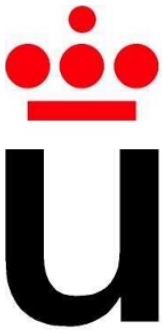

Universidad Rey Juan Carlos

ESCUELA DE INGENIERÍA INFORMÁTICA

INGENIERÍA INFORMÁTICA

Curso Académico 2010/2011

Proyecto Fin de Carrera

ActiveInterface++.

Framework de integración de sistemas. ESB Ligero

Autor: **Óscar Pernas Plaza**

Tutor: **Luis López Fernández**

Agradecimientos

De un niño que no sabía diferenciar la a con la e, a un adolescente que no sabía diferenciar la derecha de la izquierda (todavía no sé), a esto... todos habéis contribuido en algo.

Mis padres, mi hermana, Rus, Luis Lopez, Pedro, Aitor, Marco, Dani, Alberto, Javi Moreno, Chechu, Gustavo, todo lo sobrevalorado, la auténtica salud, las fábulas, la programación, las oportunidades, las barbacoas, los de Módulo, la chapa de cocacola, Cañamero, el reloj de tuenti, Moraleja de Enmedio, la imaginación, los sueños y los soñadores...

Gracias a todos!... Checkpoint!

Resumen

En este proyecto se ha realizado una librería de comunicaciones software libre denominada ActiveInterface, desarrollada en C++.

ActiveInterface ofrece un framework de integración de sistemas, e implementa una serie de patrones denominados *Enterprise Integration Patterns*. Estos patrones describen formas para hacer de la integración una tarea más sencilla, tanto contra aplicaciones externas como con procesos de la misma aplicación.
Todo este tipo de patrones están englobados en el ámbito de los sistemas de comunicación basados en mensajes o *MOM* y están orientados a permitir la distribución de mensajes de manera sencilla, flexible, optimizada y normalizada.

ActiveInterface se embebe en aplicaciones desarrolladas en C++, y actúa como módulo de comunicaciones que usa JMS como protocolo de comunicaciones de alto nivel. Bajo este protocolo de alto nivel permite establecer comunicaciones usando TCP, UDP o multicast.

El objetivo principal de esta librería es ofrecer una API que permita la creación y modificación de flujos de datos complejos de una forma sencilla, dando además la posibilidad de aplicar reglas de enrutamiento y de filtrado de mensajes. El flujo de datos de ActiveInterface se define a partir entidades que abstraen a los desarrolladores del concepto de conexión agrupándolas en servicios, y de entidades denominadas enlaces, que aplican determinados filtrados a los mensajes. Las asociaciones entre estas entidades producen diferentes flujos de datos, generando la topología de red por la que los mensajes viajan desde un proceso productor a un proceso consumidor.

Además, ofrece mecanismos de persistencia, inicialización de topologías mediante ficheros XML, mecanismos de control de congestión, gestión de flujos de datos en tiempo real, y es un desarrollo *thread safe* y multiplataforma. Sencillez, abstracción, flexibilidad y rendimiento son otras de las cualidades más importantes.

ActiveInterface es un software estable y usable, implantado en cerca de 10.000 máquinas, en proyectos tanto nacionales como internacionales. Esto le convierte en una opción a tener en cuenta en desarrollos C++ orientados a sistemas de mensajería.

Listado de acrónimos

- **MOM:** Message Oriented Middleware
- **SCADA:** Supervisory Control And Data Acquisition
- **FIFO:** First in-First Out.
- **JMS:** Java Message Service.
- **ESB:** Bus de servicio para empresas ó Enterprise Service Bus
- **DSL:** Lenguaje específico del dominio ó Domain Specific Language.
- **STOMP:** Streaming Text Orientated Messaging Protocol.
- **EAI:** Enterprise application integration.
- **EIP:** Enterprise Integration Patterns.
- **OMG:** Object Management Group.
- **PUD:** Proceso Unificado de desarrollo.
- **ANSI:** American National Standards Institute
- **W3C:** Worl Wide Web Consortium

Índice general

Índice de figuras

Capítulo 1

Introducción

La Ingeniería Informática es una ciencia nueva en la que problemas como la obsolescencia del software no habían aplicado hasta hace poco tiempo. Unas veces provocado por la escasa Ingeniería del Software, otras por la cultura del dicho popular: *Lo que funciona no se toca* y otras por nuevas funcionalidades que sólo se cubren usando nuevas tecnologías, se ha llegado a un punto de no retorno en el que nuevos requerimientos hacen imposible seguir desarrollando más plantas en un edificio con una arquitectura antigua no preparada para los requerimientos actuales.

Empresas con productos con cierta antigüedad, muy estandarizados, necesitan seguir implementando nuevas funcionalidades para hacer sus productos competitivos. Pero hay ciertas funcionalidades que derivarían en unos costes de desarrollo muy altos al tener que reimplementar parte de su núcleo actual. Además, nuevas tecnologías más específicas para una determinada funcionalidad se abren paso, reduciendo costes de desarrollo y facilitando las tareas de mantenimiento.

En este punto, obviamente, lo más deseado sería poder implementar esas nuevas funcionalidades usando esas nuevas tecnologías, modularizando y estandarizando las formas en la que nuestros nuevos requisitos se integran con nuestro núcleo de producto más antiguo, abriendo nuestro núcleo y permitiendo la migración, en caso de que fuese necesario, de determinadas partes.

Por todo esto, la palabra integración, se hace cada vez más necesaria. Integración entre tipos de máquinas, integración entre sistemas operativos, Integración entre aplicaciones, integración entre procesos e integración entre lenguajes de programación. En otras palabras, la industria del *software* se tiene que preparar para un entorno multi-tecnológico, dejando de apostar por una tecnología para todo un producto y usando la necesaria en

cada parte del mismo.

Seguir todos estos consejos orientan todos nuestros desarrollos a arquitecturas distribuidas, agregando a cada parte de la arquitectura distintos interfaces para poder adaptarse a nuevos requerimientos o nuevas integraciones. Procesos pequeños, orientados a funcionalidades específicas, usando la red como mecanismo central de comunicaciones.

Particularizando un poco más en lo referente a este proyecto de fin de carrera, en ciertos entornos industriales, por la exigencia del tiempo real y del acceso al *hardware*, siempre se ha tendido hacia desarrollos en entornos C/C++. A pesar de haber multitud de soluciones en estos mismos lenguajes, otros factores a tener en cuenta, como principalmente la complejidad del desarrollo (al estar el lenguaje más orientado a ello), la facilidad de mantenimiento, y las posibilidades de tecnologías más nuevas, están llevando a la industria a la necesidad de dar un pequeño giro, mirando hacia lo que ofrecen tecnologías como Java, y las nuevas plataformas móviles como Android o iPhone.

En este proyecto se ha tratado de dar una vuelta de tuerca más al concepto de conector entre lenguajes. Se ha intentado dar un mecanismo de comunicación multi-lenguaje y multi-plataforma siempre orientando a la facilidad de uso y a la versatilidad, tratando de llevar conceptos y formas de desarrollos más propias de lenguajes como Java a entornos C/C++.

Estos conceptos permiten establecer un primer paso para migraciones hacia arquitecturas SOA, siendo este proyecto parte importante de esta migración, haciendo funciones de enrutador y en cierta parte de tratamiento automático de mensajes.

1.1. Ámbito

En esta memoria se van a usar una serie de conceptos que constituyen el ámbito de los sistemas de mensajería basados en colas.

- **Sistemas de mensajería:** Los sistemas de mensajería o MOM son sistemas basados en el concepto de intercambio de mensajes entre clientes-servidores de manera **asíncrona**.

  En las comunicaciones tradicionales cliente-servidor, para establecer una comunicación entre ambos, se necesita que ambos entes estén disponibles para ello.

Los sistemas de mensajería incluyen un concepto intermedio o *middleware* sobre el que ambos clientes establecen comunicación síncrona y que aporta asincronismo de comunicaciones entre los clientes.

Aunque cada fabricante tiene sus especificaciones, esta comunicación asíncrona presenta una serie de ventajas comunes:

- **API:** Este tipo de sistemas siempre proveen a los usuarios de una API de sencillo manejo, con la que se pueden conseguir abarcar prácticamente todas las funcionalidades de cualquier comunicación síncrona pero con un grado de abstracción y por tanto sencillez mucho más alta.
- **Multi-plataforma y multi-lenguaje:** Muchos de estos sistemas de mensajería incorporan clientes implementados en multitud de lenguajes de programación y para distintas plataformas.
- **Sin bloqueos:** El modo de comunicación asíncrono aporta un sistema de envío sin bloqueos también denominado *Send and forget*. Cada mensaje que un cliente envía, es dirigido hacia el *middleware*, el usuario sólo envía y se olvida.
- **Contención de saturación:** En sistemas tradicionales un productor de mensajes podría verse condicionado por la velocidad de procesamiento del receptor. En los sistemas de mensajería asíncronos, el *middleware* está optimizado para almacenar los mensajes que se le envían y permite al cliente receptor consumirlos con los tiempos que sea posible sin afectar al productor de mensajes.
- **Comunicación fiable:** En los MOM se garantiza que una vez los mensajes lleguen al *middleware*, nunca se van a perder. Además, el *middleware* garantiza que los mensajes sólo serán eliminados cuando el cliente receptor los procese.
- **Funcionamiento no conectado:** Estos sistemas permiten al productor de mensajes enviar los mensajes sin tener en cuenta cuando serán recibidos, pero teniendo la seguridad de que cuando el receptor los quiera procesar, los mensajes seguirán disponibles.
- **Ordenamiento:** Garantizar que un mensaje enviado hacia el *middleware* siempre será procesado antes que un mensaje enviado posteriormente es una condición indispensable. Con esta ventaja se puede intuir una idea básica de este tipo de sistemas de comunicaciones, como son las colas FIFO de mensajes.

- **Simulación de comunicaciones síncronas:** Aunque son comunicaciones en su base asíncronas, existen medios de capas superiores que permiten simular las comunicaciones síncronas, que no son más que respuestas desde el proceso receptor hacia el emisor (a través del *broker*) de que un determinado mensaje ha sido procesado.
- **Comunicaciones punto a grupo:** Los sistemas de mensajería permiten, sin alterar sus conceptos básicos, las comunicaciones entre un emisor puntual y un grupo de consumidores.
- **Escalabilidad:** La escalabilidad es un concepto muy importante en arquitecturas industriales como los sistemas SCADA. En arquitecturas tradicionales, cuando un proceso sufre mucha carga la optimización de dicho proceso suele ser la única forma de no caer en el cuello de botella. Con sistemas MOM es muy sencillo escalar horizontalmente, estableciendo más procesos consumidores sin necesidad de cambios en el software.

Aunque también nos generan una serie de desventajas:

- **No aplicable en determinados escenarios:** Para determinados tipos de comunicaciones, el concepto de asincronismo provoca que los procesos receptores no tengan constancia del estado de los productores y viceversa.
- **Tasas de transferencia:** Aunque son sistemas muy optimizados, es evidente que los distintos pasos de los que se compone una emisión de mensajes va a aplicar una determinada ralentización en el envío de mensajes respecto a los envíos con *sockets* puros.
- **Dependencia del *middleware*:** En los MOM, el *middleware* se convierte en el centro de las comunicaciones, con lo que se tiene una gran dependencia de él. Este punto no sería aplicable si se aplican conceptos de arquitecturas distribuidas que se explicarán en posteriores capítulos de esta memoria.

- **Colas:** Los sistemas de mensajería inherentemente están relacionados con el concepto de colas de mensajes. Cuando un mensaje llega al *middleware*, éste se introduce en una cola FIFO. Estas colas pueden ser configuradas para estar en memoria, en base de datos o en fichero proporcionando persistencia. Un mensaje no será sacado de la cola hasta que no sea consumido por un receptor o por varios de ellos si es así requerido.

- **JMS:** JMS es una API estándar para la plataforma de desarrollo Java que permite construir aplicaciones que utilicen sistemas de mensajería. JMS no es el sistema

de mensajería, es un conjunto de clases que permite trabajar con estos sistemas. Gracias a ser un estándar, permite construir sistemas portables entre distintas implementaciones de sistemas de mensajería, lo que permite abstraerse del producto software *middleware* en concreto que se utilice.

- **CMS:** CMS es la implementación del estándar JMS pero para plataformas basadas en lenguajes C/C++.
- **Broker:** Una vez introducidos los conceptos anteriores, y estableciendo un vocabulario más habitual, el llamado anteriormente *middleware* es comúnmente llamado broker. El *middleware* (a partir de este momento *broker*) es el encargado de recibir los mensajes, almacenarlos y reenviarlos donde sea necesario. En el mercado existen multitud de implementaciones de *brokers* como pueden ser:

  - **WebSphereMQ:** Su escalabilidad, estabilidad y el soporte de la empresa que lo desarrolla (IBM) lo convierte el producto líder del mercado. Su elevado precio por licencia lo hace privativo en determinados entornos y proyectos como es este proyecto de fin de carrera.
  - **HornetQ:** Es uno de los productos más utilizados en entornos *OpenSource*, pero la no existencia de clientes en determinados lenguajes evita su proliferación en determinados entornos. Actualmente, sólo tiene soporte para lenguajes C/C++ mediante un protocolo denominado STOMP que reduce las posibilidades de interactuación con el *broker*.
  - **Open Message Queue:** Está incluido en el proyecto *Glassfish*, anteriormente promocionado por Sun Microsystem. Es un producto *OpenSource* y que tiene una versión no OpenSource soportada por Oracle llamada *Java Message Queue*. No dispone de un conector para plataformas desarrolladas en C/C++.
  - **ActiveMQ:** Es el producto implementado por *Apache Software Foundation*. Autodenominado el broker *Open Source* más rápido y eficiente del mercado. Dispone de multitud de conectores en distintos lenguajes. Es un producto estable, y avalado por empresas que lo usan como Yahoo, IBM o Google. Fue el producto elegido para este proyecto de fin de carrera por ser una alternativa *OpenSource*, estable y con conectores para plataformas desarrolladas en C/C++.

- **ActiveMQ-CPP:** Es el cliente para conectar con el *broker* ActiveMQ desde lenguajes C/C++. Está desarrollado bajo licencia Apache y al igual que ActiveMQ

es un proyecto en estado de incubación. ActiveMQ-CPP proporciona una API para desarrollar aplicaciones desde lenguajes C++ muy similar a la API proporcionada en lenguaje Java.

La decisión de elegir un producto u otro fue una de las tareas más difíciles. Los requerimientos iniciales eran herramientas *OpenSource* válidas, estables, contrastadas y con una comunidad activa que lo apoyase. ActiveMQ cumplía todas las características. Además, otro requisito es que tuviera un conector en lenguajes C/C++ que permitiese una API sencilla de programación. Muchos proyectos aportaban esta capacidad mediante un protocolo común de comunicaciones contra el *broker* llamado *STOMP*. Este protocolo permite la publicación de mensajes, pero además de provocar lentitud en el sistema, restringe la funcionalidad.

ActiveMQ-CPP ofrece una API sencilla, muy parecida a la sintaxis de Java. Desde un primer momento se vio que no era un producto maduro, pero que estaba bajo un desarrollo activo. Barajando las posibilidades, se tuvo en cuenta que ayudar en el testeo de un producto *OpenSource* como este, podía dar un valor añadido a este proyecto de fin de carrera.

Casi un año después de haber hecho esta elección, este proyecto de fin de carrera ha ayudado a sacar a la luz más de 10 *bugs*, ha ayudado a testear nuevas versiones y en general ha contribuido a que se desarrolle y mejore para el uso común.

1.2. Arquitecturas orientadas a servicios ó SOA

En este punto, y teniendo un conocimiento general sobre las funcionalidades que aporta este proyecto de fin de carrera, es obligatorio dar una breve introducción a que son las arquitecturas orientadas a servicios (en adelante SOA).

SOA es un conjunto de patrones y técnicas de desarrollo software que aplicadas, ayudan a que dicho desarrollo sea más aplicable a los entornos empresariales actuales.

En todo proyecto software la integración siempre es una tarea crítica. Es por ello que arquitecturas como SOA se encuentran muy de moda en todo nuevo desarrollo, ya que es un nuevo conjunto de técnicas que aplicadas consiguen abrir un sistema hacia

fuera, facilitando la integración externa e interna de nuevos módulos. Las arquitecturas orientadas a servicios están basadas en *interfaces* y tienen como núcleo la comunicación entre esas *interfaces*.

Poniendo como ejemplo una fábrica, en ella se puede establecer una cadena de trabajo muy sencilla en la que cada parte de la fábrica recibe un producto, y sobre él se aplican determinadas modificaciones para así generar un producto que servirá de entrada para el siguiente proceso de la cadena.

Las arquitecturas SOA intentan imitar este tipo de comportamiento humano en el desarrollo *software*. Cada una de las etapas serían los denominados servicios, y las modificaciones que se realizan en el producto que sirven como entrada al siguiente servicio serían la interfaz de comunicación.
Con esta breve descripción es posible apreciar que este tipo de arquitecturas son capaces de adaptarse muy fácilmente a los cambios en el proceso de negocio. Nuevos productos o nuevas formas de proceso de productos antiguos derivarían en un nuevo servicio que exportaría una interfaz nueva y que generaría un nuevo producto.

Es necesario destacar que cada etapa o servicio es auto-contenida y no depende de ningún otro servicio.

Como ventajas de las arquitecturas SOA se podrían enumerar [1]:

- Formado por servicios, poco acoplados y altamente interoperables.
- Componentes totalmente reusables.
- Total independencia de la plataforma y del lenguaje.
- No está asociado a ningún protocolo de transporte o infraestructura de objetos distribuido.
- Permite la interoperabilidad casi con cualquier entorno.

Y una serie de desventajas:

- Las soluciones suelen ir orientadas a usar XML para intercambio de mensajes. Lo

cual es pesado y lento tanto en parseo como en envío (Puede no usarse XML como medio de envío).

- Lentitud por el procesado y envío de mensajes entre servicios.
- Determinadas políticas de seguridad no implementadas.

1.2.1. ESB ó Enterprise Service Bus

Los *Enterprise Service Bus* se han convertido en el núcleo de las comunicaciones SOA, y se podrían definir como un *software* distribuido que proporciona una infraestructura para integrar aplicaciones y, en particular, para integrarlas según el modelo de SOA [2]. Un ESB suele tener una serie de características [3]:

- Enrutamiento y redireccionamiento de mensajes.
- Multiplicidad de tipos de transporte y protocolos de enlace.
- Transformación de contenido.
- Soporta múltiples patrones de interacción (petición-respuesta, publicación-suscripción, notificación de eventos, etc.)
- Presencia de adaptadores a múltiples plataformas.
- Características de garantía de la calidad del servicio (QoS), como transaccionalidad, seguridad y persistencia.
- Gestión y monitorización.
- Repositorio de objetos.

Como se puede observar por sus características, los ESB ayudan a desacoplar aun más los servicios, pues abstraen a los servicios de protocolos, transporte, tipos de interacción y otras muchas cosas.

1.2.2. Frameworks de integración de aplicaciones empresariales

Una vez se ha hablado sobre los ESBs, especificando sus funciones, y reseñando que este proyecto de fin de carrera sólo permite cierta funcionalidades, ActiveInterface se podría englobar en ese punto intermedio entre un simple cliente C++ y un completo ESB.

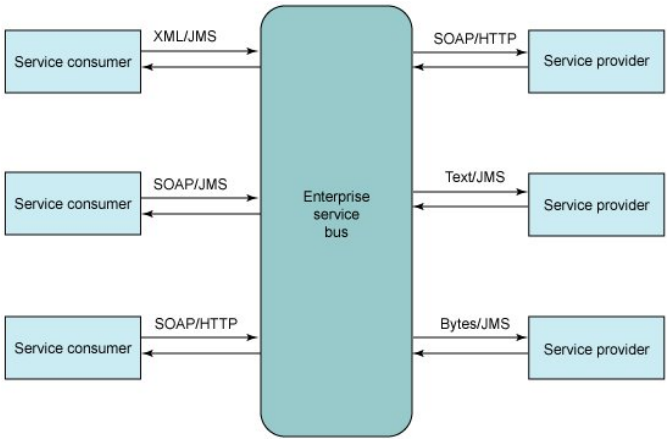

Figura 1.1: Representación gráfica de un ESB.

Los patrones de integración o EIP son un conjunto de reglas, principios y metodologías que seguidas, ayudan a que la integración entre distintas aplicaciones, plataformas y en general, la integración entre distintos servicios de una empresa. Todos estos principios y metodologías fueron inicialmente recogidos en un libro con nombre: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*[4].

El objetivo principal de estos patrones de integración es dar una forma no restrictiva de compartir datos entre procesos de negocio, entendiendo como proceso de negocio, cualquier proceso, aplicación o repositorio de datos de una empresa [5].

A continuación, se van a introducir los patrones de integración más importantes:

- **Mensajería:** Es el patrón usado para describir como los datos son enviados de una manera asíncrona y confiable, con formatos definibles y modificables por el usuario.

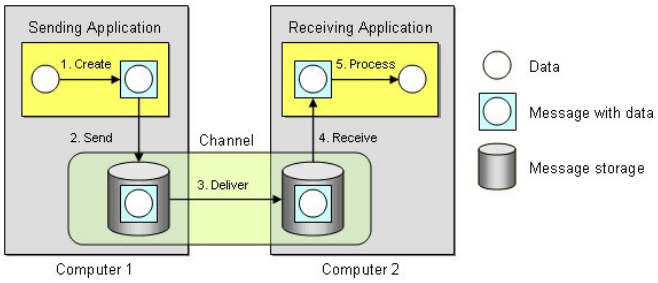

Figura 1.2: Representación gráfica de un sistema de mensajería.

- **Canal de mensaje:** Los canales son los patrones por los que se envían los mensajes y los que conectan las aplicaciones. Cuando una aplicación quiere enviar un mensaje

lo envía a un canal específico.

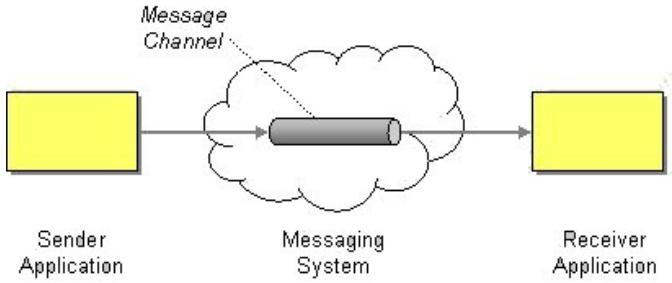

Figura 1.3: Canal de mensajes.

- **Mensaje:** Patrón que empaqueta los datos y que permite enviarse a través del canal.
- **Tuberías y filtros:** Patrón arquitectural cuyo objetivo es dividir en partes más pequeñas un determinado problema, asociando estas pequeñas partes mediante tuberías y aplicando filtros.

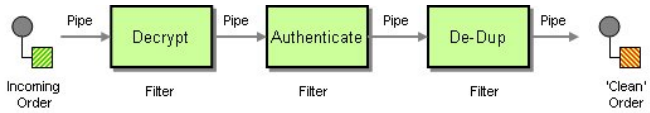

Figura 1.4: Tuberías y filtros sobre mensajes.

- **Enrutamiento de mensajes:** Es un filtro especial en el que cuando un mensaje es consumido existe la posibilidad de reenviarlo a otro canal dependiendo de unas condiciones.

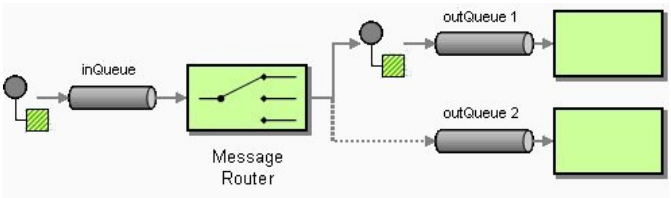

Figura 1.5: Enrutamiento de mensajes.

- **Traducción de mensajes:** Este patrón permite que cuando un mensaje es consumido, aplicarle determinados cambios para transformar el tipo de dato recibido.

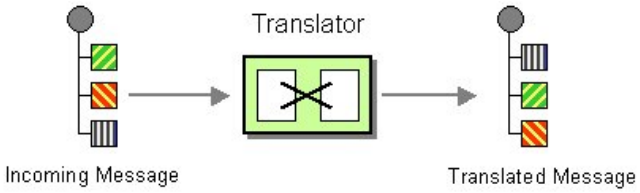

Figura 1.6: Traducción de mensajes.

- **Recepción final de mensaje:** Es el patrón *endpoint*, mediante el cual se da la posibilidad de que haya un primer cliente que es el que origina el mensaje, o un destino final del mensaje, donde es procesado.

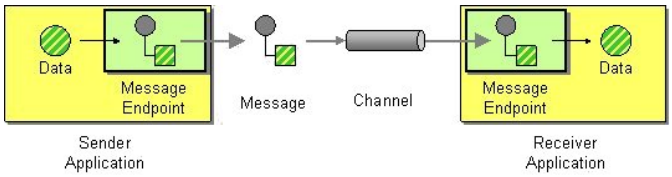

Figura 1.7: Recepción final de mensaje.

Como se puede apreciar, todos estos patrones formarían parte de los anteriormente estudiados ESBs, pero los ESB's tienen otras características que se salen fuera de esta metodología. Por tanto, la definición más precisa para este proyecto de fin de carrera sería que **ActiveInterface++ podría considerarse como un *framework* de integración de aplicaciones empresariales basado en los denominados patrones de integración para empresas o EIP**.

En los siguientes capítulos de esta memoria, se describirán, por ejemplo el estado del arte, que incluye una descripción de herramientas similares que actualmente competirían con este proyecto de fin de carrera. Además, se presentarán las metodologías de desarrollo elegidas y se introducirán más en profundidad conceptos acerca de las tecnologías utilizadas en el desarrollo.
Se describirá todo el proceso de desarrollo que se ha llevado a cabo para obtener una versión estable final, especificando los requisitos, mostrando determinadas pinceladas del diseño y describiendo brevemente la implementación desarrollada. Además, se mostrarán los casos de prueba que se han llevado a cabo, y se indicarán usos reales donde se ha utilizado esta tecnología.

Para finalizar, se mostrarán las conclusiones de este proyecto, incluyendo los logros alcanzados y los trabajos futuros.

Capítulo 2

Objetivos

Como se apuntó en los primeros puntos de la introducción, los desarrollos en C/C++ suelen ser desarrollos muy específicos, como procesos industriales o procesos en los que el máximo rendimiento es condición indispensable.

A la vez que es innegable el mejor rendimiento de lenguajes como C/C++ frente a lenguajes interpretados como Java, es innegable también que Java o cualquier otra plataforma de desarrollo más nueva, ofrece una serie de ventajas de otro tipo. Ventajas que pueden hacer que determinados procesos comiencen a migrarse a otros lenguajes más específicos que ofrecen menores costes de desarrollo debido a su más alta abstracción.

En mi opinión, el principal problema de los desarrollos C/C++ frente a otros lenguajes es la carencia de *frameworks* que simplifiquen el desarrollo de determinados tipos de requerimientos. Por ello, uno de los objetivos de esta librería es dar a los programadores C/C++ de un **framework de desarrollo que permita al desarrollador programar con un grado de abstracción alto perdiendo la menor funcionalidad y rendimiento posible.**

2.1. Descripción general

ActiveInterface es una librería de comunicaciones desarrollada en C++ que se basa en el estándar de comunicaciones Java Message Service (JMS) que provee la librería nativa ActiveMQ-CPP.

Por tanto, ActiveInterface se puede entender técnicamente como un recubrimiento que añade una serie de funcionalidades y que facilita el desarrollo de aplicaciones en lenguajes

C sobre arquitecturas MOM basadas en el *broker* ActiveMQ.

- **Sencillez en la creación de redes de productores y consumidores:** ActiveInterface provee a los usuarios de una API de desarrollo muy simple para distribuir un flujo de consumo y producción de mensajes. Su diseño está orientado a ser de propósito general, intentando cubrir todos los requerimientos tanto de desarrollos simples como complejos.
  ActiveInterface está basado en tres conceptos que definen su arquitectura:

  - **Conexión:** Es la parte más baja de la arquitectura. Es una entidad que abstrae el concepto de conexión con el *broker*, y que puede ser entendible como el verdadero productor o consumidor de los mensajes.
  - **Enlace:** Es la capa intermedia que permite al usuario agrupar las conexiones y modificar los mensajes que son enviados a través de un enlace determinado.
  - **Servicio:** Es la parte visible. Es el concepto a través del cual el usuario interactúa con la librería. Todos los mensajes son enviados a través de un identificador de servicio.

  Sobre estos tres conceptos, ActiveInterface permite cubrir desde escenarios simples, como un sólo productor o consumidor a una cola, a soluciones más complejas como por ejemplo el caso de necesitar el envío del mismo mensaje a distintos destinos.
  ActiveInterface permite agrupar conexiones dentro del mismo identificador de servicio. Además, usando el concepto de enlace, ActiveInterface permite poder definir propiedades por defecto que siempre son adjuntadas en todos los mensajes que se envían a través de dicho enlace.

  Por ejemplo, si en nuestra arquitectura hubiera dos niveles de *backup* que recibiesen todos los mensajes y los almacenasen, ActiveInterface permitiría enviar dichos mensaje a través de un identificador de servicio único, que permite al usuario abstraerse del número de niveles de *backup*. Además, si el mensaje que se debe enviar a cada nivel necesita tener una serie de propiedades específicas, mediante el concepto de enlace, ActiveInterface permitiría especificar las propiedades por defecto que son enviadas a cada nivel particular.

  En la figura 1.1, se puede ver una topología típica de ActiveInterface en la que los servicios están relacionados con los enlaces y estos con las conexiones. En posteriores

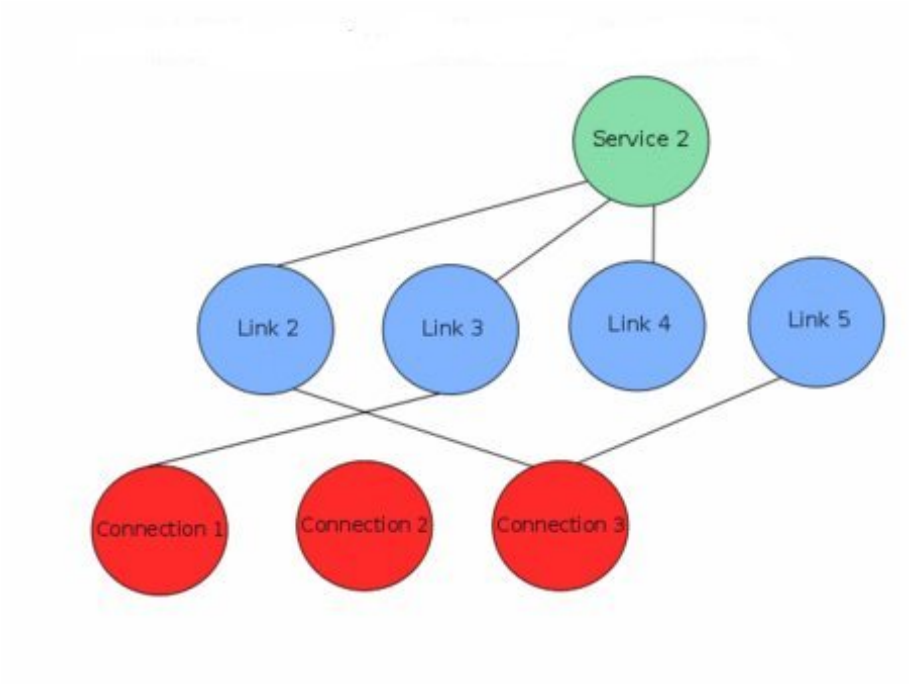

Figura 2.1: Ejemplo de topología.

secciones de esta memoria, se entrará más en profundidad sobre estos tres conceptos.

- **Creación y modificación de la topología en tiempo real:** ActiveInterface ofrece al usuario una forma fácil de borrar, modificar y crear nuevas conexiones, enlaces y servicios en tiempo real. Por ejemplo, si un determinado paquete o un determinado evento pudiera provocar un cambio en el destino de un mensaje, usando ActiveInterface, se podría modificar la topología de la red, cambiando el destino de la cola, modificándolo o borrándolo si fuera necesario.

  Como se verá en capítulos posteriores de esta memoria, ActiveInterface ofrece una API muy sencilla que permite al usuario crear toda la topología (conexiones-enlaces-servicios),modificarla y utilizarla para el envío y recepción de mensajes.

- **Inicialización XML:** A parte de la API, ActiveInterface ofrece a los usuarios la posibilidad de inicializar su topología en base a un fichero de inicialización XML. Este fichero permite inicializar la librería, instanciando todos los productores, consumidores, enlaces y servicios mediante un fichero muy sencillo.

```
<?xml version="1.0" ?>
```

```
<connectionslist>
  <connection
    id="producer1"
    ipbroker="failover:///(tcp://localhost:61616)"
    type=1
    destination="alarms"
    persistent=0
    topic=0
  />
</connectionslist>

<linkslist>
  <link id="link1" name="alarmas" connectionid="producer1" />
</linkslist>

<serviceslist>
  <service id=1>
    <link id=1 />
  </service>
</serviceslist>
```

Todas las funcionalidades de creación de elementos de la API se pueden realizar igualmente mediante este fichero de configuración. La idea general es que usando esta configuración, cualquier persona sea capaz de generar una topología de red sin ningún conocimiento de programación.

- **Mensajes multi-tipo:** Cuando se desarrollan formas de comunicación simples, como podrían ser textos con marcación XML, el envío y la recepción no suponen ningún problema. En cuanto los mensajes a enviar son más complejos como es envío de estructuras, objetos binarios o cualquier requerido entre cliente y servidor, se requiere un determinado protocolo de datos para ponerse de acuerdo parte emisora y receptora.

  Una de las mejores funcionalidades de ActiveInterface es que proporciona al usuario de una interfaz simple para mandar mensajes de cualquier tipo, compuestos por cualquier tipo de información. Simulando un acceso a una tabla *Hash* (clave-valor), ActiveInterface proporciona un recubrimiento sobre los mensajes JMS nativos que aporta una forma de crear el mensaje e insertar en él cualquier tipo de dato (entero, real, cadena o bytes). Además, provee una forma simple de recuperar en destino los datos insertados.

  Este mensaje multi-tipo está basado en el concepto de parámetros, que son los datos reales que van en el cuerpo del mensaje, y las propiedades, que son propiedades

asociadas a todo el mensaje y sobre las cuales se pueden aplicar filtros. Usando este tipo de mensajes, el desarrollo de un protocolo de intercambio de cualquier tipo de dato sería tan sencillo como definir un identificador para el tipo de dato que se está enviando y recoger el mismo identificador en destino.
Por ejemplo, si en una aplicación de *chat* se quiere poder enviar fotografías, se podría definir un mensaje que contuviese la foto y una descripción de la siguiente forma:

```
insertStringParameter("tipoMensaje",1),
insertBytesParameter ("foto", foto);
insertStringProperty ("descripcion", descripcion);
```

Como se ve en el ejemplo anterior, mediante estos tres simples comandos, se estaría definiendo un tipo de mensaje con una foto y una descripción. En la parte receptora se podría actuar por el tipo de mensaje, o leyendo los parámetros y propiedades iterativamente.
Todos estos métodos están descritos en la API de ActiveInterface, y permiten al

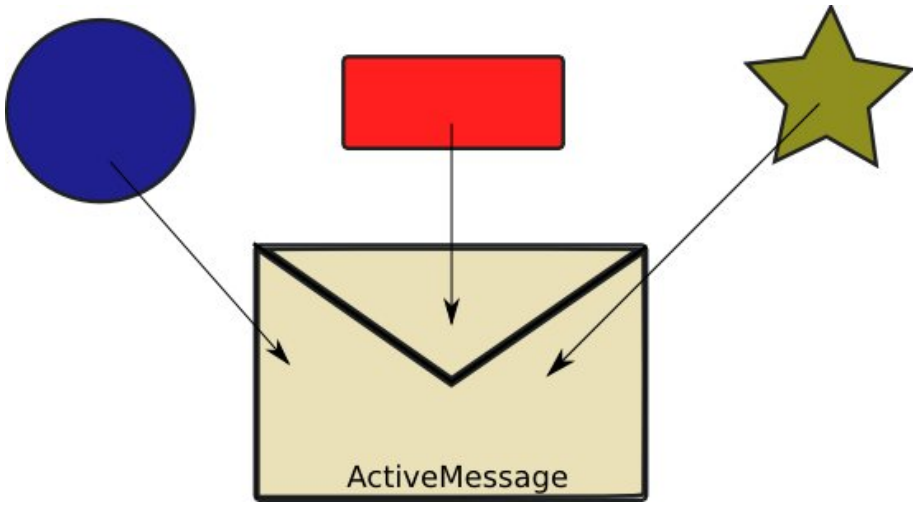

Figura 2.2: ActiveMessage. Mensaje por defecto de ActiveInterface++.

usuario insertar datos de una manera fácil, y recuperarlos sin ni siquiera tener en cuenta el protocolo y recuperando los datos que vengan en el mensaje que se haya recibido.

- **Cola interna y Persistencia:** Una de los razonamientos que pueden complicar un desarrollo usando cualquier sistema de mensajería es la posibilidad de perdida de mensajes.

Nativamente, cuando el *broker* está fuera de servicio, un cliente se podría comportar ya sea bloqueando el envío de mensajes, lo que bloquearía toda la ejecución que ha hecho ese envío (aplicaciones no multihilo), o descartando los mensajes una vez se ha llegado a un número máximo de reintentos.
En el caso del bloqueo, si una aplicación no ha tenido en cuenta este problema, cualquier parada del broker supondría un bloqueo de la aplicación ya que pararía todo el proceso de ejecución del programa que está intentando enviar.
Si se opta por la solución anterior pero usando múltiples hilos por envío, se podría llegar a exceder el número de hilos posibles de una máquina si el *broker* está fuera de servicio durante un tiempo prolongado.
Se podría optar por pensar en descartar los mensajes una vez se reintenten una serie de veces, pero en ese caso, se estaría pasando por alto un punto muy importante en una librería de comunicaciones, y es que cuando se envía algo, tiene que llegar siempre abstrayendo al usuario de **cómo o cuando**.

Para resolver estos problemas, ActiveInterface ofrece una cola interna y un sistema de persistencia que se puede manipular para optimizarlo a los requerimientos específicos de cada usuario. Todos los mensajes que se envían son almacenados en una cola interna que actúa de almacén de mensajes hasta que se puedan enviar. El tamaño de esta cola es definible por el usuario. Además, esta funcionalidad establece un pequeño sistema de congestión frente a problemas de lentitud en conectividad de red o frente a altas tasas de envíos de mensajes en las que la red actúa como cuello de botella.

Llegados a este punto, la única posibilidad de perder un mensaje sería con un apagado repentino de la máquina que haría perder los mensajes que estaban almacenados en el almacén de mensajes. Por esto, ActiveInterface da al usuario la posibilidad de usar o no usar un mecanismo de persistencia en origen.

Cuando un mensaje es enviado a una cola de mensajes definida como persistente, estos mensajes son escritos en un fichero local de la máquina y posteriormente encolados. Cuando se produce una falta de servicio del *broker*, los mensajes se van acumulando en el almacén hasta que este se desborda, momento en el que la librería entra en el determinado *modo de persistencia*.
En este modo de persistencia, el usuario podrá seguir enviando mensajes, y el sistema

de persistencia los almacenará e irá encolando y enviando según vaya siendo posible. Este sistema le concede al usuario el concepto de *"enviar y olvidar"*, dándole la seguridad de que el mensaje se enviará en cuanto sea posible.

- **Multi-plataforma y *thread-safe*:** ActiveInterface está desarrollado y probado bajo varios entornos (como Windows o Linux) y está preparado para ser usado en entornos multihilo de manera segura.

Actualmente existen más de 10.000 máquinas en entornos de producción que usan ActiveInterface. Hasta ahora los resultados son muy buenos, soportando tasas de más de 1000 mensajes por segundo (dependiendo del tamaño del mensaje y de la máquina), en entornos de alta disponibilidad y rendimiento. Además, se estima que el tiempo de desarrollo se reduce en más de un 80 por ciento respecto a usar la librería nativa.

2.1.1. Profundidad

ActiveInterface puede considerarse algo más que un simple cliente JMS escrito en C++, ya que provee muchas características que lo hacen dar ciertas funcionalidades de las que daría un ESB. Usando la lista de características de un ESB descrita en anteriores secciones, se van a particularizar sobre este proyecto de fin de carrera haciendo ver con un poco más de detalle hasta que profundidad se ha llegado.

- **Enrutamiento y redireccionamiento de mensajes:** Teniendo un conocimiento previo adquirido en las secciones anteriores de esta introducción, se puede decir que ActiveInterface provee esta funcionalidad, pero de una manera menos configurable. La mayoría de los ESB proporcionan un lenguaje denominado DSL.
  Un DSL es un lenguaje específico que ha sido diseñado para resolver un determinado tipo de problema (en contraposición a los lenguajes de propósito general como C++ o Java). A través de estos DSL, los ESB son capaces de definir rutas, reenvío de paquetes y cualquier redirección de los paquetes desde una entrada a una salida.
  Como se ha dicho, ActiveInterface permite realizar este tipo de reenvíos, pero de manera no configurable, es decir, incrustado en el código fuente. Es una desventaja pues cualquier cambio en la tabla de rutas te obliga a recompilar código de la aplicación cliente.

- **Multiplicidad de tipos de transporte y protocolos de enlace:** ActiveInterface soporta distintos tipos de enlace como TCP, UDP o multicast, pero actualmente sólo soporta como tipo de transporte JMS. Es decir, por ejemplo, no soporta que comunicación entre servicios con interfaces en SOAP con servicios en JMS.

- **Transformación de contenido:** ActiveInterface soporta filtros pero actualmente sólo para el envío de información. Es decir, se pueden definir filtros para que siempre que se envíe un mensaje se le añadan cierto tipo de propiedades. En cuanto a recepción no existe esta capacidad.
  La capa de enlace de ActiveInterface sería la capa donde se implementaría la posibilidad de incluir esta transformación y aplicación de filtros a los mensajes.

- **Soporte de múltiples patrones de interacción:** ActiveInterface soporta distintos tipos de interacción, comunicaciones punto a punto, punto a grupo, petición-respuesta, publicación-suscripción, etc. .

- **Presencia de adaptadores a múltiples plataformas:** Al ser multi-plataforma, ActiveInterface ha sido testeado sobre plataformas Microsoft y Unix (Linux). Su desarrollo en C/C++ hace que se utilización sea fácil desde lenguajes de *scripting* como python (Python-C++) o desde Java con JNA ó JNI.

- **Características de garantía de la calidad del servicio:** ActiveInterface posee persistencia en cliente, y mediante la librería nativa asegura la transaccionalidad de un mensaje. En cuanto a seguridad, ActiveInterface soporta SSL y además permite la creación de colas o tópicos con contraseña.

- **Gestión y monitorización:** ActiveInterface no provee de ningún sistema de monitorización y gestión, pero gracias a su API sería muy fácil de desarrollar pues el cliente que monitorizase sería un cliente JMS más, que usaría la API de una manera remota y que permitiría acceder a una serie de características que serían necesario implementar.

2.1.2. Conclusiones

Por tanto, se puede ver que ActiveInterface se encuentra a medio camino entre un cliente simple y un ESB que proporciona características más avanzadas.
Hay que reseñar que, como se verá en las descripciones de diseño posteriores, ActiveInterface está diseñado para soportar la mayoría de estas ampliaciones futuras de una manera sencilla.

Una vez conocido totalmente el contexto que rodea a este proyecto de fin de carrera, se podrían resumir como objetivos cumplidos:

- **Adaptación de metodologías orientadas a servicios:** Una vez entendido el concepto de Arquitecturas orientadas a servicios (SOA), es de reseñar como ActiveInterface podría cambiar la forma de desarrollar, pensar y estructurar aplicaciones escritas en C/C++.
  Aunque es bastante complicado adaptar arquitecturas heredadas antiguas a arquitecturas SOA, introducir ActiveInterface en nuevos procesos añadiendo una pasarela para la conversión de datos entre el sistema heredado y el nuevo, podría ser una de las formas con las que ir transformando los productos antiguos a la vez de ir proveyéndoles de mayor funcionalidad.

- **Conexión con otros lenguajes:** Nuevos servicios desarrollados con nuevas herramientas, mucho más específicas para su función, con un menor coste de desarrollo y con una integración sencilla en un sistema antiguo. Conexión con Interfaces gráficas Java, con interfaces web, con lenguajes de *scripting* como Python, tareas que anteriormente necesitaban de un gran desarrollo y una ardua tarea de integración, ahora estarían acotadas y simplificadas a como los datos son enviados dentro del mensaje.

- **Distribuir contenido:** Usando ActiveInterface es mucho más simple distribuir contenido. Muchas de los desarrollos C/C++ que necesitan de comunicación de red usan la implementación de *sockets* nativa del sistema *host* sobre el que se esté desarrollando, ya sea windows sockets o las primitivas de sockets.
  Usar este tipo de comunicación conlleva grandes cantidades de código, de depuración y de definición del protocolo.
  Usando ActiveInterface, el envío de datos se resume a unas pocas líneas de código, además de permitir multitud de funcionalidades (descritas anteriormente) que

codificarlas llevaría una gran cantidad de esfuerzo y de desarrollo.

- **OpenSource:** ActiveInterface es *OpenSource*, y aunque todavía se encuentra a falta de determinadas funcionalidades para poder referirse a él como tal, se podría ver como el primer *framework* de integración de arquitecturas de comunicaciones basadas en mensajes, escrito y desarrollado nativamente en C/C++.

2.2. Estado del arte

Una vez conocidas las funcionalidades que provee ActiveInterface, es necesario explicar el estado del arte, entendiendo estado del arte como todas las herramientas similares que existen actualmente, cual es su estado y dar unas breves pinceladas sobre algunas de ellas.

2.2.1. Sistemas orientados a mensajes y C++

Es difícil establecer un estado del arte en el mismo contexto en el que se establece ActiveInterface ya que como se ha intentado resaltar en el capítulo anterior, este tipo de arquitecturas y metodologías de programación no son las más comunes en lenguajes C/C++.

Si un desarrollador *software* C/C++ tuviera como tarea encontrar una nueva plataforma de comunicaciones y la idea de los sistemas orientados a mensajes le pareciera buena, uno de los escollos más importantes que tendría sería encontrar una forma de conectar con él. Como se describió en la introducción, la mayoría de los *brokers* (incluidos los más importantes) carecen de conectores C/C++ y si los tienen se encuentran limitados en funcionalidad y con APIs más complicadas de entender. Se enfrentarían a un mundo con mucha menos documentación y soporte que se podrían encontrar si desarrollasen en lenguajes como Java.

Muy pocos *broker* tienen un conector C/C++ maduro, y obviando las soluciones propietarias como WebsphereMQ, que dispone de un espacio de cliente muy parecido al entorno de desarrollo para Java, existen escasas soluciones *OpenSource*, estables, con una funcionalidad amplia y con una comunidad activa, a las que poder tener en cuenta en entornos reales de producción.

La mayoría de las soluciones MOM disponen de un conector C++ usando el protocolo *STOMP*. Como identifican sus siglas, STOMP es un un protocolo de red orientado a

mensajes de texto. STOMP provee a los clientes de una forma de operar con casi cualquier *broker* de mensajería, ya que la mayoría de los *broker* entienden este protocolo al ser un protocolo más antiguo. El problema de utilizar STOMP es que es un protocolo antiguo, basado en texto y sin muchas de las nuevas funcionalidades que han dado más rendimiento y funcionalidad a este tipo de sistemas de mensajería como compresión de mensajes, la posibilidad de usar SSL, propiedades de mensajes, colas temporales o los selectores.
Por estas limitaciones, pese a que STOMP es un protocolo que dispone de librerías en muchos lenguajes de programación como C++, Java, .NET, Ruby, Python, etc., y multiplataforma, para este proyecto era inviable intentar dar una funcionalidad mayor, si el protocolo nativo no la soportaba.

ActiveMQ-CPP

En este punto y en el momento de escritura de esta memoria, se puede afirmar que ActiveMQ es la única solución de mensajería que dispone de un cliente C++ medianamente estable y con una API y funcionalidad muy parecidas a las de otros lenguajes de programación. Hay que reseñar, que la primera versión del cliente C++ (a partir de este momento ActiveMQ-CPP) data de mediados del año 2007, y aunque se encuentra en una fase estable, continúa en desarrollo, aportando nuevas funcionalidades y resolviendo problemas existentes.

Actualmente sobre ActiveMQ-CPP no existe ningún desarrollo que agregue a su API nativa nuevas funcionalidades que son requeridas en casi cualquier desarrollo que se vaya a realizar usando esta tecnología. ActiveInterface se encuadra en este contexto, siendo la referencia del estado del arte en sistemas de comunicaciones basados en mensajes OpenSource y en entornos C/C++.

En los siguientes apartados de este capítulo se va a proceder a comparar nuestro desarrollo con productos dentro del mismo contexto, pero bajo otros lenguajes de programación, especialmente Java.

2.2.2. Sistemas orientados a mensajes y Java

Los sistemas orientados a mensajes en entornos Java se encuentran en una posición muy distinta a la que se encuentran en C++. Hacia el año 2008 fue cuando empezaron a utilizarse, y en ese año se pronosticaba que un 80 por ciento de los nuevos desarrollos que se empezarían, sería desarrollado bajo metodologías SOA[6].

En Java existen soluciones de todo tipo para estos tipos de entornos de mensajería. Desde ESB's completos a otras aplicaciones llamadas *lightweight ESB's* que podrían ser las aplicaciones más comparables con este proyecto de fin de carrera.

Apache Camel

Como se especificó en la introducción, la definición más idónea para definir este proyecto de fin de carrera es que es un *framework* de integraciones OpenSource. Y esto es exactamente lo que es Apache Camel.[7]
Apache Camel es un producto desarrollado bajo el proyecto Apache y con licencia GPL. Para describir un poco más informalmente lo que es Apache Camel, se podría decir que Apache Camel tiene muchas características comunes con los ESBs cómo enrutamiento inteligente, transformación de datos, mediación y monitorización de mensajes. Pero no dispone de otras características como repositorio de objetos.

Apache Camel implementa todos los patrones EIP descritos en la introducción de esta memoria:

- **Canales de mensajes:** Apache Camel permite mensajes punto a punto o publicación-suscripción.
- **Construcción de mensajes:** Apache Camel permite envío de respuestas a mensajes simulando comunicaciones síncronas, también llamado *Request Reply*.
- **Enrutado de mensajes:** Apache Camel permite casi cualquier tipo de operación sobre los mensajes en cuanto a operaciones de filtrado, reenvío y enrutado de mensajes. Permite enviar un mensaje a listas de destinatarios, permite hacer un enrutado dinámico de mensajes mediante definiciones en lenguajes DSL como Scala DSL, XML o JavaDSL.
- **Transformación de mensajes:** Permite modificar los mensajes mediante los lenguajes DSL, como mediante una API en lenguaje Java. Permite agregar, filtrar e incluso comprimir mensajes.
- **Extremos de las comunicaciones:** Puede actuar de simple cliente en ambos extremos de la comunicación tanto como un productor de mensajes como un consumidor. Permite consumir mensajes selectivamente en base a propiedades, como lanzar múltiples consumidores sobre la misma cola de mensajes.

- **Monitorización:** Permite la monitorización de su actividad de distintas formas, con JMX y mediante el modulo BAM *Business Activity Monitor*.

Por tanto, Apache Camel es una interesante herramienta, que incluida en todos los procesos que componen un determinado producto de una empresa, independizaría cada uno de los procesos y proporcionaría una forma fácil de hacer la conversión de esos datos entre servicios. Además permitiría enrutarlos dinámicamente, filtrarlos o completarlos, dejando al desarrollador una interfaz muy simple para hacer todo esto, o incluso proveyendo a los administradores de sistemas de una forma de hacerlo dinámicamente si fuera necesario, sin necesidad de tener ningún conocimiento de programación.

Apache Camel está desarrollado en Java, y aunque puede actuar como proceso, **sólo** puede ser embebido en desarrollos Java.

2.2.3. Conclusiones

Como conclusión a esta sección, es necesario reseñar la gran diferencia del estado del arte entre dos lenguajes de programación de alto nivel como C++ y Java. Si en Java todo este tipo de arquitecturas están muy generalizadas, y casi cualquier desarrollo o integración se hace usando este tipo de patrones y metodologías, en C++ las posibilidades son mínimas.

También es necesario reseñar el estado del arte de este proyecto de fin de carrera. Intentar comparar este proyecto de fin de carrera con aplicaciones maduras, totalmente implantadas y con una comunidad de mucha gente quizá es demasiado ambicioso, pero creo que es un buen comienzo para ofrecer una serie de funcionalidades en un entorno ciertamente innovador a veces no suficientemente maduro. Las comparaciones se utilizan como ejemplo para poner en contexto el desarrollo llevado a cabo en este proyecto de fin de carrera.

Como se ha dicho anteriormente, ActiveInterface es un cliente mejorado, que ofrece ciertas funcionalidades que ofrecen otros productos pero que en ningún momento fueron las directrices de desarrollo. ActiveInterface partió de la idea de utilizar los sistemas orientados a mensajes como una plataforma para intercambio de datos P2P, y el primer paso era tener una librería que abstrajera lo máximo posible de las comunicaciones.

Creo que la mejor descripción que se podría hacer para esta librería es que muchas de las funcionalidades que ofrece, cualquier desarrollador sobre estas tecnologías, tendría que implementarlas de una manera u otra si quiere tener un sistema flexible, confiable, multi-plataforma y configurable.

2.3. Metodología y tecnologías

En este capítulo se van a describir las metodologías de desarrollo que se han seguido desde sus comienzos hasta su implementación final. Se describirán varias metodologías y se especificará cual ha sido la elegida para la implementación de este PFC.
Además, se describirán las tecnologías usadas, especificando las razones por las que fueron elegidas, mostrando sus ventajas y desventajas.

2.3.1. Metodologías.

La elección de una metodología es uno de los aspectos más importantes, y que a posteriori más efectos puede tener en un desarrollo *software*. En las siguientes secciones se va a comparar la metodología tradicional, como es el proceso unificado de desarrollo ó PUD, con metodologías ya no tan nuevas como *Extreme Programming*. A partir del estudio de ambas se justificará la decisión elegida.

El proceso unificado de desarrollo

El Proceso Unificado de Desarrollo fue elaborado por Ivar Jacobson, Grady Booch y James Rumbaugh, en 1999[8]. El PUD es un proceso de desarrollo *software*. Un proceso de desarrollo software es el conjunto de actividades necesarias para transformar los requisitos de un usuario en un sistema software[8].
Pero el PUD es algo más que un proceso, se trata de un marco de trabajo genérico que puede especializarse para una gran variedad de sistemas, para diferentes áreas de aplicación, diferentes tipos de organizaciones y diferentes niveles de aptitud [8].
Una de las partes esenciales del PUD es el denominado Lenguaje Unificado de Modelado ó UML. UML es el formato en el que el PUD prepara todos los esquemas de un sistema software.

Como características más importantes del PUD se pueden citar:

- **Dirigido por casos de uso:** Todo sistema software tiene sentido si alguien o algo interactúa con él, es decir, un usuario ejecuta acciones sobre el sistema que producen una respuesta. Cada una de esas interacciones entre un usuario define lo que se denomina **caso de uso**.
  Cada uno de estos casos de uso especifica una funcionalidad que el sistema proporciona al usuario.
  Los casos de uso no sólo son una herramienta para especificar los requisitos del sistema, sino que también guían su diseño, implementación y pruebas, es decir, guían todo el proceso del desarrollo del *software*.

- **Centrado en la arquitectura:** La arquitectura en un sistema software se describe mediante diferentes vistas del sistema en construcción. El PUD asume que no existe una arquitectura única con la que poder definir todos los desarrollos posibles, ya que la arquitectura refleja los casos de uso y otros muchos factores como la arquitectura hardware, el sistema operativo, la base de datos, o los protocolos de comunicaciones. La arquitectura es una vista del diseño completo con las características más importantes resaltadas, dejando los detalles de lado. Debe de tener en cuenta la comprensibilidad, la facilidad de adaptación al cambio y la reutilización.

- **Iterativo e incremental:** El PUD tiene en cuenta que los procesos de desarrollo software suelen ser tareas de gran esfuerzo que duran una considerable cantidad de tiempo. Por esto, es necesario dividir el trabajo en partes más pequeñas.
  Cada una de esas partes es una iteración que resulta en un incremento del valor del producto. Las iteraciones hacen referencia a pasos en el flujo de trabajo, y los incrementos al crecimiento del producto.
  En cada iteración se recorren los flujos de trabajo definidos: requisitos, análisis y diseño, implementación y pruebas. Cada iteración termina convirtiendo en código ejecutable los casos de uso que se desarrollan en dicha iteración. En cada iteración se amplía el sistema con nuevos casos de uso, se identifican nuevos riesgos y se corrigen los conocidos.
  Cada iteración se agrupa en fases secuenciales: inicio, elaboración, construcción y transición. Cada una de ellas hace más hincapié en una etapa concreta del desarrollo.

Estas características definen el PUD y son todas de igual importancia. La arquitectura proporciona la estructura sobre la cual guiar las iteraciones, mientras que los casos de uso definen los objetivos y dirigen el trabajo de cada iteración.

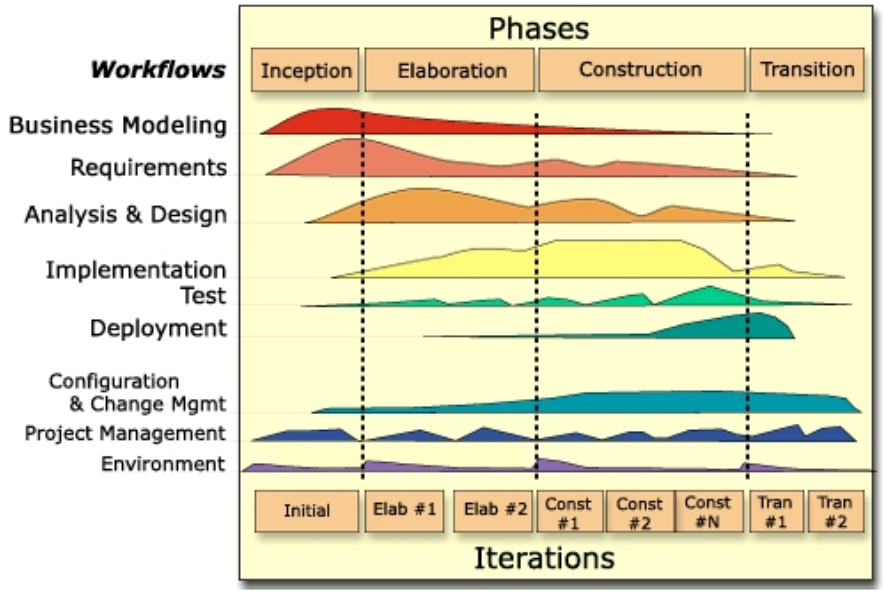

Figura 2.3: Ciclo de vida del PUD.

Por tanto, se puede ver que el PUD es un proceso ciertamente estricto, que da más importancia a las etapas de diseño y análisis que al desarrollo. Aunque aún es la metodología de desarrollo más usada, las metodologías ágiles como la que se va a explicar a continuación están ganando terreno cada vez más.

Extreme Programming

La metodología Extreme Programming está dentro de las metodologías de desarrollo ágiles, y quizá sea la más popular. Fue formulada por Kent Beck, quien en 1999 publicó el libro: *Extreme Programming Explained: Embrace Change*.
XP es una metodología simple, eficiente, con bajo riesgo, flexible, predecible, científica y sobre todo es una metodología que hace divertido el desarrollo software[9]. Esta metodología es fácilmente distinguible de otras ya que:

- Desarrolla evolutivamente, en ciclos cortos.
- Flexibiliza la planificación de cambios en la implementación de funcionalidades.
- Garantiza la calidad del software, ya que en cada evolución tanto los programadores como los clientes generan y ejecutan pruebas para monitorizar el estado del software.
- Estrecha la colaboración entre los programadores.
- Define un diseño del proyecto que se va evolucionando desde los primeros modelos hasta los últimos desarrollos.

XP es una disciplina del desarrollo software, ya que obliga a realizar una serie de pasos, pero ninguna de las ideas de esta metodología son nuevas, y la mayoría se han estado aplicando desde que se empezaron a desarrollar los primeros proyectos software.

XP propone dar más importancia a la adaptabilidad que a la previsibilidad. Es decir, XP da por hecho que los cambios en el software son un aspecto natural en cualquier desarrollo software y que son difíciles de prever. Los que defienden esta metodología creen que es mucho más fácil desarrollar un software adaptable a cambios, que definir todos los requisitos de manera exhaustiva e ir cambiándolos según se van modificando.

Esta metodología esta basada en cuatro valores:

- **Realimentación continua** entre el cliente y el equipo.
- **Comunicación** entre todos los participantes.
- **Simplicidad** en el enfoque de las soluciones implementadas.
- **Coraje** al enfrentarse a los cambios.

Además, la metodología XP define una serie de buenas prácticas que toda persona que quiera aplicar esta metodología a su desarrollo debería seguir. Son las siguientes:

- **Planificar como un juego:** Esta característica describe que se debe determinar el ámbito de la siguiente evolución del desarrollo rápidamente, combinando el conocimiento del negocio por parte de los clientes y las estimaciones técnicas por parte de los desarrolladores. El cliente podría definir fechas de entrega o determinados aspectos o interfaces, y el técnico se encargaría de organizar y planificar técnicamente el proyecto.
- **Versiones pequeñas:** Sacar nuevas versiones nuevas rápidamente. Si en una versión hay algún error, se deben corregir rápidamente en el siguiente ciclo.
- **Metáforas:** Estas metáforas se refieren a pequeños conceptos resumidos en una frase, y que van a definir una iteración. Esto va a ayudar a tener muy claro el objetivo de la siguiente versión.
- **Diseño simple:** El sistema debe ser diseñado de la manera más simple que se pueda en cada momento. Si se descubre un diseño demasiado complejo, debe eliminarse en la siguiente versión.

- **Pruebas:** Los programadores deben estar continuamente escribiendo test unitarios sobre las nuevas funcionalidades que se van agregando en cada versión. Además los clientes deben proveer a los técnicos de tests que validen sus requisitos.
- **Refactorización:** La refactorización es una tarea de los programadores que significa cambiar código sin cambiar el comportamiento, mejorando la calidad, simplificando y flexibilizando el software.
- **Programación en parejas:** Dos programadores en la misma máquina, para hacer un código más consensuado y con más calidad.
- **Propiedad colectiva:** Cada una de las personas que componen el grupo puede ser el creador de cualquier parte del software. No se restringen ámbitos.
- **Integración continua:** Se debe construir e integrar el software cada día, en cada versión.
- **Máximas horas laborables:** Se define un máximo de 40 horas semanales de trabajo para cada persona. Este rango de horas es un rango que permite tener una cierta credibilidad de que las partes integrantes están en plenas facultades para tomar las mejores decisiones.
- **Cliente participa:** El cliente debe estar muy cercano al desarrollo, participando en él y dando su *feedback* de las evoluciones de los técnicos.
- **Codificación en base a estándares:** Todo el código debe de estar estandarizado con unas metodologías comunes para todos los integrantes del mismo. Esto mejorará la comunicación y la flexibilidad del desarrollo.

Una vez descritas las buenas prácticas, se puede llegar a la conclusión de que es una metodología ágil, que une la innovación que los desarrolladores pueden dar a un producto con la estricta funcionalidad que los clientes exigen.

Aunque XP es una de las metodologías más habituales siempre que se habla de metodologías ágiles, actualmente se están empezando a tener en cuenta otras metodologías ágiles como SCRUM, o Crystal Clear.

Para finalizar con esta sección se va a describir cuales han sido las conclusiones de ambas metodologías de desarrollo aplicadas al ámbito de este proyecto de fin de carrera. Se especificará cual ha sido la elegida, las motivaciones para ello y como se han ido aplicando al proceso de desarrollo de la misma.

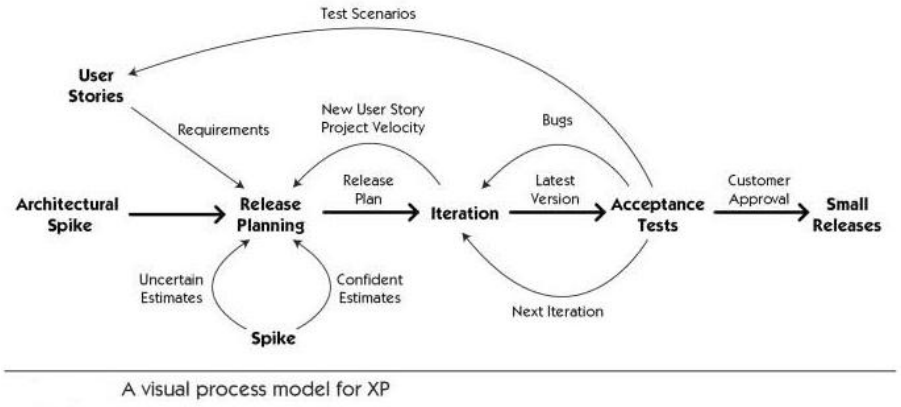

Figura 2.4: Prácticas de modelado XP.

Conclusiones

Como se ha podido observar en la descripción general de cada una de las metodologías anteriores, se puede ver que la metodología ágil XP encajaba mejor en el desarrollo de este PFC. Esta metodología fue elegida por una serie de motivaciones:

- **Innovación:** La metodología XP da más flexibilidad al programador de hacer un desarrollo más ágil y más abierto. Las tecnologías tradicionales son más rígidas y dejan al desarrollador sin la capacidad de improvisar e innovar, capacidades que la metodología XP fomenta en mayor grado. En este PFC se han ido definiendo funcionalidades incrementales, pero dichas funcionalidades fueron creando nuevas ideas a partir de funcionalidades desarrolladas en iteraciones anteriores.

- **Equipo de desarrollo:** Aunque la XP está pensada para desarrollo en parejas, el uso de esta metodología siguiendo el resto de pautas, sigue generando una buena relación entre esfuerzo de desarrollo y resultados. Las tecnologías tradicionales exigirían un esfuerzo mayor que reduciría los resultados.

- **Construcción de una API:** El desarrollo de ActiveInterface construye una API que será usada por otros programadores. Usar XP incluye al cliente o usuario en el proceso de desarrollo y la construcción de una API hace que el *feedback* de los usuarios en cuanto a facilidad de uso o correcciones de errores sea una parte muy importante.

- **Desarrollo incremental:** Es importante para valorar el proceso de desarrollo de ir disponiendo de versiones del producto incrementales. XP beneficia este tipo de desarrollo incremental en base a nuevas funcionalidades.

Por tanto, el enfoque a la innovación, la continua integración de cambios para encajar nuevas funcionalidades, y la continua interacción con el usuario, hicieron que XP fuese la metodología elegida. XP ofrece un contexto ágil, con capacidad de reacción y de adaptación a nuevos requisitos y con baterías de pruebas incrementales por cada una de las nuevas funcionalidades. Además, XP deja un lugar de privilegio a la imaginación para la búsqueda de nuevas funcionalidades que a los usuarios les pudieran ser útiles.

2.3.2. Tecnologías

En este capítulo se van a describir desde las tecnologías de desarrollo más obvias como la orientación a objetos, hasta el lenguaje de programación usado o las distintas librerías usadas.
ActiveInterface tiene dependencias de distintas librerías, con la motivación de la reusabilidad de código y con la idea de centrarse específicamente en las funcionalidades que se querían ofrecer. Usar tecnologías ya existentes facilita la implementación y permite centrarse únicamente en el negocio.

Orientación a objetos

La orientación a objetos es el paradigma de programación más utilizado actualmente en el desarrollo de aplicaciones. Actualmente, todos los lenguajes de programación, pueden ser desarrollados bajo este paradigma, ya sea nativamente como Java, o mediante recubrimientos como C++ sobre C.
El paradigma de la orientación a objetos une la información y la forma de manipular dicha información. Es decir, poniendo como ejemplo los datos personales de una persona, un objeto sería la propia información personal como el nombre, dirección o fecha de nacimiento, y las formas de interactuar con ella, como por ejemplo la forma de saber si dicha persona es mayor de edad. Une la lógica sobre datos con los propios datos.

Los conceptos más importantes de la orientación a objetos son los siguientes:

- **Clase:** El concepto principal en la orientación a objetos es el concepto de *clase*. Una clase está formada por la propia información de la clase o **atributos** y una determinada lógica sobre esos atributos, **métodos**. En el caso de los datos personales, un atributo podría ser la *edad* y un método podría ser *esMayorEdad()*.

- **Objeto:** A partir del concepto de clase, es necesario definir el concepto de objeto. Un objeto es la instanciación de una clase, es decir, un conjunto de datos reales que

siguen las pautas de la clase que usa para definirse.

- **Herencia:** Heredar es la capacidad de construir una clase **clase hija** en base a otra clase previamente construida **clase padre** con la que comparte una determinada información o características. Por ejemplo, la clase triángulo, podría heredar de la clase figura, ya que un triángulo es una figura y todas las figuras tendrán una serie de características comunes.

- **Polimorfismo:** El polimorfismo es otra de las características más importantes de la orientación a objetos y que surge de ciertos problemas que aparecían en la programación procedimental. Está muy ligada a la herencia. Mediante el polimorfismo se pueden crear una serie de acciones que son válidas para todas las clases que heredan de una misma clase padre, permitiendo definir acciones distintas para cada una de ellas o definiendo la misma acción para todas.

C++

El lenguaje de desarrollo elegido para este proyecto de fin de carrera ha sido C++. C++ es un recubrimiento de C que además de otras cosas incluye la orientación a objetos. Fue desarrollado por Dennis Ritchie en 1973 a partir de una versión anterior denominada *B*. Dennis Ritchie desarrollo los principios básicos de C, los sistemas de inclusión, mejoras sobre el manejo de vectores y de punteros, añadiendo el concepto de portabilidad y convirtiéndolo a un lenguaje de desarrollo de alto nivel.
Todas estas mejoras se incluyeron llamado: *The C Programming Language*. Este libro sirvió como estándar de facto para que finalmente en 1989, C se convirtiera en un estándar bajo la publicación del ANSI standard X3J11.
[10]

Entre los años 1980 y 1990, Bjarne Stroustrup empezó a desarrollar lo que sería C++, un refinamiento de C, del que finalmente en 1985 se publicó la primera edición del libro *The C++ programming languaje*. Fué a partir de 1990 cuando el comité ANSI X3J16 comenzó a desarrollar un estándar específico para C++. Este estándar se publicó en 1998.

A partir de esa fecha, C/C++ en conjunto, se convirtieron en el lenguaje más usado para todo tipo de desarrollos, desde desarrollos de bajo nivel (más orientados a C) y desarrollos de alto nivel, usando la potencia de la orientación a objetos de C++. Por esto, se dice que C++ es un lenguaje **multiparadigma**.

Las características más importantes de este lenguaje son:

- **Orientación a objetos:** La posibilidad de orientar el código a objetos permite al programador diseñar las aplicaciones enfocándose a la comunicación entre objetos y no en secuencias de acciones. Esto ayuda a que el código sea más reusable y productivo.
- **Portabilidad:** Aunque se puedan usar distintos compiladores, si se desarrolla bajo los estándares ISO C++ el código C++ podrá ser compilado en cualquier máquina sin hacer ningún cambio. C++ es el lenguaje más usado y portado del mundo.
- **Programación modular:** Un código C++ modular, puede ser compilado por partes y luego *linkado*. Esto hace que no sea necesario recompilar toda la aplicación cuando se produce sólo un cambio en una determinada parte de un módulo.
- **Compatibilidad con C:** C++ es compatible con C. Esto permite que cualquier código C sea fácilmente integrado en código C++.
- **Velocidad:** El resultado de compilación de código C++ es muy eficiente y rápida.

Position Apr 2011	Position Apr 2010	Delta in Position	Programming Language	Ratings Apr 2011	Delta Apr 2010	Status
1	2	↑	Java	19.043%	+0.99%	A
2	1	↓	C	16.162%	-1.90%	A
3	3	=	C++	9.225%	-0.48%	A
4	6	↑↑	C#	7.185%	+2.75%	A
5	4	↓	PHP	6.584%	-3.08%	A
6	7	↑	Python	4.931%	+0.73%	A
7	5	↓↓	(Visual) Basic	4.682%	-1.71%	A
8	11	↑↑↑	Objective-C	4.386%	+2.10%	A
9	8	↓	Perl	1.991%	-1.56%	A
10	10	=	JavaScript	1.513%	-0.96%	A
11	12	↑	Ruby	1.482%	-0.74%	A
12	20	↑↑↑↑↑↑↑↑	Lua	1.035%	+0.51%	A

Figura 2.5: Ranking de los lenguajes más usados en la actualidad.

A continuación se van a describir brevemente las librerías C++ que se han usado para el desarrollo de este PFC.

Apache APR

Apache Portable Runtime es una librería de desarrollo que tiene como objetivo ofrecer una interfaz consistente y confiable para el desarrollo de aplicaciones en C++

independientes de la arquitectura y sistema operativo.
El lenguaje C++ para ciertos aspectos como operaciones con sockets, semáforos, hilos o secciones críticas, accede directamente a la API del sistema operativo, que son las que le proveen de esas primitivas. Cada sistema operativo ofrece esas herramientas de una manera distinta. Esta disformidad de herramientas y formas de uso, hace que determinadas aplicaciones no puedan ser portadas directamente de un sistema operativo a otro (por ejemplo entre Windows o Linux).

Para pasar por alto este tipo de cambios entre sistemas operativos o arquitecturas y no tener que aplicar cambios en el código dependiendo de ello, APR provee de una interfaz común para acceder a las primitivas de cualquier sistema operativo, permitiendo desarrollar código portable y multiplataforma.

Aplicaciones como Mozilla Firefox están desarrolladas usando APR.

log4cxx

Apache log4cxx es un *framework* de traceo de código para C++ desarrollada por el proyecto Apache. Está basada en las funcionalidades que ofrece su homónima en Java log4java. Toda aplicación medianamente grande necesita de un sistema de traceo de errores o de depuración para hacerla más mantenible y depurable en entornos de administración de sistemas.
Log4cxx está centrada en la velocidad, ya que muchos sistemas de traceo infieren lentitud al sistema. Además, log4cxx tiene multitud de funcionalidades, desde sistemas de traceo rotativos, por fechas, envío de trazas de error por email, distintos niveles de traceo, y otras muchas características que se describen en su sitio web *http://logging.apache.org/log4cxx/*.

Boost

Las librerías Boost son un conjunto de librerías generales que ofrecen un *framework* de desarrollo para lenguajes C++. Boost recubre ciertos aspectos de C++, dando al programador una interfaz más sencilla para trabajar con sockets, expresiones regulares, operaciones matemáticas, serialización de datos, trabajo con imágenes, ficheros de configuración, fechas y muchas más funcionalidades que se pueden leer en su página web *http://www.boost.org/doc/libs/*.

Popularmente se dice que las librerías Boost son las que marcan las nuevas funcionalidades que posteriormente serán aceptadas en el nuevo ANSI de C++ que será denominado C++0x.

En este proyecto de fin de carrera sólo se ha usado la parte de serialización de datos de Boost. Esta librería permite la serialización de datos a fichero, y la recuperación posterior de ese dato desde el fichero a memoria. El módulo de persistencia de ActiveInterface tiene como núcleo central esta librería ya que la usa para escribir los mensajes a disco y operar con ellos.

XML

XML ó *eXtensible Markup Language* es un lenguaje de etiquetas desarrollado por el W3C. Se puede definir como un lenguaje de definición de datos que usa etiquetas. Su flexibilidad en cuanto a la posibilidad de definición de etiquetas y la forma en la que puede estructurar la información le ha llevado a convertirse en el estándar para intercambio de información entre aplicaciones, independientemente de plataformas o arquitecturas.

En este proyecto de fin de carrera, XML es usado para definir el fichero inicial de configuración. Además, es importante destacar que muchos de los usuarios que usen *ActiveInterface* usarán XML insertado en un mensaje de texto para definir la información a intercambiar entre clientes.

ActiveMQ-CPP

ActiveMQ-CPP ha sido descrita en capítulos anteriores, por lo que en esta sección, va a ser solamente nombrada.

LLegados a este punto, se han definido en el capítulo 1 todo el ámbito relevante para poder entender el problema y las funcionalidades de este PFC que son descritas en el capítulo 2. En los capítulos siguientes, se hará una descripción informática centrada en la metodología de desarrollo elegida, definiendo requisitos, diseño e implementación.

Para finalizar se comentarán las validaciones y pruebas efectuadas, terminando con unas conclusiones generales y unos trabajos futuros.

Capítulo 3

Descripción Informática

En este capítulo, se explicará todo el proceso de desarrollo a nivel técnico de este PFC. Se entrará en profundidad en todos los aspectos técnicos que intervienen en el desarrollo de un proyecto *software*.

En la especificación de requisitos se explicarán los requisitos previos que se analizaron y que coinciden con los objetivos mostrados en el capítulo anterior. En la sección de diseño se especificarán los casos de uso, las arquitecturas tanto lógicas como físicas y se describirán los diseños del núcleo y de la API de ActiveInterface. En la sección siguiente se introducirán los detalles más importantes de la implementación, para más tarde en las pruebas describir la batería de pruebas que ha sido implementada para verificar el correcto funcionamiento de las funcionalidades ofrecidas.

Gracias a esta descripción informática el lector podrá validar los objetivos conseguidos a partir de los requisitos especificados. Podrá valorar la extensibilidad, las metodologías y patrones aplicados al diseño. Además, podrá verificar la calidad del *software* y la estabilidad gracias a la descripción en profundidad de los capítulos de implementación y pruebas.

3.1. Especificación de requisitos

La captura de requisitos comprende todas las tareas relacionadas con la determinación de las necesidades o de las condiciones a satisfacer por un *software*, tomando en cuenta los diversos requisitos de los clientes (o usuarios), que pueden entrar en conflicto entre ellos.

Un requisito podría ser una descripción formal y detallada de una función del sistema,

una declaración de alto nivel de una funcionalidad del sistema o simplemente una restricción del mismo. [11]

3.1.1. Requisitos funcionales

Se puede decir que un requisito funcional son las acciones o restricciones que debe realizar o tener un sistema. Enfocándolo a este PFC, se han dividido las capturas de requisitos agrupándolas en torno a las principales funcionalidades del sistema.

- **RF.1 Proporcionar una topología de estructuras flexible para el usuario.**
  - **RF.1.1** Proporcionar estructura que abstraiga el concepto de conexión.
  - **RF.1.2** Proporcionar estructura intermedia o enlace que asocie conexiones para permitir agruparlas.
  - **RF.1.3** Proporcionar estructura de servicio que asocie estructuras intermedias.
  - **RF.1.4** Permitir definir propiedades por defecto a nivel de estructura intermedia.
  - **RF.1.5** Permitir asociar distintas conexiones a una determinada estructura intermedia.
  - **RF.1.6** Permitir asociar un enlace a distintas conexiones.
  - **RF.1.7** Permitir asociar un servicio a varios enlaces.
  - **RF.1.8** Permitir definir múltiples instancias de cada una de las estructuras.
  - **RF.1.9** Permitir al usuario conocer el origen y el canal por el que ha recibido el mensaje.
  - **RF.1.10** Permitir definición de entidades como cadenas de caracteres.
  - **RF.1.11** Proporcionar una cola intermedia que prevenga de inundaciones y perdidas de mensajes.
  - **RF.1.12** Proporcionar una cola intermedia por conexión real.
  - **RF.1.13** Establecer un algoritmo de retardos para evitar llenados de la cola intermedia.
  - **RF.1.14** Permitir mandar un mismo mensaje y replicarse a distintas conexiones.
- **RF.2 Requisitos de la API**

- **RF.2.1** Permitir creación de estructura que abstraiga el concepto de conexión.
- **RF.2.2** Permitir creación de estructura intermedia que asocie conexiones.
- **RF.2.3** Permitir creación de estructura de servicio.
- **RF.2.4** Permitir definir asociaciones entre conexiones y enlaces.
- **RF.2.5** Permitir definir asociaciones entre enlaces y servicios.
- **RF.2.6** Permitir obtener el servicio asociado a una conexión.
- **RF.2.7** Permitir obtener el servicio asociado a un enlaces.
- **RF.2.8** Permitir obtener el enlace asociado a una conexión.
- **RF.2.9** Permitir obtener los enlaces asociados a un servicio.
- **RF.2.10** Permitir obtener las conexiones asociadas a un enlace.
- **RF.2.11** Permitir obtener las conexiones asociadas a un servicio.
- **RF.2.12** Permitir eliminar las asociaciones entre servicio y enlace.
- **RF.2.13** Permitir eliminar las asociaciones entre enlace y conexión.
- **RF.2.14** Permitir eliminar enlaces.
- **RF.2.15** Permitir eliminar servicios.
- **RF.2.16** Permitir eliminar conexiones.
- **RF.2.17** Permitir parar conexiones.
- **RF.2.18** Permitir reiniciar conexiones.
- **RF.2.19** Permitir enviar mensajes a servicios.
- **RF.2.20** Permitir enviar respuestas a conexiones.
- **RF.2.21** Permitir obtener la posición de la cola intermedia donde se almacena el mensaje.
- **RF.2.22** Permitir tener un único punto de entrada para la recepción de mensajes.
- **RF.2.23** Permitir obtener el estado de la conexión.
- **RF.2.24** Permitir obtener cuando la conexión ha sido interrumpida.
- **RF.2.25** Permitir obtener cuando la conexión se ha restablecido.
- **RF.2.26** Permitir conocer al usuario cuando un mensaje no puede ser enviado.
- **RF.2.27** Permitir parada total de la librería.

- **RF.2.28** Permitir definir la cadena de conexión con el broker respetando las posibilidades de la librería nativa.
- **RF.2.29** Permitir definir el destino de la cola o tópico.
- **RF.2.30** Permitir definir si los mensajes enviados al broker son persistentes o no.
- **RF.2.31** Permitir definir si la conexión es un tópico o una cola.
- **RF.2.32** Permitir definir selectores de tópicos.
- **RF.2.33** Permitir definir si una conexión es duradera o no.
- **RF.2.34** Permitir definir si el *acknowledge* es automático o no.
- **RF.2.35** Permitir especificar un usuario y contraseña para conectar a una cola o tópico.
- **RF.2.36** Permitir especificar un identificador de cliente unívoco para conexiones duraderas.
- **RF.2.37** Permitir especificar un certificado para conexiones *SSL*.

▪ **RF.3 Requisitos de abstracción de mensajes**

- **RF.3.1** Permitir la creación de una estructura de datos propia donde alojar contenido.
- **RF.3.2** Permitir definir parámetros que componen el mensaje.
- **RF.3.3** Permitir definir parámetros de tipo entero.
- **RF.3.4** Permitir definir parámetros de tipo real.
- **RF.3.5** Permitir definir parámetros de tipo cadenas de caracteres.
- **RF.3.6** Permitir definir parámetros de cadenas de bytes.
- **RF.3.7** Permitir definir propiedades que componen el mensaje.
- **RF.3.8** Permitir definir propiedades de tipo entero.
- **RF.3.9** Permitir definir propiedades de tipo real.
- **RF.3.10** Permitir definir propiedades de tipo cadenas de caracteres.
- **RF.3.11** Permitir definir mensajes como texto.
- **RF.3.12** Permitir definir propiedades en un mensaje de texto.
- **RF.3.13** Permitir eliminar propiedades.
- **RF.3.14** Permitir eliminar parámetros.

- **RF.3.15** Permitir eliminar todas las propiedades.
- **RF.3.16** Permitir eliminar todos los parámetros.
- **RF.3.17** Permitir asociar cada propiedad con una clave.
- **RF.3.18** Permitir asociar cada parámetro con una clave.
- **RF.3.19** Permitir extraer cada propiedad con una clave.
- **RF.3.20** Permitir extraer cada parámetro con una clave.
- **RF.3.21** Permitir especificar la prioridad del mensaje.
- **RF.3.22** Permitir especificar el tiempo de vida del mensaje.
- **RF.3.23** Permitir extraer el origen del mensaje recibido.
- **RF.3.24** Permitir especificar el destino del mensaje.
- **RF.3.25** Permitir extraer si el mensaje recibido requiere respuesta.
- **RF.3.26** Permitir especificar que el mensaje a mandar requiere respuesta.
- **RF.3.27** Permitir recorrer la lista de parámetros de manera iterativa.
- **RF.3.28** Permitir recorrer la lista de propiedades de manera iterativa.
- **RF.3.29** Permitir obtener el tipo del mensaje.
- **RF.3.30** Permitir especificar el tipo del mensaje.

▪ **RF.4 Requisitos de inicialización**

- **RF.4.1** Permitir definir un fichero de configuración XML de inicialización.
- **RF.4.2** Permitir definición de conexiones.
- **RF.4.3** Permitir definición de enlaces.
- **RF.4.4** Permitir definición de servicios.
- **RF.4.5** Permitir definición de propiedades por defecto.
- **RF.4.6** Permitir asociación entre entidades.
- **RF.4.7** Permitir definir la cadena de conexión con el broker respetando las posibilidades de la librería nativa.
- **RF.4.8** Permitir definir el destino de la cola o tópico.
- **RF.4.9** Permitir definir si los mensajes enviados al broker son persistentes o no.
- **RF.4.10** Permitir definir si la conexión es un tópico o una cola.

- **RF.4.11** Permitir definir selectores de tópicos.
- **RF.4.12** Permitir definir si una conexión es duradera o no.
- **RF.4.13** Permitir definir si el *acknowledge* es automático o no.
- **RF.4.14** Permitir especificar un usuario y password para conectar a una cola o tópico.
- **RF.4.15** Permitir especificar un identificador de cliente unívoco para conexiones duraderas.
- **RF.4.16** Permitir especificar un certificado para conexiones *SSL*.

- **RF.5 Requisitos de persistencia:**
  - **RF.5.1** Permitir persistencia en origen.
  - **RF.5.2** Permitir almacenar mensajes en cola de mensajes.
  - **RF.5.3** Permitir almacenar mensajes en fichero cuando la cola está llena.
  - **RF.5.4** Permitir recuperar mensajes almacenados cuando sea posible enviarlos.
  - **RF.5.5** Ofrecer un fichero de datos donde se almacenan los mensajes no enviados.
  - **RF.5.6** Ofrecer un fichero de configuración donde se almacena el último mensaje enviado.
  - **RF.5.7** Todos los mensajes enviados por el usuario son almacenados.
  - **RF.5.8** Borrado dinámico del fichero de persistencia cuando los datos enviados han superado un umbral.
  - **RF.5.9** Ofrecer un umbral a partir del cual los datos ya enviados del fichero de persistencia son borrados.
  - **RF.5.10** Permitir activación o desactivación de persistencia.
  - **RF.5.11** Permitir recuperación de mensajes no enviados ante un cierre repentino de la aplicación.

- **RF.6 Requisitos generales.**
  - **RF.6.1** Permitir diferentes de protocolos de conexión (Multicast, UDP, TCP).
  - **RF.6.2** Permitir conexiones usando SSL.
  - **RF.6.3** Ofrecer trazas de la aplicación para depuración y control de errores.
  - **RF.6.4** Ofrecer entorno de compilación en Linux.
  - **RF.6.5** Ofrecer entorno de compilación en Windows.

3.1.2. Requisitos no funcionales

Los requisitos no funcionales son aquellos requisitos que no se refieren específicamente a las funciones del sistema, sino que son requisitos generales que ayudar a juzgar el modo de operación de un sistema.

- **Flexibilidad:** Por el tipo de software en el que se enmarca este PFC, la flexibilidad es un requisito imprescindible. Debe ser un software flexible, que se adapte a las necesidades de los posibles desarrolladores. Flexible en sus estructuras y en su manera de operar con él.

- **Confiabilidad:** ActiveInterface debe proveer a los usuarios de una gran confiabilidad ya que los mensajes pueden ser críticos. Establecer mecanismos de persistencia y de tolerancia a fallos.

- **Mantenibilidad:** La API debe ser flexible para que se pueda mantener y mejorar fácilmente. Este es un detalle muy importante ya que intentar crear una comunidad de desarrollo es muy difícil, y para que surjan nuevos desarrolladores el código actual debe ser mantenible y fácil de leer.

- **Portabilidad:** ActiveInterface debe de ser portable en, al menos, plataformas Windows y Unix.

- **Eficiencia y tiempos de respuesta:** Debe de ser eficiente, retardos grandes en la producción de mensajes pueden hacer que mensajes antiguos ya no sean válidos. ActiveInterface debe limitar al mínimo el retardo desde que el usuario envía un mensaje hasta que lo procesa y lo envía.

- **Utilización de memoria:** Al poder arrancarse en entornos embebidos con pocos recursos hardware, la utilización de la memoria debe ser limitada. El desarrollo del software debe estar protegido frente a agujeros de memoria.

- **Seguridad:** Debe proveer la manera de establecer conexiones seguras.

- **Documentación:** Debe proveer a los usuarios de una documentación clara y con mucho detalle, que permita a los usuarios utilizarla y a los nuevos desarrolladores aportar.

- **Testeabilidad:** Proveer de una batería de pruebas que pruebe funcionalidad.

3.2. Análisis

En este capítulo se van a describir los aspectos más importantes que se han tenido en cuenta para diseñar este PFC.

En los primeros puntos se describirán los diagramas de casos de uso, que guiarán el análisis y las posteriores etapas. Con un detalle más técnico, se describirán en profundidad los diagramas de clases principales de los que se compone ActiveInterface.

Desde el punto de vista arquitectural, se describirá tanto la arquitectura a nivel lógico como a nivel físico. A nivel lógico se podrá entender como funciona ActiveInterface desde el punto de vista del proceso que lo usa. Gracias a la arquitectura física, se podrá enmarcar este proceso en un diagrama más general, más enfocado a las posibles topologías de red y a determinados aspectos hardware.

Todos estos puntos ayudarán a tener una visión superficial de ActiveInterface, que será profundizada en el capítulo de implementación, donde se recogerán los aspectos más importantes en dicho nivel.

3.2.1. Casos de uso

Los casos de uso vienen muy marcados por los requisitos expuestos en la sección anterior. Se han englobado en el mismo diagrama los casos de uso que tienen características comunes.

Caso de uso general

Como se puede ver en la figura 3.1, los casos de uso principales son los que engloban la funcionalidad de envío de mensajes, respuestas y la recepción de nuevos mensajes. Ambas acciones incluyen un caso de uso denominado *encolar mensaje* que encola el mensaje en la cola intermedia.

Caso de uso creación de entidades

En la figura 3.2 se pueden observar los diagramas de casos de uso para la creación de entidades que corresponden con los requisitos de creación tanto por fichero de inicialización

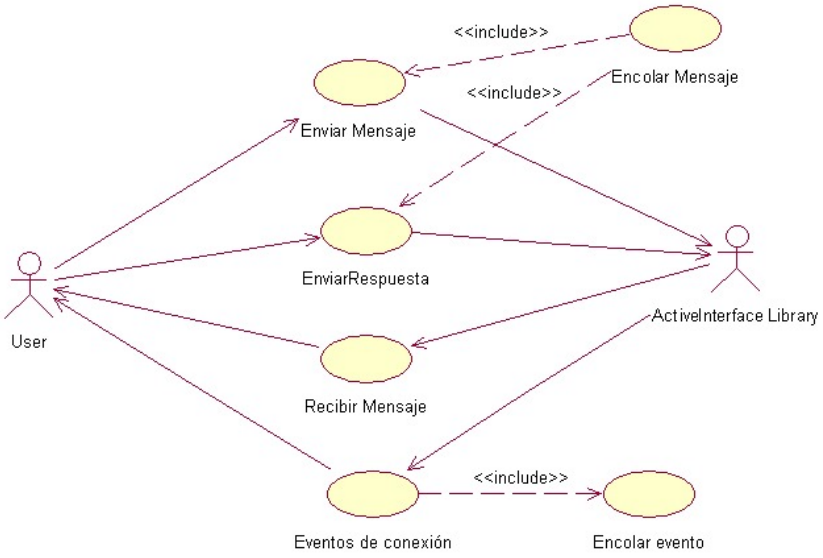

Figura 3.1: Diagrama de casos de uso de envío y recepción de mensajes.

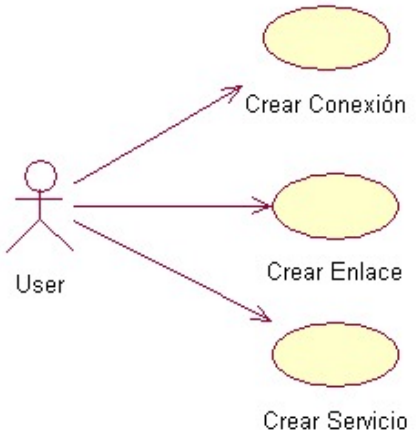

Figura 3.2: Diagrama de caso de uso de creación de entidades.

como mediante la API. El borrado de una entidad obliga a conocer el identificador de esa entidad, pero el caso de uso de obtención de información de las entidades está recogido en casos de uso posteriores.

Caso de uso de borrado de entidades

En la figura 3.3 se puede observar los casos de uso para los borrados de entidades.

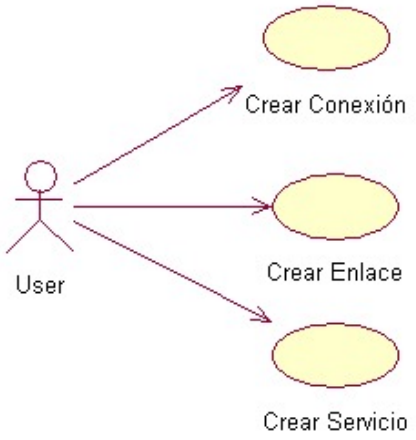

Figura 3.3: Diagrama de caso de uso de borrado de entidades.

Caso de uso de asociación de entidades

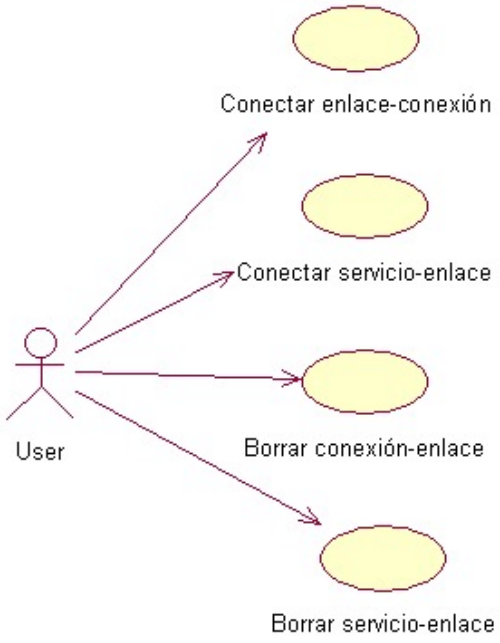

Figura 3.4: Diagrama de caso de uso de asociación entre entidades.

En la figura 3.4 se muestra el diagrama de casos de uso mediante el cual se especifica la necesidad de poder eliminar y crear asociaciones entre las distintas entidades.

3.2.2. Diagrama de clases

Los diagramas de clases son diagramas estáticos que se ubican en el análisis y diseño de un sistema software. Muestran el conjunto de clases estáticas que forman la estructura de un sistema y las asociaciones entre ellas.

Los diagramas de clases describen los atributos y los comportamientos, y son muy útiles para ilustrar las relaciones entre las clases, como las generalizaciones, asociaciones o agregaciones.

A continuación, en la figura 3.5, se expone el diagrama de clases general que permite ver toda la estructura de clases para luego poder ir estudiándola funcionalmente en los módulos principales que la componen.

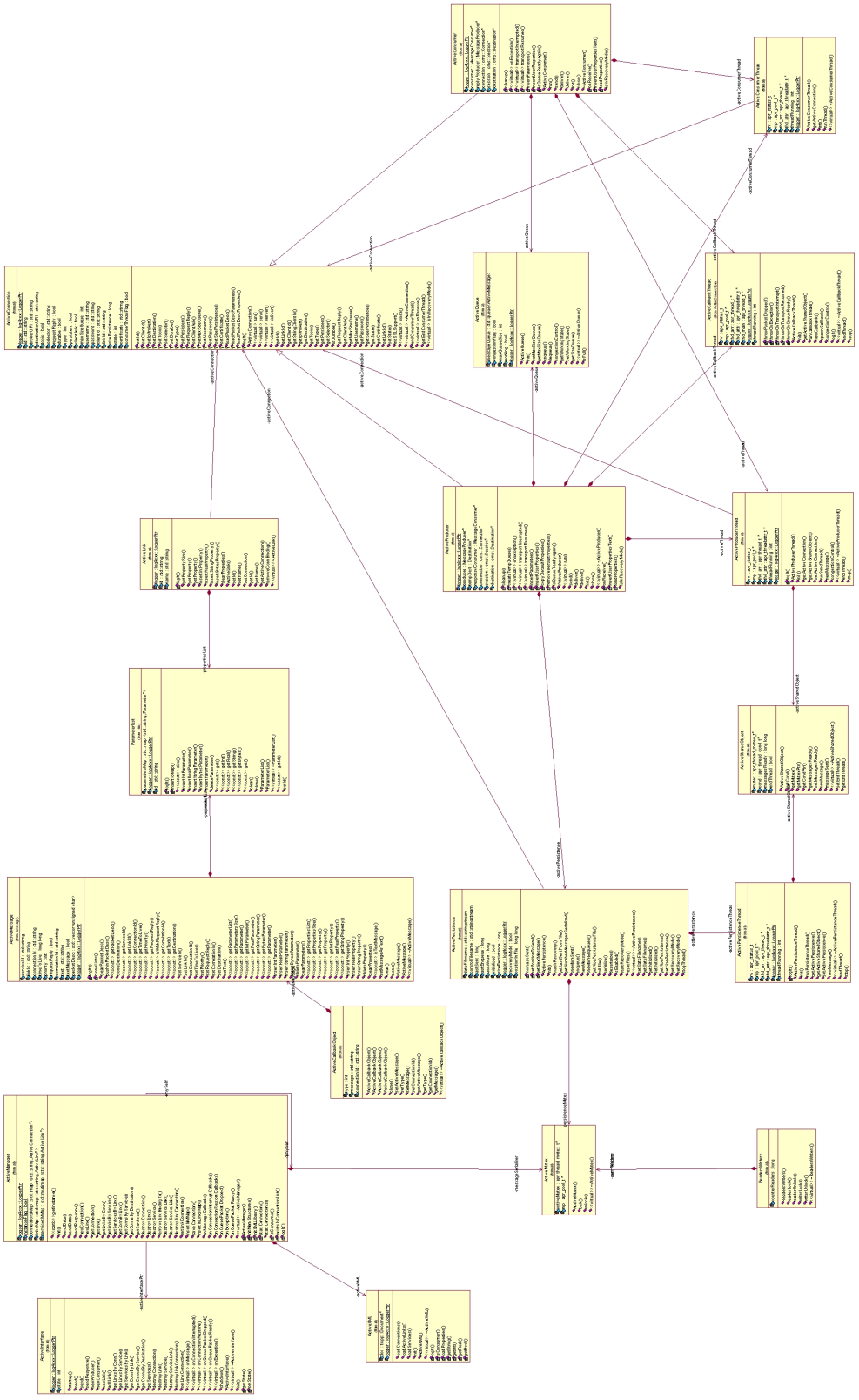

Figura 3.5: Diagrama de clases general.

Interfaz y Mánager

Como se puede apreciar en el diagrama de clases, existe una clase que provee de las funciones al usuario llamada ActiveInterface. Esta clase implementa el patrón fachada, y únicamente actúa de paso para la clase que provee la lógica de la aplicación denominada ActiveManager.

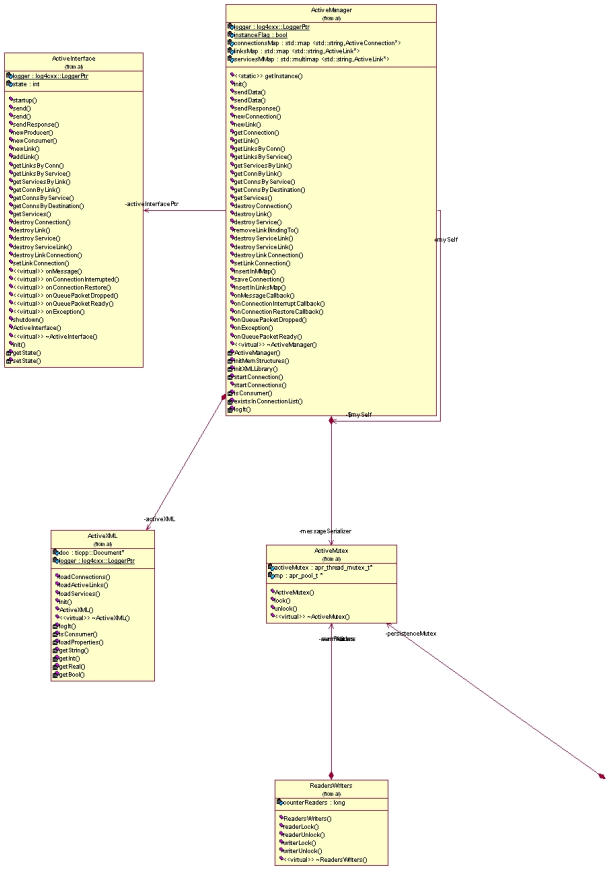

Figura 3.6: Diagrama de clases para la interfaz con el usuario.

ActiveManager implementa el patrón *singleton* para actuar como punto único de entrada e impedir varias instanciaciones de la misma. Además, se puede apreciar como esta clase implementa el algoritmo de lectores y escritores. Este algoritmo, en inglés *readers and writers*, bloquea las diferentes acciones que conllevan una modificación de la topología de red, mientras se está utilizando la misma. Además, esta clase serializa los mensajes al entorno de usuario, haciendo que el usuario no tenga que tener en cuenta el soporte multihilo.

Además, se puede apreciar como es el *manager* el que se encarga de la inicialización de

la librería mediante XML usando la clase ActiveXML.

Mensaje

Otra de las partes que es importante destacar son las clases relacionadas con ActiveMessage.

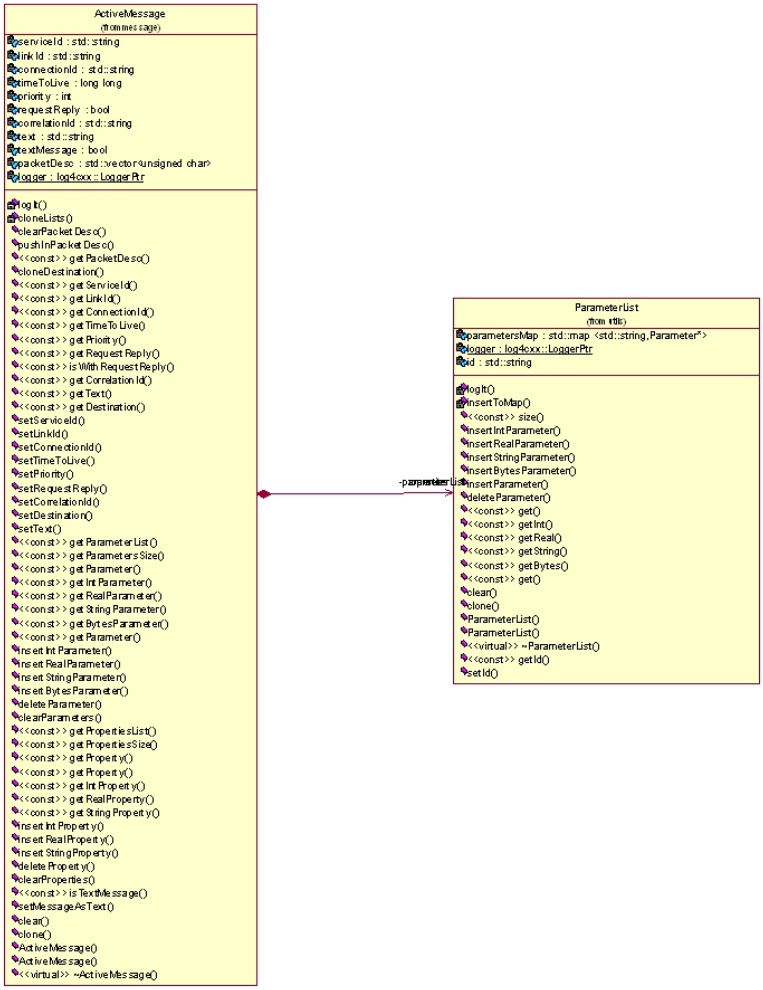

Figura 3.7: Diagrama de clases para las clases relacionadas con el mensaje.

Como se puede ver en la figura anterior, la clase mensaje provee de los métodos para trabajar con cualquier tipo de dato y está compuesta por una clase que representa una lista de parámetros.

Conexión

Aunque se podría entrar más en profundidad, del resto de clases es relevante destacar la clase ActiveConnection. De esta clase heredan todas las posibles conexiones que están desarrolladas, como JMS, o otras pendientes de desarrollar como clases para conexiones mediante puerto serie o mediante protocolos de red como TCP o UDP.

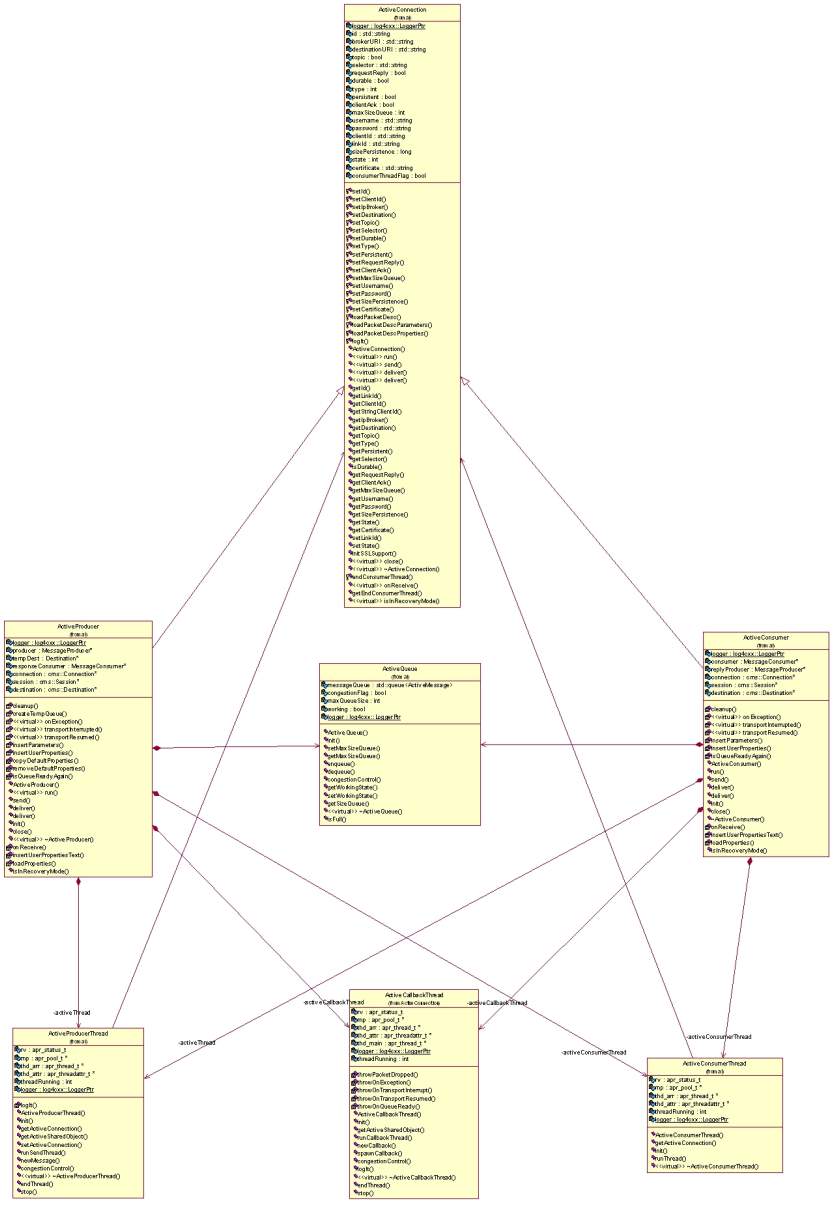

Figura 3.8: Diagrama de clases para las clases relacionadas con la conexión.

Sobre la figura anterior, se puede observar como las clases productoras y consumidoras

de mensajes heredan de la clase padre ActiveConnection. Además, gracias a esta figura, se puede apreciar el concepto de cola intermedia introducido en capítulos anteriores, con la clase ActiveQueue.

Cada clase que hereda de la clase conexión, es implementada en un hilo que permite al usuario poder pararlo, para parar de consumir o producir. Además, como ya se ha apuntado, todos los mensajes que son enviados al espacio de usuario se envían a través de la clase ActiveCallback y de su hilo ActiveCallbackThread.

3.2.3. Arquitectura lógica

Mediante la arquitectura lógica se va a poder entender desde un punto de vista lógico, las capas de trabajo de una aplicación que use ActiveInterface. El siguiente diagrama ayudará a explicarlas.

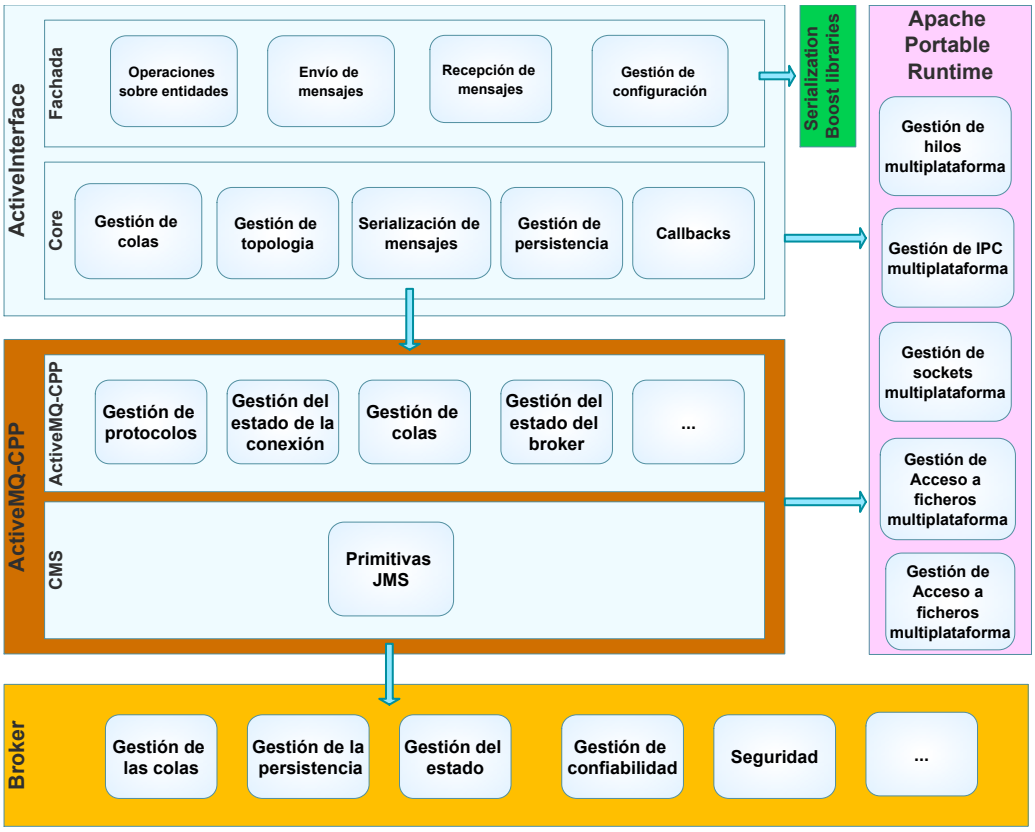

Figura 3.9: Diagrama de arquitectura lógica de una aplicación basada en ActiveInterface.

Aunque gracias a capítulos anteriores se podía tener una visión sobre la arquitectura lógica, este diagrama ayuda a representarla perfectamente.

Como se puede ver, ActiveInterface es el punto de entrada. A través de su fachada se realizan todas las posibles acciones del usuario como pueden ser las modificaciones de la topología de red, o operaciones sobre mensajes. Una vez el usuario ha mandado el evento, el núcleo de ActiveInterface realizará sus tareas de gestión de colas (en caso de operar sobre mensajes), cambios en la topología de red, o gestión de la configuración (ficheros XML) internamente.
Como se aprecia en el gráfico, ActiveInterface utiliza *boost*[12] para la serialización de objetos, y además usa *Apache Portable Runtime* para ofrecer una programación multiplataforma en el acceso a primitivas del sistema. ActiveMQ-CPP también usa *Apache Portable Runtime* para ofrecer portabilidad.

Una vez ActiveInterface ha concluido su flujo de trabajo, ésta accede a la librería ActiveMQ-CPP, que a través de su fachada, ofrece una serie operaciones, que incluyen todas las gestiones de estado de conexión, protocolo de conexión y muchas más funcionalidades para la conexión correcta con el *broker*.
Una parte importante de ActiveMQ-CPP es *CMS*. *CMS* es la implementación en C++ de las primitivas de conexión con los brokers a través de JMS.

Una vez explicada la arquitectura lógica a nivel de proceso, se procederán a explicar las diferentes posibilidades de arquitecturas físicas.

3.2.4. Arquitectura física

La arquitectura física se va a centrar en explicar, a nivel físico como se podrían organizar los distintos procesos que compondrían una arquitectura real.

Arquitecturas centralizadas

La primera de las arquitecturas posibles es la denominada *Arquitectura centralizada*. Se denominan arquitecturas centralizadas a las arquitecturas que tienen un centro común a todos los clientes. Este centro común se convierte, por tanto, en el punto más crítico pues todos los demás elementos de la arquitectura dependen de él.

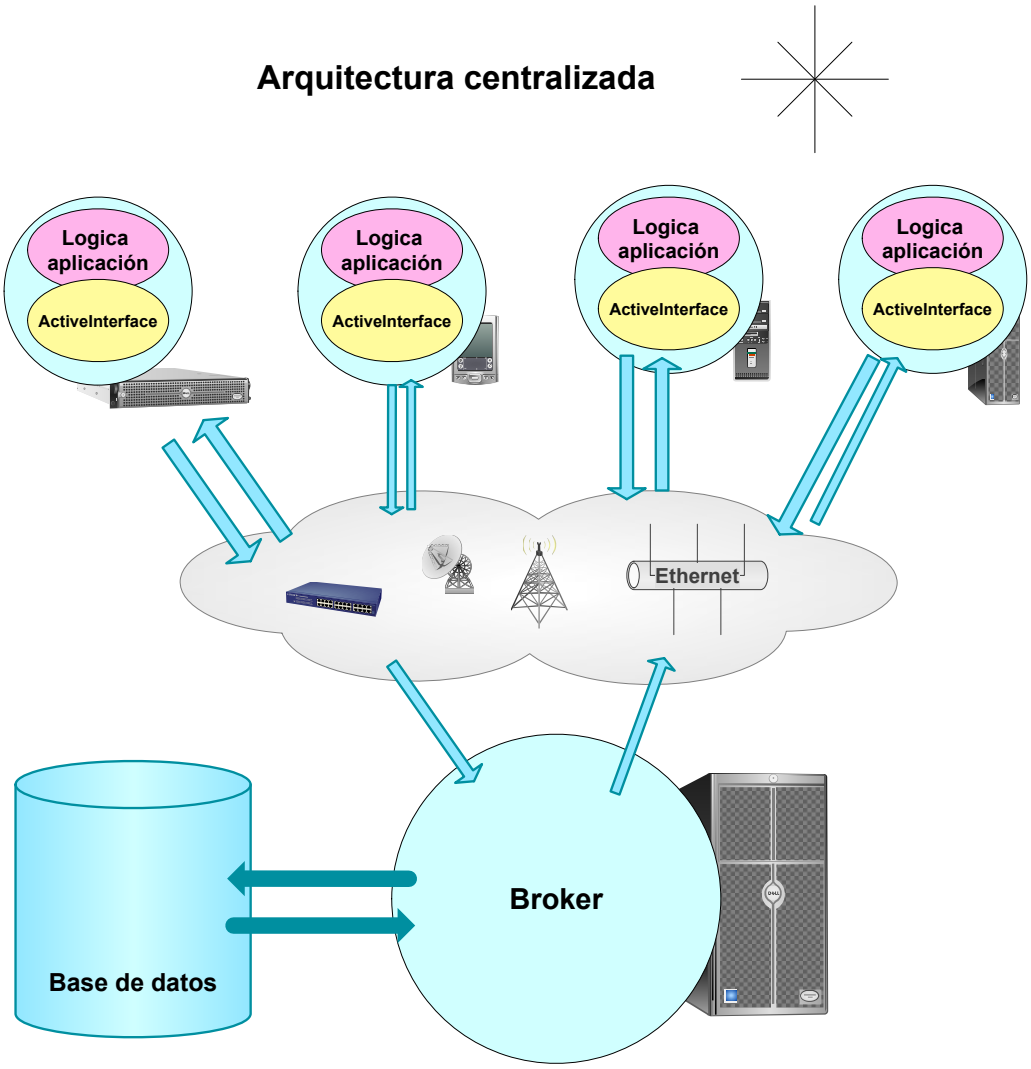

Figura 3.10: Diagrama de arquitectura física centralizada.

En las arquitecturas centralizadas, el *broker* es el elemento más crítico del sistema, pues es el que recibe y reenvía los mensajes desde y hacia los clientes.
En los sistemas de mensajería, las arquitecturas centralizadas son las más usuales. Un broker, configurado o no con alta disponibilidad, se convierte en el centro de nuestra red, por el que pasan todos los datos que se generan desde los clientes.

Estas arquitecturas centralizadas tienen desventajas desde el punto de vista de rendimiento y disponibilidad. En cuanto a disponibilidad es un problema ya que centralizar tu lógica te hace tener un punto único de fallo. En cuanto a rendimiento, tener demasiada lógica agrupada supone estar más cerrado al hardware que en otras arquitecturas, además de problemas de rendimiento ante avalanchas de tráfico.

Aunque las arquitecturas basadas en mensajes y en especial el *broker* ActiveMQ

permite escalar fácilmente agregando nuevos nodos y formando un clúster, ActiveInterface ayuda a descentralizar las arquitecturas, permitiendo que los clientes embeban un *broker* convirtiéndose parte de una arquitectura descentralizada de brokers.

Arquitecturas distribuidas

Una arquitectura distribuida es una arquitectura en la que no hay un elemento central que acapara más lógica que otro, es decir, la lógica del sistema está distribuida entre todos los nodos del sistema.

Cuando un cliente quiere mandar un mensaje común a un grupo de clientes, lo más común es usar el tipo de cola llamada *tópico*. Mediante este tipo de colas, el mensaje que se envía a este tópico, es reenviado a todos los clientes que estén asociados a ese tópico.

Teniendo la idea anterior en mente, se puede ver claramente, que el problema puede venir cuando queremos descentralizar la arquitectura, repartiendo el broker en los propios clientes. En ese momento, el usuario tendría que programar que el envío de un mensaje vaya a diferentes brokers, repitiendo el envío y inflexibilizando la arquitectura, pues tendría que especificarse a nivel de código.

En este punto es donde más puede ayudar ActiveInterface. Por ejemplo, mediante servicios, se puede definir un conjunto de clientes para un determinado flujo de mensajes. Simplemente haciendo una asociación lógica de clientes, bajo un identificador, y de forma totalmente configurable y flexible, se podrían descentralizar los brokers. Con un simple envío, ActiveInterface abstraería al usuario de esa replicación del mensaje, de manera similar a cómo lo haría el *broker*.

Como se ve en la figura 3.11, para formar una arquitectura distribuida es necesario distribuir los brokers, embebiéndolos en cada uno de los clientes. Es evidente que las arquitecturas distribuidas son más complejas, pero con la ayuda de ActiveInterface, la complejidad se reduce, pudiendo agrupar, modificar y añadir nuevos clientes.

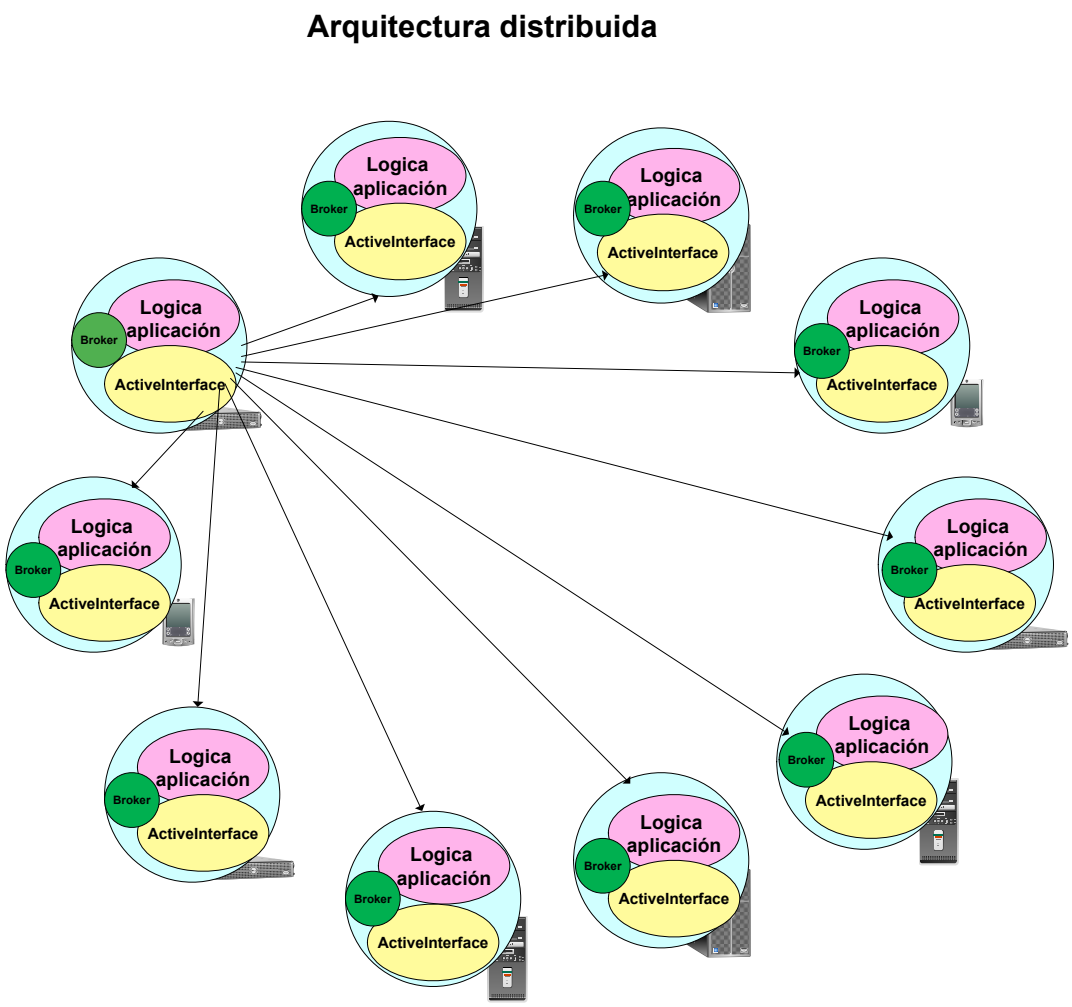

Figura 3.11: Diagrama de arquitectura física distribuida.

3.3. Implementación

En este capítulo se tratarán de enfocar todos los conceptos reflejados en la sección anterior hacia entornos puramente más técnicos. Se comentarán todas los detalles importantes, o con una determinada complejidad, que se han tomado desde el punto de vista de la implementación.

3.3.1. Entorno de desarrollo

El entorno de desarrollo para ActiveInterface depende de la plataforma para la que se quiera compilar la librería.

En entornos Windows, ActiveInterface usa Visual Studio. Visual Studio es una suite de desarrollo de aplicaciones para sistemas operativos Windows desarrollada por Microsoft. Soporta distintos tipos de lenguajes, desde C#, Visual Basic o Visual C++.

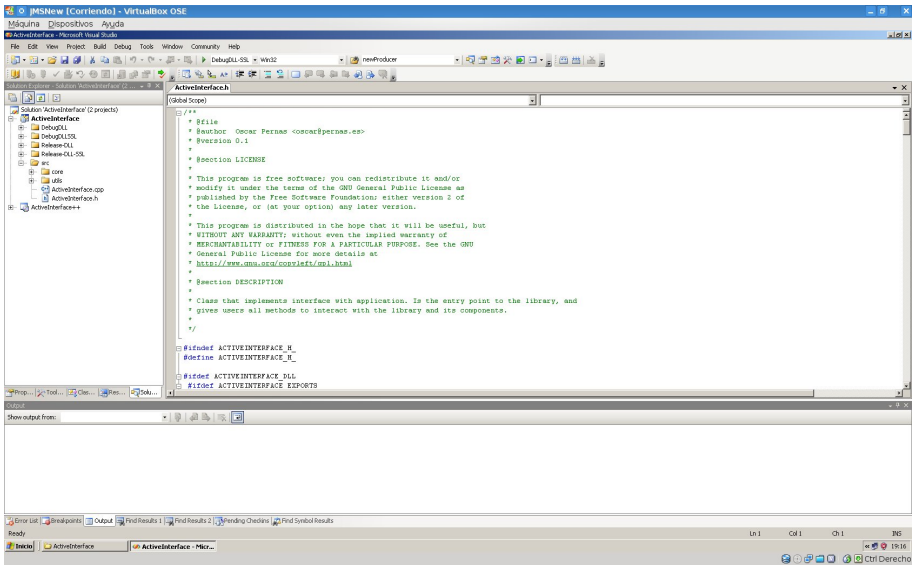

Figura 3.12: Captura del proyecto ActiveInterface en Visual Studio 2005.

ActiveInterface provee de un proyecto de configuración para Visual Studio, el cual cuando es cargado permite al usuario modificar las rutas de las librerías de dependencias. Con el proyecto cargado y modificadas las rutas, sólo es necesario elegir la opción de compilar el proyecto. ActiveInterface aporta ficheros de configuración para las versiones de Microsoft Visual Studio C++ 2005 y 2008.

Además de esto, ActiveInterface tiene un instalador que permite al usuario mediante unos simples clicks instalar todas las dependencias. Muy recomendable para entornos de producción.

En entornos Linux, ActiveInterface ha sido desarrollado usando Eclipse IDE C++. Eclipse es un entorno de desarrollo *Software libre* y multiplataforma, que aunque tradicionalmente es usado para el desarrollo de aplicaciones Java, ha sido adaptado para desarrollos en multitud de lenguajes de programación.

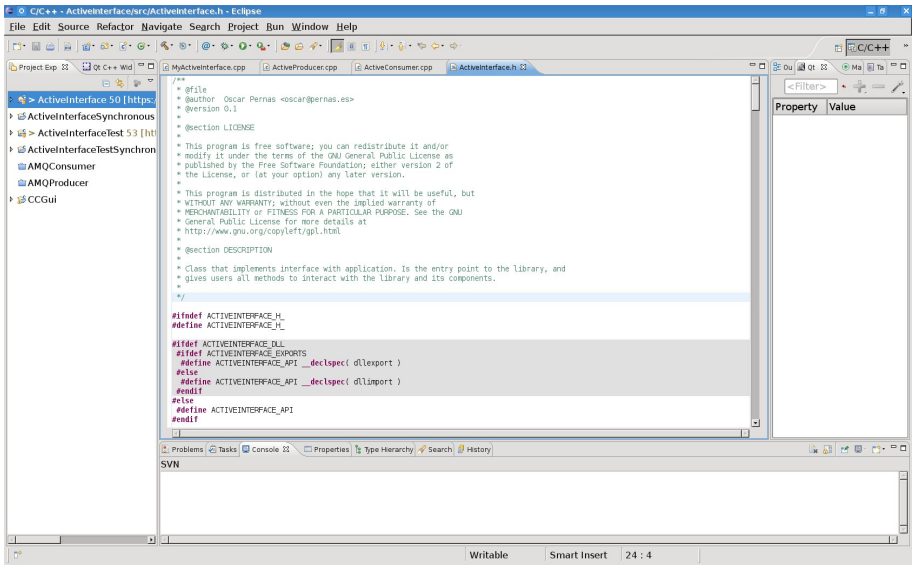

Figura 3.13: Captura del proyecto ActiveInterface bajo Eclipse IDE.

La arquitectura de eclipse basada en *plugins* permite a los desarrolladores fabricar nuevos módulos e integrados con Eclipse. Esto convierte a Eclipse en un entorno de desarrollo extensible y con capacidades para controlar sistemas externos, lo que le convierte en muy intuitivo y útil.

3.3.2. Distribución de los paquetes

Uno de los primeros conceptos a entender a nivel de implementación es como se han distribuido los paquetes (en C++ son exclusivamente directorios). La distribución de los paquetes se ha hecho en base a funcionalidad. La siguiente imagen es una captura real de los paquetes en los que se ha dividido el desarrollo.

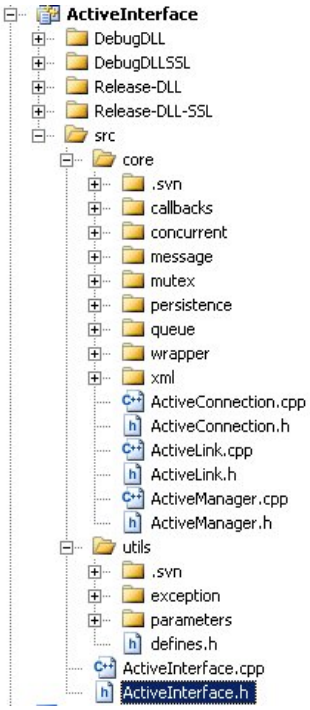

Figura 3.14: Distribución de los paquetes de ActiveInterface.

A continuación se van a describir brevemente cada una de las carpetas que se pueden apreciar en la figura anterior.

- **Core:** Es el paquete central, en él se encuentran las principales funcionalidades.
  - **Callbacks:** En este paquete se encuentran todas las clases relacionadas con los callbacks hacia el entorno de usuario.
  - **Concurrent:** En este paquete se encuentran las clases que implementan el patrón de concurrencia de *lectores y escritores.*
  - **Message:** En este paquete se encuentran las clases que manipulan los mensajes *ActiveMessage.*
  - **Mutex:** Este paquete implementa una abstracción de mecanismos de IPC.
  - **Persistence:** En este paquete se han implementado las clases que recogen toda la lógica relativa a la persistencia de datos.
  - **Queue:** Paquete que implementa el patrón de cola de acceso concurrente.
  - **Wrapper:** Son las clases que encapsulan las funcionalidades de ActiveMQ-CPP.

    - **XML:** Implementa las clases que recogen las funcionalidades de inicialización por XML.

- **Utils:** En esta carpeta principal se encuentran agrupadas utilidades varias.

    - **Exception:** Clase que implementa las excepciones de toda la librería.
    - **Parameters:** Paquete que encapsula una lista de parámetros multitipo.

Además, en el paquete raíz se encuentran las clases ActiveManager y ActiveInterface que representan el núcleo de la librería junto con la clase ActiveConnection.

A parte de la distribución de paquetes, es importante destacar las diferentes opciones de precompilación de la que dispone ActiveInterface. Como se puede ver en la parte superior de la imagen, ActiveInterface se puede compilar como librería estática (.lib o .a) o como dinámica (.dll o .so). Esta elección depende de como el usuario quiera que se genere el ejecutable de su aplicación.

Además, ActiveInterface provee de variables de precompilación para agregar seguridad a los sistemas de comunicaciones.
Si en los requisitos del cliente es necesario proporcionar seguridad bajo SSL a las comunicaciones, es necesario compilar ActiveInterface activando ActiveInterface.

3.3.3. Principales detalles técnicos

En esta sección se van a detallar los principales detalles técnicos que han aplicado a este PFC. Se han organizado en distintas subsecciones, en cada una de ellas se explicará el problema que resuelve y se detallará la forma en las que se ha llevado a cabo la resolución.

API

Uno de los principales requisitos no funcionales que se ha querido satisfacer es la *facilidad de uso*. Ofrecer al usuario una API fácil de usar, conlleva una abstracción mayor, y por tanto una pérdida de funcionalidad. ActiveInterface ha intentado modelar esta abstracción para que a nivel de implementación, el usuario escriba un código claro y con muy pocas líneas de código.

Para facilitar el uso, y siguiendo el patrón de desarrollo software denominado *fachada* se decidió proveer de una clase sobre la que se tendría que aplicar herencia para su uso.

```
class MyActiveInterface: public ActiveInterface
```

Al heredar de la clase ActiveInterface, esta te obliga a implementar unos serie de métodos (virtuales puros). Estos métodos serán invocados desde la librería por ejemplo para la recepción de mensajes.

```
virtual void onMessage(const ActiveMessage& activeMessage)
  abstract;

virtual void onConnectionInterrupted (std::string& connectionId)
  abstract;

virtual void onConnectionRestore(std::string& connectionId)
  abstract;

virtual void onQueuePacketDropped(const ActiveMessage& activeMessage)
  abstract;

virtual void onQueuePacketReady(std::string& connectionId)
  abstract;

virtual void onException(std::string& connectionId)
  abstract;
```

A partir del momento en que estos métodos son declarados en el espacio de usuario, la librería ya podría actuar como receptora de mensajes, volcando los mensajes en el método *onMessage* descrito anteriormente.

Además, en el momento en el que se implementa la herencia, toda una serie de métodos que exporta son accesibles por parte del usuario. Todos estos métodos se usarían desde el entorno de usuario, formando un flujo de acciones y datos como muestra la figura siguiente.

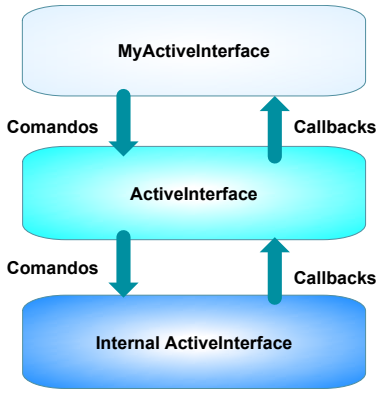

Figura 3.15: Diagrama de interacción entre usuario y ActiveInterface.

Una vez explicados los métodos donde el usuario recibe los datos mediante *callbacks* o eventos, hay que explicar como el usuario interactúa con el núcleo de ActiveInterface. El método principal para enviar datos a la librería es el método *send*.

```
void send (std::string& serviceId, ActiveMessage& activeMessage)
  throw (ActiveException);

void sendResponse ( std::string& connectionId, ActiveMessage& activeMessage)
  throw (ActiveException);
```

Como se puede observar, el método *send* únicamente tiene dos atributos, que serán el servicio al cual se mandará el mensaje, y el propio mensaje. El método *sendResponse* tiene los mismos atributos, pero se utilizará en caso de que se haya recibido un mensaje que requiera una respuesta.

Además de los métodos donde se recibe y se envía información, la API se completa con otros métodos enfocados a operaciones de modificación, creación y borrado de las entidades que forman la topología de red de ActiveInterface.

Los métodos de creación son métodos que requieren de un previo conocimiento de como funciona la API de JMS pues es necesario argumentar estas creaciones con una serie de argumentos que modifican la forma de actuación de esta entidad.

```
1
2 ActiveConnection* newProducer ( std::string& id,
3                                 std::string& ipBroker,
4                                 std::string& destination,
5                                 bool requestReply=false,
6                                 bool topic=false,
```

```
7                                   bool persistent=false,
8                                   bool clientAck=false,
9                                   int maxSizeQueue=0,
10                                  const std::string& username="",
11                                  const std::string& password="",
12                                  const std::string& clientId="",
13                                  long persistence=0,
14                                  const std::string& certificate="")
15    throw (ActiveException);
16
17  ActiveConnection* newConsumer ( std::string& id,
18                                  std::string& ipBroker,
19                                  std::string& destination,
20                                  bool requestReply=false,
21                                  bool topic=false,
22                                  bool durable=false,
23                                  bool clientAck=false,
24                                  const std::string& selector="",
25                                  const std::string& username="",
26                                  const std::string& password="",
27                                  const std::string& clientId="",
28                                  const std::string& certificate="")
29    throw (ActiveException);
30
31
32  void newLink(    std::string& serviceId,
33                   std::string& linkId,
34                   std::string& name,
35                   std::string& connectionId,
36                   ai::utils::ParameterList& parameterList)
37    throw (ActiveException);
38
39
40  void addLink (std::string& serviceId, std::string& linkId)
41    throw (ActiveException);
42
43  bool setLinkConnection( std::string& linkId,
44                          std::string& connectionId)
45    throw (ActiveException);
```

Con estos cuatro simples métodos, se pueden crear todas las entidades que usa ActiveInterface para crear su topología de red. Como se puede observar, en los métodos, se van uniendo las distintas entidades.

Primero se crearán las conexiones, ya sean productores de mensajes o consumidores con una serie de parámetros como la dirección de destino, si el mensaje requiere un acuse de recibo, si es un tópico o una cola y otra serie de parámetros que son descritos en el anexo de inicialización. Una vez están creados los *endpoints* es necesario crear un *link* y asociárselo; como se puede apreciar en el método *newLink*. En el método de creación de *link* se le argumenta una lista de parámetros que serán incorporados a cada mensaje que use este *link*.

Una vez definidas las conexiones y enlaces, es preciso crear el último eslabón que es el servicio. Este identificador de servicio (*serviceId*) será por el que el usuario acceda a sus elementos asociados.

Además de tareas relacionadas con la creación de la topología de ActiveInterface, también se permite consultar el estado de la topología actual. Para esto, se dan al usuario de una serie de métodos a partir de los cuales se pueden extraer las asociaciones entre entidades, partiendo de una u otra entidad.

```
void getLinksByConn( std::string& connectionId,
                     std::list<ActiveLink*>& linkList)
  throw (ActiveException);

void getLinksByService( std::string& serviceId,
                        std::list<ActiveLink*>& linkList)
  throw (ActiveException);

void getServicesByLink( std::string& linkId,
                        std::list<std::string>& linkList)
  throw (ActiveException);

const ActiveConnection* getConnByLink(std::string& linkId)
  throw (ActiveException);

void getConnsByService (std::string& serviceId,
                        std::list<ActiveConnection*>& connectionListR)
  throw (ActiveException);

void getConnsByDestination(std::string& destination,
                           std::list<ActiveConnection*>& connectionListR)
  throw (ActiveException);

void getServices(std::list<std::string>& servicesList)
  throw (ActiveException);
```

Mediante estos métodos de consulta, es posible extraer todas las asociaciones de la topología actual. ActiveInterface permite extraer cualquier entidad asociada a otra a partir de una ellas. Por ejemplo, es posible obtener todas las conexiones a partir de un enlace, o obtener todos los servicios o conexiones a partir del mismo.
Estos métodos de consulta son muy útiles para el desarrollador pues permiten desarrollar código flexible, basándose sólo en nombres de servicios. En general, estos métodos serán utilizados para, a partir de ellos, realizar cambios en la topología ya sean modificaciones o borrados.

Para hacer borrados o crear nuevas asociaciones entre entidades, se pueden usar los siguientes métodos.

```
bool destroyConnection(std::string& connectionId)
  throw (ActiveException);

bool destroyLink(std::string& linkId)
  throw (ActiveException);

bool destroyService(std::string& serviceId)
  throw (ActiveException);

bool destroyServiceLink ( std::string& linkId,
                          std::string& serviceId)
  throw (ActiveException);

bool destroyLinkConnection (std::string& linkId)
  throw (ActiveException);
```

Además de los métodos relacionados con la topología, es necesario detallar la manera en la que ActiveInterface permite al usuario agregar, modificar o eliminar información de los mensajes que crea o recibe. ActiveMessage no es más que un contenedor de parámetros y propiedades, que pueden ser de tipo entero, real, bytes o cadenas de caracteres.

Muchos de los métodos que se muestran a continuación describen las formas de interactuar con estos atributos.

```
1
2  const ParameterList& getParameterList() const;
3
4  int getParametersSize() const;
5
6  Parameter* getParameter (std::string& key) const;
7
8  IntParameter* getIntParameter(std::string& key) const
9    throw (ActiveException);
10
11 RealParameter* getRealParameter(std::string& key) const
12   throw (ActiveException);
13
14 StringParameter* getStringParameter(std::string& key) const
15   throw (ActiveException);
16
17 BytesParameter* getBytesParameter(std::string& key) const
18   throw (ActiveException);
19
20 Parameter* getParameter (int index, std::string& key) const;
21
22 void insertIntParameter(std::string& key, int value);
23
24 void insertRealParameter(std::string& key, float value);
25
```

```
26 void insertStringParameter(std::string& key, std::string& value);
27
28 void insertBytesParameter(std::string& key,
29                           std::vector<unsigned char>& value);
30
31 void deleteParameter(std::string& key);
32
33 void clearParameters ();
```

Como se puede observar, la inserción de distintos tipos de datos es muy simple, asociando al tipo de dato una clave que permitirá el acceso directo a dicha información. Además, se puede acceder de manera iterativa a los parámetros existentes en un ActiveMessage. Esto ayudaría a no tener que discernir entre tipos de mensajes y actuar en base a las propiedades que el emisor haya mandado.

Cola de acceso concurrente

Una cola de acceso concurrente es una cola de la que se puede extraer y insertar elementos desde distintos hilos de una manera segura.

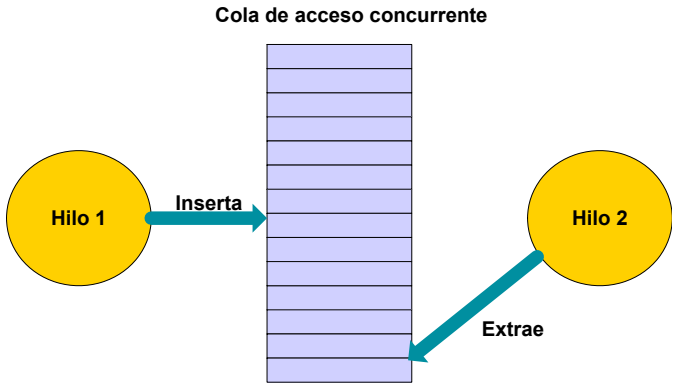

Figura 3.16: Imagen que ilustra una cola de acceso concurrente.

Aunque la cola de acceso concurrente no es una tarea compleja, se ha introducido este concepto ya que es un concepto íntimamente relacionado con la implementación de los hilos productores y consumidores sobre y hacia esta cola.

Implementación de threads e IPC

La implementación de threads que se ha utilizado tenía el requisito de ser multiplataforma. Para esto había que abstraerse de las API's nativas de los sistemas

operativos en los que se desarrollaba y compilaba.

Esta abstracción es la que se ha conseguido gracias a la librería *Apache Portable Runtime*. Esta librería ofrece al programador toda una serie de llamadas que abstraen de tareas como lectura de ficheros, semáforos, sockets, mapeos de memoria y muchas más tareas.

En ActiveInterface se han utilizado las primitivas de hilos y de sincronización de hilos, también llamadas IPC ó Interprocess communications.

Uno de las funcionalidades que se especificaron en los requisitos era que un envío no podía ser bloqueante, respetando el modelo de *Send and Forget*. Para solucionar este problema fue necesario desarrollar la aplicación orientándola a multihilo.

El problema de los bloqueos principalmente ocurre cuando el *broker* no se encuentra operativo y los mensajes no se pueden enviar. En este caso, hay varias opciones de implementación. Una de ellas podría ser lanzar cada mensaje en un hilo, lo que podría provocar un desbordamiento del límite de hilos del sistema operativo.

En este punto se decidió optar por una cola concurrente señalizando los mensajes enviados mediante eventos. La primera parte de la arquitectura de envío de mensajes se explicó en el capítulo anterior, y los eventos se han implementado mediante un mutex condicional.

La implementación de un mutex condicional podría ser la siguiente:

```
1
2  while(true){
3
4      apr_thread_mutex_lock(mySharedObject->getMutex());
5
6      while (mySharedObject->getMessagesReady() == 0 &&
7             !mySharedObject->getEndThread()) {
8
9        apr_thread_cond_wait( mySharedObject->getCond(),
10                             mySharedObject->getMutex());
11     }
12
13     apr_thread_mutex_unlock(mySharedObject->getMutex());
14 }
```

Como se puede apreciar en el código, un mutex condicional utiliza dos mutex internamente. Uno actúa de semáforo para entrada en la sección crítica (línea 4). El

siguiente mutex es el condicional por sí sólo, implementado en la línea 9. Este mutex condicional sólo se desbloqueará cuando el número de mensajes sea mayor que 0. Para aumentar el valor de la variable que controla el desbloqueo del mutex condicional se hace a través de un *objeto compartido*.

```
1
2 apr_thread_mutex_lock(activeSharedObject.getMutex());
3 if (received){
4         activeSharedObject.newMessage();
5 }else{
6         activeSharedObject.messageSent();
7 }
8 apr_thread_cond_signal(activeSharedObject.getCond());
9 apr_thread_mutex_unlock(activeSharedObject.getMutex());
```

El acceso al objeto compartido se realiza de la manera que se puede observar en el código anterior. Se protege el acceso concurrente al mismo, se aumenta la variable que desbloquea el hilo, y se lanza un evento hacia el mutex del hilo servidor (línea 8).

Una vez explicado como se ha implementado el mutex condicional, el siguiente diagrama explica gráficamente la secuencia de eventos desde que un usuario envía un mensaje a ActiveInterface.

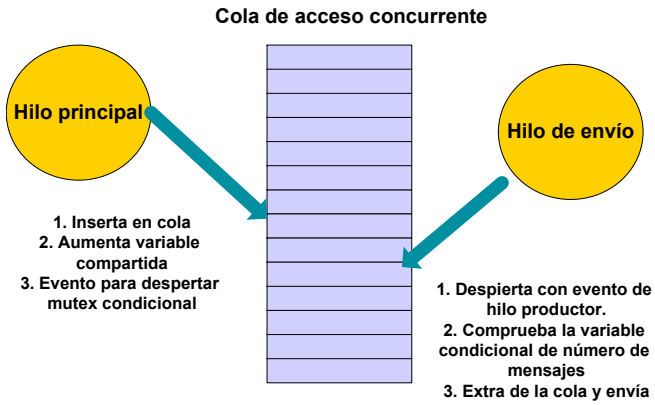

Figura 3.17: Imagen de mecanismos de sincronización de procesos.

Serialización de datos

Como se ha introducido en capítulos anteriores, la serialización de datos se ha utilizado para la escritura de datos en fichero en relación a la funcionalidad de persistencia.

Uno de los requisitos principales es que la persistencia era una funcionalidad que podía ser activada o desactivada según los requisitos del usuario. Esta funcionalidad puede tener

sentido en ámbitos en los que los mensajes antiguos no son importantes, pues por ejemplo, lo importante es reflejar el estado en tiempo real. Sin embargo, en ciertas situaciones como sistemas de registros, o cualquier otro tipo de mensaje del que se requiera seguridad de envío, la persistencia de ActiveInterface puede ser una gran aliada.

Mediante la siguiente figura se va a intentar describir gráficamente los distintos estados por los que pasa ActiveInterface tanto para el caso de que la persistencia no haya entrado en funcionamiento (no se ha desborado el buffer intermedio) como en el caso en que se desborde.

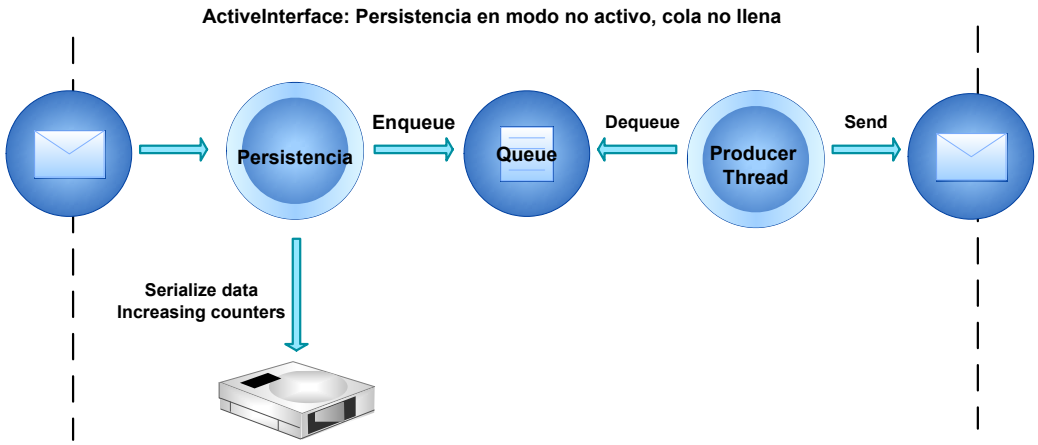

Figura 3.18: Persistencia activada pero no activa.

Cuando la persistencia se encuentra activada, pero no activa, su funcionamiento es simplemente ir incrementando los contadores de mensajes enviados, encolados y serializados, además de serializar el fichero a disco.

En este caso la persistencia es un módulo pasivo preparado para actuar en caso necesario. Si la conexión con el broker se ralentiza, o se pierde, la cola de mensajes empezaría a crecer hasta el límite establecido por el usuario. Si antes de que se restablezca la conexión, el buffer llega a su máximo tamaño, la persistencia se activaría, funcionando de la manera que representa la siguiente figura.

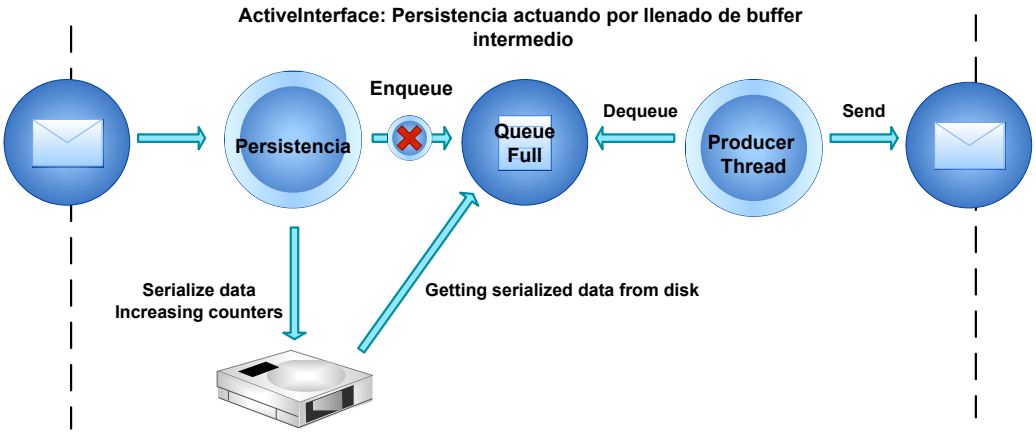

Figura 3.19: Persistencia activada pero no activa.

Como se puede ver en la figura, en cuanto un mensaje es incapaz de encolarse, el módulo de persistencia entra en funcionamiento evitando la entrada de ningún mensaje más a la cola por el camino normal. Cuando la persistencia está activa, los mensajes son serializados a disco siempre en la última posición, utilizando el fichero como si fuera una cola de mensajes. Cuando la conexión se restablece y los mensajes que están en el buffer se pueden volver a enviar, es el módulo de persistencia el que se encarga de deserializar los mensajes de disco, y encolarlos en la posición libre que se ha quedado en la cola.
Cuando todos los mensajes que estaban serializados en disco se hayan enviado, el modo de persistencia se desactivará y volverá a funcionar el cauce normal de envío de mensajes.

Es destacable como esta persistencia ayuda a controlar grandes picos de envíos de información. Si en ciertas situaciones se producen envíos de grandes cantidades de información, el módulo de persistencia ayuda a serializar esos datos e ir mandándolos sin saturarse y sin llegar a perder ningún mensaje.

El módulo de persistencia ha sido desarrollado para intentar ser lo más óptimo posible y no entrar en un posible caso de persistencia continua, que añadiría un retardo a cualquier envío de mensaje. Por parte de los usuarios es muy importante definir un tamaño de cola basado en su entorno.

Para hacer una breve introducción a la implementación de la serialización usando *boost* es de destacar la gran abstracción que provee al usuario. Por ejemplo, en el cuadro siguiente se muestra el código completo para la serialización de la clase ActiveMessage.

```
//making activemessage as a serializable class
friend class boost::serialization::access;
template<class Archive>

/**
 * Method that describes atts that are going to be serialized
 */
void serialize(Archive & ar, const unsigned int version){
  ar & serviceId;
  ar & linkId;
  ar & connectionId;
  ar & timeToLive;
  ar & priority;
  ar & requestReply;
  ar & correlationId;
  ar & text;
  ar & textMessage;
  ar & packetDesc;
  ar & parameterList;
  ar & propertiesList;
}
```

Con estas simples líneas, la clase ActiveMessage es serializable a disco mediante el operador: <<.

Esta implementación de serialización funcionaría siempre que sólo se tratase de tipos de datos simples. En ActiveMessage hay distintos tipos de datos, herencia y tablas *hash* que forman parte del mensaje, lo que conlleva aplicar este método a todas las clases con las que se relaciona.

Programación orientada a eventos

La programación orientada a eventos es una programación basada en acciones que son generadas por o para el usuario. ActiveInterface es una aplicación dirigida por eventos.

Como se muestra en la figura siguiente, todas las posibles interacciones desde el núcleo de ActiveInterface hacia la parte de usuario, son generados a través de eventos:

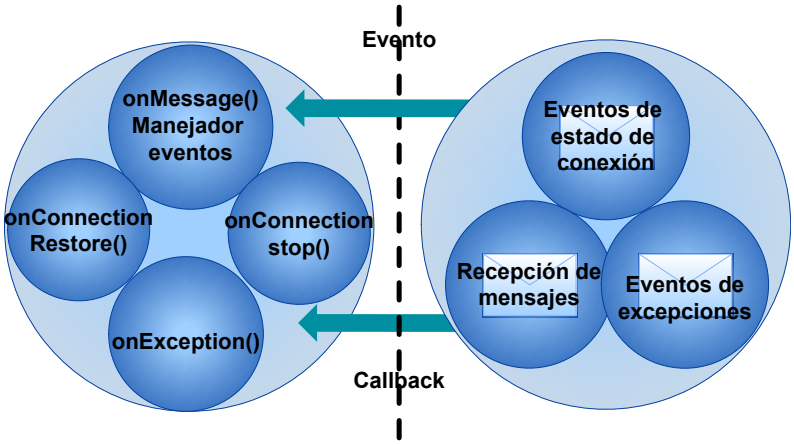

Figura 3.20: Gráfico que representa los eventos que genera ActiveInterface.

Uno de los principales problemas de la programación orientada a eventos, es que posibles errores en el desarrollo del usuario, puede influir en el rendimiento de la librería. Si por ejemplo, en el método manejador de eventos de recepción de mensajes, el usuario por un determinado motivo bloquea la llamada o la ralentiza, esto podría provocar la no recepción de más mensajes o la ralentización de todo el proceso de recepción de mensajes.

3.4. Pruebas

3.4.1. Batería de pruebas

Esta sección es una de las secciones más importantes pues asegura una determinada calidad en cada iteración y en el resultado final del proyecto.
Para probar ActiveInterface se tiene una batería de pruebas que cubre la mayoría de funcionalidades. En las siguientes imágenes se puede ver como se han organizado las pruebas y que funcionalidad prueba cada una.

Cada una de las pruebas está organizada en niveles:

- **Failover:** Las pruebas que tienen failover son pruebas orientadas a probar clústeres de brokers para obtener alta disponibilidad y rendimiento. La diferencia principal frente a No-failover es que en las últimas, la cadena de conexión se especifica únicamente un *broker*.

  - **No persistence:** Las pruebas sin persistencia, son pruebas en las que el módulo de persistencia no está activado.

- **SSL:** Pruebas de conexión con brokers a través de *Secure Sockets Layer*.
- **NO-SSL:** Pruebas de conexión bajo conexión no segura.

- **Persistence:** Pruebas con failover en las que la persistencia se encuentra activada para verificar el reenvío de mensajes con broker parado.
  - **SSL:** Misma descripción que en caso anterior.
  - **NO-SSL:** Misma descripción que caso anterior.

- **No-failover:** Son pruebas orientadas a un único broker o múltiples brokers sin sistemas de clustering.
  - **No persistence:**Misma descripción que en caso anterior.
    - **SSL:**Misma descripción que en caso anterior.
    - **NO-SSL:**Misma descripción que en caso anterior.
  - **Persistence:**Misma descripción que en caso anterior.
    - **SSL:**Misma descripción que en caso anterior.
    - **NO-SSL:**Misma descripción que en caso anterior.

Para tener una visión más gráfica de como se organizan las pruebas se puede ver las siguientes imágenes:

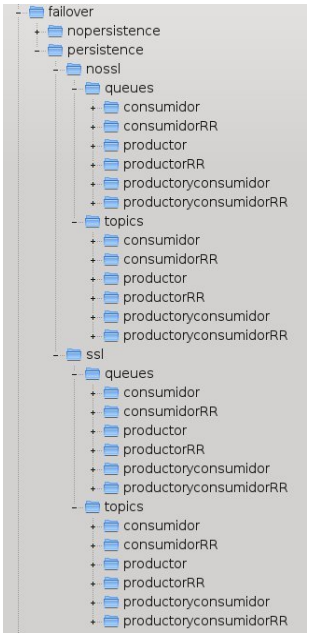

Figura 3.21: Gráfico que representa la estructura de directorios para pruebas con failover y persistencia.

Como se puede ver en la figura anterior, dentro de cada directorio final, se encuentran los distintos comportamientos que podría tener cada *endpoint* que puede establecerse en

la librería. Hay que destacar que las baterías de pruebas se realizan usando el mismo programa ejecutable, cambiando únicamente la configuración de cada uno de los tipos de entidades. Esta configuración se incluye dentro de cada directorio con el nombre *ActiveConfiguration.xml*.

Por ejemplo, en la imagen se puede observar como para una configuración con *failover* y con persistencia, existe la posibilidad de establecer conexión mediante SSL o mediante *sockets* no cifrados. Dentro de cada de los tipos de conexión, se pueden tener los siguientes comportamientos:

- **Productor:** Si se arranca el programa de prueba denominado productor, significa que se va a generar una topología de red formada por sólo un productor de mensajes hacia una cola determinada por el fichero de configuración.
- **Consumidor:** Si se arranca el binario con el fichero de configuración bajo la carpeta consumidor, se está arrancando una topología formada por un sólo consumidor de mensajes de una determinada cola.
- **Productor y Consumidor:** Bajo este directorio se encuentra un fichero de configuración que instancia un productor y un consumidor sobre la misma cola, en el mismo proceso.
- **ProductorRR:** Se instanciaría una topología de red basada en un productor configurado para pedir y recibir respuestas a los mensajes que emite. Se abrevia RR a lo que se denomina *Request Reply*.
- **ConsumidorRR:** Bajo este directorio se encontraría el fichero de configuración que instanciaría un consumidor que genera respuestas a cada mensaje recibido.
- **Productor y Consumidor con RR:** Bajo este directorio se instanciaría un productor y un consumidor que envían y reciben respuestas a los mensajes de cada uno. Ambos bajo el mismo proceso.

Por tanto como se puede ver, cualquier usuario puede probar con cambios simples en el fichero de configuración casi cualquier funcionalidad de ActiveInterface. Además, todos estos ficheros de configuración sirven como ayuda y ejemplo para los usuarios finales.

Además, de estos ficheros de configuración, existen distintos ficheros de ejemplo que ayudan a los usuarios a empezar a utilizar ActiveInterface. Cada uno de ellos ayuda a los usuarios a implementar tareas como la generación de la topología de red usando la API,

envío de mensajes que requieran multithreading, o ejemplos simples que usan los ficheros de configuración para inicializarse.

3.4.2. Pruebas en entornos de producción

Una de las características más importantes de ActiveInterface es que es un software real, que funciona y que se encuentra en producción en muchos ámbitos de uso.

Sistema de venta y control de acceso de Metro de Madrid

El sistema de venta y control de acceso de Metro de Madrid está compuesto por todas las máquinas expendedoras de billetes, tanto automáticas como las gestionadas por los trabajadores de la estación y los tornos de acceso a la misma estación. Con este sistema se monitorizan la venta de billetes, los accesos a las estaciones y el propio hardware que se utiliza para ello.

ActiveInterface se encuentra en todas las máquinas clientes de la red de metro, es decir, en todas las máquinas que participan en el sistema de cobro o de control de acceso. Por tanto, en la red de metro habrá un total de **4000 máquinas** que utilizan ActiveInterface para enviar todos los registros a un servidor central donde se procesa y monitoriza.

Figura 3.22: Logo de metro de madrid.

En la parte servidora ActiveInterface está embebido en la aplicación que procesa y gestiona la gestión de alarmas y notificaciones, recibiendo el tráfico de todas las máquinas clientes, con un número máximo de mensajes estimado de 300 mensajes por segundo.

Aunque la topología de red no es muy compleja, se han utilizado distintos tipos de entidades, agrupación de conexiones, uso del sistema de persistencia y manejo de la API

por parte de integradores.

Además de todos estos datos, la flexibilidad por la necesidad de adaptación a nuevos sistemas, la seguridad y estabilidad son otras de las funcionalidades que se han tenido muy en cuenta dado que forma parte de un sistema crítico.

Sistemas de control de túneles

En el entorno de tráfico terrestre, está a un nivel más alto que en la implementación de Metro de Madrid.

ActiveInterface se ha implantado en los nuevos desarrollos de túneles en Colombia y Marruecos, actuando de interfaz entre el núcleo de la aplicación, y un sistema de monitorización del estado general del túnel.

En este caso, la interfaz gráfica que usan los operadores está desarrollada en Java, y ActiveInterface ha permitido conectar el núcleo de la aplicación de control, desarrollada en C++ con una nueva interfaz más flexible y amigable para el usuario.

Sistema de control de paso de vehículos

ActiveInterface también se está usando en aplicaciones con objetivo de conteo de vehículos en los denominados *peajes en sombra*.

En este ámbito, existen una serie de procesos en la propia vía que se encargan de controlar los dispositivos de detección y clasificación de vehículos; además existen una serie de niveles superiores que se encargan de controlar los accesos, generación de informes y todo tipo de tareas administrativas.

ActiveInterface ha sido el conector entre las vías y los niveles superiores, tanto de sistemas de monitorización de hardware como conteos o sistemas de respaldo.

Peaje de vehículos.

ActiveInterface también se está usando en implementaciones de peajes tradicionales de carreteras. Actualmente se ha incluido en proyectos en México y Colombia.

El uso de ActiveInterface en este ámbito es muy similar al caso anterior, proveedor de comunicaciones entre los distintos niveles que forman la arquitectura de un peaje convencional.

Por tanto, y para terminar este capítulo, es importante destacar que ActiveInterface es un software estable, confiable y probado en muchos y distintos entornos.

Este hecho no hace más que destacar su estabilidad y su flexibilidad para adaptarse a multitud de entornos, ámbitos y aplicaciones con objetivos distintos.

3.5. Validación

Cuando se evalúa una tecnología, es muy importante conocer sus límites. Estos límites marcan los contextos en los que puede ser útil o no. Por ejemplo, si en una aplicación es necesario una librería de comunicaciones en tiempo real, es decir, que ofrezca una latencia menor de 30 milisegundos, gracias a este apartado el evaluador podría tener datos suficientes para evaluar si ActiveInterface cumple esos requisitos.

En este apartado se van a aportar una serie de pruebas de validación, enfocadas a conocer el rendimiento y el comportamiento de esta librería. Esto ayudará a determinar si es válida para aplicarse a determinados problemas y entornos.

Se van a generar casos de prueba consistentes en el envío y recepción de 1000 mensajes, con distintos tamaños de paquete y distintos tamaños de cola. A partir de las gráficas generadas, se estudiará la latencia de cada mensaje, la latencia media, el tiempo de envío total y como se comporta la cola de ActiveInterface.

Gracias a estos datos se habrá realizado un pequeño *benchmark* que debería ser contrastado en trabajos futuros en distintas máquinas, y en diferentes entornos. Es importante destacar que estas pruebas se han realizado sobre una máquina personal,

considerando la red como ideal.

Es importante destacar que los tiempos sobre los que se toman estas medidas son tiempos muy pequeños, por lo que las políticas de *performance* del planificador del procesador, junto con los requisitos de otras aplicaciones que se están ejecutando, pueden influir en determinadas pruebas afectando a los resultados finales.

3.5.1. Envío y recepción de 1000 paquetes, con tamaño de 2KB

En el primer contexto se han realizado envíos masivos de mensajes en entorno monohilo, desplegando sobre la misma máquina un proceso configurado como consumidor y otro como productor.

Al ser un tamaño de paquete pequeño, y al ser la red ideal, la latencia envío/recepción de mensajes es muy pequeña, con un *throughput* muy alto.

Se van a realizar distintas pruebas con diferentes tamaños de la cola gestionada por ActiveInterface:

Tamaño de cola de 5 mensajes.

Un tamaño de cola de 5 mensajes significa que ActiveInterface va a disponer de un buffer intermedio en memoria volátil de 5 mensajes. Esto quiere decir que si la aplicación que genera mensajes es mucho más rápida produciendo, que la librería en enviarlos hacia el broker, podría suponer un llenado del buffer con la consiguiente pérdida de mensajes.

Los datos obtenidos son:

- **Latencia media de envío/recepción:** 0,013005 segundos, en torno a 10 milisegundos.
- **Tiempo total de envío/recepción:** 0,582000017 segundos.
- **Número de mensajes enviados por segundo:** 1718,213008 mensajes por segundo.

Como se puede observar por los datos, la latencia está en torno a 10 milisegundos, generando una gráfica como la siguiente:

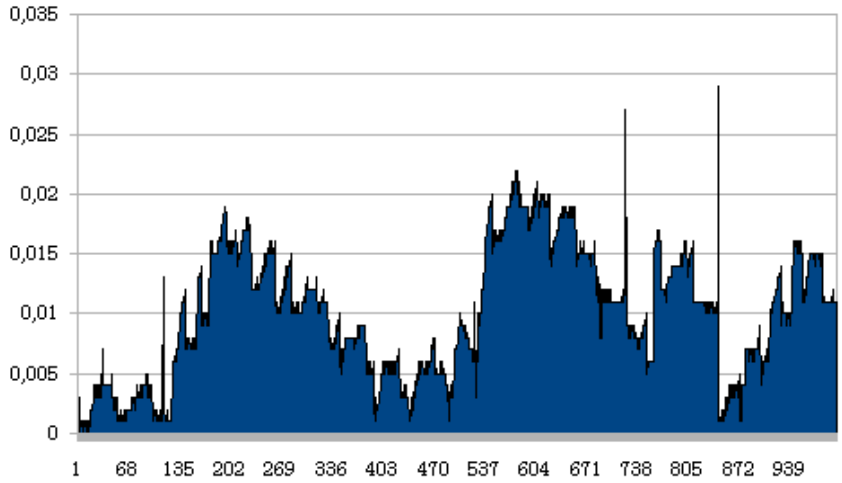

Figura 3.23: Latencia para 1000 mensajes de 2KB con tamaño de cola 5.

Además de esta gráfica, es interesante comparar la latencia de los mensajes con el llenado de la cola que maneja ActiveInterface.

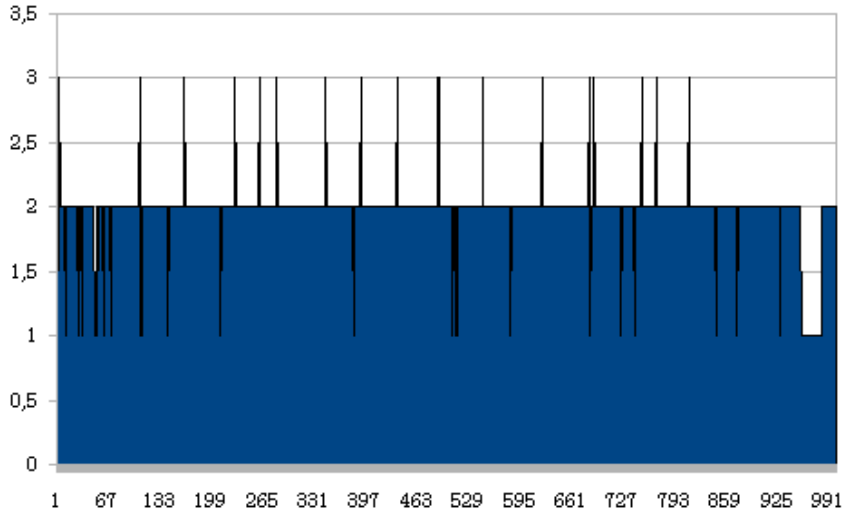

Figura 3.24: Llenado de cola de mensajes de ActiveInterface con tamaño máximo de 5.

Estudiando la gráfica de latencia se puede observar como ningún mensaje supera los 30 milisegundos de latencia, por lo que se puede considerar un rendimiento que encajaría en sistemas de comunicaciones en tiempo real.

En cuanto a la gráfica de llenado de la cola, es importante destacar como los algoritmos de congestión que se han introducido evitan el llenado de la cola, manteniendo por lo

general un buffer estable en el 50 por ciento de su capacidad.

Por tanto en este contexto, cualquier aplicación que requiera un tamaño de mensajes de 2KB y tiempos inferiores a 30 milisegundos, podría utilizar ActiveInterface como su librería de comunicaciones.

Tamaño de cola de 10 mensajes.

Respecto a la prueba anterior se ha aumentado el tamaño de la cola de mensajes. La latencia debe de generar un resultado similar, pues sólo mide el tiempo desde que se envía finalmente hasta que se recibe en el consumidor.
Aumentar el tamaño de la cola, debe producir un aumento en el número de mensajes enviados por segundo, lo que produce un decremento en el tiempo total que se necesita para envíar los 1000 mensajes generados en la prueba.

Los datos obtenidos son:

- **Latencia media de envío/recepción:** 0,019005 segundos, en torno a 20 milisegundos.

- **Tiempo total de envío/recepción:** 0,552999964 segundos.

- **Número de mensajes enviados por segundo:** 1776,199049 mensajes por segundo.

Como se puede observar por los datos, la latencia está en torno a 20 milisegundos, generando una gráfica como la siguiente:

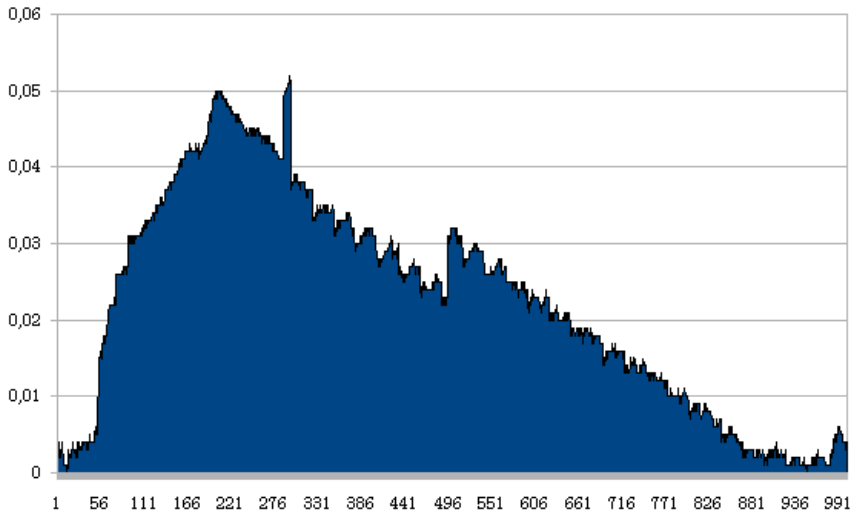

Figura 3.25: Latencia para 1000 mensajes de 2KB con tamaño de cola 10.

Además de esta gráfica, es interesante comparar la latencia de los mensajes con el llenado de la cola que maneja ActiveInterface.

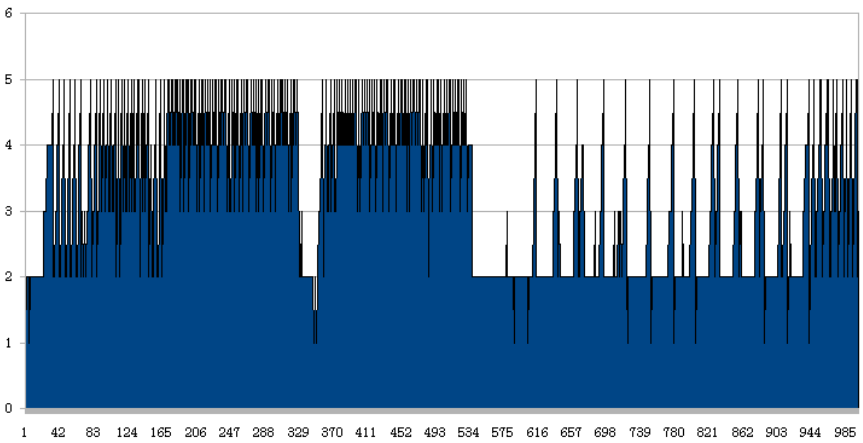

Figura 3.26: Llenado de cola de mensajes de ActiveInterface con tamaño máximo de 10.

Estudiando todos los detalles se puede llegar a la conclusión de que aumentando el tamaño de la cola se ha conseguido una mejora de performance ya que el buffer intermedio dispone en todo momento de más mensajes a enviar.

Tamaño de cola de 50 mensajes

El tamaño de la cola se ha aumentado a 50 posiciones. Es importante destacar que no es directamente proporcional el aumento del tamaño de la cola con el aumento del *throughput* de ActiveInterface. Esto se debe a que el mecanismo de control de congestión de ActiveInterface siempre intenta que la cola se encuentre en un punto medio de su tamaño máximo, aplicando determinadas ralentizaciones cuando se superan esos límites.

Por tanto, en este contexto se han obtenido los siguientes resultados:

- **Latencia media de envío/recepción:** 0,021899997 segundos, en torno a 20 milisegundos.
- **Tiempo total de envío/recepción:** 0,559000015 segundos.
- **Número de mensajes enviados por segundo:** 1788,908717 mensajes por segundo.

Se puede observar una latencia de entorno a 20 milisegundos, representada en la siguiente gráfica:

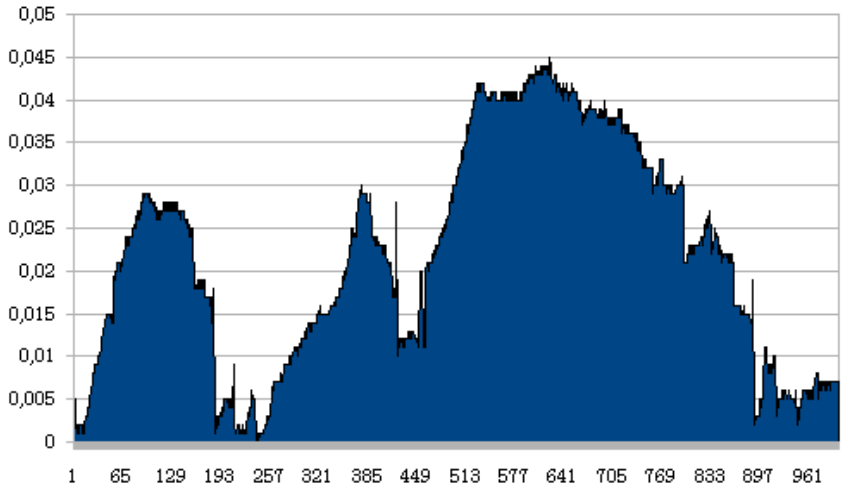

Figura 3.27: Latencia para 1000 mensajes de 2KB con tamaño de cola 50.

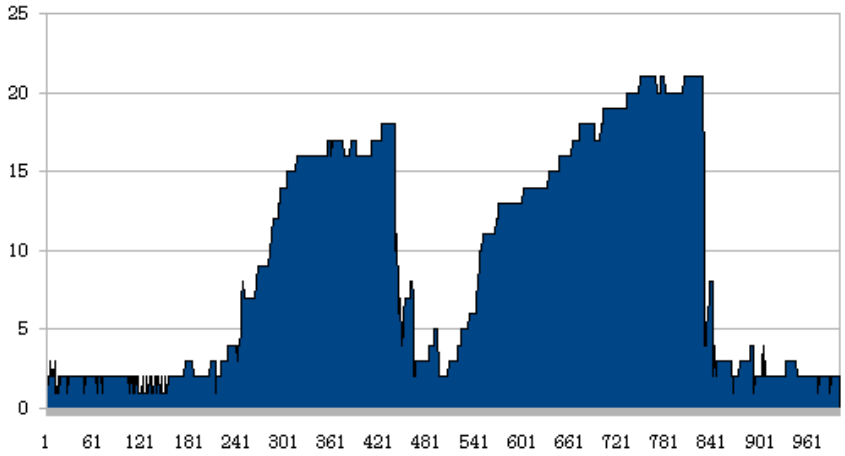

Figura 3.28: Llenado de cola de mensajes de ActiveInterface con tamaño máximo de 50.

Se puede ver que el número de mensajes por segundo ha crecido respecto a tamaños de cola menores, disminuyendo el tiempo en el que la aplicación es capaz de enviar y recibir los mensajes totales.

Además, gracias a las gráficas se puede intuir que cuando el planificador del procesador da más prioridad al hilo que desencola mensajes respecto al que encola, el tamaño de la cola baja, y la latencia de envío baja. En tiempos tan pequeños como los que se están midiendo, es imposible determinar como el planificador va asignando tiempos de proceso a unos hilos respecto a otros.

Tamaño de cola de 500 mensajes

Con un tamaño de cola de 500 mensajes, ActiveInterface sería capaz de almacenar en memoria volátil la mitad de los mensajes a enviar, por lo tanto, el hilo de encolado puede insertar más mensajes en la cola antes de que el hilo recolector empiece a aplicar un mecanismo de congestión y adquiera más prioridad que el hilo encolador.

Es muy importante dimensionar el tamaño de la cola con respecto el máximo número de mensajes en pico por segundo, ya que desbordar la cola activaría el mecanismo de persistencia, que induciría mucha más latencia por el acceso a disco. También es importante definir el tamaño respecto a la media general de mensajes, esto hace ganar rendimiento, ofreciendo un buffer con un llenado medio.

Los datos obtenidos en este contexto son:

- **Latencia media de envío/recepción:** 0,015203001 segundos, en torno a 15 milisegundos.

- **Tiempo total de envío/recepción:** 0,578999996 segundos.

- **Número de mensajes enviados por segundo:** 1727,115728 mensajes por segundo.

En este ejemplo, la cola está sobredimensionada respecto al número de mensajes que se mandan, por lo que se observa una pequeña perdida de rendimiento en número de mensajes por segundo y por tanto el tiempo en enviar y recibir dichos mensajes.

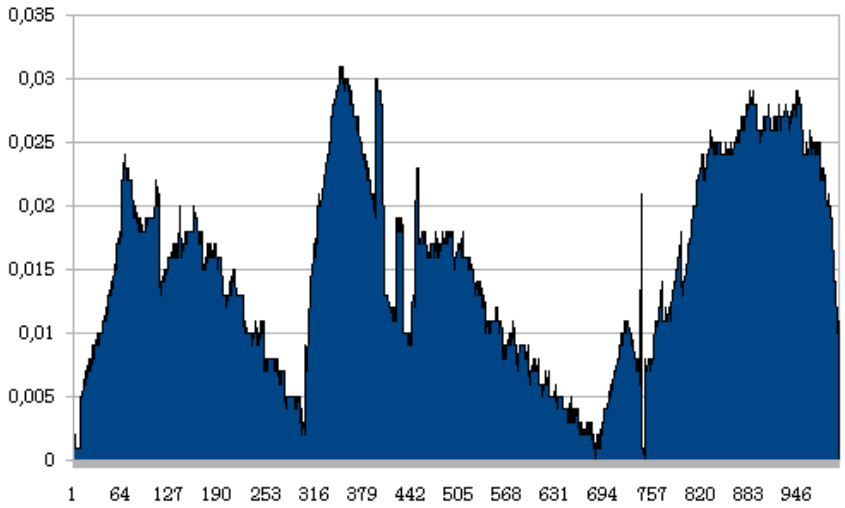

Figura 3.29: Latencia para 1000 mensajes de 2KB con tamaño de cola 500.

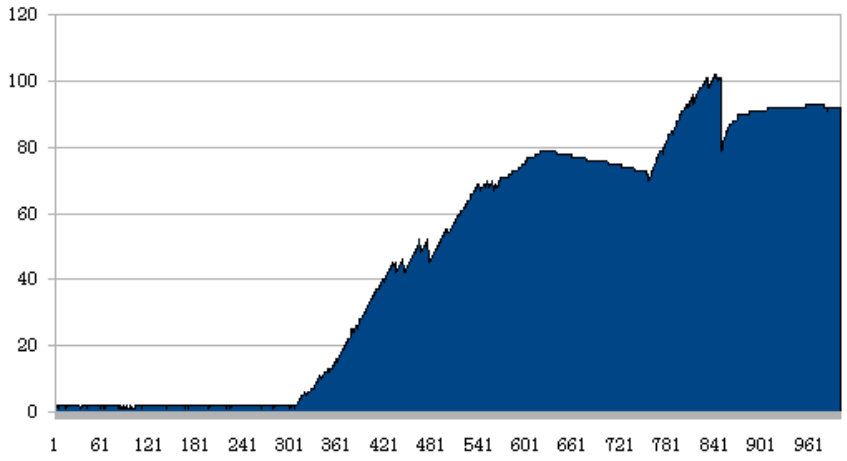

Figura 3.30: Llenado de cola de mensajes de ActiveInterface con tamaño máximo de 500.

En las gráficas se puede ver un comportamiento irregular. El envío de mensajes en estas pruebas se hace en un único hilo, que realiza los 1000 envíos iterativamente, por lo que es el planificador del procesador el encargado de dar más tiempo de proceso a unas u otras tareas.

3.5.2. Envío y recepción de 1000 paquetes con tamaño de 8KB

En esta sección se ha aumentado el tamaño de mensaje. Este cambio va a provocar que la latencia se vea afectada ya que aunque la red sea ideal, el tiempo de proceso de cada una de las partes va a ser mayor al tener que procesar más información.

Tamaño de cola de 10 mensajes.

- **Latencia media de envío/recepción:** 0,023475996 segundos, en torno a 20 milisegundos.
- **Tiempo total de envío/recepción:** 0,641000032 segundos.
- **Número de mensajes enviados por segundo:** 1560,062324 mensajes por segundo.

Como se puede observar por los datos, la latencia está superando los 20 milisegundos, generando una gráfica como la siguiente:

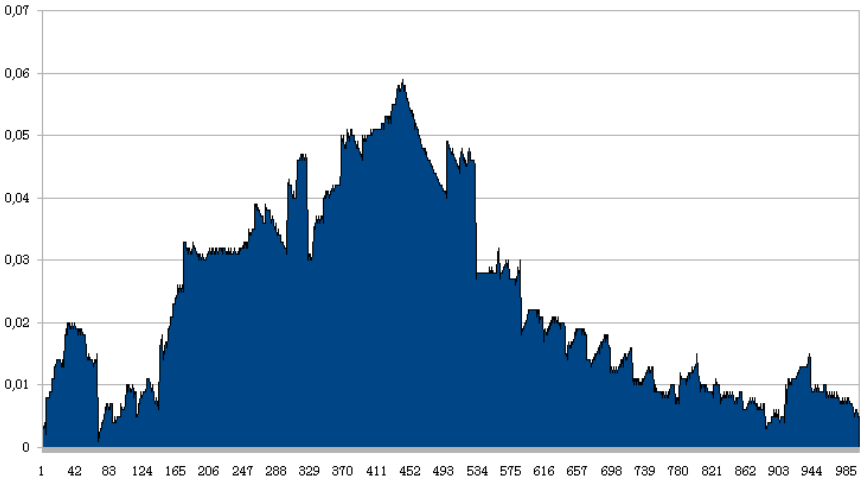

Figura 3.31: Latencia para 1000 mensajes de 8KB con tamaño de cola 10.

Además de esta gráfica, es interesante comparar la latencia de los mensajes con el llenado de la cola que maneja ActiveInterface.

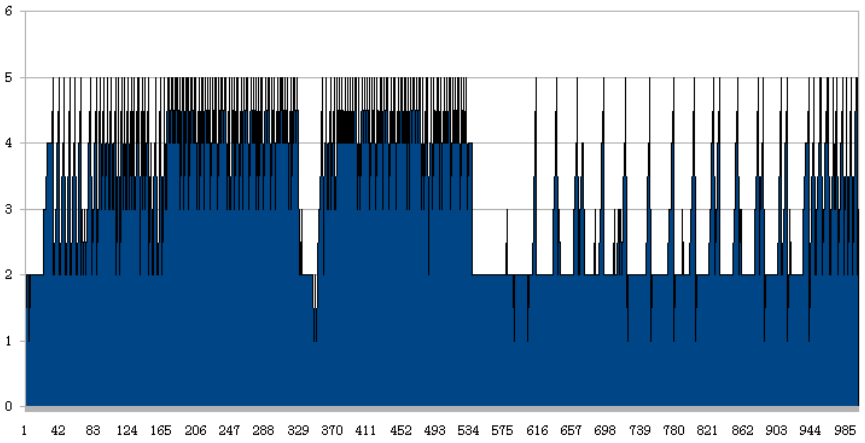

Figura 3.32: Llenado de cola de mensajes de ActiveInterface con tamaño máximo de 10.

Se puede observar por los datos anteriores, que el aumento del tamaño de paquete ha hecho disminuir el *throughput* y ha provocado un aumento de la latencia media de envío/recepción de los mensajes haciéndola superar los 20 milisegundos.

Tamaño de cola de 100 mensajes.

- **Latencia media de envío/recepción:** 0,056931 segundos, en torno a 55 milisegundos.

- **Tiempo total de envío/recepción:** 0,652999878 segundos.
- **Número de mensajes enviados por segundo:** 1531,393854 mensajes por segundo.

Como se puede observar por los datos, la latencia está superando los 55 milisegundos, generando una gráfica como la siguiente:

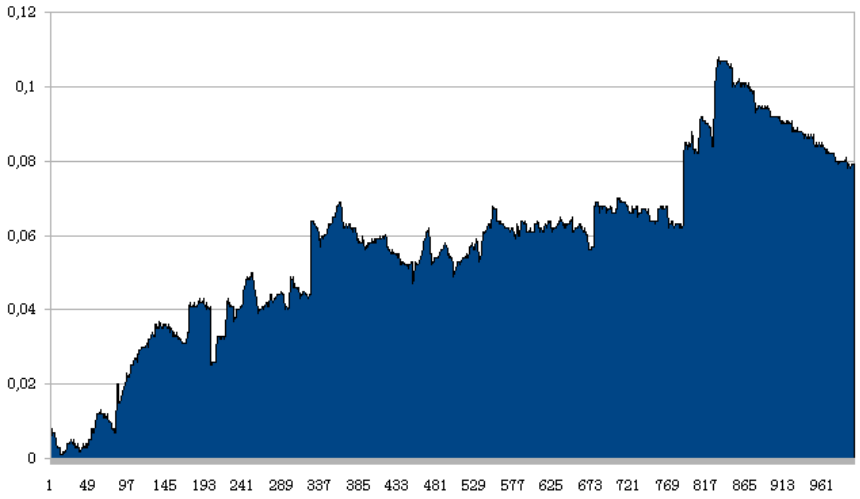

Figura 3.33: Latencia para 1000 mensajes de 8KB con tamaño de cola 100.

Además de esta gráfica, es interesante comparar la latencia de los mensajes con el llenado de la cola que maneja ActiveInterface.

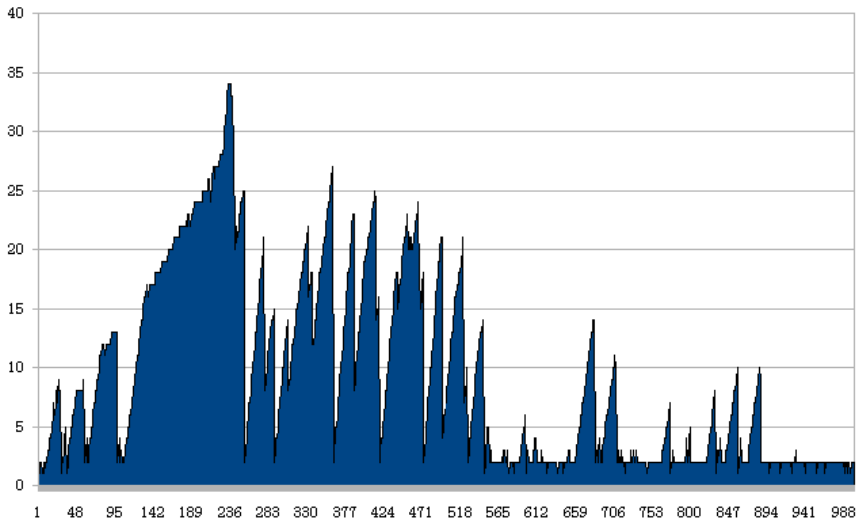

Figura 3.34: Llenado de cola de mensajes de ActiveInterface con tamaño máximo de 100.

Se puede observar por los datos anteriores, que el aumento del tamaño de paquete ha hecho disminuir el *throughput* y ha provocado un aumento de la latencia media de envío/recepción de los mensajes haciéndola superar los 50 milisegundos.

3.5.3. Envío y recepción de 1000 paquetes con tamaño de 1MB

Aumentando el tamaño de mensaje a 1MB, supone aumentar considerablemente el tamaño de mensaje respecto a las pruebas anteriores. Al aumentar el tamaño de mensaje, el tiempo desde que un mensaje es enviado hasta que se recibe es mucho mayor por tener mucha más información que enviar. Por esto, al caer el *throughput*, el proceso de encolado es capaz de encolar más mensajes en el mismo tiempo, lo que provoca una saturación de la cola, y por consiguiente una perdida de paquetes.

Por tanto para este tamaño de paquetes, lo óptimo sería declarar una cola que permitiese almacenar una cantidad de mensajes que coincida con el máximo número de mensajes que la aplicación es capaz de enviar en pico de carga.

Tamaño de cola de 500 mensajes

Los datos obtenidos en este contexto son:

- **Latencia media de envío/recepción:** 0,057835102 segundos, en torno a 60 milisegundos.
- **Tiempo total de envío/recepción:** 3,02432769 segundos.
- **Número de mensajes enviados por segundo:** 330,6520002 mensajes por segundo.

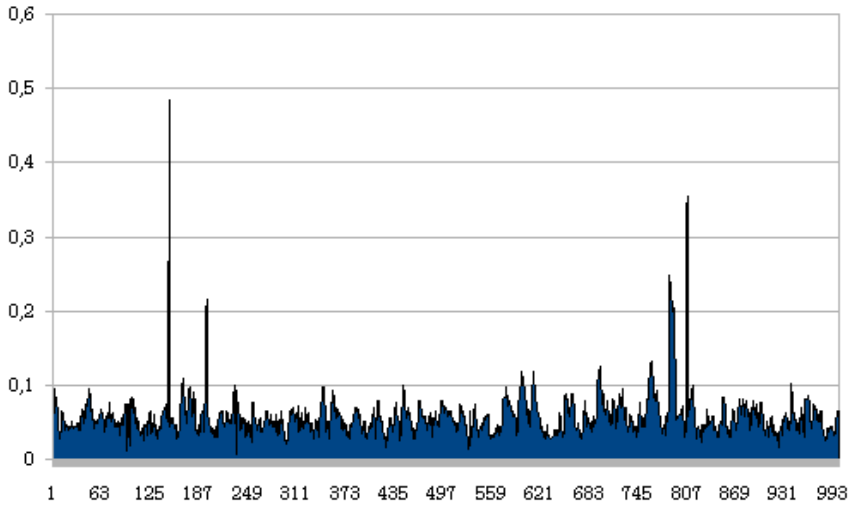

Figura 3.35: Latencia para 1000 mensajes de 1MB con tamaño de cola 500.

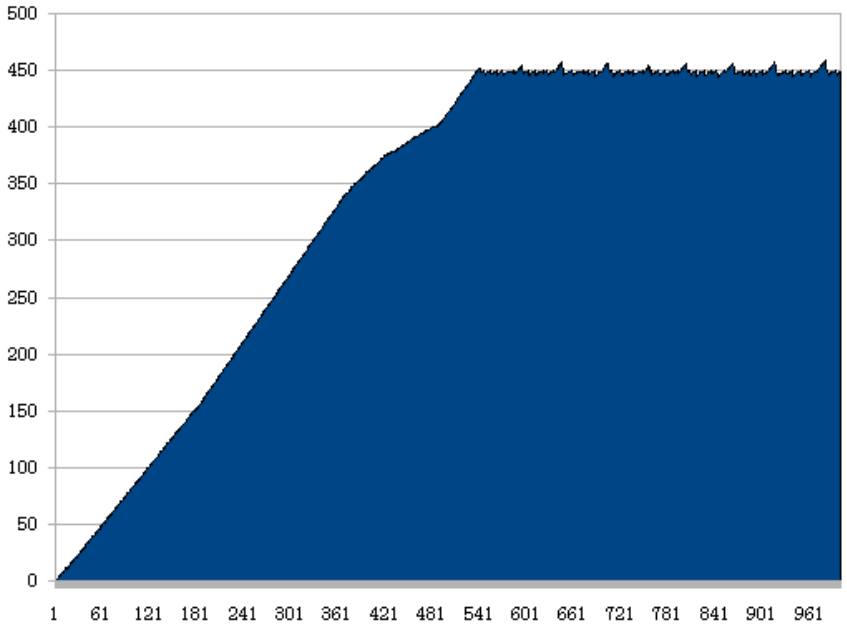

Figura 3.36: Llenado de cola de mensajes de ActiveInterface con tamaño máximo de 500.

La latencia se comporta de manera regular, entorno a 60 milisegundos, y el llenado va aumentando hasta que el buffer ha llegado a su estado medio donde entra el sistema de congestión.

3.5.4. Envío y recepción de 4000 paquetes con tamaño de 2KB con persistencia.

Esta sección se encarga de validar el sistema de persistencia. Mediante el fichero de configuración definido en las pruebas, se va a definir una cola de tamaño pequeño y se va a estresar la aplicación al máximo, haciendo que 4 threads simultáneos produzcan 1000 mensajes cada uno.

El sistema de persistencia es un sistema que se activa cuando un mensaje no puede ser encolado. En ese momento el mensaje es guardado en disco, lo que genera tiempos totales de envío/recepción mayores.

Tamaño de cola de 2 mensajes

Los datos obtenidos en este contexto son:

- **Latencia media de envío/recepción:** 0,000715499 segundos.
- **Tiempo total de envío/recepción:** 10,37900019 segundos.
- **Número de mensajes enviados por segundo:** 385,3935763 mensajes por segundo.

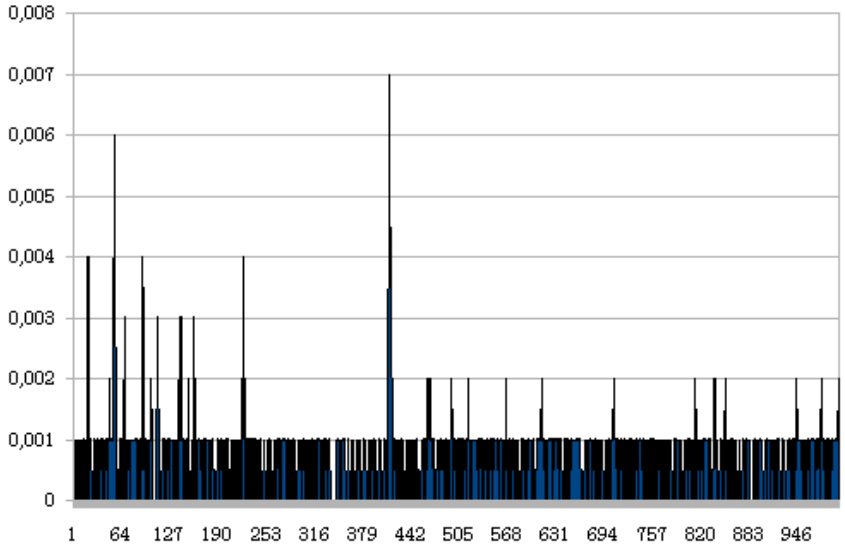

Figura 3.37: Latencia para 1000 mensajes de 1MB con tamaño de cola 2.

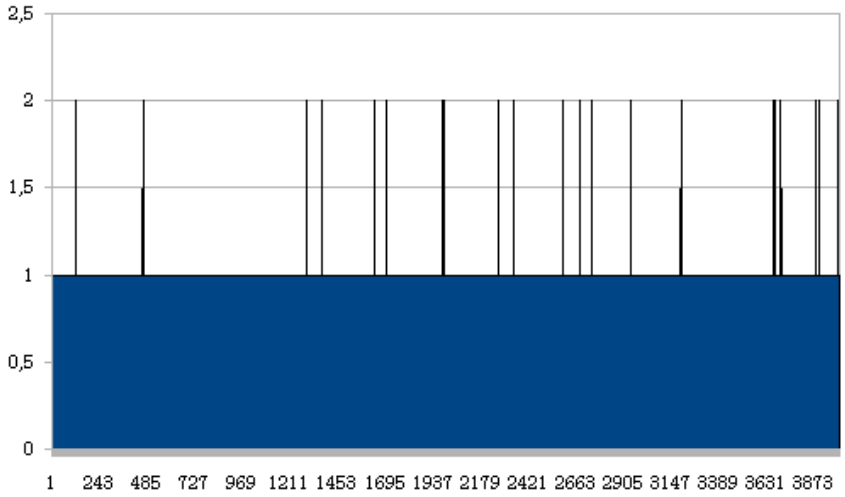

Figura 3.38: Llenado de cola de mensajes de ActiveInterface con tamaño máximo de 2.

Observando la gráfica de llenado, se puede observar como el límite máximo de tamaño de la cola es alcanzado en numerosas ocasiones, obligando a la librería a activar el sistema de persistencia y volcar los datos a disco.

Este proceso es el que induce la reducción drástica del *throughput* a un 33 por ciento del alcanzado sin pérdida de paquetes.

Tamaño de cola de 5 mensajes

Los datos obtenidos en este contexto son:

- **Latencia media de envío/recepción:** 0,001687251 segundos, en torno a 3 milisegundos.
- **Tiempo total de envío/recepción:** 7,519999981 segundos.
- **Número de mensajes enviados por segundo:** 531,914895 mensajes por segundo.

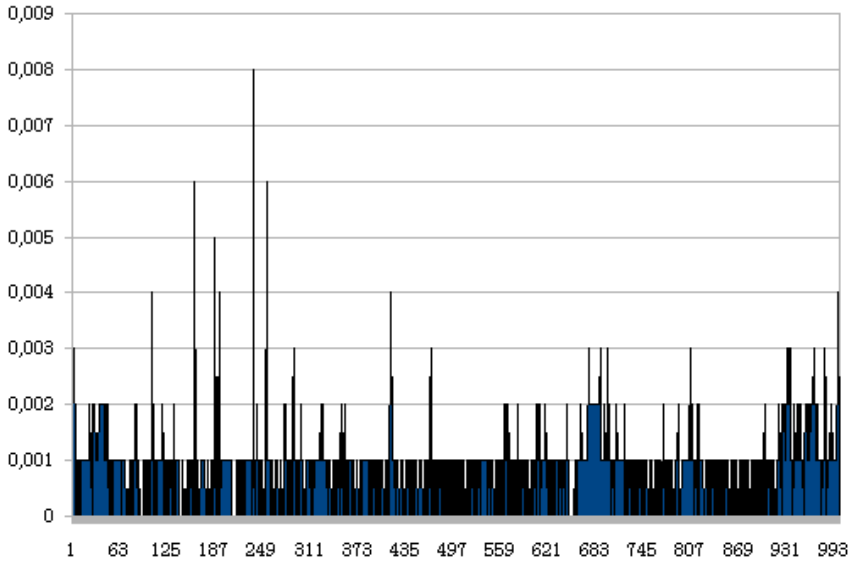

Figura 3.39: Latencia para 1000 mensajes de 1MB con tamaño de cola 5.

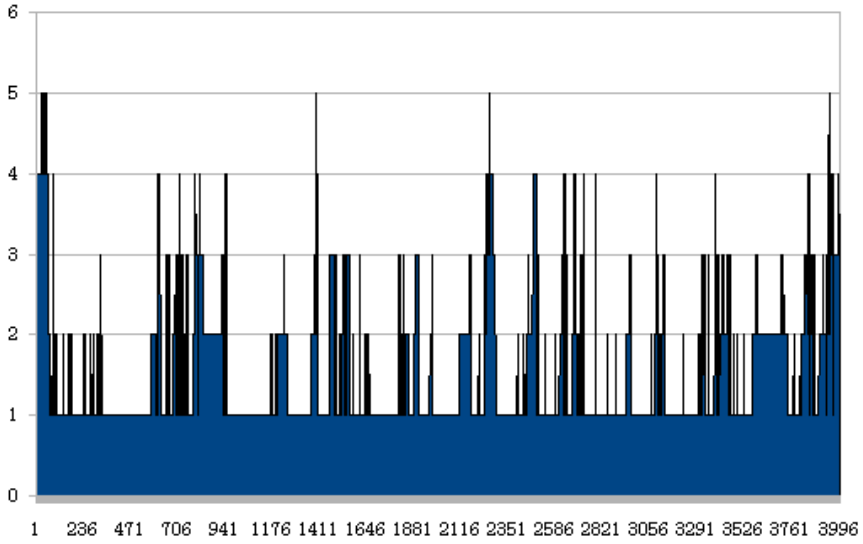

Figura 3.40: Llenado de cola de mensajes de ActiveInterface con tamaño máximo de 5.

Es importante observar la gráfica de llenado de la cola. En esta gráfica se ve que la cola en determinadas ocasiones llega a su tamaño máximo, impidiendo nuevos envíos de mensajes y provocando la activación de la persistencia. Como se puede comprobar con la gráfica de tamaño de cola 2, el número de veces en el que la cola llega a su máximo es mucho menor, y por tanto, la persistencia se activa en menos ocasiones.

Como se observa, la persistencia influye mucho en el rendimiento, y el número de mensajes por segundo baja un 50 por ciento respecto a un comportamiento normal sin desbordamiento del buffer.

Aun así, un mecanismo que ofrezca esa seguridad y además que permita mantener un *throughput* de más de 500 mensajes por segundo es muy valorable.

3.5.5. Envío y recepción de 4000 paquetes con tamaño de 1MB con persistencia.

En esta última prueba se va a intentar validar como trabaja la persistencia con tamaños de ficheros grandes. El acceso a disco es lento y la escritura y lectura en disco de ficheros grandes, hace que el número de mensajes por segundo baje mucho con respecto al caso anterior.

Tamaño de cola de 50 mensajes

En este contexto se han obtenido los siguientes resultados:

- **Latencia media de envío/recepción:** 0,137023998 segundos.
- **Tiempo total de envío/recepción:** 146,8679998 segundos.
- **Número de mensajes enviados por segundo:** 6,808835154 mensajes por segundo.

Se puede observar las siguientes gráficas:

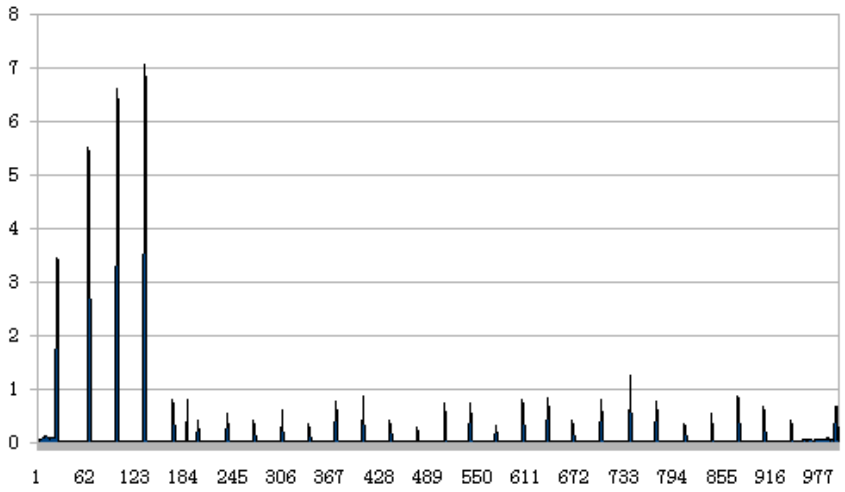

Figura 3.41: Latencia para 1000 mensajes de 1MB con tamaño de cola 50.

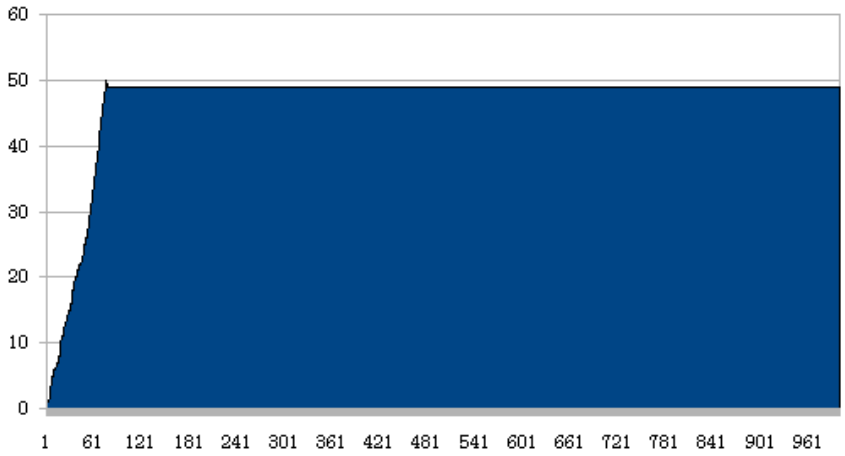

Figura 3.42: Llenado de cola de mensajes de ActiveInterface con tamaño máximo de 50.

Al observar las gráficas de llenado se puede ver como la persistencia es activada sobre el mensaje 100, a partir de ese momento, los mensajes de 1MB son volcados a disco, reduciendo drásticamente el *throughput*.

Capítulo 4

Conclusiones y trabajos futuros

En esta última sección se van a evaluar los resultados del trabajo realizado en este Proyecto de fin de carrera. Se describirán las conclusiones personales, los principales aportes de ActiveInterface, y se hará una breve descripción de los trabajos futuros.

4.1. Principales aportaciones

A lo largo de esta memoria se han descrito las principales aportaciones, pero en este capítulo se van a describir aportaciones más generales, aportaciones que un perfil no técnico podría percibir.

Una de las aportaciones más importantes es sin duda la **reducción del tiempo de desarrollo** que conlleva usar ActiveInterface.
ActiveInterface ha conseguido reducir los tiempos de desarrollo drásticamente. Si cada desarrollador de una empresa implementase su propia librería para distribuir contenido, el conocimiento de la tecnología debería ser más profundo, lo que conllevaría un mayor tiempo de aprendizaje, y un mayor tiempo de codificación.
Cualquier mínima implementación que conlleve usar este tipo de tecnologías, conllevaría un tiempo mucho mayor que integrarse con ActiveInterface.
Además, hay que tener en cuenta el ahorro de tiempo que conlleva el mantenimiento de código. Tener un único producto flexible ayuda a no volver a caer en fallos ya descubiertos en proyectos anteriores.

Algo que ha convertido a ActiveInterface en una herramienta a tener en cuenta ha sido su **flexibilidad**. Se ha integrado en proyectos distintos con objetivos muy distintos. ActiveInterface ha sido diseñado para facilitar y soportar casi cualquier topología de red, entendiendo como topología a cualquier tipo de interacción entre aplicaciones.

Gracias a su flujo de datos configurable, ActiveInterface es capaz de adaptarse y de proveer a los desarrolladores de herramientas para configurar y modificar la forma en la que se comporta tanto estáticamente como en tiempo real.

Como se ha comentado en anteriores capítulos, la mayoría de los desarrollos en C/C++ suelen ser desarrollos costosos, desarrollados íntegramente, sin utilización de *frameworks* o herramientas que lo faciliten, o si se usan, con muy poca abstracción. Por esto ActiveInterface **ayuda a cambiar la forma en la que se desarrolla** en estos entornos.
ActiveInterface es un *framework* de integración *OpenSource* al estilo de los *frameworks* desarrollados en Java. Enfocar los desarrollos C/C++ usando herramientas *OpenSource* ayudaría a crear una serie de librerías OpenSource más estables, permitiendo al desarrollador enfocarse al verdadero negocio de la empresa.

Otra aportación a destacar es que ActiveInterface **contribuye a la apertura al exterior del sistema**. JMS es un protocolo soportado en multitud de lenguajes, como Java, Perl, Python, Ruby, PHP y muchos más. Esto significa que en el momento en el que ActiveInterface se integra en una aplicación nativa C/C++ dicha aplicación es accesible desde toda esta cantidad de lenguajes de programación. Por tanto, este PFC contribuye a crear sistemas multilenguaje sin restricciones, permitiendo usar la potencia de cada uno de ellos en tareas específicas.

Además, el desarrollo de este proyecto ha ayudado a mejorar la librería nativa de conexión con el *broker*, ActiveMQ-CPP. Las pruebas masivas y tests de funcionalidad realizados sobre ActiveInterface, revelaron determinados errores en la librería nativa, que fueron resueltos conjuntamente con una pequeña aportación. Además, en las etapas finales de desarrollo, se ayudó al desarrollador líder de la comunidad ActiveMQ-CPP a realizar pruebas sobre nuevas funcionalidades y a realizar tests para verificar la resolución de *bugs* anteriores.

Por último, hay que destacar una de las principales aportaciones de este Proyecto Fin de Carrera y es su **estabilidad y su funcionalidad**. ActiveInterface es una aplicación no terminada pero sí totalmente funcional, estable, probada y utilizada en entornos de producción.

4.2. Conclusiones personales

Personalmente, y hablando en primera persona, creo que lo que más puedo valorar es que he sido capaz de llevar a cabo una idea planteada por mi mismo y obtener unos resultados buenos.

Personalmente, para mí, lo más importante ha sido *aprender*. Gracias a este proyecto he adquirido conocimientos en una tecnología que no conocía. Y esto es una de las cosas que debería motivarnos a todos.

Gracias a este proyecto he mejorado mis conocimientos generales de desarrollo C++. Con una experiencia no tan corta ya, creo que conseguir desarrollar un producto estable y totalmente funcional, en un entorno no tan orientado a estos aspectos, me ha hecho aplicar una serie de principios metodológicos que me van a ayudar en mi carrera profesional.

Además, creo que desarrollar software que va a ser usado por otros programadores, te hace ver las cosas desde un punto de vista diferente, con una perspectiva más amplia. Para mí, desarrollar una API conlleva muchos aspectos desde la facilidad de uso, hasta implementar una serie de patrones de diseño software mínimos. Desarrollar un producto que cumpla estos principios, implica que no sólo tienes que pensar en que tu software funciona, sino que tu reto es ofrecer lo que te gustaría que te ofrecieran a ti.

Las metodologías SOA probablemente sea una de la metodologías de desarrollo software más usadas en la actualidad. Ha quedado claro por esta memoria que actualmente en C++ las metodologías SOA estaban acotadas a protocolos antiguos e implementaciones C menos usables por el usuario. Aprender una tecnología nueva, sobre un lenguaje potente y con menos abstracción que otros, creo que me ha hecho tener un conocimiento más interno de como funciona la librería nativa, lo que implica tener más conocimiento de la tecnología desde su interior.

Otra conclusión que puedo aportar es que gracias a esta librería he tenido que empezar a hacer las cosas medianamente bien. Acogerse a metodologías ágiles te deja ese punto de posible innovación que creo que ayuda mucho a un programador. Pero, intentar hacer un software que usen otras persona, o incluso intentar crear una comunidad a la que se unan a desarrollar implica tener documentación a todos los niveles, diseño, implementación y manual de usuario. Esto me ha hecho esforzarme más en tareas de documentación, y de

diseño, entrando muy en profundidad, recordando y aprendiendo nuevos conceptos.

Un detalle importante es que aunque ha sido mínimamente, he aportado algo a un proyecto OpenSource que están usando otras muchas personas. Esto me ha hecho aprender como funcionan, y a valorarlas mucho más. Para mí, son gente admirable tanto intelectualmente como personalmente.

Por último, creo que me he comprometido hasta un nivel aceptable con la estabilidad y con el rendimiento. En cuanto a estabilidad, he intentado obviar ciertas partes del desarrollo C más antiguo ya que, en mi opinión, produce un índice de inestabilidad mayor. Inestabilidad que en desarrollos a más bajo nivel es más difícil de depurar. La batería de pruebas también ha ahorrado mucho tiempo de desarrollo y agregado estabilidad. Aunque esta batería no prueba el cien por cien de los posibles caminos, depura muchos aspectos, y con sólo un clic te asegura que tu software realiza las tareas convenientemente.

En cuanto al alto rendimiento, es evidente que nadie usaría una librería que implicase una perdida de rendimiento notable. Tener en cuenta este tipo de conceptos te implica que el software debe tener una calidad al menos media.

Por último, una vez en una charla técnica sobre directrices de desarrollo para un proyecto software, el ponente se despidió con una última frase: *Y sobre todo divertíos.* Y creo que eso es lo que he hecho en este proyecto fin de carrera, divertirme.

4.3. Trabajos futuros

Creo que ActiveInterface ahora mismo cumple una funcionalidad concreta muy valorable, pero hay ciertos desarrollos, que bien acometidos, podrían convertir a ActiveInterface en una librería de comunicaciones completa, acercándose más a soluciones de integración de aplicaciones Java como se ha explicado en capítulos anteriores.

Aunque podría enumerar muchas más, como trabajo en un futuro cercano podría enumerar las siguientes ideas:

- **Ampliación de los protocolos de comunicaciones soportados:** Como ya se ha explicado, actualmente ActiveInterface permite comunicaciones mediante diferentes protocolos pero añadiendo la capa JMS por encima de ellos.

Como trabajo futuro sería importante añadir nuevos protocolos de comunicación para comunicar con distintos tipos de dispositivos. El grado de complejidad del desarrollo sería medio-bajo ya que estos protocolos usarían las estructuras de alto nivel ya desarrolladas en ActiveInterface.

Por ejemplo, sólo con añadir la funcionalidad de conexión mediante puerto serie y mediante TCP/UDP nativamente, se conseguiría una librería capaz de hablar tres distintos protocolos y que simplificaría todo el desarrollo sobre dispositivos que se conectan mediante estos protocolos.

Esto es importante ya que ActiveInterface podría convertirse en el núcleo de comunicaciones de casi cualquier dispositivo, y abstraería a los desarrolladores del medio de comunicación, lo cual en determinados entornos provoca un despilfarro de tiempo y de recursos.
Además, este desarrollo convertiría a ActiveInterface en un elemento de comunicación multiprotocolo y multiplataforma, permitiendo, por ejemplo, un paso de mensajes sencillo entre N dispositivos, cada uno basado en un protocolo distinto.

- **Soporte de lenguaje DSL:** Como se ha explicado en otros capítulos, la mayoría de aplicaciones que se han comparado con este PFC disponen de un lenguaje específico de dominio. Implementar esta funcionalidad, significaría la posibilidad de generar un lenguaje específico para ActiveInterface a partir del cual automatizar determinadas acciones.

  En este caso el desarrollo se ha empezado pero se encuentra en estado muy prematuro para presentarlo en este PFC. Este desarrollo se ha ideado con la idea de proveer de una serie de funcionalidades que puedan ser modificadas en tiempo real mediante un sistema de scripting. Este sistema de scripting permitiría a los usuarios acciones como el reenvió automático de mensajes entre colas, tratamiento de los mensajes en tiempo real, y muchas más tareas que convertirían a ActiveInterface en un router con capacidad de cambio en tiempo real.
  Para implementar esto, se ha empezado a desarrollar con LUA. LUA es un lenguaje de scripting destacable por su rapidez, al que se invocaría, dándole una API para que los usuarios pudieran modificar los mensajes, reenviarlos. Este fichero de scripting permitiría aplicar casi cualquier decisión en tiempo real, sólo cambiando un fichero

de texto usando una serie de primitivas definidas en ActiveInterface.

Gracias a estos dos trabajos futuros descritos, ActiveInterface podría considerarse un framework de integración completo de ayuda que aportaría la mayoría de los patrones de integración descritos en los primeros capítulos de esta memoria.
Sería capaz de tomar decisiones en tiempo real sin tocar ni una línea de código. Además, gracias a su alto grado de abstracción, sin perdida de funcionalidad, ayudaría a los programadores en cualquier tarea de comunicación, facilitandolas, mejorando los tiempos de desarrollo y permitiendo a cada usuario centrarse en su negocio.

Anexo 1: Revisiones de ActiveInterface

ActiveInterface se encuentra actualmente en la versión 1.2. Aunque hay una rama 2.0 sobre la que se desarrollarán los trabajos futuros. La primera versión de ActiveInterface fue lanzada sobre Septiembre de 2010.

A continuación se exponen las distintas revisiones y versiones que se han lanzado.

```
-- Version 1.2 - 12/04/2011

* CHANGES

  - Added some changes in methods of connections to start and stop
    correctly.
  - Added changes needed for activemq-cpp-3.4.0

* FIXED BUGS

  - Producer that never stablish connection.

* KNOWN BUGS

  - Callback thread is property of the connection, if you want
    to delete a connection (by the API) in a callback method, a
    crash ocurrs.

* TO IMPROVE:

  - Support native StreamMessage
  - Check the termination of the thread when a callback is
    stopped in the user part.
```

```
-- Version 1.1 - 07/02/2011

* CHANGES

  - Added some changes to API creating consumers and producers.
  - Added some Windows examples for instantiating the library with
    multithreading support.

* KNOWN BUGS:

  - Related with ActiveMQ-CPP:

      * If a producer never stablish connection with broker, at the
        exit, a non caught exception will be thrown and not caught.

  - ActiveInterface:

      * Callback thread is property of the connection, if you want to
        delete a connection (by the API) in a callback method, a
        crash ocurrs.
```

```
-- Version 1.0 January 2011

* CHANGES:

    - Set version of the library. This is the first version 1.0.
    - Added SSL support.

* FIXED BUGS

    - ActiveMQ-CPP: Fixed the high CPU consuming when a non-blocking
      connection to broker could not be stablished.

    - ActiveInterface: Fixed bug that affects to persistence only in
      Windows environments.

* KNOWN BUGS:

  - Related with ActiveMQ-CPP:

      * If a producer never stablish connection with broker, at the
        exit, a non caught exception will be thrown and not caught.

      * High CPU consuming when we use the non-blocking connection
        to broker.

  - ActiveInterface:

      * Callback thread is property of the connection, if you want
        to delete a connection (by the API) in a callback method,
        a crash ocurrs.
```

```
-- December 22, 2011

* CHANGES

      * First version of persistence system.
      * Added text message to ActiveInterface.

* KNOWN BUGS:

  - Related with ActiveMQ-CPP:

      * If a producer never stablish connection with broker,
        at the exit, a non caught exception will be thrown
        and not caught.

      * High CPU consuming when we use the non-blocking
        connection to broker.

  - ActiveInterface:

      * Callback thread is property of the connection, if you want
        to delete a connection (by the API) in a callback method,
        a crash ocurrs.
```

```
-- October 1, 2011

* KNOWN BUGS:

  - Fixed:

      * When an exception is thrown by the producer thread,
        it produces a crash.

      * Caught some exceptions that if the user does not catch
        it produces a crash.

  - Related with ActiveMQ-CPP:

      * If a producer never stablish connection with broker,
        at the exit, a non caught exception will be thrown
        and not caught.

      * High CPU consuming when we use the non-blocking
        connection to broker.

  - ActiveInterface:

      * Callback thread is property of the connection, if you want
        to delete a connection (by the API) in a callback method,
        a crash ocurrs.
```

Anexo 2: Tabla de opciones de configuración

En la memoria se ha hablado sobre una multitud de conceptos sobre los que se basa ActiveInterface y sus entidades. En la siguiente tabla, se recogen todos los parámetros que se pueden usar tanto en la configuración por XML como en la API.

Parameter	Description	Could be defined when…
id	Is the identifier of the parameter std::string	Allways
ipbroker	Connection uri to broker. Allow all parameters that accept the native URI of CMS	Allways
type	ActiveInterface has 4 types of connections: 0 – ActiveConsumer 1 – ActiveProducer 2 – ActiveConsumer with Response 3 – ActiveProducer with response	Allways
destination	Is the name of the queue/topic from/to you are receiving/sending messages. Std::string with name destination	Allways
persistent	Is used to define if we want that messages produced by producer should be persistent in broker 0 for non persisten 1 for persistent	Only for producers Default 0
topic	Is used to define when a producer or consumer is for a topic or queue. 0 . queue 1. topic	Allways Default 0
selector	Defines the selector string std::string	Only for consumers Default empty
durable	It defines the property of durability of a consumer of a topic (see JMS help) 0 . non durable 1. durable	Only for consumers of a topic Default 0
clientack	Defines the way that a consumer respond to broker to say it that a message was consumed. 0. Auto acknowledge in native CMS code 1. ActiveInterface sent acknowledge when user part ends of processing it	Allways Default 0
maxsizequeue	Defines the size of the queue, this could affect so much to the performance. The size of the queue will be explained in advanced settings.	Only to producers Default unlimited queue
username	Defines a username to connect to a defined queue or topic (std::string)	Allways Default empty
password	Defines the password to connect to a defined queue or topic (std::string)	Allways Default empty
clientid	This parameter is used to identify a client univocally.	Allways, but is obligatory for topic consumers
persistence	Defines the size of the persistence file before to be deleted. More details in advance section.	Only for producers. Default to 0 (no persistence).
certificate	It is used to establish a SSL connection. It is a std::string that should contain the path to a valid certificate in .pem. More details in advance section.	Only when SSL is going to be used. Default empty.

Figura 4.1: Tabla que recoge todas las posibilidades de configuración XML de ActiveInterface.

Bibliografía

[1] Universidad Europea de Madrid. Service oriented architecture [en línea]. 2011. www.esi.uem.es/jccortizo/temasConcu/soa.pdf.

[2] SafeLayer. Enterprise service bus (esb): la infraestructura de interconexión para soa. 2011. http://labs.safelayer.com/es/tecnologia/articulos/386-esbspi.

[3] Juan José Vázquez. Enterprise service bus (esb): la infraestructura de interconexión para soa. 2011. http://blogs.tecsisa.com/articulos-tecnicos/por-que-un-enterprise-service-bus/.

[4] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison Wesley, 2004.

[5] Gartner company. Garnet eip. 2008.

[6] Christy Pettey. Soa will be used in more than 50 percent of new mission-critical operational applications and business processes designed in 2007. 2008. http://www.gartner.com/it/page.jsp?id=503864.

[7] Apache Camel. Qué es apache camel. pages 3–30, 2008.

[8] Grady Booch y James Rumbaugh Jacobson. *El proceso unificado de desarrollo*. Pearson - Prentice Hall, 1999.

[9] Kent Beck. *Extreme programming: Embrace change*. Addison - Wesley, 1999.

[10] www.cplusplus.com. C++ guide. 2000.

[11] Departamento kybele URJC. Ingeniería de requisitos. 2010.

[12] HTML. http://boost.org.