Universidad de las Ciencias Informáticas

Facultad 6

Centro de desarrollo Geoinformática y Señales Digitales

Trabajo de Diploma para optar por el título de

Ingeniero en Ciencias Informáticas

Desarrollo de *middleware* de comunicación para el sistema Video Vigilancia Xilema Suria.

Autor: Alberto Marturelo Lorenzo

Tutor: Ing. Cesar Santos Sanabria

La Habana 2014.

"Año 56 de la Revolución"

Declaración de autoría

Declaro ser el autor de la presente tesis y reconozco a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

Para que así conste firmo la presente a los ____ días del mes de _____ del año _____.

Alberto Marturelo Lorenzo	**Ing. Cesar Santos Sanabria**
_____	_____
Firma del autor	Firma del tutor

Datos de contacto

TUTOR

Ing. Cesar Santos Sanabria
Universidad de las Ciencias Informáticas, La Habana, Cuba.
E-mail: csanabria@uci.cu

Resumen

El Sistema de Video Vigilancia Xilema Suria desarrollado en la Universidad de las Ciencias Informáticas (UCI), permite el control de redes de video vigilancia, el sistema tiene un módulo llamado Gestor, este es el encargado de la comunicación de los demás, así como la coordinación de los procesos y persistencias de los datos. El presente trabajo se basó en el desarrollo de un *middleware* de comunicación orientado a objetos remotos, logrando la comunicación entre los módulos del Sistema Video Vigilancia Xilema Suria utilizando tecnologías libres, posibilitando la escalabilidad y la flexibilidad en el sistema, teniendo servicios de invocación de operaciones en objetos remotos y la publicación y subscripción de eventos. El desarrollo del sistema está sustentado por la metodología ágil XP, con la utilización de la biblioteca de comunicación ZMQ y el marco de trabajo Qt.

Palabras claves: Invocación, *Middleware*, Publicación, Subscripción.

Abstract

The Video Surveillance System Xylem Suria developed at the University Of Informatics Sciences (UCI), all ows the control of video surveillance networks. The system has a module called Manager; this is the respo nsible for the communication of others, as well as coordinating processes and data persistence. This proje ct was based on the development of a *middleware* oriented communication to remote objects, making com munication between the modules of the Video Surveillance System Xylem Suria, using free technologies, e nabling scalability and flexibility in the system, taking invoke services operations on remote objects and pu blish and subscribe to events. The system development is supported by XP agile methodology; with the us e of communication library ZMQ and the framework Qt.

Key Word: Invocation, *Middleware*, Publication, Subscription.

ÍNDICE

TABLA DE CONTENIDOS

TABLA DE CONTENIDOS

ÍNDICE DE LAS ILUSTRACIONES

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE LAS TABLAS

ÍNDICE DE TABLAS

ÍNDICE DE TABLAS

Capítulo 1: Fundamentación teórica

Introducción

En la actualidad la dinámica de los cambios tecnológicos se acelera exponencialmente, en este sentido las empresas requieren proteger sus bienes y controlar a su personal de forma *online* cada segundo. Frente a este escenario los sistemas de video vigilancia han desarrollado nuevas soluciones de hardware y software; dejando atrás los problemas que planteaban los tradicionales circuitos cerrados de televisión (CCTV) como la poca escalabilidad de sus plataformas, respuestas lentas y lineales basadas únicamente en el accionamiento de operadores, limitado acceso al monitoreo y la baja probabilidad de detección de fallas y eventos.

En la Universidad de las Ciencias Informáticas, se encuentra el centro GEYSED[1] el cual posee varios proyectos encaminados al trabajo con redes de video vigilancia, uno de los productos desarrollados es Video Vigilancia Xilema Suria Visión versión 1.0, este contiene los módulos: Análisis, Grabador, Almacenamiento, Visor, Recuperador y Gestor; este último se encarga de que exista comunicación entre los demás módulos. El sistema fue desarrollado en tecnologías privativas, usando *framework .NET* como base para el desarrollo del sistema. El módulo Gestor encargado de la coordinación de los procesos y persistencias de datos, actúa como una pizarra en el sistema. Esto se logró mediante el uso de la tecnología *.NET Remoting,* que permite la comunicación entre los objetos que se ejecutan en procesos diferentes, ya estén en el mismo equipo o en equipos conectados por una red de datos. De esta forma los diferentes módulos se pueden comunicar con el gestor. La versión 2.0 del sistema se desarrolla usando tecnologías libres y multi-plataforma, teniendo como base el *framework Qt* y el lenguaje de desarrollo C++. El estilo arquitectónico continúa siendo Pizarra, por tanto surge la necesidad de comunicar los módulos del sistema con el gestor. El software en su primera versión además de resolver los problemas para los cuales fue concebido, presenta dificultades en cuanto a organización, provocando que la aplicación no sea escalable y flexible a futuras modificaciones.

Debido a la situación problemática antes expuesta se identifica como **problema a resolver**: ¿Cómo lograr la comunicación entre los módulos de Sistema Video Vigilancia Xilema Suria?

Atendiendo al problema planteado y sobre la base de la necesidad de investigación en el marco de esta situación, se tiene como **objeto de estudio** Las tecnologías de comunicación entre aplicaciones y como **campo de acción** *Middleware* de comunicación.

[1] GEYSED: Geo informática y Señales Digitales

Capítulo 1: Fundamentación teórica

En esta investigación se plantea como **objetivo general** desarrollar un *middleware* de comunicación para el Sistema Video Vigilancia Xilema Suria.

Para guiar la investigación se plantean las siguientes **preguntas científicas**:

- ¿Cuáles son los referentes teóricos relacionados con la implementación del *middleware* de comunicación para el sistema Video Vigilancia Xilema Suria?
- ¿Cuáles son los requisitos a implementar para el desarrollo del *middleware* de comunicación para el sistema Video Vigilancia Xilema Suria?
- ¿Cómo realizar el diseño para el desarrollo del *middleware* de comunicación en el sistema Video Vigilancia Xilema Suria?
- ¿Cómo realizar la implementación del *middleware* de comunicación en el sistema Video Vigilancia Xilema Suria?
- ¿Cómo validar el *middleware* de comunicación en el sistema Video Vigilancia Xilema Suria?

Para cumplir con el objetivo general y dar solución al problema planteado se definen las siguientes **tareas de la investigación**:

- Caracterizar los sistemas informáticos que permitan la comunicación entre aplicaciones.
- Caracterizar las herramientas y metodologías a utilizar durante el proceso de desarrollo.
- Definir los requisitos funcionales y no funcionales del *middleware* de comunicación.
- Modelar el sistema haciendo uso de una herramienta CASE[2].
- Definir una arquitectura escalable y flexible.
- Implementar el *middleware* de comunicación para Suria.
- Realizar pruebas unitarias y de aceptación para verificar el correcto funcionamiento del sistema diseñado.

Con la realización de este trabajo de diploma se obtendrán los siguientes resultados:

[2] *Computer Aided Software Engineering*, Ingeniería de Software Asistida por Ordenador.

- Artefactos vinculados a la metodología utilizada.
- Código fuente de la aplicación.
- Informe resultante del proceso de desarrollo del *middleware* de comunicación para el Sistema Video Vigilancia Xilema Suria.

Durante el proceso investigativo de este trabajo fueron utilizados varios métodos científicos, tanto teóricos como empíricos.

Métodos teóricos:

- **Analítico-sintético**: Fue utilizado con el objetivo de establecer, una división del objeto de estudio en varias partes con el fin de realizar un mejor análisis de las características de las mismas.
- **Análisis histórico lógico**: Se utilizó para hacer un estudio sobre la evolución y desarrollo que han tenido los *middleware* de comunicación, lo que permite ver en qué etapa se encuentran dichos sistemas y cuál es la tecnología factible para el desarrollo de los mismos.

Métodos empíricos:

- **Observación**: Se utilizó para obtener información a partir de lo observado de los *middleware* de comunicación que se encuentran actualmente implementados, permitiendo tener una idea de las características posibles que puede tener el sistema.

La **estructura del documento** queda comprendida en: Resumen, Introducción, tres Capítulos, Conclusiones Generales, Recomendaciones, Referencias Bibliográficas, Bibliografía, Anexos y Glosario de términos.

Capítulo 1: En este capítulo se evidenciarán los resultados obtenidos tras la investigación de los diferentes tipos de *middleware* y las posibles soluciones, así como las herramientas, tecnologías y metodología para el desarrollo del sistema.

Capítulo 2: Muestra los resultados obtenidos en el desarrollo de los procesos de análisis y diseño del sistema, así como los diagramas que fueron necesarios para obtener una mayor claridad a la hora de elaborar la solución que se propone.

Capítulo 3: Muestra el modelo de implementación y pruebas como resultado del análisis y diseño estando compuesto por su respectivo diagrama de despliegue, y por su diagrama de componentes, así como las pruebas hechas al producto final.

Capítulo 1: Fundamentación teórica del *middleware* de comunicación

1.1. Introducción

En este capítulo se abordan conceptos relacionados con los sistemas distribuidos y los *middleware*. Se hace referencia a dos posibles soluciones de *middleware* existentes abordando su aplicabilidad en los diferentes entornos, además se hace un análisis sobre las principales tecnologías, metodologías y herramientas a ser usadas en el desarrollo de este trabajo.

1.2. Sistemas distribuidos

Las computadoras de escritorio tienen más potencia hoy en día de lo que poseían los *mainframes*[3] hace algunos años atrás. En el área de las comunicaciones; podemos apreciar avances tales como los sistemas de comunicación vía satélite y teléfonos digitales, provocando con esto que sea posible conectar sistemas informáticos separados físicamente.

Según Omar Hurtado Jara en una investigación sobre el tema plantea que"…los sistemas distribuidos son sistemas cuyos componentes hardware y software, que están en ordenadores conectados en red, se comunican y coordinan sus acciones mediante el paso de mensajes, para el logro de un objetivo. Se establece la comunicación mediante un protocolo prefijado por un esquema cliente-servidor (Jara, 1997)."

Otro concepto asociado a este tema se puede consultar en el libro *Distributed Systems Architecture a Middleware Appoach* donde se plantea que "Un sistema distribuido es un sistema de procesamiento de información en que contiene un número de equipos independientes que cooperan entre sí a través de una comunicación la red con el fin de alcanzar un objetivo específico" (Puder, et al., 2006).

[3] Tipo de computadora más potente y más rápida que existen en un momento dado. Es de gran tamaño, la más grande entre sus paredes. Puede procesar enormes cantidades de información en poco tiempo, pudiendo ejecutar millones de instrucciones por segundo. Está destinada a una tarea específica y poseen una capacidad de almacenamiento enorme.

Capítulo 1: Fundamentación teórica

Un sistema distribuido se define como una colección de computadores autónomos conectados por una red, y con el software distribuido para que el sistema sea visto por los usuarios como una única entidad capaz de proporcionar facilidades de computación.

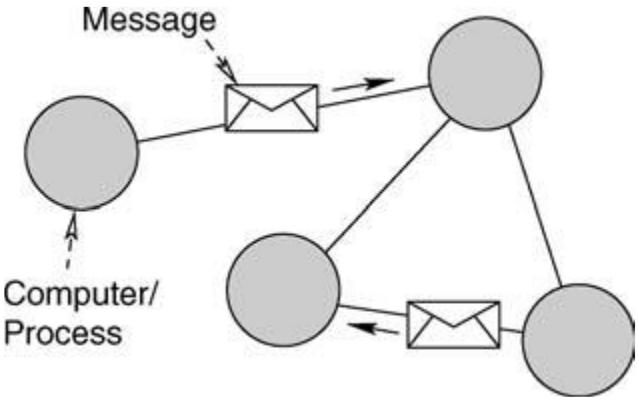

Ilustración 1 Sistema Distribuido (Puder, et al., 2006).

Características de los sistemas distribuidos (Pressman, 2002)

- **Concurrencia**.- Esta característica de los sistemas distribuidos permite que los recursos disponibles en la red puedan ser utilizados simultáneamente por los usuarios y/o agentes que interactúan en la red.
- **Carencia de reloj global**.- Las coordinaciones para la transferencia de mensajes entre los diferentes componentes para la realización de una tarea, no tienen una temporización general, está más bien distribuida a los componentes.
- **Fallos independientes de los componentes.-** Cada componente del sistema puede fallar independientemente, con lo cual los demás pueden continuar ejecutando sus acciones. Esto permite el logro de las tareas con mayor efectividad, pues el sistema en su conjunto continúa trabajando.

Ventajas de los sistemas distribuidos

- **Rendimiento:** El rendimiento de muchos tipos de sistemas distribuidos se puede incrementar añadiendo simplemente más computadoras. Normalmente esta es una opción más sencilla y más barata que mejorar un procesador en un *mainframe*. Los sistemas típicos donde se pueden lograr este incremento en el rendimiento son aquellos en donde las computadoras distribuidas llevan a cabo muchos procesos, y donde la relación trabajo de comunicaciones y procesos es bajo.

- **Recursos compartidos:** Un sistema distribuido permite a sus usuarios acceder a grandes cantidades de datos que contienen las computadoras que componen el sistema. En lugar de tener que reproducir los datos en todas las computadoras se pueden distribuir por un pequeño número de computadoras. Un sistema distribuido también proporciona acceso a servicios especializados que quizás no se requieran muy frecuentemente, y que se puedan centralizar en una computadora del sistema.
- **Tolerancia a fallos:** Un sistema distribuido se puede diseñar de forma que tolere los fallos tanto del hardware como del software. Por ejemplo, se pueden utilizar varias computadoras que lleven a cabo la misma tarea en un sistema distribuido. Si una de las computadoras funciona mal, entonces una de sus hermanas puede hacerse cargo de su trabajo. Una base de datos de una computadora se puede reproducir en otras computadoras de forma que si la computadora original tiene un mal funcionamiento, los usuarios que solicitan la base de datos son capaces de acceder a las bases de datos reproducidas (Jara, 1997).

Desventajas de los sistemas distribuidos

El principal problema del software, es el diseño, implantación y uso del software distribuido, pues presenta numerosos inconvenientes. Las principales interrogantes son las siguientes:

- ¿Qué tipo de S. O. (Sistema Operativo), lenguaje de programación y aplicaciones son adecuados para estos sistemas?
- ¿Cuánto deben saber los usuarios de la distribución?
- ¿Qué tanto debe hacer el sistema y qué tanto deben hacer los usuarios?
- Otro problema tiene que ver con las redes de comunicación. Por ejemplo: Pérdida de mensajes, saturación en el tráfico.
- Un problema que puede surgir al compartir datos es la seguridad de los mismos.

Desafíos de los sistemas distribuidos

Heterogeneidad de los componentes: La interconexión, sobre todo cuando se usa Internet, se da sobre una gran variedad de elementos hardware y software, por lo cual necesitan de ciertos estándares que permitan esta comunicación. Los *middlewares*, son elementos de software que permiten una abstracción de

la programación y el enmascaramiento de la heterogeneidad subyacente sobre las redes. También el *middleware* proporciona un modelo computacional uniforme.

Extensibilidad: Determina si el sistema puede extenderse y ser implementado en diversos aspectos (añadir y quitar componentes). La integración de componentes escritos por diferentes programadores es un auténtico reto.

Seguridad: Reviste gran importancia por el valor intrínseco para los usuarios. Tiene tres componentes:

- Confidencialidad: Protección contra individuos no autorizados.
- Integridad: Protección contra la alteración o corrupción.
- Disponibilidad: Protección contra la interferencia con los procedimientos de acceso a los recursos.

Escalabilidad: El sistema es escalable si conserva su efectividad al ocurrir un incremento considerable en el número de recursos y en el número de usuarios.

Tratamiento de Fallos: La posibilidad que tiene el sistema para seguir funcionando ante fallos de algún componente en forma independiente, pero para esto se tiene que tener alguna alternativa de solución.

Técnicas para tratar fallos:

- **Detección de fallos.** Algunos fallos son detectables, con comprobaciones por ejemplo.
- **Enmascaramiento de fallos.** Algunos fallos detectados pueden ocultarse o atenuarse.
- **Tolerancia de fallos.** Sobre todo en Internet se dan muchos fallos y no es muy conveniente ocultarlos, es mejor tolerarlos y continuar. Ej. Tiempo de vida de una búsqueda.
- **Recuperación frente a fallos.** Tras un fallo se deberá tener la capacidad de volver a un estado anterior.
- **Redundancia.** Se puede usar para tolerar ciertos fallos (DNS, BD)

Concurrencia. Compartir recursos por parte de los clientes a la vez.

Transparencia. Es la emasculación al usuario y al programador de aplicaciones de la separación de los componentes en un sistema distribuido. Se identifican 8 formas de transparencia:

- **De acceso.** Se accede a recursos locales y remotos de forma idéntica.
- **De ubicación.** Permite acceder a los recursos sin conocer su ubicación.
- **De concurrencia.** Usar un recurso compartido sin interferencia.
- **De replicación.** Permite utilizar varios ejemplares de cada recurso.
- **Frente a fallos.** Permite ocultar los fallos.
- **De movilidad.** Permite la reubicación de recursos y clientes sin afectar al sistema.
- **De prestaciones.** Permite reconfigurar el sistema para mejorar las prestaciones según su carga.
- **Al escalado.** Permite al sistema y a las aplicaciones expandirse en tamaño sin cambiar la estructura del sistema o los algoritmos de aplicación (Jara, 1997).

1.3. Descripción del objeto de estudio

El actual sistema de Video Vigilancia Xilema Suria es un sistema distribuido estructurado de la siguiente forma representada por la Ilustración 2: Estructura del sistema Video Vigilancia Xilema Suria:

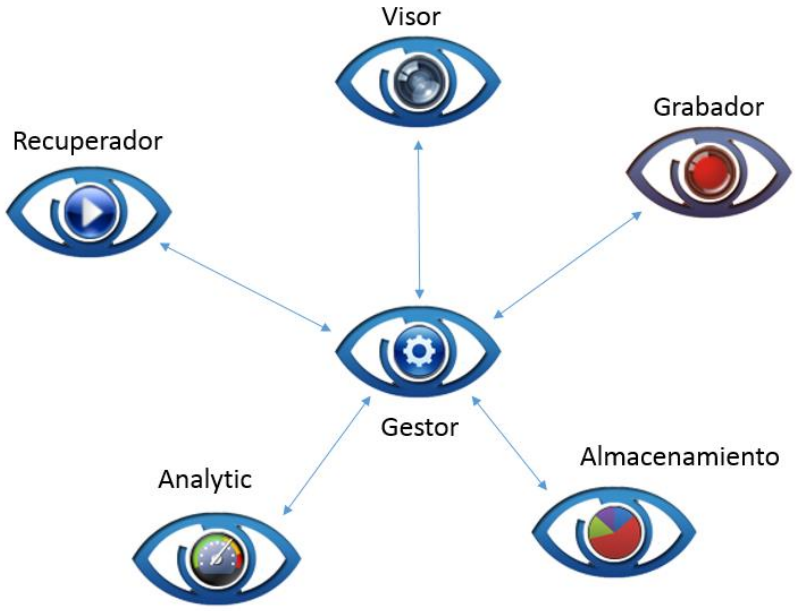

Ilustración 2 Estructura del sistema Video Vigilancia Xilema Suria.

Recuperador: Es el encargado de recuperar los videos almacenados en los servidores, hacer búsquedas, servir los resultados al usuario y darle la posibilidad de la reproducción de los mismos. Pueden existir varias instancias de esta estación corriendo en los dominios físicos del sistema.

Análisis: Es el módulo encargado de realizar el procesamiento de los flujos de videos capturados por las cámaras o almacenados en el sistema bajo petición de un cliente o por configuración, que puede ser por horarios establecidos o por la ocurrencia de algún evento. Pueden existir varias instancias de esta estación corriendo en los dominios físicos del sistema.

Grabador: Es el módulo encargado de almacenar los flujos de video obtenidos de las cámaras, bajo petición de un cliente determinado o por configuración, que puede ser por horarios establecidos o por la ocurrencia de algún evento. Pueden existir varias instancias de esta estación corriendo en los dominios físicos del sistema.

Visor: Es el módulo que permite visualizar los flujos de videos capturados por las cámaras. Pueden existir varias instancias de esta estación corriendo en los dominios físicos del sistema. Tiene la capacidad de reflejar todo el aspecto organizativo con que se manejan las cámaras internamente, además de poder visualizarlas de manera independiente o colectiva (a manera de vistas). También permite manipularlas en medida de las capacidades de cada una. Este módulo puede trabajar de manera cooperativa con el Recuperador, tras previa coordinación del Gestor, para la recuperación de videos almacenados.

Gestor: Es el módulo fundamental de la aplicación, y todos los demás son agentes que se encargan de realizar tareas específicas. Todo el tráfico de información pasa por el Gestor facilitando el control y supervisión, permitiendo además la flexibilidad del sistema. Es el único que interactúa directamente con la Base de datos.

Almacenamiento: Es el módulo encargado de controlar la capacidad en disco en el almacén de grabaciones, capaz de tomar decisiones según las reglas definidas por el usuario.

El Gestor debe de estar siempre activo en el sistema, y desde que se inicia mantiene una lista de todas las cámaras, de su configuración y todo tipo de información, que obtiene de la Base de Datos. Para cada cliente registra información de su actividad, asigna un identificador temporal a ese cliente, que lo distinguirá de las demás estaciones del sistema y le permitirá realizar operaciones futuras.

Del Gestor se descargará toda la información relacionada a las cámaras y otros elementos de configuración necesarios para el funcionamiento de los diferentes módulos. El operador podrá entonces manipular los videos e imágenes, así como las cámaras y hacer cambios sobre estas, si tiene los permisos. Cualquier

modificación a los datos existentes se le solicita al Gestor que la procesa y avise a los demás módulos del sistema que se encuentren activos, para que reflejen dicho cambio.

1.4. *Middleware*

La presencia de las redes tanto a nivel local (por ejemplo, redes de área local) y a nivel global (por ejemplo, internet) permite la proliferación de aplicaciones distribuidas, ya que la esencia es la ejecución de una aplicación en diferentes ubicaciones físicas. Por tanto surge la interrogante: ¿Qué se necesita para implementar un sistema distribuido?, se necesita una infraestructura que convenientemente apoye el desarrollo y la ejecución de aplicaciones distribuidas. Una plataforma de *middleware* presenta dicha infraestructura, proporciona un amortiguador entre aplicaciones y la red. La red simplemente provee un mecanismo de transporte, el acceso a ella depende a gran medida de factores y diferentes tecnologías entre diferentes plataformas físicas como se muestra en la (Ilustración 3 *Middleware*).

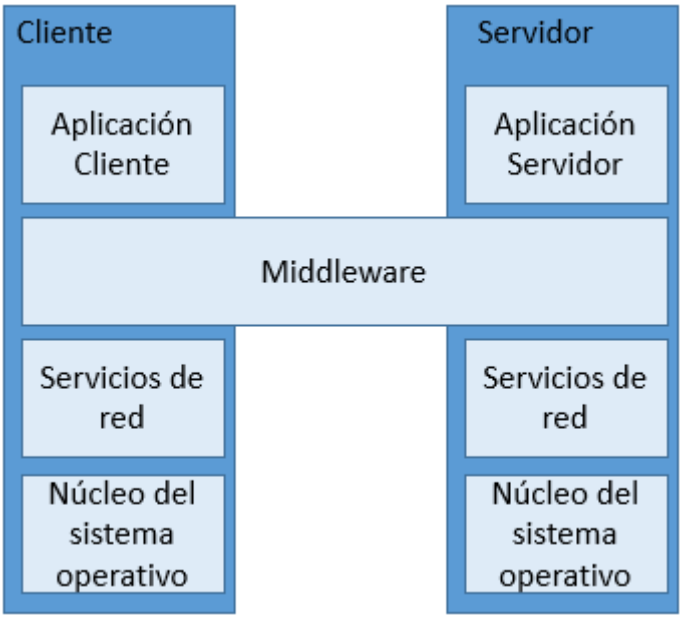

Ilustración 3 *Middleware*

El *middleware* es una clase de tecnología de software diseñado para ayudar a gestionar la complejidad y heterogeneidad inherente a sistemas distribuidos. Se define como una capa de software encima del sistema operativo, pero a continuación el programa de aplicación que proporciona una abstracción de programación común a través de un sistema distribuido, como se muestra en Fig. 4 (Bakken, 2003).

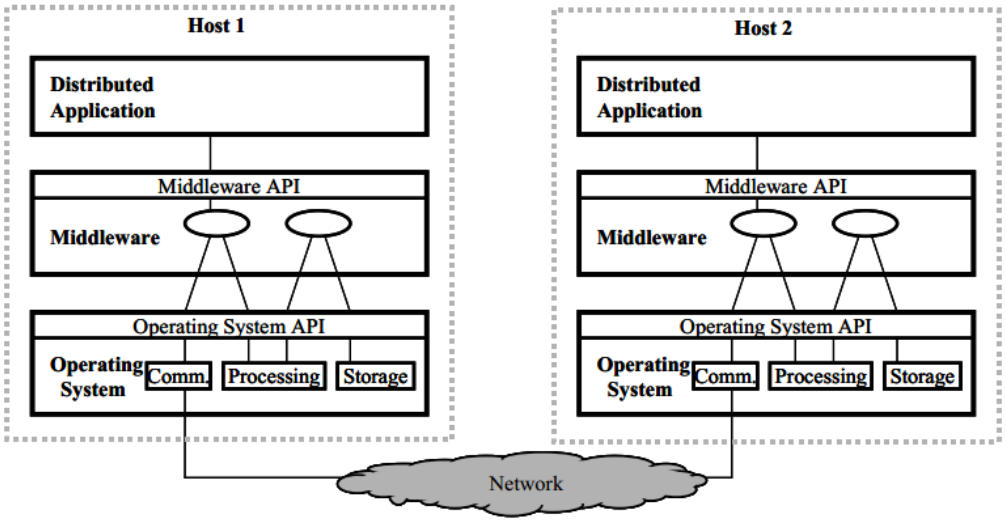

Ilustración 4 Arquitectura de un *Middleware*.

Origen y evolución

En la ilustración 5 se puede apreciar el comportamiento que ha tenido la evolución de los *middleware* a lo largo de las últimas décadas, mostrándose el uso y el avance que estos han tenido a través de los años.

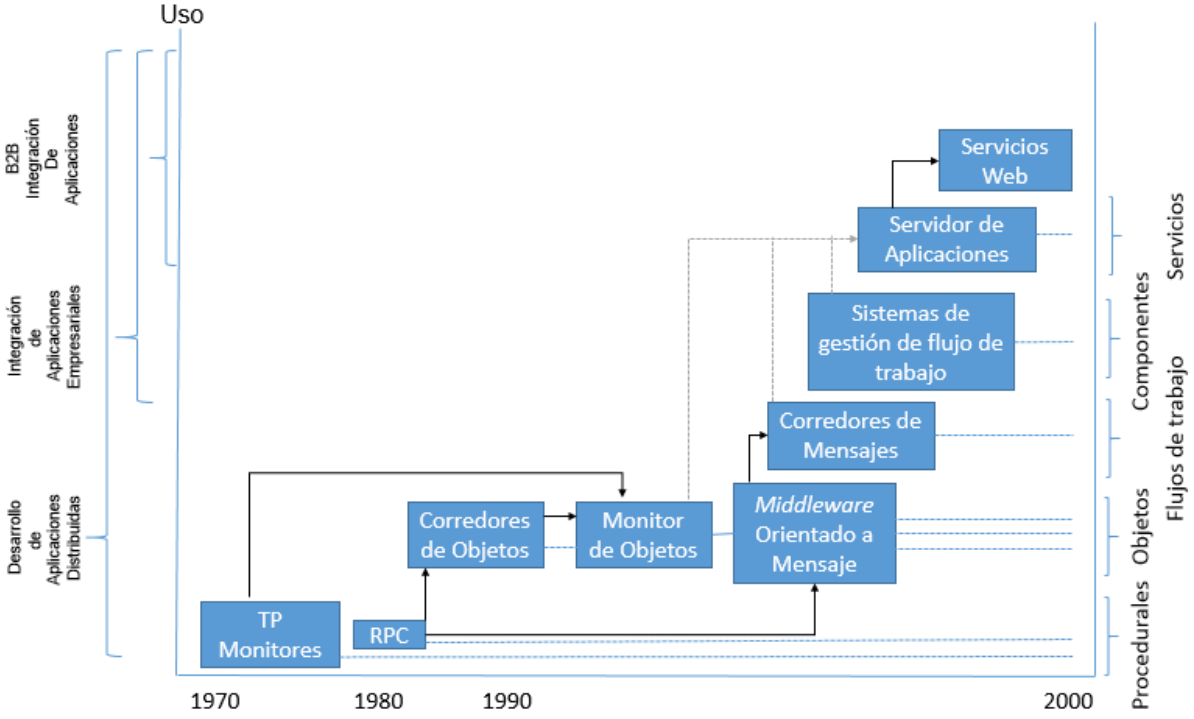

Ilustración 5 Origen y evolución de los *middleware*.

1.5. Tipos de *middleware*

Existen diferentes tipos de *middleware*. Estos varían en cuanto a función de la programación, abstracciones que prestan y los tipos de heterogeneidad que ofrecen más allá de la red y hardware.

- ***Distributed Tuples***: Ofrece la abstracción de tuplas distribuidas, y son las que mayor despliegue tienen hoy en día. Su lenguaje de consulta estructurado (SQL) permite a los programadores manipular conjuntos de estas tuplas (una base de datos) en un idioma como la semántica intuitiva y rigurosa, fundamentos matemáticos basados en la teoría de conjuntos y el cálculo de predicados. Bases de datos relacionales distribuidas también ofrecen la abstracción de una transacción.

- ***Remote Procedure Call*:** RPC[4] *middleware* amplía la interfaz de llamada a procedimiento familiar para prácticamente todos los programadores con el objetivo de ofrecer una abstracción de la posibilidad de invocar un procedimiento cuyo cuerpo es a través de una red. Los sistemas RPC son generalmente síncronos, y por lo tanto no ofrecen ninguna posibilidad de paralelismo sin necesidad de utilizar múltiples hilos, y que suelen tener limitadas las instalaciones de manejo de excepciones.

- ***Message Oriented Middleware*:** MOM [5] se pueden dividir en dos tipos, espera y publicación/subscripción. El paso de espera se puede dividir en mensaje y espera. El paso de mensaje inicia cuando la aplicación envía un mensaje a uno o más clientes, con el MOM del cliente. El servidor MOM, recoge las peticiones de la cola (*Message Broker*) en un orden o sistema de espera predeterminado. Los actos del servidor MOM son como un *router* y usualmente no interactúan con estas. El MOM de publicación y subscripción actúa de manera ligeramente diferente, es más orientado a eventos. Si un cliente quiere participar por primera vez, se une al bus de información. Dependiendo de su función, si es como publicador, suscriptor y ambas, este registra un evento. El publicador envía una noticia de un evento al bus de memoria. El servidor MOM envía un anuncio al subscriptor registrado cuando la información está disponible.

[4]Llamada a procedimiento remoto

[5]Middleware orientado a mensajes

- ***Distributed Object Middleware*:** DOM[6] proporciona la abstracción de un objeto remoto cuyos métodos se pueden invocar, al igual que las de un objeto en el mismo espacio de direcciones que el que realiza la invocación. Hacen que todas las técnicas de encapsulación orientadas a objetos sean beneficiosa, herencia y polimorfismo a disposición del desarrollador de aplicaciones distribuidas (Bakken, 2003).

Por el estudio realizado de los diferentes tipos de *middleware* existentes, se llega a la conclusión que el *Distributed Object Middleware* es el que cumple con las condiciones para darle solución a la problemática, el sistema será un servidor que posee un conjunto de objetos y los diferentes clientes conectados invocarán sus operaciones.

1.6. Posibles soluciones

En este acápite serán analizadas varias tecnologías que son candidatas para la solución del problema planteado. La biblioteca de mensajes asíncronos ZeroMQ licenciado bajo LPGL[7] (GPL reducida) y escrito en C++. El *middleware* de mensajería RabbitMQ licenciado bajo MPL (*Mozilla Public License)*, también dos tecnologías de *middleware*, DDS[8] e ICE[9]. DDS (Sistema de Datos Distribuidos) es un estándar del OMG[10] (Grupo de Administración de Objetos), y se pondrá un énfasis en la implementación que hace la empresa RTI[11]. Por otro lado, ICE, *middleware* desarrollado por la empresa ZeroC y licenciado bajo GPL. Este estudio se basará en analizar las ventajas e inconvenientes que puedan presentar los distintos tipos de aplicaciones en que estas se enmarcan.

[6]Middleware de objetos distribuidos

[7] **Licencia Pública General Reducida de GNU**, o más conocida por su nombre en inglés ***GNU Lesser General Public License*** (antes *GNU Library General Public License* o Licencia Pública General para Bibliotecas de GNU), o simplemente por su acrónimo del inglés **GNU LGPL**, es una licencia de software creada por la Free Software Foundation que pretende garantizar la libertad de compartir y modificar el software cubierto por ella, asegurando que el software es libre para todos sus usuarios.

[8]Data Distribution Sevice

[9]Internet Communications Engine

[10]Object Management Group

[11]Remote Technologies Inc

1.6.1. DDS (Data Distribution Service)

DDS es un *middleware* de tipo tiempo real de publicación-subscripción centrado en los datos, son varias empresas que implementan este estándar, pero esto no presenta ningún inconveniente ya que todas se basan en el estándar OMG.

Es necesario recalcar que un *middleware* sea realmente independiente de plataforma y lenguaje, esto facilita la implantación de la tecnología en distintos dispositivos, la escalabilidad será un elemento clave, junto a otras características de DDS para poder entender por qué es realmente aplicable en los sistemas de video vigilancia.

DDS es un *middleware* de publicación-subscripción. En el paradigma publicación-subscripción se definen dos entidades, un "*publisher*", que es el encargado de enviar los datos a la plataforma, y un "*subscriber*", cuyo cometido es recoger de la plataforma aquellos datos a los cuales esté suscrito.

La característica que presenta el paradigma publicación-subscripción de independencia es clave de DDS, se dice que es una tecnología centrada en los datos, precisamente por esta característica que consigue un desacoplamiento muy grande en la aplicación. Además, esta independencia no sólo presenta una característica clave que trae como consecuencia una gran flexibilidad a la hora de desarrollar para distintas plataformas y lenguajes, sino que también potencia la escalabilidad del sistema. Se puede entender por escalabilidad porque en cualquier momento se puede añadir un "*publisher*" o un "*subscriber*" al sistema sin tener que tomar en cuenta el resto de los elementos que haya.

No se puede pasar por alto otras alternativas de *middleware* con características de tiempo real, como por ejemplo RT-CORBA**,** que ha sido con diferencia relevante el *middleware* más utilizado en los últimos años, ha caído en desuso debido al alza de los sistemas basados en publicación-subscripción y el auge del estándar DDS (Fernández., 2009).

Esta solución aparte de ser útil, queda descartada ya que las implementaciones de CORBA se encuentran bajo licencias diferentes, una privativa y otra libre, pero en muchos casos las versiones libres no tienen la calidad y soporte necesario debido a que la mayoría de los esfuerzos se dedican a la privativa; por ejemplo TAO, ORBit, MICO y JacORB son gratis y libres, pero las mismas son básicas y para obtener las versiones extendidas de cada una hay que comprarla.

1.6.2. ICE (Internet Communications Engine)

ICE es un *middleware* libre licenciado bajo GPL, desarrollado por la empresa ZeroC y cuya motivación es responder a la complejidad de CORBA[12] proporcionando una funcionalidad similar.
Existen muchas tecnologías de *middleware* distintas. Entre las cuales se encuentran RMI[13] de Java, WCF[14] de Microsoft, el cual es dependiente de la plataforma Windows, uno de los más extendidos, independientemente de la plataforma y de lenguaje, y también uno de los más complejos en su utilización (ZeroC, 2011).

Los objetivos principales de ICE son:

- Proveer un *middleware* orientado a objetos, disponible para plataformas heterogéneas.
- Facilitar la utilización de distintos paradigmas, uno de ellos el de publicación-subscripción, que es en el que se centrará la atención.
- Hacer una plataforma más sencilla, favoreciendo la implantación, uso y aprendizaje de ésta.
- Que la plataforma sea eficiente en cuanto al uso de CPU, memoria y ancho de banda (ZeroC, 2011).
- Que la plataforma sea segura por defecto, sin que haya necesidad de añadir parches posteriormente.

Mediante el estudio realizado sobre este *middleware* se descarta su utilización ya que está liberado bajo la licencia GPL y a la universidad no le es factible producir el software bajo estas condiciones.

[12]Common Object Recuest Broker Architecture

[13]Remote Method Invocation

[14]Windows Communication Foundation

1.6.3. ZeroMQ

ZeroMQ (también conocido como 0MQ o ZMQ) es una librería de mensajería desarrollada por *iMatix Corporation* junto a una gran comunidad de colaboradores. En un principio se presentó como "*middleware* de mensajería" y en este momento se define como una "nueva capa en la pila de red". ZeroMQ no es un sistema completo, sino que es una biblioteca simple de usar mediante programación. Básicamente ofrece una interfaz socket que permite construir rápidamente un sistema de mensajería. Puede hacer uso de los protocolos de transporte tales como IPC y Multicast, además de hacer uso de la dosificación inteligente de mensajes lo que le permite utilizar eficientemente una conexión TCP/IP reduciendo al mínimo no sólo la sobrecarga de protocolo, sino también las llamadas al sistema **(Piël, 2010)**.

Los patrones ZeroMQ básicos son:

- ***Request-reply*:** Se conecta un conjunto de clientes a un conjunto de servicios. Esto es una llamada a procedimiento remoto y patrón de distribución de tareas como se muestra en la ilustración siguiente.

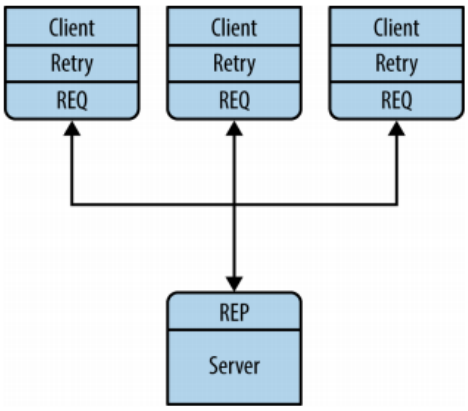

Ilustración 6 Patrón *Request and Repier*

- ***Publish-subscribe*:** Se conecta un conjunto de editores para un conjunto de *subscriptores*. Este es un patrón de distribución de datos como se muestra en la ilustración siguiente.

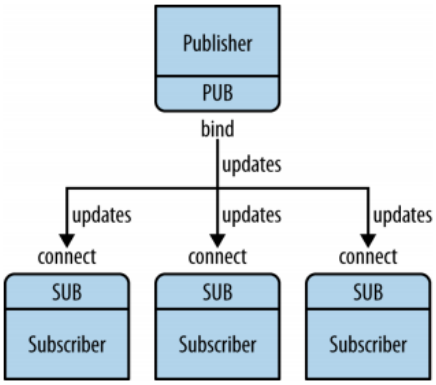

Ilustración 7 Patrón *Publisher and Suscriber*

- ***Push-pull:*** Conecta los nodos de un patrón de abanico en un abanico de salida que puede tener varios pasos, y lazos. Se trata de una distribución de tareas en paralelo y el patrón colección.
- ***Exclusive pair:*** Conecta dos sockets en un par exclusivo.(Este es un patrón avanzado de bajo nivel para los casos de uso específicos)

Cada patrón define una topología de red en particular. ***Request-reply*** define la llamada " bus de servicio", ***Publish-subscribe*** define el "árbol de la distribución de datos", ***push-pull*** define el "pipeline paralelizados". Todos los patrones se han diseñado deliberadamente de tal manera como para ser infinitamente escalable y por lo tanto utilizable en escala de Internet (iMatrix, 2013).

Algunas de las ventajas:

- Actúa como framework de concurrencia.
- Más rápido que TCP para productos en clúster.
- Capaz de intercambiar mensajes dentro de un mismo proceso, IPC, TCP y multicast.
- Soporta mensajería *Publish-Suscribe* y *Request-Reply*.
- Soporte I/O asíncrono.
- *Binding* para más de 30 lenguajes incluidos C, C++, Java, .Net.
- Soporte para la mayoría de sistemas operativos como Linux, AIX, Windows,
- Software LGPL con soporte comercial desde iMatix.

1.6.4. RabbitMQ

RabbitMQ es un software de negociación de mensajes, que entra dentro de la categoría de *middleware* de mensajería, desarrollado y mantenido por *Rabbit Technology Ltd*. Implementa el estándar AMQP lo que proporciona que tenga un esquema de enrutamiento flexible. El servidor RabbitMQ está escrito en *Erlang* y utiliza el framework OTP para construir sus capacidades de ejecución distribuida y conmutación ante errores. RabbitMQ es compatible con varios clientes, siendo tres de ellos considerados como oficiales, estos son Java, .NET/C# y Erlang. Sin embargo, debido al apoyo sustancial de la comunidad hay una serie de otros clientes en Python, Perl, Ruby entre otros. Ofrece varios *plugins* para ampliar su funcionalidad entre los que se encuentra un plugin para el protocolo STOMP. Posee soporte para SSL y asume una conexión TCP. Al igual que ActiveMQ soporta ambos modelos de mensajería Punto-a-Punto y Publicación-Subscripción. Es un software de código abierto y está liberado bajo la licencia MPL. Está basado en una plataforma comprobada, ofreciendo alta confiabilidad, disponibilidad y escalabilidad, junto con un rendimiento y latencia predecibles y consistentes. Tiene un código fuente compacto y fácil de mantener, que permite una rápida adaptabilidad (Jankolovska, et al., 2012).

El proyecto RabbitMQ consta de diferentes partes:

- El servidor de intercambio RabbitMQ en sí mismo
- Pasarelas para los protocolos HTTP, XMPP y STOMP.
- Bibliotecas de clientes para Java y el *framework* .NET.
- El *plugin Shovel* (pala) que se encarga de copiar (replicar) mensajes desde un corredor de mensajes a otros.

1.7. Comparativa entre ZeroMQ y RabbitMQ

El protocolo AMQP está desarrollado sobre RabbitMQ (junto con Apache Qpid). Por lo tanto, se implementa una arquitectura de agente, lo que significa que los mensajes se ponen en cola en un nodo central antes de ser enviado a los clientes.

Este enfoque hace RabbitMQ muy fácil de usar e implementar, porque los escenarios avanzados como enrutamiento, balanceo de carga o colas de mensajes persistentes se apoyan en unas pocas líneas de código. Sin embargo, también hace que sea menos escalable y " lento " debido a que el nodo central agrega latencia sobre mensajes que son bastantes grandes (Nawaz, 2009).

ZeroMQ es un sistema de mensajería muy ligero especialmente diseñado para alto rendimiento bajo escenarios de latencia como el que se puede encontrar en el mundo financiero. ZMQ es compatible con muchos escenarios de mensajería avanzados pero contrariamente a RabbitMQ. ZMQ es muy flexible (Nawaz, 2009).

Quedando explicita la selección de la solución después de analizar la anterior comparativa, se llega a la conclusión de que ZMQ es la opción más viable para la solución de la problemática, por sus diferentes facilidades y patrones implícitos ya implementados.

1.8. Metodología, tecnologías y herramientas a usar en el desarrollo del sistema

Las tecnologías a usar a la hora de desarrollar cualquier aplicación están en estrecha dependencia con los lenguajes de programación a utilizar.

1.8.1. Lenguaje de programación: C++ 98

Algunas de las características de este lenguaje expuestas a continuación fue lo que llevó a ser seleccionado como lenguaje de desarrollo de dicho sistema.

1. El C++ es un lenguaje orientado a objetos el cual posee características y cualidades de las que carecía el lenguaje C con lo cual fue enriquecido y mejorado.

2. El proyecto definió Qt como *framework* para el desarrollo de las aplicaciones en el proyecto y el mismo está escrito en C++.

3. El C++ es uno de los lenguajes más potentes porque permite programar a alto y a bajo nivel, es complicado porque se debe hacer casi todo de manera manual.

El C++ por lo tanto es un lenguaje muy completo, el mismo permite la programación multiplataforma, y es a su vez conciso, eficaz y claro. Desde su surgimiento fue el lenguaje utilizado por la mayoría de los

desarrolladores para sus proyectos. Por esta razón este lenguaje es ideal para el desarrollo de cualquier aplicación en general (Delgado, 2011).

1.8.2. Framework de desarrollo: Qt 5.0.2

Desde el punto de vista del desarrollo de software, un *framework* es una estructura de soporte definida en la cual otro proyecto de software puede ser organizado y desarrollado.

Para el desarrollo del sistema se utilizó como *framework* de desarrollo Qt en su versión 5.0.2, debido a que es un *framework* multiplataforma, lo cual facilita que la solución pueda adaptarse tanto para el sistema operativo Linux, Windows o Mac, permitiendo que una aplicación pueda ser compilada y utilizada en cualquier plataforma sin necesidad de cambiar el código constantemente. Utiliza C++ como lenguaje de programación brindando una serie de facilidades a la hora de implementar, al ser orientado a objetos y contar con un conjunto de bibliotecas con clases y herramientas incluidas, las mismas están bien documentadas y son muy fáciles de usar.

1.8.3. Herramienta de modelado: Visual Paradigm para UML 8.0

Visual Paradigm for UML es una herramienta ampliamente utilizada en el mundo del software que permite a los profesionales modelar sus diseños. Esta herramienta fue desarrollada para una amplia gama de usuarios, incluyendo ingenieros de software, analistas de sistemas, analistas del negocio y arquitectos de sistemas. Es un software privativo, pero brinda una versión libre para uso no comercial. Proporciona un diseño centrado en casos de uso y enfocado al negocio que genera un software de mayor calidad. Es capaz de generar código e ingeniería inversa para varios lenguajes de programación. Permite manejar grandes estructuras de manera eficiente. Soporta el ciclo de vida completo del desarrollo de software: análisis y diseño orientado a objetos, construcción, pruebas y despliegues. *Visual Paradigm for UML* ha sido desarrolla para todos los sistemas operativos compatibles con Java, incluyendo Windows, Linux y Mac OS X (Luis Díaz, et al., 2013).

Luego del estudio realizado de la herramienta Visual Paradigm, se determinó que es conveniente el uso de la misma para el trabajo con la ingeniería de software, ya que posee versiones multiplataforma, logrando que el producto tenga la calidad requerida. Es la herramienta CASE para la modelación que se utiliza en el proyecto Video Vigilancia Xilema Suria.

1.8.4. Entorno de Desarrollo Integrado: Qt Creator 2.6.2

Qt Creator es un IDE de desarrollo multiplataforma, viene acompañado de un conjunto de herramientas para facilitar su uso.

Principales características de Qt Creator:

- Posee un avanzado editor de código C++.
- Soporta los lenguajes: C#/.NET, Python: PyQt y PySide, Ada, Pascal, Perl, PHP y Ruby.
- Posee herramienta para proyectos y administración.
- Posee ayuda sensible al contexto integrado.
- Posee depurador visual.
- Resaltado y auto-completado de código (Cambiaso, 2010).

1.8.5. *Metodología a usar en el desarrollo del software del sistema: Programación Extrema*

El desarrollo de un software se rige sobre las bases de la guía de una metodología acorde al mismo, la cual sirve de guía en el proceso de producción del producto final. Las metodologías de desarrollo se dividen en dos grandes grupos de acuerdo al énfasis realizado en la documentación del desarrollo, son conocidas como metodologías ágiles y metodologías tradicionales o robustas.

El tema de la metodología a seleccionar es uno de los más complicados en cualquier proceso de desarrollo de software, son muchos los factores que se deben incluir en el análisis. A continuación se hace un estudio de las metodologías más utilizadas y que se pueden adaptar a proyectos de solo una persona.

Rational Unified Process (RUP):

Sus principales características son:

- Guiado por Casos de uso: Los cuales sirven para describir el comportamiento del sistema y elaborar los casos de prueba con los que se comprueba que el sistema desarrollado hace lo que el cliente quiere.

- Centrado en la arquitectura: La arquitectura del sistema es la columna vertebral de todo el desarrollo del mismo. Cada iteración gira en torno a la misma, fortaleciendo y corrigiendo sus características.
- Iterativo e incremental: En cada ciclo de iteración se produce una nueva versión del software.
- Utiliza UML como lenguaje de modelado y cuenta con varias fases de trabajo en las cuales se desarrolla una serie de flujos fundamentales del desarrollo del proyecto (Kruchten, 2002).

Programación Extrema (del inglés "*Extreme Programming*", en adelante XP) es la metodología más popular de las relativamente recientes metodologías ágiles. Sus principales características son:

- **Retroalimentación con el cliente:** Conceptualmente, al menos uno de los miembros del equipo de trabajo del proyecto, es un cliente. Esto propicia una constante interacción del mismo con el producto en desarrollo.
- **Cortas iteraciones:** En cada iteración, se obtiene un producto listo para entregar y que tiene valor para el cliente. La entrega continua de resultados compromete a ambas partes en la evolución del proyecto e indirectamente influye de forma positiva en la calidad del producto final.
- **Muy flexible a cambios:** Una de las características más reconocidas de XP. La constante retroalimentación con los clientes permite prever futuros cambios y evita llegar a momentos que paralizan el desarrollo del producto (Warden, 2007).

Cuenta con una serie de prácticas que van en vías de aumentar la productividad del equipo de trabajo, tales como la programación en pares, reuniones diarias y planes de entrega a corto plazo, por mencionar algunos. Al igual que RUP, también utiliza UML si el equipo de desarrollo lo decide.

En ambas metodologías se puede hacer un recorte amplio de roles y artefactos para adaptar el proyecto a equipos de trabajos compuesto por solo 1 persona. RUP es conocido por la robustez de su proceso de desarrollo a largo plazo y XP por la rapidez a corto plazo de las entregas. Es debido a ello y a las cuestiones que aparecen a continuación, que XP es la metodología seleccionada para el desarrollo de la presente solución:

- El período de desarrollo es corto: El desarrollo de la solución se limita a solamente 4 meses de trabajo continuo.
- El cliente forma parte del equipo de desarrollo.
- Las dimensiones del proyecto son pequeñas.

- Uno de los objetivos específicos es publicar versiones de pruebas para obtener retroalimentación de diferentes probadores de la universidad y así optimizar el producto. Para ello es necesario tener un período de entregas corto, respaldado por iteraciones cortas y un proceso de desarrollo bastante ágil (Gutiérrez, 2009).

Conclusiones parciales

El tiempo y la calidad son variables notablemente importantes en el desarrollo de software, el desarrollo de un *middleware* para la comunicación de los diferentes módulos son aspectos que permiten la disminución y el aumento de estas variables respectivamente, por lo cual se considera propicio el desarrollo de un sistema que permita la comunicación de los diferentes módulos del Sistema Video Vigilancia Xilema Suria.

El análisis de las herramientas existentes para la comunicación permitió identificar cual era la más viable para la propuesta de solución, seleccionando a ZMQ como biblioteca de comunicación, el mismo posee ya implementado los patrones de publicación (Pub) y subscripción (Sub) que permitieron el desarrollo del servicio de notificador de eventos y el de solicitador (REQ) y respuesta (REP), que contribuirán a la comunicación entre los módulos del sistema Video Vigilancia Xilema Suria.

2. Capítulo 2: Exploración y planificación

2.1. Introducción

En el siguiente capítulo se presenta la propuesta del *middleware* de comunicación**.** Se hacen especificaciones sobre sus características, además sobre sus requisitos funcionales y no funcionales. Se definen las historias de usuarios y se realiza un diagrama de clases del diseño que se utiliza como complemento de la metodología definida. Para el negocio se establece un modelo de dominio a través del cual se puede entender mejor el problema. Así mismo se exponen las tareas de ingeniería y el plan de iteraciones en las cuales serán implementadas cada historia de usuario y se realiza una estimación de la duración de las tareas de la ingeniería garantizando una correcta implementación.

2.2. Propuesta del sistema

El sistema que se necesita desarrollar debe garantizar la comunicación entre los diferentes módulos de Video Vigilancia Suria, brindando servicios de publicación y subscripción de eventos así como la invocación síncrona y asíncrona obteniendo respuesta de estos métodos en el servidor. El sistema debe presentar una estructura lo más escalable, flexible y genérica posible con el objetivo de garantizar futuras modificaciones. Debe permitir la configuración de la subscripción o la invocación al servidor deseado.

2.3. Fase de Exploración

En esta fase, los clientes plantean a grandes rasgos las historias de usuario que son de interés para la primera entrega del producto. Al mismo tiempo el equipo de desarrollo se familiariza con las herramientas, tecnologías y prácticas que se utilizarán en el proyecto. Se prueba la tecnología y se exploran las posibilidades de la arquitectura del sistema construyendo un prototipo. Debe quedar claro que las estimaciones realizadas en esta fase son primarias, puesto que estarán basadas en datos de muy alto nivel y podrían variar cuando se analicen más en detalle en cada iteración. Esta fase dura típicamente un par de semanas y el resultado es una visión general del sistema, y un plazo total estimado (Joskowicz, 2008).

2.4. Historias de usuario

Una historia de usuario es una representación de un requisito de software escrito en una o dos frases utilizando el lenguaje común del usuario. Dentro de la metodología XP las historias de usuario deben ser escritas por los clientes. Son una forma rápida de administrar los requisitos de los usuarios sin tener que elaborar gran cantidad de documentos formales y sin requerir de mucho tiempo para administrarlos. Las historias de usuario permiten responder rápidamente a los requisitos cambiantes.

Estas definen lo que se debe construir en el proyecto de software, tienen una prioridad asociada definida por el cliente de manera que se indiquen cuáles son las más importantes para el resultado final, serán divididas en tareas y su tiempo será estimado por los desarrolladores. Generalmente se espera que la estimación de tiempo de cada historia de usuario se sitúe entre unas 10 horas y un par de semanas. Estimaciones mayores a dos semanas indican que la historia es muy compleja y debe ser dividida en varias historias. Entre sus principales características resaltan, que deben ser independientes una de otra y debe ser entendida por el cliente (Cohn, 2004)

.

Tabla 1 Historia de Usuario Publicar evento remotamente.

Historia de Usuario: Publicar evento remotamente	
Número: 1	**Usuario:** Programador
Nombre historia: Publicar evento remotamente.	
Puntos estimados:1	**Puntos reales:**2
Prioridad en negocio: Alta	**Riesgo en desarrollo:** Alto
Programador responsable: Alberto Marturelo Lorenzo	
Descripción: El sistema debe permitir la publicación de eventos en el servidor para que los subscriptores que se han registrado al evento publicado se notifiquen de su aparición.	
Observaciones: Un cliente conectado publica la aparición de un evento en el servidor.	

Tabla 2 Historia de Usuario Suscribirse a un evento remotamente.

Historia de Usuario: Subscribir a un evento remotamente	
Número: 2	**Usuario:** Programador
Nombre historia: Subscribir a un evento remotamente.	
Puntos estimados:1	**Puntos reales:**2
Prioridad en negocio: Alta	**Riesgo en desarrollo:** Alto

Programador responsable: Alberto Marturelo Lorenzo
Descripción: El sistema debe permitir que un cliente se subscriba a un evento o eventos en específico.
Observaciones: El cliente conectado y autentificado puede subscribirse a uno o más eventos para que en un futuro cuando sean publicados le llegue la notificación de la aparición.

Tabla 3 Historia de Usuarios Eliminar subscripción del cliente a un evento remoto.

Historia de Usuario: Eliminar subscripción de un evento remoto	
Número: 3	**Usuario:** Programador
Nombre historia: Eliminar subscripción de un evento remoto.	
Puntos estimados: 0.5	**Puntos reales:** 1
Prioridad en negocio: Medio	**Riesgo en desarrollo:** Medio
Programador responsable: Alberto Marturelo Lorenzo	
Descripción: El sistema debe ser capaz de eliminar subscripción al cliente de un evento en específico.	
Observaciones: Para eliminar una subscripción debe estar registrada la misma con anterioridad.	

2.5. Lista de reserva del producto

La lista de reserva del producto refleja los requisitos funcionales que debe cumplir el *middleware* a desarrollar, ordenados según la prioridad en el negocio (Alta, Media). Se realiza una estimación por semanas de cada uno y se especifica el rol que lo estimó. La tabla contiene además los requisitos no funcionales que requiere el sistema, teniendo en cuenta que los mismos no poseen alta prioridad en el negocio.

Tabla 4 Lista de reserva del producto.

	Prioridad	Descripción	Estimación	Estimado por
1	Alta	Publicar evento remotamente.	1 semana	Programador
2	Alta	Subscribir a un evento remotamente.	1 semana	Programador
3	Alta	Invocación síncrona de operaciones en el objeto remoto.	1 semana	Programador
4	Alta	Invocación asíncrona de operaciones en el objeto remoto.	1 semana	Programador
5	Alta	Invocación asíncrona sin retorno de objetos remoto.	1 semana	Programador
6	Alta	Retornar la invocación asíncrona mediante *Result CALLBACK*.	1 semana	Programador
7	Alta	Retornar la invocación asíncrona mediante *Poll Object*.	1 semana	Programador
8	Alta	Retornar la invocación asíncrona mediante *Sync With Server*.	1 semana	Programador
9	Alta	Creación de objetos de forma estática en el servidor.	1 semana	Programador
10	Alta	Creación de objetos en el momento que el cliente solicite.	1 semana	Programador
11	Media	Eliminar subscripción de un evento remoto.	2.5 días	Programador

2.6. Fase de Planificación

Esta es una fase muy corta en la que el cliente establece la prioridad de cada historia de usuario con el propósito de que el programador conozca el orden en que serán implementadas las mismas. Una vez conocido el orden en que serán implementadas las HU (Historias de Usuario) el equipo de desarrollo define cuáles son las que estarán listas para la primera liberación. Por tanto la prioridad:

- **Alta:** se le otorga a las HU que resultan funcionalidades fundamentales en el desarrollo del sistema, a las que el cliente define como principales para el control integral del sistema.
- **Media:** se le otorga a las HU que resultan para el cliente como funcionalidades a tener en cuenta, sin que estas tengan una afectación sobre el sistema que se esté desarrollando.
- **Baja:** se le otorga a las HU que constituyen funcionalidades que sirven de ayuda al control de elementos asociados al equipo de desarrollo, a la estructura y no tienen nada que ver con el sistema en desarrollo.

2.6.1. Prioridad de las Historias de Usuario

La acción del cliente al establecer la prioridad para cada historia de usuario es la que le permite conocer al programador en qué orden deberá implementar las mismas. Aclarar que las historias de usuario de menor prioridad son las más importantes y por tanto deben desarrollarse de primeras.

Tabla 5 Prioridad de las Historias de Usuario.

Historia de usuario	Prioridad
Publicar evento remotamente.	1
Subscribir a un evento remotamente.	1
Invocación síncrona de operaciones en el objeto remoto.	1
Invocación asíncrona de operaciones en el objeto remoto.	1
Invocación asíncrona sin retorno de objetos remoto.	1
Retornar la invocación asíncrona mediante *Result CALLBACK*.	1
Retornar la invocación asíncrona mediante *Poll Object*.	1
Retornar la invocación asíncrona mediante *Sync With Server*.	1
Creación de objetos de forma estática en el servidor.	1
Creación de objetos en el momento que el cliente solicite.	1
Eliminar subscripción de un evento remoto.	2

El riesgo en su desarrollo

- **Alto:** cuando en la implementación de las HU se considera la posible existencia de errores que lleven a la inoperatividad del código.
- **Medio:** cuando pueden aparecer errores en la implementación de las HU que puedan retrasar la entrega de la versión.
- **Bajo:** cuando pueden aparecer errores que serán tratados con relativa facilidad sin que traigan prejuicios para el desarrollo del proyecto. El cliente y los desarrolladores trabajan en conjunto para definir cómo agrupar las HU para su lanzamiento.

2.6.2. Estimación de esfuerzo de las Historias de Usuario

Según la prioridad asignada a las historias de usuario es imprescindible estimar el esfuerzo necesario para el desarrollo de ellas. La estimación se basa en los conocimientos y velocidad que presente el equipo de desarrollo para llevar a cabo las tareas ingenieriles. Los puntos estimados son expresados en días ideales y se debe tener presente que los puntos estimados por historias de usuario no siempre se cumplen de acuerdo a como se planifica, puede que existan casos donde varíe.

Tabla 6 Estimación de esfuerzo de Historias de Usuarios.

Historia de Usuario	Esfuerzo necesario (Puntos estimados)
Publicar evento remotamente.	7
Subscribir a un evento remotamente.	7
Invocación síncrona de operaciones en el objeto remoto.	7
Invocación asíncrona de operaciones en el objeto remoto.	7
Invocación asíncrona sin retorno de objetos remoto.	7
Retornar la invocación asíncrona mediante *Result CALLBACK*.	7
Retornar la invocación asíncrona mediante *Poll Object*.	7
Retornar la invocación asíncrona mediante *Sync With Server*.	7
Creación de objetos de forma estática en el servidor.	7
Creación de objetos en el momento que el cliente solicite.	7
Eliminar subscripción de un evento remoto.	3.5

La metodología XP plantea que se debe llevar a cabo una liberación cada vez que culmine una iteración; proponiendo una vez lanzando un producto completamente funcional. La planificación de la liberación es realizada entre el cliente y el equipo de desarrolladores, el primero decide qué historias de usuario tienen la mayor prioridad y el segundo estima el tiempo que le llevará implementar las mismas. A continuación se presenta el cronograma de liberación de la investigación en curso:

Tabla 7 Cronograma de iteración.

Iteración	Fecha de liberación
Primera	1ra semana de marzo 2014
Segunda	2da semana de marzo 2014
Tercera	3ra semana de marzo 2014
Cuarta	1ra semana de abril 2014
Quinta	2da semana de abril
Sexta	3ra semana de abril

2.7. Fase de Iteración

Una vez identificadas y descritas por el cliente las historias de usuario y con ello la estimación del esfuerzo de cada una de ellas por el equipo de desarrollo se procede a realizar la planificación de las etapas de implementación del sistema. Por tanto se establece un plan de iteraciones donde se especifican las historias de usuario y el orden en que serán implementadas en cada iteración de acuerdo a su duración. La cantidad de iteraciones (IN) a realizar para el desarrollo de las historias de usuario está determinada por la suma de los puntos de esfuerzo (PE) para cada una de ellas dividida por la velocidad de iteración del equipo (VIE).

IN=PE / VIE

PE= 73.5 (14.7 semanas)

IN=14.7 /2.5

IN =5.88

La velocidad de iteración del equipo (VIE) se obtiene dividiendo la cantidad de desarrolladores (CD) entre el factor de dedicación (FD) al proyecto (en el caso de la presente investigación es de 4 [100%]) y multiplicado por el tiempo de duración máximo de una iteración (DMI) (en el caso de la presente investigación es de 10 días máximo).

VIE= (CD / FD)*DMI

CD=1, FD =100% (4), DMI =10

VIE= 0.25*10

VIE=2.5

Teniendo en cuenta los cálculos realizados anteriormente se obtiene por valor de la velocidad de equipo aproximadamente 4 y la cantidad de iteraciones necesarias para desarrollar las historias de usuario 2. El plan de iteraciones queda conformado de la siguiente forma:

Tabla 8 Primera iteración.

Iteración		
Número: 1	**H.U (por orden): 1 y 2**	**Duración total: 10**
Descripción: esta iteración tiene como objetivo la implementación de las HU con prioridad alta. Esta iteración es la encargada de la publicación y subscripción de los eventos permitiendo que cada uno de los subscriptores registrados a los eventos especificados le llegue la publicación del mismo en el momento de su ocurrencia. Al final de esta iteración se contará con una primera versión de prueba, la cual será mostrada al cliente con el objetivo de obtener una retroalimentación para el grupo de trabajo.		

Tabla 9 Segunda iteración.

Iteración		
Número: 2	**H.U (por orden): 3 y 5**	**Duración total: 10**
Descripción: esta iteración tiene como objetivo la implementación de las HU con prioridad alta. Esta iteración es la encargada de realizar invocaciones de forma síncrona y asíncrona sin retorno. Al final de esta iteración se contará con una versión de prueba, la cual será mostrada al cliente con el objetivo de obtener una retroalimentación para el grupo de trabajo.		

Tabla 10 Tercera iteración.

Iteración		
Número: 3	**H.U (por orden): 4 y 6**	**Duración total: 10**
Descripción: esta iteración tiene como objetivo la implementación de las HU con prioridad alta. Esta iteración es la encargada de realizar invocaciones asíncronamente y de retornar el resultado de la invocación mediante *Result CALLBACK*. Al final de esta iteración se contará con una versión de prueba, la cual será mostrada al cliente con el objetivo de obtener una retroalimentación para el grupo de trabajo.		

Tabla 11 Cuarta iteración.

Iteración		
Número: 4	**H.U (por orden): 7 y 8**	**Duración total: 10**
Descripción: esta iteración tiene como objetivo la implementación de las HU con prioridad alta. Esta iteración es la encargada de retornar el resultado de las invocaciones mediante *Poll Object* y mediante *Sync With Server*. Al final de esta iteración se contará con una primera versión de prueba, la cual será mostrada al cliente con el objetivo de obtener una retroalimentación para el grupo de trabajo.		

Tabla 12 Quinta iteración.

Iteración		
Número: 5	**H.U (por orden): 9 y 10**	**Duración total: 10**
Descripción: esta iteración tiene como objetivo la implementación de las HU con prioridad alta. Esta iteración es la encargada de la implementación ya sea de forma estática o dinámica de los diferentes objetos que pueden ser invocados en el servidor. Al final de esta iteración se contará con una versión de prueba, la cual será mostrada al cliente con el objetivo de obtener una retroalimentación para el grupo de trabajo.		

Tabla 13 Sexta iteración.

Iteración		
Número: 6	**H.U (por orden): 11**	**Duración total: 2.5**
Descripción: esta iteración tiene como objetivo la implementación de las HU con prioridad media. Esta iteración es la encargada de eliminar la subscripción de eventos remotos.		

2.8. Diseño del Sistema

XP enfoca sus esfuerzos para conseguir diseños simples y sencillos. Es necesario procurarlo todo lo menos complicado posible para conseguir un diseño fácilmente entendible y que sea posible implementarlo. Esto garantizará menos tiempo y esfuerzo durante el proceso de desarrollo.

2.8.1. Propuesta de Arquitectura del Sistema

Arquitectura de software se entenderá como la organización fundamental de un sistema encarnada en sus componentes, las relaciones de los componentes con cada uno de los otros y con el entorno, y los principios que orientan su diseño y evolución (Warden, 2007).

Arquitecturas en Capas

Los sistemas o arquitecturas en capas constituyen uno de los estilos que aparecen con mayor frecuencia mencionados como categorías mayores del catálogo, o, por el contrario, como una de las posibles encarnaciones de algún estilo más envolvente. En [GS94] Garlan y Shaw definen el estilo en capas como una organización jerárquica tal que cada capa proporciona servicios a la capa inmediatamente superior y se sirve de las prestaciones que le brinda la inmediatamente inferior (Reynoso, et al., 2004).

Arquitectura cliente servidor

La arquitectura cliente-servidor es un modelo de aplicación distribuida en el que las tareas se reparten entre los proveedores de recursos o servicios, llamados servidores, y los demandantes, llamados clientes. Un cliente realiza peticiones a otro programa, el servidor, quien le da respuesta una representación clara se muestra en la ilustración siguiente.

Esta idea también se puede aplicar a programas que se ejecutan sobre una sola computadora, aunque es más ventajosa en un sistema operativo multiusuario distribuido a través de una red de computadoras.

En esta arquitectura la capacidad de proceso está repartida entre los clientes y los servidores, aunque son más importantes las ventajas de tipo organizativo debidas a la centralización de la gestión de la información y la separación de responsabilidades, lo que facilita y clarifica el diseño del sistema.

Una disposición muy común son los sistemas multicapa en los que el servidor se descompone en diferentes programas que pueden ser ejecutados por diferentes computadoras aumentando así el grado de distribución del sistema.

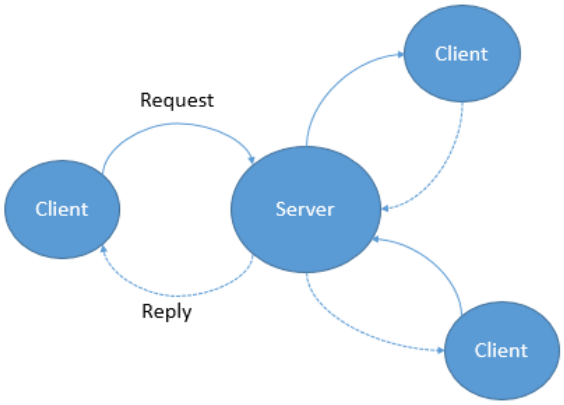

Ilustración 8 Arquitectura cliente servidor.

Estilo arquitectónico *Broker*

El *Broker* es un patrón de arquitectura que se utiliza para estructurar sistemas de software distribuidos con componentes desacoplados que interactúan por invocaciones de servicios remotos. Utilizando el patrón *Broker*, una aplicación puede acceder a los servicios distribuidos enviando mensajes al objeto apropiado, en vez de enfocarse en la comunicación entre procesos de bajo nivel. Además, el patrón *Broker* es flexible porque permite cambiar, añadir, quitar y reubicar objetos dinámicamente.

El patrón *Broker* reduce la complejidad en el desarrollo de aplicaciones no centralizadas porque hace que la distribución sea transparente al desarrollador, mediante la introducción de un modelo de objetos en el que los servicios distribuidos se encapsulan en objetos. Por lo tanto, los sistemas *Broker* ofrecen una ruta para la integración de dos tecnologías: distribución y objetos. Así mismo, los sistemas *Broker* extienden los modelos de objetos desde aplicaciones individuales hasta aplicaciones distribuidas que constan de componentes desacoplados que pueden ejecutarse en máquinas heterogéneas, y que pueden escribirse en lenguajes de programación diferentes (Chapa, 1996).

El patrón *Broker* comprende seis componentes participantes: clientes, servidores, *Brokers*, puentes, *Proxy* del lado del cliente y *Proxy* del lado del servidor, como se muestra en la siguiente ilustración:

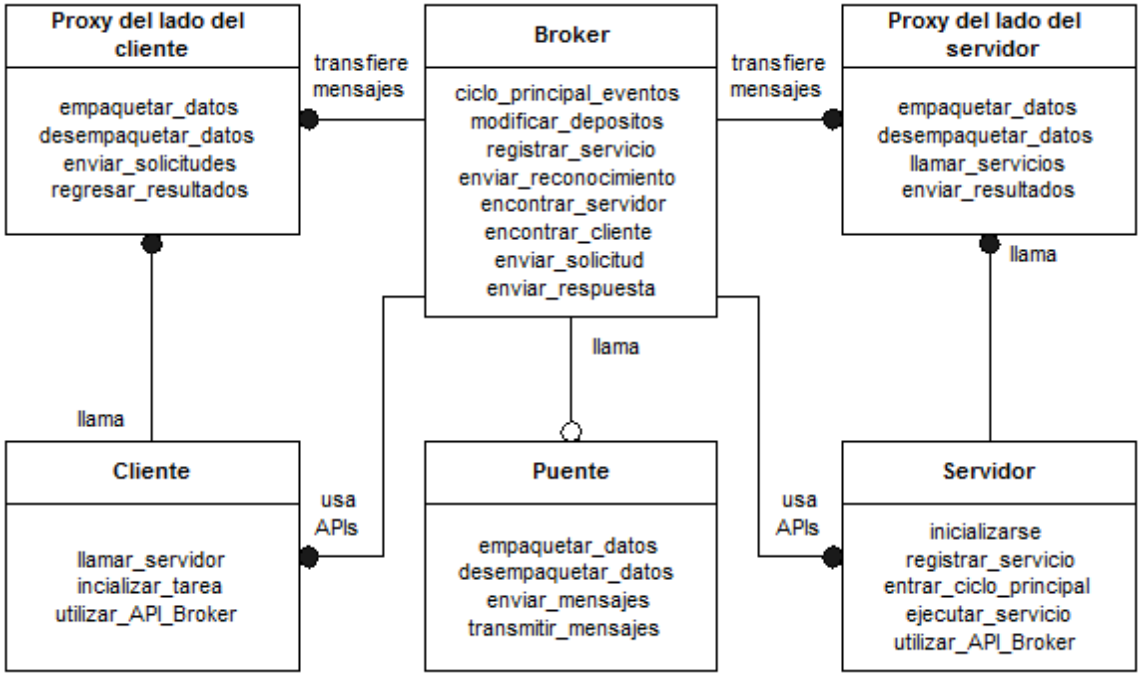

Ilustración 9 Estructura del patrón *Broker*.

Servidor

Un servidor implementa objetos que exponen su funcionalidad a través de interfaces que consisten de operaciones y atributos. Las interfaces están disponibles a través de un lenguaje de definición de interfaz (IDL) o un estándar binario este último es el utilizado en el desarrollo de la aplicación para la utilización de interfaces comunes.

Cliente

Los clientes son aplicaciones que accedan a los servicios de, al menos, un servidor. Para invocar servicios remotos, los clientes envían solicitudes al *Broker*.

Broker

Un *Broker* es un mensajero, responsable de la transmisión de solicitudes de clientes a servidores, así como de la transmisión de respuestas y excepciones de servidores a clientes. Localiza al receptor de una solicitud basándose en un sistema de identificadores únicos. Ofrece API's a clientes y servidores que incluyen operaciones para el registro de servidores, y la invocación de métodos de servidores.

***Proxy* del lado del cliente**

Los *Proxy* del lado del cliente representan una capa adicional entre los clientes y el *Broker*, para proveer transparencia en el sentido que un objeto remoto aparece como local ante el cliente, es decir esconden los detalles de implementación tales como:

- El mecanismo de comunicación entre procesos utilizado para transferir un mensaje entre clientes y *Brokers*.
- La creación y eliminación de bloques de memoria.
- El *marshaling* de parámetros y resultados.

***Proxy* de lado del servidor**

Los *Proxy* del lado del servidor generalmente son análogos a los *Proxy* del lado del cliente, la diferencia es que son responsables de recibir solicitudes, desempaquetar los mensajes de entrada, el *unmarshaling* de los parámetros, llamar al servicio apropiado, y el *marshaling* de resultados y excepciones antes de enviarlos al cliente.

Puentes

Los puentes son componentes opcionales utilizados para esconder los detalles de implementación cuando dos *Brokers* interoperan. Supóngase que un sistema *Broker* se ejecuta en una red heterogénea. Si se

transmiten solicitudes sobre la red, se deben comunicar *Brokers* diferentes independientemente de las redes y de los sistemas operativos utilizados.

Mediante la bibliografía consultada se logra la unión de todas las arquitecturas mediante la implementación de los patrones que se evidencian en la siguiente ilustración.

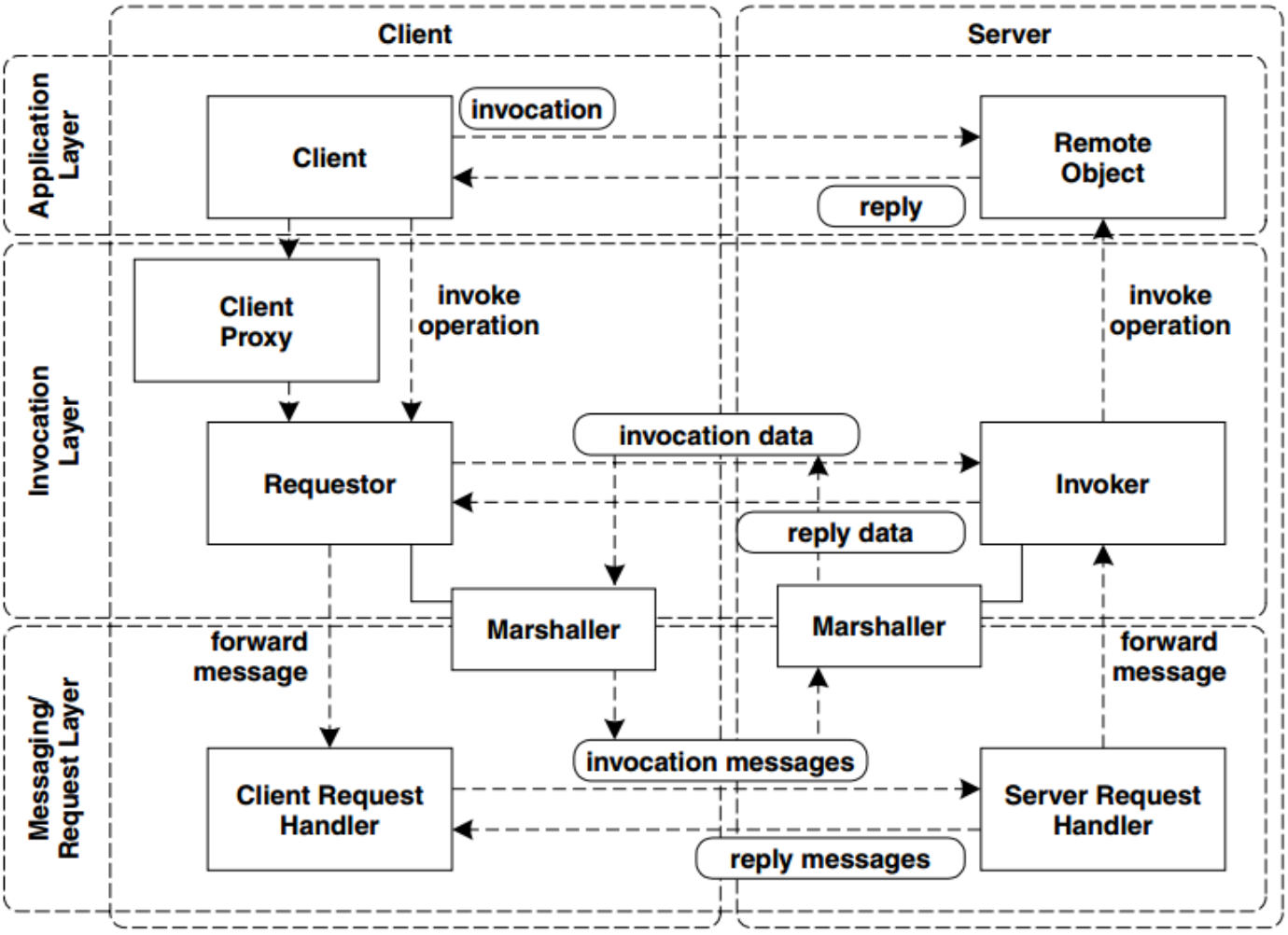

Ilustración 10 Integración de las arquitecturas a utilizar. (2 p. 66)

En Ilustración se muestra el flujo de datos de una invocación donde se puede apreciar cómo interactúan los diferentes componentes, la responsabilidad de cada capa ya sea del lado del cliente cómo de lado del servidor y el nivel de transparencia que aporta cada una.

Patrones arquitectónicos del *Middleware*

Patrón Requestor

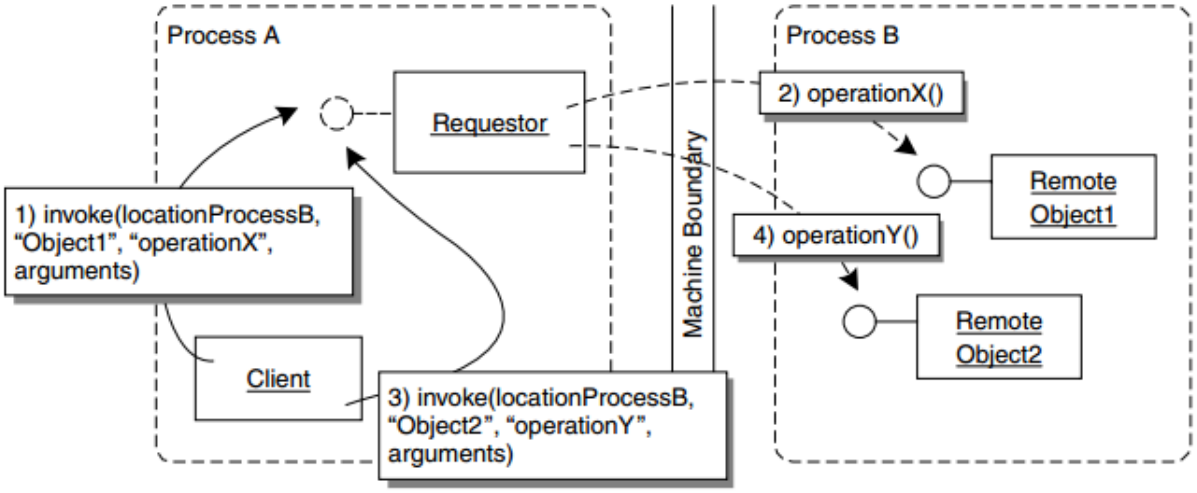

Ilustración 11 Patrón Requestor.

La invocación de objetos remotos requiere que la operación de los parámetros sea recogida y calculadas las referencias del flujo de bytes, ya que las redes solo permiten flujos de bytes a enviar. Una conexión también necesita ser establecida y la solicitud de información ser enviada al objeto remoto de destino. Dichas tareas tienen que ser realizadas para cada objeto remoto al cual tiene acceso al cliente, y por tanto; puede llegar convertirse en algo tedioso para los desarrolladores del cliente (Puder, et al., 2006).

Patrón Client Proxy

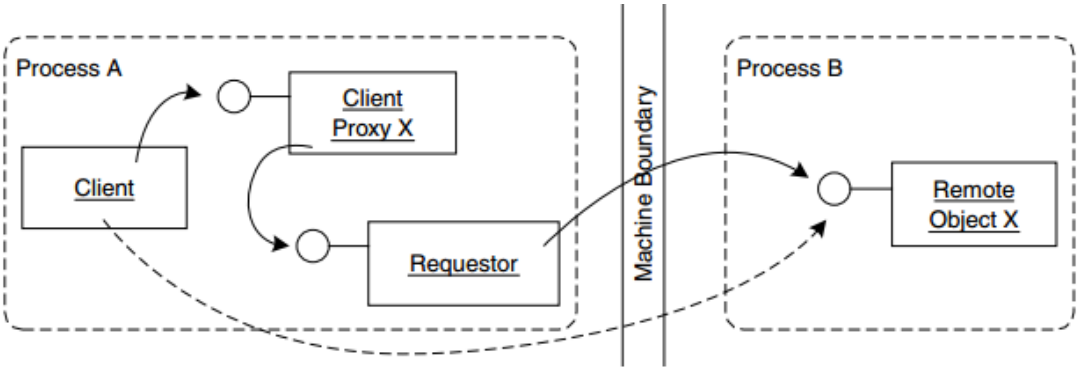

Ilustración 12 Patrón Client Proxy.

Uno de los objetivos principales de la utilización de objetos remotos es apoyar un modelo de programación que permita acceder a objetos en aplicaciones distribuidas similar al acceso a objetos locales. El patrón *Requestor* resuelve parte de este problema ocultando muchos detalles de la red. Sin embargo, utilizando el *Requestor* es engorroso, ya que los métodos que se invocan en el objeto remoto, sus parámetros, así como la ubicación y la información de identificación del objeto remoto, tendrán que ser aprobada por el cliente en un formato definido por el solicitante para cada invocación. *El Requestor* tampoco proporciona comprobación de tipos en tiempo de compilación o estática, que además complica el desarrollo del cliente (Puder, et al., 2006).

Patrón Invoker

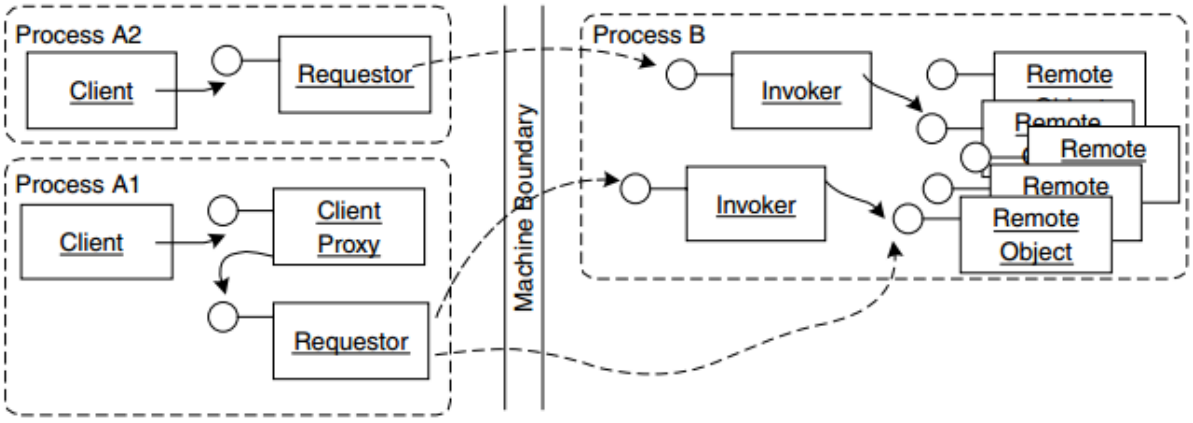

Ilustración 13 Patrón Invoker.

Cuando un cliente envía los datos de invocación a través del cliente al servidor, el objeto remoto destino tiene que ser alcanzado de alguna manera. La solución más sencilla es dejar que cada objeto remoto sea abordado a través de la red directamente. Pero esta solución no funciona para un gran número de objetos remotos, ya que no se pueden hacen los suficientes puntos finales de red para todos los objetos remotos. Además, el objeto remoto tendría que lidiar con el manejo de las conexiones de red, la recepción y el des serializado de los mensajes, y así sucesivamente. Esto es engorroso y excesivamente complejo (Puder, et al., 2006).

Patrón Client Request Handler

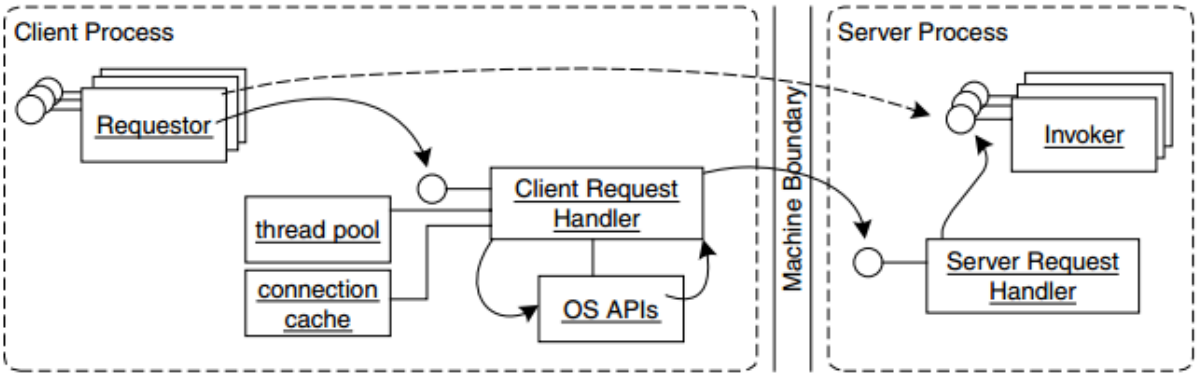

Ilustración 14 Patrón Client Request Handler.

Para enviar las solicitudes del cliente para la aplicación servidor, varias tareas tienen que ser realizadas: establecimiento de la conexión y la configuración, manejo de resultados, manejo de tiempo de espera, y detección de errores. En el caso de los tiempos de espera o los errores del *Requestor* y, posteriormente, el *Client Proxy,* tiene que estar informado. Gestionando la conexión del lado del cliente, los hilos, y del coordinado y optimizado de las operaciones (Puder, et al., 2006).

Patrón Server Request Handler

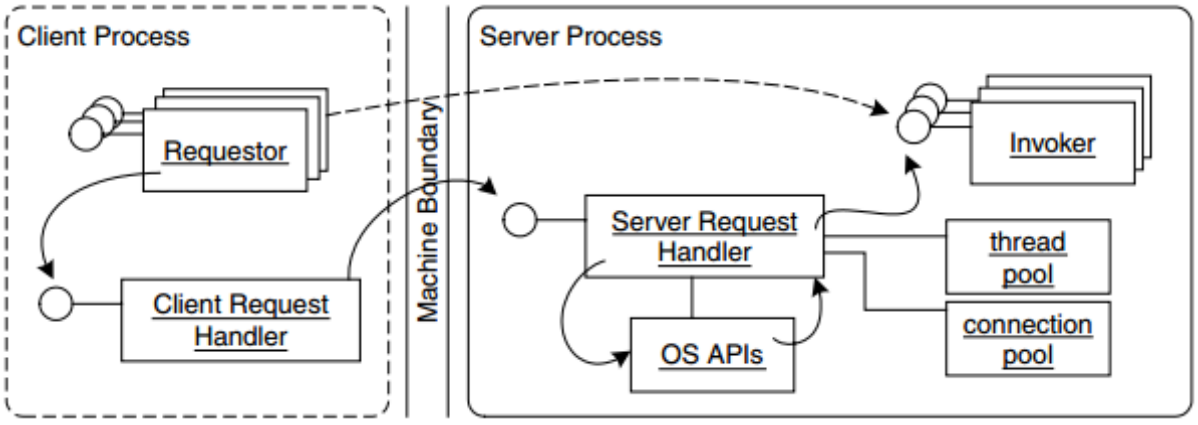

Ilustración 15 Patrón Server Request Handler.

Antes de que una solicitud sea enviada a un *Invoker,* la aplicación de servidor tiene que recibir el mensaje de solicitud de la red.

La gestión de los canales de comunicación de manera eficiente y eficaz es, ya que por lo general muchas peticiones esenciales pueden ser manipuladas, posiblemente, incluso al mismo tiempo. La comunicación de red tiene que ser gestionada de forma coordinada y optimizada (Puder, et al., 2006).

Patrón Marshaller

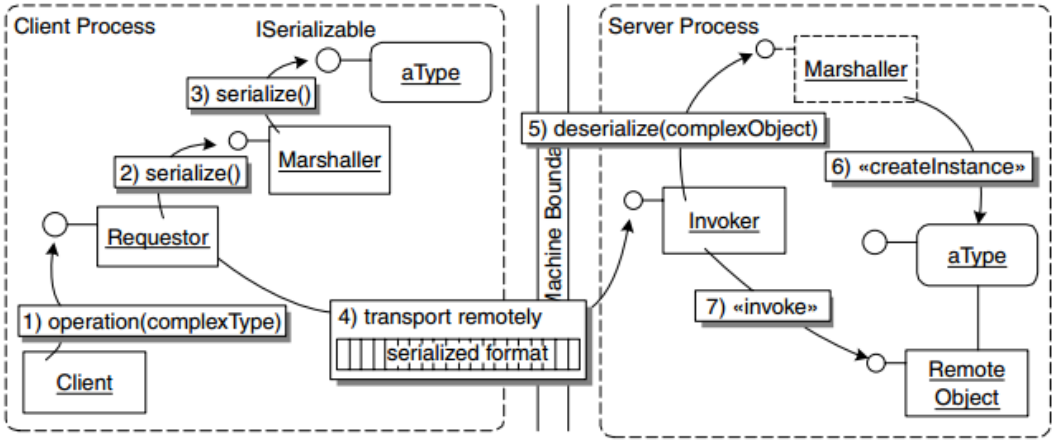

Ilustración 16 Patrón Marshaller.

Para ejecutar una invocación de forma remota, ya sea en diferentes ordenadores o en el mismo tiene que ser encapsulada de modo que se pueda enviar del cliente al servidor y viceversa. Los datos necesarios para describir las invocaciones se compone de un ID del objeto remoto destino, el nombre de la operación, los parámetros, el valor de retorno. Solo en flujo de bytes son los adecuados para ser transportados por la red (Puder, et al., 2006).

Patrón de identificación de objeto Object ID

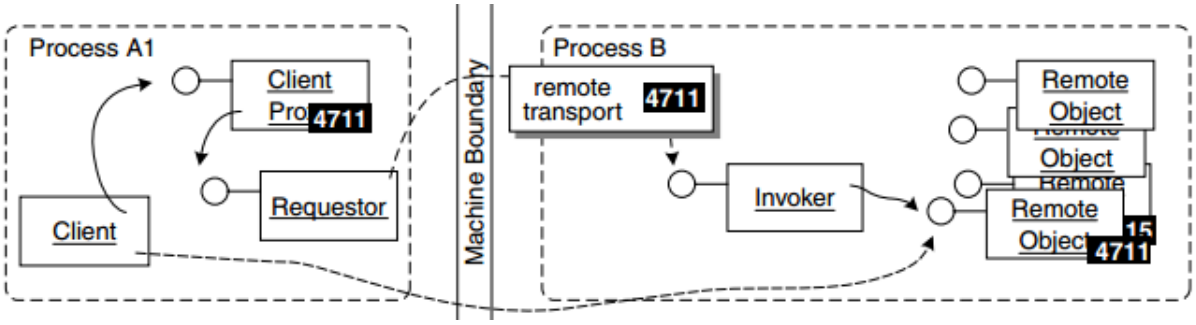

Ilustración 17 Patrón de identificación de objetos *Object ID*.

El *Invoker* es responsable de despachar las invocaciones recibidas desde el *Server Request Handler.* Se invoca la operación de un objeto remoto en nombre de un cliente. Sin embargo, el *Invoker* posee varios objetos remotos, y tiene que determinar que objeto corresponde a una ejecución en particular (Puder, et al., 2006).

Patrón de invocación asíncrona: Fire And Forget

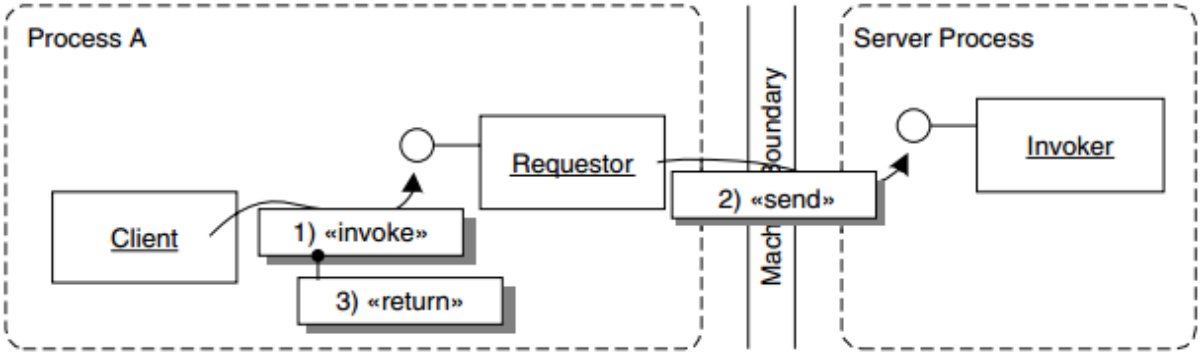

Ilustración 18 Patrón de invocación asíncrona: ***Fire And Forget***.

Una aplicación cliente necesita invocar una operación en un objeto remoto simplemente para notificar al objeto remoto de un evento, el cliente no necesita la espera de un valor de retorno. La fidelidad de la invocación no es crítica, ya que la invocación se utiliza simplemente como una notificación (Puder, et al., 2006).

Patrón de invocación asíncrona: SyncWithServer

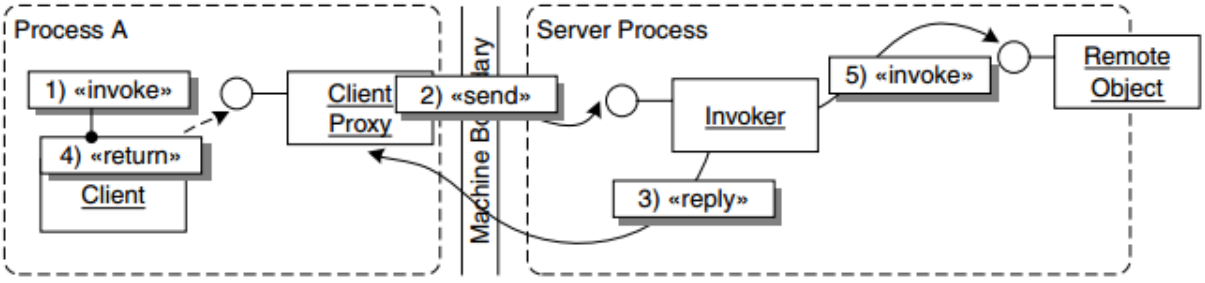

Ilustración 19 Patrón de invocación asíncrona: ***SyncWithServer***.

Dispara y olvida es una solución útil, pero extrema, en el sentido de que solo se puede utilizar si el cliente realmente puede correr el riesgo de no tener en cuenta de que una invocación remota no ha alcanzado el objeto.

El otro extremo es una invocación síncrona, en el que un cliente se bloquea hasta que el método remoto ha ejecutado con éxito y el resultado devuelto. A veces un equilibrio entre estos extremos se necesita (Puder, et al., 2006).

Patrón de invocación asíncrona: Poll Object

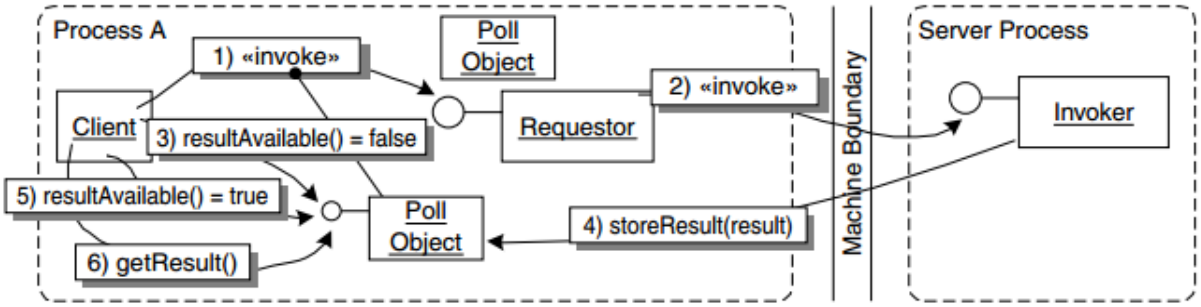

Ilustración 20 **Patrón de invocación asíncrona: Poll Object.**

Hay situaciones en las que una aplicación necesita invocar una operación asíncrona, pero todavía tiene que saber los resultados de la invocación. El cliente no necesita los resultados de inmediato para continuar su ejecución, y puede decidir por sí mismo cuando utilizar los resultados devueltos (Puder, et al., 2006 pp. 170, 171, 172).

Patrón de invocación asíncrona: Result CallBack

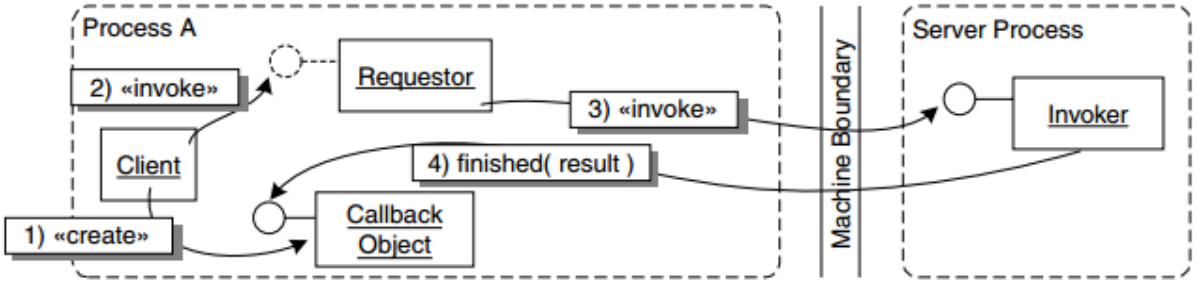

Ilustración 21 **Patrón de invocación asíncrona: Result CallBack**

El cliente debe ser informado de forma activa sobre los resultados de las operaciones invocadas de forma asíncrona sobre un objeto remoto. Es decir, si el resultado ya está disponible, el cliente es informado inmediatamente, de modo que pueda reaccionar sobre la disponibilidad del resultado. Mientras tanto, el cliente se ejecuta concurrentemente (Puder, et al., 2006).

***Patrón de administración de ciclo de vida:* Static Instance**

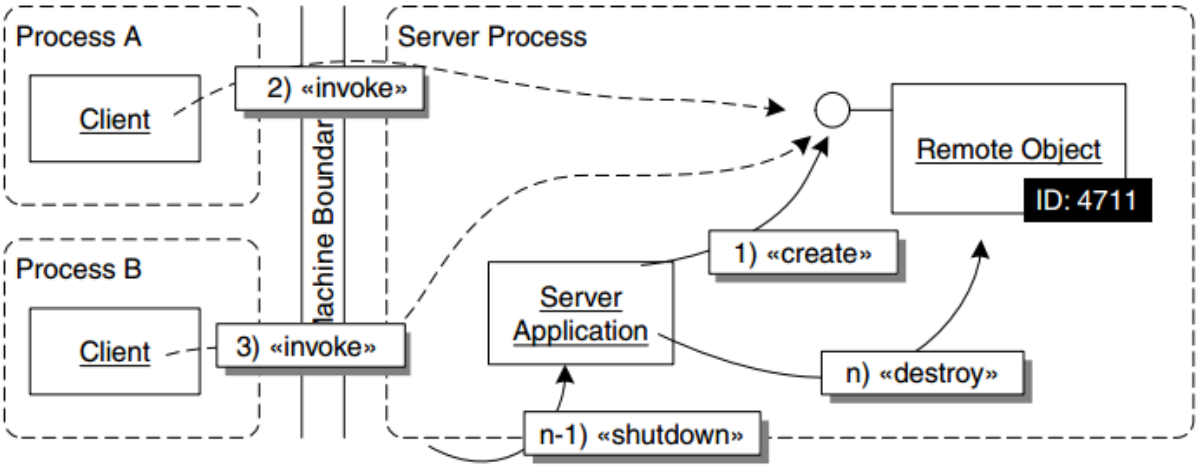

Ilustración 22 **Patrón de administración de ciclo de vida: Static Instance.**

La aplicación servidor tiene que proporcionar una serie objetos remotos previamente conocida. Los objetos remotos deben estar disponibles durante un largo período sin ningún tiempo de espera ni de expiración predeterminado. El objeto debe estar disponible para todos los clientes (Puder, et al., 2006).

Patrón de administración de ciclo de vida: Per-Request Instance

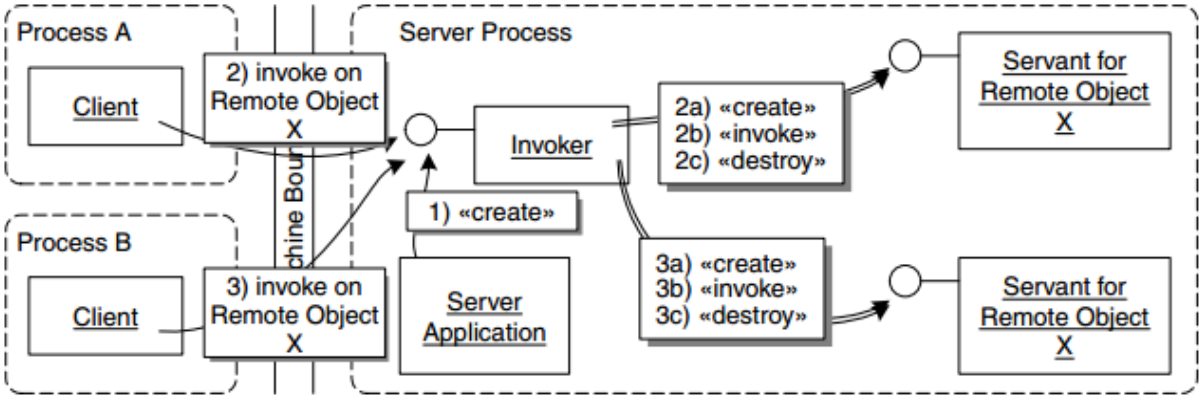

Ilustración 23 **Patrón de administración de ciclo de vida: Per-Request Instance**

Una aplicación de servidor contiene varios objetos remotos a los cuales acceden un gran número de clientes, y el acceso a los objetos remotos tiene que ser altamente escalable con respecto al número de clientes. Cuando muchos clientes acceden al mismo tiempo, el rendimiento de la aplicación de servidor se reduce drásticamente debido a la sincronización (Puder, et al., 2006).

Patrón Event Notifier

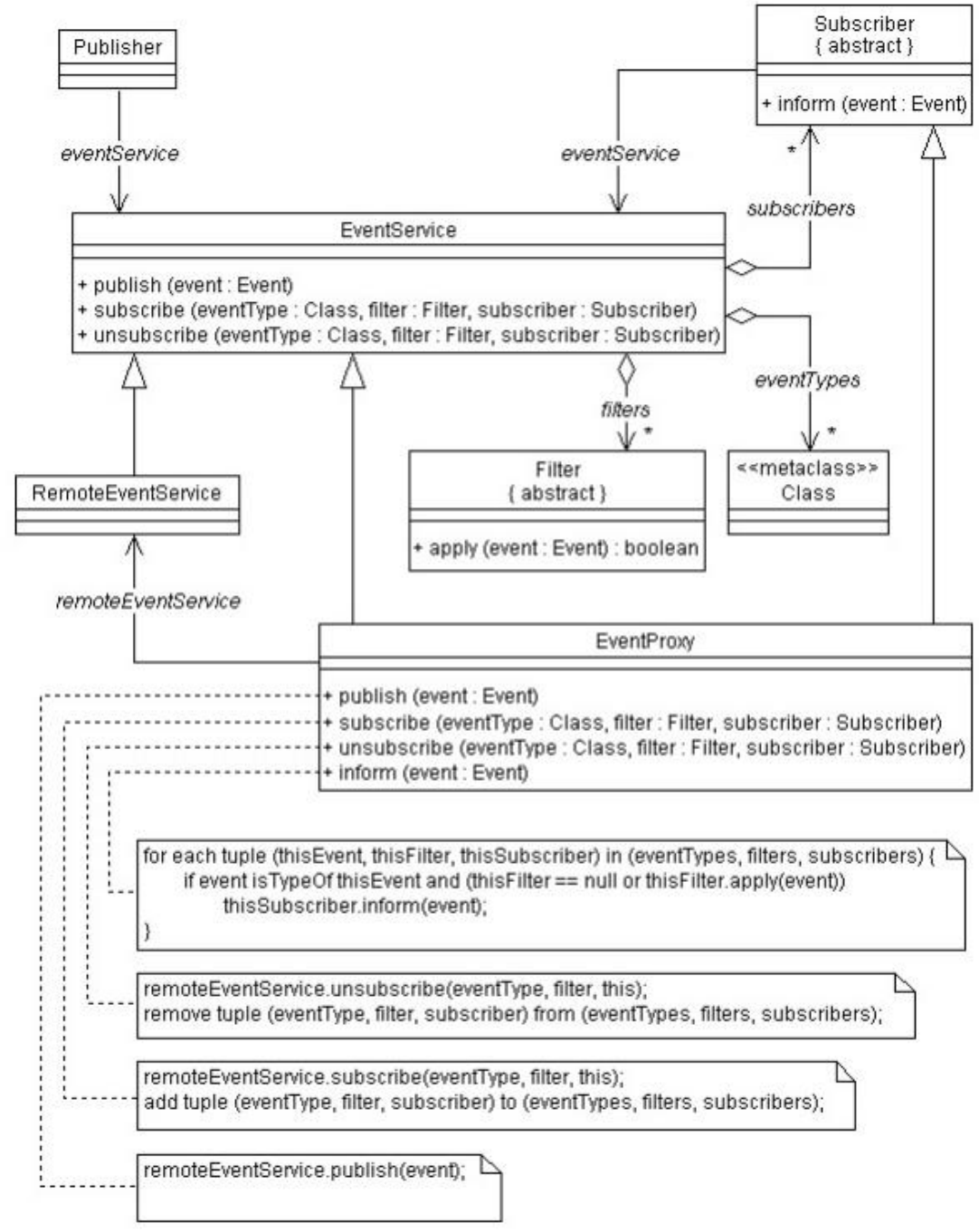

Ilustración 24 Diagrama de clases: Event Notifier (Event Notifier, a Pattern for Event Notification, 1998)

El patrón Notificador de Eventos o por sus siglas en inglés Event Notifier, está compuesto por dos patrones, el observador y el mediador. El diagrama anterior está compuesto por varias partes, un *Event Service* o Servicio de Eventos, que actúa como mediador entre un proceso que publica y otro como observador que se subscribe a sus publicaciones. Para el uso del servicio de forma remota se adiciona un nuevo objeto, el Event Proxy o Proxy de eventos, el cual es el encargado mediante el uso del patrón ya implementado de ZMQ de realizar las publicaciones a todas las aplicaciones subscritas ya sea en el mismo ordenador o en equipos diferentes interconectados por una red de datos.

Para realizar una subscripción se necesita un *Event* (Evento), un *Filter* (Filtro) y un *Subscriber* (Subscriptor). Cuando un evento se publica este antes de ser informado al subscriptor relacionado necesita pasar por un filtro el cual es una clase abstracta la misma tiene un método *apply,* el cual devuelve verdadero o falso si el evento es o no. En caso de ser verdadero se procederá a informar con el uso del método *inform* que contiene la clase abstracta *Subscriber.* El diagrama de clases resultante del uso del patrón propuesto por ZMQ y Event Notifier se muestra en la siguiente ilustración.

2.9. Diagrama de clases

El diagrama de clases es una herramienta esencial durante el proceso de análisis y diseño del sistema. Representa de una manera estática la estructura de información del sistema y la visibilidad que tiene cada una de las clases así como sus relaciones con los demás en el modelo.

A la hora de programar es necesario realizar un diagrama con todas o algunas de las clases que el programador tiene en mente para empezar la implementación. Esto hace que se dificulte menos el trabajo y se realice de manera organizada.

2.9.1. *Diagrama de clases: Servicio de Publicación* – Subscripción

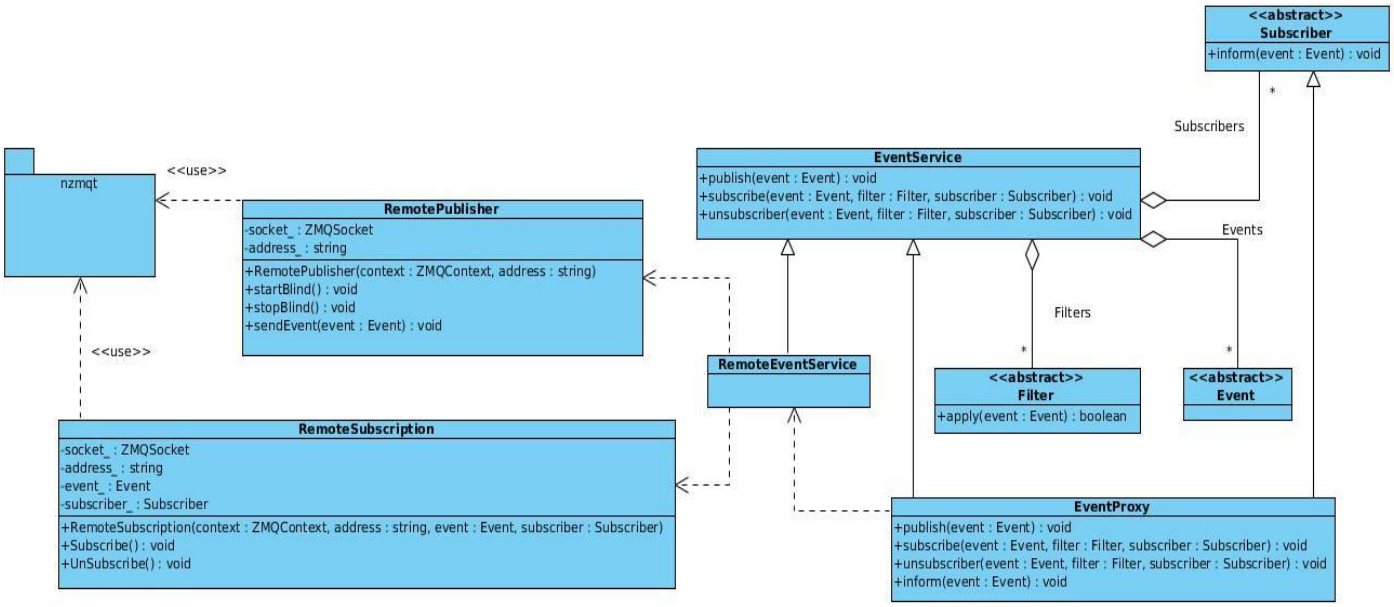

Ilustración 25 Diagrama de clases: Servicio de Publicación - Subscripción.

2.9.2. *Diagrama de clases: Servicio de XPub–XSub*

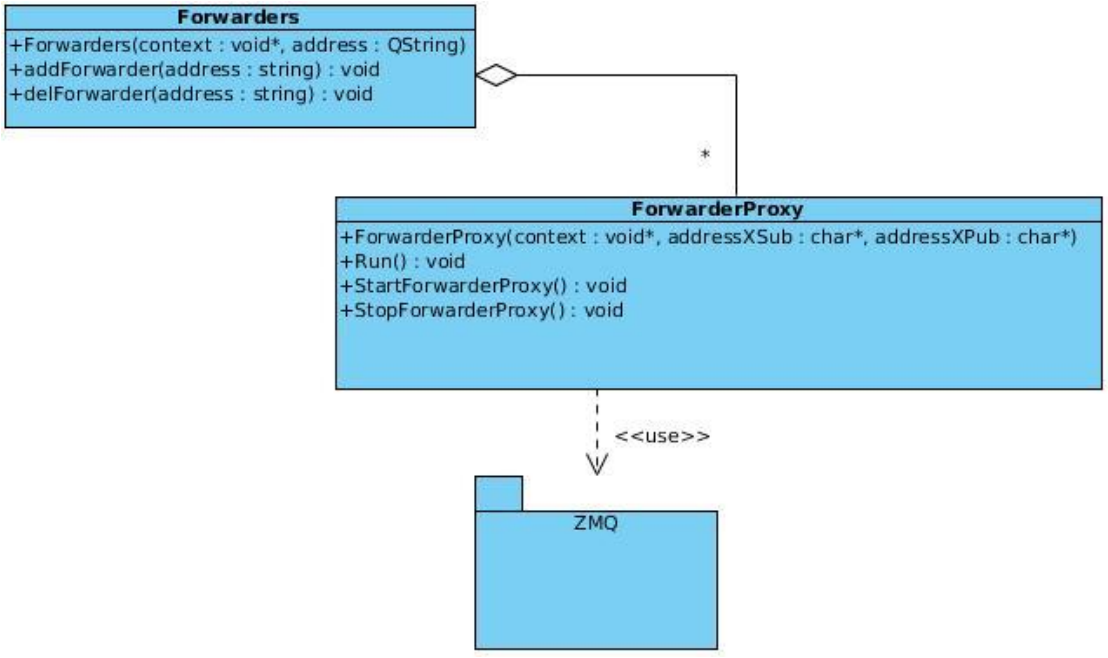

Ilustración 26 Diagrama de clases: Servicio XPub – XSub.

El uso de los patrones publicación y subscripción ya implementados por ZMQ como Pub y Sub, trae consigo un problema, ya que cada cliente tiene que conocer de dónde van a ser publicados los eventos para subscribirse a los mismos. Lo antes descrito no es objetivo, atenta contra la flexibilidad ya que si se quiere agregar un nuevo cliente al sistema todos los ya existentes tienen que conocer la dirección IP del mismo. Para hacer frente a dicha problemática ZMQ implementa la alternativa de publicación y subscripción XPub y XSub, que no es más que un proxy que actúa de tubería, el cliente que ingresaría nuevo al sistema solo deberá conocer la dirección IP del servidor proxy de XPub y XSub para suscribirse al evento o a los eventos que desee y enviar una petición al servidor para que este se subscriba a sus publicaciones.

2.9.3. Diagrama de clases: Servicio de invocación remota lado Cliente

El siguiente diagrama muestra la relación entre clases del lado del cliente del servicio de invocación remota, el cual está compuesto por varias partes importantes, la clase *ClientProxy* la cual implementa los métodos definidos por la interfaz común entre el cliente y el servidor, y que contiene las instancia de los tipos de solicitudes ya sea asíncrona *AsyncRequestor* o síncrona *SyncRequestor.* La clase OutputManager juega un papel muy importante ya que es la encargada de actuar como bandeja de entrada y salida, administrando los mensajes que no han sido atendidos en el tipo definido tras la inicialización del *ClientRequestHandler.* El *ClientRequestHandler*es el encargado de enviar o recibir el mensaje al servidor utilizando el patrón *Request* (REQ) que implementa ZMQ.

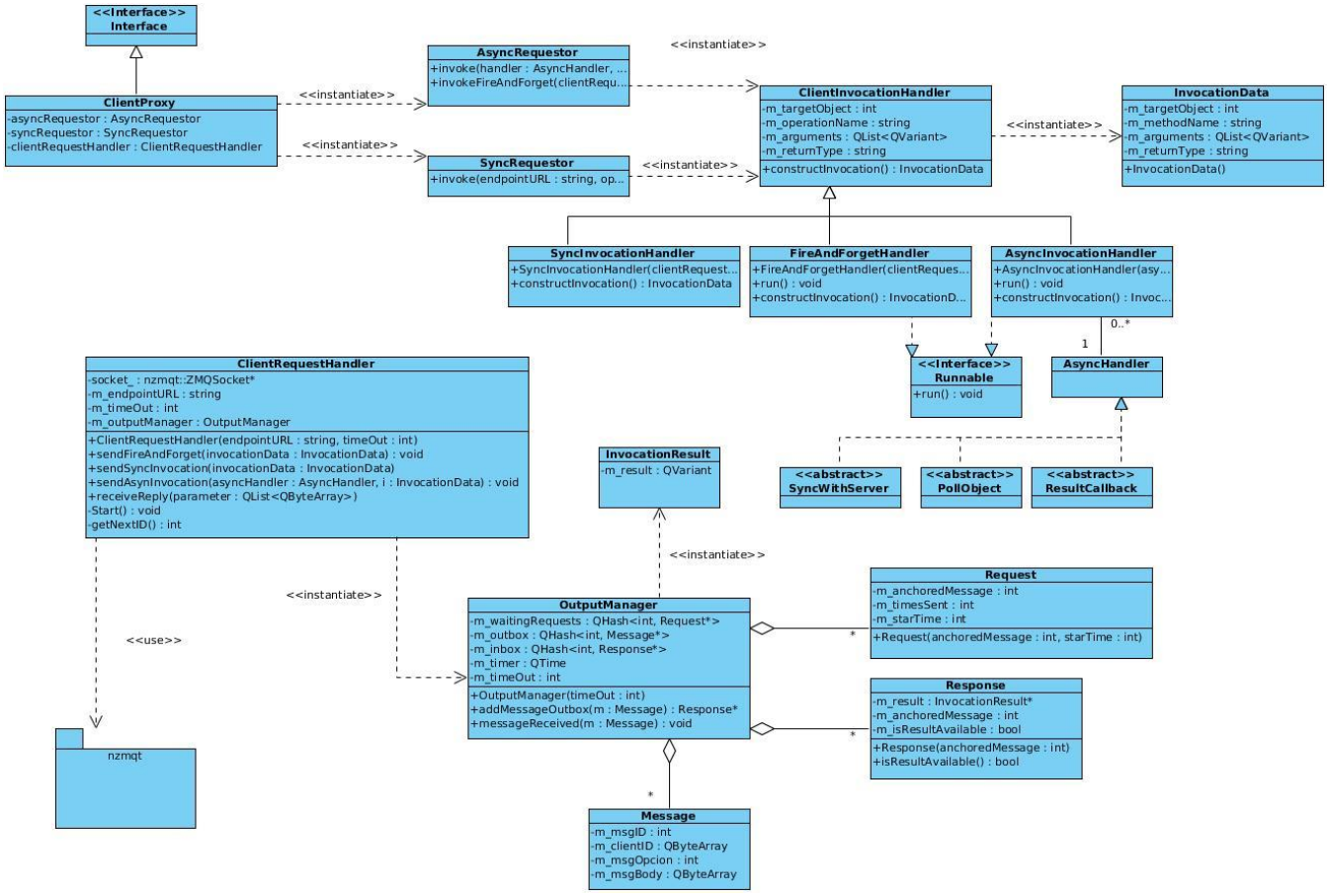

Ilustración 27 Diagrama de clases: Servicio de invocación remota lado Cliente.

2.9.4. Diagrama de clases: Servicio de invocación remota lado Servidor

El siguiente diagrama muestra la relación existente entre las clases del servicio de invocación remota del lado del servidor, resaltando por su importancia la clase *LifecycleManagerRegistry,* la cual es la encargada de almacenar una instancia de cada tipo de administración de vida que exista, ya sea en este caso instancia estática *StaticInstanceLifecycleManager* o por invocación *PerRequestLifecycleManager,* la clase *LifecycleManagerRegistry* contiene un método llamado: “getLifecycleManager” el cual busca mediante el identificador del objeto pasado por parámetro en que administrador se encuentra y devolviendo el mismo. La clase *ServerRequestHandler* es la encargada con el uso del patrón que implementa ZMQ *Repier* (REP) de enviar las respuestas de las solicitudes que llegan desde el cliente.

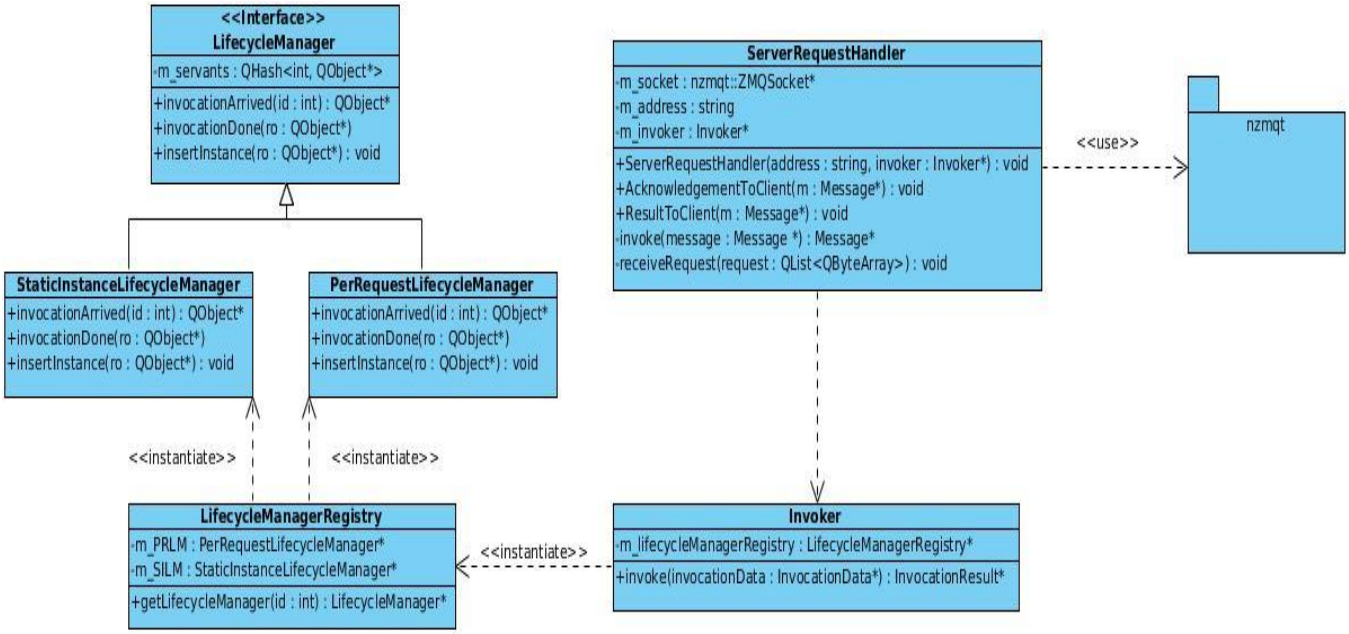

Ilustración 28 Diagrama de clases: Servicio de invocación remota lado Servidor.

2.10. Tarjetas CRC

Aunque en general el diseño es realizado por los propios desarrolladores en ocasiones se reúnen aquellos con más experiencia o incluso se involucra al cliente para diseñar las partes más complejas. En estas reuniones se emplean un tipo de tarjetas denominadas CRC (*Class, Responsabilities and Collaborator* - Clase, Responsabilidad y Colaborador) cuyo objetivo es facilitar la comunicación y documentar los resultados. Para cada clase identificada se rellenará una tarjeta de este tipo, se especificará su finalidad y las clases con las que interactúa. Las tarjetas CRC son una buena forma de cambiar de la programación estructurada a una filosofía orientada a objetos.

Tarjetas CRC del patrón Event Notifier

Tabla 14 Tarjeta CRC "EventService"

Tarjeta CRC	
Clase: EventService	
Responsabilidades: Clase abstracta que posee las operaciones necesarias para la publicación, subscripción y de subscripción.	Colaboradores: Filter Event Subscriber

Tabla 15 Tarjeta CRC "EventProxy"

Tarjeta CRC	
Clase: EventProxy	
Responsabilidades: Clase que hereda de la clase EventService y de Subscriber redefiniendo los métodos para realizar publicaciones y subscripciones remotamente.	Colaboradores: EventService Subscriber

Tabla 16Tarjeta CRC "RemoteEventService"

Tarjeta CRC	
Clase: RemoteEventService	
Responsabilidades: Clase abstracta que posee las operaciones necesarias para la publicación, subscripción y de subscripción remota.	Colaboradores: Filter Event

	Subscriber

Tabla 17 Tarjeta CRC "RemoteEventPublisher"

Tarjeta CRC	
Clase: RemoteEventPublisher	
Responsabilidades: Clase encargada de mediante el uso de la bibliotecaZeroMQy el uso de sus patrones Subscripción y Publicación, publicar un evento a todos los subscritos.	Colaboradores: Event Subscriber

Tabla 18 Tarjeta CRC "RemoteEventSubscriber"

Tarjeta CRC	
Clase: RemoteEventSubscriber	
Responsabilidades: Clase encargada de mediante el uso de la biblioteca ZeroMQ y el uso de sus patrones Subscripción y Publicación, suscribirse a un evento específico.	Colaboradores: Event Subscriber

Conclusiones parciales

Se concluye con la identificación de 11 Historias de Usuarios, 10 son de prioridad Alta y 1 de prioridad Media, las cuales sirvieron como base para el desarrollo de un sistema que cumpliera con los requerimientos del cliente. Mediante el estudio realizado la bibliografía consultada propone el uso de dos arquitecturas importantes la Arquitectura Cliente Servidor y la N-Capas, las cuales sirvieron para estructurar el sistema de clases y sus responsabilidades, así como el estilo arquitectónico para el servicio de invocación a objetos remotos *Broker,* y el patrón de diseño *Event Notifier* para el servicio de publicación y subscripción, que permitirán la comunicación entre los diferentes módulos del sistema Video Vigilancia Xilema Suria, dándole organización, escalabilidad y flexibilidad al sistema.

3. CAPÍTULO 3 Implementación y Prueba

En este acápite se pasará una vez realizado el análisis y diseño del sistema a la implementación, el cual estará guiado por las tareas que componen cada historia de usuario descrita en la fase de exploración. Este capítulo está dividido en dos secciones: Implementación y Pruebas. En la primera sección se realizan las implementaciones de las clases y objetos descritas en las anteriormente elaboradas tarjetas CRC, contando para ello con la realización de las tareas de la ingeniería. En la segunda sección se describen las pruebas a las que fue sometido el *middleware* de comunicación en cada iteración, en función de identificar y corregir fallos cometidos durante el desarrollo de las Historias de Usuario.

3.1. Implementación del sistema

Los estándares de codificación son pautas de programación que no están enfocadas a la lógica del programa, sino a su estructura y apariencia física para facilitar la lectura, comprensión y mantenimiento del código.(19)

En esta fase se plantea la implementación de las Historias de Usuario en su correspondiente iteración, obteniéndose en cada una de ellas una versión funcional del producto. Para ello se descomponen las historias de usuario en tareas de desarrollo para ser realizadas, describiendo cada una de estas.

3.2. Estándares de Codificación

Los estándares de codificación, también llamados estilos de programación o convenciones de código, son convenios para escribir código fuente en ciertos lenguajes de programación. Estos estándares facilitan el mantenimiento del código, sirven como punto de referencia para los programadores, mantienen un estilo de programación y ayudan a mejorar el proceso de codificación, para lograr una mayor eficiencia.

- El código será escrito en inglés.
- El estándar de codificación utilizado es el mismo que el que utiliza el *framework Qt*.

```
class OBJECTREMOTEBROKERSHARED_EXPORT OutputManager: public QObject
{
    Q_OBJECT
public:
    OutputManager();
    Response* addMessageOutbox(Message* m);


    QHash<int, Message *> outbox() const;
    void setOutbox(const QHash<int, Message *> &outbox);

    QHash<int, Request *> waitingRequests() const;
    void setWaitingRequests(const QHash<int, Request *> &waitingRequests);

    QHash<int, Response *> inbox() const;
    void setInbox(const QHash<int, Response *> &inbox);

    void messageReceived(Message* m);
private slots:
    void checkIn();

private:
    QHash<int, Request*> m_waitingRequests;
    QHash<int, Message *> m_outbox;
    QHash<int, Response*> m_inbox;

    QTime m_timer;
signals:
    void sendMessage(Message* m);

};
```

Ilustración 29 Ejemplo de aplicación del Estándar de Codificación.

3.3. Tareas de la ingeniería por Historias de Usuarios

Una vez definidas las HU el equipo de desarrollo divide cada una de ellas en una serie de tareas que contribuyan al desarrollo de las mismas. Las tareas pueden ser descritas en un lenguaje técnico que no necesariamente garantice el entendimiento del cliente.

Iteración 1

Tabla 19 Tarea de ingeniería "Implementar la publicación"

Tarea: Implementar la publicación de eventos remotos	
Número de tarea: 1	Historia de usuario: 1
Nombre de la tarea: Implementar la publicación de eventos remotos.	
Tipo de tarea: Desarrollo	Puntos estimados: 1
Fecha inicio:	Fecha fin:
Programador responsable: Alberto Marturelo Lorenzo	
Descripción: Con la utilización de la biblioteca de comunicación mediante socket ZMQ permitir la publicación de eventos.	

Tabla 20 Tarea de Ingeniería "Implementar la subscripción de eventos remotos"

Tarea: Implementar la subscripción de eventos remotos	
Número de tarea: 2	Historia de usuario: 2
Nombre de la tarea: Implementar la subscripción de eventos remotos.	
Tipo de tarea: Desarrollo	Puntos estimados: 1
Fecha inicio:	Fecha fin:
Programador responsable: Alberto Marturelo Lorenzo	
Descripción: Con la utilización de la biblioteca de comunicación mediante socket ZMQ permitir la subscripción de eventos.	

Iteración 2

Tabla 21 Tarea de Ingeniería "Implementación de invocaciones síncronas"

Tarea: Implementación de invocaciones síncronas.	
Número de tarea: 3	Historia de usuario: 3
Nombre de la tarea: Implementación de invocaciones síncronas.	
Tipo de tarea: Desarrollo	Puntos estimados: 1
Fecha inicio:	Fecha fin:
Programador responsable: Alberto Marturelo Lorenzo	
Descripción: Con la utilización de la biblioteca de comunicación mediante socket ZMQ permitir la ejecución síncrona de operaciones en objetos remotos.	

Tabla 22 Tarea de Ingeniería "Implementación de invocaciones asíncronas sin retorno (*Fire and forget*)."

Tarea: Implementación de invocaciones asíncronas sin retorno (*Fire and forget*).	
Número de tarea: 4	Historia de usuario: 5
Nombre de la tarea: Implementación de invocaciones asíncronas sin retorno (*Fire and forget*).	
Tipo de tarea: Desarrollo	Puntos estimados: 1
Fecha inicio:	Fecha fin:
Programador responsable: Alberto Marturelo Lorenzo	
Descripción: Con la utilización de la biblioteca de comunicación mediante socket ZMQ permitir la ejecución asíncrona sin retorno de operaciones en objetos remotos.	

Iteración 3

Tabla 23 Tarea de Ingeniería "Implementación de invocaciones asíncronas con retorno"

Tarea: Implementación de invocaciones asíncronas con retorno.	
Número de tarea: 5	Historia de usuario: 4
Nombre de la tarea: Implementación de invocaciones asíncronas con retorno.	
Tipo de tarea: Desarrollo	Puntos estimados: 1
Fecha inicio:	Fecha fin:
Programador responsable: Alberto Marturelo Lorenzo	
Descripción: Con la utilización de la biblioteca de comunicación mediante socket ZMQ permitir la ejecución asíncrona con retorno de operaciones en objetos remotos.	

Tabla 24 Tarea de Ingeniería "Implementación de tipo de retorno mediante *Result CALLBACK*".

Tarea: Implementación de tipo de retorno mediante *Result CALLBACK*.	
Número de tarea: 6	Historia de usuario: 6
Nombre de la tarea: Implementación de tipo de retorno mediante Result CALLBACK.	
Tipo de tarea: Desarrollo	Puntos estimados: 1
Fecha inicio:	Fecha fin:
Programador responsable: Alberto Marturelo Lorenzo	
Descripción: Permitir el tipo de retorno *Return CALLBACK* para las invocaciones asíncronas.	

Iteración 4

Tabla 25 Tarea de Ingeniería "Implementación de tipo de retorno mediante Poll Object."

Tarea: Implementación de tipo de retorno mediante *Poll Object.*	
Número de tarea: 7	Historia de usuario: 7
Nombre de la tarea: Implementación de tipo de retorno mediante *Poll Object.*	
Tipo de tarea: Desarrollo	Puntos estimados: 1
Fecha inicio:	Fecha fin:
Programador responsable: Alberto Marturelo Lorenzo	
Descripción: Permitir el tipo de retorno Poll Object para las invocaciones asíncronas.	

Tabla 26 Tarea de Ingeniería "Implementación de tipo de notificación de invocación remota *Sync With Server*".

Tarea: Implementación de tipo de notificación de invocación remota Sync With Server.	
Número de tarea: 8	Historia de usuario: 8
Nombre de la tarea: Implementación de tipo de notificación de invocación remota Sync With Server.	
Tipo de tarea: Desarrollo	Puntos estimados: 1
Fecha inicio:	Fecha fin:
Programador responsable: Alberto Marturelo Lorenzo	
Descripción: Permitir el tipo de notificación de invocación remota Sync With Server para las invocaciones asíncronas.	

Iteración 5

Tabla 27 Tarea de Ingeniería "Implementación del administrador de vida para la invocación de objetos estáticos".

Tarea: Implementación del administrador de vida para la invocación de objetos estáticos	
Número de tarea: 9	Historia de usuario: 9
Nombre de la tarea: Implementación del administrador de vida para la invocación de objetos estáticos.	
Tipo de tarea: Desarrollo	Puntos estimados: 1
Fecha inicio:	Fecha fin:
Programador responsable: Alberto Marturelo Lorenzo	
Descripción: Permitir la instancia de objetos estáticos en el servidor, de forma que siempre estén activos mientras el servidor lo este.	

Tabla 28 Tarea de Ingeniería "Implementación del administrador de vida para la invocación de objetos no estáticos".

Tarea: Implementación del administrador de vida para la invocación de objetos no estáticos	
Número de tarea: 10	Historia de usuario: 10
Nombre de la tarea: Implementación del administrador de vida para la invocación de objetos no estáticos	
Tipo de tarea: Desarrollo	Puntos estimados: 1
Fecha inicio:	Fecha fin:
Programador responsable: Alberto Marturelo Lorenzo	
Descripción: Permitir la instancia de objetos no estáticos en el servidor, de forma que estos se instancien solamente cuando el cliente requiera ejecutar una de sus operaciones.	

Iteración 6

Tabla 29 Tarea de Ingeniería "Implementación del sistema para eliminar la subscripción de eventos remotos."

Tarea: Implementación del sistema para eliminar la subscripción de eventos remotos.	
Número de tarea: 11	Historia de usuario: 11
Nombre de la tarea: Implementación del sistema para eliminar la subscripción de eventos remotos.	
Tipo de tarea: Desarrollo	Puntos estimados: 1
Fecha inicio:	Fecha fin:
Programador responsable: Alberto Marturelo Lorenzo	
Descripción: Permitir al sistema eliminar la subscripción de eventos remotos.	

3.4. Pruebas de software

El proceso de pruebas es uno de los pilares fundamentales de la metodología XP, el cual ayuda al cliente a verificar y concretar las funcionalidades de las HU, por lo que favorece la comunicación entre el cliente y el equipo de desarrollo. Esta filosofía ayuda a identificar y corregir fallos u omisiones cometidas en las mismas, por lo que se reduce el número de errores no detectados así como el tiempo entre la introducción de este en el sistema y su detección. Permite identificar HU adicionales que no fueran obvias para el cliente o en las que cliente no hubiese pensado de no enfrentarse a dicha situación. Todo esto contribuye a elevar la calidad de los productos desarrollados y a la seguridad de los programadores a la hora de introducir cambios o modificaciones.

XP divide las pruebas en dos grupos: pruebas unitarias, desarrolladas por los programadores, se escriben antes que el código y son las encargadas de verificar el mismo de forma automática y las

pruebas de aceptación, destinadas a evaluar si al final de una iteración se obtuvo la funcionalidad requerida, además de comprobar que dicha funcionalidad sea la esperada por el cliente.

3.4.1. Pruebas unitarias

Las pruebas unitarias son una de las piedras angulares de XP. Todos los módulos y componentes deben de pasar las pruebas unitarias antes de ser liberados o publicados. Todo código liberado debe pasar correctamente las pruebas unitarias lo que habilita que funcione la propiedad colectiva del código. En este sentido, el sistema y el conjunto de pruebas debe ser guardado junto con el código, para que pueda ser utilizado por otros desarrolladores, en caso de tener que corregir, cambiar o recodificar parte del mismo.
Las pruebas unitarias son pruebas de caja blanca, definidas por el programador, que se le hacen a pequeñas porciones de código, por separados, para verificar su correcta funcionalidad, las mismas se pueden ir efectuando desde que se está implementando, no necesariamente se tiene que esperar al final de la implementación del producto (Ver Ilustración 31 Método de Caja Blanca).

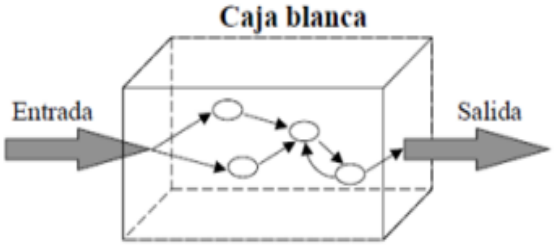

Ilustración 30 Método de Caja blanca

La prueba de caja blanca se basa en el diseño de casos de prueba que usa la estructura de control del diseño procedimental para derivarlos. Mediante la prueba de la caja blanca el ingeniero del software puede obtener casos de prueba que garanticen que se ejerciten por lo menos una vez todos los caminos independientes de cada módulo, programa o método; ejerciten todas las decisiones lógicas en las vertientes verdadera y falsa; ejecuten todos los bucles en sus límites operacionales y ejerciten las estructuras internas de datos para asegurar su validez (SOMMERVILLE, 2002).

Pruebas de Camino Básico

La prueba del camino básico es una técnica de prueba de la Caja blanca propuesta por Tom McCabe. Esta técnica permite obtener una medida de la complejidad lógica de un diseño y se usa como guía para la definición de un conjunto básico. La idea es derivar casos de prueba a partir de un conjunto dado de caminos

independientes por los cuales puede circular el flujo de control. Para obtener dicho conjunto de caminos independientes se construye el grafo de flujo asociado y se calcula su complejidad ciclomática.

Los pasos que se siguen para aplicar esta técnica son:

1. A partir del diseño o del código fuente, se dibuja el grafo de flujo asociado.
2. Se calcula la complejidad ciclomática del grafo.
2. Se determina un conjunto básico de caminos independientes.
3. Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

El cálculo de la complejidad ciclomática puede ser realizado de tres maneras: (Pressman, 2002)

1. El número de regiones del grafo de flujo coincide con la complejidad ciclomática.
2. La complejidad ciclomática V (G), de un grafo de flujo G se define como: V (G) = A - N + 2, donde: A es el número de aristas del grafo de flujo y N es el número de nodos del mismo.
3. La complejidad ciclomática V (G), de un grafo de flujo G también se define como: V (G) = P + 1, donde P es el número de nodos predicado contenido en el grafo G.

En la siguiente **¡Error! No se encuentra el origen de la referencia.** se detalla el modelo general que eguirán los casos de pruebas para cada camino encontrado.

Tabla 30 Casos de pruebas del camino básico.

Casos de pruebas del camino básico	
Camino	
Descripción	Se hace la entrada de los datos necesarios, validando que ningún parámetro obligatorio pase nulo al procedimiento y no se entre ningún dato erróneo.
Condición de ejecución	Se especifica cada parámetro para que cumpla una la condición deseada para ver el funcionamiento del procedimiento.
Entrada	Se muestran los parámetros que entran al procedimiento.
Resultados esperados	Se expone el resultado que se espera que devuelva el procedimiento.

A continuación se muestran las pruebas de caja blanca realizadas a algunos de las funcionalidades fundamentales de la aplicación.

Funcionalidad: **receiveRequest** (slot encargado de activarse en el momento que el servidor recibe un mensaje de alguno de los clientes).

La Ilustración 32. Código de la funcionalidad: **receiveRequest** muestra las rutinas de código y posteriormente se realiza el procedimiento de pruebas unitarias para validar su correcto funcionamiento.

```
void ServerRequestHandler::receiveRequest(const QList<QByteArray> &request)
{
    Message* m = new Message; ———————— 1
    QByteArray dataMsg = request[1];
    QDataStream streamM(&dataMsg, QIODevice::ReadWrite);
    streamM >> *m;
    //qDebug() << m->msgID();

    if(m->msgOpcion() == 00) ———————— 2
        invoke(m); ———————————————— 3
    if(m->msgOpcion() == 10){ ———————— 4
        AcknowledgementToClient(m); ———— 5
        invoke(m); ———————————————— 3
    }
    if(m->msgOpcion() == 11){ ———————— 6
        AcknowledgementToClient(m); ———— 5
        ResultToClient(m); ———————————— 7
    }
}
```

Ilustración 31 Código de la funcionalidad *receiveRequest*

A partir del código fuente mostrado en la ilustración 32 se construye el grafo de flujo asociado.

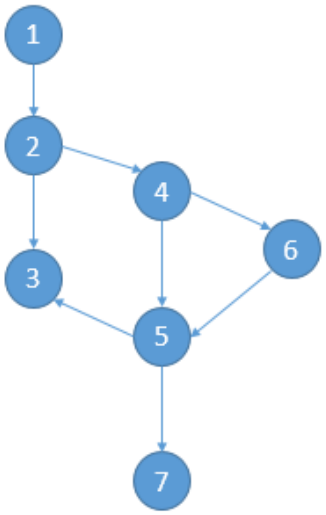

Ilustración 32 Grafo de control de flujo asociado a la funcionalidad: receiveRequest

Para el caso del algoritmo de la funcionalidad presentada, se toma como referencia el grafo de flujo de la Tabla 31 Caminos básicos de la funcionalidad: *receiveRequest* y se calcula la complejidad ciclomática de la manera que se lista a continuación:

- ✓ V(G) = 8 aristas – 7 nodos + 2 = 3

A partir del resultado obtenido, se determina que el algoritmo presenta una complejidad ciclomática de 3, lo que deriva que presenta a lo sumo 3 caminos lógicos por donde ejecutarse dicha funcionalidad.

Tabla 31 Caminos básicos de la funcionalidad: *receiveRequest*

No	Camino básico
1	1,2,3
2	1,2,4,5,3
3	1,2,6,5,7

Luego de haber identificado los caminos básicos del flujo se procede a realizar los casos de prueba para la funcionalidad en cuestión. Se debe elaborar al menos un caso de prueba por cada camino básico. Seguidamente se detallan los casos de prueba creados para realizar las pruebas del procedimiento: *receiveRequest*.

Tabla 32 Caso de prueba para el camino 1

Camino	1,2,3
Descripción	No existen instrucciones para traducir a seudocódigo C.
Condición de ejecución	Se ejecuta cuando se emite la señal del socket, que llego un mensaje al servidor.
Entrada	El contenido de la variable request donde el mensaje de serializado tiene como opción 00 refiriéndose a la invocación sin acuse de recibo y sin retorno del servidor.
Resultados esperados	Se espera que se realice la invocación en el objeto remoto especificado por la invocación.

Tabla 33 Caso de prueba para el camino 2

Camino	1,2,4,5,3
Descripción	No existen instrucciones para traducir a seudocódigo C.
Condición de ejecución	Se ejecuta cuando se emite la señal del socket, que llego un mensaje al servidor.
Entrada	El contenido de la variable request donde el mensaje de serializado tiene como opción 10 refiriéndose a la espera por parte del cliente del acuse de recibo por parte del servidor pero no el retorno de un resultado.
Resultados esperados	Se espera que el cliente reciba el acuse de recibo y luego se ejecute la invocación en el objeto remoto sin esperar retorno.

Tabla 34 Caso de prueba para el camino 3

Camino	1,2,6,5,7
Descripción	No existen instrucciones para traducir a seudocódigo C.
Condición de ejecución	Se ejecuta cuando se emite la señal del socket, que llego un mensaje al servidor.
Entrada	El contenido de la variable Request donde el mensaje de serializado tiene como opción 10 refiriéndose a la espera por parte del cliente del acuse de recibo por parte del servidor y el retorno de un resultado.
Resultados esperados	Se espera que el cliente reciba el acuse de recibo y luego el resultado de la invocación realizada en el objeto remoto.

3.4.2. Pruebas de aceptación

Como técnica para garantizar que los requerimientos hayan sido cumplidos y que la aplicación es realmente lo que el cliente necesita, además de asegurar su correcto funcionamiento son realizadas las pruebas de aceptación. Las cuales son creadas a partir de las historias de usuario y desde la perspectiva

del cliente como parte del equipo de desarrollo con el objetivo de evaluar al final de cada iteración si se alcanzaron las metas propuestas. Una historia de usuario puede tener todas las pruebas de aceptación que necesite para asegurar su correcto funcionamiento. Estas pruebas funcionan como una caja negra, pues cada una de ellas representa una salida esperada del sistema, donde es responsabilidad del cliente verificar la corrección de las pruebas y tomar decisiones acerca de las mismas.

El objetivo final de las mismas es lograr que los requerimientos sean cumplidos y que el sistema sea aceptable. Una vez que todas las historias de usuario hayan pasado sus pruebas de aceptación se considera entonces terminada la aplicación.

Tablas de casos de pruebas

Las partes que componen los casos de prueba son las siguientes:

1- El código está dado por las letras claves (HU) refiriéndose a las Historias de Usuario, seguido por el número de Historia de Usuario, un guion bajo y a letra “p” refriéndose a las pruebas, además el número de prueba según la Historia de Usuario.
2- El número de Historia de Usuario.
3- Nombre de las pruebas realizada.
4- Una breve descripción del objetivo que se quiere lograr con las pruebas.
5- Condiciones necesarias para ejecutar las pruebas.
6- Lo que se debe hacer para realizar las pruebas.
7- El resultado que se espera al finalizar las pruebas.
8- Evaluación que se le da a las pruebas: Satisfactoria- Parcial- Insatisfactoria.

Tabla 35 Prueba 1 de la HU “Publicar evento remotamente.”

Caso de prueba de aceptación	
Código: HU1_p1	**Historia de usuario:** 1
Nombre: Publicar evento remotamente.	
Descripción: En este caso de prueba se pretende comprobar que la funcionalidad Publicar evento remoto.	
Condiciones de ejecución: El evento que se desea publicar debe haberse implementado como clase anteriormente.	
Entrada/Paso de ejecución: Se crea una instancia de la clase *RemoteEventService*, la cual crea una instancia del socket de publicación, luego se crea una instancia de la clase *RemotePublisher*	

que tiene un método que se llama *sendEventse* le pasa por parámetro el evento que se desea publicar.
Resultado esperado: Se espera que todos los clientes subscritos al evento publicado reciban la notificación.
Evaluación de la prueba: Satisfactoria.

Las restantes tablas de los casos de prueba de aceptación se encuentran en los Anexos del documento (Anexos 3: Pruebas de aceptación.).

Cumplimiento de los principios de la computación distribuida

A continuación se describe con cuales de los principios de la computación distribuida cumple el sistema.

Escalabilidad:

El sistema es escalable si conserva su efectividad al ocurrir un incremento considerable en el número de recursos y en el número de usuarios.

Una de las estrategias fundamentales llevadas cabo con el fin de garantizar que el sistema desarrollado sea escalable fue la utilización del middleware de mensajería ZMQ, el cual soporta una gran cantidad de computadoras suscritas a sus colas. Además, el sistema ejecuta las invocaciones y la manera de informar la ocurrencia de un evento al subscriptor de manera concurrente.

Todo lo anteriormente planteado garantiza que el sistema pueda continuar funcionando sin necesidad de realizar cambios independientemente de la cantidad de la cantidad de clientes conectados al sistema y la cantidad de tareas ejecutándose en el servidor.

Tratamiento de Fallos: La posibilidad que tiene el sistema para seguir funcionando ante fallos de algún componente en forma independiente, pero para esto se tiene que tener alguna alternativa de solución.

Por más confiable que pudiese parecer un sistema es necesario estar siempre preparado para cuando este eventualmente falle. Específicamente en un sistema distribuido los fallos que se producen son parciales, pues algunos componentes fallan, mientras otros siguen funcionando, dejando de brindarse solo los servicios ofrecidos por los componentes afectados.

Detección de fallos: Para dar tratamiento a un fallo es necesario primeramente detectar que este ha ocurrido. El sistema desarrollado hace uso del estilo arquitectónico *Broker* el cual define que la respuesta a la invocación puede ser una respuesta o una excepción, y la aparición de este será notificada al cliente.

Enmascaramiento de fallos: El sistema realiza el enmascaramiento de fallos mediante la corrección de los mismos apenas hayan sido detectados, de manera que el usuario no percibe en ningún momento que está ocurriendo un fallo.

Tolerancia de fallos: La tolerancia a fallos es garantizada mediante la definición de un tiempo de vida a las invocaciones, las cuales si superan el tiempo de espera, si esto sucede se será notificado al cliente y el sistema seguirá con su ejecución normal.

Transparencia: El sistema con el uso de los patrones de publicación y suscripción XPub y XSub, se garantiza que cada cliente solo conozca la dirección IP del servidor, desinteresándole de donde vino la publicación del evento, en el servicio de invocación de eventos remotos, se garantiza ya que cada cliente ejecutará las operaciones locales y remotas por igual siendo transparente la diferencia.

Flexibilidad: El sistema permite la inserción de eventos, filtros y subscriptores al sistema sin cambiar la estructura del mismo. Permite la aparición de nuevos clientes sin importar su responsabilidad al sistema ya que el servidor da respuesta a cada una de las peticiones sin importar quien la realizó.

Conclusiones parciales

Se usó el estándar de codificación que propone Qt garantizando que la implementación sea organizada y sea entendida por futuros desarrolladores y facilite su mantenimiento.

Se implementaron los dos servicios definidos como requisitos del sistema, el servicio de publicación y subscripción y de invocación de objetos remotos.

Se realizaron pruebas unitarias y de aceptación permitiendo que se comprobara que el sistema cumpla con cada uno de los requisitos definidos por el cliente.

Conclusiones Generales

Con la realización del presente trabajo se logró:

- La identificación de ZMQ como biblioteca de comunicación para el diseño e implementación del *middleware* de comunicación; la cual permitió la implementación de la capa de comunicación entre la aplicación cliente y el servidor.
- Se seleccionó XP como metodología rectora para guiar el desarrollo del software; por el énfasis que hace la misma a la implementación y dedicándole poco esfuerzo a la documentación.
- La investigación aportó un servicio de notificación de eventos y de invocación de operaciones a objetos; el cual surte las necesidades de conexión entre los módulos del Sistema Video Vigilancia Xilema Suria Visión.
- La investigación aportó un sistema que cumple con los requerimientos por los cuales fue pensado, la publicación y subscripción de eventos remotamente y la invocación de operaciones en objetos remotos.
- Por las pruebas arrojadas durante el desarrollo del sistema, se comprobó que el sistema responde a los requerimientos funcionales descritos, evaluando los resultados a través de las pruebas unitarias y de aceptación realizadas.

Recomendaciones

Como parte del proceso de mejora continua del desarrollo de software se propone la siguiente recomendación tributaria a un producto de mayor calidad.

- Implementar los patrones *Client-Dependent Instance* que es otro tipo de ciclo de vida de objetos remotos y *Passivation* que permite al sistema liberar memoria de forma tal que un objeto al cabo de un tiempo sin usarse pasar a un estado de inactividad, eliminándolo de la memoria y guardando su instancia en un archivo SQL Lite (Puder, et al., 2006).
- Agregar una capa de seguridad mediante el uso del protocolo de seguridad SSL.

Referencias

1. **Bakken, David E. 2003.** *Middleware.* Washington : Washington State University, 2003.
2. **Cambiaso, Diego. 2010.** pixelcoblog. *Pixelco Blog.* [En línea] 10 de Junio de 2010. [Citado el: 20 de Enero de 2014.] http://pixelcoblog.com/qt-creator-completo-entorno-de-desarrollo-multiplataforma/.
3. **Chapa, Sonia Guadalupe Mendoza. 1996.** *A System of Patterns -Pattern Oriented Software Architecture.* Mexico : Buschmann et al, 1996.
4. **Cohn, Mike. 2004.** *User Stories Applied.* UUEE : Addison Wesleey, 2004. ISBN 0-321-20568-5.
5. **Delgado, Jordan. 2011.** Teoría de programación. *Teoría de programación.* [En línea] 28 de Junio de 2011. [Citado el: 08 de 12 de 2013.] https//www.Caracteristicas del lenguaje C++.html.
6. *Event Notifier, a Pattern for Event Notification.* **Suchitra, Gupta, Hartkopf, Jeff y Ramaswamy, Suresh. 1998.** 7, s.l. : SIGS Publications, 1998, Vol. 3.
7. **Fernández., Raúl Castro. 2009.** *Desarrollo de software de videovigilancia para sistemas embarcados distribuidos con ICE.* Madrid : Universidad Carlos III de Madrid, 2009.
8. **Gutiérrez, J.J, Escalona, M. Mejías M.J, Torres J. 2010.** *Pruebas del sistema en Programación.* Sevilla : s.n., 2010.
9. **Gutiérrez, Jorge Antonio Díaz. 2009.** *Desarrollo de un IDE libre y multiplataforma para la .* Habana : Universidad de Ciencias Informaticas, 2009.
10. **iMatrix. 2013.** ZeroMQ. *iMatrix Corporation.* [En línea] iMatrix Corporation, 5 de Septiembre de 2013. [Citado el: 15 de Enero de 2014.] http://zeromq.org.
11. **Jankolovska, S. y Zherajikj, B. 2012.** *Access Point Framework:-towards a more scalable ESB solution.* Moscu : KTM, 2012.
12. **Jankolovska, S. y Zherajikj, B. 2012.** *Access Point Framework:-towards a more scalable ESB.* s.l. : KTM, 2012.
13. **Jara, Omar Hurtado. 1997.** *Sistemas Distribuidos.* España : Departamento de Investigación de la Universidad Carlos III de Madrid, 1997.
14. **Joskowicz, José. 2008.** *Reglas y Prácticas en Extreme Programming.* 2008.
15. **Kruchten, P. 2002.** *A Software Development Process for a Team of One.* 2002.

16. **Luis Díaz, Heidy y Berrillo Borrero, Sael José. 2013.** *Plataforma de procesamiento distribuido en entornos .* Habana : Universidad de las Ciencias Informáticas, 2013.
17. **—. 2013.** *Plataforma de procesamiento distribuido en entornos colaborativos.* Habana : Universidad de las Ciencias Informáticas, 2013.
18. **Nawaz, Ing. Sarfaraz. 2009.** Stackoverflow. *Stackoverflow.com.* [En línea] 8 de 9 de 2009. [Citado el: 10 de 1 de 2014.] http://stackoverflow.com/questions/731233/activemq-or-rabbitmq-or-zeromq-or.
19. **Piël, N. 2010.** ZeroMQ an introduction. *ZeroMQ.* [En línea] 15 de Septiembre de 2010. [Citado el: 18 de Enero de 2014.] http://nichol.as/zeromq-an-introduction.
20. **Pressman, Roger S. 2002.** *Ingeniería de Software. Quinta Edición.* Madrid : McGraw-Hill Interamericana, 2002.
21. **Puder, Arno, Kay, Romer y Pilhfer, Frank. 2006.** *Distributed Systems Architecture a Middleware Appoach.* Ansterdam : Morgan Kaufmann Publishers, 2006. 978-1-55860-648-7.
22. **Reynoso, Carlos y Kicillof, Nicolás. 2004.** *Estilos y Patrones en la Estrategia de Arquitecturade Microsoft.* Argentina : UNIVERSIDAD DE BUENOS AIRES, 2004.
23. **Riehle, Dirk. 2010.** *The Event Notification Pattern Integrating-Implicit Invocation with Object-Orientation.* Zurich : UBILAB, 2010.
24. **Serrano Cuayahuitl, Victor Hugo. 2011.** *Pruebas de unidad, Estándares de codificación y generación.* Universidad Autonoma de Tlaxcala : Univercidad Autónoma de Tlaxcala, 2011.
25. **SOMMERVILLE, I. 2002.** *Ingeniería de Software. 6ta. Edición.* s.l. : Prentice-Hall, 2002. ISBN 970-260206-8.
26. **2012.** Visual Paradigm para UML. *FREEDOWNLOADMANAGER.* [En línea] 18 de 12 de 2012. http://www.freedownloadmanager.org/es/downloads/Paradigma_Visual_para_UML_(Iglesia_Anglicana)_%5BMac_OS_X_cuenta_14717_p/.
27. **Warden, Shore and. 2007.** *The Art of Agile Development Primera Edición.* O'Reilly : Cambridge University Press, 2007.
28. **ZeroC. 2011.** *ICE-Manual.* 2011.

Glosario

GNU LGPL: La Licencia Pública General Reducida de GNU (GNU LGPL, acrónico en ingles de GNU (Lesser General Public License) es una licencia de software creada por la *Free Software Foundation* que pretende garantizar la libertad de compartir y modificar el software cubierto por ella, asegurando que el software es libre para todos sus usuarios.

UML: Lenguaje Unificado de Modelado (LUM) o (UML, por sus siglas en inglés, Unified Modeling Language) es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad; está respaldado por el OMG (Object Management Group).

Middleware: Capa de software mediante la cual el sistema puede abstraerse de las redes, hardware, lenguajes de programación y sistemas operativos, sobre los que se ejecuta.

ZMQ: *Middleware* de mensajería multiplataforma que implementa completamente el protocolo AMQP.

Protocolo AMQP: Acrónico en ingles de *Advanced Message Queuing Protocol*, es un protocolo de estándar abierto en la capa de aplicaciones de un sistema de comunicación orientado a mensajes.

4. Anexos

Anexo #1 Historias de Usuarios

Tabla 36 Historias de Usuarios Invocación síncrona de operaciones en el objeto remoto.

Historia de Usuario: Invocación síncrona de operaciones en el objeto remoto.	
Número: 4	**Usuario:** Programador
Nombre historia: Invocación síncrona de operaciones en el objeto remoto.	
Puntos estimados: 1	**Puntos reales:** 1.5
Prioridad en negocio: Alta	**Riesgo en desarrollo:** Alto
Programador responsable: Alberto Marturelo Lorenzo	
Descripción: El sistema debe ser capaz de invocar operaciones y esperar por la respuesta de forma síncronamente.	
Observaciones:	

Tabla 37 Historias de Usuarios Invocación asíncrona de operaciones en el objeto remoto.

Historia de Usuario: Invocación asíncrona de operaciones en el objeto remoto.	
Número: 5	**Usuario:** Programador
Nombre historia: Invocación asíncrona de operaciones en el objeto remoto.	
Puntos estimados:1	**Puntos reales:**1.5
Prioridad en negocio: Alta	**Riesgo en desarrollo:** Alto
Programador responsable: Alberto Marturelo Lorenzo	
Descripción: El sistema debe ser capaz de invocar operaciones y continuar con la ejecución de otras actividades hasta que la respuesta en el servidor este lista.	
Observaciones:	

Tabla 38 Historias de Usuarios Invocación asíncrona sin retorno de objetos remoto.

Historia de Usuario: Invocación asíncrona sin retorno de objetos remoto.	
Número: 6	**Usuario:** Programador
Nombre historia: Invocación asíncrona sin retorno de objetos remoto.	
Puntos estimados:1	**Puntos reales:**1.5
Prioridad en negocio: Alta	**Riesgo en desarrollo:** Alto

Programador responsable: Alberto Marturelo Lorenzo
Descripción: El sistema debe permitir la invocación de operaciones en el servidor sin esperar respuesta del mismo.
Observaciones:

Tabla 39 Historias de Usuarios Retornar la invocación asíncrona mediante Result CALLBACK.

Historia de Usuario: Retornar la invocación asíncrona mediante *Result CALLBACK*.	
Número: 7	**Usuario:** Programador
Nombre historia: Retornar la invocación asíncrona mediante *Result CALLBACK*.	
Puntos estimados:1	**Puntos reales:**1.5
Prioridad en negocio: Alta	**Riesgo en desarrollo:** Alto
Programador responsable: Alberto Marturelo Lorenzo	
Descripción: El sistema debe permitir la invocación de operaciones en el servidor continuar la ejecución y permitir el aviso de la respuesta mediante *Result CALLBACK*.	
Observaciones:	

Tabla 40Historias de Usuarios Retornar la invocación asíncrona mediante Poll Object.

Historia de Usuario: Retornar la invocación asíncrona mediante *Poll Object*.	
Número: 8	**Usuario:** Programador
Nombre historia: Retornar la invocación asíncrona mediante *Poll Object*.	
Puntos estimados:1	**Puntos reales:**1.5
Prioridad en negocio: Alta	**Riesgo en desarrollo:** Alto
Programador responsable: Alberto Marturelo Lorenzo	
Descripción: El sistema debe permitir la invocación de operaciones en el servidor continuar la ejecución y permitir el aviso de la respuesta mediante *Poll Object*.	
Observaciones:	

Tabla 41 Historias de Usuarios Retornar la invocación asíncrona mediante SyncWithServer.

Historia de Usuario: Retornar la invocación asíncrona mediante *SyncWithServer*.	
Número: 9	**Usuario:** Programador
Nombre historia: Retornar la invocación asíncrona mediante *SyncWithServer*.	
Puntos estimados:1	**Puntos reales:**1.5
Prioridad en negocio: Alta	**Riesgo en desarrollo:** Alto

Programador responsable: Alberto Marturelo Lorenzo
Descripción: El sistema debe permitir la invocación de operaciones en el servidor continuar la ejecución y permitir el aviso de la respuesta mediante *SyncWithServer*.
Observaciones:

Tabla 42 Historias de Usuarios Creación de objetos de forma estática en el servidor.

Historia de Usuario: Creación de objetos de forma estática en el servidor.	
Número: 10	**Usuario:** Programador
Nombre historia: Creación de objetos de forma estática en el servidor.	
Puntos estimados:1	**Puntos reales:**1.5
Prioridad en negocio: Alta	**Riesgo en desarrollo:** Alto
Programador responsable: Alberto Marturelo Lorenzo	
Descripción: El sistema debe permitir la creación de objetos estáticos en el servidor.	
Observaciones: Los objetos estáticos se crearan una vez que el servidor sea activado y los mismos dejaran de estar publicados para la ejecución de sus métodos cuando el servidor se finalice.	

Tabla 43 Historias de Usuarios Creación de objetos en el momento que el cliente solicite.

Historia de Usuario: Creación de objetos en el momento que el cliente solicite.	
Número: 11	**Usuario:** Programador
Nombre historia: Creación de objetos en el momento que el cliente solicite.	
Puntos estimados:1	**Puntos reales:**1.5
Prioridad en negocio: Alta	**Riesgo en desarrollo:** Alto
Programador responsable: Alberto Marturelo Lorenzo	
Descripción: El sistema debe permitir la creación de objetos en el momento que el cliente solicite la invocación de un método en dicho objeto.	
Observaciones: Los objetos requeridos por el cliente se instanciarán en el momento en el que se requiera una invocación en el mismo.	

Anexo # 2 Tarjetas CRC

Tarjetas CRC Event Notifier

Tabla 44 Tarjeta CRC "Subscriber"

Tarjeta CRC: Subscriber.	
Clase: Subscriber	
Responsabilidades: Clase abstracta que posee la responsabilidad de darle el tratamiento requerido al Evento ocurrido mediante el método *inform()*.	Colaboradores:

Tabla 45 Tarjeta CRC "Filter"

Tarjeta CRC: Filter.	
Clase: Filter.	
Responsabilidades: Clase abstracta que posee la responsabilidad de en dependencia del filtro si el evento debe ser informado o no a sus subscriptores mediante el método apply.	Colaboradores: Event

Tabla 46 Tarjeta CRC "Event"

Tarjeta CRC: Event.	
Clase: Event.	
Responsabilidades: Clase abstracta que de la cual tienen q implementar los futuros nuevos eventos con sus atributos.	Colaboradores:

Tabla 47 Tarjeta CRC "ForwarderProxy"

Tarjeta CRC: ForwarderProxy.	
Clase: ForwarderProxy	
Responsabilidades: Clase encargada de establecer un proxy entre el servidor y un cliente de publicación.	Colaboradores:

Tabla 48 Tarjeta CRC "Forwarders"

Tarjeta CRC: Forwarders.	
Clase: Forwarders	
Responsabilidades: Clase encargada de tener todos los proxy de todos los clientes de publicación.	Colaboradores: Forwarder

Tarjetas CRC Object Remote *Broker*

Tabla 49 Tarjeta CRC "ClientProxy"

Tarjeta CRC: ClientProxy.	
Clase: ClientProxy.	
Responsabilidades: Clase con la responsabilidad de crear una interfaz con los métodos que se deseen invocar en el servidor.	Colaboradores: Interface

Tabla 50 Tarjeta CRC "Interface"

Tarjeta CRC: Interface.	
Clase: Interface.	
Responsabilidades: Clase interfaz que posee todos los métodos intocables en el servidor.	Colaboradores:

Tabla 51Tarjeta CRC "Forwarders"

Tarjeta CRC: Forwarders.	
Clase: Forwarders.	
Responsabilidades: Clase encargada de tener todos los proxy de todos los clientes de publicación.	Colaboradores: Forwarder

Tabla 52 Tarjeta CRC "Invoker"

Tarjeta CRC: Invoker.	
Clase: Invoker.	
Responsabilidades: Clase con la responsabilidad de realizar las invocaciones en el objeto especificado por el cliente.	Colaboradores: LifecycleManagerRegistry

Tabla 53 Tarjeta CRC "LifecycleManagerRegistry"

Tarjeta CRC: LifecycleManagerRegistry.	
Clase: LifecycleManagerRegistry	
Responsabilidades: Clase que tiene la responsabilidad de albergar los tiempos de LifeCycleManage que existan, encargada de devolver el LifeCycleManage asociado al identificador del objeto ingresado.	Colaboradores: LifecycleManager PerRequestLifecycleManager StaticInstanceLifecycleManager

Tabla 54 Tarjeta CRC "Marshaller"

Tarjeta CRC: Marshaller.	
Clase: Marshaller	
Responsabilidades: Clase encargada de la serialización y de serialización de los diferentes tipo de invocaciones.	Colaboradores: InvocationData InvocationResult

Tabla 55 Tarjeta CRC "AsyncRequestor"

Tarjeta CRC: AsyncRequestor.	
Clase: AsyncRequestor	
Responsabilidades: Clase encargada de contener los métodos encargados de las invocaciones asíncronas.	Colaboradores: FireAndForgetHandler AsyncInvocationHandler

Tabla 56 Tarjeta CRC "SyncRequestor"

Tarjeta CRC: SyncRequestor.	
Clase: SyncRequestor.	
Responsabilidades: Clase encargada de contener los métodos encargados de las invocaciones síncronas.	Colaboradores:

Tabla 57 Tarjeta CRC "InvocationData"

Tarjeta CRC: InvocationData.	
Clase: InvocationData.	
Responsabilidades: Clase encargada de contener los datos necesarios para realizar una invocación, como ID del objeto remoto, el	Colaboradores:

nombre de la operación, los parámetros y el tipo de retorno.	

Tabla 58 Tarjeta CRC "Message"

Tarjeta CRC: Message.	
Clase: Message.	
Responsabilidades: Clase encargada de transportar los datos del cliente al servidor y viceversa.	Colaboradores:

Tabla 59 Tarjeta CRC "AsyncHandler"

Tarjeta CRC: AsyncHandler.	
Clase: AsyncHandler	
Responsabilidades: Clase interfaz de la cual heredan los diferentes tipo de retorno, SyncWithServer, PollObject y ResultCallBack.	Colaboradores:

Tabla 60 Tarjeta CRC "AsynsInvocationHandler"

Tarjeta CRC: AsyncInvocationHandler.	
Clase: AsyncInvocationHandler	
Responsabilidades: Clase encargada de realizar invocaciones asíncronas.	Colaboradores: AsyncHandler InvocationData ClientRequestHandler

Tabla 61 Tarjeta CRC "ClientInvocationHandler"

Tarjeta CRC: ClientInvocationHandler.	
Clase: ClientInvocationHandler.	
Responsabilidades: Clase encargada de construir la invocación en dependencia del tipo, clase interfaz de la que heredan todos los tipo de invocaciones ya sea síncrona, asíncrona y dispara y olvida (Fire and forget).	Colaboradores: ClientRequestHandler InvocationData

Tabla 62 Tarjeta CRC "ClientRequestHandler"

Tarjeta CRC: ClientRequestHandler.	
Clase: ClientRequestHandler.	
Responsabilidades: Clase encargada de enviar con el uso de ZMQ os diferentes mensajes al servidor y recibir las respectivas respuestas.	Colaboradores: OutputManager Request

Tabla 63 Tarjeta CRC "FireAndForgetHandler"

Tarjeta CRC: FireAndForgetHandler.	
Clase: FireAndForgetHandler.	
Responsabilidades: Clase encargada de realizar una invocación asíncrona de modo dispara y olvida, no esperando confirmación de ejecución ni respuesta.	Colaboradores: InvocationData

Tabla 64 Tarjeta CRC "OutputManager"

Tarjeta CRC: OutputManager.	
Clase: OutputManager	
Responsabilidades: Clase con la responsabilidad de funcionar como bandeja de entrada y salida, enviando y dando respuesta a las invocaciones llegadas desde el servidor.	Colaboradores: Request Response Message

Tabla 65 Tarjeta CRC "PollObject"

Tarjeta CRC: PollObject.	
Clase: PollObject.	
Responsabilidades: Clase encargada de almacenar la respuesta llegada desde el servidor.	Colaboradores:

Tabla 66 Tarjeta CRC "Request"

Tarjeta CRC: Request.
Clase: Request.

Responsabilidades: Clase encargada de controlar el tiempo en que el mensaje fue enviado, las veces que ha sido enviado sin respuesta.	Colaboradores:

Tabla 67 Tarjeta CRC "Response"

Tarjeta CRC: Response.	
Clase: Response.	
Responsabilidades: Clase encargada de controlar la llegada de la respuesta a la solicitud realizada.	Colaboradores: InvocationResult

Tabla 68 Tarjeta CRC "ResultCallback"

Tarjeta CRC: ResultCallback	
Clase: ResultCallback.	
Responsabilidades: Clase encargada de notificar la llegada de la respuesta esperada.	Colaboradores:

Tabla 69 Tarjeta CRC "SyncInvocationHandler"

Tarjeta CRC: SyncInvocationHandler.	
Clase: SyncInvocationHandler.	
Responsabilidades: Clase encargada de realizar una invocación síncrona al servidor.	Colaboradores: ClientRequestHandler

Tabla 70 Tarjeta CRC "SyncWithServer"

Tarjeta CRC: SyncWithServer.	
Clase: SyncWithServer.	
Responsabilidades: Clase encargada de notificar que la invocación llego al servidor con éxito.	Colaboradores:

Tabla 71 Tarjeta CRC "ServerRequestHandler"

Tarjeta CRC: ServerRequestHandler.
Clase: ServerRequestHandler.

Responsabilidades: Clase encargada de recibir las peticiones y enviar las respectivas repuestas a los clientes específicos.	Colaboradores:

Anexo # 3 Casos de prueba

Tabla 72 Prueba 2 de la HU "Suscribirse a un evento remotamente."

Caso de prueba de aceptación	
Código: HU2_p2	**Historia de usuario:** 2
Nombre: Suscribirse a un evento remotamente.	
Descripción: En este caso de prueba se pretende comprobar que la funcionalidad Suscribirse evento remoto.	
Condiciones de ejecución: El evento al que se va a subscribir, el filtro que debe cumplir el evento y el subscriptor para el tratamiento de dicha publicación tienen que haberse implementado con anterioridad.	
Entrada/Paso de ejecución: Se crea una instancia de la clase RemoteEventService, la cual crea una instancia del socket de subscripción, luego se crea una instancia de la clase RemoteSubscriber que tiene un método que se llama Subscribe se le pasa por parámetro el evento, el filtro y el subscriptorel cual tiene el tratamiento para la aparición del evento.	
Resultado esperado: Se espera que el cliente reciba la aparición al evento al cual se suscribió.	
Evaluación de la prueba: Satisfactoria.	

Tabla 73 Prueba 3 de la HU "Invocación síncrona de operaciones en el objeto remoto."

Caso de prueba de aceptación	
Código: HU3_p3	**Historia de usuario:** 3
Nombre: Invocación síncrona de operaciones en el objeto remoto.	
Descripción: En este caso de prueba se pretende comprobar que la funcionalidad Invocación síncrona de operación en un objeto remoto en el servidor.	
Condiciones de ejecución: El objeto al cual se le va a efectuar la invocación en el servidor tiene que antes haberse declarado por cualquier tipo de ciclo de vida, para poder ejecutar operaciones en el mismo.	

Entrada/Paso de ejecución: El Client Proxy crea una instancia de la clases *SyncRequestor*, mediante el cual utilizando el método *invoke* y pasándole por parámetros los atributos necesarios para hacer una invocación la realiza, y espera la respuesta síncronamente.
Resultado esperado: Se espera el bloqueo permanente del sistema hasta que la petición no se haya respondido o hasta que no retorne una excepción.
Evaluación de la prueba: Satisfactoria.

Tabla 74 Prueba 4 de la HU "Invocación asíncrona de operaciones en el objeto remoto."

Caso de prueba de aceptación	
Código: HU4_p4	**Historia de usuario:** 4
Nombre: Invocación asíncrona de operaciones en el objeto remoto.	
Descripción: En este caso de prueba se pretende comprobar que la funcionalidad Invocación asíncrona de operación en un objeto remoto en el servidor.	
Condiciones de ejecución: El objeto al cual se le va a efectuar la invocación en el servidor tiene que antes haberse declarado por cualquier tipo de ciclo de vida, para poder ejecutar operaciones en el mismo.	
Entrada/Paso de ejecución: El Client Proxy crea una instancia de la clases As*yncRequestor*, mediante el cual utilizando el método *invoke* y pasándole por parámetros los atributos necesarios para hacer una invocación la realiza, y espera la respuesta asíncronamente.	
Resultado esperado: Se espera mediante el uso de los diferentes tipos de retornos la respuesta del servidor.	
Evaluación de la prueba: Satisfactoria.	

Tabla 75 Prueba 5 de la HU "Invocación asíncrona de operaciones en el objeto remoto sin remoto."

Caso de prueba de aceptación	
Código: HU5_p5	**Historia de usuario:** 5
Nombre: Invocación asíncrona de operaciones en el objeto remoto sin retorno.	
Descripción: En este caso de prueba se pretende comprobar que la funcionalidad Invocación asíncrona de operación sin retorno en un objeto remoto en el servidor.	
Condiciones de ejecución: El objeto al cual se le va a efectuar la invocación en el servidor tiene que antes haberse declarado por cualquier tipo de ciclo de vida, para poder ejecutar operaciones en el mismo.	

Entrada/Paso de ejecución: El Client Proxy crea una instancia de la clases As*yncRequestor*, mediante el cual utilizando el método *fireAndForget* y pasándole por parámetros los atributos necesarios para hacer una invocación la realiza sin esperar respuesta del servidor.
Resultado esperado:
Evaluación de la prueba: Satisfactoria.

Tabla 76 Prueba 6 de la HU "Creación de objetos de forma estática en el servidor."

Caso de prueba de aceptación	
Código: HU9_p6	**Historia de usuario:** 9
Nombre: Creación de objetos de forma estática en el servidor.	
Descripción: En este caso de prueba se pretende comprobar la funcionalidad de la creación de un objeto remoto de forma estática en el servidor.	
Condiciones de ejecución: Debe de estar implementada la clase (Objeto remoto) que se desea registrar en el ciclo de vida.	
Entrada/Paso de ejecución:	
Resultado esperado:	
Evaluación de la prueba: Satisfactoria.	

Tabla 77 Prueba 7 de la HU "Creación de objetos en el momento que el cliente solicite."

Caso de prueba de aceptación	
Código: HU10_p7	**Historia de usuario:** 10
Nombre: Creación de objetos en el momento que el cliente solicite.	
Descripción: En este caso de prueba se pretende comprobar la funcionalidad de la creación de un objeto remoto en el momento que lo requiera un solicitante.	
Condiciones de ejecución:	
Entrada/Paso de ejecución:	
Resultado esperado:	
Evaluación de la prueba: Satisfactoria.	

Tabla 78 Prueba 8 de la HU "Eliminar subscripción de un evento remoto."

Caso de prueba de aceptación	
Código: HU11_p8	**Historia de usuario:** 11
Nombre: Eliminar subscripción de un evento remoto.	

Descripción: En este caso de prueba se pretende comprobar la funcionalidad de eliminar una subscripción de un evento en un objeto remoto.
Condiciones de ejecución:
Entrada/Paso de ejecución:
Resultado esperado:
Evaluación de la prueba: Satisfactoria.