ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN
UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

PROTOCOLO DE COMUNICACIONES PARA ROBOTS DE SERVICIOS BASADO EN SOCKETS

AUTOR: Adolfo José Martínez Lamberto

DIRECTOR(ES): Juan Ángel Pastor Franco

Septiembre / 2002

Autor	Adolfo José Martínez Lamberto
E-mail del Autor	mlamberto@terra.es
Director(es)	Juan Ángel Pastor Franco
E-mail del Director	juanangel.pastor@upct.es
Codirector(es)	
Título del PFC	Protocolo de comunicaciones para robots de servicios basado en sockets
Descriptores	Sockets, Cliente/Servidor, Comunicaciones

Resúmen

Desarrollo de un sistema de comunicaciones cliente/servidor que opere con el robot Goya que consta de un sistema robotizado sobre el cual se monta un cabezal de limpieza y que es teleoperado para poder recorrer casi toda la superficie del barco (aproximadamente un 90%). La plataforma de teleoperación provee al usuario final de todos los servicios necesarios para realizar la operación de limpieza, dicha plataforma se comunica vía TCP con la unidad de control, la cual es la encargada de controlar los movimientos de los diversos accionadores que constituyen el conjunto robot-cabezal de limpieza, así como, de registrar el modo de operación y estados en los que se puede encontrar el mismo. El protocolo de comunicación a través de la red estará basado en los servicios provistos por TCP/IP utilizando el mecanismo de comunicación mediante sockets como vía de acceso a dichos servicios.

Titulación	Ingeniero Técnico de Telecomunicación, esp. Telemática
Intensificación	
Departamento	Departamento de Tecnologías de la Información y las Comunicaciones
Fecha de Presentación	Septiembre - 2002

ÍNDICE

1. El Proyecto Goya. 5
1.1. El robot Goya. 5
1.2. Objetivos. 5
2. Estado del arte. 8
2.1. Introducción a los Sistemas Distribuidos. 8
2.1.1. La computación distribuida. 9
2.1.2. Sistemas distribuidos. 10
2.1.3. El modelo de objeto distribuido de Java. 10
2.1.4. Enfoques para la construcción de aplicaciones distribuidas en Java. 11
2.2. Aplicaciones Cliente/Servidor. 12
2.2.1. Introducción. 12
2.2.2. Antecedentes. 13
2.2.3. La Arquitectura Cliente/Servidor. 14
2.2.3.1. Características. 15
2.2.3.2. Componentes. 16
2.2.3.3. Relación entre componentes y recursos de la arquitectura Cliente/Servidor. 17
2.2.3.4. Modelos de la Arquitectura Cliente/Servidor. 20
2.2.3.5. Servidores orientados a conexión frente a servidores sin conexión. 21
2.2.3.6. Ventajas e inconvenientes de la Arquitectura Cliente/Servidor. 22
2.3. Comunicación basada en sockets. 24
2.3.1. Introducción. 24
2.3.2. Tipos de sockets. 25
2.3.3. Sockets Java. 27
2.3.4. Streams. 28
3. Concurrencia en una red. 29
3.1. El problema del servidor de sockets. 29
3.2. Servidor iterativo. 29
3.3. Servidor concurrente. 30
3.3.1. Algoritmo de Servidor Concurrente 30
3.3.2. Servidor Concurrente en Java. 31
4. Patrones. 32
4.1. Introducción a los patrones. 33
4.2. Definiciones. 33

4.3. Patrones de software. 34
4.3.1. Características de los patrones software. 34
4.3.2. Tipos de patrones software. 35
4.3.2.1. Patrones de diseño. 35
4.3.2.2. Clasificación de los patrones de diseño. 36
4.4. Patrón Observador. 37
4.4.1. Estructura. 38
4.4.2. Implementación. 39
4.4.3. Consecuencias de su uso. 40
4.4.4. Patrones relacionados. 40
4.4.5. Implementación en la aplicación Goya. 41
4.5. Patrón Proxy. 42
4.5.1. Implementación. 43
4.5.2. Estructura. 44
4.5.3. Componentes. 44
4.5.4. Diagrama. 45
4.5.5. Aplicaciones. 45
4.5.6. Consecuencias de su uso. 46
4.5.7. Implementación en la aplicación Goya. 46
5. UML. 48
5.1. Orígenes. 48
5.2. Un poco de historia. 48
5.3. ¿Qué es UML? 49
5.4. Metas de UML. 50
5.5. Vistas de un sistema. 50
5.5.1. Vista de Casos de Uso. 50
5.5.2. Vista Lógica. 52
5.5.3. Vista de Componentes. 54
5.5.4. Vista de Implantación. 56
5.5.5. Vista de Concurrencia. 58
6. Modelado del sistema mediante UML. 54
6.1. Vista de Casos de Uso. 54
6.2. Vista Lógica. 62
6.3. Vista de Implantación. 71
6.4. Vista de Concurrencia. 73
6.5. Vista de Componentes. 81
7. Arquitectura de la aplicación. 82

7.1. Organización de la aplicación. 82
7.2. Descripción de la aplicación. 84
7.3. Dependencias significativas. 104
8. Etapas de creación de la aplicación. 110
9. Aspectos característicos de la implementación. 113
9.1. Diseño de la arquitectura cliente/servidor. 113
9.2. Creación de los enlaces de comunicaciones. 115
9.3. Creación de la arquitectura de la aplicación. 117
9.4. Soporte multicliente. 122
9.5. Envío y recepción de mensajes. 126
9.6. Servicio de registro de los servidores. 129
9.6.1. Registro mediante notificación de eventos. 129
9.6.2. Registro mediante notificación periódica. 133
9.7. Lista de clientes registrados. 138
9.8. Comunicación asíncrona vs. síncrona. 140
9.9. Cierre de las comunicaciones. 143
9.9.1. Cierre de un cliente. 143
9.9.2. Cierre de la aplicación. 144
10. Metodología. 147
11. Manual de usuario de la aplicación. 149
11.1. Ejecución de la aplicación. 149
11.2. Extremo cliente. 150
11.3. Extremo servidor. 153
12. Documentación del código. 154
13. Conclusiones. 156
14. Trabajos futuros. 158
14.1. Servicios de seguridad. 158
14.1.1. Modelo de seguridad Java. 158
14.1.2. Control de Acceso. 160
14.2. SSL (Secure Socket Layer) con Sockets. 163
14.2.1. Análisis de las propiedades de seguridad SSL. 163
14.2.2. La suite de cifrado. 165
14.2.3. Factorías de sockets. 166
14.3. Implementación mediante middleware (RMI - CORBA). 167
15. Bibliografía. 170
16. Documentos y ficheros adjuntados al proyecto. 173

1. El Proyecto Goya.

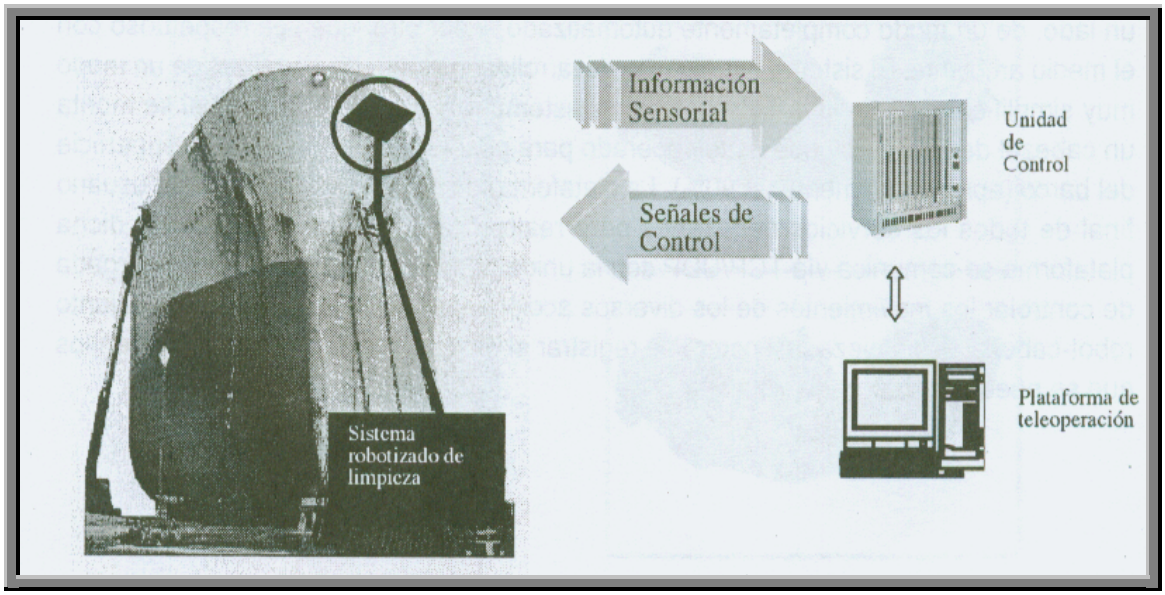
GOYA: ROBOT PARA LA LIMPIEZA DE CASCOS DE BUQUES.
Sector: Control remoto mediante sistemas cliente/servidor.
Palabras clave: Control remoto, Teleoperación, Sistemas distribuidos, Arquitectura cliente/servidor.
Objetivo: Desarrollar un protocolo de comunicaciones que permita controlar remontamente un robot que incorpore un sistema de limpieza de casos de buques, con el que se pueda trabajar con la unidad de control en modo teleoperado de forma fiable a través de una red TCP/IP (red local e incluso Internet).

1.1. El robot Goya.

El mantenimiento de los buques requiere la eliminación periódica de las adherencias marinas y la capa de pintura que recubre el casco del barco, con el objetivo de preservar la integridad del mismo, así como mantener un casco con un acabado superficial lo suficientemente suave. De este modo se consigue minimizar el consumo de combustible, reducir los costes operativos asociados al gasto del mismo, y disminuir la emisión de elementos contaminantes a la atmósfera.

Los períodos de renovación de las pinturas modernas, adheridas a los cascos de los buques, están comprendidos entre los cuatro y cinco años, realizándose alguna limpieza parcial del casco cada dos años. En estas últimas operaciones no se contempla la eliminación de la capa de pintura, tan sólo la eliminación de las adherencias marinas.

Una de las tecnologías más ampliamente difundidas para la eliminaci ón de la capa de pintura y adherencias marinas presentes en el casco del barco es el grit blasting a cielo abierto. Esta tecnología consiste en utilizar manualmente mangueras que proyectan escorias a alta velocidad mediante la inyección de aire a presión, una vez que se ha realizado la limpieza parcial mediante chorros de agua. Las principales desventajas de esta metodología son: (1) emisión de importantes cantidades de polvo, y (2) producción de elevadas cantidades de materiales de desecho.



En la actualidad, la metodología anterior está siendo parcialmente sustituida por la utilización de sistemas de chorreado con agua a presión o water-blasting. Mediante estos sistemas se consigue disminuir el impacto medioambiental del anterior método, y a su vez se evita la limpieza parcial del casco, con agua a presión, que se hace necesaria antes del grit-blasting. Sin embargo, no se consiguen tan buenas prestaciones como las obtenidas con la anterior tecnología. Los principales problemas son: (1) un acabado superficial insuficiente, y (2) un tiempo de ejecución elevado.

Todas estas circunstancias están produciendo en la práctica que los armadores, para la realización de operaciones de mantenimiento, estén acudiendo cada vez más a astilleros en los cuales los costes del servicio de limpieza son más reducidos y en los que se permite todavía realizar el grit-blasting (países del Sur y Este de Europa, Corea y China).

Adicionalmente a estos problemas que están íntimamente relacionados con la tecnología de blasting, hay que añadir el hecho de que en la actualidad no existen sistemas que operen íntegramente en modo automático. En la actualidad, la mayor parte de los astilleros utilizan para la limpieza de los cascos de barcos sistemas de blasting semiautomatizados que son manipulados muy rudimentariamente. Mediante la utilización de estos sistemas, se obtienen prolongados tiempos de ejecución para le prestación de los servicios de limpieza, así como un esfuerzo importante en horas hombre, y por consiguiente en costes de operación.

Todo ello conlleva una importante pérdida de trabajos de reparación en los astilleros del norte de Europa (donde el chorreado con arena a cielo abierto ha sido prohibido), así como, en el Sur de Europa donde los costes de producción son elevados (debidos a los costes de la hora-hombre) y en los que, en un futuro muy próximo, existe el riesgo de que la legislación medioambiental se aplique con mayor rigor, y por tanto se ponga en peligro la existencia de esta importante actividad productiva.

1.2. Objetivos.

El objetivo del presente proyecto es el desarrollar un sistema de comunicaciones cliente/servidor que opere con el robot Goya. El sistema que se esta desarrollando se puede visualizar de un modo muy simplificado en la figura. Consta de un sistema robotizado sobre el cual se monta un cabezal de limpieza y que es teleoperado para poder recorrer casi toda la superficie del barco (aproximadamente un 90%). La plataforma de teleoperación provee al usuario final de todos los servicios necesarios para realizar la operación de limpieza, dicha plataforma se comunica vía TCP con la unidad de control, la cual es la encargada de controlar los movimientos de los diversos accionadores que constituyen el conjunto robot-cabezal de limpieza, así como, de registrar el modo de operación y estados en los que se puede encontrar el mismo. El protocolo de comunicación a través de la red estará basado en los servicios provistos por TCP/IP utilizando el mecanismo de comunicación mediante sockets como vía de acceso a dichos servicios.

2. Estado del arte.

2.1. Introducción a los Sistemas Distribuidos.

Como ya es conocido por todos, el mundo de la informática se mueve hacia las aplicaciones distribuidas, y esto es debido principalmente a la migración que se está realizando desde una arquitectura en la que domina una única gran computadora hacía otra arquitectura totalmente distribuida caracterizada por las redes de estaciones de trabajo (mucho mas reducidas que la macro-computadora).

El desarrollo y abaratamiento del hardware, la ampliación de las empresas y la demanda de aplicaciones cada vez más complejas y adaptadas a la estructura de la empresa llevó a que se tuvieran que idear nuevas formas de organizar los Sistemas de Información (SI) de las empresas. A medida que el hardware se fue desarrollando, la demanda de aplicaciones de gestión automatizada de información fue creciendo. Cuando se necesitaron sistemas de información que se fueran ajustando a las necesidades de las empresas, la única solución que podían aportar los primeros sistemas, debido en gran parte al elevado coste del hardware, era una configuración en la que un único equipo o mainframe relativamente grande y adaptado a las necesidades de la empresa gestionaba todo el sistema de información.

La primera etapa de la utilización comercial de las computadoras estuvo marcada por los gigantescos macro-computadores que mimetizaban el modelo de desarrollo centralizado que se practicaba en los negocios. Los sistemas de cómputo tendían a ser homogéneos, dependientes de un solo proveedor, el cual tenía mucha influencia sobre la empresa. Todas las herramientas de desarrollo eran proporcionados por el mismo proveedor ya que los macro-computadores de diferentes marcas no eran compatibles entre si. La única ventaja de esta plataforma homogénea era que facilitaba la comunicación entre usuarios y hacía posible compartir dispositivos.

Los distintos puntos en los que se requería el acceso a la información centralizada eran conectados al gran mainframe a través de líneas de comunicación utilizando terminales cuya única funcionalidad era la de mostrar caracteres en la pantalla y enviar la información del usuario en forma de pulsaciones del teclado. Estos terminales eran mucho más baratos que el gran mainframe, y, por tanto, una empresa podía distribuir un número relativamente grande de éstos en distintos puntos estratégicos a un coste no muy elevado.

Dos causas llevaron a que esta organización fuera evolucionando: primero, el hardware se hizo cada vez más barato, por lo que en los puntos de acceso de información se podían colocar, a un menor coste, ordenadores con una cada vez más grande capacidad de

procesamiento, que quedaba ampliamente desaprovechada al utilizarlos éstos como terminales; sin embargo, la segunda y la causa más importante era la falta de flexibilidad y escalabilidad de la solución basada en mainframe.

En primer lugar, todo el código del sistema junto con sus datos residía en el ordenador principal de la empresa. Esto hacía que, para empresas medianamente grandes, el sistema de información fuera un largo y, a la vez, en muchos casos, incomprensible programa al que los programadores se tenían que enfrentar a la hora de realizar alguna modificación, actualización, etc. En segundo lugar, el sistema quedaba muy poco escalable, y esto era por varias razones: todos los terminales se conectaban a un mismo ordenador, quedando éste limitado incluso por el número de interfaces físicos para conexión de terminales de que dispusiera; todos los terminales requerían del servicio del mainframe de forma más o menos simultánea, lo que hacía que, al ir añadiendo más terminales, el servicio de las peticiones de cada una de ellas se hacía más lento.

2.1.1. La computación distribuida.

Con el progreso de la electrónica, tanto el tamaño y co sto de las computadoras fue reduciéndose. Se hizo factible que en algunas aplicaciones, como por ejemplo la generación de gráficas, se asignara una computadora para uso exclusivo de una sola persona. La velocidad de respuesta ya no dependía del número de usuarios conectados y se podía disponer de poderosas interfaces gráficas gracias a la potencia de cómputo de que disponía el usuario en su propia estación de trabajo. La tendencia de reducción del tamaño de los equipos continuó. Muy pronto fue económicamente posible asignar una computadora a cada usuario. La necesidad de compartir información y dispositivos periféricos caros, tal como se hacía en la época del macro-computador, motivó la interconexión de las computadoras mediante redes de comunicaciones. Las redes no se utilizaban para compartir funcionalidad sino únicamente dispositivos.

Hay que destacar, que el potencial de una arquitectura distribuida sólo superará al de una centralizada, si las aplicaciones cooperan entre si; pero esta cooperación tiene un gran inconveniente, y es que al estar distribuido, es muy probable que las plataformas sean de diferentes fabricantes, por lo que habrá que añadir el gran esfuerzo de programación necesario para resolver estas cuestiones de comunicaciones.

Aparecieron, en cambio, múltiples proveedores de equipos para la red y surgieron estándares para los equipos (redes, interfaces, etc.) buscando garantizar la compatibilidad. Cualquier pieza de equipo, incluyendo las computadoras personales, o estaciones de trabajo,

podía ser remplazada por otra pieza similar de otro fabricante. Por primera vez fue posible construir sistemas heterogéneos. Surge la visión de compartir también la funcionalidad de las aplicaciones, colocando la funcionalidad común a varias aplicaciones (cliente-s) en un sistema servidor. Se crea entonces el modelo Cliente/Servidor (C/S).

2.1.2. Sistemas distribuidos.

Los sistemas distribuidos son el último paso en la computación Cliente/Servidor. En vez de diferenciar entre las distintas partes de la aplicación, los sistemas distribuidos ofrecen toda la funcionalidad en forma de objetos, con un significado muy en la línea del término Objeto de la programación Orientada a Objetos. No existen los roles explícitos de cliente y servidor, sino que toda la funcionalidad está ahí para ser utilizada. Los procesos que componen la aplicación y que se están ejecutando en las distintas máquinas de la red son clientes y servidores y cooperan para conseguir la funcionalidad total de la aplicación. Esto da la máxima flexibilidad.

El mundo de los sistemas distribuidos es un mundo de entidades pares (peer-to-peer), esto es, elementos de procesamiento o nodos con distintas disponibilidades de recursos, distinta capacidad de almacenamiento, distintos requerimientos, etc., que cooperan ofreciendo servicios en forma de objetos y requiriendo otros servicios de otros objetos implementados en otros nodos de la red.

En general, un sistema distribuido es un sistema Cliente/Servidor multinivel con un número potencialmente grande de entidades que pueden desempeñar roles de clientes y servidores según sus necesidades. El hecho de ofrecer una serie de servicios en forma de objetos hace que el diseño y desarrollo se haga en base a interfaces bien definidos que facilitan y apoyan la modularidad y reutilización, a la vez que permiten un diseño mucho más flexible. Los sistemas distribuidos ofrecen, por lo general, un conjunto de servicios añadidos, como el servicio de directorio, que permite localizar servicios por nombre, gestión de transacciones, etc.

2.1.3. El modelo de objeto distribuido de Java.

El modelo de objeto distribuido que usa Java permite que los objetos que se ejecutan en una JVM invoquen a los métodos de objetos que se ejecutan en otras JVM.

Estas otras JVM pueden ejecutarse como proceso separado en la misma computadora o en otras computadoras remotas. El objeto que hace la invocación del método se denomina

objeto cliente u objeto local, mientras que el objeto cuyos métodos se están invocando se denomina objeto servidor u objeto remoto.

En el modelo de objeto distribuido de Java, un objeto cliente nunca hace referencia directa a un objeto remoto. En lugar de ello, hace referencia a una interfaz remota que implementa el objeto remoto.

El uso de interfaces remotas permite:

- Que los objetos servidor se diferencien entre sus interfaces locales y remotas.
- Que los objetos servidor presenten modos diferentes de acceso remoto.
- Que la posición del objeto servidor dentro de una clase jerárquica se abstraiga de la manera en que esta se utiliza, lo que nos permite compilar los objetos cliente por medio de la interfaz remota, sin necesidad de que los archivos de clases del servidor estén presentes localmente durante el proceso de compilación.

2.1.4. Enfoques para aplicaciones distribuidas en Java.

A la hora de construir una aplicación distribuida podemos seguir varios enfoques:

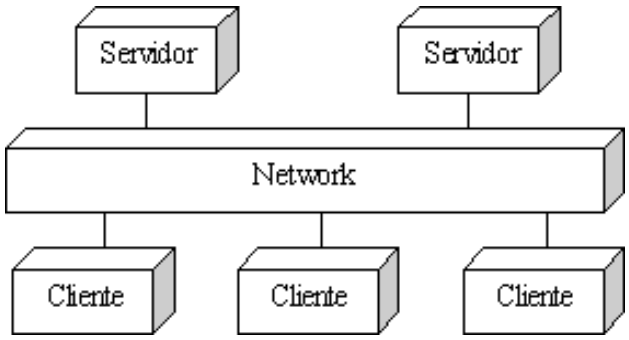
- **Sockets TCP:** Java los admite, y con ellos podrá construir aplicaciones tradicionales cliente/servidor. Aunque para ello deberá definirse un protocolo que le permita interpretar los mensajes que se envían durante la comunicación entre el cliente y el servidor.
- **Entorno de computación distribuido (DCE):** Basado en la RPC (llamadas de procedimientos remotos), constituye un enfoque orientado a procedimientos para el desarrollo de aplicaciones distribuidas. Las RPC no se acoplan bien con las aplicaciones distribuidas orientadas a objetos. El enfoque de la invocación remota de métodos que admite Corba se adapta mucho mejor al modelo de objetos de Java.
- **Modelo de objeto componente distribuido (DCOM):** Se basa en la RPC de DCE, pero ofrece posibilidades de programación orientadas a objetos a través de objetos, interfaces y métodos del COM. Además, el DCOM proporciona servicios de seguridad amplios. El entorno de desarrollo Java de Microsoft, Visual J++, ofrece la posibilidad de acceder a objetos Com y Dcom desde Java. No obstante, esta posibilidad constituye más bien un puente a las tecnologías de herencia que una extensión distribuida del modelo de objetos de Java.

- **Arquitectura de intermediación de solicitud de objetos comunes (CORBA):** Proporciona un enfoque excelente a la construcción de aplicaciones distribuidas orientadas a objetos, ya que está orientada a objetos y es una solución no vinculada a ningún sistema operativo. Java admite CORBA, sin embargo, Corba esta diseñada para admitir un modelo de objetos independiente del lenguaje. La RMI de Java posee todas las ventajas de Corba, pero está especialmente adaptada al modelo de objetos Java. Esto hace que la RMI de Java sea mucho más eficaz y fácil de usar que Corba en lo que respecta a aplicaciones de Java puro.

2.2. Aplicaciones Cliente/Servidor.

2.2.1. Introducción.

La arquitectura Cliente/Servidor es la plataforma abierta por excelencia, por la variedad de combinaciones de clientes y servidores que permite conectar en red. Sin embargo, elegir las plataformas, las herramientas, los proveedores y las bases de administración de la arquitectura Cliente/Servidor, además de la tecnología de creación, es una decisión difícil de tomar.
Por tal motivo, se ha elaborado el presente apartado con la finalidad de ofrecer una guía para el desarrollo de proyectos basados en arquitectura Cliente/Servidor.



2.2.2. Antecedentes.

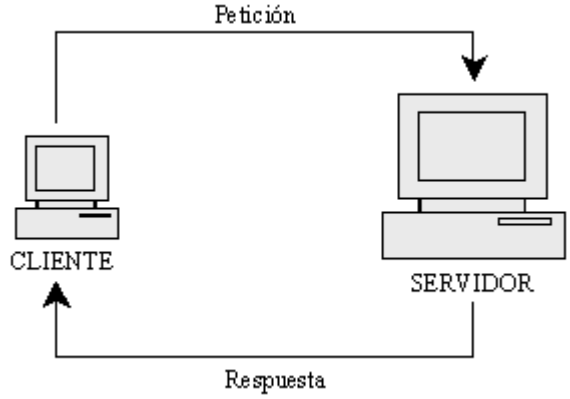
Las estaciones de trabajo individuales y los paquetes de software de aplicaciones empezaron a proliferan comercialmente. Con estas computadoras, que estaban conectadas en una red LAN, los usuarios podían compartir aplicaciones y datos. Simultáneamente, gracias a nuevas tecnologías de distribución de funciones y datos en red, se desarrollaron aplicaciones distribuidas.

A mediados de los 90's, se decía que el desarrollo de aplicaciones Cliente/Servidor era inevitable por varias razones: era más eficiente que el procesamiento centralizado, sobre todo cuando aumentaba mucho la cantidad de usuarios; ya existían en ese momento servidores razonablemente eficientes y confiables, de los cuales la mayoría había adoptado el estándar para una interfaz Cliente/Servidor. Con esto, una cantidad importante de empresas, en todo el mundo, empezaron a utilizar aplicaciones Cliente/Servidor

Hoy en día, el modelo Cliente/Servidor se considera clave para abordar las necesidades de las empresas. El proceso distribuido se reconoce actualmente como el nuevo paradigma de sistemas de información, en contraste con los sistemas independientes. Este cambio fundamental ha surgido como consecuencia de importantes factores (negocio, tecnología, proveedores), y se apoya en la existencia de una gran variedad de aplicaciones estándar y herramientas de desarrollo, fáciles de usar que soportan un entorno informático distribuido.

2.2.3. La Arquitectura Cliente/Servidor.

La arquitectura Cliente/Servidor es un modelo para el desarrollo de sistemas de información, en el que las transacciones se dividen en elementos independientes que cooperan entre sí para intercambiar información, servicios o recursos.



En esta arquitectura la computadora de cada uno de los usuarios, llamada cliente, inicia un proceso de diálogo: produce una demanda de información o solicita recursos. La computadora que responde a la demanda del cliente, se conoce como servidor. Bajo este modelo cada usuario tiene la libertad de obtener la información que requiera en un momento dado proveniente de una o varias fuentes locales o distantes y de procesarla como según le convenga. Los distintos servidores también pueden intercambiar información dentro de esta arquitectura.

Los clientes y los servidores pueden estar conectados a una red local o una red amplia, como la que se puede implementar en una empresa o a una red mundial como lo es la Internet. Cliente/Servidor es el modelo de interacción más común entre aplicaciones en una red. No forma parte de los conceptos de la Internet como los protocolos IP, TCP o UDP, sin embargo todos los servicios estándares de alto nivel propuestos en Internet funcionan según este modelo. Se puede decir que la arquitectura Cliente/Servidor es la integración distribuida de un sistema en red, con los recursos, medios y aplicaciones que definidos modularmente en los servidores, administran, ejecutan y atienden las solicitudes de los clientes; todos interrelacionados física y lógicamente, compartiendo datos, procesos e información; estableciendo así un enlace de comunicación transparente entre los elementos que conforman la estructura. No existe una definición específica adoptada universalmente de la Arquitectura Cliente/Servidor, las empresas de cómputo enfocan el concepto basándose en la funcionalidad que representa según los servicios que ellas mismas ofrecen.

2.2.3.1. Características.

Entre las principales características de la arquitectura Cliente/Servidor destaca que el servidor presenta a todos sus clientes una interfaz única y bien definida, y que los cambios que se realicen en el servidor implican pocos cambios en los clientes. Además, el cliente no necesitará conocer la lógica del servidor, sólo su interfaz externa; y además, no dependerá de la ubicación física del servidor, ni del tipo de equipo físico en el que se encuentra, ni de su sistema operativo.

Todos los sistemas desarrollados en arquitectura Cliente/Servidor poseen las siguientes características distintivas de otras formas de software distribuido:

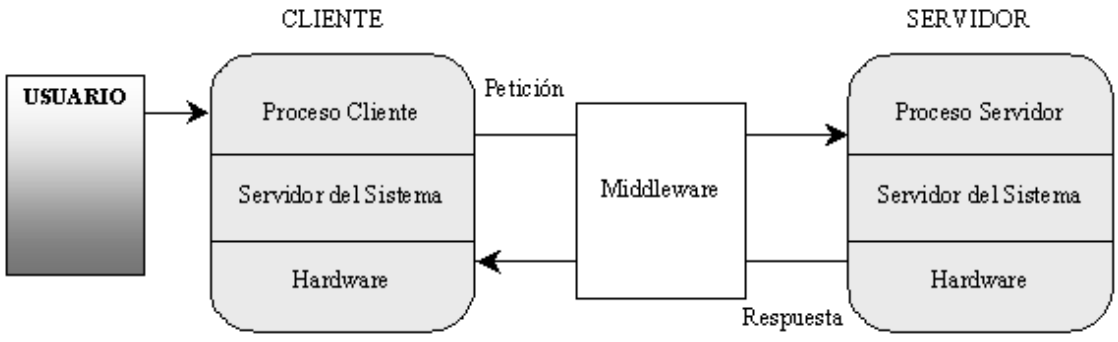
- Servicio. El servidor es un proveedor de servicios; el cliente es un consumidor de servicios.
- Recursos compartidos. Un servidor puede atender a muchos clientes al mismo tiempo y regular su acceso a recursos compartidos.
- Protocolos Asimétricos. La relación entre cliente y servidor es de muchos a uno; los clientes solicitan servicios, mientras los servidores esperan las solicitudes pasivamente.
- Transparencia de ubicación. El software Cliente/Servidor siempre oculta a los clientes la ubicación el servidor.

- Mezcla e igualdad. El software es independiente del hardware o de las plataformas de software del sistema operativo; se puede tener las mismas o diferentes plataformas de cliente y servidor.
- Intercambio basados en mensajes. Los sistemas interactúan a través de un mecanismo de transmisión de mensajes: la entrega de solicitudes y respuestas del servicio.
- Encapsulamiento de servicios. Los servidores pueden ser sustituidos sin afectar a los clientes, siempre y cuando la interfaz para recibir peticiones y ofrecer servicios no cambie.
- Facilidad de escalabilidad. Los sistemas Cliente/Servidor pueden escalarse horizontal o verticalmente. Es decir, se pueden adicionar o eliminar clientes (con apenas un ligero impacto en al desempeño del sistema); o bien, se puede cambiar a un servidor más grande o a servidores múltiples.
- Integridad. El código y los datos del servidor se conservan centralmente; esto implica menor costo de mantenimiento y protección de la integridad de los datos compartidos. Además, los clientes mantienen su individualidad e independencia.

La arquitectura Cliente/Servidor es una infraestructura versátil modular y basada en mensajes que pretende mejorar la portabilidad, la interoperabilidad y la escalabilidad del cómputo; además es una apertura del ramo que invita a participar a una variedad de plataformas, hardware y software del sistema.

2.2.3.2. Componentes.

Conceptualmente, los componentes de la arquitectura Cliente/Servidor son el cliente, el servidor y la infraestructura de comunicaciones (middleware), como a continuación se explica.



Cliente

El cliente es la entidad por medio de la cual un usuario solicita un servicio, realiza una petición o demanda el uso de recursos. Este elemento se encarga, básicamente, de la presentación de los datos y/o información al usuario en un ambiente gráfico. Se comunica con procesos auxiliares que se encargan de establecer conexión con el servidor, enviar el pedido, recibir la respuesta, manejar las fallas y realizar actividades de sincronización y de seguridad; además, requiere el uso de los recursos de la computadora para cualquier actividad y puede interactuar con uno o varios servidores. Los clientes se suelen situar en PC's o en estaciones de trabajo se encargan de realizar el FRONT END, que es la parte de la aplicación que interactúa con el usuario, en ellos permanecen las aplicaciones particulares de cada usuario, y realizan funciones como:

- Manejo de la interfaz del usuario.
- Captura y validación de los datos de entrada.
- Generación de consultas e informes sobre las bases de datos.

Servidor

El servidor es la entidad física que provee un servicio y devuelve resultados; ejecuta el procesamiento de datos, aplicaciones y manejo de la información o recursos. En el servidor se realiza el BACK END que es la parte destinada a recibir las solicitudes del cliente y dónde se ejecutan los procesos. La plataforma asociada con los servidores es más poderosa que la de los clientes debido a que estos primeros tienen que realizar un gran número de operaciones; así, se utilizarán PC’s poderosas, estaciones de trabajo, sistemas grandes. Por su parte los servidores realizan, entre otras, las siguientes funciones:

- Gestión de periféricos compartidos.
- Control de accesos concurrentes a bases de datos compartidas.
- Enlaces de comunicaciones con otras redes de área local o extensa.
- Siempre que un cliente requiere un servicio lo solicita al servidor correspondiente y éste, le responde proporcionándolo.

Middleware

Para que los clientes y servidores puedan comunicarse se requiere de una infraestructura lógica que proporcione los mecanismos básicos de direccionamiento y transporte. A dicha infraestructura se le denomina Middleware, el cual es un término que abarca a todo el software distribuido necesario para el soporte de interacciones entre clientes y servidores.

El middleware es un módulo intermedio que no pertenece a los dominios del servidor, ni a la interfaz de usuario, ni a la lógica de la aplicación en los dominios del cliente; tampoco debe confundirse con la red física en sí (cableado, señales de radio…). Sus funciones son:

- Independizar las dos entidades: El cliente y el servidor no necesitan saber comunicarse entre ellos, sino cómo comunicarse con el módulo de middleware.
- Traducir la información de una aplicación y pasarla a la otra: acepta consultas y datos recuperándolos de la aplicación cliente, los transmite y envía la respuesta de regreso. También genera los códigos de error.
- Controlar las comunicaciones: da a la red las características adecuadas de desempeño, confiabilidad, transparencia y administración.

Existen dos tipos de middleware:

1) El Middleware general es el sustrato de la mayoría de las interacciones de Cliente/Servidor. Incluye las pilas de comunicación, directorios distribuidos, servicios de autenticación, horario de la red, llamadas a procedimientos remotos (RPC's), y servicios en cola.

2) El Middleware de servicios específicos es necesario para cumplir un tipo particular de servicio de Cliente/Servidor; así, existe un middleware específico para los servidores dedicados: Middleware para bases de datos, middleware para objetos...

El middleware es una herramienta adecuada, que no sólo es flexible y segura, sino que también protege la inversión en tecnología y permite manejar diferentes ambientes de computación.

2.2.3.3. Relación entre componentes y recursos.

Conjuntando ambas estructuras y adicionado los recursos que serán compartidos, todos los componentes están estrechamente vinculados de tal forma que los procesos, datos y presentación de cualquier tipo de aplicación se ejecutan compartiendo los recursos del sistema en red.

Presentación Distribuida.

En este modelo, se distribuye la presentación entre el cliente y el servidor, pero éste último ejecuta todos los procesos y almacena la totalidad de los datos. Es similar a la arquitectura tradicional de un host y terminales; el cliente se utiliza solo para mejorar la

presentación desde un punto de vista estrictamente cosmético. Tiene un bajo costo de desarrollo, pero dificulta el mantenimiento del sistema y no se aprovecha la red.

Presentación Remota.

Aquí, la interfaz del usuario está completamente en el cliente, la presentación soporta la captura de datos, incluyendo una validación parcial de los mismos y una presentación de las consultas. La lógica de la aplicación y los datos está en el servidor. Con este modelo, el cliente aprovecha bien la presentación y la red; sin embargo, los procesos de la aplicación pueden ser complejos de desarrollar y si existe un alto tráfico en la red, puede ser difícil la operación de aplicaciones muy pesadas.

Proceso Distribuido.

Se da cuando la presentación está en el cliente, la base de datos está en el servidor y la lógica de la aplicación está distribuida entre el cliente y el servidor. Este modelo permite distribuir los programas del sistema al componente más apropiado, aunque es difícil de diseñar, ya que el diseñador debe definir los servicios y las interfaces de manera que los papeles de cliente y servidor sean intercambiables (excepto el control de los datos, que es responsabilidad exclusiva del servidor).

Gestión de Datos Remota.

Para este modelo, tanto la presentación como los procesos de la aplicación residen en el cliente, mientras que las bases de datos permanecen centralizadas en el servidor. Es fácil de desarrollar ya que la lógica de la aplicación no está distribuida y además, libera la carga del servidor; pero, como desventaja, la totalidad de los datos viajan a través de la red ya que no hay procesamiento en el mismo.

Bases de Datos Distribuidas.

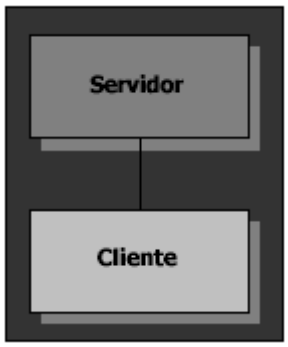
Este último modelo, la presentación, los procesos de la aplicación y parte de los datos de la Base de Datos están en el cliente; el resto de los datos están en el servidor. Esto permite que la ubicación de los datos sea transparente para la aplicación y que se pueda accesar a datos almacenados en ambientes heterogéneos, aunque éste acceso es dependiente del proveedor del software administrador de Base de Datos, el cual divide sus componentes entre el cliente y el servidor.

2.2.3.4. Modelos de la Arquitectura Cliente/Servidor.

Dado que la percepción sobre la Arquitectura Cliente/Servidor es algo abstracto, algunos autores la definen en base a modelos de dos y tres niveles de acuerdo al número de estratos respectivos para representar a sus componentes:

- **Modelos de Dos Niveles (Two Thier).**

Se considera como modelo Cliente/Servidor de dos niveles o capas a la estructura más simple, cuyos componentes son:

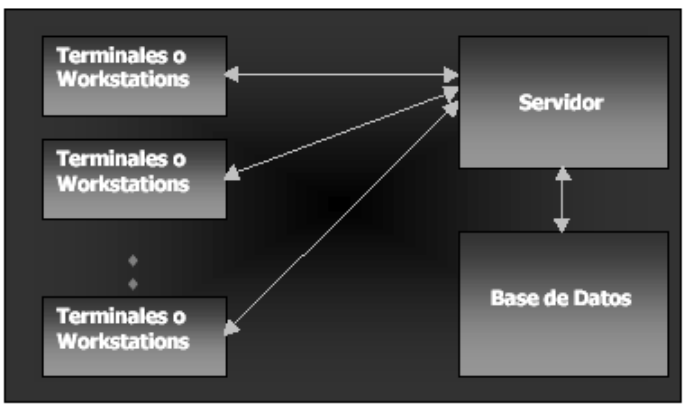


Clientes: Por medio de la interfaz con el usuario vía una petición se solicita un servicio, el uso de un recurso, o bien el acceso a un conjunto de datos.

Servidores: Satisfacen la solicitud del usuario recibiendo la petición, direccionándola y enviando la respuesta al cliente, ya sea la consulta respectiva de datos, ejecutando el proceso requerido o permitiendo el acceso y/o uso del recurso. En este modelo se acostumbra instalar las bases de datos dentro del servidor, por las ventajas de almacenamiento y velocidad que ofrece en comparación con las del cliente.

- **Modelo de Tres Niveles (Three Thier).**

El objetivo de este modelo es dividir las funciones de una aplicación en tres componentes:



Presentación: Este componente se encarga de la interacción hombre máquina a través del monitor, teclado, ratón, o bien mediante algún otro medio como reconocedor de voz.
Servidores: Compuesto por varios servidores o componentes de Software localizados en una o más plataformas que se encargará de conectar los sistemas existentes.
Información: En este componente se incluye la información en sí, los sistemas y aplicaciones existentes.

2.2.3.5. Servidores orientados a conexión frente a servidores sin conexión.

Cuando los programadores diseñan software cliente/servidor, deben escoger entre dos tipos de interacción: un estilo de poca conexión y un estilo orientado a la conexión. Los dos estilos de interacción correspondes a los principales protocolos de transporte que TCP/IP soporta. Si el cliente y el servidor se comunican usando UDP, la interacción es de poca conexión. Si se usa TCP la interacción es orientada a conexión.

Desde el punto de vista del programador, la distinción entre interacciones de poca conexión y orientadas a conexión es crítica debido a que:

- Determina el nivel de fiabilidad que proporciona el sistema.
- TCP proporciona toda la fiabilidad necesaria para comunicarse a través de Internet.
- Verifica que datos llegan y automáticamente retransmite los segmentos que no han llegado.
- Realiza comprobación de checksum sobre los datos y garantiza que no se han corrompido durante la transmisión.
- Usa secuencias de números para asegurarse de que os paquetes llegan en el orden correcto, y automáticamente elimina los duplicados.
- Proporciona un control de flujo que asegura que el emisor no envía datos demasiado rápido para que el receptor pueda consumirlos.
- Informa al cliente y al servidor si la red se vuelve inoperativa por cualquier motivo.

En contraste, los clientes y servidores que usan UDP se caracterizan por no tener garantías sobre la fiabilidad de la entrega. Cuando un cliente envía una petición, la petición puede perderse, duplicarse, retrasarse o entregarse fuera de orden. De forma similar las respuestas que retorna el servidor pueden perderse, duplicarse, retrasarse o entregarse fuera de orden. Las aplicaciones clientes o servidoras deben tomar las medidas apropiadas para detectar y corregir estos problemas.

UDP no introduce errores. Simplemente depende de la red IP subyacente para ent regar los paquetes. IP en cambio, depende del hardware y de los gateways intermedios. Desde el punto de vista del programador, la consecuencia de usar UDP es que trabaja bien si la red subyacente trabaja bien.

2.2.3.6. Ventajas e inconvenientes de la Arquitectura Cliente/Servidor.

Las condiciones que pueden aconsejar la implantación del modelo Cliente/Servidor en una organización son los cambios estructurales y organizativos, cambios en los organigramas, la respuesta a la dinámica del mercado o los cambios en los procesos del negocio. La capacidad de aproximación de los productos y servicios, a la medida de las necesidades del cliente, exige diseñarlos, producirlos y suministrarlos con rapidez y mínimos costos.

Ventajas

Las razones que impulsan el crecimiento de las aplicaciones Cliente/Servidor son:

a) Aumento de la productividad:

- Los usuarios pueden utilizar herramientas que le son familiares, como hojas de cálculo y herramientas de acceso a bases de datos.
- Mediante la integración de las aplicaciones Cliente/Servidor con las aplicaciones personales de uso habitual, los usuarios pueden construir soluciones particularizadas que se ajusten a sus necesidades cambiantes.
- Una interfaz gráfica de usuario consistente reduce el tiempo de aprendizaje de las aplicaciones.

b) Menores costos de operación:

La existencia de plataformas de hardware cada vez más baratas. Ésta constituye a su vez una de las más palpables ventajas de este esquema, la posibilidad de utilizar máquinas considerablemente más baratas que las requeridas por una solución centralizada, basada en sistemas grandes:

- Permiten un mejor aprovechamiento de los sistemas existentes, protegiendo la inversión.
- Se pueden utilizar componentes, tanto de hardware como de software, de varios fabricantes, lo cual contribuye considerablemente a la reducción de costos y favorece la flexibilidad en la implantación y actualización de soluciones.

- Proporcionan un mejor acceso a los datos. La interfaz de usuario ofrece una forma homogénea de ver el sistema, independientemente de los cambios o actualizaciones que se produzcan en él y de la ubicación de la información.
- El movimiento de funciones desde una computadora central hacia servidores o clientes locales origina el desplazamiento de los costos de ese proceso hacia máquinas más pequeñas y por tanto, más baratas.

c) Mejora en el rendimiento de la red:

- Las arquitecturas Cliente/Servidor eliminan la necesidad de mover grandes bloques de información por la red hacia las computadoras personales o estaciones de trabajo para su proceso. Todo esto reduce el tráfico de la red, lo que facilita que pueda soportar un mayor número de usuarios.
- Si se utilizan interfaces gráficas para interactuar con el usuario, el esquema cliente/Servidor presenta la ventaja, con respecto a uno centralizado, de que no es siempre necesario transmitir información gráfica por la red, pues ésta puede residir en el cliente, lo cual permite aprovechar mejor el ancho de banda de la red.
- Se pueden integrar PC's con sistemas medianos y grandes, sin que todas las máquinas tengan que utilizar el mismo sistema operativo.
- Tanto el cliente como el servidor pueden escalar para ajustarse a las necesidades de las aplicaciones. Las CPU's utilizados en los respectivos equipos, pueden dimensionarse a partir de las aplicaciones y el tiempo de respuesta que se requiera.
- La existencia de varias CPU's proporciona una red más fiable: una falla en uno de los equipos, no significa necesariamente que el sistema deje de funcionar.
- En una arquitectura como ésta, los clientes y los servidores son independientes los unos de los otros, con lo que pueden renovarse para aumentar sus funciones y capacidad de forma independiente, sin afectar al resto del sistema.
- La arquitectura modular de los sistemas Cliente/Servidor, permite el uso de computadoras especializadas (servidores de bases de datos, servidores de archivos, estaciones de trabajo, etc.).
- Permite centralizar el control de sistemas que estaban descentralizados, como por ejemplo la gestión de las computadoras personales que antes estuvieron aisladas. Es más rápido el mantenimiento y el desarrollo de aplicaciones, pues se pueden emplear las herramientas existentes.
- El esquema Cliente/Servidor contribuye además a proporcionar a las diferentes direcciones de una institución soluciones locales, pero permitiendo además la integración de la información relevante a nivel global.

Desventajas

a) Hay una alta complejidad tecnológica al tener que integrar una gran variedad de productos. Por una parte, el mantenimiento de los sistemas es más difícil, pues implica la interacción de diferente hardware y software, distribuidos por distintos proveedores, lo cual dificulta el diagnóstico de fallas.

b) Requiere un fuerte rediseño de todos los elementos involucrados en los sistemas de información (modelos de datos, procesos, interfaces, comunicaciones, almacenamiento de datos, etc.). Además, en la actualidad existen pocas herramientas que ayuden a determinar la mejor forma de dividir las aplicaciones entre la parte cliente y la parte servidor. Por otra parte, es importante que los clientes y los servidores utilicen el mismo mecanismo, lo cual implica que deben tenerse mecanismos generales que existan en diferentes plataformas.

c) Escasas herramientas para la administración y ajuste del desempeño de los sistemas.

d) Es más difícil asegurar un elevado grado de seguridad en una red de clientes y servidores que en un sistema con una sola computadora centralizada. Deben hacerse verificaciones en el cliente y en el servidor.

e) A veces, los problemas de congestión de la red pueden degradar el rendimiento del sistema por debajo de lo que se obtendría con una única máquina (arquitectura centralizada). También la interfaz gráfica de usuario puede a veces ralentizar el funcionamiento de la aplicación.

f) El quinto nivel de esta arquitectura (bases de datos distribuidas) es técnicamente muy complejo y en la actualidad, hay muy pocas implantaciones que garanticen un funcionamiento totalmente eficiente.

g) Existen multitud de costos ocultos (formación en nuevas tecnologías, licencias, cambios organizativos, etc.) que encarecen su implantación.

El cambio hacia Cliente/Servidor no implica solamente cambios técnicos y de productos sino también un cambio cultural y organizacional. Pues, para que un sistema se realice con éxito, debe anteceder el soporte gerencial y la aprobación por parte de la dirección; por lo tanto, también debe culturizarse a los usuarios para que una vez implantado, conciban al cambio como un beneficio colectivo. La decisión dependerá de las necesidades, requerimientos y capacidad económica de la empresa.

2.3. Comunicación basada en sockets.

2.3.1. Introducción.

Los Sockets son una innovación del sistema UNIX de Berkeley y permiten al programador trabajar con un canal de comunicación de red como si fuera un caudal de bytes que puede leerse y al que puede escribirse, ocultando los detalles de bajo nivel al programador.
La comunicación con sockets sigue el modelo Cliente-Servidor y en la mayoría de los casos un programa servidor fundamentalmente envía datos, mientras que un programa cliente recibe esos datos, aunque es raro que un programa exclusivamente reciba o envíe datos. Una distinción confiable se logra si consideramos cliente al programa que inicia la comunicación y servidor al programa que espera a que algún otro inicie comunicación con él.

La comunicación con sockets se hace mediante un protocolo de la familia TCP/IP en la mayoría de los casos, y es el programador quien decide qué protocolo utilizar dependiendo de las necesidades de la aplicación que desarrolla.

En general los dos protocolos de la familia TCP/IP más utilizados son TCP y UDP, el primero es orientado a conexión y confiable, mientras que el segundo no lo es, pero ofrece otras ventajas como rapidez y facilidad de recibir datos de varios anfitriones usando un solo socket.

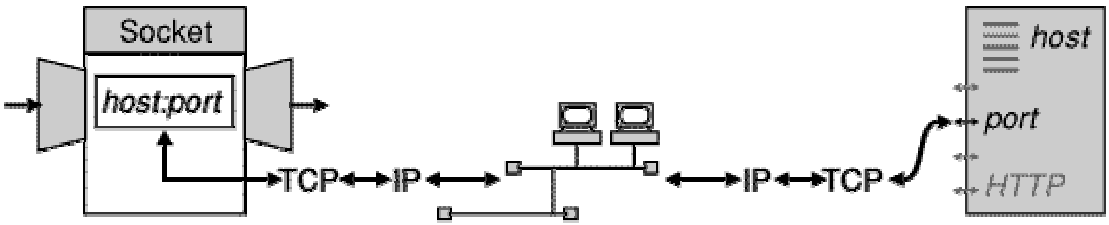
Si ahora pensamos en terminología de Java, toda comunicación entre dos procesos, entiéndase aplicaciones, deberá abrir un canal de comunicaciones (socket), realizar las operaciones de lectura y escritura en el canal y tras esto, cerrar dicho canal.

Como acabamos de comentar, la comunicación en Java se basa en la creación y uso de sockets. Un socket es un punto final en un enlace de comunicación de dos vías entre dos programas que se ejecutan en la red. Las clases Socket son utilizadas para representar conexiones entre un programa cliente y otro programa servidor. El paquete java.net proporciona dos clases -- Socket y ServerSocket -- que implementan los lados del cliente y del servidor de una conexión, respectivamente.

Una aplicación servidor normalmente escucha a un puerto específico esperando una petición de conexión de un cliente. Cuando llega una petición de conexión, el cliente y el servidor establecen una conexión dedicada sobre la que poder comunicarse. Durante el proceso de conexión, el cliente es asignado a un número de puerto, y ata un socket a ella. El cliente habla al servidor escribiendo sobre el socket y obtiene información del servidor cuando lee de él. Similarmente, el servidor obtiene un nuevo número de puerto local (necesita un nuevo puerto

para poder continuar escuchando para petición de conexión del puerto original.) El servidor también ata un socket a este puerto local y comunica con él mediante la lectura y escritura sobre él. El cliente y el servidor deben ponerse de acuerdo sobre el protocolo -- esto es, debe ponerse de acuerdo en el lenguaje para transferir la información de vuelta a través del socket.

La clase Socket del paquete java.net es una implementación independiente de la plataforma de un cliente para un enlace de comunicación de dos vías entre un cliente y un servidor. La clase Socket se sitúa en la parte superior de una implementación dependiente de la plataforma, ocultando los detalles de los sistemas particulares a un programa Java. Utilizando la clase java.net.Socket en lugar de tratar con código nativo, los programas Java pueden comunicarse a través de la red de una forma independiente de la plataforma.



2.3.2. Tipos de sockets.

A continuación se comentaran las diferentes formas de trabajar con sockets según se transfiera la información a través del canal de comunicaciones.

Sockets TCP:

Basan su forma de trabajo en el protocolo de comunicaciones Transfer Control Protocol y proporcionan un servicio orientado a la conexión con las características de proporcionar una comunicación que nos avisará si se producen errores en la transmisión.

La forma de trabajar es: Crear una conexión entre los dos extremos de interés y transmitir por el canal datos. Un extremo es el que atiende a peticiones de conexión y actúa como servidor mientras que el otro realiza las peticiones y está actuando como cliente. Una vez se haya establecido la conexión se garantiza un flujo ordenado de datos en ambas direcciones.

Sockets Datagrama:

Basan su forma de trabajo en el protocolo de comunicaciones User Datagram Protocol (UDP) que no es orientado a la conexión y que se caracteriza por un envío de los datos en

paquetes, los cuales pueden perderse, duplicarse o llegar desordenados. Al ser un protocolo no orientado a la conexión será necesario implementar un mecanismo que secuenciamiento y control de la transmisión.

Sockets Raw:

Es un tipo particular de sockets que no vamos a ver y que sirven para depurar los protocolos de comunicaciones ya que nos permiten entrar en protocolos de más bajo nivel respecto del que estamos tratando.

Vistos los tipos de sockets y dadas sus características, nos inclinaremos por un tipo u otro en función de nuestras necesidades. La elección tendrá su justificación:

En UDP, cada vez que se envía un datagrama, hay que enviar también el descriptor del socket local y la dirección del socket que va a recibir el datagrama, luego éstos son más grandes que los TCP. Como el protocolo TCP está orientado a conexión, tenemos que establecer esta conexión entre los dos sockets antes de nada, lo que implica un cierto tiempo empleado en el establecimiento de la conexión, que no existe en UDP.

En UDP hay un límite de tamaño de los datagramas, establecido en 64 kilobytes, que se pueden enviar a una localización determinada, mientras que TCP no tiene límite; una vez que se ha establecido la conexión, el par de sockets funciona como los streams: todos los datos se leen inmediatamente, en el mismo orden en que se van recibiendo.
UDP es un protocolo desordenado, no garantiza que los datagramas que se hayan enviado sean recibidos en el mismo orden por el socket de recepción. Al contrario, TCP es un protocolo ordenado, garantiza que todos los paquetes que se envíen serán recibidos en el socket destino en el mismo orden en que se han enviado.

Los datagramas son bloques de información del tipo lanzar y olvidar. Para la mayoría de los programas que utilicen la red, el usar un flujo TCP en vez de un datagrama UDP es más sencillo y hay menos posibilidades de tener problemas. Sin embargo, cuando se requiere un rendimiento óptimo, y está justificado el tiempo adicional que supone realizar la verificación de los datos, los datagramas son un mecanismo realmente útil.

En resumen, TCP parece más indicado para la implementación de servicios de red como un control remoto, mientras que UDP es menos complejo y tiene una menor sobrecarga sobre la conexión, lo que sea el indicado en la implementación de aplicaciones cliente/servidor en sistemas distribuidos montados sobre redes de área local.

La aplicación cliente/servidor que se presenta en el proyecto utiliza los sockets TCP por varias razones obtenidas como conclusión a los puntos anteriores. Aunque la aplicación en cuestión trabajará normalmente bajo una red local, el entorno de trabajo puede presentar muchos problemas en relación a los errores de transmisión por el hecho de que el servidor esté localizado en ambientes industriales donde los ruidos pueden alterar las comunicaciones. Si se usara UDP, sería responsabilidad del programador el comprobar que los paquetes llegan a su destino y que la información se transmite en orden. Aunque existen multitud de mecanismos para conseguir esto, la complejidad de su implementación hace más recomendable la utilización de un protocolo orientado a la conexión como TCP. Además de esto, TCP es un protocolo ordenado y garantiza que todos los paquetes que se envíen serán recibidos en el socket destino en el mismo orden en que se han enviado, evitando que el servidor reciba ordenes del cliente repetidas, desordenadas o erróneas.

2.3.3. Sockets Java.

En Java un socket TCP corresponde a un objeto de la clase Socket. Tanto el cliente como el servidor se comunican mediante un objeto de esta clase, uno en el cliente y otro en el servidor. Un socket realiza 7 operaciones básicas:

1. Conectarse a un anfitrión remoto
2. Enviar datos.
3. Recibir datos.
4. Cerrar la Conexión
5. Acoplarse a un puerto
6. Esperar conexiones
7. Aceptar conexiones de anfitriones remotos

La clase Socket tiene métodos que implementan las primeras 4 operaciones básicas, las últimas 3 son solo necesarias para los servidores y se implementan como métodos de la clase ServerSocket, que se discutirá mas adelante. En un programa Cliente un socket se utiliza como sigue:

1. El Cliente crea un objeto Socket
2. El socket intenta conectarse al anfitrión remoto

3. Ya conectados el cliente y el servidor envían y reciben datos

4. Al terminar la transmisión de datos uno o ambos cierran la conexión.

La clase ServerSocket proporciona al programador lo necesario para implementar programas Servidores en Java, principalmente proporciona métodos para esperar conexiones en un puerto dado y métodos que regresan un objeto Socket cuando se establece la conexión con el cliente, este socket se utiliza para intercambiar datos con el cliente, de modo que la comunicación con el cliente (que tiene su socket) se realiza por medio de otro socket del mismo tipo y no por medio del objeto ServerSocket.

A continuación se presenta el ciclo de vida de un servidor:

1. Crea nuevo objeto ServerSocket

2. El objeto ServerSocket espera conexiones en un puerto dado

3. Cuando un cliente solicita conexión, crea un objeto Socket para comunicarse con él.

4. Cliente y servidor actúan de acuerdo a un protocolo.

5. Uno o ambos cierran la conexión.

6. El Servidor regresa al paso 2 y espera la próxima conexión.

En el paso 2 la "espera" se hace con el método accept(), este bloquea al servidor hasta que un cliente solicita una conexión, cuando esto ocurre regresa un objeto Socket para comunicarse con el cliente en cuestión.

2.3.4. Streams.

Los streams sirven para enviar y recibir datos de un socket, generalmente en bytes, sin embargo el tipo de datos puede cambiar, un stream puede manejar caracteres o algún otro tipo de dato, finalmente el tipo de dato que será enviado al socket será byte. El stream es un mecanismo que permite manejar otro tipo de dato sin tener que realizar las operaciones básicas para el manejo de bytes.

En java los "streams" se explican en base al tipo de datos que manejan, el primer tipo maneja datos del tipo byte, y se definen dos clases InputStream y OutputStream, que son abstractas y de las que se derivan todos los "streams", y el segundo caso se dan el InputStreamReader y el OutputStreamWriter, éstos últimos realizan el manejo de caracteres. Y funcionan como un puente entre los "streams" para bytes y los "streams" para caracteres, puede

leer del stream para bytes y traducir su contenido a caracteres de acuerdo a la codificación en la que se estén dando los bytes. Y en el caso del OutputStreamWriter escribe los datos dados en caracteres a un stream de bytes. En java los streams pertenecen a la biblioteca java.io.

3. Concurrencia en una red.

Como ya sabemos, el modelo cliente/servidor comprende dos partes bien diferenciadas: cliente y servidor. El cliente es el que lleva la iniciativa en la comunicación, enviando sus peticiones al servidor. Éste, por el contrario, permanece a la espera de recibir las solicitudes que llegan de los clientes. Cuando recibe una solicitud la atiende y sigue esperando nuevas peticiones. El comportamiento a la hora de atender al cliente difiere según el tipo de servidor.

3.1. El problema del servidor de sockets.

El problema del servidor de sockets es uno de los problemas más habituales que se presentan en la arquitectura cliente/servidor en redes.

El problema consiste en suponer que tenemos una aplicación servidor en un host, que se encarga de resolver las peticiones que le llegan de aplicaciones cliente que se encuentran en cualquier otro host conectado a la red o en el propio host local, devolviéndoles una determinada respuesta. El problema se puede resolver de dos formas, mediante un servidor iterativo o mediante un servidor concurrente.

3.2. Servidor iterativo.

El comportamiento de este tipo de servidores es el siguiente: el servidor escucha una petición de un cliente, la procesa, responde al cliente, y vuelve a bloquearse a la espera de una nueva petición. Lógicamente mientras se procesa la petición del primer cliente, pueden llegar nuevas peticiones que deberán esperar a que el servidor finalice con la primera petición, para ser atendidas. En el sistema solo existirá una petición como máximo que se esté resolviendo. Esta solución solo se emplea cuando la petición del cliente se puede gestionar en un tiempo muy pequeño.

3.3. Servidor concurrente.

En este tipo de servidores, cuando se escucha una petición de un cliente, se crea un proceso de servicio que procesa y responde al cliente, mientras el servidor vuelve a bloquearse a la espera de una nueva petición. Esta solución es mucho más efectiva, ya que si llega una nueva petición mientras se sirve otra, el servidor crea un nuevo proceso concurrente para atender la nueva petición, con lo cual en el sistema puede haber más de una petición resolviéndose de forma concurrente. Esta es la solución más habitual, ya que normalmente no se conoce a priori la cantidad de tiempo que llevará la resolución de la petición, pues puede depender del tipo de petición. Este tipo de servidores son más complejos y consumen más recursos del sistema que los iterativos; pese a esto, se utilizan en muchas aplicaciones distribuidas.

3.3.1. Algoritmo de servidor concurrente.

El programa principal del servidor debe ser del siguiente tipo:

1- Crear un Socket de servidor

En un ciclo infinito:

2 - Aceptar requerimientos de clientes.

3 - Cuando llega una petición de un cliente crear un nuevo proceso “esclavo” que atienda paralelamente la petición (esto no debe bloquear la ejecución del programa principal del servidor).

4 - Volver a 2.

Mientras que el patrón que debe seguir el proceso “esclavo” sería el siguiente:

1 - Recibir los parámetros de la comunicación (socket o flujos de entrada y/o salida).

2 - Atender al cliente.

3 - Retornar (desaparecer).

3.3.2. Servidor concurrente en Java.

En Java se prefiere usar Threads en lugar de procesos, y que tienen las siguientes características:

- Un thread es una secuencia o flujo de de instrucciones que se ejecutan dentro de un programa. Tiene un comienzo y un fin.

- El thread sólo puede ser creado dentro de un proceso. Y un proceso (programa) puede crear varios threads dentro de él que se ejecutan en paralelo.
- El programa principal es consciente de los threads que existen, hay variables que los identifican. Pueden ser creados, inicializados, suspendidos, reactivados o parados por el programa que los creó.
- El programa principal puede darles parámetros distintos a cada thread. Los thread se pueden programar con la cantidad de variables necesarias para su ejecución (no lo heredan todo).

La forma de implementar servidores que atiendan a varios clientes paralelamente se consigue combinando a la vez threads con sockets:

- El servidor abre un ServerSocket desde donde oye cualquier intento por conectarse con él de un cliente.
- Una vez establecida la conexión, abre un socket normal e inicia un thread que atiende a este cliente. El socket abierto se pasa como parámetro, de manera que se puede seguir oyendo por el ServerSocket sin estar bloqueado.
- El thread tiene un método run que atiende los pedidos del cliente.
- El cliente se conecta al servidor sin saber que finalmente será un thread el que está atendiéndolo.

El procedimiento que se debe seguir para la creación de un servidor concurrente mediante threads debe ser similar al siguiente:

- Hacer una clase Thread que tenga como variables de un objeto socket y flujos de entrada y/o salida.
- Programar el constructor de modo que reciba como parámetro un socket y haga todo lo necesario para dejar inicializado el ambiente para empezar a atender al cliente (por ejemplo, abrir flujos de datos de entrada y/o salida del socket recibido).
- Programar el método run de modo que implemente el protocolo necesario.
- Programar un método main que en un ciclo infimito se ponga a escuchar en un puerto dado la llegada de clientes.
- Con cada cliente nuevo crear un thread nuevo y pasar como parámetro el socket.

4. Patrones.

Tanto el desarrollo como la comprensión del software es una tarea complicada, que depende en gran medida de la experiencia de los desarrolladores; y es que, a la hora del mantenimiento y evolución de las aplicaciones también se producen muchas complicaciones que no siempre son tenidas en cuenta

Continuamente se requieren sistemas más complejos y cada vez más grandes. Los recursos para desarrollarlos cada vez son más escasos. Y por tanto debe existir un mecanismo de reutilización. Las tecnologías orientadas a objetos son las más utilizadas en los últimos años para el desarrollo de aplicaciones software y se ha comprobado como estas tecnologías de programación presentan muchas ventajas. Uno de los objetivos que se buscan al utilizar la programación orientada a objetos es conseguir la reutilización, que implica por si sola características tan beneficiosas como: reducción de tiempos, disminución del esfuerzo de mantenimiento, eficiencia, consistencia, fiabilidad y prtección de la inversión en desarrollos.

Entre los diferentes mecanismos de reutilización se encuentran:

Componentes: elemento de software suficientemente pequeño para crearse y mantenerse pero suficientemente grande para poder utilizarse.
Frameworks: bibliotecas de clases preparadas para la reutilización que pueden utilizar a su vez componentes.
Objetos distribuidos: paradigma que distribuye los objetos de cooperación a través de una red heterogénea y permite que los objetos interoperen como un todo unificado.
Patrones de diseño.

Sin embargo estos mecanismos de reutilización también presentan algunos obstáculos, como son el síndrome “No Inventado Aquí” por el cual los desarrolladores de software no se suelen fiar de lo que no está supervisado por ellos mismos, el acceso a las fuentes de componentes, los impedimentos legales, comerciales o estratégicos, el formato de distribución de componentes, la inercia de cada persona a realizar cambios y el dilema entre reutilizar y rehacer.

4.1. Introducción a los patrones.

Los patrones como elemento de reutilización, comenzaron a utilizarse en la arquitectura de edificios con el objetivo de reutilizar diseños que se habían aplicado en otras construcciones y que se catalogaron como completos. En especial, fue Chistopher Alexander el primero en intentar crear un formato específico para patrones en la arquitectura, describiendo algunos diseños para tratar de conseguir sus metas (la calidad en sus trabajos). De este modo el patrón trata de extraer la esencia de ese diseño para que pueda ser utilizada por otros arquitectos cuando se enfrentan a problemas parecidos a los que resolvió dicho diseño. Alexander intenta resolver problemas arquitectónicos utilizando estos patrones. Para ello tratará de extraer la parte común de los buenos diseños (que pueden ser dispares), con el objetivo de volver a utilizarse en otros diseños.

4.2. Definiciones.

Algunos autores crean su propia definición de lo que es un patrón software:

1. Christopher Alexander:

"Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces sin hacerlo ni siquiera dos veces de la misma forma".

2. Dirk Riehle y Heinz Zullighoven:

"Un patrón es la abstracción de una forma concreta que puede repetirse en contextos específico".

3. Otras definiciones más aceptadas por la comunidad software son:

"Un patrón es una información que captura la estructura esencial y la perspicacia de una familia de soluciones probadas con éxito para un problema repetitivo que surge en un cierto contexto y sistema".

"Un patrón es una unidad de información nombrada, instructiva e intuitiva que captura la esencia de una familia exitosa de soluciones probadas a un problema recurrente dentro de un cierto contexto".

4.3. Patrones de software.

Los patrones para el desarrollo de software son uno de los últimos avances de la Tecnología Orientada a Objetos. Los patrones son una forma literaria para resolver problemas de ingeniería del software, que tienen sus raíces en los patrones de la arquitectura. Los diseñadores y analistas de software más experimentados aplican de forma intuitiva algunos criterios que solucionan los problemas de manera elegante y efectiva. La ingeniería del software se enfrenta a problemas variados que hay que identificar para poder utilizar la misma solución (aunque matizada) con problemas similares.

Por otra parte, las metodologías Orientadas a Objetos tienen como uno de sus principios "no reinventar la rueda" para la resolución de diferentes problemas. Por lo tanto los patrones se convierten en una parte muy importante en las Tecnologías Orientadas a Objetos para poder conseguir la reutilización.

Debe existir una comunicación entre los distintos ing enieros para compartir los resultados obtenidos. Por tanto debe existir también un esquema de documentación con el objetivo de que la comunicación pueda entenderse de forma correcta. Esta comunicación no se debe reducir a la implementación sino a unos niveles de abstracción, desde un proceso de desarrollo hasta la utilización eficiente de un lenguaje de programación. El objetivo de los patrones es crear un lenguaje común a una comunidad de desarrolladores para comunicar experiencia sobre los problemas y sus soluciones.

Dos términos que aparecerán en muchas ocasiones y que son un objetivo permanente del diseño orientado a objetos, como son la cohesión y el acoplamiento.

Podríamos definir la cohesión de una clase (o de un paquete, o de lo que sea) como la relación entre los distintos elementos de la clase, normalmente sus métodos. La cohesión dice que todos los elementos de una clase tienen que trabajar en la misma dirección, es decir, hacia un mismo fin. Por ejemplo, una clase "Coche" debería ocuparse de cosas relacionadas con el coche en si, como acelerar y frenar, pero no de cosas ajenas a él como manipular información referente a su seguro. La cohesión es una medida relativa, en el sentido de que depende de lo que cada uno piense que es la función de la clase, pero lo importante es mantener una cohesión lo más alta posible.

Respecto al acoplamiento, se podría decir que es la interdependencia existente entre dos clases, paquetes, etc. Esto ocurre normalmente cuando una clase (o paquete) necesita saber demasiados detalles internos de otra para su funcionamiento, es decir, rompe el encapsulamiento del que tanto se habla en la programación orientada a objetos. También existen diversos tipos de acoplamiento (funcional, de datos, etc.), pero al igual que ocurría con la cohesión, eso no nos importa ahora, no es el objetivo de estos artículos. Por supuesto, para tener un diseño correcto, fácil de mantener y modular, cuanto más bajo acoplamiento haya entre las clases (o paquetes) mejor.

4.3.1. Características de los patrones software.

Hay que tener en cuenta que no todas las soluciones que tengan, en principio, las características de un patrón son un patrón, sino que debe probarse que es una solución a un problema que se repite, ya que un buen patrón debe tener las siguientes características:

- Solucionar un problema: los patrones capturan soluciones, no sólo principios o estrategias abstractas.
- Ser un concepto probado: los patrones capturan soluciones demostradas, no teorías o especulaciones.
- La solución no es obvia: muchas técnicas de solución de problemas tratan de hallar soluciones por medio de principios básicos. Los mejores patrones generan una solución a un problema de forma indirecta.
- Describe participantes y relaciones entre ellos: los patrones no sólo describen módulos sino estructuras del sistema y mecanismos más complejos.

Los patrones indican repetición, si algo no se repite, no es posible que sea un patrón. Pero la repetición no es la única característica importante. También necesitamos mostrar que un patrón se adapta para poder usarlo, y que es útil. La repetición es una característica cuantitativa pura, la adaptabilidad y utilidad son características cualitativas. Podemos mostrar la repetición simplemente comprobando que se adapta al menos en 3 sistemas existentes); mostrar la adaptabilidad explicando “como” el patrón es exitoso; y mostrar la utilidad explicando “por qué” es exitoso y beneficioso. Así que aparte de la repetición, un patrón debe describir “como” la solución resuelve sus objetivos, y “por qué” está es una buena solución.

4.3.2. Tipos de patrones software.

Existen diferentes ámbitos dentro de la ingeniería del software donde se pueden aplicar los patrones: patrones de arquitectura, patrones de programación, patrones de análisis, patrones organizacionales y patrones de diseño. La diferencia entre estas clases de patrones está en los diferentes niveles de abstracción y detalle, y del contexto particular en el cual se aplican. Pero de todos ellos solo nos vamos a centrar en los patrones de diseño, que los patrones utilizados en el proyecto pertenecen sólo a este último tipo.

Los patrones de diseño proporcionan un esquema para refinar los subsistemas o componentes de un sistema software, o las relaciones entre ellos. Describen estructuras repetitivas en los que se comunican componentes que resuelven un problema de diseño en un contexto particular.

4.3.2.1. Patrones de diseño.

Una cosa que los diseñadores expertos no hacen es resolver cada problema desde el principio. A menudo reutilizan las soluciones que ellos han obtenido en el pasado. Cuando encuentran una buena solución, utilizan esta solución una y otra vez. Estos patrones solucionan problemas específicos del diseño y hacen los diseños orientados a objetos más flexibles, elegantes y por último reutilizables. Si siempre se pudieran recordar los detalles de los problemas anteriores y como se solucionaron, se podrían reutilizar en vez de tener que volver a pensarlo. Es por esto que la experiencia es muy importante. El objetivo de los patrones de diseño es guardar esa experiencia en diseños de programas orientados a objetos.

Cada patrón de diseño nombra, explica y evalúa un importante diseño en los sistemas orientados a objetos. Es decir se trata de agrupar la experiencia en diseño de una forma que la gente pueda utilizarlos con efectividad, además estos patrones ayudarán al diseñador a conseguir un diseño correcto rápidamente.

Los patrones de diseño tienen un cierto nivel de abstracción. Son descripciones de las comunicaciones de objetos y clases que son personalizadas para resolver un problema general de diseño en un contexto particular. Un patrón de diseño nombra, abstrae e identifica los aspectos clave de un diseño estructurado, común, que lo hace útil para la creación de diseños orientados a objetos reutilizables. Los patrones de diseño identifican las clases participantes y las instancias, sus papeles y colaboraciones, y la distribución de responsabilidades. Los patrones de diseño se pueden utilizar en cualquier lenguaje de programación orientado a objetos, adaptando los diseños generales a las características de la implementación particular.

4.3.2.2. Clasificación de los patrones de diseño.

Existen seis categorías para englobar el gran número de patrones que hay:

- Patrones de diseño fundamentales. Los patrones de esta categoría son los más utilizados e importantes.
- Patrones de creación. Los patrones de creación muestran la guía de cómo crear objetos cuando sus creaciones requieren tomar decisiones. Estas decisiones normalmente serán resueltas dinámicamente decidiendo que clases instanciar o sobre que objetos un objeto delegará responsabilidades.
- Patrones de participación. En la etapa de análisis, se examina el problema para identificar los actores, casos de uso, requerimientos y las relaciones que constituyen el problema. Los patrones de esta categoría proveen la guía sobre como dividir actores complejos y casos de uso en múltiples clases.
- Patrones estructurales. Los patrones de esta categoría describen las formas comunes en que diferentes tipos de objetos pueden ser organizados para trabajar unos con otros.
- Patrones de comportamiento. Los patrones de este tipo son utilizados para organizar, manejar y combinar comportamientos.
- Patrones de concurrencia. Los patrones de esta categoría permiten coordinar las operaciones concurrentes. Se dirigen principalmente a dos tipos diferentes de problemas: Recursos compartidos y Secuencia de operaciones (si las operaciones son protegidas para acceder a un recurso compartido una cada vez, podría ser necesario garantizar que ellas acceden a los recursos compartidos en un orden particular).

En el proyecto que nos ocupa se han utilizado 2 patrones; estos son: el patrón Proxy que pertenece a la familia de patrones de diseño fundamentales, y el patrón Observador que pertenece a los patrones de comportamiento. Ambos serán comentados en detalle a continuación.

4.4. Patrón Observador.

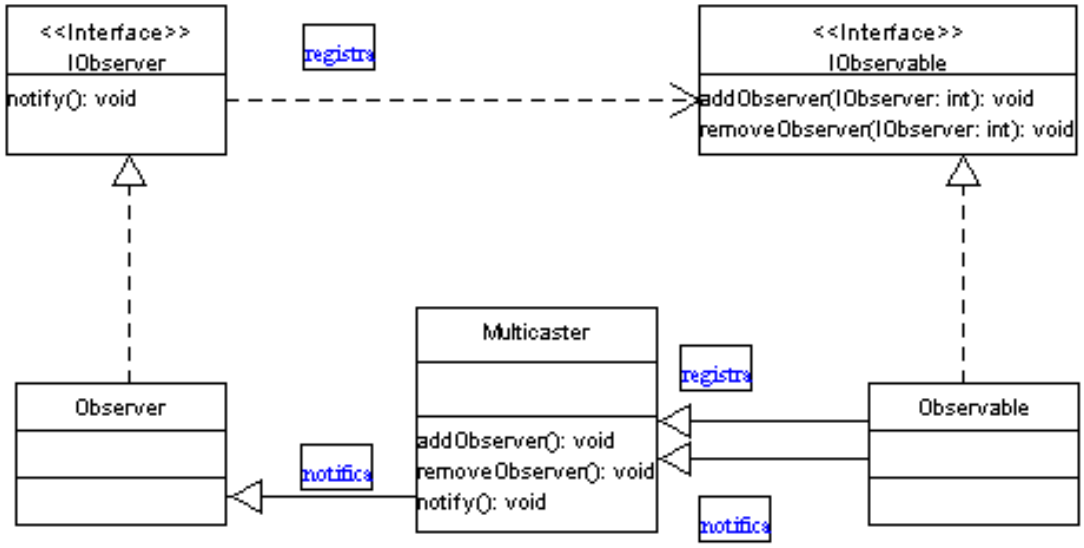
El objetivo de este patrón es permitir a los objetos captar dinámicamente las dependencias entre objetos, de tal forma que un objeto notificará a los que dependen de él cuando cambia su estado, siendo actualizados automáticamente. Este patrón se utilizará en las siguientes situaciones:

- Cuando un sistema requiere que unos elementos sean conscientes de los cambios producidos en otros.

- Cuando la dependencia entre instancias se produce de forma dinámica, en tiempo de ejecución y no siempre.
- Cuando la dependencia se produce de un objeto hacia muchos y un sistema simple de eventos no sirve porque solo permite notificar a una sola instancia.
- En ocasiones se implementan sistemas de eventos mediante este sistema, por ejemplo en el API de Java, de forma que los objetos que notifican de eventos implementan la interfaz IObservable y los que reciben las notificaciones de eventos implementan la interfaz IObserver. Esto permite que muchos objetos reciban eventos de otro objeto en lugar de los sistemas de eventos básicos que solo permiten notificar a un único objeto.

4.4.1. Estructura.

El diagrama de clases que muestra la organización de las clases e interfaces que participan en el patrón Observer es el siguiente:



A continuación se muestran las descripciones de los papeles que juegan las clases e interfaces en la organización citada anteriormente:

IObserver. Una interfaz en este papel define un método normalmente llamado notifica o actualiza. Un objeto Observable llama a este método para proporcionar una notificación de que su estado ha cambiado, pasándole los argumentos apropiados. En muchos casos, una referencia al objeto Observable es uno de los argumentos que permite al método conocer que objeto proporciona la notificación.

Observer. Las instancias de clases en este papel implementan la interfaz IObserver y reciben notificaciones de cambio de estado de los objetos Observable.

IObservable. Los objetos Observable implementan una interfaz en este papel. La interfaz define dos métodos que permiten a los objetos Observer registrarse y desregistrarse para recibir notificaciones.

Observable. Una clase en este papel implementa la interfaz IObservable . Sus instancias son las responsables de manejar la inscripción de objetos IObserver que quieren recibir notificaciones de cambios de estado. Sus instancias son también responsables de distribuir las notificaciones. La clase Observable no implementa directamente estas responsabilidades. En lugar de eso, delega estas responsabilidades a un objeto Multicaster.

Multicaster. No es imprescindible, ya que sus funciones las puede llevar a cabo la propia clase Observable, pero delegar el registro de observers y la notificación en una sola clase permite reutilizar fácilmente el mecanismo.

Una descripción más detallada de las interacciones mostrada en la figura anterior es la siguiente:

1. Los objetos que implementan una interfaz IObserver son pasados al método addObserver de un objeto IObsevable.

    1.1. El objeto IObservable delega la llamada a addObserver a su objeto asociado Multicaster. Este añade el objeto IObservable a la colección de objetos IObserver que él mantiene.

2. El objeto IObservable necesita notificar a otros objetos que son dependientes de él que su estado ha cambiado. Este objeto inicializa la notificación llamando al método notify de su objeto asociado Multicaster.

    2.1. El objeto Multicaster llama al método notify de cada uno de los objetos IObserver de su colección.

4.4.2. Implementación.

Generalmente el objeto Observable al notificar al objeto Observer le pasa una referencia de si mismo como parámetro del método notify, de manera que el objeto receptor de la notificación puede acceder a los atributos del objeto que observa. Hay que tener en cuenta que un mismo objeto Observer puede haberse registrado como observador de varios objetos observables, de forma que cuando le llega la notificación necesita saber de quien le viene. Independientemente de como se implemente la solución la forma más prácticas requiere que la clase Observer conozca concretamente la clase Observable o se utilicen interfaces especificas para cada tipo de notificación según la clase origen.

En ocasiones conviene reducir el número de notificaciones simplemente por que se van a producir más cambios, en este caso se espera a que se produzcan todos los cambios y se retrasa la notificación hasta que se han completado todos. Se puede implementar añadiendo dos métodos a la clase Observable para indicar el comienzo de cambios y el final de los mismos, de forma que las notificaciones están desactivadas entre tanto.

4.4.3. Consecuencias de su uso.

Este patrón permite enviar notificaciones desde un objeto a muchos, de forma dinámica y sin que las clases implicadas sean conscientes de las clases del resto. Un objeto observable puede ser a su ver observer respecto de otros.

En ocasiones se pueden producir consecuencias no deseables. Cuando existen muchos observers registrados o cuando se producen varias notificaciones en cascada se puede ralentizar notablemente el envío de notificaciones. Otro caso ocurre cuando se producen ciclos de notificaciones, de forma que la notificación de un cambio en el objeto Observable produce de forma indirecta una cadena de notificaciones que vuelve a provocar un cambio en el objeto (reproduciendose el ciclo) o simplemente el objeto observable está de forma indirecta dentro de un ciclo de observers. En ambos casos se produce un bucle infinito que termina cuando se agota la memoria.

Este problema no obstante no es inherente al patrón Observer, también se puede producir en la notificación de eventos normales en cuanto se produzca un ciclo de objetos que capturan eventos y los lanzan a su vez. Una forma trivial de evitar esto consiste en utilizar un flag que indica que se está procesando una notificación, de forma que el objeto no aceptará nuevas notificaciones hasta que haya terminado con la anterior, de forma que se interrumpe el ciclo.

4.4.4. Patrones relacionados.

Otros patrones que se pueden estudiar por estar relacionados con el patrón Observer son:

- Adapter El patrón Adapter puede ser utilizado para permitir a los objetos que no implementen la interfaz requerida participar en el patrón Observer para recibir notificaciones.
- Delegation El patrón Observer utiliza el patrón Delegation.

- Mediator El patrón Mediator es utilizado algunas veces para coordinar cambios de estado inicializados por múltiples objetos a un objeto Observable.

4.4.5. Implementación en la aplicación Goya.

Se ha decidido utilizar el patrón proxy en la aplicación para permitir a los objetos servidores captar dinámicamente las dependencias entre objetos clientes, de tal forma que un servidor notificará a los que dependen de él cuando cambia su estado, siendo actualizados automáticamente. El servicio de registro de clientes en el servidor se distinguirá por las siguientes ventajas que presenta el patrón observador:

- El sistema requiere que los clientes sean conscientes de los cambios producidos en los servidores (robot).
- La dependencia entre cliente y servidor se produce de forma dinámica, en tiempo de ejecución y no siempre.
- La dependencia se produce de un servidor hacia muchos clientes y un sistema simple de eventos no sirve porque solo permite notificar a una sola instancia.

A continuación se especifica qué clases de la aplicación desarrollada coinciden con el papel de las clases explicadas anteriormente:

Las interfaces MechanismListener y ToolListener del paquete Listeners son las interfaces IObserver que se comento anteriormente, ya que contienen varios métodos para notificar o actualizar la información del estado del robot. Esta interfaz la implementaran los clientes, que son las instancias que recibirán las actualizaciones de cambio de estado de los servidores del robot, por ello, a los clientes se les conoce como observadores (Observer).

Los servidores en cambio, implementan las interfaces MechanismCtrl y ToolCtrl que definen los métodos que permiten a los objetos Listeners registrarse y desregistrarse para recibir notificaciones. Sin embargo, los servidores son también responsables de distribuir las notificaciones a todos los observadores registrados (Listeners), pero las clases MechanismServer y ToolServer no implementan directamente estas responsabilidades. En lugar de eso, delega estas responsabilidades a un objeto EventControl y TimerControl (realizan la función de Multicaster).

Las clases TimerControl y EventControl no son imprescindibles, ya que sus funciones las pueden llevar a cabo los propios servidores, pero delegar el registro de observadores y la notificación en una clase permite reutilizar fácilmente el mecanismo.

Una descripción más detallada de las interacciones entre escuchadores y servidores es la siguiente:

- Los objetos que implementan las interfaces Listener (clientes) son pasados al método addListener de un objeto Ctrl (servidor).
  - El objeto Ctrl (servidor) añade el objeto cliente a la colección de objetos Listener que él mantiene.
- El servidor necesita notificar a otros objetos que son dependientes de él que su estado ha cambiado. Éste inicializa la notificación llamando al método update (notify) de su objeto asociado EventControl o TimerControl en cada caso (eventos o tiempo).
  - El objeto EventControl o TimerControl llama al método update (notify) de cada uno de los objetos Listener de su colección.

Podemos comprobar que este patrón permite enviar notificaciones desde un objeto a muchos, de forma dinámica y sin que las clases implicadas sean conscientes de las clases del resto. Por lo que el patrón esta hecho a medida para implementarse en el servicio de registro de clientes en un servidor de la aplicación desarrollada.

Nota: Este patrón también se utiliza en la aplicación cuando se crea una instancia de la clase Timer de la API de Java en el servidor cuando se registra el primer cliente. Al crear una instancia de la clase Timer, se crea un reloj que notificara cada cierto periodo de tiempo a una clase que implemente la interfaz ActionListener mediante el método actionPerformed. El proceso se comenta en detalle en el apartado de Servicio de registro de los servidores, dentro del capítulo de Aspectos característicos de la implementación.

4.5. Patrón Proxy.

Este patrón (también conocido como Surrogate o Virtual Proxy) consiste en comunicar a los clientes de un componente a través de una entidad intermedia. Introducir esta entidad intermedia puede servir para muchas cosas, como mejorar la eficiencia del sistema, facilitar el acceso o proteger usos no autorizados del sistema. Y es que, a menudo no es lo mejor el acceso directo a un componente por parte de los clientes. Los motivos pueden ser variados: eficiencia,

coste, seguridad, transparencia de ubicación, interfaz simple y homogénea para todos los componentes que se acceden, etc.

El proxy es un componente intermedio que no sólo es una interfaz de acceso al componente, sino que además realiza un pre- y un postprocesamiento del flujo de información que pasa a través de él.

Un posible ejemplo es el caso de un sistema de acceso a una base de datos de una compañía. La mayoría de los accesos son muy parecidos. Una forma de abaratar costes y de bajar los tiempos de espera del sistema es que haya un proxy intermedio que haga de cache intermedia de los datos que se van leyendo. Esta solución no implica cambiar la aplicación, de gestión de base de datos y mejora las prestaciones del sistema.

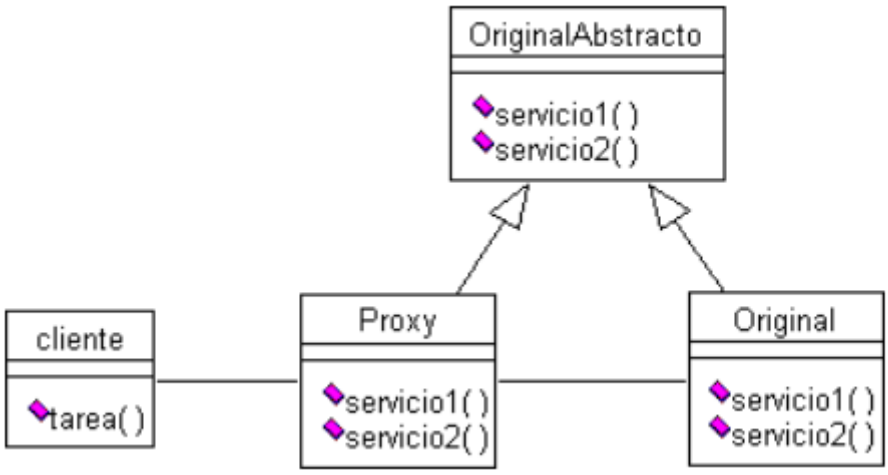
4.5.1. Implementación.

Los pasos para la implementación del patrón Proxy son muy sencillos. En el primero de ellos se ha de identificar las responsabilidades para gestionar el acceso a un componente, asignando todas estas responsabilidades al proxy. Seguidamente, si es posible, se introducirá una clase abstracta, de la que hereden el proxy y el original, y que contendrá las partes comunes a ambas clases. Si las interfaces de acceso al proxy y al componente son diferentes, puede introducirse un "adaptador" para hacerlas compatibles.

El siguiente paso será implementar las funciones del proxy, eliminar de los clientes y de los originales todo lo que se haya metido en el proxy y también asociar al proxy con el original mediante algún mecanismo. El mecanismo puede ser un puntero, una referencia, un identificador, un puerto, etc.

El último paso de todos será eliminar en los clientes todas las referencias al original que existan y sustituirlas por referencias al proxy.

4.5.2. Estructura.



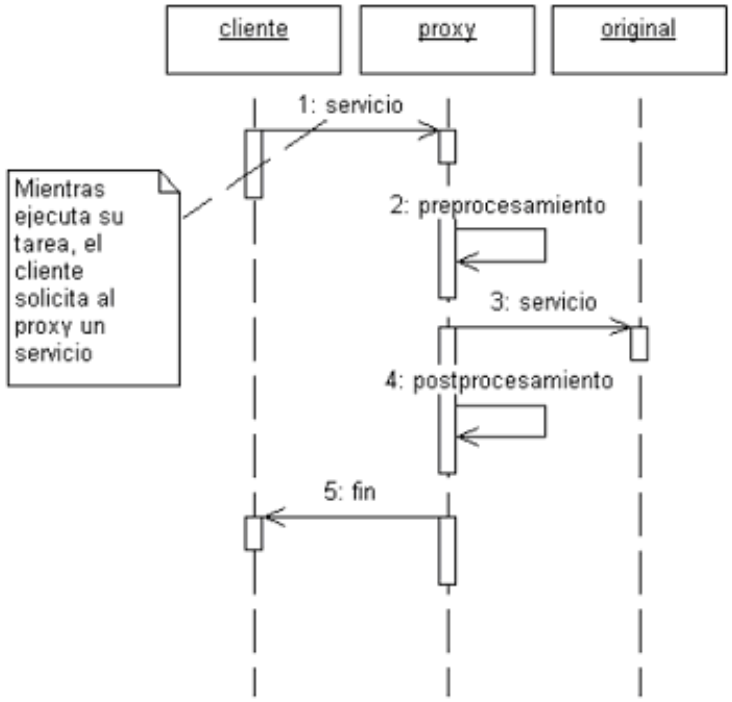
4.5.3. Componentes.

Cliente. La aplicación, utiliza el interfaz de la clase proxy para hacer uso de la clase original.

Proxy. Ofrece un interfaz equivalente al de la clase Original, pudiendo derivar de una clase común. Puede realizar un preprocesamiento y un postprocesamiento sobre los servicios ofrecidos por la clase original, este procesamiento adicional puede tener los objetivos de: transformar infomación (parametros de llamada y resultados), control de acceso, tareas de conexión remota, retraso en la instanciación, destrucción de instancias y caché.

Original. Es la clase que implementa los servicios ofrecidos, puede ser una instancia local o remota.

4.5.4. Diagrama.



4.5.5. Aplicaciones.

Entre las aplicaciones encontradas donde usar este patrón destaca el llamado "Proxy Remoto", que se da cuando el objeto está en un sistema remoto y oculta detalles de acceso a la red. Otras aplicaciones son el "Proxy Virtual" y "Proxy de proctección"; el primero de ellos se utiliza en el caso en que se tenga que retrasar la creación de objetos hasta que sean necesarios; así, se cargará sólo la parte de componente remoto que se necesita, ahorrando de este modo recursos. El Proxy de Protección se aplica para controlar el acceso a un objeto, protegiéndolo frente a accesos no autorizados desde el exterior.

Aunque esas 3 aplicaciones son las más importantes, el patrón proxy también se aplica como "Proxy de Sincronización" para gestionar accesos de múltiples clientes a un recurso o componente o como "Proxy contador", mediante el cual se realizarían estadísticas de uso, evitando también borrados accidentales.

Otro uso aunque un poco más complejo se da en las referencias inteligentes, usadas para la gestión y mantenimiento del acceso a un objeto real, permitiendo contabilizar su utilización de cara a gestionar sus recursos e incluso destruirlo cuando ya no se necesita.

4.5.6. Consecuencias de su uso.

Al usar el patrón proxy en nuestro diseño aporta algunas ventajas e inconvenientes. En cuanto a las ventajas destacan que se puede mejora al controlar la instanciación de recursos y al hacer de caché, los clientes ya no se preocuparán de la ubicación de los componentes, el código de control estará separado de la funcionalidad del cliente, y el aumento de la seguridad.

Entre los inconvenientes encontramos que la eficiencia también disminuirá al existir un nivel más de indirección, y además si el proxy es muy complicado, puede sobrecargar el sistema innecesariamente.

4.5.7. Implementación en la aplicación Goya.

Este patrón se ha implementado en la aplicación por dos veces, de forma que se ha realizado en el extremo del cliente y en el extremo del servidor. En este caso, el patrón proxy del extremo del cliente consiste en comunicarse con los servidores a través de una entidad intermedia llamada proxy, mientras que en el extremo del servidor el patrón consiste en comunicarse con los clientes a través de una entidad intermedia llamada stub.

El motivo por el que se ha implementado este patrón es para conseguir una transparencia de ubicación de clientes y servidores, y utilizar unas interfaces simples y homogéneas para todos los componentes que se acceden.

A continuación se va a comentar en detalle los pasos para la implementación del patrón proxy en el extremo del cliente, quedando también explicado el patrón en el extremo del servidor ya que son muy similares. La diferencia radica en que el proxy situado en el extremo del cliente actua como representante del servidor, ocultando al cliente la localización del mismo. Mientras que el proxy del extremo del servidor (stub) actúa como un representante de un cliente, ocultando al servidor la distribución de todos los posibles clientes existentes en el sistema.

Los pasos para la implementación del proxy en la aplicación en cu estión son muy sencillos. En el primero de ellos se ha de identificar las responsabilidades para gestionar el acceso al servidor, asignando todas estas responsabilidades al proxy. Seguidamente, se introducirá una interfaz, de la que hereden el proxy y el servidor (original), y que contendrá las partes comunes a ambas clases (estas interfaces son las definidas en el paquete Controls).

El siguiente paso será implementar las funciones del proxy, eliminar de los clientes y de los originales todo lo que se haya metido en el proxy y también asociar al proxy con el servidor mediante los enlaces de comunicaciones correspondientes (véase apartado de Aspectos característicos de la implementación).

El último paso de todos será eliminar en los clientes todas las r eferencias al servidor que existan y sustituirlas por referencias al proxy. Todo ello se realizará mediante interfaces, de forma que el cliente solo conoce un objeto cualquiera que implemente las interfaces Controls, pero no conoce la implementación de la misma (no sabe si la instancia es realmente una clase servidor o proxy).

Finalmente, es importante comentar que el proxy del extremo del servidor (stub) debe implementar las interfaces Listeners y Updaters, ya que actúa como representante del cliente, y los clientes implementan estas interfaces para que el servidor pueda comunicarse con ellos.

Los componentes que formarían este patrón en la aplicación serían los siguientes:

- Cliente (clase Client):
  El cliente utiliza el interfaz de la clase proxy para hacer uso de la clase original (servidores).

- Proxy (clases Mechanism Proxy, Tool Proxy y Mission Proxy):
  Ofrece unas interfaces equivalentes al de los servidores (interfaces Controls de tipo Mechanism, Tool y Misión). Realiza un preprocesamiento y un postprocesamiento sobre los servicios ofrecidos por los servidores, este procesamiento adicional tiene los objetivos de transformar información (parámetros de llamada y resultados) y tareas de conexión remota.

- Servidor (clases MechanismServer, ToolServer y MissionServer):
  Es la clase que implementa los servicios ofrecidos, es una instancia remota.

Resumiendo, el patrón diseñado en la aplicación es conocido concretamente como llamado “Proxy Remoto”, que se da cuando el objeto está en un sistema remoto y oculta detalles de acceso a la red.

5. UML (Lenguaje de Modelado Unificado).

5.1. Orígenes.

Durante muchos años en la industria del software se ha hablado de la "crisis del software" dado que los proyectos de software no cumplían con los requisitos y necesidades de los usuarios y además se excedían los costos y estimaciones de tiempo.

Mediante el uso de nuevas técnicas de esta última década de los noventa tales como la orientación a objetos, los lenguajes visuales y entornos de desarrollo avanzados se ha conseguido incrementar la productividad pero los principales problemas del desarrollo de software se suelen deber a que muchos proyectos empiezan pronto con la codificación y destinan demasiado esfuerzo en ello apremiados por las prisas de entregar el producto acabado lo antes posible al cliente. También se debe a que los programadores se sienten mucho más seguros con sus líneas de código que construyendo modelos abstractos del sistema que están creando y a que los directores de proyecto desconocen el proceso de desarrollo de software y se vuelven ansiosos porque su equipo no produce código.

Había que cuestionar la calidad de muchos de las primeras metodologías orientadas a objetos dado que fueron pensadas principalmente para pequeños sistemas con funcionalidad limitada y, por ello, no tenían la capacidad de escalar a sistemas de mayores dimensiones. Además la falta de una notación bien definida sobre las cuales varias metodologías y herramientas puedan coincidir ha mantenido confundidos a los desarrolladores y ha hecho que sea más difícil aprender a utilizar una metodología correctamente.

Actualmente, la construcción de modelos de los sistemas antes de implementarlos se ha convertido en una práctica aceptada por la comunidad de la ingeniería del software dado que los sistemas se están volviendo cada vez más grandes y distribuidos en varios ordenadores mediante arquitecturas cliente-servidor y el modelado y la programación están altamente integrados.

5.2. Un poco de historia.

Grady Booch y James Rumbaugh comenzaron los trabajos de UML en 1994 con el objetivo de crear un nuevo método: "Método Unificado", que uniera el de Booch con el de OMT-2 del cual Rumbaugh era líder del desarrollo. En 1995 se les unió Ivar Jacobson, responsable de OOSE y Objectory. En este momento, los futuros desarrolladores de UML,

remarcaron que su trabajo estaba destinado a crear un lenguaje de modelado estándar y rebautizaron su trabajo como "Unified Modeling Language o Lenguaje de Modelado Unificado".

Booch, Rumbaugh, y Jacobson entregaron a la comunidad de orientación a objetos una serie de versiones preliminares de UML. La retroalimentación les dio una serie de ideas y sugerencias a incorporar para mejorar el lenguaje y así apareció la versión 1.0 de UML en Enero de 1997. Durante 1996, un número de empresas se unió a Rational (la empresa de Booch, Rumbaugh y Jacobson) para formar el consorcio de UML Partners. Esas organizaciones adaptaron UML como estrategia para sus propios negocios y estaban ansiosos por contribuir a la definición de UML. Todas estas empresas auspiciaron la propuesta de adoptar UML como el lenguaje de modelado estándar del Object Management Group (OMG).

Cuando el OMG hizo una convocatoria para elegir un lenguaje de modelado estándar, los desarrolladores de UML sostuvieron que éste bien podría ser aceptado como tal lo cual originó una mayor demanda basada en una definición más precisa de UML y mejoró la calidad del lenguaje. Finalmente fue aceptada la propuesta y se estandarizó formalmente lo cual es un paso muy importante para muchas industrias por razones ya conocidas por todos.

Actualmente UML está llamado a ser el lenguaje de modelado dominante utilizado por la industria. Tiene un amplio rango de uso, está construido en base a técnicas para modelar sistemas, bien definidas y probadas, y cuenta con el respaldo de la comunidad del software para establecer un estándar en todo el mundo. UML también está muy bien documentado con metamodelos del lenguaje y con una especificación formal de la semántica del lenguaje además de estar soportados la mayoría de sus modelos con una herramienta CASE disponible en versión evaluación en http://www.rational.com/uml.

5.3. ¿Qué es UML?

UML es un lenguaje para especificar, visualizar, construir y documentar el desarrollo de sistemas software, así como para el modelado de negocios y otros sistemas que no son de software. UML representa una colección de las mejores prácticas de ingeniería que han probado tener éxito en el modelado de sistemas grandes y complejos. Sin embargo, como ya se adelantaba, UML no da una metodología de desarrollo. Sólo define unos diagramas (es un lenguaje de modelado como su propio nombre indica), cada uno de ellos más o menos útil para determinadas tareas.

5.4. Metas de UML.

Las principales metas al diseñar UML fueron:

1. Proveer a los usuarios de un lenguaje de modelado visual, expresivo y listo para usar, que les permita intercambiar modelos coherentes.
2. Proveer mecanismos de extensibilidad y especialización para extender los conceptos básicos.
3. Ser independiente de cualquier lenguaje de programación o proceso de desarrollo.
4. Proveer bases formales para el entendimiento del lenguaje de modelado.
5. Estimular el crecimiento del mercado de herramientas orientadas a objetos.
6. Dar soporte a conceptos de desarrollo de alto nivel como: patrones, colaboraciones, componentes y frameworks.
7. Integrar las mejores prácticas.

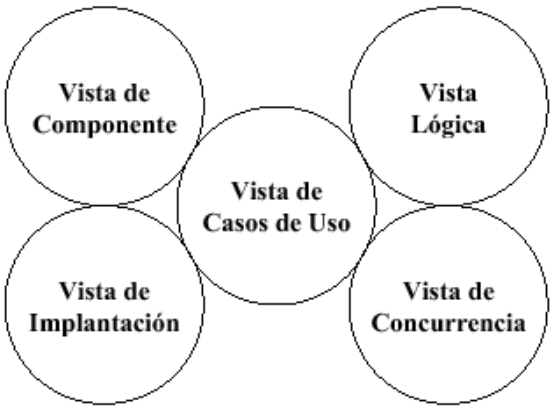
Los modelos se expresan en un lenguaje de modelado el cual consiste en un conjunto de símbolos y de reglas sintácticas, semánticas o pragmáticas que indican cómo usarlo. La sintaxis indica cómo deben verse los símbolos y cómo se combinan en el lenguaje de modelado. Las reglas semánticas nos indican qué significa cada símbolo y cómo debe interpretarse en el contexto en el que está inmerso. Las reglas pragmáticas definen las intenciones de los símbolos mediante los cuales se alcanza el propósito de modelado y se vuelve entendible para los usuarios.

Para usar correctamente un lenguaje de modelado, es necesario conocer todas estas reglas pero aunque el lenguaje esté bien definido no pueden garantizarse los modelos que se construyan con él al igual que al escribir una historia en el lenguaje natural no lleva al autor de la misma a escribir una buena historia.

5.5. Vistas de un sistema.

La descripción de los sistemas se realiza en UML a través de Vistas, las cuales a su vez están integradas por Diagramas. Esta estrategia parte del hecho de que un solo diagrama no puede expresar toda la información que se requiere para describir un sistema. Sí se hace un símil con una edificación, no es posible elaborar un plano que contenga todos los detalles de su construcción; en lugar de ello, se dibujan planos que presentan diferentes aspectos del edificio: estructura, instalaciones eléctricas, instalaciones hidráulicas, diseño exterior, etc. Así pues, es necesario utilizar conjuntos separados de diagramas, las vistas, para representar proyecciones del sistema relacionadas con aspectos particulares.

La figura de la parte inferior muestra las diferentes vistas consideradas en UML (a estas vistas se las conocen también como “las 4+1 vistas de Kruchten”).



5.5.1. Vista de Casos de Uso.

El detalle de representar la Vista de Casos de Uso en el centro de todas las figuras no es una casualidad, ya que esta vista hace el papel de enlace, siendo el hilo conductor de todo el proceso de desarrollo. Es la única vista que no describe aspectos de la construcción del sistema sino de su comportamiento. La Vista de Casos de uso muestra la funcionalidad del sistema, tal como es percibida por actores externos.

La Vista de Casos de Uso es utilizada por todos los participantes en el proceso de desarrollo: los clientes, pues a través de ella se definen y expresan los requerimientos del sistema; y los equipos de diseño, desarrollo, y pruebas que conducen todo el proceso de desarrollo y verificación. Esta vista utiliza principalmente los diagramas de Casos de Uso.

5.5.2. Vista Lógica.

Muestra el diseño de la funcionalidad del sistema en sus dos aspectos esenciales: su estructura (los componentes que lo integran) y su comportamiento (dinámica de interacción de dichos componentes).

Esta vista es utilizada fundamentalmente por los equipos de diseño y desarrollo, y consta de los siguientes diagramas:

- Para la descripción de estructura:
    - Diagramas de Clases y de Objetos.

- Para la descripción del comportamiento:
    - Diagramas de Estado, Secuencia, Colaboración y Actividad.

5.5.3. Vista de Componentes.

UML no se limita a ofrecer una notación para representar los modelos obtenidos en el proceso de desarrollo de los programas, que al fin y al cabo constituyen una abstracción de los mismos, sino que también ofrece elementos para representar las entidades, que son el resultado de todo el trabajo de desarrollo; los archivos.

Mediante la Vista de Componentes se muestra la organización del código y archivos que hacen parte del sistema, tanto los que han sido desarrollados (programas fuente, ejecutables-etc.) como los que han sido obtenidos (bibliotecas de funciones o de servicios, componentes reutilizados, etc,); además, muestra también las relaciones de dependencia que existen entre ellos. Es utilizado por el grupo de desarrollo y consiste en el Diagrama de Componentes.

5.5.4. Vista de Implantación.

Muestra la implantación del sistema en la arquitectu ra física, indicando dónde se localizan los ejecutables del sistema y cómo se comunican entre si. Para ello se utiliza una descripción de los nodos del sistema, que son los computadores donde éste se ejecuta, y los dispositivos periféricos relevantes.

Es utilizado por los grupos de desarrollo, integración y pruebas, y consiste en el Diagrama de Implantación.

5.5.5. Vista de Concurrencia.

Es una combinación de las vista Lógica, de Componentes y de Implantación, en la que se muestra el manejo de los aspectos de concurrencia en el sistema, especialmente los de comunicación y sincronización. Para ello, el sistema se divide en procesos, que manejan su propio flujo de control al procesador; y se presentan tanto los aspectos estáticos de la asignación de los componentes a la arquitectura física, como los aspectos dinámicos de su interacción.

Esta es una vista de gran importancia para los sistemas distribuidos y de tiempo real, y es utilizada principalmente por lo grupos de desarrollo e integración. Consta de los siguientes diagramas:

- Para la descripción de la implementación:
    - Diagramas de Componente e Implantación.
- Para la descripción dinámica:
    - Diagramas de Estado, Secuencia, Colaboración y Actividad.

6. Modelado del sistema mediante UML.

6.1. Vista de casos de uso.

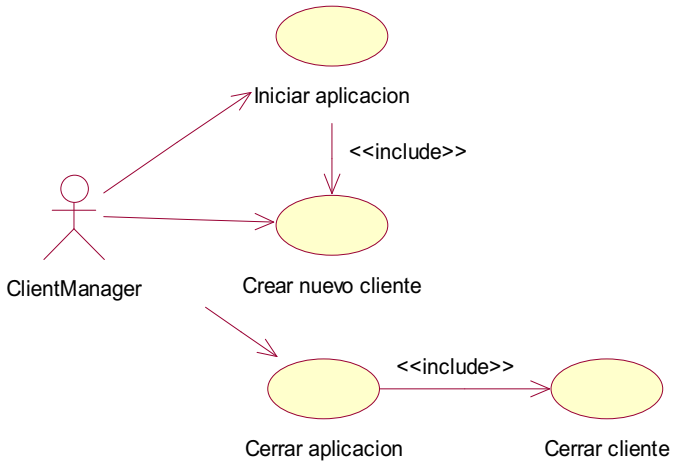


Diagrama 1: Casos de uso ClientManager

Iniciar aplicación

1. Nombre. Iniciar aplicación.
2. Actores. ClientManager.
3. Propósito. Englobar el conjunto de procesos que tienen como resultado final la puesta en marcha de la aplicación.
4. Descripción. Para poder enviar cualquier orden a la máquina servidor, primero se han de dar una serie de procesos mediante los cuales se crean los enlaces entre las máquinas clientes y el servidor.
5. Pasos. Todos los pasos se muestran en los diagramas de secuencia que se adjuntan.
6. Condiciones. para iniciar la aplicación es necesario, que ésta no esté ya "corriendo" y que las conexiones entre los extremos funcionen correctamente.
7. Resultado. Se muestra una interfaz para el usuario, mediante la cual éste puede manejar el robot remotamente.

Crear nuevo cliente

1. Nombre. Crear nuevo cliente.
2. Actores. ClientManager.

3. Propósito. Englobar el conjunto de procesos que tienen como finalidad la creación de un nuevo cliente.
4. Descripción. Si uno de los clientes se ha cerrado anteriormente, se puede recurrir a crear un nuevo cliente para seguir manejando el robot.
5. Pasos. Los pasos a realizar son muy pocos ya que lo único que se requiere es que se muestre una nueva interfaz con el usuario, de tipo ClientGUI. Con ésta, se utilizarán los mismos objetos, enlaces y recursos utilizados para el cliente anterior (de la misma máquina).
6. Condiciones. La única condición necesaria es que la aplicación esté en funcionamiento. Se puede dar el caso que en una misma máquina estén funcionando simultáneamente varios clientes, pero su utilidad no aumenta en comparación con tener un solo un cliente.
7. Resultado. El resultado es muy simple ya que lo único que aparece es una nueva interfaz con el usuario donde manejar el robot.

Cerrar aplicación

1. Nombre. Cerrar aplicación.
2. Actores. ClientManager.
3. Propósito. Englobar el conjunto de procesos que tienen como resultado final el cierre de todos los componentes que intervenían en la aplicación.
4. Descripción. Cuando se cierra la aplicación, tanto los servidores, como los clientes y los enlaces existentes entre ellos se han de cerrar, ya que sino, saltarían un gran número de excepciones quedando estos elementos muy inestables.
5. Pasos. Todos los pasos se muestran en los diagramas de secuencia que se adjuntan.
6. Condiciones. Para cerrar la aplicación es necesario , que ésta esté ya “corriendo” y que las conexiones entre los extremos funcionen correctamente.
7. Resultado. Todas las interfaces con el usuario serán eliminadas así como todos los procesos pertenecientes a la aplicación.

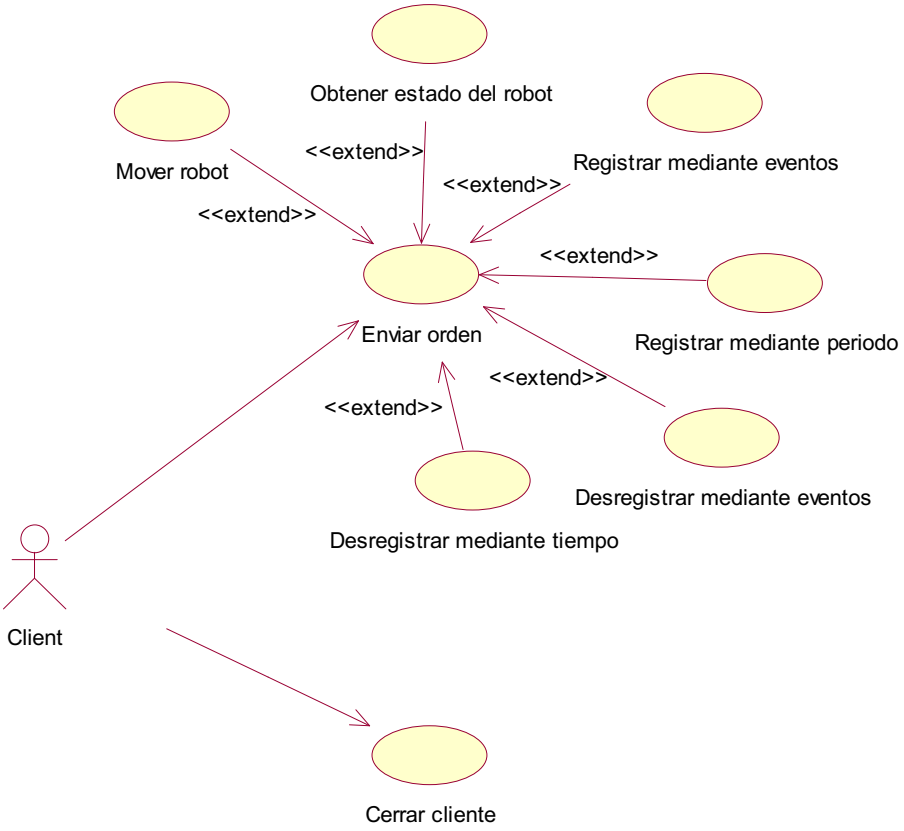


Diagrama 2: Casos de uso Client

Mover robot

1. Nombre. Mover robot.
2. Actores. Client.
3. Propósito. englobar el conjunto de procesos que tienen como resultado final el movimiento del robot.
4. Descripción. Una de las acciones que se pueden hacer desde el cliente, es desplazar el robot. Para esto, se han implementado un gran número de métodos en los que se varía la posición del robot.
5. Pasos. Todos los pasos se muestran en los diagramas de secuencia que se adjuntan.
6. Condiciones. Para cualquiera de las ordenes que se pueden dar desde los clientes, es necesario que la aplicación esté en funcionamiento, y además las conexiones entre los extremos deben funcionar correctamente.
7. Resultado. El robot se desplazará según los parámetros indicados.

Obtener estado del robot

1. Nombre. Obtener estado del robot.
2. Actores. Client.
3. Propósito. Englobar el conjunto de procesos que permiten al cliente recibir el estado del servidor.
4. Descripción. El cliente, para poder tener un control total y efectivo sobre el robot, es necesario que conozca en todo momento el estado del robot, por ello, el cliente necesitará realizar esta acción en muchas ocasiones.
5. Pasos. Todos los pasos se muestran en los diagramas de secuencia que se adjuntan.
6. Condiciones. Para cualquiera de las ordenes que se pueden dar desde los clientes, es necesario que la aplicación esté en funcionamiento, y además las conexiones entre los extremos deben funcionar correctamente.
7. Resultado. El cliente recibirá un objeto del mismo tipo que el que representa el estado del robot, con el valor actual.

Registrar mediante eventos

1. Nombre. Registrar mediante eventos.
2. Actores. Client.
3. Propósito. Englobar el conjunto de procesos que tienen como resultado final el registro mediante eventos del cliente.
4. Descripción. Cuando el cliente necesita que cada vez que surja un evento determinado conozca el estado del robot, realizará esta acción para que así, el servidor le mande dicho estado.
5. Pasos. Todos los pasos se muestran en los diagramas de secuencia que se adjuntan.
6. Condiciones. Para cualquiera de las ordenes que se pueden dar desde los clientes, es necesario que la aplicación esté en funcionamiento, y además las conexiones entre los extremos deben funcionar correctamente.
7. Resultado. El cliente recibirá la actualización del estado del robot cada vez que ocurra un evento. Además, si el cliente ya está registrado su efecto será nulo.

Registrar mediante periodo

1. Nombre. Registrar mediante periodo.
2. Actores. Client.
3. Propósito. Englobar el conjunto de procesos que tienen como resultado final el registro mediante periodo del cliente.

4. Descripción. Cuando el cliente necesita periódicamente la actualización del estado del robot, realizará esta acción para que así, el servidor le mande dicho estado cada vez que pase un periodo (indicado por el cliente).
5. Pasos. Todos los pasos se muestran en los diagramas de secuencia que se adjuntan.
6. Condiciones. Para cualquiera de las ordenes que se pueden dar desde los clientes, es necesario que la aplicación esté en funcionamiento, y además las conexiones entre los extremos deben funcionar correctamente.
7. Resultado. El cliente recibirá la actualización del estado del robot cada vez que ocurra un evento. Además, si el cliente ya está registrado con el mismo periodo su efecto será nulo. En cambio, si el periodo no es el mismo, el servidor actualizará el valor del periodo de actualización.

Desregistrar mediante eventos

1. Nombre. Desregistrar mediante eventos.
2. Actores. Client.
3. Propósito. Englobar el conjunto de procesos que tienen como resultado final la eliminación del cliente en el "registro de eventos" del servidor.
4. Descripción. Cuando el cliente ya no desea recibir más actualizaciones del estado del robot cada vez que surge un evento, realizará esta acción.
5. Pasos. Todos los pasos se muestran en los diagramas de secuencia que se adjuntan.
6. Condiciones. Para cualquiera de las ordenes que se pueden dar desde los clientes, es necesario que la aplicación esté en funcionamiento, y además las conexiones entre los extremos deben funcionar correctamente.
7. Resultado. El cliente ya no recibirá este tipo de actualizaciones. Si el cliente que realiza esta acción no está registrado, el resultado será nulo.

Desregistrar mediante periodo

1. Nombre. Desregistrar mediante periodo.
2. Actores. Client.
3. Propósito. Englobar el conjunto de procesos que tienen como resultado final la eliminación del cliente en el "registro periódico" del servidor.
4. Descripción. Cuando el cliente ya no desea recibir más actualizaciones del estado del robot cada vez que pase un periodo, realizará esta acción.
5. Pasos. Todos los pasos se muestran en los diagramas de secuencia que se adjuntan.
6. Condiciones. Para cualquiera de las ordenes que se pueden dar desde los clientes, es necesario que la aplicación esté en funcionamiento, y además las conexiones entre los extremos deben funcionar correctamente.

7. Resultado. El cliente ya no recibirá este tipo de actualizaciones. Si el cliente que realiza esta acción no está registrado, el resultado será nulo.

Enviar orden

1. Nombre. Enviar orden.
2. Actores. Client.
3. Propósito. Englobar el conjunto de procesos que tienen como resultado final el envío de una orden al servidor.
4. Descripción. Sin importar si la orden es un tipo de registro, un movimiento del robot o cualquier otra acción, el cliente necesitará realizar una serie de acciones que permitan a los clientes manejar al robot.
5. Pasos. Todos los pasos se muestran en los diagramas de secuencia que se adjuntan.
6. Condiciones. Para cualquiera de las ordenes que se pueden dar desde los clientes, es necesario que la aplicación esté en funcionamiento, y además las conexiones entre los extremos deben funcionar correctamente.
7. Resultado. El servidor de la aplicación recibirá una orden determinada.

Cerrar cliente

1. Nombre. Cerrar cliente.
2. Actores. Client.
3. Propósito. Englobar el conjunto de procesos que tienen como resultado final el cierre de un cliente.
4. Descripción. Cuando un cliente ya no desea manejar el robot éste se cerrará, de tal modo, que ya no podrá realizar acción alguna. Cuando el usuario pretenda volver a manejar el robot, y solo si la aplicación no ha sido cerrada, se tendrá que crear un nuevo cliente.
5. Pasos. Todos los pasos se muestran en los diagramas de secuencia que se adjuntan.
6. Condiciones. Para cualquiera de las ordenes que se pueden dar desde los clientes, es necesario que la aplicación esté en funcionamiento, y además las conexiones entre los extremos deben funcionar correctamente.
7. Resultado. El GUI que utilizaba el usuario para manejar el robot será cerrado. Pero hay que tener muy en cuenta, que tanto los enlaces como el resto de componentes de comunicaciones seguirán activos.

Diagrama 3: Casos de uso ServerManager

Iniciar aplicación

1. Nombre. Iniciar aplicación.
2. Actores. ServerManager.
3. Propósito. Englobar el conjunto de procesos que tienen como resultado la puesta en funcionamiento de la aplicación.
4. Descripción. Para que el servidor pueda ejecutar cualquier orden es necesario que antes se creen los enlaces y el resto de componentes de comunicaciones.
5. Pasos. Todos los pasos se muestran en los diagramas de secuencia que se adjuntan.
6. Condiciones. Para cualquiera de las ordenes que se pueden dar desde el servidor, es necesario que la aplicación esté en funcionamiento, y además las conexiones entre los extremos deben funcionar correctamente.
7. Resultado. El resulta a obtener que se pretende es que a partir de este instante, el servidor puede recibir ordenes de los clientes, mostrándose también las dos interfaces con el usuario que permiten realizar simulaciones de eventos.

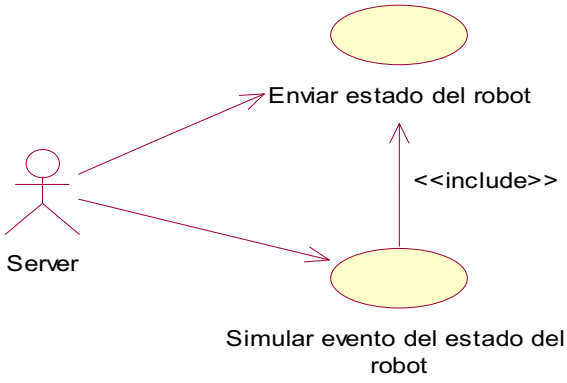


Diagrama 4: Casos de uso Server

Enviar estado del robot

1. Nombre. Enviar estado del robot.
2. Actores. Server.

3. Propósito. Englobar el conjunto de procesos que tienen como resultado final el envío a un cliente del estado del robot.
4. Descripción. Ya sea mediante el registro de los clientes o a través de una petición de tipo “get”, el servidor enviará su estado al cliente para que éste conozca la situación en la que se encuentra el robot.
5. Pasos. Los pasos que lleva a cabo el servidor, son similares a los de cualquier acción que puede realizar el servidor
6. Condiciones. Para cualquiera de las ordenes que se pueden dar desde el servidor, es necesario que la aplicación esté en funcionamiento, y además las conexiones entre los extremos deben funcionar correctamente.
7. Resultado. El cliente recibirá el estado del robot.

Simular evento del estado del robot

1. Nombre. Simular evento del estado del robot.
2. Actores. Server.
3. Propósito. Englobar el conjunto de procesos que tienen como resultado la simulación de un cambio de estado en el robot.
4. Descripción. La simulación de un evento de estas características permite al cambiar de forma virtual es estado del robot, se produzca una actualización en los clientes que estén “registrados mediante eventos”.
5. Pasos. Todos los pasos se muestran en los diagramas de secuencia que se adjuntan.
6. Condiciones. Para cualquiera de las ordenes que se pueden dar desde el servidor, es necesario que la aplicación esté en funcionamiento, y además las conexiones entre los extremos deben funcionar correctamente.
7. Resultado. Al simular este evento todos los clientes registrados recibirán su actualización. En cambio, si no hay ningún cliente registrado para este efecto, el resultado será nulo.

6.2. Vista Lógica.

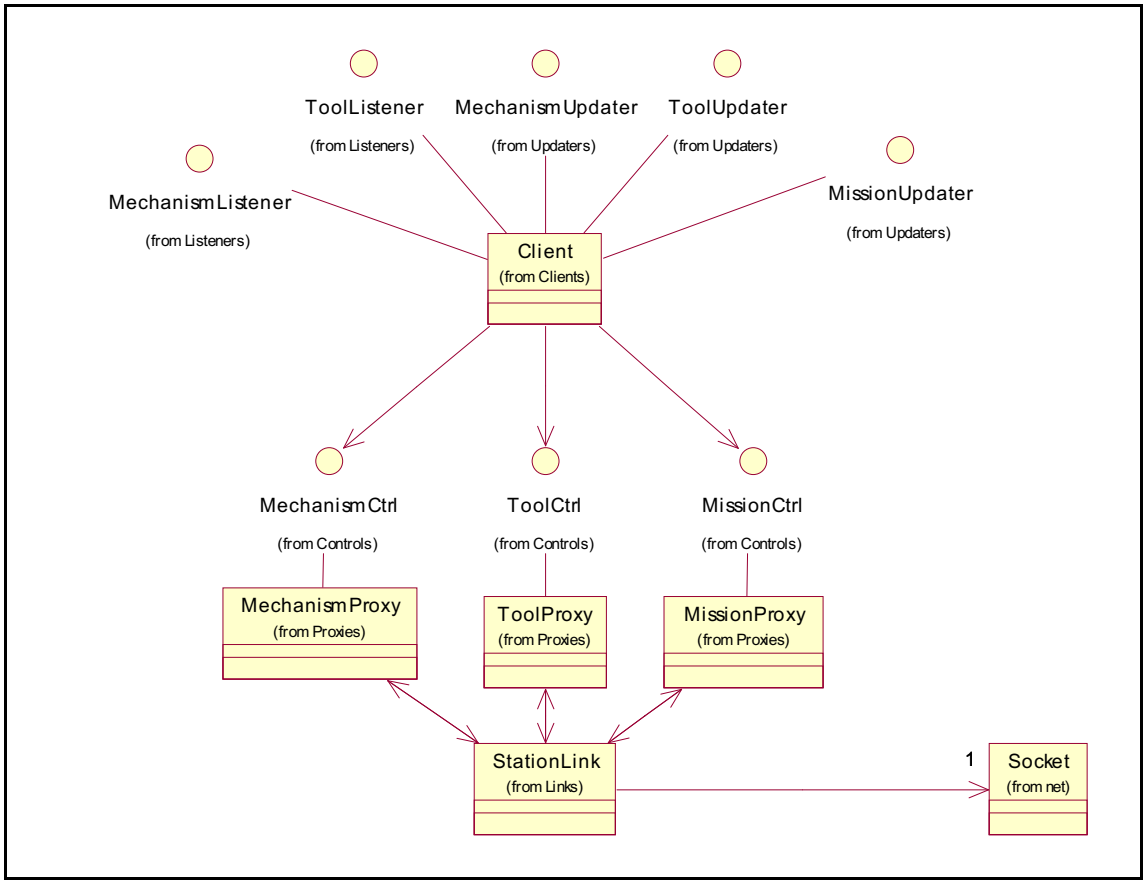


Diagrama 5: Arquitectura cliente

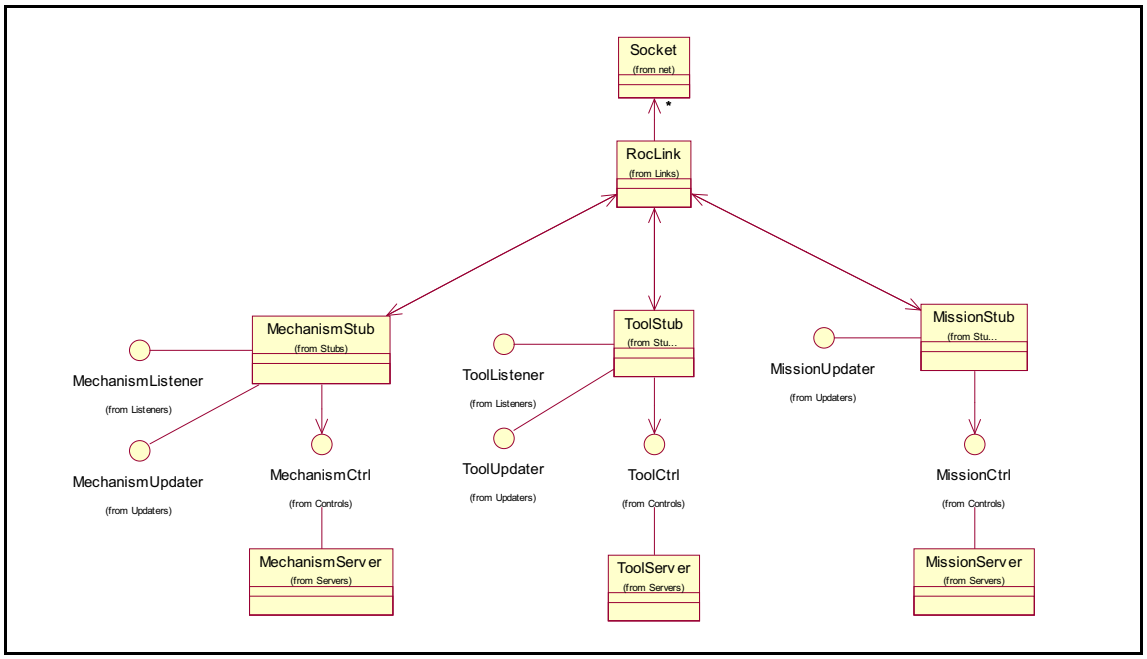


Diagrama 6: Arquitectura servidor

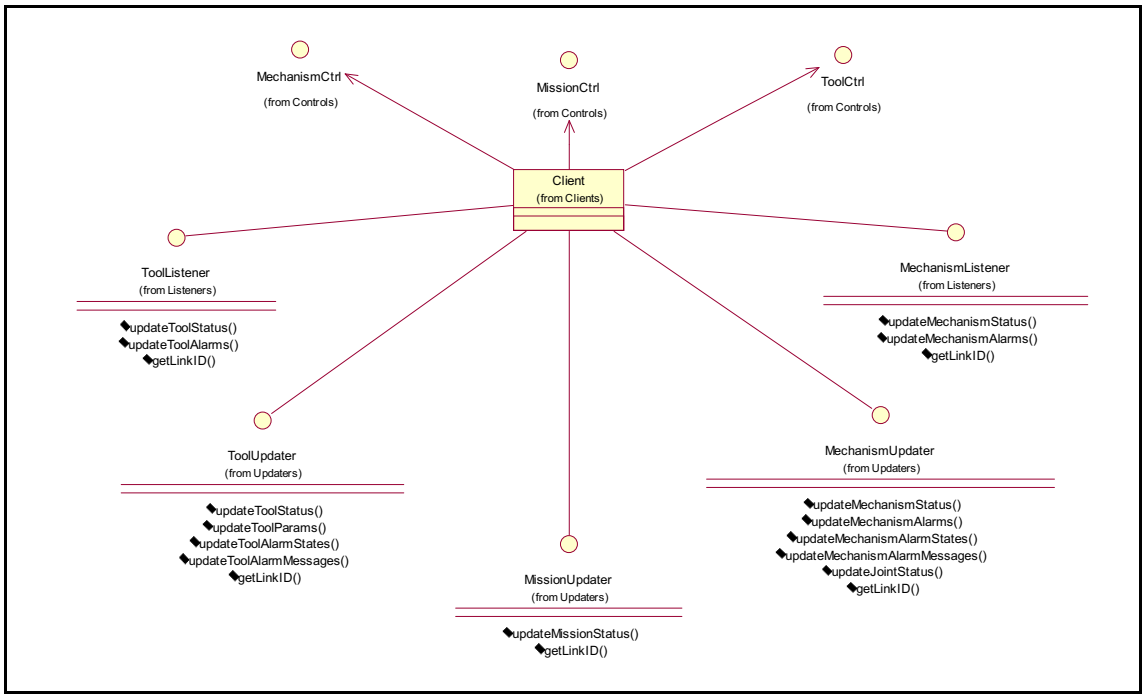


Diagrama 7: Cliente

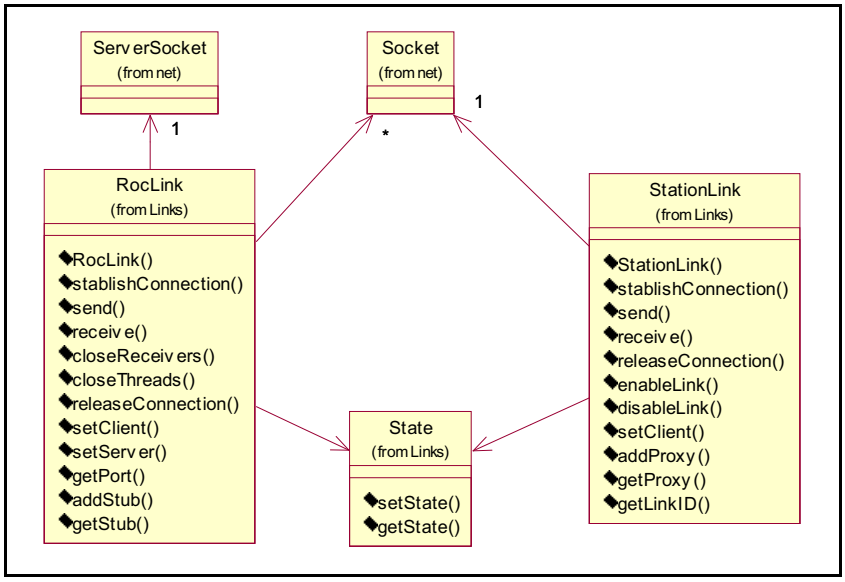


Diagrama 8: Enlaces

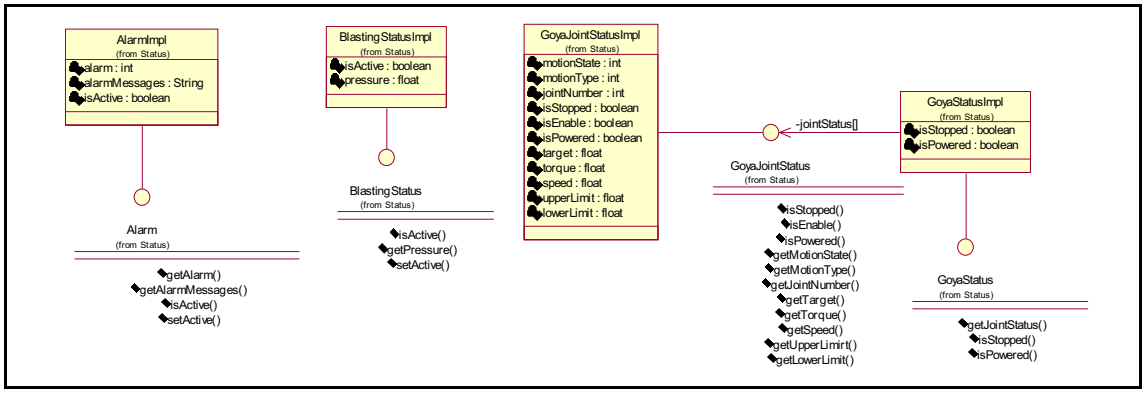


Diagrama 9: Estado robot

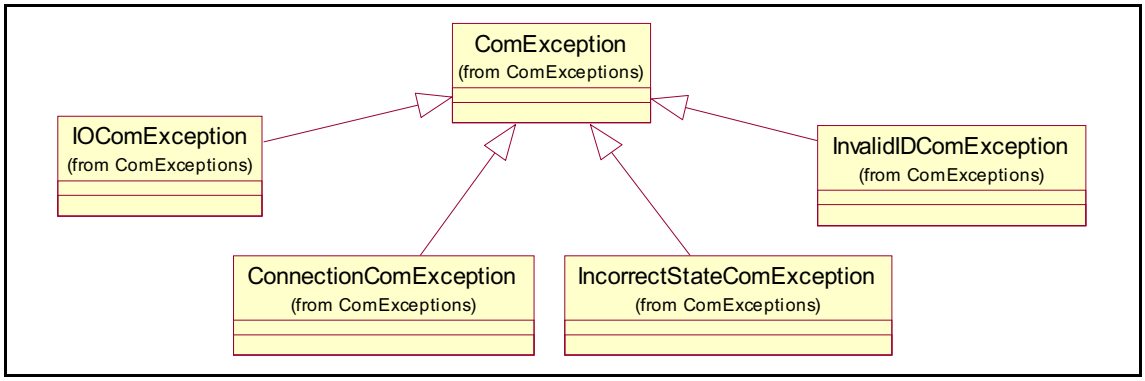


Diagrama 10: Excepciones

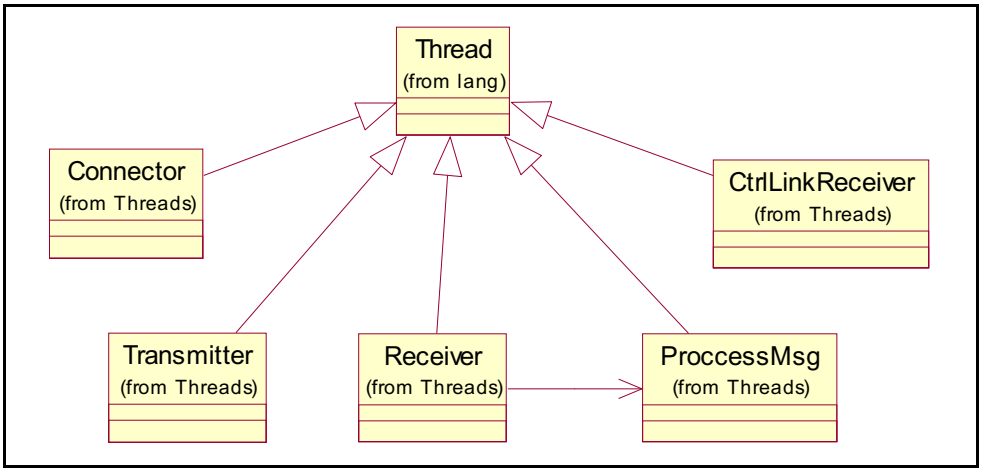


Diagrama 11: Hilos

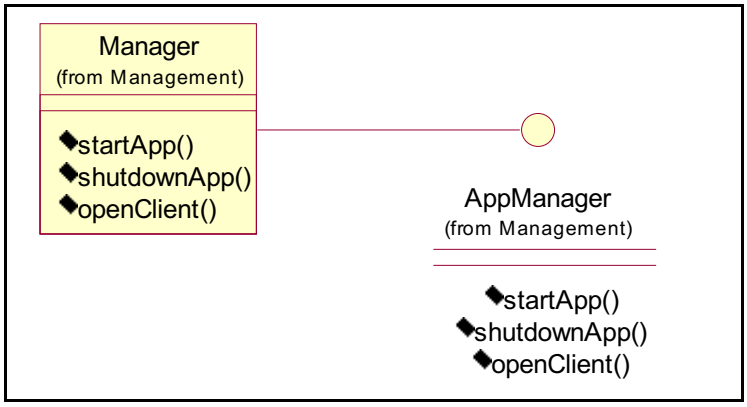


Diagrama 12: Manager

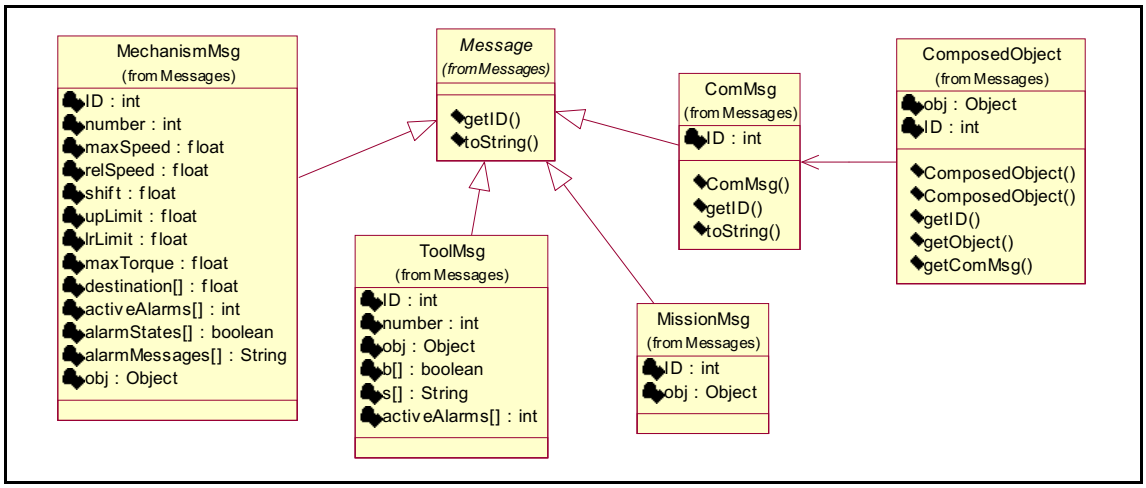


Diagrama 13: Mensajes

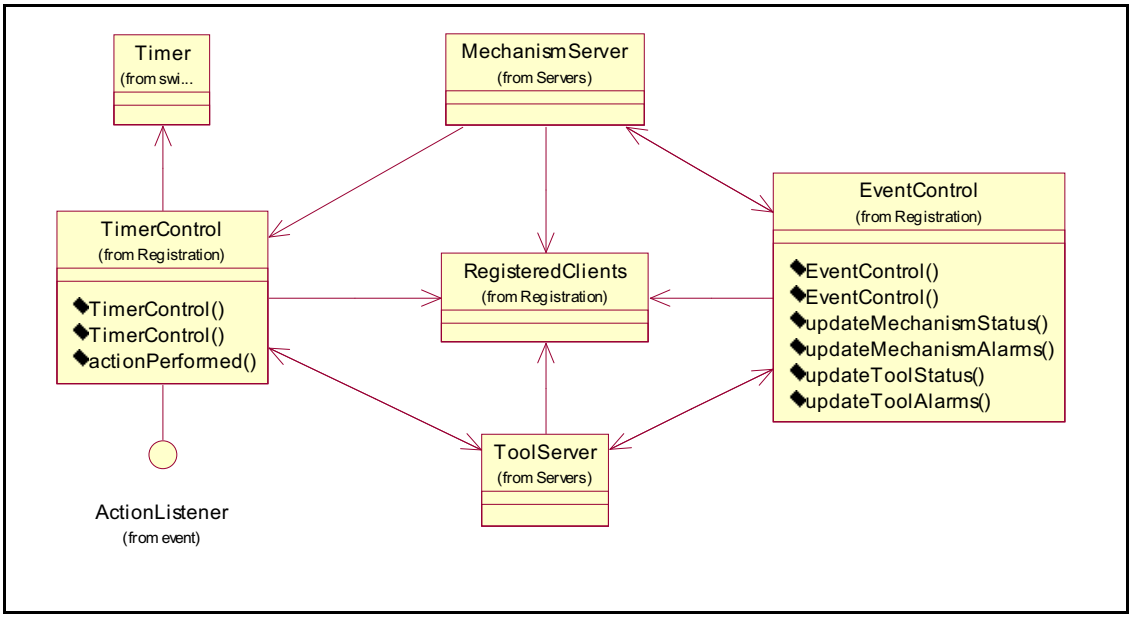


Diagrama 14: Registro

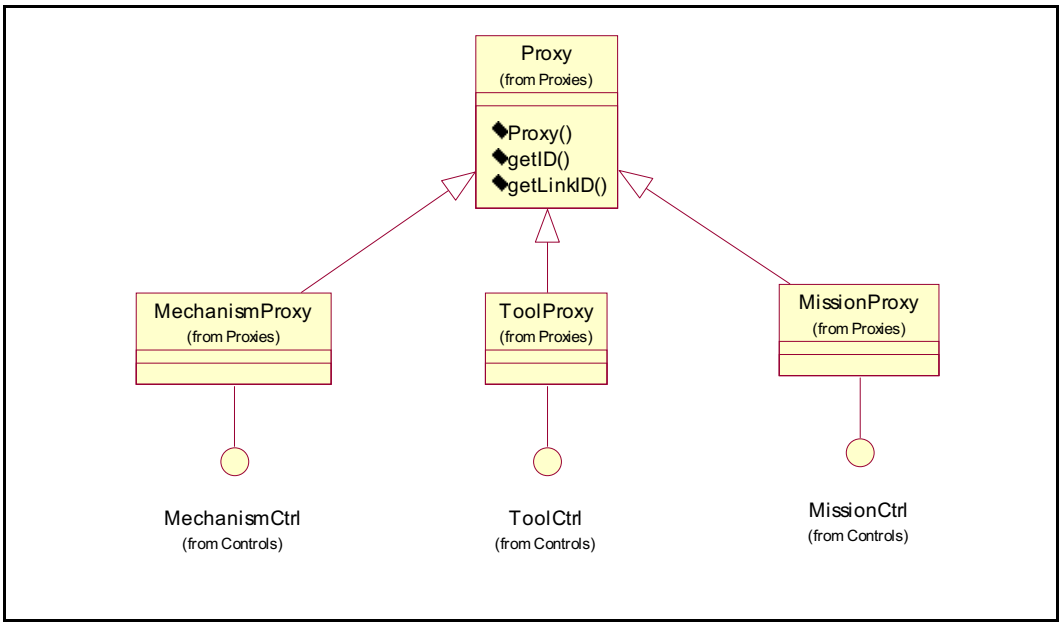


Diagrama 15: Proxies

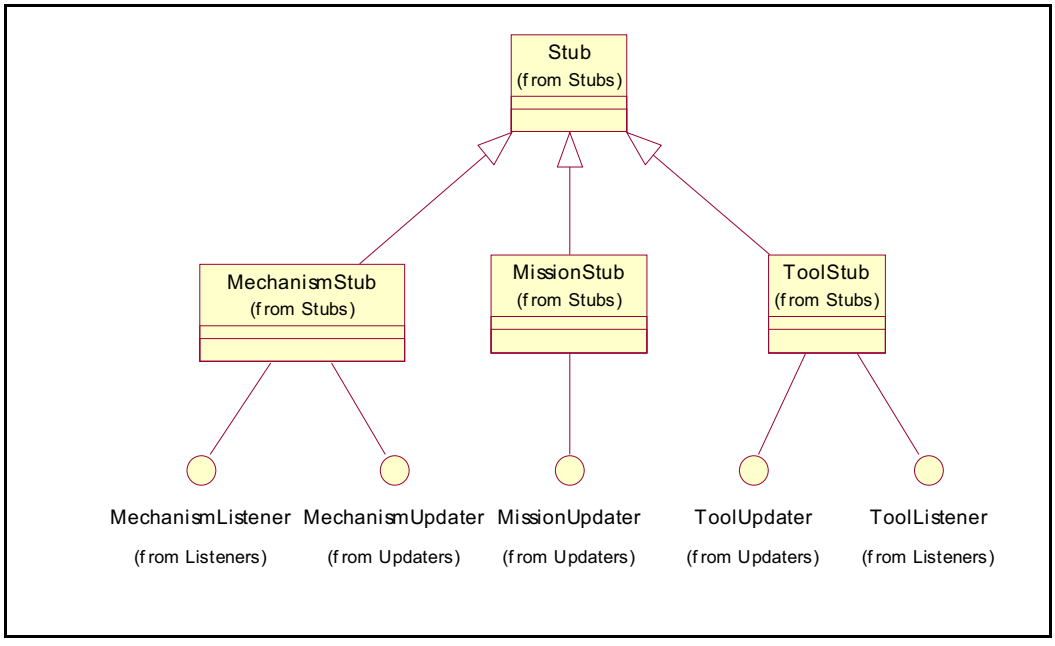


Diagrama 16: Stubs

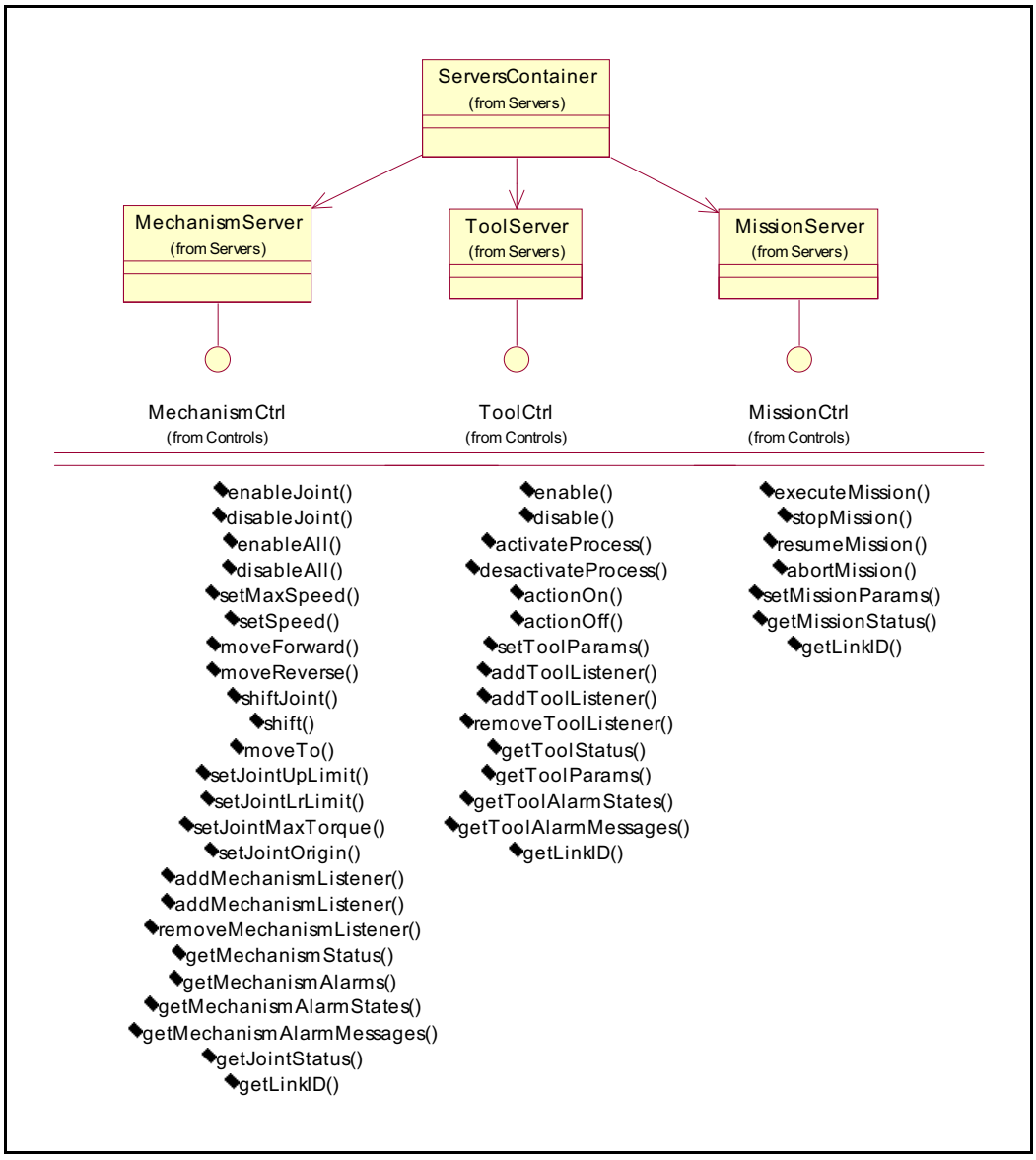


Diagrama 17: Servidores

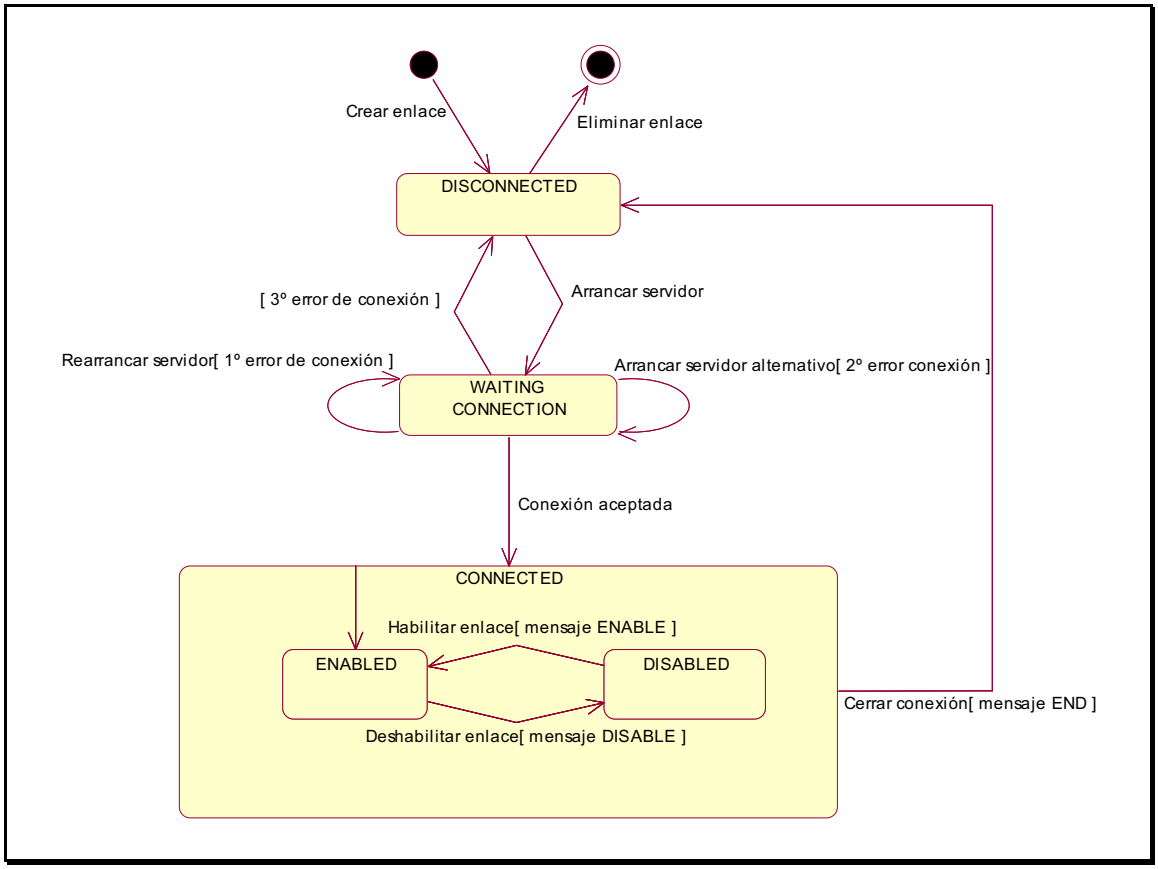


Diagrama 18: Servidor (clases RocLink y CtrlLinkRoc)

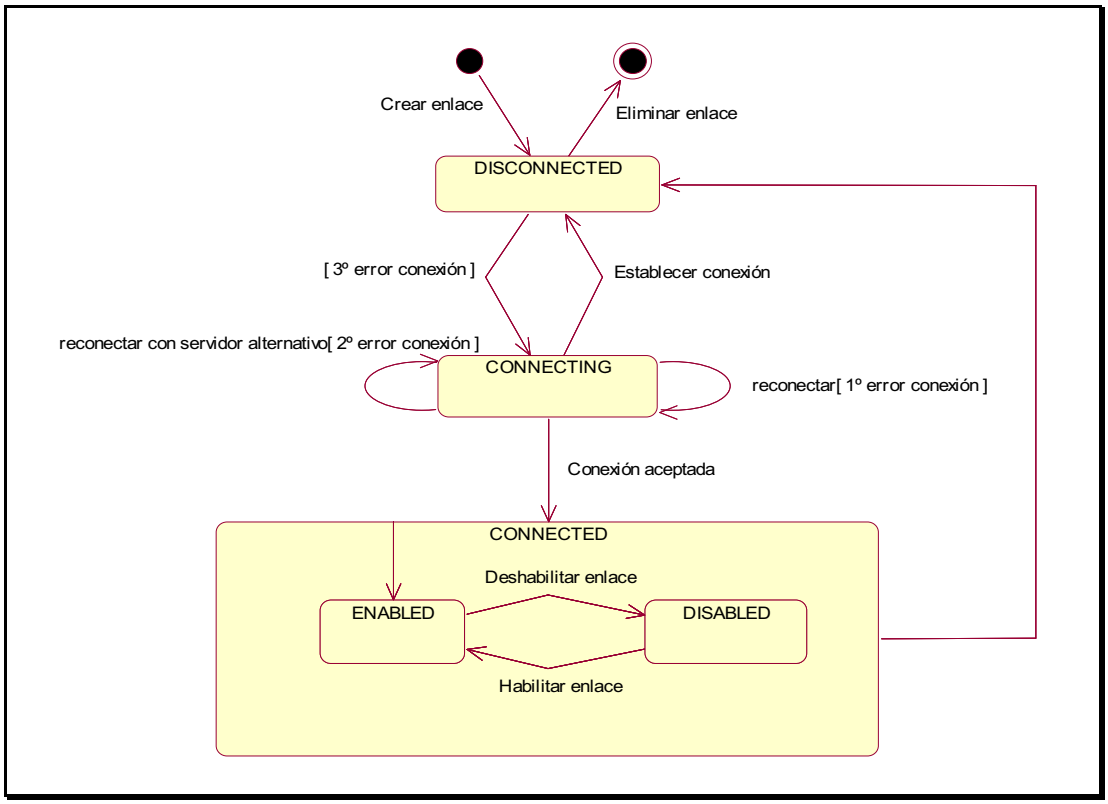


Diagrama 19: Cliente (clases StationLink y CtrlStationLink)

6.3. Vista de Implantación.

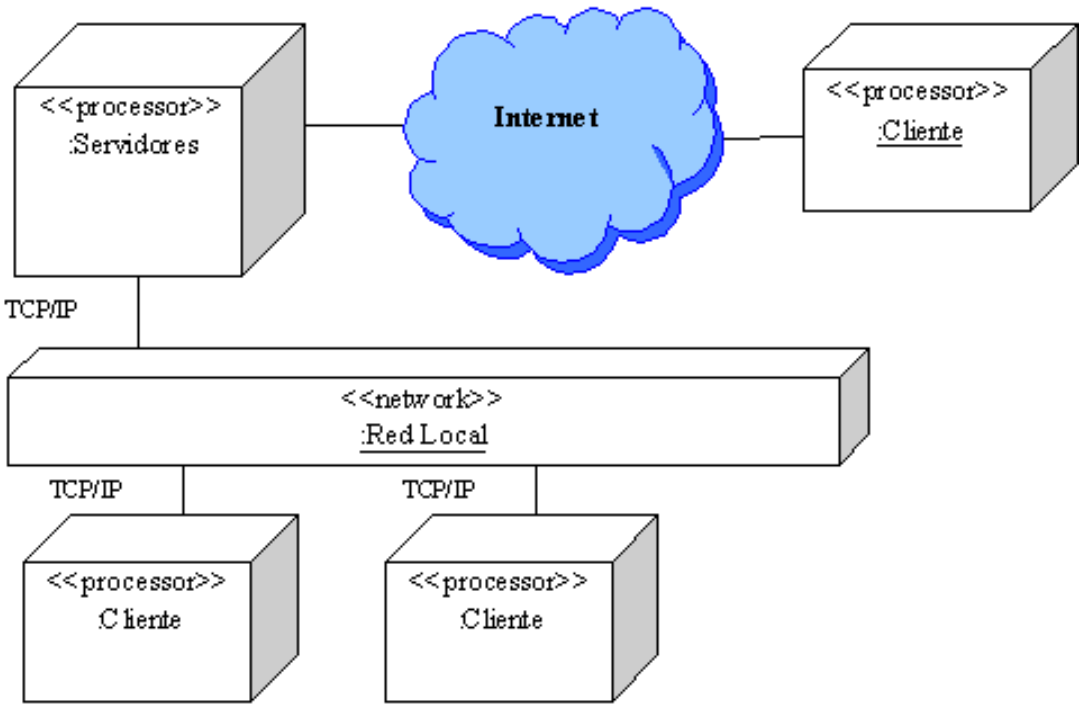


Diagrama 20: Ejemplo de arquitectura

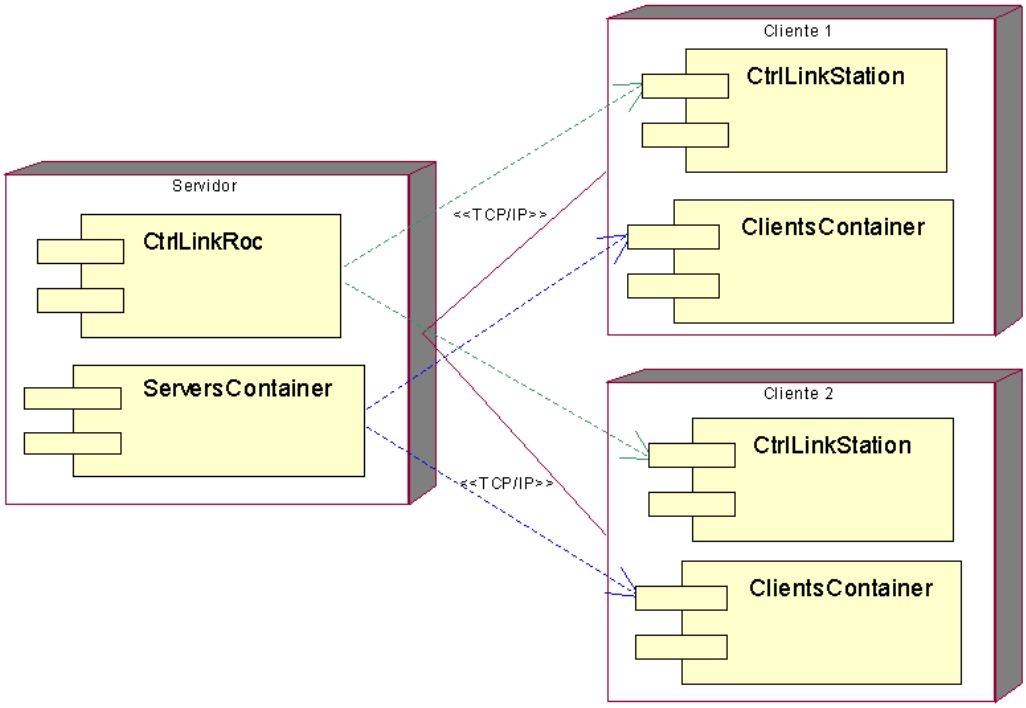


Diagrama 21: Despliegue del sistema

6.4. Vista de Concurrencia.

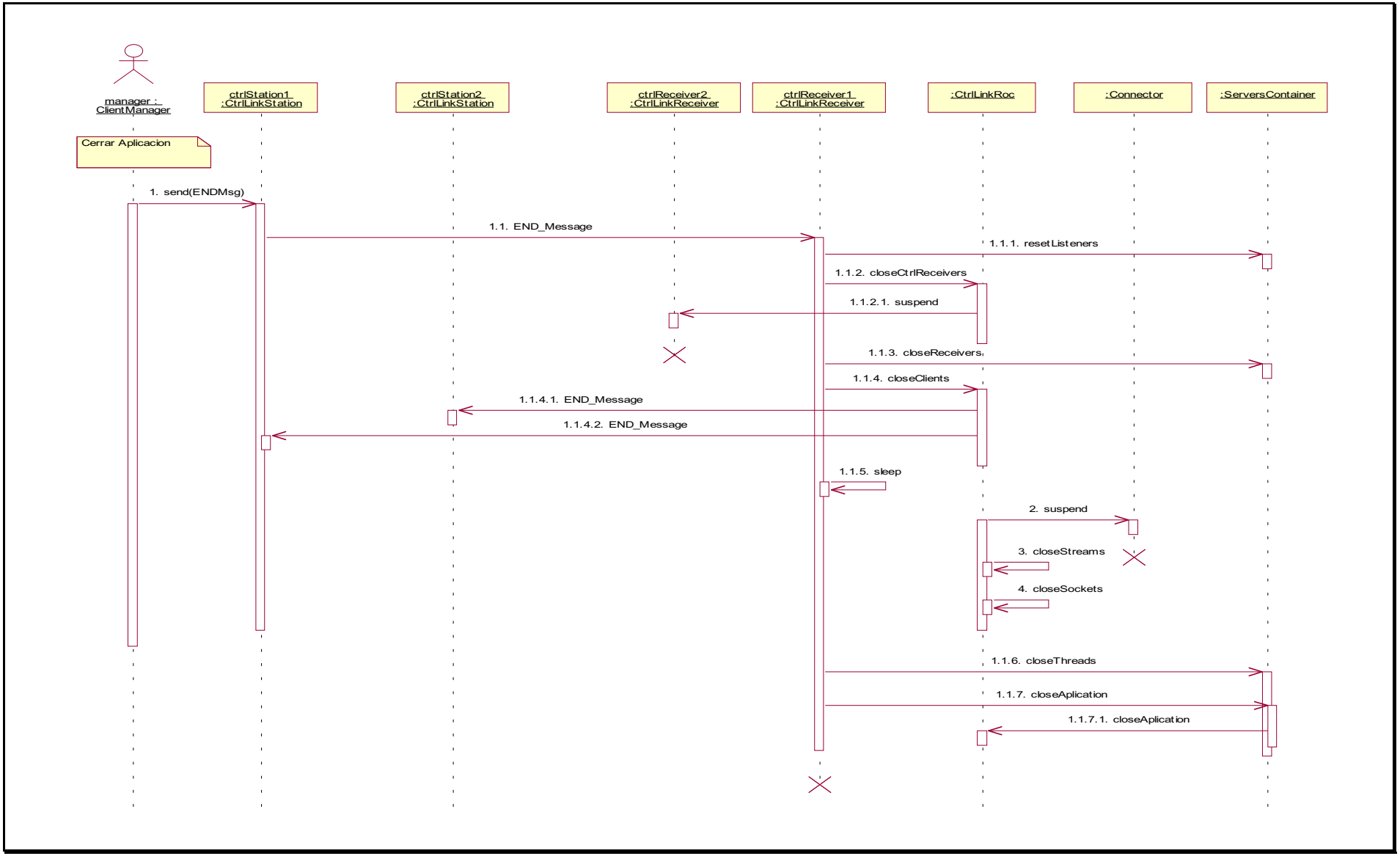


Diagrama 22: Cerrar aplicación (general)

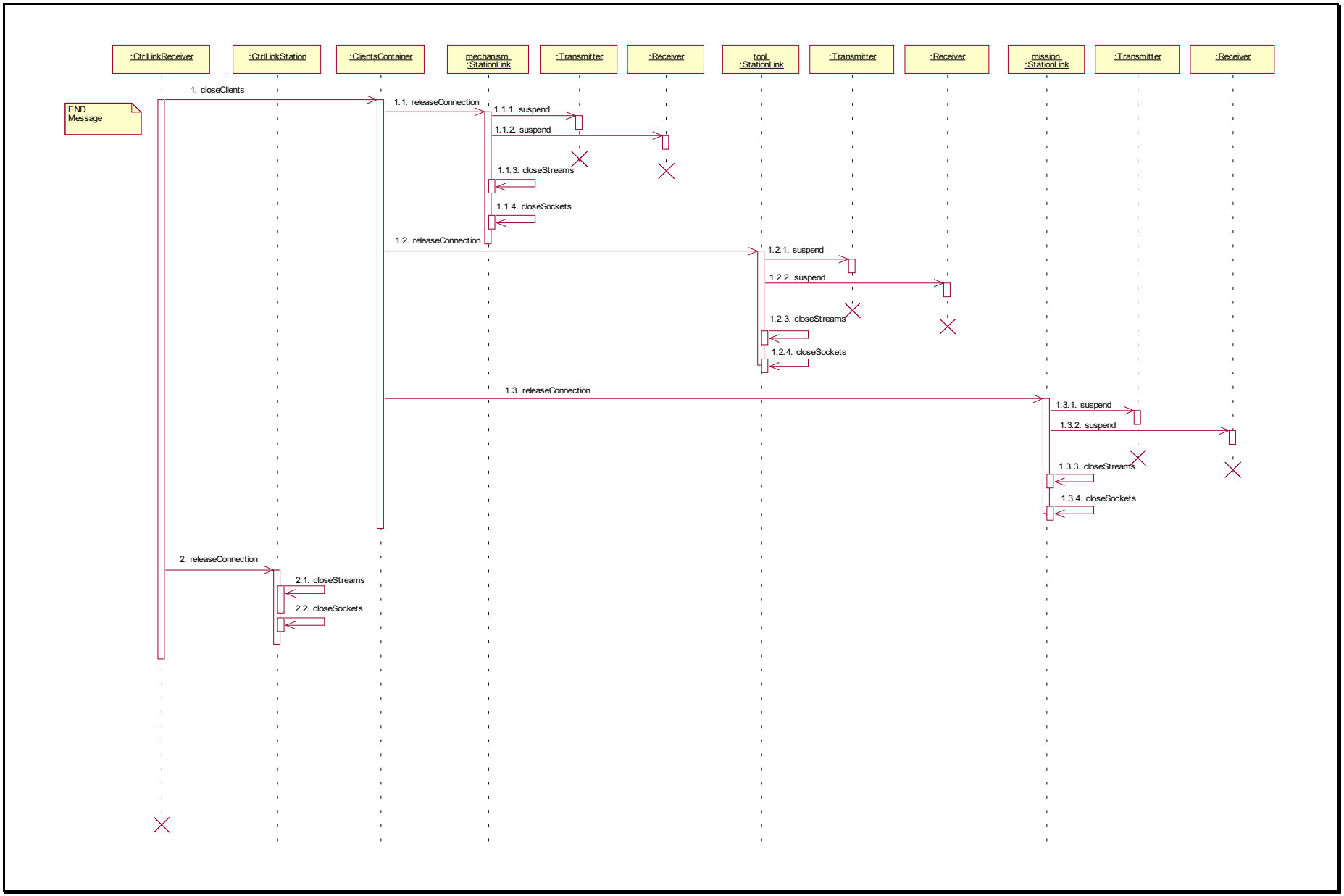


Diagrama 23: Cerrar aplicación (cliente)

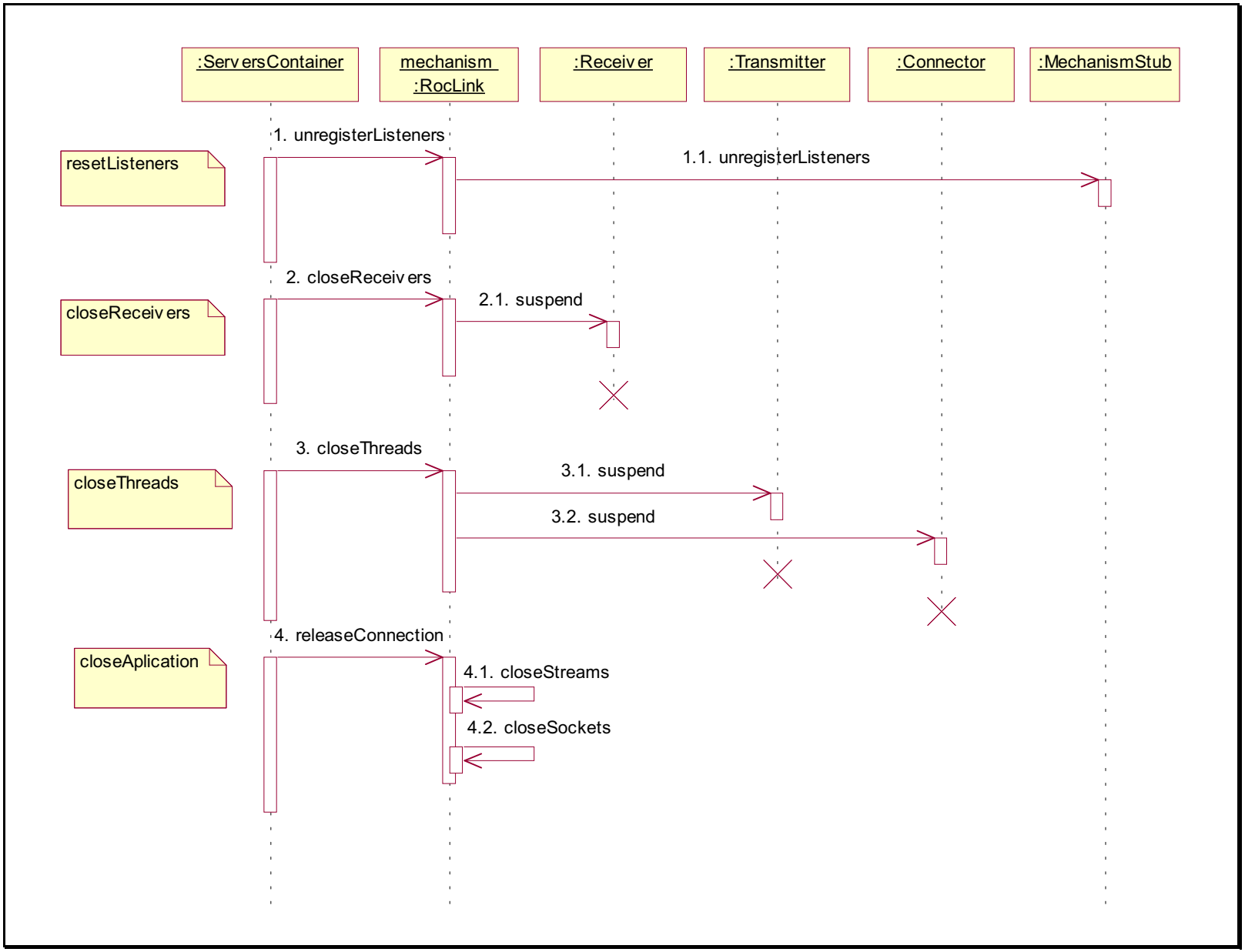


Diagrama 24: Cerrar aplicación (servidor)

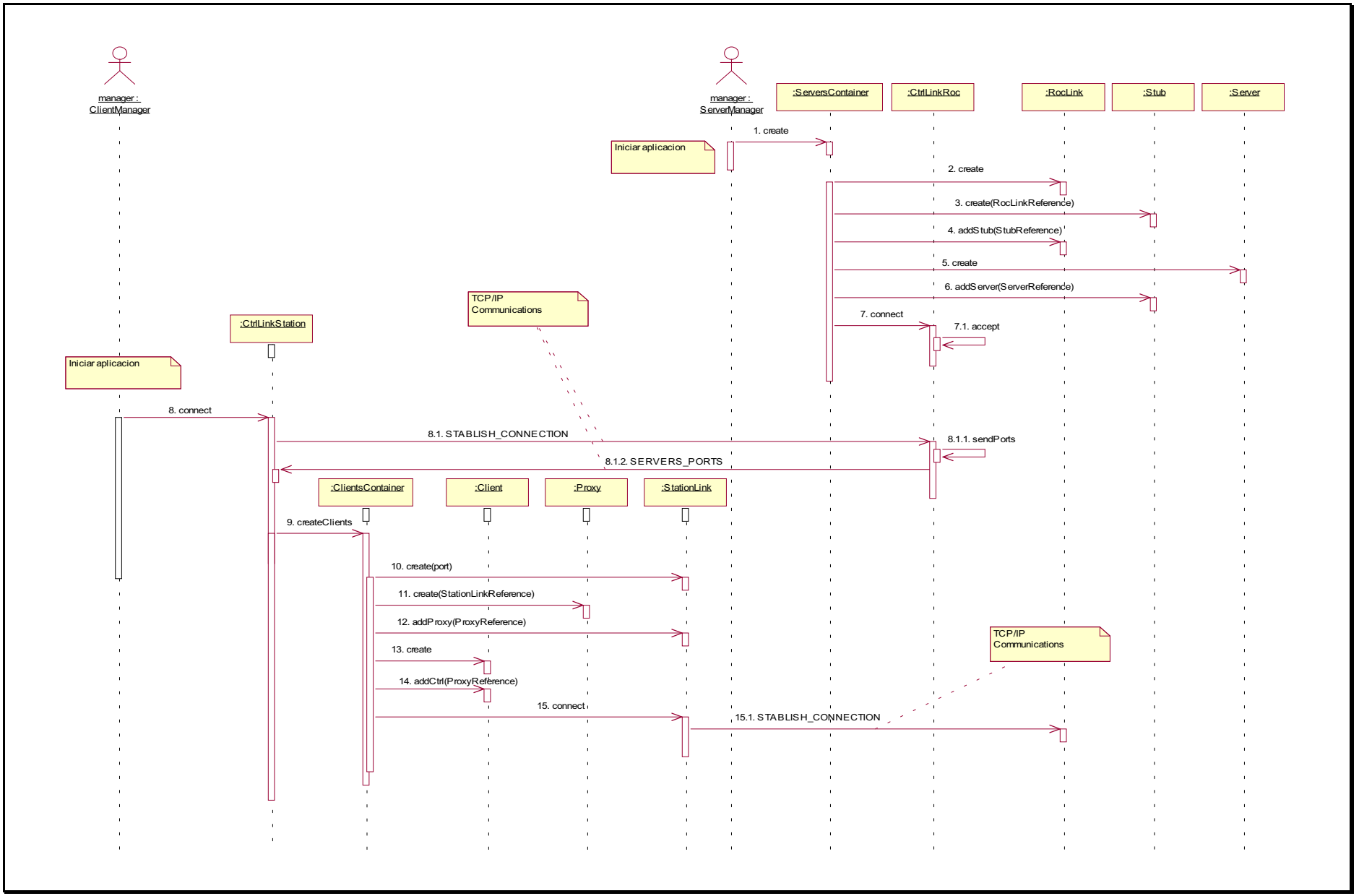


Diagrama 25: Inicio de la aplicación

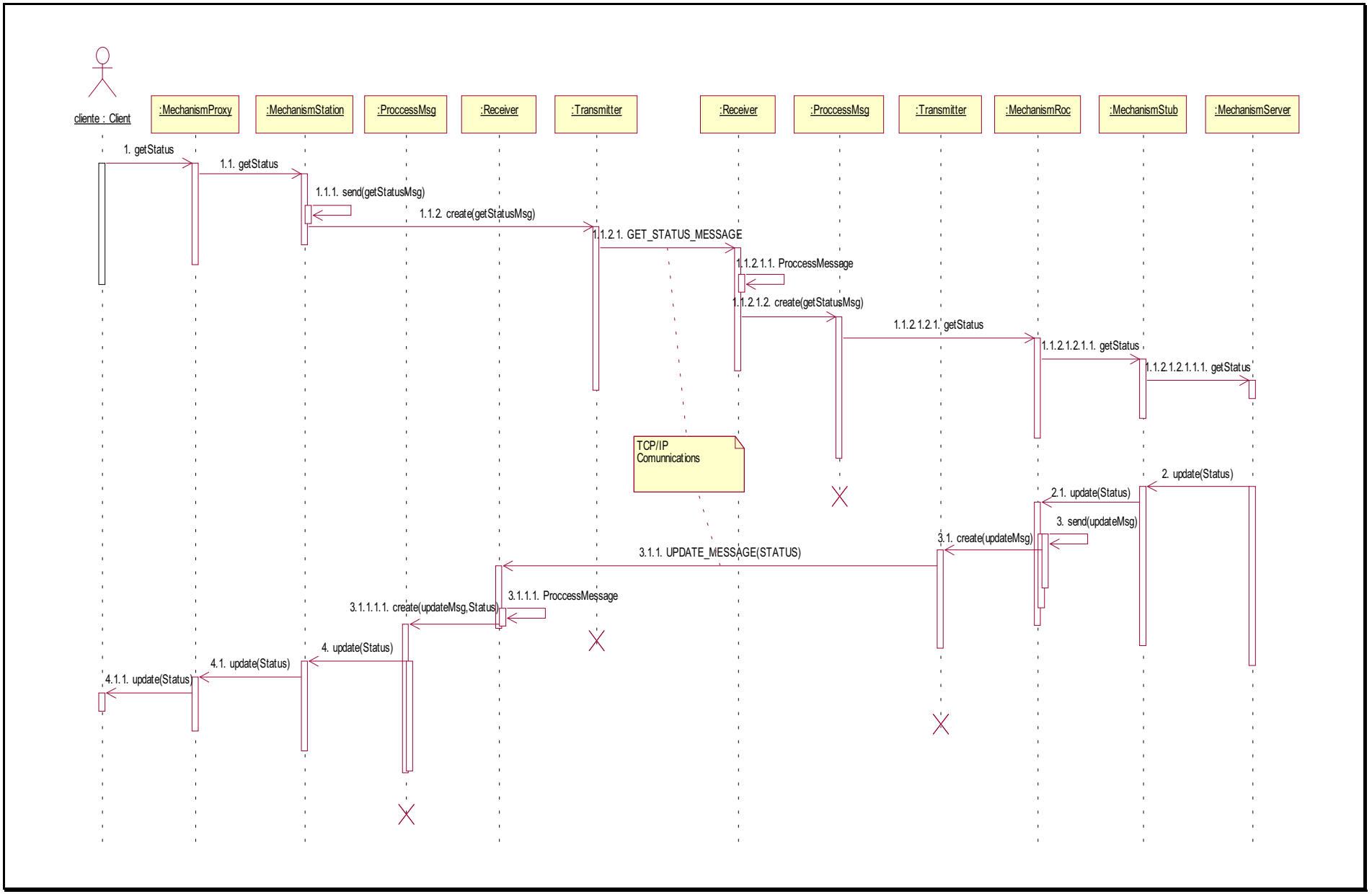


Diagrama 26: Enviar mensaje

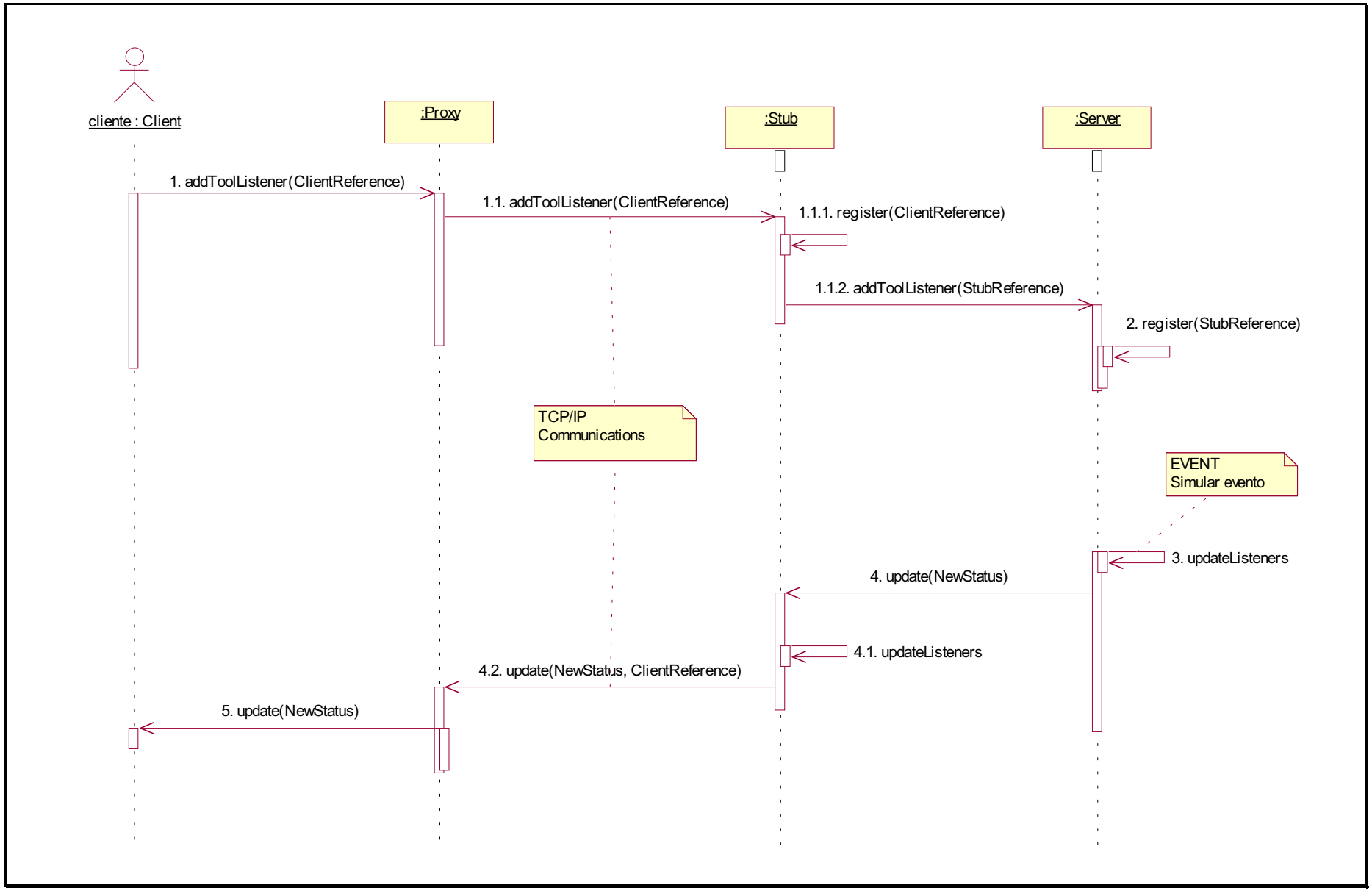


Diagrama 27: Registro con eventos

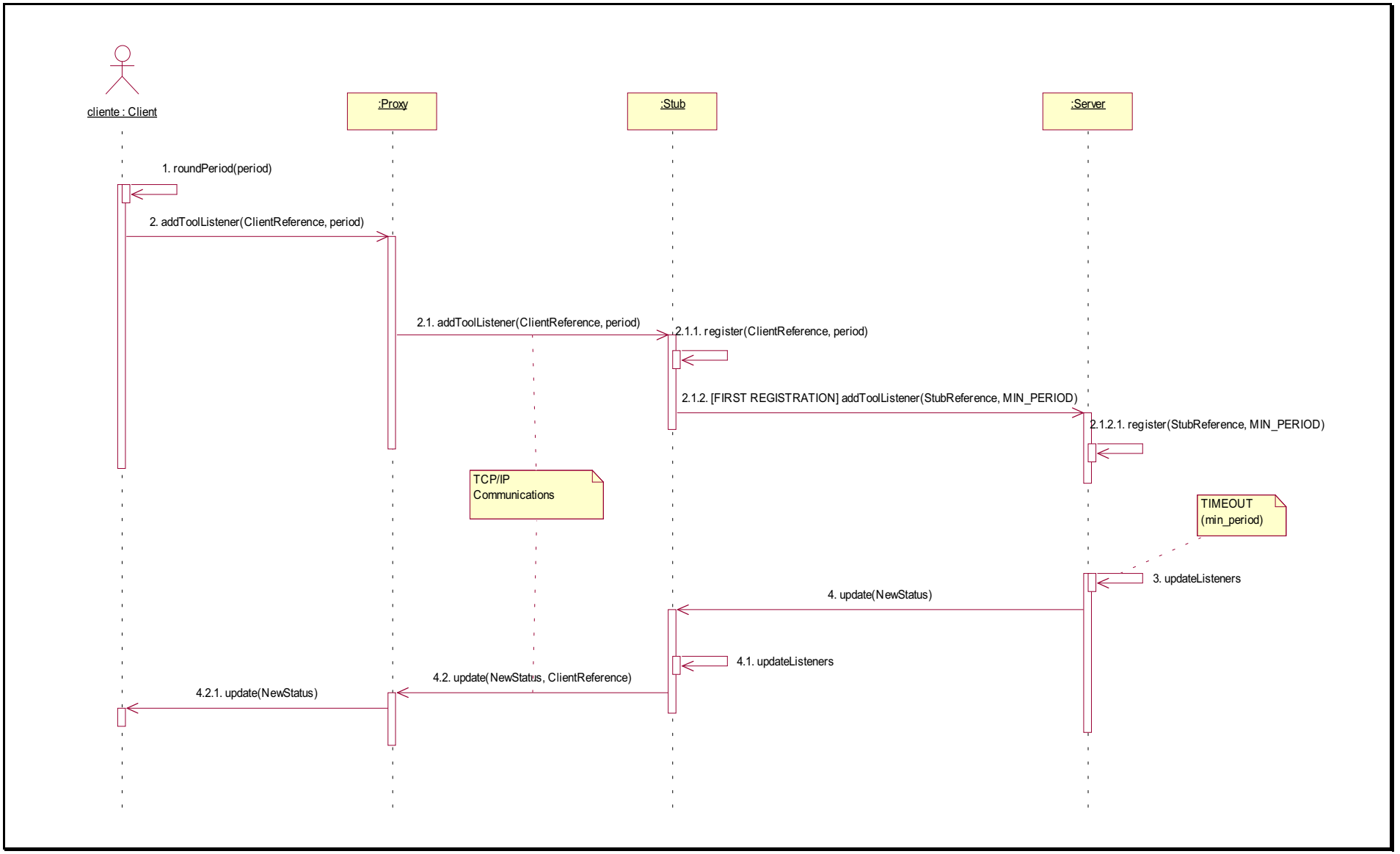


Diagrama 28: Registro con periodo de tiempo

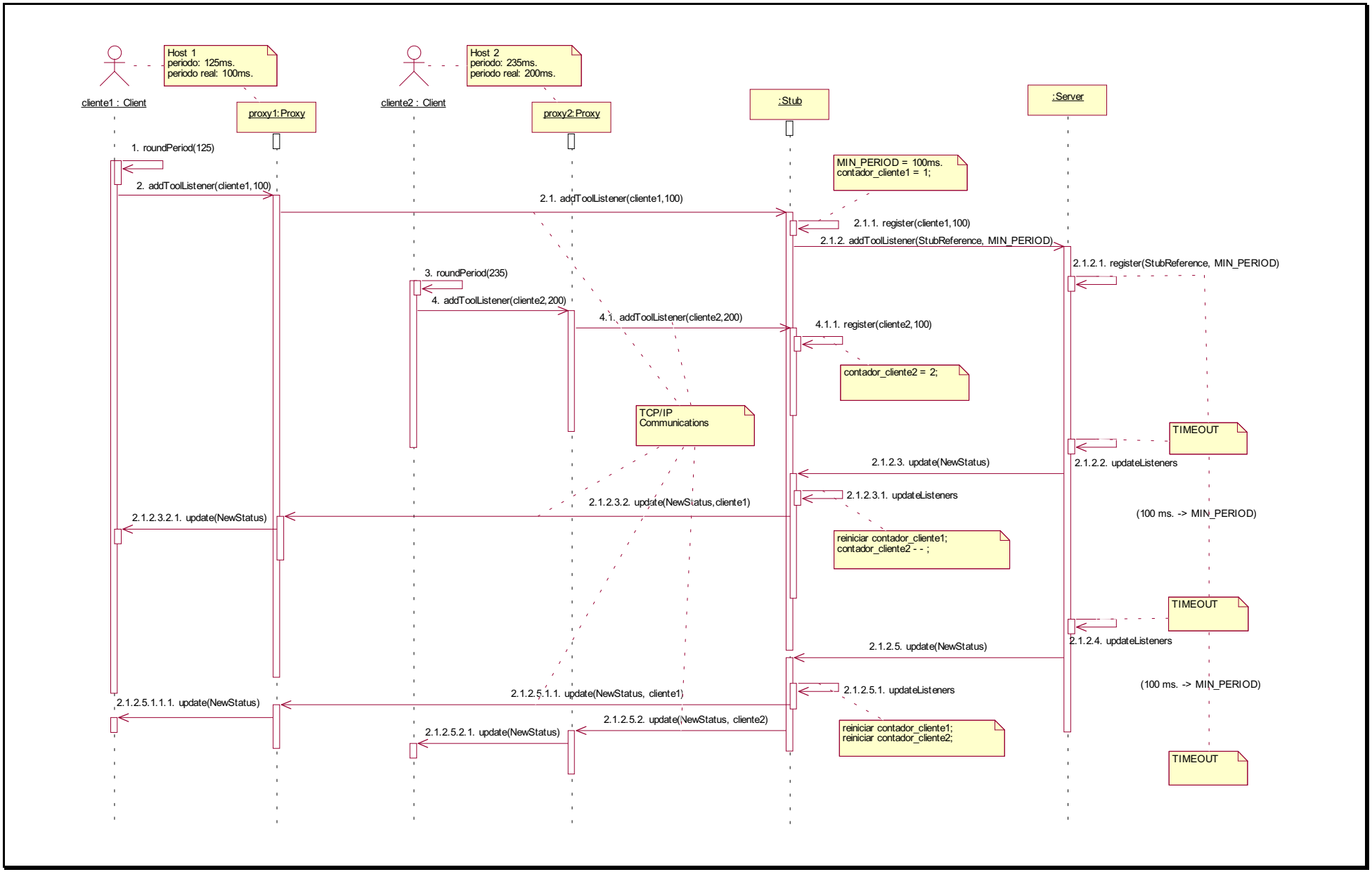


Diagrama 29: Registro con periodo de tiempo (2 clientes)

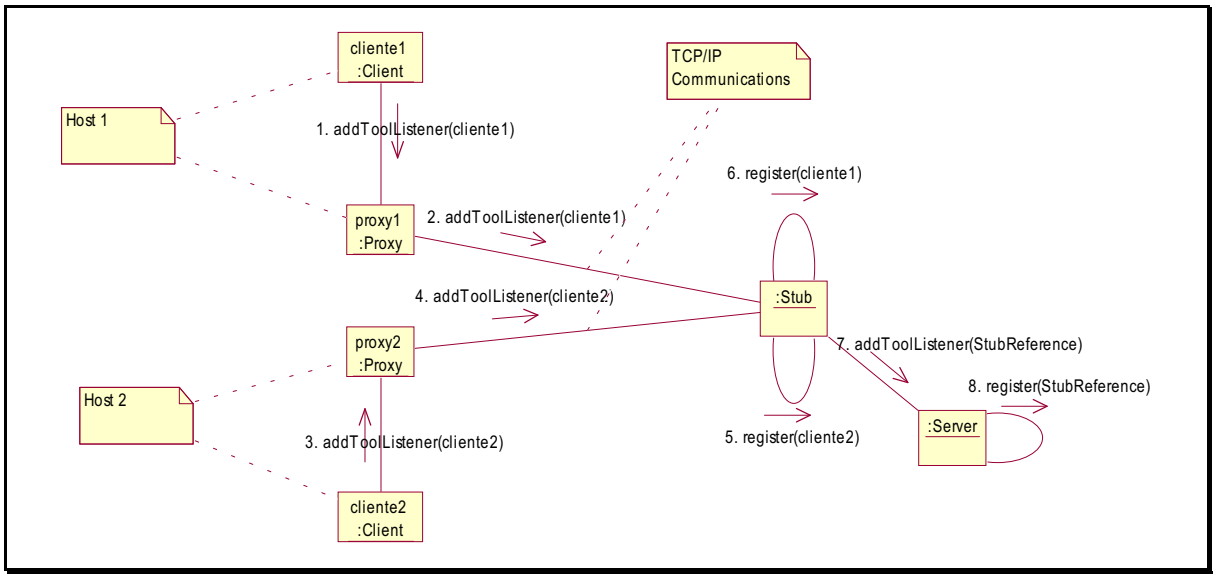


Diagrama 30: Registro con eventos

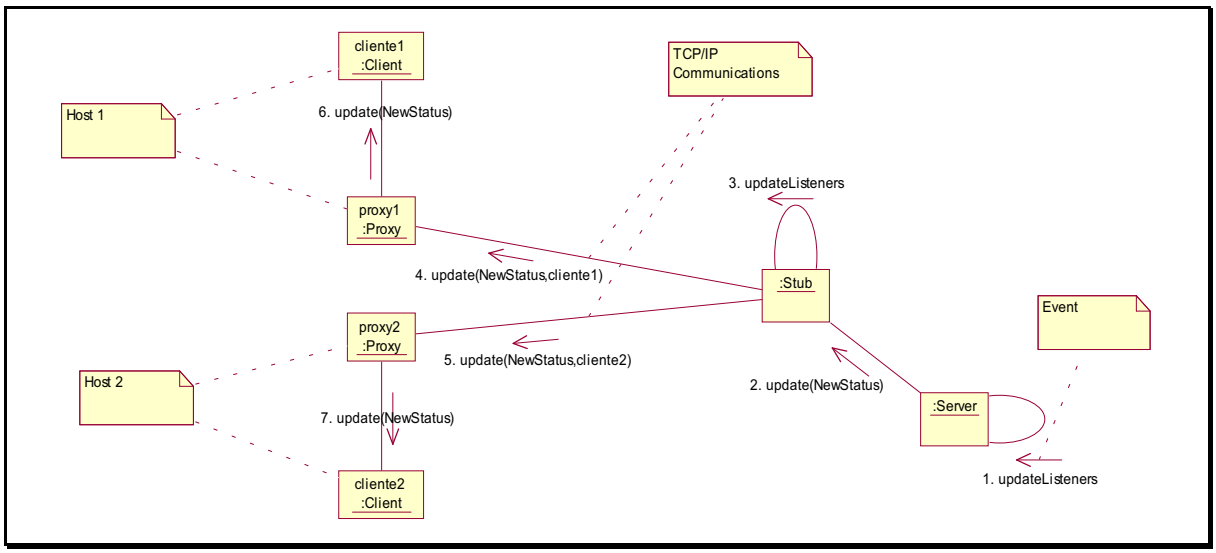


Diagrama 31: Actualización de clientes registrados

6.5. Vista de Componentes.

Los diagramas asociados a la vista de componentes se encuentran en el apartado "Dependencias significativas", donde se presentan las dependencias a destacar entre los componentes que forman la aplicación, tanto paquetes como clases.

7. Arquitectura de la aplicación.

7.1. Organización de la aplicación.

Para la realización de la aplicación se optó por la distribución de todas las clases en 13 paquetes: Controls, Updaters, Listeners, Clients, Servers, Management, Protocol, Proxies, Stubs, ComExceptions, Status, IDs y GUIs. A su vez, estos paquetes estarán localizados dentro de un paquete general llamado Sockets.

Esta organización de la aplicación es debida a que en Java es posible agrupar varias clases en una estructura llamada paquete. Un paquete no es más que un conjunto de clases, generalmente relacionadas entre sí de alguna manera. Es habitual diseñar una aplicación distribuyendo su funcionalidad entre varios paquetes, cuyas clases se comunican entre sí a través de interfaces bien definidas.

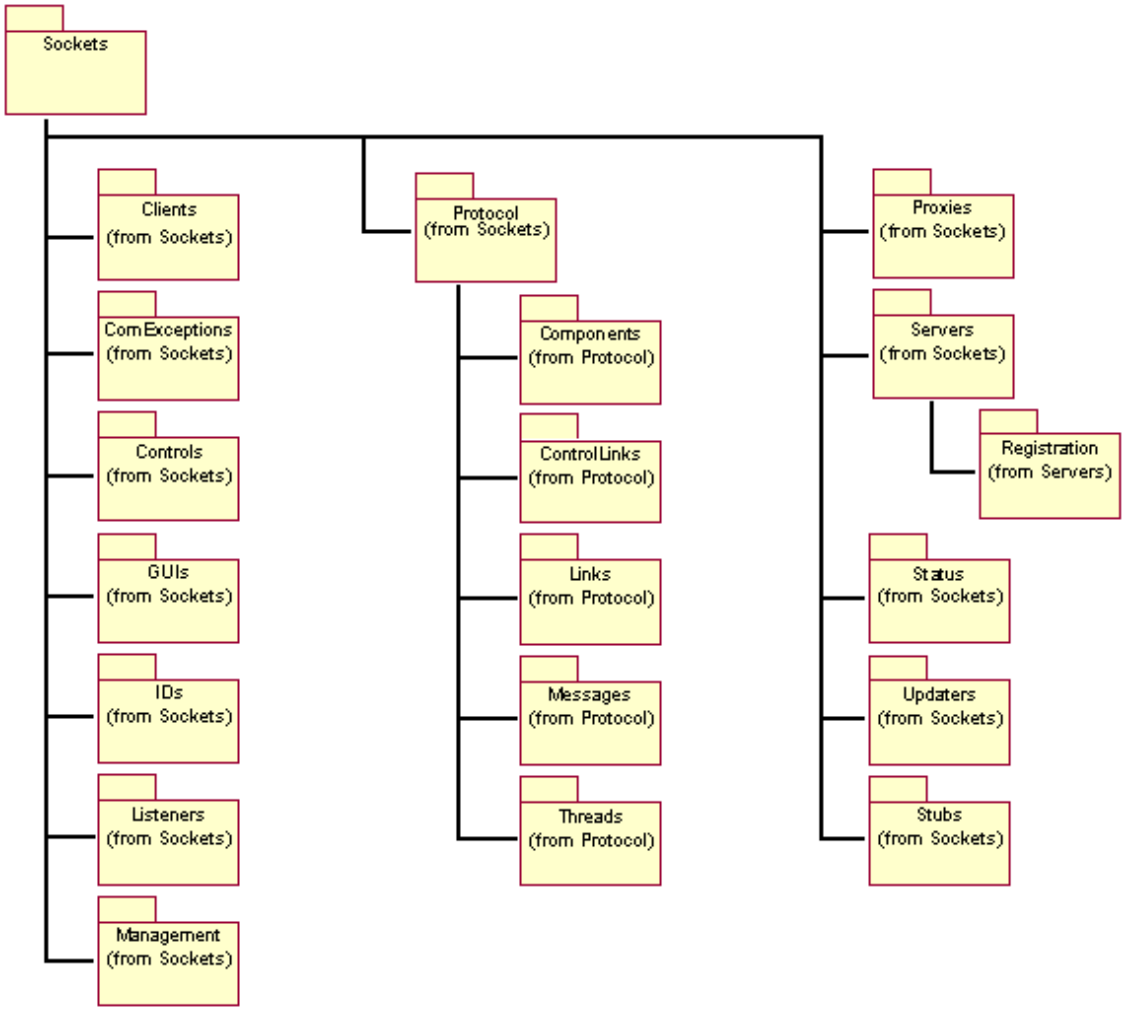
El uso de paquetes aporta varias ventajas frente a la programación sin paquetes. En primer lugar, permite encapsular funcionalidad en unidades con un cierto grado de independencia, ocultando los detalles de implementación. De esta forma se pueden conseguir diseños (e implementaciones) más limpios y elegantes.

Por otra parte, se potencia la reutilización de las clases desarrolladas. Es posible definir interfaces de uso de cada paquete, para que otros paquetes o aplicaciones puedan utilizar la funcionalidad implementada.

Además, el uso de paquetes permite la reutilización de los nombres de las clases, ya que el espacio de nombres de un paquete es independiente del de otros paquetes. El lenguaje Java impone la restricción de que una clase debe tener un nombre único, dentro del paquete al cual pertenece. Sin embargo, es posible que dos clases tengan el mismo nombre, siempre y cuando pertenezcan a paquetes distintos.

El uso de paquetes en Java nos proporciona ventajas en cuanto a encapsulamiento, reusabilidad, independencia de espacios de nombres, etc. Únicamente se exige respetar la estructura jerárquica de los directorios, tanto para almacenar los ficheros .class como a la hora de hacer las invocaciones a las aplicaciones.

Una vez que sabemos las ventajas de la organización en paquetes de la estructura de un software Java, vamos a estudiar la estructura concreta de la aplicación en cuestión. Para una mejor visualización de la distribución de clases de la aplicación, se muestra a continuación la jerarquía de paquetes del sistema:



Una vez conocida la estructura del código de la aplicación, se comentan a continuación las finalidades que tiene cada paquete, aunque el nombre asignado a cada uno de ellos nos facilita la tarea. No obstante, se comentara de un modo sencillo y rápido para poder hacerse una idea general del contenido de la aplicación. Seguidamente se procederá a una explicación en profundidad de cada clase contenida en los paquetes en cuestión. De este modo, tenemos los siguientes paquetes:

- **Controls**: Interfaces que implementaran los servidores del robot.
- **Updaters**: Interfaces que implementará el cliente para poder recibir respuestas del servidor.
- **Listeners**: Interfaces que implementará el cliente para registrarse en el servidor.

- **Clients**: Implementación de los clientes.
- **Servers**: Implementación de los servidores de control del robot.
- **Servers.Registration**: Permite el servicio de registro de clientes en los servidores, para que sean informados del estado del robot automáticamente cada vez que se modifique el estado, o cada cierto periodo de tiempo.
- **Management**: Permite iniciar y cerrar la aplicación desde un cliente, e incluso abrir más clientes.
- **Protocol**: Protocolo que se usa para comunicar clientes y servidores a través de un enlace de comunicaciones TCP.
- **Protocol.Links**: Permite crear un enlace de comunicaciones entre cliente y servidor para controlar el robot.
- **Protocol.ControlLinks**: Proporciona clases para crear un enlace de comunicaciones entre cliente y servidor que permita gestionar y controlar el protocolo y la aplicación.
- **Protocol.Messages**: Conjunto de mensajes que se envían a través de los enlaces de comunicaciones creados por la aplicación. Cada mensaje tiene su correspondiente orden asociada en los extremos cliente y servidor.
- **Protocol.Threads**: Hilos encargados de realizar las operaciones de comunicaciones que puedan resultar bloqueantes para el sistema.
- **Protocol.Components**: Utilidades necesarias para el funcionamiento del protocolo.
- **Proxies**: Permitirán ocultar al cliente todo el protocolo que facilita la distribución de los servidores.
- **Stubs**: Ocultaran al servidor los aspectos de distribución de los clientes.
- **ComExceptions**: Excepciones de definición propia que pueden lanzarse en la aplicación.
- **Status**: Contiene las clases que almacenaran información acerca del estado del robot.
- **IDs**: Contiene las clases que recogen todos los parámetros de configuración de la aplicación.
- **GUIs**: Interfaces gráficas usadas para la simulación del sistema.

7.2. Descripción de la aplicación.

Ahora se conoce la finalidad de cada paquete de la jerarquía, por lo que podemos entrar ya en detalle con las clases que se encuentran en dichos paquetes para describir en detalle la estructura de la aplicación resumida en el punto anterior.

- **Controls**:

El paquete Controls contiene 3 clases: MechanismCtrl, MissionCtrl y ToolCtrl, que son las interfaces que definen los métodos encargados del control del mecanismo y herramientas del robot en cuestión. Es por esto, por lo que la interfaz MechanismCtrl será implementada por el servidor MechanismServer; ocurriendo lo mismo para las interfaces MissionCtrl y ToolCtrl con los servidores MissionServer y ToolServer respectivamente.

Como se puede observar, todos los métodos de estas interfaces, deben tener cláusulas donde se relancen las excepciones de tipo ComException, que representa la superclase que abarca todas las excepciones que pueden ocurrir durante la ejecución de la aplicación.

En cuanto a los parámetros que pueden tener estos métodos, vemos como no hay ninguna limitación en cuanto al número de éstos, ni al tipo, ya que la única condición a cumplir es que implementen la interfaz Serializable para poder enviarlos al servidor a través de la red.

- **Updaters**:

Este paquete es muy similar al comentado anteriormente, en cuanto que también contiene 3 clases: MechanismUpdater, MissionUpdater y ToolUpdater y ambas son interfaces. Además, todos los métodos de estas interfaces también relanzan las excepciones de tipo ComException.

La principal diferencia de estas interfaces con las del paquete anterior, es que éstas definen métodos que llamará el servidor para comunicarse con el cliente, y que por tanto, deberán ser implementados por el cliente (clase Client), mientras que los métodos de las interfaces del paquete Controls los llamaban los clientes para comunicarse con los servidores (los métodos eran implementados por los servidores).

En cuanto a la funcionalidad de los métodos que se describen en las int erfaces, estos se centran en la actualización de algunos de los parámetros que definen al robot. Esta actualización se llevaría a cabo como respuesta a una llamada por parte del cliente a un método de tipo get.

- **Listeners**:

Este es el tercer y ultimo paquete que engloba las interfaces que proporcionan la funcionalidad del cliente y servidor a nivel de la aplicación, y éste si es totalmente similar al anterior, ya que además de las características comunes que tenía con las interfaces de los paquetes anteriores, éstas además se utilizan también en la clase cliente (Client). La aplicación

de los métodos de estas interfaces se centra también en la actualización de los parámetros del robot, pero en este caso, la actualización en el cliente se realizara de forma automática mediante la escucha de eventos por parte del servidor, o cada cierto periodo de tiempo fijado por el cliente. No será necesario de que el cliente invoque un método get para que este reciba los valores actualizados de los parámetros.

Las clases que pertenecen a este paquete son: MechanismListener y ToolListener; y son solo 2 porque el componente "misión" no tiene la propiedad de registro. Como es de suponer, la primera de estas interfaces se encargará de definir los métodos de actualización para los componentes relacionados con el mecanismo, y la segunda de las interfaces para los componentes relacionados con la herramienta.

- **GUIs:**

Este paquete no pertenece en principio a la aplicación, ya que engloba a las clases utilizadas para la visualización del correcto funcionamiento del protocolo. Así, este paquete solo incluye clases que heredan de la clase Frame, y que tienen un conjunto de botones, menús y paneles que dotan al protocolo de una sencilla interfaz mediante la cual permite simular y verificar su correcto funcionamiento. Las clases pertenecientes al protocolo son: MainGUI (interfaz gráfica perteneciente al cliente, mediante la cual se podrá dar inicio y fin a la comunicación y abrir un cliente nuevo), ClientGUI (interfaz gráfica perteneciente al cliente, encargado de dar todas las ordenes a los servidores, además de fijar los parámetros de la actualización de los parámetros, ya sea periódico o mediante eventos) y MechanismServerGUI y ToolServerGUI (interfaces gráficas encargadas de simular la variación de los parámetros del robot).

- **IDs:**

El paquete IDs solo contiene una clase, llamada AppIDs, que almacena una serie de identificadores necesarios para el funcionamiento de la aplicación y configuración de la misma, con la idea de facilitar la realización de cambios en las mismas previamente a la puesta en marcha de la aplicación, y así disponer de una agrupación centralizada de dichos parámetros de configuración. Todos los identificadores son públicos para poder acceder a ellos desde cualquier clase del sistema, y sin presentar problemas de seguridad ya que son constantes y no se pueden modificar (variables de tipo "final" para impedir la modificación de su valor a lo largo de las ejecuciones). Además, esta clase es estática para que de este modo se pueda acceder con mucha facilidad a todas estas variables. Se optó por realizar un paquete nuevo para esta sola clase, ya

que se pensó que tras una ampliación de la aplicación se debería necesitar otro fichero con identificadores que se podría colocar en este paquete, evitando así una mala distribución de las clases.

- **Status:**

Este paquete incluye un conjunto de componentes que determinan el estado del robot en un momento determinado. Actualmente solo hay 4 componentes: GoyaStatus, GoyaJointStatus, Alarm y GoyaBlastingStatus, pero la implementación de otros componentes podría ser necesaria en un futuro si se añaden nuevas funcionalidades a la aplicación, o si se desea tener un mayor control sobre ésta. Cada uno de los componentes descritos anteriormente está formado por una interfaz y su correspondiente implementación.

De este modo, la clase Alarm es una interfaz que representa una alarma que se lanzaría en caso de urgencia o amenaza del robot, y AlarmImpl es una posible implementación de esta alarma y que nos permitirá crear instancias (objetos) de la misma. En cuanto al componente GoyaJointStatus, se puede observar que es otra interfaz, y una posible implementación está realizada en la clase GoyaJointStatusImpl. La utilidad de esta interfaz es que permite definir el estado de cada una de las articulaciones del robot. El resto de componentes (GoyaStatus y GoyaBlastingStatus) se presentan de la misma forma que los anteriormente comentados, es decir, cada componente contiene su interfaz asociada y una posible implementación que nos permite crear instancias de este componente para que sea enviado a través de la red cuando el cliente solicite una petición de estado del robot. Cada componente tiene un significado atendiendo al estado de la parte del robot que representa el mismo.

Por último destacar que las clases que implementen las 4 interfaces anteriores, además de implementar su correspondiente interfaz, debe implementar también la interfaz Serializable para permitir así que se puedan transmitir objetos de este tipo por la red.

El diagrama de clases número 9 que se presenta en el apartado de modelado del sistema mediante UML presenta la estructura estática de estas clases y sus relaciones entre ellas.

- **Management**:

En este paquete se encuentran el conjunto de clases encargadas del manejo y control de la aplicación en el extremo del cliente; en total: AppManager y Manager. Ambas clases están muy relacionadas ya que la primera de ellas representa la interfaz en la que se definen los

métodos encargados de la apertura y cierre de la aplicación. Así, por ejemplo, podemos encontrar métodos tales como startApp (arrancar aplicación), shutdownApp (cerrar aplicación) y openClient (abrir cliente) encargados de tales funciones.

La clase Manager supone la implementación de cada uno de los métodos de la interfaz anterior, y es por esto, por lo que en la clase podemos encontrar un vector (similar a array pero con funciones más avanzadas) en el que se almacenen los clientes y sus correspondientes interfaces gráficas. En la implementación realizada del método startApp se pretenden crear tantos clientes como se indique en la constante NUMBER_OF_CLIENTS de la clase AppIDs, de tal modo que si se desea, se puede configurar la aplicación para que el robot pueda ser manejado desde varios clientes en una misma máquina. Para realizar estas acciones, en la implementación se crearon 2 vectores, en los cuales se almacenarán tanto los clientes como las interfaces gráficas asociadas a estos clientes.

El diagrama de clases número 12 que se presenta en el apartado de modelado del sistema mediante UML presenta la estructura estática de esta clase y sus interfaz.

- **Clients:**

Este paquete contiene solamente 3 clases: Client, ClientsContainer y RegistrationPeriod. En cuanto a la clase Client, hay que destacar que junto a las clases "servidores" es una de las clases más importantes ya que representa al cliente que maneja los servidores del robot. Tan solo analizando la sentencia inicial podemos descubrir su estructura y relación con las otras clases de la aplicación: "class Client implements MechanismListener, ToolListener, MechanismUpdater, MissionUpdater, ToolUpdater, Serializable". Las sentencias de implements nos indican que esta clase implementa los métodos de todas esas interfaces. Empezando por la más conocida, la interfaz Serializable es necesario implementarla debido a que es necesario pasar objetos de este tipo a través de la red. Las otras interfaces implementadas se corresponden con las que definen todos los métodos que el cliente debe poder realizar; así, distinguimos por un lado las interfaces de tipo "update" que engloban a los métodos encargados de la actualización de los parámetros (o variables) tras una petición realizada por parte del cliente, mientras que las interfaces "listeners" engloban a los métodos que realizan esta misma función pero en este caso de forma periódica o a partir de un evento.

El diagrama de clases número 7 que se presenta en el apartado de modelado del sistema mediante UML presenta de forma más clara las interfaces que implementa un cliente.

Analizando brevemente las asociaciones de composición que tiene esta clase con otras de la aplicación, vemos que la clase Client contiene 3 referencias a 3 posibles objetos que implementen las interfaces que contenía el paquete Controls (MechanismCtrl, MissionCtrl y ToolCtrl), estos objetos serán realmente los proxies, aunque el cliente crea que sean directamente los 3 servidores (que realmente están distribuidos remotamente y los proxies se encargaran de realizar la conexión de forma transparente al cliente).

Nota: En ocasiones puede interesar que un atributo concreto de un objeto no sea serializado, como por ejemplo lo son las 3 referencias MechanismCtrl, MissionCtrl y ToolCtrl que cada cliente contiene. Esto se puede conseguir utilizando el modificador transient, que informa a la JVM de que no nos interesa mantener el valor de ese atributo para serializarlo o hacerlo persistente. En el cliente, estos 3 objetos no serán incluidos en la secuencia de bytes resultante de serializar un objeto de clase Client.

En cuanto a los métodos que presenta esta clase, hay que destacar que además de haber los propios de la implementación de las interfaces anteriormente indicadas, también aparecen todos los de las interfaces de los servidores, como es el caso de enableJoint(int jointNumber), ya que cuando se ordena una acción se debe invocar a un método de un servidor y para ello, como éste se encuentra remotamente primero se ha de pasar dicha orden al cliente.

La existencia de una clase que interviene en el periodo de registro de los clientes en el servidor (clase RegistrationPeriod) dentro del paquete Clients y no dentro de Servers.Registration puede resultar un poco irregular, pero se decidió colocar la clase dentro de este paquete ya que solo la usamos desde el cliente. La utilidad que permite esta clase es la de redondear el periodo a utilizar en el registro mediante el método roundPeriod; y esta acción se realiza desde el cliente, para así, ofrecer al servidor el periodo ya ajustado y liberar al servidor el realizar otra operación más. El redondeo del periodo de registro del cliente es muy importante debido al patrón seguido a la hora de registrase en el servidor, y el por qué se ha de redondear el periodo bajo cierto factor se comentará detalladamente más adelante, concretamente cuando se estudie en profundidad el registro de un cliente en el servidor.

Por último, la clase ClientsContainer es la clase que representa a un objeto contenedor que agrupa el conjunto de clientes, proxies y stationLinks (enlaces) para asociarlos entre si de forma centralizada. Esta clase permite acceder a todos estos objetos de forma centralizada ya que la clase es estática y podemos acceder a ella sin necesidad de crear una instancia de la misma. A través de esta clase se creara toda la arquitectura en el extremo del cliente, ya que ella será la que contenga almacenados todos los clientes y permita acceder a ellos de forma sencilla.

El contenido de esta clase se estudiara en profundidad en el apartado de creación de la arquitectura, en el extremo del cliente.

- **Servers:**

Este paquete engloba tanto a los servidores como a las clases encargadas del registro, por lo que se decidió hacer un nuevo subpaquete que perteneciera a éste. En el paquete principal se encuentran los servidores: MechanismServer, MissionServer y ToolServer, cada uno encargado del control de una parte concreta del robot. Junto a estas clases se encuentra la clase ServersContainer, que pretende actuar como un contenedor de los servidores anteriores. Como es de suponer, la estructura de cada uno de estos servidores es muy similar, así que, con el análisis de uno de ellos queda explicado el comportamiento de los 2 restantes.

Cada servidor implementa su interfaz correspondiente del paquete C ontrols, de este modo, la clase MechanismServer implementa la interfaz MechanismCtrl ("class MechanismServer implements MechanismCtrl"), en este caso esta interfaz corresponde con la que define el manejo del brazo del robot. Esta interfaz (junto con ToolCtrl y MissionCtrl) son las que definen los métodos para controlar el robot, y estos métodos serán los que el cliente tiene que llamar, aunque sea de forma indirecta a través de las comunicaciones Proxy/Stub correspondientes.

El diagrama de clases número 17 que se presenta en el apartado de modelado del sistema mediante UML presenta de forma más clara las interfaces que implementa un cliente y los métodos que forman cada una de ellas.

Analizando en esta clase también las asociaciones de composición, vemos como al contrario que en el cliente (donde se tenían objetos que representarían al servidor), la clase MechanismServer no va a estar formada por objetos que representan al cliente, liberando de esta forma al servidor de conocer todos sus clientes. Además de esto, el servidor únicamente tiene que conocer a un cliente cuando dicho cliente solicita que le envíe el estado del robot (ya que en el resto de órdenes no se debe devolver nada al cliente como respuesta). Es por esto por lo que cada cliente debe incluir en cada método que implique una respuesta por parte del servidor (métodos get) un parámetro adicional donde insertara una referencia a si mismo, de forma que cuando el servidor reciba esa petición pueda enviarle la respuesta utilizando dicha referencia.
Más adelante se comprobará que realmente se trabaja además con referencias sobre el Proxy y el stub, aunque todo ello es transparente tanto para el cliente como para el servidor. Sin embargo, no se debe olvidar del servicio de registro de clientes en el servidor, en donde en éste

caso si que se almacenan los clientes en el servidor correspondiente, de forma que cada cierto periodo de tiempo o evento en el estado del robot, el servidor tenga alguna referencia de los clientes que se registraron en el servicio para que pueda contestarles.

La clase MechanismServer estará formada por un conjunto de objetos de tipo EventControl, TimerControl, y RegisteredClients, que permitirán la implementación de un registro particular de clientes. Todo ello se comentará detalladamente en la parte de registro de clientes.

Para finalizar, si se analizan los métodos existentes en esta clase vemos como además de los métodos propios de la interfaz implementada, existen también métodos homónimos a los que hay en la clase Client, y es que al igual como ocurría en la clase Client, cada clase tiene que tener métodos similares a los del extremos opuesto, para que al invocar al "método local", éste haga una invocación remota del método del otro extremo.

Por ultimo, y teniendo en cuenta que el comportamiento del resto de los servidores es similar al comentado con la salvedad de que MissionServer no tiene la capacidad de registro, solo queda comentar la clase ServersContainer. Como ya se dijo anteriormente, esta clase se comporta como un contenedor de los servidores anteriores. Esta clase será estática (todos sus métodos son estáticos) ya que no es necesario crear varias instancias de la misma en la aplicación, y de esta forma podemos acceder a sus métodos de forma más sencilla (utilizando el propio nombre de la clase, y sin necesidad de crear ninguna instancia de la clase). Es importante destacar que ésta clase, además de contener a los servidores, contiene todos los stubs y rocLinks (enlaces) existentes en la aplicación. Con esto se consigue asociar servidores, stubs y enlaces entre si de forma centralizada. En definitiva, esta clase contiene los métodos que permiten crear y arrancar todos los servidores de control del robot para iniciar la aplicación, además de cerrar por completo la aplicación; el proceso seguido para ello se comenta en el apartado de aspectos característicos de la implementación

- **Servers.Registration:**

Este paquete se encuentra dentro del paquete Servers porque en este se encuentran todas las clases relacionadas con el registro de clientes y este registro solo se encontrará en los servidores. Las clases englobadas en esta paquetes son: EventControl, RegisteredClients, TimerControl, TimeRegisteredObject. Aunque el registro de los clientes sea un tema muy complejo y se dedique un apartado independiente para su comentario, a continuación se va a comentar brevemente el uso de cada clase.

Dentro de este paquete podemos distinguir las clases que se encargan de los registros de clientes en los servidores, que se encargan de almacenar los clientes que se registran, para recibir la actualización de variables mediante eventos, y también (de forma separada) para los de la actualización de forma periódica. La utilidad de este tipo de registro permite que cada vez que se produzca un evento o cada vez que expire un periodo de tiempo, haya que dirigirse al registro que corresponda para invocar algún método remoto de los clientes almacenados.

Para la implementación del servicio de registro de clientes, se necesitan las siguientes clases: EventCotrol, TimerControl, RegisteredClients y TimeRegisteredClients. Las dos primeras se utilizan para el tratamiento de los elementos a ocurrir para que se lleve a cabo una actualización; y las 2 siguientes para almacenar los clientes.

La clase RegisteredClients está formada principalmente de 4 vectores en los que almacenar los clientes, ordenándolos éstos según el tipo de clientes que sean y según el tipo de registro deseado; así se dispondrá de los vectores:

- mechanismEventListeners: vector para almacenar los clientes de tipo mechanism que se registran para recibir las actualizaciones de las variables cada vez que ocurra un evento definido anteriormente.
- toolEventListeners: vector para almacenar los clientes de tipo tool que se registran para recibir las actualizaciones de las variables cada vez que ocurra un evento definido anteriormente.
- mechanismTimeListeners: vector para almacenar los clientes de tipo mechanism que se registran para recibir las actualizaciones de las variables cada vez que expire un periodo.
- toolTimeListeners: vector para almacenar los clientes de tipo tool que se registran para recibir las actualizaciones de las variables cada vez que expire un periodo.

Además, para cada uno de estos vectores se crearán un método que especifique claramente al llamarlo, en que vector se ha de almacenar el cliente; así se tiene: addToolTimeListener, addMechanismTimeListener, addMechanismEventListener y addToolEventListener. En este registro de clientes si nos interesa el poder eliminar clientes, ya que si un cliente desea no recibir más actualizaciones de variables, habrá que dejar de realizarle este servicio; así, se dispondrán de métodos con la misma nomenclatura que los anteriores cambiando “add” por “remove”.

A la hora de almacenar los clientes, hay que distinguir si éste se registra periódicamente o bien por eventos, ya que si es del primer tipo, el cliente se almacenará en el vector como un objeto de tipo TimeRegisteredObject, mientras que si es por eventos se almacenará solo como un String. Esto es debido a que en el primero de los casos, además de la cadena de acceso al cliente también necesitaremos almacenar el periodo deseado; mientras que para el segundo tipo, con almacenar la cadena de acceso será suficiente. Así, los vectores toolTimeListeners y mechanismTimeListeners contendrán objetos TimeRegisteredObject y el resto solo Strings.

Como es de suponer tras el párrafo anterior, los objetos de la clase TimeRegisteredObject solamente constan de un String que representan la cadena de acceso y de una variable de tipo entero que indique el periodo. Además esta clase dispondrá de métodos para modificar el periodo de ese cliente, ya que se puede dar el caso que se pretenda almacenar un cliente ya almacenado con otro periodo, con lo que implica que tan solo habrá que modificar su periodo.

Por último, ya solo queda comentar las clases de tipo “Control” de este subpaquete. La clase TimerControl será encargada de generar un evento cada cierto tiempo y esto se utilizará para que cada evento se envíen actualizaciones a los clientes registrados. Esta clase estará formada por un objeto de tipo Timer que delega en otro que implemente la interfaz ActionListener, la acción a realizar cada vez que expire un cierto periodo (que toma como parámetro en mseg.). Para facilitar el entendimiento, se decidido que la clase a la que delegar ese trabajo sería esta misma clase, de tal modo que cada X mseg el objeto Timer llamara a su método actionPerformed de esta clase.

La lista de clientes registrados (ya sea con peri odo o mediante eventos) a cada servidor será recorrida desde la clase EventControl. Además, desde esta clase se llamará al método del servidor que corresponda (gracias a un referencia que se dispone de éste) .

Nota: La utilización de vectores en lugar de arrays para almacenar algunos objetos, esta justificada debido a que aunque si se hubiera utilizado la segunda opción el acceso a sus elementos hubiera sido mucho más rápido y eficiente, al utilizar los vectores se permite poder almacenar tantos objetos como se deseen, evitando de este modo los desbordamientos del array, sin olvidar también la facilidad ofrecida por los métodos de los vectores para insertar y eliminar sus elementos.

El diagrama de clases número 14 que se presenta en el apartado de modelado del sistema mediante UML presenta de forma más precisa las relaciones entre las clases asociadas con el registro de clientes y los servidores.

- **ComExceptions:**

Debido a las múltiples situaciones excepcionales que se pueden ocasionar en esta aplicación se decidió definir una serie de excepciones propias del sistema. Este paquete es el que recoge todas las excepciones propias del protocolo que puede lanzar la aplicación, y la arquitectura seguida para su diseño es la siguiente: En primera instancia tenemos la clase madre del resto de excepciones, conocida como ComException, esta clase representa la excepción de la cual heredaran todas las excepciones generadas por el protocolo. Dicha clase heredara de Exception, de forma que cada excepción propia del sistema sea también una excepción Java. El resto de clases del paquete son las excepciones más específicas que puede lanzar la aplicación, y todas ellas heredaran de ComException. De este modo, la aplicación dispone de las siguientes excepciones concretas:

- ConnectionComException: Se lanzara en caso de que no se pueda realizar la conexión entre cliente y servidor.
- IOComException: Se lanzarán cuando ocurran problemas de entrada/salida con los flujos de comunicaciones de los enlaces.
- IncorrectStateComException: Se lanzará en caso de que el estado del enlace no sea el correcto para una petición concreta.
- InvalidIDComException: Se lanzará cuando se utilice un identificador y este no corresponda a ninguno de los permitidos en ese momento.

El resto de excepciones que no abarcan las anteriores se lanzan directamente con una ComException, indicando en su respectivo constructor el motivo por el cual es lanzada la misma. Es importante definir aparte las 4 excepciones anteriores debido a la gran cantidad de posibilidades en la que se pueden lanzar, permitiendo así una mejor depuración y simulación del protocolo en cuestión.

El diagrama de clases número 10 que se presenta en el apartado de modelado del sistema mediante UML presenta de forma más explícita la relación de las excepciones propias con la clase Exception de Java.

- **Proxies:**

Este paquete contiene los 3 tipos de proxies necesarios para la aplicación (MechanismProxy, ToolProxy, MissionProxy), y todos ellos heredan de la clase "Proxy", que será la clase madre de los anteriores. El motivo por el que se definen proxies esta detallado en el capítulo de creación de la arquitectura, pero resumiendo se puede decir que cada proxy pretende ocultar al cliente los aspectos de distribución de los servidores de control del robot, esto es, cada proxy se encarga de contactar con el servidor remoto cada vez que el cliente realiza una petición o de comunicarse con el servidor cuando dicho servidor devuelve una respuesta a una petición. Lo realmente importante es que todo esto es transparente para el cliente, ya que el proxy implementa la misma interfaz que su correspondiente servidor y el cliente solo conoce esta interfaz, pero no la implementación de la misma.

Resumiendo todo el proceso, podemos decir que el proxy actúa como represent ante local del servidor en la maquina del cliente, de forma que el cliente actué sobre la interfaz del proxy (que es la misma que la del servidor) pensando que es el propio servidor.

A continuación se detallara el contenido de la clase MechanismProxy, da ndo por hecho que el contenido del resto de clases (ToolProxy y MissionProxy) son de contenido similar, pero con distintos métodos implementando su respectiva interfaz de su servidor asociado (interfaces ToolCtrl y MissionCtrl respectivamente). Continuando con MechanismProxy, podemos decir que representa a un intermediario entre un cliente y un servidor de tipo Mechanism, ya que implementa la interfaz MechanismCtrl para poder actuar como servidor para el cliente. El cliente contendrá de esta forma una referencia a un objeto MechanismCtrl que luego se instanciará con un objeto MechanismProxy. De esta forma el cliente no conoce a ningún proxy y lo trata como si fuera directamente el servidor. MechanismProxy, al igual que ToolProxy y MissionProxy, hereda de la clase Proxy, y esto es porque es necesario para englobar todos los tipos de proxies en una clase generalizada, ya que al obtener una referencia del proxy asociado a un determinado enlace (StationLink) no se conoce el tipo al que puede estar asociado dicho enlace (Mechanism, Tool o Mission) y es necesaria esta clase que los englobe a todos. El tipo de proxy concreto que es necesario una vez tenemos un objeto "Proxy" se obtiene haciendo cast sobre la superclase Proxy, el tipo de clase a hacer cast dependerá de un identificador que contiene la clase madre (se comentará en detalle el capítulo de creación de la arquitectura).

El diagrama de clases número 15 que se presenta en el apartado de modelado del sistema mediante UML presenta de forma más las relaciones entre las clases relacionadas con los proxies de la aplicación.

- **Stubs:**

Este paquete contiene los 3 tipos de stubs necesarios para la aplicación (MechanismStub, ToolStub, MissionStub), y todos ellos heredan de la clase "Stub", que será la clase madre de los anteriores. El motivo por el que se definen stubs esta detallado en el capítulo de creación de la arquitectura, pero resumiendo se puede decir que cada stub pretende ocultar al servidor de control del robot los aspectos de distribución de los clientes, esto es, cada stub se encarga de comunicarse con el proxy remoto correspondiente (cliente) para obtener una petición que realice el cliente o una respuesta que envíe el servidor. Al igual que ocurría en el extremo del cliente, lo realmente importante es que todo esto es transparente para el servidor, ya que el stub implementa las mismas interfaces que cada cliente y el servidor solo conoce esta interfaz, pero no la implementación de la misma.

Resumiendo todo el proceso, podemos decir que el stub actúa al igual que el proxy, pero en este caso en el extremo del servidor, de forma que cada stub sea un representante local de cada cliente en la maquina del servidor, de forma que el servidor actué sobre las interfaces del stub (que son las mismas que las del cliente) pensando que es el propio cliente De este modo, el servidor realmente dispondrá de un cliente, el stub, y será el stub el que redireccione las respuestas del servidor a todos los clientes que estén localizados remotamente.

A continuación se detallara el contenido de la clase MechanismStub, dando por hecho que el contenido del resto de clases (ToolStub y MissionStub) son de contenido similar, pero con distintos métodos implementando las interfaces respectivas del tipo de cliente asociado (interfaces ToolListener-ToolUpdater y MissionUpdater respectivamente). Nota: MissionStub no implementa la interfaz Listener correspondiente ya que no existe servicio de registro para el servidor de Misión, por lo que tampoco lo implementan los clientes de tipo Mission.

Continuando con MechanismStub, podemos decir que representa a un intermediario entre los clientes y el servidor de tipo Mechanism, ya que implementa las interfaces MechanismListener y MechanismUpdater para poder actuar como cliente para el servidor. De esta forma, el servidor conocerá solamente referencias a objetos que implementen las interfaces anteriores que luego se instanciarán con un objeto MechanismStub. De esta forma el servidor no conoce a ningún stub y lo trata como si fuera directamente el cliente. MechanismStub, al igual que ToolStub y MissionStub, hereda de la clase Stub por la misma razón que los proxies comentados anteriormente heredaban de Proxy, es decir, porque es necesario para englobar todos los tipos de stubs en una clase generalizada, ya que al obtener una referencia del Stub asociado a un determinado enlace (RocLink) no se conoce el tipo al que puede estar asociado dicho enlace (Mechanism, Tool o Mission) y es necesaria esta clase que los englobe a todos. El

tipo de Stub concreto que es necesario una vez tenemos un objeto "Stub" se obtiene haciendo cast sobre la superclase Stub, el tipo de clase a hacer cast dependerá de un identificador que contiene la clase madre (se comentará en detalle el capítulo de creación de la arquitectura).

El diagrama de clases número 16 que se presenta en el apartado de modelado del sistema mediante UML presenta de forma más las relaciones entre las clases relacionadas con los proxies de la aplicación.

- **Protocol**:

El protocolo de comunicaciones definido trabaja por encima de TCP/IP y por debajo de nuestra aplicación del robot (capa de aplicación), esto es debido a que la arquitectura cliente – servidor no ofrece un middelware que convierta el paso de mensajes a través de los enlaces de comunicaciones en llamadas a procedimiento (métodos).

La arquitectura cliente – servidor requiere definir un protocolo de bajo nivel, que en nuestro caso se basa en el paso de mensajes (objetos Java) a través de enlaces de comunicaciones que serán procesados de forma que se invoque un método del robot o del cliente, o se procese internamente si es un mensaje de control y mantenimiento del protocolo. Esta capa de comunicaciones de bajo nivel esta definida en el paquete "Protocol", que contiene a su vez otros paquetes que dividen el protocolo en varias secciones:

- **Protocol.ControlLinks:**

El protocolo trabaja bajo una arquitectura en la que se dispone de 3 enlaces reservados para el control del robot y un enlace adicional de control y mantenimiento del protocolo. El canal de control permite iniciar, mantener y cerrar las comunicaciones de los otros 3 enlaces pertenecientes a los servidores de control del robot. De esta forma, el paquete ControlLinks contiene las clases necesarias para crear un enlace de control de la aplicación, que son la clase que representa al servidor de control (CtrlLinkRoc) y la que representa a un cliente de este servidor de control (CtrlLinkStation).

La clase CtrlLinkRoc representa al extremo servidor del enlace de control, y contiene métodos para crear un servidor que permita abrir dicho enlace, para enviar y recibir mensajes a través de dicho enlace, y finalmente para cerrar el enlace al terminar la aplicación. Solo habría una instancia de esta clase en el host del servidor, por lo que definimos todos sus métodos y

variables estáticas para que no sea necesario crear una instancia de la misma, llamando así a todos sus métodos usando como referencia el nombre de la propia clase.

La clase CtrlLinkStation representa al extremo cliente del enlace de con trol, y contiene métodos para conectarse al servidor de control del protocolo creado por la clase CtrlLinkRoc, y poder enviar y recibir mensajes a través de dicho enlace. Al igual que la clase anterior, debido a que en el host de un posible cliente solo habría una instancia de esta clase, se decidió definir todos sus métodos y variables estáticas para que no sea necesario crear una instancia de la misma, llamando así a todos sus métodos usando como referencia el nombre de la propia clase.

Por último, cabe destacar la importancia de estas dos clases en el funcionamiento del protocolo debido a que son las encargadas de crear el enlace de control sobre el que, como se verá más adelante, se envían los mensajes que permitirán iniciar y cerrar una conexión con los servidores del robot Mechanism, Tool y Mission.

- **Protocol.Links:**

Como ya se comento anteriormente, además del enlace de control del protocolo existen 3 enlaces encargados del control del robot, que se realizara mediante los servidores de Mechanism, Tool y Mission. Este paquete contiene las clases necesarias para crear un enlace de control del robot, validas para los tres tipos de servidores Mechanism, Tool y Mission. Una de las clases es la que representa a un servidor del robot (RocLink) y la que representa al cliente (StationLink), contiene además la clase que define el estado de un enlace (State).

Para comprender mejor los enlaces que crea cada clase en la aplicación se puede observar el diagrama de despliegue (número 21) que se presenta en el apartado de modelado del sistema mediante UML.

La clase RocLink representa el extremo servidor del enlace al nivel más interno del protocolo. Se encarga de proporcionar servicio a la clase StationLink (extremo del cliente al nivel más interno). De este modo, esta clase contiene métodos para crear un servidor que permita abrir un enlace, para enviar y recibir mensajes a través de dicho enlace, para procesar cada mensaje que recibe del cliente, para traducir una respuesta que el servidor ha de enviar al cliente en un mensaje, y finalmente para cerrar el enlace al terminar la aplicación.

La clase StationLink representa el extremo cliente del enlace al nivel más interno del protocolo. Se encarga de usar el servicio que proporciona la clase RocLink (extremo del

servidor al nivel más interno). De este modo, esta clase contiene métodos para conectarse al servidor que la clase RocLink creó, para enviar y recibir mensajes a través de dicho enlace, para procesar cada mensaje que recibe del servidor, para traducir una orden que el cliente ha de enviar al servidor en un mensaje, y finalmente para cerrar el enlace al terminar la aplicación.

Estas dos últimas clases son las encargadas de crear un enlace sobre el que se podrá controlar una parte del robot (Mechanism, Mission o Tool) mediante una comunicación cliente – servidor. De este modo, será necesario crear 3 enlaces, por lo que dispondremos de 3 instancias de la clase RocLink y 3 instancias de la clase StationLink. Cada una de ellas se emaprejará con su correspondiente extremo opuesto (station - roc) para cada tipo de servicio (Mechanism, Mission o Tool). Cada pareja representara un extremo del enlace creado, y este enlace se ha monitorizado definiendo una clase adicional, la clase State. Esta clase es la encargada de representar el estado del enlace sobre el que trabajan las instancias de RocLink y StationLink. Cada estado del enlace se representa en la clase como un número entero, y de esta forma tenemos los siguientes estados:

- DISCONNECTED: Desconectado del enlace.
- CONNECTING: Conectando con el servidor (en cliente) o creando servidor (en servidor).
- CONNECTED: Conectado al enlace.
- BLOCKED: Enlace bloqueado.

Cada extremo del enlace dispondrá de una instancia de la clase State, y atendiendo al transcurso de la aplicación el estado ira modificándose. Cada extremo deberá comprobar el estado del extremo opuesto antes de realizar una operación, para comprobar si es viable o no dicha operación en ese momento.

El diagrama de clases número 6 que se presenta en el apartado de modelado del sistema mediante UML presenta la estructura estática de estas dos clases y sus relacion con las clases Socket, ServerSorcket y State. Mientras que en los diagramas 18 y 19 forman los diagramas de estado de las clases RocLink y StationLink atendiendo a los estados comentados anteriormente.

- **Protocol.Threads:**

Algunas de las operaciones que realizan las clases de los paquetes ControlLinks y Links anteriores pueden resultar bloqueantes, y esto no es viable en una aplicación que tenga que trabajar con múltiples clientes. De este modo se han definido una serie de clases que actúen

como threads para realizar las operaciones que pueden ser bloqueantes para el protocolo. Este paquete es el que contiene dichos threads, que veremos uno por uno para explicar con más detalle que función realiza cada uno. Los hilos son los siguientes:

Conector: Cuando el thread se ejecuta en el extremo servidor, únicamente crea un servidor en un puerto determinado y empieza a aceptar peticiones (aunque se servirán en otro thread), evitando que el sistema se bloquee por el hecho de que los servidores esperen una petición de conexión que no llega. Cuando el thread se ejecuta en el extremo cliente, únicamente se encarga de conectarse con el servidor creado en el extremo opuesto, iniciando la conexión y creando los flujos de entrada / salida, evitando que el sistema se bloquee debido a un fallo de conexión de los clientes al intentar conectarse a dicho servidor.

Transmitter: Se encarga de enviar cualquier información a través de un enlace mediante un mensaje, es decir, transmite un objeto a través de un enlace representado por un flujo de escritura, evitando que un envió de un mensaje bloquee a otro por criterios de longitud del mensaje o fallo en la transmisión.

Receiver: Representa a un thread que estará continuamente recibiendo en un extremo del enlace de comunicaciones de control del robot, y procesando los mensajes que reciba de forma adecuada. De este modo se evita que una recepción de un mensaje bloquee el sistema por criterios de longitud del mensaje o fallo en la recepción.

ProccessMsg: Para cada mensaje que recibe el thread anterior (Receiver) se crea un thread de este tipo que sea capaz de procesar dicho mensaje en una llamada al método asociado a dicho mensaje, es decir, es el encargado de convertir los mensajes enviados por la red a llamadas locales a métodos. Al arrancar este thread se encarga de analizar el tipo de mensaje, obtener todos sus paramteros (si los tiene), y llamar al método correspondiente del Proxy o del Stub. El Thread tiene 2 constructores, uno para cada extremo (roc y station) ya que los mensajes a procesar en cada extremo son diferentes.

CtrlLinkReceiver: Es el Thread receptor del enlace de control, que esta continuamente esperando recibir algún mensaje de control y lo procesa según el identificador del mismo. Este thread realiza la función que realizaban los threads Receiver y ProcessMsg, pero en este caso en el enlace de control del protocolo. El procesamiento que se lleva a cabo permite recibir mensajes con identificadores de ACK, NACK, END y de puertos, cuyo significado se explicar en detalle en el apartado de aspectos significativos de la implementación).

Todos los threads anteriores pueden trabajar indistintamente en los 2 extremos de la aplicación (cliente y servidor), ya que en el caso de que el procesamiento del thread en un extremo sea distinto al del extremo opuesto, se usaran distintos constructores en cada caso.

El diagrama de clases número 11 que se presenta en el apartado de modelado del sistema mediante UML contiene todos los threads anteriores y su relación con la clase Thread de Java.

- **Protocol.Messages:**

La información que se envía a través de todos los enlaces de la aplicación se encapsula en un objeto cuyo formato esta definido por las clases de tipo "mensaje" (Msg) que se definen en este paquete. En primer lugar esta la clase Message, que no únicamente es una clase abstracta que contiene dos métodos (getID y toString), que serán los métodos que todo mensaje de la aplicación debe implementar como mínimo. Por ello, el resto de mensajes heredaran de esta clase e implementaran estos dos métodos. El método getID() devuelve el identificador del mensaje, que indica el tipo de información que contiene, y el método t oString() devuelve una cadena de caracteres asociada al mensaje, necesaria para la simulación y depuración de la aplicación.

El resto de mensajes pueden ser de varios tipos en función de la información que contienen, y atendiendo a dicha información se pueden clasificar en los siguientes:

- ComMsg: Representa al objeto que se envía por la red con carácter interno (gestión y mantenimiento del protocolo), por lo que no representa a operaciones del robot, sino a aspectos como cierre de conexión, acuse de recibo y habilitación / deshabilitación del enlace.

- MechanismMsg: Mensajes relacionados con el enlace de control del servidor que controla el mecanismo.
- ToolMsg: Mensajes relacionados con el enlace del servidor que controla la herramienta.
- MissionMsg: Mensajes relacionados con el enlace del servidor que controla una tarea del robot.

Aunque los 4 últimos mensajes hereden de Message, el contexto en el que se usa ComMsg no es el mismo que cuando se usa MechanismMsg, ToolMsg y MissionMsg, ya que ComMsg se utiliza para enviar información de carácter interno al protocolo (mantenimiento y

gestión de las comunicaciones), mientras que el resto se usa para enviar ordenes al servidor del robot o recibir respuestas del mismo. Es por ello por lo que se decidió encapsular todos los posibles mensajes en uno común, cuyo formato lo define la clase ComposedObject. Esta clase representa al objeto que encapsulara todo lo que se enviara a través de la red, es decir, en una instancia de esta clase se encapsularan tanto los mensajes de comunicación internos (inicio, mantenimiento y cierre de conexión) como los externos (movimientos del robot, estados, alarmas,...). Para que el receptor sepa si lo que esta encapsulado es realmente un mensaje interno (comunicaciones) o externo (robot), se añade un identificador. Una característica importante de esta clase es que si vamos a enviar un mensaje externo, el constructor a usar es ComposedObject(int ID, Object obj), de manera que como se usa un Object, podemos enviar de extremo a extremo cualquier cosa, ya sea un mensaje o una referencia a algún otro objeto (que implemente Serializable).

La última clase que falta comentar de este paquete es CtrlComposedObject, cuya finalidad es la misma que la de la clase ComposedObjet, pero en este caso es el objeto que se envía a través del enlace de control del protocolo (véase paquete CtrlLinks), mientras que los objetos ComposedObject se enviarán a través de los enlaces de los servidores de control del robot en cuestión. Con este diseño, podemos encapsular cualquier cosa necesaria fácilmente y enviarla por la red encapsulada en un objeto ComposedObject, porque solo seria necesario definir un nuevo identificador asociado al objeto que se encapsula en la clase ComposedObject.

Por último, es importante comentar que todas estas clases que representan los objetos que se envían a través de un enlace deben implementar la interfaz Serializable.

La serialización de un objeto consiste en obtener una secuencia de bytes que represente el estado de dicho objeto. Esta secuencia puede utilizarse de varias maneras (puede enviarse a través de la red, guardarse en un fichero para su uso posterior, utilizarse para recomponer el objeto original, etc.). Básicamente consiste en guardar el estado de los campos de un objeto y si el objeto a serializar tiene campos que a su vez son objetos, habrá que serializarlos primero. Éste es un proceso recursivo que implica la serialización de todo un grafo (en realidad, un árbol) de objetos. Además, también se almacena información relativa a dicho árbol, para poder llevar a cabo la reconstrucción del objeto serializado.

Un objeto serializable es un objeto que se puede convertir en una secuencia de bytes. Para que un objeto sea serializable, debe implementar la interfaz java.io.Serializable. Esta interfaz no define ningún método. Simplemente se usa para 'marcar' aquellas clases cuyas instancias pueden ser convertidas a secuencias de bytes (y posteriormente reconstruídas).

Objetos tan comunes como String, Vector o ArrayList implementan Serializable, de modo que pueden ser serializados y reconstruídos más tarde. El sistema de ejecución de Java se encarga de hacer la serialización de forma automática.

Todas las relaciones de herencia y composición asociadas a los mensajes y objetos encapsuladores de información se encuentran en el diagrama de clases número 13 que se presenta en el apartado de modelado del sistema mediante UML.

- **Protocol.Components:**

Contiene una serie de utilidades necesarias para el funcionamiento del protocolo, y que conviene estudiar con detalle una por una. Las clases en cuestión son las siguientes:

GeneratePorts: Es la clase encargada de gestionar todos los puertos que se usan en el protocolo, ésta añade y elimina puertos de una lista de números puertos disponibles. Solo habría una instancia de esta clase en todo el proyecto, por ello definimos todos sus métodos y variables estáticas para que no sea necesario crear una instancia del mismo, llamamos a los métodos usando como referencia la propia clase.

NewPorts: Representa al objetos que se envían por el enlace de control encapsulado en un CtrlComposedObject y que contiene el par de enteros "puerto" y "tipo de enlace", que usara el cliente para conectarse a los servidores deseados. Al igual que los mensajes, debe implementar la interface Serializable porque instancias de esta clase se enviaran por la red.

ComIDs: Esta clase almacena una serie de identificadores necesarios para el funcionamiento del protocolo y su configuración, con la idea de facilitar la realización de cambios en el mismo previamente a la puesta en marcha de la aplicación, y así disponer de una agrupación centralizada de dichos parámetros de configuración. Todos los identificadores son públicos para poder acceder a ellos desde cualquier clase del sistema, y sin presentar problemas de seguridad ya que son constantes y no se pueden modificar (variables de tipo “final” para impedir la modificación de su valor a lo largo de las ejecuciones). Además, esta clase es estática para que de este modo se pueda acceder con mucha facilidad a todas estas variables.

El resto de clases del paquete (MechanismIDs, ToolIDs y MissionIDs) también almacenan identificadores, pero en este caso son necesarios para la creación de mensajes, es decir, para la creación de los mensajes es necesario un identificador que indique el significado y el contenido que va a tener ese mensaje. Las clases MechanismIDs, ToolIDs y MissionIDs

contienen los identificadores necesarios para los mensajes de tipo MechanismMsg, ToolMsg y MissionMsg respectivamente.

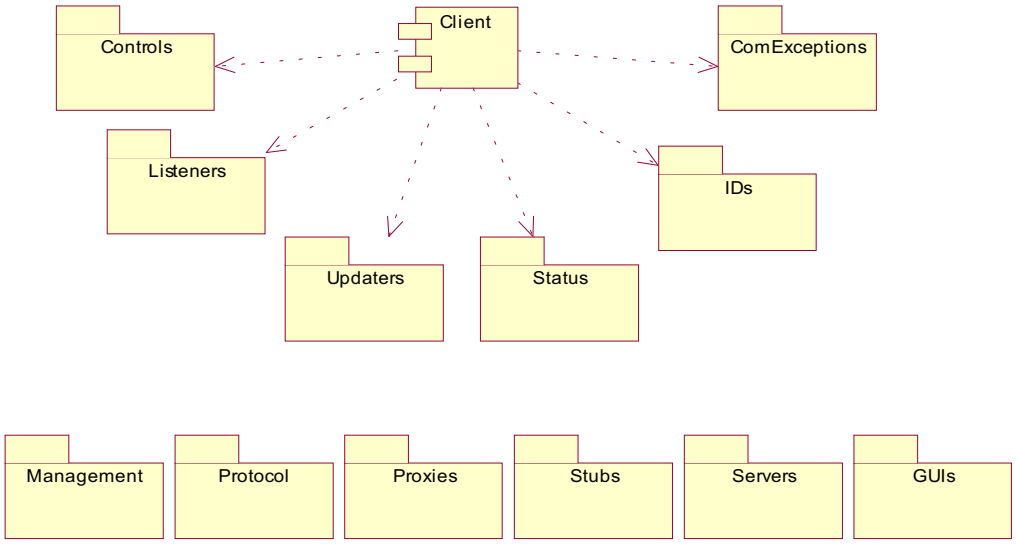
7.3. Dependencias significativas.

Una vez examinada la estructura de la aplicación, es importante conocer las dependencias creadas entre los paquetes del sistema. Las dependencias permiten visualizar que paquetes necesitan usar otros paquetes, de forma que si cambia el contenido de un paquete, habrá que tener en cuenta dichos cambios en los paquetes que dependan del paquete modificado. A continuación comprobaremos las dependencias más importantes entre paquetes y entre clases y paquetes, donde se comprobara en cada caso que las dependencias son las mínimas posibles, surgiendo lo que se conoce como bajo acoplamiento. El acoplamiento ocurre normalmente cuando una clase (o paquete) necesita saber demasiados detalles internos de otra para su funcionamiento, es decir, rompe el encapsulamiento del que tanto se habla en la programación orientada a objetos. Por supuesto, para tener un diseño correcto, fácil de mantener y modular, cuanto más bajo acoplamiento haya entre las clases (o paquetes) mejor.

Algunas de las dependencias más importantes en la aplicación en cuestión son las siguientes:

- **Clase Client:**

La clase que representa al cliente depende obviamente de las interfaces Listener y Updater ya que son las interfaces que él mismo debe implementar para que el servidor pueda contestarle a las peticiones de obtención del estado del robot. En cuanto a la dependencia con la clase Controls, es obvia ya que esta interfaz corresponde con la que implementaran los servidores y son sus métodos los que tiene que llamar para actuar sobre el robot a través de dichos servidores. La clase Status debe ser visible para el cliente ya que contiene todos los objetos que representan las alarmas y estados que puede recibir del robot. Y por último, el paquete IDs es necesario para utilizar los parámetros de configuración de la aplicación (que se comentaron anteriormente), además del paquete ComExceptions necesario ya que los métodos que el cliente implementa de las interfaces Listener y Updater lanzan excepciones contenidas en este paquete.

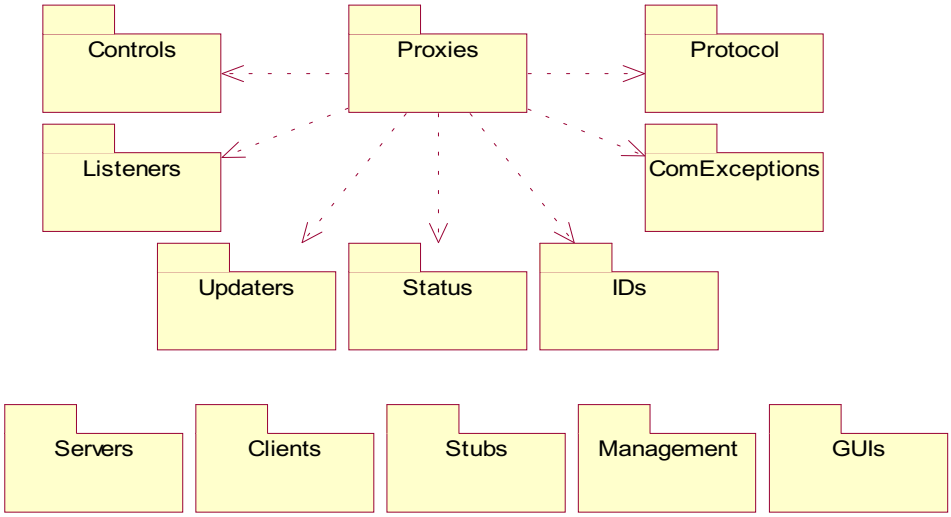


Una característica importante es que el cliente no depende del paquete Protocol, y esto permite usarlo en otra posible aplicación en la que se use otro tipo de protocolo, y todo ello beneficia la modularidad de la aplicación. El cliente además, no conoce a ningún proxy de la aplicación ya que solo trabaja con interfaces, siendo completamente independiente de la implementación de dichas interfaces por otras posibles clases (ejemplo: El cliente solo conoce la interfaz MechanismCtrl, pero no el objeto MechanismProxy que la implementa, es decir, la implementación es transparente al cliente). Lo mismo ocurre con el paquete Servers, en donde se presentan una posible implementación de los servidores, de la que el cliente no tiene por que depender ya que trabaja solo con interfaces "Control", de forma que se pueden usar otro tipo de servidores, solamente se pide que implementen las interfaces correspondientes (Control). El cliente tampoco conoce a ningún stub ya que al igual que el Proxy, entra dentro de la capa del middleware, que es transparente a la capa de aplicación donde se encuentra el cliente. El resto de paquetes de los que no depende son obvios, como son las interfaces gráficas (GUIs) y las clases de control de inicio y cierre de aplicación (Management).

- **Paquete Proxies:**

Los proxies definidos en la aplicación dependen del protocolo ya que son los encargados de transformar las peticiones de los clientes en envíos de mensajes mediante el protocolo definido a un nivel más bajo. Todo ello puede lanzar excepciones de tipo ComException, y necesita usar tanto los parámetros (identificadores) de configuración de la aplicación como los objetos de estado del robot. Es indiscutible que los proxies dependan directamente de las interfaces Controls ya que son las que ellos implementan para poder realizar

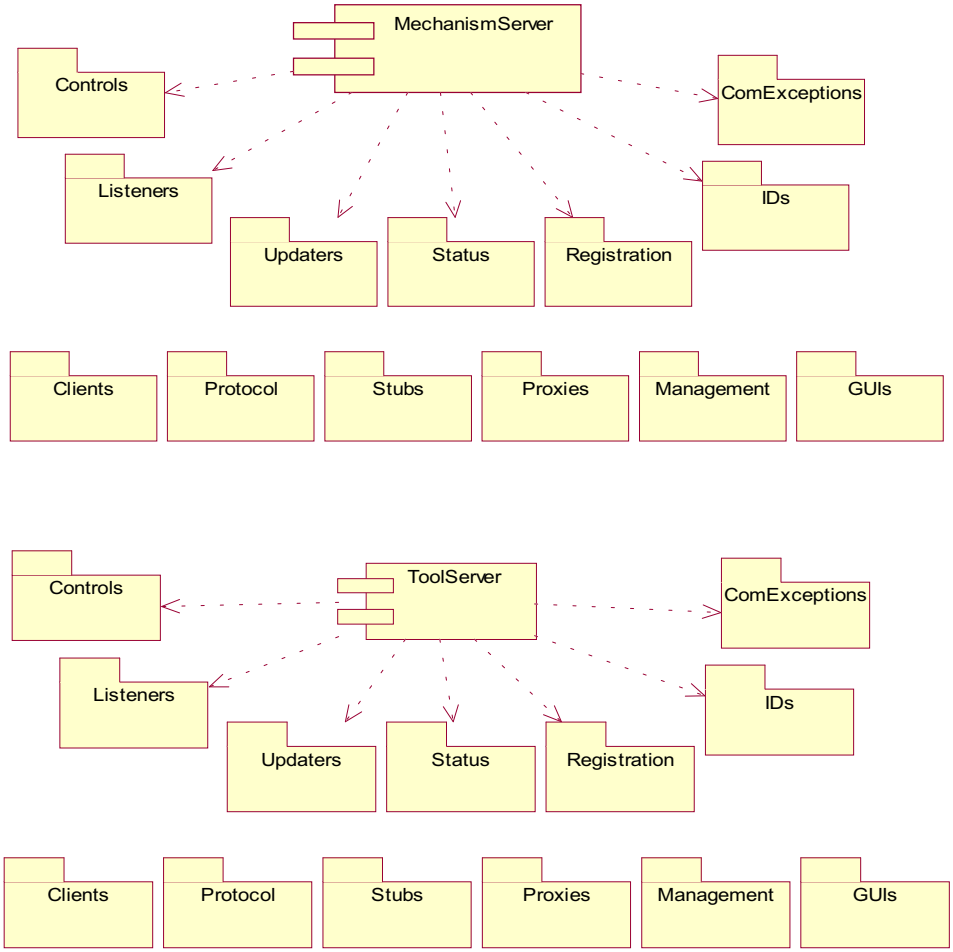
su labor de representante de los servidores. Las interfaces Listeners y Updaters son necesarias para poder trabajar con el cliente cuando es necesario devolver una respuesta a un cliente.



Los proxies no dependen de la implementación que se use como servidor, ni como cliente, ya que trabaja con las interfaces que deben implementar, no con la propia implementación. Los stubs no los conocen porque trabajan en el extremo opuesto de la comunicación, y tampoco depende de las interfaces gráficas ni de las clases de control de inicio y cierre de aplicación (Management).

- **Clases MechanismServer y ToolServer:**

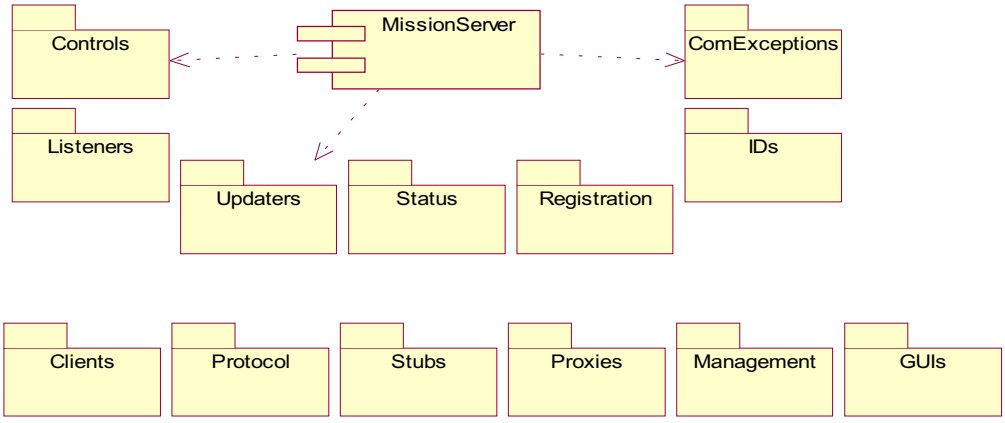
Estudiaremos las dependencias que existen entre estas dos clases conjuntamente debido a las similitudes que presentan, en las que únicamente existe una mínima diferencia debido al tipo de servidor que son cada uno (Mechanism vs Tool). La principal dependencia es con la interfaces Control que deben implementar por el hecho de ser servidores, mientras que las interfaces de tipo Listener y Updater junto con el paquete Status deben ser visibles para poder responder a los clientes que soliciten el estado del robot. Por otro lado, son necesarias las excepciones para relanzarlas en los métodos que implementen, y también los parámetros de configuración de la aplicación. Por último, el paquete Registration debe ser accesible para poder usar el servicio de registro de clientes comentado inicialmente.



Como era de esperar, los servidores son independientes de la implementación de los clientes, de los stubs y del protocolo usado a nivel más bajo, ya que trabajan únicamente con las interfaces definidas en los paquetes Controls, Listeners y Updaters.

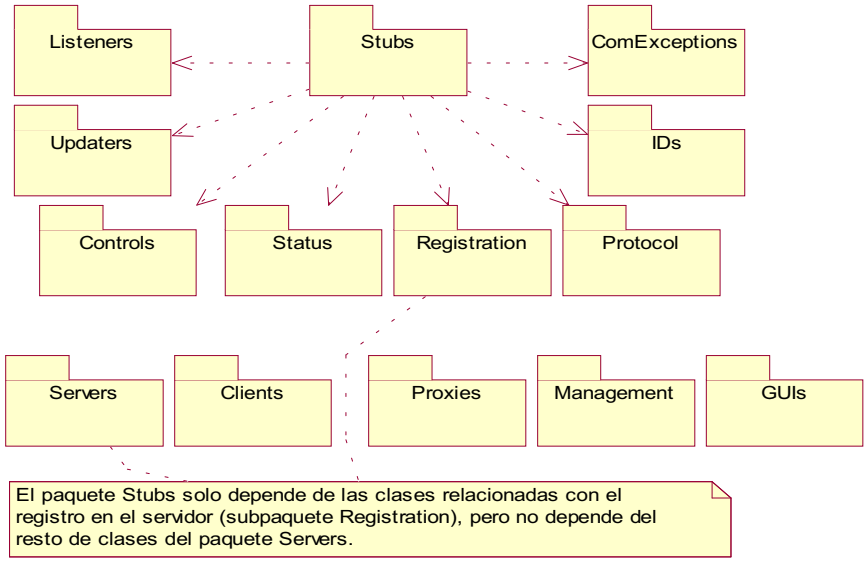
- **Clase MissionServer:**

Esta clase hay que estudiarla separadamente de los servidores de Mecha nism y Tool debido a que este tipo de servidor no necesita implementar el servicio de registro de clientes, por lo que contiene las mismas dependencias que los otros dos tipos de servidores, pero exceptuando los paquetes relacionados con el registro de clientes (Registration, Listeners, Status e IDs).



- **Paquete Stubs:**

Por último, podemos comprobar que los stubs también deben estar relacionados con el protocolo ya que son los encargados de transformar los mensajes que recibe del proxy a través del enlace en llamadas a métodos del servidor. Todo ello puede lanzar excepciones de tipo ComException, y necesita usar tanto los parámetros (identificadores) de configuración de la aplicación como los objetos de estado del robot. Es incuestionable que los stubs dependan directamente de las interfaces Listeners y Updaters ya que son las que ellos implementan para poder realizar su labor de representante de los clientes. Las interfaces Controls son necesarias para poder trabajar con los servidores, ya que acceden a ellos a través de dichas interfaces. Cabe destacar que los stubs necesitan el paquete Registration ya que serán los stubs los que almacenen a los clientes reales que se registren, y el propio stub se registrará en el servidor (no los clientes). Aunque el paquete Registration pertenezca al paquete Servers, los stubs no dependen de los servidores, ya que trabajan con las interfaces Controls, por lo que la implementación de dichas interfaces es indiferente para los stubs.



Comprensiblemente, los stubs tampoco dependen de la implementación de los clientes ya que trabajan únicamente con las interfaces Listener y Updater. Los stubs tampoco conocen a los proxies porque están localizados en el extremo opuesto de la comunicación, que el protocolo oculta a las capas de nivel más alto.

Ahora ya conocemos las dependencias más importantes entre algunos componentes de la aplicación, y podemos llegar a la conclusión de que el uso de interfaces bien definidas beneficia el ocultamiento de la información y ayuda a tener un diseño correcto, fácil de mantener y modular. La idea es ofrecer una serie de servicios a un cliente, pero sin importarle a dicho cliente la implementación de dichos servicios.

8. Etapas de creación de la aplicación.

A continuación se describirán las etapas seguidas en el diseño de la aplicación Goya, que ayudarán a comprender mejor el orden que se debe seguir a la hora de implementar una aplicación cliente / servidor de estas características, y que requiera la creación de un protocolo interno que defina las reglas necesarias para el paso de mensajes a través de los enlaces de comunicación y, en definitiva, que permita una comunicación transparente entre cliente y servidor.

- Protocolo interno.

El estudio que se realizo sobre los protocolos orientados a conexión en el capítulo referente a la arquitectura cliente / servidor, proporciona una serie de razones que son causa directa de que el protocolo creado trabaje sobre el Protocolo de Control de Transmisión (TCP). La implementación de dicho protocolo, debía ofrecer un conjunto de clases y métodos que permitieran la transmisión de mensajes entre maquinas distintas de una forma transparente, es decir, sin tener en cuenta la localización de las maquinas. Una de las decisiones a tomar para el diseño del protocolo, fue el formato de los mensajes a enviar, ya que su complejidad no debía ser muy elevada y su tamaño tampoco excesivo ya que podría dar lugar a problemas de rendimiento. Finalmente, y tras varios estudios, se decidió que el formato final de estos mensajes constaría de un identificador de tipo de mensaje y un conjunto de parámetros (números enteros, cadenas de caracteres, etc...) según el tipo de mensaje a enviar.

Otro de los detalles a tener en cuenta en diseño era el número de hilos (Threads) q ue se debían tener en ejecución, ya que aunque este parámetro de depende en gran medida del rol que se realice en cada situación (ya sea cliente o servidor), no es aconsejable abusar del rendimiento del procesador porque podría dar lugar a unas necesidades hardware muy exigentes. Finalmente la decisión fue la de tener un hilo encargado de la transmisión de mensajes, dos mas para la recepción y análisis de los mensajes entrantes para cada enlace.

Además de los enlaces propios de la transmisión, para permit ir un control centralizado de cada uno de dichos enlaces (como por ejemplo determinar los puertos donde trabajar), se decidió la implementación de un enlace principal que permitiera la creación y la eliminación de cada uno de los enlaces de transferencia. Así de este modo, mediante la creación de un objeto de este tipo se podrían crear todos los elementos de comunicaciones (sockets) necesarios para el correcto funcionamiento de la aplicación.

El ultimo detalle que se decidió implementar para el protocolo interno, fue la inclusión de un conjunto de excepciones de creación propia llamadas "ComException" que determinaran con su mensaje donde se produjo dicha excepción, para que de este modo se pueda solucionar de forma más rápida los problemas ocasionados.

- Proxies y Stubs.

Una vez creado el protocolo y probado en las diversas situaciones con una interfaz de usuario sencilla, se decidió implantar los que se podrían denominar "elementos de transparencia", los proxies y los stub, es decir, elementos cuya función, como ya se ha explicado con detalle en puntos anteriores, sea suplantar al extremo opuesto para que, de este modo, los usuarios de la aplicación no adviertan ninguna diferencia si el cliente y el servidor esta en la misma maquina o no. Como es normal, estos elementos quedarán en un nivel superior al protocolo interno, y de este modo, siempre la implementación de los métodos de estos elementos, tendrán su "traducción" con los elementos de dicho protocolo.

Con estos elementos, ya se produce una especialización de la aplicación, ya que hasta este punto, es decir el protocolo interno, estaba implementado para ser utilizado para cualquier aplicación distribuida, sin importar su funcionalidad. Pues bien, una vez implantados los proxies y stubs, se marcan los métodos, es decir, la utilidad que se le va a otorgar a la aplicación. Es por esto, que estos elementos implementaran todas las interfaces que se nos ofrecieron como objetivos de la aplicación: Updaters, Listeners y Controls.

- Clientes y Servidores.

Estando ya creados los stubs y proxies, ya se pueden utilizar todos los métodos que estas clases ofrecen sin dar importancia a las comunicaciones entre las maquinas, ya que estos elementos se ocuparan de esto. De este modo, los elementos que utilizarán estas facilidades serán los clientes y los servidores que implementaran las mismas interfaces que los stubs y proxies, y ya serán los elementos más externos de la aplicación y con los que el usuario tendrá "contacto". Así, al igual que a los stubs y proxies le "pasaban" los métodos a los enlaces de transferencia, los clientes y servidores realizan la misma operación, llamando para cada uno de sus métodos, a los métodos de los stubs y proxies, formándose una cadena bidireccional (cliente ←→ proxy ←→ link) y (servidor ←→ stub ←→ link).

- Contenedores.

Por ultimo, hay que destacar que para ofrecer un acceso centralizado a cada uno de los servidores y clientes (y sus elementos necesarios), se decidió implementar dos contenedores (uno para cada extremo) donde se encontraran todos los elementos necesarios para el control total de la aplicación, como son: clientes, proxies, y links; y por otro lado: servidores, stubs y links.

- Registration.

Este es el nombre del paquete donde se encuentran todas las clases que ofrecen las utilidades de registro y "temporización" de servidores. Al ser estos últimos los que utilizaran dichas clases, se decidió colocar este paquete dentro del propio paquete "Servers". La principal utilidad que se tuvo que implementar en estas clases fue el registro de los servidores y el control del tiempo para que de este modo, los clientes pudieran recibir las actualizaciones oportunas de forma periódica y exacta.

- Status.

Por ultimo, y ya de forma muy especializada para la aplicación en cuestión, se implementaron un conjunto de clases e interfaces como son Alarm, BlastingStatus…que se incluirán dentro del paquete "Status" y que definen algunas de las características del robot para el cual se diseñó la aplicación, pero sin olvidar en todo momento, que la aplicación esta adecuada para poder ser utilizada por cualquier robot con unos simples cambios.

9. Aspectos característicos de la implementación.

A continuación se comentan los puntos más importantes de la aplicación Goya, y que han presentado mayores problemas a la hora de su implementación. Sin embargo, todos los diagramas UML adjuntados en este proyecto pueden facilitar la comprensión de muchos de los aspectos conflictivos que presenta la aplicación.

9.1. Diseño de la arquitectura cliente/servidor.

En la realización de una arquitectura cliente – servidor donde la aplicación es distribuida es muy común el uso del Proxy y Stub (representantes), que se encarguen de separar completamente a los clientes y servidores del tipo de protocolo utilizado por la aplicación.
La idea se basa en que cada cliente crea que su servidor asociado no esta distribuido, sino en la misma maquina donde él se encuentra, y de igual forma con el servidor y su cliente asociado en el extremo opuesto de la aplicación. Sin embargo, tanto servidor como cliente pueden estar distribuidos. Esta arquitectura se puede conseguir con el uso de proxies y Stubs, en el que un Proxy será un representante del servidor en el extremo del cliente, y un Stub será un representante del cliente en el extremo del servidor. Esta arquitectura conviene estudiarse en cada extremo de la aplicación:

Extremo del cliente: En el extremo del cliente se implementara un Proxy que implemente la misma interfaz que el servidor asociado a dicho cliente, de forma que el cliente considere que su servidor asociado sea el propio Proxy. De este modo, el cliente trabaja con su servidor de forma local, y el encargado de transformar estas llamadas locales a llamadas remotas al servidor real será el mismo Proxy.

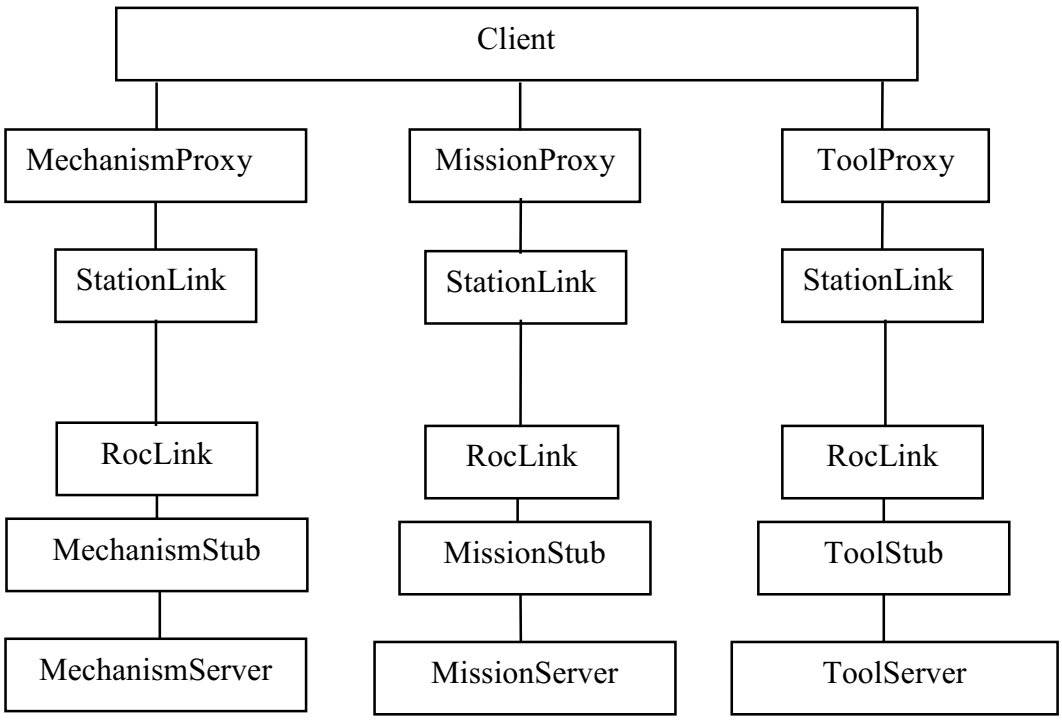
Extremo servidor: En el extremo del servidor se implementara un Stub que implemente la misma interfaz que el cliente asociado a dicho servidor, de forma que el servidor considere que su cliente asociado sea el propio Stub. De este modo, el servidor trabaja con su cliente de forma local, y el encargado de transformar estas llamadas locales a llamadas remotas al cliente real será el mismo Stub.

Consecuentemente, el Proxy se comunica con el Stub y viceversa, convirtiendo ambos también las llamadas remotas de su opuesto (Proxy o Stub en cada caso) en llamadas locales a su cliente o servidor asociado. Esta arquitectura Proxy – Stub es interesante ya que hace que el cliente y el servidor sean independientes del protocolo que trabaje por debajo de ellos, ya que estará oculto por el Proxy y el Stub.

El protocolo que se usara debajo de la capa del Proxy y el Stub esta definido por dos clases, una clase que representa al servidor de comunicaciones (RocLink), y otra que representa al cliente de comunicaciones que se conectara a dicho servidor (StationLink). Estas clases trabajan a su vez sobre el protocolo TCP/IP y proporcionan toda la funcionalidad necesaria para la creación de un enlace de comunicaciones, transMission y recepción de mensajes y cierre del enlace. De este modo ya hemos definido la aplicación en varias capas de abstracción, que cabe destacar:

Servidor	**Cliente**
Stub	**Proxy**
RocLink	**StationLink**
Protocolo TCP/IP	

La aplicación sigue esta arquitectura, pero debido a que esta formada por varios servidores y de distinto tipo (Mechanism, Tool y Mission) conviene hacer de nuevo el planteamiento, que es el que muestra la figura:



Aunque existen tres tipos de proxy (MechanismProxy, ToolProxy, MissionProxy), todos ellos heredan de la clase “Proxy”. Esta arquitectura es necesaria para englobar todos los tipos de proxies en una clase generalizada, ya que al obtener una referencia del proxy asociado a un determinado enlace (StationLink) no se conoce el tipo al que puede estar asociado dicho enlace (Mechanism, Tool o Mission) y es necesaria esta clase que los englobe a todos. El tipo de proxy concreto que es necesario una vez tenemos un objeto “Proxy” se obtiene haciendo cast sobre la superclase Proxy, el tipo de clase a hacer cast dependerá de un identificador que contiene la clase madre.

Para el caso de los stubs ocurre lo mismo, ya que existen tres tipos de stub (MechanismStub, ToolStub, MissionStub), todos ellos heredan de la clase “Stub”. Esta arquitectura es necesaria para englobar todos los tipos de stubs en una clase generalizada, ya que al obtener una referencia del Stub asociado a un determinado enlace (RocLink) no se conoce el tipo al que puede estar asociado dicho enlace (Mechanism, Tool o Mission) y es necesaria esta clase que los englobe a todos. El tipo de Stub concreto que es necesario una vez tenemos un objeto “Stub” se obtiene haciendo cast sobre la superclase Stub, el tipo de clase a hacer cast dependerá de un identificador que contiene la clase madre.

9.2. Creación de los enlaces de comunicaciones.

En el caso en que se trabaje con un cliente la aplicación estará formada por un enlace de control y mantenimiento de la aplicación y tres enlaces de control del robot (un enlace para controlar cada división del robot). En el caso de trabajar con múltiples clientes (distribuidos en varias maquinas) existirán tantos enlaces de control de la aplicación como maquinas cliente existan, y los tres enlaces adicionales de control del robot.

En la maquina servidor se crean por tanto un servidor de control de la aplicación, y otros tres servidores que controlaran el robot y que proporcionan la funcionalidad necesaria para controlar y mantener el robot a través de varios enlaces de comunicaciones. Estos tres servidores dividen en tres capas el comportamiento del robot, estas capas son las formadas por la “herramienta” (Mechanism), el “brazo” del robot (Tool), y la Mission actual del robot (Mission).

El primer paso para la creación de la aplicación es arrancar el servidor de control, que se encargara de arrancar los tres servidores del robot en una serie de puertos que el mismo elegirá,

una vez que estos tres servidores estén a la espera de alguna petición el servidor de control comenzará también a esperar peticiones de conexión.

Cuando un cliente desee conectarse a los servidores del robot necesitará conocer la dirección IP y los puertos donde se ejecutan los mismos. El protocolo tiene establecido que la única información que el cliente debe conocer acerca del extremo opuesto es la dirección IP de la maquina y el puerto del servidor de control de la aplicación. Este puerto y dirección IP serán fijos y necesarios para que el cliente pueda acceder al robot. El proceso a seguir por el cliente para acceder al robot es el siguiente:

Conocida la dirección IP y el puerto donde esta corriendo el servidor de control de la aplicación, el cliente se conectara al mismo y enviara una petición de acceso a los tres servidores que controlan el robot. El servidor de control atenderá dicha petición enviando al cliente los puertos donde están corriendo los servidores del robot, de forma que el cliente ya dispondrá de los dos parámetros necesarios para acceder a ellos (la dirección IP que ya conocía y los puertos recibidos por el enlace de control).

El cliente se conectara conociendo dicha información a cada uno de los tres servidores del robot, y acaba de esta forma el proceso de creación de los enlaces necesarios para la comunicación cliente/s – servidor/es.

En la creación de los enlaces del protocolo diseñado se ha utilizado un mecanismo de reconexión automático en caso de fallo en las comunicaciones. De este modo, tanto los enlaces de control del robot (Links) como los enlaces de control del protocolo (CtrlLinks) están dotados del siguiente mecanismo:

Si ocurre un error de conexión en el cliente, éste intentara conectarse de nuevo al servidor creando un nuevo Socket. En caso de que ocurra un segundo error de conexión el cliente intentara conectarse al servidor pero esta vez en un puerto distinto (evitando posibles bloqueos de un puerto determinado). Es importante que en el servidor se cree el ServerSocket asociado al puerto alternativo (reservado de antemano por la aplicación) en caso de fallo para que el cliente pueda conectarse. Si un tercer error surgiera, el cliente dará por perdidas las comunicaciones y no intentara reconectar de nuevo con el servidor. Para que este mecanismo funcione correctamente es necesario sincronizar cliente y servidor en cuanto al numero de errores se refiere; esto se ha gestionado con un contador que indica el numero de errores que se han sucedido en la conexión de un cliente, y en función de dicho contador se optará por reconectar de un modo u otro, o cesar los intentos.

Aunque se ha explicado el proceso de creación de conexión con un único cliente, pueden recibirse varias peticiones sobre el mismo puerto y consecuentemente sobre el mismo servidor. Las peticiones de conexiones de clientes se almacenan en el puerto, para que el servidor pueda aceptarlas de forma secuencial. Sin embargo, puede servirlas simultáneamente a través del uso de threads.

9.3. Creación de la arquitectura de la aplicación.

Una vez creados los enlaces sobre los que trabaja la aplicación se ha de crear el resto de la arquitectura que anteriormente se ha mencionado.

Extremo servidor:

El primer paso es la creación de los stubs en el extremo del servidor, y que se enlazaran a los objetos RocLink creados para poder convertir las llamadas a procedimiento locales en envío de mensajes a través del enlace de comunicaciones; la manera de enlazar cada stub con su enlace servidor asociado (RocLink) es mediante el constructor de cada stub, que toma una referencia a su correspondiente objeto RocLink. De este modo conseguimos que el stub se pueda comunicar con el enlace, pero no el enlace con el stub, necesario para que los mensajes que envía el proxy (cliente) sean convertidos a llamadas a procedimiento del servidor, por lo que es necesario también pasar una referencia de su stub asociado al objeto RocLink, mediante el método *addStub(Stub Stub).*

El siguiente paso es crear los servidores del robot de tipo Mechanism, Tool y Mission, que proporcionen la funcionalidad necesaria para el control del robot. Para que cada Stub pueda comunicarse con su servidor asociado es necesario pasarle una referencia de dicho servidor después de crearse, mediante los métodos *addServer(MechanismServer server)*, *addServer(MissionServer server)*, *addServer(ToolServer server)*. Se usara el método que proceda dependiendo del tipo de servidor a enlazar.

La comunicación en sentido stub → servidor es única, pero la comunicación en sentido servidor → stub no es tan evidente ya que pueden existir varios clientes para un mismo servidor. Esto se solventa pasando a cada método que el cliente llame y que requiera una comunicación con sentido servidor a stub (métodos de actualización) una referencia al propio cliente, para que el servidor pueda distinguir a qué cliente devolver la respuesta que dicho método solicita. El problema en este caso es que el stub es el único representante de todos los posibles clientes que hay para el servidor, por lo que dicho stub almacena todas las referencias a los clientes que

hayan llamado a estos métodos de actualización, y el que se enlaza como cliente en el servidor es el propio stub (ya que implementa la misma interfaz que todos los clientes). Cuando el servidor devuelve la respuesta a dicho stub, éste se encarga de redireccionarla al cliente correspondiente al que iba dirigida la respuesta, de forma que el servidor ignore los múltiples clientes que pueda tener distribuidamente.

Extremo cliente:

El primer paso en el extremo del cliente una vez creados los enlaces de comunicaciones sobre los que trabajara la aplicación, es la creación de los proxies en el extremo del servidor, que se enlazaran a los objetos StationLink creados para poder convertir las llamadas a procedimiento locales en envío de mensajes a través del enlace de comunicaciones; la manera de enlazar cada Proxy con su enlace asociado (StationLink) es mediante el constructor de cada Proxy, que toma una referencia a su correspondiente objeto StationLink. De este modo conseguimos que cada Proxy se pueda comunicar con su enlace correspondiente, pero no el enlace con el Proxy, necesario para que los mensajes que envía el Stub (servidor) sean convertidos a llamadas a procedimiento del cliente, por lo que es necesario también pasar una referencia de su Proxy asociado al objeto StationLink, mediante el método *addProxy(Proxy Proxy)*.

El siguiente paso es crear los clientes que se deseen para controlar los tres tipos de servidores (Mechanism, Tool y Mission). Estos clientes se deben de enlazar con su servidor, que realmente será el Proxy (representante del servidor en el cliente) ya que implementa la misma interfaz que los servidores. La forma de enlazar cada cliente con su Proxy correspondiente es pasándole una referencia al cliente mediante el método *addCtrl(MechanismCtrl ctrl)*, *addCtrl(MissionCtrl ctrl)* y *addCtrl(ToolCtrl ctrl)*. Se usara el método que proceda atendiendo al tipo de servidor (Proxy) a enlazar. Como el método puede tomar una referencia a cualquier objeto que implementa la interfaz del servidor (MechanismCtrl, MissionCtrl, y ToolCtrl), se le pasara una referencia al Proxy aunque el cliente considerara que es el propio servidor, con el que trabajara localmente. Sin embargo, el Proxy convertirá las llamadas a métodos locales del cliente en llamadas remotas al servidor, que generalmente se encontrara en otra maquina.

Con los procedimientos anteriores conseguimos la comunicación en sentido cliente $\rightarrow$ proxy, pero no en sentido proxy $\rightarrow$ cliente. Como este sentido de llamadas solo tiene sentido cuando el servidor devuelve una respuesta al cliente y el hecho de que puedan existir varios clientes en la misma maquina, la solución es la de pasar una referencia del cliente que quiera una respuesta en el método de petición de dicha respuesta. Esta referencia del cliente traspasara

los enlaces de comunicaciones y se insertara en el método de respuesta de la anterior llamada, de forma que el proxy ya pueda referirse al cliente que solicitaba la respuesta.

Llegados a este punto ya se ha creado la arquitectura completa de la aplicación, que básicamente simplifica la implementación del servidor y del cliente, ya que son completamente independientes del protocolo en el que se trabaja a nivel más bajo. Tanto el servidor como el cliente consideran que sus opuestos están en la misma maquina y pueden comunicarse mediante llamadas locales (entre objetos), pero realmente hay un protocolo por debajo de esta capa que se encarga de distribuir todas las llamadas locales en llamadas remotas a sus clientes o servidores correspondientes, y todo ello bajo el protocolo TCP/IP con el que los sockets de Java trabajan.

Resumiendo los pasos necesarios para obtener una comunicación bidireccional entre los objetos de nuestra arquitectura, obtenemos el siguiente esquema:

- Se crean tres objetos RocLink.
- Se crean tres stubs (Mechanism, Tool y Mission), se le pasa en el constructor una referencia a su RocLink asociado.
- A cada objeto RocLink se le pasa una referencia a su stub asociado con *addStub(...)*.
- Se crean tres servidores (Mechanism, Tool y Mission), y a cada stub se le pasa su servidor asociado con el método *addServer(...)*.
- El servidor no contiene ninguna referencia a el cliente (stub) hasta que ninguno se registre o solicite una respuesta del servidor, en el primer momento se registrara el stub ya que es el representante del cliente, y dicho stub distribuirá la respuesta del servidor al cliente que envió la petición.
- Se crean tres objetos StationLink (que se enlazan con sockets a los objetos RocLink).
- Se crean tres proxies (Mechanism, Tool y Mission), se le pasa en el constructor una referencia a su StationLink asociado.
- A cada objeto StationLink se le pasa una referencia a su proxy asociado con *addProxy(...)*.
- Se crea un cliente, y se le pasan las tres referencias a los servidores (proxies) con *addCtrl(...)* para que pueda enviar peticiones y ordenes. Cada Proxy representa a el tipo de servidor asociado en la maquina del cliente, y convertirá sus llamadas locales en llamadas remotas a través de los objetos StationLink. Para la comunicación en sentido Proxy a cliente, se resuelve enviando el cliente una referencia a si mismo en cada método que requiera una respuesta del servidor, el servidor al recibir este tipo de peticiones, insertara dicha referencia en la respuesta, de forma que al llegar al Proxy a

través del enlace de comunicaciones, el sabrá con dicha referencia a que cliente va dirigida.

Ejemplo:

servidor.getAlarm(this); (en el cliente)

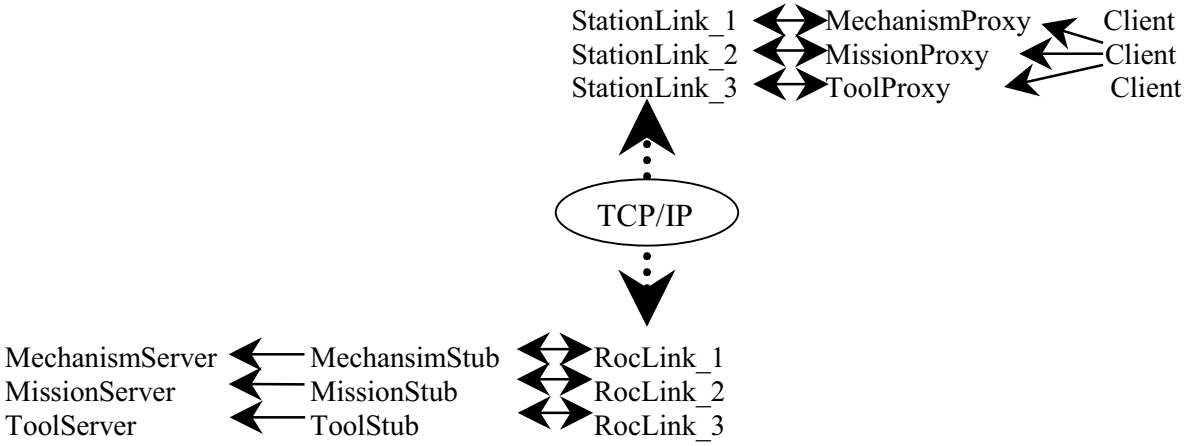
(proxy)

void getAlarm(Client c){ cliente.updateAlarm(c); } (en el servidor)

(stub)

c.updateAlarm(); (en el proxy)

- Se pueden crear varios clientes en la misma maquina realizando el ultimo punto, y varios clientes en distinta maquina realizando los últimos cuatro pasos.

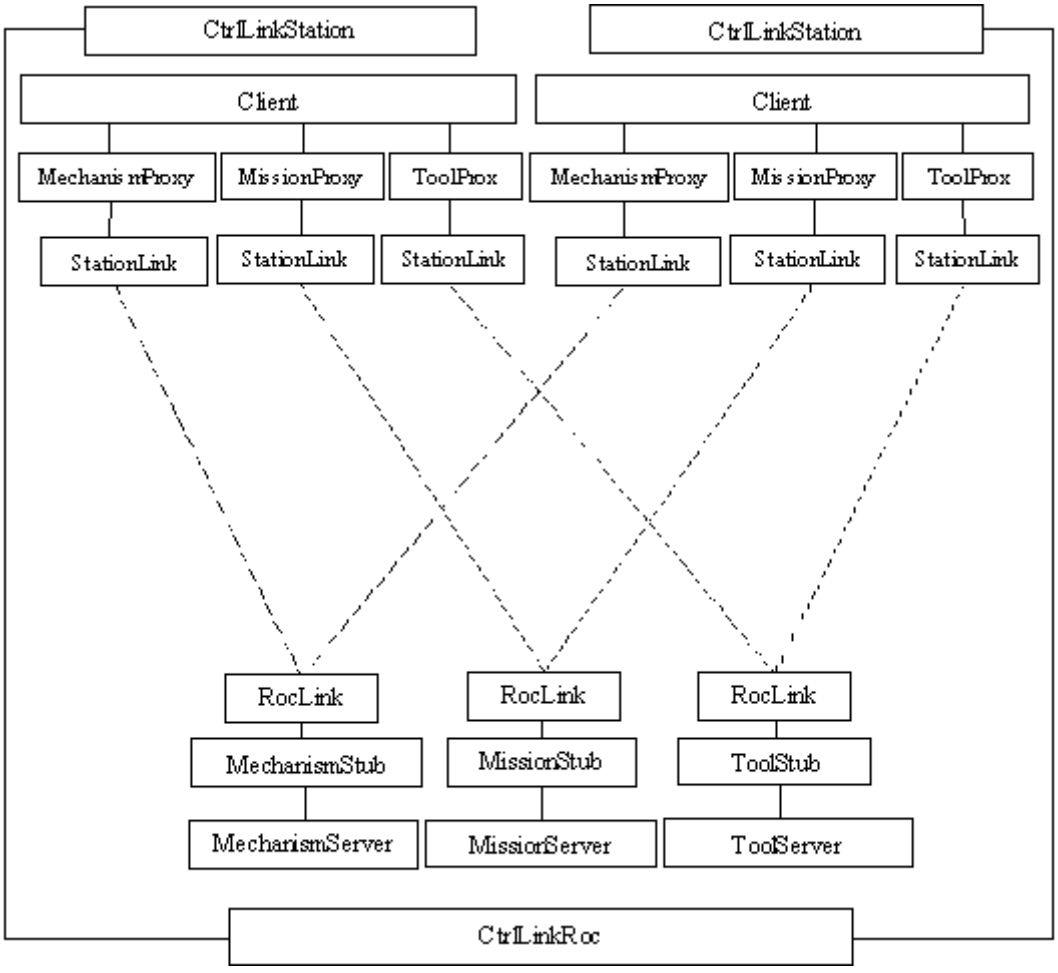


Esta arquitectura permite que la implementación del cliente y el servidor sea completamente independiente del protocolo que trabaje debajo, y las dos peculiaridades que permiten esto son:

- El cliente contiene una referencia a un objeto que implemente la interfaz del servidor (Ctrl), pero realmente dicho objeto no es el mismo servidor sino un Proxy que implementa esta interfaz y actúa de representante del servidor en la maquina del cliente. El cliente trata a este Proxy como al servidor, desconociendo la implementación del mismo.

- En el extremo opuesto ocurre algo similar, ya que el servidor contiene una referencia a un objeto que implemente la interfaz del cliente (puede contener más ya que admite varios clientes), pero este objeto no es realmente un cliente sino un representante del mismo llamado stub, que implementa la misma interfaz del cliente. De este modo, el servidor solo considera la interfaz, sin conocer la implementación del objeto que la implementa.

La arquitectura formada en el caso de que dos clientes se conecten al servidor, y cada uno de ellos en distintas máquinas, sería como la de la figura:



Además de esta figura, las figuras número 5 y 6 que se presentan en el apartado de modelado del sistema mediante UML contienen la arquitectura de la aplicación en cada extremo (cliente y servidor), incluyendo relaciones entre enlaces, proxies, stubs, clientes y servidores. Si no ha quedado claro el proceso de creación de la aplicación, se puede observar el diagrama de sencuencia número 25, que indica de forma concisa todo el proceso comentado anteriomente.

9.4. Soporte multicliente.

Es importante que el protocolo que trabaje por debajo de la aplicación permita la existencia de múltiples clientes de forma distribuida, y la implementación de esta funcionalidad en el servidor es de máxima importancia. Principalmente se basa en la existencia de threads que permite que se atienda una nueva petición de conexión mientras esta conectado con otra máquina, sin que se bloquee el sistema.

Cabe destacar el procedimiento utilizado por el servidor para distribuir las respuestas de peticiones de los clientes a las distintas maquinas donde se encuentran éstos. Básicamente se basas en que cada enlace de comunicaciones entre el cliente y el servidor contiene un socket y unos flujos de lectura y escritura asaciados, que se van almacenando en arrays conforme se van creando estos enlaces. El punto mas importante en esta parte es el saber en que posición de los arrays se almacena cada socket para poder obtener el enlace correcto asociado al cliente al que se debe dirigir la respuesta.

Estos sockets (y flujos I/O) no se van almacenando en cualquier posición de los arrays ya que esta posición es crítica para el funcionamiento de la aplicación, por lo que se debe seguir cierto criterio a la hora de almacenarlos y obtenerlos. Esta posición donde se almacenara debe ser igual a un identificador que cada cliente contiene de forma que conociendo al cliente donde va dirigida una respuesta se pueda obtener el enlace por donde enviarla, y en consecuencia la posición del array donde esta almacenado el socket en cuestión.

El criterio fijado para este posicionamiento de los sockets de los enlaces es el siguiente: Los sockets se van almacenando secuencialmente en los arrays conforme se van creando enlaces, por lo que la posición del primer enlace creado será el 0, al siguiente enlace creado le corresponde el 1, al siguiente el 2, etc. Una vez que están almacenados dichos sockets, solo falta saber acceder a la posición adecuada en cada caso en que se quiera enviar una respuesta a un cliente determinado.

Esta cuestión plantea ciertos problemas de implementación, pero finalmente el hecho de que cada método que requiera una respuesta del servidor se insertara una referencia al cliente que solicita dicha respuesta ha solventado el problema. Simplemente con añadir un método al cliente que lo identifique con el enlace al que esta asociado, de forma que usando ese método en el servidor, usando la referencia a dicho cliente que se inserto en la petición, se pueda obtener el enlace por donde distribuir la respuesta.

Ejemplo:

servidor.getAlarm(this); (en el cliente)

(proxy)

void getAlarm(Cliente cliente){ cliente.updateAlarm(cliente); } (en el servidor)

(stub)

void updateAlarm(Cliente cliente) { rocLink.send(mensaje, cliente.getLinkID());

(en el stub)

void send(Mensaje msg, int linkID) { outStreams[linkID].writeObject(msg); }

(en RocLink)

Es importante destacar que aunque en el ejemplo se especifique que en el servidor se trabaja con referencias a objetos de la clase "Client" para una simplificación de la descripción, realmente se trabaja con referencias a objetos que implementan las interfaces "Listener" y/o "Updater" que implementa el cliente. Con esto se consigue que el servidor no dependa de la implementación del cliente, siendo solo visibles para él las interfaces anteriores que implementa. Incluso se puede disponer de clientes con distintas implementaciones (que implementen las mismas interfaces de forma distinta) y el código del servidor permanecería inalterable, lo que se conoce como bajo acoplamiento entre objetos.

Una vez que se conoce el criterio a seguir en el servidor, solo falta conocer como se adjudica a cada cliente su identificador de enlace asociado. Este procedimiento se realiza al crear cada cliente de la siguiente forma:

- Existe una variable entera que contendrá el número de enlaces creados hasta el momento.
- Si se crea un enlace se incrementa esta variable.
- Si se crea un cliente se adjudica el valor de dicha variable como identificador de enlace.

Este criterio es análogo al utilizado en el servidor para almacenar los sockets, de forma que coincidirá con la posición solicitada por el identificador de cada cliente. Este procedimiento de distribución de peticiones y respuestas hacia todos los posibles clientes se lleva a cabo gracias la existencia de varios threads como se comento anteriormente. Estos threads evitan que las comunicaciones sean bloqueantes y un servidor pueda atender a varios clientes simultáneamente. El flujo lógico básico en el servidor sería como éste:

while (true) {
aceptar una conexión;
crear un thread para tratar a cada cliente;
}

Los métodos de lectura y escritura que proporciona Java son bloqueantes. Esto quiere decir que si pedimos leer datos, el proceso se detiene hasta que los datos hayan sido leídos. Similarmente ocurre con la escritura, que el proceso que quiere escribir se ve detenido hasta que los datos han sido escritos. Consecuentemente, las circunstancias bloqueantes que se pueden presentar en la aplicación son:

- Esperar petición de un cliente en el servidor → método *accept()* de la clase ServerSocket.
- Enviar un objeto a través de un enlace de comunicaciones → *writeObject()* de la clase ObjectOutputStream.
- Leer un objeto enviado a través de un enlace de comunicaciones → método *readObject()* de la clase ObjectInputStream.

La forma de evitar que algunos de estos métodos pueda bloquear la aplicación es encapsulándolos en threads que permitan que la aplicación pueda aceptar peticiones, y transmitir y recibir mensajes simultáneamente.

El thread encargado de despachar peticiones y que permite trabajar simultáneamente con otras conexiones ya creadas es la clase "Connector", que se usa tanto en el extremo del servidor como del cliente. En el extremo del servidor (tanto de los enlaces del robot como el de control) se encarga de crear el servidor en el correspondiente puerto y estar permanentemente aceptando peticiones de conexión de los clientes. Cada vez que llega una nueva petición se encarga de invocar a un método que crea otro thread que estará escuchando a través del enlace creado para despachar las peticiones de los clientes. Mientras que en el extremo del cliente el thread "Connector" se usa para conectarse al cliente que, aunque sea no bloqueante evita retardos a causa de posibles problemas en el establecimiento de una conexión.

Conjuntamente con el thread Connector existe otro tipo de thread que se encarga de enviar mensajes (objetos) a través de un enlace de comunicaciones de forma no bloqueante, ya que el método que proporciona Java para enviar objetos a través de un flujo de comunicaciones es bloqueante, y si el mensaje a enviar es de un tamaño considerable la aplicación puede bloquearse espontáneamente. Este thread permite al procesador compartir la acción de enviar

mensajes con el resto de funciones de la aplicación, evitando cualquier tipo de bloqueo. La clase se conoce con el nombre de "Transmitter", y simplemente se encarga de enviar un objeto a través de un flujo de escritura.

Anteriormente se comento que por cada conexión que se creaba entre el servidor y un nuevo cliente se creaba además un thread encargado de despachar las peticiones de dicho cliente, este thread es el llamado "Receiver" ya que se encarga de recibir cualquier información que el cliente envía a través del flujo de comunicaciones formado en el enlace de comunicaciones entre cliente y servidor. Este thread puede crearse en los dos extremos (cliente y/o servidor) pero actúa de forma similar en cada extremo. Este thread puede recibir dos tipos de mensajes básicamente, mensajes relacionados con el robot y mensajes del protocolo de comunicaciones (mantenimiento y servicios de conexión). Si el mensaje recibido es referente al protocolo éste se procesara en el propio thread Receiver, mientras que si el mensaje es vinculado al control del robot no se procesa dentro de este thread, sino que se crea otro thread encargado de procesar el mensaje. Este tipo de thread se comentara más adelante.

Aunque todos los enlaces de control del robot usen los threads de tipo Receiver, el enlace de control del protocolo usa otro tipo de thread ("CtrlLinkReceiver") ya que el procesamiento de los mensajes que se enviaran a través de este enlace especial son distintos al resto. A través de thread se recibirán y procesaran mensajes relacionados con los puertos donde están corriendo los servidores del robot, los relacionados con la terminación de la conexión, etc., que serán procesados de forma distinta si el thread los recibe en el extremo del servidor o del cliente.

El thread encargado de procesar los mensajes relacionados con el control y notificaciones del robot es el llamado "ProccessMsg". El método que se sigue para procesar cada mensaje es el siguiente:

- En el extremo del servidor, cuando se recibe un mensaje se crea este tipo de thread que lo procese, y este thread atendiendo al identificador del mensaje invocara al método correspondiente del stub asociado al enlace, que a su vez llamara al método solicitado del servidor. Este tipo de mensajes que se envía en la dirección cliente a servidor son los relacionados con todas las órdenes del robot.

- En el extremo del cliente, cuando se recibe un mensaje se crea este tipo de thread que lo procese, y este thread atendiendo al identificador del mensaje invocara al método correspondiente del proxy asociado al enlace, que a su vez llamara al método solicitado

del cliente. Este tipo de mensajes que se envía en la dirección servidor a cliente son los relacionados con las actualizaciones sobre el estado del robot.

Básicamente el procesamiento de mensajes consiste en obtener un identificador del mensaje e invocar al método asociado a dicho identificador en el correspondiente stub o proxy, ya que son los que actúan de intermediarios entre cliente/servidor y el protocolo de comunicaciones.

9.5. Envío y recepción de mensajes.

A continuación se van a explicar con detalle los pasos a realizar du rante el envío de un mensaje por parte de uno de los clientes (servidor).

Cualquier método del cliente (servidor) que aparezca debido a la implementación de las interfaces como es MechanismCtrl (MechanismListener, MechanismUpdater…), es decir, cualquier método que suponga el envío de una petición al otro extremo de la comunicación, ya sea de cliente a servidor o viceversa, supondrá el envío de un conjunto de bits por el enlace, que el otro extremo debe ser capaz de entender.

Siempre que el cliente (servidor) llame a uno de esos métodos comentados anteriormente, por ejemplo de la interfaz MechanismCrtl (MechanismListener), y como ya anteriormente se comentó que el cliente (servidor) tiene una "referencia" al servidor (cliente), se invocará a ese mismo método pero de la referencia que posee- como ya se debe saber, esa referencia representa a un proxy (stub) obtenida gracias al método addControl-. Pues bien, dicho proxy (stub) contiene un objeto de tipo stationLink (rocLink) que representa el enlace de comunicaciones creado durante el proceso de establecimiento de conexión que permitirá la comunicación entre un proxy y un stub, es decir, entre un cliente y un servidor.

Todos los métodos que tiene el proxy (stub), que son los mismos que tiene el servidor (cliente) por implementar la misma interfaz que el servidor (cliente), tienen la misma implementación: "link.send( )", y con éste lo que se realiza es generalizar todos los posibles métodos que surjan para el envío de cualquier orden, de este modo, sea cual sea la orden a enviar al otro extremo, siempre se utilizará este método cambiando el parámetro. El parámetro a introducir corresponde a un objeto de tipo Message, en especial de: MechanismMsg, MissionMsg y ToolMsg, cuyos constructores estarán adaptados para cada una de las ordenes a generar, además de los posibles parámetros internos que pueden llevar si se desea enviar un

objeto de cualquier tipo adjuntado: int, float… Todos estos mensajes tendrán un formato en común, y es que todos tienen un objeto de tipo entero que representa el identificador de tipo de orden a enviar; además de esto y según el valor del identificador introducido se deberán adjuntar otros datos que se adjuntaran en el envío del mensaje.

El último paso para el envío de la orden de un extremo a otro es la implementación del método send de la clase stationLink (rocLink). En esta clase, encontramos un objeto de tipo Transmitter que será el hilo encargado de enviar la orden al otro extremo mediante el socket que posee. A este hilo, hay que introducirle en el constructor un flujo de datos de salida, en especial un objeto de tipo ObjectOutputStream y otro objeto de tipo ComposedObject que será el tipo de dato que va a circular por la red ya sea para ordenes internas del protocolo o para ordenes de movimiento de robot. De este modo, como al método send tan solo le pasábamos un objeto de tipo Message, tendremos que empaquetar ese mensaje dentro de uno de tipo ComposedObject; esto se consigue gracias al constructor de esta ultima clase y a un identificador que permita distinguir si la orden que se está enviando es de tipo interna o bien es una orden de manejo de robot.

Ya para finalizar el envío, solo quedará hacer una llamada al método run del hilo Transmitter para que mediante writeObject de la clase ObjectOutputStream (recuerde que se le introducía como parámetro un objeto de este tipo) se pueda enviar el objeto de tipo ComposedObject creado.

El proceso de recepción queda en gran parte ya aclarado y comentado porque el proceso es bastante similar al de transmisión pero al revés. El proceso de intercambio de métodos entre rocLink/stationLink – stub/proxy– servidor/cliente es igual al de la transmisión, pero no así el paso previo de recepción, así que será ese punto al que mayor énfasis se le de en la explicación. Además hay que destacar que este proceso es también adaptable tanto para la recepción del cliente como para el servidor.

Recordando la parte de transmisión, veíamos que existía un hilo que se encargaba de transmitir por la red objetos de tipo ComposedObject, por lo que eso es lo que se espera recibir. De este proceso también se encarga un hilo, con la diferencia de que para el transmisor se creaba un hilo para cada mensaje a transmitir mientras que en recepción solo existe un hilo encargado de recibir todos los mensajes, y será un objeto de tipo Receiver. La primera operación a realizar por el hilo tras recibir el objeto ComposedObject con el método readObject es leer el identificador de dicho objeto para diferenciar si lo recibido es una orden interna

(ComIDs.COM_MSG) cuya finalidad será la de cambiar el estado del enlace; o bien es un mensaje de movimiento de robot (ComIDs.EXTERNAL_MSG).

A la primera de las opciones no se le dará importancia en este punto ya que quedará perfectamente explicada en el punto que se encarga del protocolo interno utilizado.

Si el mensaje corresponde a la segunda opción, lo que se debe realizar es analizar el contenido de ese mensaje, pero no el del objeto ComposedObject, sino el del tipo Message que lleva adjuntado al objeto recibido. Para ello se creara un hilo de la clase CheckChanges al que se le debe pasar como parámetro el mensaje a analizar y un objeto de tipo RocLink o StationLink según corresponda al extremo cliente o al servidor. El método run de este hilo, analizará el identificador del mensaje y a partir de éste utilizará un tipo de stub o proxy (mechanism, mission o tool) según el mensaje esté especificado para uno u otro tipo para llamar al método que corresponda con ese identificador. Así por ejemplo si el identificador del mensaje es ToolIDs.ACTION_ON se llamará al método toolStub.actionOn. Además, con la llamada al método, será necesario introducirle algunos parámetros que corresponden a la vez con los objetos que venían adjuntos al mensaje que se recibió. Así, tras esta operación ya volvemos al traspaso de métodos entre proxy/stub y servidor/cliente pero en orden inverso a como se producía al transmitir.

Al hacer una llamada a un método de cualquiera de los stubs/proxies, como estos poseen una referencia del servidor o cliente asociado respectivamente, estos primeros volverán a pasarles el mismo método con los mismos parámetros. Esto no esta implementado siempre así, ya que para las operaciones de registro de clientes, hay que realizar otras muchas operaciones, pero eso ya se comentará en el siguiente apartado.

Finalmente, si no ha quedado claro el proceso de envio y recepción de mensajes sobre el protocolo, se puede observar el diagrama de sencuencia número 26 que se presenta en el apartado de modelado del sistema mediante UML, que indica de forma concisa todo el proceso comentado anteriomente.

9.6. Servicio de registro de los servidores.

9.6.1. Registro mediante notificación de eventos.

Los servidores del robot de tipo Mechanism y Tool disponen de un servicio en el que los clientes pueden recibir información sobre el estado del robot cuando éste varía, registrándose previamente en dichos servidores. Los métodos que permiten a un cliente registrarse en estos servidores son los siguientes:

void addMechanismListener(MechanismListener listener);
void addToolListener(ToolListener listener);

Estos métodos son los que contiene el servidor de tipo Mechanism, y nos basaremos únicamente en el proceso de registro en este tipo de servidor ya que para el de tipo Tool se usa el mismo proceso, aunque con distintos métodos. Con estos métodos el cliente ya no tiene que estar permanentemente preguntándole a los servidores el estado del robot, ya que el propio servidor notificará al cliente el cambio de estado del robot cada vez que se modifique el mismo. De este modo, los clientes son "escuchadores" y delegan la tarea de comprobar el estado del robot a los servidores.

El proceso que se lleva a cabo para registrar un cliente es el siguiente:

1. Un cliente llama a un método cualquiera de los dos anteriores, y le pasa como parámetro una referencia a si mismo (this) ya que dicho cliente debe implementar las interfaces MechasnimListener y/o ToolListener para poder registrarse en el servidor.

2. Dicha invocación al método del servidor llega realmente al Proxy asociado y este lo convierte en un mensaje que se envía a través del enlace de comunicaciones; este mensaje contiene la referencia al cliente que se introdujo como parámetro.

3. El mensaje llega al extremo del servidor y se procesa, convirtiéndose por ejemplo en una llamada al método *addMechanismListener(MechanismListener mechanismListener)* del stub correspondiente, al que se le pasara como parámetro la referencia al cliente que contenía el mensaje. Como el stub es el representante del cliente en el servidor, pues el que realmente se registra en el servidor es el stub (que también implementa las interfaces Listener), y los clientes reales que pueden estar distribuidos en distintas maquinas se registran en el stub. De esta forma que cada vez que el robot cambie de

estado el servidor notificara al stub (para él el único cliente registrado), y el stub notificará a los clientes que se hayan registrado realmente.

Hay que tener en cuenta que el stub solo debe registrarse una vez en el servidor, que será cuando se registre el primer cliente. Tanto el stub como el servidor almacenan a sus escuchadores en una lista que esta implementada por la clase "RegisteredClients", que se comentara mas adelante.

4. Los servidores que permiten el registro de clientes deben crear cada uno un objeto de tipo "EventControl" que se encargue de avisar al servidor que en el robot se ha modificado alguna variable de estado. Para conseguir que le pueda notificar esos cambios, al crear este objeto el servidor le pasa una referencia a si mismo.

El proceso que se sigue para notificar a los clientes que el robot ha cambiado de estado es el siguiente:

1. El primer paso es simular el cambio de estado del robot, que se ha resuelto implementado una interfaz gráfica de usuario para cada tipo de servidor que permite registrarse de este modo (MechanismServerGUI y ToolServerGUI). Estas interfaces contienen varios botones que simulan el cambio de estado del robot (modificando variables internas, etc.).

2. Cuando se simula un cambio de variable de estado de una parte del robot en concreto, el objeto "EventControl" asociado al servidor que controla esa parte del robot que se ha alterado notifica a dicho servidor que ha de actualizar dicha variable de estado a los clientes registrados. Los métodos existentes permiten notificar el cambio del estado del robot y las alarmas.

3. Es el propio objeto EventControl el que recorre la lista de clientes registrados en el servidor y notificándoles el cambio. En este caso solo se le notificara al stub ya que es el único cliente registrado para el servidor. El proceso que se realiza en cada método del objeto EventControl es el siguiente:

public void updateMechanismStatus(Object status)

{

- Comprueba si hay alguien registrado, si no hay nadie termina el método.

- Recorre la lista de registrados (Listener).
- Para cada escuchador "listener" registrado invoca al método:

mechanismServer.updateMechanismStatus(listener,AppIDs.EVENT,status);

- Invoca al método del servidor, indicándole a dicho servidor que el propio objeto EventControl esta recorriendo la lista de clientes registrados y que el cliente al que apunta la referencia "listener" esta registrado y debe notificarle el cambio de estado. El resto de parámetros indica que el tipo de registro solicitado por el cliente es mediante eventos (cambio estado) y el nuevo estado del robot esta representado por el objeto "status".

}

- El método del servidor que ejecuta dicho objeto EventControl es de la forma:

```
public void updateMechanismStatus(MechanismListener listener,int
                                                    regMode,Object status)
{
        listener.updateMechanismStatus(regMode,status);
```

- El servidor únicamente invoca al método de actualización del escuchador que le paso como parámetro el objeto EventControl asociado. Este escuchador es realmente el stub, ya que es el único cliente registrado en el servidor. Se puede comprobar que para el servidor el cliente esta en su misma maquina, y puede acceder al mediante llamadas locales.

}

4. Una vez que el servidor notifica al stub el cambio de estado del robot, éste ya se encarga de distribuir a los clientes reales dichas notificaciones. Cada stub recorre la lista de registrados en los propios métodos de la siguiente forma (se explica solo la parte correspondiente al registro mediante eventos, el resto en el siguiente apartado):

```
public void updateMechanismStatus(int registrationMode,Object mechStatus)
    {
```

```
if (registrationMode == AppIDs.EVENT)
        {
```

- Comprueba si hay alguien registrado, si no hay nadie termina el método.

```
        }
```

- Recorre la lista de registrados (Listener).
- Para cada escuchador "listener" registrado le envía un mensaje asociado al método de actualización de estado, este mensaje se envía a través del identificador del enlace asociado al cliente que se ha registrado.

```
    MechanismMsg m = new MechanismMsg
        (MechanismIDs.UPDATE_MECHANISM_STATUS,listener,mechStatus
        );

    link.send(m,listener.getLinkID());
}
```

5. El mensaje que el stub envía a través del enlace es procesado al llegar al extremo del cliente, y se convierte en una llamada a un método del proxy correspondiente. Este proxy utilizara la referencia del cliente registrado que incluía el mensaje para notificarle de forma local el nuevo estado del robot. Ejemplo:

```
public void updateMechanismStatus(int regMode,MechanismListener
                                        listener,Object mechStatus)
{
        listener.updateMechanismStatus(registrationMode, mechStatus);
}
```

6. De este modo el cliente recibe la notificación del nuevo estado del robot, sin necesidad de estar el preguntando por el estado del mismo.

Este proceso se sigue en todos lo métodos de actualización de estado y alarmas de cada servidor (Mechanism y Tool).

Todo el proceso de registro de clientes mediante eventos comentado anteriormente se presenta en los diagramas de sencuencia e interacción número 27,30 y 31 que se presentan en el apartado de modelado del sistema mediante UML.

9.6.2. Registro mediante notificación periódica.

Además del servicio de notificación de eventos en el estado del robot, los servidores del robot de tipo Mechanism y Tool disponen de otro servicio en el que los clientes pueden recibir información sobre el estado del robot cada cierto periodo de tiempo fijado por ellos, registrándose previamente en dichos servidores. Los métodos que permiten a un cliente registrarse en estos servidores de este modo son los siguientes:

void addMechanismListener(MechanismListener listener, int period)
void addToolListener(MechanismListener listener, int period)

Estos métodos son los que contiene el servidor de tipo Mechanism, y nos basaremos únicamente en el proceso de registro en este tipo de servidor ya que para el de tipo Tool se usa el mismo proceso, aunque con distintos métodos. El proceso que se lleva a cabo para registrar un cliente es el siguiente:

1. Un cliente llama a un método cualquiera de los dos anteriores, y le pasa como parámetro una referencia a si mismo (this) ya que dicho cliente debe implementar las interfaces MechasnimListener y/o ToolListener para poder registrarse en el servidor, y además le introduce un entero que es el periodo de tiempo en milisegundos con el que el servidor notificará el estado del robot.

   Antes de insertar como parámetro el periodo de tiempo, éste entero se debe redondear a un numero que sea múltiplo del periodo mínimo fijado en la aplicación para poder implementar el contador en el servidor (se explicará más adelante), este redondeo lo realiza un método de la clase "RegistrationPeriod" que sigue la siguiente regla de redondeo:
   - Si periodo < periodo mínimo, periodo redondeado = periodo mínimo.
   - Si periodo > periodo máximo, periodo redondeado = periodo máximo.
   - Si periodo esta comprendido entre el periodo máximo y mínimo, periodo = entero superior más cercano que sea múltiplo del periodo mínimo.

2. Dicha invocación al método del servidor llega realmente al Proxy asociado y este lo convierte en un mensaje que se envía a través del enlace de comunicaciones; este mensaje contiene la referencia al cliente y el periodo que se introdujo como parámetro.

3. El mensaje llega al extremo del servidor y se procesa, convirtiéndose por ejemplo en una llamada al método *addMechanismListener(MechanismListener mechanismListener, int period)* del stub correspondiente, al que se le pasara como parámetro la referencia al cliente y el periodo que contenía el mensaje. Al igual que para el registro mediante eventos en el estado del robot, quien realmente se registra en el servidor es el stub (que también implementa las interfaces Listener que implementa el cliente), y los clientes reales que pueden estar distribuidos en distintas maquinas se registran en el stub. A partir de este punto se complica la operación de controlar el periodo, que se realiza del siguiente modo:

   Cuando se invoca el método del stub *addMechanismListener(MechanismListener listener, int period)* al procesarse el correspondiente mensaje recibido por el enlace, el escuchador que desea recibir la notificación representado por la referencia "listener" se almacena junto al periodo de registro en la lista de registrados que contiene el stub. A continuación el stub debe registrarse en el servidor, pero solo lo hace cuando se registra el primer cliente. Sin embargo, una característica importante es que el stub se registra en el servidor con el periodo mínimo fijado por la aplicación, de forma que cada periodo mínimo el servidor notifique al stub el estado del robot, y sea el stub el que considere el tiempo de registro de los clientes registrados y les notificará el estado del robot cuando les corresponda. Se consigue de esta forma que solo exista un reloj para cada enlace (el que crea cada servidor).

4. Cuando el stub se registra en el servidor lo hace con el periodo mínimo, y el servidor se encarga de registrar al stub en su lista de clientes registrados con dicho periodo. Seguidamente el servidor crea el objeto de tipo "TimerControl" que controla el periodo solamente si es la primera vez que se registra un cliente, ya que el reloj solo se debe crear una vez, y debe ser con el periodo mínimo. Este objeto cada periodo mínimo notificará al servidor que debe actualizar el estado del robot en los clientes registrados en su lista, pero el problema es que cada cliente se registra con su periodo y pueden ser distintos entre si. Para solventar este problema se redondeó el periodo en el cliente antes de enviarse a través del enlace, de forma que el periodo de cada cliente en el extremo del servidor sea múltiplo del periodo mínimo.

Con esto se consigue que cada cliente contenga en la lista una "cuenta atrás" fijada por el entero resultante entre la división del periodo de registro y el periodo mínimo, esta cuenta atrás se va decrementando por cada periodo del reloj (periodo mínimo), y cuando esta llegue a "1" ha llegado el momento de actualizar el estado de dicho cliente con esa cuenta atrás. Con este ejemplo se deduce fácilmente el algoritmo:

- Periodo mínimo fijado por la aplicación de 100 milisegundos.
- Un cliente quiere registrarse con periodo 1995 milisegundos, pero antes de registrarlo la aplicación redondea automáticamente el periodo a 2000 mseg.
- El servidor registra al cliente con una cuenta atrás de 2000/100 = 20, de forma que cada 100 mseg. se va decrementando (20, 19, 18, 17,...). Cuando esta cuenta atrás llegue a 1 el servidor notificará al cliente el nuevo estado del robot y reiniciará la cuenta atrás de nuevo a 20 para la siguiente notificación. De esta forma el servidor solo tiene que crear un reloj que es válido para todos los clientes, ya que cada cliente esta asociado a una cuenta atrás que define el periodo con el que realmente se registró.

Como se explico en el registro mediante eventos, en el servidor únicamente se registrara su stub asociado que se registrará siempre con el periodo mínimo, por lo que en la lista de registrados del servidor solo se dispondrá de un solo escuchador cuya cuenta atrás se iniciará con el valor a "1", es decir, que cada 100 miliseg. (periodo mínimo) el servidor notificará al stub el estado del robot. El stub contiene otra lista de registrados donde se almacenan los clientes reales, que dispondrán cada uno de su cuenta atrás asociada obtenida mediante el algoritmo explicado anteriormente. De este modo, como el stub se registró con el periodo mínimo en el servidor, cada 100 mseg. el stub decrementara la cuenta atrás de sus clientes registrados, cuando la de un cliente llegue a "1" le notificara el estado del robot, seguidamente reiniciará la cuenta atrás para que reciban la siguiente notificación cuando expire de nuevo el periodo.

Es importante destacar que si se usa la aplicación de forma centralizada (sin usar ningún protocolo), ya no se usan stubs ni proxies, por lo que los clientes se registran directamente en el servidor, con el mismo procedimiento que se seguía cuando se registraba el stub.

Ahora que se conoce la metodología usada para registrar un cliente, el proceso que se sigue para notificar a los clientes el estado del robot es el siguiente:

1. Cada servidor crea un objeto de tipo TimerControl y le pasa en su constructor una referencia que apunta a si mismo, de forma que este objeto pueda notificarle a su servidor asociado cada vez que expira el periodo establecido también en el constructor, que será siempre de 100 mseg. (periodo mínimo).

timerControl = new TimerControl(this,AppIDs.MIN_PERIOD);

Esta clase contiene un objeto de la clase javax.swing.Timer de la API de Java, cuyo constructor es de la forma:

timer = new Timer(period, this);

Este objeto envía un evento cada periodo indicado como primer parámetro (en milisegundos) a un objeto que implemente el método actionPerformed(ActionEvent evt) de la clase java.awt.event.ActionListener de Java. En este caso la clase que implementa este método es la propia clase TimerControl donde se crea el objeto Timer (this), ya simplemente basta que esta clase implemente la clase ActionListener y sobrescriba el método actionPerformed. Este método recorrerá la lista de registrados del servidor y les notificará a cada uno el estado del robot cuando les corresponda, es decir, atendiendo al valor de la cuenta atrás de cada uno de ellos. Sin embargo, para el caso que se estudia en el que el único cliente registrado en el servidor es el stub y con periodo mínimo la cuenta atrás siempre estará a "1", por lo que cada periodo mínimo se notifica al stub el estado del robot. En el método actionPerformed de la clase TimerControl se realizan las siguientes operaciones (cada 100 mseg.):

- Recorro la lista de clientes registrados (únicamente estará registrado el stub).
- Para cada cliente:
  - Si su cuenta atrás es "1" le notifico el estado (y alarmas) del robot.
  - Si su cuenta es todavía mayor que "1" la decremento.

La forma de notificar al cliente (stub) el estado del robot es mediante estos métodos:
mechanismServer.updateMechanismStatus(mechanismListener, AppIDs.TIME,status);
mechanismServer.updateMechanismAlarms(mechanismListener, AppIDs.TIME,alarms);

2. Lo que se hace es enviarle al servidor la referencia del cliente al que tiene que notificar el estado, que en el caso que estamos estudiando únicamente será la referencia al stub. Por lo que el servidor únicamente debe usar esa referencia para notificarle el estado al stub (listener):

```
public void updateMechanismStatus(MechanismListener listener,int regMode,Object
mechStatus)
{
        listener.updateMechanismStatus(registrationMode,mechStatus);
}
```

Nota: También se envían el estado de las alarmas, pero solo se explicará como se notifica el estado el robot ya que son procedimientos idénticos (solo cambian los métodos).

3. El stub ahora recibe el estado del robot y debe redirigirlo a los clientes registrados. El stub además debe controlar el valor de la cuenta atrás de dichos clientes al igual que se hacia en el servidor:

```
public void updateMechanismStatus(int registrationMode,Object mechStatus)
{
        if (registrationMode == AppIDs.TIME)
        {
```

- Recorro la lista de clientes registrados.
- Para cada cliente:
  - Si su cuenta atrás es "1" le notifico el estado (y alarmas) del robot.
  - Si su cuenta es todavía mayor que "1" la decremento.

```
        }
        else if (registrationMode == AppIDs.EVENT)
        {…se explicó anteriormente…}
}
```

La forma de notificar a los clientes el estado del robot es mediante el envío de mensajes a través del enlace asociado a cada cliente. El mensaje que se envía contiene el identificador asociado al método de actualización del estado del robot, una referencia al cliente que se registró, y por último el estado del robot a actualizar.

```
        message = new MechanismMsg ( MechanismIDs.UPDATE_MECHANISM_STATUS,
                                        obj.getMechanismListener(),mechStatus);

        link.send(message,obj.getMechanismListener().getLinkID());
```

4. El mensaje que se envía a través del enlace es procesado al llegar al extremo del cliente, y se convierte en una llamada a un método del proxy correspondiente. Este proxy utilizara la referencia del cliente registrado que incluía el mensaje para notificarle de forma local el nuevo estado del robot. Ejemplo:

```
public void updateMechanismStatus(int regMode,MechanismListener listener,Object
mechStatus)
{
        listener.updateMechanismStatus(registrationMode, mechStatus);
}
```

5. De este modo el cliente recibe la notificación del nuevo estado del robot, sin necesidad de estar el preguntando por el estado del mismo.

Este proceso se sigue en todos lo métodos de actualización de estado y alarmas de cada servidor (Mechanism y Tool).

9.7. Lista de clientes registrados.

La listas donde se almacenan los clientes registrados, tanto los registrados mediante eventos como mediante un periodo de tiempo, se encapsulan en la clase "RegisteredClients". Esta clase contiene los vectores donde se almacenarán los clientes del servicio de registro y los métodos necesarios para su inserción y extracción.

- Para el servicio de registro mediante eventos se dispone de los métodos:

```
public void addTimeListener(MechanismListener client,int period) {...}
public void addTimeListener(ToolListener client,int period) {...}
```

Insertan en la lista de registrados de tipo Mechanism o Tool en cada caso un nuevo cliente que implemente las interfaces MechanismListener o ToolListener en cada caso. Antes de insertar un cliente en su lista correspondiente se comprueba si el cliente estaba ya registrado, de forma que no se pueda registrar mas veces ya que recibiría varias veces la misma notificación de estado, y en consecuencia ocupando un espacio innecesario en el canal de comunicaciones.

- Para el servicio de registro mediante un periodo de tiempo se dispone de los métodos:

public void addTimeListener(MechanismListener client,int period) {...}
public void addTimeListener(ToolListener client,int period) {...}

Insertan en la lista de registrados de tipo Mechanism o Tool en cada caso nuevo un cliente que implemente las interfaces MechanismListener o ToolListener en cada caso. Antes de insertar un cliente en su lista correspondiente se comprueba si el cliente estaba ya registrado, de forma que si no esta registrado se inserta en la lista, y si no esta registrado pueden suceder dos casos:

- Si esta registrado con el mismo periodo $\rightarrow$ no hace nada.
- Si esta registrado con periodo distinto $\rightarrow$ actualiza el periodo.

De esta forma si un mismo cliente se registra con varios periodos, solo permanecerá el último con el que se registró. Esto permite al cliente modificar el periodo de actualización con el que el servidor le notifica el estado del robot, permitiendo la corrección de posibles registros con un periodo indeseado.

Por último también es interesante destacar los métodos que eliminan un cliente del registro, ya sea mediante eventos o periodo de tiempo. Estos métodos son:

public void removeMechanismListener(MechanismListener listener, int regMode) {…}
public void removeToolListener(ToolListener listener, int regMode) {…}

Estos métodos eliminan el cliente al que apunta la referencia listener de la lista de registrados de tipo Mechanism o Tool en cada caso. El entero que se inserta como segundo parámetro indica si se quiere eliminar del registro de eventos o de periodo de tiempo (véase clase AppIDs).

Para terminar con todo el proceso de registro en la aplicación cabe destacar que para almacenar en las listas (vectores) los escuchadores registrados mediante eventos se insertaban directamente las referencias a dichos escuchadores (MechanismListener y ToolListener). Sin embargo, para almacenar los clientes que se registran con un periodo de tiempo en las listas necesario almacenar el valor del periodo además de la referencia al cliente; por ello es necesario crear un objeto que encapsule estos dos valores incluyendo el valor de la cuenta atrás para cada cliente. Estos objetos son instancias de la clase TimeRegisteredObject, y a través de ellos se accederá al cliente registrado, su periodo y su cuenta atrás para la actualización.

Todo el proceso de registro de clientes mediante periódo de tiempo comentado anteriormente se presenta en los diagramas de sencuencia número 28 y 29 que se presentan en el apartado de modelado del sistema mediante UML.

9.8. Comunicación asíncrona vs. síncrona.

La funcionalidad que inicialmente presentaban los servidores del robot definía una serie de métodos que permitían a los clientes obtener el estado del robot y otros parámetros del mismo, estos métodos estaban definidos en las interfaces que los servidores deberían implementar (MechasnimCtrl, ToolCtrl y MissionCtrl). Algunos de estos métodos eran:

public Object getMechanismStatus();
public int[] getMechanismAlarms();
public Object getToolStatus();
...

En definitiva, los métodos a los que nos referimos son a los métodos que devuelven algún valor u objeto, ya que el resto de métodos definidos en las interfaces señaladas anteriormente y que no devuelvan ningún valor (void) no se tendrán en cuenta en este punto ya que no presentan la problemática que a continuación se explica.

Estos métodos tienen el problema de que el hecho de que devuelvan un valor implica que el cliente tiene que esperar la respuesta que el método defina para poder continuar, es decir, la comunicación es bloqueante ya que el cliente se bloquea hasta que no reciba la respuesta del servidor. Este tiempo de espera puede ser grande si la aplicación se ejecuta bajo redes que presentan cierto retardo (Internet), por lo que se optó por redefinir los métodos que presentaban esta problemática de forma que la comunicación sea completamente asíncrona, es decir, que el cliente no tenga que esperar la respuesta del servidor para poder continuar con otras acciones.

Este problema se solventó definiendo una serie de métodos en el cliente, de forma que cada método del servidor que devuelva algún valor esté asociado con uno de estos métodos nuevos en el cliente. La idea es que cuando el cliente llame a un método del servidor para obtener un valor, el cliente haga la petición y termine el método sin obtener respuesta, y el servidor cuando pueda le enviará la respuesta invocando al método del cliente asociado a la petición que realizó, de forma que la respuesta se inserte como parámetro en dicho método del cliente. Con este proceso la comunicación es asíncrona ya que el cliente no se bloquea

esperando la respuesta, sino que el envía la petición y se olvida, pudiendo seguir con su trabajo. El servidor le enviará la respuesta, que puede retrasarse y el cliente no estara bloqueado esperándola, sino que será procesada por los threads receptores como una respuesta más del servidor.

El primer paso para implementar esta comunicación fue definir tres nuevas interfaces (MechanismUpdater, ToolUpdater y MissionUpdater) que precisen los métodos que el servidor invocara para pasar la respuesta a la petición que el cliente realizó. Como se comentó anteriormente estas interfaces definen un método de respuesta para cada método de petición al servidor. Estas tres interfaces las debe implementar el cliente para que el servidor les pueda devolver la respuesta invocando a dichos métodos.

El segundo paso fue redefinir los métodos de las interfaces MechanismCtrl, ToolCtrl y MissionCtrl que devuelvan algún valor u objeto de forma que no devuelvan nada, es decir, sustituyendo el tipo que devuelven por el tipo "void". Además de ello cada método va a tener de un parámetro obligatorio que será una referencia a un objeto de tipo MechanismUpdater, ToolUpdater o MissionUpdater, que representará al cliente que quiere recibir la respuesta.

Estos pasos se detallan con el siguiente ejemplo:

Si antes disponíamos en la interfaz MechanismCtrl de los siguientes métodos:

```
public Object getMechanismStatus();
public int[] getMechanismAlarms();
```

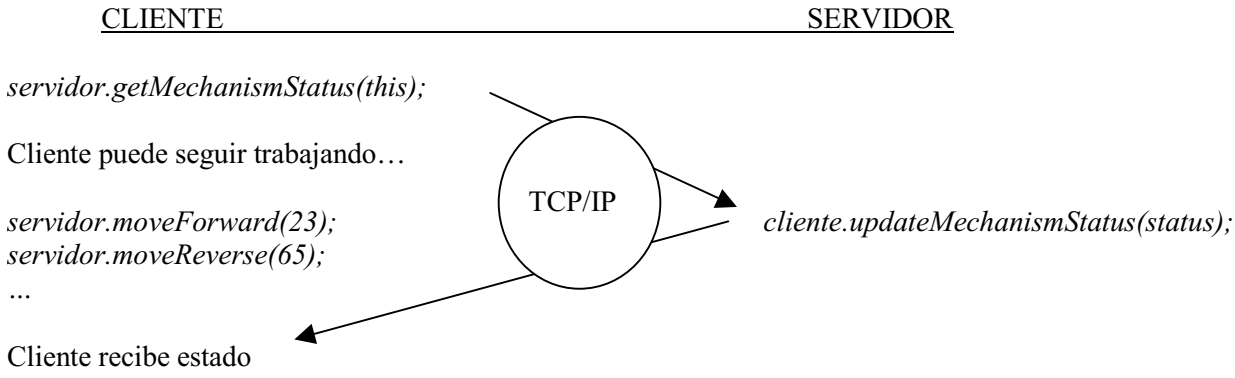
1. Se define una nueva interfaz ("MechanismUpdater") que implemente los métodos de respuesta:

```
public void updateMechanismStatus(Object mechStatus);
public void updateMechanismAlarms(int[] activeAlarms);
```

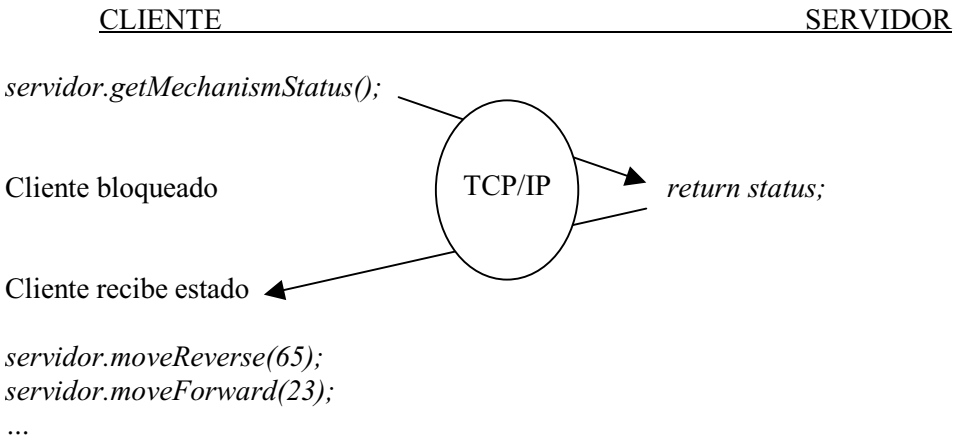
2. Se redefinen los métodos anteriores de la interfaz MechanismCtrl de la siguiente forma:

```
public void getMechanismStatus(MechanismUpdater updater);
public void getMechanismAlarms(MechanismUpdater updater);
```

De esta forma, un posible ejemplo de invocación de métodos sería la siguiente:



Mientras que de forma síncrona el mismo ejemplo sería la siguiente:



Aunque se ha explicado de forma generalizada, todas las invocaciones de métodos que impliquen cliente y servidor son transformadas en mensajes que son enviados a través de los enlaces de comunicaciones. Los proxies y stubs procesaran dichos mensajes en llamadas a métodos y viceversa.

Es necesario insertar en los métodos anteriores como parámetro una referencia al cliente mediante MechanismUpdater, ToolUpdater o MissionUpdater ya que será necesaria para cuando se devuelva la respuesta, ya que al recibir el proxy la respuesta del servidor necesita tener la referencia del cliente que hizo la petición para devolverle dicha respuesta localmente.

Para terminar, se puede comprobar que el proceso de hacer una petición con algún método "get" es similar al proceso de registrarse en el servidor ya que la metodología usada para enviarle un dato al cliente es el mismo en los dos casos. La única diferencia que existe es que en el caso del registro en el servidor el cliente no se preocupa de estar haciendo peticiones de información de estado, ya que registrándose en el servidor éste se encarga automáticamente de enviarle la información.

9.9. Cierre de las comunicaciones.

9.9.1. Cierre de un cliente.

Una vez entendido el traspaso de métodos entre cliente-stub-enlace, es fácil de comprender el mecanismo implementado para cerrar un cliente. Como es de suponer la clase cliente dispone de un método llamado "close" que es el encargado de cerrar dicho cliente. Esto supone eliminarse del registro del servidor, y es que los enlaces que se crearon para dar servicio a los clientes de esa maquina, no se cerrarán hasta que no se mande la orden de cerrar la aplicación (shutdownApp), para posibilitar de este modo la posible creación de otros clientes bajo ese mismo enlace; pero ese método ya no pertenece a la clase Client, sino a Manager, así que se comentará mas adelante.

Atendiendo a la implementación, y como ya se descrito anteriormente, al hacer una llamada al método close, se llamaran a los removeMechanismListener() y removeToolListener() pasando como parámetro tanto el identificador de registro por eventos (AppIDs.EVENT) como por espacio temporal (AppIDs.TIME) para ambos métodos, además de una referencia del propio cliente. Esos métodos, que están implementados en la misma clase cliente, se irán pasando tal y como se detalló en el punto de "envío y recepción de mensajes" hasta llegar al stub. Y es que como ya se comentó para el registro de los clientes, debido a que el servidor no debe tener información de los clientes, son los stubs los encargados de almacenar a los clientes registrados.

La implementación de estos métodos en los stubs es muy simple ya que se compone de una llamada al método con el mismo nombre y con los mismos parámetros pero en este caso de la clase "RegisteredClients", que se encarga de todo el proceso de registro de los clientes en sus distintas versiones. Por ultimo, en el método de esta clase se va a hacer un recorrido por los vectores destinados al registro y se eliminara la entrada que coincida con los parámetros introducidos. Pero hay que tener en cuenta un detalle muy importante, y es que solo se eliminan los clientes pero en ningún caso se pierde el "contacto" con los proxies ya que no hay que

olvidar que aunque se cierre un cliente, puede que existan otros futuros clientes que usen esos enlaces.

El proceso de cerrado del cliente queda finalizado con solo eliminar a éste del registro, pero razonando podemos comprender que si no se cierra la interfaz con el usuario de dicho cliente, éste podría seguir mandando ordenes al servidor, sin distinguir este último si el cliente ha sido cerrado anteriormente o no.

9.9.2. Cierre de la aplicación.

Mediante el método shutdownApp() conseguimos dar la orden a todos los servidores para que eliminen todos los datos almacenados en sus registros y además cierren todos los enlaces (sockets) que permitían comunicarse con los clientes. Este método puede ser invocado por cualquiera de los clientes que este corriendo ya que la implementación de éste solo provoca el envío de un objeto de tipo CtrlComposedObject (tipo de objeto que se envía entre CtrlLinkStation y CtrlLinkRoc) con identificador ComIDs.END. Lo que realmente se está haciendo es enviar un mensaje, con un identificador especial (fin de aplicación), a través del enlace de control de comunicaciones. Este mensaje llegará al host donde estén corriendo los servidores y lo procesaran de forma adecuada, siguiendo el proceso que sigue a continuación.

El objeto (mensaje) será recogido por el hilo encargado de la recepción de dicho enlace de control: CtrlLinkReceiver. Éste, al analizar el identificador que acompaña al objeto y comprobar que corresponde a la orden de cierre de aplicación considerará que está trabajando en el extremo del servidor de control (CtrlLinkRoc) y ejecutara secuencialmente varios métodos para cerrar por completo la aplicación. Veamos paso a paso como se cierran todas las comunicaciones y registros del servidor, y el resto de clientes:

La primera acción a realizar por el servidor será la de limpiar todos sus registros de actualización de clientes del mismo modo que se hacía al cerrar un cliente. Para ello se llamará al método estático resetListeners() de la clase ServersContainer, que eliminará todos los clientes registrados en los servidores. Sin embargo, como realmente los clientes están almacenados en los stubs, ya que el objeto que está registrado en el servidor no es un cliente sino un stub (representante del cliente), pues basta con eliminar los clientes registrados en el stub. Lo que buscamos desregistrando todos los clientes es evitar que se envíen más mensajes a través de los enlaces de comunicaciones puesto que se van a cerrar a continuación, y los objetos que realmente convierten dichas actualizaciones de los clientes registrados son los objetos stub, es

por ello que basta con eliminar los clientes registrados del stub (no es necesario eliminarlos del servidor puesto que solo contendría al stub como "cliente" registrado). Resumiendo todo lo comentado anteriormente sobre el desregistro de todos los clientes, lo único que se debe hacer es evitar que el stub redirija las actualizaciones a posibles clientes mediante los enlaces de comunicaciones.

El siguiente paso a realizar es cerrar todos los hilos receptores que se ocupaban de recibir los mensajes de cada uno de los enlaces de control (CtrlLinkReceiver) de la aplicación, exceptuando el hilo que haya recibido el objeto con el identificador END, ya que será este el que se encargue de cerrar la aplicación. Esto es, si por ejemplo disponemos de 3 clientes distribuidos en 3 maquinas distintas, en el servidor se crearán 3 threads receptores sobre cada stream de comunicaciones asociado a cada cliente respectivamente; con esta arquitectura, si por el primer enlace de control (asociado al cliente 1) se recibe un mensaje de cierre de aplicación, el thread que recibe dicho mensaje debe cerrar el resto de threads receptores de control (clientes 2 y 3), pero sin cerrarse a si mismo, ya que éste thread (donde se ha recibido el mensaje de cierre) es el encargado de cerrar todas las comunicaciones. Todo este proceso se realiza con el método estático closeCtrlReceivers() de la clase CtrlLinkRoc.

A continuación se cerraran el resto de hilos receptores, pero en este caso, los hilos que actúen en los enlaces de comunicaciones (anteriormente se cerraron los receptores de los enlaces de control), ya que si se desea cerrar la aplicación no debemos seguir recibiendo ordenes de los clientes. Estos hilos se cierran con el método estático closeReceivers() de la clase ServersContainer, la cual se encargará de cerrar los threads receptores de los enlaces asociados al servidor de Mechanism, Tool y Misión.

El siguiente paso corresponde al cierre de los clientes que todavía permanecen activos con el servidor, ya que es imprescindible para el correcto cierre de las comunicaciones que se cierren antes los clientes que los propios servidores. Esto se consigue enviando un objeto, similar al recibido en el hilo receptor de control (objeto con identificador END), a cada uno de los enlaces de control que todavía permanecen abiertos. En el otro extremo de la comunicación (clientes), cuando por cada uno de los enlaces de control se reciba este objeto (también mediante un hilo de tipo CtrlLinkReceiver), se cerrarán todos los enlaces (StationLink) que dependan de esos enlaces de control con el método estático closeClients() de la clase ClientsContainer. Éste método se encarga de cerrar los flujos de entrada/salida y los sockets abiertos para la comunicación con el servidor, y todo esto para cada uno de los 3 enlaces (Mechanism,Tool y Mission) abiertos. Todo este procesamiento se recibe en todos los clientes que tengan establecida una conexión con el servidor (distribuidos en varios hosts). Por último y de forma

similar a como ocurrirá en el extremo servidor, se van a cerrar todos los flujos y sockets que permanezcan abiertos correspondientes al enlace de control; esto se consigue con el método releaseConnection() de la clase CtrlLinkStation.

A partir de ahora todos los posibles clientes conectados al servidor están cerrados correctamente, por lo que ahora nos disponemos a cerrar los servidores completamente. Volviendo al procesamiento en el extremo servidor, y tras haber dormido el hilo que procesa todo el cierre de las comunicaciones (hilo receptor que recibió el mensaje de fin de aplicación) durante cierta duración (~ milisegundos) para dar tiempo a que se realicen las acciones anteriores en los clientes, se van a cerrar el resto de los hilos que permanecen abiertos, es decir, tanto los posibles transmisores (Transmitter) que queden abiertos como los encargados de aceptar peticiones (Conector). Para esto se llamará al método closeThreads de la clase ServersContainer, la cual indicará a cada uno los enlaces rocLinks existentes que deben cerrar sus threads conectores y transmisores.

Por último y para dejar la aplicación totalmente cerrada, solo falta cerrar todos los flujos y sockets de la maquina servidor. Esto se consigue con la llamada al método closeAplication de la clase ServersContainer, que realiza dos operaciones: Primero se encarga de indicar a cada uno de los servidores (enlaces rocLinks) existentes que deben cerrar todos los flujos y sockets que permanezcan abiertos, y por último se encarga de indicar al servidor de control (CtrlLinkRoc) que debe cerrar todos sus threads activos y seguidamente los flujos entrada/salida y sockets.

En este momento, todas las comunicaciones se han cerrado ordenadamente y sin provocar ningún tipo de excepción, teniendo la garantía de que nuestros equipos permanecen estables y ningún puerto permanece abierto en el sistema.

Todo el proceso de cierre la aplicación comentado anteriormente se presenta en los diagramas de sencuencia número 22, 23 y 24 que se presentan en el apartado de modelado del sistema mediante UML.

10. Metodología.

Una vez realizada la implementación de la aplicación, se procedió a su comprobación y simulación. Para esto, y para facilitar el proceso se desarrollaron un conjunto de interfaces con el usuario (GUIs), que permitieran el envío de órdenes de un extremo a otro mediante un simple click.

Como la primera parte de la aplicación a implementar fue el protocolo de comunicaciones interno, para su puesta en marcha se realizó la primera interfaz con el usuario mediante la cual se permitía tanto la creación y cierre de enlaces como el envío y recepción de objetos por la red.

Una vez que se fueron implementando otros componentes de la aplicación y que se fueron añadiendo al protocolo, la interfaz con el usuario se tuvo que remodelar al estado actual para permitir de este modo el envío de mensajes predefinidos y la actualización de los parámetros de registro de clientes. Con este formato, la interfaz con el usuario del cliente esta formado por 3 menús desplegables donde aparecen todas la ordenes que se pueden mandar a cada uno de los servidores. Además, dicha interfaz estará formada por una caja de texto donde introducir el periodo al que se desea registrar y por un botón para cerrar el cliente.

La interfaz de usuario del servidor es muy sencilla y esto es debido a que en princi pio, éste no dispondrá de interfaz con el usuario. Así, la única razón de implementarla es la de simular un cambio en las variables que supuestamente describen al robot, para de este modo, poder comprobar el correcto funcionamiento del registro de clientes mediante eventos. Como el registro de los clientes puede ser tanto para el servidor de mecanismo como para el de herramienta, en el monitor de la maquina servidor (si es que existe) aparecerán dos interfaces de usuario que simularan el proceso de cambio de dichas variables.

Ya habiendo creado las interfaces con el usuario, es conveniente probar la aplicación en cualquiera de las situaciones posibles en las que puede tener utilidad la aplicación. La mas sencilla y la que en mas ocasiones se suele instalar consiste en una línea punto a punto entre dos maquinas, haciendo una de ellas de servidor y otra de cliente. Ampliando esta situación y disponiendo de más de dos máquinas en red, se consigue que el robot puede ser manejado mediante varios clientes situados cada uno de ellos en maquinas distintas.

Hay también una situación, que aunque elimina la propiedad de aplicación distribuida, puede resultar muy beneficiosa, ya que ésta se puede adaptar a una situación centralizada, es decir, que se encuentre tanto el cliente como el servidor en una única maquina. Esto supone unos pequeños cambios en el código, ya que no hay que olvidar que no habría que crear ningún socket ni ningún elemento de comunicación, es decir, se eliminaría el paquete del protocolo interno, el de los stubs y el de los proxies.

Debido al boom existente actualmente por la red Internet, también se ejecutó la aplicación sobre esta red, ya que disponiendo de una dirección IP de la Red para cada una de las maquinas, el proceso es similar a si se estuviese ejecutando la aplicación sobre una red local. Esta propiedad es de gran importancia, ya que al disponer actualmente casi todas las maquinas conexión a Internet, y provoca que pueda ser manejado el robot desde un gran número de maquinas sin importar su localización.

Un posible ejemplo de despliegue del sistema sería el disponer de un servidor que disponga de una dirección pública de Internet, y al que pueden acceder tanto clientes pertenecientes a la red local como a Internet. El diagrama de despliegue número 20 que se presenta en el apartado de modelado del sistema mediante UML es un ejemplo de una posible configuración de esta aplicación.

11. Manual de usuario de la aplicación.

11.1. Ejecución de la aplicación.

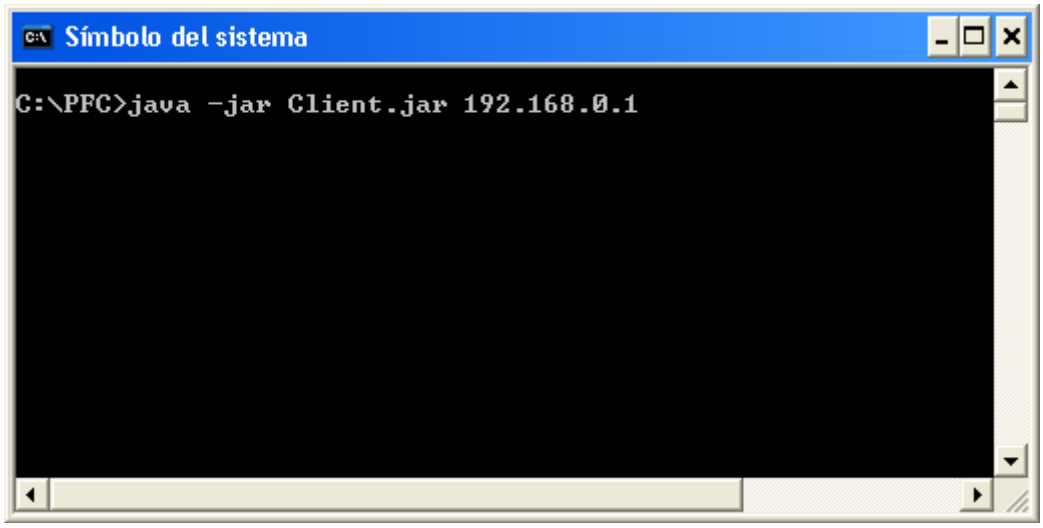
A pesar de que la aplicación es bastante sencilla de ejecutar, se ha elaborado esta pequeña referencia para el usuario que desee probar su ejecución sobre una red local, sobre Internet o incluso sobre un único PC.

Java proporciona principalmente 2 métodos para ejecutar una aplicación de este tipo, la primera de ellas es la más básica y consiste en utilizar el comando "java" (que proporciona la maquina virtual de Java) sobre las clases consideradas como "main", sin embargo, debido al gran número de clases que abarca la aplicación, se opto por almacenar todas las clases en un archivo JAR que permita su ejecución de forma más rápida y sencilla. El segundo método para ejecutar la aplicación se basa en los archivos JAR a los que nos referíamos anteriormente, y el proceso que se sigue es el siguiente:

Al igual que todas las aplicaciones basadas en la arquitectura clientes/servidor, el primer paso consiste en la ejecución del servidor. Para ello se usará el comando java pero con la opción "–jar" que indica que estamos ejecutando la aplicación a partir de un fichero JAR. El main del servidor no recoge ningun parámetro, por lo que para ejecutarlo simplemente escribimos "java –jar Server.jar" en la línea de comandos. El resultado debe ser similar al de la figura, donde podemos comprobar que se han creado los 3 servidores de control del robot y el servidor de control del protocolo.

Una vez esté corriendo el programa servidor se procede a ejecutar el cliente, que únicamente recoge un parámetro que indica la dirección del host donde se está ejecutando el servidor. Este parámetro se puede omitir pero el cliente tomara como dirección por defecto la IP 127.0.0.1 conocida como "localhost", que da a entender que el servidor esta corriendo en la misma máquina que el cliente. No obstante, si el servidor está ejecutándose en un host perteneciente a una red local a la que el cliente tenga acceso, el cliente debe especificar la dirección IP que tenga asignada el servidor en dicha red. El procedimiento sería el mismo en el caso de que el servidor esté ejecutándose en un host de Internet (se especificara la IP de dominio público).

La figura muestra el comando que se utilizaría para ejecutar el cliente suponiendo que el servidor este corriendo en el host con la dirección 192.168.0.1.



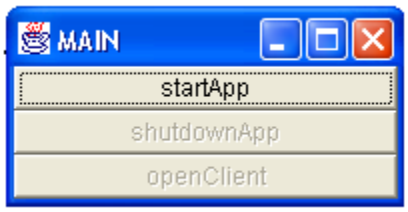
De este modo se pueden ejecutar otros clientes si se desea controlar el robot desde diferentes hosts, ya que el servidor puede soportar servir peticiones de forma concurrente.

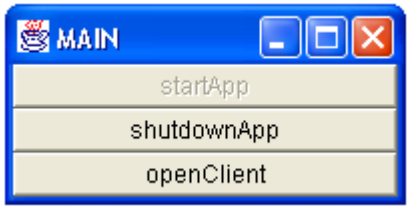
11.2. Extremo cliente.

Cuando se inicia la aplicación desde el extremo cliente, se nos muestra una ventana desde la cual se permiten realizar 3 acciones: "startApp", "shutdownApp" y "openClient". Con la primera de ellas se iniciará la aplicación con la consecuente creación de enlaces y de todos los elementos necesarios para una comunicación segura y fiable. Con el segundo botón conseguimos el cierre de la aplicación y con ella, el cierre también de todos los clientes y sus enlaces creados para su funcionamiento. Con el último de los botones lo que se consigue es crear un nuevo cliente para manejar el robot, y es que, aunque no tenga mucho sentido la

existencia de 2 clientes en una misma máquina, si es posible que se haya cerrado el cliente con el que se estaba trabajando, y más tarde se desee crear otro nuevo para seguir manejando la aplicación.

En la primera ejecución, las 2 ultimas acciones están deshabilitadas ya que no tiene ningún sentido crear un nuevo cliente o mas aun cerrar la aplicación si ésta todavía no se ha iniciado. Así análogamente, tras pulsar el botón de inicio de aplicación, será éste el que se desactive y los otros 2 se habilitarán.



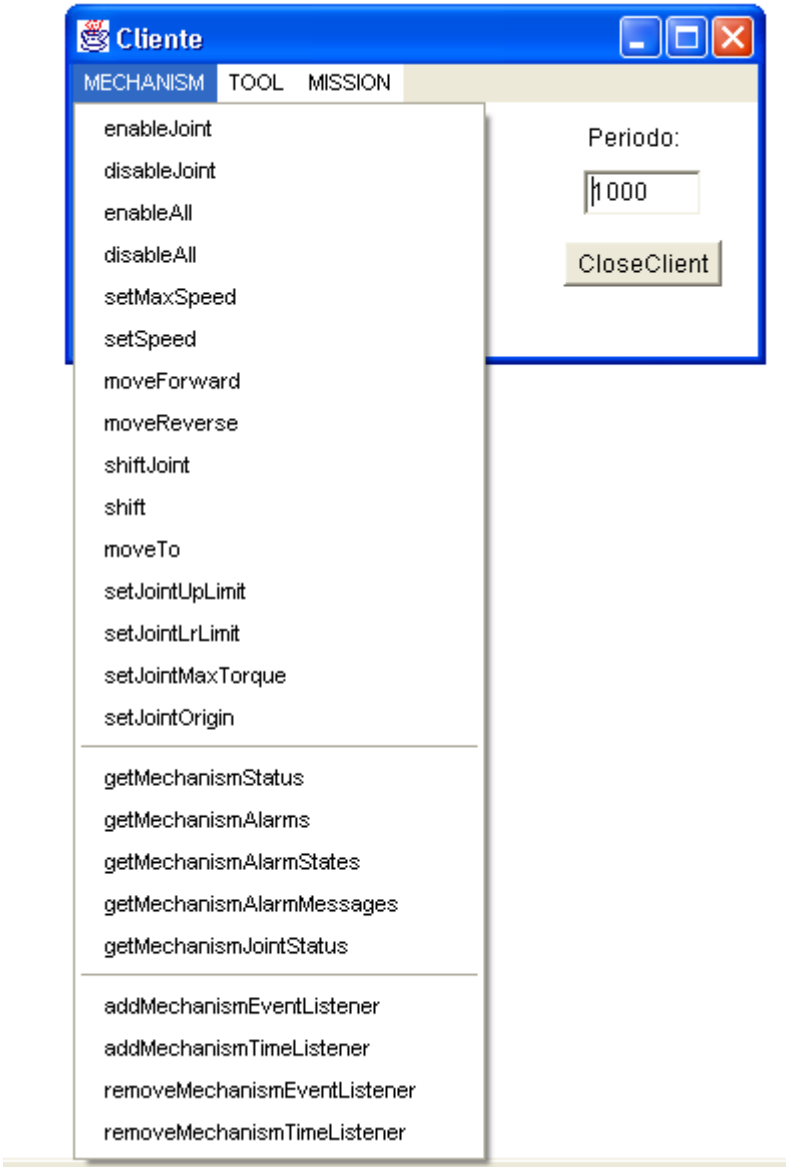


Una vez iniciada la aplicación, aparecerá una nueva ventana en la que podemos ver: un botón de cierre de cliente (“closeClient”), un textbox en el que introducir un valor entero (“periodo”) y una barra de menú con 3 elementos desplegables: “mechanism”, “misión” y ”tool”. Empezando por el botón “closeClient”, su acción es la opuesta a la del otro botón que aparecía en la primera ventana (“openClient”). Aunque parezca un poco extraño que dos botones cuya acción es complementaria aparezcan en 2 ventanas separadas, esto tiene su motivo y es que sino fuera así, la ventana principal tendría que tener un botón para cada uno de los clientes creados hasta el momento y eso resultaría poco eficiente.

Pues bien, una vez que se ha iniciado la aplicación y se ha creado la segunda ventana, ya podemos manejar el robot mandándole ordenes. Para ello, y como las ordenes son muchas se han dispuesto 3 botones desplegables en los que se ordenan las posibles acciones a realizar según cual sea su cometido: “mechanism”, “misión” y ”tool”.

Dentro de cada uno de estos distinguimos 3 grupos de ordenes: en el primer grupo se encuentran las ordenes tras las que no se espera una contestación del servidor, como son los casos de “enableJoint” o “shift”. En el segundo grupo aparecen los métodos para los que se espera una respuesta, que coinciden por se los de tipo “get” (el cliente desea conocer el valor de un parámetro o variable del robot); tras su invocación el servidor llamará a alguno de sus métodos para “enviar” el resultado de la petición. El ultimo grupo de métodos, representa a las ordenes que provocarán un cambio en el registro de clientes que dispone el servidor, como son los métodos: “addMechanismEventListener” y “removeMechanismTimeListener”, que permiten

que un cliente se añada o elimine del registro para obtener periódicamente o tras un evento algún parámetro del robot. Al pulsar uno de los métodos "add" de tipo "TimeListener", se producirá una lectura del checkbox de la parte derecha de la interfaz y de ahí se obtendrá el valor del periodo tras el cual se recibirá el valor del parámetro al que se haya suscrito. Por defecto, se toma como parámetro 1000 milisegundos , para evitar de esta forma que el usuario no introduzca ningún valor. Si se desea conocer más sobre el funcionamiento del registro deberá dirigirse a la sección donde se explica esto con un mayor detalle.



Hay que destacar, aunque ya se ha advertido varias veces, que el componente "misión" no tiene registro de clientes, y por lo tanto no dispondrá de métodos de este grupo.

Para cerrar la aplicación, bastará con pulsar sobre el botón de la ventana principal "shutdownApp", tras lo cual se cerrarán todos los clientes, estén o no en nuestra misma máquina, es por esto por lo que hay que limitar el uso de este botón, ya que tras esto, se cerrará incluso el servidor, impidiendo que ningún otro cliente pueda volver a ponerlo en marcha remotamente. Así, se aconseja utilizar siempre el cierre de clientes, dejando el servidor a la escucha de nuevos clientes; y es que, no hay que olvidar que si se realiza esta aplicación para poder controlar el robot de forma remota, carece de sentido que cada vez que se desee utilizar, tenga que haber un operario que se dirija al servidor (que en principio se encontrará muy próximo al robot) para reiniciarlo.

11.3. Extremo servidor.

La existencia de una interfaz gráfica en el servidor carece de sentido alguno ya que en principio éste carecerá de monitor. Esta interfaz se creó para permitir la simulación por parte del programador de una variación en los parámetros o variables del robot. Asi, de este modo, se creo una simple ventana en la que destacan 2 botones y un textbox. Cada vez que haga clic sobre uno de esos botones, se simula que se ha variado o el estado o las alarmas de uno de los componentes de este robot. Así, dispondremos de 2 ventanas, una para el componente Mechanism y otra para Tool; y cada una de ellas con los 2 componentes: estado y alarmas. La caja de texto de la parte inferior supondría el nuevo valor que ha tomado una variable determinada, que a su vez, se tendría que enviar al cliente para que el usuario de la aplicación pudiera actualizar la suya.

12. Documentación del código.

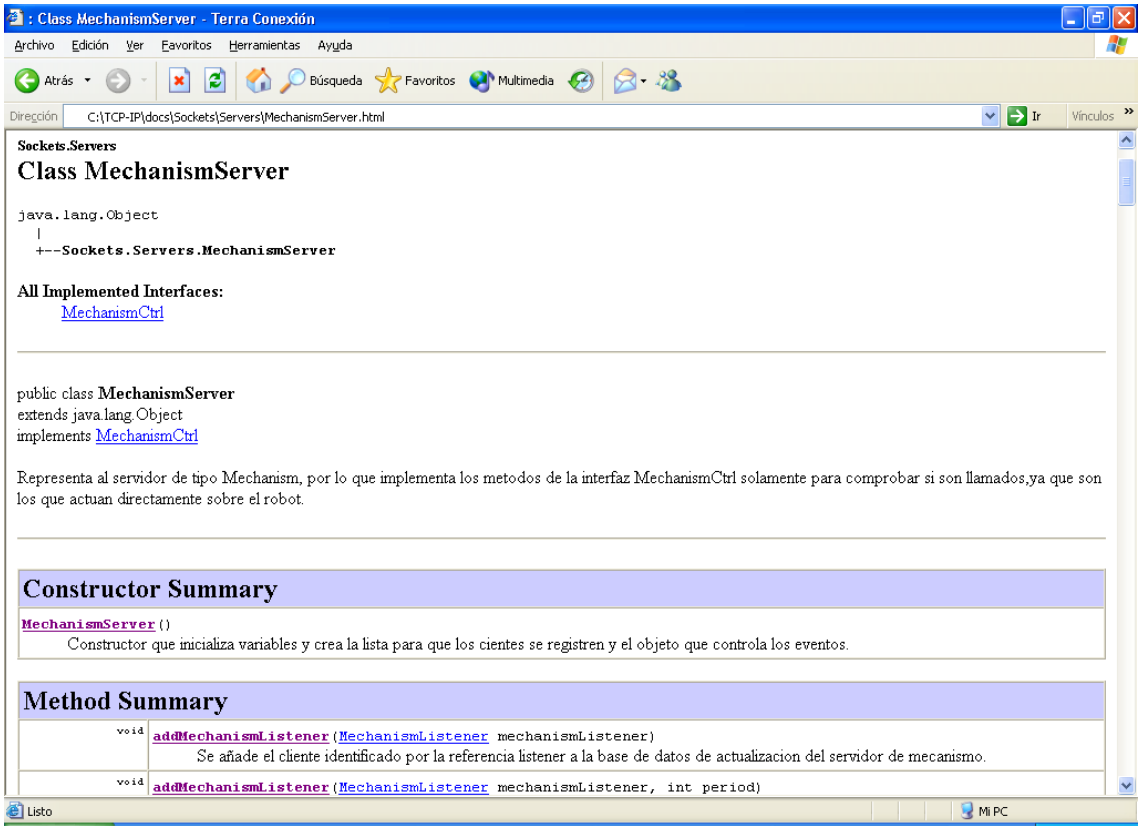
El código implementado para esta aplicación está comentado siguiendo las normas de JavaDoc. Las convenciones de código son importantes para los programadores por un gran número de razones:

- El 80% del coste del código de un programa va a su mantenimiento.
- Casi ningún software lo mantiene toda su vida el auto original.
- Las convenciones de código mejoran la lectura del software, permitiendo entender código nuevo mucho más rápidamente y más a fondo.
- Si se distribuye código fuente como un producto, es necesario asegurarse de que esta bien hecho y presentado como cualquier otro producto.

No es necesario comentar todas las líneas de código (salvo algún caso excepcional que requiera ser remarcado por su importancia), realmente lo único imprescindible a comentar es la clase y cada uno de los métodos que empleemos, bien para quien quiera entender qué hace esa clase o la función que desempeñan cada uno de los métodos o bien para que luego la herramienta de generación automática de documentación nos cree la nuestra.

Así, gracias a la aplicación Javadoc que se acompaña en el JDK, se pueden crear documentos en formato HTML similares a los que se distribuyen en la ayuda del API de Java. Así, aplicando esta utilidad a nuestro código vamos a obtener un conjunto de estos ficheros que se suministrarán a todos aquellos programadores que deseen reutilizar la aplicación, y por tanto, utilizar las clases y métodos ya creados. Estos ficheros se adjuntan en el proyecto entregado.

La imagen de la figura presenta un ejemplo de la documentación generada pa ra la clase MechanismServer, donde se puede comprobar que el documento contiene todas las características que pueden representar a una clase (paquete, interfaces, métodos, constructores, etc.). No obstante, esta imagen es solamente un ejemplo para hacerse una idea de la importancia de la documentación del código a la hora de distribuir una aplicación. Para visualizar la documentación completa de la aplicación diríjase a los ficheros que se adjuntan con el proyecto.


: Class MechanismServer - Terra Conexión
Archivo Edición Ver Favoritos Herramientas Ayuda
Atrás Búsqueda Favoritos Multimedia
Dirección C:\TCP-IP\docs\Sockets\Servers\MechanismServer.html
Ir Vínculos
Sockets.Servers
Class MechanismServer
java.lang.Object
+--Sockets.Servers.MechanismServer
All Implemented Interfaces:
MechanismCtrl
public class MechanismServer
extends java.lang.Object
implements MechanismCtrl
Representa al servidor de tipo Mechanism, por lo que implementa los metodos de la interfaz MechanismCtrl solamente para comprobar si son llamados,ya que son los que actuan directamente sobre el robot.
Constructor Summary
MechanismServer()
Constructor que inicializa variables y crea la lista para que los cientes se registren y el objeto que controla los eventos.
Method Summary
void addMechanismListener(MechanismListener mechanismListener)
Se añade el cliente identificado por la referencia listener a la base de datos de actualizacion del servidor de mecanismo.
void addMechanismListener(MechanismListener mechanismListener, int period)
Listo
Mi PC

13. Conclusiones

Una vez finalizada la aplicación Goya, y analizando las características que ofrece la implementación final, podemos distinguir dos tipos de cualidades. Analizando las que se pueden observar mediante la ejecución, destacan: la fiabilidad (habilidad del sistema para mantenerse operativo en el tiempo) y la funcionalidad (habilidad del sistema para hacer el trabajo para el que fue creado). En cuanto a los factores internos que se pueden observar destacan la modificabilidad, ya que el sistema con modificar las interfaces que se implementan y algunos pequeños cambios más, se puede adaptar a la funcionalidad de cualquier aplicación; portabilidad: esta cualidad queda más que probada por el hecho de que la implementación está realizada en Java; y reusabilidad: ya que tanto el protocolo interno o bien el conjunto formado por protocolo - proxy/stub puede ser reutilizado para cualquier otra aplicación cliente/servidor.

Aunque se trate de una simple aplicación cliente/servidor de las que existen muchas en la actualidad, donde el cliente realiza peticiones y el servidor las responde, no hay que olvidar que el servidor implementado tiene una gran importancia al permitir que los clientes que utilicen sus servicios puedan encontrarse en cualquier localización: ya sean uno o varios clientes sin importar si están todos o alguno de ellos en la misma o distinta maquina que el servidor. Teniendo también presente, que se han llegado a realizar pruebas del protocolo en redes tan dispares como pueden ser desde una simple red local hasta Internet, sin encontrar ninguna complicación para ello. También hay que destacar la implementación realizada para permitir el almacenamiento de los clientes que, según el tipo de registro realizado, recibirán la actualización de las variables deseadas.

Tras la realización de este protocolo en Java, que es uno de los lenguajes de programación más actuales, y tras haber realizado otras muchas aplicaciones utilizando la interfaz Socket (protocolo TCP) con otros lenguajes como son C y Delphi, se puede llegar a la conclusión de que realizar este tipo de aplicaciones resulta mucho más sencillo y ágil utilizando el lenguaje Java. Y es que, además de la ya comentada ventaja de portabilidad, la simplificación que ofrecen los métodos de la implementaron Java de la interfaz Socket recomiendan claramente su utilización. Como es de suponer, esta ventaja tiene un principal inconveniente y es que los requerimientos hardware necesarios para el desarrollo de una aplicación en Java son altos. De este modo, por ejemplo, puede resultar muy ineficiente la compilación del código fuente en una maquina que no posea de un gran procesador. Es por esto, que el rendimiento percibido al estar ejecutando la aplicación puede ser escaso, si la maquina sobre la que esta corriendo no tiene un potente hardware.

En cuanto a la utilización del protocolo TCP como nivel de transporte sobre el que trabaja el protocolo creado, no cabe duda de que ha sido una decisión acertada, ya que además de las características técnicas ofrecidas, al disponer de la interfaz Socket la utilización se simplifica de forma considerable.

En definitiva podemos calificar a la aplicación desarrollada como un sistema orientado a objetos, modular y robusto, pero que ha presentado ciertas dificultades en aspectos de seguridad y comunicaciones. Se puede concluir con que el uso de sockets dificulta el diseño de aplicaciones distribuidas (a pesar de las facilidades que proporciona la clase Socket de Java) y actualmente ya se están integrando en la empresa aplicaciones basadas en middleware como RMI y CORBA. Estas tecnologías proporcionan la infraestructura de comunicaciones necesaria para la implementación de sistemas de objetos distribuidos, sin que sea necesario implementar un protocolo de comunicaciones de bajo nivel como el desarrollado en este proyecto. No obstante, cada tecnología presenta sus ventajas y desventajas respecto al uso de sockets, por lo que en el siguiente apartado se presentara una comparativa más detallada sobre las facilidades y dificultades del uso de sockets y tecnologías middleware.

14. Trabajos futuros.

Aunque la aplicación Goya desarrollada en el proyecto dispone de todos los servicios necesarios para el manejo del robot Goya mediante una arquitectura cliente / servidor, existen diversas implementaciones adicionales que serían de gran importancia para la evolución de esta aplicación. Entre ellas destacamos el desarrollo de servicios de seguridad que proporcionen los permisos necesarios para poner a disposición los servidores de control a través de Internet de forma fiable y robusta. Se estudiaran 2 posibles casos en los que se pueda añadir seguridad a la aplicación, estos son la seguridad que ofrece Java y seguridad mediante SSL.

Por último, se propondrá el diseño de la aplicación Goya mediante el uso de un middleware que sustituya al protocolo de bajo nivel definido en este proyecto. Se realizará un pequeño estudio comparativo sobre las ventajas y desventajas del uso de tecnologías middleware como RMI y CORBA respecto al uso de Sockets.

14.1. Servicios de seguridad.

La seguridad en las aplicaciones cliente / servidor es importante, y especialmente en la aplicaciones como la desarrollada en este proyecto que permite distribuir la funcionalidad de los servidores a través de Internet. Como ya se sabe, Internet es una red a la que tiene acceso todo el mundo y podría ser peligroso conceder permiso al código para aceptar o crear conexiones sobre host remotos porque código malevolente podría fácilmente transferir y compartir datos confidenciales. Todo ello podría tener graves consecuencias, ya que se puede dar el caso de que cualquier host conectado a Internet podría conectarse a nuestros servidores y controlar el robot sin la autorización correspondiente. Por ello, es importante conocer todos los elementos capaces de proporcionar seguridad a la aplicación desarrollada en el proyecto y aunque no es objetivo de este proyecto detallar todos los elementos que forman parte de la Maquina Virtual de Java (JVM en adelante), si hay que destacar aquellos que estén involucrados en el modelo de seguridad de Java. Estos elementos son:

- ClassLoader: antes de que la JVM pueda ejecutar un programa, ésta debe localizar las clases que componen un programa y cargarlas en memoria. Este proceso no es tan fácil como puede resultar en otros lenguajes, ya que en Java las clases pueden ser cargadas tanto del sistema local, como de la red. La JVM divide las clases en 2 grupos: las clases seguras que son aquellas cuyo comportamiento es seguro y correcto (solo se consideran de este tipo aquellas distribuidas por el JRE); y las clases inseguras que son las

restantes y que se localizarán en la ruta especificada en el Classpath. Con este elemento se asegura cómo y de donde se cargarán las clases, evitando que clases que acceden al sistema durante la ejecución puedan ser remplazadas. De este modo evitamos por ejemplo, que la clase String que usamos en un desarrollo no haya sido modificada por un usuario malicioso que pretenda realizar otras acciones que no son las que se detallan en la especificación del JDK.

- Verificador de clases: se encargará de comprobar la integridad del bytecode de las clases inseguras que van a ser cargadas, comprobando que su tamaño, estructura y características de ejecución sean correctas.

- Security Manager: como incluso después de pasar por el verificador de clases, el código está sujeto a restricciones en tiempo de ejecución, este elemento se encargará de gestionar el control de acceso durante la ejecución. Éste comprobará si los métodos invocados en las clases tienen permisos para realizar determinadas acciones como pueden ser leer ficheros, abrir sockets, etc. El Security Manager pertenece al paquete java.lang y puede ser extendido, permitiendo al usuario crear su propia implementación del manager. Así por ejemplo, cuando se ejecuta un applet el Security Manager se encarga de que no se intenten realizar acciones que no están permitidas.

Estos 3 elementos de la JVM junto con el Control de Acceso forman el modelo de seguridad completo de Java. Este Control de Acceso ha sufrido varias modificaciones en las diferentes versiones del JDK. En la versión 1.0 se consideraba que todo el código local era seguro, por lo que tenía acceso a todos los recursos del sistema. En cambio, todo el código remoto (el descargado de la red) se considera inseguro, no teniendo acceso a casi ningún recurso. Según esta versión, el código descargado tenía muy pocas aplicaciones.

En la versión 1.1 se amplió el concepto dividiendo el código inseguro en firmado y no firmado. Así, todo código descargado de la red que este acompañado de su correspondiente firma, tendrá acceso a todos los recursos; y aquel que no esté firmado, seguirá sin poder acceder a dichos recursos. En la ultima versión desarrollada (Java2), se ha "globalizado" el tratamiento del código y ya no importa la procedencia de éste. Ahora, todo el código está sujeto al Control de Acceso y a las políticas de seguridad. El código estará asociado a un dominio de seguridad que es una asociación entre el código, su firma y los permisos de este. Será el Security Manager el que se encargue de comprobar estos permisos. Por ejemplo, cuando una clase intenta abrir un socket, la implementación de la clase Socket llamará al método SecurityManager.checkPermission() antes de abrirlo. Este método comprueba si en la política

de seguridad esa clase tiene permiso para abrir un socket. Si lo tiene seguirá adelante, sino, se lanzará una excepción de seguridad.

Resumiendo, el modelo de seguridad tiene 2 componentes: por un lado, los elementos de la JVM que verifican la correcta ejecución del código; y por otro lado, el Control de Acceso asignará permisos al código simplificando enormemente el desarrollo de los programas, porque ahora el programador no necesita programar la seguridad; simplemente hará el programa y luego le asignará unos permisos.

14.1.1. Control de Acceso.

Para comprobar y asignar unos permisos se usa un fichero llamado java.policy, localizado por defecto en el directorio raíz de Java, dentro de lib, dentro de security. Normalmente se usan 2 ficheros de políticas de seguridad: el que viene por defecto y uno creado por el usuario. Existe además otro fichero llamado java.security también localizando en el mismo directorio que el anterior, que se utiliza para indicarle a la JVM donde tiene que buscar el fichero de políticas de seguridad.

En el fichero java.policy que servía para gestionar el control de acceso se almacena una lista de URLs junto con los permisos para cada una de estas URLs. A continuación se muestra un pequeño ejemplo:

```
grant codeBase "file:/C:${/}work${/}MiClase.class"
{
        permission java.net.SocketPermission "192.168.4.23","accept,listen";
        permission java.io.FilePermission "<<ALL FILES>>", "write, read";
};
```

Como puede observarse el formato de este fichero es muy simp le, y consiste en una serie de cláusulas del tipo:

```
Grant [signedBy "signers"] [,codeBase "URL"]{
        permission permission_class ["target"] [ ,action_list] [, signedBy "signer"];
        ..
.};
```

Cada cláusula dice que el código firmado por "signers" y/o localizado en "URL" tiene los permisos "permission_class". Es decir, en el fragmento anterior, el código localizado en "file:/C:/work/MiClass.class" tiene permiso para aceptar conexiones procedentes de una determinada direccion IP y también para leer y escribir todos los ficheros del sistema. Hay que destacar que los permisos se asignan al código localizado en cierto URL o al código firmado por alguien, y las dos condiciones se pueden combinar.

Los permisos se definen usando la clase java.security.Permission y sus e xtensiones. En realidad, en el fichero solo se pone el nombre completo de la clase, como por ejemplo, java.io.FilePermission. Además del nombre de la clase, se pone un objetivo "target" y unas acciones "actions". En el ejemplo de la clase FilePermission, el objetivo es el fichero sobre el cual se adquieren permisos y las acciones son lo que pueden hacer con el objetivo, como leer y escribir. A continuación se muestra una lista de todas las clases de Permisos:

java.security.Permission, java.security.PermissionCollection, java.security.Permissions, java.security.UnresolvedPermission, java.io.FilePermission, java.net.SocketPermission, java.security.BasicPermission, java.util.PropertyPermission, java.lang.RuntimePermission, java.awt.AWTPermission, java.net.NetPermission, java.lang.reflect.ReflectPermission, java.io.SerializablePermission, java.security.SecurityPermission, java.security.SecurityPermission.

Estas clases pueden ser usadas tanto para especificar los permisos en el fichero java.policy, como para ser usadas dentro de un programa Java a la hora de hacer llamadas al SecurityManager. Para verificar si el programa que se está ejecutando tiene permisos, se utiliza el SecurityManager y para ser más exactos, se hace una llamada a SecurityManager.checkPermission(), cuyo argumento es un objeto de tipo Permission. Si al modificar los permisos del fichero "java.policy" se le añaden las líneas:

```
grant {
permission java.security.AllPermission;
};
```

tendremos la menor seguridad posible. Se puede (y se debe) tomar una política más restrictiva en la que únicamente se le den acceso a las situaciones que conozcamos que no supondrán peligro alguno, pero a la hora de realizar el desarrollo, se optó por modificar el fichero como se indica en las líneas de arriba, ya que así no aparecerá ningún problema relacionado con la seguridad, permitiendo el acceso a la aplicación a cualquier cliente.

De todas las clases que proporciona el control de acceso Java, la clase que más interesa utilizar para proporcionar seguridad a la aplicación Goya es la clase SocketPermission, ya que puede conceder permisos al código para aceptar o crear conexiones sobre host remotos. La funcionalidad de esta clase se resume a continuación.

El permiso java.net.SocketPermission concede acceso a una red mediante sockets. La fuente es un nombre de host y la dirección del puerto, y las acciones una lista que especifica las formas de conexión con ese host. Las conexiones posibles son accept, connect, listen, y resolve.

Ejemplos de ficheros policía:

Esta entrada de fichero de policía permite que una conexión acepte conexiones al puerto 7777 en el host client.goya.com.

```
grant {
  permission java.net.SocketPermission
       " client.goya.com:7777",
"connect, accept";
};
```

Esta entrada de fichero de policía permite a la conexión, aceptar conexiones para escuchar cualquier puerto entre el 1024 y el 65535 en el host local.

```
grant {
  permission java.net.SocketPermission
       "localhost:1024-",
"accept, connect, listen";
};
```

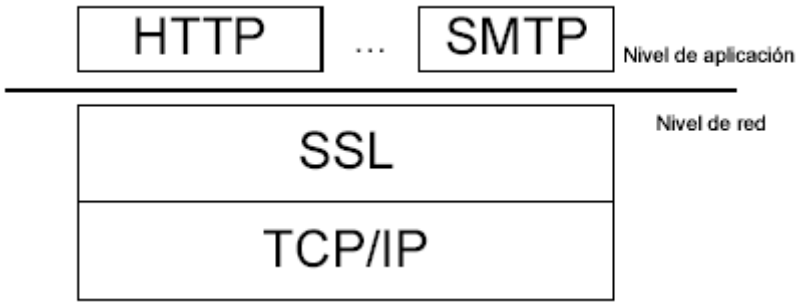
El host se expresa con la siguiente sintaxis como un nombre DNS, una dirección IP numérica, o como localhost (para la máquina local). El comodín asterisco (*) se puede incluir una vez en una especificación de nombre DNS. Si se incluye debe estar en la posición más a la izquierda, como en *.goya.com.

```
host = (hostname | IPaddress)[:portrange]
portrange = portnumber | -portnumber |
portnumber-[portnumber]
```

El puerto o rango de puertos es opcional. Una especificación de puerto de la forma N-, donde N es un número de puerto, significa todos los puertos numerados N y superiores, mientras que una especificación de la forma -N indica todos los puertos numerados N e inferiores. La acción listen es sólo importante cuando se usa con el localhost, y resolve (resuelve la dirección del servicio host/ip) cuando cualquiera de las otras opciones está presente.

14.2. SSL (Secure Socket Layer) con Sockets.

SSL son las siglas de Secure Socket Layer o capa segura de sockets. Es una tecnología desarrollada por Netscape para asegurar la privacidad y fiabilidad de las comunicaciones entre dos aplicaciones. SSL representa una alternativa a los sockets TCP/IP utilizados normalmente, pero hay que destacar que estos nuevos sockets se encuentran en una capa intermedia entre los protocolos TCP/IP donde trabajan los sockets "originales" y entre los protocolos de aplicación, como pueden ser HTTP o SMTP. Es por esto, que las aplicaciones que trabajan sobre SSL deberán tener un diseño adaptado a esta capa, ya que ahora las llamadas deben hacerse a la capa SSL y no a la de transporte como ocurría antes. Hubiera sido muy beneficioso para este nuevo protocolo que su uso tuviera una implementación transparente porque SSL necesita cierta información criptográfica adicional para establecer un socket.



Utiliza un sistema de encriptación asimétrico basado en claves publica/privada para negociar una clave que se utiliza para establecer una comunicación basada en encriptación simétrica. SSL es el protocolo de encriptación más utilizado en Internet en estos momentos y es el más usado en servidores web donde se solicita información confidencial. SSL es utilizado por el nivel de aplicación como capa de transporte de forma totalmente transparente independiente del protocolo utilizado, por lo que es una opción ideal para dotar a nuestra aplicación de un alto nivel de seguridad con muy poco esfuerzo.

Las principales propiedades de seguridad proporcionadas por SSL son:

- Comunicación segura basada en encriptación simétrica.
- Autentificación y negociación basada en encriptación asimétrica.
- Comunicación fiable basada en protocolos de integridad de mensajes.
- Privacidad: la información se transmite cifrada.
- Integridad de datos: Se usan funciones hash para asegurar que no se modifica la información.
- Autenticidad: Se intercambian certificados digitales.
- No repudio: El uso de firmas y certificados digitales asegura que no se pueda repudiar una transacción.

SSL puede ser utilizado en Java para obtener un nivel de sockets seguro. SUN todavía no proporciona un conjunto de clases que permitan utilizar SSL aunque si recomienda una serie de implementaciones comerciales que pueden ser utilizadas

14.2.1. Análisis de las propiedades de seguridad SSL.

SSL tiene dos fases en su proceso de comunicación. En la primera fase se establece una comunicación basada en encriptación asimétrica donde el cliente y el servidor intercambian los primeros mensajes y realizan la negociación de los parámetros de la sesión. Esta fase esta soportada por un protocolo conocido como HandShake (estrechamiento de manos). En la segunda fase se establece la verdadera sesión de comunicación donde las aplicaciones intercambian información.

Se pueden producir ataques sobre el protocolo en la comunicación a través de la red, sobre los mensajes intercambiados entre el cliente y el servidor. Se supone que un posible atacante podría realizar diversas operaciones ilícitas sobre los mensajes como:

- Sustitución.
- Eliminación.
- Interceptación.
- Desencriptacion.

SSL ha sido diseñado teniendo en cuenta este tipo de ataques e implementa mecanismos para detectarlos, aunque su comentario no es el objetivo de este apartado.
Entre las opciones de comunicación de SSL existe la posibilidad de autentificar a los participantes de la conexión por medio del uso de Certificados. Un Certificado no es mas que

una identificación que puede ser comprobada por una Entidad de Verificación como puede ser VeriSing. Sin embargo, SSL permite también que los participantes no sean autentificados. Por lo tanto, podemos destacar los siguientes modos de comunicación de SSL:

- Comunicación anónima: Ninguno de los participantes esta autentificado.
- Server autentificado.
- Cliente autentificado.
- Ambos autentificados.

Evidentemente, la opción más segura es aquella en la que tanto el cliente como el servidor están autentificados. SSL puede sufrir un ataque directo a la comunicación encriptada por métodos de fuerza bruta. Los métodos de encriptación de clave publica/privada se basan en funciones matemáticas de "un solo sentido". Esto no quiere decir realmente que no se pueda conseguir calcular la función inversa, si no que es mucho mas costoso computacionalmente que calcular la función normal. El tiempo necesario para desencriptar un mensaje depende en gran medida de la longitud de las claves utilizadas en la encriptación. Actualmente se considera que una clave de 40 bits es poco segura y una clave de 154 bit es prácticamente invulnerable. La longitud de claves utilizada por SSL depende de la implementación utilizada.

14.2.2. La suite de cifrado.

Como ya se dijo anteriormente, la realizac ión de un protocolo de encriptación es un proceso muy costoso y que no está al alcance de todos los programadores. Por tanto, es necesario recurrir en muchos casos a algoritmos ya implementados. La suite de cifrado es una combinación de algoritmos usados por una conexión SSL y los parámetros asociados a los mecanismos criptográficos usados.

Hay una serie de suites de cifrado definidas en el estándar para proporcionar diferentes servicios criptográficos, con nombres asociados que describen su contenido; así por ejemplo, podemos encontrar la suite SSL_RSA_EXPORT_WITH_DES_40_MD5 en la que se comprende que se usa: RSA para el cifrado de la clave pública, DES como algoritmo de cifrado simétrico con una clave de 40 bits y MD5 como algoritmo de resumen o hash. Con esto vemos que una suite de cifrado engloba:

- El algoritmo de clave pública usado para el intercambio de claves.
- El algoritmo de cifrado simétrico usado.
- El algoritmo de resumen usado.

14.2.3. Factorías de sockets.

Si queremos entender como SSL se integra en nuestra aplicación debemos comprender como funcionan las Custom Socket Factories. Los pasos a realizar para crear una nueva Socket Factory son:

1. Crear de un flujo (stream) de entrada y otro de salida que extiendan de las clases FilterInputStream y FilterOutputStream respectivamente. Esto es necesario porque la capa socket utiliza el concepto de Stream para implementar la conexión a la red.
2. Crear nuestro nuevo socket que debe extender de la clase Socket utilizando los Streams creados en el punto. Al crear un nuevo tipo de socket tenemos que redefinir los streams que va a devolver nuestro nuevo socket.
3. Crear la nueva Socket Factory que utilice los nuevos sockets creados en el punto anterior.

En la implementación de los métodos write y read de los streams c reados es donde definimos realmente el comportamiento de nuestra capa de transporte, es decir, ahí es donde se deberá implementar la encriptación, compresión o cualquier otra acción deseada. En cambio los métodos de la nueva clase socket no será necesario volver a implementarlos y simplemente bastará con rescribirlos. Por ultimo, lo único que nos queda por hacer es utilizar nuestra SocketFactory en un programa A continuación se muestra un fragmento de código en el que se instancia un socket SSL utilizando la factoría que proporciona Java por defecto:

```
SSLServerSocketFactory serversf =
                        (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
SSLServerSocket s = (SSLServerSocket) serversf.createServerSocket(1250);
SSLSocket c = (SSLSocket) s.accept();
```

Podemos comprobar que utilizando el API de referencia de Sun se puede añadir de forma sencilla, la funcionalidad de SSL en los programas Java, aunque el rendimiento sea bastante pobre ya que se necesita gran capacidad de procesado.

14.3. Implementación mediante middleware (RMI - CORBA).

Una vez realizada la implementación de la aplicación Goya mediante Sockets, y siendo necesario el diseño de un protocolo que defina las reglas y criterios necesarios para el paso de mensajes a través de los enlaces de comunicaciones, podemos considerar el esfuerzo y tiempo necesario para realizar un protocolo de estas características. Sin embargo, al tener conocimiento de la existencia de otras tecnologías orientadas a objetos distribuidos como son RMI (Remote Method Invocation) y CORBA, se considera que utilizando el middleware que ofrecen estas tecnologías el desarrollo de la aplicación se vería simplificado en gran medida. A continuación se presentará un resumen comparativo de las características que las tecnologías Sockets, RMI y CORBA presentan para el desarrollo de arquitecturas cliente / servidor similares a la aplicación desarrollada en este proyecto, que nos servirá para evaluarlas en base a una serie de criterios. De forma resumida, las características que se considerarán se engloban en cinco aspectos:

1) Proceso de desarrollo Cliente/Servidor.
2) Grado de integración con Java.
3) Instalación y despliegue.
4) Rendimiento.
5) Escalabilidad.

Los puntos que se consideran en la tabla comparativa son los siguientes:

Nivel de Abstracción. A medida que el nivel de abstracción ofrecido por la tecnología aumenta, nuestra aplicación debe realizar menos tareas. Con los Sockets teníamos que definir convenciones de paso de parámetros, tipos de datos, marshaling, definición de servicios, etc. CORBA y RMI poseen un mayor nivel de abstracción.

Integración con Java. CORBA y RMI proveen la mejor integración con Java. Los Sockets ofrecen librerías de bajo nivel.

Soporte multiplataforma. Tanto CORBA, RMI y Sockets se ejecutan en la mayoría de las plataformas ya que es Java.

Implementación total en Java. Hay implementaciones de CORBA, Sockets y RMI escritas completamente en Java.

Soporte de tipos. Las tecnologías basadas en Objetos Distribuidos proveen soporte para tipos de datos y chequeo en tiempo de compilación y ejecución. CORBA permite especificar parámetros de salida y de entrada/salida; RMI sólo de entrada; y los Sockets no proveen ninguno.

Facilidad de configuración. La configuración para las tres tecnologías es sencilla. No obstante, el trabajo con sistemas distribuidos no es fácil.

Invocación de métodos distribuida. Sólo las basadas en Objetos Distribuidos permiten la invocación distribuida de métodos. CORBA además soporta referencias a objetos únicas.

Estado entre invocaciones. Con Sockets podemos mantener el estado, pero una vez más, esto va por cuenta del programador. Las dos tecnologías de Objetos Distribuidos conservan el estado de sus objetos, pero sólo CORBA ofrece un identificador único a cada objeto, que permite reconectar al mismo objeto (con su mismo estado) en un momento posterior.

Soporte de Meta-información. Sólo CORBA soporta introspección, permiten descubrir de forma dinámica los interfaces que ofrece un objeto. CORBA soporta también Repositorios de Interfaces inter-ORB.

Invocación dinámica. Una vez más, sólo CORBA ofrece esta posibilidad.

Rendimiento. CORBA y los Sockets son los que ofrecen un mejor rendimiento, mientras que RMI está muy lejos de estas cifras.

Seguridad. Los Sockets proveen seguridad a través de SSL (Secure Socket Layer). CORBA define también un servicio de seguridad, integrado en el ORB. Tanto CORBA como RMI soportan SSL.

Objetos persistentes. Sólo CORBA permite objetos persistentes que también poseen referencias persistentes.

Soporte multilenguaje. Todos los lenguajes soportan Sockets. RMI sólo funciona con Java. CORBA ofrece también soporte multilenguaje, definiendo un estándar de mapping de alto nivel.

Protocolo común multilenguaje. CORBA provee protocolos estándar de comunicación entre entidades independientes del lenguaje, el Sistema Operativo y la plataforma hardware.

Escalabilidad. Los Sockets soportan escalabilidad a través de redes TCP/IP. Sin embargo, este soporte es a un nivel de abstracción muy bajo. En contraste, CORBA ofrece federaciones de ORBs, dominios de nombres, etc. IIOP establece una manera estándar de propagar transacciones, seguridad, etc., a través de ORBs de distintos fabricantes. La idea básica es que provee la infraestructura para conectar ORBs débilmente acoplados.

Estándar abierto. Una infraestructura de aplicaciones distribuidas a escala mundial que posiblemente se integre con Internet no puede pertenecer a una única compañía. Debe basarse en un conjunto de estándares bien definido en el que los distintos vendedores puedan competir, sin alcanzar ninguno el control total. CORBA y Sockets son abiertos: están controlados por un cuerpo de estandarización. RMI no lo es, ya que pertenece a Sun.

Característica	CORBA	RMI	Sockets
Nivel de abstracción	★★★★	★★★★	★
Integración con Java	★★★★	★★★★	★★
Soporte multiplataforma	★★★★	★★★★	★★★★
Implementación total en Java	★★★★	★★★★	★★★★
Soporte de tipos	★★★★	★★★★	★
Facilidad de configuración	★★★	★★★	★★★
Invocación de métodos distribuida	★★★★	★★★	☆
Estado entre invocaciones	★★★★	★★★	★★
Meta-información	★★★★	☆	☆
Invocaciones dinámicas	★★★★	★	☆
Rendimiento (ping)	★★★★ 3.3 ms	★★★ 5.5 ms	★★★★ 2.0 ms
Seguridad	★★★★	★★★★	★★★
Objetos persistentes	★★★★	☆	☆
Soporte multilenguaje	★★★★	☆	★★★★
Protocolo común multilenguaje	★★★★	☆	☆
Escalabilidad	★★★★	★	★★★★
Estándar abierto	★★★★	★★	★★★★

★★★★ = *mejor*; ★ = *peor*; ☆ = *N/A, muy mala o inexistente.*

Como se desprende de la tabla, en la que se analizan los puntos más importantes que hemos ido descubriendo durante el estudio de todas las tecnologías, la mejor tecnología es la integración de Java con CORBA.

15. Bibliografía.

Cliente/Servidor

[Boar93] Bernard H. Boar. Implementing client/server computing: a strategic perspective. McGraw-Hill, 1993.

[OHE94] Robert Orfali ... [et al.]. Essential client/server survival guide. John Wiley and Sons, 1994.

[Harold97] Elliotte Rusty Harold. Java network programming. O'Reilly, 1997.

[Allama00] Subrahmanyam Allamaraju... [et al.]. Professional Java Server programming : J2EE Edition. Wrox Press, 2000.

[Farley98] Jim Farley. Java distributed computing. O'Reilly, 1998.

- "La Arquitectura Cliente-Servidor Y Las Logicas De Medición" Dr. Alvaro Rendón Gallón. Universidad Del Cauca.
- "Sistemas en arquitectura Cliente Servidor" Departamento de control de calidad y auditoría.
- "Sistemas Operativos Distribuidos", Fernando Pérez Costova, Jose María Peña Sanchez
- "Métodos De Computación Distribuida", D. Juan José González Cid, Universidad de Vigo, Escuela Superior de Ingeniería Informática.
- "Arquitectura Cliente/Servidor", Armando Suárez Cueto, GPLSI.

Concurrencia

[Lea01] Doug Lea. Programación concurrente en Java principios y patrones de diseño. Addison Wesley, 2ª ed, 2001.

[Hyde99] Paul Hyde. Java tread programming. Sams, 1999.

[Kleiman96] Steve Kleiman ... [et al.]. Programming with threads. Sunsoft Press, 1996

[Lea00] Doug Lea. Programación concurrente en JAVA : principios y patrones de diseño. Addison-Wesley, 2ª ed, 2000.

Patrones

[Grand98] Mark Grand. Patterns in Java. Volume 1 : a catalog of reusable design patterns illustrated with UML. John Wiley and Sons, 1998

[Gamma95] Erich Gamma... [et al.]. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[Gamma94] Erich Gamma... [et al.]. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

- Articulo:"Diseño de Software con patrones", Alberto Molpeceres
- http://www.aqs.es/web/files/designpatterns.Html

UML

[Larman99] Craig Larman. UML y patrones: introducción al análisis y diseño orientado a objetos. Prentice Hall, 1ª ed, 1999.

[Reed02] Paul Reed. Developing applications with Java and UML. Addison Wesley, 2002.

[Quatrani00] Terry Quatrani. Visual modeling with Rational Rose 2000 and UML. Addison-Wesley, 2000.

[Rational02] Rational Software Corporation, Rational Rose 02 Enterprise Edition, http://www.rational.com/, 2002

Java

- Using Java 1.1", Joseph Weber. 3ª Edición. QUE Corporation. 1997.
- "Core Java 1.1", Cay S. Harstmann Y Gary Cornell. Sun Microsystems Press. 1998.
- "Java 1.2 Al Descubierto" De Jaime Jaworski (1999), Prentice Hall.
- "Thinking In Patterns With Java" De Bruce Eckel. President, Mindview, Inc.
- SUN Microsystems. The J2SE Documentation, http://java.sun.com/docs/ 2002.
- Using Java 1.1", Joseph Weber. 3ª Edición. QUE Corporation. 1997.
- "Core Java 1.1", Cay S. Harstmann Y Gary Cornell. Sun Microsystems Press. 1998.
- "Java 1.2 Al Descubierto" De Jaime Jaworski (1999), Prentice Hall.
- "Thinking In Patterns With Java" De Bruce Eckel. President, Mindview, Inc.

Seguridad

[Thomas00] Stephen Thomas. SSL and TLS essentials: securing the Web. John Wiley and Sons, 2000.

[Pistoia99] Marco Pistoia ... [et al.]. Java 2 network security. Prentice Hall, 1999.

[Jaworski01] Jamie Jaworski, Paul J. Perrone. Seguridad en Java: edición especial, Prentice-Hall, 2001.

[Oaks98] Scott Oaks. Java security. O'Reilly, 1998.

[Sun2002] Sun Microsystems. JavaTM Security. http://java.sun.com/security/ 2002.

Otros Proyectos Fin de Carrera

- "Objetos Distribuidos y persistencia con CORBA, Java Y C++". Javier Carlos Ros Sanchez, Facultad de Informática, Universidad de Murcia.
- "Comercio Electrónico, Tecnología y Aplicaciones", Felipe J. Sevilla Garnica
- "Plataforma de acceso a juegos multijugador en Internet", Juan María Calvo Tirado, Universidad de Valladolid, Escuela Técnica Superior de Ingeniería Informática.
- "Guía de construcción de software en java con patrones de diseño", Francisco Javier Martínez Juan, Escuela Universitaria de Ingeniería técnica en informática de Oviedo.
- "Aplicaciones Distribuidas en Internet/Intranets: De Los Sockets a los Objetos Distribuidos", Diego Sevilla Ruiz, Facultad de Informática, Universidad de Murcia.

16. Documentos y ficheros adjuntados al proyecto.

Junto a la presente documentación se entrega un CD-ROM, este disco contiene la versión digital de este documentos en formato Word, los documentos en formato digital empleados para la realización del proyecto, los ficheros fuente empleados para la realización de la aplicación desarrollada.

El contenido de este disco por directorios es el siguiente:

Documentación: Contiene toda la documentación entregada con el proyecto impresa en papel.

Código Fuente: Incluye todos los ficheros necesarios para compilar la aplicación Goya, así como los ficheros de Proyecto de Kawa 5.0.

Javadocs: Dispone de la documentación HTML de los paquetes y clases que forman parte de la aplicación Goya, generada automáticamente con la herramienta Javadoc proporcionada por JavaSoft en el JDK 1.3.

JAR: Contiene los ficheros JAR que almacenan las clases de la aplicación.

- Client.jar → JAR del cliente de la aplicación.
- Server.jar → JAR del servidor de la aplicación.

UML: Incluye los ficheros de proyecto de Rational Rose 98, que contienen todos los diagramas UML presentados en la memoria.