ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN
UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

PROTOCOLO DE COMUNICACIONES PARA ROBOTS DE SERVICIOS BASADO EN MIDDLEWARE

AUTOR: Jorge Angel Martínez Navarro
DIRECTOR(ES): Juan Angel Pastor Franco
Septiembre/ **2002**

Autor	Jorge Angel Martínez Navarro
E-mail del Autor	ajorgemartinez@hotmail.com
Director(es)	Juan Angel Pastor Franco
E-mail del Director	juanangel.pastor@upct.es
Codirector(es)	
Título del PFC	PROTOCOLO DE COMUNICACIONES PARA ROBOTS DE SERVICIOS BASADO EN MIDDLEWARE
Descriptores	Protocolo, Java, RMI, CORBA.

Resúmen

Desarrollo de 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). Se comparan las ventajas e inconvenientes del desarrollo utilizando la tecnología RMI y CORBA, usando en ambas el lenguaje de programación Java.

Titulación	Ingeniera Técnica Telecomunicación, especialidad Telemática
Intensificación	
Departamento	Departamento de Tecnologías de la Información y Comunicaciones
Fecha de Presentación	Septiembre- 2002

INDICE

1. Documentos y ficheros adjuntados al proyecto.

Pfc.doc → Memoria del Proyecto.

Presentacion.ppt → Presentación del proyecto.

CORBAClient.jar → Ejecutable del cliente de la implementación con CORBA.

CORBAServer.jar → Ejecutable del servidor de la implementación con CORBA.

RMIClient.jar → Ejecutable del cliente de la implementación con RMI.

RMIServer.jar → Ejecutable del servidor de la implementación con RMI.

Rmiregisty.exe → Aplicación de Sun necesaria para ejecutar implementación RMI.

Tnameserv.exe → Aplicación de Sun necesaria para ejecutar implementación CORBA.

- Directorio RMI:
  - Subdirectorio Clases → Contiene los ficheros .class de la implementación RMI.
  - Subdirectorio Código → Contiene el código fuente de la implementación RMI.
  - Subdirectorio Ayuda → Contiene los ficheros de ayuda html autogenerados por Javadoc.
  - Subdirectorio UML → Incluye los ficheros de proyecto creados con la aplicación Racional Rose, que contiene todos los diagramas UML presentados en la memoria.
- Directorio CORBA:
  - Subdirectorio Clases → Contiene los ficheros .class de la implementación CORBA.
  - Subdirectorio Código → Contiene el código fuente de la implementación CORBA.
  - Subdirectorio Ayuda → Contiene los ficheros de ayuda html autogenerados por Javadoc.
  - Subdirectorio UML → Incluye los ficheros de proyecto creados con la aplicación Racional Rose, que contiene todos los diagramas UML presentados en la memoria.

2. 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).

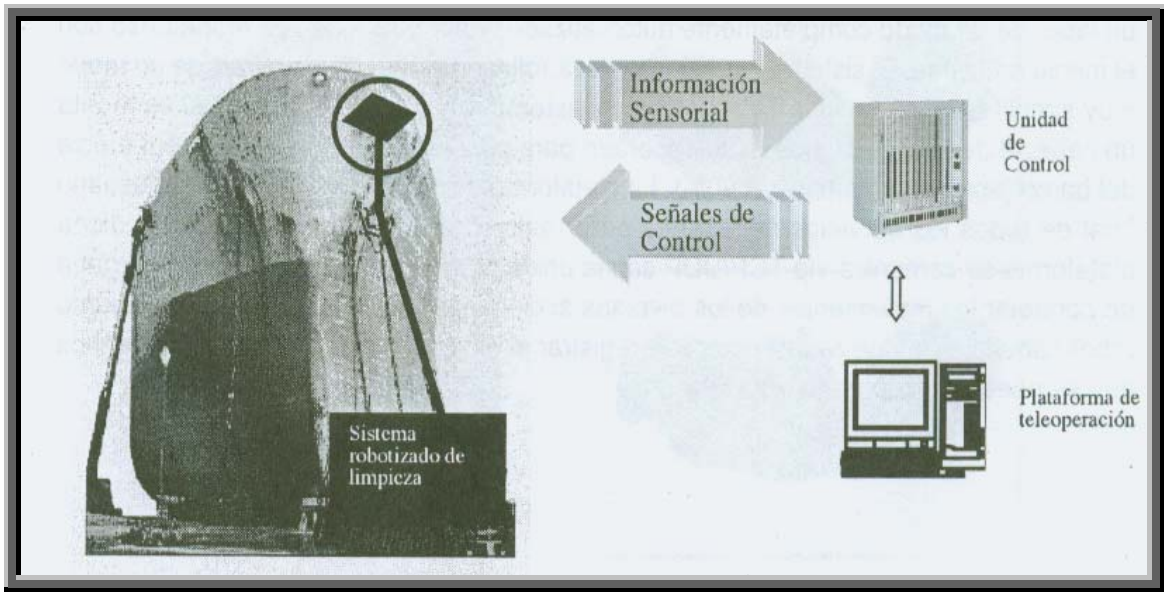
2.1 El robot Goya.

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 eliminación 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.

2.2 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.

3. Estado del arte.

3.1 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…

3.1.1 La computación distribuida.

Con el progreso de la electrónica, tanto el tamaño y costo 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).

3.1.2. Arquitectura Cliente/Servidor

El modelo Cliente/Servidor está basado en el concepto de servicio. Este modelo describe la funcionalidad de una aplicación mediante dos tipos de entidades lógicas independientes: clientes, o consumidores, y productores, o servidores, de forma que los servidores ofrecen una serie de servicios que pueden ser solicitados por los clientes para completar la funcionalidad de la aplicación. Para implementar este esquema de interacción, en el modelo cliente/servidor se distinguen los tres elementos software siguientes:

- Cliente. Aquel elemento software que realiza la petición de un servicio a un servidor capaz de proporcionarlo.
- Middleware. Conjunto de elementos software situados tanto en el cliente como en el servidor que permiten una comunicación transparente y fiable entre ambos. De esta forma la aplicación global puede abstraerse de aquellos elementos concretos que realizan la comunicación. El middleware incluye: los componentes que emplea el cliente para invocar un servicio, la transmisión de la solicitud por la red y los componentes que emplea el servidor para recibir la petición y transmitir la respuesta.
- Servidor. Aquel elemento software que atiende a los clientes interesados en cierto servicio y los recursos o datos que posee el servidor.

Desde un punto de vista arquitectónico, adaptar una aplicación al modelo cliente/servidor requiere definir y agrupar las diferentes funciones que ofrece la aplicación, y distribuirlas entre los diferentes elementos que conforman el modelo. Típicamente, se distinguen tres elementos: presentación, lógica de negocio y el modelo de datos. La presentación implementa una Interfaz Gráfica de Usuario (GUI) o una Interfaz en modo texto para ofrecer al cliente la posibilidad de interaccionar con el servidor mostrando los resultados de dicha interacción. La lógica de negocio, por su parte, implementa la funcionalidad central de la aplicación, realizando todos los procesos requeridos para llevar a término todos los requisitos de la misma. El último elemento (el modelo de datos) implementa el sistema de gestión de los datos y define la estructura de los mismos.

3.1.3 Aplicaciones de dos y tres capas y multicapa.

Tomando como base la manera en que la funcionalidad se divide entre cliente y servidor, las aplicaciones Cliente/Servidor se dividen en dos grupos: aplicaciones de dos niveles y aplicaciones de tres o más niveles.

Las aplicaciones de dos niveles fueron el primer paso en el desarrollo de aplicaciones Cliente/Servidor. Típicamente, la lógica de presentación, la de negocio y la de datos quedaban en la parte cliente, dejando a los servidores encargados sólo de guardar los datos, es decir, como servidores de Bases de Datos. Esta organización es sin duda un paso adelante con respecto a las soluciones basadas en mainframes, ya que permite una cierta escalabilidad y que varios clientes se puedan beneficiar de los datos residentes en los servidores. Sin embargo, no está libre de inconvenientes. Aunque es verdad que a lo largo del ciclo de vida de una aplicación los datos son más estables que los procedimientos, con esta configuración, un cambio en las bases de datos requería la programación de una nueva aplicación cliente y la distribución a todos los usuarios.

Además, esta solución también presenta una baja escalabilidad, ya que a medida que el número de clientes aumenta, el servidor de bases de datos se ve cargado de forma proporcional al número de clientes. Las aplicaciones son específicas para realizar cierta función y sólo comparten los datos. No se pueden reutilizar partes de aplicaciones ya creadas y se queda ligado al producto utilizado como servidor de base de datos.

Un refinamiento de la arquitectura anterior es la arquitectura de tres ó n niveles. Aquí, el cliente se encarga de mantener el interfaz gráfico de usuario, mientras que existen una serie de componentes intermedios en el sistema que se encargan de implementar la lógica de la aplicación. Por último, hay un último nivel que se encarga del la lógica de datos. En el momento en el que los componentes de este último nivel se conviertan en clientes de otros componentes, se convierte en una estructura multinivel. Esta configuración permite que los clientes se construyan en base a unos servicios encapsulados en los procesos que implementan la lógica de la aplicación, y por lo tanto son más inmunes a cambios tanto en la lógica como en los datos. Aún así, la funcionalidad que los clientes implementan es tan sencilla que los cambios son muy superficiales. Varios clientes pueden reutilizar servicios estándar definidos en el nivel intermedio. La aplicación, al estar dividida en partes más pequeñas, hace que el proceso de distribución de funcionalidad en los procesadores más adecuados sea muy flexible.

Las arquitecturas multicapa se obtienen cuando la capa que contiene la lógica de negocio en una aplicación de tres capas, se convierte en cliente de otras aplicaciones similares. En este esquema no está unívocamente definido el papel cliente o servidor de las distintas entidades que corren en las diferentes máquinas de la red, ya que todas cooperan unas con otras para conseguir la funcionalidad total de la aplicación de la que forman parte.

3.1.4 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.

3.2. Tecnología RMI.

3.2.1. Motivación.

Un protocolo diseñado para sistemas distribuidos requiere que la computación sea ejecutada en espacios de dirección diferentes (en distinta máquina) que se puedan comunicar. Java, al igual que otros muchos lenguajes posee la Interfaz Socket que es un mecanismo de comunicación básico. Sin embargo, los sockets requieren que el cliente y el servidor trabajen en varios niveles de protocolos y aplicaciones para codificar y decodificar los mensajes y, por tanto, el diseño de un protocolo utilizando esto puede ser engorroso y conducir a errores.

Una alternativa a los sockets son las Llamadas a Procedimiento Remoto (RPC), que abstrae la interfaz de comunicación al nivel de una llamada a procedimiento. Aunque el programador piensa que está realizando una llamada a un procedimiento local, en realidad se está invocando a uno remoto, empaquetando los argumentos en la llamada y enviándolos al host servidor. RPC codifica argumentos y retorna valores usando una representación externa de los datos; sin embargo, no trabaja bien en sistemas de objetos distribuidos, donde la comunicación entre objetos y programa residente en espacios de dirección diferentes es requerida. Es por esto, que para los sistemas de objetos distribuidos se utiliza más RMI o la Invocación del Método Remoto.

El sistema de invocación de método remoto de Java se ha diseñado específicamente para operar en el ambiente Java, mientras que RPC estaba orientado al lenguaje C. El sistema RMI de Java parte de la idea que las máquinas virtuales son similares, y por lo tanto puede aprovechar por consiguiente, el diseño del modelo de objeto de Java.

3.2.2 Objetivos de Java RMI.

Los objetivos del soporte de distribución de objetos de Java son:

- Soporte de invocación remota de objetos en diferentes máquinas virtuales.
- Integración del modelo de objeto distribuido en el lenguaje Java de una manera natural, manteniendo la mayor parte de la semántica de objeto de Java.
- Diferenciar claramente entre el objeto distribuido y el diseño de objetos locales.
- Escritura de aplicaciones distribuidas fiables lo más simple posible.
- Mantener las características de seguridad del ambiente de ejecución de Java.
- Varios tipos de semántica de referencia para objetos remotos; por ejemplo, referencias live (referencias no persistentes), referencias persistentes, y activación perezosa (lazy).

- Mantener el ambiente seguro que Java proporciona para los administradores de seguridad y cargadores de clase.
- Junto con estos objetivos planteados, es requisito general que el modelo RMI sea a la vez simple (fácil usar) y natural (se inserte bien en el lenguaje Java).

3.2.3 Java RMI.

RMI se introduce en JDK 1.1. Aunque es relativamente fácil usar, es una tecnología notablemente poderosa que proporciona al diseñador un nuevo paradigma de programación, aplicaciones de objetos distribuidos. Permite a un objeto que se está ejecutando en una Máquina Virtual Java llamar a métodos de otro objeto que está en otra diferente.

Las aplicaciones RMI normalmente comprenden dos programas separados: un servidor y un cliente. La aplicación servidor típica crea objetos remotos, hace accesibles referencias a dichos objetos remotos, y espera a que los clientes llamen a estos métodos u objetos remotos. Una aplicación cliente típica obtiene una referencia remota de uno o más objetos remotos en el servidor y llama a sus métodos. RMI proporciona el mecanismo por el que se comunican y se pasa información del cliente al servidor y viceversa.

El objetivo principal de RMI es permitir a los programadores diseñar programas distribuidos en Java con la misma sintaxis y semántica usadas en programas no distribuidos. La arquitectura RMI define el comportamiento de los objetos, como y cuando pueden ocurrir las excepciones, la administración de la memoria y como se pasan parámetros a y desde los métodos remotos.

3.2.4 Comparación de Programas Java Distribuidos y No distribuidos.

La arquitectura RMI trata de hacer el uso de objetos distribuido similar a usar objetos Java locales. Sin embargo hay ciertas diferencias que se muestran a continuación:

	Objeto local	Objeto remoto
Definición del objeto	Un objeto local es definido por clase un Java.	El comportamiento de un objeto remoto exportado se define por una interfaz extendida de la interfaz “remote”.

Implementación del objeto	Un objeto local es implementado por su clase Java.	El comportamiento de un objeto remoto es ejecutado por una clase Java que implementa la interfaz “remote”.
Creación del objeto	Una nueva instancia de un objeto local es creada por el operador “new”.	Una nueva instancia de un objeto remoto se crea en el servidor con el operador “new”. Un cliente no puede crear directamente un objeto nuevo remoto.
Acceso al Objeto	Se accede un objeto local directamente a través de una variable referencia del objeto.	Se accede a un objeto remoto a través de una variable referencia del objeto que apunta a una implementación (proxy o stub) de la interfaz remota.
Referencias	En una JVM simple, una referencia del objeto apunta directamente a un objeto en el heap (zona de memoria destinada a la ejecución de los programas).	Una "referencia remota" es un puntero a un objeto proxy (un "stub") en el heap local. Ese stub contiene información que permite conectarse a un objeto remoto, que contiene la implementación de los métodos.
Referencias activas	Se considera un objeto "vivo" si hay por lo menos una referencia a él.	En un ambiente distribuido, JVMs remotas podrían caerse, y pueden perderse las conexiones de la red. Se considera que se tiene una referencia activa a un objeto remoto si se ha accedido dentro de un cierto período de tiempo.
Colección de la basura	Cuando se han eliminado todas las referencias locales a un objeto, el objeto se vuelve	El basurero distribuido trabaja con el basurero local. Si no hay referencias remotas y se

	un candidato para la recolección de basura.	han eliminado todas las referencias locales a un objeto remoto, entonces se vuelve un candidato para la recolección de basura por los medios normales.
Excepciones	El compilador Java fuerza al programa a manejar todo tipo de excepciones.	RMI fuerza a los programas a tratar cualquier posible "RemoteException". Esto se realizó para asegurar la robustez de las aplicaciones distribuidas.

3.2.5 Arquitectura Java RMI.

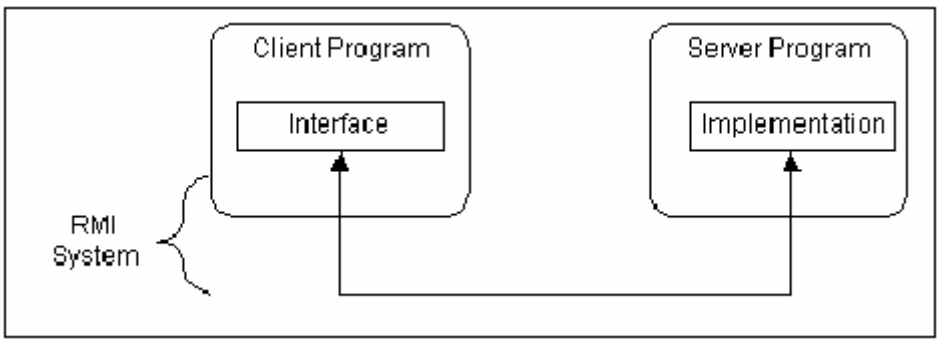
El objetivo del diseño de la arquitectura RMI era crear un modelo de objeto Java distribuido que se integrara naturalmente en el lenguaje de programación Java y en el modelo de objeto local. RMI ha logrado crear un sistema que extiende la seguridad y robustez de la arquitectura Java al mundo de la computación distribuida.

3.2.5.1. Interfaces: El Corazón de RMI.

La arquitectura RMI se basa en un principio importante: la definición de conducta y la implementación de esa conducta son conceptos separados. RMI permite que el código que define el comportamiento y la codificación que lleva a cabo la implementación estén separados se puedan ejecutar en JVM separadas. Esta idea encaja con las necesidades de un sistema distribuido donde:

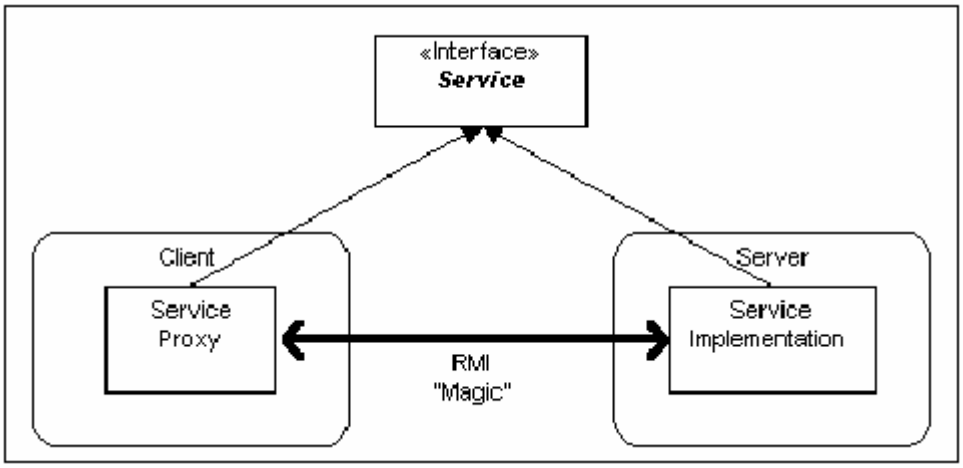
- El cliente se centra en la definición y utilización de un servicio.
- El servidor se enfoca en proporcionar el servicio.

En RMI, la definición de un servicio remoto se codifica en una interfaz Java y la implementación del servicio se codifica en una clase. Por consiguiente, la clave para la comprensión de RMI es recordar que las interfaces definen conducta y clases definen implementación. El siguiente dibujo ilustra la idea mencionada (recordando que en la interfaz no hay código):



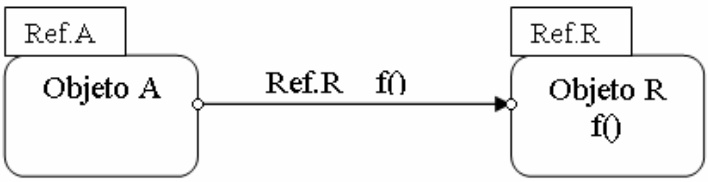
RMI soporta 2 clases que implementan la misma interfaz:

- Primera clase: implementación del comportamiento (se ejecuta en el servidor).
- Segunda Clase: actúa como proxy para el servicio remoto (se ejecuta en el cliente). Pero hay que destacar que esta clase ya no la implementa el programador, sino que se crea automáticamente mediante la utilidad "rmic".

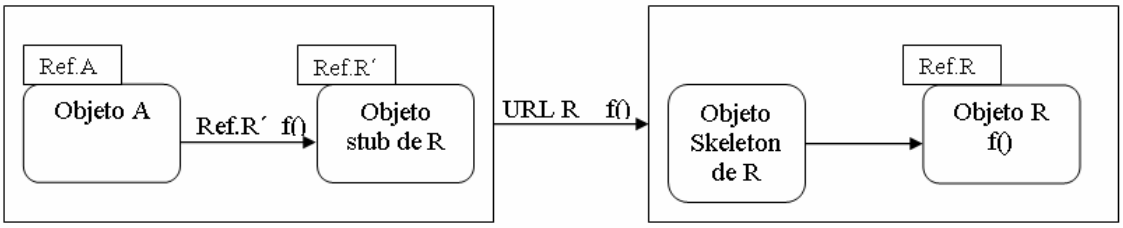


3.2.5.2. Modelo de llamada remota.

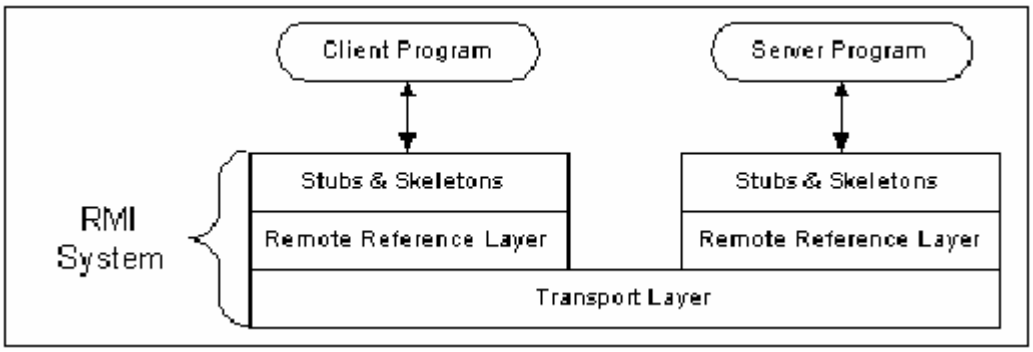
Como ya se comentó anteriormente, al usar RMI un objeto llama a métodos del objeto remoto como si fuera local; así, el usuario piensa que se sigue este esquema:



Pero en realidad, lo que se produce es que un programa cliente hace una llamada a un método en el objeto proxy, RMI envía el requerimiento a la JVM (que es reenviado hacia la implementación). Los valores retornados son enviados de vuelta al proxy y después al programa cliente. El esquema es el siguiente:



Para comprender este esquema de la arquitectura RMI, se van a comentar las 3 capas que constituyen la arquitectura RMI. Pero antes de nada hay que destacar que al usar una arquitectura de capas, cada uno de ellas puede ser reforzada o reemplazada sin alterar el resto del sistema. Por ejemplo, la capa del transporte puede ser reemplazada por UDP/ IP sin alterar las capas superiores.



3.2.5.3. Capa de Stub y Skeleton.

En esta capa, RMI usa el patrón proxy, donde un objeto esta representado por otro (el proxy) en un contexto distinto. El proxy sabe como enviar las llamadas a los métodos entre los objetos que participan.

El skeleton es una clase de ayuda generada por RMI y se encarga de:

- Comunicarse con el stub a través de un enlace RMI.
- Mantener una conversación con el stub.
- Leer los parámetros de las invocaciones.
- Hacer llamadas al objeto que implementa el servicio remoto.

- Acepta los valores de retorno.
- Escribir los valores de retorno de vuelta al stub.

3.2.5.4. Capa de referencia remota.

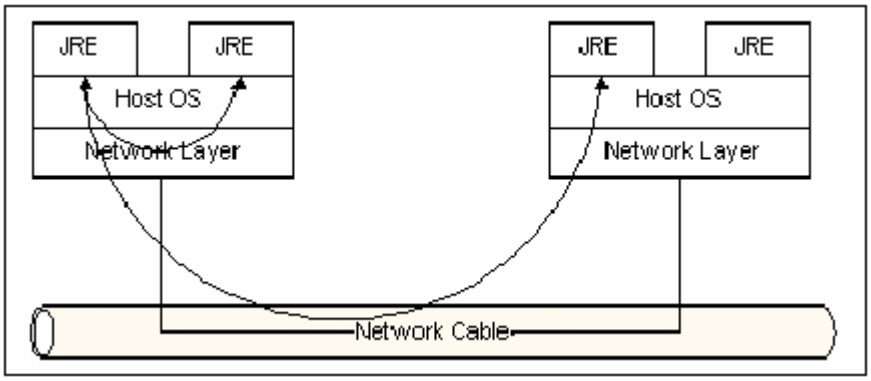
La capa de referencia remota define y soporta la semántica de una invocación en una conexión RMI. Esta capa provee un objeto (RemoteRef) que "hace" de enlace al objeto que implementa el servicio remoto .El objeto stub usará el método "invoke" del objeto RemoteRef para enviar la llamada al método.

RMI provee varias formas para que los clientes se conecten a las implementaciones de los servicios remotos:

- unicast, conexión punto a punto. Antes de que el cliente pueda usar un servicio remoto, el servicio remoto debe ser instanciado en el servidor y exportado al sistema RMI. (Si éste es el servicio primario, debe ser denominado y registrado en el registro RMI).
- Soporta objetos activables en forma remota. Cuando se hace una llamada a un método del proxy para un objeto activable, RMI determina si el objeto que implementa el servicio remoto está inactivo. Si es así, RMI instancia el objeto y restaura su estado. Una vez en memoria se comporta de forma similar al punto anterior.
- Multicast: un proxy puede enviar una petición del método a múltiples implementaciones simultáneamente y validar la primera contestación (esto mejora tiempo de respuesta y mejora la disponibilidad).

3.2.5.5. Capa de transporte.

La capa de transporte hace la conexión entre todas las conexiones de JVMs que utilizan el protocolo TCP/IP. Incluso si dos JVMs se están ejecutando en la misma máquina, se conectan con el protocolo de red TCP/IP de dicha máquina. El diagrama siguiente muestra el uso de las conexiones del TCP/IP entre JVMs :



TCP/IP proporciona una conexión persistente entre dos máquinas basadas en una dirección IP y un puerto de acceso en cada extremo. Mientras que la capa de transporte prefiere utilizar conexiones múltiples del TCP/IP, algunas configuraciones de red permiten solamente una sola conexión del TCP/IP entre un cliente y un servidor. En este caso, la capa de transporte multiplexa conexiones virtuales múltiples dentro de una sola conexión del TCP/IP.

3.2.5.6. RMIRegistry.

La respuesta implica añadir un nuevo elemento a la arquitectura de RMI, y es que es a través del servicio de nombrado o directorio como los clientes encuentran servicios remotos. Este servicio se ejecutará en un servidor y puerto bien conocidos, de tal modo que se puedan acceder fácilmente a él.

RMI puede utilizar muchos servicios de directorio, pero el más común y simple es el que incluye RMI, llamado el registro de RMI, rmiregistry. El registro del RMI se ejecuta en cada máquina servidor de objetos remotos y acepta las consultas para los servicios, el valor por defecto del puerto es 1099. En la máquina servidor, se crea un servicio remoto creando primero un objeto local que implementa ese servicio y lo exporta a RMI. Cuando se exporta el objeto, se crea un servicio que escucha los requerimientos de los clientes para conectar y solicitar el servicio. Después de exportar, el servidor coloca el objeto en el registro del RMI bajo nombre público.

En el lado del cliente, se accede al registro RMI a través del método lookup de la clase estática Naming; gracias a éste, un cliente podrá preguntar al registro por un determinado servicio. El método lookup acepta un URL que especifica el nombre de la máquina servidor y el

nombre del servicio deseado. El método devolverá una referencia remota al objeto de ese servicio. El URL toma la forma:
rmi://<host_name> [:<name_service_port>] /<service_name>, donde:

- host_name es un nombre reconocido en la red de área local o un nombre DNS.
- name_service_port: solamente será especificado si el servicio de nombre se está ejecutando en un puerto diferente al valor por defecto 1099.

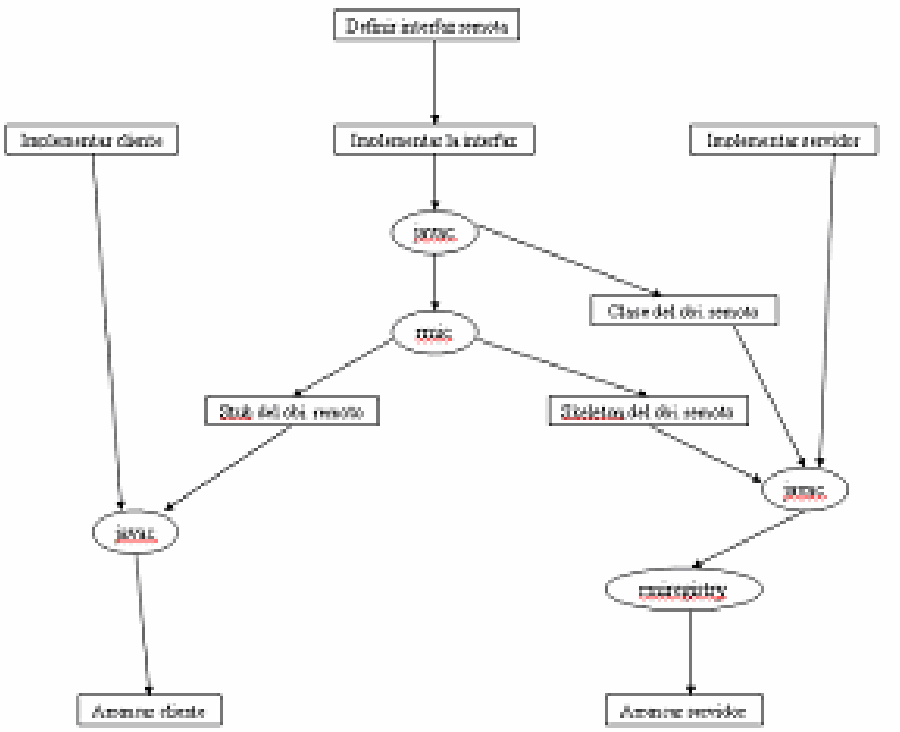
3.2.6 Usando JAVA RMI.

Un sistema RMI se compone de varias partes:

- Las definiciones de interfaz para servicios remotos.
- Implementaciones de servicios remotos.
- Archivos Stub y Skeleton.
- Un servidor de servicios remotos.
- Un servicio de nombre RMI que permita a los clientes encontrar los servicios remotos.
- Un programa cliente que utilice el servicio remoto.

Asumiendo que el sistema RMI ya está diseñado, se necesitan los siguientes pasos para construir el sistema:

- Escribir y compilar código Java para las interfaces.
- Escribir y compilar código Java para las clases de implementación.
- Generar los archivos de las clases Stub y Skeleton desde las clases de implementación.
- Escribir código Java para el servidor remoto (en el computador servidor).
- Desarrollar el código para los programas del cliente RMI.
- Instalar y ejecutar el sistema RMI



3.2.7 Parámetros en RMI.

RMI utiliza llamadas a métodos de objetos remotos, pero el problema es como trata esta tecnología si algunos de estos métodos llevan parámetros o devuelve algún valor. El tratamiento de éstos dependerá de si los parámetros son tipos primitivos de datos, objetos u objetos remotos.

- **Parámetros en una JVM sencilla**

La semántica normal para la tecnología de Java es la de “pasar por valor”. Cuando un parámetro se pasa a un método, la JVM hace una copia del valor y después ejecuta el método utilizando esa copia. Los valores retornados de métodos son también copias; así que, cuando un tipo primitivo de datos (boolean, byte, short, int, long, char, float, o double) se pasa como parámetro a un método, la mecánica de “pasar por valor" es directa. En cambio, la mecánica de pasar un objeto como parámetro es más compleja. Un objeto reside en el heap de memoria y es accesible con unas o más variables referencia.

- ***Parámetros Primitivos***

Los valores se pasan entre JVMs en un formato estándar, independiente de la máquina. Esto permite que JVMs que se ejecutan en diversas plataformas puedan comunicarse unas con otras de forma correcta. Cuando un objeto se pasa a un método remoto, la semántica cambia en el caso de JVM simple. RMI envía el objeto mismo, no su referencia; es el objeto que es “pasado

por valor", no la referencia al objeto. Semejantemente, cuando un método remoto devuelve un objeto, se devuelve al quien ha realizado la invocación una copia del objeto entero.

- ***Parámetros Objetos***

Un objeto de Java puede ser simple y autónomo, o podría hacer referencia a otros objetos Java. Dado que diversos JVMs no comparten heap en memoria, RMI debe enviar el objeto referido y todos los objetos que él referencias. (puede utilizar mucho tiempo de CPU y ancho de banda de red).

RMI utiliza una tecnología llamada Object Serialization para transformar un objeto en un formato lineal que se pueda enviar por la red. La serialización del objeto consiste en aplanar un objeto y cualquier objeto que él referencie. Los objetos serializados se pueden de-serializar en la memoria del JVM remoto y dejarlo operacional para un programa Java.

- ***Parámetros Objetos Remotos***

RMI introduce un tercer tipo de parámetro para considerar: objetos remotos. Un programa cliente puede obtener una referencia a un objeto remoto con el programa registro del RMI.

3.2.8 Distribución de Clases RMI.

Para ejecutar una aplicación RMI, los archivos de las clases que se utilizan deben estar en localizaciones que pueden ser encontradas por el servidor y los clientes. Para el servidor, las siguientes clases deben estar disponibles para su cargador de clase:

- Definiciones de interfaz de servicio remoto.
- Implementaciones del servicio remoto.
- Skeletons para las clases de implementación.
- Stub para las clases de implementación.
- El resto de las clases del servidor.

Para el cliente, las siguientes clases deben estar disponibles para su cargador de clase:

- Definiciones de interfaz de servicio remoto.
- Stub para las clases de implementación del servicio remoto.
- Clases del servidor usadas por el cliente (tal como valores de retorno).
- El resto de las clases del cliente.

3.2.8.1. Carga dinámica.

Java permite cargar stubs que inicialmente no están en el cliente. Esto permite una gran utilidad ya que de este modo se pueden utilizar clientes con código dinámico. El cargador de clases de Java (RMIClassLoader) buscará el stub en la variable Classpath y si no lo encuentra, lo buscará en el parámetro java.rmi.codebase que contendrá la URL donde se encuentra la clase del stub (equivaldría al classpath remoto).

3.2.9 Componentes.

Ahora se analizarán un conjunto de clases e interfaces que proporciona el API y los documentos de ayuda que provee JDK para más tarde poder entender que características deben de tener nuestras clases para tener el comportamiento remoto deseado.

Interfaz Remote del paquete java.rmi

La interfaz Remote sirve para identificar aquellos métodos que pueden ser invocados desde una máquina virtual remota. Cualquier objeto que desee tener un comportamiento remoto, deberá directa o indirectamente implementar esta interfaz. Solo aquellos métodos especificados en una "interfaz remota", cualquier interfaz que extienda de java.rmi.Remote serán válidos remotamente. Las clases de implementación pueden implementar tantas interfaces remotas como se desee y además pueden extender de otras implementaciones de clases remotas.

RMI provee un gran número de clases a implementar remotamente para, de este modo, facilitar la creación de objetos remotos. Estas clases son java.rmi.server.UnicastRemoteObject y java.rmi.activation.Activatable.

Clase UnicastRemoteObject

Esta clase define un objeto remoto "no replicado" cuyas referencias son válidas solo mientras el proceso servidor esta vivo. La clase UnicastRemoteObject class provee soporte para referencias de objetos activos punto a punto usando flujos TCP. Los objetos que requieren un comportamiento remoto deberán extender de las clase RemoteObject, normalmente será mediante UnicastRemoteObject. Si no extienden de ésta, la clase de implementación deberá asumir la responsabilidad sobre la sintaxis del código; de tal modo que estas se tendrán que comportar apropiadamente para los objetos remotos.

Clase RemoteException

La clase RemoteException es la superclase común para un gran número de excepciones relacionadas con las comunicaciones que pueden ocurrir durante las llamadas remotas a métodos. Cada método de una interfaz remota (cualquier interfaz que extienda de java.rmi.Remote) deberá tener una cláusula de tipo throws con esta excepción RemoteException.

Una vez comentados estos 3 componentes del jdk, queda mucho más claro porque en el desarrollo de la aplicación encontramos clases que heredan de alguna de las clases arriba comentadas.

3.2.10 Conclusiones y observaciones.

A continuación se presenta una seria de observaciones y conclusiones obtenidas al estudiar y analizar JAVA RMI:

- Java RMI mantiene características positivas de Java.
    - Programación: orientada a objetos, permite modularidad en el diseño de un programa logrando una implementación conceptualmente clara de la solución a un determinado problema.
    - Robustez: manejo de excepciones.
    - Seguridad: acceso restringido a los objetos permitiendo ocultación de datos y código.
- Separación clara entre cliente y servidor.
    - Interfaz.
    - Implementación.

Ambos lados del servicio, cliente-servidor, están claramente identificados, lo que permite un entendimiento fácil de una solución ya implementada, o una modularización de la solución ideal para trabajos en equipo.

- Arquitectura de capas.

La arquitectura de capas permite el mejoramiento continuo del sistema RMI permitiendo cambiar una capa por una mas actualizada, o por una capa en particular requerida por el entorno donde se implementa la solución.

- Servicio de nombre.

JAVA RMI provee un sistema sencillo de servicio de nombres, liberando al programador de implementar un sistema para la ubicación de objetos.

- Varios tipos de paso de parámetros.

Uno de los aspectos mas relevantes en los lenguajes de programación es el paso de parámetros a funciones, esta acción se hace menos trivial al incorporarla a un sistema distribuido. JAVA RMI incorpora las semánticas necesarias para hacer transparente al programación la distribución de los objetos y los correspondientes pasos de parámetros a los métodos.

- Llamadas a cliente.

Si bien, JAVA RMI, provee una clara separación entre cliente y servidor, permite que los procesos puedan intercambiar roles en el caso de que deba realizarse una petición de servidor a cliente, evitando de esta manera la necesidad de crear nuevos procesos para realizar esta tarea.

3.3 Tecnología CORBA.

En este capítulo se presentará la tecnología CORBA, y se comentarán los motivos que han llevado a ésta a ser actualmente la mejor tecnología existente para desarrollo de aplicaciones Cliente/Servidor distribuidas.

CORBA (Common Object Request Broker Architecture), es decir, Arquitectura de Bus Común de Gestión de Peticiones de Objetos es un marco de trabajo estándar de Objetos Distribuidos creado por el OMG (Object Management Group), un grupo de más de 760 empresas que se creó en 1989 con fundadores de gran relevancia como son Hewlett-Packard ó Sun Microsystems. Esta es una organización sin ánimo de lucro cuyos fines son promover la Orientación a Objetos en la ingeniería del software y el establecimiento de una plataforma arquitectural común para el desarrollo de programas basados en Objetos Distribuidos.

Lo primero que hizo este grupo fue definir la OMA (Object Management Architecture), que es la arquitectura donde se recoge toda la la aportación tecnológica del OMG. Esta arquitectura incluye:

- Un modelo de objetos base, que define los elementos básicos de la programación Orientada a Objetos (clases, objetos, implementación, interfaces, invocación, cliente, servidor, etc.).

- Un modelo de referencia, que está formado por cinco componentes:
    - Un Bus común de gestión de peticiones y respuestas entre objetos en el que puedan ser integrados componentes de forma estándar. Este bus lo han llamado ORB (Object Request Broker)
    - Servicios de Objetos (Object Services). Conjunto de servicios disponibles para los objetos. Destacan los servicios de nombrado, eventos, ciclo de vida, transacciones, etc.
    - Servicios Comunes (Common Facilities). Especifican servicios comunes a todos los objetos, como documentos compuestos, facilidades de Agentes Móviles, de Manejo de Sistemas, etc.
    - Interfaces de Dominio. Son frameworks específicos para dominios de desarrollo, como por ejemplo, programas médicos, control del tráfico aéreo, etc. Todos ellos, al igual que los anteriores, especificados como interfaces.
    - Interfaces de Aplicación. Estos son los interfaces que constituyen las aplicaciones específicas desarrolladas.

Dado este marco, podremos ver como CORBA, al extender las definiciones y conceptos dadas por el "modelo de objetos base" utilizando los mismos conceptos, pero más específicos y concretos, es un refinamiento de este modelo. Establece también la diferencia entre "Implementación de Objetos" y "Referencias de Objetos", la semántica de los interfaces, la semántica de las llamadas a operación, excepciones… sin olvidar tampoco el lenguaje de especificación de interfaces: IDL (Interface Definition Languaje), que será un lenguaje de especificación independiente del lenguaje de implementación utilizado.

3.3.1 Beneficios del uso de CORBA con Java.

Las ventajas de su utilización se verán desde varias dos perspectivas: en primera instancia se estudiarán las ventajas qué ofrecen otros lenguajes de script a CORBA, comparándolos con Java. Seguidamente, se hará un estudio de las características que ofrece Java a CORBA y viceversa; con este último, y uniendo sus dos partes veremos como al combinar estas dos tecnologías se mejora con creces las características de las otras existentes.

3.3.1.1 Comparación con otros lenguajes de script.

Actualmente hay otros lenguajes que comparten con Java muchas de sus características y que están tan capacitados como éste. La única razón por la que Java es popular es porque está

diseñado y pensado para desarrollar programas que se ejecutan en el navegador. Pitón se ha utilizado para construir un navegador con características similares y puede ser considerado un competidor de Java si no fuera por la falta de marketing, es decir, la gran popularidad de Java es debida a su simbiosis con los browsers de la Web. Sin embargo, otros lenguajes como Python o Perl que ofrecen características similares (también son orientados a objetos, compilan a bytecode, etc.) son serios competidores desde el punto de vista de la adaptación que estos lenguajes exhiben a la hora de escribir aplicaciones distribuidas.

Un punto importante en el que éstos fallan es en la cuestión de la seguridad. La mayoría de estos lenguajes permiten que el código del cliente se distribuya, bien en forma de bytecodes (como Java, Python), bien en forma de código fuente (Tcl, Perl). Esto requiere que se pueda garantizar una seguridad en el sentido de que cualquier trozo de código que se descargue en nuestro ordenador cliente no va a ocasionar la pérdida malintencionada de información.

3.3.1.2 Ventajas proporcionadas a CORBA por Java.

La portabilidad y la ya comentada buena integración con los navegadores de Internet le hacen un lenguaje que tiene mucho que ofrecer a la programación distribuida y a CORBA. Sin embargo, aparte de su inmenso plan de márketing y el respaldo de compañías como Sun Microsystems, las características más sobresalientes de Java son ahora compartidas por varios lenguajes de nueva generación, como Python o Perl.

A continuación se muestran una serie de ventajas que Java ofrece a la tecnología CORBA:

- Portabilidad de las aplicaciones entre distintas Plataformas. Una de las principales características de Java es que es a la vez compilado e interpretado. La portabilidad entre plataformas se consigue dividiendo el proceso de compilación en dos fases: primero la compilación hacia un código de una máquina virtual (bytecode); y después la interpretación de esos bytecodes teniendo como resultado la ejecución real. La ventaja es que el bytecode es independiente de la plataforma, y para cada una de ellas, se puede escribir un intérprete de bytecodes específico. El código en bytecodes puede ser pasado de una forma portable entre las distintas plataformas.

  Concretamente para CORBA, esto significa, por ejemplo, que el código de los servidores puede ser trasladado de un host a otro, dependiendo de las características de la aplicación; y no sólo sin cambiar los clientes al mantener intacto el interfaz (como

permite CORBA por si mismo), sino sin tener que recompilar ni adaptar el código al nuevo host, que posiblemente tendrá una plataforma [Hardware + Sistema Operativo] distinta. También significa que los clientes CORBA pueden viajar en forma de applets Java y ejecutarse en plataformas clientes distintas, con el único requisito de que soporten una máquina virtual Java.

- Programación para Internet. Los clientes CORBA se pueden desarrollar como applets que se pueden integrar de forma directa en los navegadores. Esto proporciona a la tecnología CORBA un acceso inmediato al mundo de Internet.

- Simplifica (o elimina) la gestión de la configuración en la parte del cliente. Otra vez, utilizando applets como clientes CORBA, el coste de instalación de un nuevo cliente para una aplicación, o el coste de un cambio en el cliente no implica coste alguno en términos de reconfiguración del cliente: simplemente el cliente debe descargar del servidor la nueva versión del applet. La gestión de la configuración e instalación se convierte en un proceso muy complicado para sistemas grandes Cliente/Servidor distribuidos.

- Un lenguaje de programación Orientado a Objetos relativamente sencillo y legible. Java fue diseñado entre otros objetivos para mejorar la simplicidad de sintaxis del lenguaje. Tomando un lenguaje ampliamente aceptado, como es C++, y eliminando ciertas características engorrosas, como la herencia múltiple, desarrollaron un lenguaje de sintaxis clara, orientado a objetos y seguro.

- Programación multihilo (Multithreading). Java soporta de forma integrada en su máquina virtual el uso de varios hilos de ejecución simultáneos, proporcionando monitores para conseguir exclusión mutua. Esto hace muy fácil el proceso de construir servidores que atienden a varios clientes de forma simultánea.

- Simplifica la recolección de basura (Garbage Collection). Uno de los principales problemas en los sistemas distribuidos es la recolección de basura, esto es, la eliminación de los objetos a los que ningún cliente referencia. En C++, por ejemplo, el programador debe programar de forma específica cuándo y cómo sus objetos son destruidos. En Java esto es automático, y por lo tanto, la programación se hace más sencilla.

3.3.1.3 Ventajas proporcionadas a Java por CORBA.

Lo que CORBA ofrece a Java puede resumirse en los siguientes puntos:

- Interfaces definidas utilizando IDL. La separación entre interfaz e implementación en CORBA está implícita, y se consigue definiendo todos los componentes de la aplicación e incluso los servicios genéricos disponibles utilizando un lenguaje de descripción independiente de la implementación: IDL.

- Un sistema de meta-información. Un objeto CORBA puede acceder a toda la (meta)información sobre los interfaces que definen los demás objetos que están presentes en el sistema. Esto permite a los objetos localizar servicios genéricos o comunicarse y conocer otros objetos nuevos que se van integrando en el sistema dinámicamente. Esto hace muy sencillo el proceso de desarrollo y adaptación de las aplicaciones ante cambios en la especificación

- Permite que las peticiones se generen de forma dinámica. Al poseer la metainformación anteriormente comentada, los clientes pueden actuar de una manera que no se pensó en el momento de su creación, ejecutando métodos de forma dinámica en objetos que ni siquiera se conocían en el momento de la compilación del primero.

- Independencia del lenguaje de programación utilizado. Al contrario que RMI, por ejemplo, CORBA ofrece ligaduras con C, C++, COBOL, Smalltalk, Java, etc., que permiten que los métodos definidos en el IDL sean implementados utilizando cualquiera de estos lenguajes. Los clientes de estos objetos no son capaces de distinguir en qué lenguaje fueron implementados los objetos que les proveen servicio. Esta independencia no sólo ayuda a que cada parte del sistema sea implementada en el lenguaje adecuado para su funcionalidad, sino que permite la integración de viejos sistemas existentes en las integrándolos de forma transparente en un sistema de Objetos Distribuidos estándar.

- Transparencia de localización y activación del servidor. El corazón de CORBA (el ORB) ofrece a los objetos un mecanismo por el que pueden realizar invocaciones sobre objetos remotos de forma que éstas aparezcan ante el programador como locales. El ORB se encarga de alcanzar los objetos servidores reales y enrutar la petición en beneficio del usuario. Esto descarga a la aplicación de todo el manejo de la comunicación, dando la visión al programador de que trabaja con un sistema grande sin importarle el tipo y numero de redes que se incluyen.

- Generación automática de código Stub y Skeleton. Los sistemas distribuidos requieren mucha programación de bajo nivel para manejar el inicio, flujo y finalización de la comunicación, codificar y decodificar argumentos en el formato en el que se transmitirán, etc.; estos son los stubs del cliente y skeletons del servidor. A través de los compiladores de IDL, el código que realiza todas estas funciones se crea automáticamente. Un cliente de un objeto sólo necesita su IDL para construir de forma automática el código que le permitirá realizar invocaciones remotas de forma transparente. Tanto este punto como el anteior también se conseguían con RMI.

- Reutilización de CORBAServices y CORBAFacilities. Los objetos que forman una aplicación distribuida normalmente requieren una serie de servicios adicionales. Estos servicios, en CORBA forman parte de lo que se conoce como CORBAServices y CORBAFacilities. Los servicios disponibles y más comunes que se detallarán mas adelante son: servicio de localización de objetos, servicio de eventos, servicio de transacciones, de seguridad, etc. Estos servicios están disponibles para ser reutilizados una y otra vez en las aplicaciones, liberando así a los desarrolladores de la carga de tener que implementarlos.

- Independencia del fabricante a través de la interoperatividad de los ORBs y la portabilidad de código. CORBA define un protocolo estándar a través del que varios ORBs de distintos fabricantes se pueden comunicar de forma estándar (GIOP, General Inter-ORB Protocol). Esto significa que cualquier ORB que sea compatible con CORBA puede ser accedido desde cualquier otro. El desarrollador puede utilizar el ORB del fabricante que mejores prestaciones de para una plataforma o lenguaje específico, sabiendo que, hacia el exterior, sus objetos van a presentar un interfaz uniforme y compatible.

3.3.2. Introducción a CORBA.

CORBA es una arquitectura estándar para el desarrollo de aplicaciones distribuidas basadas en Objetos. Permite que las clases que forman parte de las aplicaciones puedan ser implementadas en distintos lenguajes, se ejecuten en distintas plataformas o estén dispersas por una red heterogénea. Para conseguir esto, CORBA se centra en tres ideas clave.

La primera de ellas es la separación entre interfaz e implementación; y es que, a través del lenguaje de definición de interfaces (IDL) se especifican todos los componentes CORBA. Es independiente del lenguaje utilizado en la implementación, existiendo ligaduras para diversos lenguajes de programación (C, C++, Java, Ada, Smalltalk, COBOL, etc). Permite especificar las clases de las que un componente hereda, la signatura de operaciones, los atributos, las excepciones que lanza, y la signatura de métodos (incluyendo argumentos de entrada, de salida y valores de retorno y sus tipos de datos), etc.

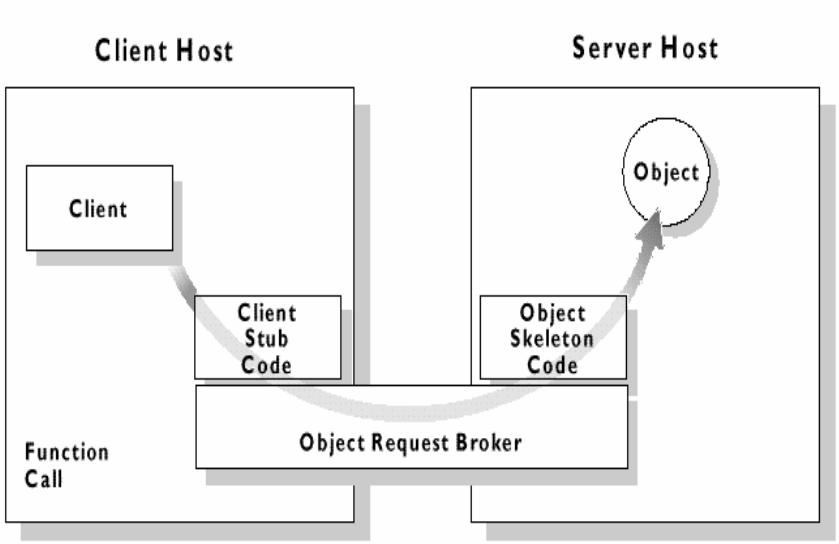
La segunda idea fundamental es la independencia de localización. El ORB, que es el componente más importante de cualquier implementación CORBA, se encarga de hacer transparente la localización de los objetos, enrutando las peticiones de manera que un objeto pueda comunicarse con otros independientemente de si ambos objetos se ejecutan en la misma máquina o en otras a través de redes heterogéneas.

La independencia del fabricante y la integración de sistemas a través de la interoperatividad también es otra idea muy importante en la que se centra CORBA. El protocolo GIOP (General Inter-ORBProtocol), que es un estándar para que ORB's de distintos fabricantes puedan integrarse, y su especifico para Internet, IIOP, permiten a ORB's de distintos fabricantes comunicarse de una manera estándar. Esta característica ofrece dos grandes ventajas: la independencia del fabricante del ORB y una invocación de métodos independiente de si ambos objetos (cliente y servidor) están o no en el mismo ORB.

3.3.3. El ORB (Object Request Broker).

La parte más importante del OMA es el ORB. El ORB es la única parte de CORBA que debe estar presente cuando vayamos a crear una aplicación en CORBA. Muchos ORBs funcionan sin los servicios (CORBAServices) ni facilidades (CORBAFacilities), pero siempre tiene que estar presente el ORB, sino una aplicación de CORBA no podría funcionar.

La función más visible del CORBA ORB es responder a las demandas de tu aplicación o desde otro ORB. En la siguiente figura vemos de forma muy simple para que sirve:



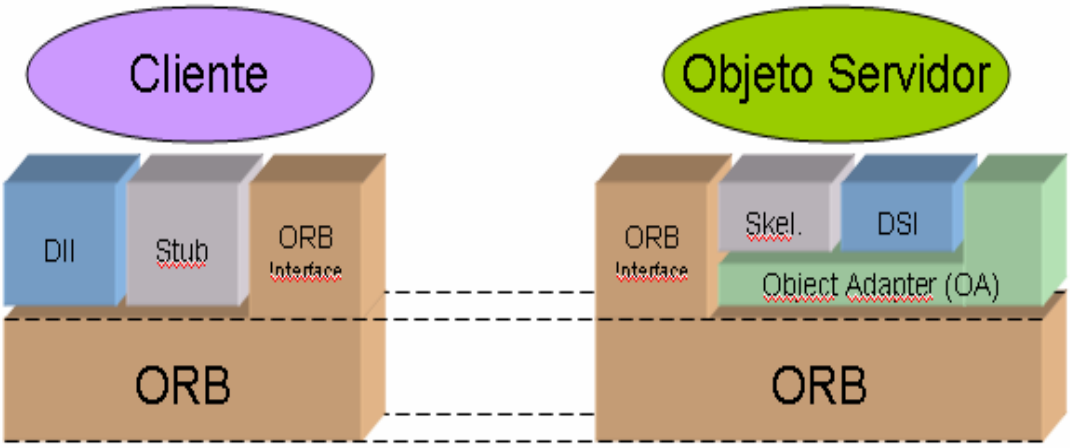
Durante el ciclo de vida de la aplicación CORBA que estamos usando, al ORB se le puede pedir que haga varias cosas, entre ellas:

- Buscar e implementar a los objetos en las maquinas remotas.
- Ordene los parámetros de un lenguaje de programación, como C++, a otro lenguaje, como Java.
- Manejar la seguridad en los límites locales de la máquina.
- Recuperar y publicar metainformación en los objetos en el sistema local para otro ORB.
- Invocar los métodos de los objetos remotos.
- Invocar los métodos de los objetos remotos usando los métodos dinámicos de invocación.
- Puesta en marcha automática de los objetos que no están en ese momento funcionando.
- Enrutado a las llamadas de regreso de los métodos al objeto local apropiado que se esté manejando.

El proceso de invocación es similar al de RMI, es decir, que también se divide en dos pasos: en el primero de ellos se produce la obtención de la referencia del objeto remoto; tras esto se da la invocación de la operación propiamente dicha y el acceso a los atributos de este objeto, definidos a su vez a través del IDL de la clase a la que este último pertenece.

3.3.4. Componentes del ORB

La figura siguiente muestra la estructura del ORB de CORBA.



En el esquema podemos localizar todos los componentes y observar las interfaces que el ORB ofrece a todos los objetos CORBA. Como se ve ofrece un conjunto de servicios comunes (bajo el nombre de ORBInterface) que pueden ser utilizados tanto por objetos clientes como por implementaciones. Entre las operaciones ofrecidas por el interfaz del ORB se encuentran la posibilidad de convertir referencias a objetos en cadenas de caracteres y viceversa para poder comunicar de manera sencilla referencias a objetos, obtener la clase de un objeto, etc.

Como en RMI, tanto los clientes como los servidores necesitan unos adaptadores que transformen, en la parte cliente, una invocación local a una petición al ORB; y, en la parte servidor, una invocación por parte del ORB en una invocación en el objeto implementación (o servidor) real. En la parte cliente, estos adaptadores se llaman Stubs y en la parte servidor, Skeletons.

También se provee de interfaces para construir invocaciones de forma dinámica; para ello se necesita de la DII (Dynamic Invocation Interface) en la parte cliente y de la DSI (Dynamic Skeleton Interface) en la parte del objeto implementación. De este modo, se provee una interfaz para que los objetos puedan invocar (y recibir) llamadas a operaciones para las que no poseen stubs, especificando el objeto, el nombre de la operación que se invocará sobre él y los parámetros de la llamada. Estas facilidades dinámicas se apoyan en la meta-información dada por el Interface Repository (IR). Finalmente, el adaptador de objetos (Object Adapter) se apoya en el Implementation Repository para controlar el registro de servidores, activación y desactivación de implementaciones, instanciación en base a varias políticas, etc.

A continuación se explicarán en detalle cada uno de estos componentes.

3.3.4.1 El interfaz del ORB.

El interfaz del ORB ofrecen los pocos servicios que son de utilidad tanto para los objetos clientes como para los servidores. Mediante las operaciones “string_to_object()” y “object_to_string()”, se nos permite transmitir referencias a objetos en forma de cadenas de caracteres. También ofrece otras:

- Operaciones de manejo de referencias:
  - is_nil(); para comprobar que una referencia es no nula.
  - duplicate(); crea copias de un objeto.
  - release(); destruye copias de un objeto.
- Funciones de meta-información:
  - get_interface(); para obtener el tipo de un objeto.
  - is_a(); informa de si un objeto “puede ser visto” como de un tipo específico, es decir, si pertenece a una clase que hereda de cierta otra.
- Funciones para que los objetos consigan las referencias que necesitan:
  - ist_initial_services(); para obtener referencias a los objetos que implementan servicios adicionales del bus, por ejemplo: el servicio de nombrado, el de transacciones, el Repositorio de Interfaces, etc.

3.3.4.2. Los stubs del cliente

Recordando los pasos para realizar una invocación sobre un método de un objeto remoto, el primero de ellos era la obtención de una referencia de este objeto. Una vez obtenida ésta, el objeto puede invocar operaciones sobre esa referencia como si el objeto servidor fuera local (aunque éste no lo sea). El cliente necesita ser linkado con el Stub para que de este modo, se permita que las invocaciones que éste realiza de forma local se transformen en peticiones de ejecución de un método sobre el objeto remoto. El papel del stub en CORBA sigue la misma idea que tomaba para la tecnología RMI; el stub transformará los parámetros de las peticiones a un formato que se pueda transmitir. A este proceso se le conoce como marshalling (en RMI se llamaba serializing).

En el objeto cliente, existe una operación (y un atributo) equivalente para cada una de los definidos en el IDL de la clase a la que pertenece. Los stubs son también llamados objetos

proxy. Los stubs son generados por el compilador de IDL para el lenguaje y el ORB específico que esté utilizando el cliente.

3.3.4.3 La interfaz de invocación dinámica (DII).

Si se quiere utilizar siempre invocación estática, todo cliente debe ser linkado con todos los stubs de los objetos servidores de los que pide servicios. Pero hay que destacar que CORBA ofrece otro mecanismo de invocación más dinámico: el DII. Con este interfaz, un objeto cliente puede invocar cualquier operación en objetos aunque no posea stubs.

Una vez que ha obtenido la información necesaria sobre el objeto sobre el que quiere realizar la invocación (como por ejemplo, saber sus métodos, número de argumentos de cada uno y tipo, etc.) puede construir un objeto del tipo Request (Petición) que encapsula toda la información sobre la invocación de un método a un objeto y la respuesta que este devuelve (incluyendo las posibles excepciones que se lanzan durante la ejecución). Esto, junto con el sistema de meta-información, hace a CORBA ideal para entornos dinámicos, en los cuales, se incluyen en el sistema nuevos componentes en tiempo de ejecución y en los que los componentes se descubren unos a otros y son capaces de trabajar en organizaciones que no fueron pensadas en el momento de la creación de ninguno de ellos de forma independiente.

Cabría destacar, que el servidor no distingue si su cliente lo invoca desde un stub estática o utilizando el interfaz dinámico.

3.3.4.4 Los skeletons del servidor.

Una vez que llega la petición del cliente al servidor, ésta debe llegar al método correcto del método correcto; además, se deben traducir los parámetros a estructuras de datos legibles por el objeto. Estas son las acciones que los skeletons deben realizar. Así, su función será la de transformar las peticiones que el ORB realiza (que son a su vez producidas por invocaciones de clientes, ya sea estática o dinámicamente) en invocaciones sobre el objeto servidor o implementación real, además de retornar de vuelta el resultado de la invocación.

Como se puede comprobar los skeletons son para los servidores como los stubs para los clientes. Además, al igual como ocurría con los anteriores, los skeletons también son generados por el compilador IDL

3.3.4.5 La interfaz de skeletons dinámicos (DSI).

Esta interfaz permite a los servidores responder a peticiones de invocación de forma dinámica. Esto significa que el servidor puede responder a peticiones interpretando en tiempo de ejecución el método invocado, los argumentos y su tipo y dándole semántica dinámicamente. Al igual como la DII estaba íntimamente ligada a los stubs, del mismo modo y con las mismas características, la DSI está relacionada con los skeletons.

3.3.4.6 Comunicación entre ORBs: El protocolo IIOP.

El ORB se encarga de enrutar las invocaciones de un objeto cliente a un objeto servidor de forma transparente independientemente de si ambos están en el mismo ORB o en distintos, posiblemente de distintos fabricantes. Para conseguir esto, se necesita un estándar de interoperatividad entre ORBs. Esto lo da el protocolo GIOP (General Inter-ORB Protocol) y su específico de Internet, IIOP, que definen el conjunto de mensajes y la codificación a nivel del cable que se utiliza para comunicar ORBs.

Esta interoperatividad obliga a que las referencias a objetos dentro de un ORB se conviertan en referencias universales al comunicarse con otros ORBs. Estas referencias universales son las que se comunican entre ellos; son llamadas IORs (Interoperable Object References, Referencias Interoperables de Objetos). Una IOR contiene toda la información que permite a un ORB dirigir las peticiones y las respuestas hacia un objeto que reside en otro ORB.

3.3.4.7 El adaptador de objetos (OA, Object Adapter).

El adaptador de objetos se encarga del manejo de la implementación de los objetos. Así, las funciones del OA (tareas conjuntas al ORB Core y otros componentes) son:

- Generar e interpretar referencias de objetos CORBA.
- Invocar operaciones.
- Activar y desactivar objetos e implementaciones.
- Registrar implementaciones.
- Asociar referencias de objetos a implementaciones.
- Autentificación

El estándar CORBA define dos adaptadores: el BOA (Basic Object Adapter), genérico, obsoleto y caracterizado porque cada desarrollador de ORB definía su propia interfaz BOA, por

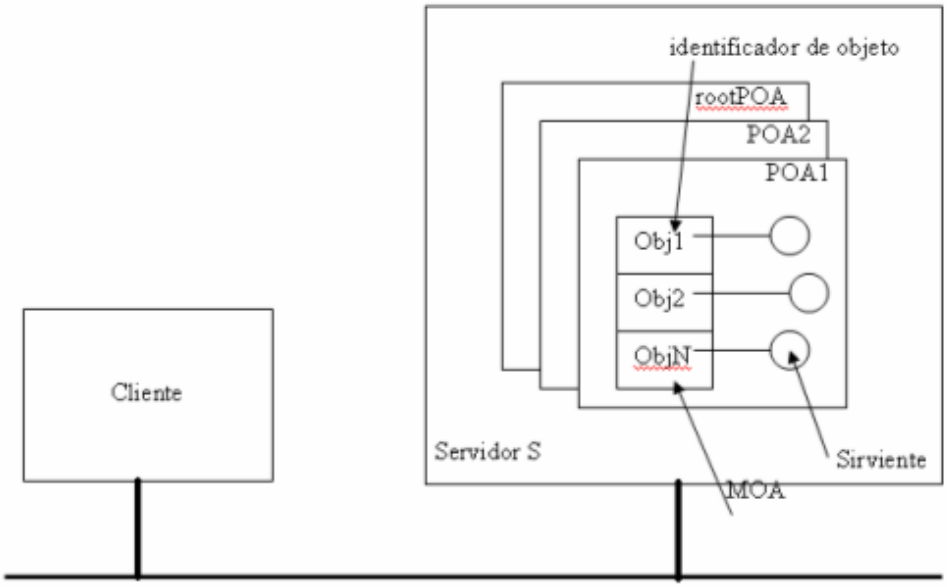
lo tanto, el código del servidor no era reutilizable. El otro adaptador es POA (Portable Object Adapter), que es una estandarización más rígida, portable, mucho más utilizada y que se caracteriza porque la implementación es invariable entre ORBs.

Una vez que ya se ha visto que el adaptador POA es el más utilizado hay que destacar una serie de conceptos muy importantes para entender el funcionamiento del adaptador:

- Sirviente: código que implementa uno o más objetos CORBA.
- ObjetoID: identificador de un objeto CORBA, único para un POA.
- Mapa de objetos activos (MOA): tabla que asocia sirvientes con ObjetoIDs.
- Encarnación: acción de asociar un sirviente con un ObjetoID.
- Eternizar: acción de destruir la asociación entre sirviente y ObjetoID.
- Sirviente por defecto: objeto que "despacha" las peticiones que no tienen el ObjetoID en el Mapa de Objetos Activos.

El proceso para localizar un objeto con POA es muy sencillo a partir del esquema de la parte inferior. Será necesario introducir todos los campos que aparecen en la parte superior del esquema, que representa la IOR de un objeto (referencia remota a un objeto de un servidor determinado). El primer campo (IDLx) representa el identificador de interfaz, seguidamente va IIOP que es el protocolo utilizado para enviar las peticiones, en el ejemplo, es una implementación de GIOP para TCP/IP. El tercer parámetro representa el servidor donde se encuentra nuestro objeto. Una vez dentro de este servidor, para poder localizar el objeto deseado, tendremos que mirar el 4º parámetro que determina el adaptador donde está "registrado" el objeto. En este POA, mas exactamente en su MOA (tabla de objetos activos), debe haber una fila donde se asocia la referencia del objeto que buscamos con su implementación (ruta del servidor donde se encuentra ésta).

IDLx	IIOP	S	POA1	Obj1



3.3.4.8 El Repositorio de Interfaces (IR).

El Repositorio de Interfaces es una base de datos que contiene la descripción de la interfaz (escrita en IDL) de los objetos, de manera que ésta es accesible en tiempo de ejecución. Esta meta-información puede ser utilizada por los clientes para construir invocaciones dinámicas. Este repositorio también es un lugar común donde se puede guardar información adicional sobre los interfaces, como información de depuración, etc.

Esta base de datos es actualizada por el compilador de IDL , y ofrece al programador un conjunto de clases que describen la (meta-)información que ésta posee y que permiten obtener esta información de una manera jerárquica.

3.3.4.9 El Repositorio de Implementaciones (IM).

Este repositorio es opcional y va a contener información que permite al ORB localizar las implementaciones de los objetos. Esta información suele ser la del control de políticas, la información de instalación, y también otras informaciones adicionales, como información de depuración, control administrativo, reserva de recursos, seguridad, etc. Así, de este modo, si uno de los objetos servidores cae, el repositorio de interfaces se encargará de volver a activarlo.

El IM será necesario si el tiempo de vida del objeto es mayor que el tiempo de vida del servidor; en este caso los objetos serán persistentes. En cambio, si el tiempo de vida del objeto es igual al tiempo de vida del servidor, los objetos serán transitorios y el IM no será necesario.

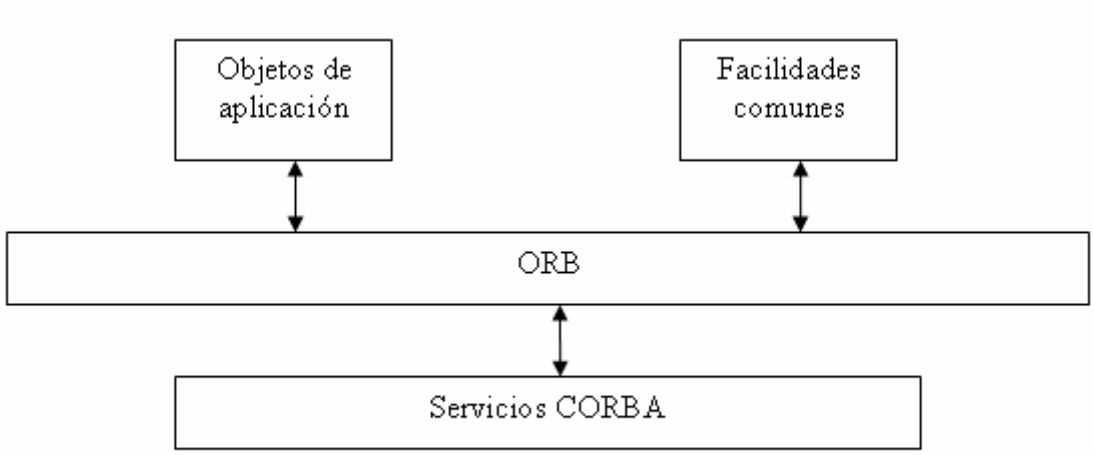
A partir de este comentario, resulta muy comprensible, que el IOR (recordad que era una referencia remota) de un objeto transitorio apuntará a uno de los adaptadores de un determinado servidor (como ya se vio en el esquema anterior), y el IOR de un objeto persistente primero apuntará al IM, para ello los campos host y puerto en vez de ser del servidor, serán del IM. Desde éste, después se obtendrá la ruta donde se encuentra el objeto deseado.

3.3.5 La cuestión de la interoperatividad.

Una de las características más aclamadas de CORBA es la posibilidad de integrar de forma estándar ORBs de distintos fabricantes. Esta capacidad se consigue a través de los protocolos GIOP y sus derivados, como IIOP. Cada referencia a un objeto debe ser convertida a un string a través del métodos del interfaz del ORB_object_to_string() de manera que esta referencia identifique al objeto de forma universal. Estas referencias son lo que hemos venido llamando IORs. Esto significa que cada IOR debe contener la dirección del host donde reside su ORB y una referencia interna en ese ORB.

3.3.6. CORBAServices y CORBAFacilities.

Esta sección trata de ser un pequeño ejemplo de cómo pueden ser utilizados los servicios añadidos al ORB en forma de IDL. Estos servicios están definidos como parte de los CORBAServices y CORBAFacilities. Aunque de cara al ORB todos ellos presentan un conjunto de interfaces, existen una serie de programas de apoyo específicos de la implementación concreta del ORB. Esto no significa que al utilizar los servicios de un vendedor específico quedemos ligados a él: el OMG ha definido muy bien estos servicios de forma estándar, por lo que el código puede ser transportado sin casi modificación entre distintos ORBs. Además, siempre queda la posibilidad de la interoperatividad entre ORBs.



El conjunto de servicios estándar es grande y sigue creciendo. Entre ellos podemos nombrar:

- Nombrado: El servicio de nombres permite ligar objetos a nombres de forma relativa a un contexto jerárquico.
- Eventos: Con el servicio de eventos se permite que distintos objetos se registren como interesados en distintos eventos que se producen en el sistema, bien lanzados por algún objeto como parte de su funcionalidad o de forma espontánea. El diseño de este servicio no requiere de ningún servidor centralizado, y está especialmente adaptado para ambientes distribuidos.
- Ciclo de Vida: Provee servicios de copia, movimiento y eliminación de grafos de objetos relacionados.
- Servicio de Persistencia: Provee servicios de mantenimiento de objetos persistentes. El diseño permite que varias implementaciones coexistan en un mismo ambiente, ya que es posible, por ejemplo, que los requerimientos de almacenamiento para guardar documentos no sean los mismos que para guardar Bases de Datos.
- Relaciones: Permite establecer relaciones entre objetos. En particular, se añaden los tipos de objetos “relación” y “rol”. Un rol representa a un objeto que toma parte en una relación. Se pueden añadir atributos específicos de roles y de relaciones, además de cardinalidades, etc. que serán comprobadas en tiempo de ejecución.
- Externalización: Este es el equivalente a la “Serialización” Java. Ofrece estándares e interfaces que permiten convertir a objetos en streams de datos y viceversa.
- Transacciones: El servicio de transacciones ofrece a los objetos la semántica de una transacción atómica: las acciones entre el inicio y el final de la transacción se realizarán todas o no se realizará ninguna.
- Control de la Concurrencia: Este servicio permite a varios clientes organizar su acceso concurrente a recursos compartidos.

- Licencias: El servicio de Licencias permite a los productores controlar el uso de su propiedad intelectual.
- Consulta: El servicio de consulta permite realizar consultas en conjuntos de objetos. Estas consultas están realizadas en un lenguaje declarativo y pueden indicar valores de atributos, invocar operaciones, etc.
- Propiedades: Permite asociar a los objetos un conjunto de propiedades que pueden ir modificándose en tiempo de ejecución. Las propiedades son conjuntos de pares <nombre; valor>. Los nombres son cadenas de caracteres, y los valores, objetos del tipo Any de CORBA.
- Seguridad: Todos los temas relacionados con la seguridad, incluyendo identificación, autentificación, autorización, encriptación, etc.
- Tiempo: Permite a un conjunto de objetos obtener el tiempo junto con un error asociado. Esto permite ordenar eventos que se producen en el sistema.
- Colecciones: Provee un mecanismo uniforme de creación y manejo de colecciones de objetos.
- Trading: Ofrece un servicio de páginas amarillas para los objetos, en el que se puede buscar objetos por funcionalidad, por tipos de servicios, calidad de servicio (QOS), etc. Este servicio es uno de los más flexibles y potentes actualmente disponibles por algunas implementaciones.

Esto da una idea del tamaño y ámbito de los servicios definidos para los objetos y, en general, del esfuerzo que el OMG está realizando para conseguir un conjunto de estándares de calidad.

3.3.7 Java, CORBA y el Web.

En este punto veremos cómo Java y CORBA se integran con el Web, ofreciendo una serie de ventajas que superan con mucho a las ofrecidas por la tecnología dominante en Internet: CGI. El primer punto a destacar es que CORBA es mucho más adecuado para el desarrollo de aplicaciones distribuidas que CGI, ya que calculando tiempos medios de invocación de estas dos tecnologías, observamos que CORBA es casi un 200% más rápido que CGI, siendo CORBA solo superada por los sockets a bajo nivel. En definitiva, CORBA nos permite programar para conseguir funcionalidad, independientemente de en qué lenguaje, host o plataforma hardware esté implementada: siempre está lista para usar.

Es cierto que cualquier aplicación CORBA escrita en Java se puede transmitir hacia una máquina cliente y ejecutarse allí (siempre que ésta tenga un ORB que de soporte a una funcionalidad CORBA de la aplicación), sin embargo, en todos los demás casos, no se puede estar seguro de que las aplicaciones CORBA escritas en Java que se ejecutarán en la máquina cliente no son malintencionadas. Por lo tanto, se requiere una seguridad similar a la garantizada por los applets: prohibición de escritura en el sistema de ficheros local, prohibición de uso directo de dispositivos de la máquina cliente, imposibilidad de conexión por red a una máquina distinta de la que se descargó el applet, etc.

El uso de applets ofrece además la ventaja de integrarse perfectamente con los navegadores, y por ello, con la Web. Nótese, sin embargo, que los applets se ejecutarán en el ambiente proporcionado por el navegador, con lo que éste no sólo debe ser compatible con Java y ser capaz de ejecutar bytecodes, sino que debe ser compatible con CORBA y contener un ORB. Los ORBs Java van todavía más allá eliminando la necesidad de que el browser contenga un ORB, ya que implementan el ORB completamente en Java. Así, un browser que sea compatible con Java, a la vez que realiza la petición del applet puede realizar la petición de las restantes clases necesarias, siendo ahora el único requisito expuesto al cliente es que posea un browser compatible con Java.

3.4 RMI Vs. CORBA.

Pero, ¿porque utilizar RMI y no utilizar CORBA? Sin duda, es casi obligatorio realizar una comparativa entre ambas tecnologías debido a que, efectivamente, podemos elegir entre las dos para implementar nuestras aplicaciones distribuidas. Numerosos artículos cubren esta discusión y prácticamente llegan a una misma conclusión: si nuestra aplicación distribuida se desarrolla totalmente en Java, utiliza RMI.

Por otro lado, CORBA es una tecnología asentada con el paso de los años y con un sólido soporte de muchas empresas. Ciertamente RMI no esta en la misma situación y si elegimos utilizarla debemos fiarnos de la "promesa" de SUN de seguir soportando la tecnología RMI, mejorándola e incrementándola, de forma que nuestras aplicaciones podrán seguir funcionado en el futuro. En cualquier caso, el objetivo público de SUN es hacer compatibles ambas tecnologías, quizás de la mano de IIOP (Internet Inter-Orb Protocol).

Sin tener en cuenta las posibles similitudes y diferencias en su diseño y arquitectura, la verdad es que tanto la tecnología RMI como CORBA tienen procesos de compilación y

ejecución muy similares; y este hecho es muy comprensible ya que no hay que olvidar que ambas realizan lo mismo.

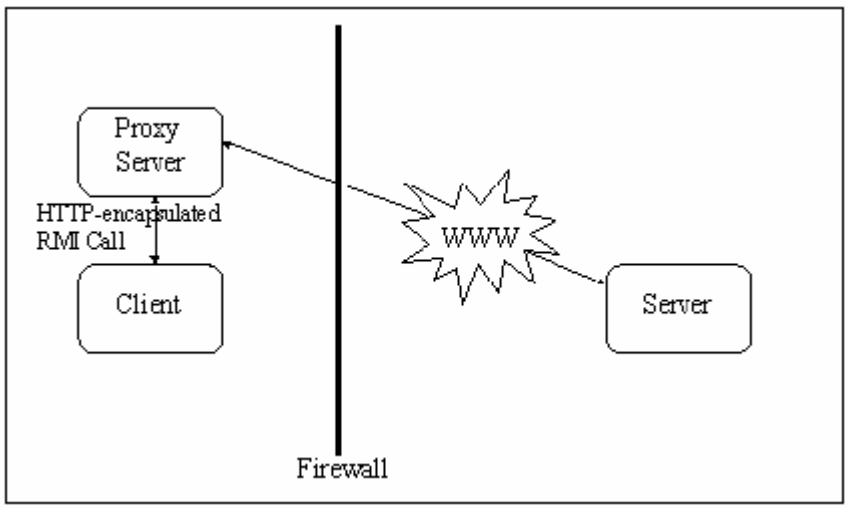
Como ya se ha comentado, una vez creado todo el código necesario, el primer paso consiste en crear el proxy y el stub a partir de las interfaces “remotas” creadas. Así para RMI se utiliza la utilidad “rmic” para crear estas clases, mientras que para CORBA se utiliza “idlj”; no hay que olvidar que las interfaces en esta última tecnología están escritas en IDL, mientras que en RMI están en IDL. El siguiente paso será el de la activación de los elementos que se encargarán de “almacenar” cada uno de estos objetos remotos: así, para RMI se ejecuta la utilidad “rmiregisty” y para CORBA se usa “tnameserv”. Una vez hechos estos pasos, ya se podrá ejecutar las aplicaciones en ambas tecnologías.

3.5 Utilización de estas tecnologías con firewall.

Los firewall son encontrados inevitablemente por cualquier aplicación en red de una empresa que tenga que funcionar más allá de los límites de una Intranet. Típicamente, los firewall bloquean todo el tráfico de la red, a excepción de algunos puertos " bien conocidos".

Puesto que la capa de transporte de estas tecnologías abre conexiones dinámicas de sockets entre el cliente y el servidor para facilitar la comunicación, el tráfico de JRMP es bloqueado típicamente por la mayoría de las implementaciones de firewall.

Para pasar a través de firewall, se hace uso de tunneling HTTP encapsulando las llamadas dentro de una petición POST del HTTP. El diagrama siguiente muestra el caso donde un cliente RMI situado detrás de un firewall se comunica con un servidor externo.



En el caso anterior, cuando la capa de transporte intenta establecer una conexión con el servidor, es bloqueado por el firewall. Cuando sucede esto, la capa de transporte reintenta encapsulando los datos de la llamada dentro de una petición POST de HTTP.

Si un cliente está detrás de un firewall, es importante que fijar la característica http.proxyHost del sistema apropiadamente. Dado que casi todos los firewall reconocen el protocolo del HTTP, el proxy server especificado debe poder remitir la llamada directamente al puerto en el cual el servidor remoto está escuchando en el exterior. Una vez que los datos encapsulados en HTTP se reciben en el servidor, son decodificados y enviados automáticamente por la capa de transporte de estas tecnologías. La respuesta se envía de nuevo a cliente como datos encapsulados en HTTP.

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 ingenieros 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.

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.

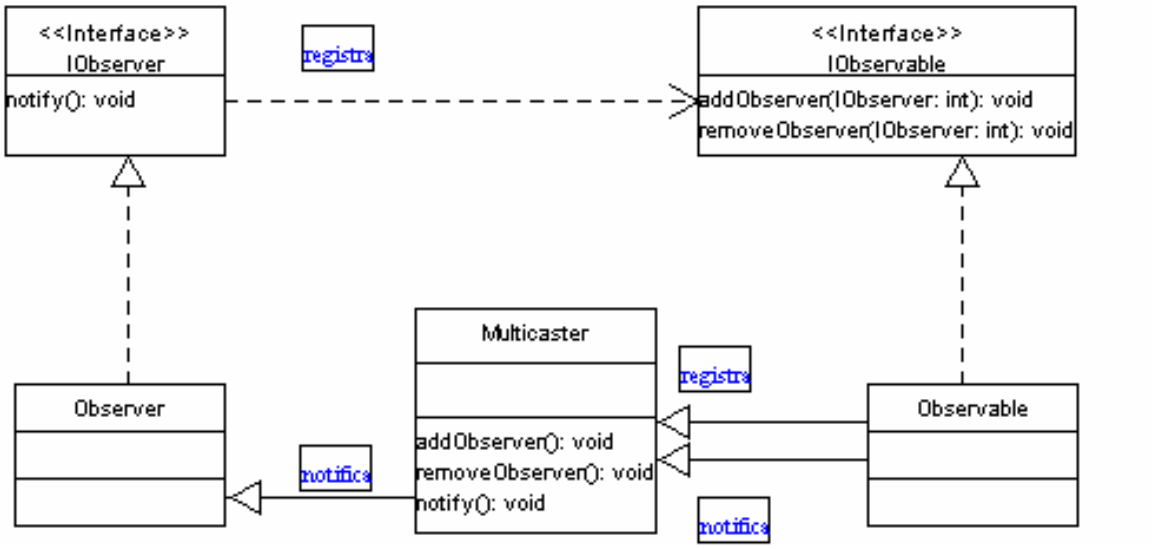
4.4 Patrón Observer.

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 requerido 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 proxy:

- 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.

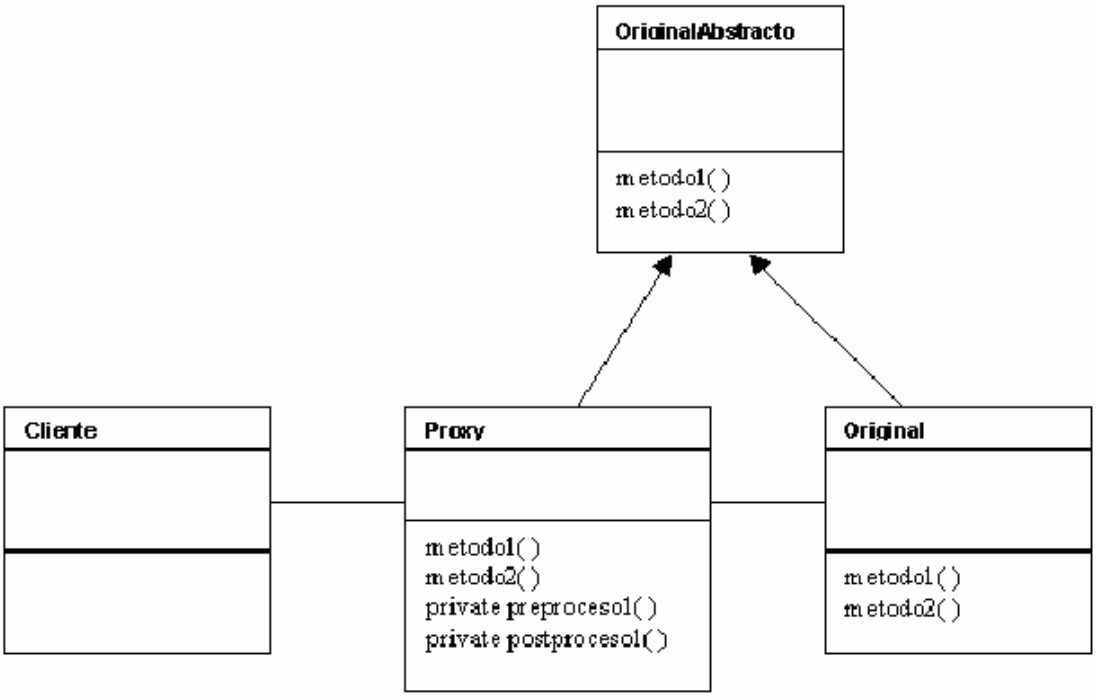
4.5.1 Implementación.

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.

4.5.1. Estructura.



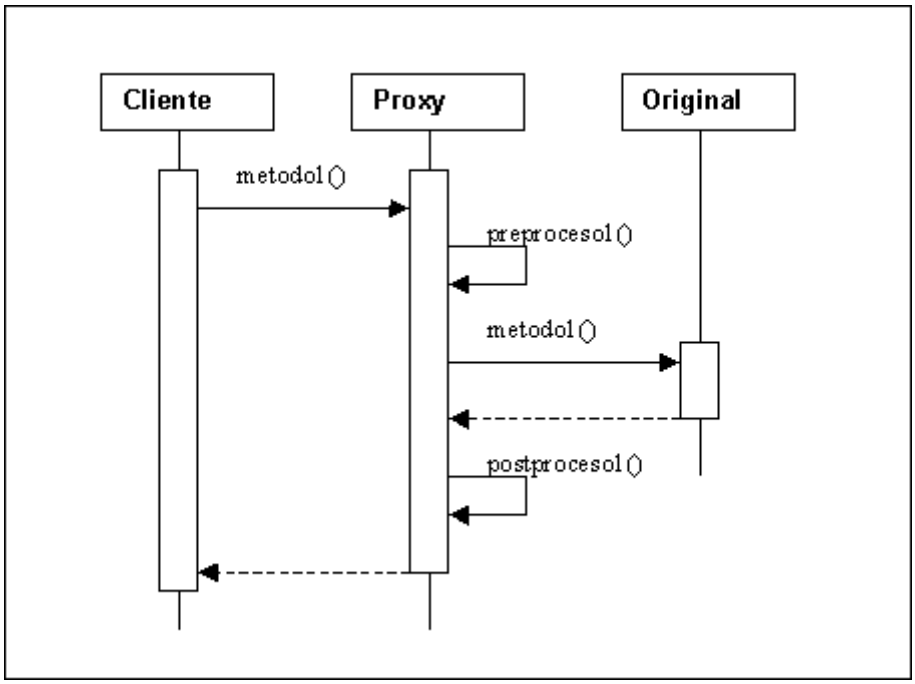
4.5.2. Componentes.

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.

4.5.3 Diagrama.



4.5.4 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. También permite sincronizar accesos concurrentes al mismo, bloqueándolo mientras está en uso.

4.5.5 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.

4.5.6 Implementación en la aplicación Goya.

La implementación de este patrón en la aplicación queda totalmente demostrada ya que en la propia arquitectura de tanto la tecnología CORBA como RMI, aparecen los conceptos de proxy y stub que realizan las mismas funciones que se describe en los roles de este patrón y que ya han suficientemente comentadas en los puntos en los que se detallaban ambas tecnologías.

5. UML

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. Las diferentes compañías fueron: DEC, HP, IBM, Icon Computing, MCI System House, IntelliCorp, Microsoft, Oracle, Texas Instrument, Unisys, I-Logix y Rational. 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.

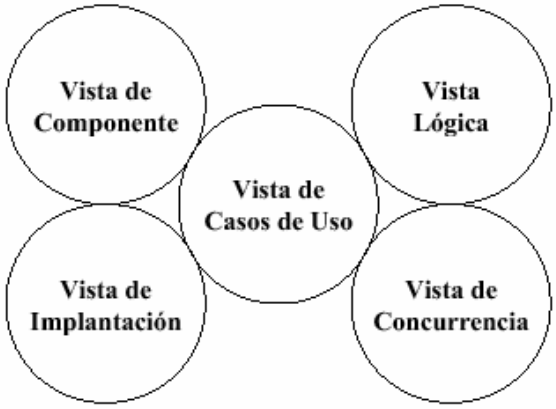
Para usar correctamente un lenguaje de modelado, es necesario conocer todas estas reglas pero aunque el lenguaje esté bien definido no pueden garantizarse los modelos que se construyan con él al igual que al escribir una historia en el lenguaje natural no lleva al autor de la misma a escribir una buena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

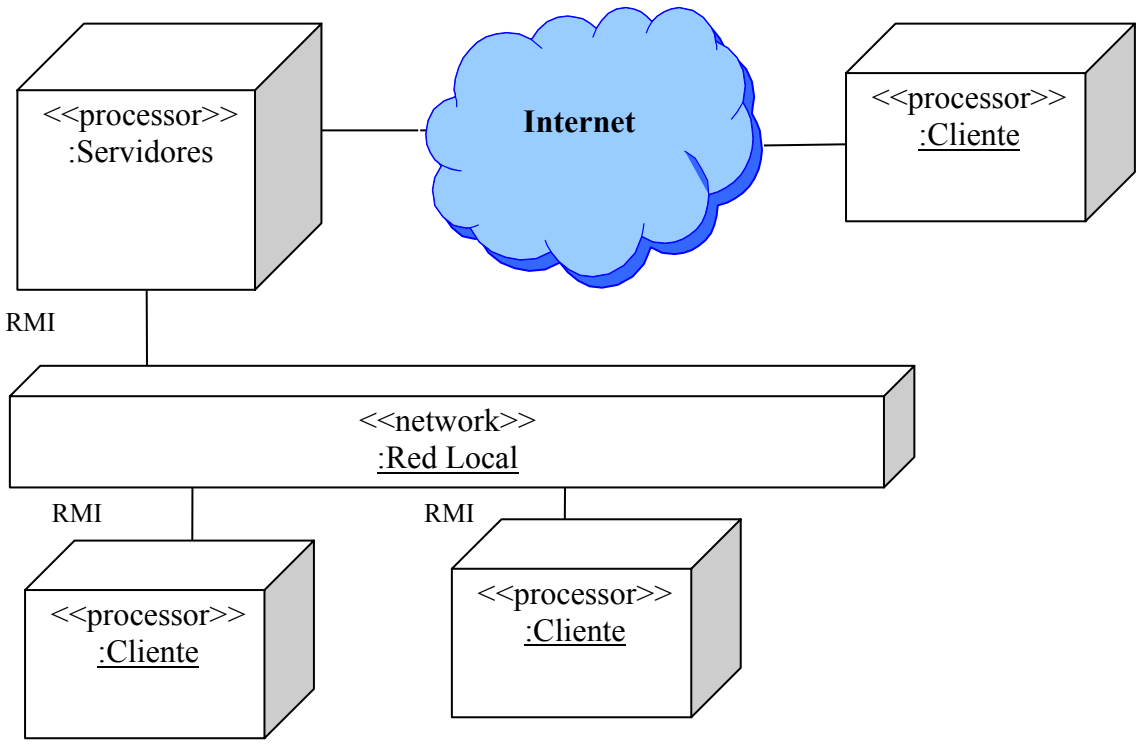
propio flujo de control al procesador; y se presentan tanto los aspectos estáticos de la asignación de los componentes a la arquitectura física, como los aspectos dinámicos de su interacción.

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. Diagrama de despliegue.



6.2. Diagramas de Casos de Uso.

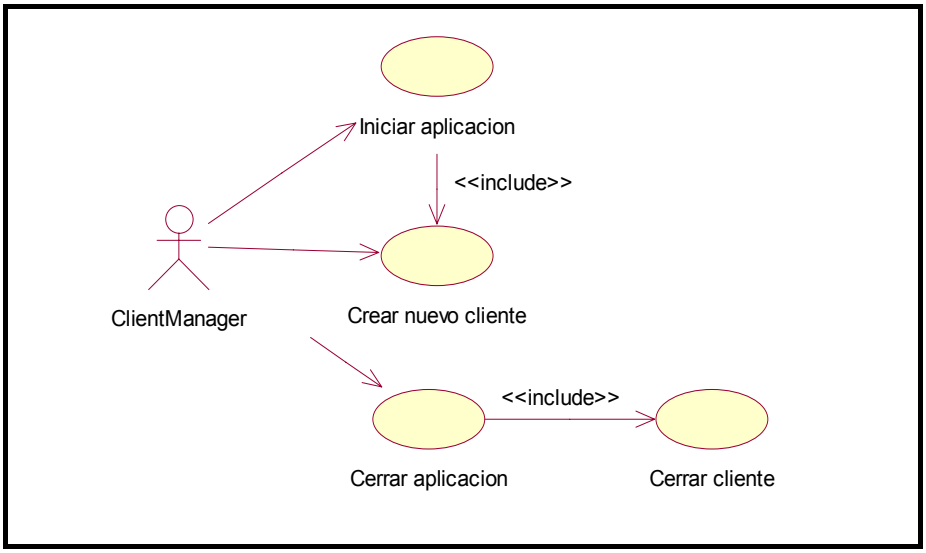


Ilustración 0: Casos de uso de ClientManager

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.

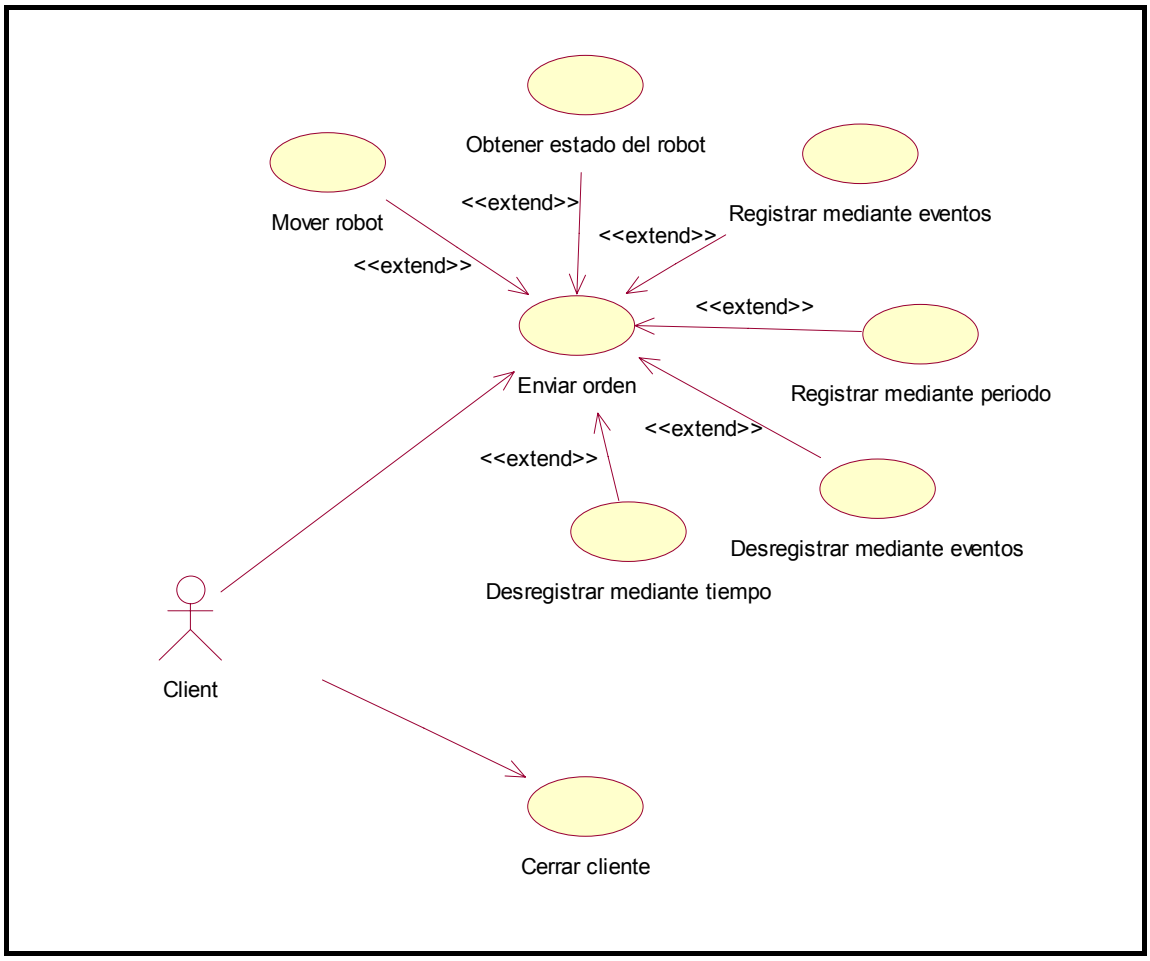


Ilustración 1: Casos de uso de Client

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 órdenes 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 órdenes 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 órdenes 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 órdenes 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 órdenes 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 órdenes 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 órdenes 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 órdenes 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.

Ilustración 2: Casos de uso de 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 órdenes 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.

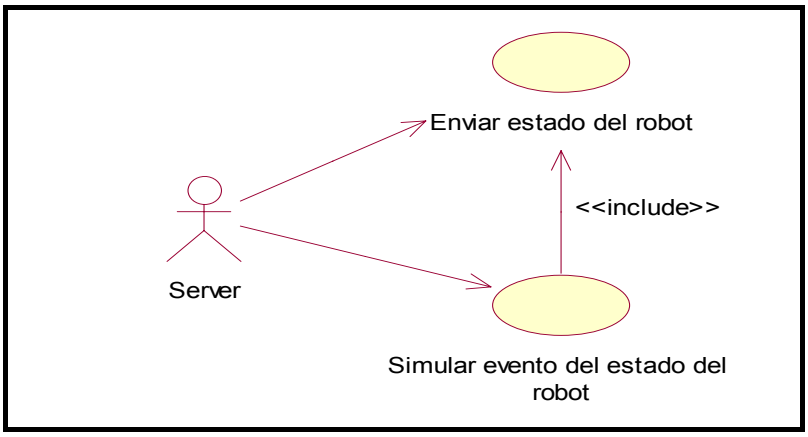


Ilustración 3: Casos de uso de Server

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 órdenes 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 órdenes 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.

6.3 Diagramas de Estructura Estática.

6.3.1 Diagramas de clase de la implementación RMI.

Client-Controls

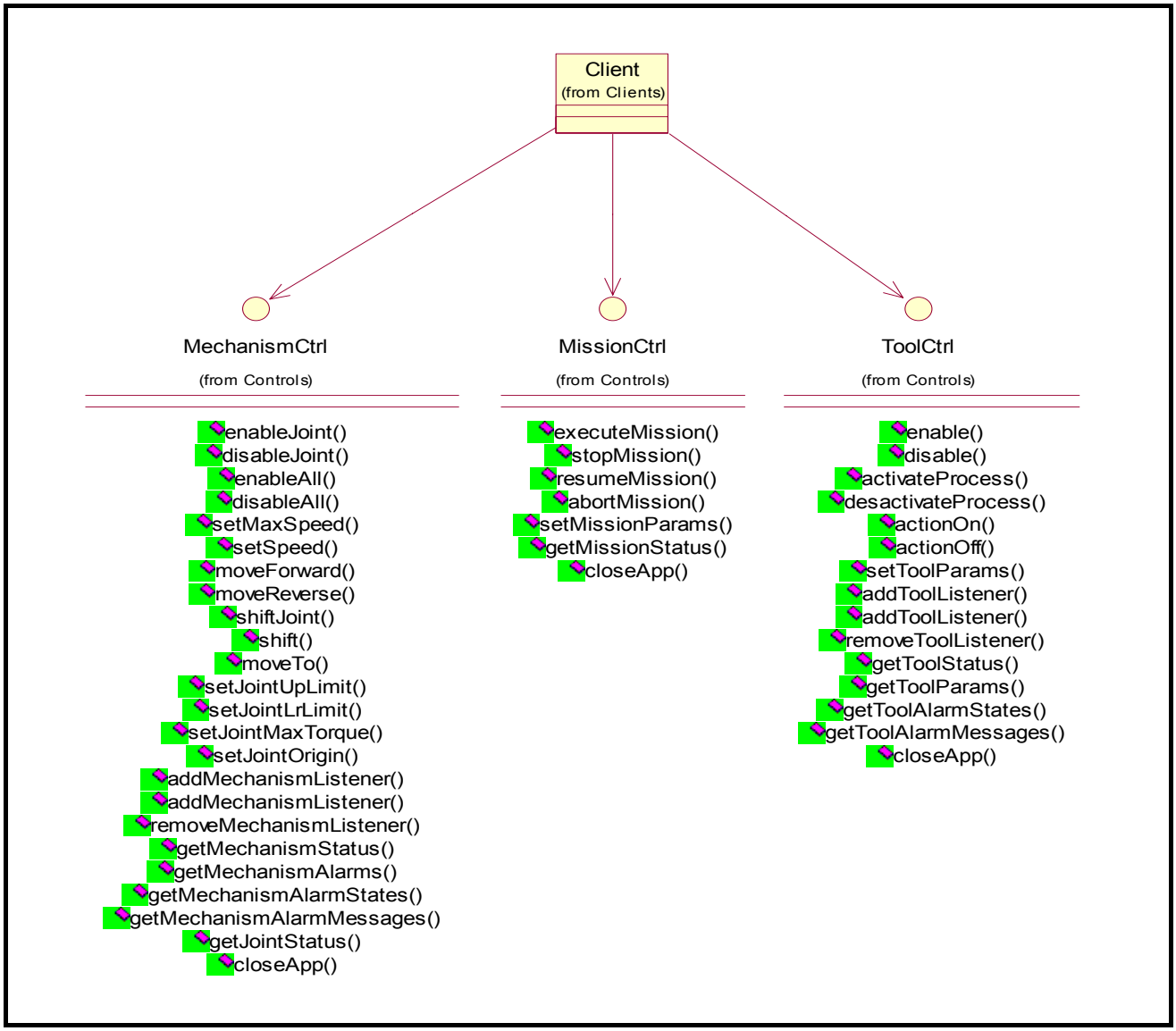


Ilustración 4: Diagrama de clase Client-Controls

Client-Listeners

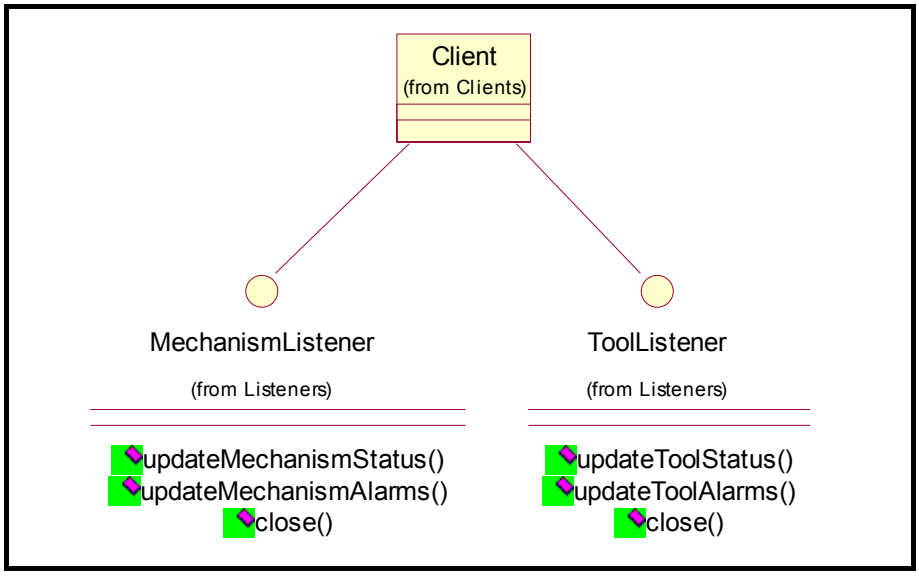


Ilustración 5: Diagrama de clase Client-Listeners

Client-Updaters

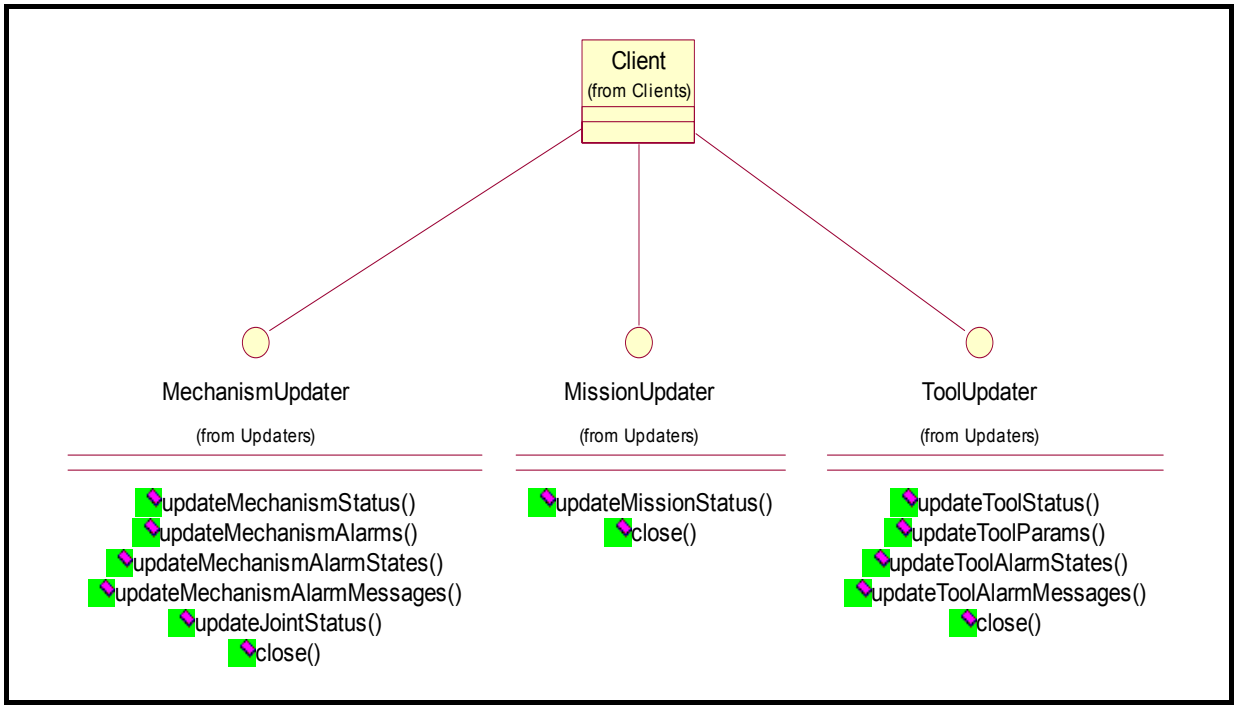


Ilustración 6: Diagrama de clase Client-Updaters

Manager

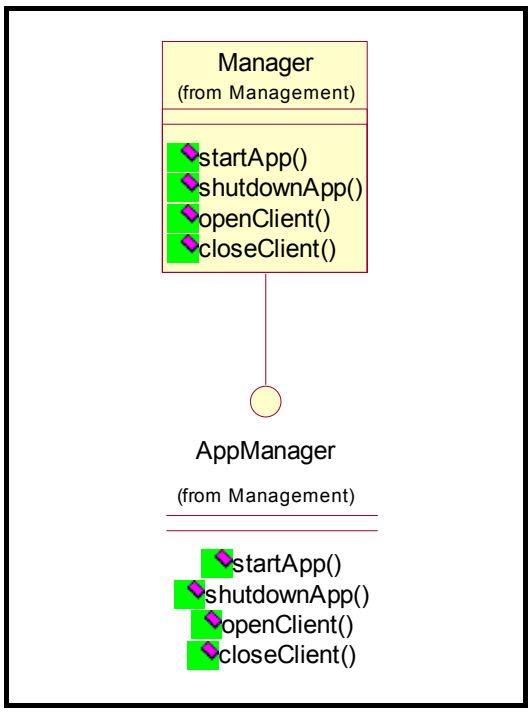


Ilustración 7: Diagrama de clase Manger

Servers

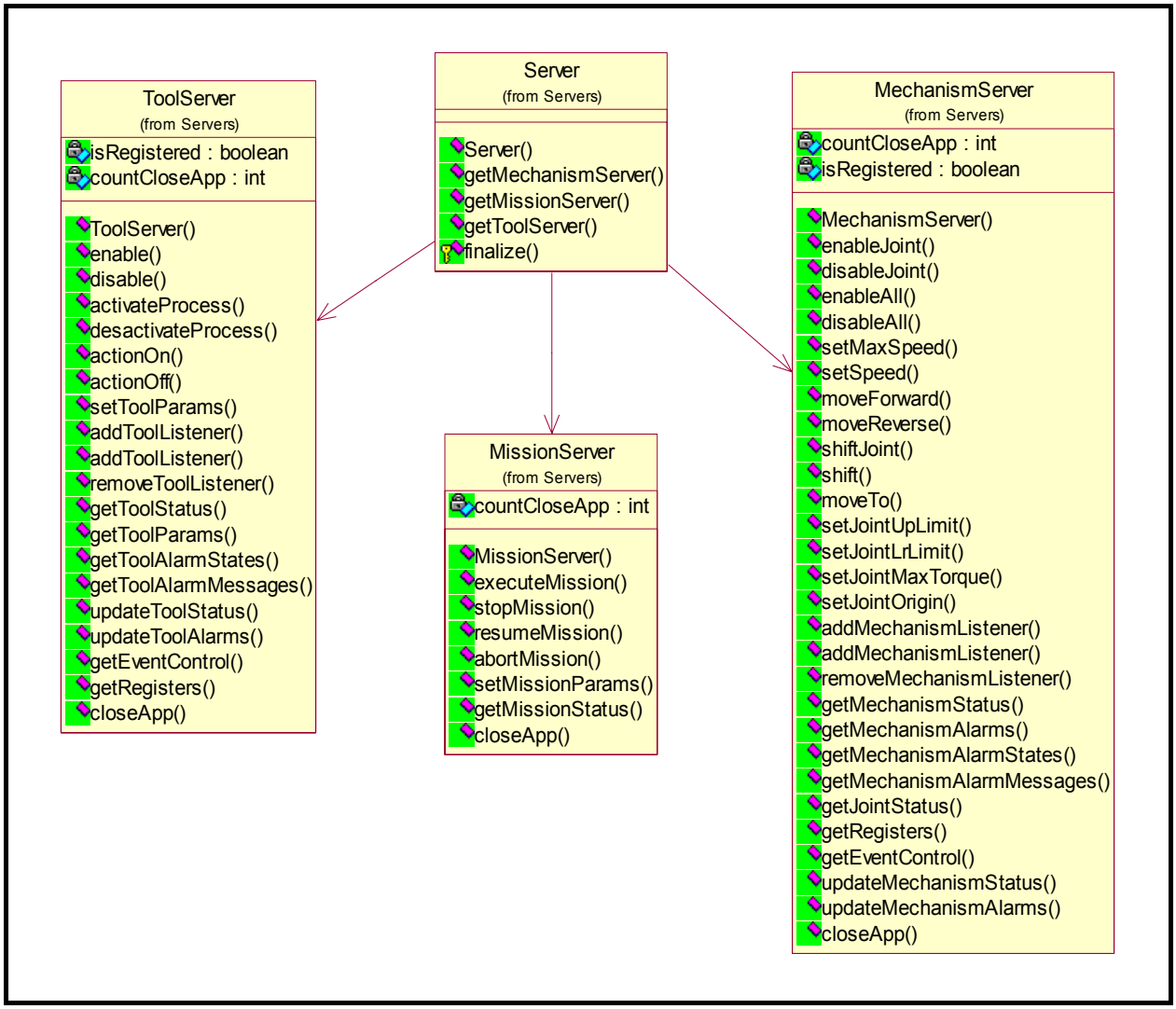


Ilustración 8: Diagrama de clase Servers

Servers-Controls

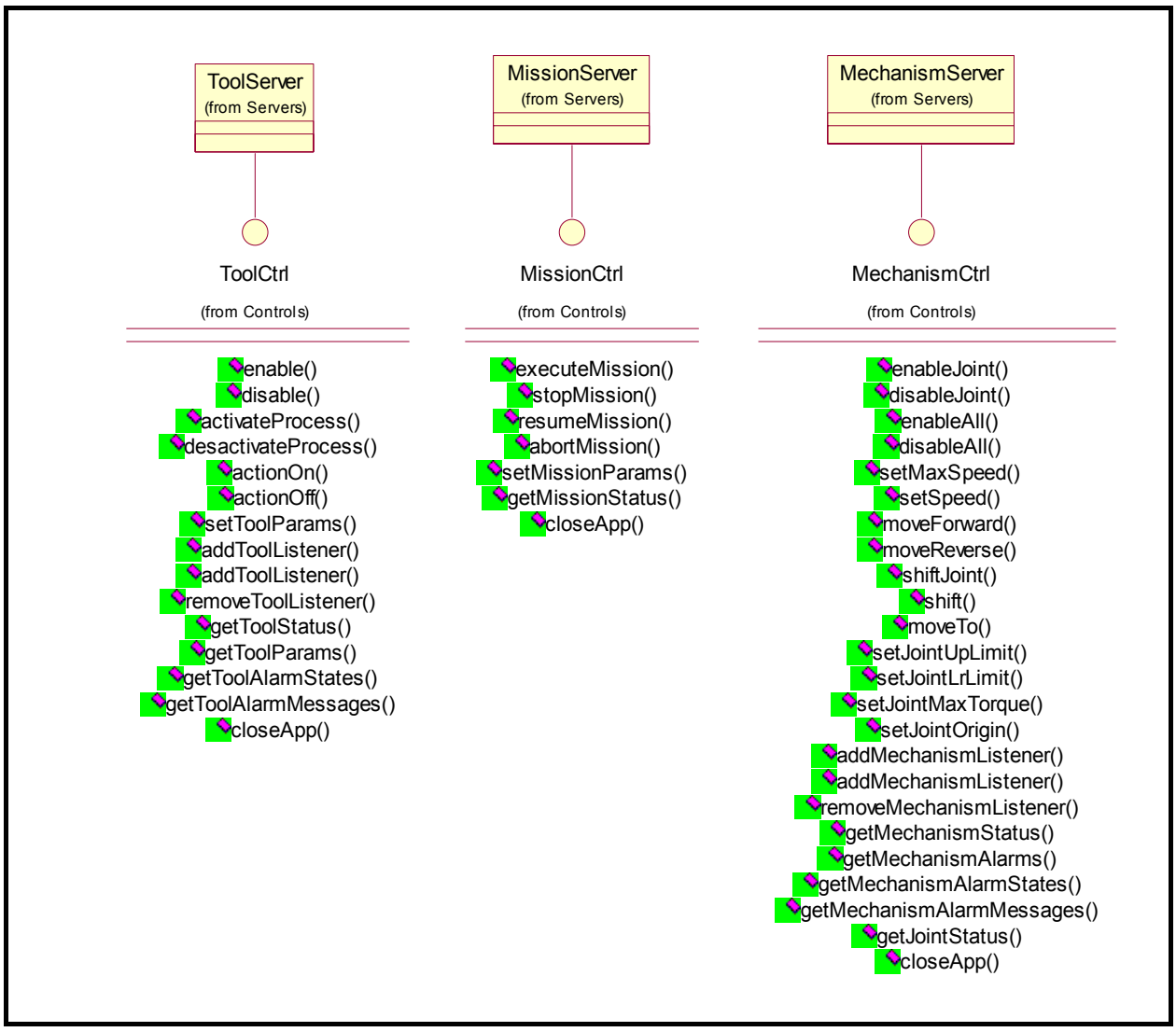


Ilustración 9: Diagrama de clase Servers-Controls

MechanismServer

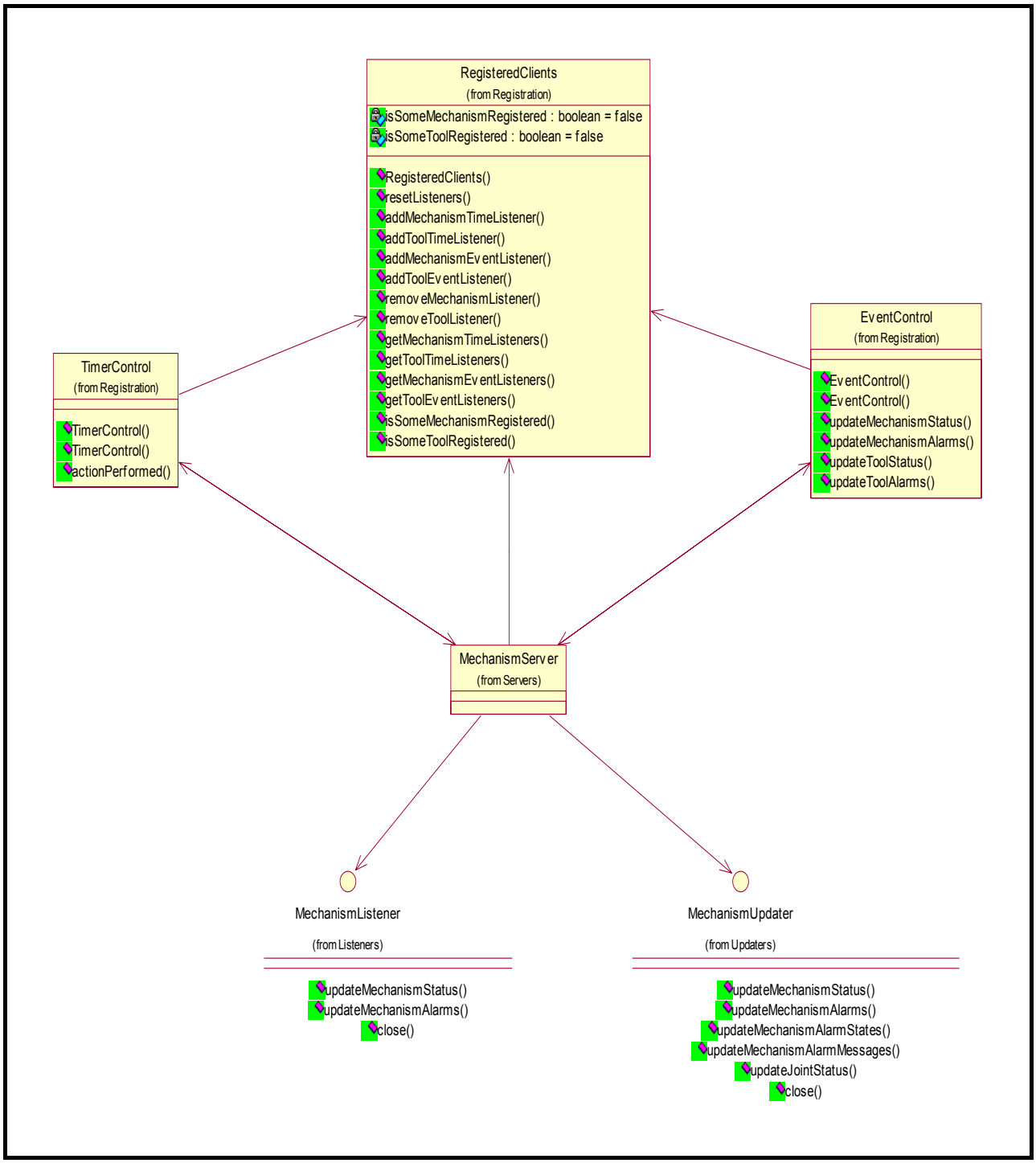


Ilustración 10: Diagrama de clase MechanismServer

ToolServer

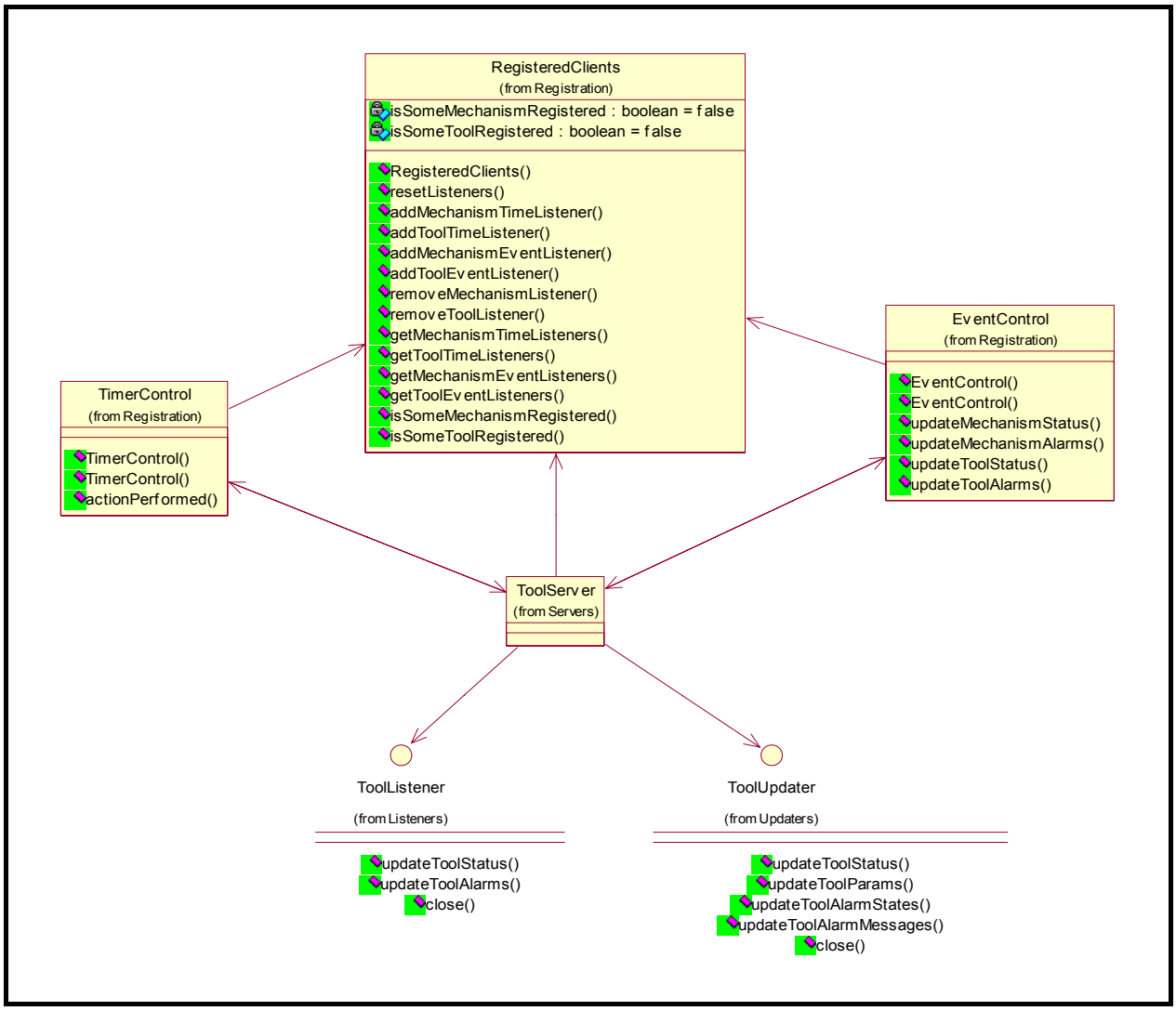


Ilustración 11: Diagrama de clase de ToolServer

MissionServer

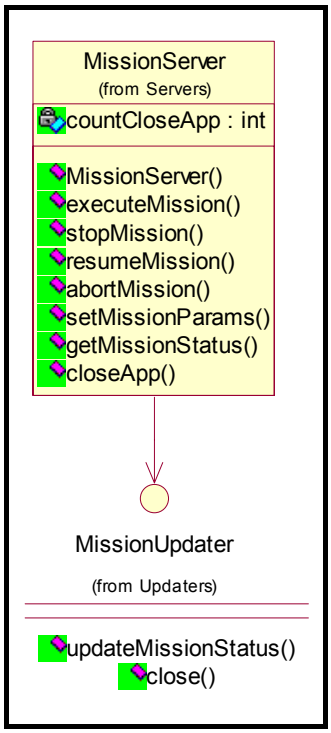


Ilustración 12: Diagrama de clase de MissionServer

6.3.2. Diagramas de clase de la implementación CORBA.

Alarm

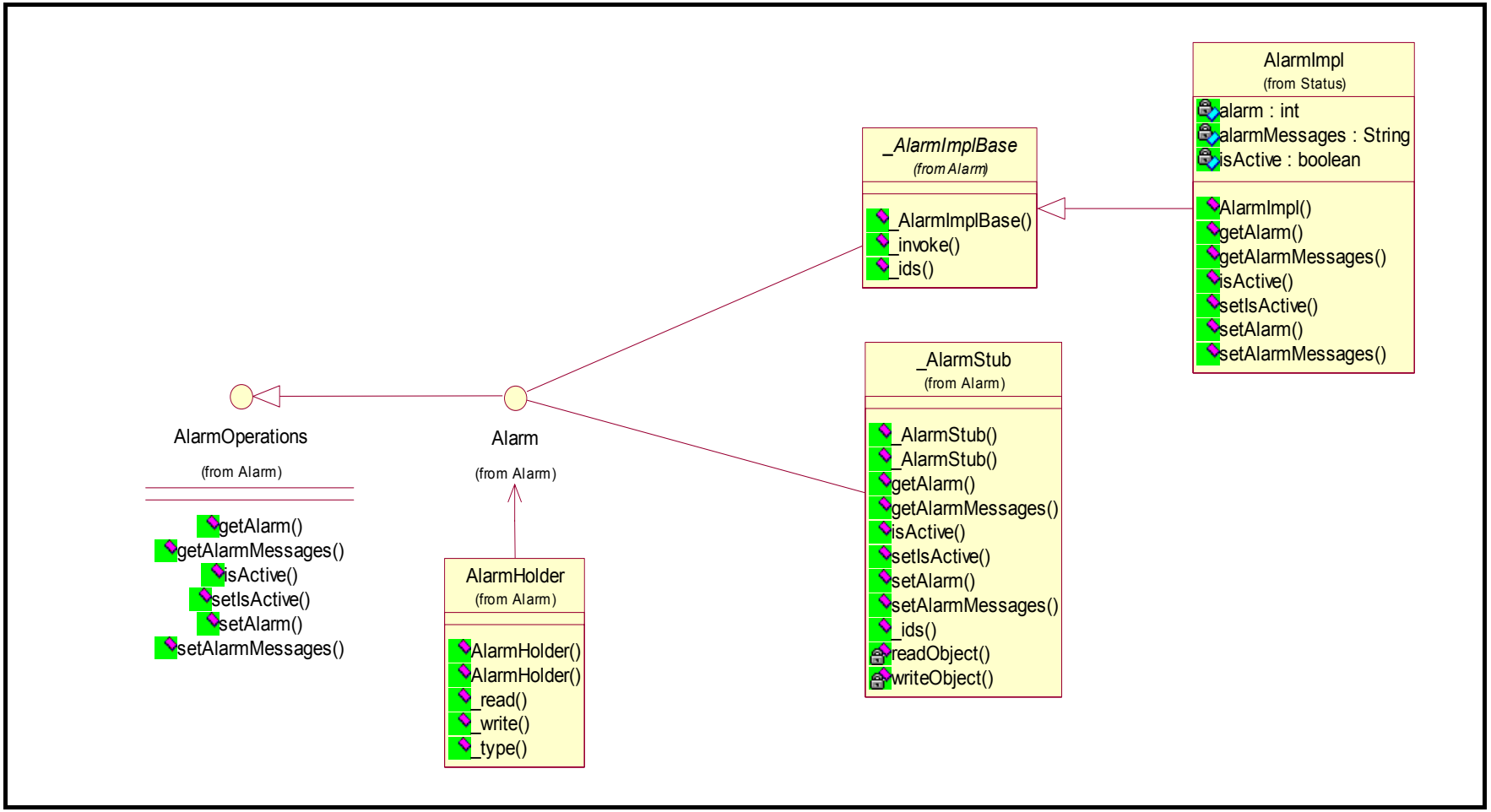


Ilustración 13: Diagrama de clase de Alarm

BlastingStatus

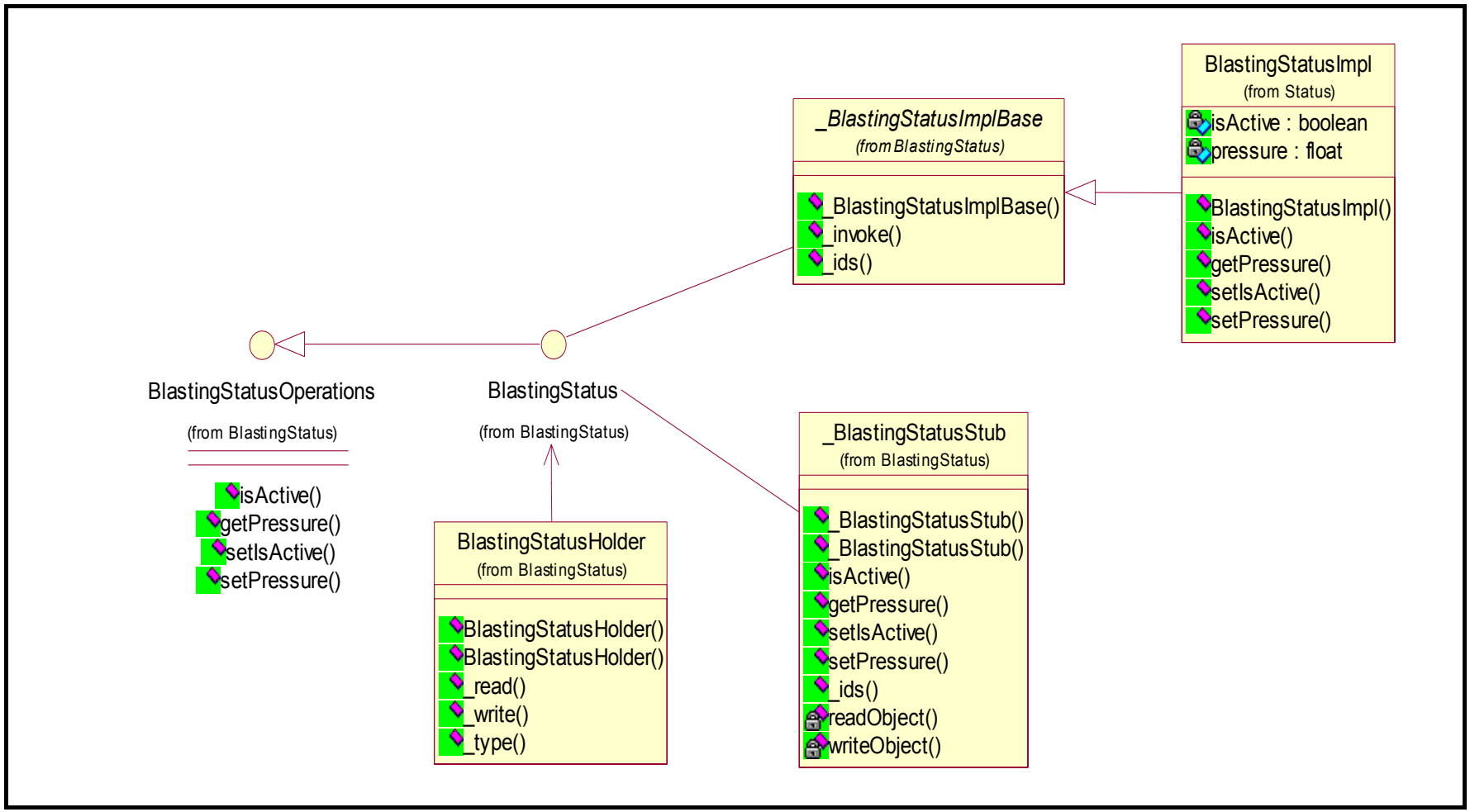


Ilustración 14: Diagrama de clase de BlastingStatus

GoyaJointStatus

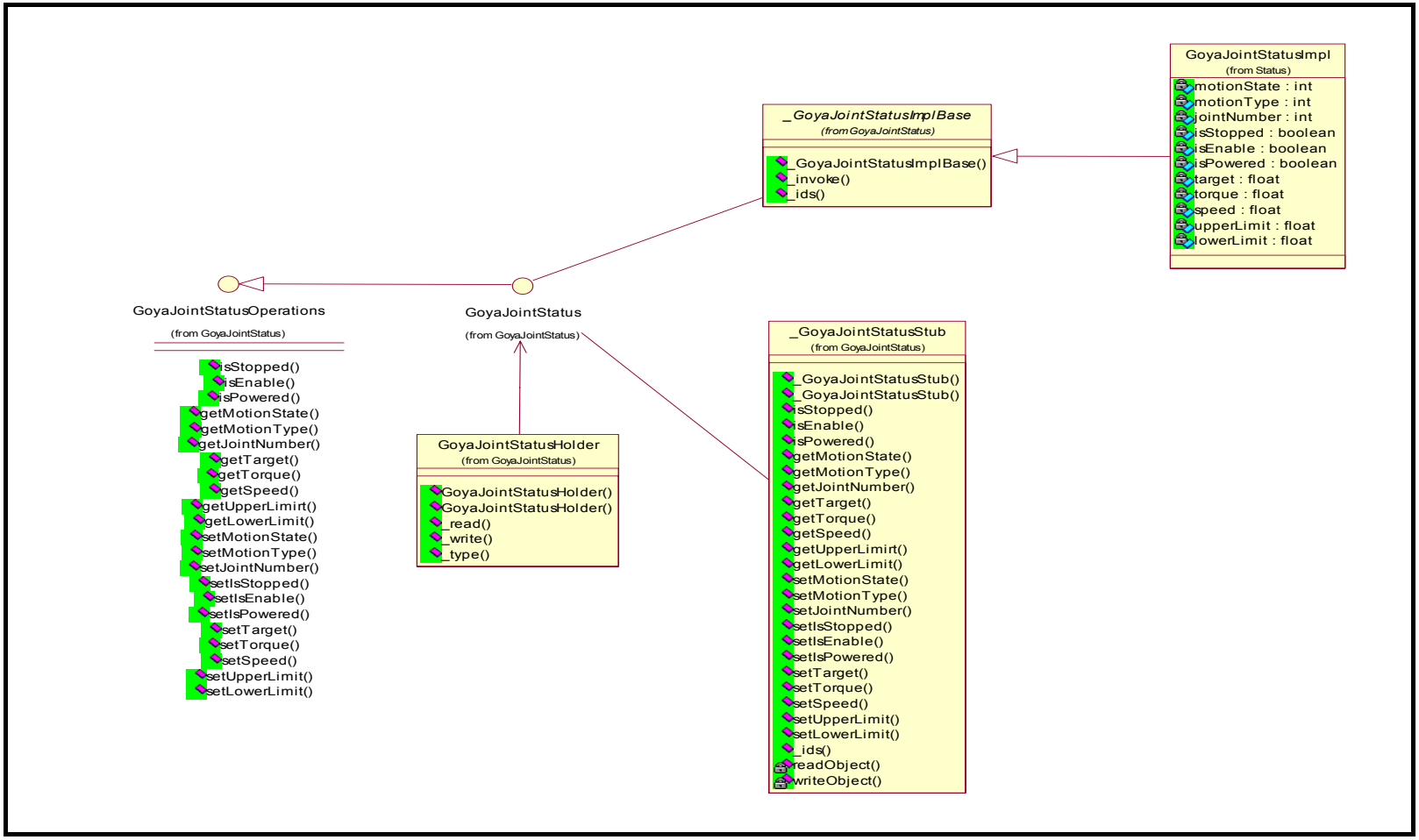


Ilustración 15: Diagrama de clase de GoyaJointStatus

GoyaStatus

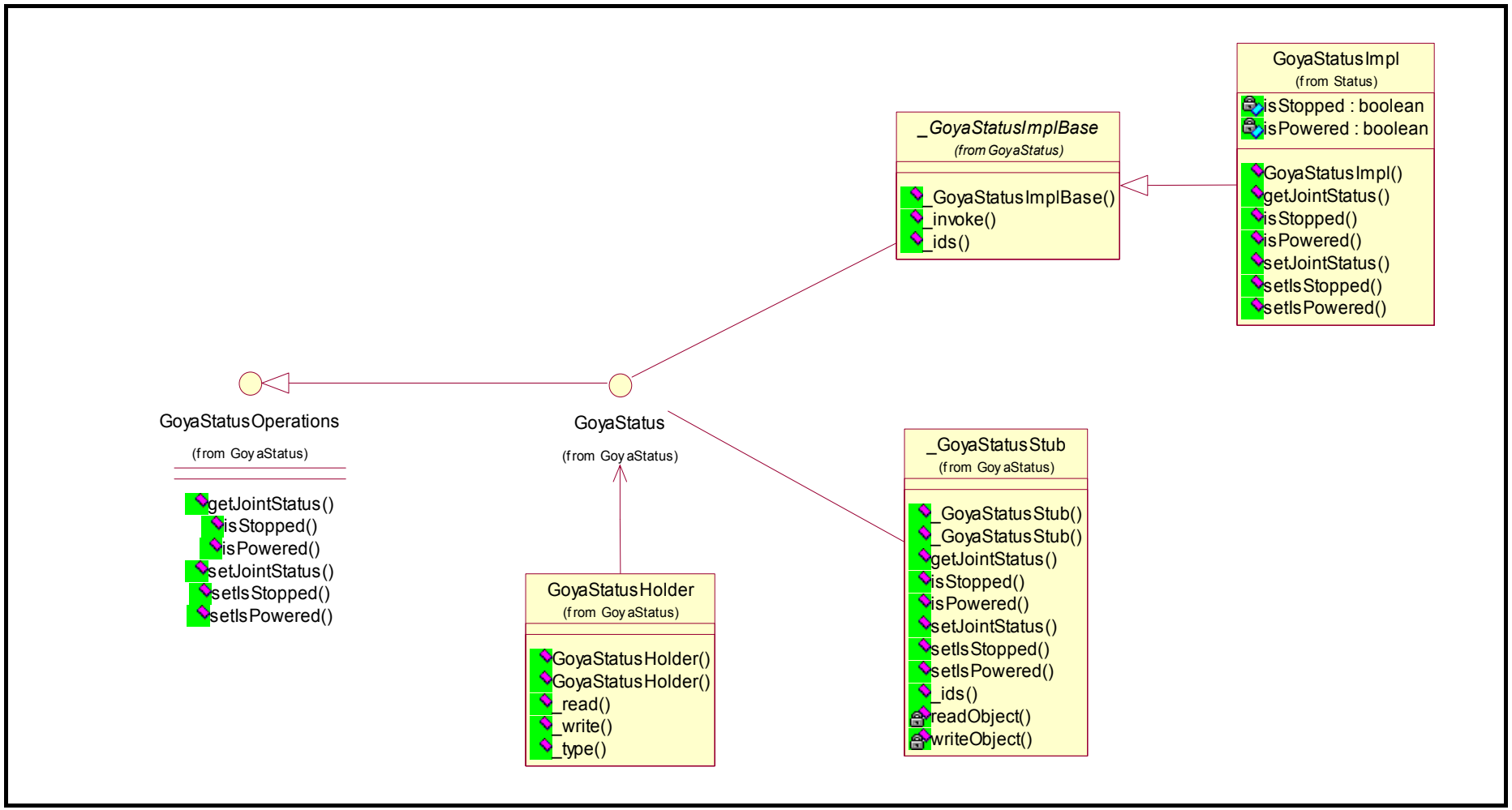


Ilustración 16: Diagrama de clase de GoyaStatus

Manager

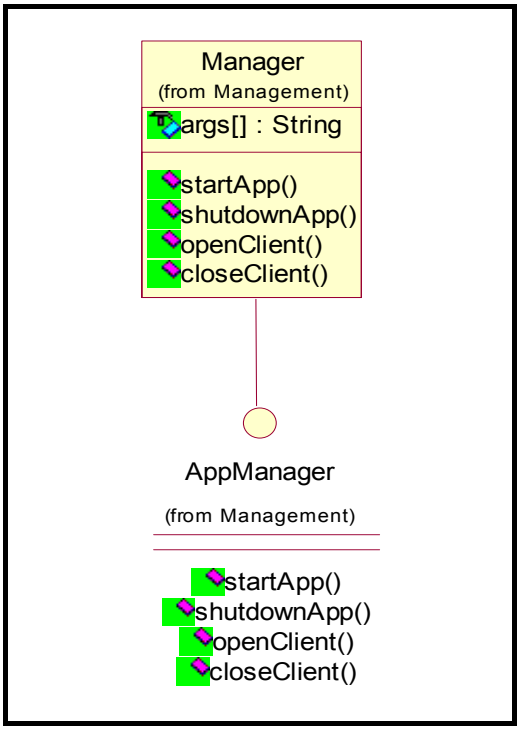


Ilustración 17: Diagrama de clase de Manager

Clients

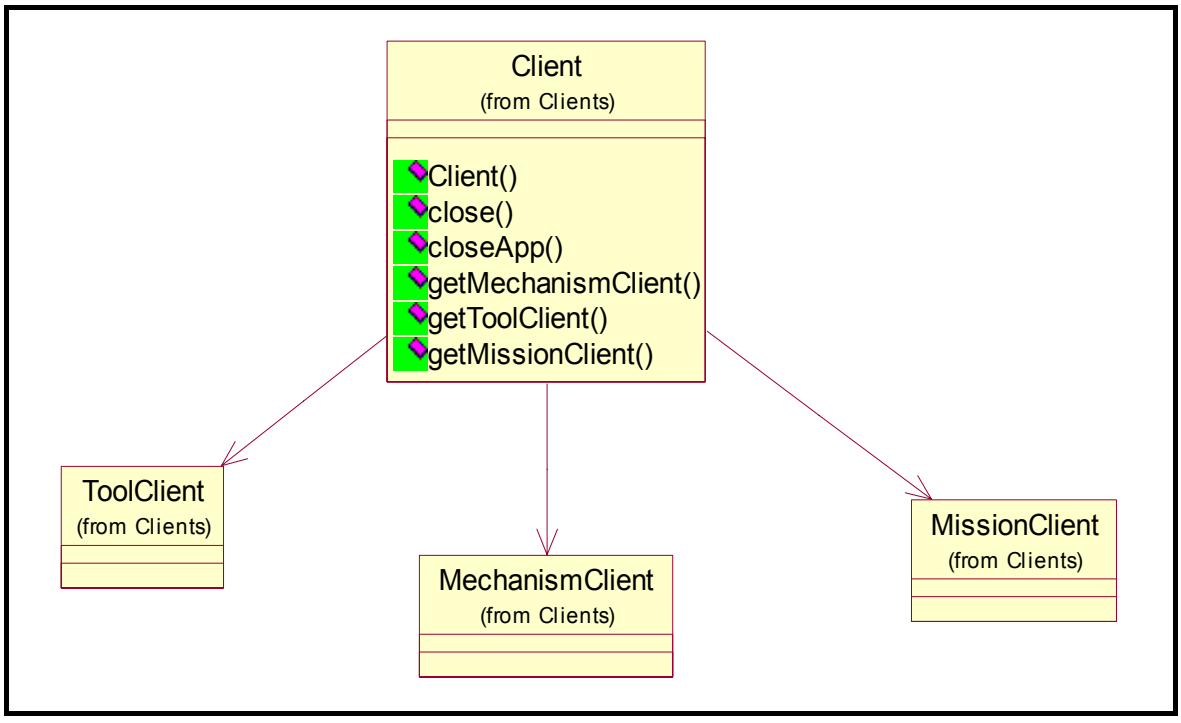


Ilustración 18: Diagrama de clase de Clients

MechanismClient

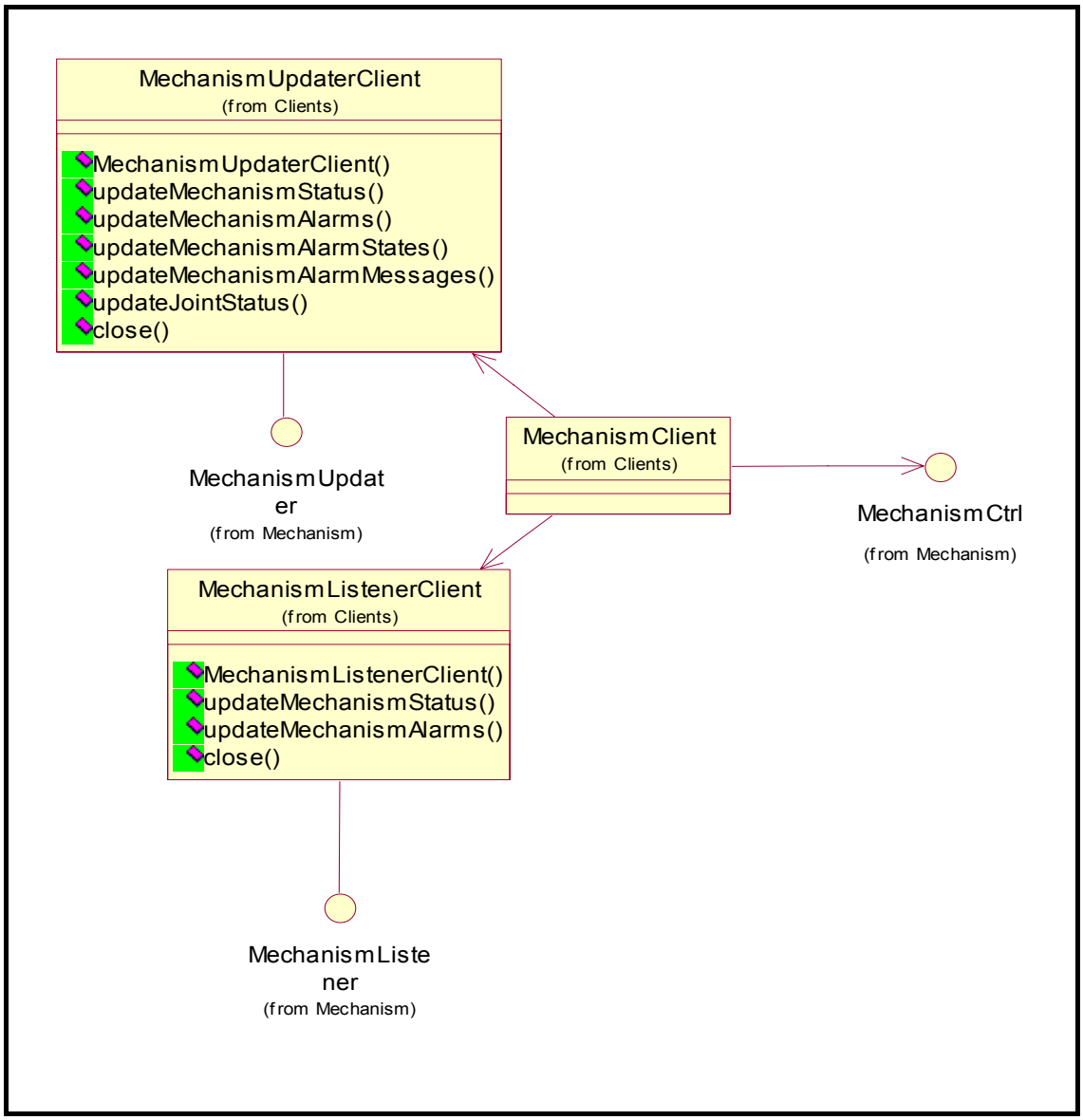


Ilustración 19: Diagrama de clase de MechanismClient

MissionClient

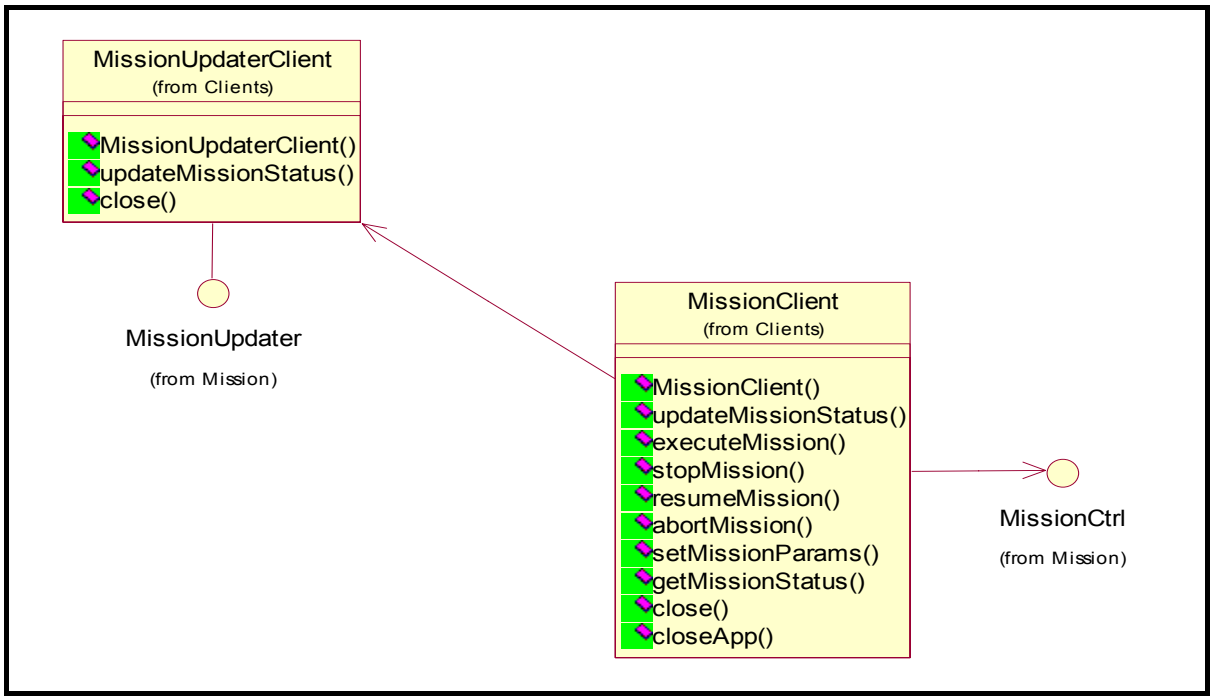


Ilustración 20: Diagrama de clase de MissionClient

ToolClient

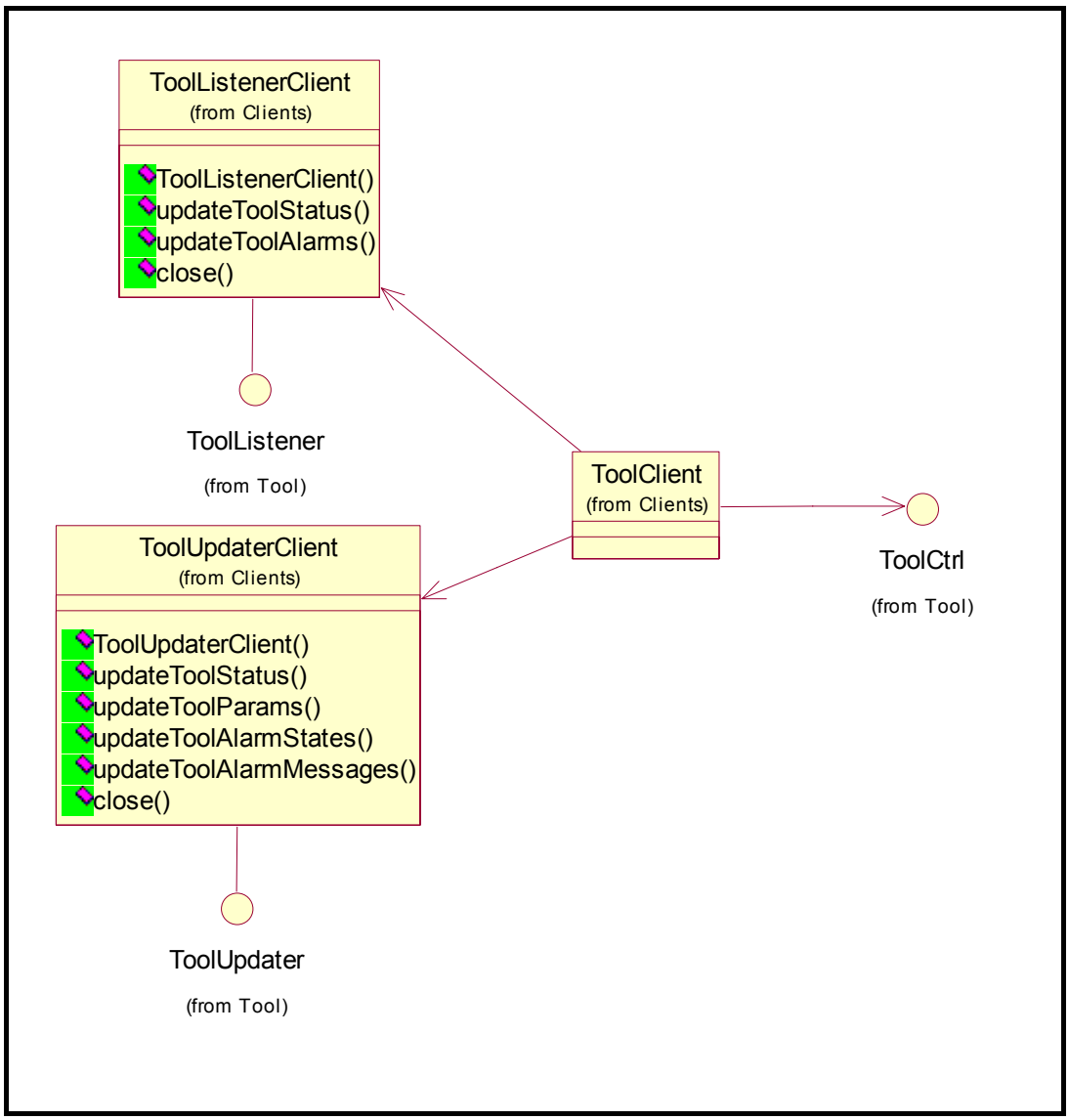


Ilustración 21: Diagrama de clase de ToolClient

MechanismCtrl

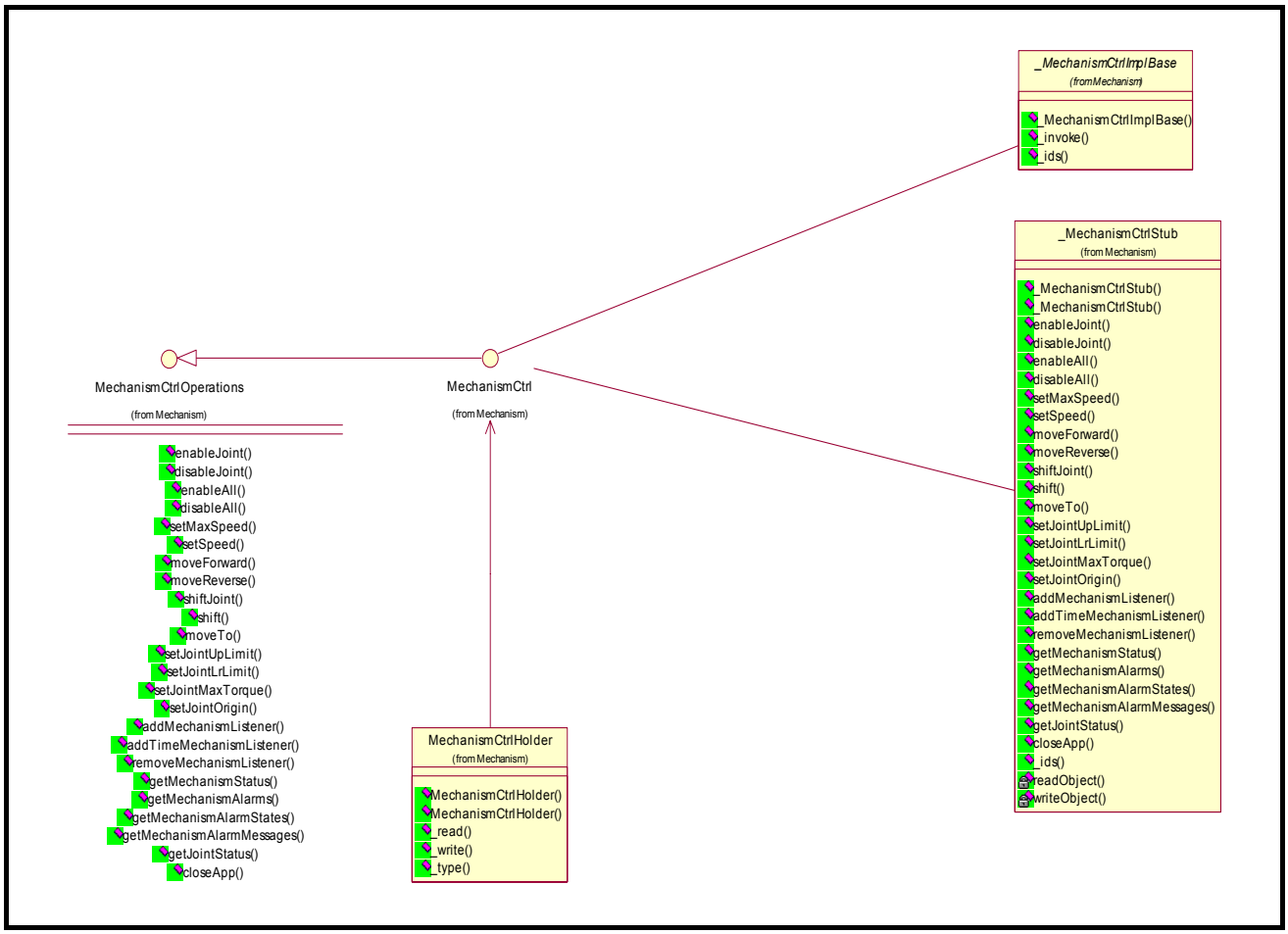


Ilustración 22: Diagrama de clase de MechanismCtrl

MissionCtrl

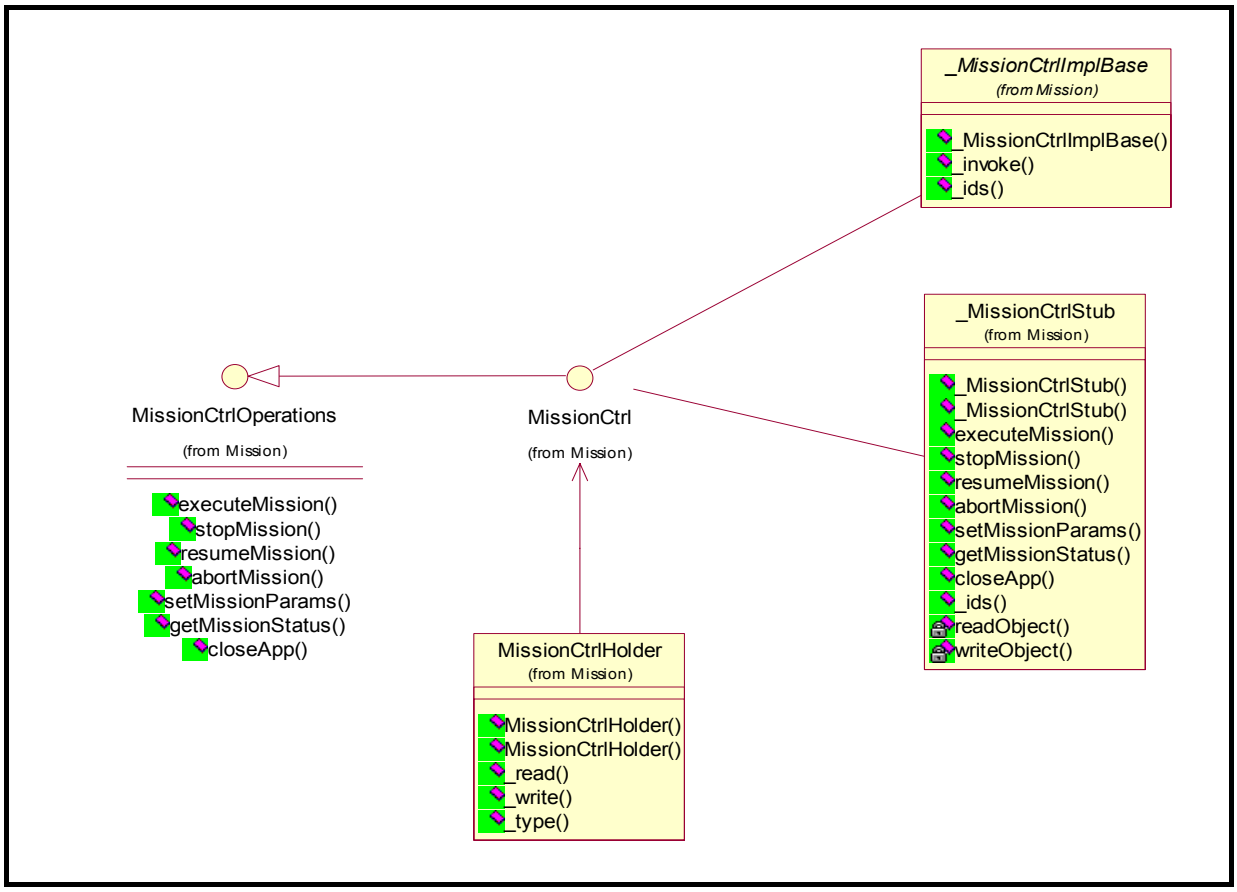


Ilustración 23: Diagrama de clase de MissionCtrl

ToolCtrl

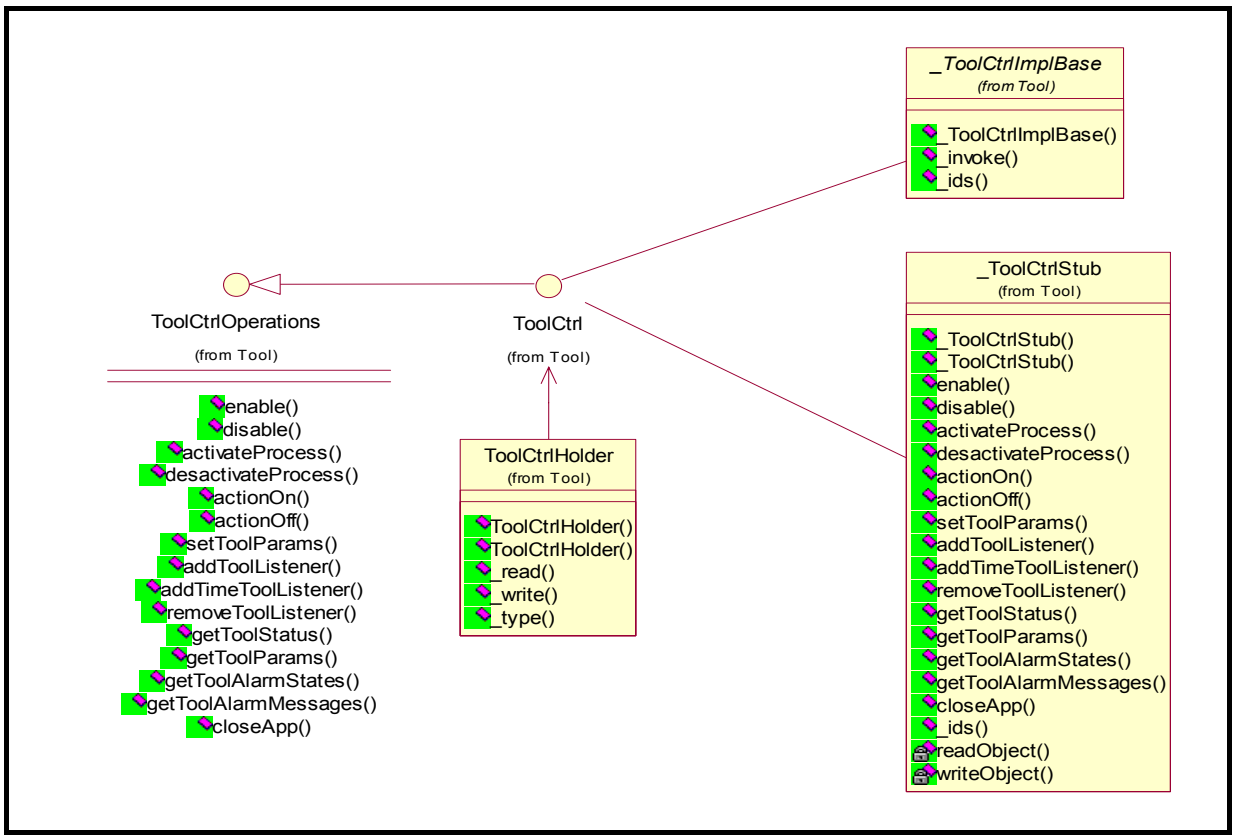


Ilustración 24: Diagrama de clase de ToolCtrl

MechanismListener

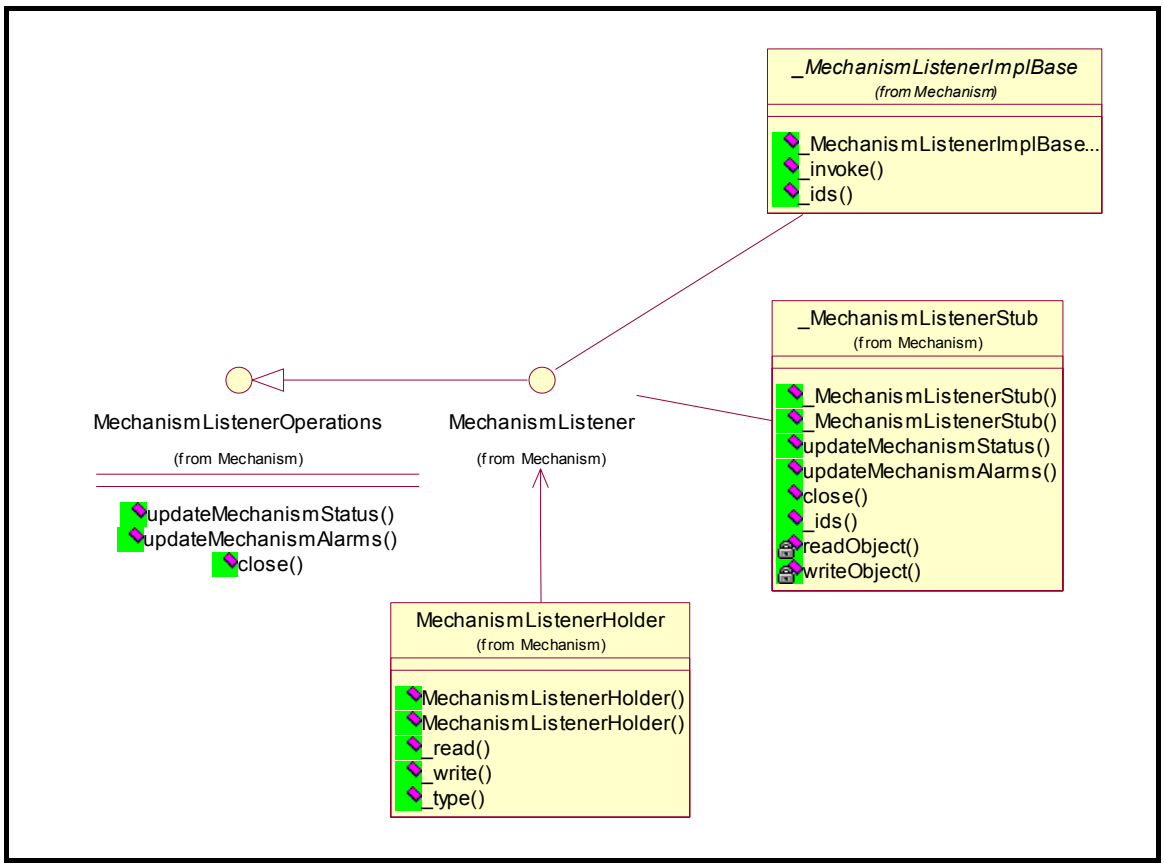


Ilustración 25: Diagrama de clase de MechanismListener

ToolListener

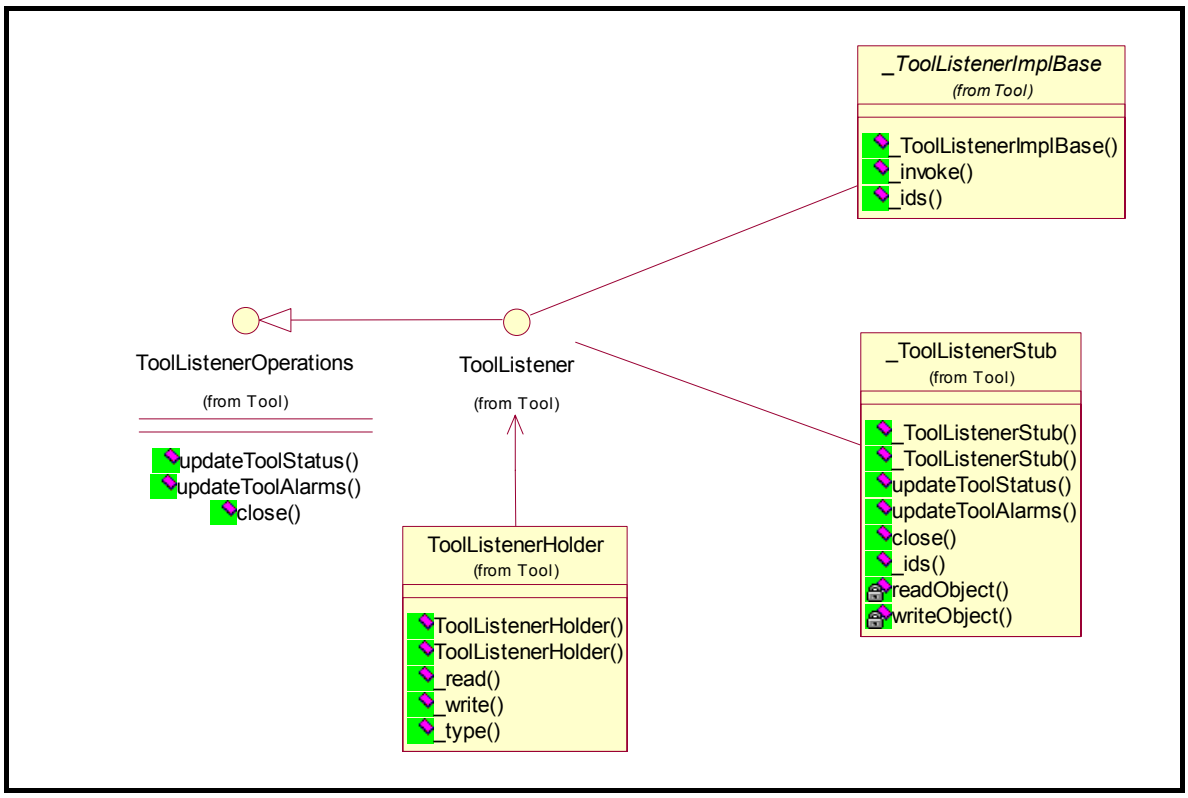


Ilustración 26: Diagrama de clase de ToolListener

MechanismUpdater

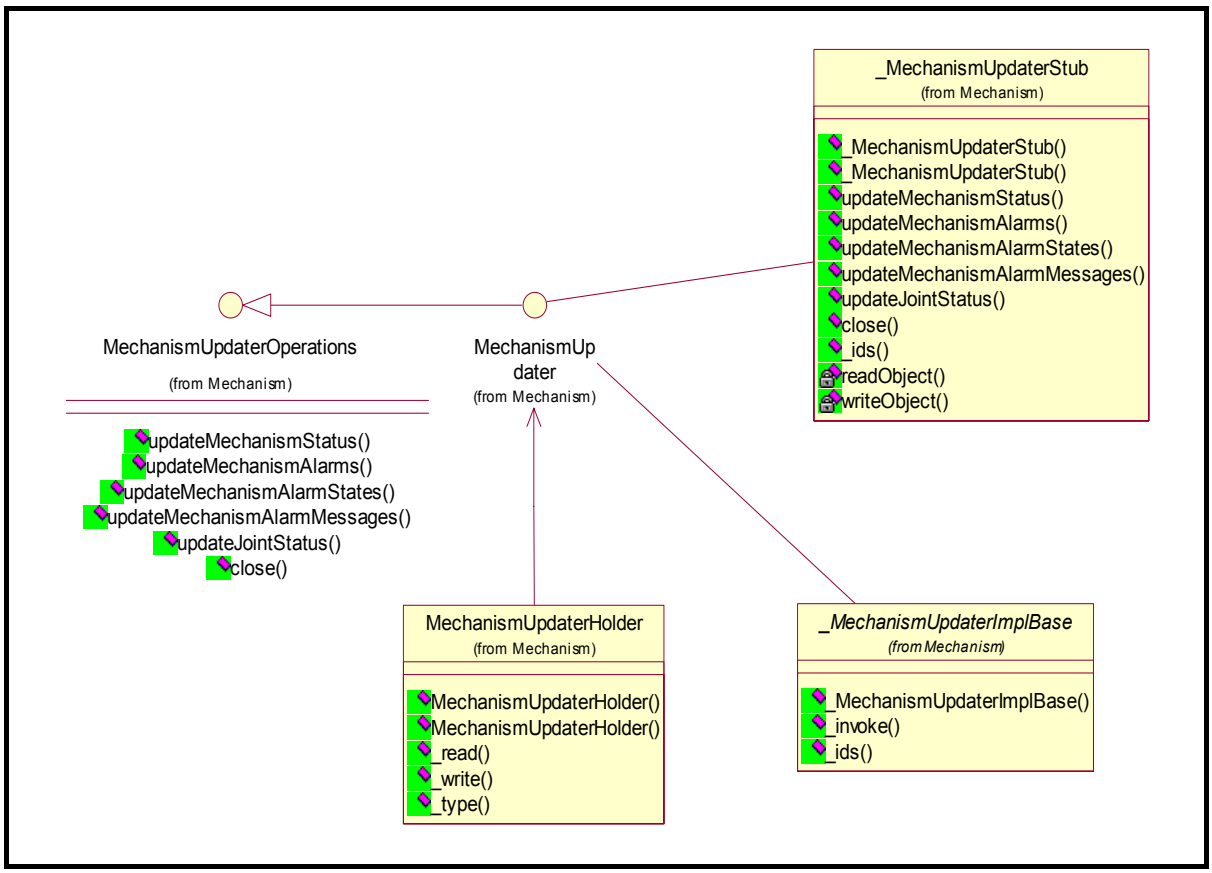


Ilustración 27: Diagrama de clase de MechanismUpdater

MissionUpdater

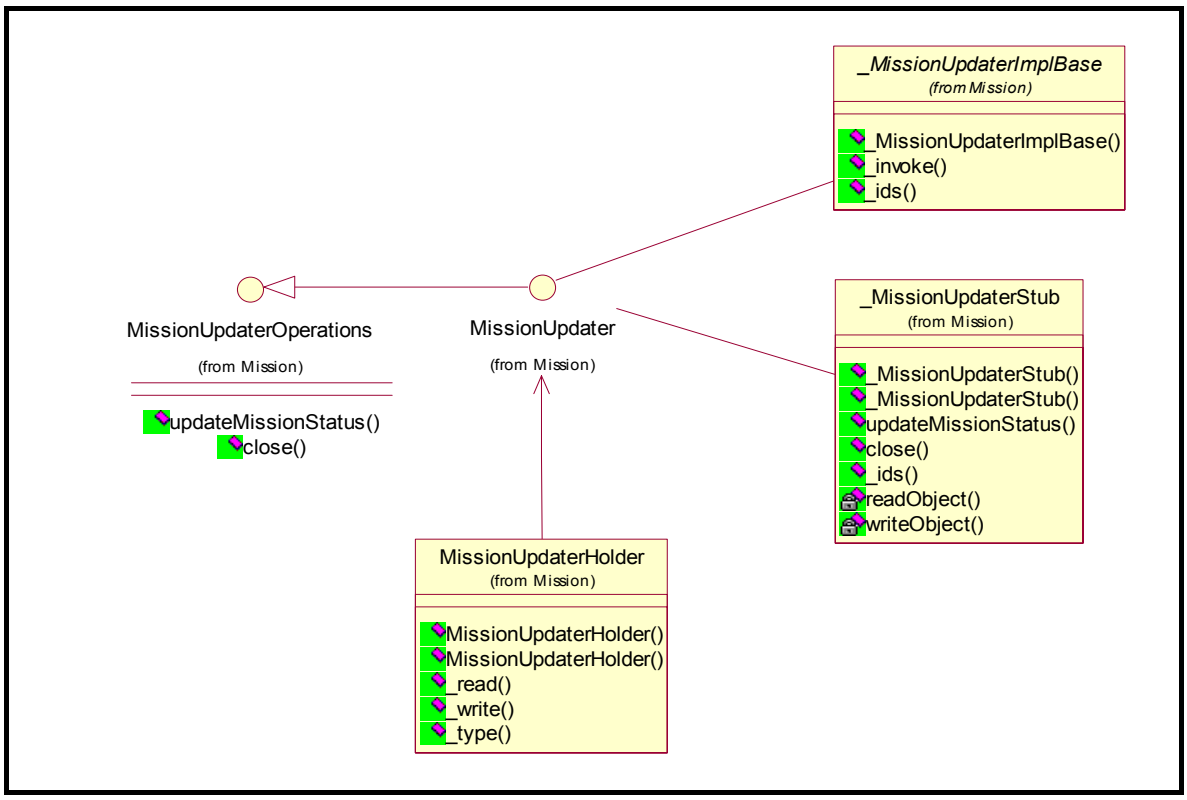


Ilustración 28: Diagrama de clase de MissionUpdater

ToolUpdater

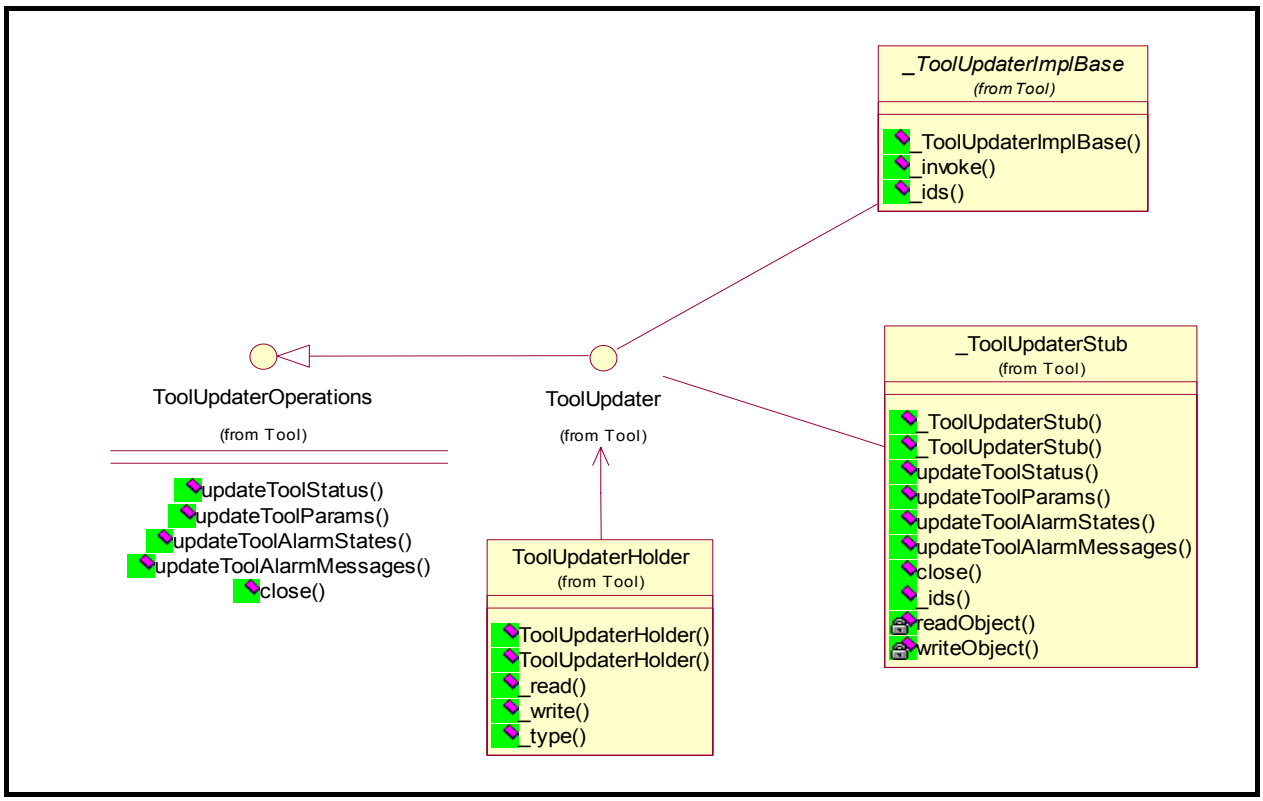


Ilustración 29: Diagrama de clase de ToolUpdater

Servers

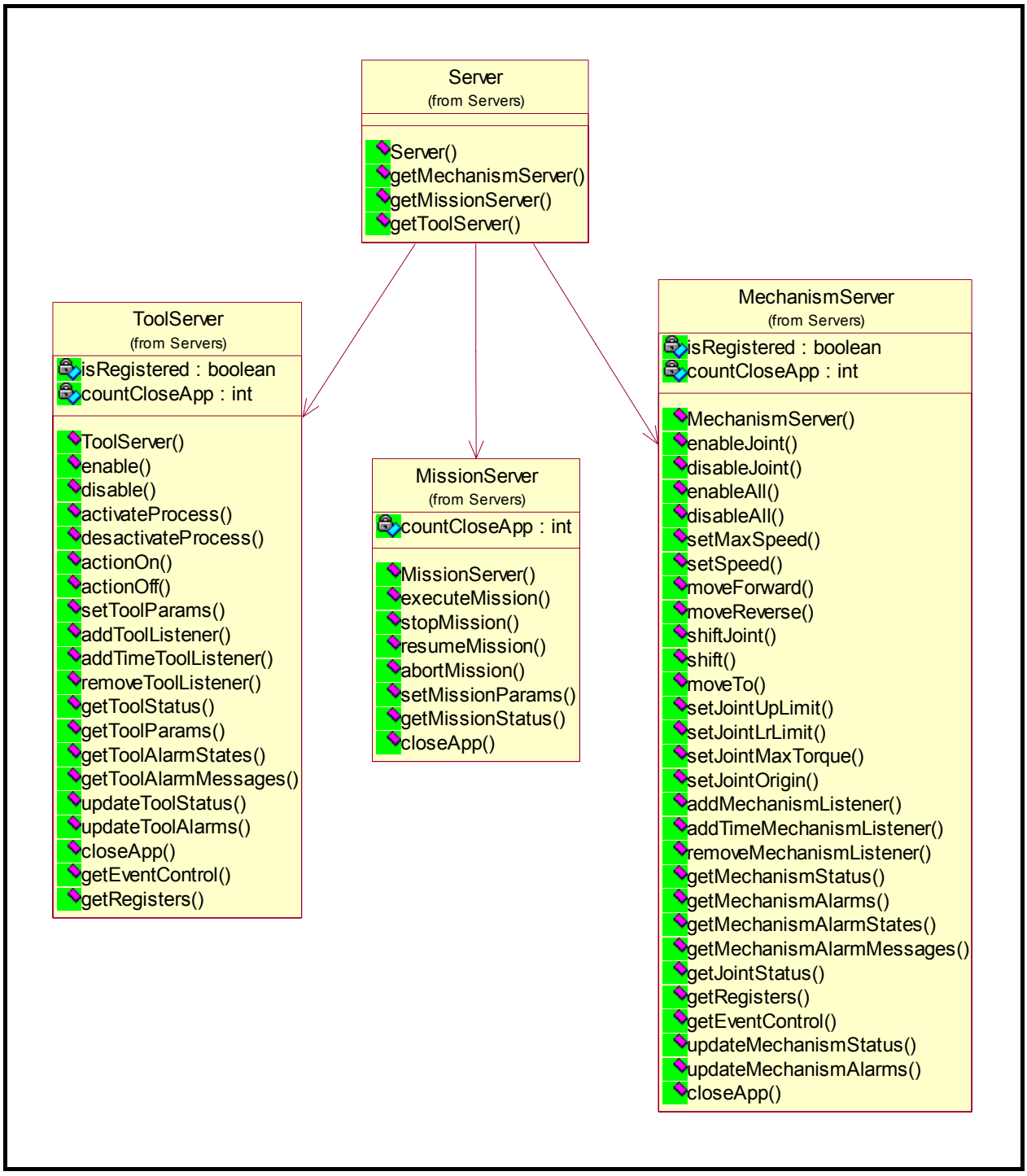


Ilustración 30: Diagrama de clase de Servers

MechanismServer

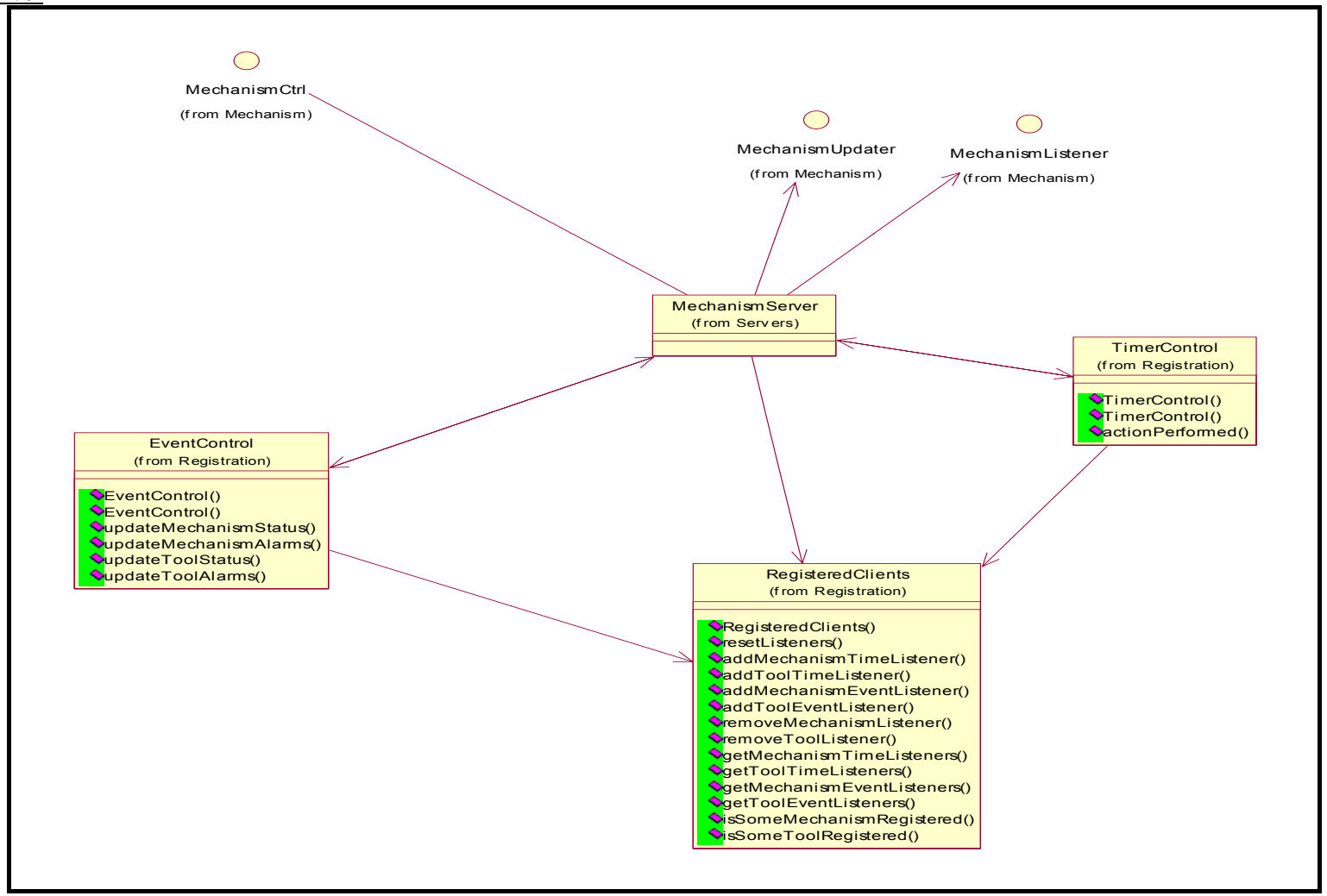


Ilustración 31: Diagrama de clase de MechanismServer

MissionServer

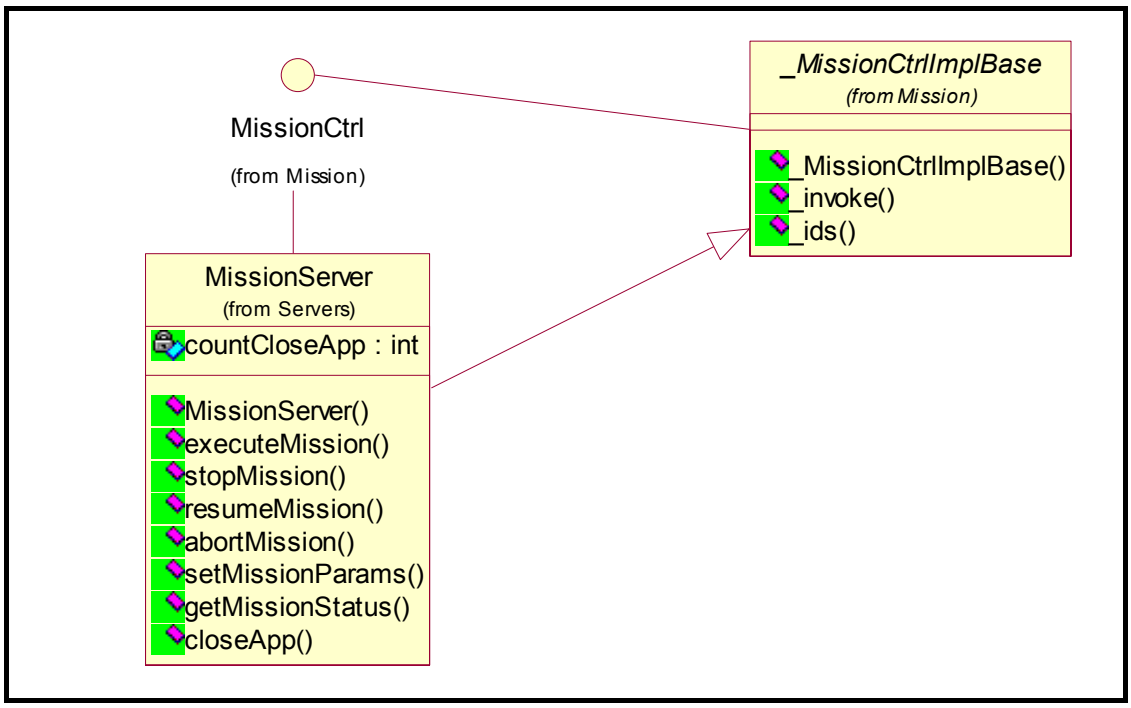


Ilustración 32: Diagrama de clase de MissionServer

ToolServer

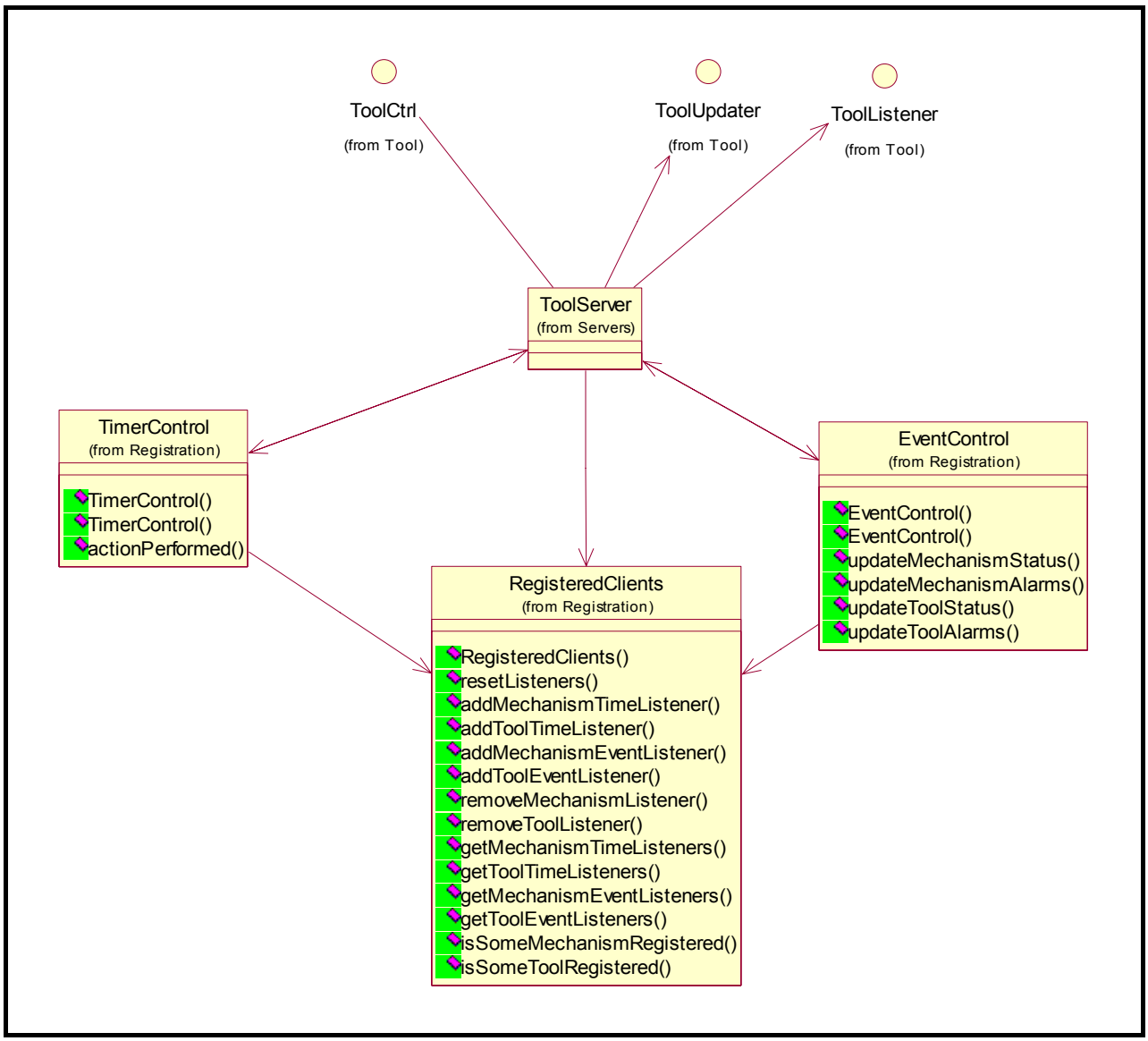


Ilustración 33: Diagrama de clase de ToolServer

Excepciones

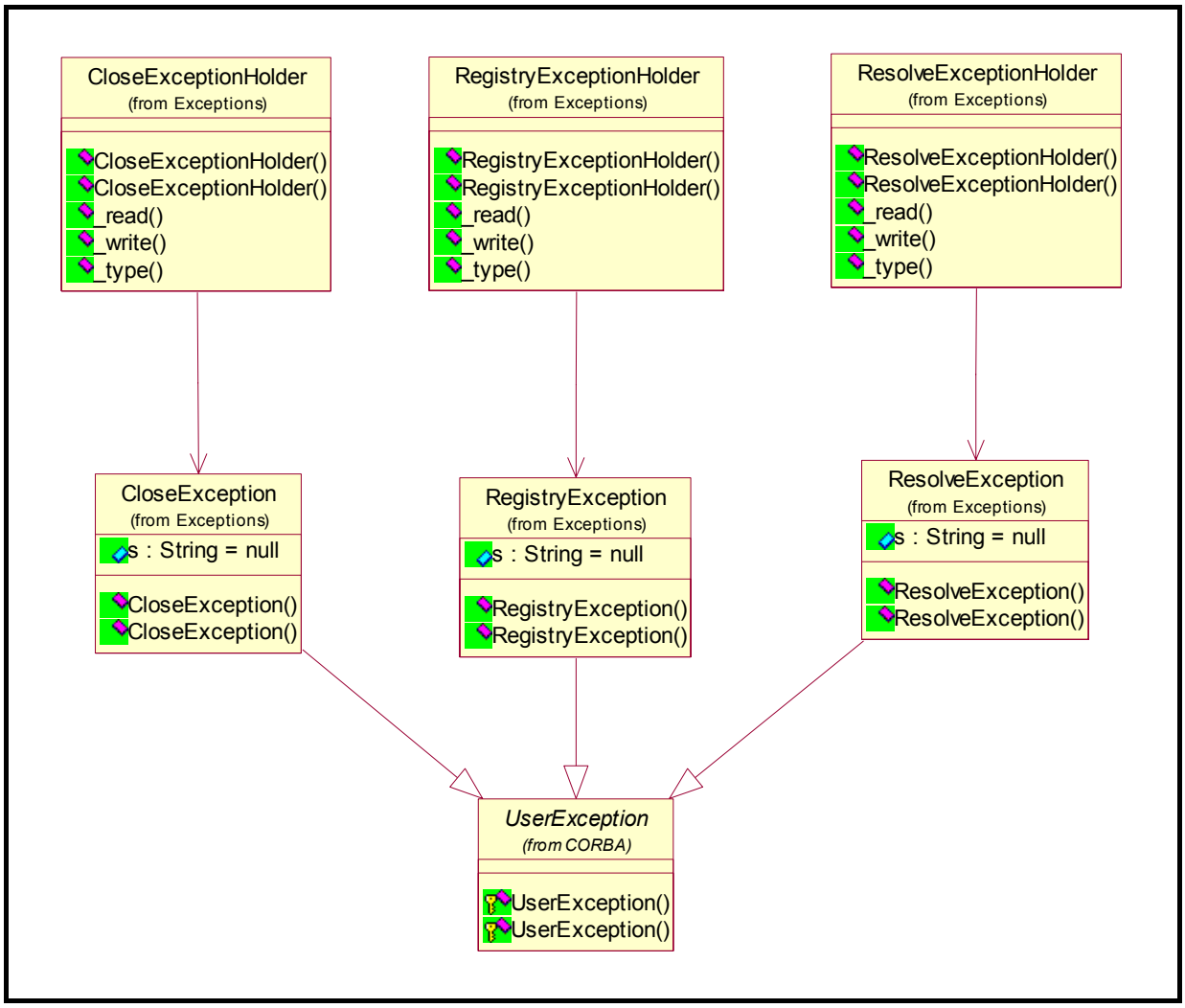


Ilustración 34: Diagrama de clase de Excepciones

6.4 Diagramas de Secuencia.

Inicio aplicación

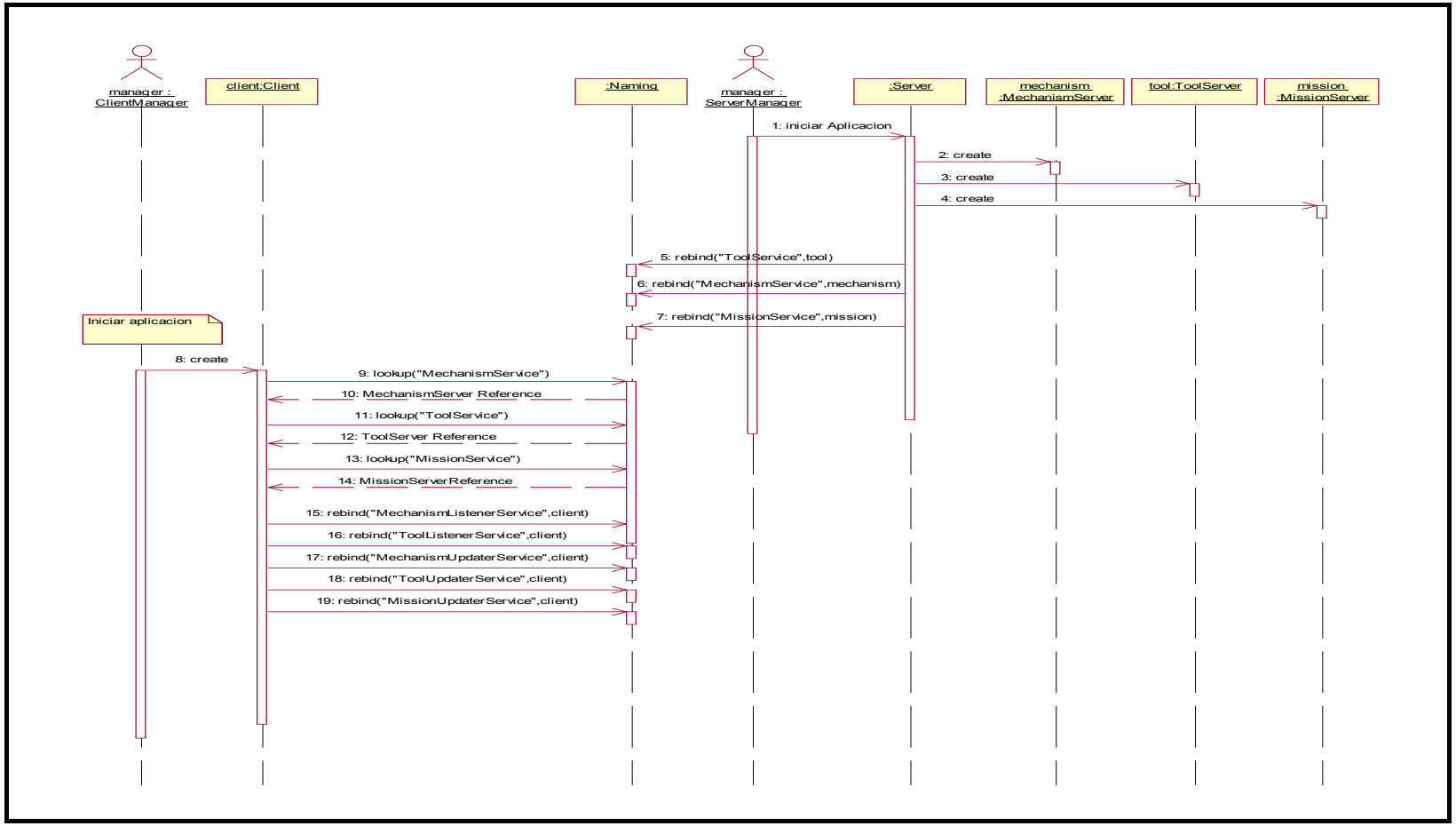


Ilustración 35: Diagrama de secuencia de inicio de aplicacion

Cierre aplicación

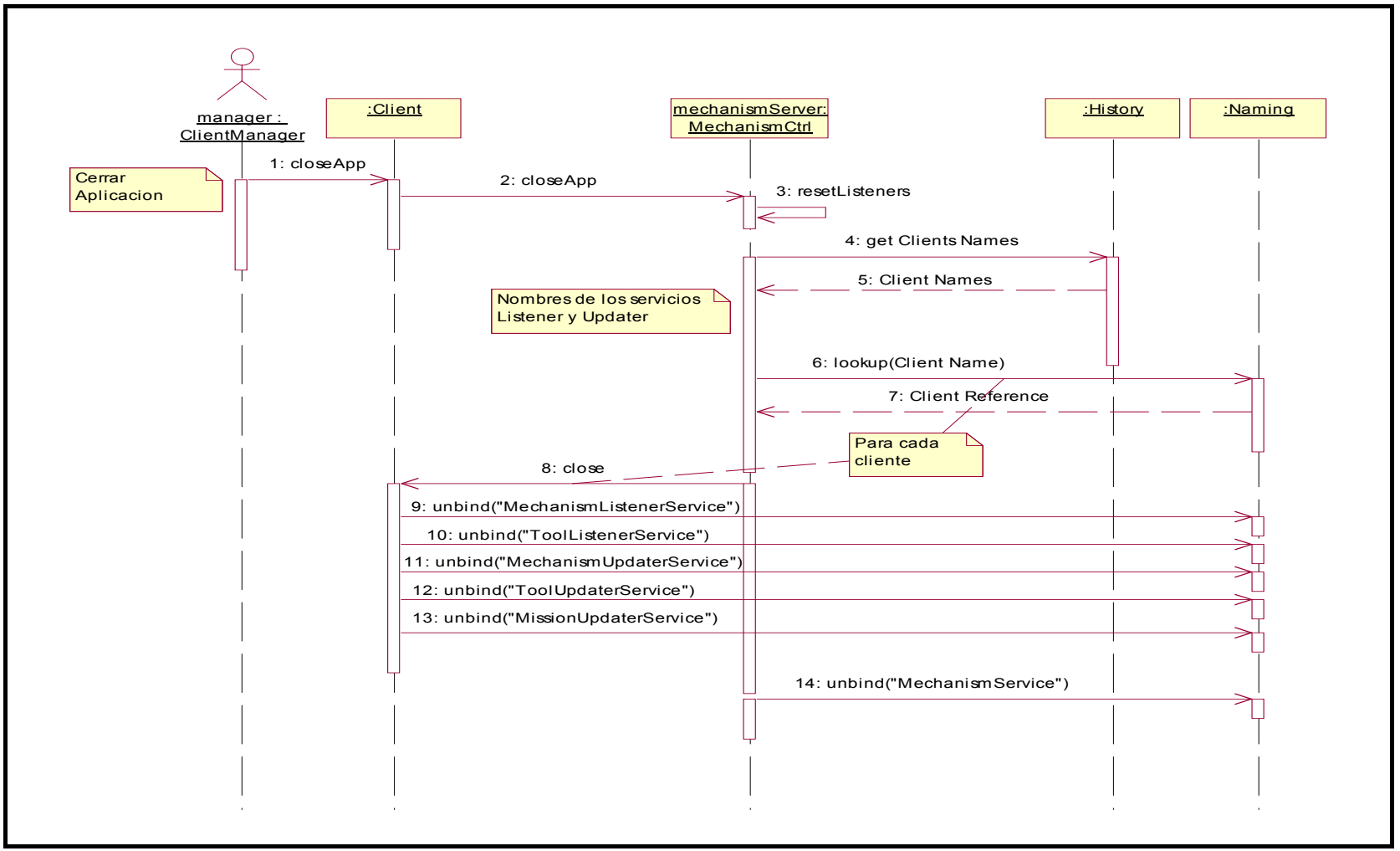


Ilustración 36: Diagrama de secuencia de cierre de aplicacion

Registro eventos

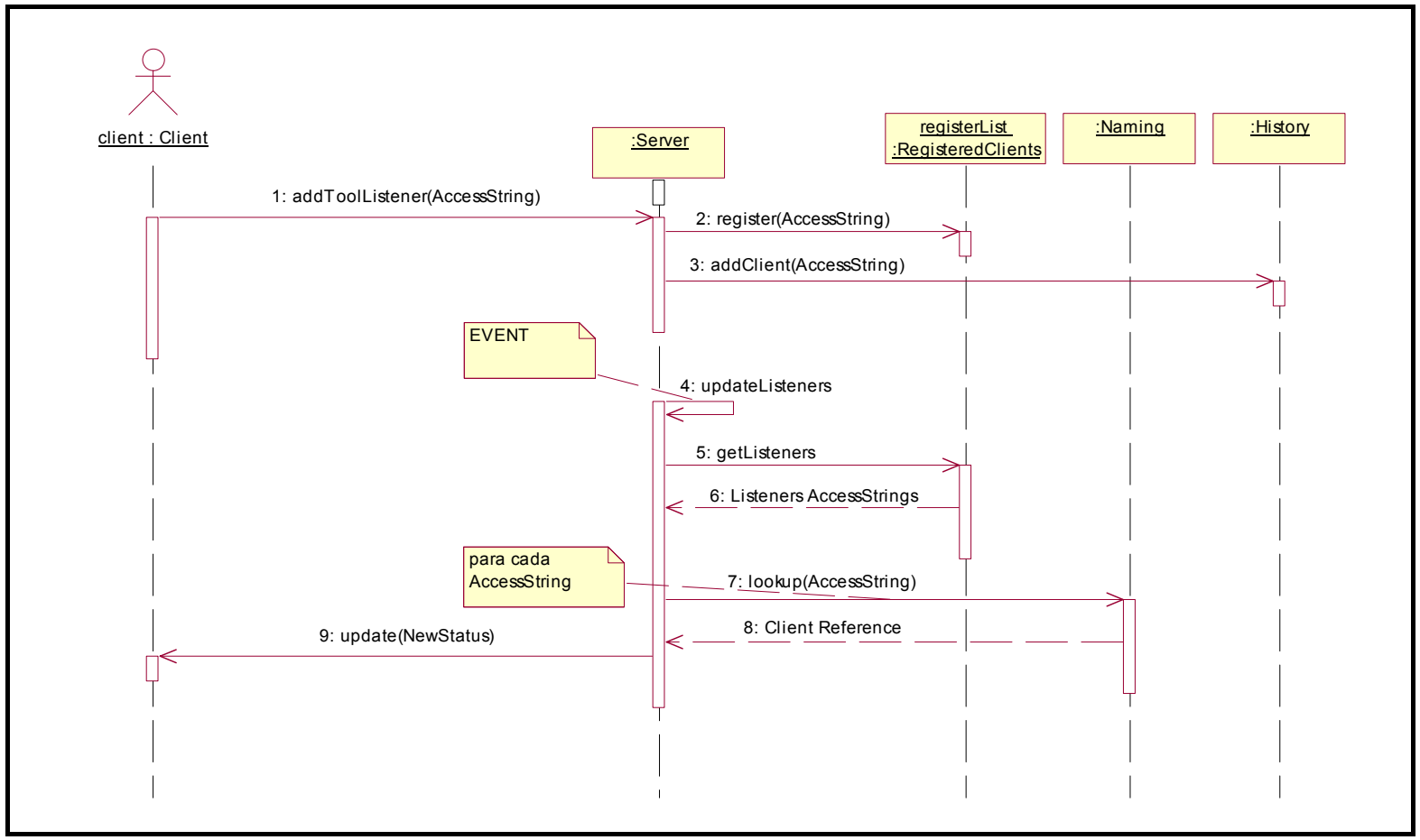


Ilustración 37: Diagrama de secuencia de registro mediante eventos

Registro eventos 1

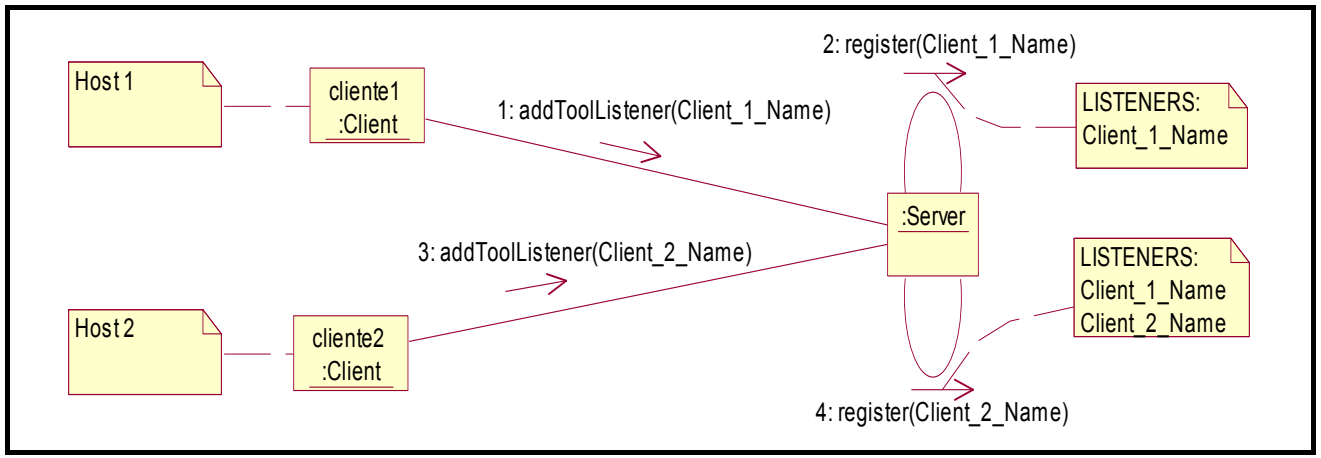


Ilustración 38: Diagrama de secuencia de registro mediante eventos

Registro eventos 2

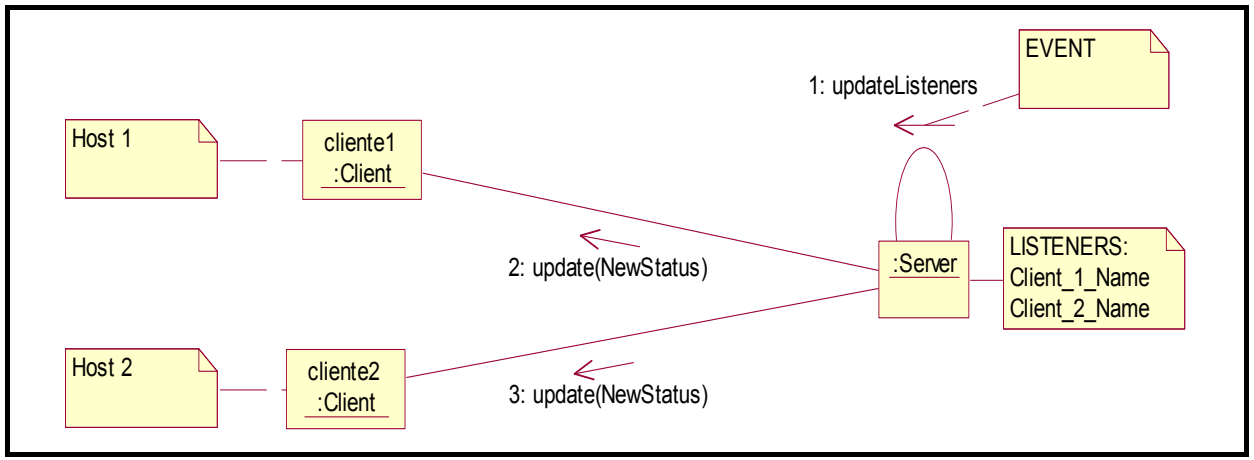


Ilustración 39: Diagrama de secuencia de registro mediante eventos

Registro tiempo

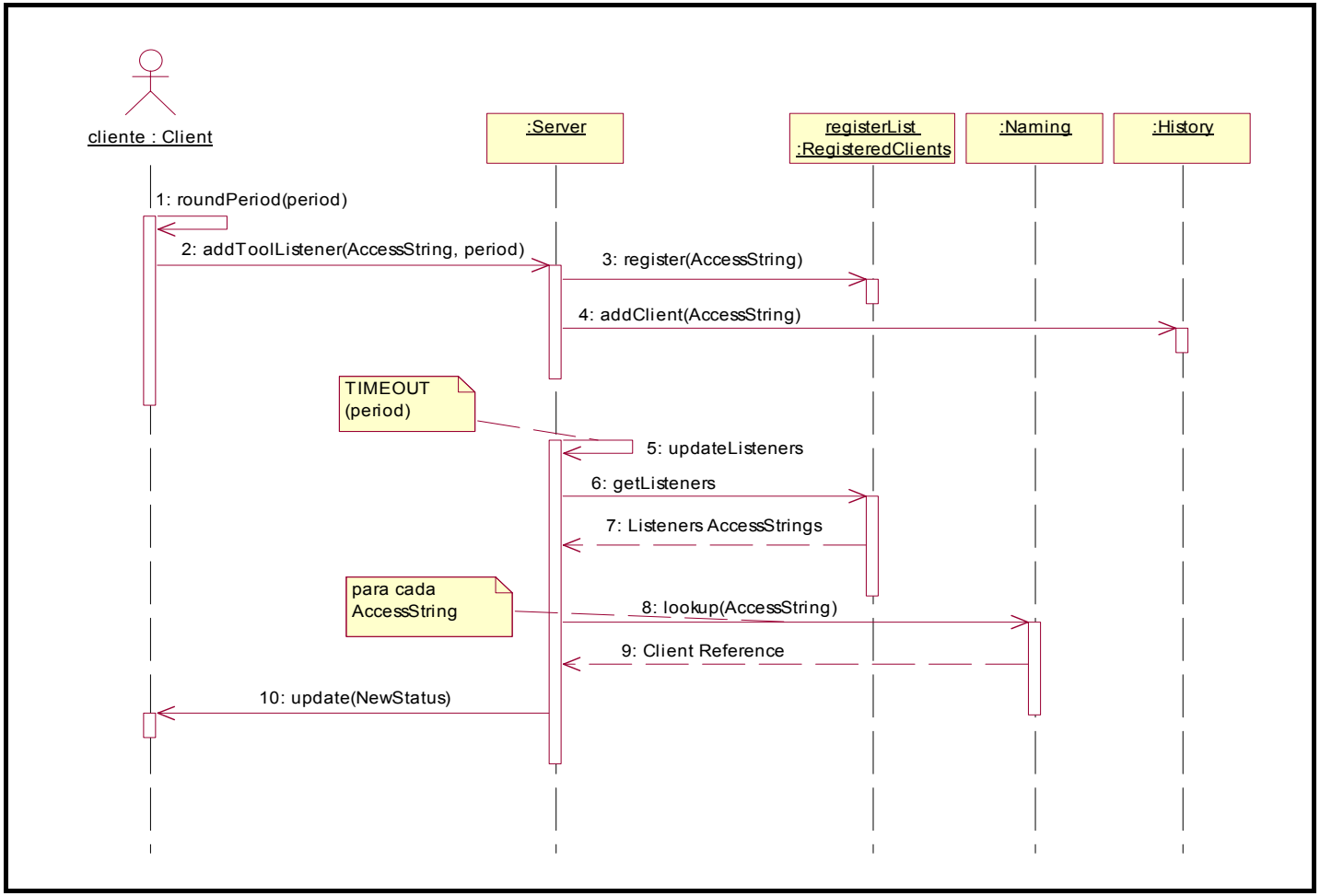


Ilustración 40: Diagrama de secuencia de registro mediante tiempo

Registro tiempo con 2 clientes

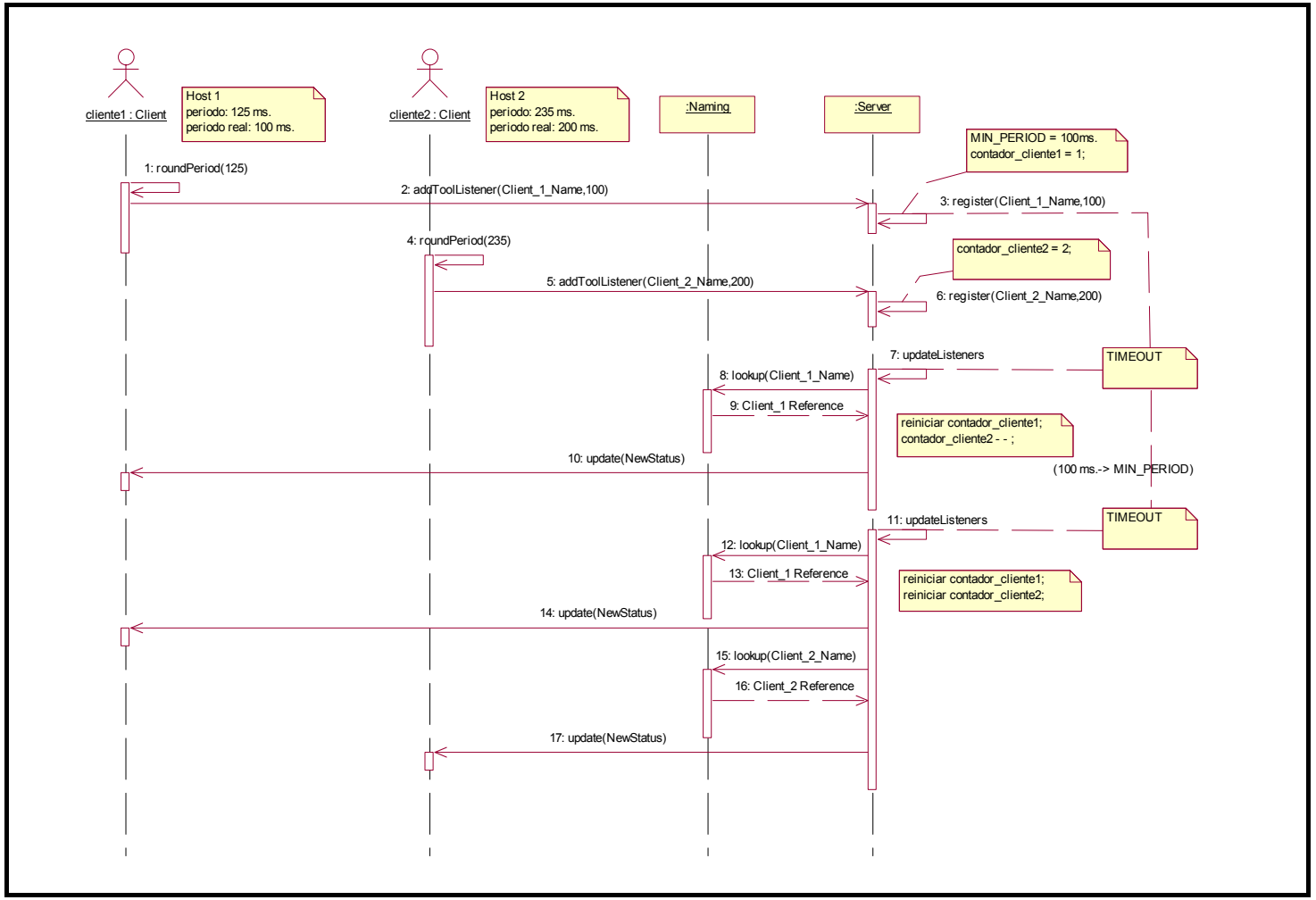


Ilustración 41: Diagrama de secuencia de registro mediante tiempo de 2 clientes

6.5. Diagramas de paquetes.

6.5.1. Diagramas de paquetes de la implementación RMI.

Client

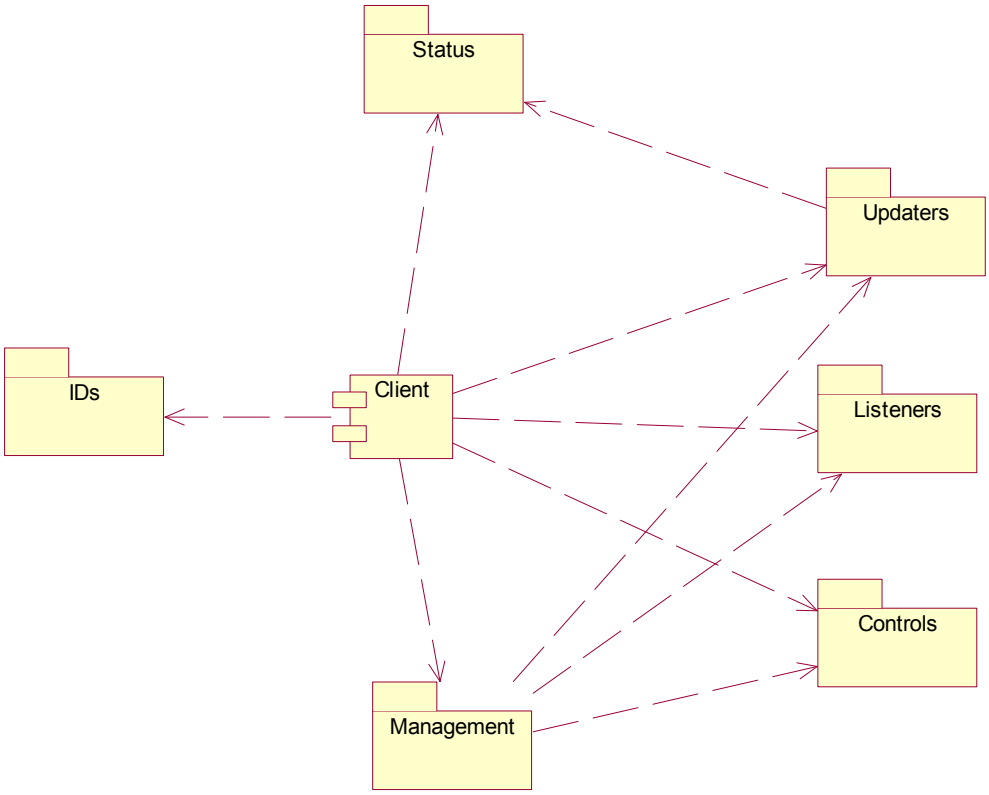


Ilustración 42: Diagrama de paquetes de Client

Servers

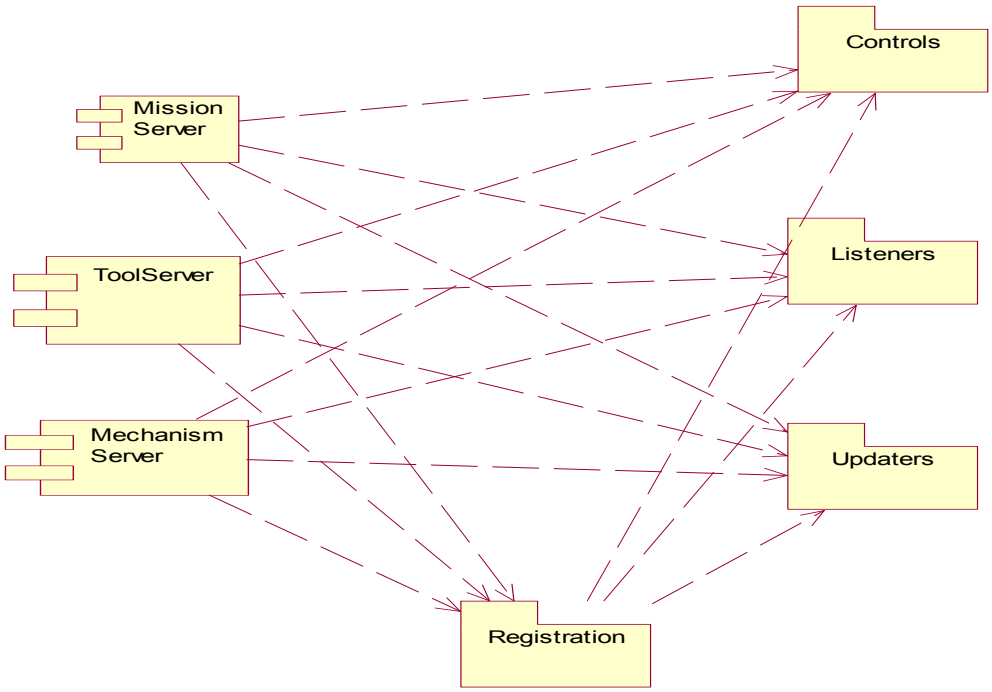


Ilustración 43: Diagrama de paquetes de Servers

Resto de componentes

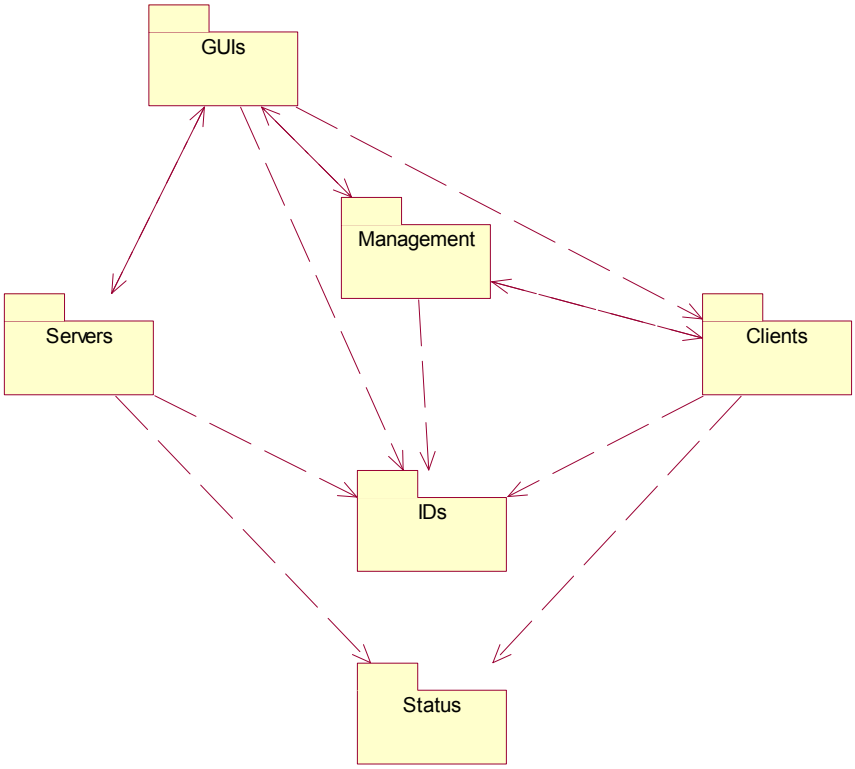


Ilustración 44: Diagrama de paquetes del resto de componentes

6.5.2. Diagramas de paquetes de la implementación CORBA.

Clients

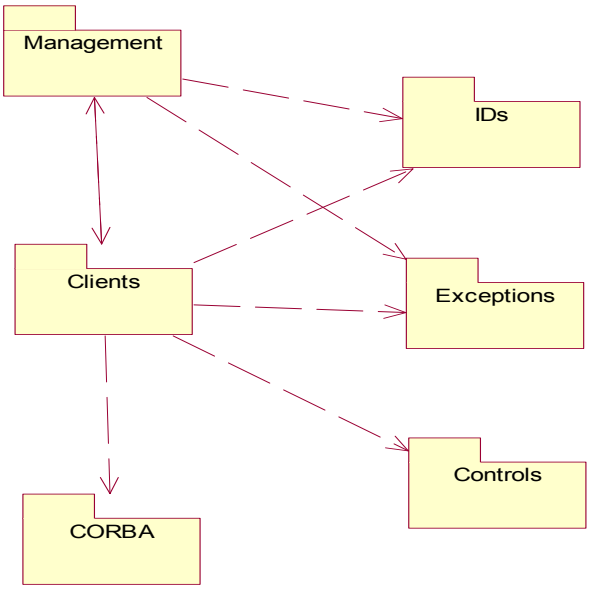


Ilustración 45: Diagrama de paquetes de Clients

Servers

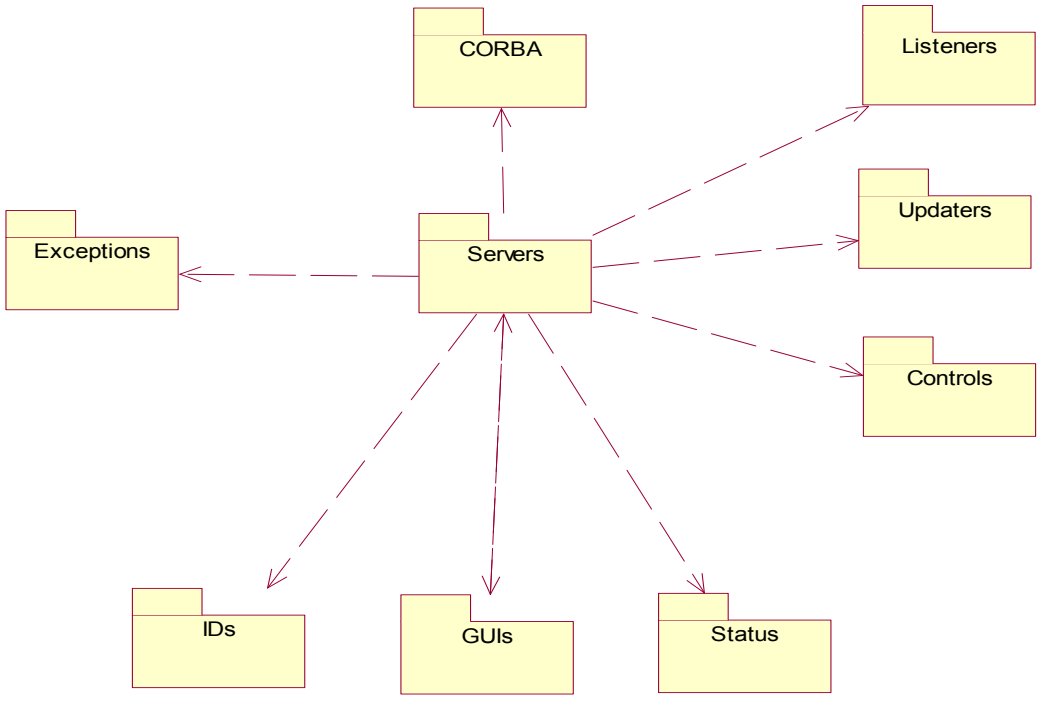


Ilustración 46: Diagrama de paquetes de Servers

Resto de componentes

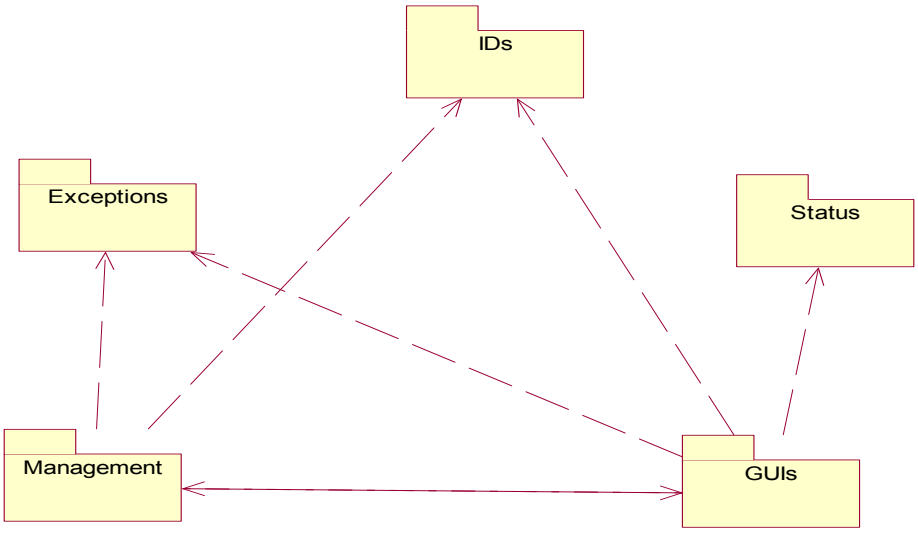
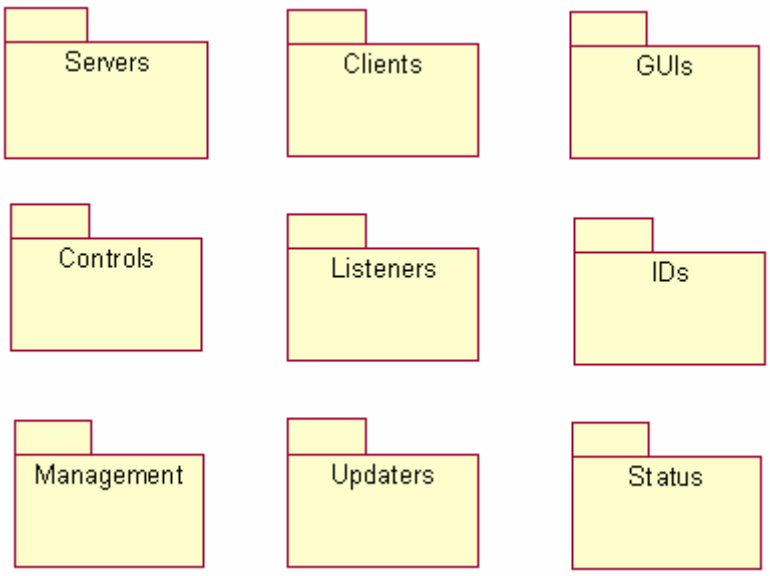


Ilustración 47: Diagrama de paquetes del resto de componentes

7. Composición de paquetes y clases.

7.1 Composición de los paquetes y clases de la implementación RMI.

Para la realización de la aplicación para la tecnología RMI, se optó por una distribución de clases en 9 paquetes: Controls, Clients, GUIs, IDs, Listeners, Managment, Servers, Status y Updaters.



Controls.

El paquete Controls contiene 3 clases: MechanismCtrl, MissionCtrl y ToolCtrl, que son las interfaces (vease Ilustración 22, 23 y 24) 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 (vease Ilustración 9); ocurriendo lo mismo para las interfaces MissionCtrl y ToolCtrl con los servidores MissionServer y ToolServer respectivamente.

Cada una de estas interfaces heredarán de la clase java.rmi.Remote, ya que se pretende que sean remotas, es decir, que se puedan exportan objetos de esos tipos y que puedan ser utilizados desde otras máquinas remotas. Así por ejemplo, si exportamos un objeto de un tipo que implemente la interfaz MechanismCtrl y lo "recogemos" en una máquina remota, desde ésa se podrán invocar remotamente a sus métodos. Como se puede observar, todos los métodos que deseen tener la propiedad de la invocación remota, es decir todos los métodos de estas interfaces, deben tener cláusulas donde se relancen las excepciones de tipo java.rmi.RemoteException, que representa la superclase que abarca todas las excepciones que pueden ocurrir durante la ejecución de llamadas remotas. Existen otras, que además de esa excepción pueden lanzar otras y por tanto se ha optado por relanzar las excepciones de la superclase Exception (de este modo se abarcan todas).

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.

Updaters.

Este paquete es muy similar al comentado anteriormente, en cuanto que también contiene 3 clases: MechanismUpdater, MissionUpdater y ToolUpdater y ambas son interfaces (Ilustración 27, 28 y 29). Además, estas interfaces también heredan de la clase java.rmi.Remote por el mismo motivo que ocurría en las clases del paquete Controls; y también, los métodos de éstas relanzan las excepciones de tipo java.rmi.RemoteException.

La principal diferencia de estas interfaces con las del paquete anterior, es que éstas definen métodos a utilizar por el cliente, y que por tanto, estas clases deberán ser implementadas por la clase Client (vease Ilustración 6), mientras que las interfaces del paquete anterior se utilizaban en los servidores.

En cuanto a la funcionalidad de los métodos que se describen en las interfaces, 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 interfaces, 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 Client (vease Ilustración 5). 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 se realiza de forma periódica a mediante la escucha de eventos. 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 (vease Ilustración 25 y 26); 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 y paneles que dotan al protocolo de una sencilla interfaz mediante la cual se puede probar. Las clases pertenecientes al protocolo son: MainGUI (interfaz principal mediante la cual se podrá dar inicio y fin a la comunicación y abrir un cliente nuevo), ClientGUI (frame 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 (frames encargados de simular la variación de los parámetros del robot).

IDs.

El paquete IDs solo contiene una clase, llamada ComIDs, que tan solo contiene 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 2 componentes: Alarm y GoyaJointStatus, 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. La clase Alarm es una interfaz que representa una alarma que se lanzaría en caso de urgencia o amenaza del robot. Para la utilización de éstas, sería necesaria la implementación de alguna alarma específica que implementará los métodos que se describen en la interfaz.

En cuanto al componente GoyaJointStatus, se puede observar como es otra interfaz pero que en este caso si tiene una implementación realizada en la clase GoyaJointStatusImpl. La utilidad de esta interfaz es que permite definir el estado de cada una de las articulaciones del robot. Por último destacar que la clase GoyaJointStatusImpl además de implementar esta interfaz, implementa también la interfaz Serializable para permitir así que se puedan transmitir objetos de este tipo por la red.

Management.

En este paquete se encuentran el conjunto de clases encargadas del manejo y control de la aplicación; 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, shutdownApp y openClient 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 ComIDs, 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.

Clients.

Este paquete contiene también solo 2 clases: Client (vease Ilustración 18) y RegistrationPeriod. En cuanto a la clase Client, hay que destacar que junto a las clases "servidores" es una de las clases más importantes. Tan solo analizando la sentencia inicial podemos descubrir su estructura y relación con las otras clases de la aplicación: "extends java.rmi.server.UnicastRemoteObject implements MechanismListener, ToolListener, Serializable, MechanismUpdater, ToolUpdater, MissionUpdater (vease Ilustración 4, 5, 6)". La sentencia extends implica que dicha clase hereda los métodos de UnicastRemoteObject, y esto es necesario ya que es necesario que la clase tenga comportamiento remoto, es decir, que objetos de este tipo puedan ser exportados a través de la red.

Las sentencias de implements nos indican también 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.

Analizando brevemente las asociaciones de composición que tiene esta clase con otras de la aplicación, vemos que la clase Client esta formada por 3 objetos de tipo servidor, o dicho de una forma mas correcta, por 3 objetos que representarán a los servidores de la maquina remota, y para poder representar a dichos servidores, estos objetos tienen que ser de la misma interfaz (MechanismCtrl, MissionCtrl y ToolCtrl).

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 clientes (clase RegistrationPeriod) dentro del paquete Clients y no dentro de Servers.Registration puede resultar un poco extraña. 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 ajusta y liberar al servidor el realizar otra operación más.

Servers.

Este es el último paquete de la aplicación, y por el hecho de englobar tanto a los servidores como a las clases encargadas del registro, se decidió hacer un nuevo subpaquete que perteneciera a este. En el paquete principal se encuentran tanto los servidores individuales: MechanismServer, MissionServer y ToolServer (vease Ilustración 31, 32, 33) como la clase Server 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.

La clase MechanismServer tiene una “sentencia de inicio” muy similar a la de la clase Client: “extends java.rmi.server.UnicastRemoteObject implements MechanismCtrl” y esto es debido a que tanto el cliente como el servidor tienen un comportamiento muy similar, diferenciando eso si, que cada uno trabaja en un extremo diferente de la comunicación y que implementan interfaces diferentes. Si recordamos el inicio de la clase Client, vemos como ambas heredan de UnicastRemoteObject, y es que, estos servidores también deben tener un comportamiento remoto, es decir, que también vamos a necesitar exportarlo a través de la red. También como ocurría en la clase cliente, se implementa una interfaz que en este caso corresponde con la que define el manejo del brazo del robot (MechanismCtrl).

Analizando en esta clase también las asociaciones de composición, vemos como al igual que en el cliente se tenían objetos que representarían al servidor, aquí la clase MechanismServer también va a estar formada por objetos que representan al cliente, pero en este caso, no es solo un objeto, sino que son 2 objetos: uno de tipo MechanismListener y otro MechanismUpdater, correspondiendo cada uno con las interfaces que definían los métodos del cliente. De este modo, se divide el comportamiento del cliente en 2 partes; así, algunos métodos del cliente se implementarán con el objeto MechanismListener y otros con el otro objeto. Además de estos objetos, la clase MechanismServer estará formada por un conjunto de objetos de tipo

EventControl, TimerControl, RegisteredClients… que permitirán la implementación de un registro particular 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 de 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 Server. Como ya se dijo anteriormente, esta clase se comporta como un contenedor de los servidores individuales y como estos servidores pueden atender a varios clientes, no tiene sentido la existencia de varios objetos de tipo Server en la misma aplicación por lo que esta clase será estática.

En cuanto a los métodos de esta clase, como ya no hay necesidad de implementar ninguna interfaz, los únicos métodos de esta clase, serán métodos "get" para acceder a los servidores.

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, History, HistoryObject, 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 para que sirve cada clase.

Dentro de este paquete podemos distinguir clases que se encargan de dos tipos de registros de clientes; por un lado existe un registro de clientes al que llamaremos historial que será el encargado de almacenar todos los clientes que pasen por la aplicación, sin importar si ya han sido cerrados. Este registro que se implementa fundamentalmente con las clases History y HistoryObject, permitirá que a la hora de cerrar la aplicación se puedan o se intenten cerrar todos estos clientes. Por otro lado, el otro registro será el encargado de almacenar los clientes que se registran para recibir la actualización de variables mediante eventos, y también (de forma separada) y también para los de la actualización de forma temporal. La utilidad de este último registro es incuestionable ya que cada vez que se produzca un evento o cada vez que expire un

periodo, habrá que dirigirse al registro que corresponda para invocar algún método remoto de los clientes ahí almacenados. Las clases encargadas de estas acciones serán las restantes del subpaquete.

Las clases History y HistoryObject están íntimamente relacionadas, ya que la primera de ellas es un registro (elemento de tipo Vector) en el que se almacenarán objetos de tipo HistoryObject. Estos objetos están formados por 2 elementos: el primero de ellos es un String que representa la cadena de acceso que permitirá importar estos clientes desde los servidores remotos; además, a la hora de importar estos hay que hacer una operación de cast, y para esto, debemos conocer bien si el cliente se trata de un objeto de tipo "listener" o "updater", para hacer por ejemplo el cast a un objeto MechanismListener o a MechanismUpdater. Para aclarar esta característica en cada cliente, se almacena también en el objeto HistoryObject un identificador que nos permita hacer esta distinción, que contendrán una cadena de acceso que permita identificar y acceder a cualquier cliente. En cuanto a la clase History, se limitará a disponer de un vector en el que almacenar los objetos HistoryObject y de un método para añadir clientes, en el que antes de hacer cada registro, se verificará si no está ya almacenado el cliente. Esta clase no dispone de método de eliminación de clientes porque no hay que olvidar que la utilidad de este registro no es otro que el de almacenar todos los clientes que pasen por la aplicación.

Para la implementación del otro tipo de registro de clientes, se necesitan de muchas más clases: EventControl, 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 vez que ocurra un 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 periodo 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 1: La utilización de vectores en vez 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 acceder a sus elementos.

Nota 2: Si se ha observado con detenimiento la implementación de la aplicación, podemos encontrar que en muchas clases aparece el método finalize. Este no ha sido comentado anteriormente ya que éste no se puede englobar en el mismo "grupo" que el resto. El comportamiento de este método, no es similar al resto, ya que éste no es necesario invocarlo, sino que se invoca de forma automática. Antes de terminar la ejecución de cualquier aplicación, se produce una llamada a este método (si existe).

Es por esto, que se puede aprovechar para implementar en éste acciones que queremos que se realicen al cerrar la aplicación. Si además nos encontramos ante una aplicación de comunicaciones en la que se han abierto enlaces y flujos con el otro extremo, su utilización se puede encontrar muy útil, ya que se puede aprovechar para cerrar estos enlaces y flujos.

7.2 Composición de los paquetes y clases de la implementación CORBA.

Como es de suponer, la implementación para esta tecnología es muy similar a la realizada para la otra tecnología y por tanto la organización de las clases también será similar; y es que sin contar las sentencias propias de cada tecnología como son en las que se invocan a los métodos lookup y rebind o cuando se trabaja con el ORB, el resto de sentencias son casi equivalentes.

De este modo, los paquetes y las clases que contienen son iguales, destacando eso si, que debido a que es la que utiliza la tecnología más potente y que además fue la primera implementación realizada, se decidió también implementar un conjunto de excepciones que aclararan la situación en la que se produjo la situación anómala. Es por esto, por lo que la implementación CORBA dispone de un paquete más (Exceptions) que la de RMI.

Si comenzamos analizando los paquetes que tan solo incluían las interfaces en las que se definían los métodos de los servidores y clientes, es decir, **Updaters, Listeners** y **Controls**, vemos como dentro de estos, no se encuentran las interfaces, sino que hay otros subpaquetes y en éstos, es donde se encuentran dichas interfaces. Esto es debido a que al utilizar la tecnología CORBA, las interfaces se deben implementar en IDL y por tanto después, se han de volver a pasar a lenguaje Java. Esta conversión, como ya se comentó anteriormente se realiza mediante una utilidad que provee JDK (idlj) pero ésta, además de crear la oportuna interfaz en lenguaje Java, también crea otros ficheros en ese lenguaje como son: _nombreInterfazImplBase.java, _nombreInterfazStub.java, nombreInterfazHelper.java, nombreInterfazHolder.java y nombreInterfazOperations.java. La utilidad de cada uno de estos ficheros queda detalladamente explicada en los puntos anteriores dedicados a este hecho. Además de estas clases, la propia utilidad idlj también creará otro subpaquete dentro de éste, donde se incluirán los ficheros "holder" y "helper" de cada uno de los tipos no primitivos (incluidos arrays) que se usen en las interfaces; así, como en la interfaz MechanismUpdater, se utilizan arrays de booléanos, existe un subpaquete (Updaters.Mechanism.MechanismUpdaterPackage) donde se encuentran los ficheros Java "arrayOfBooleanHelper" y "arrayOfBooleanHolder".

Detallado esto, queda aclarada la existencia de los siguientes subpaquetes dentro del paquete Updaters: Mechanism, MechanismUpdaterPackage, Tool, ToolUpdaterPackage y Misión. Como se puede razonar, en la interfaz MissionUpdater no se usan ni arrays ni tipos no primitivos.

Como en esta implementación lo único que se ha modificado es la tecnología utilizada, la interfaz con el usuario sigue siendo la misma, por lo tanto, las clases implicadas en esta interfaz seguirán siendo las mismas, formarán el mismo paquete **GUI** que en la implementación RMI.

El paquete **Status** en principio debería ser igual al de la otra implementación, pero si se razonan los párrafos anteriores, nos daremos cuenta como, debido a que en éste aparecían interfaces cuyos objetos (de sus implementaciones) deben ser enviados a través de la red, éstas deberán ser escritas en IDL y por tanto después se formará un subpaquete con los ficheros en

Java creados a partir de esa interfaz. Es por esto, que el paquete contiene prácticamente las mismas clases (se han añadido otras de complemento pero que no tienen importancia), salvando los subpaquetes de cada una de las interfaces que se encuentran en el paquete. Así, tendremos dentro de este paquete los siguentes subpaquetes: Alarm, BlastingStatus, GoyaJointStatus y GoyaStatus.

En cuanto al paquete **Management**, vemos como es similar al de RMI, y es que aunque éste también incluya en su interior una interfaz (AppManager), como no van a haber objetos de este tipo para enviar por la red, no habrá que escribir esta interfaz en IDL. Como es obvio, la implementación de sus clases y su utilidad será la misma que en RMI.

Otro paquete muy sencillo de comentar por su similitud con el de RMI es el **IDs**, y es que, al igual como ocurría anteriormente, tan solo contiene la clase ComIDs, que será la clase estática a la cual se tendrá que acceder para obtener los identificadores de los clientes y servidores.

El paquete **Servers** también es completamente similar al de la implementación RMI, y además para destacar la reutilización de código, se puede comprobar como el subpaquete **Registration,** que se ocupaba de todo lo relacionado con el registro de clientes, es el mismo que el de RMI, con la salvedad de que en unos pocos métodos (como es el caso de updateToolStatus de la clase EventControl) en los que había que introducir como parámetro un "Object" ahora se ha de colocar un objeto pero de CORBA: "org.omg.CORBA.Object"; exceptuando esos casos, el resto del código se puede reutilizar, y tiene la misma utilidad.

En cuanto a las clases del propio paquete Servers (vease Ilustración 30), seguirán haciendo la misma función de servidores, pero ahora utilizando la tecnología CORBA, es decir, que a la hora de recoger los clientes y publicarse ellos mismos para que le puedan hacer invocaciones remotas los propios clientes en vez de utilizar los métodos de la clase Naming propios de RMI, ahora en cambio utilizarán otros como son los de las clases clase NamingContext o NamingContextHelper.

La clase Server actuará como contenedor de los servidores individuales, y es por esto por lo que tendrá objetos de tipo: MechanismServer, MissionServer y ToolServer. Además de estos, esta clase también estará formada por un objeto de tipo ORB, que representará al propio ORB de la arquitectura CORBA. El motivo por el que este objeto se encuentra aquí es debido a

que al iniciar el objeto Server, se creará e inicializará el ORB, y además, después se pasará como parámetro este mismo objeto a cada uno de los servidores, para que cada uno de estos puedan utilizar el ORB a su gusto (ya sea tanto para publicar como recoger objetos).

Las ultimas clases a analizar en este paquete serán las de los "servidores individuales": MechanismServer, MissionServer y ToolServer; la composición de las clases es la misma. Cada una de estas, además de implementar las interfaces correspondientes del paquete Controls, extenderán de la clase esqueleto de las interfaces anteriores (para la interfaz MechanismCtrl, la clase extenderá de _MechanismCtrlImplBase). Este esqueleto será una clase abstracta puesto que su funcionalidad es únicamente servir como clase base de otra, escrita por nosotros, en la que realmente se implementen los métodos de la interfaz MechanismCtrl. (Si se desea más información sobre las clases autogeneradas por el compilador de CORBA, por favor, diríjase al Anexo 1, donde se detallan cada una de estas clases).

Además de los objetos que ya aparecían en la implementación RMI, como son los de tipo EventCtrl, TimerCtrl..., las clases servidores (por ejemplo MechanismServer) disponen también de objetos Listener y Updater (MechanisListener y MechanismUpdater) que representarán a los clientes. Además de estos, se necesita también de un objeto de tipo NamingContext propio de la tecnología CORBA que nos permitirá obtener el Servidor de Nombrado (necesario para exportar y recoger objetos).

El paquete que mas se ha visto modificado es: **Clients,** que ha sufrido una gran transformación ya que ha pasado de tan solo contener 2 clases (Clients y RegistrationPeriod) ahora además contiene otras: MechanismListenerClient, MechanismUpdaterClient, MissionClient, MissionUpdaterClient, ToolClient, ToolListenerClient yToolUpdaterClient. El motivo por el que ahora existen tantas clases es el siguiente:

El mecanismo de creación de una aplicación CORBA por herencia, a diferencia del mecanismo de creación por delegación, obliga a la implementación del objeto que se está desarrollando a derivar de un esqueleto generado a partir del interfaz IDL. La implementación de un objeto distribuido en Java requiere que se herede de la clase *skeleton*. Esto supone una limitación para la programación de los objetos distribuidos, ya que Java utiliza herencia simple.

Este problema se presento en la creación del cliente como objeto CORBA, ya que el la clase Client debía implementar las interfaces MechanismListener, MechanismUpdater, ToolListener, ToolUpdater y MissionUpdater. Para cada interfaz, el cliente debería heredar la clase esqueleto correspondiente (generadas por la compilación de las interfaces anteriores en

IDL). Sin embargo, como Java no permite la herencia múltiple, se tuvo que implementar el cliente de la siguiente manera:

El cliente se separa en tantos tipos de clientes como interfaces debía implementar, y se crean de este modo las siguientes clases:

MechanismListenerClient	→ implementará MechanismListener
MechanismUpdaterClient	→ implementará MechanismUpdater
ToolListenerClient	→ implementará ToolListener
ToolUpdaterClient	→ implementará ToolUpdater
MissionUpdaterClient	→ implementará MissionUpdater

De este modo, cada cliente anterior deberá heredar únicamente del sirviente correspondiente a la interfaz que implementa. Finalmente se creo una clase que encapsulara los clientes anteriores según sean de tipo Mechanism, Tool o Misión.

Analizando cada una de las clases del paquete y comenzando por la clase Client, podemos comprobar como existe una gran similitud con la composición de la clase Server, ya que si esta actuaba como contenedor de los servidores individuales, ahora la clase Client también actuará de contenedor para los clientes individuales (MechanismClient, MissionClient y ToolClient). Análogamente, esta clase también dispondrá por el mismo motivo que ocurría para Server, un objeto de tipo ORB que también se pasará como parámetro en la inicialización de los clientes individuales.

Como ya se ha comentado anteriormente, ahora cada cliente está representado por un conjunto de 2 clases, de tipo Updater y Listener (MechanismListenerClient y MechanismUpdaterClient para el componente Mechanism). Pues bien, ahora estas clases estarán también englobadas en un contenedor (clase MechanismClient). Este contenedor dispone de todos los objetos necesarios para la captura de los servidores y poder exportarse él mismo; así, estará formado por objetos de tipo: org.omg.CORBA.Object, NamingContext, NameComponent….

Las últimas clases a comentar serán las que representan a los clientes. Al igual como ocurría para los servidores individuales, estas clases implementarán interfaces de tipo Updater o Listener (según corresponda) y además extenderán de los esqueletos de estas interfaces (_MechanismListenerImplBase para la interfaz MechanismListener). Estas además, también estarán formadas por un objeto de tipo NamingContext que contenga el servidor de nombrado, ya que puede darse el caso en el que estos clientes deseen recoger o exportar objetos.

El último paquete a comentar es **Exceptions** y, como se puede pensar, contiene las nuevas excepciones que se han creado. En total son 3: CloseException, RegistryException y ResolveException. Como se puede comprobar, para cada una de estas existen dos clases más: "holder"y "helper", y si recordamos lo que ocurría con las interfaces escritas en IDL, podemos comprobar como las propias excepciones también están escritas en IDL, y las clases en Java que aparecen han sido autogeneradas por la aplicación idlj. El hecho de generar las excepciones mediante IDL, es debido a que mientras se definen las interfaces, si algunos de sus métodos lanzan una excepción, ésta debe ser reconocida por cualquier lenguaje de programación (por supuesto, dentro de los que soporte CORBA); o también se puede dar el hecho que se defina alguna nueva excepción para definir mejor la situación en la que se ha producido la situación anómala.

Solo con el nombre las excepciones podemos entender la utilidad de cada una de ellas: CloseException se lanzarán en caso de que ocurra alguna situación anómala a la hora de cerrar; RegistryException para situaciones a la hora del registro y ResolveException para situaciones anómalas a la hora de recoger los "clientes remotos".

9. Comentario de los aspectos más conflictivos e importantes de la implementación.

Aunque todos estos detalles han sido debidamente comentados mediante diagramas de secuencia, se ha decidido comentar de nuevo los puntos más importantes de la implementación, para que de este modo no quede ninguna duda sobre el funcionamiento de la aplicación. En principio estos aspectos solo se detallan para la implementación realizada con la tecnología RMI.

9.1. Pasos iniciales y establecimiento de conexiones.

Al utilizar la Llamada Remota de Métodos (RMI), la forma en la que el servidor y los clientes crean los enlaces para comunicarse queda muy simplicada (vease ilustración 35). Como ya se comentó anteriormente, todo el proceso de establecimiento de conexión se limita a la recogida por parte de uno de los extremos de los objetos remotos que se exportan desde el otro extremo de la comunicación.

Los pasos a seguir en el proceso cliente (Client) y servidor (Server) son muy similares, ya que en ambos lo único que hay que hacer es, además de controlar que exista algún gestor de seguridad (SecurityManager), publicar y recoger los objetos necesarios para que cada uno de estos extremos pueda acceder a todas las características del otro extremo.

Analizando en primer lugar el proceso cliente (clase Client), se observa como contiene 3 objetos de tipo MechanismCtrl, MissionCtrl y ToolCtrl que representan todas las interfaces que definen las funciones del servidor; mediante estos 3 simples objetos se podrá tener un control total sobre el servidor y cada una de sus características. Antes de hacer la asignación de estos objetos y como ya se comenta en puntos anteriores es necesario comprobar si hay algún gestor de seguridad activado, esto se consigue mediante:

```
if (System.getSecurityManager()==null)
        System.setSecurityManager(new RMISecurityManager());
```

Tras esto, se recogerán los 3 objetos que anteriormente el servidor ha publicado. Para ello se utiliza el método lookup de la clase estática Naming que permite la "captura" de un objeto remoto mediante la introducción de una cadena de caracteres donde se especifique la dirección IP de la máquina y un alias del objeto remoto; este alias es necesario ya que en una misma máquina se pueden publicar tantos objetos como se deseen. Así por ejemplo, mediante la sentencia:

```
mission = (MissionCtrl)Naming.lookup(ComIDs.MISSION_SERVER_ACCESS_STRING)
```

se consigue recoger un objeto de tipo MissionCtrl. La constante introducida como parámetro es: "rmi://" + SERVER_IP + "/ServiceOfMissionServer" donde SERVER_IP representa la dirección IP de la máquina servidor y "ServiceOfMissionServer" será el alias del objeto publicado.

Mediante esta sentencia se recogerá un objeto de cada interfaz: MechanismCtrl, MissionCtrl y ToolCtrl, cambiando en cada una de ellas el "identificador de objeto": MECHANISM_SERVER_ACCESS_STRING, MISSION_SERVER_ACCESS_STRING y TOOL_SERVER_ACCESS_STRING; además se tendrá que cambiar tambien la interfaz con la que se realiza el "cast".

Tras la recogida de los objetos, el cliente procederá a su publicación a través de la red. Pero su publicación no es tan sencilla como la de cualquier objeto ya que la clase Client implementa un gran numero de interfaces: MechanismListener, ToolListener, MechanismUpdater, ToolUpdater, MissionUpdater cuyos métodos se pretenderán utilizar

mediante llamadas remotas. Es por esto por lo que se tendrá que publicar un objeto por cada una de estas interfaces; así tendremos las siguientes sentencias:

Naming.rebind(ComIDs.MECHANISM_LISTENER_ACCESS_STRING,this);
Naming.rebind(ComIDs.TOOL_LISTENER_ACCESS_STRING,this);
Naming.rebind(ComIDs.MECHANISM_UPDATER_ACCESS_STRING,this);
Naming.rebind(ComIDs.MISSION_UPDATER_ACCESS_STRING,this);
Naming.rebind(ComIDs.TOOL_UPDATER_ACCESS_STRING,this);

Como se puede observar en estas sentencias, para la publicación se utiliza también un método de la clase estática Naming, pero en este caso se usa "rebind" que permite la publicación de un objeto. Se podría utilizar también el método "bind" de la misma clase, pero gracias a utilizar el primero de ellos, si existiera un objeto con el mismo nombre en vez de lanzar la excepción NotBoundException como ocurriría con bind, éste sería reemplazado. A partir del hecho de que se han realizado tantas publicaciones como interfaces tenía el cliente, ahora el servidor, cuando desee recoger al cliente, tendrá que recogerlas todas, es decir, que tendrá que tener un objeto para cada una de esas interfaces a los cuales se les asignará el resultado de los métodos lookup.

Como primer parámetro de este método hay una cadena de caracteres que sigue el mismo esquema que el introducido en el método lookup, es decir, que es una cadena que contiene un alias del objeto y la dirección IP de la máquina donde se encuentra dicho objeto y que permitirá poder identificar este objeto. Como segundo parámetro de este método habría que introducir el objeto a exportar; éste será el mismo para las 5 sentencias debido a que el objeto Client, que es el que se introduce, es un objeto de tipo MechanismListener, ToolListener, MechanismUpdater, ToolUpdater, MissionUpdater por implementar todas estas interfaces.

Además de todos los métodos de implementación de estas interfaces, la clase Client debe tener todos los métodos de las acciones que pueda realizar el servidor, para que de este modo, cuando se desee realizar una llamada a un método del servidor, se llamará al método homónimo de la clase Client. La implementación de estos métodos será muy sencilla ya que solo habrá que pasar la llamada con los mismos parámetro al objeto servidor correspondiente según sea su funcionalidad, así por ejemplo, si el cliente desea que la herramienta habilite una articulación, se llamará al método enableJoint(int jointNumber) de la clase Client y éste a su vez llamará al mismo método pero del objeto MechanismCtrl que la parte remota del servidor encargada de esas funciones: mechanism.enableJoint(jointNumber).

Lo primero que hay que observar cuando se analiza y compara la clase Server con Client es que la primera de ellas es una clase estática, mientras que la segunda no. Esto se debe a que en nuestra aplicación solo tiene que haber un solo servidor, pero puede haber varios clientes.

El extremo servidor (Clase Server) es muy similar al cliente con la salvedad de que el servidor se ha dividido en 3 partes: MechanismServer, MissionServer y ToolServer. De este modo, cada uno de estos "servidores" tienen un comportamiento muy similar al de la clase Client. Al igual como ocurre en la clase Client, el primer paso que se da en la clase Server es asignar un gestor de seguridad (SecurityManager) si no se tiene uno asignado. Tras esto, se crearán 3 servidores: MechanismServer, MissionServer, ToolServer que se ocuparán de cada uno de los aspectos del robot. Para finalizar y también de forma similar a la clase Client, el ultimo paso consistirá en la exportación de cada uno de los servidores nombrados anteriormente mediante el método rebind de la clase estática Naming:

Naming.rebind(ComIDs.MECHANISM_SERVER_ACCESS_STRING,mechanism);
Naming.rebind(ComIDs.MISSION_SERVER_ACCESS_STRING,mission);
Naming.rebind(ComIDs.TOOL_SERVER_ACCESS_STRING,tool);

Pero si observamos, en esta clase no se produce la recogida de los objetos clientes necesarios. Pero esta acción no hay que olvidarla ya que necesitamos de estos objetos para hacer invocaciones remotas de los métodos del cliente desde la máquina servidor. La recogida de los objetos que representan el cliente solo se producirá cuando el cliente invoque una orden del servidor que necesite o implique una respuesta inmediata o periódica por parte del servidor, como es el caso de los métodos "get" del servidor, ya que tras esto, el servidor tendrá que hacer una invocación remota de un método "update" del proceso cliente. Así, si se diera la situación en la que el cliente solo llamara a métodos que no soliciten la respuesta del servidor, el cliente en ningún momento recogerá los objetos clientes exportados. Este detalle es muy importante resaltarlo, y es que puede resultar un poco extraño que una clase que en principio se creaba como contenedor de los servidores individuales, ahora se va a encargar de exportarlos. Y esta es la principal razón por la que se decidió hacer desde esta clase, ya que de este modo la publicación se hace de forma centralizada y se hace tras la creación de estos servidores.

Hay que destacar que cuando se crearon los servidores MechanismServer y ToolServer justo antes de las llamadas a los métodos "bind", se producirá el tratamiento de variables de tipo RegisteredClients y EventControl que permitirá el control del registro de clientes para la actualización de parámetros del robot. Pero esto se comentará en puntos posteriores.

9.2. Registro de clientes.

9.2.1. 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 (vease ilustración 37, 38, 39). Los métodos que permiten a un cliente registrarse en estos servidores son los siguientes:

void addMechanismListener(String accessString);

void addToolListener(String accessString);

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 cambia 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 cadena de acceso que permite identificar al cliente entre todos los restantes para que así el servidor pueda incluirle en su registro. Un ejemplo de cadena de acceso sería: "rmi://"+ 127.0.0.1 +"/ServiceOfMechanismListenerClient"; como se puede observar, además de la dirección IP de la máquina que solicita el registro, también se introduce el nombre con el que se exporta dicho cliente por la red; de este modo, el servidor en cuestión sólo tendrá que recoger el cliente con esa cadena de acceso e invocar su método remoto de actualización de variables.

2. Cuando el cliente invoca al método del paso número 1, en la implementación de este método se vuelve a invocar al método homónimo, pero en este caso, del objeto servidor que corresponda; es decir, si el método invocado es *addMechanismListener* ahora se llamará al método del servidor mechanismCtrl. Pero hay un detalle muy importante a destacar, y es que el servidor con el que se hace la invocación, es el mismo objeto al que se le asigno la captura del servidor remoto, por lo que ese objeto ya no es local. Así, la orden que se invoca, se está invocando directamente al servidor de la máquina remota.

3. Aunque el servidor local sea un objeto de la interfaz MechanismCtrl, el servidor remoto es sobretodo un objeto de la clase MechanismServer que implementa dicha interfaz por lo que habrá que dirigirse a la implementación en esta clase del método invocado. En su implementación, además de las acciones propias del robot que se deseen realizar, se ejecutarán las siguientes sentencias:
   History.addClient(accessString,ComIDs.MECHANISM_LISTENER);
   registers.addMechanismEventListener(accessString);

4. Con estas 2 sentencias, lo que se produce es un tratamiento del registro de los clientes, que como ya se ha comentado en puntos anteriores, se centra en las clases en todas las clases del subpaquete registration: EventControl, History, HistoryObject, RegisteredClients, TimerControl y TimerRegisteredObject. En la primera sentencia, se llama al método addClient de la clase estática History para que se añada al historial el cliente representado por la cadena de caracteres introducida como parámetro. Dicha cadena será la misma que en un principio se introdujo desde el cliente en el paso 1. Además de este parámetro, al método hay que introducirle un identificado indicando el tipo de cliente que se está registrando.

5. Tras esto, se creará un objeto de tipo HistoryObject que es el único que se puede introducir en la base de datos (objeto de tipo HistoryObjectVector). A la hora de introducirlo, se ha de mirar si el cliente ya ha sido almacenado anteriormente y el tipo de registro también. Si ambos parámetros coinciden el cliente no se almacenará.. No hay que confundir este almacenamiento del cliente con el que se producirá en el siguiente paso, ya que con éste lo que se pretende es tener todos los clientes que hayan utilizado la aplicación almacenados para que cuando llegue el momento de cerrar la aplicación se cierren (si todavía siguen abiertos); mientras que el otro registro de los clientes se utiliza para poder acceder a aquellos clientes que deseen una actualización periódica o por eventos de las variables del robot,

6. Por último, la segunda sentencia a ejecutar será registers.addMechanismEventListener(accessString) a la cual también se le pasará la cadena de acceso como parámetro. Mediante esta llamada al objeto de la clase RegisteredClients, se va a almacenar otra vez el cliente, pero esta vez especificando si el cliente se está registrando para recibir la actualización de las variables periódicamente o bien, cada vez que se produzca un evento, que puede ser una modificación de estas variables o cualquier otro acontecimiento. En este registro, del mismo modo que en el punto anterior, se comprobará si el cliente ha sido ya almacenado o no. La clase

RegisteredClients dispone de 4 vectores para almacenar por un lado los clientes de los servidores Mechanism y Tool y por otro lado, y dentro de este subgrupo, almacenar los que estén para registro periódico o por eventos. Así, según se llame al unos métodos de registro o a otros, el almacenamiento se producirá en un vector o en otro. Además, hay que destacar que si el cliente se registra para actualizaciones periódicas, además de la cadena de acceso habrá que almacenar el periodo de actualización deseado; y si un cliente con estas características se vuelve a registrar, este periodo será actualizado por el nuevo valor.

Ahora ya solo queda estudiar el comportamiento del servidor cuando tenga que actualizar las variables de los clientes almacenados en sus registros para los 2 modos. Del mismo modo que se hizo en el punto anterior, solo se analizará la actualización con un solo método ya que para todos los pasos son análogos.

1. Para que cada servidor pueda "percatarse" de que se ha producido un cambio en alguna de las variables, cada uno debe crear un objeto de tipo "EventControl" que se encargue de avisar a estos servidores de 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.

2. 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.); cada vez que se hace clic sobre uno de los botones, se llama al método actionPerformed asignado a escuchar los eventos de los botones y desde aquí se invocará al método update (del servidor en cuestión) que corresponda para cada botón. Cuando se simula un cambio de variable de estado de una parte del robot en concreto, el objeto "EventControl" asociado a cada servidor será el que controle esa parte del robot que se ha alterado y notificará a dicho servidor que ha de actualizar dicha variable de estado a los clientes registrados. Los métodos existentes actualmente solo permiten notificar el cambio del estado del robot y las alarmas.

3. Al llamar al método del servidor del que se hablaba en el punto 2, es el propio objeto EventControl el que recorrerá la lista de clientes registrados en el servidor y les notificará el cambio. 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.
- Recoge cada una de las cadenas de accesos de los clientes registrados: *Object[] mechanismEventRegisters = registers.getMechanismEventListeners();*
- Recorre la lista de los clientes registrados e invocará con el objeto servidor el método encargado de actualizar las variables de los clientes que están identificados y localizados gracias a su cadena de acceso: *mechanismServer.updateMechanismStatus(accessString,ComIDs.EVENT,status);*

siendo mechanismServer el servidor que se encarga de actualizar las variables a actualizar; accessString: la cadena de acceso de cada uno de los clientes registrados; comIDs.EVENT: la constante con la que indicamos que el registro es mediante eventos (ya que estamos en la clase EventControl); y status: la variable a actualizar en los clientes.

}

4. Tras esto y como ya se ha dicho, se invoca al método del servidor que actualiza las variables y desde allí, se recogerá el cliente identificado por la cadena de acceso introducida como parámetro invocando.

5. Con el cliente importado, ya solo quedará invocar su método remoto con el que actualice sus variables; esto se conseguirá llamando método del mismo nombre que posee dicho cliente. 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).

9.2.2. Registro mediante notificación cada cierto periodo de tiempo.

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 (vease ilustración 38 y 39). Los métodos que permiten a un cliente registrarse en estos servidores de este modo son los siguientes:

void addMechanismListener(int period)
void addToolListener(int period)

Estos métodos son los que contienen 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 introduciendo un valor entero que corresponda con el periodo deseado para la actualización de las variables.

2. Tras esto, utilizando un objeto servidor que se haya recogido anteriormente con una sentencia lookup (en nuestro ejemplo un objeto de tipo MechanismCtrl) se llamará al método homónimo pero introduciendo en este caso le pasa como parámetro una cadena de acceso que permita identificar al cliente que está realizando la "petición y además le introduce un entero que es el periodo de tiempo en milisegundos con el que el servidor notificará el estado del robot; éste coincidirá con el parámetro introducido en la invocación anterior.

3. 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.

4. Tras esto, y ya desde los métodos del servidor remoto, lo primero que se va a hacer es introducir ese cliente dentro del historial de clientes que han "pasado" por la aplicación. Esto se consigue mediante la sentencia:

   History.addClient(accessString,ComIDs.MECHANISM_LISTENER);

   Este hecho ya ocurría para el registro mediante eventos, así que ya resulta muy conocida su utilización. Se recuerda que para este almacenamiento se tendrá que indicar como parámetros la cadena de acceso del cliente a introducir y un identificador que determine el tipo de cliente introducido.

5. Seguidamente y mediante el objeto de tipo RegisteredClients asignado al servidor con el que estamos trabajando, se invocará al método addMechanismTimeListener() introduciéndole además los mismos parámetros utilizados en el método en que estamos; es decir, el periodo y la cadena de acceso. Con este método lo que se consigue es que se añada el cliente al registro para que cada vez que se cumpla un intervalo de tiempo igual al periodo, reciba la actualización de las variables del servidor de mecanismos.

6. A la hora de introducirlo, se ha de verificar si ese cliente ya está almacenado y si es así, se deberá actualizar el valor del nuevo periodo. Antes de verificar esto, se ha de crear un objeto de tipo TimeRegisteredObject que englobe la cadena de acceso y el periodo; y este será el que se añadirá al vector-almacén en caso de que no se encuentre ya en éste. Si solo hubiera que modificar el periodo de algún objeto de este tipo ya existente, con una simple sentencia utilizando el método setNewPeriod(int period) el proceso se simplificaría notablemente.

   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 0 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.

En la lista de registrados del servidor se dispondrá de todos los clientes registrados con su correspondiente cuenta atrás. En el momento en el que se registre el primer cliente se creara un reloj (Timer) que notificara al servidor cada 100 mseg, cada periodo de 100 mseg marcado por el reloj se irán decrementando la cuenta atrás de cada cliente hasta que lleguen a 1 y se les notifique el estado del robot al cliente asociado a dicha cuenta atrás. De esta forma se consigue crear un único reloj para todos los clientes y distinguir la diferencia de periodo de cada uno de ellos con su cuenta atrás.

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, ComIDs.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.

En el método actionPerformed de la clase TimerControl se realizan las siguientes operaciones (cada 100 mseg.):

- 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.

La forma de notificar al cliente el estado del robot es mediante estos métodos:
mechanismServer.updateMechanismStatus(obj.getAccessString(), ComIDs.TIME, new String());
mechanismServer.updateMechanismAlarms(obj.getAccessString(), ComIDs.TIME, new int[3]);

2. Aunque estas sentencias son ya muy sencillas de entender ya que han aparecido muchas a lo largo del comentario, se explicarán otra vez: se llamarán a los métodos de actualización de variables del servidor pasándole como referencia la ya muy conocida cadena de acceso del cliente a actualizar. El hecho de introducir el tipo de registro que tiene el cliente en este método no es por otra razón sino para que tanto los métodos de los servidores tengan los mismos parámetros que los de los clientes, aunque este parámetro no sea utilizado aquí. El tercer parámetro introducido simplemente son los objetos que se quieren actualizar en el cliente. Como ya suele siendo común, en la implementación de este método simplemente se realizará una llamada a un método homónimo de la clase cliente que se ha importado de la red gracias a la cadena de acceso introducida como parámetro. Los parámetros a introducir en esta clase serán los mismos con los que se invocó al método en el que nos encontramos, sin incluir la cadena de acceso, ya que ya se está llamando a un método del cliente y ya no vamos a necesitar acceder a éste remotamente.

3. Por último, en la implementación del método del cliente, se deberá actualizar la variable introducida como último parámetro, además de todas las acciones que se deseen de visualización para el usuario de la aplicación.

9.2.3. Lista de registrados y objetos encapsuladotes.

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(String stringAccess, int period) {...}
  public void addTimeListener(String stringAcces , 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(String stringAccess, int period) {...}

  public void addTimeListener(String stringAccess ,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 caso:

· Si esta registrado con el mismo periodo => no hace nada.

· Si esta registrado con periodo distinto => 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(String stringAccess, int regMode) {…}

public void removeToolListener(String stringAccess , 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 ComIDs).

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.

9.3. Cierre del cliente

Un detalle muy importante a aclarar es la diferencia entre el cierre de la aplicación y el de un cliente; y es que la primera acción ya implica la segunda, pero a la inversa. Cuando se desea cerrar un cliente, que normalmente será único para esa máquina (recordad que se comentó que no tenia mucho sentido la creación de varios clientes en una misma máquina), se supone que ya no se desea manejar el robot desde esa máquina remota, y que por tanto, se puede cerrar esa aplicación cliente.

Si se recuerda, cada interfaz gráfica del cliente dispone tanto de un objeto Manager encargado de implementar los métodos de apertura y cierre de la interfaz AppManager, como de un objeto Client al que pasarle la orden cada vez que se pulsa un botón. Pues bien, cuando el usuario hace clic sobre el botón “closeClient” de su interfaz, inmediatamente, el objeto encargado de escuchar los eventos de ese botón, hará una llamada al método closeClient del objeto Manager, pasando como parámetro el cliente a cerrar.

Como el objeto Manager almacena todos los clientes que se encuentran en una misma máquina, al invocar el método anterior se hará una llamada al método close del cliente introducido como parámetro y tras esto se eliminará este cliente del vector donde se guardan. Al llamar al método close del cliente, lo primero que hará será borrarse de todos los registros de los servidores, para que de este modo, no le envíen mas actualizaciones de variables; no importa que este cliente no este almacenado en estos registros, ya que los propios servidores comprueban la existencia o no del cliente en sus registros. Para ello se invocarán a los métodos removeToolListener y removeMechanismListener para cada servidor y para cada tipo de registro.

Al invocar estos métodos remotos de los servidores, éstos lo único que deben hacer es buscar la cadena de acceso del cliente (que se introdujo como parámetro) dentro de sus vectores, en especial, los de los objetos RegisteredClients de que disponen. El borrarlo de un vector u otro (había uno para los clientes registrados mediante eventos y otro mediante periodo) dependerá del identificador introducido como último parámetro; así, si el identificador es

ComIDs.EVENT, el cliente se intentará eliminar del vector toolEventListeners o MechanismEventListener, mientras que si el identificador es ComIDs.TIME, se eliminará de los vectores mechanismTimeListeners o toolTimeListeners.

Por último, y volviendo otra vez a la actuación de los clientes, mediante las sentencias :
Naming.unbind(ComIDs.MECHANISM_LISTENER_ACCESS_STRING)
se hará el proceso inverso al de los métodos bind, es decir, que ya no se exportarán estos objetos a partir de la red. Para ello, solo habrá que introducir la cadena de acceso que corresponda con ese cliente. Si recordamos, se tuvieron que ejecutar 5 sentencias de tipo bind para publicar "todo" el cliente, ya que este implementaba 5 interfaces que queríamos "aprovechar" remotamente; pues bien, ahora, por la misma razón también se tendrán que ejecutar 5 sentencias con este comando. Tras esto, ninguno de los servidores podrá invocar algún método remotamente de los clientes; con esta simple acción los clientes pierden su participación en la aplicación.

Para finalizar, y para que se pueda observar estas acciones desde la interfaz del usuario se esconderá el frame perteneciente a dicho cliente. 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, ya que los servidores siguen estando publicados en la red.

9.4. Cierre de la aplicación.

En la interfaz gráfica principal, disponemos de un botón "shutdownApp" que una vez abierta la aplicación quedará habilitado. Con éste, conseguimos cerrar la aplicación completamente, es decir, cerrando también todos los clientes abiertos en ese momento (vease ilustración 36), estando éstos o no en la misma máquina.

Tras pulsar sobre el botón correspondiente, se producirá un evento que recogerá el objeto encargado de escuchar sus eventos (asignado al inicio de la aplicación). Cuando se produce el evento se llamará a estas 3 sentencias:

manager.shutdownApp();
shutdownApp.setEnabled(false);
newClient.setEnabled(false);

Con la primera de ellas, como la clase Manager tiene un vector donde almacena los clientes de una misma máquina, necesitará a uno de estos Client para invocar al método closeApp de esta clase. Por defecto, se tomará el primero de los clientes, es decir, el primer elemento del vector correspondiente, ya que no hay que olvidar que lo más común es que solo exista un cliente por máquina. Al invocar el método closeApp el cliente, se llamará al método homónimo de cada uno de los servidores que se recogieron de la red mediante el método lookup. Así, se llamará al método closeApp tanto del servidor Mechanism, Tool y Mission; pero no hay que olvidar que estos objetos representan a los servidores remotos y que por lo tanto las invocaciones de estos métodos también debe ser remota.

La implementación de este método, será similar tanto para los objetos de tipo MechanismServer como MissionServer y ToolServer, por lo tanto aquí solo se analizarán los pasos para el primero de éstos. Al llamar al método closeApp de uno de los servidores, este se tendrá que encargar de extraer todos los clientes de sus registros invocar con todos ellos el método close() para de este modo cerrar todos los clientes que permanezcan abiertos. Antes de esto, también se procederá a borrar el registro de los clientes que se encontraban almacenados para recibir las actualizaciones de las variables, ya que si queremos cerrar la aplicación, los servidores deben dejar de enviar actualizaciones a los clientes y "olvidarse" de ellos definitivamente. Para ello, bastará con llamar al método resetListeners de la clase RegisteredClients, que si recordamos era la base de datos donde almacenar todos los clientes registrados.

Puede existir un gran problema a la hora de cerrar cada uno de los clientes almacenados en el historial (clase History) de cada servidor ya que será muy normal que estos historiales tengan muchos clientes en común, y por tanto, si en el primer servidor a tratar, ya se cierran estos clientes comunes, cuando desde otro servidor se intente cerrar otra vez, ésta ya no existirá. Para ello, se debe experimentar con la recogida y relanzamiento de excepciones, ya que al cerrar un cliente que no existe, saltará una excepción; así, las sentencias quedarían del siguiente modo:

```
{
registers.resetListeners();
HistoryObject obj;
for (int i=countCloseApp; i<History.getHistory().length; i++)
{
        countCloseApp++;
        obj = (HistoryObject)(History.getHistory())[i];

        if ( obj.getTypeOfClient() == ComIDs.MECHANISM_UPDATER) {
```

```
        mechanismUpdaterClient =
(MechanismUpdater)Naming.lookup(obj.getAccessString());
        mechanismUpdaterClient.close();}

        else if ( obj.getTypeOfClient() == ComIDs.MECHANISM_LISTENER) {
        mechanismListenerClient =
(MechanismListener)Naming.lookup(obj.getAccessString());
        mechanismListenerClient.close();}
}
countCloseApp = -1;
Naming.unbind(ComIDs.MECHANISM_SERVER_ACCESS_STRING);}
catch (Exception e){
if (countCloseApp == -1){throw new Exception();}
else {closeApp();}
}
```

La primera sentencia corresponda al borrado del registro de los clientes almacenados para su actualización. Después se creará un objeto de tipo HistoryObject para poder sacar cada uno de los elementos del historial (clase estática History) y almacenarlos ahí. Pues bien, para cada cliente que se asigne a ese objeto, se va a comprobar si es de tipo updater o bien listener (si es del primer tipo el "cast" a realizar á de tipo MechanismUpdater, y si es del segundo será a MechanismListener), para así, capturar de la red el cliente que corresponda con ese objeto; no hay que olvidar que en los objetos HistoryObject también se incluía la cadena de acceso del cliente, por lo tanto, conociendo el tipo de cliente y su cadena, ya podemos acceder sin ningún problema al cliente remoto. No hay que olvidar que los clientes que se almacenaban en el historial eran todos aquellos que han utilizado alguna vez la aplicación.

Una vez capturado mediante el método lookup, solo quedará invocar remotamente al "close" de dicho cliente. Las acciones que tendrá que realizar el cliente a continuación serán las mismas que las comentadas en el punto anterior, es decir, eliminarse del registro de todos los servidores y eliminar su objeto remoto de la red mediante el método unbind.

Si repasamos el proceso realizado puede resultar un poco complejo, y es que, analizando a fondo nos damos cuenta, como primero es un cliente el que invoca el método cerrar aplicación, después, desde éste se llamará al método cerrar de los servidores y desde éstos, se volverá a hacer una invocación remota al método close remoto de cada uno de los clientes. Por ultimo, éste volverá a invocar a un método de los servidores para que eliminen al

cliente del registro, para asegurarse que se han borrado de forma correcta cada cliente del registro. Se podría haber eliminado una de las dos acciones de eliminar, ya que se obtiene el mismo resultado al invocar el método *resetListeners* de la clase RegisteredClients, que cada uno de los clientes se eliminen del registro de sus servidores; pero la verdad, es que al poder reutilizar el método close de la clase Client se obtenía una gran ventaja en cuanto a tiempo de programación, y además, resultaba muy tentador eliminar desde el servidor su propio registro, ya que con una simple sentencia nos asegurábamos que se borraría correctamente el registro.

La existencia de la variable *countCloseApp* es muy importante, ya que gracias a este contador, evitamos que se bloque la aplicación debido al problema explicado anteriormente en el que cada cliente cierra sus servidores, y se puede dar el caso en que existan varios clientes que de forma escalonada pretenda cerrar el mismo servidor. La forma de controlar todo este proceso es muy sencilla:

- Inicialmente esa variable tiene valor 0. El valor de este contador, se le asigna a "i" que indicará el numero de iteraciones a realizar de la sentencia for.
- Esta sentencia se hará tantas veces como clientes hayan en el historial. En cada iteración se incrementará el countCloseApp y se recogerá e importará un cliente, mediante el cual, como ya se dijo, se invocará el método close.
- Lo normal, es que la sentencia for termine; que al contador se le asigne el valor -1 y que se elimine el objeto remoto de ese servidor de la red mediante el método unbind. Pero si ocurre alguna excepción por algún motivo como los comentados anteriormente, saltará una excepción y por lo tanto, nos saldremos del for. Esto ya no será un problema, porque al ver que el contador no tiene el valor de -1 que se asigna al final, podemos retomar esta sentencia volviendo a invocar el mismo método, con la diferencia de que ahora el contador no tendrá el valor 0 si no el de la iteración en la que nos encontrábamos.
- Por ultimo, si al lanzar la excepción, el valor del contador es -1, eso nos indica que ya se han superado todas las iteraciones del for y donde ha saltado esa excepción es al realizar el unbind, por lo cual, esa excepción ya si la podemos relanzar hacia el exterior sin ningún problema.

De las dos bases de datos de clientes, la del historial y la de los clientes registrados, solo se borra la segunda, y es que en verdad no es necesario hacerlo en ninguna, pero se hace para la de los clientes registrados, para que los servidores se desvinculen totalmente de la actualización de las variables y de los propios clientes a la vez; así, nunca se podrá el caso de que mientras se están cerrando todos los clientes, uno de los servidores intente actualizar las variables de algunos de estos. Así, de todos modos, como la aplicación se cerrará completamente, los

registros se vaciarán y la próxima vez que se ejecute la aplicación, estos volverán a funcionar desde 0.

Recordando las 3 sentencias que se comentaban al principio del punto, vemos como ya solo queda tratar los botones para que el usuario de la aplicación pueda visualizar que la acción ha sido realizada con éxito.

9.5 Diferencias destacables en los aspectos comentados para la implementación CORBA.

Como se puede comprobar, todos estos aspectos se han comentado para el código RMI, pero para la implementación usando la tecnología CORBA la solución aplicada es la misma, salvando por supuesto, las diferencias propias de cada tecnología en cuanto a los procesos de publicación y recogida de objetos. Estas diferencias serán las que se comenten aquí:

El diseño de la implementación con CORBA tiene muchas similitudes con la implementación con RMI, y podemos destacar como principales diferencias las relacionadas con el acceso al servicio de nombrado. A continuación se detalla el proceso que se ha seguido en la aplicación para publicar los objetos y acceder a ellos mediante el servicio de nombres de CORBA.

El primer que se debe llevar a cabo antes de publicar cualquier objeto CORBA en una máquina es crear e inicializar el ORB, ya que es el middleware que da acceso al resto de objetos distribuidos de la aplicación. Para ellos usamos el método estático “init" de la clase ORB que proporciona la API de Java.

ORB orb = ORB.init(args, null);

A continuación el objeto CORBA que va a ser publicado en el servicio de nombrado debe conectarse con el ORB mediante el método estático “connect” de la clase ORB. Con este método conseguimos conectar el sirviente del objeto CORBA al ORB, para que el ORB pueda reconocer las llamadas a éste y enviarlas al sirviente correcto.

orb.connect(this);

El método connect requiere que se le indique el objeto CORBA que se desea conectar, y éste se tomará como parámetro. En el caso de que este comando se utilice dentro del constructor del objeto CORBA podemos utilizar el parámetro “this” de Java.

A partir de ahora ya se puede publicar el objeto en cuestión mediante el servicio de nombrado que proporciona CORBA. A diferencia de RMI, en el que se proporcionaba una clase que permitía el acceso al servicio de nombres (clase Naming), en CORBA dicho servicio esta integrado en el ORB como un objeto CORBA. Por ello es necesario recoger dicho objeto para poder usar el servicio de nombrado, y esto se consigue mediante el siguiente código:

```
org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
NamingContext ncRef = NamingContextHelper.narrow(objRef);
```

La primera línea permite encapsular el servicio de nombres en un objeto CORBA, y la segunda línea pretende que se puede acceder a ese objeto a través de una referencia a NamingContext, ya que esta clase representa el contexto de nombrado raíz y es la que proporciona los métodos que permiten publicar y recoger objetos CORBA en el servicio de nombrado. El método narrow de la clase NamingContextHelper permite convertir cualquier objeto CORBA a un objeto NamingContext, similar al cast de objetos en Java.

A partir de ahora ya podemos publicar nuestro objeto utilizando los métodos del objeto NamingContext obtenido del ORB. Para ello se debe crear previamente un objeto que encapsule el nombre que va a ser asociado a dicho objeto en el servicio de nombrado (y que será el que usen otros objetos para acceder a él). El objeto que encapsulara ese nombre será del tipo NameComponent, y éste es el código necesario para crearlo:

```
NameComponent name = new NameComponent(“Servicio_1”, "");
NameComponent pathName[] = {name};
```

Es importante destacar que se pueden utilizar varios nombres para identificar a un mismo objeto CORBA, por lo que el método que publica un objeto en el servicio de nombres admite un array de nombres. En nuestro caso solamente publicaremos el objeto con un solo nombre (en este ejemplo el nombre del objeto sería “Servicio_1”).

Finalmente publicamos el objeto CORBA mediante el método rebind que dispone servicio de nombres, indicando como parámetros el array de nombres asociado y el objeto CORBA a publicar.

ncRef.rebind(pathName, this);

A partir de ahora cualquier aplicación CORBA puede acceder a nuestro objeto a través del nombre asignado “Servicio_1”. A continuación se detalla el proceso que se seguirá para acceder a él y poder usarlo como un objeto local.

El primer paso es obtener una referencia al contexto de nombrado raíz de la misma forma que se explicó anteriormente, y encapsular en un array de objetos NameComponent el nombre del objeto al que deseamos acceder.

NameComponent name = new NameComponent(“Servicio_1”, "");
NameComponent pathName[] = {name};

El siguiente paso es obtener el objeto CORBA deseado con el método resolve del servicio de nombrado. Al igual que el método para publicar un objeto, el método para acceder a él necesita como parámetro el array de nombres necesario para identificar el objeto deseado. A través de ese nombre el método resolve resuelve la referencia al objeto en cuestión en el nombrado.

MechanismCtrl server = MechanismCtrlHelper.narrow(ncRef.resolve(pathName));

Sin embargo, el método resolve devuelve una referencia a un objeto CORBA, y para poder trabajar con los métodos del objeto deseado es necesario hacer la conversión de objeto CORBA a objeto Java a través del método narrow de la clase Helper asociada a dicho objeto (esta clase es generada automáticamente por el compilador IDL). En el ejemplo se utiliza la clase MechanismCtrlHelper ya que el objeto al que deseamos acceder es del tipo MechanismCtrl.

Finalmente, para desregistrar del servicio de nombrado un objeto publicado anteriormente, utilziaremos el método unbind del servicio de nombres, que al igual que el resto de métodos anteriores admite como parámetro el array de objetos que encapsulan el nombre del objeto a desregistrar.

NameComponent name = new NameComponent(“Servicio_1”, "");
NameComponent pathName[] = {name};
ncRef.unbind(pathName);

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 clic.

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 principio, 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 que se explicarán con detalle en un apartado de la memoria, 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.

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.

11. Manual de usuario de la aplicación

11.1 Compilación y ejecución.

A continuación se detallarán los pasos a seguir para compilar y ejecutar la aplicación, ya sea con la implementación RMI o CORBA.

11.1.1 Compilación y ejecución para la implementación RMI.

El primer paso a realizar es compilar los archivos Java. Esto generará los archivos de clase (.class) correspondientes a cada uno de los archivos fuente (.java). Esto se realiza con:
*C:\jdk1.3\bin\javac *.java*

Seguidamente habrá que crear los archivos stub y skeleton para cada objeto remoto mediante el uso del compilador rmic, que recibe como parámetro el nombre de la clase que contiene la implantación de cada objeto remoto. Estos se encuentran en los directorios y paquetes Servers y Management. Así, para el servidor habrá que ejecutar el comando:
C:\jdk1.3\bin\rmic Servers.ToolServer Servers.MechanismServer Servers.MissionServer,
y para el cliente:
C:\jdk1.3\bin\rmic Management.Client

Con estos dos comandos se generarán los siguientes ficheros:
MechanismServer_Skel.class
MechanismServer_Stub.class

MissionServer_Skel.class
MissionServer_Stub.class
ToolServer_Skel.class
ToolServer_Stub.class
Client_Skel.class
Client_Stub.class

Una vez compilados todos los archivos, es necesario ejecutar el servicio de nombres de RMI (rmiregistry), mediante el cual los clientes pueden obtener una referencia remota de un objeto, preguntando por su nombre. Para ejecutar el rmiregistry se necesita introducir el siguiente comando: rmiregistry & (en el caso de Unix) o start rmiregistry (en el caso de Windows). Por defecto, el registro de nombres de RMI escucha las peticiones de los clientes en el puerto 1099, pero es posible indicarle un puerto específico desde la línea de comandos. Por ejemplo, para ejecutar el registro en el puerto 5000 se debe introducir: start rmiregistry 5000. Si se ejecuta el registro en un puerto diferente al designado por defecto, es necesario que en los clientes, al momento de hacer uso de los servicios de la clase java.rmi.Naming, se especifique el puerto en el que está escuchado el registro. Por ejemplo:
Naming.bind("//localhost:5000/Server");

El siguiente paso es ejecutar el programa de establecimiento. Aquí es importante definir las propiedades de políticas de seguridad, que especifican los permisos o privilegios que poseerán los códigos obtenidos desde diferentes fuentes. Estas políticas están representadas por una subclase de java.security.Policy que proporciona una implantación de sus métodos abstractos. La implantación por defecto de Policy obtiene la información necesaria a partir de archivos de configuración estáticos, donde se definen los permisos para el código. Existen dos maneras de crear los archivos de configuración. La primera utilizando un editor de texto cualquiera o usando la herramienta llamada policytool. Mediante cualquiera de estas dos opciones debemos crear un fichero de texto que contenga las siguientes líneas:

```
Grant
{// Habilita todos los permisos.
        permission java.security.AllPermission;
};
```

En el caso de que usemos un fichero de políticas creado por nosotros se debe especificar donde se localiza dicho fichero al ejecutar el código, usando el parámetro de java "Djava.security.policy" de la siguiente forma:

```
java -Djava.security.policy=C:/rmi/policy.txt mainServer
```

Otra opción es modificar el propio fichero de políticas de seguridad que proporciona el JDK, que se encuentra en C:\jdk1.3\jre\lib\security\java.policy. De este modo no es necesario definir su localización con el parámetro Djava.security.policy ya que java usa este fichero como políticas de seguridad por defecto.

Es muy importante notar que habilitar todos los permisos puede resultar sumamente riesgoso, debido a que se deshabilitan todos los mecanismos de seguridad de la MV Java. Políticas de este estilo sólo se deben implementar en fases de depuración o en situaciones donde las aplicaciones o applets sean completamente confiables.

A la hora de ejecutar una aplicación Java, hay que tener en cuenta algunas cosas:
En primer lugar, hay que indicar el nombre completo de la clase a ejecutar. Por ejemplo, si tenemos una clase Client que pertenece al paquete goya.management, el comando para ejecutarla será:
java goya.management.Client

Para que el anterior comando funcione tal cual, es necesario que el directorio actual desde el que se lanza la orden sea el codebase (por ejemplo, C:\java\codebase, si la clase Manager está en C:\java\codebase\com\ibm\manager\Manager.class). Si trabajamos en otro directorio, es posible evitar el tener que cambiar de directorio, indicando a la máquina virtual qué directorio debe tomar como codebase, con una orden como:
java -classpath C:\java\codebase goya.management.Client

Puede ser que en nuestro árbol de directorios, tengamos todas las clases juntas a partir de un único directorio codebase, o que las tengamos separadas en varios. Puede ser que tengamos unas clases a partir de C:\java\codebase y otras a partir de D:\clases, por ejemplo.

Para facilitar la ejecución de las clases y evitar el tener que incluir parámetros demasiado largos en las invocaciones a la máquina virtual, es posible definir una variable de entorno llamada CLASSPATH que contenga todos los directorios codebase de nuestro sistema. Utilizando esta variable, es posible invocar y utilizar clases que estén en cualquier parte, sin tener que pasarle un parámetro -classpath a la máquina virtual. Por ejemplo, con CLASSPATH=C:\java\codebase;D:\clases podremos utilizar clases que estén situadas en esos dos directorios. En el ejemplo anterior ya no es necesario el parámetro -classpath, por lo que la orden
java goya.management.Client

puede ser ejecutada desde cualquier directorio. (No es posible suprimir goya ni goya.management, puesto que forman parte del nombre completo de la clase).

Ahora estamos listos para ejecutar el programa que dará de alta nuestros objetos remotos en el sistema de activación de RMI.

Nota: Ademas del directorio donde se encuentran las clases, con la implementación RMI también se suministran dos ficheros de tipo Jar que engloban las clases del extremo cliente y servidor; estos son: RMIServer.jar y RMIClient.jar. Para ejecutar estos, además de activar RMIRegistry en ambos extremos, se tendrán que introducir los siguientes parámetros:

- C:\>c:\jdk1.3\bin\java.exe -jar RMIClient.jar <<IP_Server>> <<IP_Client>>

  Para el extremos cliente, el fichero jar se ha creado de tal forma que haya que pasar como parámetro la IP del servidor y la IP del cliente con la que se quiere trabajar, ya que en un primero momento, la aplicación no es capaz de conocer cual es su IP con la que se comunicará con el servidor.
- C:\>c:\jdk1.3\bin\java.exe -jar RMIServer.jar

  Para el extremo servidor no habrá que poner ningún parámetro.

Hay que destacar que la aplicación se ejecuta mediante el comando java del directorio jdk, ya que sino, no se utilizaría el fichero de seguridad creado para el correcto funcionamiento de la aplicación (java.policy).

11.1.2 Compilación y ejecución para la tecnología CORBA.

Definir la Interfaz Remota.

Definimos la interfaz para el objeto remoto usando el lenguaje para la definición de interfaces del OMG. Usamos IDL en lugar de Java porque el compilador idlj automáticamente genera los ficheros de stub y skeleton en Java a partir de la definición IDL, así como toda la infraestructura para conectar con el ORB. También, usando IDL, hacemos posible a los desarrolladores implementar clientes y servidores en cualquier otro lenguaje compatible con CORBA.

Nótese que si estamos implementando un cliente para un servicio existente de CORBA, o un servidor para un cliente ya existente, deberíamos coger los interfaces IDL del implementador. Deberá ejecutar el compilador idlj pasando como parámetro esos interfaces e implementar los métodos que necesitemos.

Compilar la Interfaz Remota

Cuando ejecutamos el compilador IDL de a Java (idlj) con la definición del interface, genera una versión en Java del interface, así como el código de las clases de los ficheros de stub y skeleton que le permiten enganchar su aplicación al ORB. Por favor, si desea conocer más sobre cualquier aspecto relacionado con la interfaz IDL, diríjase al anexo adjuntado relacionado con este tema.

Para ejecutar el compilador idlj (del JDK 1.3) nos basamos en el siguiente patrón:
idlj -fall -td <destino de los ficheros generados> <ruta del IDL>

Así por ejemplo, para la interfaz MechanismCtrl.idl se compila de la siguiente forma:
C:\jdk1.3\bin>idlj -fall -td C:\CORBA\ C:\CORBA\IDLs\MechanismCtrl.idl

Con el parámetro –f podemos definir los enlaces que se generaran sobre los skeleton dependiendo de si estamos compilando una interfaz que es parte del cliente (client), servidor (server), o ambos (all). En nuestro caso, como cada objeto extremo de la aplicación corresponde tanto con un cliente como con un servidor usamos –fall. El parámetro –td <dir> únicamente indica que el directorio de salida de los ficheros generados es el indicado a continuación (dir).

El compilador idlj genera una serie de ficheros, basándose en las opciones que le pasamos en la línea de comandos. Los ficheros generados se almacenan en un directorio si se especificó explícitamente con la sentencia "module" en la interfaz IDL (para el caso de la interfaz MechanismCtrl el directorio es Controls). Veamos a continuación los ficheros generados por idlj para la interfaz MechanismCtrl.idl usando los parámetros comentados anteriormente:

MechanismCtrl.java

Esta interface contiene la versión en Java del interface en IDL. El interface MechanismCtrl.java hereda de org.omg.CORBA.Object, proporcionando también una funcionalidad CORBA estándar. En esta interfaz no se encuentran los métodos definidos en la interfaz IDL, sino que se encuentran en otra clase (MechanismCtrlOperations.java), la cual hereda para poder tener acceso a esos métodos. Los clientes desean referencias a objetos que implementen este interfaz, al estilo de la programación Java normal.

MechanismCtrlOperations.java

El compilador de IDL idlj mapea todas las operaciones definidas en la interfaz IDL en este fichero, que es compartido tanto por los stubs como por los skeletons.

MechanismCtrlHelper.java

Esta clase es una clase abstracta que implementa métodos estáticos para operar con objetos de tipo MechanismCtrl. En general, para un interfaz "<interfaz>", las clases Helper contienen, entre otros, los siguientes métodos (todos ellos public static):

- void insert(org.omg.CORBA.Any any, <interfaz> t); para crear un objeto del tipo Any que contenga un objeto de tipo <interfaz>. El tipo Any, puede contener a cualquier otro tipo IDL definido. También existe la función extract() que realiza la función inversa, <interfaz> read(org.omg.CORBA.portable.InputStream _input), que ofrece una utilidad de serialización (también existe el método write()) <interfaz> narrow(org.omg.CORBA.Object obj). Este es uno de los métodos más importantes, ya que permite trasladar a un objeto a alguna de sus subclases (lo que en Java se conoce como "narrowing" y en C++ como "casting"). Esto es muy útil, por ejemplo, para convertir referencias obtenidas a través de los métodos del ORB string_to_object().

- TypeCode type(). Retorna un identificador del tipo tal y como será encontrado en el Interface Repository. Esto es muy conveniente para indicar el tipo de los parámetros y el resultado de las invocaciones que se construyen de forma dinámica a través del DII (Dynamic Invocation Interface).

Todas las clases definidas en IDL son acompañadas por una clase Helper.

MechanismCtrlHolder.java

IDL soporta el uso de parámetros in (entrada), out (salida) e inout (entrada/salida). Java sólo soporta parámetros de entrada. Para ello, se crean unas clases "Holder" que "guardan" un valor de cada tipo. Para un interfaz llamado "<interfaz>", la clase puede ser la siguiente:

```
// -*- Java -*-
final public class <interfaz>Holder implements org.omg.CORBA.portable.Streamable {
public <interfaz> value;
public <interfaz>Holder() {}
public <interfaz>Holder(<interfaz> initial){ value = initial; }
public void _read(org.omg.CORBA.portable.InputStream i) { ... }
public void _write(org.omg.CORBA.portable.OutputStream o) { ... }
public org.omg.CORBA.TypeCode _type() { ... }}
```

Como se ve, la semántica de los métodos es fácilmente deducible de su signatura. Es de señalar que se genera una clase "Holder" para cada tipo que se pase en un parámetro out ó inout.

MechanismCtrlImplBase.java

Esta clase abstracta es el skeleton del servidor, proporcionando una funcionalidad CORBA básica para el servidor. Implementa el interface MechanismCtrl.java. La clase del servidor MechanismServer debemos de hacer que herede de esta clase (public class MechanismServer extends _MechanismCtrlImplBase implements MechanismCtrl), el código para los métodos del interfaz.

MechanismCtrlStub.java

Esta clase es el stub del cliente, proporcionando funcionalidad CORBA al cliente. Implementa el interface MechanismCtrl.java.

Una vez definidas las interfaces IDL, ya hacemos todo el trabajo para generar todos estos ficheros necesarios para la aplicación distribuida. El único trabajo adicional que debemos hacer es implementar la funcionalidad del cliente y del servidor.

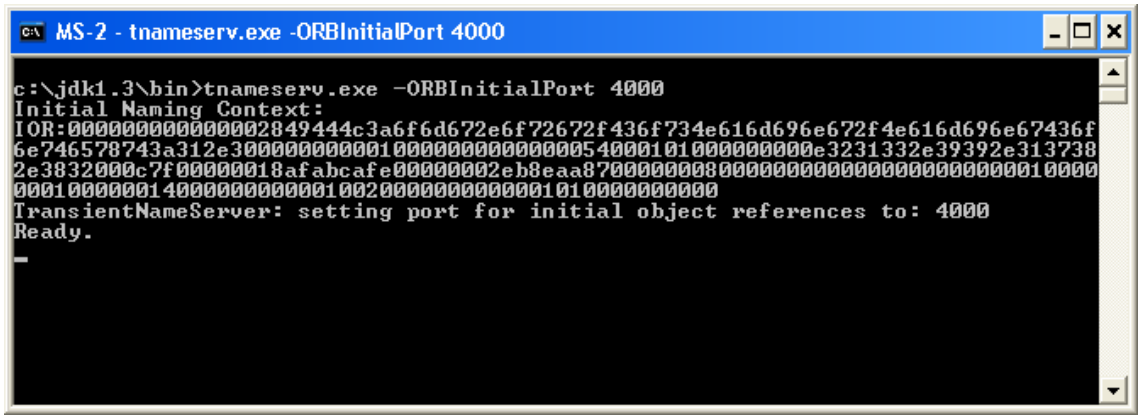
Iniciar el servicio de nombrado.

El servicio de nombres CORBA es una especificación, de la cual cada fabricante realiza su implementación. La implementación java IDL es un programa, llamado tnameserv, que es necesario ejecutar en un puesto de la red. Este servicio forma parte de la plataforma Java 2.

Al ejecutar el servicio de nombres obtendremos una salida similar a la mostrada en la figura. La larga secuencia de dígitos hexadecimales es la IOR del objeto CORBA que actúa como servicio de nombres. Una IOR es una referencia a un objeto CORBA independiente de ORB, lo cual significa que puede ser usada para facilitar la comunicación entre objetos que usan ORB de distintos fabricantes. En este caso el IOR del servicio de nombre podría servir para obtener una referencia a él sin usar el habitual método de resolución.

Otro de los datos que podemos apreciar en la figura es el número de puerto que deberá utilizar cualquier servicio o cliente que necesite acceder al servicio de nombres. Por defecto el puerto usado es el 900. Si desea usar otro puerto basta con indicarlo al ejecutar el tnameserv, usando la opción –ORBInitialPort tal y como se hace en la siguiente sentencia. En este caso el puerto a usar es el 4000.

Nota: Si estamos en Unix hay que tener en cuenta que sólo root puede ejecutar un proceso en un puerto menor que 1024. Por esta razón, recomendamos que se elija un puerto superior a 1024.



Se ejecuta sin parámetros comprobamos que el puerto por defecto es el 900.

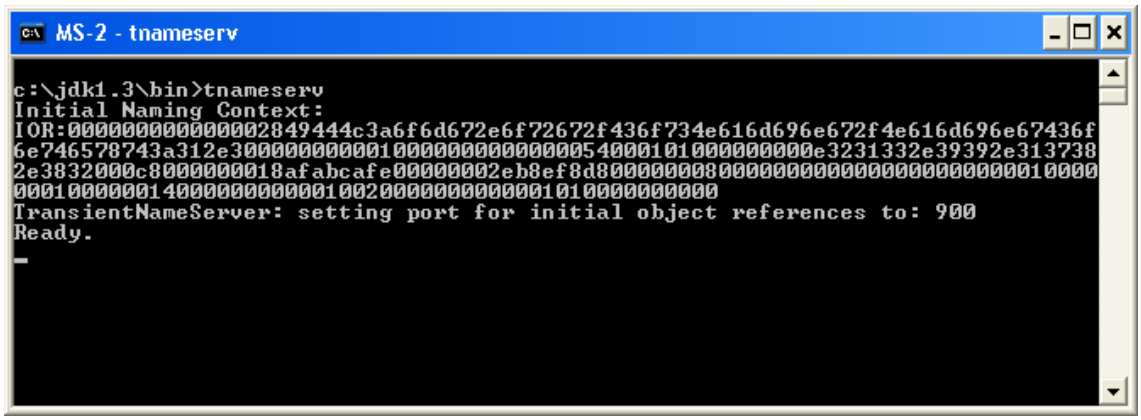


Figura: Al poner en marcha el servicio de nombres de Java IDL obtenemos el IOR correspondiente y el número de puerto por el que se queda a la escucha.

Para poder acceder al servicio de nombres desde cualquier aplicación es preciso saber el nombre del servidor y número de puerto en que está funcionando. Generalmente tnameserv se ejecutara en un ordenador de la red, un servidor, usando un puerto bien conocido por servidores y clientes CORBA. Un método alternativo consiste en usar la IOR devuelta por el servicio de nombres cuando es puesto en marcha, de tal forma que podría estar ejecutándose en un punto u otro de la red de forma indistinta. En cualquier caso, esa IOR debería estar accesible para las aplicaciones de algún punto bien conocido, como un servidor web o una base de datos.

Ejecución desde la terminal.

Ejecutaremos el servicio de nombrado tnameserv como se comento anteriormente y desde una seguna terminal iniciamos el servidor:

> *java mainServer -ORBInitialHost nameserverhost –ORBInitialPort nameserverport*

Desde una tercera terminal ejecutamos el cliente:

> *java mainClient -ORBInitialHost nameserverhost -ORBInitialPort nameserverport*

Nótese que nameserverhost es el host donde el servidor de nombres IDL está ejecutándose. Se puede omitir -ORBInitialHost nameserverhost si el servidor de nombres está ejecutándose en la misma máquina que el cliente. También se puede omitir -ORBInitialPort nameserverport si el servidor de nombres se está ejecutando en el puerto asignado por defecto (900).

Nota: Ademas del directorio donde se encuentran las clases, con la implementación CORBA también se suministran dos ficheros de tipo Jar que engloban las clases del extremo cliente y servidor; estos son: CORBAServer.jar y CORBAClient.jar. Para ejecutar estos, además de activar Tnameserv en ambos extremos (situándolo tan solo en el extremo servidor tambien funciona), se tendrán que introducir los siguientes parámetros:

- C:\>c:\jdk1.3\bin\java.exe -jar CORBAClient.jar -ORBInitialHost 192.168.0.1

  Para el extremos cliente, el fichero jar se ha creado de tal forma que haya que pasar como parámetro la IP del servidor (donde se ejecute Tnameserv) antecedido por la palabra ORBInitialHost.

- C:\>c:\jdk1.3\bin\java.exe -jar CORBAServer.jar

  Para el extremo servidor no habrá que poner ningún parámetro.

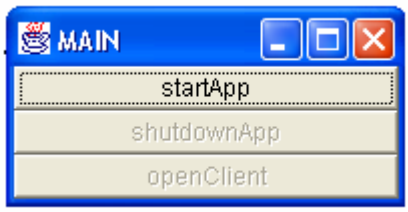
Hay que destacar que la aplicación se ejecuta mediante el comando java del directorio jdk, ya que sino, no se utilizaría el fichero de seguridad creado para el correcto funcionamiento de la aplicación (java.policy).

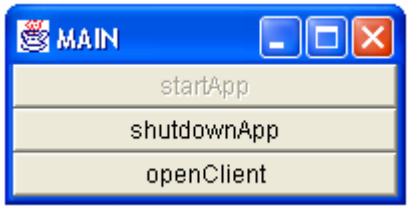
11.2 Extremo cliente.

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.



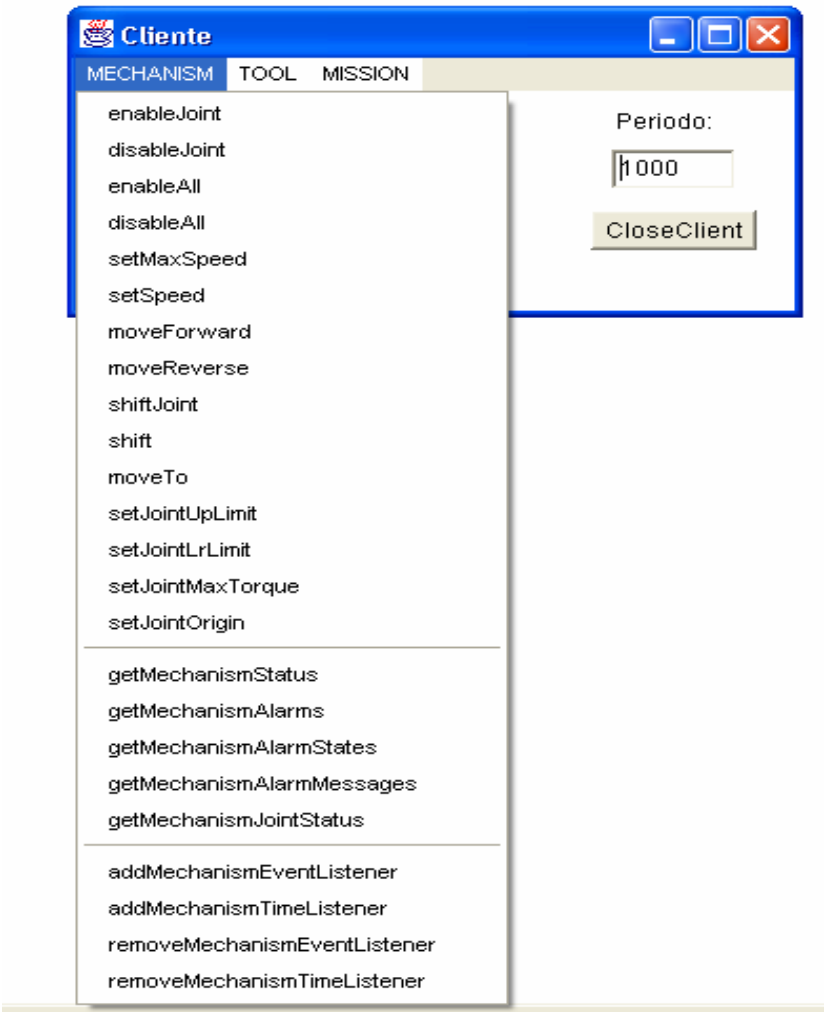


Una vez iniciada la aplicación, aparecerá una nueva ventana en la que podemos ver: un botón de cierre de cliente (“closeClient”), un textbox en el que introducir un valor entero (“periodo”) y una barra de menú con 3 elementos desplegables: “mechanism”, “misión” y ”tool”. Empezando por el botón “closeClient”, 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 una 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: JavaDoc.

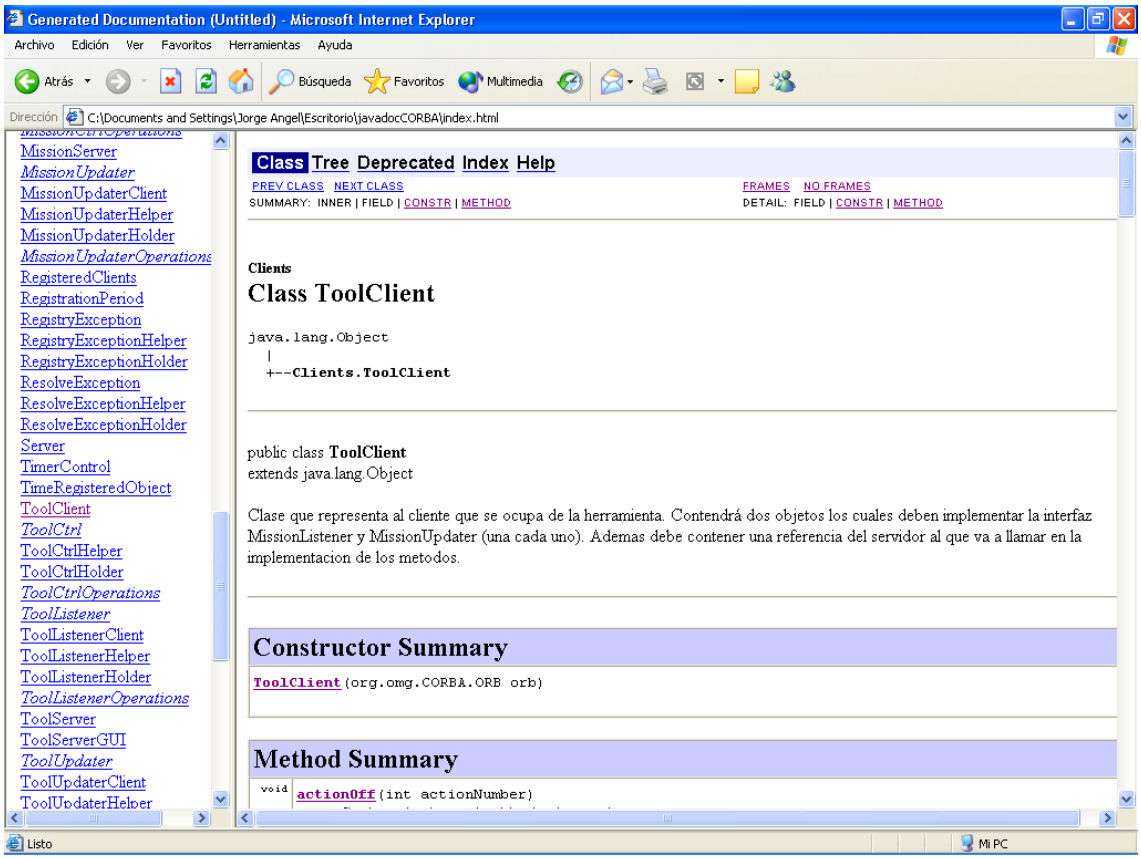
El código implementado para esta aplicación está comentado siguiendo las normas de JavaDoc. Las convenciones de código son importantes para los programadores por un gran número de razones:

- 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 distribuyes tu código fuente como un producto, necesitas asegurarte de que está bien hecho y presentado como cualquier otro producto.

Para que funcionen las convenciones, cada persona que escribe software debe seguir la convención. Todos.

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.



13. Conclusión.

A continuación se presentará un resumen comparativo de las características que cada tecnología presenta, 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) El proceso de desarrollo Cliente/Servidor; 2) el grado de integración con Java; 3) instalación y deployment (despliegue); 4) rendimiento; y 5) Escalabilidad. Las distintas tecnologías estudiadas son Sockets, RMI y la integración Java / CORBA.

Los puntos que se consideran en la tabla 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. 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.

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.*

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 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.

Transacciones. Sólo CORBA soporta transacciones.

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. 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.

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.

13. Trabajos futuros.

Si en un comienzo la aplicación se implementó utilizando la tecnología Socket, gracias a este proyecto se ha hecho un nuevo desarrolló utilizando dos tecnología mucho más actuales. Como ya se comentó en el capítulo de metodología, esta aplicación se puede utilizar tanto en entornos de red local, como en Internet, y es esta última posibilidad la que nos permite pensar en algún posible trabajo futuro a desarrollar.

Por todos es conocida, la necesidad de una gran seguridad para cualquier aplicación que utilice como medio de transporte la red Internet, y es que, la posibilidad de que un usuario malicioso pueda utilizar de forma incorrecta la aplicación, puede suponer graves consecuencias. Es por esto, que en este proyecto, se propone como trabajos futuros la aplicación de las tecnologías de seguridad más actuales. Estas es actualmente SSL (Secure Socket Layer).

13.1. Modelo de seguridad Java.

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 codigo 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.

13.1.1. El Control de Acceso.

Para comprobar y asignar unos permisos se usa un fichero llamado java .policy, localizado por defecto en el directorio raiz 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 simple, 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 extensiones. 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 lineas de arriba, ya que así no aparecerá ningun problema relacionado con la seguridad, permitiendo el acceso a la aplicación a cualquier cliente.

13.2. SSL (Secure Socket Layer) con RMI.

Como ya se ha de suponer, aunque se trabaje en un nivel superior al de la capa de transporte, ésta sigue siendo la responsable del establecimiento y mantenimiento de la conexión, además de atender y establecer la comunicación para las llamadas entrantes proporcionando una canal de comunicación fiable entre las capas de la referencia remota del cliente y del servidor. La capa de transporte de RMI está implementada internamente por medio de sockets. Estos sockets son creados por unos objetos denominados SocketFactories, que no son mas que fabricas de sockets. En JAVA 1.2 se han incluido las Custom Socket Factories, que son clases que representan fábricas de sockets definidas por el programador, que le permiten crear sus propios sockets. La utilización de éstas pone al alcance del programador la implementación de cualquier técnica de compresión, encriptación, etc. que el programador desee en el nivel de transporte de RMI.

13.2.1 Análisis de las propiedades de seguridad de RMI

RMI mantiene la misma filosofía en cuanto a seguridad de JAVA para el controlar la ejecución de las clases; el Security Manager controlará la descarga y funcionamiento de las clases provenientes de la red. Es obligatorio que en cualquier implementación RMI se cree una instancia del SecurityManager para garantizar un nivel de seguridad similar al que tienen los applets. Pero si nos damos cuenta, RMI no implementa ninguna política de seguridad en la capa de transporte, por lo que las comunicaciones a través de la red se realizan directamente, es decir, sin tener en cuenta conceptos básicos de seguridad como: autentificación, confidencialidad, etc. El servidor RMI no comprueba quien realiza las peticiones de acceso sobre sus objetos, de forma que un posible cliente “maligno” podría tener acceso a los objetos. Esto es un grave inconveniente en aplicaciones que funcionen sobre redes de comunicación inseguras, como puede ser Internet.

Este problema evidentemente se tiene que poder resolver, y su solución se encuentra en el uso de las Custom Socket Factories, que como ya se dijo permiten al programador crear su propio protocolo de encriptación e implementarlo en sus propios sockets. Pero la creación de un protocolo de encriptación seguro es una acción muy compleja, a la que no todos los programadores pueden aspirar; es por esto por lo que se utilizan las implementaciones existentes y probadas que se encuentran en el mercado, que suelen ser una garantía de calidad. La propuesta de SUN para encriptar las comunicaciones RMI es SSL (Secure Socket Layer).

13.2.2 Secure Socket Layer.

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 nuestras aplicaciones RMI 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, que a su vez puede ser utilizado por RMI. SUN todavía no proporciona un conjunto de clases que permitan utilizar SSL aunque si recomienda una serie de implementaciones comerciales que pueden ser utilizadas

13.2.2.1. 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 mas 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.

13.3 La suite de cifrado.

Como ya se dijo anteriormente, la realización 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.

13.4 Comprendiendo las socket factories.

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 anteriorExtender la clase Socket utilizando los Streams creados en el punto anterior. 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 creados 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. Para ello solo habrá que sustituir la Socket Factory por defecto de RMI mediante la línea de código:
RMISocketFactory.setSocketFactory(new "nueva_socket_factory"());

14. Anexo. Especificación IDL.

14.1. IDL y el mapping de IDL a Java.

La separación entre interfaz e implementación en CORBA está implícita, y se consigue definiendo todos los componentes de la aplicación e incluso los servicios genéricos disponibles utilizando un lenguaje de descripción independiente de la implementación: el IDL ó Interface Definition Language.

Las ventajas que supone la utilización de este lenguje son muchas, y una de ellas es que independientemente del lenguaje utilizado para el desarrollo de la aplicación, este lenguaje siempre debe ser utilizado para definir las interfaces. Al contrario que RMI, por ejemplo, CORBA ofrece ligaduras (language bindings) con C, C++, Ada, COBOL, Java, etc., que permiten que los métodos definidos en el IDL sean implementados utilizando cualquiera de estos lenguajes. Los clientes de estos objetos no son capaces (ni les interesa) distinguir en qué lenguaje fueron implementados los objetos que les proveen servicio. Esta independencia no sólo ayuda a que cada parte del sistema sea implementada en el lenguaje adecuado para su funcionalidad, sino que permite la integración de viejos sistemas existentes en las empresas (legacy systems) dotándolos de un nuevo look e integrándolos de forma transparente en un sistema de Objetos Distribuidos estándar.

Otra de las ventajas es que gracias al compilador idlj la generación del stub y del skeleton es automática. Los sistemas distribuidos requieren mucha programación de bajo nivel para manejar el inicio, flujo y finalización de la comunicación, codificar y decodificar argumentos en el formato en el que se transmitirán, etc.; estos son los stubs del cliente y skeletons del servidor. A través de los compiladores de IDL, el código que realiza todas estas funciones se crea automáticamente. Un cliente de un objeto sólo necesita su IDL para construir

de forma automática el código que le permitirá realizar invocaciones remotas de forma transparente. Recuérdese que estos dos puntos anteriores también los conseguíamos con RMI.

La separación entre interfaz e implementación es también muy importante. Y es que a través del lenguaje de definición de interfaces (IDL) se especifican todos los componentes CORBA. IDL es un lenguaje puramente declarativo con una sintaxis muy parecida a la de C++, pero sin estructuras programáticas. Permite especificar las clases de las que un componente hereda, la signatura de operaciones, los atributos, las excepciones que lanza, y la signatura de métodos (incluyendo argumentos de entrada, de salida y valores de retorno y sus tipos de datos), etc.

Por tanto, IDL es un lenguaje de especificación de interfaces independiente del lenguaje de implementación y que se va a encargar también de estandarizar los tipos y estructuras de manera que todos los objetos que divulgan el ORB las exhiban de igual forma. El estándar CORBA define cómo las distintas construcciones IDL se transforman en signaturas de métodos (o funciones, si el lenguaje de implementación no es Orientado a Objetos) y en atributos (o variables). Aquí se presenta el lenguaje IDL y su adaptación (mapping) para Java. Esta adaptación contempla la equivalencia de tipos de datos, de estructuras de programación y un punto muy importante: la portabilidad y estandarización de los stubs y skeletons.

14.2. Equivalencia de tipos de datos.

La tabla muestra la equivalencia entre los tipos de datos IDL y los tipos de Java. Nótese que el IDL soporta números con signo y sin signo, lo cual deberá ser tratado con mucho cuidado por los programadores Java. En la tabla también se muestran las distintas excepciones que pueden ser lanzadas en caso de un error de conversión.

Tipo IDL	Tipo Java	Excepciones
boolean	boolean	
char	char	CORBA::DATA_CONVERSION
wchar	char	
octet	byte	
string	java.lang.String	CORBA::MARSHAL CORBA::DATA_CONVERSION
wstring	java.lang.String	CORBA::MARSHAL
short	short	
unsigned short	short	
long	int	
unsigned long	int	
long long	long	
unsigned long long	long	
float	float	
double	double	

Después se verán las correspondencias para otros tipos más complejos, como las estructuras, las secuencias, los arrays y el tipo CORBA Any.

14.3. Módulos.

A partir de aquí comenzamos a estudiar el mapping para Java siguiendo la estructura de módulos de IDL. La estructura más externa es el módulo; los módulos pueden contener interfaces u otros módulos, y así. En general, para una definición en IDL como la siguiente:

```
// -*- IDL -*-
module Servers {...};
```

el código Java generado es:

```
// -*- Java -*-
package Servers;
...
```

Para los módulos compuestos como:

```
// -*- IDL -*-
module org {
module omg {
module CORBA {...};
...
};
};
```

se genera:

```
// -*- Java -*-
package org.omg.CORBA;
```

Nótese sin embargo, que para referenciar dentro de IDL, el nombre del módulo es ::org::omg::CORBA, esto es, en vez de utilizar puntos (“.”) como en Java, se utilizan dobles dos puntos (“::”) como en C++ para resolver el ámbito cuando sea necesario.

14.4. Interfaces.

Dentro de un módulo puede haber interfaces, excepciones, estructuras, uniones, enums, secuencias, arrays, etc. El mapping para los interfaces es más complejo. Recuérdese que gracias a los interfaces se podían generar de forma automática los stubs y skeletons. Así, se generan varias clases por cada interfaz. La declaración de una interfaz IDL genera una serie de clases que se describirán mas adelante.

14.5. Estructuras.

El mapping para estructuras (struct), uniones (union) y tipos enumerados (enum) es muy similar. Básicamente se genera la clase con el mismo nombre que la estructura, además de las típicas Helper y Holder. Como un ejemplo, el siguiente IDL:

```
// -*- IDL -*-
...
struct State {
string toolState;
string mechanismState;
string missionState;
short level;
};
...
```

generaría:

```
// -*- Java -*-
final public class State{
public java.lang.String toolState;
public java.lang.String mechanismState;
public java.lang.String missionState;
public short level;
public State() { }
public State(java.lang.String toolState, java.lang.String mechanismState, java.lang.String
missionState, short level) {
        this.toolState = toolState;
        this. mechanismState = mechanismState;
        this. missionState = missionState;
        this. level = level;
}
```

// ...}

14.6. Secuencias y arrays.

El IDL de CORBA ofrece dos tipos de colecciones ordenadas: secuencias y arrays. Una secuencia se mapea a un array unidimensional de Java. Un array puede ser multidimensional y su longitud queda fijada en tiempo de compilación. Hay dos tipos de secuencias: limitada e ilimitada. Un ejemplo se puede ver en el siguiente código IDL:

```
// -*- IDL -*-
typedef sequence<string> ilimitada;
typedef sequence<short,30> limitada;
```

Este fragmento IDL no genera ninguna definición, tan sólo las clases Holder. El programador es el encargado de definir en el interior de su programa las variables adecuadas:

```
// -*- Java -*-
public java.lang.String[] valor; // ilimitada
public short[] valor; // limitada
```

Del mismo modo ocurre con los arrays.

14.7. Atributos y operaciones.

Dentro de una interfaz, como es normal en la mayoría de los lenguajes Orientados a Objetos, se pueden definir atributos y métodos (operaciones en terminología CORBA). Los atributos son llevados a Java como un par de operaciones sobrecargadas, una de acceso y otra de cambio del atributo. Los atributos se pueden definir como de sólo lectura (readonly), en cuyo caso sólo el método de acceso es dado (sólo pueden consultarse, no modificarse). La sintaxis utilizada es del tipo:

[readonly] attribute tipo_de_atributo nombre_de_atributo ;

En general se corresponde a dos operaciones:

1)tipo_de_atributo getNombre_de_atributo();
2)void setNombre_de_atributo(in tipo_de_atributo nuevo_valor);

```
// -*- IDL -*-
interface Server {
attribute string name;
```

readonly attribute string IP;
// ..};

generaría:
// -- Java -*-*
public interface Server extends org.omg.CORBA.Object {
public String name();
public void name(String val);
public string IP();
// ...};
y las clases Helper y Holder si son necesaria. Como se ve, dos métodos para el atributo de lectura y escritura, y un método para el que sólo es de lectura.

La definición de métodos tiene la siguiente sintaxis:
[oneway] <tipo> <identificador> (<parámetros>) [<raises>] [<context>]

Donde oneway especifica que la operación se llama sólo una vez de forma informativa y no se espera a su ejecución (normalmente utilizado en métodos de finalización o de ping), <tipo> especifica el tipo del valor devuelto (o void si no devuelve nada), <identificador> es el nombre de la operación. Los <parámetros> son una lista separadas por comas de valores: {in | out | inout} <tipo> <identificador>, donde in especifica un parámetro de entrada, out uno de salida e inout uno de entrada y salida. <raises> utiliza la palabra "raises" y una lista entre paréntesis de excepciones que la operación puede lanzar y <context> utiliza la palabra "context" seguida de una lista de identificadores encerrados entre paréntesis y separados por comas. Esto último se utiliza para establecer el contexto de la operación, es decir, para pasarle información adicional.

Para permitir que los tipos estándar de Java puedan participar en parámetros out ó inout, se han añadido unas clases en el paquete org.omg.CORBA: ShortHolder, IntHolder, etc. tan sencillas como:
// -- Java -*-*
package org.omg.CORBA;
final public class ShortHolder {
public short value;
public ShortHolder() {}
public ShortHolder (short initial) {
value = initial;}

```
}
```

Como un ejemplo de mapping para una operación, considérese el siguiente ejemplo:

```
// -*- IDL -*-
interface MechanismCtrl {
void move(in short x, inout double y);
};
```

que generaría en Java:

```
// -*- Java -*-
public interface MechanismCtrl extends org.omg.CORBA.Object {
void move(short x, DoubleHolder y)
}
```

El uso que se haría de esta operación move() sería el siguiente:

```
// -*- Java -*-
MechanismCtrl MechanismServer = new MechanismCtrl ();

// Parámetros. Para el tipo 'inout' se debe usar un 'Holder'
DoubleHolder y = new DoubleHolder(0.122343);
short x = 3;

// Invocar la operación de ajuste
MechanismServer.move(x,y);
```

14.8. Excepciones.

CORBA, al igual que Java, también da la posibilidad de definir excepciones por parte del usuario. Además, CORBA también posee unas excepciones del sistema. Todas las excepciones del usuario extienden a org.omg.CORBA.UserException (que, a su vez, extiende a java.lang.Exception). Las excepciones del sistema extienden a org.omg.CORBA.SystemException (que extiende a java.lang.RuntimeException). Como ejemplo, considérese el siguiente código IDL:

```
// -*- IDL -*-
module Exceptions
{
```

```
        exception CloseException{string reason;};
};
```

generaría:

```
// -*- Java -*-
package Exceptions;
public final class CloseException extends org.omg.CORBA.UserException implements
org.omg.CORBA.portable.IDLEntity {
public String reason = null;

public CloseException ()  { }  // Constructor por defecto
public CloseException (String _reason) { reason  = _reason; }  // Constructor}
```

Hay varias excepciones del sistema, que son adaptadas como clases Java en el paquete org.omg.CORBA. Entre ellas, CORBA::MARSHAL a la clase org.omg.CORBA.MARSHAL, CORBA::BAD_OPERATION a la clase org.omg.CORBA.BAD_OPERATION, etc.

14.9. El tipo Any.

El tipo IDL Any sirve para especificar que un parámetro puede ser de cualquier tipo. Se adapta a Java utilizando la clase org.omg.CORBA.Any. La clase posee un constructor por defecto que construye un objeto Any vacío. La clase ofrece métodos insert_<tipo>() y extract_<tipo>(), donde <tipo> es el tipo IDL en cuestión. Los métodos “insert” retornan un Any, y los “extract” retornan un <tipo>. Los últimos son declarados como posibles lanzadores de la excepción org.omg.CORBA.BAD_OPERATION.

Por ejemplo, para crear un objeto de tipo Any que contenga un short, podemos escribir:

```
// -*- Java -*-
Any x = new Any().insert_short( 6 );

// Y ahora una actualización
x.insert_short( 12 );
```

Para el tipo Any es fundamental el interfaz TypeCode que se adapta a Java como la clase org.omg.CORBA.TypeCode. Este tipo define el tipo de los objetos que populan el ORB y que pueden encontrarse en el Interface Repository. Cada interfaz definido en IDL, puede

devolver un objeto TypeCode que identifica unívocamente del tipo que es. Un TypeCode contiene un "kind" (un entero que identifica al tipo) y unos parámetros específicos de ese tipo. Los identificadores de tipo para los tipos estándar IDL se encuentran en constantes estáticas de la clase org.omg.CORBA.TCKind como TCKind.tk_<tipo>. Existen, además, unos objetos TypeCode que representan a cada tipo estándar: TCKind.tc_boolean, TCKind.tc_char, TCKind.tc_TypeCode, etc.

14.10. Stubs y Skeletons portables.

Cuando el mapping a otros lenguajes hubiera terminado, en Java se necesita algo más. La razón es clara: las clases stub y skeleton viajan por la red en forma compilada (bytecodes) y son cargadas en ORBs que pueden ser de distintos fabricantes. Se debe definir unos stubs y skeletons estándar de manera que puedan ser cargados directamente en tiempo de ejecución y funcionar independientemente del fabricante del ORB. Nótese que hasta la aparición del mapping a Java, el código interno de los stubs y skeletons no era especificado por CORBA y eran, por tanto, propietarios.

La portabilidad se consigue a través del paquete org.omg.CORBA.portable. Todos los stubs portables extienden a la clase org.omg.CORBA.potable.ObjectImpl. Todos los skeletons heredan de org.omg.CORBA.DynamicImplementation.

Es realmente admirable cómo se ha conseguido esta portabilidad. Los stubs son sorprendentemente sencillos, utilizan un interfaz común en todos los ORBs compatibles con CORBA 2.0: el DII (Dynamic Invocation Interface, Interfaz de Invocación Dinámica). Las llamadas al objeto stub del cliente se traducen en llamadas al interfaz de invocación dinámica con el método específico. Así los stubs son totalmente portables. Normalmente, los compiladores de IDL poseen una opción para generar este tipo de stubs, ya que normalmente son más lentos que los propietarios, al utilizar invocación dinámica.

Al compilar con idlj –portable se genera una clase llamada _portable_stub_<interfaz>. Para los skeletons se utiliza el interfaz simétrico: el DSI (Dynamic Skeleton Interface, Interfaz de Skeletons dinámico). La clase de la que heredan, DynamicImplementation tiene la siguiente signatura:

```
// -*- Java -*-
package org.omg.CORBA;
public abstract class DynamicImplementation extends org.omg.CORBA.portable.ObjectImpl{
```

```
        public abstract void invoke(org.omg.CORBA.ServerRequest request);
}
```

Implementando el método invoke(), los skeletons pueden recibir todo tipo de peticiones, ya que la información de cada una se guarda en el objeto ServerRequest. Así, tanto stubs como skeletons se construyen a partir del API estándar de CORBA y son, por ello, portables.

14.11. Interfaces IDL.

Una vez que sabemos como se especifica una interfaz IDL y su mapeo a Java, vamos a analizar algunas interfaces importantes del proyecto y las clases Java generadas por el compilador del JDK1.3. idlj. Veamos, por ejemplo, la interfaz MechanismCtrl.idl, que define las operaciones que proporcionara el objeto servidor remoto que manejan al brazo del robot (Mechanism), junto con los ficheros generados por el compilador de IDL a Java:

```
// Interfaz MechanismCtrl
/*  Como los métodos que se definirán a continuación lanzan excepciones propias (definidas
por usuario), debemos de definir dichas excepciones en esta interfaz. Estas excepciones se
almacenaran dentro del paquete Exceptions. */
module Exceptions
{
        exception RegistryException{string s;};
        exception CloseException{string s;};
        exception ResolveException{string s;};
};
/* Una vez que se han definido las excepciones que pueden lanzarse en los métodos que
proporciona la interfaz MechanismCtrl, estamos listos para especificar las operaciones que
define en si la interfaz. */

/* Esta interfaz se almacenará en el paquete Controls.Mechanism */
module Controls{
module Mechanism{
interface MechanismCtrl{

/* Definimos una array ilimitado que contenga float, ya que en algunos métodos es necesario
pasarlo como parámetro. */
typedef sequence<float> arrayOfFloat;
```

```
/* Métodos definidos en la interfaz. Muchos de ellos requieren parámetros de entrada de
distintos tipos basados en la especificación de la interfaz. */
void enableJoint(in long jointNumber);
void disableJoint(in long jointNumber);
void enableAll();
void disableAll();
void setMaxSpeed(in float maxSpeed);
void setSpeed(in float relSpeed);
void moveForward(in long jointNumber);
void moveReverse(in long jointNumber);
void shiftJoint(in long jointNumber,in float shift);
void shift(in arrayOfFloat destination);
void moveTo(in arrayOfFloat destination);
void setJointUpLimit(in long JointNumber, in float upLimit);
void setJointLrLimit(in long JointNumber, in float lrLimit);
void setJointMaxTorque(in long JointNumber,in float maxTorque);
void setJointOrigin(in long JointNumber);
void addMechanismListener(in string stringAccess);
void addTimeMechanismListener(in string stringAccess, in long period);
void removeMechanismListener(in string stringAccess ,in long registrationMode);

/* Métodos que lanzan las excepciones que se definieron anteriormente. */
void getMechanismStatus(in string stringAccess) raises(Exceptions::ResolveException);
void getMechanismAlarms(in string stringAccess) raises(Exceptions::ResolveException);
void getMechanismAlarmStates(in string stringAccess) raises(Exceptions::ResolveException);
void getMechanismAlarmMessages(in string stringAccess)
raises(Exceptions::ResolveException);
void getJointStatus(in string access,in long jointNumber)
raises(Exceptions::ResolveException);
void closeApp() raises(Exceptions::CloseException); };
};
};
```

Los ficheros Java generados por el compilador idlj para esta interfaz son los siguientes:

- Para las excepciones se crea un directorio (paquete) Exceptions, que contiene las clases Java asociadas a las excepciones y las clases Helper y Holder correspondientes.

Veamos el código Java generado solamente para la excepción CloseException. Podemos comprobar que deriva de org.omg.CORBA.UserException debido a que es una excepción propia (definida por el usuario). Esta clase únicamente contiene el constructor y el constructor asignado por defecto.

```
// Exceptions/CloseException.java
package Exceptions;

public final class CloseException extends org.omg.CORBA.UserException implements
org.omg.CORBA.portable.IDLEntity {
  public String s = null;

  public CloseException ()  {  }
  public CloseException (String _s)  {  s = _s;  }
}
```

Esta clase es una clase abstracta que implementa métodos estáticos para operar con excepciones (objetos) de tipo CloseException. Verificamos que contiene los métodos visto en el apartado de compilación de interfaces IDL. Estos métodos se explican junto al código a continuación:

```
// Exceptions/CloseExceptionHelper.java
package Exceptions;
abstract public class CloseExceptionHelper
{
  private static String  _id = "IDL:Exceptions/CloseException:1.0";

// Inserta un objeto CloseException dentro de un objeto Any.
  public static void insert (org.omg.CORBA.Any a, Exceptions.CloseException that) {
    org.omg.CORBA.portable.OutputStream out = a.create_output_stream ();
    a.type (type ());
    write (out, that);
    a.read_value (out.create_input_stream (), type ());
  }

// Extrae un objeto CloseException de un objeto Any (inverso a insert).
  public static Exceptions.CloseException extract (org.omg.CORBA.Any a) {
```

```
    return read (a.create_input_stream ());
  }

  private static org.omg.CORBA.TypeCode __typeCode = null;
  private static boolean __active = false;

// Retorna un identificador del tipo tal y como será encontrado en el Interface Repository.   //
Esto es muy conveniente para indicar el tipo de los parámetros y el resultado de las        //
invocaciones que se construyen de forma dinámica a través del DII (Dynamic Invocation   //
Interface).
  synchronized public static org.omg.CORBA.TypeCode type ()  {
    if (__typeCode == null) {
          synchronized (org.omg.CORBA.TypeCode.class)  {
                if (__typeCode == null){
                      if (__active)  {
                          return org.omg.CORBA.ORB.init().create_recursive_tc ( _id ); }
                      __active = true;
        org.omg.CORBA.StructMember[] _members0 = new org.omg.CORBA.StructMember [1];
        org.omg.CORBA.TypeCode _tcOf_members0 = null;
        _tcOf_members0 = org.omg.CORBA.ORB.init ().create_string_tc (0);
        _members0[0] = new org.omg.CORBA.StructMember ( "s", _tcOf_members0, null);
        __typeCode = org.omg.CORBA.ORB.init ().create_struct_tc
(Exceptions.CloseExceptionHelper.id (), "CloseException", _members0);
        __active = false;
       }
     }
    }
    return __typeCode;
  }
  public static String id () {
    return _id;  }

// Utilidad de serialización.
 public static Exceptions.CloseException read (org.omg.CORBA.portable.InputStream istream){
    Exceptions.CloseException value = new Exceptions.CloseException ();
    // read and discard the repository ID
    istream.read_string ();
```

```
    value.s = istream.read_string ();
    return value; }

// Utilidad de serialización.
  public static void write (org.omg.CORBA.portable.OutputStream ostream,
Exceptions.CloseException value) {
    // write the repository ID
    ostream.write_string (id ());
    ostream.write_string (value.s); }
}
```

IDL soporta el uso de parámetros in (entrada), out (salida) e inout (entrada/salida). Java sólo soporta parámetros de entrada. Para ello, se crean unas clases "Holder" que "guardan" un valor de cada tipo (véase el objeto "value" de tipo Exceptions.CloseException).

```
// Exceptions/CloseExceptionHolder.java
package Exceptions;

public final class CloseExceptionHolder implements org.omg.CORBA.portable.Streamable{
  public Exceptions.CloseException value = null;
  public CloseExceptionHolder () { }
  public CloseExceptionHolder (Exceptions.CloseException initialValue) {
    value = initialValue; }
  public void _read (org.omg.CORBA.portable.InputStream i) {
    value = Exceptions.CloseExceptionHelper.read (i); }
  public void _write (org.omg.CORBA.portable.OutputStream o) {
    Exceptions.CloseExceptionHelper.write (o, value); }
  public org.omg.CORBA.TypeCode _type () {
    return Exceptions.CloseExceptionHelper.type (); }
}
```

- Para la interfaz MechanismCtrl se crea un directorio (paquete) donde se almacenan las clases generadas. Podemos comprobar los visto anteriormente sobre el mapeado de IDL a Java:

Los módulos IDL se corresponden con los paquetes Java. Esto implica que el módulo Controls genere como resultado un paquete con ese mismo nombre, y el modulo Mechanism genere otro paquete con ese mismo nombre a su vez dentro del paquete Controls, cuyo elementos se almacenan en una carpeta también llamada Controls y Mechanism

respectivamente. En la carpeta Mechanism (que está dentro de Controls) se almacena un archivo con la interfaz en lenguaje Java, un stub, un skeleton y dos archivos con clases auxiliares. Vamos a analizar cada uno de estos elementos con algo más de detalle.

La interfaz MechanismCtrl.java contiene la versión en Java del interfaz en IDL. En esta interfaz no se encuentran los métodos definidos en la interfaz IDL, sino que se encuentran en otra clase (MechanismCtrlOperations.java), la cual hereda para poder tener acceso a esos métodos. Los clientes desean referencias a objetos que implementen este interfaz, al estilo de la programación Java normal.

No hay que olvidar que esta interfaz Java se utilizará para implementar un objeto CORBA o bien para facilitar que un cliente acceda a dicho objeto. Por ello la interfaz MechanismCtrl está basada en org.omg.CORBA.Object, una interfaz que cuenta con los métodos necesarios para mantener, comprobar y manipular la referencia a un objeto CORBA. La otra interfaz de la que desciende MechanismCtrl, llamada org.omg.CORBA.portable.IDLEntity, no aporta método alguno y sirve tan sólo como una marca utilizada para saber si un cierto objeto es una implementación IDL y, por lo tanto, dispone de una clase Helper que facilita su gestión.

```
// Controls/Mechanism/MechanismCtrl.java
package Controls.Mechanism;

public interface MechanismCtrl extends MechanismCtrlOperations, org.omg.CORBA.Object,
org.omg.CORBA.portable.IDLEntity { }
```

El compilador de IDL idlj mapea todas las operaciones definidas en la interfaz IDL en este fichero (MechanismCtrlOperations), que es compartido tanto por los stubs como por los skeletons. Comprobamos que los métodos que lanzaban excepciones en IDL (raises) las lanzan también en su equivalente Java (throw); y que algunos tipos como long en IDL equivalen a int en Java, lo mismo ocurre con string y String respectivamente (como indica la tabla de conversión de tipos vista anteriormente).

```
// Controls/Mechanism/MechanismCtrlOperations.java
package Controls.Mechanism;

public interface MechanismCtrlOperations {
void enableJoint (int jointNumber);
void disableJoint (int jointNumber);
```

void enableAll ();
void disableAll ();
void setMaxSpeed (float maxSpeed);
void setSpeed (float relSpeed);
...

14.12 El esqueleto de servidor.

Partiendo de la interfaz Java que se comento anteriormente, el compilador IDL ha generado un esqueleto en el cual se implementan los métodos de bajo nivel y carácter genérico. Dicho esqueleto está contenido en el archivo _MechanismCtrlImplBase.java. Como se puede ver, _MechanismCtrlImplBase es una clase abstracta que deriva de org.omg.CORBA.portable.ObjectImpl e implementa las interfaces Controls.Mechanism.MechanismCtrl y org.omg.CORBA.portable.InvokeHandler.

Este esqueleto de servidor es abstracto puesto que su funcionalidad es únicamente servir como clase base de otra, escrita por nosotros, en la que realmente se implementen los métodos de la interfaz MechanismCtrl. _MechanismCtrlImplBase incorpora la implementación de los métodos necesarios para identificar a un objeto de esta clase, así como para realizar llamadas dinámicas a los métodos de la interfaz MechanismCtrl. Lo primero que encontramos en _MechanismCtrlImplBase es un constructor por defecto, que no necesita parámetro alguno y se limita a ejecutar el constructor de la clase base. A continuación encontramos una cadena con la identificación de la clase, cadena que es devuelta por el método _ids() implementado poco más abajo.

Un cliente que dispone del stub apropiado puede ejecutar llamadas estáticas a los métodos expuestos en la interfaz de un servidor CORBA. Esto, de hecho, es lo habitual. Es posible, no obstante, realizar llamadas dinámicas prescindiendo del stub y sirviéndose de un repositorio de interfaces. En cualquier caso, el método invoke() es el encargado de recibir cualquier invocación a un método de la interfaz MechanismCtrl, que se traducirá en una llamada al método que corresponda, para lo cual se busca en una tabla de búsqueda poco antes. Dicha tabla contiene el nombre de cada método y un índice, como se puede ver en el código de esta clase. En este caso existen 24 elementos, cada índice del 0 al 23 corresponde a un método diferente.

Como veremos más adelante, la clase _MechanismCtrlImplBase realmente nos servirá para crear la implementación propiamente dicha de nuestro servidor. No es necesario conocer el

funcionamiento de _MechanismCtrlImplBase para poder desarrollar un servidor CORBA, pero siempre resulta interesante tener una idea de lo que está ocurriendo.

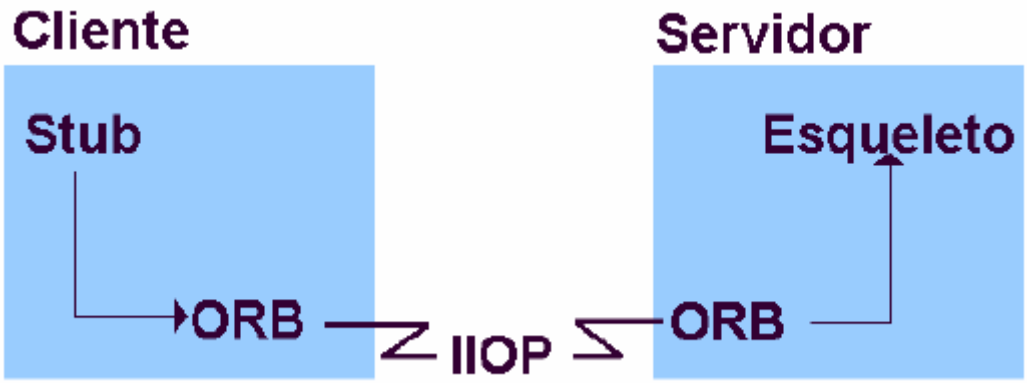
14.13 El stub para el cliente.

Un componente CORBA que actúa como servidor puede estar ejecutándose en cualquier maquina conectada a una red, recibiendo peticiones por parte de clientes que pueden ser tanto locales como remotos. Indistintamente de ello, el cliente siempre tendrá la ilusión de que la llamada que esta realizando se ejecuta localmente. Esto es así gracias a la clase stub generado por el compilador IDL. Al compilar nuestra interfaz IDL MechanismCtrl el código generado en el archivo _MechanismCtrlStub.java es el que puede verse más abajo.

La finalidad del stub es, por lo tanto, evitar que el programador tenga que ocuparse de dar todos los pasos que sean necesarios para poder ejecutar un método de un objeto remoto. Para ello la clase _MechanismCtrlStub implementa la interfaz MechanismCtrl, de tal forma que es posible llamar directamente a cualquiera de sus métodos facilitando los parámetros necesarios (en cada uno de ellos) y obteniendo el resultado correspondiente.

Como puede verse si se analiza el el código, la implementación de los métodos de la interfaz que hay en el stub no realiza realmente ninguna operación con los parámetros. Seremos nosotros los que tengamos que escribir el código de cada método, cuando implementemos el servidor. En el stub lo que se hace es utilizar las clases del ORB para preparar y ejecutar la llamada remota, básicamente creando un objeto org.omg.CORBA.Request y usando el método invoke() para llamar a cada método de la interfaz en cada caso. Obsérvese que tras crear el mencionado objeto se establece el tipo del valor de retorno, en este caso una cadena, procediéndose a continuación a preparar una lista con los parámetros de entrada. Dados estos pasos se ejecuta la llamada y se obtiene el valor de retorno, que es lo que finalmente se devuelve.

Si obviamos los detalles aislados en la clase que actúa como stub, usando este módulo sin preocuparnos de su contenido, a la hora de crear el cliente veremos que básicamente es como su obtuviésemos una referencia a un objeto y realizásemos una llamada local a uno de sus métodos. La realidad, sin embargo, será mucho más compleja, tal y como se muestra de forma simplificada en la siguiente figura:



Esquema simplificado que muestra la estructura de un cliente y un servidor, con sus correspondientes ORB usando el protocolo IIOP para comunicarse por una red IP.

14.14 Clases auxiliares.

Además de los archivos comentados hasta ahora, el compilador IDL genera dos más que contienen clases auxiliares. Se trata de clases cuya finalidad es contener básicamente métodos estáticos, es decir, que pueden ser ejecutados sin necesidad de crear un objeto de esa clase. Las dos clases en cuestión son MechanismCtrlHelper y MechanismCtrlHolder.

Las referencias genéricas a objetos java pueden ser convertidas en referencias de un cierto tipo mediante la habitual técnica de modelado, existente también en otros lenguajes como C/C++ y Pascal. Una referencia a un objeto genérico CORBA, por el contrario, no puede convertirse mediante un modelado de tipo en Java, siendo necesario unos métodos alternativos. El método más interesante de la clase MechanismCtrlHelper, llamado narrow(), tiene la finalidad de facilitar la conversión de una referencia genérica a una del tipo que nos interesa lo que, en definitiva, permitirá usar esa referencia para llamar a los métodos establecidos originalmente en la interfaz. Lógicamente cada interfaz que se defina en un módulo IDL contará con su propia clase XXXXHelper, con un método narrow() específico para ella. Éste método se usara posteriormente a la hora de implementar un cliente que utilice alguno de los métodos de la interfaz MechanismCtrl.

Además de estas clases, se generan otras clases en un paquete llamado MechanismCtrlPackage, donde se almacenan el código Java generado por la definición del un array en la interfaz IDL. Concretamente, la línea “typedef sequence<float> arrayOfFloat;” en la interfaz que estamos estudiando genera únicamente las clases Holder y Helper asociadas a este array de float.

15. Bibliografía

RMI & CORBA:

- Http://www1.Ceit.Es/Sitr/Trabajos/Grupo10/Trabajo2/Trabajo_2_10.Htm
- Http://www.Programacion.Com/Java/Idl/Index.Php
- Http://www.Programacion.Com/Java/Rmi/Index.Php
- Http://www.Javahispano.Com/
- Http://www.Osmosislatina.Com/Java/Rmi.Htm
- Http://Turing.Gsi.Dit.Upm.Es/~Vferval/Introduccion.Html
- Http://Developer.Java.Sun.Com/Developer/Technicalarticles/RMI/
- Http://Developer.Java.Sun.Com/Developer/Onlinetraining/Rmi/
- Http://Developer.Java.Sun.Com/Developer/Earlyaccess/Idlc/
- Http://Java.Sun.Com/Products/Jdk/Rmi/
- Http://Docs.Rinet.Ru:8083/Jsol/Ch19.Htm
- www.Unidata.Ucar.Edu/Staff/Robb/Gemstone/Corba/Concepts/Idlcompiling.Html
- Http://www.Javaworld.Com/Javaworld/Jw-10-1997/Jw-10-Corbaja V A _ P Html
- "Introducción a CORBA", Juan Pavón Mestras, Dep. Sistemas Informáticos y Programación, Universidad Complutense Madrid.
- Http://Java.Sun.Com/J2se/1.4/Docs/Guide/Idl/Jidlexample.Html
- Http://Turing.Gsi.Dit.Upm.Es/~Vferval/Pfcvic.Html
- "Java RMI" Seminario De Sistemas Distribuidos Marco Aravena V., Rudy Malonnek Universidad Técnica Federico Santa María, Departamento de Electrónica.
- Invocación Remota De Objetos RMI Joaquin Salvachua
- "CORBA con Java IDL" Francisco Charte Ojeda
- "Taller Java. RMI mano a mano con SSL: Construyendo aplicaciones distribuidas seguras", Carlos Beltrán González.
- [Orfali, 1998] Client/Server programming with Java and CORBA, Orfali, Edición: 2ª, ISBN: 047124578X, 1998.
- [Vogel, 01] Java programming with CORBA, Vogel, 3ª Edición, ISBN: 0471376817, SOFTCOVER.
- [Castanet , 99] Java cliente-servidor: JDK 1.1, JavaBeans, JDBC, Corba/RMI, Marimba Castanet, ISBN: 84-8088-268-9

Sistemas Distribuidos:

- "Introducción a Los Sistemas Distribuidos", Raúl Pérez.
- Http://Uxmcc1.Iimas.Unam.Mx/~Cursos/Algoritmos/Algoritmos-Distrib.Html

- Comunicación De Procesos En Sistemas Distribuidos Félix García Carballeira

Arquitectura Cliente-Servidor:

- "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.

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.
- 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.

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.