Benemérita Universidad Autónoma de Puebla

Facultad de Ciencias de la Computación

Infraestructura Genérica de Flujo de Objetos para aplicaciones distribuidas bajo ProActive

Tesis

que para obtener el grado de:

Maestro en ciencias de la Computación

Presenta:

Adriana Hernández Beristain

Asesores:

Dr. Denis Caromel
M. C. David Eduardo Pinto Avendaño

Puebla, Pue. **Mayo del 2004**

Contenido

Índice de Ilustraciones

Introducción

La transmisión de contenido multimedia a través de Internet es cada vez más importante, en este caso es necesario considerar el envío de objetos tales como, archivos de audio, video, imagen, texto, etc. Más aún, el uso de flujos de transferencia (streaming en inglés) ha venido a ser una tendencia. La tecnología de streaming es un mercado con futuro y grandes compañías ya están luchando por el dominio del mismo. La velocidad de las redes aumentará indudablemente con el tiempo y con ella aumentará la calidad de las transmisiones. Mientras tanto, es necesario desarrollar recursos que apoyen en la administración de dichas transferencias.

La tecnología de streaming se utiliza para aligerar la descarga y ejecución de audio y vídeo, permitiendo escuchar y visualizar los archivos mientras se están descargando.

Si no se utiliza el streaming para mostrar un contenido multimedia, se debería descargar primero el archivo completo en la computadora y posteriormente ejecutarlo, para finalmente ver y oír el contenido del mismo. Sin embargo, el streaming permite que esta tarea se realice de una manera rápida y que sea posible ver y escuchar el contenido aún cuando la descarga está siendo realizada.

El streaming funciona de la siguiente manera: inicialmente la computadora (el cliente) solicita un flujo a un equipo servidor, el cual comienza a enviar el archivo. El cliente entonces recibe parte del archivo y construye un *buffer* en el cual guarda la información. Cuando se ha llenado el *buffer*, el cliente empieza a mostrar el contenido y a la vez continúa con la descarga. El sistema está sincronizado para que el archivo pueda ser visualizado mientras se descarga, de modo que cuando el archivo acaba de descargarse el archivo también ha acabado de visualizarse. Si en algún momento la conexión sufre descensos de velocidad se utiliza la información que existe en el *buffer*, de modo que es posible aplazar un tiempo más el descenso. Si la comunicación se interrumpe demasiado tiempo, el *buffer* se vacía y la ejecución del archivo se interrumpe también hasta que la señal sea restaurada.

El desarrollo de infraestructuras genéricas para el flujo de objetos ha sido propuesto en algunos trabajos; básicamente para el envío de multimedia como se propone en [6]; otra aproximación está siendo desarrollada por HP [7], donde su principal objetivo es investigar sobre sistemas de flujo de medios sobre Internet. En algunos casos, como en [8], se motiva el uso de flujo de objetos para aplicaciones particulares, como es el caso de realidad aumentada. Sin embargo, un módulo particular para el flujo de objetos en la biblioteca de ProActive PDC ha sido una necesidad desde la creación de la misma biblioteca de funciones. El objetivo principal de este trabajo de tesis es proporcionar un conjunto de primitivas básicas para el control y flujo de objetos que sea utilizado en un futuro en el

desarrollo de aplicaciones distribuidas, paralelas y distribuidas que hagan uso de ProActive.

Aunque ProActive es una biblioteca para cómputo paralelo, distribuido y concurrente Orientado a Objetos en Java en constante crecimiento, la carencia este tipo de primitivas que incluso incluyan la administración de flujo de objetos, hacen que el objetivo de ésta tesis sea importante al ampliar el contenido de la biblioteca.

Se propone un marco de trabajo paralelo y distribuido para la construcción de aplicaciones que usen flujo de objetos. Esta infraestructura genérica está basada en ProActive, una capa intermedia (ambiente y modelo de programación) orientada a objetos para cómputo paralelo, móvil, y distribuido. El objetivo es extender la biblioteca de ProActive mediante la implementación de un modelo de componentes jerárquico y dinámico para el flujo de objetos, y mediante el cual se puede tomar ventaja de esta infraestructura genérica para el desarrollo de aplicaciones complejas basadas en ProActive.

Un objetivo adicional de esta tesis es proponer una arquitectura para la automatización, identificación e integración de componentes reutilizables de software, en particular para el flujo de objetos.

La estructura de la tesis se encuentra distribuida como a continuación se indica.

En el capítulo 1 se describe el marco teórico de las aplicaciones distribuidas así como la evolución de las distintas herramientas para su desarrollo.

Un enfoque detallado de la biblioteca ProActive PDC se presenta en el capítulo 2, describiendo sus principales componentes.

El capítulo 3 muestra diversas técnicas y propuestas para el flujo de objetos.

El diseño de la infraestructura genérica se presenta en el capítulo 4. En este capítulo se comenta sobre la filosofía de funcionamiento. El modelado de la infraestructura se presenta a través del lenguaje de modelado unificado (UML).

El capítulo 5 muestra algunos ejemplos de utilización del conjunto de primitivas desarrollado. El lector podrá usar este capítulo como referencia y asociar las actividades de las clases desarrolladas para implementar algún sistema basado en ProActive que necesite del flujo de objetos en ambientes distribuidos, paralelos y concurrentes.

Finalmente, se presentan las conclusiones del trabajo realizado, las metas alcanzadas así como las líneas de trabajo derivadas del presente trabajo.

Capítulo I Marco Teórico.

I.1 Cómputo Distribuido

El cómputo Paralelo y Distribuido emergió como una técnica importante para acelerar cálculos que son demasiado lentos en computadoras tradicionales con un solo procesador. La idea consiste en descomponer un problema en problemas más pequeños cuyas soluciones se integran como solución al problema original. Es sin embargo más fácil decirlo que hacerlo y, después de casi 50 años de estudio, el cómputo paralelo y distribuido todavía lucha por la atención de la comunidad del cómputo de alto rendimiento.

Hoy en día no sólo es posible, sino fácil, reunir sistemas de cómputo compuestos por un gran número de CPU´s, conectados mediante una red de alta velocidad. Estos reciben el nombre genérico de sistemas distribuidos, De manera formal se definen a los sistemas distribuidos de la forma siguiente: “Un sistema distribuido es una colección de computadoras independientes que aparecen ante los usuarios del sistema como una única computadora” [1]. Así, los sistemas distribuidos son una de las grandes áreas de las Ciencias de la Computación, la cual se dedica al estudio y solución de los problemas que se presentan en el desarrollo de sistemas de cómputo distribuido. Esta área se divide en las siguientes subáreas:

- Redes de comunicaciones para computadoras (Internet e Intranet).
- Bases de datos distribuidos.
- Sistemas operativos distribuidos y de red.
- Sistemas cliente-servidor.
- Sistemas multimedia distribuidos.
- Cómputo paralelo.
- Sistemas de tiempo real distribuidos.
- Sistemas de control distribuido.

Por otro lado los sistemas distribuidos necesitan además un software radicalmente distinto al de los sistemas centralizados. En particular los sistemas operativos necesarios para estos sistemas distribuidos están apenas en una etapa de surgimiento.

I.1.1 Antecedentes

Históricamente el problema que trata de resolver el cómputo distribuido es la manera de distribuir el cómputo entre varios sistemas que trabajan en conjunto para resolver un problema dado. El concepto abstracto más utilizado en el campo de cómputo distribuido es el RPC (Remote Procedure Calls - llamados a

procedimientos remotos". Los RPC permiten a una función remota ser llamada como si se tratara de una local. Los sistemas distribuidos orientados a objetos requieren RPC basados en objetos (ORPC).

La historia del cómputo distribuido y objetos distribuidos es un poco complicada; la siguiente cronología presenta algunos eventos que permitirán al lector entenderla [Internet 1].

1987

- La compañía Sun Microsystems desarrolla el RPC de cómputo de red abierta (ONC - Open Network Computing) como el mecanismo básico de comunicación para su sistema de archivos de red (NFS - Network File System).
- La compañía Apollo Computer desarrolla el RPC de su sistema de cómputo en red (NCS - Network Computer System) para su sistema operativo Dominio.

1989

- La Fundación de Software Abierto (OSF - Open Software Foundation, ahora conocida como The Open Group) lanzó una convocatoria para un sistema RPC. Fué elegida la propuesta NCS de la compañía HP/DEC como el mecanismo RPC para su Ambiente de Cómputo Distribuido (DCE - Distributed Computing Environment).
- Se conforma el Grupo de Administración de Objetos (OMG - Object Management Group) para coordinar las especificaciones para cómputo distribuido independiente de plataforma y lenguaje. El OMG inició el desarrollo de las especificaciones de una plataforma de objetos distribuidos (CORBA - Common Object Request Broker Architecture).

1990

- Microsoft basa sus iniciativas de RPC en una versión modificada de DCE/RPC.

1991

- La OSF libera el DCE 1.0.
- Se libera CORBA 1.0, solamente para el lenguaje C. Se populariza el ORB (Object Request Broker)

1996

- Microsoft presenta el Modelo de Objeto Componente Distribuido (DCOM - Distributed Component Object Model) cercano a la familia de componentes de Microsoft como el OLE (Object Linking and Embedding), COM no distribuido (OLE2) y ActiveX. La parte central de las habilidades de DCOM están basadas en las tecnologías RPC de Microsoft. DCOM es un protocolo ORPC.
- Se presenta CORBA 2.0 es presentado con muchas mejoras en el modelo de cómputo distribuido especialmente en servicios de alto nivel que pueden utilizar los objetos distribuidos. El Protocolo de Internet Inter-ORG (IIOP - Internet Inter-ORB) presentado permite la ínter operación de múltiples ORBs.

1997

- La compañía Sun Microsystems presenta la herramienta de desarrollo de Java (JDK 1.1) que incluye el Método de Invocación Remota (RMI - Remote Invocation Method). El RMI presenta un modelo de cómputo distribuido utilizando objetos de Java. El RMI es similar al CORBA y DCOM, pero sólo trabaja con objetos Java. El RMI es un protocolo ORPC llamado Protocolo de Método Remoto de Java (JRMP - Java Remote Method Protocol).
- Microsoft anuncia su primera versión de COM+, el sucesor de DCOM. Las características de COM+ se acercan más al modelo CORBA de cómputo distribuido.

1998

- Sun Microsystems presenta J2EE (Java 2 Platform Enterprise Edition). La plataforma de Java 2 integra el RMI con IIOP, facilitando la ínter operación entre sistemas Java y CORBA.
- Aparece el Protocolo de Acceso de Objeto Simple (SOAP - Simple Object Access Protocol). Se inicia la era de los servicios Web.

1998

- Sun Microsystems presenta J2EE (Java 2 Platform Enterprise Edition). La plataforma de Java 2 integra el RMI con IIOP, facilitando la ínter operación entre sistemas Java y CORBA.
- Aparece el Protocolo de Acceso de Objeto Simple (SOAP - Simple Object Access Protocol). Se inicia la era de los servicios Web.
- Aparece XML and RELATED, desarrollado y certificado por IBM, El comercio electrónico ha proliferado en el Web, en los últimos años y XML acelera este crecimiento más y más. Algunas de las tecnologías relacionadas que hacen uso de XML son: XML Sherna, XSLT y Xpath.

1999 al 2002

- Aparece XML-RPC, el cual es una implementación del protocolo de procedimientos remotos RPC usando XML y HTTP como agente de transporte, el protocolo permite la comunicación sobre diferentes sistemas operativos y diferentes lenguajes.
- WDDX es un acrónimo para Web Distributed Data Exchange. Fue pensado como una solución para el intercambio entre aplicaciones web, de estructuras de datos complejos, como arreglos y recordsets de base de datos. Usando un lenguaje común e independiente de la plataforma de representación de datos y basado enteramente en XML y un conjunto de módulos que entienden este lenguaje y cuya implementación es soportada por una buena variedad de lenguajes (Asp, .NET,java, javascript, Vb, etc.).
- SOAP 0.9 (Simple Object Access Protocol), Creado por DevelopMentor, Microsoft, y Userland Software. Protocolo para mensajería y comunicación entre dos procesos.

2001 al 2002

- WSDL (Web Services Description Language), es una gramática XML, orientada a describir en forma estructurada, la funcionalidad de un Web Service y la forma en que esa funcionalidad se hace disponible. Describe

un servicio, como una colección de “comunication endpoints” (puertos) capaces de intercambiar mensajes.

2001

- El Consorcio de World Wide Web Protocolo XML (XP) es un grupo que estandarizo el SOAP, que se llama XP.

2001 hasta nuestros días

- WS_FTP Server es un servidor de protocolo de transferencia de archivos (FTP), extraordinariamente potente.
- WSE 1.0 El marco de trabajo .NET se ha liberado.

El futuro de los servicios Web, y servicios para arquitecturas web son los que sobresalen es esta etapa hasta nuestros días, dentro de estos se encuentran:

- XML 1.0 (segunda edición) basado en la codificación de documentos, XSD, DTD (Document Type Definition), WSDL 1.1 descriptor de servidos Web, SOAP basado en la codificación de mensajes, UDDI 2.0 (Universal Description, Discovery and Integration), HTTP (Hyper Text Transport Protocol), XSLT – XML Transformations, PSVI – Post Schema Validation Infoset, DIME (Direct Internet Message Encapsulation), MIME (Multipurpose Internet Mail Extension).

I.1.1.1 Comunicación entre procesos

La diferencia más importante entre un sistema distribuido y un sistema con un procesador es la comunicación entre procesos. En un sistema con un procesador, la mayor parte de la comunicación entre procesos supone de manera implícita la existencia de la memoria compartida. Un ejemplo típico es el problema de los productores y los consumidores, donde un proceso escribe en un buffer compartido y otro proceso lee de él. Incluso en la forma más básica de sincronización, en el uso de semáforos, hay que compartir una palabra (la propia variable del semáforo). En un sistema distribuido, no existe tal memoria compartida, por lo que toda la naturaleza de la comunicación entre procesos debe replantearse a partir de cero [1].

La comunicación entre procesos permite la interacción entre aplicaciones y servicios del sistema, existen dos modelos de comunicación entre procesos, los cuales son la memoria compartida (Sólo un/multiprocesador no distribuido) y el paso de mensajes. El nivel de abstracción en la comunicación (Figura I.1) de procesos es dado gracias al paso de mensajes puro (Cliente-Servidor), las llamadas a procedimientos remotos y a los modelos de objetos futuros.
Los diferentes mecanismos de comunicación entre procesos se caracterizan por diferentes factores tales como: el rendimiento, la escalabilidad, la fiabilidad, la seguridad, la movilidad, la calidad de Servicio QoS y la comunicación en grupo.

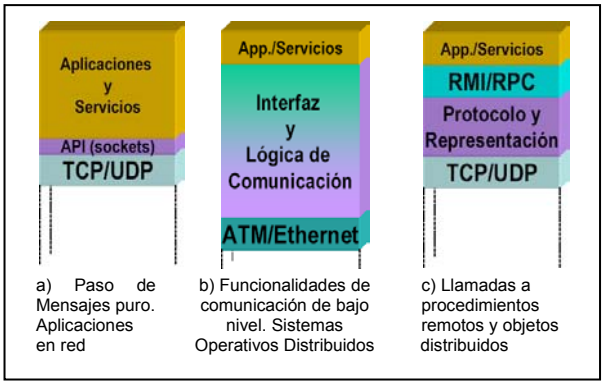


Figura I.1 Niveles de comunicación

En la comunicación entre procesos es de gran importancia hablar acerca de las primitivas de comunicación básicas ofrecidas por el microkernel (Figura I.2); las cuales son cada una de las funciones de comunicación de una tecnología determinada. Estas primitivas de comunicación básicamente son el envío ***send(mandar un mensaje a un destinatario)*** y recepción: ***receive(esperar un mensaje de una fuente).*** Aunque existen otras primitivas de comunicación como lo son conexión y desconexión la primera utiliza ***connect(destino)*** y la segunda **close()**.

Cada una de estas primitivas tiene las siguientes características:

- Bloqueantes (conocidas como primitivas síncronas)

En un envío bloqueante, el emisor especifica un destino y un mensaje a mandar a éste; mientras se manda el mensaje, el proceso emisor se bloquea; el emisor no continua hasta recibir un asentimiento del envío. Mientras que por el lado de la recepción bloqueante, el receptor especifica un emisor del cual recibir, el receptor se bloquea hasta que no recibe el mensaje y lo almacena en un buffer interno; en algunos sistemas puede especificarse un tiempo máximo de espera

- No bloqueantes (conocidas como primitivas asíncronas)

En un envío no bloqueante el hilo emisor especifica un destino y un mensaje a mandar a éste, ordena el envío del mensaje; y vuelve inmediatamente, antes de mandar el mensaje. Por el otro lado, en la recepción no bloqueante el hilo receptor especifica un origen del mensaje, indica una ubicación donde guardar el mensaje; en una recepción condicional el hilo receptor especifica una condición de recepción, retorna inmediatamente, devolviendo el mensaje recibido o un error.

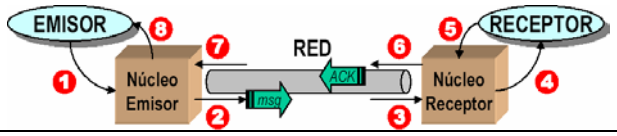
- Síncronas vs Asíncronas

Esta característica no afecta tanto a la primitiva como a la transmisión en sí. En la comunicación síncrona el envío y recepción se realiza de forma simultánea, además la comunicación asíncrona usa un buffer de almacenamiento, esto implica ciertas condiciones de bloqueo en envío y recepción mientras que en la comunicación asíncrona el envío no requiere que el receptor esté esperando.

- Fiables vs no-fiables

El envío fiable de los datos garantiza que un mensaje enviado ha sido recibido por el receptor, bajo este esquema, el usuario debe contemplar la posible pérdida de mensajes, con primitivas fiables, se pueden tener dos esquemas, “Asentimientos

individuales" request-ACK-reply-ACK, y "Respuesta utilizada como asentimiento" request-reply-ACK. El más utilizado es un compromiso entre los dos esquemas, cuando llega una petición al kernel del receptor, se arranca un temporizador. Si el receptor contesta rápido, la respuesta es tomada como asentimiento, si se vence el temporizador, se envía un asentimiento separado.
La fiabilidad puede garantizar el protocolo de comunicación (TCP si y UDP no) y así también como los elementos emisor y receptor.



Envío no bloqueante: [1:8]	El emisor continua al pasar el mensaje al núcleo
Envío bloqueante: [1:2:7:8]	El emisor espera a que el núcleo transmita por red el mensaje.
Envío bloqueante fiable: [1:2:3:6:7:8]	El emisor espera a que el núcleo receptor recoja el mensaje.
Envío bloqueante explicito: [1:2:3:4:5:6:7:8]	Identifica al anterior, pero es la aplicación receptora la que confirma la recepción.
Petición-Respuesta: [1:2:3:4:<servicio>:5:6:7:8]	El emisor espera a que el receptor procese la operación para reanudar la ejecución.

Figura I.2 Primitivas de comunicación

El direccionamiento de procesos es muy importante dentro de la comunicaron entre procesos, ya que sin éste no se podría relacionar un proceso a una máquina (machine.process), aunque esta relación no es transparente; el usuario sabe dónde esta el proceso que busca; de igual forma no seria posible localizar procesos con broadcasting, esto debido a que los procesos escogen una dirección de entre un gran rango. Se mandan paquetes de localización que el kernel escucha; es importante aclarar que esto genera mucha carga; la localización de procesos vía un servidor de nombres, es dada por el direccionamiento de procesos, el servidor puede fallar. Si se replica, pueden producirse problemas de consistencia. Una solución alternativa al direccionamiento consiste en utilizar hardware especial que mantenga direcciones de procesos y examinan el tráfico de red.
Aunque muchas tareas pueden realizarse en procesos aislados, la gran mayoría requieren la intervención de más de un proceso, es por ello la importancia de la comunicación entre los mismos. Para que dichos procesos cooperantes lleven a buen término una tarea común es necesario algún tipo de comunicación entre ellos. Los mecanismos de comunicación entre procesos (IPC's) habilitan mecanismos para que los procesos puedan intercambiar datos y sincronizarse. A la hora de comunicar dos procesos, se consideran dos situaciones diferentes: que los procesos se estén ejecutando en una misma máquina, o bien que los procesos se ejecuten en máquinas diferentes.
En el caso de comunicación local entre procesos, aunque existen diferentes mecanismos que se engloban bajo esta denominación común, cada uno de ellos tiene su propósito específico.

En el caso de ProActive (ver Capítulo II) los procesos son hilos que se ejecutan de forma concurrente, estos son creados dentro de un Objeto Activo, una vez que un objeto activo es creado su actividad (el hecho de que este tenga su propio hilo) y su localización (local o remota) son perfectamente transparentes. El modelo de comunicación en el que se basa ProActive es el paso de mensajes; en ProActive no existe diferencia alguna entre procesos locales y procesos remotos

Cuando una aplicación levanta una Máquina Virtual Java (MVJ), y esta a su vez crea un Objeto Activo, el hilo de esté se relaciona con la Máquina Virtual Java que lo crea; de esta forma un proceso se relaciona a una máquina, sin necesidad del direccionamiento de procesos en dónde el usuario sabe dónde esta el proceso que busca (esta relación no es transparente). En ProActive es posible localizar procesos con broadcasting de manera transparente sin la necesidad de que los procesos escojan una dirección de entre un gran rango, y sin la necesidad de mandar paquetes de localización que el kernel escuche. En ProActive no se genera mucha carga, ya que la localización de procesos vía un servidor de nombres no es dada por el direccionamiento de procesos, por lo cual el servidor no falla, no produce problemas de consistencia, y no es necesario examinar el trafico de la red.

I.1.1.2 Comunicación en los sistemas distribuidos

I.1.1.2.1 Modelo Cliente-Servidor

Bajo esta filosofía, en absoluto exclusiva de los sistemas operativos, se esconde la idea de liberar al núcleo del sistema operativo de una gran parte de las funciones asociadas a éste. Dado que la mayor parte de esas funciones se definen como servicios ofrecidos por el sistema, aparecen los conceptos de proceso servidor, el cual lo podemos definir como un grupo de procesos en cooperación, que ofrecen un servicio. Y los procesos cliente, que utilizan dichos servicios. Las máquinas de los clientes y servidores ejecutan por lo general el mismo microkernel (micronúcleo) y ambos se ejecutan como procesos de usuario. Una máquina puede ejecutar un proceso o varios clientes, varios servidores o combinaciones de ambos.
Para evitar un gasto excesivo en los protocolos orientados hacia la conexión como OSI o TCP/IP, lo usual es que el modelo cliente-servidor se base en un protocolo solicitud respuesta sencillo y sin conexión, no es complejo y orientado a la conexión como OSI o TCP/IP. No se tiene que establecer una conexión sino hasta que ésta se utilice. El cliente envía un mensaje de solicitud al servidor para pedir cierto servicio. El servidor hace el trabajo y regresa los datos solicitados o un código de error para indicar la razón por la cual un trabajo no se llevó a cabo, como se muestra en la Figura I.3. La principal ventaja como se puede observar, es su sencillez. El cliente envía un mensaje y obtiene una respuesta. No es necesario establecer una conexión sino hasta que ésta se utilice. El mensaje de respuesta

sirve como reconocimiento de la solicitud. Después de la sencillez se puede observar otra ventaja: la eficiencia: La pila del protocolo es más corta y por lo tanto más eficiente. Si todas las máquinas fuesen idénticas sólo se necesitarían tres niveles de protocolos, como se muestra en la Figura I.3. Las capas físicas y de enlace de datos se encargan de llevar los paquetes del cliente al servidor y viceversa. No se necesita un ruteo y tampoco se establecen conexiones, por lo que no se utilizan las capas 3 y 4. La capa 5 es el protocolo solicitud/respuesta. Define el conjunto de solicitudes válidas y el conjunto de respuestas válidas a esas solicitudes. No existe administración de la sesión, puesto que éstas no existen. Tampoco se utilizan las capas superiores. Debido a esta estructura tan sencilla, se pueden reducir los servicios de comunicación que presta el (micro) núcleo [1].

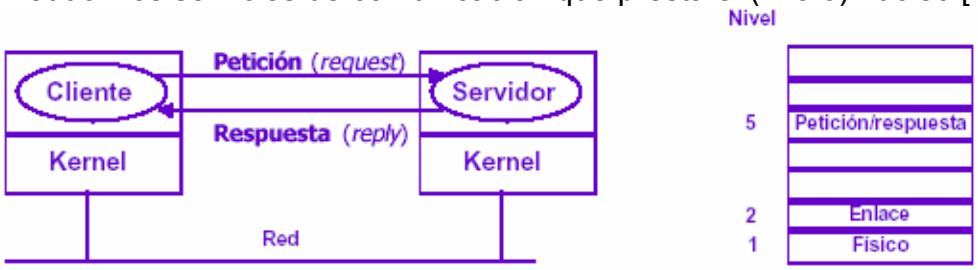


Figura I.3 Modelo cliente-servidor

Dentro del modelo cliente-servidor los tipos de servidores más comunes son:

- Servidores de archivos
- Servidores de bases de datos
- Servidores de transacciones
- Servidores de Groupware
- Servidores de objetos / componentes
- Servidores de aplicaciones web

El primer problema con el que se debe lidiar al querer comunicar dos procesos es la sincronización. Suponer por ejemplo que existen dos programas que desean comunicarse en forma síncrona a través de un pipe. La mejor forma de lograr la sincronización es que uno de los dos programas espere hasta que llegue algún indicio de que existe otro programa que se quiere comunicar con él. En el modelo cliente-servidor, el servidor siempre está esperando una conexión (inicia la conexión en forma pasiva) y el cliente inicia la conexión en forma activa.
El modelo cliente-servidor es utilizado en este trabajo de tesis para la transferencia del flujo de objetos, en donde un servidor es capaz de atender simultáneamente a varios clientes, que solicitan el servicio de transferencia.

I.1.1.2.2 Paso de Mensajes

En paso de mensajes un proceso manda a otro proceso un mensaje que va a contener la información que debe conocer. Por ello, no existe problema de control de acceso a la información, ya que, si a un proceso le llega un mensaje, este mensaje ya es correcto. Esto supone una primera ventaja muy importante a favor

de que se escoja una biblioteca de paso de mensajes. Además, la sincronización puede ir en el propio envío y recepción del mensaje haciéndolos síncronos, lo que facilita la programación.
Los modelos de comunicación basados en cliente-servidor con paso de mensajes responden al esquema de la Figura I.5, en donde cada pareja ***send-receive*** transmite un mensaje entre cliente y servidor. Por lo general de forma asíncrona; habitualmente ***send*** es no bloqueante mientras que ***receive*** es bloqueante, aunque puede hacerse no bloqueante.
Los mensajes intercambiados en el paso de mensajes, pueden ser mensajes de texto (cadenas de caracteres) o mensajes con formato (binarios). La estructura de un mensaje de texto es por ejemplo "http:www.cs.buap.mx" y la forma de enviar un mensaje es de la manera siguiente ***send (GET "www.cs.buap.mx")***, el emisor debe hacer un análisis de la cadena de caracteres transmitida.
Por otro lado la estructura de un mensaje binario es como la que a continuación se muestra:

```
Struct mensaje_st
{
  unsigned int msg_tipo;
  unsigned int msg_seq_id;
  unsigned char msg_data[1024];
};
```

Y el envío del mensaje es como podemos observar a continuación.

```
Struct  mensaje_st confirm;
Confirm.msg_tipo=MSG_ACK;
Confirm.msg seq id=129;
Send(confirm);
```

Para la transmisión de formatos binarios (Figura I.4) tanto emisor y receptor deben coincidir en la interpretación de los bytes transmitidos, deben coincidir con el tamaño de los datos numéricos, ordenación de bytes y formatos de texto ASCII vs EBCDIC.

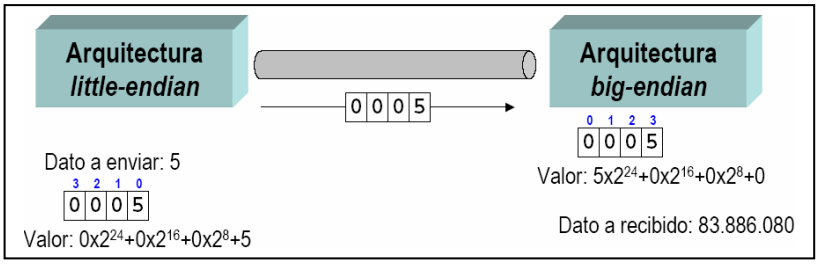


Figura I.4 Transmisión de formatos binarios

Es importante recordar que las aplicaciones definen el protocolo de comunicación, ya sea petición-respuesta, recepción explicita, sin/con confirmación.

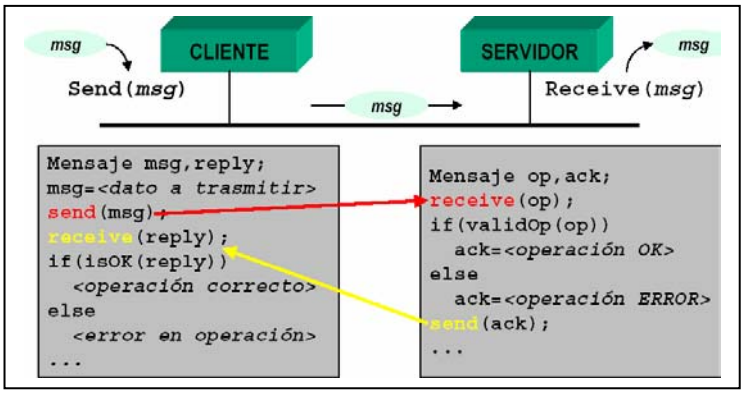


Figura I.5 Paso de mensajes

En el caso de Java él paso de mensajes esta dado por la librería ***Synchronization***, y existe una interfaz básica que es ***MessagePassing***. Las clases en java para el paso de mensajes asíncrono son ***AsyncMessagePassing*** o ***AsyncConditionalMessagePassing*** y las clases para el paso de mensajes sincrono es dado por SyncMessagePassing o ***SyncConditionalMessagePassing.*** ***MessagePassingSendOnly*** es la clase utilizada para restringir el carácter de sólo envío, y ***MessagePassingReceiveOnly*** para restringir el carácter de sólo recepción.
Las soluciones distribuidas habitualmente hacen uso del modelo de paso de mensajes, que se han popularizado para estas redes. Las soluciones paralelas hacen uso de los dos modelos paso de mensajes y memoria compartida, habitualmente en función de la arquitectura de la máquina y del lenguaje escogido; ya que en determinadas máquinas paralelas cuentan con memoria compartida.

I.1.1.2.3 Java

La programación de proporciona un mecanismo de muy bajo nivel para la comunicación e intercambio de datos entre dos computadoras, uno considerado como cliente, que es el que inicia la conexión con el otro, servidor, que está a la espera de conexiones de clientes.
El protocolo de comunicación entre ambos, determinará lo que suceda tras el establecimiento de la conexión. Para que las dos máquinas puedan entenderse, ambas deben implementar un protocolo conocido por las dos. En la programación de , la comunicación es full-duplex, (ambos sentidos a la vez), entre cliente y servidor; siendo responsabilidad del sistema el llevar los datos de una máquina a otra, dejando al programador el proporcionar significado a esos datos. Parte de la información que fluye entre las dos máquinas es, pues, para implementar el protocolo, y el resto son los propios datos que se quieren transferir.
Es muy sencilla la utilización de para establecer la comunicación entre cliente y servidor; en realidad; no es más complicada que lo que pueda serlo el escribir datos en un archivo. Enviar y recoger los datos que se intercambian es la parte fácil del asunto; porque más allá de esto ya se encuentra el protocolo de

comunicación que debe ser entendido por el cliente y servidor; que en caso de ser necesario implementarlo, se convierte en algo verdaderamente complicado [2].

El paquete java.net de Java contiene clases para establecer comunicación en Internet que facilita el desarrollo de aplicaciones cliente-servidor.

El paquete java.net incluye las siguientes clases:

1. Clase InetAddress (direcciones IP)

La clase InetAddress implementa direcciones IP, ofreciendo métodos para obtener información sobre dicha dirección.
Por ejemplo, InetAddress contiene métodos para obtener su representación de cuatro octetos. InetAddress posee métodos estáticos que permiten obtener información de una computadora sin necesidad de construir objetos InetAddres.

2. Clases URL

Las clases URL se usan para representar e interactuar con localizadores uniformes de recursos (Uniform Resorce Locators), los cuales son referencias a la información puesta en la web.

Las clases URL incluidas en el paquete java.net son las siguientes:

URL	construye objetos constantes definidos en el momento de su creación para representar direcciones estáticas URL.
URLConnection	define una clase abstracta cuyas subclases facilitan la conexión con un URL.
URLStreamHandler	define una clase abstracta cuyas subclases poseen los mecanismos requeridos para abrir flujos (streams) provenientes de recursos referenciados por URLS.
URLEncoder	permite convertir una cadena de texto a un formato apropiado para comunicación mediante URLs.

3. Socket (stream)

Los de flujo (Figura 1.6) definen flujos de comunicación en dos direcciones, fiables y con conexión. Por ejemplo Si se envian dos ítems a través del *socket* en el orden "1, 2" llegan al otro extremo en el orden "1, 2", y llegan sin errores. Cualquier error que se encuentre es producto de extravio de paquetes. Telnet usa de flujo, todos los caracteres que se teclean llegan en el mismo orden en que se teclean, También los navegadores, que usan el protocolo HTTP , usan de flujo para obtener las páginas. De hecho, si se hace telnet a un sitio de la *web* sobre el puerto 80, y se escribes " GET / ", se recibe como respuesta el código HTML. Los de flujo usan el protocolo "TCP". TCP asegura que tu información llegue secuencialmente y sin errores. TCP forma parte del acrónimo " TCP/IP ", donde "IP" significa "Protocolo de Internet". IP se encarga básicamente del encaminamiento a través de Internet y en general no es responsable de la integridad de los datos [Internet-9].

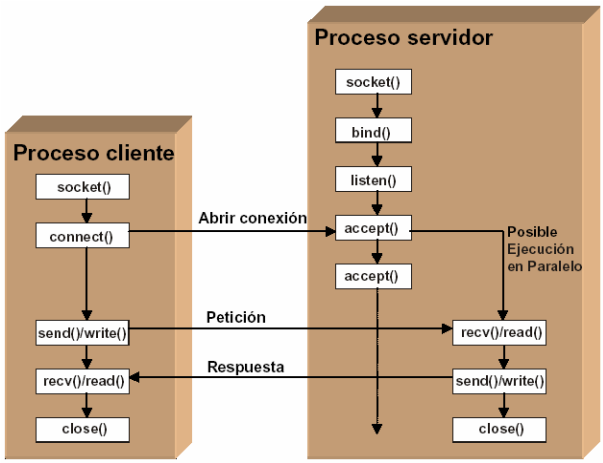


Figura 1.6 Escenario de uso de socket de flujo

4. DatagramSocket (datagram)

Si se envía un datagrama, puede que llegue. Puede que llegue fuera de secuencia. Si llega, los datos que contiene el paquete no tendrán errores.

Los de datagramas también usan IP para el encaminamiento, pero no usan TCP; usan el "Protocolo de Datagramas de Usuario" o "UDP"

Los datagram (SOCK_DGRAM) (Figura 1.7) son sin conexión, porque no se tiene que mantener una conexión abierta como con los de flujo. Simplemente se monta un paquete, se mete una cabecera IP con la información de destino y se envía. No se necesita conexión. Generalmente se usan para transferencias de información por paquetes. Aplicaciones que usan este tipo de son, por ejemplo, **tftp** y **bootp**.

El protocolo tftp establece que, para cada paquete enviado, el receptor tiene que devolver un paquete que diga, "¡Lo tengo!" (un paquete "ACK"). Si el emisor del paquete original no obtiene ninguna respuesta en, mas o menos cinco segundos, retransmite el paquete hasta que finalmente recibe un ACK . Este procedimiento de confirmaciones es muy importante si se implementan aplicaciones basadas en SOCK_DGRAM. Este tipo de no son fiables [Internet-9].

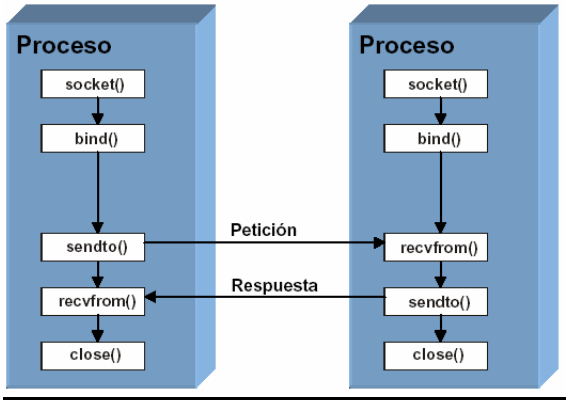


Figura 1.7 Escenario de uso de de datagramas

5. Content-Handler

Las clases socket son quizás las clases más importantes incluidas en el paquete java.net por que provee un modelo completo para realizar comunicaciones de red usando diferentes métodos.
Un socket puede compararse con un archivo remoto cuya dirección lógica en Internet esta determinada por un URL.
El socket representa una conexión entre dos computadoras (punto a punto). La computadora servidor, la cual solo se detiene a escuchar solicitudes de conexión, y la computadora cliente que solicita abrir una conexión, y para poder establecer una conexión, todo lo que se necesita es la dirección y un número de puerto asociado al servicio.

Varios programas servidores (distintos) pueden residir en la misma computadora, misma dirección IP y para distinguirlos se necesita un numero de puerto único asociado a cada uno de ellos.
Un puerto (Tabla I.1) representa una dirección (dentro del computador servidor) donde se puede localizar a un programa servidor.
Cuando un socket se crea siempre se debe especificar un número de puerto.

Los sockets tienen dos modos principales de operación:

- Orientados a conexión (TCP), los cuales ofrecen comunicación confiable.
- No orientados a conexión (UDP), en donde no se garantiza la confiabilidad de la comunicación.

En la comunicación orientada a conexión, los sockets se llaman de flujo de datos (stream), mientras que en la comunicación no orientada a conexión, se llaman sockets de datagramas (datagram).

Un datagrama es un mensaje, que además del contenido de información de la aplicación, contiene toda la información para su entrega.

echo	1	Regresa lo que se le mande
daytime	13	Regresa la hora del día
ftp	21	Transfiere archivos
telnet	23	Establece una sesión remota
Smtp	25	Maneja correo electrónico simple
Finger	19	Encuentra información sobre usuarios o computadoras
http	80	Obtener paginas web

Tabla I.1 Números de puertos de varios servidores

Comunicación con TCP (Protocolo de Control de Transmisión)

Se utilizan dos clases de socket (uno para el cliente y otro para el servidor).

Para el cliente:

- Socket(InetAddress dir, int puerto)
  Crea un socket de flujo para el cliente conectado con la dirección y puerto indicados. Existen otros constructores con diferentes argumentos

Para el servidor:

- ServerSocket(int puerto)
  Crea un socket stream para el servidor. Existen otros constructores con diferentes argumentos.
- Socket accept()
  Prepara la conexión y se bloquea a espera de conexiones. Equivale a listen y accept de BSD Socket. Devuelve un socket.

Comportamiento general de los clientes

Considérese primero la comunicación orientada a conexión, la cual define comportamientos asimétricos para clientes y servidores.
En este caso, el comportamiento de los clientes es bastante regular realizando, en general las siguientes acciones:

1. El cliente crea un socket e intenta realizar una conexión con el servidor.
2. El cliente obtiene los flujos de datos para lectura y escritura de modo que pueda conducir una comunicación con el servidor.
3. El cliente se desconecta del socket cerrando todos los flujos de datos así como el socket mismo.
4. Si la solicitud de comunicación tiene éxito, el servidor recibirá el puerto creado por el cliente

Comportamiento general de los servidores

Un servidor que usa sockets con flujos sigue esquemas bastante bien definidos:

1. El servidor crea una instancia de ***ServerSocket*** en el puerto indicado.
2. Llama al método ***accept()*** para escuchar peticiones de nuevas conexiones.
3. Crea flujos de entrada y salida para el socket.
4. Conduce comunicaciones basadas en el protocolo aceptado.
5. Cierra los flujos de datos del cliente y el socket.
6. Continua así indefinidamente entre los pasos del 2 al 5 o cierra el socket del servidor.

Los servidores no crean conexiones constantemente sino por demanda.
Un servidor puede indicar el número máximo de clientes que pueden esperar antes de establecer una conexión ***(listen stack)***; las peticiones, hasta 50 si no se indica otra cosa, podrán recibirse al mismo tiempo aunque solo una se procesa a la vez. No se intercambia ninguna comunicación e información a nivel de aplicación por medio del socket del servidor.
El servidor crea un socket nuevo en el método ***accept()*** de modo que el socket del servidor permanece abierto para recibir solicitudes de comunicación.
La operación ***accept()*** se dice bloqueante; si no hay clientes esperando servicio, el servidor espera que algún cliente llegue.

El servidor:

1. Abre los flujos de datos de entrada y salida
2. Realiza con ellos operaciones de lectura y escritura de acuerdo al protocolo de comunicación definido para la aplicación
3. Cierra los flujos de entrada y salida

Cuando la demanda por comunicación con el servidor es enorme, conviene crear un representante del servidor (usando threads) el cual realmente realiza el servicio.
A diferencia de los métodos orientados a conexión, las versiones de socket que usan datagramas son bastante simétricos, es decir se comportan similarmente.
La clase DtagramSocket se usa por ambos clientes y servidores. El patrón de comportamiento de clientes y servidores que usan datagramas es bastante simétrico, a diferencia de las comunicaciones que usan sockets basados en flujos.
La información a transmitir por los UPD se asocia a un objeto de la clase DatagramPacket. Estos objetos se construyen con un array de bytes a transmitir.
DaragramPacket(byte[] datos, int tam): Crea un datagrama para el vector de bytes a transmitir. Adicionalmente se le puede pasar una dirección IP (inetAddress) y un puerto para indicar el destino de transmisión del paquete cuando se envié.

La comunicación vía UDP se realiza por medio de objetos de la clase DatagramSocket

- DatagramSocket(int puerto, InetAddress dir)
  Crea un socket UDP con un bind a la dirección y puerto indicados. Dirección y puerto son opcionales (se elige uno libre).
- Void receive(DatagramPacket paquete)
  Se bloquea hasta la recepción del datagrama.

Otros métodos:

- Void close(): Cierra el socket
- Void setSoTimeout(int timeout):Define el tiempo de bloqueo de un receive()

<u>Comportamiento de los clientes y servidores de sockets de datagramas</u>
Los pasos que realizan son los siguientes:

1. Crea un socket para datagramas sobre cualquier puerto disponible
2. Crea la dirección a quien será dirigido el datagrama
3. Envía el datagrama de acuerdo al protocolo del servidor
4. Espera y recibe datos del servidor
5. Continua así indefinidamente entre (pasos 3 y 4) o termina cerrando el socket de datagramas

El servidor determina el número de puerto para recibir comunicaciones, también recibe datagramas conteniendo el número de puerto del cliente, además envía datagramas usando el número de puerto asociado para el cliente. Ambos cliente y servidor usan la clase DatagramPacket para enviar y recibir información.

El comportamiento del servidor es el siguiente:

1. Crea un socket de datagramas en un puerto especifico
2. Usa receive() para esperar por los datagramas entrantes
3. Responde a los datagramas recibidos de acuerdo al protocolo de comunicación
4. Regresa al paso 2 o va al paso 5
5. Cierra el socket

El comportamiento del cliente es el siguiente:

1. Crea un socket para datagramas sobre cualquier puerto disponible
2. Crea la dirección a quien será dirigido el datagrama (obtenida de un datagrama previamente recibido
3. Envía el datagrama de acuerdo al protocolo del servidor
4. Espera y recibe datos del servidor
5. Continua así indefinidamente entre (pasos 3 y 4) o termina cerrando el socket de datagramas

I.1.1.3 Llamadas a Procedimientos Remotos (RPC)

La técnica de RPC ó Remote Procedure Call, fue propuesta por Birrel y Nelson en 1985; Sun RPC es la base para varios servicios actuales (NFS o NIS). Llegaron a su culminación en 1990 con DCE (Distributed Computing Environment) de OSF han evolucionado hacia la orientación a objetos e invocación de métodos remotos (CORBA, RMI). El objetivo de los RPC es acercar la semántica de las llamadas a procedimientos convencional a un entorno distribuido (transparencia).

El funcionamiento general de RPC se muestra de forma descriptiva en la Figura I.8 En donde el cliente, es el proceso que realiza una llamada a una función; dicha llamada empaqueta los argumentos en un mensaje, se los envía a otro proceso y queda a la espera del resultado, el servidor recibe el mensaje consistente en varios argumentos, los cuales son usados para llamar una función en el servidor; el resultado de la función se empaqueta en un mensaje que se retransmite de regreso al cliente, como se muestra en la Figura I.9.

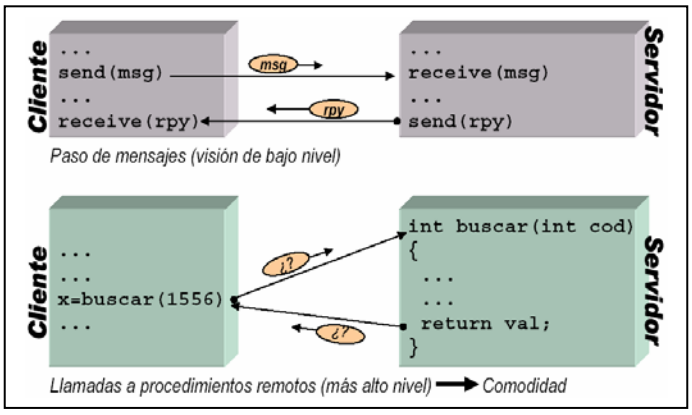


Figura I.8 Llamadas a procedimientos remotos

Los elementos necesarios para la programación de llamadas a procedimientos remotos son:

- Código cliente
- Código del servidor

- Formato de representación
- Definición del interfaz
- Localización del servidor
- Semántica de fallo

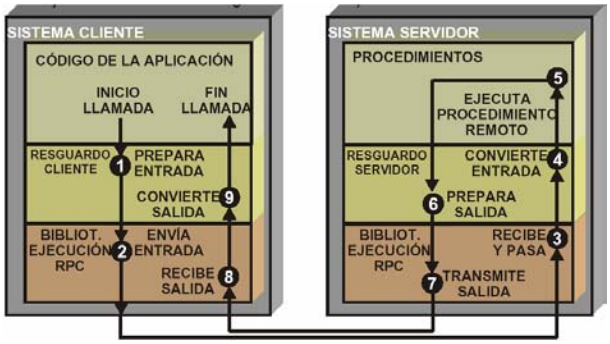


Figura I.9 Código del cliente y código del servidor

Las funciones de abstracción de una llamada RPC a intercambio de mensajes se denominan resguardos (stubs). Los Stubs se generan automáticamente por el software de RPC en base a la interfaz del servicio, estos son independientes de la implementación que se haga del cliente y del servidor; sólo depende de la interfaz. Las tareas que realizan los Stubs son localizar al servidor, empaquetar los parámetros y construir los mensajes, así como enviar el mensaje al servidor y esperar la recepción del mismo y una vez así devolver los resultados. Los Stubs se basan en una librería de funciones RPC para las tareas más habituales.
Como ya se mencionó una de las funciones de los resguardos es empaquetar los parámetros en un mensaje: aplanamiento (marshalling). Existen algunos problemas en el momento de la representación de los datos, como por ejemplo, el servidor y el cliente pueden ejecutarse en máquinas con arquitecturas distintas, problemas con los punteros, una direccion sólo tiene sentido en un espacio de direcciones de la máquina emisora, etc.

Definición de interfaces IDL (Interface Definition Language), es un lenguaje de representación de interfaces, en IDL aun existen muchas variantes, como el hecho de que está integrado con un lenguaje de programación (Cedar, Argus), es específico para describir las interfaces (RPC de Sun y RPC de DCE), define procedimientos y argumentos (no la implementación), y es usado habitualmente para generar de forma automática los resguardos.

Para la comunicación de bajo nivel entre cliente y servidor por medio de paso de mensajes, se requiere localizar la dirección del servidor, tanto IP como número de puerto en el caso de y verificar si dicho servidor esta funcionando. Estas tareas las realiza el resguardo cliente. En el caso de servicios cuya localización no es estándar se recurre al enlace dinámico (dynamic binding); este permite localizar objetos con nombre en sistemas distribuidos y servidores que ejecutan las RPC. Existen dos tipos de enlaces dinámicos, el enlace no persistente y el enlace

persistente, en el primero la conexión entre el cliente y el servidor se establece en cada llamada RPC, y en el segundo la conexión se mantiene después de la primera RPC.
Es de gran importancia mencionar acerca del enlazador dinámico (binder), el cual es el servicio que mantiene una tabla de traducciones entre nombres de servicio y direcciones, este incluye funciones para registrar un nombre de servicio, eliminar un nombre de servicio y buscar la dirección correspondiente a un nombre de servicio. Para localizar al enlazador dinámico se ejecuta en una dirección fija de un computador fijo, el sistema operativo se encarga de indicar su dirección y difundir un mensaje (broadcast) cuando los procesos comienzan su ejecución.

I.1.2 Actualidad

Con el transcurso del tiempo, se han desarrollado diferentes tipos de aplicaciones y sistemas, ya para nuestros días las necesidades de un cómputo distribuido, han ido creciendo cada vez más, necesidades como la distribución de datos de aplicación, la distribución de aplicaciones y la distribución de los usuarios de las aplicaciones entre otras, han influido en ello.
El problema del cómputo distribuido, no es un tema que haya emergido el día de hoy; existen consorcios que han hecho grandes contribuciones para proveer una solución a esta problemática, tal es el caso de la OMG (Object Management Group), proponiendo la arquitectura CORBA (Common Object Request Broker Architecture), esta será el futuro en el cómputo distribuido y el reemplazo del mecanismo RPC y esta tecnología ha tomado en cuenta a los agentes móviles para su computo móvil distribuido. Los mecanismos para la interacción entre los mismos agentes móviles y los ORB's de CORBA están siendo desarrollados y la definición de la interfaz esta definida en la especificación MAF (Móvil Agent Facility).

Actualmente el paradigma de los agentes móviles que aun se encuentra en desarrollo, tratará de solucionar los problemas del futuro tales como el mapping, es decir, asociar detalles de la información en el Internet; Por otro lado Cory Quammen sigue con una introducción al arte de programar los computadoraes paralelos que están configurados para compartir y también para distribuir memoria, y entre otros Quammen saca a la luz, las diferencias entre estas dos configuraciones y discute el *OpenMP* y el *MPI*, que son los dos estándares para programar estos diversos tipos de multiprocesadores.
RMI y algunas alternativas como CORBA y COM son mecanismos para invocar y ejecutar procedimientos remotos en computadoras y servidores distribuidos.
La gran mayoría de los sistemas empresariales hoy en día requieren de esta funcionalidad, esto se debe tanto a distancias geográficas como a requerimientos de cómputo, ya que seria iluso pensar que las necesidades de cómputo de toda una empresa fueran satisfechas por una sola computadora y/o servidor.

Diseñar Procedimientos y Aplicaciones Remotas, implica revisar diversos detalles con los que comúnmente no se trabaja al diseñar un programa para ejecutarse en una sola computadora y/o servidor.

Estos son solo algunos de los detalles con los que se debe trabajar al invocar procedimientos remotos:

1) **Marshalling y Unmarshalling**

Si se ejecuta un programa en una sola computadora se tiene la seguridad que la representación de datos esta conforme a esa plataforma, sea SunSolaris, Linux, Windows, etc... Sin embargo, ¿Que ocurre si se intentan pasar parámetros a un procedimiento en un servidor AIX de una computadora utilizando Windows NT? Marshalling y Unmarshalling es el proceso por el cual debe pasar toda información para que ésta sea utilizable en ambientes heterogéneos.

2) **Inestabilidad de la Red**

Debido a que los procedimientos son invocados a través de una Red, ¿Que ocurriría si un ***Router*** en la Red o bien el servidor y/o computadora fallara al momento de invocarse el procedimiento? Este tipo de errores deben ser contemplados al momento de definir el procedimiento.

3) **Seguridad**

Deben existir diversos criterios de Seguridad para permitir la ejecución de estos procedimientos remotos ya que pueden estar sujetos a un ambiente hostil, por el hecho de encontrarse en Red.
La razón de ser de RMI así como cualquier otro mecanismo para invocar procedimientos remotos (CORBA), es precisamente insular al programador final de todos estos detalles que deben ser contemplados al diseñar un procedimiento y/o aplicación en Red.

La tendencia actual a la globalización exige de los sistemas y aplicaciones informáticos prestaciones que van más allá de lo alcanzable por cualquier computadora aislado, por muy potente que este sea. Ello hace que las aplicaciones distribuidas se vayan convirtiendo en el modelo generalizado.
Desde las más extendidas arquitecturas cliente-servidor hasta los sistemas en cluster que ofrecen la imagen de una única máquina, los sistemas distribuidos presentan una serie de ventajas frente a los centralizados, en cuanto a su potencial rendimiento, fiabilidad, escalabilidad y efectividad de coste. Sin embargo, su diseño y programación conllevan también dificultades específicas.
La creciente necesidad de comunicación a todos los niveles entre particulares y organizaciones ha extendido el uso de los sistemas de comunicación entre computadoraes, desde las redes de área local y las redes de área extensa hasta Internet. En la actualidad el uso de computadoras aparece generalmente asociado al de algún sistema de comunicación, y es posible disponer de grupos de es que presten conjuntamente un mismo servicio. En el futuro encontraremos cada vez más redes heterogéneas de computadoras, con aplicaciones que compartan

recursos geográficamente distribuidos, flujo de información y coordinación entre sus actividades.

I.1.2.1 Entornos de Objetos Distribuidos

La extensión de los mecanismos de RPC a una programación orientada a objetos dio lugar a los modelos de objetos distribuidos. Algunas de las ventajas de estos modelos son: los métodos remotos; los cuales están asociados a objetos remotos, el hecho de que es más natural para desarrollo orientado a objetos y admite modelos de programación orientados a eventos. Por el lado contrario algunos de los problemas de los modelos de objetos remotos son: el concepto de referencia a objeto, los objetos volátiles y objetos persistentes.

I.1.2.1.1 Objetos Distribuidos

Las características de los objetos distribuidos son:

- Uso de un Middleware, este es un nivel de abstracción para la comunicación de los objetos distribuidos que oculta la localización de objetos, protocolos de comunicación, hardware de computadora y sistemas operativos.
- Modelo de objetos distribuido, describe los aspectos del paradigma de objetos que es aceptado por la tecnología (herencia, interfaces, excepciones, polimorfismo, etc.)
- Recolección de basura (Garbage Collection), determina los objetos que no están siendo usados para liberar recursos.

Actualmente existen las siguientes tecnologías de desarrollo de sistemas distribuidos basado en objetos:

- ANSA (1989-1991) fue el primer proyecto que intentó desarrollar una tecnología para modelar sistemas distribuidos complejos con objetos.
- DCOM de Microsoft
- CORBA de OMG
- Tecnologías de Java de Sun Microsystems (RM, EJB Enterprise Java Beans,Jini)
- Diferentes entornos de trabajo propietarios.

I.1.2.1.2 Java RMI

Invocación a Métodos Remotos (RMI, Remote Method Invocation) se utiliza para crear aplicaciones de Java que pueden comunicarse con otras aplicaciones de Java en una red. Para ser más específicos, la RMI permite que un objeto que se ejecuta bajo el control de una Maquina Virtual Java pueda invocar métodos de un objeto que se encuentra en ejecución bajo el control de una Máquina Virtual Java diferente. Estas dos Máquinas Virtuales pueden estar ejecutándose como dos

procesos independientes en una misma computadora o, lo que es más interesante, estar lanzadas en computadoras distintas conectadas a través de una red TCP/IP. Es decir, que como Internet es en último término una red TCP/IP, lo anterior se puede traducir en que una máquina cliente en cualquier rincón del mundo es capaz de invocar métodos de un objeto que se encuentre corriendo sobre un servidor en cualquier otro rincón del mundo.
Puesto que los mecanismos y protocolos usados en la comunicación entre objetos están definidos y estandarizados en la RMI, éste es un mecanismo más sofisticado para comunicarse entre objetos de Java distribuidos, que una simple conexión de socket. Se puede comunicar con otro programa de Java, por medio de la RMI, sin tener que saber de antemano qué protocolo utilizar , o cómo utilizarlo.
Aunque el concepto de la RMI pudiera presentar visiones de objetos repartidos por todo el mundo comunicándose alegremente entre sí, en realidad se utiliza en situaciones más tradicionales cliente/servidor. Una sola aplicación de servidor recibe conexiones y solicitudes de un cierto número de clientes. La RMI sólo es el mecanismo de comunicación entre el cliente y el servidor [3].

En el cliente siempre debe haber una línea de código para recoger la referencia al objeto remoto. Una vez que el cliente consigue esa referencia, la invocación de métodos sobre el objeto remoto no difiere en absoluto de la llamada a cualquier método de un objeto local (sin tener en cuenta la velocidad, por supuesto). Por otro lado el código del servidor debe definir la clase e instanciar un objeto remoto de esa clase. Además de eso, solamente se necesitan unas cuantas líneas más de código para registrar el objeto y dejar accesibles los métodos a los clientes, para que éstos puedan invocarlos remotamente.
Tanto el cliente como el servidor deben definir , o tener acceso, a una interfaz común, en donde se declaren los métodos que pueden ser invocados remotamente, y también el controlador de seguridad que va a tener a su cargo el control de que tanto servidor como cliente tengan los niveles de seguridad adecuados a las acciones que quieren realizar.
Cuando se invocan métodos sobre objetos remotos, el cliente puede pasar objetos como parámetros y los métodos de los objetos remotos pueden devolver objetos. Esto es posible gracias a la capacidad de serialización de objetos de Java. Por otro lado, como el cliente y el servidor están escritos en Java, el único requerimiento en cuanto a las plataformas en que se ejecutan es que en ambas se ejecuten Máquinas Virtuales Java compatibles [2].

Las metas de la RMI fueron integrar en Java un modelo de objetos distribuidos, sin interrumpir el lenguaje o el modelo de objetos existentes, e interactuar con un objeto remoto con la misma facilidad con que se interactúa con un objeto local. La RMI incluye mecanismos más sofisticados para llamar métodos de objetos remotos con el fin de pasar objetos completos o partes de objetos ya sea por referencia o por valor, así como excepciones adicionales para manejar los errores de la red que pudieran ocurrir durante la operación remota.
La RMI tiene varias capas para lograr estas metas, y una sola llamada de método cruza muchas de estas capas para llegar a su destino (Figura 1.10).

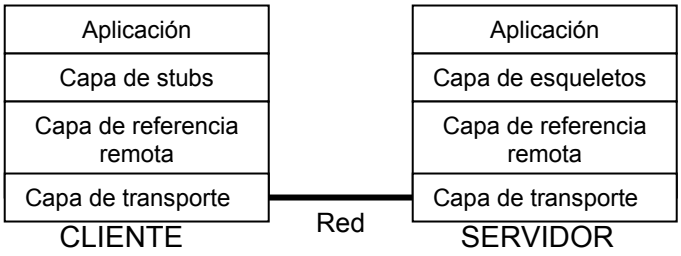


Figura I.10 Las capas de la RMI

Aunque en realidad solo cruza tres de estas capas: Primero se hace mención a las capas de “stubs” y de “esqueletos” en el cliente y el servidor, respectivamente. Estas capas se comportan como objetos sustitutos en cada lado, ocultando la “lejanía” de la llamada del método a las clases reales de implementación. Por su parte la capa de referencia remota, la cual maneja el empaquetamiento de una llamada de métodos, así como sus parámetros y valores devueltos para su transporte en la red y por ultimo la capa de transporte, que es la conexión real a la red de un sistema a otro.

Estas tres capas de la RMI permiten el control o la implementación de cada capa. Los stubs y los esqueletos permiten a las clases del cliente y el servidor comportarse como si los objetos con que tratan fueran locales, y utilizar exactamente las mismas características del lenguaje de Java para acceder a esos objetos. La capa de referencia remota separa el procesamiento de los objetos remotos en su propia capa, la cual se puede optimizar o volver a implementar, independientemente de las aplicaciones que dependen de ella. Por último, la capa de transporte de red se utiliza en forma independiente de las otras dos, de modo que se puedan usar distintos tipos de conexión de para la RMI (TCP, UDP o TCP con algún otro protocolo, como SSL).
Cuando una aplicación cliente hace una llamada a un método remoto, la llamada se pasa al stub y de ahí a la capa de referencia, la cual empaqueta los argumentos en caso necesario y los pasa al servidor por medio de la capa de trasporte, una vez en el servidor, la capa de referencia desempaca los argumentos y los pasa al stub y de ahí a la implementación del servidor. Luego, los valores devueltos para la llamada al método emprenden el viaje de regreso hacia el lado del cliente. Puesto que los objetos se deben convertir en algo que se pueda pasar por la red, la característica de empacar y pasar argumentos de método es uno de los aspectos más interesantes de la RMI. Esta conversación se denomina *serialización.* Mientras un objeto se pueda serializar, la RMI lo puede utilizar como un parámetro de método o un valor devuelto. Los objetos susceptibles de serialización incluyen todos los tipos primitivos de Java, objetos remotos de Java y cualquier otro objeto que implemente la interfaz **Serializable** (la cual incluye mechas de las clases del JDK estándar, como todos los componentes del AWT) [3].

Uno de los aspectos básicos en los que se apoya RMI es en la *persistencia de objetos,* es decir, en que los objetos no se destruyan y puedan ser enviados entre

máquinas en un entorno distribuido. En muchas aplicaciones, la persistencia de los datos se controla a través de archivos de texto o bases de datos comerciales, dependiendo de la complejidad de la aplicación y de los recursos puestos a disposición de los programadores. Para aplicaciones sencillas, los archivos de texto son suficientemente válidos, porque son flexibles, es muy fácil trabajar con ellos y no están limitados a su uso por un solo programa. Sin embargo, no son nada amigos de los objetos. Cuando el formato de un archivo se complica un poco más allá de una simple tabla o una lista de parámetros (lo que ocurre en casi todas las aplicaciones orientadas a objetos), el código para manejar estos archivos se vuelve dificil y consume mucho tiempo del programador.
Al otro lado del espectro están las bases de datos relacionales y orientadas a objetos que funcionan muy bien con programas que requieran características de bases de datos; Pero generalmente estas son muy caras. Si un diseño requiere estados en que hay que guardar datos, entonces se asume que las bases de daros son la elección. En muchos casos, todo esto es un archivo con formato orientado a objetos integrado en el propio entorno de programación.
Una situación similar existe también en el entorno de la programación distribuida. Los son flexibles y muy fáciles de utilizar, al igual que los archivos, pero presentan los mismos problemas que éstos cuando se transmiten formatos de datos complejos. Las aplicaciones distribuidas basadas en CORBA, por ejemplo, disponen de facilidades para transmitir objetos, pero es una solución costosa.
La *serialización* de objetos en Java proporciona una solución intermedia para salvar objetos en archivos y transmitirlos a través de la red. Tanto RMI como el API de los *JavaBeans* utilizan la serialización para guardar y transmitir objetos. Por lo tanto, en toda aplicación Java en que se vea involucrada la persistencia o distribución de objetos, la serialización es una poderosa herramienta de programación.
La serialización de objetos en Java permite escribir y leer objetos en *streams*, sean éstos archivos o . Esto proporciona a los programadores una forma sencilla de guardar tanto objetos individuales como grandes estructuras de objetos en archivos, o enviarlos a través de la red.
Desde la perspectiva del programador, gran parte de este trabajo se realiza automáticamente. El mecanismo de serialización mantiene control sobre los tipos de los objetos, las referencias entre ellos y muchos detalles de cómo están almacenados los datos. El API de serialización está muy estructurado, de tal modo que en muchos casos se puede manejar directamente con facilidad, mientras permite realizar acondicionasmientos muy complejos en caso necesario [2].

ProActive. Cuenta con tipos de creación de Objetos Activos (ver Capitulo II sesión I.4.1) y uno de los camino es usar ProActive.newActive y está basado en la instanciación de un nuevo objeto, en este este tipo de creación es posible poder pasar un tercer parametro a la llamada a newActive para crear el nuevo Objeto Activo en una Máquina Virtual Java especifica, posiblemente remota. La Máquina Virtual Java es identificada usando un objeto Node que ofrece los servicios minimos que ProActive necesita en una Máquina Virtual Java para comunicarse con ella. Esto se logra gracias a la utilización de RMI en la creación del nodo como se muestra en Figura I.11.

```
ActivoN a;
    org.objectweb.proactive.core.node.Node node;

 try {
      node =org.objectweb.proactive.core.node.NodeFactory.getNode("rmi://localhost/aNode");
      a    =(ActivoN)org.objectweb.proactive.ProActive.newActive("OActivoIN.ActivoN",params,node);
    }
   catch(org.objectweb.proactive.core.ProActiveException e)
        {System.err.println("Error del constructor: " + e.getMessage());
         e.printStackTrace();
        }
```

Figura I.11 Creación de un Nodo

Si el parámetro no se da, el OA es creado en la Máquina Virtual Java actual y se asigna a el nodo por defecto.

I.1.2.1.3 CORBA

CORBA (Common Object Request Broker Architecture) es una arquitectura para desarrollar aplicaciones distribuidas. Es una especificación normativa que resulta de un consenso entre los miembros de la **OMG** (Object Management Group), un consorcio que agrupa hoy por hoy a más de 700 empresas tanto de la industria informática como consumidores de la misma. La OMG fue constituida en 1989, y definieron el estándar CORBA en 1991, a partir de las necesidades existentes en la industria. Soluciona dos de los problemas derivados de la creación de software: la dificultad del desarrollo de aplicaciones cliente/servidor y la rápida integración de sistemas heredados, nuevos desarrollos, etc.
CORBA es una especificación para la tecnología de la gestión de objetos distribuidos (DOM). La tecnología DOM proporciona una interfaz de alto nivel orientado a objetos situada en la cima de los servicios básicos de la programación distribuida. El nivel más alto de la especificación es denominado arquitectura de gestión de objetos (OMA Object Management Architecture)

Esta norma cubre cinco grandes ámbitos que constituyen los sistemas de objetos distribuidos (Figura 1.12):

1) Un lenguaje de descripción de interfaces, llamado Lenguaje de definición de Interfaces **IDL** (Interface Definition Language), traducciones de este lenguaje de especificación IDL a lenguajes de implementación (como pueden ser C++, Java, ADA, etc.) y una infraestructura de distribución de objetos llamada Object Request Broker (ORB) que ha dado su nombre a la propia norma: Common Object Request Broker Architecture (CORBA).
2) Una descripción de servicios, conocidos con el nombre de **Servicios CORBA** (CorbaServices), que complementan el funcionamiento básico de los objetos de que dan lugar a una aplicación. Estas especificaciones cubren los servicios de nombrado, de persistencia, de eventos, de transacciones, etc. El número de servicios se amplía continuamente para añadir nuevas capacidades a los sistemas desarrollados con CORBA.

- **Naming Service** : Permite a un cliente encontrar un objeto, a través de su nombre.
- **Event Service**: Permite a los clientes y servidores enviar mensajes o eventos.
- **Security Service**: Permite dar permisos a objetos o grupos de objetos.
- **Trading Service**: Permite a los clientes encontrar objetos dada una interoperabilidad.
- **Transaction Service**: Permite tener interoperabilidad distribuidas bajo el protocolo de compromiso de dos fases (two phase commit).
- **Persistent State Service**:Servicio de persistencia de objetos.

3) Una descripción de servicios orientados al desarrollo de aplicaciones finales, estructurados sobre los objetos y servicios CORBA. Con el nombre de **CORBA** (CorbaFacilities), estas especificaciones cubren servicios de alto nivel, como las interfaces de usuario, los documentos compuestos, la administración de sistemas y redes, etc. La ambición es aquí bastante amplia ya que CorbaFacilities pretende definir colecciones de objetos prefabricados para aplicaciones habituales en la empresa: creación de documentos, administración de sistemas informáticos, etc.
4) Una descripción de servicios verticales denominados **Dominios CORBA** (CorbaDomains), que proveen funcionalidad de interés para usuarios finales en campos de aplicación particulares. Por ejemplo, existen proyectos en curso en sectores como: telecomunicaciones, finanzas, medicina, etc.
5) Un protocolo genérico de intercomunicación, el Protocolo General Inter-ORB **GIOP** (General Inter-ORB Protocol), que define los mensajes y el empaquetado de los datos que se transmiten entre los objetos. Además define implementación, de ese protocolo genérico sobre diferentes protocolos de transporte, lo que permite la comunicación entre los diferentes ORBs consiguiendo la interoperabilidad de elementos de diferentes vendedores. Por ejemplo el IIOP para redes con la capa de transporte TCP [4].

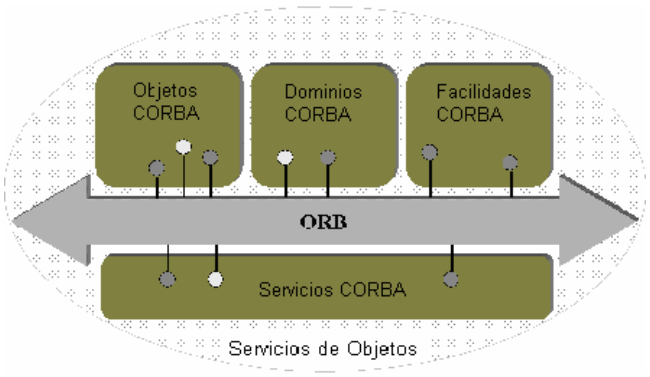


Figura 1.12 Sistemas de objetos distribuidos

Object Request Broker (ORB) es el sistema intermedio (*midleware*) que establece relaciones cliente/servidor entre objetos. Mediante la utilización de un ORB, un cliente puede invocar transparentemente un método de un objeto servidor que se encuentre en la misma máquina o en otra distinta. El ORB intercepta la llamada realizada por el objeto que implementa la petición, pasa los parámetros, invoca el método y retorna los resultados. No es necesario que el cliente sepa dónde se localiza el objeto que ejecutará el método, el lenguaje en el que está programado, el sistema operativo, ni cualquier otro aspecto que no sea su interfaz. Hay que resaltar que los papeles de cliente y servidor que se dan a los objetos son simplemente para coordinar las interacciones entre ambos; estos papeles pueden cambiar ya que un objeto puede ser cliente o servidor dependiendo de la invocación.

En la Figura 1.13 se muestra la estructura de un sistema distribuido basado en CORBA. En él se puede observar dos partes: una cliente y una servidora. En la parte cliente existirá un programa cliente propiamente dicho al que CORBA le añade cierta infraestructura para permitir la comunicación con el servidor a través de la red. Del otro lado, la parte servidor estará formado por el objeto que exporta su funcionalidad, integrado en el servidor (proceso en el que se ejecuta) y diversos elementos que permiten que las invocaciones realizadas por el cliente a los métodos del objeto lleguen a éste, sean procesadas y sus resultados devueltos.

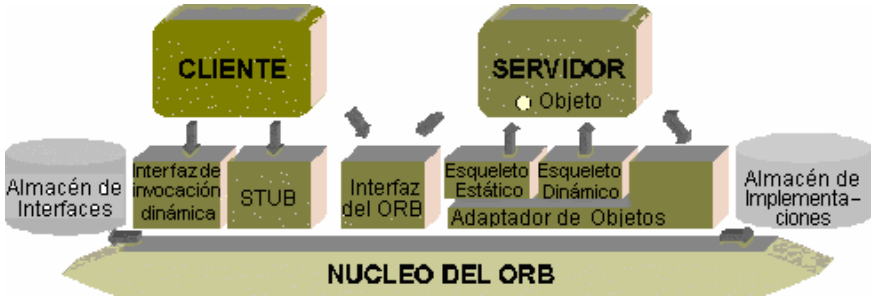


Figura 1.13 Estructura de un sistema distribuido basado en CORBA

La parte cliente estará formada por:

- **Los Stubs del Cliente**

Proporcionan una capa intermedia entre el cliente y el núcleo del ORB. Definen la manera en que los clientes cómo los clientes invocan los servicios que proporcionan los objetos servidores. Desde la perspectiva del cliente, el stub actúa como una especie de proxy, dando la impresión de que la invocación se está realizando sobre un objeto local como en cualquier aplicación orientada a objetos. El stub se encarga de codificar la operación y sus parámetros, y de enviarla de forma remota.

- **El interfaz de invocación dinámica**

Permite descubrir en tiempo de ejecución métodos para ser invocados. CORBA define una biblioteca, que permite localizar el método, generar los parámetros, realizar la llamada remota y recoger los resultados.

- **El almacén de interfaces**

Permite obtener y modificar la descripción de todos los componentes que en él están registrados, los métodos que soporta y los parámetros que requiere. CORBA llama a esas descripciones firmas. El almacén de interfaces (*Interface Repository*) es una base de datos distribuida modificable en tiempo de ejecución.

- **El interfaz del ORB**

Consiste en unas cuantas librerías de servicios locales para realizar labores auxiliares en la aplicación. Por ejemplo, CORBA proporciona APIs (*Aplication Programming Interface*, interfaces de programación de aplicaciones) para convertir referencias a objetos a cadenas. Estas llamadas pueden ser muy interesantes si se necesita comunicar o almacenar referencias a objetos.

La capacidad de soportar tanto invocaciones dinámicas como estáticas, además del almacén de interfaces, da a CORBA una gran ventaja sobre las plataformas intermedias competidoras. Las invocaciones estáticas son rápidas y fáciles de programar. Las invocaciones dinámicas proporcionan la máxima flexibilidad, pero son difíciles de programar. Esta últimas son muy usadas por herramientas que descubren servicios en tiempo de ejecución.

Por lo que respecta al lado del servidor no hay diferencia entre la invocación dinámica y estática, ya que ambas tienen el mismo mensaje semánticamente hablando. En los dos casos el ORB localiza al Adaptador de Objetos del objeto y le transmite un mensaje que contiene el nombre del servicio a invocar y sus parámetros. La implementación recibe a través del esqueleto del objeto los datos necesarios, ejecuta el servicio y retorna el resultado para que sea enviado al cliente en forma de un nuevo mensaje.

Los elementos que componen la parte servidor son los siguientes:

- **El esqueleto del servidor**:

Proporciona los elementos necesarios para que los clientes invoquen los servicios exportados por el objeto. Estos esqueletos realizan una función similar a los stubs del cliente tratando de hacer transparente todo el proceso de comunicación.

- **El esqueleto de interfaces dinámicos (DSI)**:

Proporciona un mecanismo de enlazado en tiempo de ejecución para servidores que necesitan manejar llamadas a métodos que no tienen esqueletos estáticos definidos.

- **El adaptador de objetos**:

Proporciona en tiempo de ejecución un entorno para la instanciación de objetos en el servidor (proceso que los acoge), la asignación de referencias y la gestión las peticiones que les lleguen.

- **El almacén de implementaciones**:

Proporciona en tiempo de ejecución un almacén de información acerca de las clases que el servidor soporta, los objetos instanciados y sus identificadores.

- **El interfaz del ORB**:

Consiste en unas cuantas APIs de servicios locales iguales a las de la parte cliente.

Uno de los lenguajes de implementación para los que CORBA describe traducción de IDL (Interface Definition Language) es el Java, quizás hoy por hoy el más

utilizado, sus principales características son: el mapeo para tipos básicos y estructurados, la traducción para interfaces, módulos, contextos, atributos, métodos, etc [4].

Algo muy interesante en CORBA, es la posibilidad de usar lenguajes de scripts para implementar servidores y clientes, como por ejemplo *Corbascript*, lenguaje creado por la OMG y CORBA para Python, la ventaja de usar python en lugar de corbascript es que tienes muchas librerías ya disponibles y listas para usar.

I.1.3 Aplicaciones Distribuidas

Las aplicaciones distribuidas son aquellas construidas sobre sistemas distribuidos, con las consiguientes ventajas potenciales en cuanto a rendimiento, disponibilidad y escalabilidad.
Localización. El usuario no necesita conocer la localización física de un objeto en el sistema. Asimismo, el acceso a objetos locales o remotos se realiza del mismo modo.

- Replicación. El usuario no precisa conocer el grado de replicación de un objeto (ni siquiera si está replicado), ni el modelo en que tal replicación está implementada.
- Fallos El sistema debe enmascarar (al menos en cierto grado) los fallos de alguno de sus componentes de modo que la prestación del servicio continúe.
- Concurrencia El acceso a recursos compartidos paralelamente por diversos usuarios requiere una sincronización por parte del sistema.

Los paradigmas para la construcción de aplicaciones distribuidas pueden ser clasificados en dos grupos.

- Grupo 1
  En este grupo se encuentra :
  RPC llamada a procedimiento remoto (el tradicional)
  RMI método de invocación remota (lo mas reciente de este grupo)
  CORBA arquitectura común para el manejo de peticiones de objetos

En este grupo, la funcionalidad de las aplicaciones son particionadas entre nodos participantes. Los diferentes participantes usan el paso de mensaje para coordinar el computo distribuido. Él cómputo es particionado, los nodos participantes intercambian resultados intermedios e información de sincronización.

- Grupo 2

En el segundo grupo la computación es migrada hacia los recursos. Este paradigma es útil para aplicaciones que solicitan reacciones inmediatas al flujo de datos entrantes en tiempo real y para las aplicaciones distribuidas que son fuertemente acopladas.

Con respecto a la aplicación hoy en día las demandas muestran la necesidad insaciable por poder de cómputo (Cómputo científico, Biología, Química, Física, Cómputo de propósito especial como: Video, Gráficas, CAD, Bases de Datos, etc.)

La tendencia en los sistemas de archivos distribuidos son afectados por los cambios tecnológicos de los últimos años. Además con el rápido avance que se da en las redes de comunicaciones y su incremento en el ancho de banda la creación de paquetes que ofrecen la compartición de archivos es común de encontrarse en el mercado. En la industria, el esquema más solicitado es aquel que permite acceder a los grandes volúmenes de información de los grandes servidores desde las computadoras personales o convencionales y desde otros servidores. ´
Entre los sistemas de archivos distribuidos más populares que existen en la actualidad, tenemos los que nos proporciona Netware, tales como:

- Remote Filke Sharing (RFS en UNIX)
- Network File System (NFS de Sun Microsystems)
- Andrew File System (AFS)

Por otro lado el concepto de escalabilidad propone que cualquier computadora individual ha de ser capaz de trabajar independientemente como un sistema de archivos distribuido, pero también debe poder hacerlo conectado a muchas otras máquinas.
Un sistema de archivos debería funcionar tanto para una docena de equipos como para varios millares. Igualmente no debería ser determinante el tipo de red utilizada (LAN o WAN) ni las distancias entre los equipos. Aunque este punto sería muy necesario, puede que las soluciones impuestas para unos cuantos equipos no sean aplicables para varios otros. De igual manera, el tipo de red utilizada condiciona el rendimiento del sistema, y podría ser que lo que funcione para un tipo de red, para otro necesitaría un diseño diferente.

I.2 Cómputo Paralelo (CP)

Se le llama cómputo paralelo a la ejecución de más de un cómputo (cálculo) al mismo tiempo usando más de un procesador en una computadora.
La meta es reducir al mínimo el tiempo total de cómputo distribuyendo la carga de trabajo entre los procesadores disponibles. Para fines prácticos es necesario tener en mente que la paralelización es un factor básico para tener un alto desempeño en los equipos de cómputo.
Una de las razones principales para utilizar el paralelismo en el diseño de hardware o software, es obtener un alto rendimiento o mayor velocidad al ejecutar un programa.
La velocidad de procesamiento no es solamente la razón para utilizar el paralelismo. La construcción de aplicaciones más complejas ha requerido una computadora más rápida, y las limitaciones en el desarrollo adicional de computadoras seriales han llegado a ser más y más evidentes.

I.2.1 Antecedentes del CP

A través de la historia se han establecido máquinas o computadoras paralelas, así como una taxonomía para sistemas de cómputo paralelo.

Una computadora paralela cuenta con dos o más procesadores independientes que cooperan y se comunican para la resolución de un algoritmo. Esto obliga a la descomposición del problema y a resolverlo de modo que se distribuya su procesamiento, lo cual requiere un manejo de la coordinación de acciones y comunicación entre procesadores.
Un algoritmo paralelo, es aquel con una especificación rigurosa de la secuencia de pasos a realizar por cada autómata mas las acciones de comunicación y sincronización entre ellos, de modo de alcanzar el resultado deseado en un tiempo finito, el cual es ejecutable sobre una arquitectura de múltiples autómatas.

El paralelismo es un concepto asociado con el hardware multi-procesador. Para ser útil requiere que el algoritmo admita una descomposición en múltiples procesos que se comuniquen y sincronicen (concurrencia).
Estos son algunos ejemplos de computadoras paralelas:

- SMPC (Shared Memory Parallel Computer)

En esta los procesadores trabajan sincrónicamente con el mismo programa sobre una memoria global compartida. Cuando el procesador P1 debe comunicarse con P2, lo hace leyendo la memoria escrita por P2. Cada procesador puede tener su propia entrada y salida (E/S).

- BTPP (Binary Tree Parallel Processor)

Los procesadores en estas computadoras se conectan siguiendo una estructura de árbol binario, sin memoria compartida. Cada uno se puede comunicar con un “procesador padre” y dos “procesador hijo”. Los procesadores trabajan sincrónicamente, no necesariamente con el mismo programa. Sólo el procesador raíz y las hojas externas tienen E/S.

- MIPP (Matriz Interconnected Parallel Processor)

Los procesadores dentro de estas computadoras se estructuran en filas y columnas, cada procesador puede interconectarse con alguno o todos sus vecinos, cada procesador tiene capacidad de memoria local independiente y la E/S puede estar centralizada o distribuida en alguna fila o columna

De acuerdo a Flyn [1] una clasificación de las arquitecturas de Procesamiento Paralelo:

- SISD
- MISD
- SIMD
- MIMD

De esta clasificación la arquitectura MIMD es considerada la más general, es decir aplicable a una amplia gama de problemas (al menos más amplia que las demás arquitecturas); así también es considerada la más escalable, definiendo

primeramente Escalabilidad como la capacidad de aumentar la cantidad de recursos para resolver problemas mayores (en datos y/o en procesamiento), es la más escalable, puesto que no necesita sincronismo al nivel del reloj.

Existen dos clases de MIMD:

- Memoria compartida
  - Multiprocesadores: una única máquina con muchos procesadores
  - Una única visión de le memoria (mapa de memoria)
- Memoria distribuida
  - DMPC: lo mismo
  - Multicomputadora: muchas computadoras
  - Losely coupled: independencia, asincronismo
  - Paso de mensajes: mirando al procesamiento/programas paralelos: CSP

"La computación Paralela es el procesamiento de información que enfatiza la manipulación concurrente de elementos de datos pertenecientes a uno o más procesos resolviendo un problema en común" *(Quinn 1994)*.
La computación paralela ha sido una disciplina madura y muy activa durante los últimos 20 años. Desde finales de los años 80 y en toda la década de los años 90 diversos investigadores han desarrollado trabajos de paralelización de aplicaciones numéricas y cooperativas haciendo uso de los recursos de las redes de comunicación: se trata en muchos casos de resolver problemas de elevado costo computacional con un costo económico reducido.

I.2.2 Estado actual del CP

Los últimos años han sido cruciales pues el avance de cada día ofrece nuevos equipos, aplicaciones y las tecnologías diversifican sustancialmente la forma de trabajo e imponen nuevos retos por alcanzar la necesidad de nuevos equipos, tecnologías, métodos y aplicaciones que den un servicio de cómputo de alto rendimiento en nuestros días, para lograrlo es necesaria la construcción de superclusters, el fomento a cómputo masivamente paralelo y de teraescala, así como el desarrollo de grids computacionales.

La creación de superclusters ofrece a los investigadores otras opciones de trabajo para la obtención de resultados óptimos en tiempos razonables y representa menores costos de adquisición, operación y mantenimiento. Construir y mantener un cluster para uso restringido es relativamente sencillo, sin embargo no lo es para un cluster de servicio general, y en este último caso surgen dificultades adicionales a las de construir una máquina más potente mediante la integración de computadoras personales, que consisten en revisar, configurar y aprender a usar diversos componentes de software hasta administrar, monitorear y actualizar equipos en configuraciones diversas que crecen más rápido que sus contrapartes paralelas comerciales.

Un servicio basado en clusters es tan confiable como un equipo comercial (existe la posibilidad de crecer hasta contar con un supercluster de 512 procesadores o más). Es importante comentar que una parte importante de los servicios de supercómputo en los próximos años, será sustentada por superclusters para cómputo paralelo escalable o masivo.

En nuestros días, el cómputo paralelo es una opción adicional para la comunidad científica internacional pueda trabajar sus investigaciones con mayores capacidades de procesamiento y con el uso de computadoras paralelas. Se logran reducir los tiempos, obtener resultados óptimos de un programa y realizar simulaciones de mucho mayor tamaño y volumen. Si bien es cierto que se pueden construir computadoras con cientos o miles de procesadores, el hecho de construir una aplicación que use eficientemente estos recursos, necesariamente requiere especialistas en programación del más alto nivel, con amplios conocimientos en algoritmos, programación paralela y sistemas de comunicación para superar los retos que representa la escalabilidad al usar la integración de procesadores y computadoras.

Otra forma de trabajar con cómputo de alto rendimiento, es a través del cómputo de "***Teraescala***", representado por el uso de equipos capaces de realizar billones de operaciones por segundo o ***teraflops*** y de generar miles de billones de bytes de información o ***petabytes***, particularmente útil en investigaciones biomédicas y astronómicas.
Para integrar equipos de esta magnitud, se requiere tecnologías que permitan interconectar equipos paralelos en constelaciones, lo que se conoce como clusters y medios de almacenamiento masivos. Más allá del procesamiento de información, para que realmente sean útiles, estos equipos deben permitir que los usuarios cuenten con métodos de acceso fáciles y cómodos, tanto a las unidades de procesamiento como a las de almacenamiento.
Un GRID es la integración de equipos de trabajo ubicados físicamente en forma remota que comparten recursos, instrumentos, dispositivos y tecnologías como las asociadas a redes de alta velocidad o dedicadas, equipos disponibles para el cálculo numérico como computadoras vectoriales, paralelas, clusters, redes de estaciones de trabajo, dispositivos de almacenamiento e instrumentos como telescopios, microscopios o controladores, y sistemas de software que interaccionan entre el usuario final y los dispositivos para recibir y ejecutar sus solicitudes.

Estas tecnologías de GRID buscan cambiar la forma en que se usan los recursos computacionales, de manera que los investigadores tengan la oportunidad de usar cierta cantidad de recursos, sin importar su ubicación geográfica ni su pertenencia a una organización. Los GRID tendrán a futuro la consecuencia positiva de cohesionar a las diversas comunidades de cómputo de alto rendimiento.

I.2.3 Aplicaciones del CP

La demanda de la computación paralela en las aplicaciones es debido a la insaciable necesidad de computadoras rápidas, para él cómputo científico como la Física, Química, Biología, Oceanografía, Astronomía, entre otros se ha vuelto sumamente indispensable, por el lado del cómputo de propósito general es utilizada en aplicaciones de video, Visualización, CAD, Bases de Datos, Procesamiento de transacciones, etc.

Por el lado de la Investigación, en la computación paralela actualmente se están explorando diversos algoritmos masivamente paralelos orientados a problemas intensivos en cómputo. A modo general, algunas líneas de investigación son:

- Optimización de Rendimiento de Algoritmos Paralelos para problemas NP-completos.
- Simuladores de Ambientes de Procesamiento paralelo y distribuido.
- Aplicaciones Paralelas para resolución de problemas en el Área de Optimización Combinatoria.

Por su parte la demanda en cómputo en la Ingeniería, requiere máquinas paralelas grandes, útiles en muchas industrias como:

- Petróleo (análisis de reservas)
- Automotriz (simulación de choques, análisis de arrastre, eficiencia de combustión)
- Aeronáutica (análisis de flujo de aire, eficiencia de motores, mecánica estructural, electromagnetismo)
- CAD (diseño asistido por computadora)
- Farmacéuticas (modelado molecular)
- Visualización (Entretenimiento, arquitectura, simulaciones 3D)
- Modelado financiero.

I.3 Diferencias entre cómputo Paralelo y Distribuido

Cuando se trabaja con computo paralelo, se utiliza hardware especializado (como por ejemplo transputers) y software especializado (como el lenguaje OCCAM), y no se escriben programas orientados a objetos, sino programas paralelos. En el cómputo paralelo se paralelizan tareas y generalmente se resuelven dentro del mismo hardware. MPI y PVM son bibliotecas de paso de mensajes para programar en paralelo. En cambio cuando se trabaja con cómputo distribuido se hace uso de clusters, el cual es un conjunto de computadoras independientes interconectadas, usadas como un recurso unificado de cómputo; así también se hace uso del pasó de mensajes (como PVM, o HIPPI). También se paralelizan tareas y a cada nodo se entrega la tarea paralelizada a resolver, y así al terminar cada nodo, regresa su respuesta al controlador de nodos, quien casi siempre muestra el resultado final.

En el caso de los clusters, los procesos efectivamente se ejecutan en cada nodo, lo que podría entenderse como cómputo distribuido, pero resuelven un mismo problema, ejecutan el mismo programa en paralelo.

Hoy en día la frontera entre el cómputo distribuido y el cómputo paralelo es muy tenue, cada vez lo es más, con el uso de los clusters y del GRID. La distinción hay que hacerla mas desde el punto de vista del software, de la aplicación misma que desde el hardware.
Una de las opciones para programar en paralelo es a través del paso de mensajes utilizando MPI. En la actualidad es la mas utilizada, incluso en máquinas de memoria compartida, donde se podrían utilizar otros modelos. Y se puede programar con MPI desde fortran, C, C++ y hasta java, todo depende de las virtudes de cada programador. Por el lado del software para cómputo distribuido esta HURD, que ofrece la ventaja para los clusters que es, que al ser inherente al núcleo la comunicación por paso de mensajes la adición de nodos es transparente, ya que es igual que se comunique a un proceso dentro de la misma máquina, que a un servidor remoto. Por supuesto, esto será cuando el soporte para el paso de mensajes del núcleo HURD a través de la red funcione, lo cual no sucede hasta el día de hoy.

I.3.1 Problemas a resolver en el cómputo paralelo y distribuido

- ¿Cómo determinar si una tarea es paralelizable o no?
- ¿Cómo disminuir la latencia en la comunicación para maximizar el tiempo de respuesta?

No es relevante si la solución es estructurada u orientada a objetos. Ninguna de las dos filosofías de programación ofrece una solución para paralelizar tareas, y la latencia se combate con eficiencia en la comunicación entre los elementos que realizaran el cómputo. Y en este aspecto, los lenguajes OOP tienen un "overhead" que en muchas ocasiones los hace imprácticos.
Una de las soluciones que se han explorado para disminuir la latencia es que la comunicación entre procesos en los nodos se realice desde el espacio de las aplicaciones, evitando entre otras cosas la copia de la información de la memoria del proceso a la memoria del sistema para poder enviarla. En este caso, el papel del kernel podría ser irrelevante.

I.4 Infraestructuras Genéricas

I.4.1 Concepto

Si bien una infraestructura “son los recursos requeridos para una actividad”, según diccionario Merrian Webster [Internet-8], por otro lado Larimer [ref] define a una infraestructura “como el fundamento o el marco subyacente de servicios básicos de instalaciones e instituciones sobre la cual se basa el crecimiento y el desarrollo de un área, comunidad o sistema”; por su lado la base de datos léxica para el laboratorio inglés de la ciencia cognoscitiva (WordNet), nos proporciona dos definiciones de infraestructura:

1) “Marco o características básicas de un sistema u organización”
2) “Instalaciones básicas y equipo necesario para el funcionamiento de un país o área”

En la tecnología de información, la infraestructura es vista como todo aquello que soporte el flujo y procesamiento de información.

Por otro lado según el diccionario de la lengua española dice que genérica es “Común a todas las especies”. El hecho de reunir sistemas de cómputo compuestos por un gran número de CPU´s, conectados mediante una red de alta velocidad. Se le conoce como “genérico de sistemas distribuidos”, desde el punto de vista del hardware.

Pues bien una infraestructura genérica es básicamente un conjunto mínimo necesario de primitivas para el desarrollo de aplicaciones en algún campo. Para programar en este caso aplicaciones de flujo de objetos, es obligatorio tener un conjunto de instrucciones que permitan a los programadores enfocarse en el problema y no en los detalles de la implementación.

Capítulo II ProActive

II.1 Introducción

ProActive PDC (Internet -10), es una biblioteca para cómputo Paralelo, Distribuido, Concurrente (PDC) Orientado a Objetos con seguridad y movilidad desarrollada en Java. La biblioteca está hecha a partir de clases estándar de Java. ProActive proporciona una Interfaz de Programación de Aplicaciones (API) propia.

La base de ProActive son los objetos activos, los cuales son las unidades básicas de actividad y distribución usadas para construir aplicaciones concurrentes. Un objeto activo posee su propio hilo, que ejecuta solamente los métodos invocados en este objeto activo por otros objetos activos y pasivos del subsistema al que pertenece este objeto activo.

Con ProActive PDC el programador no tiene que manipular explícitamente los objetos hilos a diferencia de Java estándar. Los objetos activos pueden ser creados en cualquiera de los anfitriones involucrados en el cómputo. Una vez que un objeto activo es creado, su actividad (el hecho de que éste tenga su propio hilo) y su localización (local o remota) son perfectamente transparentes, de hecho cualquier objeto activo puede ser manipulado tal y como si éste fuera una instancia pasiva de la misma clase.

La movilidad de los objetos activos se hace a través de la primitiva de migración *migrateTo()*.

ProActive esta construido en la cima del protocolo metaobjeto (MOP), que permite abstraer la invocación de los métodos y de las llamadas al constructor, el cual es útil para la implementación de nuevos metacomportamientos.

ProActive PDC es un proyecto del consorcio ObjectWeb, que es una comunidad de Software Open Source middleware creada a final de 1999 por FRANCE TELECOM R&D, Bull y por INRIA (Instituto Nacional de Investigación en Informática y Automatización).

ObjectWeb es un consorcio internacional que fomenta el desarrollo de aplicaciones ***middleware*** cutting-edge: EAI, e-business, clustering, grid computing, managed services y más.

El consorcio ObjectWeb es apoyado por los ministerios de economía, finanzas e industria de Francia, a través de algunos proyectos. Por otro lado, Bull es Abierto, flexible y asegura las soluciones, Bull diseña, desarrolla los servidores y el software para un ambiente abierto, integrando las tecnologías más avanzadas.

Trae a sus clientes conocimientos técnicos para ayudarles en la transformación de sus sistemas de información y para optimizarlos.
En un sistema de cómputo distribuido el middleware es definido como la capa de software que se encuentra entre el sistema operativo y la aplicación en cada sitio del sistema. Simultáneamente por el crecimiento de las aplicaciones de red básicas, las tecnologías ***middleware*** están tomando una gran importancia. Cubren una amplia gama de los sistemas de software, incluyendo objetos distribuidos, componentes, mensajes orientados a comunicación y soporte de aplicaciones móviles [Internet 7].

II.2 Antecedentes

II.2.1 Transparencia Secuencial, Multihilos y Distribuido

La mayoría de las veces, las actividades y la distribución no se conocen al principio y cambian con el tiempo. La transparencia implica la reutilización y transiciones suaves e incrementales entre el cómputo secuencial, multihilos y distribuido (Figura II.1).

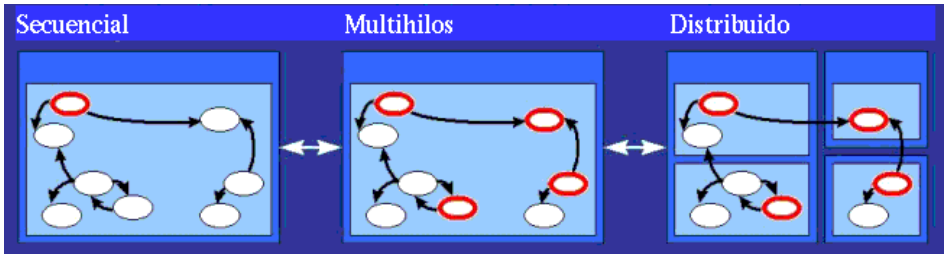


Figura II.1 Cómputo secuencial, multihilos y distribuido

Existe aún un hueco enorme entre las aplicaciones distribuidas y multihilos de Java, las cuales prohíben la reutilización de código para construir aplicaciones distribuidas a partir de aplicaciones multihilos. Tanto JavaRMI como JavaIDL son ejemplos de bibliotecas de objetos distribuidos en Java, que dificultan el trabajo del programador porque éste necesita realizar modificaciones profundas al código existente para convertir los objetos locales en objetos que puedan ser accesibles de manera remota. En estos sistemas los objetos remotos necesitan ser accesados a través de una interfase específica. Como consecuencia, estas bibliotecas de objetos distribuidos no permiten el polimorfismo entre los objetos locales y los remotos. Considerando que el polimorfismo es el primer requerimiento para un ambiente de trabajo de meta cómputo, éste se necesita fuertemente para permitir al programador concentrarse en el modelo y en las

características algorítmicas mas que en las tareas de bajo nivel tales como distribución de objetos, mapeo y balance de carga.
ProActive es una biblioteca diseñada para el desarrollo de aplicaciones en un modelo introducido por el lenguaje Eiffel//

Modelo Eiffel// :
Las Principales características de este modelo (Figura II.2) son:

- La aplicación esta estructurada en subsistemas: Hay un objeto activo (y por lo tanto un hilo) para cada subsistema y un subsistema para cada objeto activo (o hilo). Cada subsistema esta entonces compuesto de un objeto activo y de cualquier número de objetos pasivos (posiblemente ninguno). El hilo de un subsistema solamente ejecuta métodos en los objetos de esté subsistema.
- No hay objetos pasivos compartidos entre los subsistemas.

Estas dos características tienen consecuencias de gran importancia en la topología de la aplicación.

- De todos los objetos que componen a un subsistema (objetos activos y pasivos), solamente el objeto activo es conocido por los objetos que se encuentran fuera del subsistema.
- Todos los objetos (pasivos y activos) pueden tener referencias dentro de objetos activos.
- Si un objeto 1 tiene una referencia dentro de un objeto pasivo 2, entonces el objeto 1 y 2 son parte del mismo subsistema.

Esto también tiene consecuencias en la semántica en el paso de mensajes entre subsistemas

- Cuando un objeto en un subsistema llama a un método en un objeto activo, el parámetro de la llamada debe hacer referencia en objetos pasivos del subsistema (Figura II.2), lo cual podría llevar a objetos pasivos compartidos. Este es el por qué de que objetos pasivos pasados como parámetros de llamadas en objetos activos sean siempre pasados por copia profunda. Los objetos activos por otro lado son siempre pasados por referencia. Simétricamente, esto también se aplica para objetos que son regresados de métodos invocados en objetos activos.

- Cuando un método es llamado en un objeto activo, éste regresa inmediatamente un objeto futuro, el cual es un contenedor para el resultado de la invocación al método devuelto. Desde el punto de vista del subsistema, ninguna diferencia existe entre el objeto futuro y el objeto que puede ser regresado si la misma llamada es hecha dentro de otro objeto pasivo. Entonces el hilo invocante puede continuar ejecutando su código tal y como si la llamada hubiera sido efectivamente realizada. El rol del objeto futuro es bloquear este hilo si este invoca a un método en el objeto futuro y

el resultado aún no es colocado (ejemplo: el hilo del subsistema en el cual la llamada fue recibida no ha ejecutado la llamada y puesto el resultado en el objeto futuro). Esta política de sincronización entre objetos es conocida como espera por necesidad.

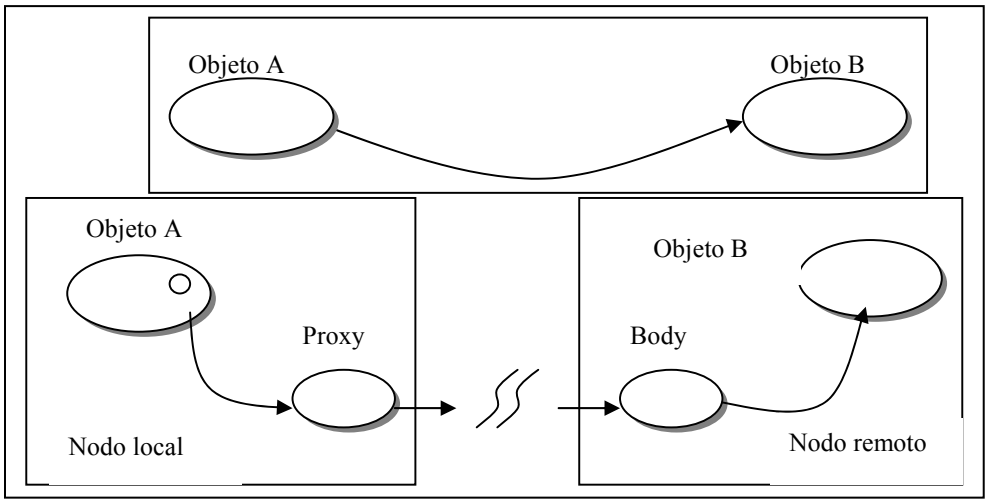


Figura II.2 Llamada dentro de un objeto activo

II.3.2 Modelo de cómputo

Este modelo esta compuesto de las características siguientes: **Modelo** heterogéneo, el cual se utiliza tanto en objetos activos como en objetos pasivos, comunicación sistemática asíncrona a través de objetos activos, objetos pasivos no compartidos (llamada por valor entre objetos activos), continuaciones automáticas (un mecanismo de delegación transparente), espera-por-necesidad, control centralizado y explicito (bibliotecas de abstracciones). En este modelo se cuenta con dos características claves para obtener una mejor reutilización y transparencia, la primer característica es espera por necesidad, que es la sincronización entre objetos futuros, transparentes, implícitos y sistemáticos; facilidad de programación, sincronización y la reutilización de métodos existentes; la segunda característica es el polimorfismo entre objetos estándar y activos, esta característica se refiere al tipo de compatibilidad para clases y no solamente para interfases, necesario también para los objetos futuros y mecanismo dinámico (el dinamismo es obtenido sí es necesario).

II.4 Objetos Activos

Los objetos activos son las unidades básicas de actividad y distribución usadas para construir aplicaciones concurrentes usando ProActive. Un objeto activo posee su propio hilo, este hilo ejecuta solamente los métodos invocados en este objeto

activo por otros objetos activos y pasivos del subsistema al que pertenece este objeto activo.

El objeto activo es actualmente la composición de dos objetos: Un cuerpo y un objeto Java estándar. El cuerpo no es visible desde afuera del objeto activo. Entonces todo luce tal y como si el objeto estándar fuera activo. El cuerpo es responsable de recibir llamadas en el objeto activo, almacenando estas llamadas en una cola de llamadas pendientes (que también se llaman peticiones), éste también ejecuta estas llamadas en un orden determinado por una política de sincronización especifica. Sí ninguna política de sincronización especifica se proporciona, las llamadas son manejadas de manera FIFO (primero en entrar primero en salir), entonces el hilo de un objeto activo escoge alternativamente un método en la cola de peticiones pendientes y la ejecuta.

Es importante notar que ningún paralelismo se proporciona dentro de un objeto activo, es una decisión importante en el diseño de ProActive PDC que habilita el uso de pre o post condiciones y clases invariantes.

Por el lado del subsistema, el cual envía una llamada a un objeto activo, este objeto activo es representado por un *proxy*, cuya responsabilidad principal es generar objetos futuros para la representación de valores futuros, transformando llamadas en objetos *Request* (en términos de metaobjeto, esto es una abstracción) y ejecuta una copia profunda de objetos pasivos pasados como parámetros.

II.4.1 Creación de Objetos Activos

En ProActive hay dos maneras de crear un objeto activo:

Un camino es usar **ProActive.newActive** y está basado en la instanciación de un nuevo objeto.
El otro camino es usar **ProActive.turnActive** y está basado en el uso de un objeto existente

1) <u>Creación basada en instanciación</u>
Cuando se usa la creación basada en la instanciación cualquier argumento pasado al constructor del objeto abstraído a través de **ProActive.newActive** es serializado y pasado por copia al objeto. Esto es debido a que el modelo detrás de ProActive es uniforme, donde el objeto activo es instanciado local o remotamente. Se garantiza que los parámetros van hacer pasados por copia al constructor. Cuando se usa **ProActive.newActive** se debe asegurar que los argumentos del constructor son serializables. Por otro lado, la clase usada para crear el objeto activo no necesita ser serializable aún en el caso en que el objeto activo sea creado remotamente

```
Activo a;
   Object[] params = new Object[] {"astring"};
   try
  {a=(Activo)org.objectweb.proactive.ProActive.newActive(Activo.class.getName(),
       params); /* Activo: clase que se dese hacer Activa */
  } catch(org.objectweb.proactive.ActiveObjectCreationException e)
          { System.err.println("Error del constructor: "+e.getMessage());
            e.printStackTrace();
          }
          catch(org.objectweb.proactive.core.node.NodeException e)
                {
                e.printStackTrace();
               }
```

Este código crea un objeto activo de la clase **A** en la máquina virtual de Java (MVJ) local. Si la invocación del constructor de la clase **A** lanza una excepción, ésta es puesta dentro del tipo de excepciones **ActiveObjectCreationExcepcion.** Cuando la llamada a newActive regresa, el objeto activo a sido creado y su hilo activo a sido inicializado.

El primer parámetro de **newActive** es una cadena que contiene el nombre totalmente válido de la clase que se desea hacer activa.
Los parámetros del constructor tienen que ser pasados como un arreglo de objetos. Entonces de acuerdo al tipo de elementos de este arreglo, en tiempo de ejecución ProActive determina que constructor de la clase **A** se llamará. Sin embargo existe espacio para alguna ambigüedad en la resolución del constructor debido a que:

a) Como los argumentos del constructor son almacenados en un arreglo de tipo **Object[ ],** los tipos primitivos tienen que ser representados por sus tipos de empaquetamiento de objetos. En el ejemplo anterior se usa el objeto **Integer** para empaquetar el valor entero *int 26*. Una ambigüedad surge si dos constructores de la misma clase solamente difieren por la conversión de los tipos primitivos a su correspondiente clase empaquetadora. En el ejemplo anterior una ambigüedad existe en el primer y segundo constructor.
b) Si un argumento es nulo, el tiempo de ejecución puede no determinar su tipo. Esta es la segunda fuente de ambigüedad, en el ejemplo anterior, una ambigüedad existe entre el tercer y cuarto constructor si el segundo elemento del arreglo es nulo.

```
public A( int i) {   // }
public A( Integer  i { // }
public A( int i, String s) { // }
pblic A( int i, Vector v) { // }
```

Es posible pasar un tercer parámetro a la llamada a **newActive** para crear el nuevo objeto activo en una MVJ específica, posiblemente remota. La MVJ es identificada usando un objeto **Node** que ofrece los servicios mínimos que

ProActive necesita en una MVJ para comunicarse con ella. Si ese parámetro no se da, el objeto activo es creado en la MVJ actual y se asigna al nodo por defecto.

Un nodo es identificado por el URL, el cual se forma usando el protocolo, con los siguientes datos: el nombre de la máquina anfitriona donde reside la MVJ y donde está ubicado el nodo y el nombre del nodo. El **NodeFactory** permite crear o buscar nodos. El método **newActive** puede tomar como parámetro el URL del nodo como una cadena o como un objeto **Node** que apunta a un nodo existente Ejemplo:

```
ActivoN a;
Object[] params = new Object[] {"astring"};
org.objectweb.proactive.core.node.Node node;
 try
{
  node=org.objectweb.proactive.core.node.NodeFactory.getNode("rmi://localhost/
  aNode");
  a=(ActivoN)org.objectweb.proactive.ProActive.newActive("OActivoIN.ActivoN",
  params,node);
}
catch(org.objectweb.proactive.core.ProActiveException e)
      {
      System.err.println("Error del constructor: " + e.getMessage());
      e.printStackTrace();
      }
```

2) Creación basada en objetos

Es usada para convertir un objeto pasivo existente a uno activo, lo anterior se ha presentado en ProActive como una respuesta al siguiente problema:

Considerando, por ejemplo que una instancia de la clase ***A*** es creada dentro de una biblioteca y regresada como el resultado de una llamada a un método. Como una consecuencia, no se tiene el acceso al código fuente donde el objeto es creado, lo cual previene de modificarlo para crear una instancia activa de ***A***. Aún si esto fuera posible, esto no puede ser ya que no se desea una instancia activa de ***A*** por cada llamada en este método.

Cuando se usa la creación basada en objetos, se crea el objeto que esta siendo abstraído como un objeto activo antes del manejo, por lo tanto no hay serialización involucrada cuando se crea el objeto.
Cuanto se invoca ***ProActive.turnActive*** en el objeto, son posibles dos casos, uno, si se crea el objeto activo localmente, este no será serializado, y dos, si se crea el objeto activo de manera remota, el objeto abstraído si será serializado, por lo tanto si ***turnActive*** es aplicado en un nodo remoto, la clase usada para crear el objeto activo de esta forma tiene que ser *Serializable.* En suma, cuando se usa ***turnActive*** se debe de tener mucho cuidado de que no sean guardadas las referencias al objeto original después de la llamada a ***turnActive***. Una llamada

directa al método del objeto original, sin pasar por un stub de ***ProActive*** en este objeto rompería el modelo.

Código para la creación basada en objetos

```
Object[] params = new Object[] {"astring"};
try
{
  ActivoO a;
  a = new ActivoO("astring");
  a=(ActivoO)org.objectweb.proactive.ProActive.turnActive(a);
} catch(org.objectweb.proactive.ActiveObjectCreationException e)
        {
          System.err.println("Error del constructor: " + e.getMessage());
          e.printStackTrace();
        }
         catch(org.objectweb.proactive.core.node.NodeException e)
                {
                 e.printStackTrace();
                }
```

Tal como ***newActive,*** el segundo parámetro de ***turnActive*** sí se da, es la posición del objeto activo a ser creado, sin parámetros o nulo significa que el objeto activo será creado localmente en el nodo actual. Cuando se esté usando este método, el programador se debe asegurar que no exista ninguna otra referencia en el objeto pasivo ***a*** después de la llamada a ***turnActive***. Si tales referencias fueran usadas para la llamada de métodos de manera directa en el objeto pasivo ***A*** (sin ir a través del cuerpo), el modelo podría no ser consistente y por lo tanto no se podría garantizar la sincronización.

II.4.2 Especificando la actividad de un objeto activo

Personalizar la actividad del objeto activo es el núcleo de ProActive debido a que éste permite especificar completamente el comportamiento de un objeto activo, por defecto un objeto que se convierte en un objeto activo sirve las peticiones entrantes de manera FIFO. Para especificar alguna otra política de servir las peticiones o para especificar cualquier otro comportamiento se pueden implementar interfaces definiendo métodos que serán llamados automáticamente por ProActive. Es posible especificar que hacer antes de que la actividad inicie, y que hacer al final de la misma. Los pasos son tres:

- la inicialización de la actividad (se hace solamente una vez)
- la actividad en sí misma
- el final de la actividad (se hace solamente una vez)

tres interfaces son usadas para definir e implementar cada paso:

InitActive
RunActive
EndActive

En caso de una migración, un objeto activo se detiene y reinicia su actividad automáticamente sin invocar las fases de inicialización o finalización. Solo la actividad es reinicializada por sí misma.
Hay dos formas posibles para definir cada una de las tres fases de un objeto activo

- Implementando uno o más de las tres interfaces directamente en la clase usada para crear el objeto activo.
- Pasando un objeto, implementando uno o más de las tres interfaces en el parámetro al método **newActive** o **turnActive** (parámetro activo en estos métodos)

Note que los métodos definidos por estas tres interfaces, garantizan que serán invocadas por el hilo activo del objeto activo.

Algoritmos para decidir que actividad invocar:
Los algoritmos que deciden para cada fase que hacer son los siguientes:
*Nota : **activity** es el objeto eventual pasado como parámetro a **newActive** o **turnActive***

InitActive
Si la actividad es no nula e implementa **InitActive** entonces
se invoca el método **InitActivity** definido en la actividad del objeto
si no
si la clase del objeto abstraído implementa **InitActive** entonces
se invoca el método **InitActivity** del objeto abstraído
si no
no se realiza ninguna inicialización

RunActive
Si la actividad es no nula e implementa **RunActive** entonces
se invoca el método **runActivity** definido en la actividad del objeto
si no
si la clase del objeto referenciado implementa **RunActive** entonces
se invoca el método **runActivity** del objeto referenciado
si no
se ejecuta la actividad estándar FIFO

EndActive
si la actividad es no nula e implementa **EndActive** entonces
invocamos el método **endActivity** definido en el objeto activity
si no
si la clase del objeto implementa **EndActive** entonces
se invoca el método **endActivity** del objeto abstraido
si no
no se realiza ninguna limpieza.

La solución más fácil, cuando se controla la clase que se va hacer activa, es implementar las interfaces directamente en la clase usada para crear el objeto

activo. Dependiendo en que fase de la vida del objeto activo se requiera personalizar, se implementa la correspondiente interfaz (una o más) entre ***InitActive***, ***RunActive*** y ***EndActive***.

Un ejemplo que tiene una inicialización y una actividad personalizada puede ser vista en la Figura II.3.

```
import org.objectweb.proactive.*;
public class A implements InitActive, RunActive
{
   private String myName;
 public String getName() {  return myName;}
//--Implementación de InitActive
public void initActivity(Body body) {myName = body.getName();}
//--Implementación de RunActive para servir peticiones en orden FIFO
public void runActivity(Body body) {
    Service service = new Service(Body);
    while (body.isActive()) { sService.blokingServeYoungest(); }
}
public static void main (string {} args) throws Exception {
 A a = (A) ProActive.newActive("A",null);
 System.out.println("Name = "+a.getName());
}
}
```

Figura II.3 Programa que inicializa y personaliza la actividad

Pasando un objeto implementando las interfaces cuando se crea un objeto activo
Esta es una solución para usarse cuando no se tiene un control de la clase que se va hacer activa o cuando se requiere escribir políticas de actividades genéricas y reutilizarlas con diversos objetos activos. Dependiendo en que fase de la vida del objeto activo se requiera personalizar, se implementa la correspondiente interfaz (una o más) entre **InitActive**, **RunActive** y **EndActive**. En la Figura II.4 se muestra cómo se debe personalizar la actividad.

Comparando la solución anterior donde las interfaces son directamente implementadas en la clase materializada, (al proceso de convertir un objeto abstracto aun objeto activo se le denomina materialización, por lo cual a partir de este punto, se hará referencia del término objeto materializado y clase materializada) hay una restricción aquí: no se puede acceder el estado interno del objeto abstraído. Usando un objeto externo debería por lo tanto ser usado cuando la implementación de la actividad es suficientemente genérica para no tener acceso a las variables miembro del objeto materializado.

```
import org.objectweb.proactive.*;
public class LIFOActivity implements RunActive {
//Implementa RunActive para servir peticiones en orden LIFO
  public void runActivity(Body body) {  Service service = new Service(Body);
                                          while (body.isActive())
                                                  {service.blockingServeYoungest();}
                                       }
}
 import org.objectweb.proactive.*;
public class A implements InitActive {
   private String myName;

 public String getName() { return myName; }

//Implemetación de InitActive
public void initactivity(Body body) { myName = body.getName(); }

public static void main(String{} args) throws Exception {
//newActive (classname, constructor parameter (null=none), node
                ( null = local), active, MetaObjectFactory (null = default
                 A a = (A) ProActive.newActive("A", null, null, new
                 Lifoactivity(), null);
                System.out.println("Name = "+a.getName()
               );
}
}
```

Figura II.4 Personalizar la actividad

II.4.3 Restricciones en los objetos abstraídos

No todas las clases pueden generar objetos activos. Existen algunas restricciones, la mayoría de ellas son causadas por la compatibilidad con java al 100% , la cual prohíbe modificar la MVJ o el compilador.
Algunas de estas restricciones trabajan a nivel de clase por ejemplo:

- Las clases **Final** no pueden generar objetos activos
- Las clases no públicas también no pueden generar objetos activos

- Las clases sin un constructor sin argumento no pueden ser abstraídas. Esta restricción será suavizada en una versión reciente de ProActive.
  Algunas otras restricciones pasan a nivel de un método en una clase específica por ejemplo:
    - Los métodos **Final** no pueden ser usados del todo. Cuando se invoca un método final en un objeto activo pueden llevar a un comportamiento inconsistente.
    - Invocando un método no público en un objeto activo levanta una excepción. Esta restricción desapareció con el JDK 1.2

II.4.4 Patrón de diseño (método de fábrica)

La creación de un objeto activo usando ProActive puede ser un poco incómodo y requerir mas líneas de código que para crear un objeto regular. Una buena solución a este problema es a través del uso del patrón de diseño denominado **factory**. Este se aplica principalmente a la creación basada en clases. Este consiste en adicionar un método estático a la clase **AA** que cuida de instanciar el objeto activo y regresarlo. El código puede ser visto en la Figura II.5

```
public class AA extends Activo
{
 public static Activo createActiveActivo(int i, String s,
org.objectweb.proactive.core.node.Node node)
 { Object[] params = new Object[] {new Integer (i),s};
   try {
       return(Activo)org.objectweb.proactive.ProActive.newActive("Factory.Activo",
      params,node);
       } catch(Exception e)  { System.err.println("la creacion de una instancia de A
                                  a levantado una  excepcion:" + e.getMessage());
                                 return null;
                               }
}
```

Figura II.5 Método fatory

Es responsabilidad del programador decidir si este método tiene que lanzar excepciones o no. es recomendable que este método solamente maneje las excepciones que aparecen en la firma del constructor abstraído, pero las excepciones no funcionales inducidas por la creación del objeto activo tienen que ser repartidas en algún lugar del código.

II.4.5 Personalizando el cuerpo del objeto activo

Hay muchos casos donde se puede desear personalizar el cuerpo cuando sé esta creando un objeto activo. El cuerpo no es un objeto cambiable que delege la mayoría de sus tareas a objetos que lo apoyan, llamados metaobjetos. Los

metaobjetos estándar son usados ya po r defecto en ProActive pero uno puede fácilmente reemplazar cualquiera de estos metaobjetos por uno personalizado.

Se ha definido la interfase ***MetaObjectFactory*** capaz de crear las fábricas para cada uno de esos metaobjetos Esta interfase esta implementada por ***ProActiveMetaObjectFactory*** que proporciona todas las fábricas por defecto usadas en ProActive.
Cuando se crea un objeto activo como se mencionó anteriormente, es posible especificar que ***MetaObjectFactory*** será usado para cada instancia en particular del objeto activo que esta siendo creado. La clase ProActive proporciona métodos extras ***newActive*** y ***turnActive*** para esto.

Primero se debe escribir una nueva fábrica ***MetaObject*** que herede de ***ProActiveMetaObjectFactory*** o implementar directamente ***MetaObjectFactory*** para redefinir cada cosa. Heredando de ***ProActiveMetaObjectFactory*** se ahorra tiempo de tal manera que solo se tendría que redefinir lo que realmente se necesita. Un ejemplo se muestra en la Figura II.6

```
public class MyMetaobjectFactory extends ProActiveMetaObjectFactory
{
 private static final MetaObjectfactory instance=new MyMetaobjectFactory();
 protected MyMetaobjectFactory(){ super(); }
 public static MyMetaobjectFactory newInstance()  { return instance; }
//Metodos protegidos
protected RequestFactory newRequestFactorySingleton()
{ return new myRequestFactory();}
//Clase Interior
protected class MyMetaobjectFactory implements RequestFactory,java.io. Serializable
{
        public Request newRequest (MethodCall
        methodCall,UniversalBody sourceBody, boolean isOneWay,
        longsequenceID) {
                return new MyRequest(methodCall,sourceBody,
                isOneWay,sequenceId,server);
               }
    }// fin de las clases interiores
}
```

Figura II.6 Redefine el Requestfactory

La fábrica anterior simplemente redefine el ***Requestfactory*** para hacer que el cuerpo utilice un nuevo tipo de petición. El método protegido ***RequestFactory newRequestFactorySingleton()*** es un método conveniente de ***ProactiveMetaObjectFactory*** para simplificar la creación de las fábricas como singlenton.
El uso de esta nueva fábrica es bastante simple. Todo lo que se tiene que hacer es pasar una instancia de la fábrica cuando está creando un nuevo OA, si se retoma el ejemplo anterior se tiene el código de la Figura II.7.

```
Object{} params = new Object{} {new Integer (26), "astring"};
 try {   A a = (A) ProActive.newActive("example.AA", params, null,
         null,MyMetaObjectFactory .newInstance());
      }
    catch (Exception e) {  e.printStackTrace(); }
```

Figura II.7 Pasar una instancia de factorycuando se crea un nuevo OA

En el caso de un ***turnActive*** se debería tener:

```
A a = new A(26, "astring");
a = (A) P`roActive.turnActive(a,null,null,MyMetaObjectFactory.newInstance());
```

II.4.6 El rol de los componentes de un objeto activo

En esta sección se tendrá una visión cercana de lo que sucede cuando un objeto activo es creado, proporcionando un mejor entendimiento de cómo trabaja la biblioteca y de donde vienen las restricciones de ProActive.

Considerando que para algún código en una instancia de la clase ***A*****,** se crea un objeto activo de la clase **B,** usando una pieza de código como:

```
 B b;
Object{} params = {};
try {
      //se crea una instancia activa de B en el nodo actual
      b = (B) ProActive.new.Active ("B",params);
    }
   catch (Exception e) {
                          e.printStackTrace();
                        }
```

Si la creación de una instancia activa de ***B*****,** es satisfactoria, el grafo de objetos es como se describe en la Figura II.8 (con flechas que denotan las referencias).

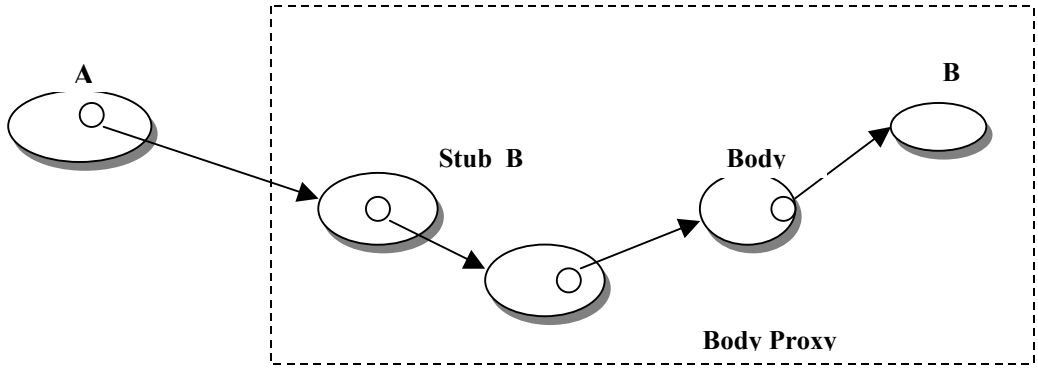


Figura II.8 Componentes de un Objeto Activo

La instancia activa de B está compuesta por 4 objetos:

- Un Stub (Stub_B)
- Un proxy (BodyProxy)
- Un cuerpo (Body)
- Una instancia de B

El rol de la clase ***Stub_B*** es abstraer todas las llamadas al método que puedan ser ejecutadas a través de una referencia de tipo ***B*** y solamente éstas como llamadas a métodos declarados en una subclase de ***B*****,** por downcasting causarían un error en tiempo de ejecución. Abstraer una llamada simplemente significa construir un objeto (en este caso, todas las llamadas abstraídas son instancias de la clase ***MethodCall***) que representa la llamada, de tal manera que ésta puede ser manipulada como cualquier otro objeto, entonces esta llamada abstraída es procesada por los otros componentes del objeto activo para obtener el comportamiento que esperamos del mismo.

La idea de usar un objeto estándar para representar elementos del lenguaje que normalmente no son objetos (tales como llamadas de métodos, llamadas del constructor, referencias, tipos, etc.), es que todo tiene que ver con la *programación metaobjecto*.

Como uno de los objetivos es proporcionar objetos activos transparentes, las referencias a un OA de la clase ***B*** necesitan ser del mismo tipo que las referencias a instancias pasivas de ***B*** (esta característica es llamada *polimorfismo* entre instancias pasivas y activas de la misma clase). Esto es, por construcción, ***Stub_B*** es una subclase de la clase ***B***, permitiendo por lo tanto que instancias de la clase ***Stub_B*** sean asignadas a variables de tipo ***B***.
La clase Stub_B redefine cada uno de los métodos heredados de su superclase. El código para cada método de la clase ***Stub_B*** construye una instancia de la clase ***MethodCall*** para representar la llamada a este método. Este objeto es entonces pasado al ***BodyProxy***, el cual devuelve un objeto que fue devuelto como el resultado de la llamada al método.
Desde el punto de vista del invocador, cada cosa luce tal como si la llamada hubiera sido ejecutada en una instancia de B.

Ahora que se sabe como trabajan los stubs, se debe entender algunas de las limitaciones de ProActive:

- Obviamente, ***Stub_B*** no puede redefinir los métodos de tipo ***final*** heredados de la clase ***B***. Por lo tanto, las llamadas a esos métodos no son abstraídas sino más bien ejecutadas en el ***stub***, lo cual puede llevar a comportamientos inexplicables si el programador no evita cuidadosamente el llamado a los métodos ***final*** en objetos activos.

Debido a que hay seis métodos finales en la clase base ***Object***, se puede imaginar como vivir sin ellos, en efecto 5 de estos 6 métodos tienen que lidiar con la sincronización entre hilos (***notify()***, ***notifyAll()*** y 3 versiones de ***wait()***). Estos métodos no deberían ser usados dado que un OA proporciona sincronización de hilos.
Usando el mecanismo de sincronización hilos estándar y el mecanismo de sincronización de hilos de ProActive al mismo tiempo podría traer conflictos y como consecuencia y una pesadilla para realizar un proceso de depuración.
El último método ***final*** en la clase Object es ***getClass()***. Cuando se invoca en un objeto activo ***getClass()*** no es abstraída y por lo tanto ejecutado en el objeto stub, el cual devuelve un objeto de clase ***Class*** que representa la clase del stub (***Stub_B*** en el ejemplo) y no de la clase del OA (***B*** es el ejemplo). Sin embargo este método raras veces es usado en aplicaciones estándar y no previene el operador ***instanceof*** para trabajar gracias a su comportamiento polimórfismo. Por lo tanto la expresión (***foo instanceof B***) tiene el mismo valor cuando ***B*** es un objeto activo y cuando no lo es.

- La obtención y la puesta de variables de instancias de manera directa (no a través de un método que lo obtenga o lo coloque) deberían ser evitado en el caso de los OA debido a que éste podría resultar en obtener o colocar un valor en el objeto stub y no en la instancia de la clase **B**. Este problema es normalmente atacado usando los métodos ***get/set*** para colocar o leer atributos. Esta regla estricta de encapsulación, puede ser encontrada en JavaBeans o en los sistemas distribuidos de objetos RMI o Corba.

El rol del proxy: El rol del proxy es manejar el asincronismo en llamadas a objetos activos. Mas específicamente, éste crea objetos futuros si van a ser posiblemente necesarios, llama mas adelante a cuerpos y objetos futuros para los stubs. Como esta clase opera en objetos ***MethodCall***, éste es absolutamente genérico y no depende de todos en el tipo del stub que alimenta las llamadas a través de su método ***reify*.**

El rol del cuerpo: El cuerpo es responsable de almacenar las llamadas (actualmente objetos **Request**) en una cola de peticiones pendientes y procesarlas de acuerdo a una política de sincronización dada, cuyo comportamiento por defecto es el FIFO. El cuerpo tiene su propio hilo, que alternativamente selecciona una petición en la cola de peticiones pendientes y ejecuta la llamada asociada.

El rol de la instancia de la clase B: Esta es una instancia estándar de la clase ***B***. Puede contener alguna información de sincronización en su método ***live*** si es que hubiera alguna. Como el cuerpo ejecuta llamadas una por una, no puede haber ninguna ejecución concurrente de dos porciones de código de este objeto por dos hilos diferentes.

Esto da la posibilidad de usar pre y post condiciones y invariantes de clase. Como una consecuencia, el uso de la palabra reservada ***synchronized*** en la clase ***B*** no debería ser necesario. Cualquier esquema de sincronización que pueda ser expresada a través de monitores y sentencias ***synchronized*** pueden ser expresadas usando mecanismos de sincronización de alto nivel de ProActive de una manera más natural y amigable.

II.5 Objetos futuros y llamadas asíncronas

II.5.1 Creación de un objeto futuro

Siempre que sea posible, una llamada a un método de un objeto activo, esté es materializado como una petición asíncrona. Si esto no es posible, la llamada es síncrona y se bloquea hasta que la respuesta sea recibida. En caso de que la petición sea asíncrona, ésta inmediatamente devuelve un objeto futuro (OF). Este objeto futuro actúa como un espacio de almacenamiento para el resultado de la invocación al método que aún no concluye. Como una consecuencia, el hilo invocador puede continuar ejecutando su código, mientras esto sucede no tiene que invocar métodos sobre el objeto devuelto, en tal caso el hilo invocante es automáticamente bloqueado sí el resultado de la invocación al método aún no está disponible.
En la sig. Tabla II.1 se muestran los diferentes casos en que se puede permitir una llamada asíncrona.

Tipo devuelto	Pasa una excepción	Creación de un objeto futuro	Asíncrono
Void	-	No	Si
Non Reifiable Object	-	No	No
Reifiable Object	Si	No	No
Reifiable Object	No	Si	Si

Tabla II.1 Llamada asíncrona

La creación de un objeto futuro no solo depende del tipo del invocador, sino también del tipo del objeto devuelto. La creación de un objeto futuro solo es posible si el objeto es abstraible. Note que aunque éste tiene una estructura bastante similar a un objeto activo, un OF no es activo. Este solo tiene un Stub y un Proxy como se muestra en la Figura II.9.

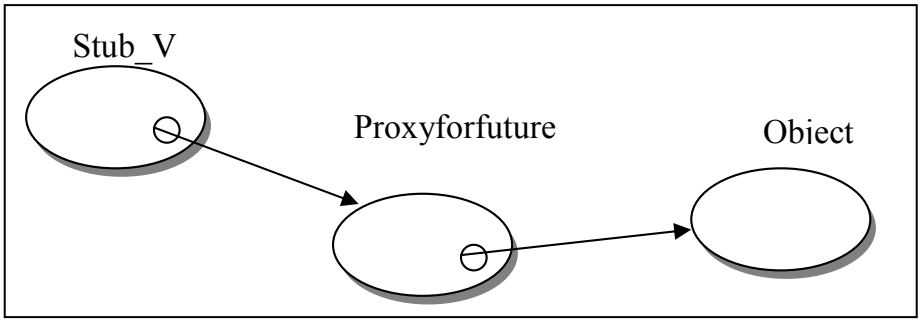


Figura II.9 Composición de un objeto futuro

Durante su tiempo de vida, un objeto activo puede crear muchos objetos futuros, estos se guardan automáticamente en un espacio de almacenamiento para objetos futuros (FuturePool).
Cada vez que un objeto futuro es creado, se inserta en el espacio de almacenamiento para OF del objeto activo correspondiente. Cuando el resultado llega a estar disponible, el OF es removido del espacio de almacenamiento. Aunque la mayoría de los métodos del FuturePool son solo para uso interno y son llamados directamente por la biblioteca proactive, se proporciona un método para que espere hasta que el resultado esté disponible. En lugar de bloquear hasta que un objeto futuro específico esté disponible, la llamada a ***waitForReply()*****,** se bloquea hasta que cualquiera de los objetos futuros actuales lleguen a estar disponibles.
Cualquier llamada a un OF es abstraída para ser bloqueada si el objeto futuro no está aún disponible y después ejecutada sobre el objeto resultado. Sin embargo, existen dos métodos que no siguen el mismo esquema: Equals y HashCode. Ellos a menudo son llamados por otros métodos desde la biblioteca de Java, como HashTable.add() y así están mas fuera del control del usuario la mayoría de las veces. Esto puede conducir fácilmente a candados mortales si ellos son llamados en un objeto que no está aún disponible.
En lugar de regresar el código hash del objeto, *hashCode()* devuelve el código hash de su proxy. Debido a que hay solo un proxy por OF, hay una única equivalencia entre ellos.
La implementación por defecto de equals() en la clase Object es para comparar las referencias en dos objetos. En ProActive, éste es redefinido para comparar el código hash de dos proxys. Como consecuencia, solamente es posible comparar dos objetos futuros, y no un OF con un objeto normal.
Hay algunas desventajas con esta técnica, la principal es la imposibilidad de que un usuario anule los métodos por defecto de ***hashcode()*** y ***equals().***
toString(): El método toString es la mayoría de las veces llamado por ***System.out.println()*** para cambiar un objeto a una cadena imprimible. En la implementación actual, una llamada a este método bloqueará al OF con cualquier otra llamada, así que hay que tener cuidado en su uso. Como ejemplo, el intentar imprimir un OF con el propósito de depuración, la mayoría de las veces conducirá a un candado mortal. En lugar de desplegar la cadena correspondiente de un OF, se debería considerar desplegar su código hash.

II.5.2 Llamadas asíncronas en detalle

Para poder entender las llamadas asíncronas describiremos un ejemplo en particular. Algunas piezas de código en una instancia de la clase ***A*** llama al

método ***foo*** en una instancia activa de la clase ***B***. Esta llamada es asíncrona y devuelve un objeto futuro de la clase **V**. Entonces, posiblemente después de haber ejecutado algún otro código, el mismo hilo que emitió la llamada llama al método bar en el OF devuelto por la llamada a ***foo***.
En una versión secuencial de la misma aplicación con un solo hilo, éste podría haber ejecutado el código del método llamado en la clase ***A*** antes, de la llamada a **foo,** después el código de ***foo*** en la clase ***B***, y posteriormente regresar al código del método invocado en la clase **A** antes de la llamada a ***bar***, enseguida el código de ***bar*** en la clase ***V*** y finalmente regresar al código del método invocado en la clase ***A*** hasta que éste finalice. El diagrama de secuencia que muestra la Figura II.10 resume esta ejecución, notese como un solo hilo sucesivamente ejecuta código de diferentes métodos en diferentes clases.

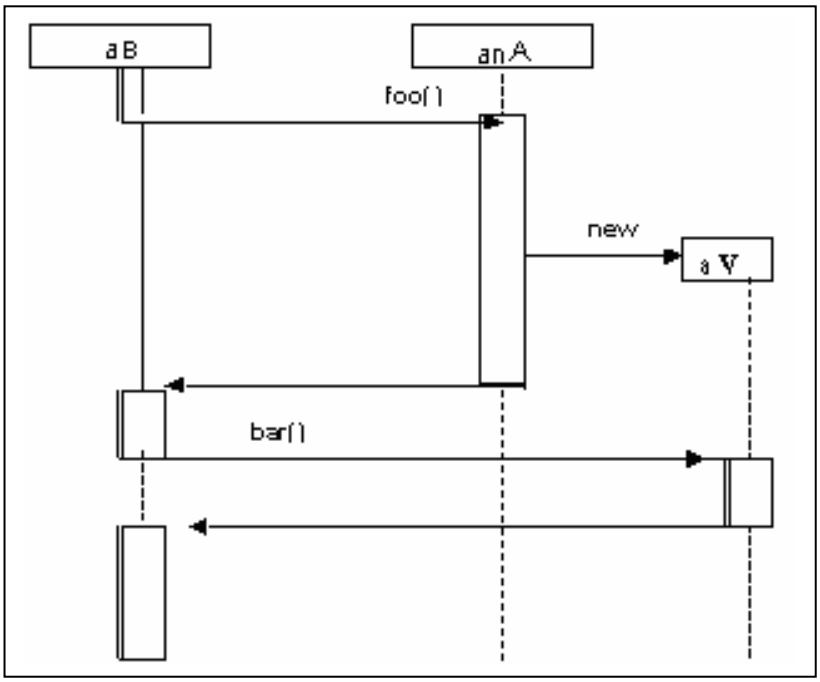


Figura II.10 Diagrama de secuencia versión de un solo hilo del programa

A continuación se obtiene una idea inicial de cómo luce el grafo de la ejecución de los objetos en tres momentos diferentes de la ejecución (los objetos con sus respectivas referencias):

- Antes de llamar a ***foo,*** se tiene exactamente la misma configuración como después de la creación de la instancia activa de ***B***, tal y como se muestra en la Figura II.11 una instancia de la clase ***A*** y una instancia activa de la clase ***B***. Como todos los OA, la instancia de la clase ***B*** esta compuesta de un esqueleto (una instancia de la clase ***Stub_B***, la cual hereda directamente de ***B***), un ***BodyProxy***, un ***Body*** y la instancia actual de **B**.

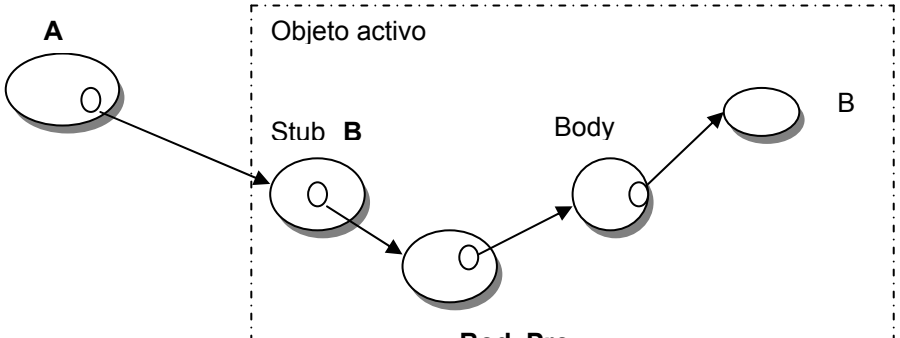


Figura II.11 Los componentes de un objeto activo

- Después de que la llamada asíncrona a **foo** ha regresado, **A** ahora contiene una referencia en un OF que representa el resultado, aun no disponible, de la llamada. Esto esta compuesto de un **Stub_V** y un **FutureProxy** como muestra la Figura II.12.

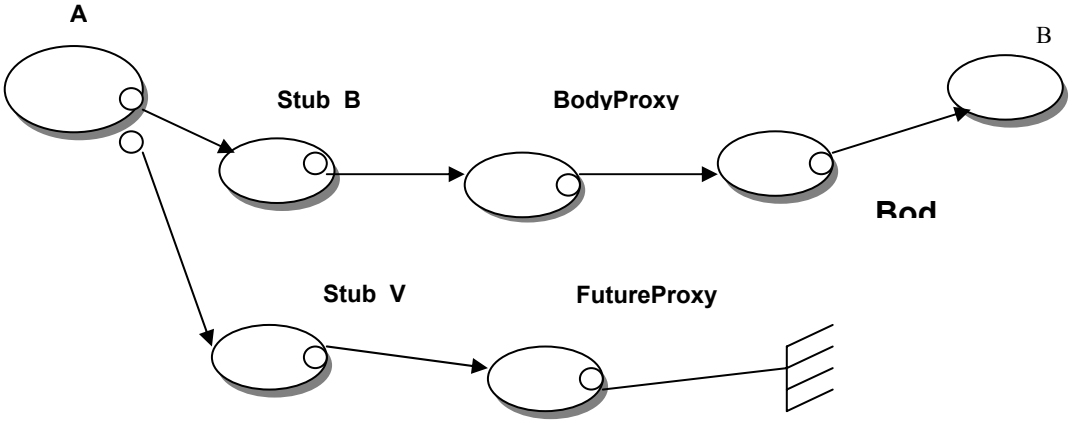


Figura II.12 Los componentes de un objeto futuro antes del resultado

- Justo después de haber ejecutado ***foo*** en una instancia de ***B***, el hilo del ***Body*** pone el resultado en el futuro, que causa que en ***FutureProxy*** tenga una referencia dentro de **V** ver Figura II.13.

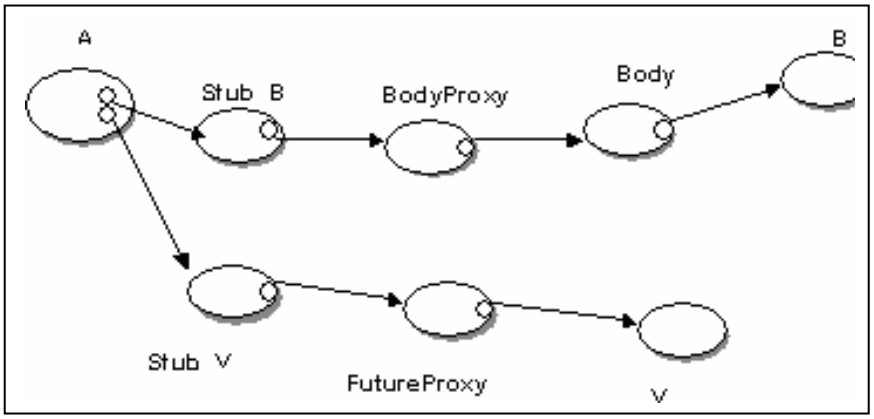


Figura II.13 Los componentes de un objeto futuro

Ahora se concentra en como diagrama de secuencia, cuando y cuales hilos de los diferentes métodos son llamados. Tenemos dos hilos: uno que pertenece al subsistema ***A*** es parte de (primer hilo) y el hilo que pertenece al subsistema ***B*** es parte de (segundo hilo).

El primer hilo invoca ***foo*** en una instancia de ***Stub_B***, el cual construye un objeto ***MethodCall*** y lo pasa a ***BodyProxy*** como parámetro de la llamada a ***reify***. El proxy entonces checa el tipo que devuelve la llamada (en este caso ***V***) y genera un objeto futuro de tipo **V** para representar el resultado de la invocación al método. El objeto futuro está compuesto de un ***Stub_V*** y un ***FutureProxy***. Una referencia dentro de este objeto futuro es puesta en el objeto **MethodCall**, el cual será útil una vez que la llamada es ejecutada. Ahora que el objeto ***MethodCall*** esta listo, este se pasa como una petición al ***Body*** del OA como parámetro. El cuerpo simplemente añade esta petición a la cola de peticiones pendientes y regresa inmediatamente. La llamada a ***foo*** que ***A*** emitió ahora regresa un OF de tipo ***Stub_V***, que es una subclase de V.
En algún punto, posiblemente después de haber servido algunas otras peticiones, el segundo hilo (el hilo activo) recoge la petición emitida, tiempo atrás por el primer hilo. Éste entonces ejecuta una llamada empotrada invocando a ***foo*** en la instancia de **B** con los parámetros actuales almacenados en el objeto ***MethodCall***. Como está especificado en su firma, ésta llamada regresa un objeto de tipo **V**. El segundo hilo es responsable de poner este objeto en un OF (esta es la razón del por que el objeto ***MethodCall*** contiene una referencia en él OF creado por el ***FutureProxy***). La ejecución de esta llamada se termina y el segundo hilo puede seleccionar otra petición de la cola y ejecutarla.

Mientras tanto, el primer hilo ha continuado ejecutando el código del método invocado en la clase ***A***. En algún punto, éste llama a ***bar*** en el objeto de tipo ***Stub_V*** que fue devuelto por la llamada a ***foo***. Esta llamada es abstraída gracias a ***Stub_V*** y procesada por el ***FutureProxy***. Si el objeto que representa el futuro esta disponible (el segundo hilo ha sido puesto en el objeto futuro, lo cual es descrito en la Figura II.14), la llamada es ejecutada sobre éste y devuelve un valor al código invocado en ***A***.

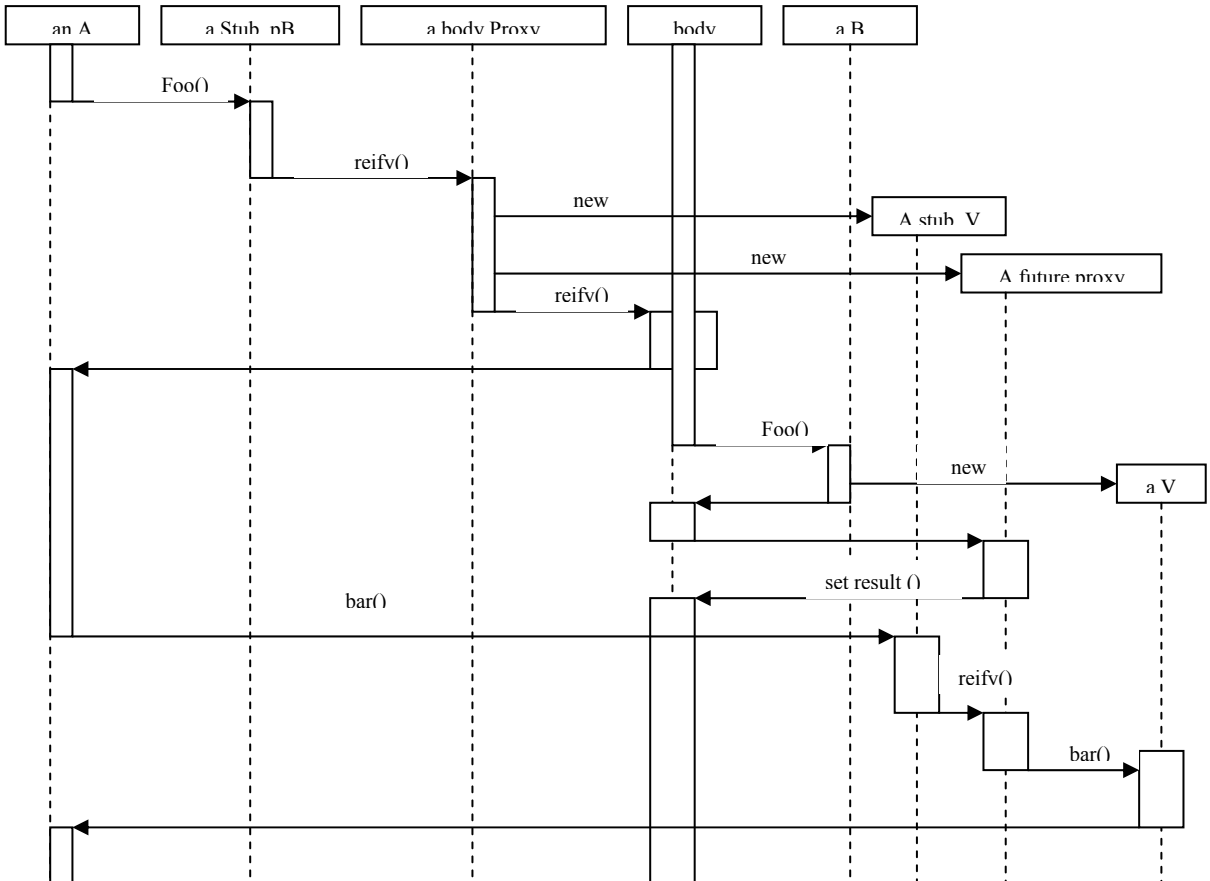


Figura II.14 Diagrama de secuencia

Si éste no esta disponible aún, el primer hilo se suspende en **FutureProxy** hasta que el segundo hilo coloque el resultado en él OF ver Figura II.15.

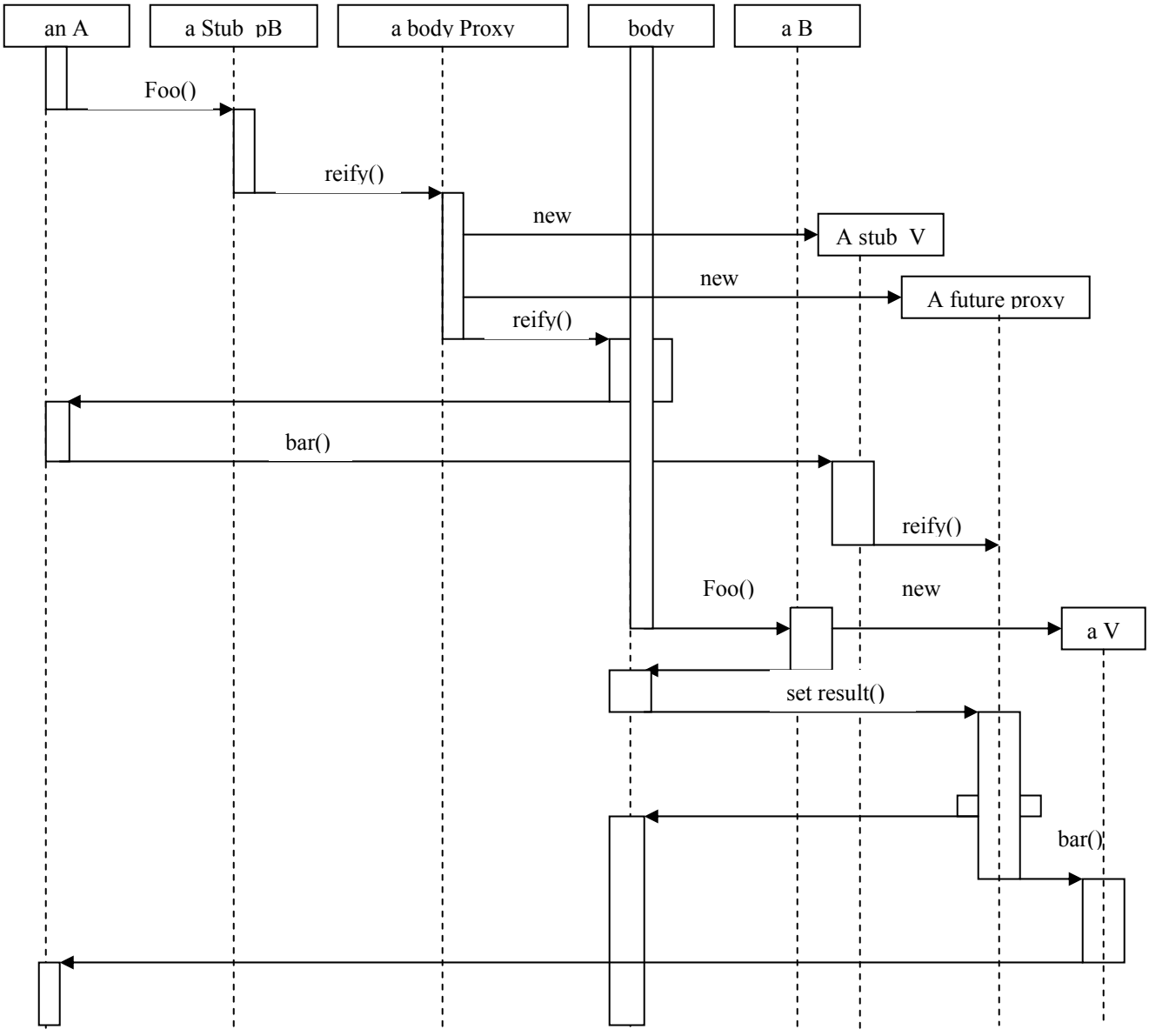


Figura II.15 Segundo diagrama de secuencia

II.6 Protocolo Metaobjeto

ProActive está construido en la cima del protocolo metaobjeto (MOP), que permite abstracción de la invocación a los métodos y de las llamadas al constructor. El MOP no esta limitado a la implementación de la biblioteca de objetos remotos transparentes, si no que éste además proporciona un ambiente de trabajo abierto para implementar poderosas bibliotecas para el lenguaje Java.
Como cualquier elemento de ProActive, el MOP está totalmente escrito en Java y no requiere modificación alguna o extensión de la maquina virtual de Java, de manera opuesta a otros protocolos metaobjeto para Java. Éste hace extensivo el uso de la API Java Reflection, el cual requiere de JDK 1.1 o mayor. Se requiere JDK1.2 para suprimir las verificaciones de control de acceso por defecto del lenguaje Java cuando se ejecuta un método no publico abstraído o llamadas al constructor.

II.6.1 Principios

Si el programador desea implementar un nuevo metacomportamiento usando el MOP, debe de escribir una clase concreta (como opuesto a lo abstracto) y una interfase. La clase concreta proporciona una implementación para el metacomportamiento que se desea obtener mientras que la interfase contiene su parte declarativa.
La clase concreta implementa la interfase Proxy y proporciona una implementación para el comportamiento dado a través del método reify:

puclic Object reify (MethodCall c) throws Throwable;

Este método toma una llamada abstraída como un parámetro y regresa el valor obtenido de la ejecución de esta llamada abstraída. Se proporciona un empaquetamiento y desempaquetamiento automático de tipos de primitivas. Si la ejecución de la llamada termina abruptamente por el lanzamiento de una excepción, ésta es propagada al método invocador, tal y como si la llamada no hubiese sido abstraída.

La interfase que contiene la parte declarativa del metacomportamiento tiene que ser una subinterface de ***Reflect*** (la interfase raíz de todos los metacomportamientos implementados usando ProActive). El propósito de esta interfase es declarar el nombre de la clase ***proxy*** que implementa el comportamiento dado. Entonces cualquier instancia de una clase que tenga implementada esta interfase será creada automáticamente con un ***proxy*** que implementa este comportamiento, considerando que esta instancia no es creada usando la palabra clave ***new*** sino a través de un método estático: ***MOP.newInstance***. Esta es la única modificación que se requiere para el código de la aplicación. Otro método estático ***MOP.newWrapper***, agrega un *proxy* a un

objeto ya existente; la función de ProActive ***turnActive***, por ejemplo esta implementada a través de esta característica.

Ejemplo de un comportamiento diferente: EchoProxy

Se muestra la implementación de un metacomportamiento simple pero útil: para cada llamada abstraída, el nombre del método invocado es impreso en el flujo de salida estándar y la llamada es entonces ejecutada. Esto puede ser un punto de inicio para la construcción de ambientes de depuración o de perfiles.

```
class EchoProxy extends Object implements Proxy {
// Aquí el constructor y las variables de declaración
// {...}
public Object reify (MethodCall c) throws Throwable {
System.out.println (c.getMethodName());
Return c.execute (targetObject);
}
}

interfase Echo extends Reflect {
public String PROXY_CLASS= "EchoProxy";
}
```

II.6.2 Instanciación con el metacomportamiento

Instanciar un objeto de cualquier clase con este metacomportamiento se puede hacer de tres formas diferentes:

- basado en instanciación
- basado en clase
- basado en objeto

Suponga que se desea instanciar un objeto Vector con un comportamiento *Echo:*

- Con código Java estándar debería ser:
  Vector v = new Vector(3);

- Con código ProActive, la declaración del metacomportamiento basado en instanciación (él ultimo parámetro es nulo por que no se tiene otro parámetro adicional para pasar al proxy) es:
  Object[] params = {new Integer (3) };
  Vector v = (Vector) MOP.newInstance("Vector", params, "EchoProxy", null);

- Con declaración basada en clase
  public class MyVector extends Vector implements Echo {}
  Object[] params = {new Integer (3)};
  Vector v = (Vector) MOP.newInstance("Vector", params, null);

- Con declaración basado en objeto
  Vector v = new Vector (3);
  v=(Vector) MOP.newWraper("EchoProxy",v);

Este el único modo de dar un metacomportamiento a un objeto que es creado en un lugar donde no se puede editar el código fuente. Un ejemplo típico puede ser un objeto devuelto por un método que es parte de una API distribuida como un archivo JAR, sin el código fuente. Note que cuando se usa ***newWrapper***, la invocación del constructor de la clase Vector no es abstraída.

II.6.3 La interfase Reflect

Todas las interfases usadas para declarar *metacomportamientos* heredan directa o indirectamente de Reflect. Esto da lugar a una jerarquía de metacomportamientos tal como se muestra en la Figura II.16.

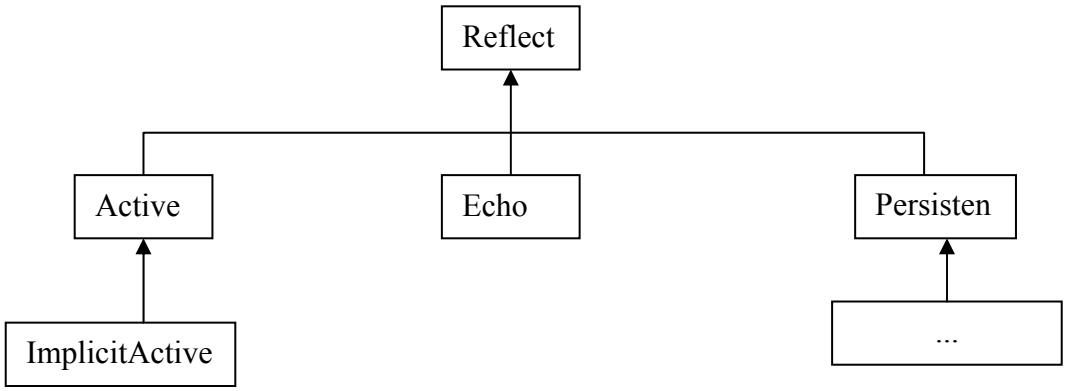


Figura II.16 Diagrama de interfase y subinterfases de Reflect

Note que ImplicitActive hereda de Active para resaltar el hecho de que la sincronización implícita siempre confía en algún mecanismo explicito oculto. Las interfaces heredan de Reflect, pueden entonces ser lógicamente agrupadas y ensambladas usando herencia múltiple para construir nuevos metacomportamientos de los ya existentes.
Debido a que la librería tiene el compromiso de ser 100 % biblioteca de Java, el MOP tiene algunas limitaciones

- Las llamadas enviadas a instancias de clases final (las cuales incluyen todos los arreglos) no pueden ser abstraídas.
- Los tipos de primitivas no pueden ser abstraída por que ellos no son instancias de clases estándar
- Las clases Final no pueden ser abstraída.

II.7 Funcionalidades de ProActive

II.7.1 Sincronización

ProActive provee mecanismos de sincronización avanzados que permiten una implementación fácil y segura con políticas de sincronización potencialmente complejas.
Lo anteriormente mencionado se explicará más a detalle con dos ejemplos para mayor exactitud:

1) En el problema de los lectores y escritores para permitir la concurrencia mientras se asegura la consistencia de los lectores, estos tienen que estar sincronizados con una política específica; para acceder a los datos. Esto es posible gracias a ProActive, en donde los accesos son garantizados de manera secuencial. La implementación en *ProActive* usa 3 objetos activos, el primero llamado ***Reader***, el segundo ***Writer*** y por último la clase controladora (***ReaderWriter***).
2) En el problema de los filósofos, el cual es un ejercicio clásico en la enseñanza de la programación concurrente. El objetivo principal es evitar los bloqueos. En la solución ha este problema usando ProActive, todos los filósofos son objetos activos, así como el controlador y la interface ver Internet-10.

II.7.2 Migración de Objetos Activos

II.7.2.1 Usando migración

Cualquier objeto activo tiene la habilidad para migrar. Si éste referencia a algunos objetos pasivos, ellos también se migrarán a la nueva ubicación. Dado que se confía en la serialización para enviar el objeto sobre la red, el OA debe implementar la interfase serializable. Para migrar, un OA debe tener un método que contenga una llamada a la primitiva de migración. Esta llamada debe ser la última en el método, es decir, el método debe regresar inmediatamente. Lo siguiente es un ejemplo de un método en un objeto activo:

```
public void moveTo(String t) {
try {
       ProActive.migrateTo(t);
      }catch (Exception e) { e.printStackTrace(); }
 }
```

No se proporciona ninguna prueba para verificar si la llamada a ***migrateTo()*** es la última en el método, entonces si la regla no se cumple, esto puede generar un comportamiento inesperado. Ahora para hacer que este objeto se mueva, solo se debe llamar a su método moveTo() como se muestra en la Figura II.17

Ejemplo completo:

```
import org.objectWeb.proactive.ProActive;
public class SimpleAgent implements Serializable
 {
    public SimpleAgent() {}
    public void moveTo(String t)
   {
        try {
             ProActive.migrateTo(t);
            }catch (Exception e) {
            e.printStackTrace();}
  }
}
public String whereAreYou() {
    try {
         return InetAddress.getLocalHost().getHostName();
        }catch (Exception e) {   return "Busqueda del local host fallida";} }
public static void main (String[] args)
{
   if (!args.length>0)) {   System.out.println("Uso: java migration.test.TestSimple
                             hostname/NodeName");
                            System.exit(-1);
                          }
   SimpleAgent t = null;
   try { //crea un SimpleAgent en esta MVJ
        t= (SimpleAgent)
       ProActive.newActive("migration.test.SimpleAgent",null);
      }catch (Exception e) { e.printStackTrace();}
  //migran el SimpleAgent a la posición identificada por el nodo URL
  //se asume que el ya nodo existe
  t.moveTo(args[10]);
  System.out.println(El objeto activo esta ahora en el host " + t.whereAreYou());
 }
}
```

Figura II.17 Llamar a moveTo()

La clase ***SimpleAgent*** implementa ***Serializable***, de tal manera que los objetos creados serán capaces de migrar. Se necesita proporcionar un constructor vacío para evitar efectos colaterales durante la creación de OA. Este objeto tiene dos métodos, ***moveTo()*** el cual lo migra a la posición especificada y, ***whereAreYou()*** el cual regresa el nombre del anfitrion de la nueva ubicación del agente.

En el método principal (***main***), primero es necesario crear un OA, el cual se hace a través de la llamada a ***newActive()***. Una vez que se ha hecho esto, se llaman a los métodos en éste como en cualquier objeto. Se invoca su método ***moveTo()*** el cual migra al nodo especificado como parámetro y entonces pregunta cuál es su posición actual.

II.7.2.2 Primitiva de migración

La migración de un OA puede ser provocado por el propio OA o por un agente externo. En ambos casos una sola primitiva tarde o temprano será llamada para aplicar la migración. Este método ***migrateTo()*** es accesible desde un cuerpo migrable (un cuerpo que hereda de ***AbstractMigratableBody***).

Para facilitar el uso de la migración, se proporcionar dos conjuntos de métodos estáticos en la clase ProActive.

El primer conjunto: Pone atención a la migración provocada por el OA que desea migrar. Los métodos confían del hecho de que el hilo que invoca es el hilo activo del objeto activo.

- ***migrateTo(Object o)***: migran a la misma posición como un objeto activo existente.
- ***migrateTo(String nodeURL)***: migran a la posición dado por el URL del nodo.
- ***migrateTo (Node node)***: migran a la posición del nodo dado.

El segundo conjunto : Pone atención a la migración provocada desde otro agente en vez del OA destino. En este caso el agente externo debe tener una referencia al cuerpo del OA que quiere migrar.

- ***migrateTo(Body body, Object o, boolean priority)***: migran a la misma posición como un objeto activo existente.
- ***migrateTo(Body body, String nodeURL, boolean priority)***: migran a la posición dado por el URL del nodo
- ***migrateTo(Body body, Node node, boolean priority)***: migran a la posición del nodo dado.

II.7.2.3 Manejo de atributos no-serializables

La migración de un objeto activo usa la serialización. Desafortunadamente, no todos los objetos en el lenguaje Java son serializables. Se vera un método simple que trata con tales atributos, en el caso de que sus valores no necesiten ser conservados. Para casos más complejos, el lector puede ver las especificaciones de Java RMI.

Cuando una excepción NotSerializable es lanzada, el primer paso para resolver el problema es identificar la variable responsable, es decir la que no es serializable, entonces antes de la declaración de la variable, poner la palabra reservada transient, esto indica que el valor de esta variable no debería ser serializable. Después de la primera migración este campo es cambiado a nulo ya que el valor

no tiene que ser guardado. La variable se tiene que reconstruir bajo el arribo del OA en su nueva ubicación. Esto se puede hacer fácilmente proporcionando un método estándar en el OA.

private void readObject(java.io.ObjectInputStream in) throws
java.io.IOException, ClassNotFoundException;

II.8 Los Paquetes de ProActive

ProActive PDC cuenta con 32 paquetes en su versión 1.0.3, los cuales hacen posible la programación, paralela, concurrente y distribuida de forma transparente, de igual forma gracias a la herencias de sus clases es posible lograr migración, sincronización de procesos, etc. Más adelante se da una breve descripción de cada uno de los paquetes.

1) org.objectweb.proactive

Este paquete proporciona la clase ***Main*** para crear objetos activos y objetos futuros. La clase pivote es la clase *ProActive* que contiene los métodos para crear los objetos activos. Cuando un objeto activo es creado, éste proporciona el camino para atender las respuestas provenientes de un método vivo. En el método vivo el objeto activo recibirá su propio cuerpo listo como la interface del cuerpo que permite manejar todo el comportamiento no funcional.

La clase *Service* es una clase ayudante creada usando un cuerpo que puede ser usado en el método vivo para simplificar los servicios de los métodos

2) org.objectweb.proactive.core

Directamente en este paquete se encuentran algunas clases que no están especificadas en ninguna parte de ProActive pero usadas en muchos lugares.

Muchas de las clases dentro del paquete stub-packages de este paquete no están apuntadas para ser usadas directamente pero son utilizadas para el funcionamiento interno de PorActive.

3) org.objectweb.proactive.core.body.future

Proporciona la definición e implementación de los objetos futuros.

Las llamadas asíncronas son útiles para reforzar las actuaciones de aplicaciones cuidadosamente diseñadas. Esto permite la llamada de un método para realizar otras tareas mientras estas peticiones están siendo procesadas.

4) org.objectweb.proactive.core.body.jini

El cuerpo de un objeto activo actúa como un cubo de comunicaciones con otros objetos remotos. Por esta razón, como JINI es la tecnología que subraya la comodidad de ProActive, el cuerpo del objeto activo es una versión remota del cuerpo que usa remotamente la referencia.

5) org.objectweb.proactive.core.body.message

“Solo uso interno” Define la capa del mensaje común de petición/respuesta.

La comunicación en ProActive es basada en mensajes que se especializan en petición/respuesta. La producción de mensaje también puede monitoreada basada

en eventos. Este paquete define también un típico productor abstracto de mensajes capaz de escuchar registro para el monitoreo de eventos.

6) org.objectweb.proactive.core.body.migration

Define las primitivas de migración de ProActive. Cualquier objeto activo, tal como este gráfico de objetos pasivos es serializable, es potencialmente migrable. Este es el cuerpo asociado al objeto activo que hace posible la migración. El ***AbstractMigratableBody*** (***AbstractMigratableBody.html***) es un cuerpo abstracto usado para la migración. El cuerpo objeto migrable debe heredar desde éste. La migración efectiva es actualmente delegada a un ***MigrationManager (MigrationManager.html***) también definido en este paquete.

7) org.objectweb.proactive.core.body.reply

Define los mensajes de respuesta basado en la capa de mensajes común. Una respuesta es un mensaje representando el resultado de una llamada a método emitido a través de una petición. Se usa por el cuerpo y sus componentes para comunicar.

Además de los mensajes de respuesta, este paquete define dos componentes del cuerpo (***ReplySender*** and ***ReplyReceiver***) en la responsabilidad de enviar y recibir las contestaciones.

8) org.objectweb.proactive.core.body.request

Define los mensajes de petición basados en la capa de mensajes común. Una petición es un mensaje empotrando en una llamada a un método emitido en un objeto activo. Es usado por el cuerpo y por sus componentes para comunicarce. Todas las peticiones vienen a el cuerpo que los proceso, entonces una por una usa al hilo activo y activa una llamada en el objeto referenciado.

Todas las peticiones entrantes son almacenadas en ***RequestQueue*** agregada al cuerpo esperando a ser procesado. Es usado por la cola de peticiones que uno puede configurar con políticas de sincronización complejas en el método *live* de un objeto activo. Varias interfaces (***RequestFilter(RequestFilter.html***) o ***RequestProcessor(RequestProcessor.html)***) proporciona un buen control sobre la cola de peticiones. Además de los mensajes de petición, este paquete define dos componentes del cuerpo (***RequestSender y RequestReceiver***) encargados de enviar y recibir peticiones.

9) org.objectweb.proactive.core.body.rmi

El cuerpo del objeto activo actúa como un cubo de comunicaciones con otros objetos remotos. Por esta razón, como RMI es la tecnología que subraya la comodidad de ProActive, el es una versión remota del cuerpo que es usado para referenciar un cuerpo remotamente.

10) org.objectweb.proactive.core.descriptor.data

Proporciona todas las clases necesarias para crear objetos java relacionados al descriptor de desarrollo XML.

El descriptor XML es representado como un objeto java por ***ProActiveDescriptor***. Cuando se leen archivos xml, una nueva instancia es creada

ProActiveDescriptor. Tales etiquetas como ***virtualnode*** y ***jvm*** en el archivo ***xml*** son representados respectivamente por objetos ***VirtualNode*** y objetos ***VirtualMachine***.

11) org.objectweb.proactive.core.descriptor.xml
Proporciona todas las clases necesarias para analizar el archivo del descriptor de desarrollo ***XML***, y construyendo objetos java relacionado.

12) org.objectweb.proactive.core.event
El objeto activo creado con ProActive genera eventos en demanda cuando escuchador registra a un componente del objeto activo.
Este paquete define todo tipo de escuchadores y todo tipo de eventos usados en ProActive.

13) org.objectweb.proactive.core.jini
Proporciona clases auxiliares para usarse con ***Jini.***

14) org.objectweb.proactive.core.mop
"Sólo uso interno" Define el protocolo Meta Objeto usado en PorActive que permite la intercepción de una llamada a un método para un objeto activo y sus abstracciones como un petición enviada al cuerpo.
Este paquete es el genérico ***MOP*** que maneja la generación de objetos ***stub*** y la abstracción de llamadas al ***proxy*** asociado.

15) org.objectweb.proactive.core.process
Define servicios para generar procesos externos desde java. Desde la clase abstracta genera un proceso genérico, varias especializaciones son incluidas por otro separador de la ***VM*** con una clase destino y creando una VM remota en otro ***host***. La creación remota es echa haciendo ***RSH, RLOGIN, SSH, LSF*** o ***GLOBUS***.

16) org.objectweb.proactive.core.process.globus
"Bajo el desarrrollo" Define servicos para crear procesos remotos usando Globos. Principalmente usando el descriptor de desarrollo ***XML***.

17) org.objectweb.proactive.core.process.lsf
Define servicos para crear un proceso remoto usando ***RLOGIN*** o ***BSUB*** para un cluster usando el protocolo ***LSF.***

18) org.objectweb.proactive.core.process.rsh
Define servicios para crear un proceso remoto usando ***RSH***. Actualmente esto sólo es posible usando RSH desde Unix a Unix o desde Windows a Unix. Las dos partes extrañas (Windows/Unix a Windows) no trabaja por la falta de un servidor RSH estándar en el sistema Windows.

19) org.objectweb.proactive.core.process.ssh

Define servicios para crear un proceso remoto usando ***SSH***. Actualmente esto solo es posible para usar SSH desde Unix a Unix o desde Windows a Unix. Las dos partes perdidas (Windows/Unix a Windows) no trabajan por la falta de un servidor SSH estándar en el sistema Windows.

20) org.objectweb.proactive.core.rmi

"Sólo uso interno" Esta clase ayuda a unir las clases usadas entre ProActive y RMI.
Proporciona un ***SocketFactory*** para objetos remotos unicast, una clase auxiliar para crear o encontrar el ***RMIRegistry*** y la clase para manejar la clase servidor ***http***.

21) org.objectweb.proactive.core.runtime

Define todas las clases necesarias para construr un ***ProActiveRuntime***.
Un ***ProActiveRuntime*** ofrece configuración de servicios necesario para trabajar ProActive con JVM. Cada JVM que es apuntada para sostener objetos activos. Debe contener una y solo una instancia de la clase ***ProActiveRuntime***.
RMI y ***JINI*** implementaciones de ***ProActiveRuntime*** son proporcionadas en sub-packages.

22) org.objectweb.proactive.core.runtime.jini

Implementación JINI de ***ProActiveRuntime***

23) org.objectweb.proactive.core.runtime.rmi

Implementación RMI i de ***ProActiveRuntime***

24) org.objectweb.proactive.core.util

"Sólo uso interno" Contiene accesorios y útiles clases

25) org.objectweb.proactive.core.util.test

Clases
CircularArrayListTest

26) org.objectweb.proactive.core.xml

"Sólo uso interno" Contiene clases útiles para analizar el descriptor de ***XML***.

27) org.objectweb.proactive.core.xml.handler

"Sólo uso interno". Contiene clases útiles.

28) org.objectweb.proactive.core.xml.io

"Sólo uso interno" Contiene clases útiles.

29) org.objectweb.proactive.ext.locationserver

Define un cuerpo asociado con un servidor localizado capaz de dar la localización de un cuerpo en demanda, en caso de migración, un cuerpo deja atrás una

petición reenviada que permite a otros objetos comunicarse al cuerpo emigrado usando la misma referencia remota que ellos tenían antes de la migración.
Otra estrategia que usa una cadena de despachadores es usar un servidor de localización que esta notificando la localización del cuerpo. En la migración un cuerpo notifica al servidor de esta nueva localización. Otros objetos contactan al servidor para localizar al cuerpo que no responde a nadie.

30) org.objectweb.proactive.ext.migration
Define estrategias de migración basadas en un itinerario.
El cuerpo define primitivas de migración básica permitiendo migrar a otra localización. Es posible construir estrategias de migración sofisticadas encima de estas primitivas. Varias estrategias hechas usan el itinerario que está definido en este paquete

31) org.objectweb.proactive.ext.mixedlocation
Resumen de clases

Clases
MigrationManagerWithMixedLocation
MixedLocationMetaObjectFactory
MixedLocationMetaObjectFactory.MigrationManagerFactoryImpl
RequestWithMixedLocation
TimeRequestWithMixedLocation
UniversalBodyWrapper

32) org.objectweb.proactive.ext.util
Proporciona algunas clases útiles, para el uso de un usuario final o también utilizado para la extensión del paquete.

Capítulo III Flujo de Objetos

III.1 Introducción

El modelo más sencillo de un algoritmo puede ser definido mediante tres componentes: datos de entrada, procesamiento y datos de salida. La entrada y salida (E/S), es un aspecto fundamental en la computación, una computadora no sería útil si no pudiera recibir datos del mundo exterior y presentar los datos calculados.
Un flujo en general es una transferencia desde un emisor hacia un receptor. La transferencia puede ser de datos específicos, como un archivo de texto, un archivo binario, etc.

El entender que son los flujos en general, es fundamental en este trabajo de tesis, y específicamente un flujo en java. Así bien un flujo es un sistema de comunicación implementado en el paquete java.io cuyo fin es guardar y recuperar la información en cada uno de los diversos dispositivos de almacenamiento, Otra definición de flujo es: “secuencia de bytes que viajan desde una fuente a un destino a través de un camino, de modo secuencial”. Java a través de java.io y java.nio, y otros paquetes relacionados, proporciona clases independientes para manipular archivos y flujos (streams).

Para el caso particular de Java es posible imaginar un flujo como un tubo donde se puede leer o escribir bytes. No nos importa lo que pueda hacer en el otro extremo del tubo: puede ser un teclado, un monitor, un archivo, un proceso, una conexión TCP/IP o un objeto cualquiera.
Todos los flujos que aparecen en Java englobados en el paquete java.io, pertenecen a dos clases abstractas comunes: java.io.InputStream para los flujos de Entrada (aquellos de los que podemos leer) y java.io.OutputStream para los flujos de salida (aquellos en los que podemos escribir).
Estos flujos, pueden tener orígenes diversos (un archivo, un socket TCP, etc.), pero una vez que tenemos una referencia a ellos, podemos trabajar siempre de la misma forma: leyendo datos mediante los métodos de la familia read() o escribiendo datos con los métodos write(). Existen flujos de bytes y flujos de caracteres y dos categorías de flujos los de entrada y los de salida, en donde se podrá leer desde un flujo de entrada, pero no se podrá escribir en él, a diferencia que se podrá escribir en un flujo de salida, pero no se podrá leer. Así bien un flujo

de entrada es un objeto que una aplicación puede usar para leer una secuencia de datos, y un flujo de salida es un objeto que una aplicación puede usar para escribir una secuencia de datos; un flujo de entrada actúa como una fuente de datos y un flujo de salida actúa como un destino de datos.

El esquema básico de trabajo con los flujos es siempre el mismo: se abren, se realizan las operaciones deseadas de escritura y lectura y luego se cierran.
Un flujo de objetos es una secuencia ordenada de objetos que tiene una fuente (flujo de entrada) y un destino (flujos de salida), el flujo nos proporciona una interfaz de objetos uniforme, de tal forma que el programador no sabe que tipo de datos se encuentra en el flujo, puesto que es tratado como un objeto como tal, dándonos así todas las cualidades que tienen los objetos.

III.1.1 Flujos estándar de java

Java al igual que cualquier otro lenguaje de programación tiene su conjunto de métodos que le permiten captar la información de flujos de entrada y enviar flujos de salida por dispositivos estándar.

Los flujos estándar son:

- Flujo de datos de entrada: Se realiza mediante la clase System.in y suele recibir los datos del teclado, utiliza el método read() para conseguir leer los caracteres del teclado.
- Flujos de datos de salida: Se realiza mediante la clase System.out y suele enviar los datos por pantalla, utiliza el método print() y println() para la salida por pantalla.
- Flujos de datos de error: Se realiza mediante la clase System.err, la salida va dirigida al monitor para enviar mensajes de error al usuario. El flujo de entrada más utilizado está asociado al teclado. El siguiente programa es bastante ilustrativo al respecto.

```
import.java.io
public class HolaOscar {
public static void main(String[] args) trows IOException
{
   InputStreamReader lector=new InputStreamReader(System. in);
   BufferedReader entrada=new BufferedReader(lector);
  System.out.print("Cual es tu nombre?");
  String nombre=entrada.readLine();
  System.out.printl( Hola  + nombre  +  !  );
}
}
```

La salida de este programa al ejecutarlo es:

Da tu nombre: Davicín
¡Hola, Davicín!

En la Figura III.1 se puede observar el proceso de tomar los datos del flujo de estándar de entrada System.in asociado al teclado con la creación del objeto ***lector*** de la clase ***InputStreamReader***. En seguida el objeto ***lector*** es utilizado como argumento del constructor de un objeto ***entrada*** de la clase ***BufferReader***, el cual tiene métodos como ***readLine()***, para leer el conjunto de bytes del buffer hasta el final de la línea. Finalmente se utiliza el método **println()** de **System.out** que está asociado al flujo estándar de salida para presentar los datos en la consola o monitor.

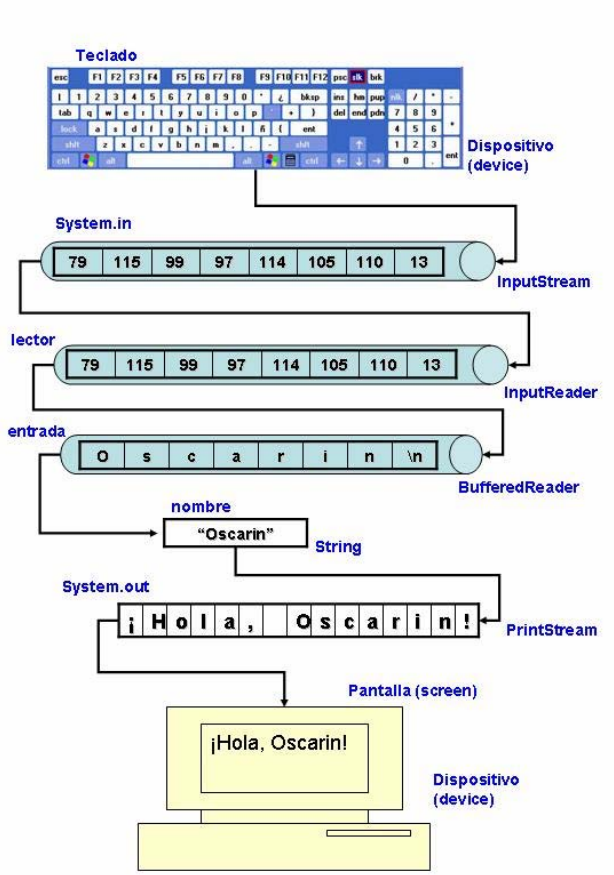


Figura III.1 Proceso tomar datos del flujo estándar de entrada asociado al teclado

III.1.2 Flujos que ofrece java.io

Java ofrece dos clases abstractas para manejar los flujos de datos procedentes de equipos remotos o archivos y son: java.io.OutputSream y java.io.InputSream, a continuación se muestra en la Figura III.2 la jerarquía de esas clases y del paquete java.io en general [5].

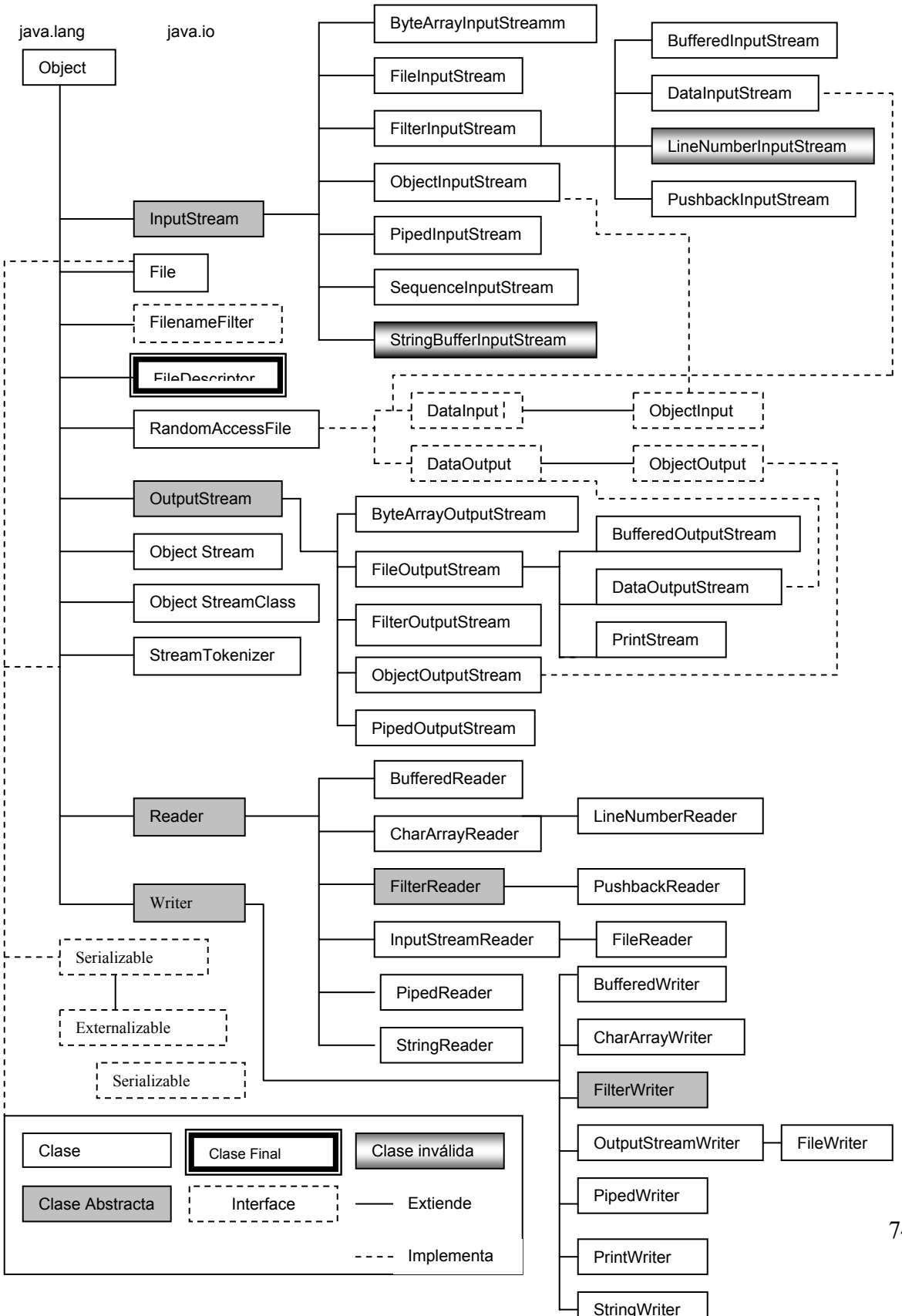

java.lang
java.io
Object
ByteArrayInputStreamm
FileInputStream
FilterInputStream
ObjectInputStream
PipedInputStream
SequenceInputStream
StringBufferInputStream
BufferedInputStream
DataInputStream
LineNumberInputStream
PushbackInputStream
InputStream
File
FilenameFilter
FileDescriptor
RandomAccessFile
DataInput
ObjectInput
DataOutput
ObjectOutput
OutputStream
Object Stream
Object StreamClass
StreamTokenizer
ByteArrayOutputStream
FileOutputStream
FilterOutputStream
ObjectOutputStream
PipedOutputStream
BufferedOutputStream
DataOutputStream
PrintStream
Reader
Writer
BufferedReader
CharArrayReader
FilterReader
InputStreamReader
PipedReader
StringReader
LineNumberReader
PushbackReader
FileReader
Serializable
Externalizable
Serializable
BufferedWriter
CharArrayWriter
FilterWriter
OutputStreamWriter
FileWriter
PipedWriter
PrintWriter
StringWriter
Clase
Clase Final
Clase inválida
Clase Abstracta
Interface
Extiende
Implementa

Figura III.2 Flujos que ofrece java.io

Como se muestra en la jerarquía del paquete java.io, éste provee una línea extensa de clases para trabajar con entrada y salida. Java provee flujos como un mecanismo general para trabajar con datos I/O. Los flujos implementan acceso secuencial de datos; las siguientes entidades pueden actuar como flujos de entrada y flujos de salida:

- Un arreglo de bytes de caracteres
- Un objeto string
- Un archivo
- Una tubería (pipe)
- Una secuencia de flujos
- Otras fuentes, como una conexión de Internet.

Los flujos en java pueden ser encadenados con filtros para proveer nueva funcionalidad. Además de trabajar con bytes y caracteres, los flujos proveen entrada y salida de objetos y valores primitivos de Java.

El paquete java.io también provee soporte para acceso aleatorio de archivos, y una interfaz genérica para interactuar con el sistema de archivos de la plataforma.

En java 1.0 la librería I/O fue totalmente orientada a bytes. En versiones posteriores se complementó con una librería orientada a caracteres, clases basadas en Unicote.

III.1.3 Flujos de bytes

Para los ejemplos que se hará mención en este punto, se supondrá que los objetos de tipo InputStream y OutputStream vienen dados, sin preocuparse de cómo han sido creados (de hecho, las clases mencionadas ya habían sido comentados en el punto anterior y se sabe que son clases abstractas que y no pueden ser instanciadas, así que son la base de otras clases concretas de donde si es posible instanciar objetos).

Lectura de bytes individuales

Se logra mediante código como:

```
nputStream is=…;
nt b=is.read();
```

Así se puede obtener el siguiente byte de InputStream. Hay que darse cuenta que el byte (8 bits) se devuelve como un dato de tipo int (32 bits), con un valor entre 0

y 255. ENCASO de que se haya alcanzado el final del archivo, el método read() devuelve un valor de -1.

Lectura de varios bytes

Primero se crea un arreglo unidimensional del tamaño adecuado. El tamaño de este arreglo es lo que indica al método read() cuántos bytes debe leer como máximo. El siguiente ejemplo de fragmento de código lee 1024 bytes de un flujo de entrada.

```
byte[]= miArreglo= new byte[1024]; //declaración del arreglo
InputStream is=…;
Int lee=is.read(miArreglo);
```

La variable lee almacena el número de bytes que se han leído en realidad. Si se llega al final del archivo, devuelve el valor de -1. No hay garantía de leer exactamente el número de bytes especificado, porque puede ser menor debido a que se está leyendo de un archivo que se ha acabado, porque los datos de una conexión de red tardan en llegar, o cualquier otra causa. De cualquier forma, cuando el método read() devuelve un valor distinto de -1 podemos seguir leyendo mediante sucesivas llamadas a read(). Veamos el siguiente ejemplo que nos muestra el uso del método read() para leer bloques de bytes.

```
Public class EscribeNombres {
  public static void main(String[] args)
  {
   final int        LON=20;
   byte[] buffer1= new byte[LON];
   System.out.println("Introduce tu apellido:");
   try { System.in.read(buffer1,0,LON); }
        Catch (Exception e) {}]
    String apellido=new String(buffer1);
    Byte[] buffer2= new byte[LON];
    System.out.println("Introduce tu nombre:");
    Try { System.in.read(buffer2,0,LON);}
        Catch (Exception e) {}]
     String nombre=new String(buffer2);
     System.out.println("Hola,  + nombre.trim()+      + apellido.trim());
   }
}
```

La salida del programa es:
Introduce tu apellido: MARQUEZ
Introduce tu nombre: MARGARITA
Hola, MARGARITA MARQUEZ

Cerrar el flujo de entrada

Cuando ya no se necesita leer datos de un flujo de entrada, se tienen que liberar los recursos reservados mediante el método close(). Si no cerrar el InputStream explícitamente, el flujo asociado se cierra cuando se destruye el objeto.

Escritura de bytes individuales

La clase OutputStream dispone de varios métodos write(). Por ejemplo:

```
OutputStream os=..;
Int dato=777;
Os.write(dato);
```

Como en el caso del método read() de la clase InputStream, el método write() recibe un byte dentro de una variable del tipo int (32 bytes).

Escritura de varios bytes

El dedazo de código siguiente es muy ilustrativo.

```
byte[] vector ={65,66,67,68,69};
        OutputStream os=..;
        Os.write(vector); //escribe los bytes 65,66,67,68,69
        Os.write(vector,1,3); //escribe los bytes 66,67,68n
```

Gestión de excepciones de entrada/salida

Todos los métodos read(), write(), avaible(), etc. Lanzan excepciones de tipo java.io.IOExcepcion. Estas excepciones se capturan de forma obligatoria, o aparecerán errores en tiempo de compilación.

En general, las partes de los programas que trabajen con los flujos deben estar dentro de una cláusula try … match.

En programas pequeños en los que no se quiera complicarse la existencia con estructuras de éste tipo, se puede tomar el camino alternativo de mandar la excepción “hacia arriba”. Por ejemplo:

```
public static void main(String[] args) IOExcepcion{
}
```

III.1.4 Flujos de acceso a archivos

Ya sea para leer o para escribir en un archivo, estos se manipulan con flujos de acceso a archivos. En java tenemos la clase ***FileInputStream***, con los métodos necesarios para abrir e interactuar con un canal de comunicación hacia un archivo de entrada para nuestra aplicación, y la clase ***FileOutputStream*** para el caso de un archivo de salida.

Las clases ***FileInputStream*** y ***FileOutputStream*** reciben en uno de sus constructores como parámetro el nombre del archivo a leer o escribir. Hay otras dos variantes: una que recibe un objeto de tipo File y otra que recibe un objeto de tipo ***FileDescriptor***.

Objetos FileInputStream

Los objetos ***FileInputStream*** típicamente representan ficheros de texto accedidos en orden secuencial, byte a byte. Con ***FileInputStream***, se puede elegir acceder a un ***byte***, varios ***bytes***, o al fichero completo.

Apertura de un FileInputStream

Para abrir un ***FileInputStream*** sobre un fichero, se le da al constructor un ***String*** o un objeto ***File***:

FileInputStream miFicheroSt;
miFicheroSt=new FileInputStream("/etc/kk");

Lectura de un FileInputStream

Una vez abierto el FileInputStream, se puede leer de él. El método ***read()*** tiene muchas opciones:

- lee un byte y devuelve -1 al final del ***stream***
  int read();
- llena todo el array, si es posible. Devuelve el número de bytes leídos o -1 si se alcanzó el final del stream.
  int read(byte b[]);
- Lee longitude bytes en b comenzando por ***b[offset]***. Devuelve el número de bytes leídos o -1 sis e alcanzó el final del stream.
  int read(byte b[], int offset, int longitud);

Cierre de FileInputStream

Cuando se termina con un fichero, existen dos opciones para cerrarlo: explícitamente, o implícitamente cuando se recicla el objeto (el *garbage collector* se encarga de ello). Para cerrarlo explícitamente, se utiliza el método close().
miFicheroSt.close();

Objetos FileOutputStream

Los objetos FileOutputStream son útiles para la escritura de ficheros de texto. Como con los ficheros de entrada, primero se necesita abrir el fichero para luego escribir en él.

Apertura de un FileOutputStream

Para abrir un objeto FileOutputStream, se tiene las mismas posibilidades que para abrir un fichero stream de entrada. Se le da al constructor un String o un objeto File.

FileOutputStream miFicheroSt;
miFicheroSt = new FileOutputStream("/etc/kk");

Escritura en un FileOutputStream

Una vez abierto el fichero, se puede escribir bytes de datos utilizando el método write(). Como con el método read() de los streams de entrada, se tienen tres posibilidades:

- Escribe un byte
  void write(int b);
- Escribe todo el array, si es posible
  void write(byte b[]);
- Escribe longitud de bytes en b comenzando por b[offset].
  void write(byte b[], int offset, int longitud);

Cierre de FileOutputStream

Cerrar un stream de salida es similar a cerrar ***streams*** de entrada. Se puede utilizar el método explicito:

miFicheroSt.close();

O, se puede dejar que el sistema cierre el fichero cuando se recicle ***miFicheroSt***.

III.1.5 Flujos en Memoria

Es posible que en alguna ocasión se desee manejar a veces un buffer en memoria (array) o una cadena de texto como un flujo. Aunque en principio puede parecer muy extraño querer manejar una cadena de este modo, esto permite escribir código para tratar del mismo modo cadenas, bloques de memoria o archivos.
Las clases que proporciona java para esto son **ByteArrayInputStream**, **ByteArrayOutputStream.** El método **available** está garantizado que devuelve el número de bytes en memoria, y además existe la particularidad de que **reset** nos lleva al comienzo del buffer, en lugar de a un marcador guardado con **mark.**

ByteArrayInputStream (flujo de entrada de matriz de bytes) es una implementación de un flujo de entrada que utiliza una matriz de bytes como origen. Esta clase tiene dos constructores, y ambos necesitan una matriz de bytes que proporcione el origen de los datos. Un ***ByteArrayInputStream*** implementa un método adicional además de los admitidos por ***FileInputStream, reset()*** reinicializa el puntero del flujo situándolo al comienzo del mismo, que en este caso es el comienzo de la matriz de bytes que se han utilizado en el constructor.

En cuanto a **ByteArrayOutputStream**, es un buffer dinámico, que crece conforme le vamos añadiendo datos. Se le puede especificar un tamaño base, o bien dejar que tenga un tamaño inicial por defecto. La clase ***ByteArrayOutputStream*** ofrece varios métodos nuevos con respecto a ***OutputStream***. Es posible saber el número de bytes que se han escrito mediante **size** Además, es posible obtener un **array** con los datos del flujo, mediante ***toByteArray***, o cadenas de texto, mediante las dos versiones de ***toString***. Esto último puede ser útil, dado que resulta muy común que lo que manejemos en memoria no sea más que una cadena de texto. Como una comodidad adicional, existe la posibilidad de pasar toda la información almacenada en la memoria por el flujo a otro flujo de salida, como un archivo, etc., mediante el método ***writeTo(flujoSalid*a**)**.

ByteArrayOutputStream tiene dos constructores. En el primer tipo de constructor se crea un buffer de 32 bytes. En el segundo, se crea un buffer con un tamaño igual al argumento en bytes, que en este caso es 1024 bytes:

```
OutputStream out0 = new ByteArrayOutputStream();
OutputStream out1 = new ByteArrayOutputStream(1024);
```

El tener una matriz de ***bytes*** como destino de la salida proporciona algunas oportunidades nuevas para un ***OutputStream***, y la clase ***ByteArrayOutputStream*** se aprovecha de ellas. Igual que la mayoría de los otros métodos de escritura, devuelven un ***void*** y lanzan una ***IOException*** cuando hay una condición de error.

III.1.6 Comunicación entre procesos/threads mediante flujos

Java proporciona unos flujos especiales para comunicación entre ***threads***, la ventaja de utilizar este modo de comunicación es que Java se encarga de todas las tareas de sincronización en el acceso a los datos, de modo que los procesos lectores y escritores no choquen. Las clases utilizadas para llevar a cabo esta taerea son **PipedOutputStream** y **PipedInputStream.**

La idea básica aquí es tener un objeto de cada clase, y los ***threads*** usan el de la clase ***PipedInputStream***, y los procesos escritores el de la clase ***PipedOutputStream***, a través de los cuales se accede a una misma información, para ponerlos de acuerdo en que esto es así, hay que conectar el flujo de entrada con el de salida, lo que se hace mediante código como el que sigue:

pipeEntrada.connect(pipeSalida); o bien
pipeSalida.connect(pipeEntrada)

Con lo cuál ambos trabajan sobre la misma información. Como se puede ver, el método connect existe para ambas clases, y es único método que añaden a sus clases base, que como de costumbre son ***InputStream*** y ***OutputStream***. Además la operación de poner de acuerdo a ambos flujos también se puede llevar a cabo mediante un constructor que proporcionan y que permite pasar como parámetro el pipe complementario, lo que hace innecesario llamar a ***connect***.

Como podemos ver, el concepto de flujo resulta muy potente, a través de él podemos obtener información o leerla de casi cualquier fuente, incluyendo archivos, memoria, cadenas, o incluso otro proceso/thread. Si fuera necesario, podría definir flujos para comunicación vía ***DDE***, o casi cualquier cosa. Además, hay flujos que actúan sobre otros, por ejemplo para concatenar dos o más fuentes de información, como ***SequenceInputStream***.

Beneficios de los flujos
La interfaz de flujos de datos de E/S en Java proporciona una abstracción limpia para una tarea compleja y a menudo incómoda. La composición de las clases de flujo filtradas permite que se construya dinámicamente una interfaz de flujo personalizada que se adapte a los requisitos de transferencia de datos. Los programas de Java que utilicen las clases abstractas y de alto nivel ***InputStream*** y

OutputStream funcionarán adecuadamente en el futuro incluso cuando se invierten clases de flujo concretas nuevas y mejoradas.

III.3 Necesidades

Con la explosión del Internet y la aparición de las microcomputadoras al público en general, los objetos multimedia se difunden más y más. Todos y cada uno desean ser capaces de difundir un texto pequeño, fotografías, películas, etc. Durante mucho tiempo, el modo de difusión de estos medios era simple, primero descargar el archivo y después verlo. Pero por el paso de los años, han aparecido técnicas que hacen esto posible, logrando difundir videos de una manera más conveniente en vez de inicialmente descargarlo y después verlo; el archivo que contiene el sonido o el video es fijado por el flujo, los datos son tratados progresivamente con su llegada, en tiempo real. Por ejemplo, para el vídeo, los paquetes de datos de los recursos de Internet serán descomprimidos inmediatamente y las imágenes progresivamente con sus llegadas. Esta técnica es llamada ***técnica de flujo***.
Hablar sobre el flujo del vídeo o sonido es algo muy común hoy en día, el usuario no tiene nada más que tener paciencia durante el cargado inicial del archivo; por otra parte, el ancho de banda de la red, tiene un tiempo de espera, Por ejemplo, es posible que un archivo que contiene un archivo encode con el formato AVI no comprimido contiene 1 Mb o más datos para un segundo del video. Tales archivos requieren no solamente capacidades de almacenamiento importantes, pero también las unidades de almacenamiento capaces de proveer los datos con un ancho de banda alto. A modo de ejemplo, el ancho de banda de un CD-ROM es de 150Kb, mientras que en un disco duro estándar es de 5Mb. Esta es una de las razones del por que los objetos multimedia son enviados en Internet y comprimidos. Es por ello que varios tipos de compresión están disponibles para los varios tipos de medios. Referente a los datos audio, cotejar los formatos con el formato definido por SUN, el wav el formato de Microsoft y el estándar famoso MP3. Para los videos se puede cotejar el formato avi que es el formato definió por Microsoft, el cuarto de galón definido por APPLE, el definido por Real Networks y los estándares ***MPEG1, MPEG2 y MPEG4***.

El envío de un flujo, es una técnica que toma más y más importancia en la red, y en particular en multimedia; El HTTP es por ejemplo un protocolo con flujo, el cual permite ver el texto antes de que las imágenes sean completamente transmitidas y estén puestas.

Por el momento con los flujos que provee java, el flujo y el envío de un flujo son técnicas que son llevadas a cabo a un nivel relativo bajo de abstracción, primeramente en el nivel de los protocolos de red (por ejemplo. RTP Protocolo de transporte en tiempo real, RTSP Protocolo de flujo en tiempo real.

Hoy en día las necesidades de definir modelos y técnicas para un flujo de objetos, con un alto nivel de abstracción de la programación con lenguajes orientados a objetos, es mucha, y en particular en java. Así uno debe poder permitir el flujo mientras el resto esta a nivel de programación analizando distintas

llamadas de métodos (RPC, RMI). Lo anterior es logrado en este trabajo de tesis, gracias a la potencialidad de ProActive, que permite olvidarse de analizar las diferentes llamadas a métodos y concentrarse únicamente en el flujo, logrando así las primitivas mínimas necesarias para el desarrollo de aplicaciones con flujos de objetos, y con lo cual se logra que el programador solo se enfoque en el problema y no en los detalles de la implementación. El capítulo siguiente llevará de la mano al lector para una mejor comprensión del concepto antes planteado.

Capítulo IV Diseño de la Infraestructura genérica

IV.1 Objetivo de la infraestructura genérica para flujo de objetos

El objetivo principal de esta infraestructura es la transmisión de objetos por un flujo, sin pérdida de objetos y el control del flujo, por un usuario; Logrando con esto mejorar la eficacia de la programación distribuida con objetos, y además, que el usuario pueda tener el control de los objetos que se encuentran dentro del mismo flujo, y lograr una infraestructura que simplemente invoque a los métodos adecuados y obtenga un flujo transparente de objetos en un ambiente paralelo y distribuido.

Gracias a los objetos activos con los que cuenta ProActive, los cuales son unidades básicas de actividad y distribución, es posible, el diseño de un conjunto de primitivas básicas para el flujo de objetos y obtener un flujo transparente de objetos en un ambiente paralelo y distribuido como ya se ha mencionado.

La implementación se llevó dentro del marco de trabajo de la biblioteca ProActive, una biblioteca 100% escrita en java, la cual proporciona como ya se vio en el capitulo II la creación de objetos distantes y activos, llamadas asincrónicas con las transparencias futuras, la migración de objetos activos (movilidad), el ofrecimiento entero de los mecanismos de la sincronización del alto nivel que simplifican la programación distribuida, entre otras cosas.

La infraestructura presentada en este trabajo de tesis tiene un desafío importante además de la simplicidad y de flexibilidad, que es el obtener un buen funcionamiento, lo cual nos da como consecuencia que el trabajo tenga que incluir como lo mas lejos posible el tomar en cuenta el nivel conceptual, la implementación de los protocolos de redes existentes de flujo y así como el curso de estandarización.

IV.2 Desarrollo de la Infraestructura genérica para flujo de objetos

El desarrollo de una infraestructura genérica para el flujo de objetos se ha propuesto básicamente para el envío de multimedia como ha sido propuesto en [6], otra aproximación está siendo desarrollada por HP [7], donde su principal objetivo es investigar sobre sistemas de flujo de medios sobre Internet. En algunos casos, como es en [8], se motiva el uso de flujo de objetos para aplicaciones particulares, como es el caso de realidad aumentada.
La importancia del desarrollo de infraestructuras genéricas puede verse por las propuestas anteriormente descritas. Cada propuesta en el área de cómputo distribuido y paralelo necesariamente necesita generar infraestructuras genéricas para facilitar el desarrollo de las aplicaciones paralelas y distribuidas. Obviamente, la idea principal detrás de cada propuesta es identificar, diseñar y crear componentes de software reutilizables que posean un conjunto de criterios, en este caso para flujo de objetos.

La infraestructura genérica propuesta en este tema de tesis está basada en ProActive, una capa intermedia (ambiente y modelo de programación) orientado a objetos para programación paralela, concurrente y distribuida. Con esta propuesta se ha extendido la biblioteca de funciones ProActive implementando un modelo de componentes jerárquico y dinámico por el flujo de objetos.

IV.3 Diseño

IV.3.1 Diagrama de Casos de Uso

Los Casos de Uso describen el comportamiento de un sistema desde el punto de vista del usuario, se compone de 3 elementos:

Actores:	Roles que se juegan en el sistema.
Sistema:	Es el sistema en si mismo.
Casos de uso:	Escenarios de iteración de los actores.

En el diagrama de casos de uso presentado en la Figura IV.1, los Actores están conformados por el o los usuarios y por ProActive, el sistema es el control del flujo de objetos. Los casos de uso son tres.

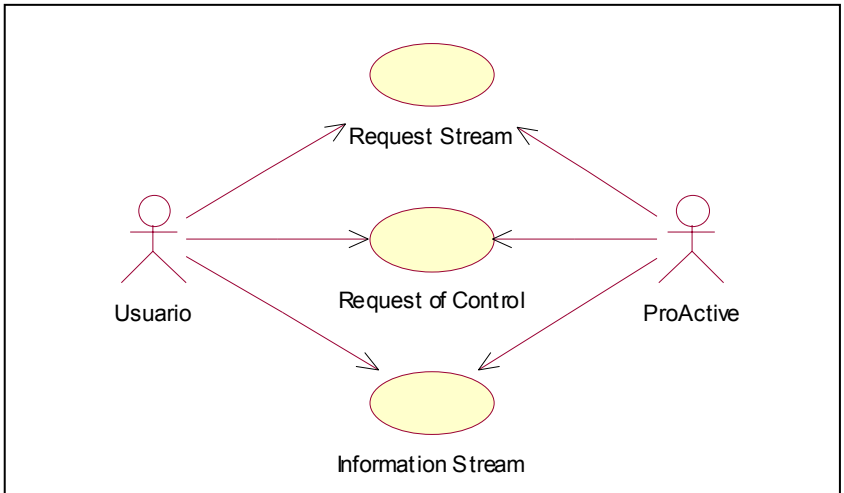


Figura IV.1 Diagrama de Casos de Uso

Como se muestra en la Figura IV.1 un usuario tiene las posibilidades de decidir entre cualquiera de los tres diferentes casos de uso Request Stream, Request of Control y Information Stream.

En el caso de uso Request Stream, un usuario realiza una petición de transferencia de flujo de objetos al sistema; éste a su vez realizará una configuración inicial para poder realizar a cabo el flujo, lo cual es llevado a cabo a través de Configure como se puede observar en la Figura IV.2 y regresa como respuesta la transferencia del flujo.

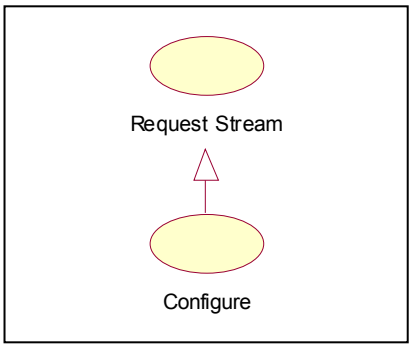


Figura IV.2 Asociación extiende de Request Stream

El usuario por otra parte puede tener también la opción de solicitar la información del flujo de objetos, a través del caso de uso Information Stream en donde podrá decidir entre conocer el estado del flujo, (State Stream), ver las prioridades de envío (Priority Stream), verificar la velocidad de transferencia (Verify Velocity Transfer), o visualizar la información acerca de los cambios que se realicen en el control del flujo, si el usuario decide conocer el estado del flujo, este le indicara como se encuentra un objeto, ya sea en proceso, detenido o simplemente ya no existe en el flujo; por su parte Priority Stream muestra la información acerca de la prioridad de transferencia para la transferencia, o el usuario podrá tener también la posibilidad de observar la velocidad de transferencia durante la misma a través de Verify Velocity Transfer, lo cual se puede observar en la Figura IV.3.

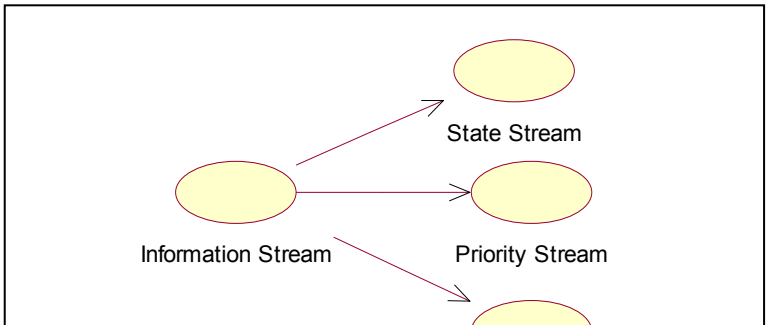


Figura IV.3 Asociación extiende Information Stream

Por otro lado y lo mas importante de este trabajo de tesis es el control del flujo, con el cual el usuario podrá interactuar gracias al caso de uso Request of Control (Figura IV.4), en donde el usuario podrá tener el control de este y así poder decidir entre reconfigurar el flujo (Reconfigure), Detener el flujo (Stop Stream), Reinicializar el flujo (Reinitialize), Eliminar el flujo (Killer Stream) o Restaurar el flujo (Restart Stream. Si el usuario decide Reconfigurar el flujo, tendrá la posibilidad de elegir entre cambiar de prioridad para la transferencia (Change Priority), Modificar el IP (IP Modify) o Modificar el puerto para la conexión (Port Modify), como es posible observar en el la Figura IV.5.

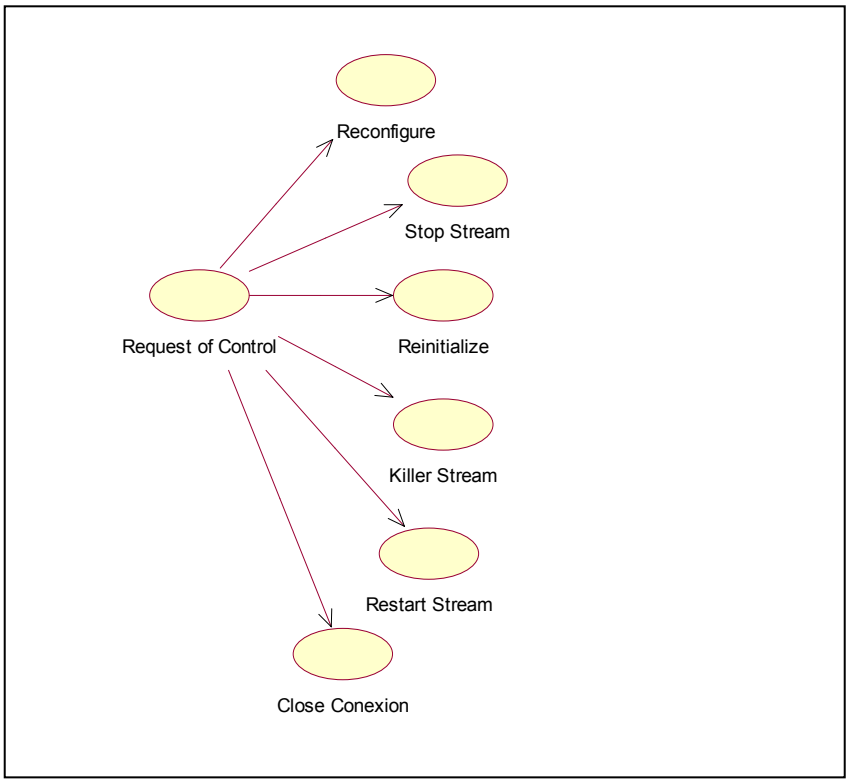


Figura IV.4 Asociación extiende Request of Control()

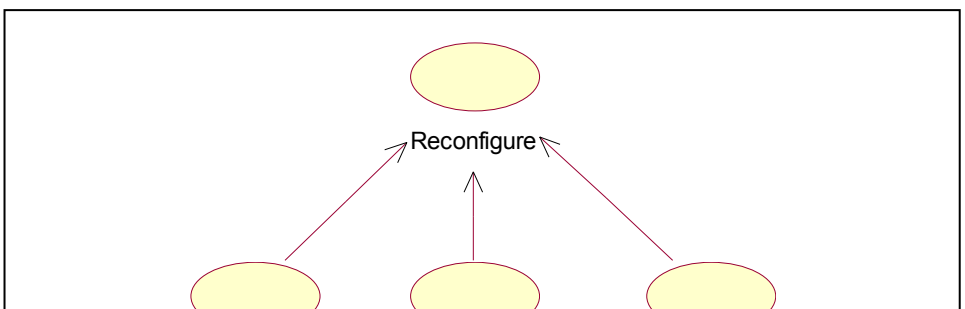


Figura IV.5 Reconfigure

IV.3.2 Identificación de Clases y Objetos

La infraestructura genérica, tal cual se indicó anteriormente, está diseñada con la ayuda de los objetos activos; se cuenta con un modelo cliente-servidor; por el lado del cliente se tienen tres objetos activos, uno al que se le llama **ViewControlAO** el cual interactúa con el usuario mediante una interfaz gráfica; un segundo objeto activo **ClientAO**, el cual es el encargado de enviar las peticiones de transferencia al servidor, y por ultimo se tiene un tercer objeto activo el cual recibirá la transferencia del objeto hecha por el servidor, al que se le ha denominado **TransferAO**, este objeto es de gran importancia puesto que tiene la tarea de controlar el flujo. Por el lado del servidor se cuenta también con tres objetos activos: **ServerAO**, es el objeto encargado de recibir las peticiones del cliente, las cuales son depositadas en una cola de peticiones, dichas peticiones son atendidas una a una, **TransferAO_Server** por su parte es el objeto encargado de realizar la conexión entre los hosts a llevar a cabo el flujo;. El usuario puede decidir en un momento dado realizar cambios de transferencia, cambiar el IP o Puerto destino, así como también cambiar las prioridades de transferencia, realizar acciones como: cancelar la transferencia, o definitivamente terminarla, muchas de estas acciones necesitarán cerrar el flujo, la conexión o quizás ambas, pues bien **TransferAO** es el objeto encargado de realizarlas. Por su lado **ViewControlAO** es el responsable de verificar que las acciones antes mencionadas sean mostradas al usuario. Existen otros objetos, involucrados tanto del lado del servidor como del cliente, pero es de gran importancia hacer un énfasis especial sobre los objetos activos, para la compresión de la arquitectura, mas adelante se vera la iteración con los demás objetos que no son objetos activos.

La arquitectura de esta propuesta puede ser vista en Figura IV.6

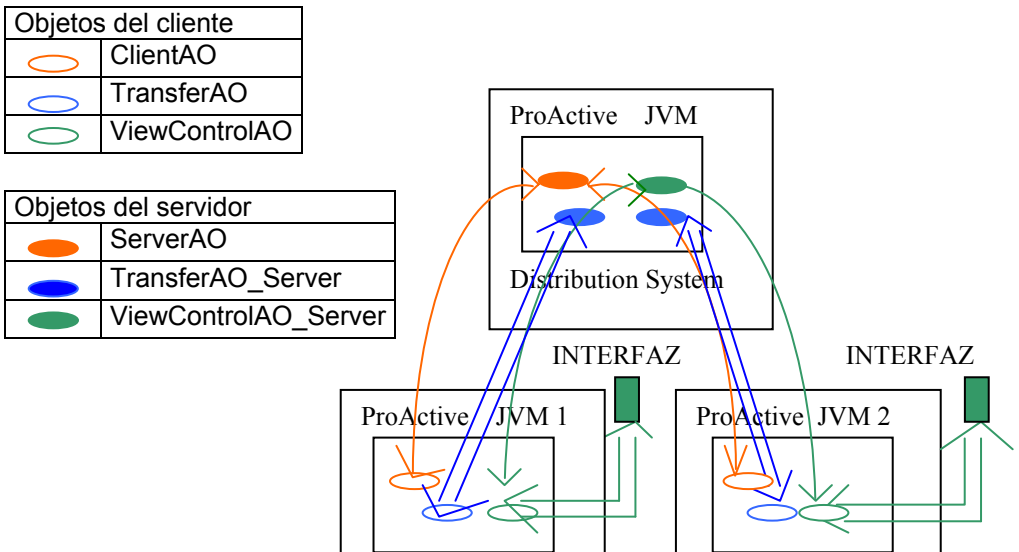
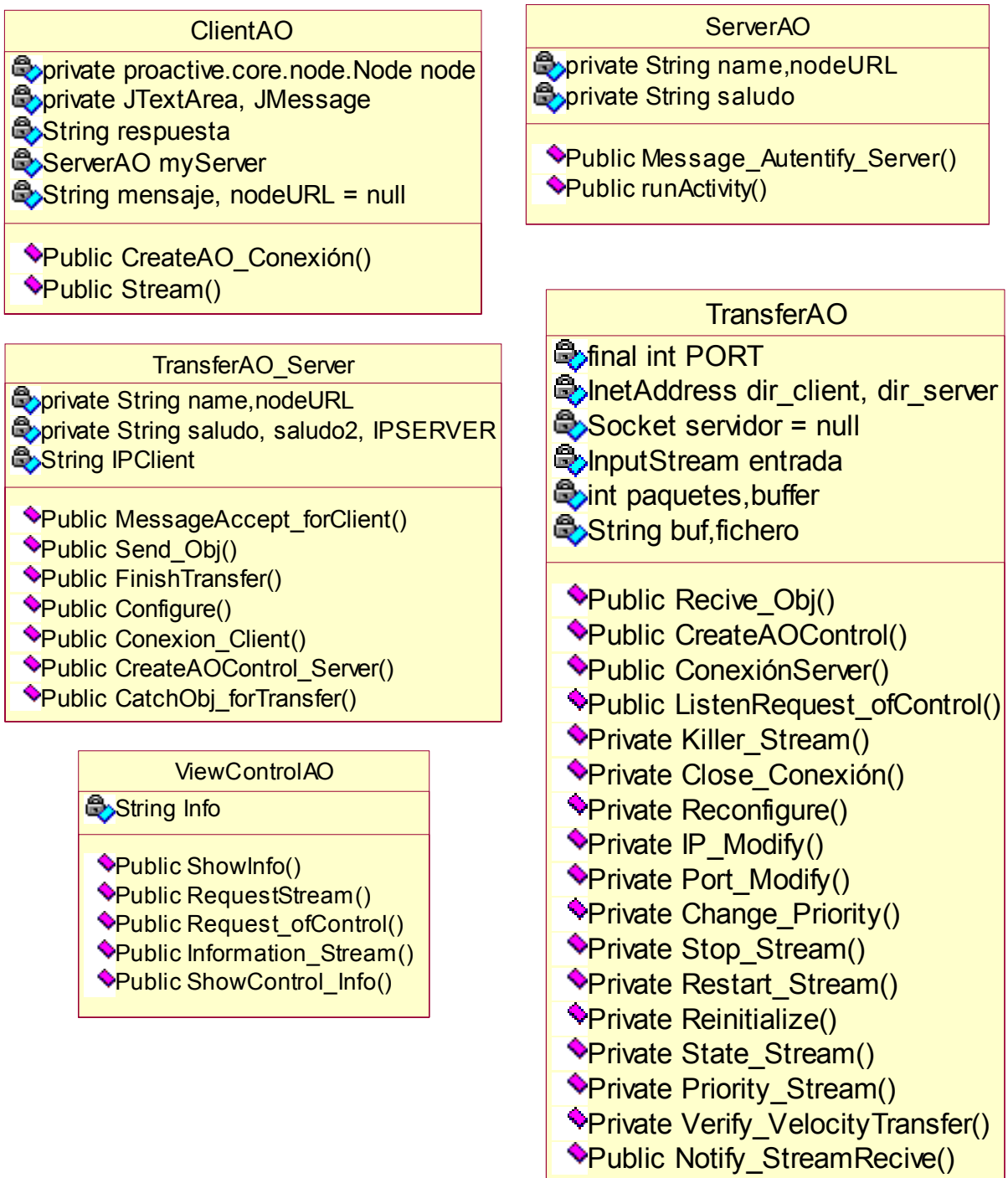


Figura IV.6 Arquitectura de la infraestructura.

Las Ventajas del diseño de esta Infraestructura son las siguientes:

- retardo de sincronización en la pérdida de enlace (LOD)
- Control remoto de la transferencia
- Transferencia paralela con grupos



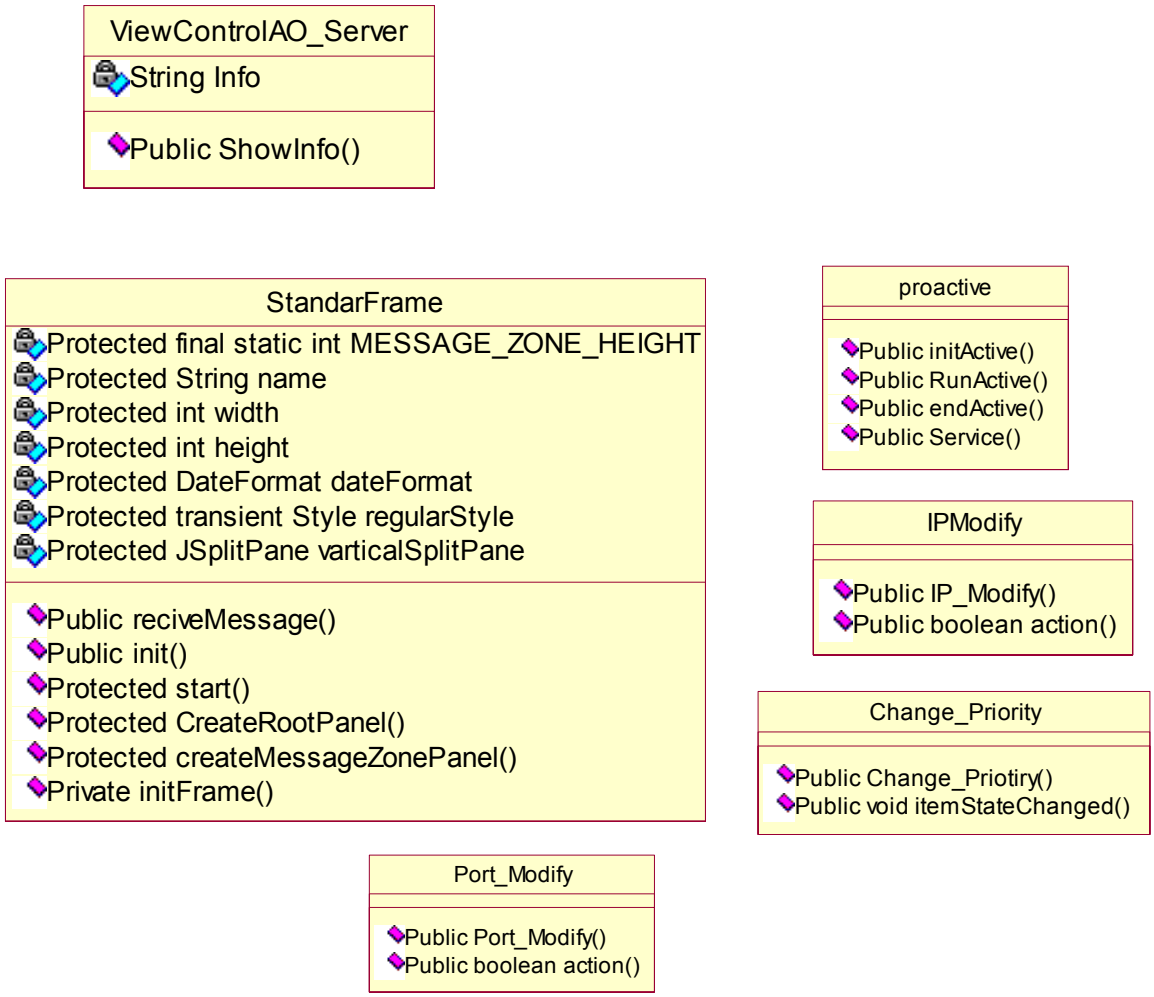


Figura IV.7 Identificación de Clases

IV.3.3 Diagrama de Asociación

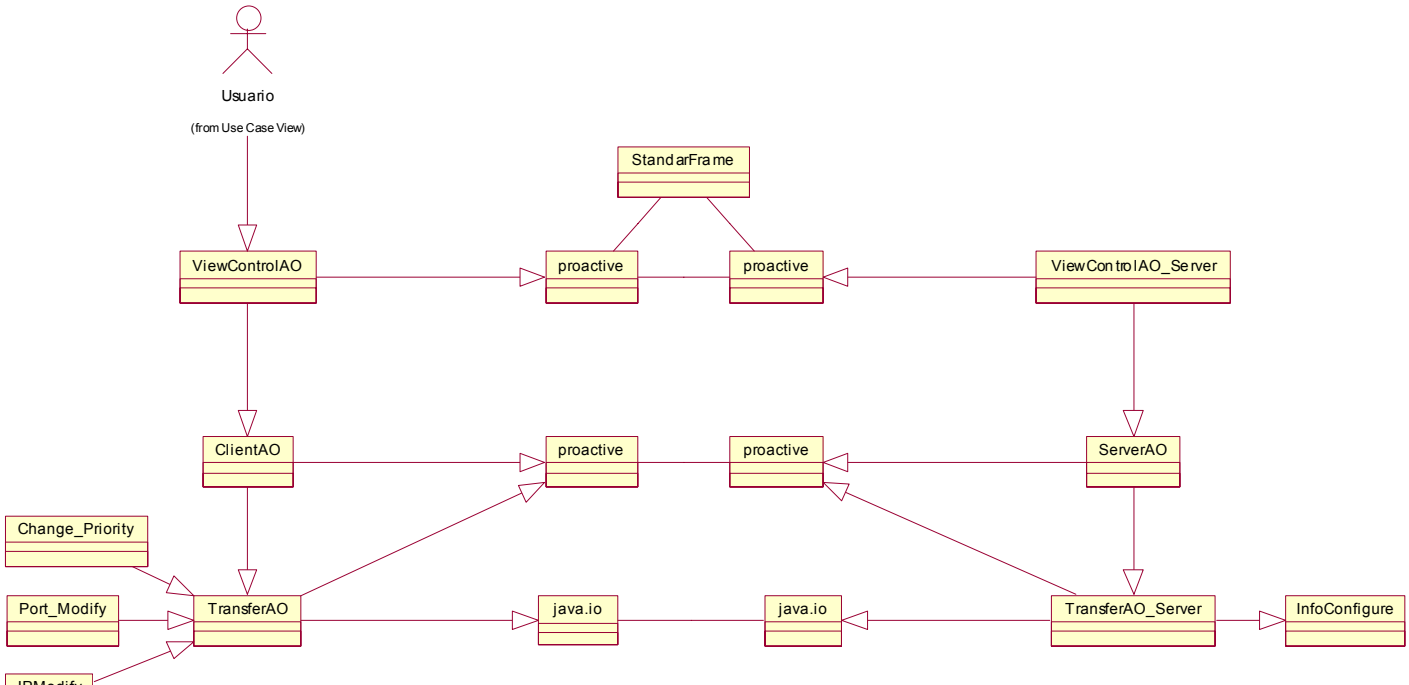


Figura IV.8 Diagrama de Asociación

IV.3.4 Diagrama de Clases

El Diagrama de Clases es el diagrama principal para el análisis y diseño. Un diagrama de clases presenta las clases del sistema con sus relaciones estructurales y de herencia. La definición de clase incluye definiciones para atributos y operaciones. El modelo de casos de uso aporta información para establecer las clases, objetos, atributos y operaciones. El mundo real puede ser visto desde abstracciones diferentes (subietividad)

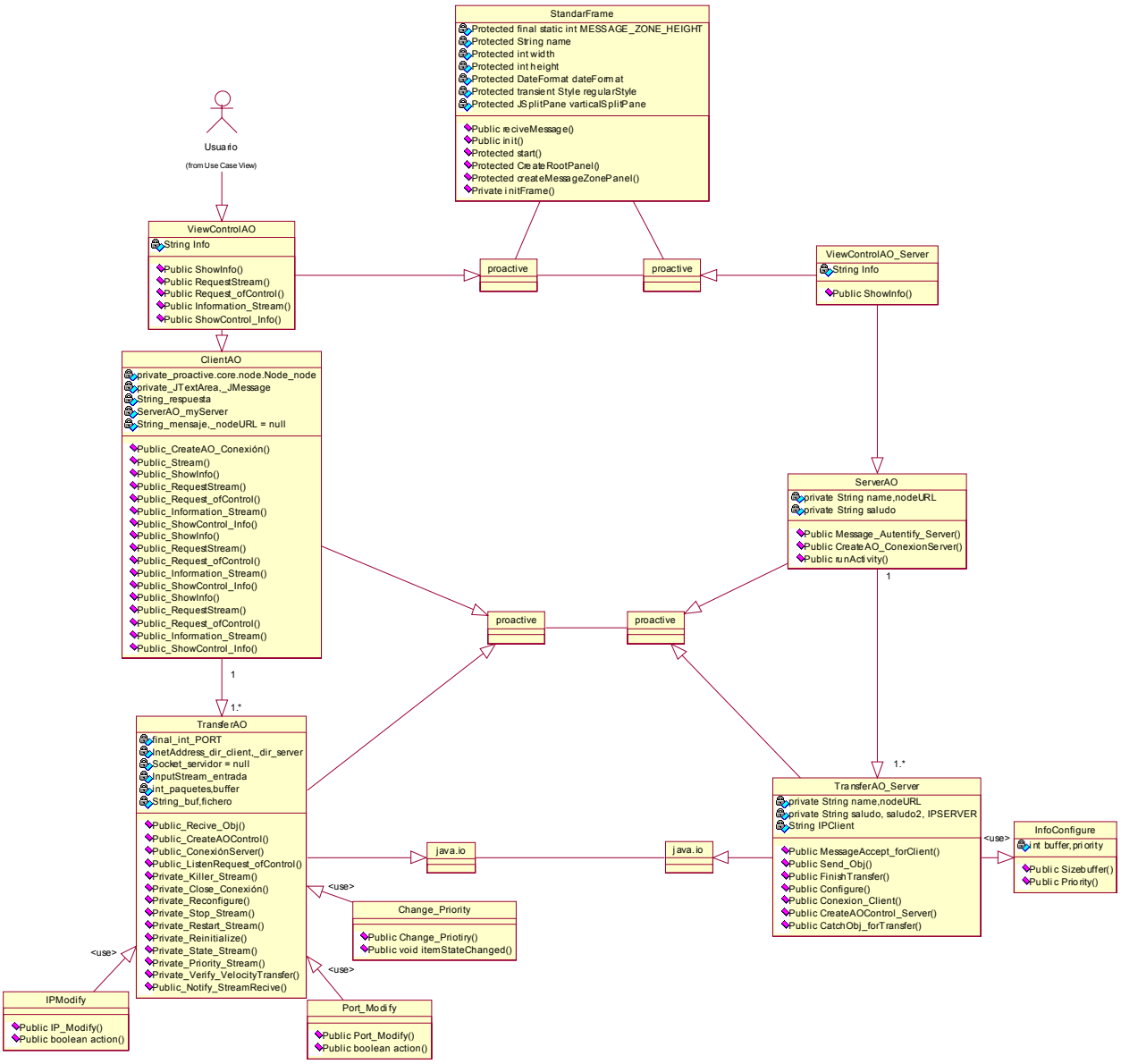


Figura IV.9 Diagrama de Clases

IV.3.5 Diagramas de Secuencia

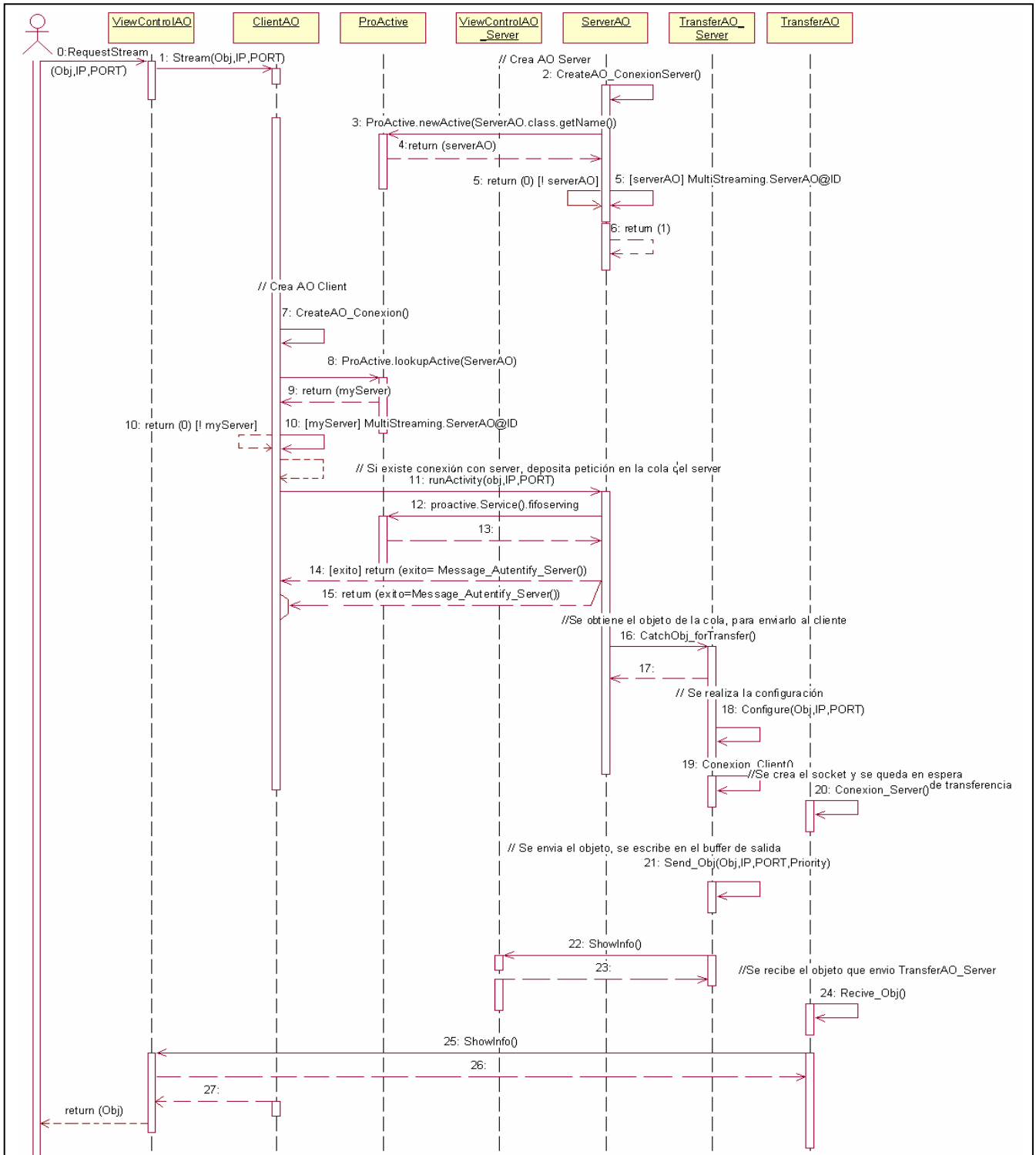


Figura IV.10 Diagrama de Secuencia de RequestStream()

El diagrama de secuencia muestra las interacciones entre los objetos organizadas en una secuencia temporal. En particular muestra los objetos participantes en la interacción y la secuencia de mensajes intercambiados. La Figura IV.10 presentada arriba muestra a los objetos participantes en el caso de uso Request Stream, de esta forma podemos observar la secuencia temporal para lograr satisfacer la petición de flujo de un usuario, en donde un usuario introduce el objeto a enviar a su IP y puerto para la conexión, estos datos los recibe ViewControlAO y éste envía la información a ClientAO, el cual será quien realmente haga la solicitud al servidor para la transferencia; el serverAO crea un objeto activo para permitirle al cliente comunicarse con el y atender las peticiones. Por su parte el ClientrAO crea un objeto activo para la comunicación con el servidor y a través de este objeto activo realizara la solicitud al ServerAO y el objeto activo es creado correctamente, la creación de este nos devolvera un identificador del objeto activo y si no regresara null.

Una vez que existe comunicación entre ClientAO y ServerAO, el ClientAO deposita la petición en la cola del ServerAO para que este pueda atenderla.

Hasta este momento como se puede observar en el diagrama TransferAO_Server ya tiene la información necesaria para poder realizar el flujo, y lo primero que hace es configurar el flujo de transferencia inicial, para poder entender la configuración para la transferencia, podemos observar la Figura IV.11, en donde como se puede observar se configura el tamaño del buffer y la prioridad de envío y recepción, en donde la prioridad nos va a dar la pautar para saber que objeto será enviado primero y cual después, y así como la cantidad de paquetes enviados por el buffer.

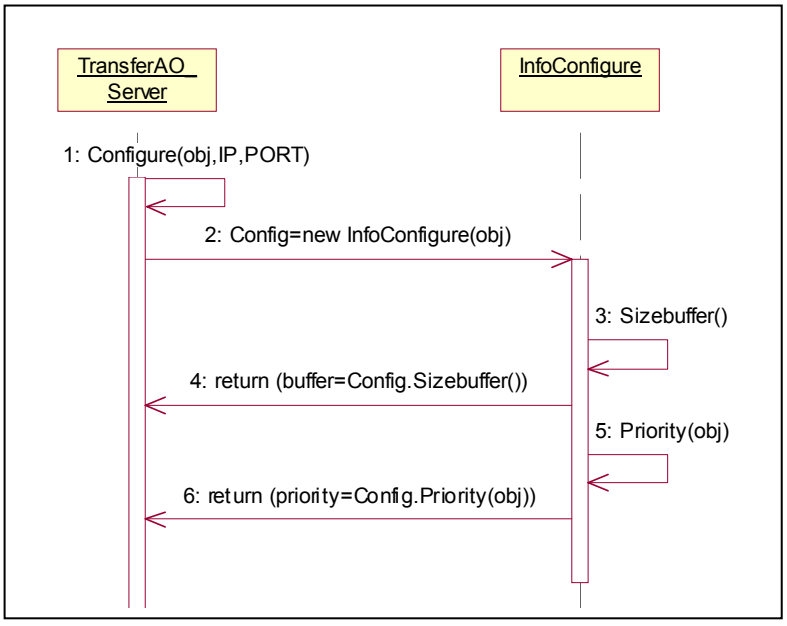


Figura IV.11 Diagrama de Secuencia de Configure()

Una vez ya echa la configuración TransferAO_Server levanta el socket para el flujo de objetos, con la ayuda del paquete java.net. Este proceso puede ser observado en la Figura IV.12.

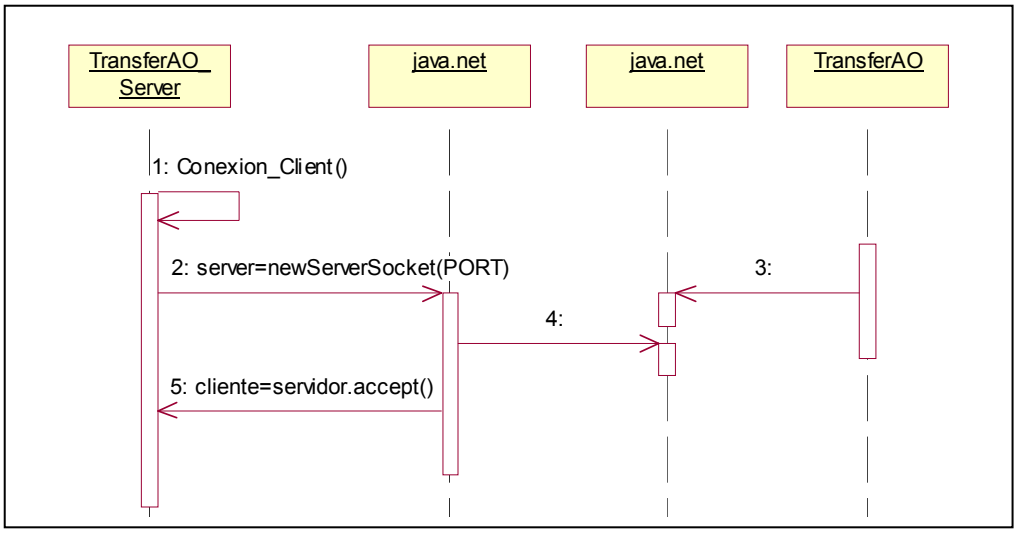


Figura IV.12 Diagrama de Secuencia de Conexion_Client()

Por su parte TransferAO busca el socket de TransferAO_Server utilizando el paquete java.net y se queda en un estado de espera de transferencia por parte de TransferAO_Servrer. La Figura IV.13 muestra lo anterior.

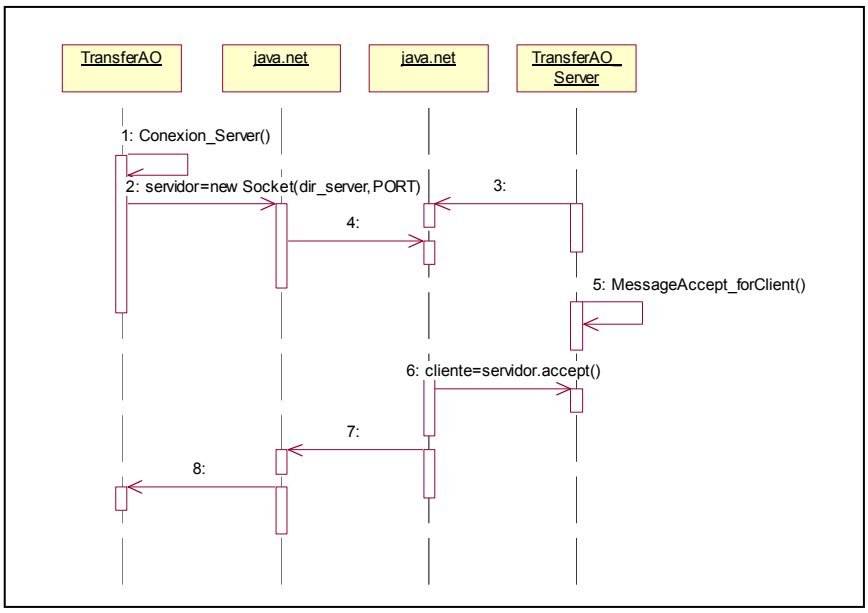


Figura IV.13 Diagrama de Secuencia de Conexion_Server()

Si la conexión es exitosa TransferAO_Server envía el objeto a TransferAO, utilizando el paquete java.io; lo anterior es posible observar en Figura IV.14.

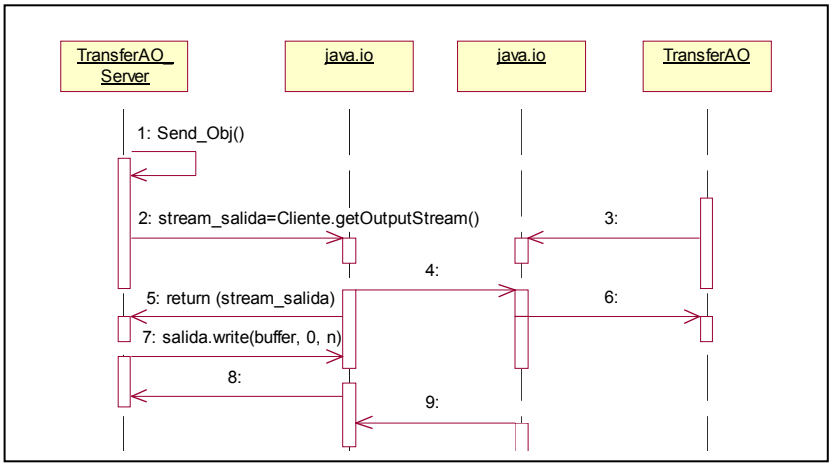


Figura IV.14 Diagrama de Secuencia de Send_Obj()

TransferAO recibe el objeto (Figura IV.15), que le envía TransferAO_Server a través del paquete java.io, atendiendo así la petición echa por el usuario.

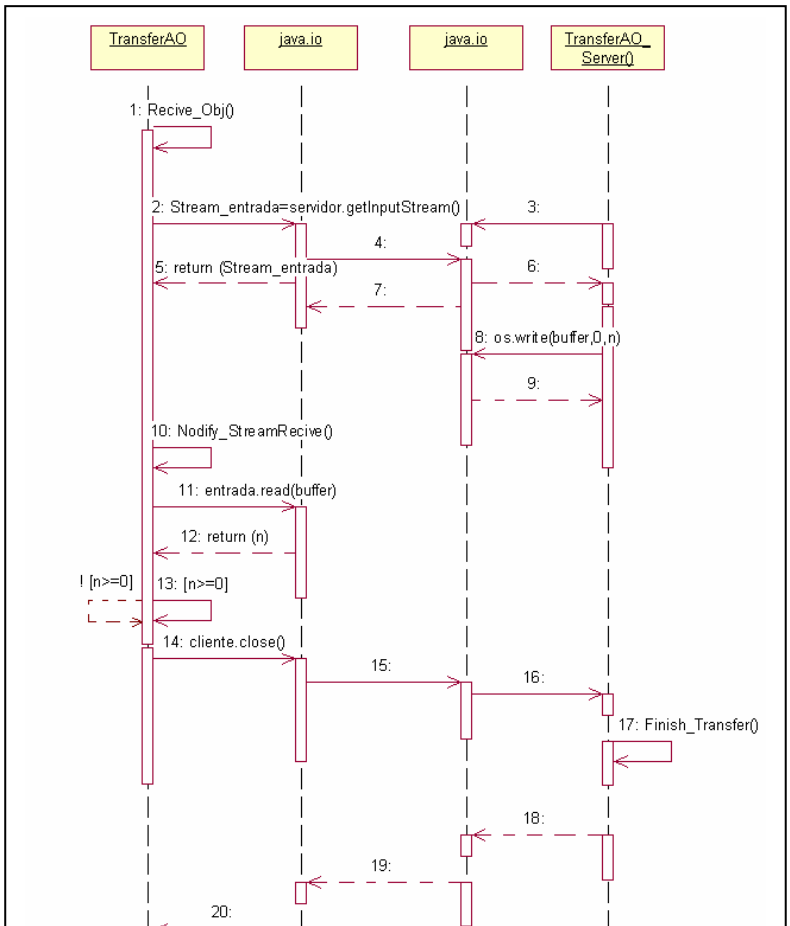


Figura IV.15 Diagrama de Secuencia de Recive_Obj()

La Figura IV.16 presentada abajo muestra a los objetos participantes en el caso de uso Request_ofControl, de esta forma podemos observar la secuencia temporal para lograr satisfacer la petición de un usuario para el control del flujo, en donde un usuario introduce el tipo de control que desee tener sobre el flujo, en donde tiene las posibilidades de elegir entre Reconfigurar el flujo lo cual se logra con el método Reconfigure(), como es posible observar en la Figura IV.5, el caso de uso Reconfigure muestra al usuario las opciones de son posibles reconfigurar entre las cuales se encuentran Modificar el IP, Modificar el puerto para la conexión y realizar un cambio de prioridad.

La Figura IV.17 nos muestra la secuencia a seguir para lograr la modificación del IP, de igual manera en la Fig. IV.18 se pueden observar los objetos que interactúan para lograr la modificación del puerto, y como es posible observar será necesario cerrar la conexión y reiniciar otra conexión con el puerto que el usuario a modificado.

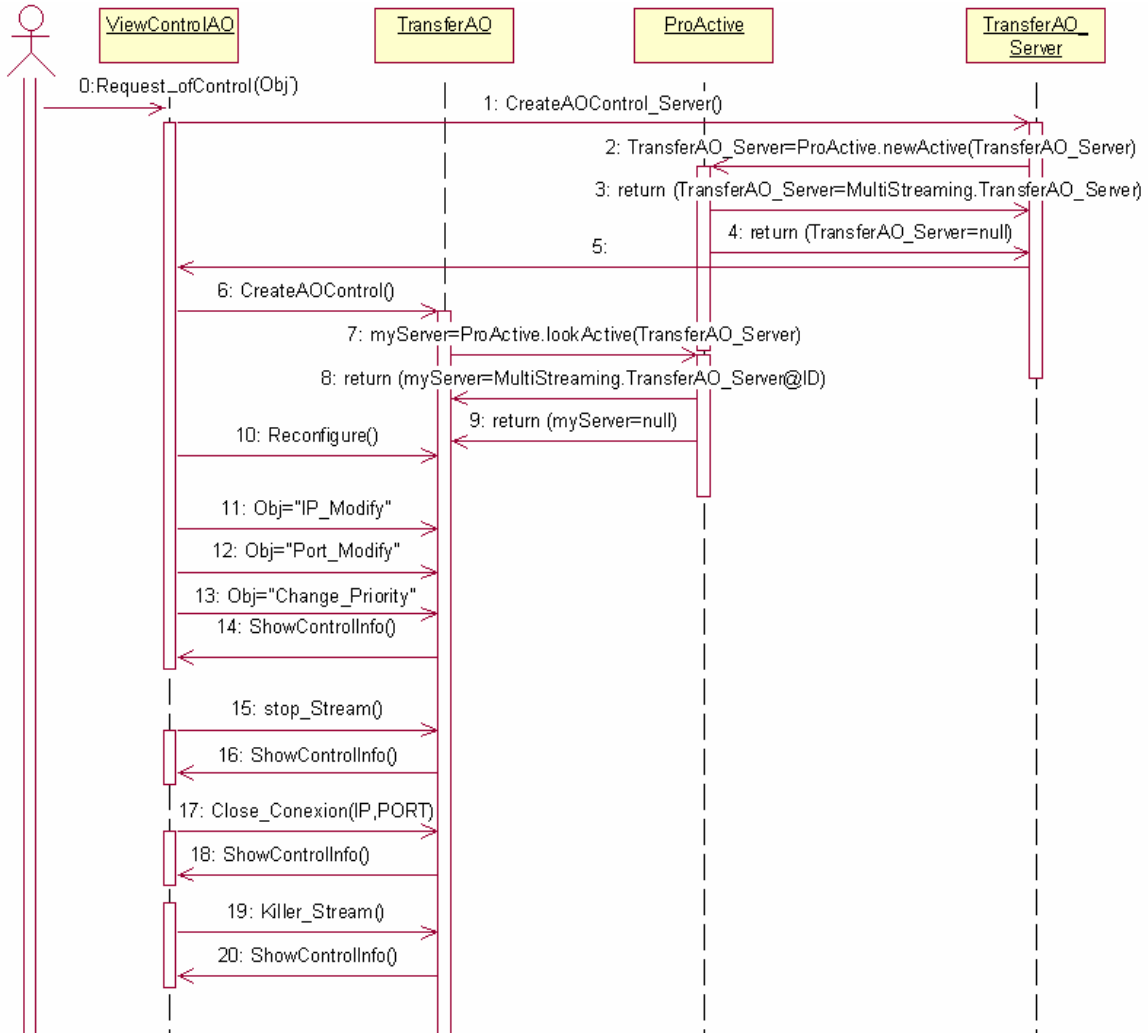


Figura IV.16 Diagrama de Secuencia de Request_ofControl()

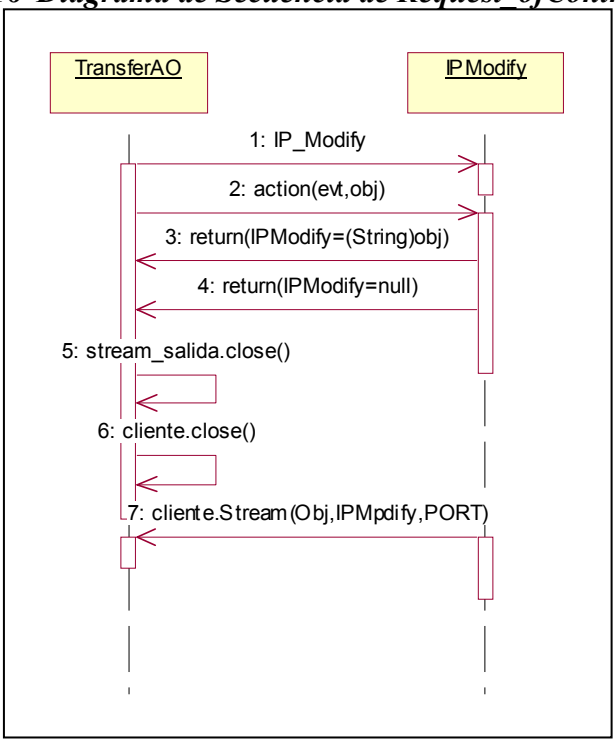


Figura IV.17 Diagrama de Secuencia de IPModify()

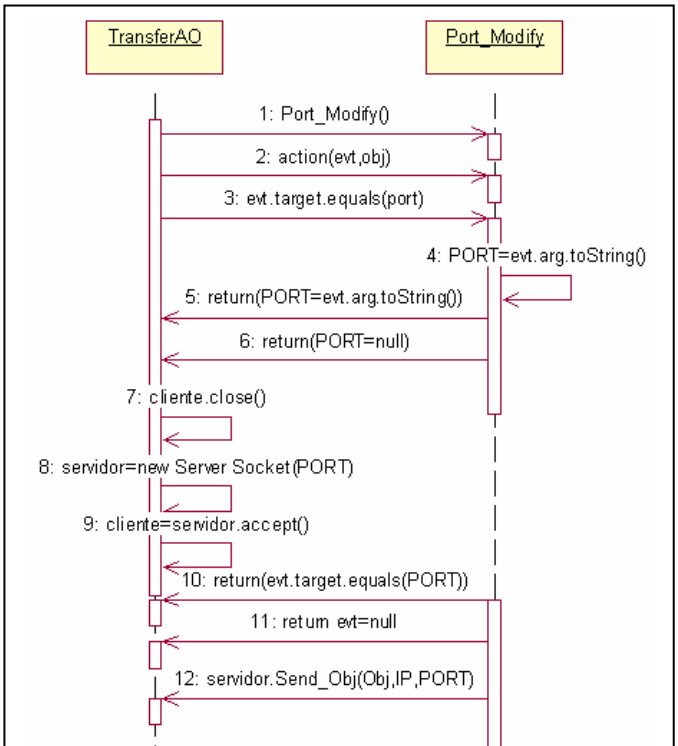


Figura IV.18 Diagrama de Secuencia de Port_Modify()

Por otro lado la política de envío de múltiples objetos a través de un flujo, recae principalmente en el uso de prioridades por objetos. La Figura IV.19 muestra los diferentes tipos de prioridades que el usuario pode elegir, gracias al uso de prioridades, es posible optimizar el canal de transferencia, enviando objetos con diferentes prioridades. Así, un archivo de video o sonido podría tener una mayor prioridad que uno de texto.

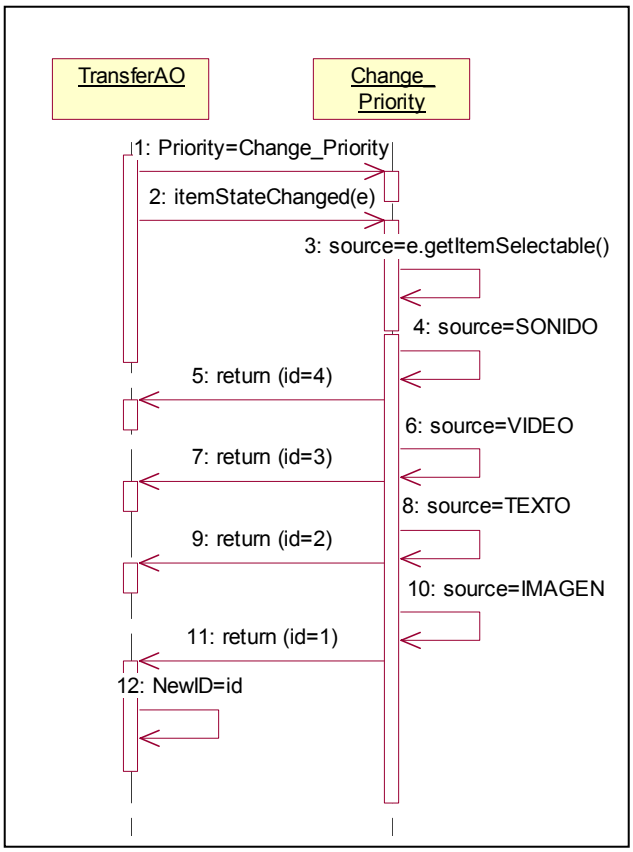


Figura IV.19 Diagrama de Secuencia de Change_Priority()

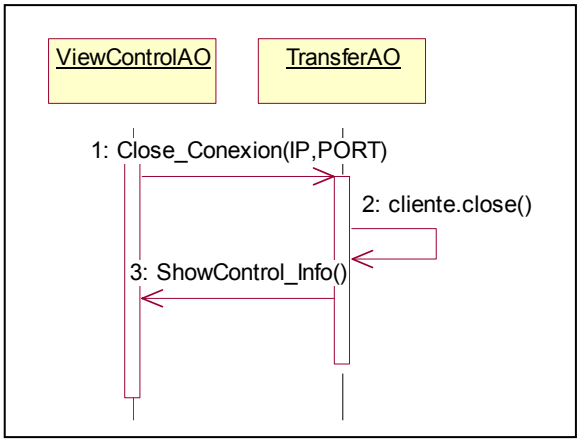


Figura IV.20 Diagrama de Secuencia de Close_Conexion()

Otras de las opciones con las que cuenta el usuario sobre el control del flujo son: detener el flujo (Stop_Stream()), lo cual es posible observar en la Figura IV.21, matar el flujo (Killer_Stream()), este se muestra en la Figura IV.22, reinicializar el flujo (Reinitialize()), restaurar el flujo (Restart_Stream()) y cerrar la conexión para el flujo (Close_Conexion()), son también opciones para el control del flujo.

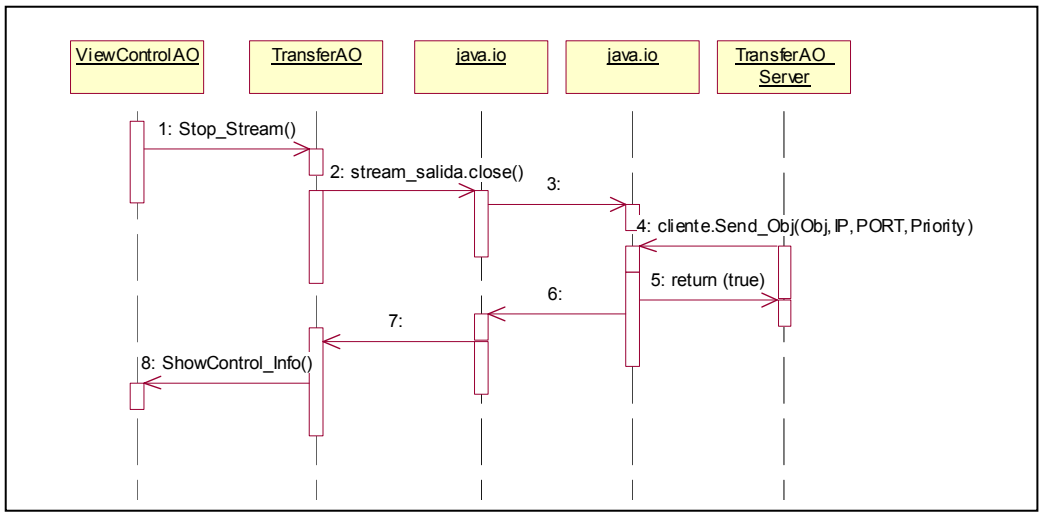


Figura IV.21 Diagrama de Secuencia de Stop_Stream()

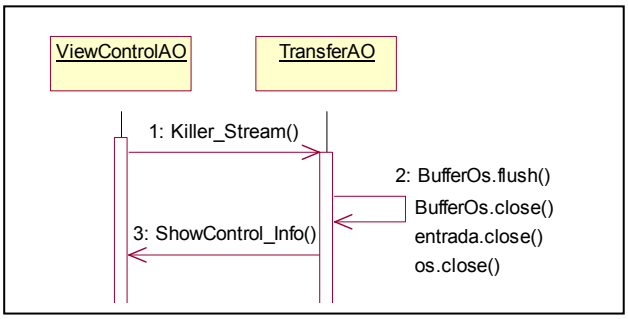


Figura IV.22 Diagrama de Secuencia de Killer_Stream()

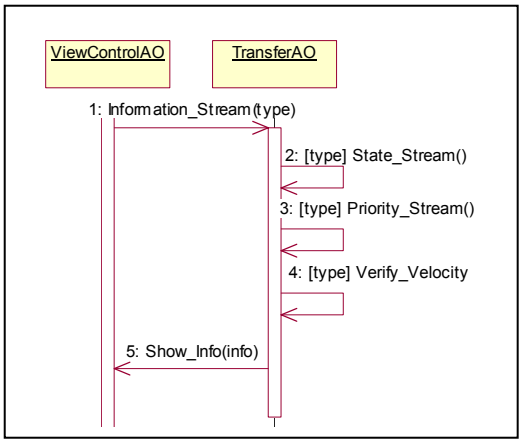


Figura IV.23 Diagrama de Secuencia de Information_Stream()

IV.3.6 Diagramas de Colaboración

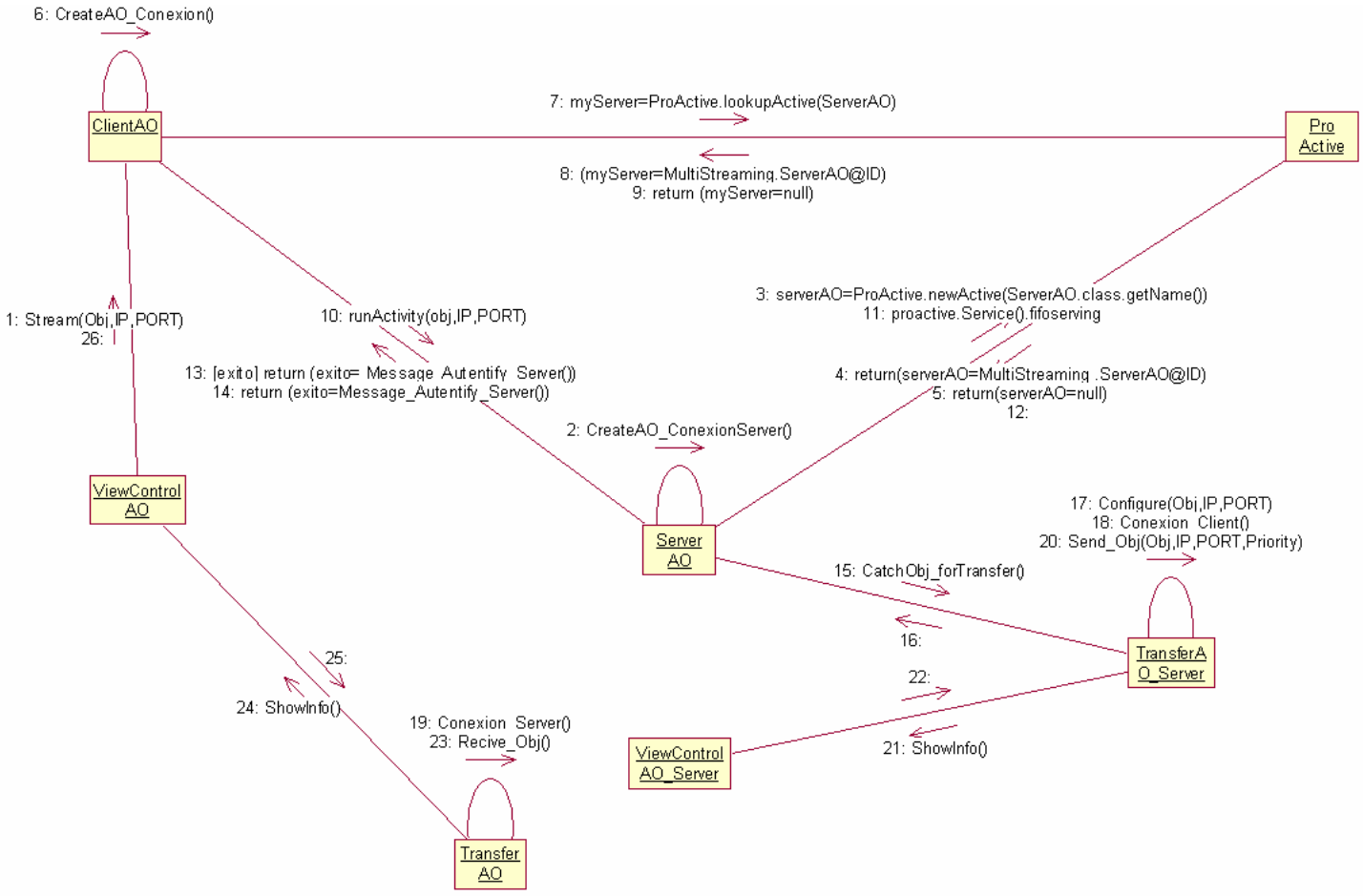


Figura IV.24 Diagrama de Colaboración de RequestStream()

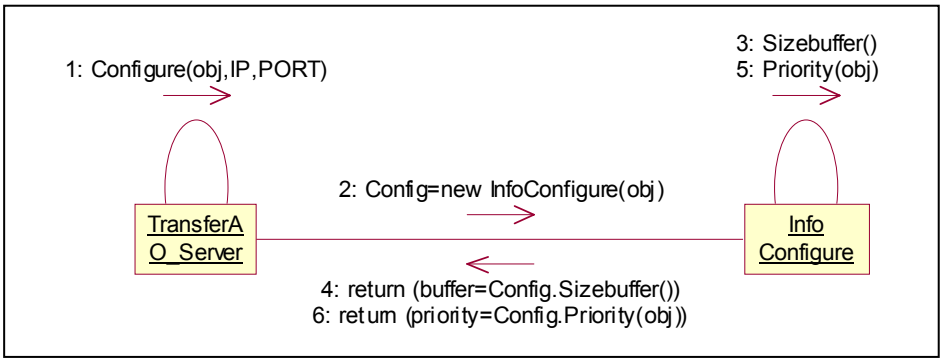


Figura IV.25 Diagrama de Colaboración de Configure()

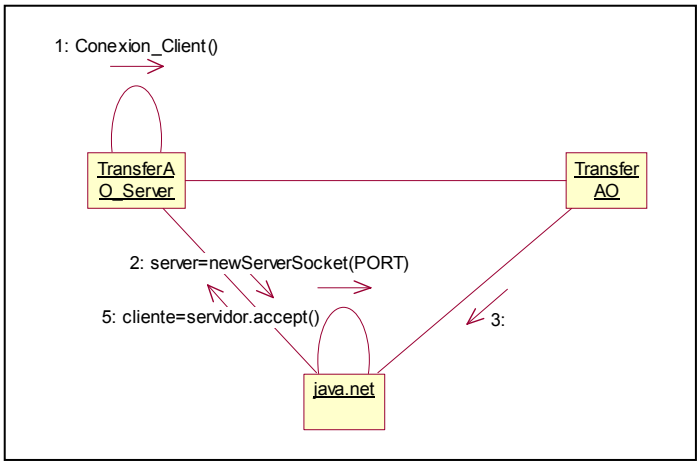


Figura IV.26 Diagrama de Colaboración de Conexion_Client()

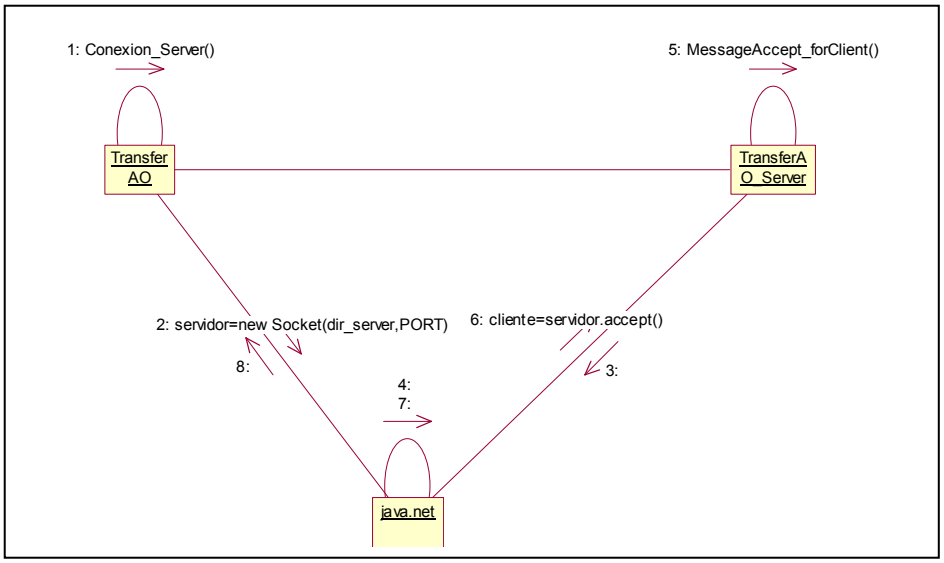


Figura IV.27 Diagrama de Colaboración de Conexion_Server()

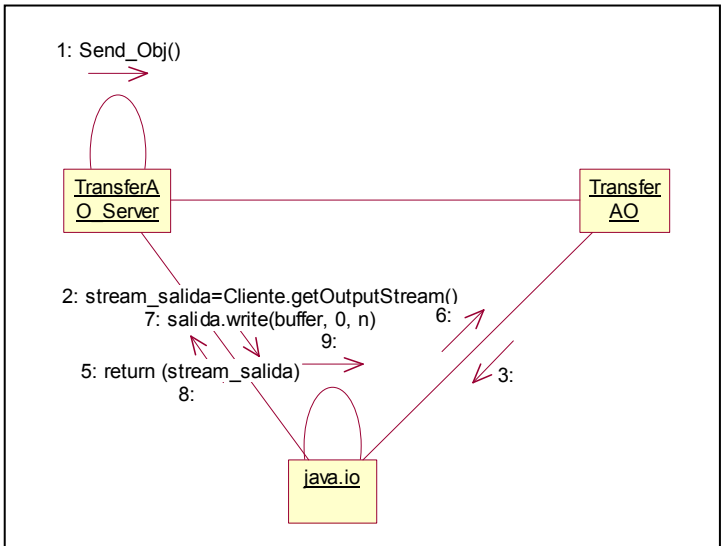


Figura IV.28 Diagrama de Colaboración de Send_Obj()

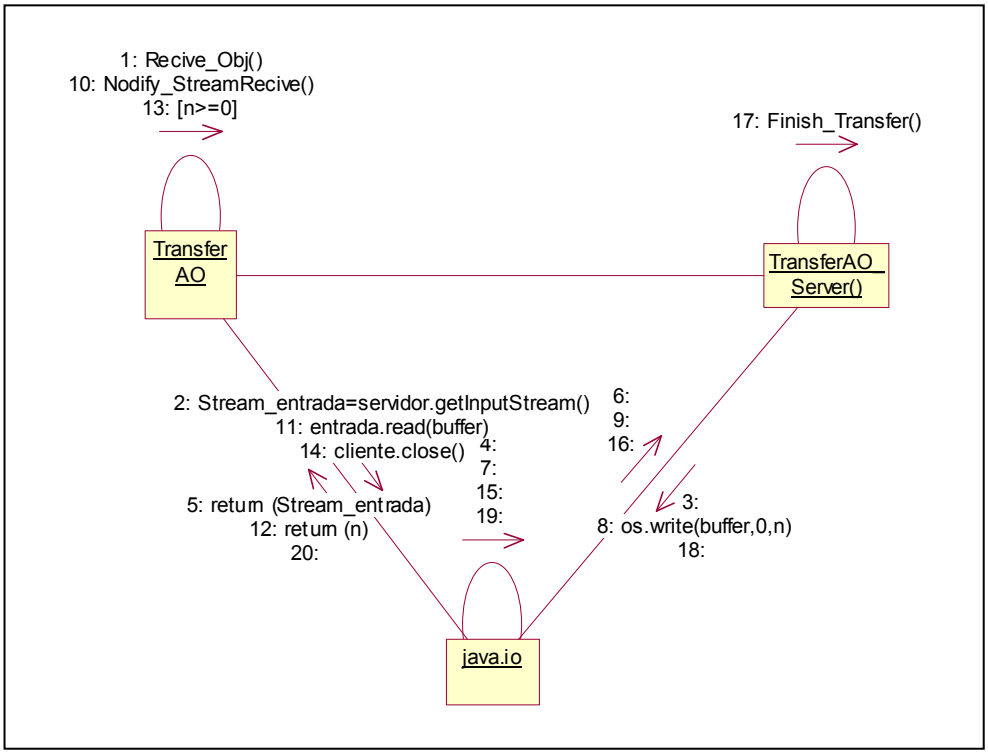


Figura IV.29 Diagrama de Colaboración de Recive_Obj()

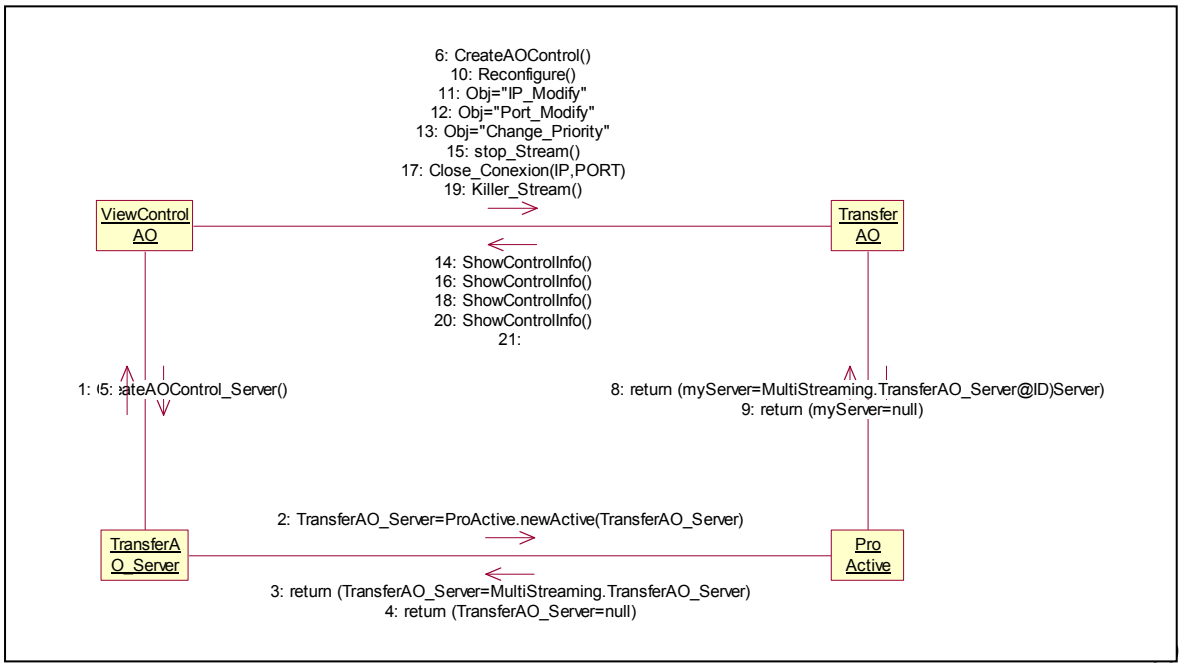


Figura IV.30 Diagrama de Colaboración de Request_ofControl()

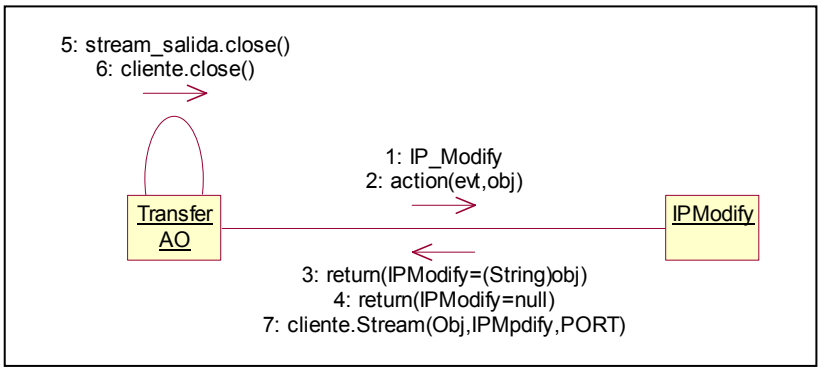


Figura IV.31 Diagrama de Colaboración de IPModify()

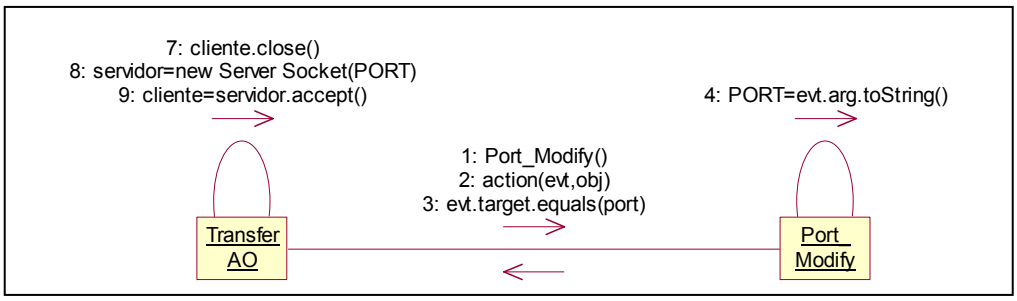


Figura IV.32 Diagrama de Colaboración de Port_Modify()

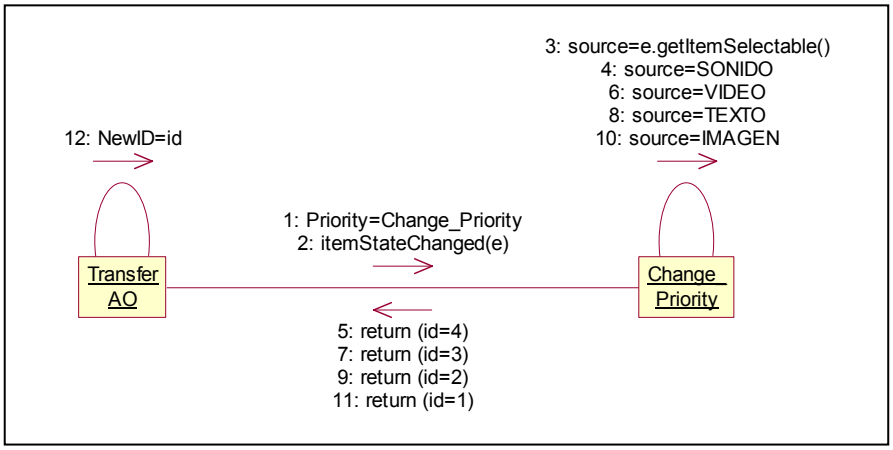


Figura IV.33 Diagrama de Colaboración de Change_Priority()

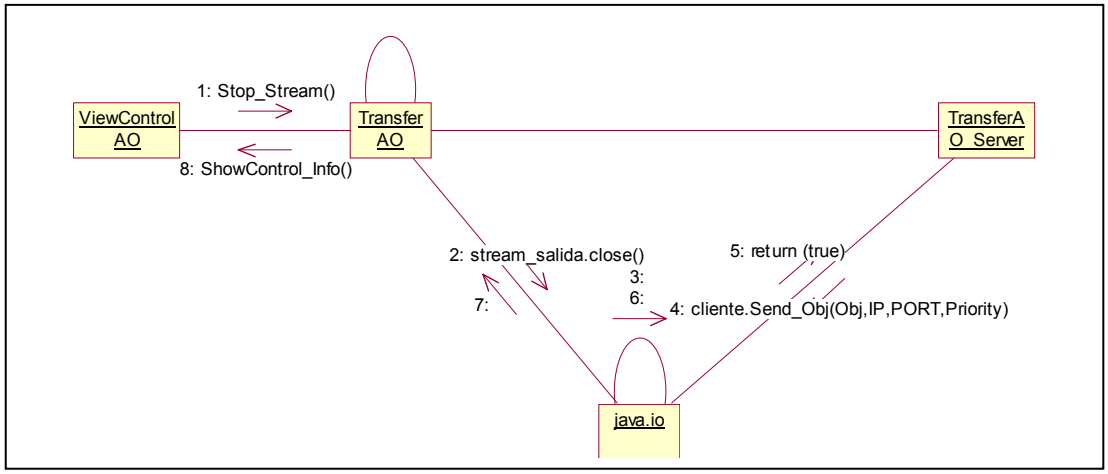


Figura IV.34 Diagrama de Colaboración de Stop_Stream()

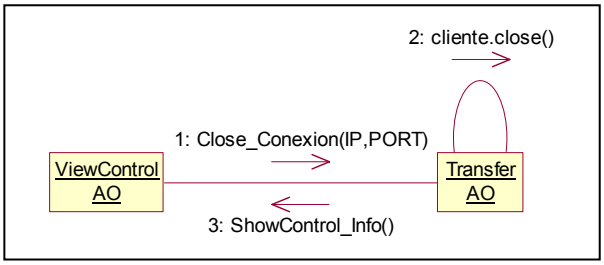


Figura IV.35 Diagrama de Colaboración de Close_Conexion()

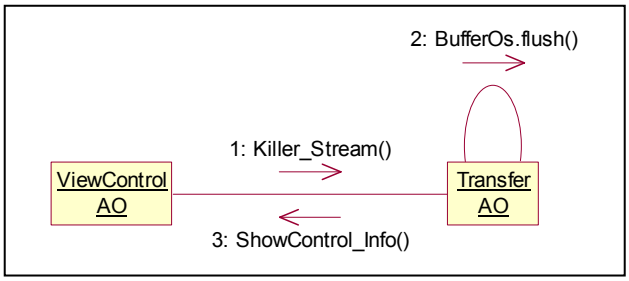


Figura IV.36 Diagrama de Colaboración de Killer_Sream()

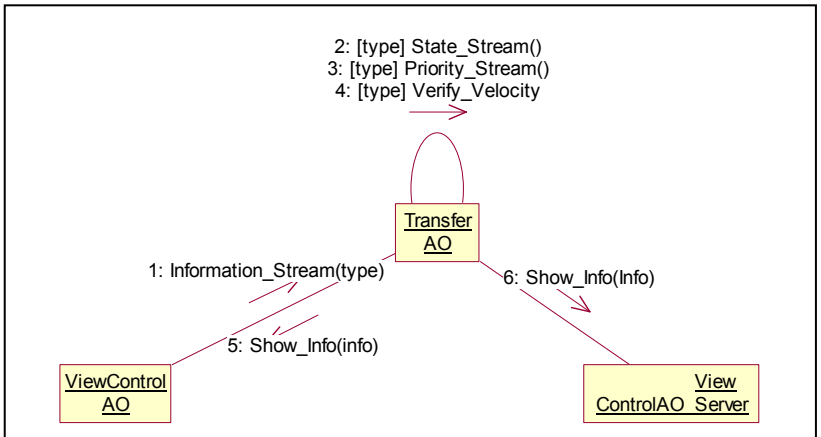


Figura IV.37 Diagrama de Colaboración de Information_Sream()

IV.3.7 Diagrama de Actividades

El Diagrama de Actividad es una especialización del Diagrama de Estado, organizado respecto de las acciones y usado para especificar:
Un método
Un caso de uso
Un proceso de negocio
Un estado de actividad representa una actividad: un paso en el flujo de trabajo o la ejecución de una operación. Un grafo de actividades des cribe grupos secuénciales y concurrentes de actividades.

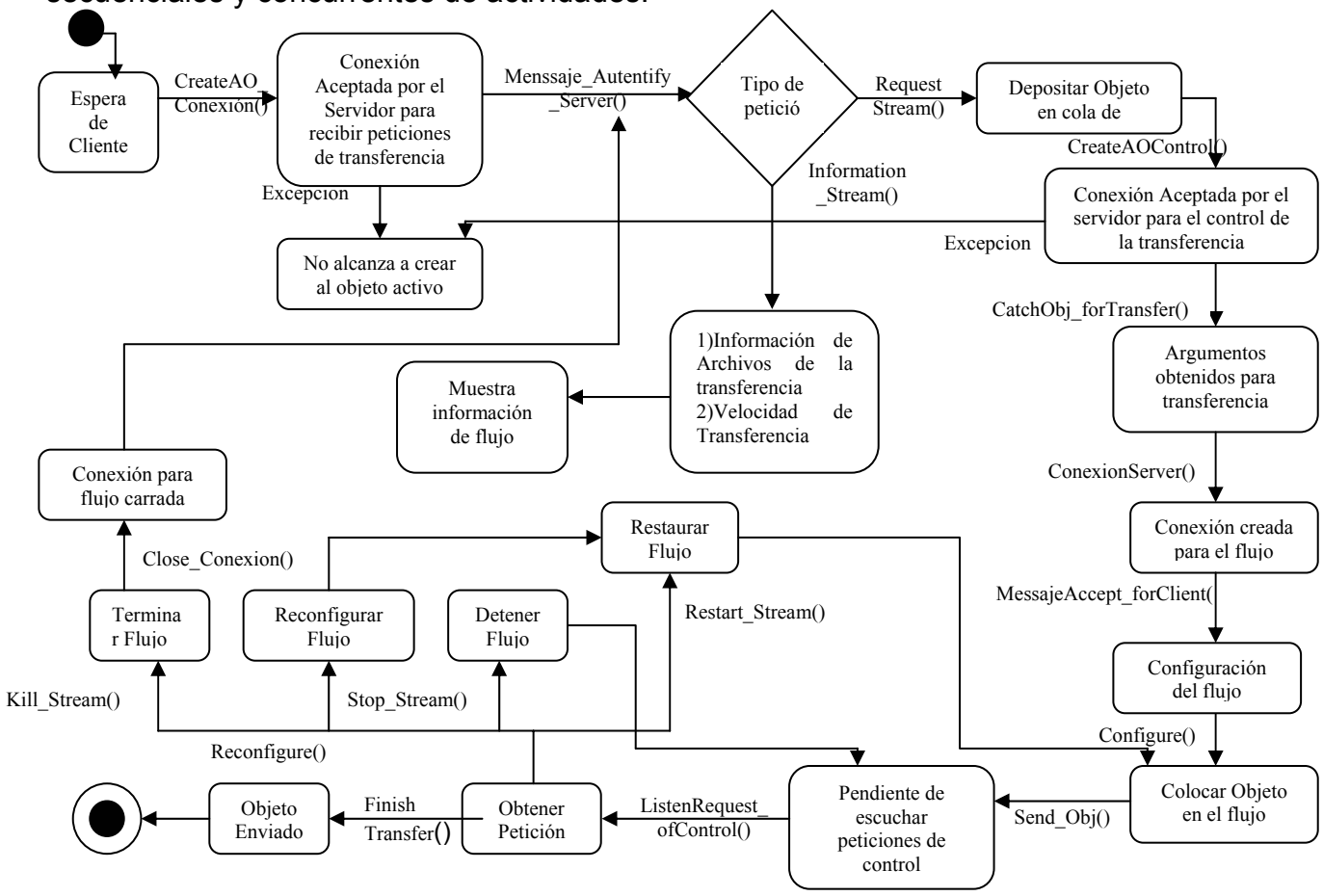


Figura IV.38 Diagrama de Actividades

IV.3.9 Diagrama de Transición de estados

Muestra el conjunto de estados por los cuales pasa un objeto durante su vida en una aplicación, junto con los cambios que permiten pasar de un estado a otro. Los Diagramas de Estado representan autómatas de estados finitos, desde el punto de vista de los estados y las transiciones. Son útiles sólo para los objetos con un comportamiento significativo.

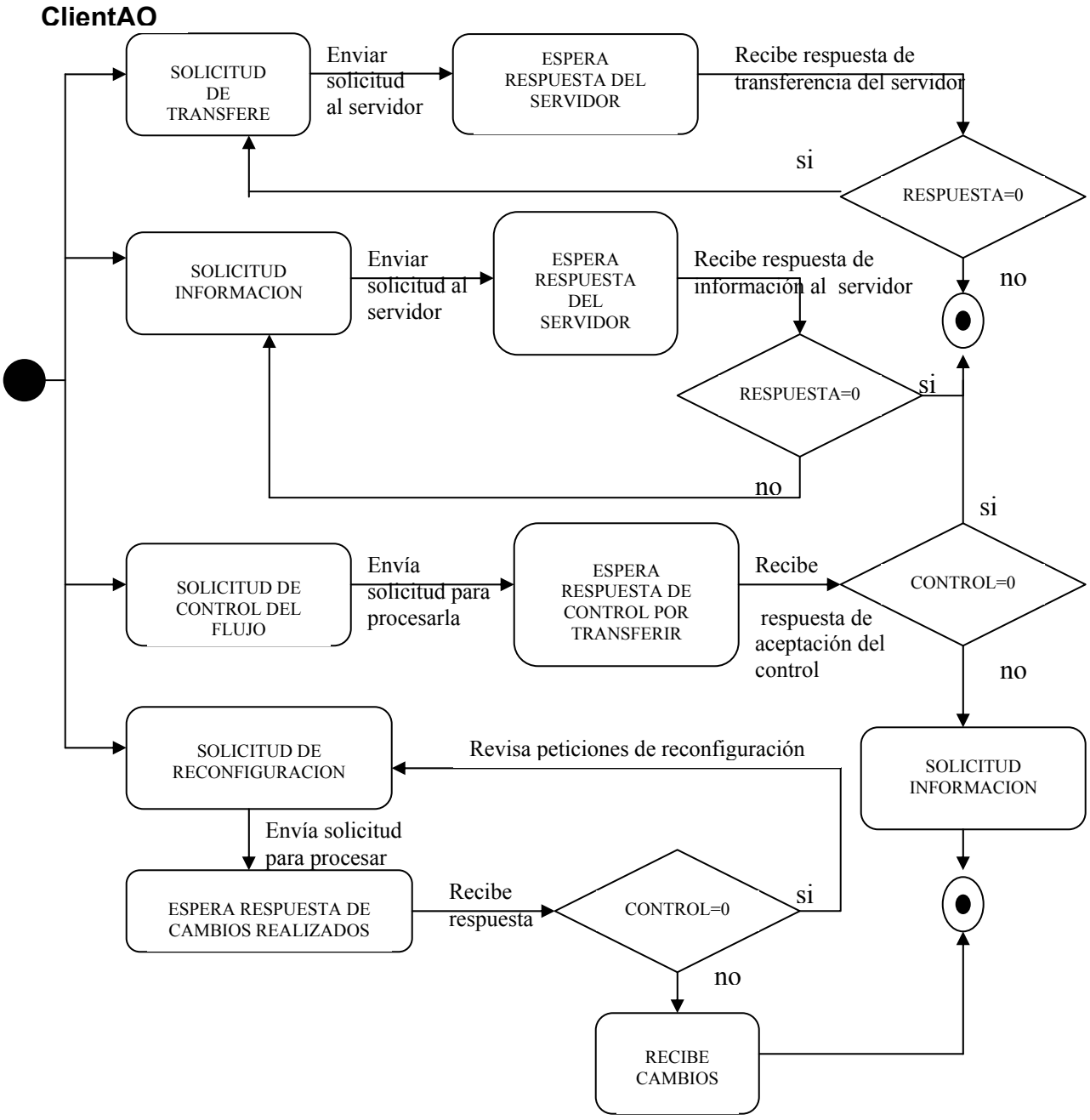


Figura IV.39 Diagrama de estados de AOClient

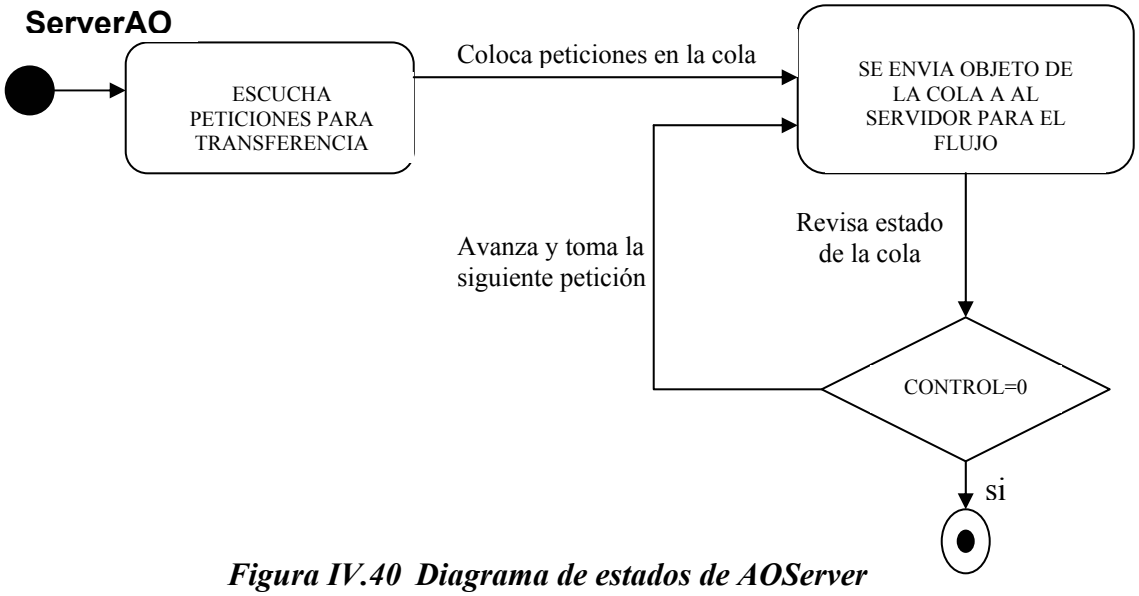


Figura IV.40 Diagrama de estados de AOServer

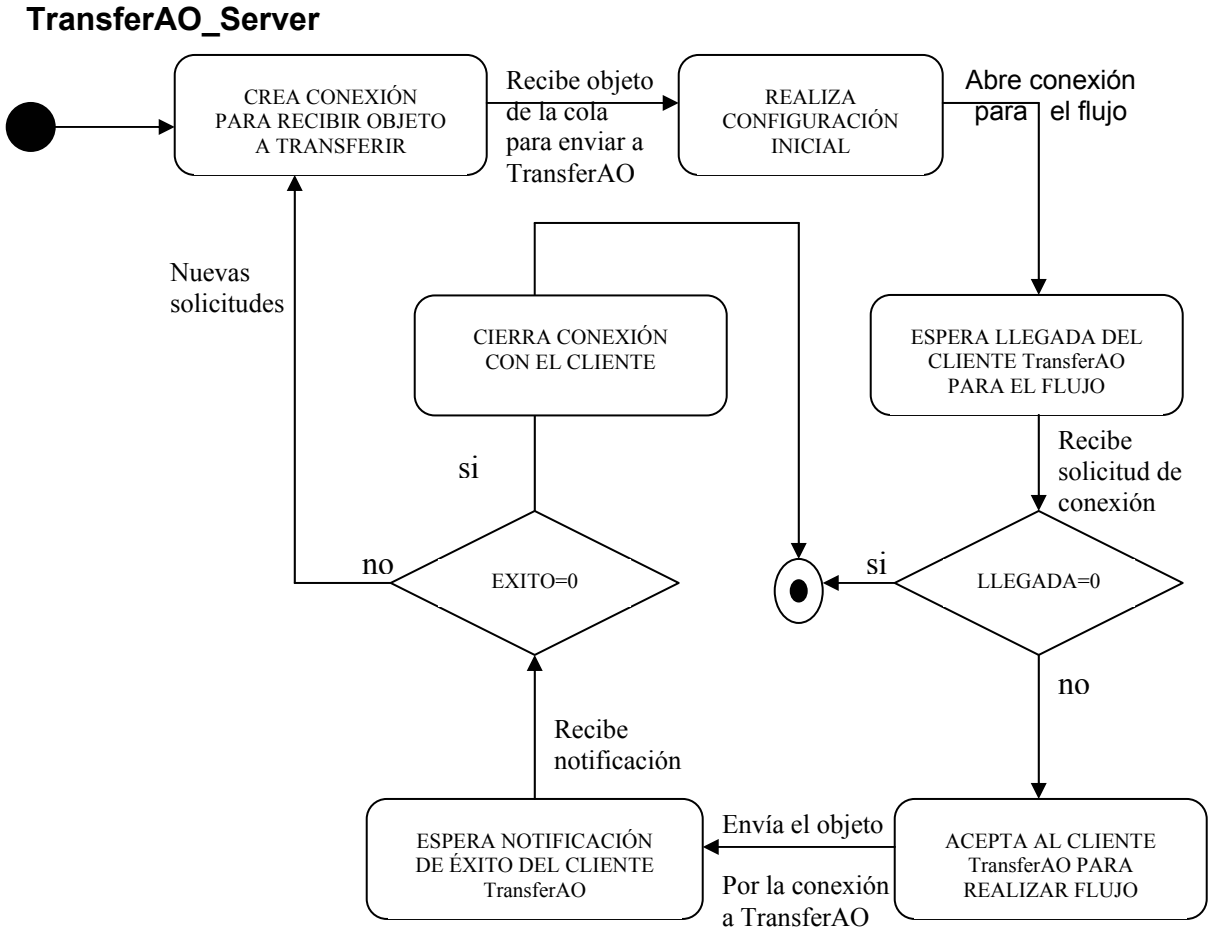


Figura IV.41 Diagrama de estados de AOTransfer_Server

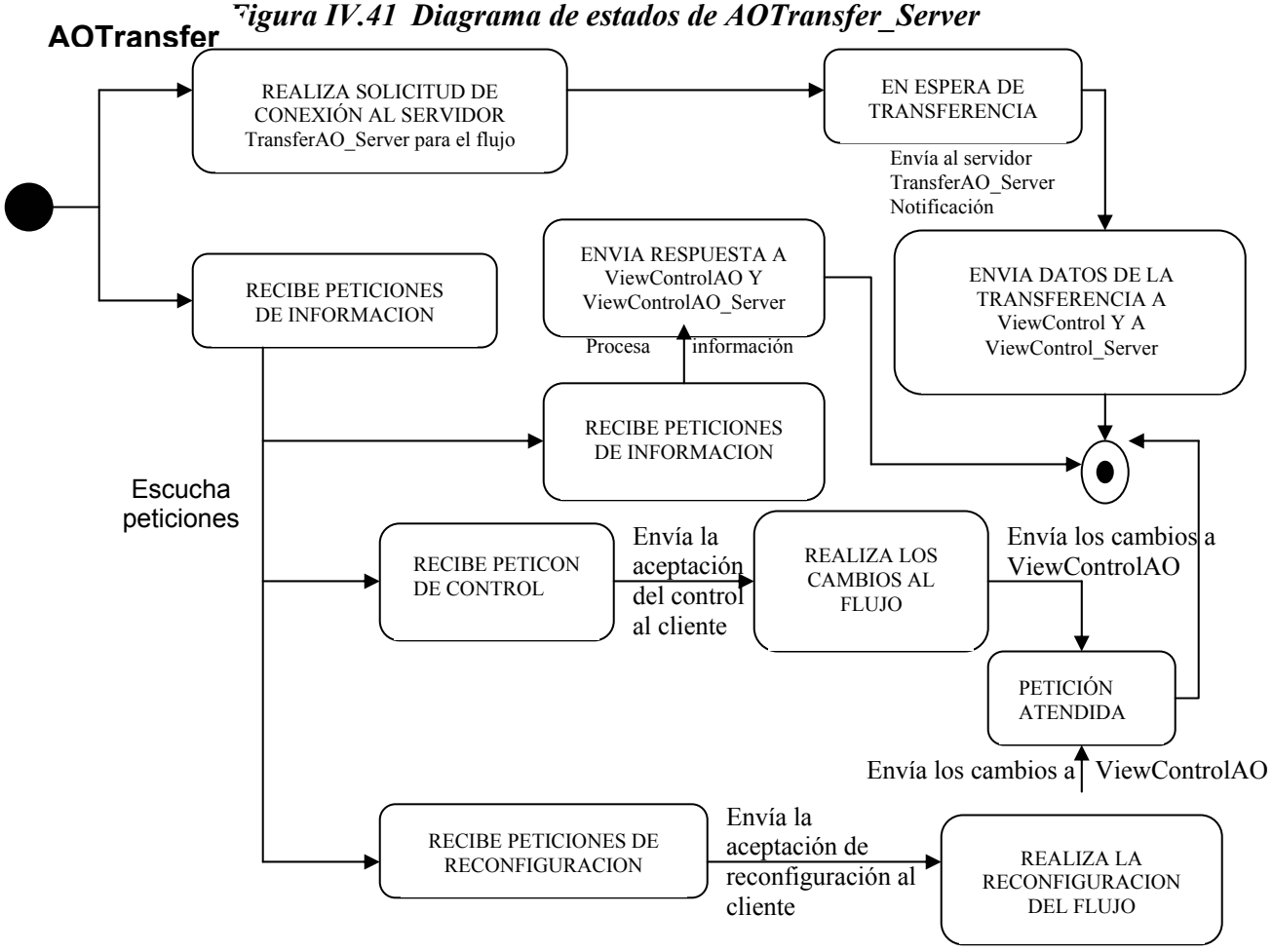


Figura IV.42 Diagrama de estados de AOTransfer

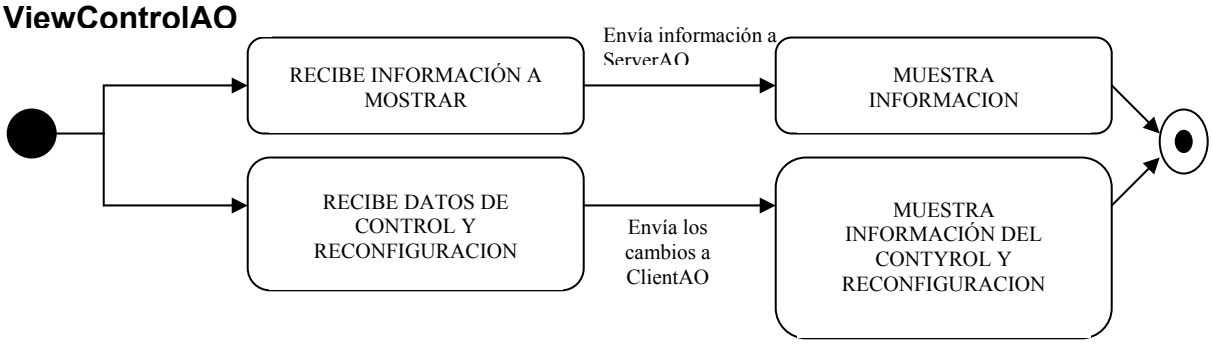


Figura IV.41 Diagrama de estados de ViewControlAO

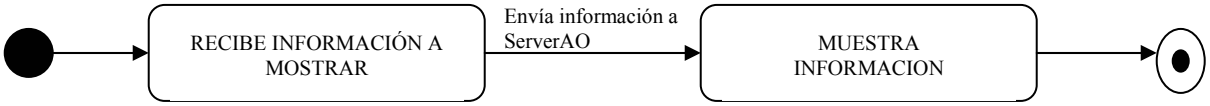


Figura IV.43 Diagrama de estados de ViewControlAO_Server

IV.3.10 Diagramas de Componentes

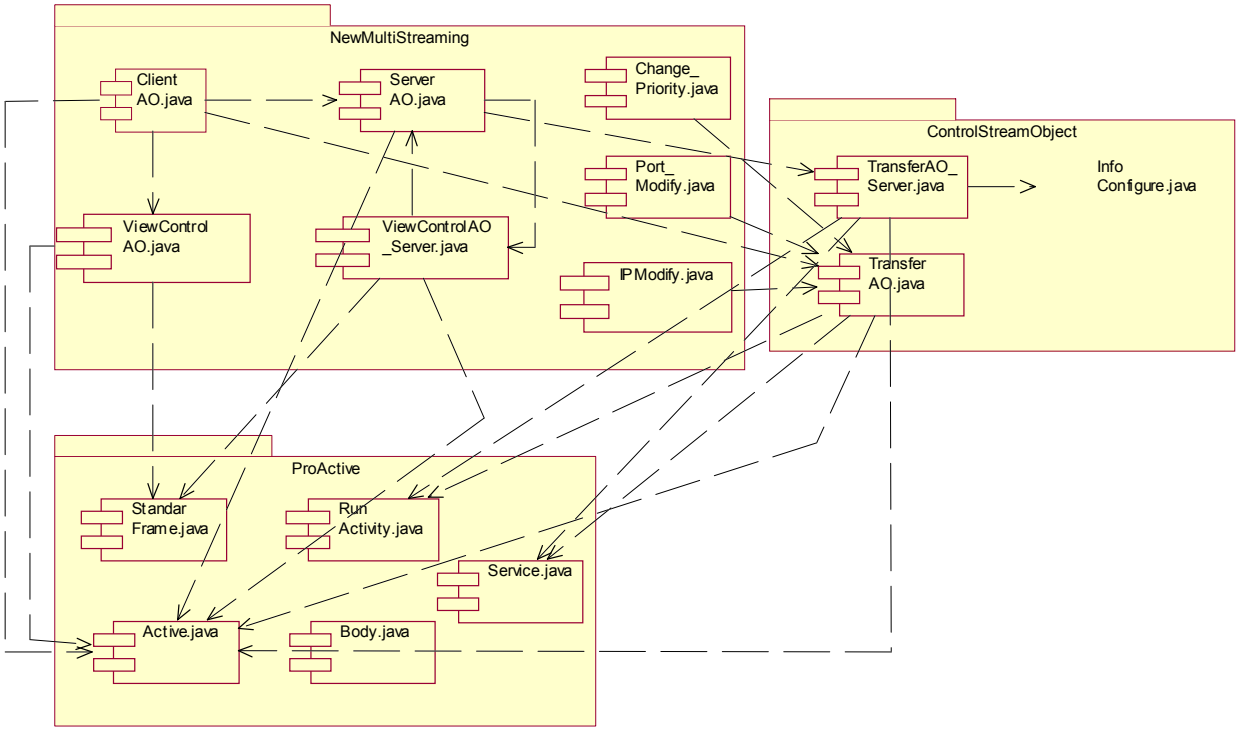


Figura IV.44 Diagrama de Componentes

IV.4 Necesidades

El envío y la recepción de un flujo, es una técnica que toma más y más importancia en la red, y en particular en multimedia. HTTP es un ejemplo de un protocolo con flujo, el cual permite ver el texto antes de que las imágenes sean completamente transmitidas y estén puestas; hasta el momento, los flujos que provee java, así como el envío de un flujo son técnicas que son llevadas a cabo a un nivel relativamente bajo de abstracción, primeramente en el nivel de los protocolos de red (por ejemplo. RTP[9] Protocolo de transporte en tiempo real, RTSP[10] Protocolo de flujo en tiempo real).

Hoy en día las necesidad de definir modelos y técnicas para flujo de objetos con un alto nivel de abstracción de la programación con lenguajes orientados a objetos, es mucha, y en particular en java. La idea fundamental es permitir el flujo de objetos a un nivel alto de abstracción, evitando que los programadores se preocupen por detalles de implementación relacionados con dicho flujo. Se debería generar una infraestructura genérica que permita simplemente invocar los métodos adecuados y obtener un flujo transparente de objetos en un ambiente paralelo y distribuido, lo es cual ya se ha logrado en este trabajo de tesis.

Capítulo V Ejemplo de utilización de la infraestructura

org.objectweb.MultiStreaming

Este es el paquete obtenido de este tema de tesis, el cual proporciona la clase ***ControlStreamObject*** para la creación de Servidores y Clientes de transferencia de objetos. La clase pivote es la clase ***MultiStreaming*** que contiene los métodos para crear los objetos activos servidor y cliente para el control del flujo. Cuando un objeto servidor de transferencia es creado, éste proporciona el camino para atender las respuestas provenientes de un cliente de transferencia.
Los métodos con los que cuenta la clase ***MultiStreaming*** son:

Control del Flujo	**Información del Flujo**
Killer_Stream()	State_Stream()
Close_Conexion()	Priority_Stream
Reconfigure()	Verify_VelocityTransfer()
Stop_Stream	
Restart_Stream()	
Reinitialize()	

La Figura V.1 se muestra un ejemplo de la instanciación de un objeto cliente para el control del flujo.

```
package pruebas;

import org.objectweb.MultiStreaming.*;
import org.objectweb.proactive.ProActive;
import org.objectweb.proactive.Service;

public class Prueba implements org.objectweb.MultiStreaming.Reconfigure, java.io.Serializable {
{
            //Constructor vacío para ProActive
  public Prueba() {
  }

  public static void main(String[] args)
  {
    Prueba cliente=(Prueba)org.objectweb.MultiStreaming.ControlStreamObject.TransferAO
                    (Prueba.class.getName(), null);
  }
}
```

Figura V.1 Ejemplo de la instanciación de un objeto cliente

De la misma forma se hace la instanciación de un objeto servidor, como es notable es muy sencillo hacer uso del paquete *MultiStreaming* así como de sus métodos, dándole con esto a ProActive las primitivas necesarias para la utilización de flujos de objetos, y el control del mismo.

Como se ve en la figura anterior, se ahorra una gran cantidad de líneas de programación al no tener que hacer uso de los diferentes tipos de flujos con los que cuenta Java; dándole así al programador la capacidad de no preocuparse en como realizar dichos flujos, como lograr la comunicación, o preocuparse por la pérdida de objetos etc. Y proporcionándole con esto todas las ventajas con las que cuenta ProActive, como la capacidad de migrar de forma transparente para el usuario entre algunas otras.

Es sencillo hacer uso de este paquete, simplemente con invocar a los métodos adecuados y obtendrá un flujo transparente de objetos en un ambiente paralelo y distribuido.

```
package pruebas;

import org.objectweb.MultiStreaming.*;
import org.objectweb.proactive.ProActive;
import org.objectweb.proactive.Service;

public class SolicitarObj implements
org.objectweb.MultiStreaming.ControlStreamObject.TransferAO,
                                                    java.io.Serializable {
{
  //Constructor vacío para proActive
  public SolicitarObj() {
  }

  public void TransferAO(Obj,IP,PORT){
  }

  public static void main(String[] args)
  {
   SolicitarObj a = (SolicitarObj)
org.objectweb.MultiStreaming.ControlStreamObject.TransferAO(SolicitarObj.class.getNam
e(), null);
  }
}
```

Figura V.2 Programas para solicita una transferencia en un ambiente distribuido

```
package pruebas;

import org.objectweb.MultiStreaming.*;
import org.objectweb.proactive.ProActive;
import org.objectweb.proactive.Service;

public class TransferirObj implements
org.objectweb.MultiStreaming.ControlStreamObject.TransferAO_Server,

java.io.Serializable {
{
            //Constructor vacío para proActive
  public TransferirObj() {
  }

  public void TransferAO_Server(Obj,IP,PORT){
  return Obj
  }

  public static void main(String[] args)
  {
    TransferirObj a = (TransferirObj)
org.objectweb.MultiStreaming.ControlStreamObject.TransferAO_Server(TransferirObj.clas
s.getName(), null);
  }
}
```

Figura V.3 Programas transfiere un objeto en un ambiente distribuido

Como es posible observar tanto en la Figura V.2 y V.3 no se tiene que realizar programación extra para indicar que se transfiere un objeto de un máquina otra en un ambiente distribuido.

Conclusiones

La biblioteca ProActive ha sido desarrollada totalmente en el lenguaje Java y ha tenido como propósito proporcionar una infraestructura genérica para el desarrollo de aplicaciones paralelas, distribuidas y concurrentes. En ese marco, este trabajo también se ha desarrollado usando funciones estándar de java, y de esta manera se ha respetado la flexibilidad que podría tener ProActive.
ProActive PDC es una biblioteca sumamente poderosa, que permite de forma transparente el cómputo paralelo, distribuido y concurrente, y que se encuentra continuamente en crecimiento, sin embargo, carece de primitivas básicas para el flujo de objetos. De hecho, esta carencia motivó este trabajo de tesis. Aunque es posible realizar flujo de objetos en ProActive, usando la biblioteca estándar de java, su uso en aplicaciones que requieren transferencias masivas de objetos es complicado. Existe una serie de elementos a ser considerados en el proceso de manipulación de flujos para múltiples objetos de manera concurrente.

De esta manera, en esta tesis se ha analizado, diseñado e implementado una infraestructura genérica para el flujo de objetos en ProActive. Tanto el análisis como el diseño fueron realizados haciendo uso del lenguaje de modelado unificado (UML). Es posible observar, que existe un alto grado de detalle en cada uno de los diagramas.

El flujo de objetos es básicamente un mecanismo de transferencia de elementos genéricos (llamados objetos), los cuales son enviados mediante el uso de servidores de transferencia, y están basados en el concepto de objetos activos de ProActive.

La administración del flujo de objetos por parte del usuario, es sin duda el núcleo del diseño de esta infraestructura. El usuario será capaz de controlar la transferencia de múltiples objetos de manera concurrente mediante acciones de cancelación, restauración o reinicialización de flujo, así como el cambio de prioridades de envío, cambio de configuración en los puntos de transferencia; como es el caso del cambio del IP y del Puerto de comunicación. Así también, es posible verificar el estado del flujo y la velocidad de transferencia. El objetivo final obtenido es que el usuario es capaz de controlar remotamente la transferencia.

La política de envío de múltiples objetos a través de un flujo, recae principalmente en el uso de prioridades por objetos. De esta manera, es posible optimizar el canal

de transferencia, enviando objetos con diferentes prioridades. Así, un archivo de video o sonido podría tener una mayor prioridad que uno de texto relacionado por ejemplo a un correo electrónico.

Una ventaja adicional se encuentra en el comportamiento de la infraestructura en el caso de pérdida temporal de enlace, ya que, por la naturaleza de ProActive, es posible mantener comunicación asíncrona bajo el uso de objetos futuros.

La construcción de una infraestructura para el flujo de múltiples objetos mediante un solo canal de transmisión, da la posibilidad incluso de realizar tareas de transferencia paralela con grupos, únicamente por medio de la instanciación de un nuevo objeto servidor de transferencia.

El hecho de haber desarrollado esta infraestructura usando únicamente bibliotecas de Java estándar, garantiza que podrá ser utilizada en ambientes heterogéneos. Particularmente, en este proyecto se usó un ambiente de equipos heterogéneos compuesto por una computadora Intel con sistema operativo Windows XP, una computadora Intel con sistema operativo Linux (RedHat 7), una computadora con procesador SPARC (SUN) con sistema operativo Solaris (versión 9) y una computadora con procesador SPARC (SUN) con sistema operativo Linux (RedHat 7.1).

Las restricciones obtenidas después de la utilización de ProActive en un ambiente heterogéneo son las siguientes:

- Es necesario utilizar la misma versión de Java en cada una de las computadoras.
- Es necesario usar la misma versión de ProActive en cada una de las computadoras.

Como se puede observar, los problemas son mínimos, comparados con las bondades en el uso de una infraestructura genérica para el cómputo paralelo, distribuido y concurrente.

La propuesta realizada en este trabajo de tesis, servirá como fundamento en la generación de bibliotecas especializadas en la transferencia de objetos en ambientes paralelos, distribuidos y concurrentes. Las necesidades pueden ser variantes de ambiente a ambiente y por lo tanto, una de las perspectivas de este trabajo será proporcionar una mayor flexibilidad.

La difusión de el conjunto de primitivas desarrolladas correrá por cuenta del grupo de soporte de ProActive PDC, a partir de la liberación de este trabajo. No queda duda, que el aporte realizado en términos de transferencia de objetos bajo ProActive, ha sido valioso. Adicionalmente, se ha logrado colaborar a nivel internacional, proporcionando un enlace entre dos universidades (BUAP-NICE) que en otro caso no hubiera sido posible.

Es necesario aún realizar experimentos para la transferencia de múltiples objetos en ambientes distribuidos en colaboración con otras actividades. Se espera que este tipo de pruebas puedan deducir un comportamiento de la infraestructura desarrollada, así como la posibilidad de mejorar los métodos y las clases. Se desconoce parcialmente al momento, el rendimiento o facilidad de desarrollo que pueda aportar el conjunto de primitivas desarrolladas, por lo que habrá que extender esta línea de trabajo a nuevos estudiantes interesados en ésta área de los sistemas distribuidos.

Finalmente, es importante mencionar que este trabajo ha sido presentado en el sexto Congreso Internacional en Telecomunicaciones e Informática llevado a cabo durante el mes de Mayo del 2004 en la Ciudad de Cancún, México. El trabajo ha sido publicado en las memorias del evento [11], así como en el “Transactions on Communications” [12].
De igual manera, el trabajo ha sido aceptado en la 3ra Conferencia Iberoamericana en Sistemas, Cibernética e Informática CISCI 200 que se llevará a cabo en el mes de julio del 2004, en la ciudad de Orlando, Florida, Estados Unidos.

Bibliografía

[1] Andrew S.Tanebaum, “Sistemas Operativos Distribuidos”, Prentice Hall, 1er. Edición.

[2] Agustín Froute,”Java 2 Manual de usuario y tutorial”, Alfa omega Ra-Ma.

[3] Laura Lemaq y Rogers Cadenhead,”Aprendiendo Java 2”, Prentice Hall.

[4] Caramazana Alberto, “Estándar CORBA C++/Java (ORBacus)”, Universidad Pontificia de Salamanca Madrid, Facultad de Informática,

[5] Mark Grand and Jonathan Knudsed, “Java Fundamental Classes Reference“, OREILLY.

[6] Duc A. Tran, Kien A. Hua and Tai Do, *ZigZag: An efficient Peer-to-Peer scheme for Media Streaming,* Infocom 2003.

[7] HP Labs: Research: *MMSL: Projects: Streaming Media Systems*, URL= http://www.hpl.hp.com/research/mmsl/projects/streaming.html.

[8] Tokunaga, E., Zee A., *Object-Oriented Middleware Infrastructure for Distributed Augmented Reality*, ISORC 2003.

[9] Henning Sinul Zrinne, Stephem L. Casner, Ron Frederick, Van Jacobson, Internet Engineering Task Force, 2001.

[10] H. Shulzrinne, A.Rao, R. Lanphier Columbia U./Netscape/Real Networks, 1998.

[11] David Pinto, Adriana Beristain & Margarita Márquez: "A Generic Infrastructure for Object Streamming and Mobile Agents on ProActive". On Proceedings of TELEINFO Conference: ISBN: 960-8052-98-X, 2004.

[12] David Pinto, Adriana Beristain & Margarita Márquez: "A Generic Infrastructure for Object Streamming and Mobile Agents on ProActive". TELE-INFO'04: Transactions on Communications: ISSN: 1109-2742, Pag. 71-75, 2004.

Referencias

Internet 1 http://www.fisica.uson.mx/carlos/WebServices/WSOverview.htm

Internet-2 http://ict2.udlap.mx/people/odette/html/tec_alt_intro.html

Internet-3 http://mail.udlap.mx/~tesis/lis/hernandez_c_ej/capitulo0.pdf

Internet-4 http://www.acm.org/crossroads/espanol/xrds8-3/intro83.html

Internet-5 http://www.cemisid.ing.ula.ve/area-compparall.html

Internet-6 http://mail.linux.org.mx/pipermail/ayuda/2002-November/005493.html

Internet-7 http://middleware.objectweb.org/

Internet-8 http://www.m-w.com/

Internet-9 http://www.arrakis.es/~dmrq/beej/theory.html

Internet-10 http://www-sop.inria.fr/oasis/ProActive/

Glosario de Términos

Cliente-servidor	Es una arquitectura que permite al usuario en una máquina, llamado el cliente, requerir algún tipo de servicio de una máquina a la que está unido, llamado el servidor, mediante una red. Aunque clientes y servidores suelen verse como máquinas separadas, pueden, de hecho, ser dos áreas separadas en la misma máquina.
RPC	Remote Procedure Calls (llamados de procedimientos remotos) Los RPC permiten a una función remota ser llamada como si se tratara de una local.
ORPC	RPC basados en objetos
ONC	Open Network Computing
NFS	Network File System
NCS	Network Computer System
OSF	Open Software Foundation
OMG	Object Management Group
CORBA	Common Object Request Broker Architecture
ORB	Object Request Broker
DCOM	Distributed Component Object Model
OLE	Object Linking and Embedding
ORG	IIOP Internet Inter ORB
JDK	
RMI	Remote Invocation Method “Método de Invocación Remota”, es un protocolo ORPC llamado Protocolo de Método Remoto de Java
JRMP	Java Remote Method Protocol “Protocolo de Método Remoto de Java”
COM+	El sucesor de DCOM.
J2EE	Java 2 Platform Enterprise Edition
SOAP	Simple Object Access Protocol
groupware	Servidores
full-duplex,	la comunicación ambos sentidos a la vez