Universidad Politécnica de Valencia

Facultad de Informática

Proyecto Final de Carrera

"Acceso a datos multimedia utilizando los métodos de invocación remota (RMI) de Java"

Director proyecto:
D. Pietro Manzoni.
Realizado por:
D. Jacinto Navarro González.

Indice:

Pág.

1. Introducción. 3
2. Herramientas Java utilizadas. 6
   2.1. Remote Method Invocation (RMI). 6
      2.1.1. Descripción de RMI. 6
      2.1.2. Creación de aplicaciones distribuidas con RMI. 8
   2.2. Java Database Connectivity (JDBC) 10
      2.2.1. Descripción de JDBC. 10
      2.2.2. Utilización de JDBC. 11
   2.3. Java Media Framework (JMF). 13
      2.3.1. Descripción de JMF. 13
      2.3.2. Utilización de JMF. 13
   2.4. Abstract Window Toolkit (AWT). 18
      2.4.1. Descripción de AWT. 18
3. Descripción de la aplicación. 20
   3.1. El servidor. 20
   3.2. El cliente 20
4. Análisis de la aplicación. 23
   4.1. Análisis de la implementación del servidor. 23
   4.2. Análisis de la implementación del cliente. 42
5. Uso de la aplicación. 85
6. Conclusiones. 89
7. Bibliografía. 90

1. Introducción.

El presente documento describe el desarrollo y funcionamiento de la aplicación elaborada para el proyecto final de carrera: "Acceso a datos multimedia utilizando los métodos de invocación remota (RMI) de Java". Dicha aplicación consta de dos componentes bien diferenciados, un servidor de archivos multimedia que proporciona información sobre dichos archivos mediante el acceso a una base de datos, y que los transmite a través de la red con el fin de servir las peticiones que le llegan, y un cliente que accede a la información presente en la base de datos del servidor mediante una serie de métodos de invocación remota ofrecidos por éste, y que recibe dichos archivos procedentes del servidor través de la red, utilizando para ello dos protocolos de transporte distintos.

Posiblemente, el lenguaje de programación más adecuado para el desarrollo de la aplicación descrita sea Java. Java es un lenguaje de desarrollo de propósito general, y como tal es válido para realizar todo tipo de aplicaciones. Pero Java añade una serie de características importantes que lo hacen especialmente atractivo para el desarrollo de una aplicación como la descrita anteriormente. Está fuertemente orientado al desarrollo de aplicaciones para intrarredes, aplicaciones cliente/servidor, aplicaciones distribuidas en redes locales y en Internet, además de proporcionar bastante fiabilidad a las aplicaciones, puesto que puede controlarse su seguridad frente al acceso a recursos del sitema y es capaz de gestionar permisos y criptografía. También es de destacar que Java, tanto su kit de desarrollo de aplicaciones (JDK), como toda la serie de paquetes adicionales que lo complementan, es público, y por tanto gratuito. Finalmente, una de las características más importantes de Java, es que los programas ejecutables, creados por el compilador de Java, son independientes de la arquitectura sobre la que se ejecutan, se ejecutan en una gran variedad de equipos con diferentes microprocesadores y sistemas operativos.

Así pues, una vez descrita la motivación para el empleo de Java en el desarrollo del proyecto, pasaremos a enumerar los principales paquetes del lenguaje que han sido utilizados en la elaboración de la aplicación.

Para el intercambio de información entre el proceso cliente y el proceso servidor se ha utilizado el paquete RMI (Remote Method Invocation). Dicho paquete proporciona una serie de clases y utilidades, para que objetos que se están ejecutando en diferentes máquinas virtuales se puedan comunicar unos con otros, residan o no ellos en la misma máquina física. RMI es similar al concepto de RPC (llamadas a procedimiento remoto). Una llamada a procedimiento remoto es una llamada a una función que se origina en una máquina, se ejecuta en otra, y el resultado de la ejecución es devuelto a la primera máquina. Debido al diseño de RMI, las comunicaciones entre dos objetos Java pueden implementarse con muy poco código, y sencillo de elaborar. Además, RMI, gracias a la serialización de objetos de Java, permite que los objetos Java se puedan

pasar fácilmente entre objetos remotos. Más adelante, en este documento, se describirá con más detalle el funcionamiento y el uso del paquete RMI de Java.

Para la organización de los archivos multimedia existentes en el servidor, y todos los datos referentes a ellos se ha utilizado una base de datos. Dicha base de datos almacena la información de los archivos multimedia que el servidor ofrece a los clientes (nombre del archivo, tamaño, tipo de archivo, fecha, hora, etc.). Para el acceso a la información almacenada en la base de datos se utiliza el paquete JDBC (Java DataBase Connectivity) de Java. Dicho paquete ofrece una serie de clases para acceder a bases de datos relacionales directamente desde código Java. Antes de la aparición de JDBC, la única manera de acceder a una base de datos desde un programa Java, era implementando el acceso a la base de datos en C, y usando métodos nativos para interactuar entre el código C y el código Java. JDBC está pensado como una API de bajo nivel lo suficientemente potente como para poder llevar a cabo la mayor parte de las operaciones posibles sobre bases de datos relacionales comerciales, pero al mismo tiempo lo suficientemente sencilla como para que la mayor parte de los programadores pueda utilizarla sin dificultades. JDBC se verá con más detenimiento posteriormente en este documento.

Los archivos multimedia son enviados por el servidor a través de la red, pero el cliente necesita una herramienta que interprete los datos que le llegan, y los procese, teniendo en cuenta que se trata de datos multimedia. Para realizar dicho trabajo se utiliza el paquete JMF (Java Media Framework) de Java. JMF especifica una arquitectura conducida por eventos para el desarrollo de aplicaciones que soportan imagen y sonido. Proporciona un completo soporte para manipular archivos multimedia. El soporte es proporcionado mediante el inicio, parada, sincronización, y acciones similares sobre archivos multimedia. La reproducción local o remota de datos multimedia incluye sonido e imagen de una gran variedad de formatos (MIDI, WAV, MPEG, AVI, MP3, RMF, etc.). JMF es uno de los paquetes básicos en el desarrollo de este proyecto, por las características de éste, y como tal, se analizará con mayor profundidad en posteriores secciones de este documento.

El proceso cliente ha de interactuar directamente con el usuario (no ocurre lo mismo con el proceso servidor), así pues, es necesaria una herramienta para crear la interfaz gráfica de usuario que permita que éste interactue con el proceso cliente. Dicha herramienta es el AWT (abstract windows toolkit) de Java. AWT es una interfaz independiente de la plataforma que permite el desarrollo de interfaces gráficas de usuario que se pueden ejecutar en la mayor parte de las plataformas. Está formado por un amplio conjunto de clases que permiten la inclusión en las interfaces de usuario creadas, de la mayor parte de los objetos presentes en cualquier interfaz de usuario (ventanas, botones, etiquetas, diálogos, menús, imágenes, etc.). Posteriormente se entrará a definir AWT y su uso con mayor detenimiento.

El intercambio de información entre el proceso cliente y el servidor se realiza, como se ha comentado antes, mediante métodos de invocación remota (RMI), pero la transmisión de los datos multimedia desde el servidor hacia al cliente ha de hacerse empleando un protocolo de transporte de datos a través de la red. En la aplicación implementada se han utilizado dos protocolos de transporte de datos, el TCP (Transmission Control Protocol), y el UDP (User Datagram Protocol). El primero de ellos es un protocolo orientado a conexión, es decir, se basa en una conexión establecida en la que queda fijado el destino de la conexión desde el momento de abrirla y hasta el momento de cerrarla, mientras que el segundo es un protocolo sin conexión, es decir, envía mensajes individuales con el destino grabado en cada uno de ellos, donde no son necesarias las fases de establecimiento y liberación de conexión alguna. Ambos protocolos han tenido que integrarse en la programación relacionada con JMF para que el proceso cliente pueda interpretar correctamente los datos multimedia que vienen por la red procedentes del servidor y pueda reproducirlos.

2. Herramientas Java utilizadas.

2.1. Remote Method Invocation (RMI).

Como se ha comentado antes, RMI es un mecanismo de Java que permite la comunicación entre dos objetos, estando éstos presentes cada uno en una máquina virtual distinta en la misma máquina física, o incluso en máquinas virtuales distintas presentes cada una en una máquina física distinta.

2.1.1. Descripción de RMI.

Las aplicaciones RMI están compuestas casi siempre de dos programas separados: un programa cliente y uno servidor. Una aplicación servidor típica crea varios objetos remotos, hace que las referencias a ellos sean accesibles y espera que los clientes invoquen métodos sobre esos objetos remotos. Una aplicación cliente típica obtiene una referencia remota a uno o varios de los objetos remotos registrados en el servidor y entonces invoca métodos sobre dichos objetos. RMI proporciona el mecanismo por el cual el cliente y el servidor se comunican y se pasan información entre ellos. Las aplicaciones citadas reciben a menudo el nombre de aplicaciones de objetos distribuidos. Una aplicación de objetos distribuidos necesita básicamente lo siguiente para su funcionamiento:

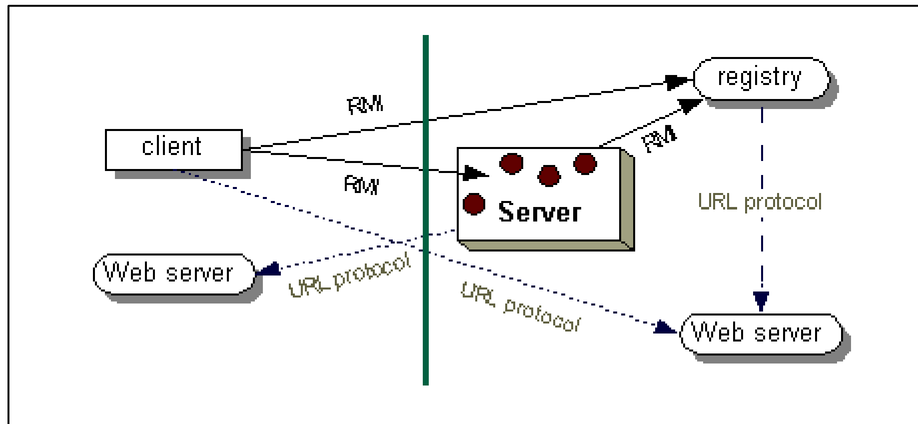
- Localizar objetos remotos: las aplicaciones pueden usar uno de dos mecanismos para obtener referencias a objetos remotos. Una aplicación puede registrar sus objetos remotos con la utilidad de RMI para el registro de objetos, el *rmiregistry*, o la aplicación puede pasar y devolver referencias a objetos remotos como parte de su funcionamiento normal.

- Comunicarse con objetos remotos: los detalles referentes a la comunicación entre objetos remotos son manejados totalmente por RMI. Para el programador, la comunicación remota es, en su uso, como una invocación de métodos estándar de Java.

- Cargar los byte-codes de las clases de los objetos que son pasados: puesto que RMI permite pasar objetos a objetos remotos, RMI proporciona los mecanismos necesarios para el código de un objeto, así como para transmitir sus datos.

El siguiente esquema representa una aplicación distribuida RMI que usa el registro para obtener una referencia a un objeto remoto. El servidor utiliza el registro para asociar un nombre con un objeto remoto. El cliente busca el objeto remoto por su nombre en el registro del servidor y entonces invoca un método sobre él. También se observa como el sistema RMI usa un servidor web existente con el fin de cargar byte-

codes de clases, del servidor al cliente y del cliente al servidor, para cuando se necesitan objetos.



Uso de RMI en una aplicación distribuida

Una de las principales características de RMI es la posibilidad de descargar los byte-codes de la clase de un objeto, si la clase no está definida en la máquina virtual Java receptora. Los tipos y el comportamiento de un objeto previamente disponible sólo en una única máquina virtual, pueden ser transmitidos hacia otra, posiblemente remota, máquina virtual. Todo esto permite que nuevos tipos sean introducidos en una máquina virtual remota, extendiendo así el comportamiento de una aplicación dinámicamente.

Como cualquier otra aplicación construida usando Java RMI está compuesto de interfaces y clases. Las interfaces definen métodos, y las clases implementan los métodos definidos en las interfaces, y posiblemente, definen métodos adicionales. En una aplicación distribuida se asume que algunas implementaciones residen en diferentes máquinas virtuales. Los objetos que poseen métodos que pueden ser invocados entre diferentes máquinas virtuales son objetos remotos. Un objeto pasa a ser remoto implementando una interfaz remota, la cual tiene las siguientes características:

- Una interfaz remota extiende la interfaz Java *java.rmi.Remote*.
- Cada método de la interfaz declara *java.rmi.RemoteException* en su claúsula *throws*, además de alguna excepción más específica de la aplicación.

RMI trata un objeto remoto de una forma diferente a como trata un objeto no remoto, cuando el objeto es pasado de una máquina virtual a otra. En lugar de hacer una copia de la implementación del objeto remoto en la máquina virtual receptora, RMI pasa un *stub* remoto por cada objeto remoto que se desea ofrecer. El *stub* actua como el representante local, o *proxy,* para el objeto remoto y se trata básicamente de una referencia remota para ser usada por el objeto invocador. El objeto invocador llama a un método en el *stub* local, y es éste último el responsable de llevar a cabo la llamada al método sobre el objeto remoto. Un *stub* para un objeto remoto implementa el mismo conjunto de interfaces remotas que el objeto remoto implementa. Esto permite a un *stub*

considerar como suya alguna de las interfaces que el objeto remoto implementa. Sin embargo, esto también significa que sólo los métodos definidos en una interfaz remota, están disponibles para ser llamados por un objeto presente en la máquina virtual receptora.

2.1.2. Creación de aplicaciones distribuidas con RMI.

Cuando se utiliza RMI para implementar una aplicación distribuida han de seguirse básicamente los siguientes pasos:

1. Diseñar e implementar los componentes de la aplicación distribuida: en primer lugar hay que decidir como va a ser la arquitectura de la aplicación y determinar que componentes han de ser objetos locales y cuales de ellos han de ser accesibles de forma remota. Para los objetos que han de ser accesibles de forma remota se han de definir las interfaces remotas. Una interfaz remota especifica los métodos que pueden ser invocados remotamente por un cliente. Parte del diseño de tales interfaces remotas consiste en determinar que objetos locales serán utilizados como parámetros y valores de retorno de los métodos definidos. Una vez definidas las interfaces remotas es necesario implementar los objetos remotos. Un objeto remoto debe implementar una o más interfaces remotas. La clase del objeto remoto puede incluir implementaciones de otras interfaces (ya sean éstas locales o remotas) y otros métodos (que son accesibles sólo localmente). Finalmente, para completar la aplicación distribuida, es necesario implementar los clientes. Clientes que usarán objetos remotos de forma similar a como usan sus objetos locales.
2. Compilar los fuentes y generar los *stubs*: éste es un proceso que consta de dos partes. En primer lugar ha de usarse el compilador *javac* de Java para compilar los archivos con los fuentes que contienen la implementación de las interfaces remotas y sus implementaciones, las clases propias del servidor, y las clases del cliente. En segundo lugar ha de usarse el compilador *rmic* para crear los *stubs* de los objetos remotos.
3. Hacer que las clases necesarias sean accesibles para los objetos distribuidos en la red: en una aplicación distribuida, como se ha comentado antes, los objetos se pueden encontrar ubicados en máquinas físicas distintas, por tanto, es necesario que los archivos que contienes las clases correspondientes a los *stubs* sean accesibles localmente por los clientes que desean tener acceso a algún objeto remoto. Una buena solución para una aplicación distribuida completa, sería disponer de un servidor web, desde el cual los clientes pudiesen descargar los archivos correspondientes a los *stubs* que necesitasen.
4. Iniciar la aplicación distribuida: iniciar la aplicación incluye ejecutar el registro de objetos remotos (*rmiregistry*) de RMI, el servidor, y el cliente. El registro de objetos remotos es un servidor de nombres de objetos que permite

a los clientes remotos obtener una referencia a un objeto remoto. Para iniciar el registro en el servidor, es necesario ejecutar el comando *rmiregistry*. Este comando no produce salida alguna y se ejecuta en segundo plano. En windows se inicia el registro de objetos remotos simplemente ejecutando el siguiente comando: *start rmiregistry.* Por defecto, el registro escucha en el puerto 1099. Para iniciar el registro en un puerto diferente, se ha de especificar el número de puerto en la línea de comandos, así por ejemplo *start rmiregistry 2002* iniciaría el registro de objetos remotos escuchando en el puerto 2002.

Más adelante en este documento se analizará el uso concreto de RMI que se ha hecho en la aplicación creada para este proyecto. Se describirán las interfaces remotas creadas, así como la implementación de dichas interfaces, y el modo en que se han de utilizar éstas.

2.2. Java Database Connectivity (JDBC).

JDBC es el paquete de Java (incluido en el JDK) que proporciona acceso a bases de datos a programas escritos en código Java. Se trata de una herramienta que proporciona un acceso SQL genérico a bases de datos, mediante una interfaz uniforme por encima de una gran variedad de módulos de conectividad a bases de datos.

2.2.1. Descripción de JDBC.

JDBC hace posible el desarrollo de aplicaciones de bases de datos independientes de la plataforma. La única premisa que se impone al fabricante del sistema gestor de bases de datos es que ofrezca un conjunto de clases, escrito en Java, al que se le llama controlador JDBC. Las clases que componen los controladores JDBC son clases abstractas que implementan para un sistema gestor de bases de datos concreto los métodos del paquete de Java *java.sql*. Estos métodos sirven para ejecutar peticiones SQL, que por tanto pueden ser enviadas desde una aplicación Java, ser traducidas por el controlador JDBC al dialecto propio del sistema gestor de bases de datos para el que esté preparado, y ejecutadas por este sistema gestor. A su vez, también se dispone de una serie de clases abstractas para recoger la información que este sistema gestor devuelve.

Los sistemas gestores de bases de datos comerciales soportan un conjunto común de comandos SQL al mismo tiempo que implementan algunos comandos más específicos de cada sistema y que en modo alguno pueden ser considerados un estándar. Frente a esta situación, JDBC actúa permitiendo pasar cualquier tipo de instrucción al sistema gestor de base de datos a través de los controladores JDBC. Esto significa que es posible lanzar a través de JDBC peticiones al sistema gestor de bases de datos que no respeten el estándar SQL y que incluso ni tan siquiera constituyan parte de SQL, sino que pertenezcan a un conjunto de instrucciones propietarias de este sistema. Este funcionamiento permite sacar partido a las funcionalidades propias de un sistema gestor de bases de datos concreto. Además, al no obligar al cumplimiento de una determinada gramática, JDBC abarca potencialmente a un conjunto bastante mayor de sistemas de bases de datos. Así pues, Java, con la inclusión de JDBC, se puede decir que es un entorno perfectamente válido para el acceso y la explotación de un sistema gestor de bases de datos.

Así pues, para que una aplicación escrita en Java se pueda comunicar con una base de datos determinada a través de JDBC, se necesita el controlador JDBC que convierta las órdenes que se le van a transmitir en Java en comandos inteligibles para esa base de datos en concreto. Existen varios tipos de controladores JDBC para conectarse mediante Java a un sistema gestor de bases de datos, pero entre ellos, el posiblemente más utilizado sea el puente JDBC-ODBC. Este es el controlador que se ha empleado en la aplicación del proyecto para que el programa servidor pueda ejecutar las

consultas sobre la base de datos que son demandadas por el cliente. El puente JDBC-ODBC no es más que un controlador JDBC para bases de datos a las que se accede a través del estándar ODBC, de forma que las aplicaciones Java puedan comunicarse con sistemas de bases de datos basados en ODBC. Puesto que ODBC es un estándar sumamente extendido en el mercado, el puente JDBC-ODBC garantiza el acceso a prácticamente cualquier base de datos comercial. La conexión de una aplicación en Java con un sistema gestor de bases de datos se realizará con el puente JDBC-ODBC del siguiente modo: en primer lugar la aplicación se comunica con el controlador JDBC que ha sido previamente cargado. Éste traduce el contenido de la comunicación a operaciones comprensibles por el controlador ODBC que también debe estar cargado en el sistema. Finalmente, el controlador ODBC pasa esa información al sistema gestor de bases de datos en concreto. La respuesta del sistema gestor de bases de datos a la aplicación Java recorre justamente el camino inverso al recién comentado.

2.2.2. Utilización de JDBC.

Todos los programas que empleen JDBC para acceder a una base de datos, deben seguir siempre el mismo esquema de conexión. A continuación se describirán los pasos fundamentales en los que se puede descomponer un acceso completo a bases de datos mediante JDBC: en primer lugar, ha de establecerse una conexión con una base de datos que puede estar o no situada en la misma máquina. Posteriormente, se puede pedir algún tipo de información presente en esa base de datos. Una vez realizado esto, hay que recoger la información que devuelve la base de datos, y por último, cuando se haya recibido la respuesta, se ha de cerrar la conexión con la base de datos hasta la próxima petición de información. Veamos con más detalle estos pasos:

1. Establecimiento de una conexión con la base de datos: en primer lugar hay que cargar el controlador JDBC que se vaya a usar, la forma de cargarlo es siempre con la siguiente sentencia de Java:

   Class.forName("controlador");

   El método *forName( )* pertenece a la clase *Class* del paquete *java.lang,* y devuelve el objeto *Class* asociado con la clase que tiene el nombre *controlador*. Con el nombre y la ruta completa hasta la clase, el método *forName( )* trata de encontrarla y cargarla. Si no lo consigue (porque o bien no existe la clase o no se encuentra en el directorio especificado), lanza una excepción del tipo *ClassNotFoundException*. Una vez cargado (o registrado) el controlador, es necesario abrir una conexión con la base de datos, es decir, preparar la base de datos para recibir y suministrar información. También hay que indicarle a los comandos Java a que máquina deben dirigir las peticiones de información. La forma de hacer esto en JDBC es crear un objeto de la interfaz *Connection*, que pertenece al paquete *java.sql* y que representará la conexión con la base de datos. Para ello se utiliza el método *getConnection( )* pertenecientes a la clase D*riverManager*, que tiene como

argumento un URL JDBC que representa la localización de la base de datos con la que se va ha trabajar. Este método tratará de conectarse con esa base de datos empleando el controlador JDBC que se ha cargado en el paso anterior. Si por algún motivo, no es posible establecer la conexión con la base de datos, el método lanza una excepción del tipo SQLException.

2. Petición de información a la base de datos: una vez que está abierta la conexión con la base de datos, el modo que tiene JDBC para hacer una solicitud de información es con la creación de un objeto de la clase *Statement* que contiene la operación específica a realizar sobre la base de datos. Esa operación está expresada en forma de sentencia SQL. Para crear una operación concreta sobre una conexión con una base de datos determinada, se utiliza el método *createStatement( )*, de la clase C*onnection*. Este método crea un objeto *Statement* asociado a la conexión con la base de datos especificada a través del cual se dan ordenes a esa base de datos. Con el objeto que devuelve el método *createStatement* ya se puede realizar cualquier operación (como la consulta o la modificación de información) sobre la base de datos a la que se está conectado. Una de las maneras más sencillas de realizar operaciones es utilizar el método *executeQuery( )* de la clase *Statement*.
3. Recepción de la información: al igual que para el envío de información se empleaban los objetos de la interfaz *Statement*, JDBC utiliza los objetos de otra clase, la interfaz *ResultSet*, para la recepción de la información que viene de vuelta. Ya el propio método *executeQuery( )* visto anteriormente, devuelve un objeto de esta clase en el que está almacenada la información solicitada a la base de datos.
4. Cierre de la conexión: por último ha de cerrarse la conexión con la base de datos una vez que la petición ha sido atendida. Aunque la máquina virtual Java forzaría automáticamente la desconexión, siempre es preferible cerrar el resultado, la orden y la conexión de forma explícita. El método que hay que utilizar, común a las interfaces *Connection*, *ResultSet* y *Statement*, es el método *close( )*.

En una sección posterior se analizará el uso de JDBC que se hace en la aplicación del proyecto, y se podrá observar como el esquema de uso que hace de JDBC se ciñe casi en su totalidad al esquema descrito anteriormente.

2.3. Java Media Framework (JMF).

JMF es el paquete de Java (no incluido en el JDK) orientado a la integración de sonido e imagen en aplicaciones Java. Mediante JMF es posible reproducir datos multimedia tanto de forma local como a través de la red desde una aplicación escrita en Java.

2.3.1. Descripción de JMF.

JMF es una herramienta independiente de la plataforma que permite la reproducción de una gran variedad de datos multimedia desde una aplicación Java, está diseñado para soportar la mayor parte de los tipos de datos multimedia existentes, tales como MPEG, AVI, WAV, MP3, MIDI, RMF, AU, etc. Usando JMF se pueden sincronizar y presentar datos multimedia procedentes de diferentes fuentes. Los reproductores multimedia existentes para ordenadores son fuertemente dependientes del código nativo para realizar tareas computacionalmente costosas como por ejemplo la descompresión. La API JMF proporciona una abstracción que esconde estos detalles de implementación para el desarrollador, trabaja con diferentes protocolos y mecanismos de entrega de datos, con multitud de tipos de datos multimedia, y proporciona un modelo de eventos para la comunicación asíncrona entre reproductores JMF y aplicaciones Java.

2.3.2. Utilización de JMF.

Dentro de los componentes que forman parte de la arquitectura JMF uno de los principales es el *DataSource* (su traducción literal sería fuente de datos, en este caso, multimedia). Un *DataSource* encapsula la localización de los datos multimedia así como el protocolo y el software usados para representar dichos datos. Cada reproductor multimedia de Java contiene un *DataSource*. La fuente de datos de un reproductor es identificada o bien por un *MediaLocator* de JMF o bien por un URL. *MediaLocator* es una clase de JMF que describe los datos multimedia que un reproductor muestra. Un *MediaLocator* es similar a un URL y puede construirse a partir de un URL. En Java, un URL sólo puede ser construido si el manejador del protocolo correspondiente está instalado en el sistema. *MediaLocator* no tiene esa restricción.

Los reproductores Java multimedia pueden presentar datos multimedia obtenidos de una gran variedad de fuentes, tales como archivos locales o remotos, así de como *broadcasts* (emisiones de datos multimedia en vivo). JMF soporta dos tipos diferentes de fuentes de datos multimedia:

- *Pull data sources*: en este tipo de orígenes de datos, el cliente inicia la transferencia de datos y controla el flujo de datos. Protocolos establecidos para este tipo de datos incluyen por ejemplo el HTTP y el FILE.

- *Push data sources*: en estos casos el servidor inicia la transferencia de datos y controla el flujo de datos. Dentro de los *Push data sources* se incluyen por ejemplo los *broadcasts* de datos multimedia, o el video por demanda (VOD). Para emisión de datos mediante *broadcast*, un protocolo muy extendido es el RTP (Real Time Transport Protocol), desarrollado por el IETF (Internet Engineering Task Force).

El grado de control que un programa cliente puede dar al usuario depende del tipo de fuente de datos que se desea presentar. Por ejemplo, un archivo MPEG puede ser reposicionado y un programa cliente podría permitir al usuario volver a reproducir el video o buscar posiciones adelante o atrás en él. Pero en contraste con esto, la emisión mediante *broadcast* de datos multimedia está bajo control del servidor y esos datos no pueden ser reposicionados de ningún modo.

Otro componente fundamental dentro de la arquitectura de JMF es el *Player*. Un *Player* JMF es un objeto que procesa una corriente de datos a lo largo de un periodo de tiempo, lee estos datos de un *DataSource* y los procesa y muestra en el momento adecuado. Un *Player* JMF implementa la interfaz *Player*. Los *players* comparten un modelo común para la sincronización y el control temporal de las operaciones. Una interfaz de usuario de un *Player* puede incluir un componente visual (para la reproducción de videos) y un componente de control (para controlar la reproducción del archivo multimedia, es decir, poder iniciar la reproducción, detenerla, controlar el volumen del sonido de dicha reproducción, etc.). Un *Player* ha de realizar una serie de operaciones antes de poder presentar datos multimedia. Puesto que dichas operaciones pueden tardar un tiempo en realizarse, JMF permite al desarrollador controlar cuando ocurren definiendo los estados operacionales de un *Player* y proporcionando un mecanismo de control para mover al *Player* entre esos estados.

El mecanismo de notificación de eventos de JMF permite a los programas desarrollados con JMF, responder a condiciones de error relativas a los datos multimedia, tales como agujeros de datos en la lectura de una corriente de datos multimedia, o falta de los recursos necesarios para llevar a cabo la reproducción de algún tipo de datos multimedia. El sistema de eventos proporciona también un protocolo de notificación de eventos; cuando un programa invoca un método asíncrono en un *Player*, sólo puede estar seguro de que la operación se ha completado en su totalidad, recibiendo el evento apropiado.

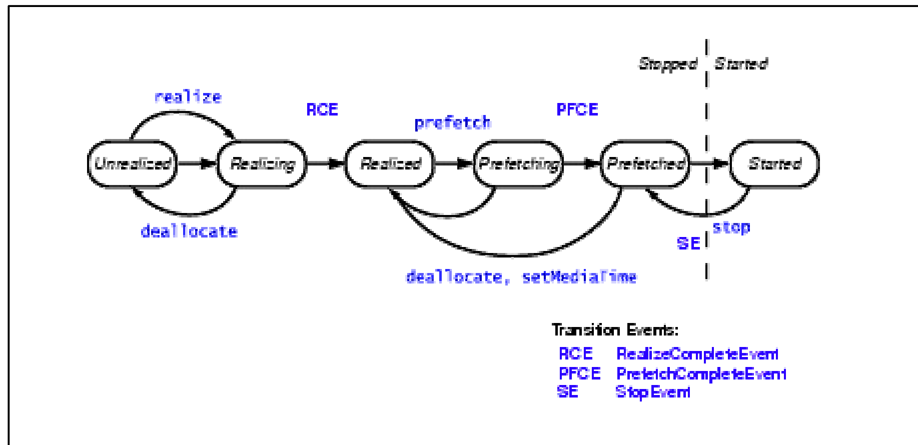
Un *Player* JMF puede estar, durante su existencia en uno de seis posibles estados. La interfaz *Clock* de JMF define los dos estados primarios, éstos son: *Stopped* y *Started*. Para facilitar el manejo de los recursos, el estado *Stopped* se divide en cinco estados de espera: *Unrealized, Realizing, Realized, Prefetching,* y *Prefetched*. Durante un funcionamiento normal, un *Player* atraviesa cada estado hasta que finalmente

alcanza el estado *Started*. A continuación vemos que ocurre con un *Player* en cada uno de los estados:

- Un *Player* en el estado *Unrealized* ha sido instanciado, pero todavía no conoce nada acerca de los datos multimedia que ha de presentar posteriormente. Así pues, cuando un *Player* está recién creado, se encuentra en el estado *Unrealized*.
- Cuando en el programa se invoca el método *realize( )*, el *Player* pasa del estado *Unrealized* al estado *Realizing*. Un *Player* en estado *Realizing* se encuentra en el proceso de determinar sus requerimientos de recursos. Mientras se encuentra en este estado, el *Player* adquiere los recursos que necesita para llevar a cabo el procesamiento posterior de los datos multimedia. Los recursos que adquiere son recursos no exclusivos, es decir, no son recursos tales como dispositivos que sólo pueden ser usados por un *Player* al mismo tiempo (tales recursos son adquiridos en un estado posterior). Un *Player* en estado *Realizing* normalmente descarga datos a través de la red (si se trata de un *Player* que ha de reproducir archivos remotos).
- Cuando un *Player* finaliza el estado *Realizing*, pasa al estado *Realized*. Un *Player* en este estado conoce ya que recursos necesita y la información acerca del tipo de datos multimedia que ha de presentar. Puesto que un *Player* en estado *Realized* conoce como manejar sus datos, puede ya proporcionar los componentes de la interfaz gráfica, es decir, los componentes visuales y los de control. Sus conexiones con otros objetos del sistema están establecidas, pero todavía no posee ningún recurso de uso exclusivo, es decir, no posee ningún recurso que pueda evitar que otro *Player* comience a presentar datos.
- Cuando el método JMF *prefetch( )* es invocado, el *Player* pasa del estado *Realized* al estado *Prefetching*. Un *Player* en este estado se encuentra preparándose para presentar sus datos multimedia. Durante esta fase, el *Player* carga sus datos multimedia, obtiene recursos exclusivos para su uso, y está totalmente preparado para comenzar a reproducir sus datos. *Prefetching* es un estado por el que un *Player* puede tener que pasar varias veces durante toda su operación, si por ejemplo, la reproducción de datos es reposicionada, o si hay un cambio en la valocidad de reproducción del *Player*, de este modo el *Player* necesita adquirir nuevos buffers para que un nuevo procesamiento de datos tenga lugar.
- Cuando un *Player* finaliza el estado *Prefetching* pasa al estado *Prefetched*. Un *Player* en este estado ya está preparado para que su reproducción sea iniciada.
- Una invocación al método *start( )* hace que el *Player* pase al estado *Started*. Un *Player* iniciado tiene su reloj ya en marcha y reproduce ya sus datos

multimedia, a no ser que haya sido programado para no mostrar los datos hasta que pasa un tiempo determinado.

Un *Player* manda *TransitionEvents* a medida que se mueve entre un estado y otro. La interfaz *ControllerListener* proporciona así a los programas un modo de determinar en que estado se encuentra un *Player*, y responder gracias a ello apropiadamente a cualquier circunstancia que pueda producirse. A continuación se muestra un esquema de todos los estados por los que puede atravesar un *Player* durante su existencia, así como los métodos que provocan estos cambios de estado, y los eventos significativos relativos a las transiciones existentes.



Estados que atraviesa un *Player*

Veamos ahora como crear un *Player* JMF, que es el objeto encargado de presentar los datos multimedia. Un objeto de este tipo se crea de forma indirecta a través del objeto *Manager*. El modo de hacerlo es invocando el método *createPlayer* de *Manager*. Éste utiliza el URL de los datos multimedia, o un *MediaLocator* (que se puede crear a partir de una cadena, similar a un URL, que indica la localización de los datos).

Como hemos comentado antes, un *Player* puede tener dos tipos de componentes gráficos, su componente visual y sus componentes de control. El componente en el que un *Player* muestra sus datos multimedia es su componente visual. Incluso en el caso de un *Player* que reproduce datos de sonido puede mostrarse un componente visual, como por ejemplo una ventana que muestre las ondas de la señal que se está reproduciendo, o una animación relativo al sonido reproducido. Para mostrar el componente visual de un *Player*, en primer lugar es necesario obtener el componente invocando el método *getVisualComponent( )*, para posteriormente añadirlo al área de presentación de la ventana o el applet en cuestión. Es posible acceder a las propiedades de presentación de un *Player*, como sus coordenadas en la pantalla, a través de su componente visual.

Un *player* a menudo tiene un panel de control que permite al usuario controlar la reproducción de los datos multimedia. Por ejemplo, un *Player* puede estar asociado con

un conjunto de botones para iniciar, detener, y pausar la corriente de datos multimedia, y con una barra con un objeto deslizable para ajustar el volumen de la reproducción. Cada *Player* JMF proporciona un panel de control por defecto. Para mostrar el panel de control por defecto de un *Player,* es necesario en primer lugar obtener dicho panel de control invocando el método *getControlPanelComponent*, y posteriormente añadir el objeto obtenido al espacio de presentación adecuado.

Hasta aquí hemos visto parte del paquete multimedia de Java, JMF. Esta arquitectura es mucho más compleja y completa que lo que aquí se ha mostrado de ella, sin embargo, lo explicado en este punto es la parte de JMF que realmente se ha utilizado en la elaboración de este proyecto. La última versión de JMF (JMF 2.0) proporciona una serie de capacidades adicionales a las ya comentadas de reproducción de datos multimedia, como por ejemplo la posibilidad de realizar capturas de imagen y sonido La utilización concreta que se ha hecho de este paquete en la aplicación construida se analizará con detalle más adelante.

2.4. Abstract Window Toolkit (AWT).

El desarrollo de applets de Java y de aplicaciones gráficas requiere trabajar con AWT, este paquete (que se incluye en el JDK) proporciona componentes básicos de interfaz, como botones, listas, menús, etiquetas, campos de texto editables, ventanas, etc.

2.4.1. Descripción de AWT.

Los componentes AWT son independientes de la plataforma, y son usados primordialmente para construir interfaces gráficas de usuario. Además, AWT proporciona también un mecanismo de manejo de eventos, soporte para transferencia de datos y manipulación de imágenes. AWT incluye también paquetes para manipulación avanzada de fuentes, impresión, accesibilidad geométrica y métodos de entrada.

AWT puede ser dividido en seis áreas principales, según el tipo de elementos AWT de que se trate:

- Componentes: son los elementos básicos de toda interfaz gráfica de usuario. Son como los bloques de construcción de las aplicaciones basadas en una interfaz gráfica de usuario. Estos elementos son implementados via una clase abstracta de componente y sus subclases asociadas que implementan componentes específicos de interfaz gráfica de usuario.
- Eventos: cada acción de usuario es convertida en una estructura de datos de tipo evento que almacena el tipo de acción y donde ocurrió dicha acción, además de otros datos necesarios. El evento es entonces enviado a la máquina virtual Java, a través del sistema operativo. Si un escuchador de eventos ha sido registrado para ese tipo de evento, entonces el escuchador es invocado para que pueda realizar alguna acción que tenga programada en tal caso.
- Escuchadores de eventos: en el mecanismo de eventos de AWT, las clases se registran para actuar ante eventos que pueden recibir, y definen escuchadores de eventos que recibe esos eventos. De este modo, se separa el manejo de eventos, de los elementos de interfaces gráficas de usuario que los generan.
- Contenedores: son componentes que almacenan otros componentes. El contenedor más común es la ventana. El panel es otro contenedor AWT muy empleado, que sirve para agrupar otros componentes dentro de una ventana.
- *Layout* y *layout* managers: se trata de una metodología para situar componentes dentro de un contenedor. Un *layout* determina donde debe ser dibujado un componente. Los *layout* managers implementan políticas específicas para situar los componentes en un contenedor.
- Pintar y modificar: aunque los componentes AWT prefabricados sean útiles, a veces es necesario pintar elementos creados a medida en las aplicaciones.

AWT proporciona mecanismos como los métodos *paint( )*, *repaint( )*, y *update( )* que permiten esta personalización de las aplicaciones.

Existen multitud de objetos pertenecientes al paquete AWT, sin embargo, prácticamente la mitad de esos objetos, de esas clases, son extensiones de la clase *java.awt.Component*. Por tanto, se puede decir que la clase *Component* y todas sus extensiones constituyen la fundación sobre la cual AWT se construye. Esta conclusión podrá corroborarse más adelante, cuando se analice la estructura del proyecto en estudio, dentro de este documento.

3. Descripción de la aplicación.

Anteriormente ya se ha comentado en que consiste básicamente la aplicación diseñada para este proyecto. Se trata de una aplicación dividida en dos partes bien diferenciadas: el servidor de archivos multimedia y el cliente reproductor de archivos multimedia.

3.1. El servidor.

El servidor es un proceso que ha de estar continuamente atendiendo las peticiones de servicio que le llegan de los clientes. No tiene una interfaz gráfica de usuario y atiende las peticiones de información sobre archivos multimedia existentes, apoyándose en una base de datos y contestando vía RMI. En cuanto a las peticiones de servicio de archivos demandados por los clientes, el servidor dispone de dos servicios que están continuamente escuchando, cada uno en un puerto diferente a la espera de la llegada de peticiones de servicio de los clientes. Se han creado dos servicios debido a que uno envía los archivos hacia el cliente mediante el protocolo de transporte TCP, mientras que el otro envía dichos archivos hacia el cliente utilizando el protocolo UDP, según sea el tipo de la petición que procede del cliente. Eso significa que se pueden recibir dos tipos de peticiones del cliente, peticiones UDP que accederán al servicio UDP del servidor, y peticiones TCP que accederán al servicio TCP del servidor.

Cuando se ejecuta el servidor, en primer lugar arranca los dos servicios para el envío de archivos multimedia (TCP y UDP). Estos dos servicios pueden servir múltiples peticiones de forma simultánea, puesto que por cada petición que llega a uno de los servicios, se crea un proceso que es realmente el que atiende la petición. Así pues, la estructura de cada uno de estos dos procesos consiste en un proceso que recibe las peticiones de servicio de los clientes, y por cada petición crea un proceso que es el encargado de enviar el archivo al cliente, empleando el protocolo adecuado.

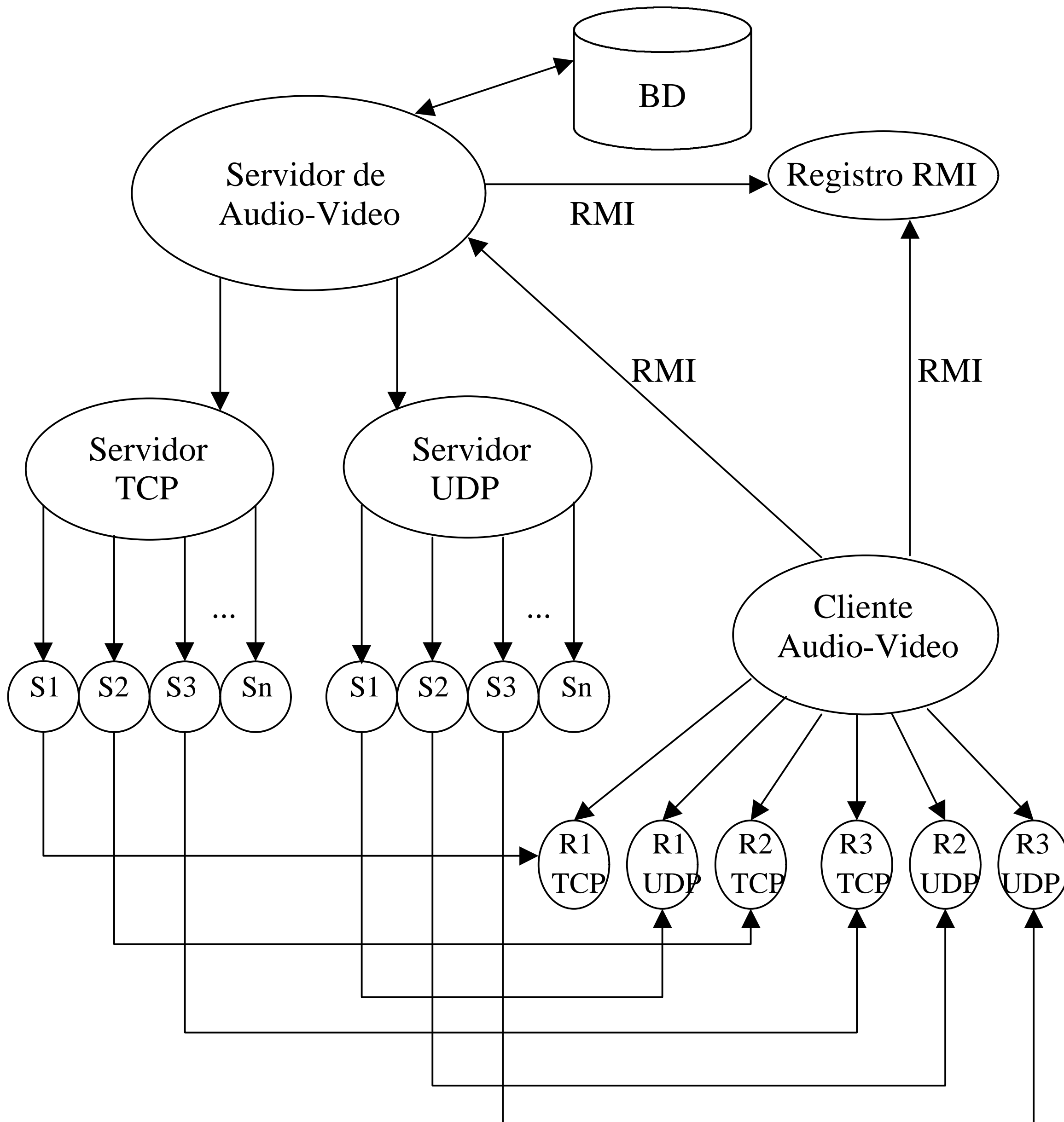
Posteriormente, el servidor registra los métodos que ofrece a los clientes para realizar consultas sobre la base de datos. Esto se hace creando una instancia del servidor que es registrada en el registro de objetos remotos (*rmiregistry*). De este modo, los clientes pueden ejecutar los métodos que el servidor ofrece, y acceder de este modo a la información de la base de datos sobre los archivos disponibles.

3.2. El cliente.

El cliente es un programa con una interfaz gráfica, que permite al usuario realizar varios tipos de búsquedas en la base de datos de archivos del servidor, de este modo el cliente conoce que archivos puede seleccionar para su reproducción. Mediante un menú es posible seleccionar el tipo de protocolo a utilizar en la descarga y procesamiento del archivo, es posible elegir entre dos variantes de descarga TCP y una

de descarga UDP. Cuando se realiza una búsqueda en el servidor, se muestra al usuario todos los archivos resultantes de dicha búsqueda. De entre esos archivos, el usuario puede seleccionar varios para su procesamiento y reproducción simultánea.

A continuación se muestra un esquema global de la aplicación anteriormente descrita:



Como puede observarse en el esquema, un programa cliente puede tener simultáneamente varios reproductores presentando datos multimedia al mismo tiempo, cada uno de los cuales recibe los datos multimedia de un archivo a través de la red, al estar conectado cada uno a un proceso servidor distinto. Se comprueba también la existencia de dos procesos servidores de archivos, creados por el proceso servidor

principal, cada uno de los cuales sirve peticiones empleando un protocolo de transporte distinto. Uno de ellos envía los archivos a un programa cliente mediante el protocolo TCP, mientras que el otro lo hace mediante UDP. Se observa también, que el servidor principal es el que se comunica con la base de datos de archivos, y al estar registrado en el registro de objetos remotos, permite que el cliente acceda a esa información vía RMI, empleando los métodos que ofrece el servidor en su interfaz remota.

4. Análisis de la aplicación.

A lo largo de este apartado analizaremos de forma detallada la implementación realizada para la elaboración de la aplicación del proyecto. Veremos los componentes Java utilizados, las clases implementadas más significativas, los métodos y variables más importantes, con el fin de que el lector de este documento tenga una idea bastante precisa de cómo funciona cada elemento que forma parte de la aplicación, y cómo se ha conseguido ese funcionamiento.

4.1. Análisis de la implementación del servidor.

La parte de la aplicación correspondiente al servidor consta de cinco archivos Java cada uno de los cuales analizaremos con detalle. Pero otro elemento imprescindible para el funcionamiento de esta aplicación es la base de datos en la que poder almacenar toda la información referente a los archivos que el servidor ofrece a los clientes. Detallamos a continuación la información significativa relativa a la base de datos utilizada:

- **Base de datos de archivos multimedia.**

La base de datos utilizada por el servidor es una base de datos Access muy simple, formada por una única tabla cuya estructura mostramos a continuación:

BdatosAV	
Nombre:	texto
Tamanyo:	numérico
Tipo:	texto
Fecha:	fecha/hora
Hora:	fecha/hora
Archivo:	texto

Como puede observarse, en la base de datos del servidor, se guarda únicamente la información de los archivos existentes. Ésta es la información que recibirá el cliente en sus consultas al servidor, sobre los archivos existentes. Puesto que a la hora de realizar accesos a la base de datos desde el código Java se utiliza el puente JDBC-ODBC (como ya se comentó en un punto anterior), es necesario, tras crear o introducir la base de datos en un sistema en el que vaya a ejecutarse el servidor, crear un ODBC para poder acceder a esa base de datos.

Lógicamente el ODBC ha de ser creado y configurado para un origen de datos de tipo Access.

Una vez vista la base de datos utilizada por el servidor analizamos con detalle los archivos implementados para su construcción:

- **ServidorAV.java.**

En este fichero se implementa la estructura básica del servidor. Es en este fichero, donde se encuentra el método *main( )*, y por tanto, la ejecución del servidor comienza por la clase implementada en este fichero. Veamos su código:

```
/*
 * @(#) ServidorAV.java
 *
 * Versión 1    22/04/99
 * Copyright 1999    Jacinto Navarro González
 *
 */

// Paquete al que pertenece la clase
package Servidor;

// Clases y paquetes importados
import java.rmi.*;
import java.rmi.server.*;
import java.sql.*;
import java.net.UnknownHostException;
import java.net.*;
import java.io.*;
import java.util.*;

/**
 * Esta clase implementa una interfaz remota y se registra como objeto
 * remoto en el registro RMI.
 * Crea una instancia de un servidor de archivos TCP y otra de uno UDP
 * para enviar datos multimedia a hosts remotos.
 * Implementa tres métodos de acceso a una base de datos de archivos
 * multimedia, para ser ofrecidos a hosts remotos vía RMI.
 *
 */

public class ServidorAV extends UnicastRemoteObject implements ServidorAVint,
        Runnable {
    private Thread servicioUDP = null; // Thread que crea el servidor de
                                   // archivos UDP
    private Thread servicioTCP = null; // Thread que crea el servidor de
                                   // archivos TCP
    private ServidorArchivosTCP servidorArchivosTCP; // Servidor de archivos
                                                     // TCP
    private ServidorArchivosUDP servidorArchivosUDP; // Servidor de archivos
                                                     // UDP

    public ServidorAV() throws RemoteException {
        super();

        // Llamadas que crean los threads servidores
        iniciaServicioUDP();
        iniciaServicioTCP();
    }
```

Como puede observarse, en el constructor de la clase *ServidorAV* se invocan los métodos *iniciaServicioUDP* e *iniciaServicioTCP* que, como veremos más adelante, lanzan los *threads* que se encargan de instanciar los objetos *ServidorArchivosTCP* y *ServidorArchivosUDP*. El motivo de encargar la creación de estos dos objetos servidores a un par de *threads*, es para evitar el bloqueo que se produciría en el caso de instanciar estos objetos durante el flujo normal de ejecución del programa. Es decir, si en el flujo de ejecución del programa instanciásemos primero, por ejemplo, el objeto *ServidorArchivosTCP*, éste tomaría el flujo de ejecución del programa y lo dejaría bloqueado, puesto que, como se verá más adelante, su código principal es un bucle que se ejecuta de forma infinita, esperando y atendiendo peticiones de los clientes mediante la creación de *threads* servidores. Así pues, de ese modo, la instanciación posterior de *ServidorArchivosUDP* nunca se llevaría a cabo.

```
/**
 * Construye la sentencia SQL que busca todos los datos de
 * todos los archivos almacenados en el servidor.
 */
public String construyeOrdenTodos() {
    StringBuffer ordenSQL =
            new StringBuffer("SELECT * FROM " + Util.devuelveBD());
          ordenSQL.append(" ORDER BY nombre ASC");
    return ordenSQL.toString();
}
```

El método *construyeOrdenTodos( )* construye la orden SQL que permite extraer de la base de datos todos los datos de todos los archivos existentes en el servidor. Dicha orden SQL se almacena, como puede observarse, en una cadena de caracteres. La cadena será recogida por el método de ejecución de la sentencia SQL.

```
/**
 * Construye la sentencia SQL que busca todos los datos de los
 * archivos que corresponden a los parámetros de busqueda recibidos.
 */
public String construyeOrdenSeleccion(String fechaIni, String fechaFin,
        int tamMin, int tamMax, String tipo) {
    StringBuffer ordenSQL =
           new StringBuffer("SELECT * FROM " + Util.devuelveBD() + " WHERE
");
    ordenSQL.append("fecha > {d '");
    ordenSQL.append(fechaIni);
    ordenSQL.append("'} AND fecha < {d '");
    ordenSQL.append(fechaFin);
    ordenSQL.append("'} AND tamanyo > ");
    ordenSQL.append(tamMin);
    ordenSQL.append(" AND tamanyo < ");
    ordenSQL.append(tamMax);
    if (tipo.equals("Todos")) {
        ordenSQL.append(" AND ( archivo = '");
        ordenSQL.append("Audio");
        ordenSQL.append("' OR archivo = '");
        ordenSQL.append("Video");
        ordenSQL.append("' ) ORDER BY nombre ASC");
    } else {
        ordenSQL.append(" AND archivo = '");
        ordenSQL.append(tipo);
        ordenSQL.append("' ORDER BY nombre ASC");
    }
    return ordenSQL.toString();
}
```

El método *construyeOrdenSeleccion(...)* construye una sentencia SQL elaborada para la selección de los datos de sólo algunos archivos de la base de datos. Los parámetros que recibe determinan el contenido de la sentencia SQL. Así pues, los parámetros *fechaIni* y *fechaFin* acotan el rango de fechas de los archivos que van a ser devueltos al ejecutar la sentencia, mientras que los parámetros *tamMin* y *tamMax* determinan el tamaño mínimo y máximo de los archivos buscados. El parámetro *tipo* indica si los archivos que se buscan son de sonido, de imagen o de ambos tipos.

```
/**
 * Construye la sentencia SQL que busca todos los datos de
 * archivos que responden al patrón de búsqueda recibido.
 */
public String construyeOrdenArchivos(String archivos) {
    StringBuffer texto = new StringBuffer();
    String nombres;
    char letra;
    int longitud;
    int contador = 0;
    int parcial = 0;
    boolean asterisco = false;
    boolean primera = true;
    StringBuffer ordenSQL =
            new StringBuffer("SELECT * FROM " + Util.devuelveBD() + "
                             WHERE ");

    nombres = archivos.trim();
    longitud = nombres.length();

    // Recorre todo el patrón de búsqueda
    while (contador < longitud) {
        parcial ++;
        letra = nombres.charAt(contador);
        // Ignora los separadores y los caracteres comodín
        while ((letra != ' ') && (letra != ';') && (letra != ',')) {
            if (letra != '*') {
                texto.append(nombres.charAt(contador));
            } else {
                texto.append('%');
                asterisco = true;
            }
            contador ++;
            parcial ++;
            if (contador != longitud) {
                letra = nombres.charAt(contador);
            } else {
                break;
            }
        }
        if (parcial > 1) {
            if (primera) {
                if (asterisco) {
                    // Si había un carácter comodín
                    ordenSQL.append("nombre LIKE '");
                    ordenSQL.append(texto);
                    texto = new StringBuffer();
                    ordenSQL.append("' ");
                    asterisco = false;
                } else {
                    // Si era un nombre de fichero
                    ordenSQL.append("nombre ='");
                    ordenSQL.append(texto);
                    texto = new StringBuffer();
                    ordenSQL.append("' ");
                }
```

```
                primera = false;
            } else {
                if (asterisco) {
                    // Si había un carácter comodín
                    ordenSQL.append("OR nombre LIKE '");
                    ordenSQL.append(texto);
                    texto = new StringBuffer();
                    ordenSQL.append("' ");
                    asterisco = false;
                } else {
                    // Si era un nombre de fichero
                    ordenSQL.append("OR nombre ='");
                    ordenSQL.append(texto);
                    texto = new StringBuffer();
                    ordenSQL.append("' ");
                }
            }
        }
        contador ++;
        parcial = 0;
    }
    ordenSQL.append(" ORDER BY nombre ASC");
    return ordenSQL.toString();
}
```

Finalmente el método *construyeOrdenArchivos(...)* construye una sentencia SQL elaborada para la selección de los datos de sólo algunos archivos de la base de datos. El método recibe un único parámetro de tipo string que guarda un patrón de búsqueda de archivos, al estilo de los permitidos en diálogos de búsqueda como por ejemplo los de Windows. Así pues, permite el uso de caracteres comodín como el asterisco, y separadores como el punto y coma o la coma. También permite la búsqueda directa por nombres completos de archivo.

```
/**
 * A partir del resultado de una consulta SQL, construye una cadena
 * de caracteres entendible por los clientes en la que va concatenando
 * los datos devueltos por la consulta, introduciendo separadores
 * entre ellos
 */
public String construirResultado(ResultSet valores) {
    StringBuffer respuesta = new StringBuffer();
    String valorSQL = new String();
    boolean resultadoOK = false;
    java.util.Date fecha;
    java.util.Date hora;
    Integer entero;
    int aux;

    try {
        // Recorre todas las filas obtenidas
        while (valores.next()) {
            // Extrae el nombre de archivo
            valorSQL = valores.getString("nombre");
            respuesta.append(valorSQL + '|');
            // Extrae la longitud del archivo
            valorSQL = valores.getString("tamanyo");
            respuesta.append(valorSQL + '|');
            // Extrae la fecha del archivo
            fecha = valores.getDate("fecha");

            // Pasa los valores a valores de dos dígitos
            entero = new Integer(fecha.getDate());
            valorSQL = new String();
            if (entero.intValue() > 9) {
```

```
                valorSQL = valorSQL + entero.toString() + '/';
          } else {
               valorSQL = valorSQL + '0'+ entero.toString() + '/';
          }
            entero = new Integer(fecha.getMonth());
            if (entero.intValue() > 9) {
                valorSQL = valorSQL + entero.toString() + '/';
          } else {
                valorSQL = valorSQL + '0' + entero.toString() + '/';
          }
            entero = new Integer(fecha.getYear());
            aux = entero.intValue() % 100;

            entero = new Integer(aux);
            if (entero.intValue() > 9) {
                valorSQL = valorSQL + entero.toString() + ' ';
            } else {
                valorSQL = valorSQL + '0' + entero.toString() + ' ';
            }

            // Extrae la hora del archivo
            hora = valores.getTime("hora");
            entero = new Integer(hora.getHours());

            // Pasa los valores a valores de dos dígitos
            if (entero.intValue() > 9) {
                valorSQL = valorSQL + entero.toString() + ':';
            } else {
                valorSQL = valorSQL + '0' + entero.toString() + ':';
            }
            entero = new Integer(hora.getMinutes());
            if (entero.intValue() > 9) {
                valorSQL = valorSQL + entero.toString();
            } else {
                valorSQL = valorSQL + '0' + entero.toString();
            }
            respuesta.append(valorSQL + '|');

            // Extrae el tipo de archivo (audio, video)
            valorSQL = valores.getString("archivo");
            respuesta.append(valorSQL);
            respuesta.append("*");
        }
        resultadoOK = true;
    } catch (SQLException e) {
        System.err.println("Error al construir resultado de consulta a la
                           base de datos");
    } finally {
        if (resultadoOK) {
            return respuesta.toString();
        } else {
            return null;
        }
    }
}
```

El método *construirResultado(...)* recibe como parámetro un valor de tipo *ResultSet*, que como se vio en el apartado de JDBC, almacena el resultado de una consulta sobre la base de datos. Dicho resultado es procesado fila a fila en el bucle de este método. En cada pasada del bucle se extraen atributo a atributo los valores que interesan. Los valores de tipo texto se extraen mediante el método *getString( )*, los de tipo fecha se extraen mediante el método *getDate( )*, etc. A los valores extraidos se les da un formato adecuado para que sean entendidos por el cliente, de ahí que entre un valor formateado y otro de una fila se introduzca un carácter separador del tipo '|', y

entre una fila formateada y otra se introduzca otro carácter separador distinto, del tipo '*'. De este modo, el cliente, cuando ejecuta vía RMI consultas sobre la base de datos, recibe los resultados ya con un mínimo formato sobre ellos, en una cadena de caracteres. Es, por tanto, el valor devuelto por este método el que llega al cliente, tras invocar éste uno de los métodos remotos ofrecidos por el servidor en su interfaz remota.

```
    /**
     * Ejecuta la consulta que busca los datos de todos los archivos
     * existentes. Es ejecutada por los clientes vía RMI, ya que
     * implementa un método de la interfaz remota.
     */
    public String obtenerBusquedaTodos() throws RemoteException {
        String respuesta = new String();

        try {
            // Carga el controlador indicado en el ini
            Class.forName(Util.devuelveControlador());

            // Crea una conexión con la fuente de datos indicada en el ini
            Connection conexion =
DriverManager.getConnection(Util.devuelveDSource());
            // Construye la sentencia de búsqueda SQL
            Statement orden = conexion.createStatement();
            String ordenSQL = construyeOrdenTodos();
            // Ejecuta la sentencia SQL
            ResultSet resultado = orden.executeQuery(ordenSQL);
            // Construye el resultado a devolver al cliente
            respuesta = construirResultado(resultado);
            // Cierra la conexión
            orden.close();
            resultado.close();
            conexion.close();

        } catch (ClassNotFoundException e) {
            System.err.println("Error al cargar controlador Jdbc-Odbc");
        } catch (SQLException e) {
            System.err.println("Error en consulta a la base de datos");
        }
        return respuesta;
    }
```

El método *obtenerBusquedaTodos( )* implementa un método de la interfaz remota del servidor (puede observarse en la cabecera del método, ya que puede lanzar la excepción *RemoteException*, en caso de un error en su ejecución). Por tanto, éste es uno de los métodos que puede ejecutar el cliente para conocer qué archivos multimedia están disponibles en el servidor. En primer lugar se carga el controlador que corresponda (el indicado en el fichero de configuración, o fichero ini), en principio el controlador JDBC-ODBC, puesto que la base de datos empleada en el proyecto, es una base de datos Access. El segundo paso es crear una conexión con la base de datos del servidor; el origen de datos también es cargado del fichero de configuración. Posteriormente se crea un objeto de tipo *Statement*, que, como vimos en el apartado dedicado a JDBC, es un objeto asociado a la conexión anteriormente creada, y mediante este objeto creado se puede realizar cualquier operación sobre la base de datos a la que se está conectado. A continuación se invoca el método *construyeOrdenTodos( )* ya descrito anteriormente. Con la cadena de caracteres que devuelve dicho método y utilizando el objeto *Statement* antes creado, se invoca el método *executeQuery(...)* que

ejecuta la sentencia SQL sobre la base de datos del servidor. El objeto *ResultSet* que devuelve el método anterior se pasa como parámetro al método *construirResultado(...)* descrito anteriormente, que transforma el resultado de la consulta a datos formateados y entendibles por parte del cliente que ha ejecutado la consulta. El último paso consiste en cerrar todos los objetos JDBC que se han creado para ejecutar la consulta, entre ellos, la conexión que se había abierto sobre la base de datos.

```
    /**
     * Ejecuta la consulta que busca los datos de los archivos
     * determinados por los parámetros recibidos. Es ejecutada
     * por los clientes vía RMI, ya que implementa un método de
     * la interfaz remota.
     */
    public String obtenerBusquedaSeleccion(String fechaIni, String fechaFin,
            int tamMin, int tamMax, String tipo) throws RemoteException {
        String respuesta = new String();

        try {
            // Carga el controlador indicado en el ini
            Class.forName(Util.devuelveControlador());
            // Crea una conexión con la fuente de datos indicada en el ini
            Connection conexion =
DriverManager.getConnection(Util.devuelveDSource());
            // Construye la sentencia de búsqueda SQL
            Statement orden = conexion.createStatement();
            String ordenSQL = construyeOrdenSeleccion(fechaIni, fechaFin,
                    tamMin, tamMax, tipo);
            // Ejecuta la sentencia SQL
            ResultSet resultado = orden.executeQuery(ordenSQL);
            // Construye el resultado a devolver al cliente
            respuesta = construirResultado(resultado);
            // Cierra la conexión
            orden.close();
            resultado.close();
            conexion.close();

        } catch (ClassNotFoundException e) {
            System.err.println("Error al cargar controlador Jdbc-Odbc");
        } catch (SQLException e) {
            System.err.println("Error en consulta a la base de datos");
        }
        return respuesta;
    }
```

El método *obtenerBusquedaSeleccion(...)* implementa otro de los métodos de la interfaz remota del servidor (lanza la excepción *RemoteException* en caso de error). Su estructura y funcionamiento es muy similar al método *obtenerBusquedaTodos( )* descrito anteriormente. La diferencia estriba en que en este caso, para construir la sentencia SQL, se invoca al método *construyeOrdenSeleccion(...)*, que elabora una sentencia SQL más compleja, gracias a los parámetros que recibe procedentes del cliente.

```
    /**
     * Ejecuta la consulta que busca los datos de los archivos que se
     * corresponden con el patrón de búsqueda recibido. Es ejecutada
     * por los clientes vía RMI, ya que implementa un método de la
     * interfaz remota.
     */
    public String obtenerBusquedaArchivos(String archivosBusqueda) throws
            RemoteException {
        String respuesta = new String();
```

```
        try {
            // Carga el controlador indicado en el ini
            Class.forName(Util.devuelveControlador());
            // Crea una conexión con la fuente de datos indicada en el ini
            Connection conexion =
DriverManager.getConnection(Util.devuelveDSource());
            // Construye la sentencia de búsqueda SQL
            Statement orden = conexion.createStatement();
            String ordenSQL = construyeOrdenArchivos(archivosBusqueda);
            // Ejecuta la sentencia SQL
            ResultSet resultado = orden.executeQuery(ordenSQL);
            // Construye el resultado a devolver al cliente
            respuesta = construirResultado(resultado);
            // Cierra la conexión
            orden.close();
            resultado.close();
            conexion.close();

        } catch (ClassNotFoundException e) {
            System.err.println("Error al cargar controlador Jdbc-Odbc");
        } catch (SQLException e) {
            System.err.println("Error en consulta a la base de datos");
        }
        return respuesta;
    }
```

El método *obtenerBusquedaArchivos(...)* implementa el tercero de los métodos definidos en la interfaz remota del servidor. Es muy similar a los dos métodos anteriores, sólo que éste recibe por parte del cliente que lo invoca, un parámetro de tipo string que representa un patrón de búsqueda de archivos, que es pasado al método *construyeOrdenArchivos(...)* para que construya la sentencia SQL correspondiente, por lo demás tiene la misma estructura y funcionamiento que los dos métodos anteriores.

```
    /**
     * Inicia el servicio que sirve archivos mediante UDP
     * lanzado el thread de servicio UDP
     */
    public void iniciaServicioUDP() {
        if (servicioUDP == null) {
            servicioUDP = new Thread(this);
            servicioUDP.start();
        }
    }

    /**
     * Inicia el servicio que sirve archivos mediante TCP
     * lanzado el thread de servicio TCP.
     */
    public void iniciaServicioTCP() {
        if (servicioTCP == null) {
            servicioTCP = new Thread(this);
            servicioTCP.start();
        }
    }
```

El método *iniciaServicioUDP( )* se encarga de lanzar el *thread* que creará el objeto *ServidorArchivosUDP*. La razón de utilizar un *thread* para realizar esta tarea se ha comentado ya anteriormente. De forma análoga, el método *iniciaServicioTCP( )* lanza el thread que crea el objeto *ServidorArchivosTCP*.

```
    /**
     * Implementa el método run de la interfaz Runnable,
     * Crea los objetos servidores según el thread que se lance.
     */
    public void run() {
        if (Thread.currentThread() == servicioUDP) {
            servidorArchivosUDP = new
                    ServidorArchivosUDP(Util.devuelvePuertoMediaUDP());
            return;
        }
        if (Thread.currentThread() == servicioTCP) {
            servidorArchivosTCP = new
                    ServidorArchivosTCP(Util.devuelvePuertoMediaTCP());
            return;
       }
       return;
    }
```

El método *run( )* es un método que ha de implementar toda clase que implemente la interfaz *Runnable*. Este método es lanzado cuando se crea algún *thread*, de ahí que mediante el método *currentThread( )*, se compruebe primero si el *thread* lanzado es uno u otro, para así crear el objeto servidor de archivos UDP o el TCP.

```
    public static void main(String args[]) {
        String localHost = new String();

        // Lee y carga los datos del fichero ini
        Util.leeFicheroIni();
        // Establece el security manager RMI por defecto
        System.setSecurityManager(new RMISecurityManager());

        try {
            InetAddress localURL = InetAddress.getLocalHost();
            localHost = localURL.getHostName();
        } catch (UnknownHostException e) {
            System.err.println("Dirección del host local no encontrada");
        }
        try {
            // Crea el objeto servidor y lo registra en el registro
            // de objetos remotos
            ServidorAV objservidor = new ServidorAV();
            Naming.rebind("//" + localHost + "/ServidorAVint", objservidor);
            System.err.println("Registrado el objeto servidor");
        } catch (MalformedURLException e) {
            System.err.println("El URL del servidor no es válido");
            // Aborta ejecución
            System.err.println("Programa abortado");
            System.exit(0);
        } catch (RemoteException e) {
            System.err.println("Error al registrar el objeto servidor");
            // Aborta ejecución
            System.err.println("Programa abortado");
            System.exit(0);
        }
    }
}
```

Este es el método principal del programa servidor, puesto que su flujo de ejecución comienza por este método. En primer lugar se invoca el método *leeFicheroIni( )* de la clase *Util*, también incluida en el paquete *Servidor*. Este método lee valores del fichero de configuración del servidor, 'ServidorAV.ini' (como el puerto

en que escuchan los servidores TCP y UDP, la base de datos a la que se accede, mediante que controlador se accede a ella, el nombre del origen de datos para esa base de datos, etc.) y los carga en unas variables estáticas de la clase *Util* comentada antes. Después se establece el *security manager* que ha de controlar que se cumplan las políticas de seguridad establecidas en el sistema. Finalmente se crea un objeto del tipo *ServidorAV*, es decir, de la clase en estudio, y se registra en el registro de objetos remotos (*rmiregistry*), para que los clientes puedan utilizar, accediendo previamente a dicho registro, los métodos que el objeto de clase *ServidorAV* ha declarado en su interfaz remota, y por tanto accesibles vía RMI. El método que registra el objeto es *rebind(...)* de la clase final *Naming*, y recibe como parámetros un objeto y un nombre de objeto mediante al cual poder referenciar a dicho objeto. El cliente, cuando desee utilizar los métodos remotos ofrecidos por el servidor, utilizará ese nombre de objeto que se registra para poder utilizar los métodos de ese objeto mediante el *stub* correspondiente, como se comentó en el apartado dedicado a RMI.

Una vez analizado el fichero central del programa servidor, vemos a continuación el fichero que define la interfaz remota implementada:

- **ServidorAVint.java.**

```
/*
 * @(#) ServidorAVint.java
 *
 * Versión 1    22/04/99
 * Copyright 1999    Jacinto Navarro González
 *
 */

// Paquete al que pertenece la clase
package Servidor;

// Clases y paquetes importados
import java.rmi.*;

/**
 * Esta interfaz define los métodos que ofrece el servidor a objetos
 * remotos. Son métodos para acceso a la información presente en la
 * base de datos del servidor.
 *
 */
public interface ServidorAVint extends Remote {
      String obtenerBusquedaTodos() throws RemoteException;
      String obtenerBusquedaSeleccion(String fechaIni, String
fechaFin, int tamMin, int tamMax, String tipo) throws RemoteException;
      String obtenerBusquedaArchivos(String archivosBusqueda) throws
RemoteException;
}
```

En el fichero anterior se definen los tres métodos ofrecidos por el servidor a objetos remotos. Puede observarse como la interfaz extiende una interfaz remota, y

como los tres métodos lanzan la excepción *RemoteException*, respondiendo así a lo definido en el apartado dedicado a RMI.

Una vez vistos el fichero del programa servidor y la interfaz remota definida, vemos a continuación el fichero que implementa la clase para la creación del objeto servidor de archivos TCP:

- **ServidorArchivosTCP.java.**

```
/*
 * @(#) ServidorArchivosTCP.java
 *
 * Versión 1    22/07/99
 * Copyright 1999    Jacinto Navarro González
 *
 */

// Paquete al que pertenece la clase
package Servidor;

// Clases y paquetes importados
import java.io.*;
import java.net.*;

/**
 * Esta clase implementa un servidor de archivos multimedia, que
 * utiliza el protocolo de transporte TCP para enviar los archivos
 * a través de la red.
 * A partir de un socket servidor se crean procesos que sirven las
 * peticiones que llegan.
 *
 */
public class ServidorArchivosTCP {
    private ServerSocket socketServidor; // Socket que recibe peticiones

    public ServidorArchivosTCP(int puerto) {
        Socket socketCliente;

        try {
            socketServidor = new ServerSocket(puerto);
        } catch (IOException e) {
            System.err.println("Error al crear el socket servidor de archivos
                               TCP");
        }

        while (true) {
            try {
                // El socket servidor acepta peticiones y crea un
                // socket para cada petición que llega
                socketCliente = socketServidor.accept();
                // Crea un thread que sirve la petición que llega
                new Thread(new HiloServidor(socketCliente)).start();
            } catch (IOException e) {
                System.err.println("Error al crear el socket que sirve la
                                   petición TCP");
            }
        }
    }
```

En el constructor de la clase *ServidorArchivosTCP*, se crea en primer lugar un objeto de la clase *ServerSocket*, dicho objeto es un *socket* que escucha en el puerto que

se pasa como parámetro al constructor. En el bucle infinito se pone el *socket* creado a escuchar peticiones en el puerto indicado, mediante la invocación del método *accept( )*. Dicho método crea un *socket* por cada petición de servicio que llega procedente de los clientes. Ese *socket*, desde ese momento se encuentra conectado, asociado, al *socket* que ha establecido la conexión en el cliente. Una vez se obtiene el *socket* conectado con el cliente, se crea un *thread* que servirá esa petición de servicio, y de este modo el bucle puede seguir su ejecución y por tanto, seguir atendiendo peticiones de servicio.

```
class HiloServidor implements Runnable {
    Socket socket;

    public HiloServidor(Socket socketCliente) {
        this.socket = socketCliente;
    }

    /**
     * Implementa el método run de la interfaz Runnable.
     * Envía el archivo solicitado al cliente.
     */
    public synchronized void run() {
        int longitud;
        int cantidad;
        InputStream entrada = null;
        OutputStream salida = null;
        FileInputStream entradaFichero = null;
        BufferedOutputStream bos;
        byte[] nombreArchivo = new byte[64];
        byte[] lectura = new byte[Util.devuelveBloqueEnvio()];
        String archivo = new String();
        File descriptor;


        try {
            entrada = socket.getInputStream();
        } catch (IOException e) {
            System.err.println("Error al crear entrada de datos del socket que
                                   sirve la petición TCP");
        }

       try {
           salida = socket.getOutputStream();
       } catch (IOException e) {
           System.err.println("Error al crear salida de datos del socket que
                                   sirve la petición TCP");
       }

        try {
            // Lee el nombre del archivo solicitado del stream de entrada
            entrada.read(nombreArchivo);
        } catch (IOException e) {
            System.err.println("Error al leer información entrante del archivo
                                   a servir en petición TCP");
        }

        archivo = new String(nombreArchivo);
        descriptor = new File(Util.devuelveDirArchivos() + '/' + archivo);
        longitud = (int) descriptor.length();

        try {
            entradaFichero = new FileInputStream(descriptor);
        } catch (FileNotFoundException e) {
            System.err.println("No encontrado el archivo solicitado '" +
                                   archivo + "' en petición TCP");
        }
```

```
        bos = new BufferedOutputStream(salida, longitud);

        try {
            // Envía el contenido del fichero por el stream de salida
            while (entradaFichero.available() > 0) {
                cantidad= entradaFichero.read(lectura);
                bos.write(lectura, 0, cantidad);
            }
            // Cierra todos los streams abiertos
            entradaFichero.close();
            bos.close();
            entrada.close();
            salida.close();

        } catch (IOException e) {
            System.err.println("Error al sevir el archivo '" + archivo + "' en
                               petición TCP");
        }
    }
}
}
```

La clase *HiloServidor* implementa la interfaz *Runnable* y en su constructor recibe como parámetro el *socket* conectado al cliente. A partir de este *socket* se crean dos *streams* (corrientes de bytes) para comunicarse con el *socket* que está conectado al otro extremo de la comunicación, en el cliente. El *stream* de entrada se crea invocando el método *getInputStream( )*, y el *stream* de salida invocando el método *getOutputStream( )*, ambos métodos de la clase *Socket*. Mediante el *stream* de entrada se lee el nombre del archivo solicitado por el cliente. Una vez se tiene el nombre del archivo a enviar, se crea un *stream* para la lectura de los datos del archivo, creando un objeto de la clase *FileInputStream*, al que se le pasa el descriptor del archivo en cuestión. A continuación se crea un objeto de la clase *BufferedOutputStream*, a partir del *stream* de salida creado anteriormente. Este objeto creado permite la escritura de caracteres o bytes empleando buffer y separadores de línea dependientes de la plataforma utilizada. Mediante este objeto creado se escribirán los datos del archivo hacia el cliente. Como puede observarse en el código, la escritura del archivo por el *stream* se realiza en un bucle que va enviando los datos de éste en bloques de un tamaño determinado (este tamaño lo determina un parámetro del archivo de configuración 'ServidorAV.ini'). El bucle finaliza su ejecución cuando ya no quedan datos por enviar del archivo. La última tarea que realiza el *thread* es cerrar todos los *streams* que se han utilizado al servir la petición de envío del archivo.

Analizamos a continuación el código del fichero en el que se implementa la clase para la creación del servidor de archivos UDP:

- **ServidorArchivosUDP.**

```
/*
 * @(#) ServidorArchivosUDP.java
 *
 * Versión 1    12/06/00
 * Copyright 2000    Jacinto Navarro González
```

```
 *
 */

// Paquete al que pertenece la clase
package Servidor;

// Clases y paquetes importados
import java.io.*;
import java.net.*;

/**
 * Esta clase implementa un servidor de archivos multimedia, que
 * utiliza el protocolo de transporte UDP para enviar los archivos
 * a través de la red.
 * A partir de un socket escuchando la llegada de paquetes de conexión
 * se crean procesos que sirven las peticiones que llegan.
 *
 */
public class ServidorArchivosUDP {
    private DatagramSocket socketServidor; // Socket que recibe peticiones
    public ServidorArchivosUDP(int puerto) {
        DatagramPacket paqueteRecepcion;
        byte [] nombreArchivo = new byte[64];

        try {
            socketServidor = new DatagramSocket(puerto);
        } catch (IOException e) {
            System.err.println("Error al crear el socket servidor de archivos
                               UDP");
        }

        while (true) {
            try {
                // El socket servidor escucha continuamente paquetes
                // que llegan y crea un thread para cada petición que llega
                // al que le pasa el paquete recibido
                paqueteRecepcion = new DatagramPacket(nombreArchivo, 64);
                socketServidor.receive(paqueteRecepcion);
                // Crea un thread que procesa el paquete recibido
                new Thread(new HiloServidor(paqueteRecepcion)).start();

            } catch (IOException e) {
                System.err.println("Error al crear el proceso que sirve la
                                   petición UDP");
            }
        }
    }
```

En el constructor de la clase *ServidorArchivosUDP*, se crea en primer lugar un objeto de la clase *DatagramSocket*, dicho objeto es un *socket* que utiliza el puerto que se pasa como parámetro para enviar y recibir datagramas. En este caso el *socket* sólo se crea para recibir datagramas. En el bucle infinito se pone el *socket* creado a recibir datagramas por el puerto indicado, mediante la invocación del método *receive( )*. Dicho método bloquea el programa hasta que llega un datagrama procedente de un cliente, con el nombre del archivo que el cliente desea reproducir. Ese datagrama es pasado como un parámetro, en la instanciación del thread que servirá la petición. Así, el bucle puede seguir su ejecución y por tanto, seguir atendiendo peticiones de servicio. Como puede observarse, la estructura de esta clase es muy similar a la del servidor de archivos TCP. Se trata de una estructura típica de proceso servidor concurrente; un proceso principal cuya tarea es ejecutar un bucle de forma infinita, en el que se reciben peticiones de servicio, que son atendidas por subprocesos creados por el proceso principal.

```
class HiloServidor implements Runnable {
    DatagramPacket paquete;

    public HiloServidor(DatagramPacket paqueteRecibido) {
        this.paquete = paqueteRecibido;
    }

    /**
     * Implementa el método run de la interfaz Runnable.
     * Envía el archivo solicitado al cliente.
     */
    public synchronized void run() {
        FileInputStream entradaFichero = null;
        byte[] lectura = new byte[Util.devuelveBloqueEnvio()];
        DatagramSocket socketEnvio = null;
        int longitud;
        int cantidad = 0;
        long disponible;
        File descriptor;
        InetAddress direccionCliente;
        int puertoCliente;
        String archivo;

        // Extrae el puerto de procedencia del paquete
        puertoCliente = paquete.getPort();
        // Extrae la dirección de procedencia del paquete
        direccionCliente = paquete.getAddress();
        // Obtiene el nombre del archivo solicitado
        archivo = new String(paquete.getData(), 0, paquete.getLength());

        descriptor = new File(Util.devuelveDirArchivos() + '/' + archivo);
        longitud = (int) descriptor.length();

        try {
            // Crea un stream para lectura del fichero
            entradaFichero = new FileInputStream(descriptor);
        } catch (FileNotFoundException e) {
            System.err.println("No encontrado el archivo solicitado '" +
                                   archivo + "' en petición UDP");
        }

        try {
            // Crea un socket UDP para envío del fichero
            // El puerto asignado es el primero que encuentra libre
            socketEnvio = new DatagramSocket();
        } catch (SocketException e) {
            System.err.println("Error al crear el socket que sirve la petición
                                   UDP");
        }

        try {
            disponible = entradaFichero.available();
            while (disponible > 0) {
                if (disponible < Util.devuelveBloqueEnvio()) {
                    lectura = new byte[(int) disponible];
                }
                cantidad= entradaFichero.read(lectura);
                // Crea el paquete de envío
                paquete = new DatagramPacket(lectura, cantidad,
                        direccionCliente, puertoCliente);
                // Envía el paquete
                socketEnvio.send(paquete);
                try {
                    // El proceso espera un tiempo hasta el siguiente
                    // envío para evitar bloqueos en los envíos
                    Thread.currentThread().sleep(Util.devuelveRetardoEnvio());
                } catch (InterruptedException ie) {
```

```
                    // Ignora excepción
                }
                disponible = entradaFichero.available();
        }
        // Cierra el stream abierto sobre el archivo
        entradaFichero.close();

        } catch (IOException e) {
            System.err.println("Error al sevir el archivo '" + archivo + "' en
                                petición UDP");
        }
    }
}
}
```

Cada *thread* que se crea, recibe un datagrama en el que va incrustado el nombre del archivo multimedia que se ha de servir, además de datos referentes al host cliente que realizó la petición. Lo primero que hace el proceso, es extraer la dirección y el puerto del cliente. A continuación se crea un *stream* para leer el contenido del archivo solicitado. Después se crea una instancia de *DatagramSocket* sin parámetro alguno, así escribirá los datos por el primer puerto que encuentre libre. Posteriormente, va leyendo los datos del archivo en bloques de un tamaño determinado en el fichero de configuración del servidor. Con cada bloque de datos que se lee del fichero, se crea un objeto de la clase *DatagramPacket*, en el que se introducen los datos, su longitud, y la dirección y puerto de destino (los del cliente). Esta operación se repite mientras quedan por enviar datos del archivo. Un detalle a comentar, es el retardo que se introduce intencionadamente entre el envío de un datagrama y otro. Esto es debido a que si el envío de datagramas por la red es muy consecutivo, se producen problemas, ya que éstos no llegan al receptor, como si se produjese algún tipo de bloqueo. De este modo se pierde rapidez en el envío del archivo, pero se gana seguridad de que no van a producirse problemas durante dicho envío. En todo caso, el retardo introducido es parametrizable modificando el parámetro correspondiente del fichero de configuración del servidor, así se consigue ajustar del mejor modo posible el funcionamiento, a las circunstancias del sistema en que corra el servidor, es decir, puede haber sistemas donde no sea necesario introducir un retardo elevado para que el servicio UDP funcione bien, pero también puede haber otros sistemas donde, a no ser que se introduzca un retardo significativo, el servicio no funcione en condiciones.

Se ha comentado ya en varias ocasiones en este apartado, la existencia de un archivo que guarda la configuración inicial para el funcionamiento del programa servidor. Veamos que parámetros almacena dicho archivo:

- **ServidorAV.ini.**

```
#Configuración
#Wed Jun 07 22:43:48 GMT+02:00 2000
controlador=sun.jdbc.odbc.JdbcOdbcDriver
datasource=jdbc:odbc:BdatosAV
basededatos=BdatosAV
puertomediaTCP=2000
```

```
puertomediaUDP=2001
directorioarchivos=bdatos
bloqueenvio=8192
retardoenvio=500
```

El parámetro *controlador* guarda el valor del controlador que se ha de cargar para acceder a la base de datos de archivos. Como puede observarse, se utiliza en principio el controlador JDBC-ODBC. El parámetro *datasource* define cuál es el origen de datos que se accede. El parámetro *basededatos* obviamente define el nombre de la base de datos que se accede. Los parámetros *puertomediaTCP* y *puertomediaUDP* definen los puertos en que escuchan los servicios TCP y UDP del servidor. El parámetro *directorioarchivos* guarda el directorio en que se encuentran los archivos multimedia. El parámetro *bloqueenvio* define el tamaño de los datagramas UDP que se envían, y en el caso TCP, define el tamaño de los bloques de datos que se mandan por el *stream* conectado al *socket* de envío de datos. Finalmente, el parámetro *retardoenvio* define el tiempo de espera entre datagramas enviados.

Una vez analizado el fichero de configuración veamos la estructura de la clase que se encarga de leerlo:

- **Util.java.**

```
/*
 * @(#) Util.java
 *
 * Versión 1    27/04/00
 * Copyright 2000    Jacinto Navarro González
 *
 */

// Paquete al que pertenece la clase
package Servidor;

// Clases y paquetes importados
import java.io.*;
import java.util.*;

/**
 * Clase que define atributos y métodos de clase,
 * implementa un método para leer el fichero de configuración.
 *
 */
public class Util {
    private static final String FICHERO_INI = "ServidorAV.ini";
    private static String controlador; // Controlador para acceso a bd
    private static String datasource; // Origen de datos al que se accede
    private static String basededatos; // Nombre de la base de datos
    private static String dirArchivos; // Directorio de archivos multimedia
    private static int puertoMediaTCP; // Puerto de escucha servicio TCP
    private static int puertoMediaUDP; // Puerto de escucha servicio UDP
    private static int bloqueEnvio; // Tamaño de envío de datagramas o streams
    private static int retardoEnvio; // Retardo de envío entre datagramas

    /**
     * Lee todos los parámetros del fichero ini y los almacena
     * en los atributos de clase
     *
     */
```

```
public static void leeFicheroIni(){
    Properties propiedades = new Properties();
    FileInputStream entrada = null;
    String valor;
    Integer entero;

    try {
        entrada = new FileInputStream(FICHERO_INI);
    } catch (FileNotFoundException e) {
        System.err.println("No se encontró el fichero de " +
                "configuración: " + FICHERO_INI + ". Programa abortado.");
        System.exit(0);
    }

    try {
        // Carga los valores del ini en un objeto
        // de propiedades
        propiedades.load(entrada);
    } catch (IOException e) {
        System.err.println("Error al leer el fichero de configuración '" +
                FICHERO_INI + "'. Programa abortado.");
        System.exit(0);
    }

    // Extrae todos los valores del objeto de propiedades
    // y los almacena en sus variables de clase
    controlador = propiedades.getProperty("controlador");
    datasource = propiedades.getProperty("datasource");
    basededatos = propiedades.getProperty("basededatos");
    dirArchivos = propiedades.getProperty("directorioarchivos");
    valor = propiedades.getProperty("puertomediaTCP", "2000");
    entero = new Integer(valor);
    puertoMediaTCP = entero.intValue();
    valor = propiedades.getProperty("puertomediaUDP", "2001");
    entero = new Integer(valor);
    puertoMediaUDP = entero.intValue();
    valor = propiedades.getProperty("bloqueenvio","8192");
    entero = new Integer(valor);
    bloqueEnvio = entero.intValue();
    valor = propiedades.getProperty("retardoenvio","1000");
    entero = new Integer(valor);
    retardoEnvio = entero.intValue();
}

public static String devuelveControlador() {
    return controlador;
}

public static String devuelveDSource() {
    return datasource;
}

public static String devuelveBD() {
    return basededatos;
}

public static String devuelveDirArchivos() {
    return dirArchivos;
}

public static int devuelvePuertoMediaTCP() {
    return puertoMediaTCP;
}

public static int devuelvePuertoMediaUDP() {
    return puertoMediaUDP;
}
```

```
    public static int devuelveBloqueEnvio() {
        return bloqueEnvio;
    }

    public static int devuelveRetardoEnvio() {
        return retardoEnvio;
    }
}
```

Esta clase actúa a modo de almacén de variables globales del programa servidor. Dichos valores globales son almacenados en los atributos de clase, y leídos del fichero de configuración por el método *leeFicheroIni( )*. Para dicha lectura, los valores se cargan en primer lugar en una estructura de la clase *Properties*, y posteriormente se extraen de dicha estructura y se almacenan en las variables de clase.

4.2. Análisis de la implementación del cliente.

La parte de la aplicación correspondiente al cliente consta de veintisiete archivos Java, así pues, analizaremos con detalle sólo las partes del código más significativas de estos archivos. Vemos en primer lugar el código del archivo que implementa la estructura básica del programa cliente:

- **ClienteAV.java.**

```
/*
 * @(#) ClienteAV.java
 *
 * Versión 1    22/04/99
 * Copyright 1999    Jacinto Navarro González
 *
 */

// Paquete al que pertenece la clase
package Cliente;

// Clases y paquetes importados
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;

/**
 * Esta clase implementa una interfaz gráfica de usuario para
 * permitir conectarse a un servidor remoto y acceder, mediante RMI
 * a la información sobre archivos multimedia almacenada en su base
 * de datos. De entre la información que se recupera, se pueden elegir
 * archivos para reproducirlos empleando tres protocolos distintos.
 *
 */
public class ClienteAV extends Frame {
    private MenuItem conectItem; // Item para mostrar diálogo de conexión
        private MenuItem configItem; // Item para mostrar diálogo de puertos
        private MenuItem desconectItem; // Item para desconectar del servidor
        private MenuItem salirItem; // Item para salir de la aplicación
        private MenuItem buscarItem; // Item para mostrar diálogo de búsqueda
                                     // de archivos
        private MenuItem seleccionItem; // Item para mostrar diálogo de
                                        // selección de archivos
```

```
    private MenuItem todosItem; // Item para buscar todos los archivos del
                                // servidor
    private MenuItem recienteItem; // Item para mostrar la última búsqueda
                                   // de archivos
    private MenuItem menuItems[]; // Guarda las últimas diez direcciones
                                  // de conexión
    private MenuItem acercaItem; // Item para mostrar diálogo de
                                 // información
    private CheckboxMenuItem tcpStreamItem; // Selecciona protocolo TCP
                                            // stream
    private CheckboxMenuItem tcpMemoryItem; // Selecciona protocolo TCP
                                            // con buffer
    private CheckboxMenuItem udpMemoryItem; // Selecciona protocolo UDP
                                            // con buffer
    private Menu recientesMenu; // Menú con las últimas direcciones de
                                // conexión
    private MenuItemListener menuItemListener; // Escuchador para los
                                               // items del menú
    private CheckboxItemListener checkboxItemListener; // Escuchador para
                                                       // items checkbox
    private Vector players = new Vector(20); // Almacena los reproductores
                                             // activos
    private int nplayers = 0; // Número de reproductores activos
    private static BarraEstado barraEstado; // Barra de estado de la
                                            // ventana principal
    private static AreaMensajes areaMensajes; // Área de mensajes durante
                                              // la reproducción
    private String ultimaBusqueda; // Resultado de la última búsqueda de
                                   // archivos
    private static String host; // Host al que se está conectado
    private int protocolo; // Protocolo activo

    private ConexionRMI conexionRMI = null; // Objeto con los métodos
                                            // remotos
    public static final String tituloVentana = "Cliente audio-video";
    private static final int ancho = 600;
    private static final int alto = 500;


public static void main(String args[]) {
    ClienteAV clienteAV;
    Image imagen;

    clienteAV = new ClienteAV(tituloVentana);
    // Lee el fichero de configuración
    Util.leeFicheroIni(clienteAV);
    clienteAV.creaMenu();

    imagen = Util.devuelveImagen(clienteAV, Util.JMF);
    clienteAV.setIconImage(imagen);

    // Centra la aplicación en pantalla
    Util.centraObjeto(clienteAV, ancho, alto);

    clienteAV.setBackground(Color.gray);
    clienteAV.setLayout(null);
    clienteAV.setVisible(true);
}
```

En el método *main( )* se instancia y se configura el objeto cliente. Al igual que ocurría en el programa servidor, existe un archivo de configuración (‘ClienteAV.ini’), que contiene una serie de valores modificables por el usuario, que el cliente utiliza por defecto (puerto del servicio TCP, del servicio UDP, últimos servidores accedidos, etc.).

```
public ClienteAV(String s) {
    super(s);
```

```
        // Protocolo seleccionado por defecto
        this.protocolo = Util.TCP_STREAM;
        // Añade la barra de estado a la ventana
        add(barraEstado = new BarraEstado(tituloVentana));

        addComponentListener(new ComponentAdapter() {
            public void componentResized(ComponentEvent ce) {
                Rectangle rect = new Rectangle();
                try {
                    rect = devuelveRectangulo();
                } catch (Exception e) {
                    // Ignora excepción
                }
                barraEstado.situa(rect.width, rect.height);
                try {
                    areaMensajes.situa(rect.width, rect.height);
                } catch (NullPointerException e) {
                    // Ignora excepción
                }
            }
        });

        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                dispose();
                System.exit(0);
            }
        });
    }
```

En el constructor de la clase *ClienteAV* se añade la barra de estado a la ventana principal de la aplicación, además de los escuchadores de eventos necesarios. Así, el *ComponentAdapter* controla los cambios de tamaño de la ventana principal (que los objetos que contiene, como la barra de estado, o el área de mensajes que muestra situaciones de error durante la reproducción de los archivos, adapten su tamaño al nuevo tamaño de la ventana) debido a la implementación que se ha realizado del método *componentResized( )*, mientras que el *WindowAdapter* escucha los eventos de ventana que se producen, y en este caso sólo procesa el de cerrado de la ventana.

```
    /**
     * Crea y añade el menú y todos sus items al Frame principal
     *
     */
    public void creaMenu() {
        MenuBar mbar;
        Menu conexionMenu;
        Menu archivosMenu;
        Menu protocoloMenu;
        Menu acercaMenu;

        menuItems = new MenuItem[10];
        mbar = new MenuBar();
        menuItemListener = new MenuItemListener();
        checkboxItemListener = new CheckboxItemListener();

        conexionMenu = new Menu("Conexión", true);
        archivosMenu = new Menu("Archivos", true);
        protocoloMenu = new Menu("Protocolo", true);
        recientesMenu = new Menu("Recientes", true);
        acercaMenu = new Menu("Acerca de", true);

        conexionMenu.add(conectItem =
                new MenuItem("Conectar con...",new
```

```
                        MenuShortcut(KeyEvent.VK_C)));
    conexionMenu.add(configItem =
            new MenuItem("Configurar puertos...",new
                        MenuShortcut(KeyEvent.VK_P)));
    conexionMenu.add(desconectItem =
            new MenuItem("Desconectar", new MenuShortcut(KeyEvent.VK_D)));
    conexionMenu.add(recientesMenu);
    conexionMenu.addSeparator();
    conexionMenu.add(salirItem =
            new MenuItem("Salir", new MenuShortcut(KeyEvent.VK_S)));
    archivosMenu.add(buscarItem =
            new MenuItem("Buscar...", new MenuShortcut(KeyEvent.VK_B)));
    archivosMenu.add(seleccionItem =
            new MenuItem("Seleccionar...", new
                        MenuShortcut(KeyEvent.VK_L)));
    archivosMenu.add(todosItem =
            new MenuItem("Todos", new MenuShortcut(KeyEvent.VK_T)));
    archivosMenu.add(recienteItem =
            new MenuItem("Reciente", new MenuShortcut(KeyEvent.VK_R)));

    protocoloMenu.add(tcpStreamItem =
            new CheckboxMenuItem("TCP stream", true));
    protocoloMenu.add(tcpMemoryItem =
            new CheckboxMenuItem("TCP memory", false));
    protocoloMenu.add(udpMemoryItem =
            new CheckboxMenuItem("UDP memory", false));
    acercaMenu.add(acercaItem =
            new MenuItem("Acerca de ...", new
                        MenuShortcut(KeyEvent.VK_A)));

    if (Util.nrecientes > 0) {
        for (int i = 0; i < Util.nrecientes; i++) {
            // Crea el menú de direcciones recientes,
            // se guardan en un vector de la clase Util
            recientesMenu.add(menuItems[i] =
                    new MenuItem((String) Util.servidores.elementAt(i)));
                    menuItems[i].addActionListener(menuItemListener);
        }
    } else {
        recientesMenu.setEnabled(false);
    }
    desconectItem.setEnabled(false);
    buscarItem.setEnabled(false);
    todosItem.setEnabled(false);
    seleccionItem.setEnabled(false);
    recienteItem.setEnabled(false);

    mbar.add(conexionMenu);
    mbar.add(archivosMenu);
    mbar.add(protocoloMenu);
    mbar.add(acercaMenu);
    setMenuBar(mbar);

    conectItem.addActionListener(menuItemListener);
    configItem.addActionListener(menuItemListener);
    desconectItem.addActionListener(menuItemListener);
    salirItem.addActionListener(menuItemListener);
    buscarItem.addActionListener(menuItemListener);
    seleccionItem.addActionListener(menuItemListener);
    todosItem.addActionListener(menuItemListener);
    recienteItem.addActionListener(menuItemListener);
    tcpStreamItem.addItemListener(checkboxItemListener);
    tcpMemoryItem.addItemListener(checkboxItemListener);
    udpMemoryItem.addItemListener(checkboxItemListener);
    acercaItem.addActionListener(menuItemListener);
}
```

En el método *creaMenu* se construye el menú de la aplicación. El bucle introducido recorre el vector que almacena las direcciones de los últimos servidores accedidos, y crea una entrada de menú para cada dirección. Dichas direcciones se almacenan, como se ha comentado antes, en el fichero de configuración.

```
/**
 * Devuelve la posición y las dimensiones del Frame principal,
 * para utilizarlo en los cambios de tamaño.
 *
 */
public Rectangle devuelveRectangulo() {
    Point puntoSupIzq;
    Rectangle rectangulo = new Rectangle();

    puntoSupIzq = this.getLocationOnScreen();
    rectangulo.x = puntoSupIzq.x;
    rectangulo.y = puntoSupIzq.y;
    rectangulo.width = this.getSize().width;
    rectangulo.height = this.getSize().height;

    return rectangulo;
}

/**
 * Devuelve la dirección del servidor al que se accede
 */
public static String devuelveHost() {
    return host;
}

/**
 * Modifica el texto de la barra de estado
 */
public static void textoBarraEstado(String contenido) {
    barraEstado.texto(contenido);
}

/**
 * Modifica el texto del área de mensajes en reproducción
 */
public static void textoAreaMensajes(String contenido) {
    areaMensajes.texto(contenido);
}

/**
 * Muestra el área de mensajes de reproducción
 */
public static void muestraAreaMensajes() {
    areaMensajes.muestra();
}

/**
 * Esconde el área de mensajes de reproducción
 */
public static void escondeAreaMensajes() {
    areaMensajes.esconde();
}

/**
 * Determina si el área de mensajes de reproducción está visible
 */
public static boolean esVisibleAreaMensajes() {
    return areaMensajes.isVisible();
}
```

```
/**
 * Devuelve el protocolo activo
 */
public int devuelveProtocolo() {
    return protocolo;
}

/**
 * Elimina un reproductor del vector que los almacena, cuando
 * un reproductor es cerrado
 */
public void eliminaPlayer(int orden) {
    players.removeElementAt(orden);
    nplayers --;
    for (int i = 0; i < nplayers; i ++) {
        // Si se elimina un reproductor intermedio
        // reordena el vector y reasigna los números de orden
        JMFPlayer jmfPlayer = (JMFPlayer) players.elementAt(i);
        jmfPlayer.fijaOrden(i);
    }
}
```

El cliente permite reproducir varios archivos multimedia simultáneamente, incluso habiendo obtenido cada uno los datos multimedia, mediante un protocolo distinto. Debido a ésto, es necesario controlar de algún modo dichos reproductores, para poder tener acceso a ellos una vez han sido instanciados, para ello se almacenan, conforme se van creando, en un vector. El método *eliminaPlayer* actualiza el vector cuando uno de los reproductores es cerrado por el usuario, o debido a algún tipo de error durante la reproducción. Al mismo tiempo, reasigna el número de orden de los reproductores que quedan abiertos, para no perder el acceso a ninguno de ellos.

```
/**
 * Intenta establecer una conexión con un servidor remoto vía RMI
 */
public void intentaConectar(String direccion) {
    boolean conexionOK;
    Rectangle rect;
    Image imagen;

    setCursor(Cursor.WAIT_CURSOR);

    barraEstado.texto("Estableciendo conexión con:   " + host);

    rect = devuelveRectangulo();
    // Añade un área de mensajes para la posible nueva conexión
    add(areaMensajes = new AreaMensajes());
    areaMensajes.situa(rect.width, rect.height);

    // Intenta instanciar el objeto remoto
    conexionRMI = new ConexionRMI(direccion, Util.devuelvePuertoRMI());
    conexionOK = conexionRMI.devuelveConexionOK();
```

En este punto del método *intentaConectar(...)* se instancia el objeto que utiliza los métodos remotos ofrecidos por el servidor en su interfaz remota. La clase de la que se instancia el objeto es *ConexionRMI* y se analizará posteriormente. Al constructor de la clase se le pasa la dirección de la máquina donde está registrado el objeto remoto, y el puerto donde escucha el registro de objetos remotos (el valor de este puerto es leido del fichero de configuración, y por defecto es el 1099). El método *devuelveConexionOK( )* informa mediante un valor booleano, del éxito o fracaso de la conexión al registro de

objetos remotos. Así pues, a efectos del programa, conectarse o no al servidor de archivos, depende del éxito o fracaso de esta conexión.

```
setCursor(Cursor.DEFAULT_CURSOR);
if (conexionOK) {

    conectItem.setEnabled(false);
    recientesMenu.setEnabled(false);
    desconectItem.setEnabled(true);
    buscarItem.setEnabled(true);
    seleccionItem.setEnabled(true);
    todosItem.setEnabled(true);

    String textoMensaje = "La conexión con el servidor ha " +
            "sido establecida satisfactoriamente.";
    imagen = Util.devuelveImagen(this, Util.INFORMACION);
    setTitle(tituloVentana + " - " + host);
    barraEstado.texto("Conectado a:   " + host);
    DialogoMensaje mensaje = new DialogoMensaje(this, tituloVentana,
            textoMensaje, imagen, true);
    mensaje.fijaCentrado(true);
    mensaje.fijaVisible(true);

    // Se añade la dirección de la nueva conexión al vector
    // que almacena las direcciones de conexión recientes
    if (!Util.servidores.contains(host)) {
        if (Util.nrecientes == 10) {
            // Si la nueva dirección no estaba ya pero el vector
            // está lleno se elimina la más antigua y mete la nueva
            for (int i = 1; i < Util.nrecientes; i++) {
                Util.servidores.set(i - 1,
                        Util.servidores.elementAt(i));
                menuItems[i - 1].setLabel((String)
                        Util.servidores.elementAt(i - 1));
            }
            Util.servidores.set(Util.nrecientes - 1, host);
            menuItems[Util.nrecientes - 1].setLabel(host);
            // Almacena la dirección de conexión en el ini
            Util.escribeFicheroIni(this);
        } else {
            // Si la nueva dirección no estaba y queda espacio
            // libre en el vector de direcciones recientes
            Util.servidores.addElement(host);
            recientesMenu.add(menuItems[Util.nrecientes] =
                    new MenuItem((String)
                     Util.servidores.elementAt(Util.nrecientes)));
      menuItems[Util.nrecientes].addActionListener(menuItemListener);
            Util.nrecientes++;
            // Almacena la dirección de conexión en el ini
            Util.escribeFicheroIni(this);
        }
    }
```

Como puede observarse, en caso de éxito en la conexión, se actualiza el vector que contiene las direcciones de las últimas conexiones, y una vez éste ha sido actualizado, se guarda esta información en el archivo de configuración para que en próximas ejecuciones del programa se disponga de esta información actualizada.

```
} else {
    String textoMensaje = "No ha podido establecerse la conexión con
            el servidor.";
    imagen = Util.devuelveImagen(this, Util.EXCLAMACION);
    DialogoMensaje mensaje = new DialogoMensaje(this, tituloVentana,
            textoMensaje, imagen, true);
    mensaje.fijaCentrado(true);
```

```
            mensaje.fijaVisible(true);
            barraEstado.texto(tituloVentana);
            conexionRMI = null;
        }
    }

    /**
     * Muestra un diálogo con una lista rellenada de los datos
     * de los archivos encontrados tras una búsqueda
     */
    public void muestraArchivos(String listaArchivos) {
        boolean busquedaOK;
        String archivo;
        long longitud;
        DialogoSeleccion dialogoSeleccion;
        Image imagen;
        int numArchivos;
        String [] seleccionados;
        CreaPlayer player;
        JMFPlayer jmfPlayer;

        busquedaOK = conexionRMI.devuelveBusquedaOK();
        if (busquedaOK) {

            if ((listaArchivos != null) && (listaArchivos.length() > 0)) {
                barraEstado.texto("Selecciona archivo a reproducir de:   " +
                        host);
                // Guarda la lista de archivos de la última búsqueda
                ultimaBusqueda = listaArchivos;
                recienteItem.setEnabled(true);
                // Muestra el diálogo que permite seleccionar archivos para
                // su reproducción
                dialogoSeleccion = new DialogoSeleccion(this, tituloVentana,
                        "Seleccione el archivo a reproducir:", 15, null,true);
                dialogoSeleccion.fijaCentrado(true);
                dialogoSeleccion.rellenaArchivos(listaArchivos);
                dialogoSeleccion.fijaVisible(true);

                if (!(dialogoSeleccion.fueCancelado())) {

                    numArchivos = dialogoSeleccion.devuelveNumArchivos();
                    // Obtiene un vector con todos los nombres de archivos
                    // seleccionados
                    seleccionados = dialogoSeleccion.devuelveArchivos();

                    for (int i = 0; i < numArchivos; i ++) {
                        archivo =
                          dialogoSeleccion.devuelveArchivo(seleccionados[i]);
                        longitud =
                          dialogoSeleccion.devuelveLongitud(seleccionados[i]);
                        // Crea el reproductor para el archivo
                        player = new CreaPlayer(this, host, archivo,longitud);
                        // Da un orden a los reproductores y los guarda en un
                        // vector
                        if (player.devuelveResultado()) {
                            jmfPlayer = player.devuelveJMFPlayer();
                            jmfPlayer.fijaOrden(nplayers);
                            players.addElement(jmfPlayer);
                            nplayers ++;
                        }
                        if (protocolo == Util.UDP_MEMORY) {
                            try {
                                // Para evitar bloqueos en la comunicación UDP
                                int retardo=Util.devuelveRetardoCreacionUDP();
                                Thread.currentThread().sleep(retardo);
                            } catch (InterruptedException ie) {
                                // Ignora excepción
                            }
```

```
                    }
                }

                barraEstado.texto("Conectado a:   " + host);
            }
```

En el método *muestraArchivos(...)* se muestra al usuario una ventana que incluye una lista, con todos los archivos resultantes de una búsqueda en el servidor. El usuario, ante esta lista puede optar por seleccionar un único archivo para su reproducción, o puede seleccionar varios de ellos. En el bucle implementado, para cada uno de los archivos seleccionados, se crea un objeto de la clase *CreaPlayer*. Dicho objeto crea un manejador de datos multimedia, a partir de un objeto de la clase *DataSource*. Como vimos en el apartado dedicado a la descripción del paquete JMF de Java, un *DataSource* es un objeto que encapsula la localización de datos multimedia así como el protocolo y el software usados para representar dichos datos. El objeto *DataSource* es el encargado de establecer una conexión con el proceso servidor de archivos multimedia, y de traerse e interpretar los datos del archivo correspondiente, creando para ello el manejador correspondiente para el tipo de datos del fichero (WAV, AVI, MPEG, etc.). Si la creación del objeto de la clase *CreaPlayer* es correcta, se extrae de dicho objeto uno de la clase *JMFPlayer*, que corresponde a un reproductor multimedia propiamente dicho, puesto que crea una interfaz gráfica para el manejo de los datos multimedia que se reciben. En este punto se añaden los reproductores creados al vector que los almacena, y se le asigna a cada reproductor un índice para poder acceder a ellos, pues sino fuese así, en el momento de crear uno perderíamos la referencia con el anterior. Se puede observar como la comunicación UDP también ocasiona algún problema en el extremo del cliente como ocurría en los envíos del servidor, puesto que es necesario introducir un retardo entre la creación de un reproductor UDP y otro, para evitar bloqueos cuando el objeto *DataSource* del reproductor, envía datagramas para establecer conexión con el servidor. El resto del método hace un tratamiento de los errores de comunicación, o de búsqueda en la base de datos.

```
        } else {
            if (listaArchivos == null) {
                String textoMensaje = "Error al acceder a la base de datos
                        del servidor.";
                imagen = Util.devuelveImagen(this, Util.SIGNOSTOP);

                barraEstado.texto("Conectado a:   " + host);
                DialogoMensaje mensaje = new DialogoMensaje(this,
                        tituloVentana, textoMensaje, imagen, true);
                mensaje.fijaCentrado(true);
                mensaje.fijaVisible(true);

            } else {
                String textoMensaje = "No encontrados archivos para ese
                        criterio de búsqueda.";
                imagen = Util.devuelveImagen(this, Util.EXCLAMACION);
                barraEstado.texto("Conectado a:   " + host);
                DialogoMensaje mensaje = new DialogoMensaje(this,
                        tituloVentana, textoMensaje, imagen, true);
                mensaje.fijaCentrado(true);
                mensaje.fijaVisible(true);
            }
        }
    } else {
```

```
            String textoMensaje = "Error en la comunicación con el servidor.";
            imagen = Util.devuelveImagen(this, Util.SIGNOSTOP);

            barraEstado.texto("Conectado a:   " + host);
            DialogoMensaje mensaje = new DialogoMensaje(this, tituloVentana,
                    textoMensaje, imagen, true);
            mensaje.fijaCentrado(true);
            mensaje.fijaVisible(true);
        }
    }

/**
 * Esta clase implmenta un escuchador para los eventos de menú
 * de tipo CheckBox
 * Permite activar sólo uno de los protocolos de transporte
 * implementados
 */
class CheckboxItemListener implements ItemListener {

    public void itemStateChanged(ItemEvent ie) {
        CheckboxMenuItem item = (CheckboxMenuItem) ie.getSource();

        if (item == tcpStreamItem) {
            if (!tcpStreamItem.getState()) {
                tcpStreamItem.setState(true);
                tcpMemoryItem.setState(false);
                udpMemoryItem.setState(false);
            } else {
                tcpMemoryItem.setState(false);
                udpMemoryItem.setState(false);
            }
            protocolo = Util.TCP_STREAM;
        }
        if (item == tcpMemoryItem) {
            if (!tcpMemoryItem.getState()) {
                tcpMemoryItem.setState(true);
                tcpStreamItem.setState(false);
                udpMemoryItem.setState(false);
            } else {
                tcpStreamItem.setState(false);
                udpMemoryItem.setState(false);
            }
            protocolo = Util.TCP_MEMORY;
        }
        if (item == udpMemoryItem) {
            if (!udpMemoryItem.getState()) {
                udpMemoryItem.setState(true);
                tcpStreamItem.setState(false);
                tcpMemoryItem.setState(false);
            } else {
                tcpStreamItem.setState(false);
                tcpMemoryItem.setState(false);
            }
            protocolo = Util.UDP_MEMORY;
        }
    }
}

/**
 * Esta clase implmenta un escuchador para los eventos de menú.
 * En el metodo actionPerformed se implementa el comportamiento
 * del programa para cada entrada de menú
 */
class MenuItemListener implements ActionListener {

    public synchronized void actionPerformed(ActionEvent ae) {
        DialogoPregunta dialogoPregunta = null;
        DialogoSiNo dialogoSiNo = null;
```

```
        DialogoBusqueda dialogoBusqueda = null;
        DialogoParametros dialogoParametros = null;
        boolean busquedaOK;
        boolean dialogoExiste = false;
        Image imagen;
        int puertoTCP;
        int puertoUDP;
        String archivos;
        String listaArchivos;
        JMFPlayer jmfPlayer = null;

        MenuItem item = (MenuItem) ae.getSource();

        if (Util.servidores.contains(item.getLabel())) {
            // Intenta conectar a uno de los servidores recientes
            host = item.getLabel();
            intentaConectar(host);
        }

        if (item == salirItem) {
            dispose();
            System.exit(0);
        }

        if (item == seleccionItem) {
            if (!dialogoExiste) {
                dialogoExiste = true;
                dialogoBusqueda = new DialogoBusqueda(ClienteAV.this,
                        tituloVentana, true);
                if (!dialogoBusqueda.fueCancelado()) {
                    String fechaIni = dialogoBusqueda.devuelveFechaIni();
                    String fechaFin = dialogoBusqueda.devuelveFechaFin();
                    int tamMin = dialogoBusqueda.devuelveTamMin();
                    int tamMax = dialogoBusqueda.devuelveTamMax();
                    String tipo = dialogoBusqueda.devuelveTipo();
                    if ((fechaIni != null) && (fechaFin != null)
                            && (tamMin != -1) && (tamMax != -1)) {
                        setCursor(Cursor.WAIT_CURSOR);
                        barraEstado.texto("Buscando archivos en:   " + host);
                        // Ejecuta un método remoto ofrecido por el servidor
                        listaArchivos = conexionRMI.buscaSeleccion(fechaIni,
                                fechaFin, tamMin, tamMax, tipo);

                        setCursor(Cursor.DEFAULT_CURSOR);
                        muestraArchivos(listaArchivos);
                    }
                }
                dialogoExiste = false;
                dialogoBusqueda = null;
            }
        }
```

El bloque anterior implementa el comportamiento del programa en caso de que el usuario desee hacer una selección elaborada de los archivos presentes en el servidor. Se muestra un diálogo al usuario en el que se piden datos para acotar la búsqueda: fechas mínima y máxima, tamaños mínimo y máximo de los archivos, y tipo del archivo (audio o video). El usuario puede introducir los datos que considere necesarios. Una vez se tienen los datos para efectuar la búsqueda, se invoca el método *buscaSeleccion(...)* del objeto de la clase *ConexionRMI* que como se ha comentado antes, utiliza los métodos ofrecidos por el servidor en su interfaz remota. El resultado que devuelve el método es procesado y mostrado al usuario.

```
        if (item == configItem) {
```

```
        if (!dialogoExiste) {
            dialogoExiste = true;
            Integer puertoTCPI = new
                    Integer(Util.devuelvePuertoMediaTCP());
            Integer puertoUDPI = new
                    Integer(Util.devuelvePuertoMediaUDP());
            String valorTCP = puertoTCP.toString();
            String valorUDP = puertoUDP.toString();

            imagen = Util.devuelveImagen(ClienteAV.this,
                    Util.INTERROGACION);

            // Muestra el diálogo que solicita los puertos TCP y UDP
            // Pone por defecto los valores del fichero ini
            dialogoParametros = new DialogoParametros(ClienteAV.this,
                    tituloVentana, "Puerto TCP:", "Puerto UDP:", valorTCP,
                    valorUDP, 5, imagen, true);
            dialogoParametros.habilitaAceptar();
            dialogoParametros.fijaCentrado(true);
            dialogoParametros.fijaVisible(true);

            if (!(dialogoParametros.fueCancelado())) {
                puertoTCP = dialogoParametros.devuelveRespuesta1();
                puertoUDP = dialogoParametros.devuelveRespuesta2()
                Util.fijaPuertoMediaTCP(puertoTCP);
                Util.fijaPuertoMediaUDP(puertoUDP);
            }
            dialogoExiste = false;
        }
    }
```

En este bloque se pide al usuario que introduzca los puertos TCP y UDP en que escucha el servidor. Se muestra un diálogo al usuario con dos campos editables, que tienen como valor inicial el determinado por los correspondientes parámetros del fichero de configuración.

```
    if (item == conectItem) {
        if (!dialogoExiste) {
            dialogoExiste = true;

            barraEstado.texto("Establece conexión con un servidor de
                    audio-video");
            imagen = Util.devuelveImagen(ClienteAV.this,
                    Util.INTERROGACION);
            // Muestra diálogo que solicita dirección del servidor
            // multimedia
            dialogoPregunta = new DialogoPregunta(ClienteAV.this,
                    tituloVentana, "Dirección del servidor:", "", 30,
                    imagen, true);
            dialogoPregunta.fijaCentrado(true);
            dialogoPregunta.fijaVisible(true);

            if (!(dialogoPregunta.fueCancelado())) {
                host = dialogoPregunta.devuelveRespuesta();
                intentaConectar(host);
            } else {
                barraEstado.texto(tituloVentana);
            }
            dialogoPregunta = null;
            dialogoExiste = false;
        }
    }
```

En el bloque anterior se solicita al usuario que introduzca la dirección del servidor multimedia al que se desea conectar. Se muestra un diálogo al usuario con un campo editable, en el cual el usuario ha de introducir la dirección del servidor .

```
if (item == desconectItem) {
    if (nplayers > 0) {
        imagen = Util.devuelveImagen(ClienteAV.this,
                Util.INTERROGACION);
        String textoDialogo = "Si cierra la conexión con " + host + "
                se " + "cerrarán los reproductores activos. ¿Desea
                desconectar?";
        // Muestra diálogo en caso de haber reproductores abiertos
        dialogoSiNo = new DialogoSiNo(ClienteAV.this, tituloVentana,
                textoDialogo, imagen, true);
        dialogoSiNo.fijaCentrado(true);
        dialogoSiNo.fijaVisible(true);

        if (dialogoSiNo.respuestaSi()) {
            conexionRMI = null;
            ClienteAV.this.setTitle(tituloVentana);
            barraEstado.texto(tituloVentana);
            conectItem.setEnabled(true);
            recientesMenu.setEnabled(true);
            desconectItem.setEnabled(false);
            buscarItem.setEnabled(false);
            seleccionItem.setEnabled(false);
            todosItem.setEnabled(false);
            recienteItem.setEnabled(false);
            areaMensajes.esconde();
            areaMensajes = null;
            for (int i = 0; i < nplayers; i ++) {
                jmfPlayer = (JMFPlayer) players.elementAt(i);

                jmfPlayer.cierraPlayer();
                try {
                   // Cierra los reproductores abiertos con un retardo
                   // entre uno y otro para que se complete la
                   // operación del todo
                   Thread.currentThread().sleep(300);
                } catch (InterruptedException ie) {
                    // Ignora excepción
                }

                jmfPlayer.dispose();

                try {
                    Thread.currentThread().sleep(300);
                } catch (InterruptedException e) {
                }
                jmfPlayer = null;
            }
            players.removeAllElements();
            nplayers = 0;
        }
    } else {
        conexionRMI = null;
        ClienteAV.this.setTitle(tituloVentana);
        barraEstado.texto(tituloVentana);
        conectItem.setEnabled(true);
        recientesMenu.setEnabled(true);
        desconectItem.setEnabled(false);
        buscarItem.setEnabled(false);
        seleccionItem.setEnabled(false);
        todosItem.setEnabled(false);
        recienteItem.setEnabled(false);
        areaMensajes.esconde();
```

```
            areaMensajes = null;
        }
    }
```

Cuando el cliente desconecta de un servidor al que estaba conectado, los reproductores que pudiera haber abiertos en ese instante han de cerrarse. En caso de que el usuario definitivamente desee desconectar, se van cerrando y eliminando los reproductores abiertos uno a uno, pero puede observarse como de nuevo es preferible introducir un retardo entre una operación y otra, para evitar que alguno de los reproductores no se cierre satisfactoriamente.

```
    if (item == todosItem) {
        if (!dialogoExiste) {
            dialogoExiste = true;
            setCursor(Cursor.WAIT_CURSOR);
            barraEstado.texto("Buscando archivos en:   " + host);
            // Ejecuta un método de la interfaz remota ofrecida por
            // el servidor
            listaArchivos = conexionRMI.buscaTodos();
            setCursor(Cursor.DEFAULT_CURSOR);
            muestraArchivos(listaArchivos);
            dialogoExiste = false;
        }
    }
```

Otro de los métodos de la clase *ConexionRMI* es invocado en el anterior bloque: *buscaTodos( )* devuelve la lista de todos los archivos existentes en la base de datos del servidor.

```
    if (item == recienteItem) {
        muestraArchivos(ultimaBusqueda);
    }

    if (item == buscarItem) {
        if (!dialogoExiste) {
            dialogoExiste = true;
            barraEstado.texto("Busca archivos en:   " + host);

            imagen = Util.devuelveImagen(ClienteAV.this,
                Util.INTERROGACION);
            // Muestra diálogo de búsqueda elaborada de archivos
            dialogoPregunta = new DialogoPregunta(ClienteAV.this,
                    tituloVentana, "Archivos:", "", 25, imagen, true);
            dialogoPregunta.fijaCentrado(true);
            dialogoPregunta.fijaVisible(true);

            if (!(dialogoPregunta.fueCancelado())) {
                setCursor(Cursor.WAIT_CURSOR);
                archivos = dialogoPregunta.devuelveRespuesta();
                barraEstado.texto("Buscando archivos en:   " + host);
                // Ejecuta un método de la interfaz remota del servidor
                // se le pasa un patrón de búsqueda de archivos
                listaArchivos = conexionRMI.buscaArchivos(archivos);

                setCursor(Cursor.DEFAULT_CURSOR);
                muestraArchivos(listaArchivos);
            }
            dialogoPregunta = null;
            dialogoExiste = false;
        }
    }
  }
}
```

```
}
```

Finalmente, en el último bloque de la clase se implementa una búsqueda elaborada de archivos. Se muestra un diálogo al usuario solicitándole que introduzca un patrón de búsqueda de archivos, o el nombre de uno o varios archivos en concreto. Empleando el último de los métodos de la interfaz remota del objeto servidor (*buscaArchivos(...)*), se obtienen los archivos que responden al patrón de búsqueda.

Los tres métodos de la interfaz remota del objeto servidor se invocan, como se ha comentado antes, en la clase *ConexiónRMI*. Veamos el código implementado:

- **ConexiónRMI.java.**

```
/*
 * @(#) ConexiónRMI.java
 *
 * Versión 1    24/04/99
 * Copyright 1999    Jacinto Navarro González
 *
 */

// Paquete al que pertenece la clase
package Cliente;

// Clases y paquetes importados
import Servidor.*;
import java.rmi.*;

/**
 * Esta clase invoca los métodos de la interfaz remota del objeto
 * servidor. Para ello busca en el registro de objetos remotos
 * la referencia al objeto servidor
 */
public class ConexionRMI {
    private ServidorAVint server; // Interfaz remota
    private boolean conexionOK; // Resultado de la busqueda en el
                                // registro de objetos remotos
    private boolean busquedaOK; // Resultado de la invocación a los
                                // métodos remotos

    public ConexionRMI(String host, int puerto) {
        Integer entero = new Integer(puerto);

        try {
            // Localiza el objeto servidor y hace un cast a la
            // interfaz remota
            server = (ServidorAVint) Naming.lookup("//" + host + ":" +
                    entero.toString() + "/ServidorAVint");
            conexionOK = true;
        } catch (Exception e) {
            conexionOK = false;
        }
    }

    /**
     * Método en el que se invoca un método de la interfaz remota,
     * devuelve en un String todos los archivos de la base de datos
     * del servidor
     */
    public String buscaTodos() {
        String resultado = new String();
```

```
        try {
            // Invoca un método de la interfaz remota
            resultado = server.obtenerBusquedaTodos();
            busquedaOK = true;
        } catch (RemoteException e) {
            busquedaOK = false;
        }
        return resultado;
    }

    /**
     * Método en el que se invoca un método de la interfaz remota,
     * devuelve en un String los archivos de la base de datos del
     * servidor que cumplen un patrón de búsqueda
     */
    public String buscaArchivos(String archivosBusqueda) {
        String resultado = new String();

        try {
            // Invoca un método de la interfaz remota
            resultado = server.obtenerBusquedaArchivos(archivosBusqueda);
            busquedaOK = true;
        } catch (RemoteException e) {
            busquedaOK = false;
        }
        return resultado;
    }

    /**
     * Método en el que se invoca un método de la interfaz remota,
     * devuelve en un String los archivos de la base de datos del
     * servidor que se ajustan a unos parámetros
     */
    public String buscaSeleccion(String fechaIni, String fechaFin, int tamMin,
                                 int tamMax, String tipo) {
        String resultado = new String();

        try {
            // Invoca un método de la interfaz remota
            resultado = server.obtenerBusquedaSeleccion(fechaIni, fechaFin,
                    tamMin, tamMax, tipo);
            busquedaOK = true;
        } catch (RemoteException e) {
            busquedaOK = false;
        }
        return resultado;
    }

    /**
     * Método que devuelve el resultado de la búsqueda en el registro
     * de objetos remotos
     */
    public boolean devuelveConexionOK() {
        return conexionOK;
    }

    /**
     * Método que devuelve el resultado de las búsquedas en la base de
     * datos del servidor
     */
    public boolean devuelveBusquedaOK() {
        return busquedaOK;
    }
}
```

En el código del constructor de la clase se observa como mediante el método *lookup* de la clase final *Naming* se busca el objeto servidor, y si se encuentra en el registro localizado en la dirección y puerto indicados, se hace un *cast* a la interfaz remota *ServidorAVint*. Con el resultado de la conversión hecha se pueden ya invocar los métodos de la interfaz remota (ahí será donde intervendrá el *stub* que se genera a partir de la compilación RMI, como se comentó en el apartado dedicado a RMI).

Al analizar el código del programa cliente, se ha comentado como se realiza la creación de los reproductores multimedia empleando la clase *CreaPlayer*. Veamos su código:

- **CreaPlayer.java.**

```
/*
 * @(#) CreaPlayer.java
 *
 * Versión 1    25/07/99
 * Copyright 1999    Jacinto Navarro González
 *
 */

// Paquete al que pertenece la clase
package Cliente;

// Clases y paquetes importados
import com.sun.media.*;
import com.sun.media.util.*;
import com.sun.media.content.*;
import java.io.*;
import java.util.*;
import javax.media.*;
import java.awt.*;

/**
 * Esta clase crea un reproductor multimedia JMF para el archivo
 * indicado, y recibiendo sus datos del host indicado y con el
 * protocolo adecuado, para ello crea un tipo de DataSource u otro
 */
public class CreaPlayer {
    private ClienteAV padre; // Objeto ClienteAV que crea el player
    private String archivo; // Archivo a reproducir por el player
    private String host; // Host origen de los datos
    private boolean resultado = false; // Resultado de la creación
    private JMFPlayer jmfPlayer = null; // Objeto JMFPlayer que se crea
    private Player player = null; // Manejador que se utiliza
    private MediaLocator mediaLocator; // Descriptor de los datos media

    public CreaPlayer(ClienteAV padre, String host, String archivo, long
                      longitud) {
        String localizador;
        String textoError;
        int protocolo;

        this.padre = padre;
        this.archivo = archivo;
        this.host = host;

        // Obtiene el protocolo seleccionado en el cliente
        protocolo = padre.devuelveProtocolo();

        switch (protocolo) {
```

```
case Util.TCP_STREAM:
    localizador = new String("tcp_stream://" + host + '/' +
                             archivo);
    // Crea el descriptor de los datos multimedia
    mediaLocator = new MediaLocator(localizador);

    // Crea el datasource que lee directamente
    // del stream de datos vía TCP
    TCPDataSourceStream tcpdss = new TCPDataSourceStream(padre,
            longitud);
   // Establece el descriptor para el datasource
    tcpdss.setLocator(mediaLocator);

    try {
        // Establece la conexión con el servidor
        tcpdss.connect();
    } catch (IOException e) {
        textoError = " Error al conectar con servidor de archivos
                al reproducir archivo '" + archivo + "'.";
        muestraError(textoError);
        return;
    }
    try {
        // Crea el manejador a partir del datasource conectado
        player = Manager.createPlayer(tcpdss);
    } catch (NoPlayerException e) {
        textoError = " No encontrado manejador para el archivo '"
                + archivo + "' con el protocolo utilizado.";
        muestraError(textoError);
        return;
    } catch (IOException e) {
        textoError = " Error de comunicación al crear manejador
                para el archivo '" + archivo + "'.";
        muestraError(textoError);
        return;
    }

    resultado = true;

    // Crea el reproductor con interfaz gráfica,
    // a partir del manejador creado
    jmfPlayer = new JMFPlayer(padre, player, host, archivo);

   break;

case Util.TCP_MEMORY:
    localizador = new String("tcp_memory://" + host + '/' +
                             archivo);
    // Crea el descriptor de los datos multimedia
    mediaLocator = new MediaLocator(localizador);

    // Crea el datasource que lee de un buffer
    // intermedio vía TCP
    TCPDataSourceMemory tcpdsm = new TCPDataSourceMemory(padre,
            longitud);
    // Establece el descriptor para el datasource
    tcpdsm.setLocator(mediaLocator);

    try {
        // Establece la conexión con el servidor
        tcpdsm.connect();
    } catch (IOException e) {
        textoError = " Error al conectar con servidor de archivos
                al reproducir archivo '" + archivo + "'.";
        muestraError(textoError);
        return;
    }
    try {
```

```
                    // Crea el manejador a partir del datasource conectado
                    player = Manager.createPlayer(tcpdsm);

                } catch (NoPlayerException e) {
                    textoError = " No encontrado manejador para el archivo '"
                            + archivo + "' con el protocolo utilizado.";
                    muestraError(textoError);
                    return;
                } catch (IOException e) {
                    textoError = " Error de comunicación al crear manejador
                            para el archivo '" + archivo + "'.";
                    muestraError(textoError);
                    return;
                }

                resultado = true;

                // Crea el reproductor con interfaz gráfica,
                // a partir del manejador creado
                jmfPlayer = new JMFPlayer(padre, player, host, archivo);

                break;

            case Util.UDP_MEMORY:
                localizador = new String("udp_memory://" + host + '/' +
                                         archivo);
                // Crea el descriptor de los datos multimedia
                mediaLocator = new MediaLocator(localizador);

                // Crea el datasource que lee de un buffer
                // intermedio vía UDP
                UDPDataSourceMemory udpdsm = new UDPDataSourceMemory(padre,
                        longitud);
                // Establece el descriptor para el datasource
                udpdsm.setLocator(mediaLocator);

                try {
                    // Establece la conexión con el servidor
                    udpdsm.connect();
                } catch (IOException e) {
                    textoError = " Error al conectar con servidor de archivos
                            al reproducir archivo '" + archivo + "'.";
                    muestraError(textoError);
                    return;
                }
                try {
                    // Crea el manejador a partir del datasource conectado
                    player = Manager.createPlayer(udpdsm);
                } catch (NoPlayerException e) {
                    textoError = " No encontrado manejador para el archivo '"
                            + archivo + "' con el protocolo utilizado.";
                    muestraError(textoError);
                    return;
                } catch (IOException e) {
                    textoError = " Error de comunicación al crear manejador
                            para el archivo '" + archivo + "'.";
                    muestraError(textoError);
                    return;
                }

                resultado = true;

                // Crea el reproductor con interfaz gráfica,
                // a partir del manejador creado
                jmfPlayer = new JMFPlayer(padre, player, host, archivo);
                break;
        }
    }
```

```
    /**
     * Método que muestra un mensaje de error en el área de mensajes
     * del cliente que creó el player. Si el área está oculta la
     * hace visible
     */
    private void muestraError(String texto) {
        padre.textoBarraEstado("Conectado a:   " + host);
        if (!padre.esVisibleAreaMensajes()) {
            padre.muestraAreaMensajes();
        }
        padre.textoAreaMensajes(texto);
    }

    public JMFPlayer devuelveJMFPlayer() {
        return jmfPlayer;
    }

    public boolean devuelveResultado() {
        return resultado;
    }
}
```

Cuando se invoca el constructor de la clase *CreaPlayer*, este recibe como parámetros, el cliente que crea el *player*, el archivo que se desea reproducir, el host donde reside el archivo, y la longitud del archivo a reproducir. Sea cual sea el protocolo que se ha de utilizar, el proceso de creación del reproductor multimedia es el mismo. En primer lugar se crea un string que describe la localización de los datos multimedia. Este string, como puede observarse es similar a un URL. El localizador se utiliza para crear un objeto de la clase *MediaLocator*, éste es un objeto de JMF que describe los datos multimedia que ha de mostrar el reproductor. Posteriormente se crea un objeto *DataSource* al que se le asigna el *MediaLocator* creado mediante el método *setLocator(...)*. El objeto *DataSource* encapsula la localización de los datos multimedia, y el protocolo y el software usados para obtener los datos. Posteriormente se establece una conexión con el servidor de archivos (TCP o UDP según el protocolo utilizado), invocando el método *connect( )* del objeto *DataSource*. Puesto que llegados a este punto, ya se conoce el tipo de los datos multimedia a reproducir, mediante el método *createPlayer(...)* de la clase *Manager* de JMF, que toma como argumento el objeto *DataSource* creado, ya se puede obtener el manejador adecuado para los datos multimedia. Una vez obtenido el manejador, se puede crear finalmente el reproductor multimedia, instanciando un objeto de la clase *JMFPlayer*, al que se le pasa, entre otros parámetros, el manejador antes comentado.

En la clase anterior, se observa como el objeto básico para la creación de un reproductor multimedia es el *DataSource*. A partir de dicho objeto, el reproductor multimedia obtiene los datos a reproducir procedentes del servidor. Analizaremos las tres clases implementadas para crear objetos *DataSource*, una de ellas (*TCPDataSourceStream*) implementa un objeto *DataSource* que lee directamente mediante el protocolo TCP, los datos del archivo conforme llegan por la red, sin realizar procesamiento o carga previa alguna. Esto redundará en una gran rapidez a la hora de mostrar los datos, pero tiene como contrapartida, que algunos tipos de datos multimedia

no podrán ser procesados por este objeto, y otros podrán serlo, pero su presentación será menos versátil. Esto es debido a que este tipo de *DataSource* no permite el reposicionamiento del puntero de lectura de datos, puesto que conforme llegan éstos, se procesan, así pues, no se podrá permitir por ejemplo, saltar a leer datos que todavía no están disponibles (ejemplos de tipos de archivos que no soportan este tipo de descarga son los AVI, WAV o MP3, mientras que por ejemplo, los MPEG soportan esta descarga, pero pierden la posibilidad de reposicionar o repetir la reproducción). La segunda clase que analizaremos implementa un objeto *DataSource* con buffer (*TCPDataSourceMemory)*, es decir, conforme lee los datos, los guarda en un buffer, y el procesamiento de datos leerá de este buffer, una vez se haya llenado con el contenido de todo el archivo. La tercera clase implementa un objeto *DataSource* con buffer (*UDPDataSourceMemory*), muy similar a la anterior, pero empleando UDP en lugar de TCP. Veamos en primer lugar el *datasource* TCP no posicionable, junto con la clase que implementa el *stream* (*TCPSourceStream*) que éste utiliza:

- **TCPDataSourceStream.java.**

```
/*
 * @(#) TCPDataSourceStream.java
 *
 * Versión 1    22/07/99
 * Copyright 1999    Jacinto Navarro González
 *
 */

// Paquete al que pertenece la clase
package Cliente;

// Clases y paquetes importados
import com.sun.media.*;
import com.sun.media.util.*;
import com.sun.media.content.*;
import javax.media.*;
import javax.media.protocol.*;
import java.io.*;
import java.net.*;
import java.awt.*;

/**
 * Esta clase crea un DataSource no posicionable que lee
 * directamente los datos mediante TCP, del stream de entrada
 */
public class TCPDataSourceStream extends PullDataSource {
    // Cliente que crea el reproductor
    private ClienteAV padre;
    // Socket de comunicación
    private Socket socket;
    // Stream de entrada
    private InputStream entrada;
    // Stream de salida
    private OutputStream salida;
    // Archivo que se lee
    private String archivo;
    // Tipo del archivo
    private String tipoArchivo;
    // Host de procedencia del archivo
    private String host;
    // Longitud del archivo
```

```
    private long longitudArchivo;
    // Stream TCP no posicionable
    private TCPSourceStream sourceStream = null;
    // Colección de streams que maneja el DataSource
    private PullSourceStream[] streams = null;
    // Indica si está iniciado el DataSource
    private boolean iniciado = false;

    public TCPDataSourceStream(ClienteAV padre, long longitudArchivo) {
        super();
        this.longitudArchivo = longitudArchivo;
        this.padre = padre;
    }

    /**
     * Extrae del MediaLocator el archivo y el host
     */
    private void parseLocator() {
        String resto = new String();
        int pos1 = 0;
        int pos2 = 0;

        // Comprueba que la conexión está inicializada
        // con un MediaLocator
        initCheck();
        resto = getLocator().getRemainder();
        pos1 = resto.indexOf("//");
        pos2 = resto.indexOf("/", pos1 + 2);
        host = resto.substring(pos1 + 2, pos2);
        archivo = resto.substring(pos2);
    }
```

El método *parseLocator( )* comprueba en primer lugar que la conexión ha sido efectivamente inicializada con un *MediaLocator* adecuado. Se encarga de extraer la dirección del servidor y el archivo a reproducir.

```
    /**
     * Conecta con el servidor de archivos TCP
     */
    public void connect() throws IOException {
        byte [] nombreArchivo = null;

        // Comprueba que la conexión está inicializada
        // con un MediaLocator
        initCheck();
        if (socket != null) {
            disconnect();
        }
        parseLocator();
        // Crea el socket conectado al servidor
        socket = new Socket(host, Util.devuelvePuertoMediaTCP());
        entrada = socket.getInputStream();
        salida = socket.getOutputStream();

        nombreArchivo = archivo.getBytes();
        // Envía al servidor el nombre del archivo solicitado
        salida.write(nombreArchivo, 0, nombreArchivo.length);
    }
```

El método *connect( )* establece una conexión con el proceso servidor de archivos TCP, además de enviar por el *stream* de salida el nombre del archivo solicitado. La conexión queda establecida cuando se instancia el *socket* del *datasource* teniendo como parámetros la dirección del servidor y el puerto en el que escucha conexiones TCP:

```
    /**
     * Desconecta la comunicación cerrando el socket
     */
    public void disconnect() {
        String textoError;

        if (socket == null) {
            return;
        }

        try {
             socket.close();
        } catch (IOException e) {
            textoError = " Error al desconectar del servidor de archivos al
                    reproducir archivo '" + archivo + "'. Protocolo TCP
                    stream";
            muestraError(textoError);
        }
        socket = null;
        entrada = null;
        salida = null;
    }

    /**
     * Inicia el datasource
     */
    public void start() {
        iniciado = true;
    }

    /**
     * Detiene el datasource
     */
    public void stop() {
        String textoError;

        try {
             socket.close();
        } catch (IOException e) {
             textoError = " Error al cerrar la conexión con el servidor de
                     archivos al reproducir archivo '" + archivo +
                     "'. Protocolo TCP stream";
             muestraError(textoError);
        }
    }

    /**
     * A partir de la extensión del archivo, obtiene el tipo
     * entendible por JMF
     */
    public String getContentType() {
        int posicion;
        String extension;
        String tipo;

        String locatorString = getLocator().toExternalForm();
        posicion = locatorString.lastIndexOf(".");
        extension = locatorString.substring(posicion + 1);
        tipo = "unknown";

        // AVI, WAV, MOV, MP3 no soportan el procesamiento
        // mediante este datasource de ahí lo de tipo desconocido
        if (extension.equals("avi")) {
            tipo = "unknown";
        } else if ((extension.equals("mpg")) || (extension.equals("mpeg"))) {
            tipo = "video.mpeg";
        } else if (extension.equals("mov")) {
```

```
            tipo = "unknown";
        } else if (extension.equals("wav")) {
            tipo = "unknown";
        } else if (extension.equals("au")) {
            tipo = "audio.basic";
        } else if ((extension.equals("mid")) || (extension.equals("midi")) ||
                   (extension.equals("rmi"))) {
            tipo = "audio.midi";
        } else if (extension.equals("rmf")) {
            tipo = "audio.rmf";
        } else if ((extension.equals("mp2")) || (extension.equals("mp3"))) {
            tipo = "unknown";
        } else if (extension.equals("viv")) {
            tipo = "video.vivo";
        } else if (extension.equals("gsm")) {
            tipo = "audio.x_gsm";
        }

        this.tipoArchivo = tipo;

        return tipo;
    }
```

El método *getContentType( )* devuelve el tipo del archivo a reproducir. La asignación del tipo a un archivo, viene determinada por la extensión del archivo. Los tipos de archivo que se devuelven para cada extensión son valores establecidos por el paquete JMF. Puede observarse como para archivos de tipo WAV, AVI, MOV, MP3, el tipo devuelto es *unknown*, esto es debido a que no se puede crear un reproductor válido basado en este *datasource* para procesar archivos de esos tipos. La razón ya se ha explicado con anterioridad, esos tipos de archivo no se pueden procesar secuencialmente, puesto que entre sus datos hay saltos entre posiciones no consecutivas.

```
    /**
     * Obtiene la colección de streams para este datasource
     */
    public PullSourceStream[] getStreams() {
        String textoError;

        streams = new PullSourceStream[1];
        try {
             // Crea un stream no posicionable TCP
             sourceStream = new TCPSourceStream(padre, archivo, host,
                     longitudArchivo, tipoArchivo, socket.getInputStream());
             streams[0] = sourceStream;
        } catch (IOException e) {
            textoError = " Error al crear el proceso receptor del archivo '"+
                    archivo + "'. Protocolo TCP stream.";
            muestraError(textoError);
        }
        return streams;
    }
```

El método *getStreams( )* devuelve el stream que maneja este *datasource*, en la instanciación se le pasa el *stream* de entrada creado para el *socket* conectado al servidor. De este *stream* lee el *datasource* los datos del archivo solicitado

```
    /**
     * Devuelve la duración del archivo
     */
    public Time getDuration() {
        return Duration.DURATION_UNKNOWN;
    }
```

```
    /**
     * Devuelve los controles soportados
     */
    public Object [] getControls() {
        return new Object[0];
    }

    /**
     * Devuelve el control que soporta la interfaz especificada
     */
    public Object getControl(String tipoControl) {
        return null;
    }

    /**
     * Muestra en el área de archivos del cliente los errores
     */
    private void muestraError(String texto) {
        padre.textoBarraEstado("Conectado a:    " + host);
        if (!padre.esVisibleAreaMensajes()) {
            padre.muestraAreaMensajes();
        }
        padre.textoAreaMensajes(texto);
    }
}
```

Finalmente, reseñamos que a los métodos implementados, heredados de la clase *DataSource*, se añade el método *muestraError*; este método abre el área de mensajes del cliente que creó el *datasource* y muestra en ese área los mensajes de error que se pueden dar durante el proceso de descarga y reproducción del archivo. Se ha optado por esta solución, en lugar de la típica de mostrar diálogos de mensaje, debido a que éstos últimos interfieren de forma negativa en la correcta descarga y reproducción de los archivos.

Veamos a continuación el archivo en el que se implementa el *stream* empleado por este *datasource*:

- **TCPSourceStream.java.**

```
/*
 * @(#) TCPSourceStream.java
 *
 * Versión 1    22/07/99
 * Copyright 1999    Jacinto Navarro González
 *
 */

// Paquete al que pertenece la clase
package Cliente;

// Clases y paquetes importados
import com.sun.media.*;
import com.sun.media.util.*;
import com.sun.media.content.*;
import java.io.*;
import javax.media.protocol.*;

/**
 * Esta clase crea un stream no posicionable que lee los datos del
```

```
 * socket TCP conectado al servidor directamente para ser procesados
 */
public class TCPSourceStream implements PullSourceStream {
    // Stream de entrada de datos
    private InputStream entradaDatos;
    // Descriptor del contenido del archivo
    private ContentDescriptor descriptor;
    // Archivo que lee el stream
    private String archivo;
    // Host del que procede el archivo
    private String host;
    // Cliente que crea el reproductor
    private ClienteAV padre;
    // Longitud del archivo leido
    private long longitudArchivo;
    // Indica si el archivo ha sido ya cargado en memoria
    private boolean iniciado = false;
    // Indica si se ha alcanzado el final del stream
    private boolean fin;

    public TCPSourceStream(ClienteAV padre, String archivo, String host,
                           long longitudArchivo, String tipo, InputStream
                           entrada) {
        this.padre = padre;
        this.host = host;
        this.archivo = archivo;
        this.entradaDatos = entrada;
        this.longitudArchivo = longitudArchivo;
        descriptor = new ContentDescriptor(tipo);
        fin = false;
    }

    /**
     * Devuelve el descriptor de los datos del archivo
     */
    public ContentDescriptor getContentDescriptor() {
        return descriptor;
    }

    /**
     * Inicia el stream
     */
    public void start() {
        iniciado = true;
    }

    /**
     * Cierra el stream de lectura de datos
     */
    public void close() {
        String textoError;

        try {
            entradaDatos.close();
        } catch (IOException e) {
            textoError = " Error al cerrar el proceso receptor del archivo '"
                    + archivo + "'. Protocolo TCP stream.";
            muestraError(textoError);
        }
    }

    /**
     * Determina si se ha alcanzado el final del stream
     */
    public boolean endOfStream() {
        return fin;
    }
```

```
    /**
     * Devuelve la cantidad de datos disponibles en el stream
     */
    public int available() {
        int cantidad = - 1;
        String textoError;

        try {
            cantidad = entradaDatos.available();
        } catch (IOException e) {
            textoError = " Error en lectura al reproducir el archivo '"
                    + archivo + "'. Protocolo TCP stream.";
            muestraError(textoError);
        }

        return cantidad;
    }

    /**
     * El manejador que procesa los datos lee del stream directamente
     */
    public int read(byte[] buffer, int offset, int length) throws IOException{
        int cantidad;

        cantidad = entradaDatos.read(buffer, offset, length);
        if (cantidad == -1) {
            fin = true;
        }

        return cantidad;
    }
```

El método *read(...)* es el método principal de la clase implementada, como puede observarse, lee directamente del *stream* de datos, el contenido del fichero, conforme le son solicitados los datos. De este modo, queda claro que sólo se permite una lectura secuencial de los datos, sin posibilidad de dar saltos a datos posteriores o anteriores.

```
    /**
     * Determina si se ha producido un bloqueo en la lectura de datos
     */
    public boolean willReadBlock() {
        boolean result = false;
        String textoError;

        if (fin) {
            result = true;
        } else {
            try {
                result = (entradaDatos.available() == 0);
            } catch (IOException e) {
                textoError = " Error en lectura al reproducir el archivo '"
                        + archivo + "'. Protocolo TCP stream.";
                muestraError(textoError);
            }
        }

        return result;
    }

    /**
     * Devuelve la longitud del archivo
     */
    public long getContentLength() {
        return longitudArchivo;
    }
```

```
    /**
     * Devuelve los controles soportados
     */
    public Object [] getControls() {
        return new Object[0];
    }

    /**
     * Devuelve el control que soporta la interfaz especificada
     */
    public Object getControl(String tipoControl) {
        return null;
    }

    /**
     * Muestra en el área de archivos del cliente los errores
     */
    private void muestraError(String texto) {
        padre.textoBarraEstado("Conectado a:   " + host);
        if (!padre.esVisibleAreaMensajes()) {
            padre.muestraAreaMensajes();
        }
        padre.textoAreaMensajes(texto);
    }
}
```

Para el análisis de la clase que implementa un *DataSource* con buffer, puesto que gran parte de su estructura es idéntica a la analizada en la clase anterior comentaremos sólo el código que distinga una clase de otra. Los métodos heredados de la clase *DataSource* que no aparezcan, es porque son idénticos a los de *TCPDataSourceStream*.

- **TCPDataSourceMemory.java.**

```
/*
 * @(#) TCPDataSourceMemory.java
 *
 * Versión 1    22/04/00
 * Copyright 2000    Jacinto Navarro González
 *
 */

// Paquete al que pertenece la clase
package Cliente;

// Clases y paquetes importados
import com.sun.media.*;
import com.sun.media.util.*;
import com.sun.media.content.*;
import javax.media.*;
import javax.media.protocol.*;
import java.io.*;
import java.net.*;
import java.awt.*;

/**
 * Esta clase crea un DataSource posicionable que lee los datos de un
 * buffer una vez son cargados en él mediante TCP, los datos del
 * stream de entrada
 */
public class TCPDataSourceMemory extends PullDataSource implements
        Positionable {
...
```

La primera diferencia entre *TCPDataSourceStream* y *TCPDataSourceMemory*, se observa en la cabecera de la segunda clase. *TCPDataSourceMemory* implementa la interfaz *Positionable* de JMF. De este modo permite realizar operaciones sobre los datos que no permite la clase que lee directamente del *stream*.

```
...
    /**
     * A partir de la extensión del archivo, obtiene el tipo
     * entendible por JMF
     */
    public String getContentType() {
        int posicion;
        String extension;
        String tipo;

        String locatorString = getLocator().toExternalForm();
        posicion = locatorString.lastIndexOf(".");
        extension = locatorString.substring(posicion + 1);
        tipo = "unknown";

        // Soporta la reproducción de todos los tipos de archivos
        if (extension.equals("avi")) {
            tipo = "video.x_msvideo";
        } else if ((extension.equals("mpg")) || (extension.equals("mpeg"))) {
            tipo = "video.mpeg";
        } else if (extension.equals("mov")) {
            tipo = "video.quicktime";
        } else if (extension.equals("wav")) {
            tipo = "audio.x_wav";
        } else if (extension.equals("au")) {
            tipo = "audio.basic";
        } else if ((extension.equals("mid")) || (extension.equals("midi")) ||
                   (extension.equals("rmi"))) {
            tipo = "audio.midi";
        } else if (extension.equals("rmf")) {
            tipo = "audio.rmf";
        } else if ((extension.equals("mp2")) || (extension.equals("mp3"))) {
            tipo = "audio.mpeg";
        } else if (extension.equals("viv")) {
            tipo = "video.vivo";
        } else if (extension.equals("gsm")) {
            tipo = "audio.x_gsm";
        }

        this.tipoArchivo = tipo;

        return tipo;
    }
```

El método *getContentType( )* de la clase *TCPDataSourceMemory*, sí devuelve el tipo de los tipos de archivo posicionables (AVI, WAV, MOV, MP3,etc.). Por tanto, esta clase es mucho más versátil que la anterior en cuanto la cantidad de tipos de archivo que maneja.

```
    /**
     * Obtiene la colección de streams para este datasource
     */
    public PullSourceStream[] getStreams() {
        String textoError;

        streams = new PullSourceStream[1];
        try {
            // Crea un stream posicionable TCP
```

```
            sourceStream = new TCPSourceMemory(padre, host, archivo,
                    longitudArchivo, tipoArchivo, socket.getInputStream());
            streams[0] = sourceStream;
        } catch (IOException e) {
            textoError = " Error al crear el proceso receptor del archivo '"
                    + archivo + "'. Protocolo TCP memory.";
            muestraError(textoError);
        }

        return streams;
    }
...
```

El método *getStreams( )* de esta clase, instancia un objeto de la clase *TCPSourceMemory*, es decir, un *stream* que lee del *stream* de datos que llega por el *socket*, pero que almacena estos datos en un buffer, para que el *datasource* lea de este buffer y pueda por tanto reposicionar la lectura, y por tanto, la presentación de los datos multimedia.

```
...
    /**
     * Establece la nueva posición en el stream del datasource
     */
    public Time setPosition(Time where, int rounding) {
        sourceStream.seek(0);
        start();
        return new Time(0);
    }

    /**
     * Determina si se puede cambiar de posición en el stream
     */
    public boolean isRandomAccess() {
        return false;
    }
```

Los dos métodos anteriores implementan la interfaz *Positionable*, y permiten que el archivo en reproducción pueda ser reposicionado.

Vemos a continuación el código que distingue la clase *TCPSourceMemory* de la anteriormente comentada *TCPSourceStream*:

- **TCPSourceMemory.java.**

```
/*
 * @(#) TCPSourceMemory.java
 *
 * Versión 1    22/04/00
 * Copyright 2000    Jacinto Navarro González
 *
 */

// Paquete al que pertenece la clase
package Cliente;

// Clases y paquetes importados
import com.sun.media.*;
import com.sun.media.util.*;
import com.sun.media.content.*;
import java.io.*;
import javax.media.protocol.*;
```

```
/**
 * Esta clase crea un stream posicionable que lee los datos del
 * socket TCP conectado al servidor y los carga en memoria para que
 * el datasource lea de ellos y pueda reposicionar durante la
 * reproducción
 */
public class TCPSourceMemory implements PullSourceStream, Seekable {
    // Stream de entrada de datos
    private InputStream entradaDatos;
    // Descriptor del contenido del archivo
    private ContentDescriptor descriptor;
    // Archivo que lee el stream
    private String archivo;
    // Host del que procede el archivo
    private String host;
    // Cliente que crea el reproductor
    private ClienteAV padre;
    // Longitud del archivo leido
    private long longitudArchivo;
    // Índice de lectura de datos
    private long indice;
    // Almacena los datos multimedia
    private byte [] datos = null;
    // Indica si el stream ha sido iniciado
    private boolean iniciado = false;
    // Indica si el archivo ha sido ya cargado en memoria
    private boolean preparado = false;
    // Indica si se ha alcanzado el final del stream
    private boolean fin;

    public TCPSourceMemory(ClienteAV padre, String host, String archivo,
            long longitudArchivo, String tipo, InputStream entrada) {

        this.padre = padre;
        this.host = host;
        this.archivo = archivo;
        this.entradaDatos = entrada;
        this.longitudArchivo = longitudArchivo;
        descriptor = new ContentDescriptor(tipo);
        fin = false;
    }
...
```

Como puede observarse, esta clase, además de implementar la interfaz *PullSourceStream*, implementa la interfaz *Seekable*, cuyos métodos veremos más adelante.

```
...
    /**
     * Determina si se ha alcanzado el final del stream
     */
    public boolean endOfStream() {
        if (indice >= longitudArchivo) {
            fin = true;
        } else {
            fin = false;
        }
        return fin;
    }
...
```

Para determinar si se ha alcanzado el fin del *stream* se mira si el puntero de lectura de éste supera en valor a la longitud del archivo que se está procesando.

```
...
    /**
     * Carga los datos del archivo en memoria
     */
    public void cargaDatos() {
        long longitud;
        int leido = 0;
        int restante;
        int cantidad = 0;
        String textoError;

        longitud = longitudArchivo;

        try {
            // Crea un buffer de longitud la del archivo
            datos = new byte[(int) longitud];
        } catch (Exception e) {
            textoError = " No hay suficiente memoria para reproducir el
                    archivo '" + archivo + "'. Protocolo TCP memory";
            muestraError(textoError);
        }
        // Mientras no se han leido todos los datos del archivo
        while (leido < longitud) {
            restante = (int) longitud - leido;
            try {
                // Lee datos del stream
                cantidad = entradaDatos.read(datos, leido, restante);
            } catch (IOException e) {
                textoError = " Error en lectura al reproducir el archivo '"
                        + archivo + "'. Protocolo TCP memory.";
                muestraError(textoError);
            }

            leido += cantidad;
        }
    }
```

El método *cargaDatos( )* va leyendo los datos del *stream* y los mete en un buffer, y será de ahí de donde se leeran y procesarán.

```
    /**
     * El manejador que procesa los datos lee de memoria
     */
    public int read(byte[] buffer, int offset, int length) throws IOException{
        long bytes;

        // Mientras el archivo no ha sido cargado en memoria
        // bloquea la lectura de datos
        if (!preparado) {
            cargaDatos();
            preparado = true;
        }

        if (endOfStream()) {
            return - 1;
        }

        bytes = longitudArchivo - indice;

        if (length < bytes) {
            bytes = length;
        }

        if (buffer != null) {
            // Copia en el buffer de lectura los datos solicitados
```

```
            System.arraycopy((Object) datos, (int) indice, (Object) buffer,
                    offset, (int) bytes);
        }

        indice += bytes;

        return (int) bytes;
    }
...
```

El método *read(...)* que implementa esta clase, presenta importantes diferencias respecto al de la clase *TCPSourceStream* que leía directamente de los datos que entraban por el *socket*. En este caso, como puede observarse, el método *read(...)* queda bloqueado hasta que no termina de ejecutarse la carga en memoria de los datos del archivo. Una vez se ha cargado el archivo en memoria, continua la ejecución del método y comienza a leer datos directamente de memoria, utilizando el método *arraycopy* de la clase *System* de Java, que como su nombre indica copia datos de una vector en otro vector.

```
...
    /**
     * Situa el puntero de lectura donde se le indica
     */
    public long seek(long donde) {
        if (donde < longitudArchivo) {
            indice = donde;
        } else {
            indice = longitudArchivo - 1;
        }
        if (donde < 0) {
            indice = 0;
        }
        return indice;
    }
```

El método *seek* implementa un método de la interfaz *Seekable*. Simplemente posiciona el puntero de lectura de datos donde indica el parámetro que se le pasa.

```
    /**
     * Devuelve a donde apunta el puntero de los datos
     */
    public long tell() {
        return indice;
    }
```

El método *tell* también implementa un método de la interfaz *Seekable*. Devuelve el valor del puntero de lectura de datos.

```
    /**
     * Determina si se puede cambiar de posición en el stream
     */
    public boolean isRandomAccess() {
        return true;
    }
```

El método *isRandomAccess( )* indica que efectivamente es posible cambiar de posición en el *stream*.

```
...
}
```

El tercer *datasource* implementado emplea el protocolo de transporte UDP para establecer la conexión con el servidor y descargar el archivo a través de la red. Puesto que las dos clases implementadas (*UDPDataSourceMemory* y *UDPSourceMemory*) son muy similares en su estructura a las dos últimas analizadas, no parece necesario incluir su código, sin embargo, sí resulta interesante comentar las diferencias relevantes. La filosofía de trabajo es la misma, el *UDPDataSourceMemory* implementa la interfaz *Positionable*, mientras que *UDPSourceMemory* implementa la interfaz *Seekable*. El archivo se recibe mediante datagramas enviados por el servidor, de los cuales se extraen los datos, que se cargan en un buffer. Se utiliza igualmente en el *stream* un puntero de lectura que determina el byte que se lee. De ese modo, el procesamiento de los datos se realiza leyendo de ese buffer, permitiendo de ese modo el reposicionamiento en los datos que se procesan. En resumen, la estructura es idéntica, sólo que a la hora de leer los datos del archivo que vienen por la red, en el caso TCP se realiza leyendo del *stream* que envuelve al *socket* TCP, mientras que en el caso UDP se realiza leyendo los datagramas que llegan al *socket* UDP, enviados por el servidor.

Hemos analizado ya como se obtienen los datos multimedia que se encuentran en el servidor, desde el primer paso, que es obtener información sobre ellos accediendo a la base de datos del servidor, hasta el último (hasta ahora), que es recoger los datos que vienen por la red y dárselos a los manejadores apropiados, de forma correcta. Pero queda un paso más que dar, y es establecer una interfaz que permita al usuario interactuar con el reproductor que se crea. Este trabajo se implementa en la clase *JMFPlayer* cuyo código comentamos a continuación:

- **JMFPlayer.java.**

```
/*
 * @(#) TCPDataSourceStream.java
 *
 * Versión 1    22/07/99
 * Copyright 1999    Jacinto Navarro González
 *
 */

// Paquete al que pertenece la clase
package Cliente;

// Clases y paquetes importados
import javax.media.*;
import com.sun.media.*;
import com.sun.media.util.*;
import com.sun.media.controls.*;
import com.sun.media.content.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
import java.util.*;

/**
 * Esta clase crea un reproductor multimedia JMF para el archivo
 * indicado, con una interfaz gráfica para el control de la
 * reproducción del archivo
```

```
 */
public class JMFPlayer extends Frame implements ControllerListener {
    // Manejador para el archivo
    private Player player = null;
    // Componente de control del reproductor
    private Component controlComp = null;
    // Componente visual del reproductor
    private Component visualComp = null;
    // Barra de progreso para la descarga
    private Component progressBar = null;
    // Cliente que crea el reproductor
    private ClienteAV padre = null;
    // Panel donde se añaden los componentes
    private Panel framePanel;
    // Bordes del contenedor de componentes
    private Insets insets;
    // Menu del reproductor
    private Menu menu;
    // Barra de menú del reproductor
    private MenuBar menubar;
    // Popup menú para permitir cambiar tamaño de los videos
    private PopupMenu zoomMenu = null;
    // Determina si el archivo de procesa en bucle
    private CheckboxMenuItem cbAutoRepeticion = null;
    // Indica si la ventana del reproductor ha sido creada
    private boolean ventanaCreada = false;
    // Indica si hay un nuevo video
    private boolean nuevoVideo = false;
    // Indica si el panel contenedor se ha redibujado
    private boolean panelRedibujado = false;
    // Indica si el reproductor está situado en su ubicación
    private boolean situado = false;
    // Indica si el reproductor está cerrado
    private boolean cerrado = false;
    // Archivo que se reproduce
    private String archivo = new String();
    // Host que envía el archivo
    private String host = new String();
    // Mensaje de error en el proceso
    private String textoError = new String();
    // Número de orden entre todos los reporductores abiertos
    private int orden;
    // Icono del reproductor
    private Image imagen;
```

Como puede observarse, la clase *JMFPlayer* implementa la interfaz *ControllerListener* de JMF, por lo tanto, los reproductores creados se registran como escuchadores de los eventos que se produzcan en el procesamiento de los datos multimedia.

```
    public JMFPlayer(ClienteAV padre, Player player, String host, String
                     archivo) {
        super(archivo + " - [" + host + "]");

        this.player = player;
        this.padre = padre;
        this.host = host;
        this.archivo = archivo;

        menubar = creaBarraMenu();
        setMenuBar(menubar);
        setLayout(new BorderLayout());

        framePanel = new Panel();
        framePanel.setLayout(null);
        add(framePanel, "Center");
```

```
        insets = getInsets();
        // Establece las medidas del frame
        setSize(insets.left + insets.right + 320, insets.top + insets.bottom +
                60);

        Point punto = new Point();
        // Inicialmente centra el reproductor en la pantalla
        Toolkit toolkit = this.getToolkit();
        Dimension dimPantalla = toolkit.getScreenSize();
        punto.x = (dimPantalla.width - 320) / 2;
        punto.y = (dimPantalla.height - 60) / 2;
        setBounds(punto.x, punto.y, 320, 60);

        // Establece el icono del reproductor
        imagen = Util.devuelveImagen(this, Util.PLAYER);
        setIconImage(imagen);

        setVisible(true);

        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                cierraPlayer();
                eliminaPlayer();
                while (JMFPlayer.this.player != null) {
                    try {
                        // Retardo para dejar concluir la operacion
                        Thread.currentThread().sleep(100);
                    } catch (InterruptedException e) {
                        // Ignora excepción
                    }
                }

                synchronized (JMFPlayer.this) {
                    dispose();
                }
            }
        });

        framePanel.addComponentListener(new ComponentAdapter() {
            public void componentResized(ComponentEvent ce) {
                // Se han modificado las dimensiones del panel
                panelRedibujado = true;
                redibuja();
            }
        });

        addComponentListener( new ComponentAdapter() {
            public void componentResized(ComponentEvent ce) {
                Dimension dimension;

                insets = getInsets();
                dimension = getSize();
                // Redibuja el panel contenedor de componentes
                framePanel.setSize(dimension.width - insets.left -
                        insets.right, dimension.height - insets.top -
                        insets.bottom);
            }
        });

        if (player != null) {
            player.addControllerListener( this );
            // Pone al player en estado realizing
            player.realize();
        }
    }
```

Cuando finaliza el constructor de la clase, se invoca al método *realize( )* para que el manejador de los datos multimedia pase del estado *unrealized* al estado *realizing*, de este modo, el manejador empieza a determinar qué recursos va a necesitar para llevar a cabo su trabajo. También en este punto, comienza a cargar los datos del archivo multimedia que proceden del servidor a través de la red.

```
/**
 * Asigna al reproductor su índice entre todos los abiertos
 * por el cliente
 */
public void fijaOrden(int orden) {
    Point punto = this.getLocation();

    this.orden = orden;

    if (!situado) {
        // Cada reproductor que se abre se situa en un punto
        // para simular una distribución en cascada

        this.setLocation(punto.x + (15 * (orden)), punto.y + (15 *
                         (orden)));
        situado = true;
    }
}

/**
 * Avisa al cliente de que este reproductor está siendo cerrado
 */
public void eliminaPlayer() {
    padre.eliminaPlayer(orden);
}
```

Los dos métodos anteriores sirven para que el cliente que ha creado este reproductor, tenga un control de lo que le sucede a éste. Así pues, el método *fijaOrden(...)* permite que el cliente asigne al reproductor un índice, mediante el cual poder acceder a él desde el vector de reproductores abiertos que vimos en el código principal del cliente. También actua haciendo que los reproductores que se van abriendo desde el cliente lo hagan en cascada, y no uno encima de otro. El método *eliminaPlayer( )* notifica al cliente, que el reproductor con el índice que se le pasa va a ser cerrado de inmediato.

```
/**
 * Duerme por un número especificado de milisegundos
 */
private void duerme(long time) {
    try {
        Thread.currentThread().sleep(time);
    } catch (Exception e) {
        // Ignora excepción
    }
}

/**
 * Crea la barra de menú
 */
private MenuBar creaBarraMenu() {
    ActionListener al = new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
        String accion = ae.getActionCommand();

        if (accion.equals("Salir")) {
```

```
                cierraPlayer();
                eliminaPlayer();
                synchronized (JMFPlayer.this) {
                    dispose();
                }
            }
            }
        };

        MenuItem item;
        MenuBar mb = new MenuBar();

        Menu menuAccion = new Menu("Acción");

        menuAccion.add(item = new MenuItem("Salir"));
        item.addActionListener(al);

        Menu menuOpciones = new Menu("Opciones");
        cbAutoRepeticion = new CheckboxMenuItem("Auto repetición");
        cbAutoRepeticion.setState(true);
        menuOpciones.add(cbAutoRepeticion);

        this.menu = menuAccion;
        mb.add(menuAccion);
        mb.add(menuOpciones);
        return mb;
    }

    /**
     * Hace que el reproductor se pueda mostrar
     */
    public void addNotify() {
        super.addNotify();
        ventanaCreada = true;
        insets = getInsets();
        validate();
    }

    /**
     * Redibuja los componentes que hay en el panel de control
     */
    public void redibuja() {
        Dimension dimension;
        int medidasVideo;

        dimension = framePanel.getSize();
        medidasVideo = dimension.height;

        if (controlComp != null) {
            medidasVideo -= controlComp.getPreferredSize().height;
            if (visualComp != null) {
                if (medidasVideo < 2)
                    medidasVideo = 2;
            }
            if (dimension.width < 80) {
                dimension.width = 80;
            }
            controlComp.setBounds(0, medidasVideo, dimension.width,
                                  controlComp.getPreferredSize().height);
            controlComp.invalidate();
        }

        if (visualComp != null) {
            visualComp.setBounds(0, 0, dimension.width, medidasVideo);
        }
        framePanel.validate();
    }
```

```
/**
 * Cierra el manejador
 */
public synchronized void cierraPlayer() {
    if (player != null) {
        player.close();
        cerrado = true;
        player = null;
        progressBar = null;
    }
}
```

El método *cierraPlayer( )* provoca, al invocar el método *close( )* del *player* que todos los recursos que envuelven a un objeto de este tipo se cierren. Así pues, se produce que el *datasource* de este manejador ejecute también su método *close( )*, y lo mismo ocurre con el *stream* del que lee el manejador, es cerrado igualmente.

```
/**
 * Maneja los eventos del player
 */
public synchronized void controllerUpdate(ControllerEvent ce) {
    int ancho = 320;
    int alto = 0;
    int prefAlto;
    int espera = 0;
    Dimension medidasVideo;
    Dimension prefMedidas;
    Time tiempo;

    // Si el player ha alcanzado el estado realize
    if (ce instanceof RealizeCompleteEvent) {

         insets = getInsets();

         if (progressBar != null) {
             // Elimina la posible barra de progreso
             framePanel.remove(progressBar);
         }
         // Puede obtener ya el componente visual
         if ((visualComp = player.getVisualComponent()) != null) {
               ancho = visualComp.getPreferredSize().width;
               alto = visualComp.getPreferredSize().height;
               framePanel.add(visualComp);
               visualComp.setBounds(0, 0, ancho, alto);
               // Añade el popup menú
               addPopupMenu(visualComp);
               nuevoVideo = true;
         }
         // Puede obtener ya el componente de control
         if ((controlComp = player.getControlPanelComponent()) != null) {
               prefAlto = controlComp.getPreferredSize().height;
               framePanel.add(controlComp);
               controlComp.setBounds(0, alto, ancho, prefAlto);
               alto += prefAlto;
               nuevoVideo = true;
         }
         // Establece el título del reproductor
         setTitle(archivo + " - [" + host + "]");

         // Pone al player en estado prefetching
         player.prefetch();
```

Cuando el manejador recibe un evento del tipo *RealizeCompleteEvent* significa que ha alcanzado ya el estado *realice*. Ya conoce que tipo de datos va a presentar, y por

ello ya puede adquirir los componentes de control y visual (en el caso de los archivos de imagen) y situarlos en pantalla, como puede observarse en el código. En la última instrucción del bloque el manejador invoca el método *prefetch( )*, que provoca que se pase al estado *prefetching*, durante el cual el manejador se prepara para presentar sus datos multimedia, pre-cargando estos datos y obteniendo recursos exclusivos para su uso.

```
// Si el player ha alcanzado el estado prefetch
} else if (ce instanceof PrefetchCompleteEvent) {
      if (nuevoVideo) {
            if (visualComp != null) {
            medidasVideo = visualComp.getPreferredSize();
            if (controlComp != null) {
                  medidasVideo.height +=
                          controlComp.getPreferredSize().height;
            }
            panelRedibujado = false;
            setSize(medidasVideo.width + insets.left + insets.right,
                    medidasVideo.height + insets.top + insets.bottom);

            while (panelRedibujado == false && espera < 2000) {
                  try {
                      espera += 50;
                      Thread.currentThread().sleep(50);
                      Thread.currentThread().yield();
                  } catch (InterruptedException ie) {}
                      // Ignora Excepción
                  }
            } else {
                  alto = 1;
                  if (controlComp != null) {
                        alto = controlComp.getPreferredSize().height;
                  }
                  setSize(ancho + insets.left + insets.right,
                        alto + insets.top + insets.bottom);
            }
            nuevoVideo = false;
      }

      if (player.getTargetState() != Controller.Started)
          // Pone en marcha la reproducción
          player.start();
```

Cuando el evento que se recibe es *PrefetchCompleteEvent* significa que el manejador ha alcanzado el estado *prefetched*, con lo cual ya está totalmente preparado para comenzar a reproducir datos. De ahí que se invoque el método *start( )*, con el que inicia su reproducción de datos multimedia.

```
// Si se alcanza el final vuelve al principio si
// está habilitado
} else if (ce instanceof EndOfMediaEvent) {
      if (cbAutoRepeticion.getState()) {
            player.setMediaTime(new Time(0));
            // Vuelve a ponerlo en prefetching
            player.prefetch();
      }
// Si ha habido algún error en el procesamiento
} else if (ce instanceof ControllerErrorEvent) {
      if (!cerrado) {
            textoError = " Error al reproducir archivo '" + archivo +
                    "', formato de datos no soportado.";
            muestraError(textoError);
```

```
                cierraPlayer();
                if (visualComp != null) {
                      visualComp.remove(zoomMenu);
                }
                framePanel.removeAll();
                visualComp = null;
                controlComp = null;
                setSize(320, 45);
                player.removeControllerListener( this );
                player.close();
                player = null;
                progressBar = null;
          }
    // El reproductor es cerrado
    } else if (ce instanceof ControllerClosedEvent) {
          if (visualComp != null) {
                visualComp.remove(zoomMenu);
                visualComp = null;
          }
          if (controlComp != null) {
                controlComp = null;
          }
          framePanel.removeAll();

          System.gc();
          System.runFinalization();
          setSize(320, 60);
          player = null;
          progressBar = null;
    // Si se modifica la duración
    } else if (ce instanceof DurationUpdateEvent) {
          tiempo = ((DurationUpdateEvent) ce).getDuration();
    // Durante la descarga muestra la barra de progreso
    } else if (ce instanceof CachingControlEvent) {
          CachingControl cc =
                ((CachingControlEvent)ce).getCachingControl();
          if (cc != null && progressBar == null) {
                progressBar = cc.getControlComponent();
                if (progressBar == null) {
                      progressBar = cc.getProgressBarComponent();
                }
                if (progressBar != null) {
                framePanel.add(progressBar);
                prefMedidas = progressBar.getPreferredSize();
                progressBar.setBounds(0, 0, prefMedidas.width,
                                  prefMedidas.height);
                insets = getInsets();
                framePanel.setSize(prefMedidas.width, prefMedidas.height);
                setSize(insets.left + insets.right + prefMedidas.width,
                        insets.top + insets.bottom + prefMedidas.height);
                }
          }
    } else if (ce instanceof StartEvent) {
    } else if (ce instanceof MediaTimeSetEvent) {
    } else if (ce instanceof TransitionEvent) {
    } else if (ce instanceof RateChangeEvent) {
    } else if (ce instanceof StopTimeChangeEvent) {
    }
}

/**
 * Abre el área de mensajes del cliente y muestra ahí el error
 */
private void muestraError(String texto) {
    padre.textoBarraEstado("Conectado a:   " + host);
    if (!padre.esVisibleAreaMensajes()) {
        padre.muestraAreaMensajes();
```

```
        }
        padre.textoAreaMensajes(texto);
    }

    /**
     * Aplica el zoom elegido por el usuario al video activo
     */
    private void aplicaZoom(float valorZoom) {
        Dimension dimension;

        insets = getInsets();

        if (visualComp != null) {
            dimension = visualComp.getPreferredSize();
            dimension.width = (int) (dimension.width * valorZoom);
            dimension.height = (int) (dimension.height * valorZoom);

            if (controlComp != null) {
                dimension.height += controlComp.getPreferredSize().height;
            }

            setSize(dimension.width + insets.left + insets.right,
                    dimension.height + insets.top + insets.bottom);
        }
    }


    /**
     * Añade menú emergente para los componentes visuales
     */
    private void addPopupMenu(Component visual) {
        MenuItem item;
        ActionListener zoomSeleccion;

        zoomMenu = new PopupMenu("Zoom");

        zoomSeleccion = new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            String accion = ae.getActionCommand();

            if (accion.indexOf("1:2") >= 0) {
                aplicaZoom(0.5f);
            } else if (accion.indexOf("1:1") >= 0) {
                aplicaZoom(1.0f);
            } else if (accion.indexOf("2:1") >= 0) {
                aplicaZoom(2.0f);
            } else if (accion.indexOf("3:1") >= 0) {
                aplicaZoom(3.0f);
            } else if (accion.indexOf("4:1") >= 0) {
                aplicaZoom(4.0f);
            }
        }
        };

        visual.add(zoomMenu);
        item = new MenuItem("Escala 1:2");
        zoomMenu.add(item);
        item.addActionListener(zoomSeleccion);
        item = new MenuItem("Escala 1:1");
        zoomMenu.add(item);
        item.addActionListener(zoomSeleccion);
        item = new MenuItem("Escala 2:1");
        zoomMenu.add(item);
        item.addActionListener(zoomSeleccion);
        item = new MenuItem("Escala 3:1");
        zoomMenu.add(item);
        item.addActionListener(zoomSeleccion);
        item = new MenuItem("Escala 4:1");
```

```
        zoomMenu.add(item);
        item.addActionListener(zoomSeleccion);

        visual.addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent me) {
                if (me.isPopupTrigger()) {
                    zoomMenu.show(visualComp, me.getX(), me.getY());
                 }
            }

            public void mouseReleased(MouseEvent me) {
                if (me.isPopupTrigger()) {
                    zoomMenu.show(visualComp, me.getX(), me.getY());
                }
            }

            public void mouseClicked(MouseEvent me) {
                if (me.isPopupTrigger()) {
                    zoomMenu.show(visualComp, me.getX(), me.getY());
                }
            }
        });
    }
}
```

Con la clase *JMFPlayer* finaliza el análisis de las clases significativas que forman parte de la implementación del programa cliente de la aplicación. El resto de clases, no vistas aquí, implementan aspectos relativos a la interfaz de usuario, por lo tanto, en principio no tienen tanta relevancia como las vistas, teniendo en cuenta la orientación que tiene este proyecto.

Finalizamos el repaso de los ficheros relativos al programa cliente, mostrando los parámetros que incluye su fichero de configuración, como hacíamos anteriormente con el del servidor.

- **ClienteAV.ini.**

```
#Configuración
#Wed Jun 14 23:13:17 GMT+02:00 2000
puertormi=1099
recientes=2
servidor1=Localhost
servidor0=127.0.0.1
puertomediaUDP=2001
puertomediaTCP=2000
bloquerecepcion=8192
retardocreacionUDP=500
```

5. Uso de la aplicación.

El presente proyecto ha sido implementado utilizando el JDK 1.2, mientras que la versión de la librería multimedia de Java que se ha utilizado es la JMF 2.1. Para ejecutar la aplicación será necesario pues, disponer de estas herramientas, o bien, de las librerías con las clases necesarias para crearse un *runtime* adecuado. Ambas herramientas son de libre distribución, por lo que conseguirlas no conlleva muchas dificultades.

La aplicación desarrollada ha construido todas sus clases en dos paquetes distintos, el paquete cliente y el servidor, así pues, habrá que tenerlo en cuenta por ejemplo, a la hora de la ejecución, para establecer correctamente la variable classpath, pues dicha variable deberá contener una entrada que apunte al directorio padre del directorio donde se instalen las clases. Además, el directorio que contenga las clases del cliente deberá llamarse cliente, y el que contenga las clases del servidor, se habrá de llamar servidor.

La aplicación se entrega en un fichero comprimido llamado 'proyecto.zip'. Dicho fichero contiene todos los archivos que forman parte de la aplicación (fuentes y compilados). Al descomprimir el fichero, se genera un directorio llamado proyecto, que contiene dos subdirectorios, uno se llama cliente y el otro servidor. En el directorio cliente se encuentran todos los archivos referentes al cliente, además de un subdirectorio llamado gifs, donde se almacenan las imágenes que utiliza el programa cliente en su interfaz, mientras que en el directorio servidor se encuentran todos los archivos referentes al servidor, además de un subdirectorio llamado bdatos, donde se encuentra situada la base de datos del servidor, y los archivos multimedia que forman parte de esa base de datos. Una vez se tiene instalada la aplicación, es necesario añadir una entrada a la variable de entorno classpath, que apunte al directorio llamado proyecto, es decir, el directorio padre del cliente y el servidor. En caso de no tener instalada alguna de las herramientas señaladas al comienzo de este apartado, será necesario por lo menos, disponer de las librerías de JDK 1.2 y de JMF 2.1 que son utilizadas por la aplicación. Junto con el software del proyecto se entregan esas librerías, para que hagan las veces de *runtime* para la aplicación. Habrá que copiarlas a la máquina en cuestión y añadir una entrada más a la variable classpath que apunte al directorio donde hayan sido ubicadas. Lo anteriormente expuesto es válido, tanto para la instalación de la aplicación en una sola máquina, como para la instalación de la aplicación en red, es decir, el cliente en una máquina, y el servidor en otra.

Una vez se encuentra todo el software correctamente instalado, podemos proceder a ejecutar la aplicación. Comenzando por el servidor, lo primero que hay que hacer es lanzar el registro de objetos remotos (rmiregistry) con el siguiente comando:

- start rmiregistry

En caso de que deseemos lanzar el registro de objetos remotos para que escuche en un puerto distinto del que tiene asignado por defecto (1099), ejecutamos el siguiente comando:

- start rmiregistry numero_puerto

Si se cambia el puerto en que escucha el registro, habrá que modificar en el fichero de configuración del cliente (‘ClienteAV.ini’) el valor del puerto RMI con el nuevo valor utilizado, para que el programa al iniciar su ejecución cargue el puerto RMI correcto. Una vez está lanzado el registro de objetos remotos, ejecutamos el programa servidor. Habrá que utilizar el siguiente comando:

- java servidor.ServidorAV

En el directorio del servidor se incluye aun archivo de proceso por lotes, llamado ‘ServidorAV.bat’ que ejecuta la orden anterior, pero que además carga las políticas de seguridad de un fichero también situado en el directorio servidor, llamado ‘ServidorAV.policy’, que asigna todos los permisos al programa servidor en su ejecución. Esto puede venir bien en caso de que, por el tipo de políticas de seguridad que haya configuradas en la máquina utilizada, el programa lance excepciones de seguridad cada vez que se ejecute, y no se pueda comprobar por lo tanto su funcionamiento.

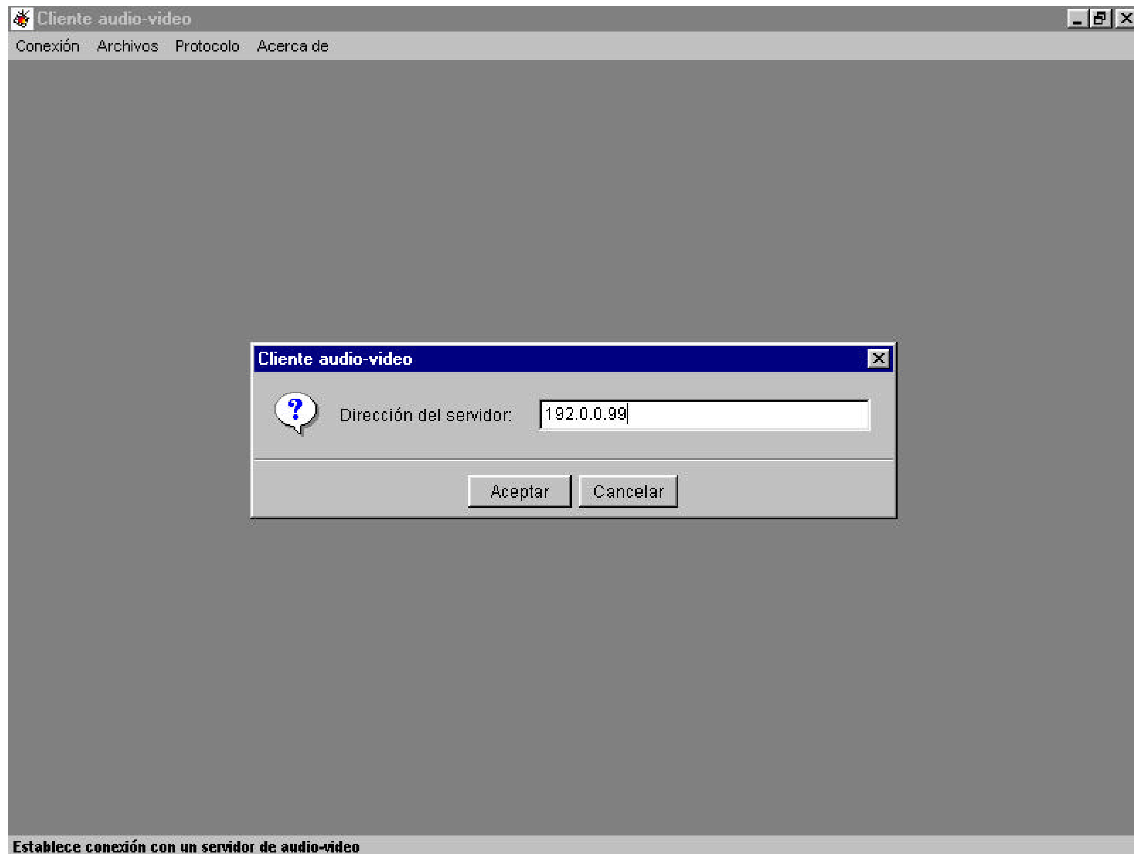
En cuanto al programa cliente, para iniciar su ejecución hay que emplear el siguiente comando:

- java cliente.ClienteAV

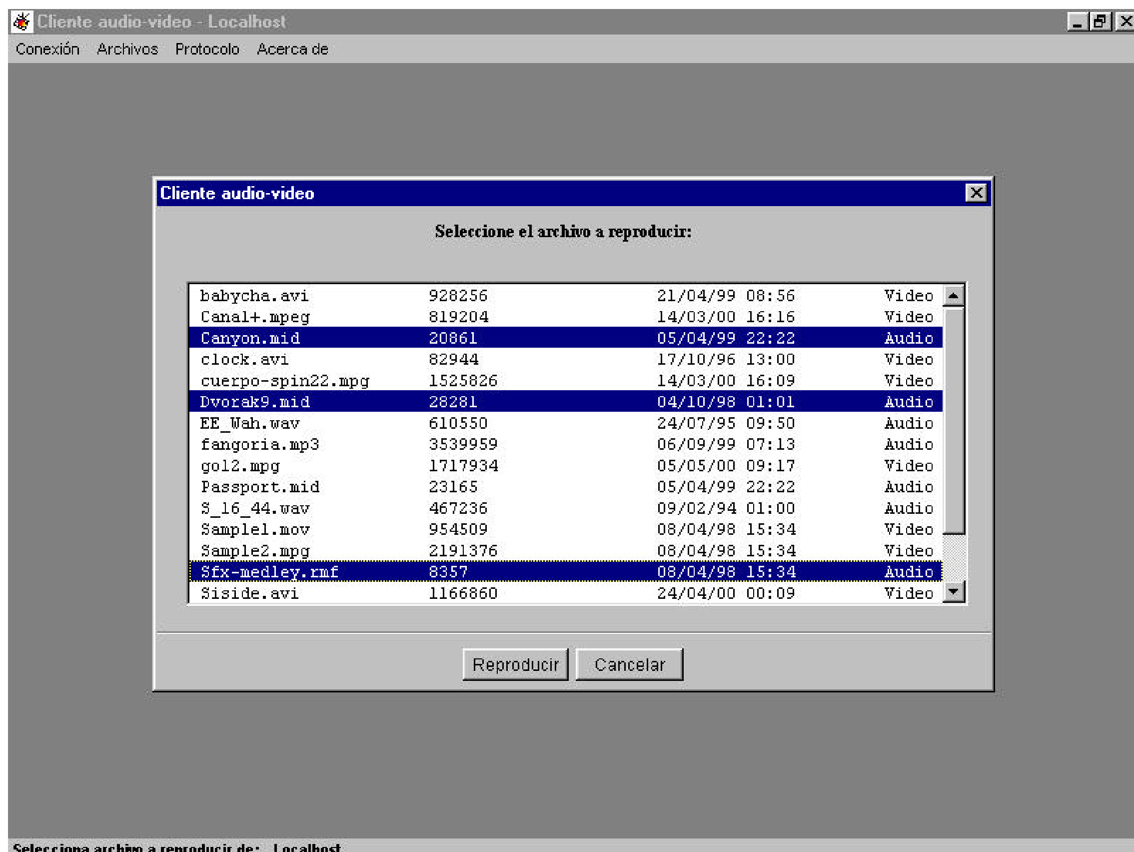
De igual modo que sucede con el servidor, también se incluye en el directorio del cliente, un archivo de proceso por lotes llamado ‘ClienteAV.bat’, que al ejecutar el programa cliente carga un fichero de políticas de seguridad llamado ‘ClienteAV.policy’, con la misma finalidad que el del servidor.

Una vez están los programas en ejecución, ya podemos interactuar con el cliente. El manejo del programa cliente es muy sencillo, su interfaz básica es un menú que permite realizar las operaciones ya descritas a lo largo de este documento: conectar con un servidor, configurar los puertos TCP y UDP, desconectar de un servidor, buscar todos los archivos de la base de datos de un servidor, o buscar sólo los determinados por las dos búsquedas elaboradas que hay disponibles, seleccionar un protocolo para la reproducción de los archivos multimedia, de entre los tres que hay, y finalmente, tras una búsqueda con éxito en la base de datos del servidor, seleccionar uno o varios archivos multimedia para reproducirlos.

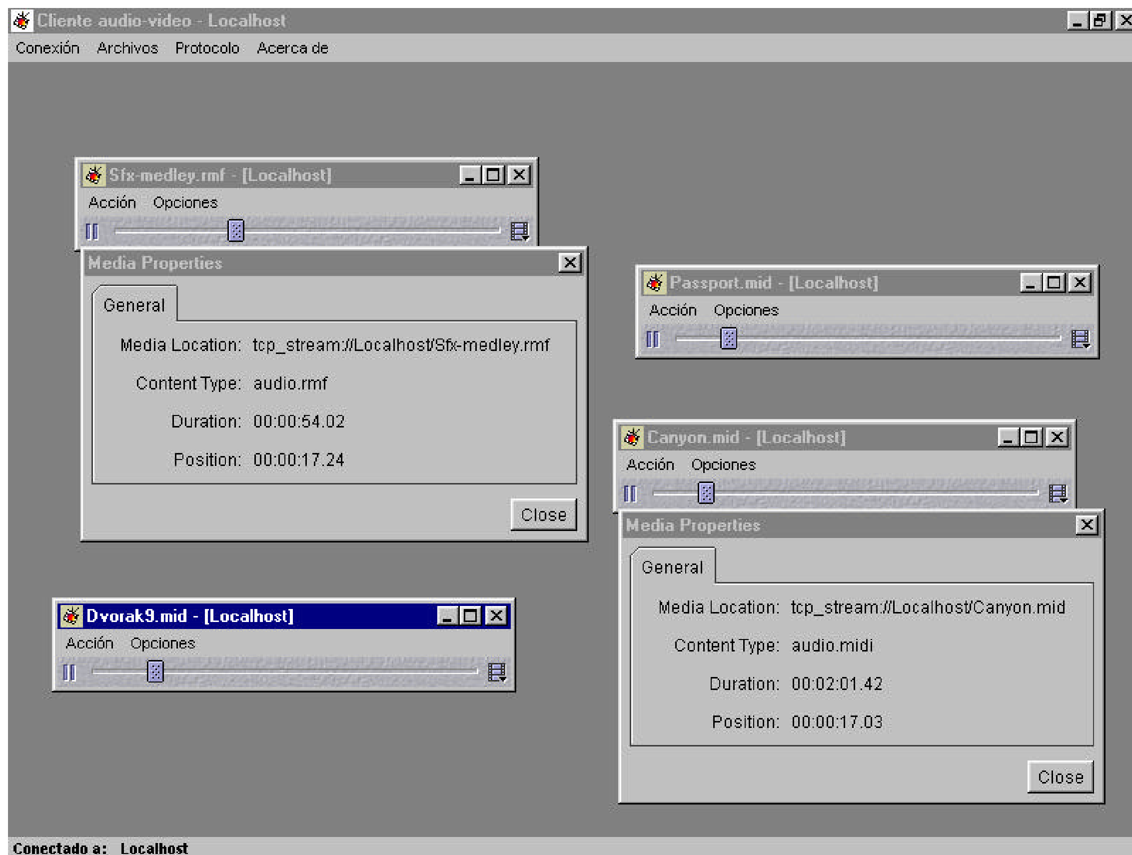
Veamos a continuación el aspecto que ofrece la aplicación en algunas de las situaciones comunes de uso. En primer lugar vemos el aspecto que tiene la aplicación cuando se abre el diálogo que solicita al usuario la dirección de un servidor de archivos multimedia:



A continuación mostramos el resultado de una búsqueda de archivos en la base de datos del servidor, donde el usuario ha seleccionado ya varios archivos multimedia:



Vemos a continuación una situación en la que están siendo reproducidos varios archivos simultáneamente empleando el protocolo TCP stream:



6. Conclusiones.

El proyecto desarrollado me ha parecido ciertamente interesante. Se trata de un tema del que se puede sacar mucho más partido del que se le ha sacado en este proyecto, sobre todo teniendo en cuenta que de un paquete como JMF, no se ha utilizado más que una mínima parte de toda la potencia que tiene (sobre todo en su última versión, que ha mejorado considerablemente a las anteriores). Sin embargo, teniendo en cuenta los objetivos que se plantearon en un inicio, y los resultados que se han obtenido al final, personalmente considero que están cubiertos, si no en su totalidad, sí al menos en gran parte. En principio, la idea era implementar un cliente que utilizase RMI para las comunicaciones y TCP para la descarga de archivos, y si funcionaba bien, adaptarlo a UDP. Ambas cosas se han conseguido.

Pero no cabe duda de que, sin salirse del tema planteado, y sin irse a buscar utilizar toda la potencia de JMF, el paso siguiente sería implementar la aplicación utilizando como protocolo de transporte el RTP. Protocolo que, por cierto está muy trabajado en el paquete JMF, aunque desconozco hasta que punto, por no haber entrado apenas en el tema. Así pues, una posible ampliación del presente proyecto sería implementar la versión RTP de la aplicación.

En el tema de las comunicaciones también podrían mejorarse los resultados y la calidad de la aplicación, utilizando CORBA en lugar de RMI. Éste último hace un buen papel en general, pero para aplicaciones de envergadura, posiblemente se podría quedar corto, sin embargo CORBA se adaptaría con facilidad. De todos modos, para una aplicación como la que nos ocupa, posiblemente meterse con CORBA no merezca la pena.

En cuanto a lo realizado que podría mejorarse, no cabe duda de que la descarga de archivos mediante el protocolo UDP no está muy afinada, por haberse realizado en poco tiempo, y de forma rápida. Funciona, pero curiosamente es más lenta que la TCP, cuando en teoría tendría que ser más rápida.

Por lo demás, destacar mi satisfacción por el trabajo realizado y por el fruto obtenido de ese trabajo, y esperar que este documento pueda ser de utilidad alguna vez para alguien interesado en el tema.

7. Bibliografía.

El siguiente material ha sido de gran ayuda en la elaboración de este proyecto fin de carrera:

- "El lenguaje de programación Java".
  Miguel Sánchez López.
  Víctor Alonso Barberán.
  UPV. Servicio de publicaciones.

- "Java 1.2 and JavaScript for C and C++ programmers".
  Michael C. Daconta.
  Wiley Computer Publishing.

- "Gestión de bases de datos en internet: JDBC".
  Jose Manuel Framiñán Torres.
  Jose Miguel León Blanco.
  Anaya multimedia.

- "Graphic Java 1.2".
  David M. Geary.
  Sun microsystems

- Sitio web de SUN: java.sun.com

- Documentación suministrada con el paquete JMF 1.0.1.