La Clase WebStream

J.G Nieto

3 March 2005

Resumen. Debido a la importancia que está tomando la computación distribuida en una gran variedad de proyectos y líneas de investigación, cada vez es más usual implementar sistemas distribuidos a través de Internet. Existen una serie de herramientas y estándares que podemos utilizar para implementar este tipo de sistemas, siendo lo más común el desarrollo de servicios que atienden peticiones realizadas desde procesos clientes. CORBA [8], SOAP [3] y RMI [4] son algunas de las herramientas más utilizadas hoy en día para realizar este tipo de sistemas, aunque la tecnología por excelencia para realizar comunicación entre procesos distribuidos en una red es `sockets` [10].

En este escrito se presenta la clase Java `WebStream`, basada en `sockets` y que proporciona una sencilla interfaz de métodos para la realización de sistemas cliente/servidor. Con esta intención, se describe las características fundamentales de esta herramienta, centrando la atención en sus métodos y constructores y finalmente se exponen sencillos ejemplos de codificación de sistemas C/S mediante `WebStream`.

1. Introducción

La interconexión entre diferentes aplicaciones y servicios a través de una red, está experimentando un gran auge actualmente debido a la necesidad de desarrollar sistemas distribuidos [10], en en los cuales es necesario realizar llamadas que deben ser encaminadas a través de *Internet* a un servicio que reside en un sistema remoto. Esto nos lleva al concepto de distribuir e integrar la lógica de la aplicación a través de la Web y utilizar protocolos como *HTTP* y *SMTP*, así como estandares como *XML* para conseguir un alcance cada vez más universal.

Podemos basarnos en varios paradigmas de diseño para desarrollar este tipo de aplicaciones como el *Paso de Mensajes* (*Síncrono* o *Asíncrono*), *Cliente/Servidor* (*C/S*), *Peer to Peer* (*ptp*) [11][12], etc. Existen herramientas que nos ayudan a trabajar en este sentido como los *Sockets* y *RMI (Remote Method Invocation)* mediante las cuales podemos establecer sistemas C/S, ya sea por paso de mensajes o por invocación de métodos remotos. También se han estandarizado especificaciones como SOAP (Simple Object Access Protocol) y CORBA, a partir de las cuales, se han desarrollado bibliotecas de funciones para la interconexión de procesos en diferentes lenguajes de programación como Java, C++, C#, etc.

Con la intención de agilizar el desarrollo de determinados sistemas distribuidos se presenta en este escrito la herramienta `WebStream`, cuya función principal es la de proveer mecanismos para la comunicación y el intercambio de información entre procesos situados en diferentes puntos de una red, generalmente a través de Internet.

Básicamente, `WebStream` es una clase Java que encapsula el funcionamiento de las archiconocidas clases Java `Socket` [4] y `ServerSocket` [4], utilizadas para la comunicación de procesos mediante el establecimiento de sistemas C/S. El objetivo de `WebStream` es el de ofrecer métodos mediante los cuales se pueden enviar y recibir tipos de datos complejos como cadenas de caracteres, ficheros e incluso objetos entre dos procesos distanciados una red, facilitando bastante el trabajo a los programadores de este tipo de sistemas.

En la siguiente sección, se describen algunas de las herramientas más utilizadas hoy en día para el desarrollo de sistemas distribuidos. En la sección 3 se presenta la clase `WebStream`, donde se define su arquitectura y principales características. En la sección 4 se enumeran los constructores y métodos de la clase dando una breve descripción de cada uno. La sección 5 contiene ejemplos de uso mediante la codificación de un sencillo programa *Hello Word!*. Otro aspecto importante observar su funcionamiento y resultados tanto en este ejemplo como en otros sistemas de mayor envergadura en los cuales se ha usado (sección 6). Finaliza en la sección 7 en la que se comentan los conclusiones y posibles actualizaciones a realizar sobre `WebStream`.

2. Implementando Sistemas Distribuidos

Como se ha comentado en la sección anterior, existen multitud de herramientas para la creación de sistemas distribuidos en la red, mediante mecanismos para el intercambio de información entre procesos que se ejecutan en diferentes máquinas situadas en puntos distanciados de una red.

Una de las tecnologías que más se han extendido en los últimos años es el estándar SOAP para el desarrollo de sistemas C/S sobre la Web. SOAP proporciona un modo abierto y extensible para que las aplicaciones se comuniquen a través de la Web usando mensajes basados en *XML*, con independencia de los sistemas operativos, modelos de objetos o lenguajes de programación que cada aplicación utilice y proporcionando, además, un modo para enviar esos mensajes sobre *HTTP*.

Esto garantiza que cualquier cliente con un navegador estándar pueda conectarse con un servidor remoto. La transmisión de datos empaquetada (serializada) con *XML* salva las incompatibilidades de otros sistemas. Por otra parte, los servidores Web pueden procesar las peticiones de usuario, empleando las tecnologías de *Servlets* [4], paginas *ASP (Active Server Pages)* [7] o *JSP (Java Server Pages)* [5], o un servidor de aplicaciones invocando objetos de los tipos *CORBA* [8], *COM* [9] o *EJB* [9].

Figura 1: Esquema del funcionamiento de *SOAP*

La especificación *SOAP* menciona que las aplicaciones deben ser independientes del lenguaje de desarrollo, por lo que las aplicaciones *Cliente* y *Servidor* pueden estar escritas con *HTML, DHTML, Java, Visual Basic* u otras herramientas y lenguajes disponibles. Lo importante es tener alguna implementación de *SOAP* (dependiendo de la herramienta de desarrollo elegida) y enlazar sus bibliotecas con la aplicación.

Aunque esto no es estrictamente necesario, es preferible trabajar usando dichas bibliotecas, con el fin de no reescribir un código ya probado. En figura la 1 se muestra el esquema del funcionamiento de *SOAP*.

Otra herramienta para el desarrollo de sistemas distribuidos es la especificación CORBA (Common Object Request Broker Architecture), que proporciona mecanismos uniformes para invocar métodos remotos. Uno de los componentes principales de la arquitectura de CORBA es el ORB (Object Request Broker) mediante el cual se establecen mecanismos transparentes de comunicación entre objetos. La clave para la interoperabilidad de objetos es el IDL (Dynamic Invocation Interface), que establece una interfaz común entre objetos remotos e implementados con diferentes lenguajes de programación. En las siguientes figuras se muestra un esquema de la arquitectura de CORBA (Figura 2 izquierda) y de su interoperabilidad de lenguajes (Figura 2 derecha).

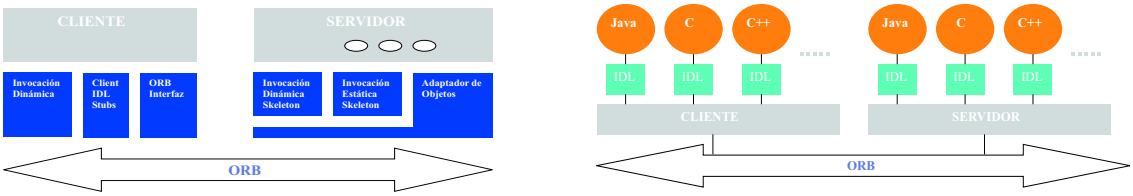

Figura 2: Esquema de la arquitectura e interoperabilidad de lenguajes de CORBA

Al compilarse las aplicaciones cliente y servidor CORBA se generan el *stub*[8] y el *skeleton*[8] correspondientes, los cuales se proporcionan para generar las referencias a los objetos remotos, similar a lo que ocurre en RMI.

Por último, con RMI (Remote Method Invocation) tenemos una herramienta que nos proporciona la plataforma J2EE de Java mediante la librería de funciones `java.rmi` para la creación y manejo de objetos remotos. El sistema RMI de Java asume el entorno homogéneo de la máquina virtual de Java (JVM), y por tanto el sistema puede tomar ventaja del modelo de objeto de Java siempre que sea posible.

Las aplicaciones RMI normalmente están compuestas por dos programas separados: un servidor y un cliente. Normalmente la aplicación servidor crea un número de objetos remotos, hace accesible las referencias a estos objetos y espera hasta que los clientes invoquen métodos en dichos objetos remotos. Una aplicación cliente típica obtiene una referencia remota a uno o más objetos remotos en el servidor e invoca métodos sobre estos objetos, de este modo RMI proporciona el mecanismo mediante el cual el servidor y el cliente se comunican e intercambian información. En la siguiente figura se muestra un resumido esquema del modo de operar de RMI en comparación con CORBA.

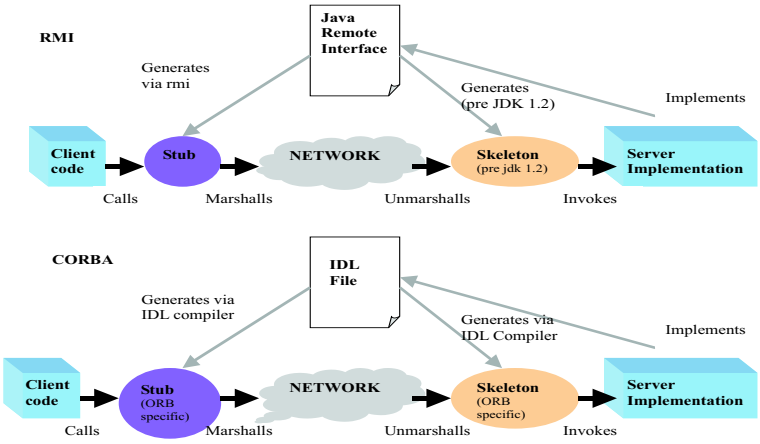

Figura 3: Operación RMI vs CORBA.

Al compilarse un objeto RMI se generan el *Stub* [4] que proporcionan la referencia remota del objeto al cliente y el *skeleton* [4] que proporciona la invocación remota al objeto remoto en el servidor. En este caso, el cliente realiza la llamada al objeto remoto mediante una instancia de su interfaz (declarada remota) que es proporcionada por el *stub*.

La gran ventaja de estas tecnologías, es que ofrecen al desarrollador la posibilidad de realizar llamadas remotas de manera transparente a la arquitectura de red que trabaja debajo de la implementación, es decir, cuando se realiza una llamada a un objeto que está en otra aplicación en un punto distanciado de una red, lo hace de manera similar a las llamadas a objetos locales así que no es necesario el manejo implícito de pasos de mensajes, sincronización, y todas las tareas a realizar en la comunicación entre procesos.

La major desventaja es que requiere una mayor carga de proceso debido a tareas como la traducción de interfaces, registro de objetos servicio de nombres, servicio de objetos, etc.

3. La Clase WebStream

`WebStream` es una herramienta para la comunicación entre procesos (C/S) mediante *Paso de Mensajes Sincrono* y con la que es posible establecer un servicio *Orientado a la Conexión*. El objetivo de `WebStream` es ofrecer una serie de mejoras sobre otros sistemas de comunicación de procesos remotos (sockets, RPC, RMI, etc), para facilitar al programador la implementación de sistemas C/S con la posibilidad de intercambiar tipos de datos construidos como arrays, ficheros e incluso objetos.

`WebStream` está implementado mediante una clase Java que encapsula el servicio de `sockets TCP` de la biblioteca `java.net`, de modo que proporciona una interfaz de métodos muy parecida a la de `sockets` como métodos para el establecimiento de una conexión, envío y recepción de mensajes, finalización de la conexión, etc.

Para implementar un sistema de comunicación entre procesos C/S mediante `WebStream` se deben crear instancia de los constructores (seción 4) y se invocan los métodos para el establecimiento de la conexión como muestra la figura 4.

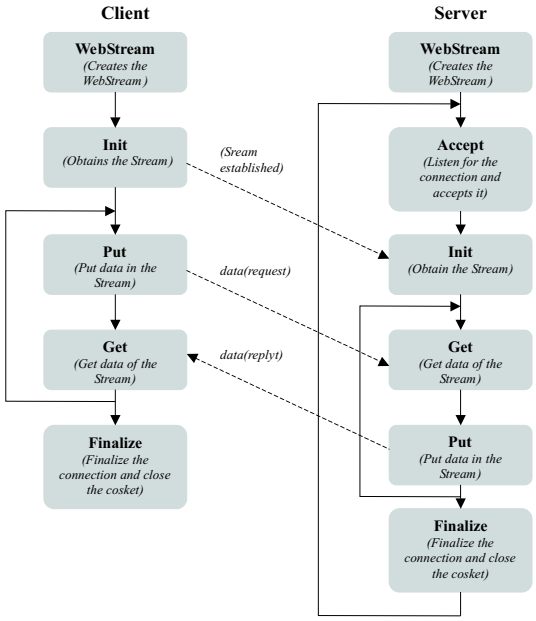

Figura 4: Funcionamiento de WebStream (C/S).

`WebStream` proporciona la posibilidad de intercambio de información *full-duplex* por lo que se realizan escrituras y lecturas en ambos sentidos de la conexión. Los pasos a seguir en una aplicación C/S típica con este servicio son los siguientes:

- En el servidor:
  1. Creación de un objeto `WebStream` servidor, introduciendo el número de puerto como argumento al constructor.
  2. Aceptación de un `stream` (flujo) de la cola de peticiones para asignarlo al `WebStream`, mediante `accept()`.
  3. Iniciación de la conexión asignando el `stream` obtenido al `WebStream`, mediante el método `init()`.
  4. Lectura de datos disponibles en el `stream`, mediante los métodos `get_`.
  5. Escritura en el `stream`, mediante los métodos `put_`.

6. Finalización de la conexión, con el método `finalize()` que cierra el `WebStream`.

- En el cliente:

1. Creación del `WebStream` cliente, introduciendo la dirección del servidor y el número de puerto de comunicaciones como argumentos.
2. Inicialización de la conexión, asignando un `stream` al `WebStream` mediante el método `init()`.
3. Escritura de datos en el `stream`, mediante los métodos `put_`.
4. Lectura de datos disponibles en el `stream`, mediante los métodos `get_`.
5. Finalización de la conexión, con el método `finalize()` que cierra el `WebStream`.

La idea fundamental de los `sockets` es dar la sensación de estar trabajando directamente sobre un fichero, el cual está situado en un lugar lejano de la red, así que se escribe y se lee en sobre él como si realmente estuviese en la máquina local. Con `WebStream` la idea es la misma, la lectura y escritura de datos conlleva implícitamente la recepción y el envío de datos de un extremo a otro de la conexión.

En la siguiente sección se muestra una relación de los constructores y métodos que proporciona la clase `WebStream`.

4. WebStream API

A diferencia de los `sockets` Java, en `WebStream` el constructor para el servidor y para el cliente es el mismo, salvo en el número de parámetros que se le pasa en cada caso. En el caso del cliente, es necesario especificar la dirección (mediante un tipo `String`) del servidor al que conectarse y el puerto, mientras que en el caso del servidor solo es necesario especificar el número de puerto en el que dará servicio (ver la figura 5).

Constructor	Description
`WebStream(int p)`	Creates a Server WebStream specifying in `p` the port number (80 default)
`WebStream(String ad, int p)`	Creates a Client WebStream specifying in `p` the port number (80 default) and the host address `ad` to connect.

Figura 5: Constructores de la clase WebStream.

La interfaz de la clase `WebStrem` se compone fundamentalmente de dos tipos de métodos: métodos de lectura de datos del `Stream` (con sintaxis `get_()`) y métodos para la escritura de datos (con sintaxis `put_()`).

Una de las características principales en el envío y la recepción es la capacidad de manejar tipos de datos compuestos, es decir, mediante una única llamada `get/put`, se reciben/envían *buffers* de datos completos (indicando la longitud), cadenas de caracteres, ficheros e incluso objetos (realizando serialización).

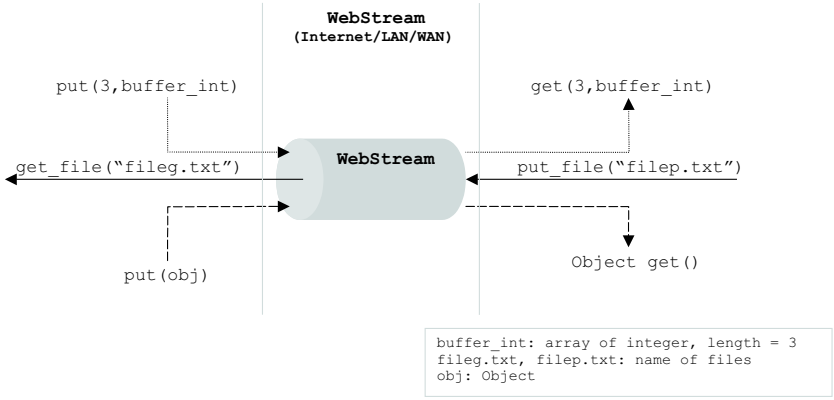

Figura 6: Operadores de lectura y escritura sobre WebStream.

Como puede observarse en la figura 6, se utilizan los operadores de lectura/escritura para intercambiar tres tipos de datos distintos. En la linea de arriba, se escribe un array de 3 enteros (`put(3,buffer_int)`) y se lee en el otro extremo el colocándolo de nuevo en otro array del mismo nombre y longitud mediante `get(3,buffer_int)`. La linea del centro representa el envío y recepción de un fichero cuyo nombre en el origen es `filep.txt` y en el destino se guarda en el fichero `fileg.txt`. Por último, en la linea de abajo el dato a intercambiar es un objeto (obj), lo cual se realiza internamente mediante *serialización* (debido a las especificaciones de Java, el objeto debe implementar la interfaz `Serializable`).

La figura 7 contiene una relación de todos los métodos de la interfaz de `WebStream`. Además de los métodos mencionados de lectura y escritura ofrece también métodos de inicialización, consulta y finalización del la conexión como `accept()`, `init()`, `flush()`, `finalize()`, y `available()`.

5. Hello Word! con WebStream

A continuación se presentan unos ejemplos de uso de WebStrem. El primero implementa un sencillo programa C/S para enviar y recibir mensajes de saludo mediante cadenas de caracteres (tipo `String`). La dirección del servidor con la que conecta el cliente es `c18.lcc.uma.es` y se establece la conexión en el puerto 4004.

Return	Method	Description
void	accept()	Accepts a new connectio an assigns a stream to the WebStream.
void	init()	Establishes the connection between two WebStreams.
void	put(Object obj)	Puts an object in the stream.
Object	get()	Gets an object from the stream.
int	put_file(String f)	Put a file specified in the stream by name f argument. Returns the length of the file.
int	put_string(int len, String s)	Put a string in the stream. Returns the length of the file.
int	put(int len, byte [] bytebuff)	Put a buffer of bytes in the stream especifying the length.
int	put(int len, int [] intbuff)	Put a buffer of ints in the stream especifying the length.
int	put(int len, boolean [] booleanbuff)	Put a buffer of booleans in the stream especifying the length.
int	put(int len, float [] floatbuff)	Put a buffer of floats in the stream especifying the length.
int	put(int len, double [] doublebuff)	Put a buffer of doubles in the stream especifying the length.
int	put(int len, long [] longbuff)	Put a buffer of longs in the stream especifying the length.
int	put(int len, short [] shortbuff)	Put a buffer of shorts in the stream especifying the length.
int	get_file(String f)	Obtains a file from the stream and writes it int a file named as f argument.
String	get_string()	Obtains a String from the stream and returns it.
int	get(int len, byte [] rec_bytebuff)	Obtains a buffer of bytes from the stream and put it in the rec_intbuff argument.
int	get(int len, int [] rec_intbuff)	Obtains a buffer of integers from the stream and put it in the rec_intbuff argument.
int	get(int len, boolean [] rec_booleanbuff)	Obtains a buffer of booleans from the stream and put it in the rec_booleanbuff argument.
int	get(int len,float [] rec_floatbuff)	Obtains a buffer of floats from the stream and put it in the rec_floatbuff argument.
int	get(int len,double [] rec_doublebuff)	Obtains a buffer of doubles from the stream and put it in the rec_doublebuff argument.
int	get(int len, long [] rec_longbuff)	Obtains a buffer of longs from t he stream and put it in the rec_logbuff argument.
int	get(int len, short [] rec_shortbuff)	Obtains a buffer of shorts from the stream and put it in the rec_shortbuff argument.
int	available()	Returns the number of bytes to read from stream.
int	flush()	Forces to clean the stream.
void	finalize()	Close the WebStream and finalizes it.

Figura 7: Relación de métodos del interfaz de la clase WebStream.

- Código del programa cliente `HelloWord_WS.java`

```
...
String s = "Hello World From Clien"; // String to send
try{
WebStream ws = new WebStream("c18.lcc.uma.es",4004);//connect with
c18.lcc.uma.es
ws.init(); // init the connection
ws.put_string(s.length(),s); // put the String in the WebStream
// receive a String from server and print it in the screen
```

```
System.out.println("Server said : "+ws.get_string());
ws.finalize(); // conclude the connection
}catch(IOException ioe){
...
```

- Código del programa servidor `HelloWord_WS_Server.java`

```
...
String s = "Hello World From Server";// String to send
try{
WebStream ws = new WebStream(4004); //Create a WebStream (Server)
System.out.println("Server listening...");
ws.accept(); Accept the connection
ws.init();// init the connection
// receive a String and print it
System.out.println(" Client said : "+ws.get_string());
ws.put_string(s.length(),s);// put a String in the WebStream
ws.finalize();// Conclude the connection
}catch(IOException ioe){
...
```

Para probar estos programas se debe ejecutar primero el `WebStream` que actúa como servidor que permanece a la espera hasta que recibe una petición del cliente. Si se ejecutara en primer lugar el cliente se generaría un error de conexión pues intentaría conectar con el servidor que no está activo aún.

- Ejecución de *Hello World Server*

```
%java HelloWorld_WS_Server
Server listening...
Client said : Hello World From Client
```

- Ejecución de *Hello World Client*

```
%java HelloWorld_WS
Server said : Hello World From Server
```

De este modo, el pequeño sistema C/S en primer lugar permanece en espera hasta que el cliente le envía el saludo (Hello Word From Client). Tras esto, el servidor escribe en pantalla el saludo y se lo devuelve al cliente (Hello word From Server).

El siguiente ejemplo realiza una tarea similar al anterior pero en este caso se envía un fichero desde el cliente al servidor.

El cliente conecta con el servidor y le envía un fichero llamado "optimization_algorithm.xml", el servidor recibe el fichero de cual obtiene la primera linea y la imprime por pantalla.

- Código del programa cliente `FileSent.java`

```
...
String filename = "optimization_algorithm.xml"; // Name of the file
int [] arrayint = new int[1];
try{
// Create a new WebStream Client
// connect with c18.lcc.uma.es
WebStream ws = new WebStream("c18.lcc.uma.es",4004);
// init the connection
ws.init();
File file = new File(filename);
ws.put_file(name);// put the file in the WebStream
//receive an array of integer with one element
int i = ws.get(1,arrayint);
System.out.println("Length of file : "+ arrayint[0]+"bytes");
ws.finalize();// conclude the connection
}catch(IOException ioe){
...
```

- Código del programa servidor `ReceiveFile.java`

```
...
int length = 0;
int [] arrayint = new int[1];
try{ //Create a new WebStream Server type
WebStream ws = new WebStream();
System.out.println("Server listening...");
// Accept the connection
ws.accept();
// init the connection
ws.init();
// receive a file and print the first line
DataInputStream f = new DataInputStream(new FileInputStream(ws.get_file()));
System.out.println("First line : "+f.readLine());
//put an array of int in the WebStream
arrayint[0] = f.available();
int i = ws.put(1,arrayint);
// Conclude the connection
f.close();
ws.finalize();
}catch(IOException ioe){
...
```

Como se puede observar, en la ejecución de este ejemplo los pasos son similares a los del ejemplo anterior. Primero se ejecuta el servidor *ReceiveFile* y permanece a la espera de peticiones, después se ejecuta el cliente que envía el fichero al servidor el cual imprime la primera línea. Seguidamente el servidor "pone" en el WebStream un `array` de enteros con un elemento (longitud del fichero), el cual es recibido por el cliente que lo termina imprimiendo por pantalla (667 bytes).

- Ejecución del servidor *ReceiveFile*

```
%java ReceiveFile
Server listening...
First line : xml version=1.0 encoding=UTF-8
```

- Ejecución del cliente *FileSent*

```
%java FileSent
Length of file : 667 bytes
```

Por último, en la siguiente sección se describe brevemente una aplicación distribuida en la que se ha utilizado `WebStream` para el intercambio de información entre sus procesos.

6. Aplicaciones de WebStream

ROS (Remote Optimization Service) [1] es una aplicación que pretende ofrecer un servicio o repositorio de algoritmos de optimización, al cual se puede acceder mediante una serie de conexiones desde una aplicación cliente.

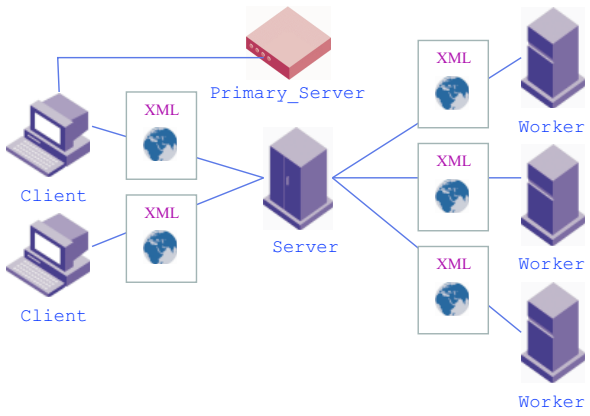

Figura 8: Esquema resumido de la arquitectura de ROS.

La arquitectura de ROS se compone de una serie de clientes y servidores mediante los que se intercambia información relativa a la ejecución de los algoritmos, direcciones de servidores, y usuarios.

En la figura 8 se muestra un resumido esquema de los procesos componentes de ROS. El proceso *client* actúa como cliente frente al los procesos *primary_ server* y *server* con los cuales intercambia información, a su vez, el proceso *server* actúa como cliente respecto a los *workers* que son los servidores finales.

Esta interacción C/S se ha implementado mediante `WebStream` en cada paso de la comunicación, en la figura se representa mediante las líneas de conexión entre las distintas máquinas, indicándose en algunos casos el intercambio de ficheros XML que se realiza durante la ejecución de ROS. Existe otra versión de este sistema empleando la implementación SOAP de Java para realizar las comunicaciones en lugar de `WebStream`, esto requiere una mayor complejidad de desarrollo, así como la necesidad de utilizar servidores Web para la ejecución, complicando de este modo su instalación y mantenimiento.

7. Conclusiones

`WebStream` es una herramienta desarrollada para facilitar la implementación de sistemas distribuidos C/S, proponiendo una alternativa a otras tecnologías como RMI, CORBA o SOAP. Su objetivo principal es proporcionar a los programadores de Java una clase con una interfaz de métodos sencilla y con la que se abstrae el procedimiento general de implementación de `sockets`.

Una de sus principales ventajas es la capacidad manejar una amplia variedad de tipos de datos, desde un simple byte hasta Objetos Java serializados. Además, proporciona el mismo constructor para el cliente y el servidor facilitando aun más su utilización.

No obstante, `WebStream` carece de propiedades importantes que se pueden incorporar en futuras versiones como la posibilidad de ofrecer servicio UDP, realizar envíos MultiCast, manejar direcciones IPv6, comunicación de alta velocidad mediante el uso "selectores" (paquete NIO de java)[14], e incluso incorporar seguridad en la transmisión incorporando el protocolo SSL. Como ocurre en la gran mayoría de los casos, esta herramienta se desarrolló para ser utilizada en un proyecto mayor, concretamente en ROS, pero una vez comprobado su correcto funcionamiento y facilidad de uso se hace susceptible de ser utilizada en otros proyectos de similares características.

Referencias

[1] The ROS site. `http://tracer.lcc.uma.es/ros/index.html`.

[2] G Booch, Object-Oriented Analysis and Design with Applications, Addison Wesley, 1993.

[3] B McLaughlin, Java and XML, O'Relly, 2001.

[4] A Pew, Instant Java, The Sunsoft Press - Pretince Hall, 1996.

[5] A Bakharia, Java Server Pages: Fast and Easy Web Development, Premier, 2001.

[6] A Froufe, Java 2 Manual de Usuario, Ra-Ma, 2000.

[7] D Espósito, Building Web Solutions with ASP .NET and ADO .NET, Microsoft Press, 2002.

[8] D Harkey, Client/Server Programming With Java and CORBA, J Wiley, 1997.

[9] J Griffin and Others, Professional EJB, Wrox Press, 2001.

[10] A Tanenbaum, Computer Networks, Prentice Hall, 1996.

[11] A Tanenbaum, M van Steen, Distributed Systems: Principles and Paradigms, Pretince-Hall, 2001.

[12] Distributed systems: concepts and design, (3rd ed.), Addison-Wesley, 2000.

[13] JSSE Reference Guide `http://java.sun.com/j2se/1.5.0/docs/guide/security/jsse/JSSERefGuide.html`

[14] Gregory M. Travis, JDK 1.4 Tutorial, Softbound, 2002.