4 Generación de servicios en red

OBJETIVOS DEL CAPÍTULO

- ✓ Aprender el concepto de "servicio" y su aplicación en el contexto de la computación distribuida.
- ✓ Aprender a programar aplicaciones distribuidas siguiendo el modelo cliente/servidor.
- ✓ Entender las características de un protocolo de nivel de aplicación y conocer los protocolos de nivel de aplicación más usados en la actualidad.
- ✓ Conocer de la existencia de tecnologías avanzadas de comunicación, más allá de los *sockets*, y aprender a programar aplicaciones usando Java RMI.
- ✓ Aprender qué son los servicios web y cuáles son sus principales ventajas.

4.1 SERVICIOS

En el capítulo anterior se han visto los conceptos fundamentales de la computación distribuida y los mecanismos básicos de comunicación en red. Dentro de estos conceptos fundamentales, se ha desarrollado el modelo de comunicaciones cliente/servidor, el más usado en la actualidad en la computación distribuida. Al definir este modelo, se dice que uno de sus elementos clave es el *servidor*, una aplicación que proporciona servicios a uno o más *clientes*. Pero ¿qué es exactamente un servicio? ¿Qué características tiene? A lo largo de este capítulo se profundizará en este y otros conceptos relacionados, y se aprenderá a desarrollar aplicaciones complejas que proporcionen servicios.

4.1.1 CONCEPTO DE SERVICIO

Desde un punto de vista básico, todo sistema presenta dos partes fundamentales: **estructura** y **función**. La estructura está formada por aquellos componentes que conforman el sistema, es decir, las piezas que unidas lo forman. En informática en general, y en los sistemas distribuidos en particular, la estructura del sistema serán aquellos componentes hardware y software que, conectados entre sí, forman el ordenador, sistema distribuido, o lo que estemos analizando. La función, por otro lado, es aquello para lo que está pensado el sistema, es decir, para qué sirve y/o se usa.

EJEMPLO 4.1

La descomposición conceptual en estructura y función se puede aplicar a cualquier sistema tecnológico, ya sea en el campo de la informática o fuera de ella. Un electrodoméstico, una lavadora por ejemplo, es un sistema que consta de estructura y función, como cualquier otro. En este ejemplo, la estructura de la lavadora son la piezas mecánicas y electrónicas que lo forman (carcasa, tambor, puerta, controlador electrónico, motor interno…). La función de la lavadora es, obviamente, lavar la ropa.

EJEMPLO 4.2

Determinar la estructura y función de un sistema distribuido es igual de sencillo que determinar la de cualquier otro sistema. Si pensamos en una aplicación de mensajería instantánea, como WhatsApp, por ejemplo, la estructura estaría formada por las aplicaciones cliente y servidor, es decir, el software que lo compone. La función de este sistema es, sencillamente, permitir que los usuarios puedan comunicarse enviando mensajes de texto, imágenes, vídeos, etc.

Todo sistema tiene, por tanto, una estructura y una función. Como se puede observar, la estructura es algo concreto, relacionado con los componentes hardware y software que forman el sistema. La función, en cambio, suele ser una serie de conceptos abstractos, que explican "lo que hace" el sistema, pero no "cómo lo hace". Para que un sistema realice su función, se deben especificar los mecanismos concretos que este proporciona a sus usuarios. A este conjunto de

mecanismos concretos que hacen posible el acceso a la función del sistema se le denomina **servicio**. Un sistema puede proporcionar uno o más servicios, en función de cuál sea su función.

Además, normalmente, este conjunto de mecanismos, al que se llama "servicio", se especifica de manera concreta e impone restricciones específicas de utilización. Cualquier usuario del sistema ha de seguir una serie de procedimientos determinados a la hora de acceder al sistema y al usarlo. Esta serie de procedimientos se denomina **interfaz del servicio** y es el punto de contacto entre el sistema y el usuario.

EJEMPLO 4.3

Continuando con el ejemplo de la lavadora, el servicio proporcionado por esta se define como el conjunto de mecanismos que esta realiza para llevar a cabo su función. En este caso podemos hablar de operaciones como lavado con agua caliente, lavado en frío, programa para tejidos sintéticos, etc. La interfaz del servicio son los componentes con los que interactúa el usuario, como la puerta de carga, el cuadro de mandos, etc. Estos condicionan los procedimientos concretos que el usuario debe seguir para obtener el servicio. Para lavar en frío, por ejemplo, el usuario debe abrir la puerta, cargar la ropa en el tambor, cerrar la puerta, llenar la cubeta del detergente, ajustar el programa adecuado en el panel y pulsar el botón de encendido.

EJEMPLO 4.4

En el ejemplo de la aplicación de mensajería instantánea (Ejemplo 4.2), los servicios son las funciones específicas de la aplicación, como buscar a un amigo, enviarle un mensaje, enviarle una foto, etc. La interfaz del servicio es la propia interfaz de la aplicación cliente, que marca la manera concreta en la que se realizan dichas operaciones. Para enviar una foto, por ejemplo, estas operaciones podrían ser seleccionar al destinatario en la lista de contactos, pulsar el botón de **Enviar foto**, seleccionar una fotografía de la galería de imágenes y pulsar el botón de **Enviar**.

ACTIVIDADES 4.1

- Identifica estructura, función, servicio e interfaz de servicio en los siguientes ejemplos:
  a. Un procesador de textos (como Word o LibreOffice Writer).
  b. Un autobús urbano.
  c. Una plataforma de juegos *on-line*, como Steam o XBOX Live.
  d. Una cámara fotográfica.

4.1.2 SERVICIOS EN RED

Durante las comunicaciones entre los diferentes elementos de una aplicación distribuida, se hace uso de multitud de servicios distintos, que cooperan entre sí para conseguir el paso de mensajes entre emisores y receptores. La pila de protocolos IP es, en efecto, un conjunto de sistemas independientes, pero montados unos sobre otros para realizar una tarea compleja. Cada nivel de la jerarquía (red, Internet, transporte y aplicación) proporciona un servicio específico, tal y como se vio en el capítulo anterior, y ofrece una interfaz de servicio al nivel superior, a través de la cual interactúa con este. Además, para que las comunicaciones puedan llevarse a cabo, cada nivel de la pila dispone de su propio protocolo de comunicaciones, que gobierna la interacción a ese nivel con los demás elementos del sistema distribuido.

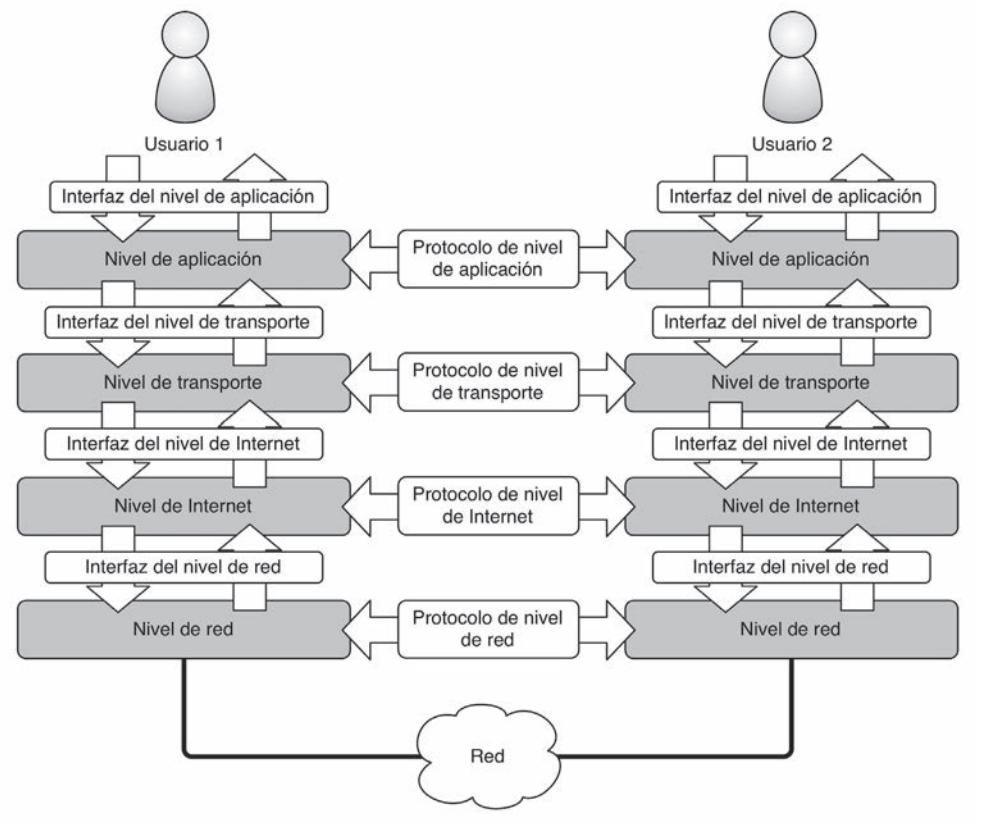

Figura 4.1. *Comunicación entre dos usuarios, usando la pila de protocolos IP*

Se puede considerar, por tanto, un servicio en red como cualquier servicio que se ubique en cualquier nivel de la pila. Las tecnologías de comunicaciones del nivel de red son servicios, los mecanismos de encaminamiento del nivel de Internet son servicios, etc.

¿SABÍAS QUE...?

El sistema de *sockets* estudiado en el capítulo anterior proporciona servicios del nivel de transporte. Su función es hacer llegar mensajes entre un emisor y un receptor, ya sea por un modelo orientado a conexión (*sockets stream*) o por medio de datagramas (*sockets datagram*). La interfaz del servicio está formada, en este caso, por las bibliotecas de programación que permiten hacer uso de los *sockets* dentro de nuestros programas. Los protocolos TCP y UDP son ejemplos de protocolos de nivel de transporte asociados a este servicio.

4.1.3 SERVICIOS DE NIVEL DE APLICACIÓN

El nivel más alto de la pila IP lo componen las aplicaciones que forman el sistema distribuido. Estas aplicaciones, al igual que en el resto de niveles, ofrecen una interfaz de servicio para que los usuarios las usen, y disponen de un protocolo de nivel de aplicación que gobierna las comunicaciones entre ellas. La mayoría de aplicaciones distribuidas más comunes se ubican en este nivel, como las páginas web, el correo electrónico o los juegos *on-line*.

Se define un **protocolo de nivel de aplicación** como el conjunto de reglas que gobiernan la interacción entre los diferentes elementos de una aplicación distribuida. A la hora de desarrollar un servicio distribuido, definir este protocolo es uno de los pasos fundamentales. En una aplicación cliente/servidor este protocolo especifica cómo se realiza la interacción entre el servidor y los clientes. A su vez, los diferentes elementos de una aplicación cliente/servidor (los clientes y el servidor) se pueden ver como sistemas independientes, con su estructura y función. En este sentido, el servidor es la pieza clave, ya que es la proporciona el servicio deseado a los clientes. Su estructura estará formada por los componentes software con los que está programado. Su función será aquella que realiza para los clientes, y su servicio el procedimiento mediante el cual la realiza. El protocolo de nivel de aplicación define cómo se interactúa con el cliente y es, por tanto, la interfaz de servicio del servidor.

¿SABÍAS QUE...?

Un servidor web es una aplicación que proporciona páginas web a los clientes que lo solicitan, normalmente navegadores web como Firefox o Internet Explorer. Si se considera el servidor web como un sistema, el servicio proporcionado por este es el acceso a la página o páginas que contiene. La interfaz de servicio del servidor web está gobernada por el protocolo de nivel de aplicación que se usa en Internet para el tráfico web, denominado HTTP *(Hipertext Transfer Protocol*). Este protocolo se estudiará (entre otros) en este capítulo.

4.2 PROGRAMACIÓN DE APLICACIONES CLIENTE Y SERVIDOR

Como se ha visto, el modelo de comunicaciones cliente/servidor es el más usado en el desarrollo de aplicaciones distribuidas. Los mecanismos de transmisión de mensajes explicados en el capítulo anterior, los *sockets stream* y *sockets datagram*, son la tecnología base sobre la que se implementa este modelo. A la hora de programar una aplicación siguiendo el modelo cliente/servidor, se deben definir de forma precisa los siguientes aspectos: **funciones del servidor**, **tecnología de comunicaciones** y **protocolo de nivel de aplicación**. A continuación veremos en detalle cada uno de estos aspectos.

4.2.1 FUNCIONES DEL SERVIDOR

A la hora de programar una aplicación distribuida siguiendo el modelo cliente/servidor, el primer paso debe ser siempre definir de forma clara las funciones del *servidor*. Esto equivale a contestar preguntas como: ¿para qué sirve nuestro servidor? Esta definición debe hacerse de manera clara y sin ambigüedades, para facilitar el desarrollo posterior y evitar problemas. Algunas de las preguntas clave que debemos hacernos a la hora de definir nuestro servidor son:

- ¿Cuál es la función básica de nuestro servidor?
- El servicio que proporciona nuestro servidor, ¿es rápido o lento?
- El servicio que proporciona nuestro servidor, ¿puede resolverse con una simple petición y respuesta o requiere del intercambio de múltiples mensajes entre este y el cliente?
- ¿Debe ser capaz nuestro servidor de atender a varios clientes simultáneamente?

El resultado de contestar estas y otras preguntas similares debe ser el hacernos una idea clara de cómo va a ser nuestro servidor, qué va a hacer y cómo va a interactuar con los clientes.

EJEMPLO 4.5

Se desea crear una aplicación cliente/servidor de consulta horaria. Como hemos dicho, el primer paso será definir las funciones del servidor. Analizamos el problema y finalmente decidimos que nuestro servidor tendrá las siguientes características:

- La función básica de nuestro servidor será "dar la hora". Cuando un cliente se conecte, realizará una petición de servicio ("preguntará la hora"). El servidor les responderá con un mensaje en el que figure la hora actual.
- Se trata de un servicio rápido, ya que solo "da la hora".
- La interacción con el cliente se limita a un único intercambio petición-respuesta ("¿Qué hora es?", "Son las 4:10").
- El servidor debe ser capaz de atender a múltiples clientes a la vez.

EJEMPLO 4.6

Se desea crear una aplicación cliente/servidor que funcione como una calculadora. Al igual que en ejemplo anterior, analizamos el problema y decidimos que nuestro servidor tendrá las siguientes características:

- La función básica de nuestro servidor será realizar operaciones aritméticas sencillas. Esto incluye sumas, restas, multiplicaciones y divisiones. Cuando un cliente se conecte, realizará una petición de servicio, indicando la operación que desea realizar ("suma", "resta", etc.) y los operandos de la operación. El servidor les responderá con un mensaje en el que figure el resultado de la operación. El cliente podrá seguir realizando operaciones aritméticas, siguiendo el mismo proceso.
- Nuestro servicio debe mantener una conexión abierta con cada cliente, ya que este debe enviar varios mensajes para solicitar una operación.
- La interacción con el cliente puede ser compleja, con múltiples mensajes intercambiados.
- Al igual que en el ejemplo anterior, el servidor debe ser capaz de atender a múltiples clientes a la vez.

ACTIVIDADES 4.2

- Se desea implementar un servicio de información meteorológica. Los clientes indicarán el nombre de la ciudad en la que viven, y el servidor les dará la temperatura prevista para el día siguiente. Basándose en los ejemplos anteriores, defina las características fundamentales que debe tener este servicio.

4.2.2 TECNOLOGÍA DE COMUNICACIONES

Una vez definidas las características de nuestro servidor, el siguiente paso es escoger la tecnología de comunicaciones que es necesario utilizar. Los dos mecanismos básicos de comunicación que se han estudiado son los *sockets stream* y los *sockets datagram*. Como ya se ha visto, cada tipo de *socket* presenta una serie de ventajas e inconvenientes. Es necesario escoger el tipo de *socket* adecuado, teniendo en cuenta las características del servicio que proporciona nuestro servidor. Los *sockets stream* son más fiables y orientados a conexión, por lo que se deben usar en aplicaciones complejas en las que clientes y servidores intercambian muchos mensajes. Los *sockets datagram* son menos fiables, pero más eficientes, por lo que es preferible usarlos cuando las aplicaciones son sencillas, y no es problema que se pierda algún mensaje.

EJEMPLO 4.7

Continuando con el Ejemplo 4.5, se escoge el tipo de *socket* más adecuado para nuestro servidor de consulta horaria. Dado que se trata de un servicio rápido, que se resuelve con un único intercambio petición-respuesta y los mensajes son cortos, escogemos utilizar *sockets datagram* para este caso.

EJEMPLO 4.8

Continuando ahora con el Ejemplo 4.6, escogemos en este caso *sockets stream* para nuestro servicio calculadora. Los *sockets stream* son más adecuados para este caso, ya que son orientados a conexión, lo que nos permite mantener un canal abierto con cada cliente mientras se realizan las operaciones aritméticas. Además, la fiabilidad de los *sockets stream* nos garantiza que los mensajes no se perderán por el camino, por lo que el servidor siempre recibirá los operandos y operadores aritméticos de forma correcta.

4.2.3 DEFINICIÓN DEL PROTOCOLO DE NIVEL DE APLICACIÓN

Llegados a este punto, se han definido de forma clara las funciones y características de nuestro servidor y hemos escogido el tipo de *socket* que vamos a usar. El último paso, antes de empezar a escribir nuestro programa, es definir el protocolo de nivel de aplicación.

Un **protocolo de nivel de aplicación** es el conjunto de reglas que gobiernan la interacción entre los diferentes elementos de una aplicación distribuida. En el caso del modelo de comunicaciones cliente/servidor, el protocolo de nivel de aplicación definirá explícitamente el formato de los mensajes que se intercambian entre cliente y servidor, así como las posibles secuencias en las que estos pueden ser enviados y recibidos. Dicho de otra forma, la definición del protocolo de nivel de aplicación debe contener todos los posibles tipos de mensajes (tanto peticiones como respuestas) que pueden ser enviados y recibidos, indicando cuándo se puede enviar cada uno y cuándo no.

EJEMPLO 4.9

Continuando con el Ejemplo 4.7, definimos el protocolo de nivel de aplicación para nuestro servicio de consulta horaria. Para ello, identificamos todos los posibles mensajes que se pueden intercambiar, y definimos su formato:

1. Mensaje 1: petición de hora. Se trata de la petición de hora por parte de un cliente.
   - **¿Quién lo envía?** El cliente.
   - **¿Cuándo se envía?** En cualquier momento, un cliente puede enviar este mensaje al servidor para realizar una petición.
   - **¿Qué contiene?** Una cadena de texto como la palabra "hora". Esto le servirá al servidor para comprobar que se trata de una petición correcta.
2. Mensaje 2: respuesta de hora. Se trata de la respuesta que envía el servidor al cliente, con la hora actual.
   - **¿Quién lo envía?** El servidor.
   - **¿Cuándo se envía?** Como respuesta a un cliente que ha realizado una petición de hora.
   - **¿Qué contiene?** Una cadena de texto con la hora, por ejemplo "10:25".

Cuando se define el protocolo de nivel de aplicación de un sistema distribuido complejo, es muy importante especificar de manera clara todas las secuencias posibles de intercambio de mensajes que pueden ocurrir en el sistema y cómo este reacciona. Asegurarnos de que el protocolo está correctamente definido y es exhaustivo es fundamental para evitar posteriormente comportamientos inesperados por parte de nuestra aplicación.

EJEMPLO 4.10

Continuando con el Ejemplo 4.8, definimos el protocolo de nivel de aplicación para nuestro servicio calculadora. Para ello, se identifican todos los posibles mensajes que se pueden intercambiar, y se define su formato:

1. Mensaje 1: inicio de operación. Se trata de la petición de inicio de operación por parte de un cliente.
   - **¿Quién lo envía?** El cliente.
   - **¿Cuándo se envía?** En cualquier momento, un cliente puede enviar este mensaje al servidor para realizar una petición, siempre y cuando otra operación no esté ya en curso por parte del mismo cliente. El cliente debe terminar dicha operación antes de comenzar una nueva.
   - **¿Qué contiene?** Una cadena de texto con un código que indique la operación que desea realizar. Los cuatro códigos posibles son "+" (para la suma), "-" (para la resta), "*" (para la multiplicación) y "/" (para la división). Al igual que en el caso anterior, esto le servirá al servidor para comprobar que se trata de una petición correcta.
2. Mensaje 2: primer operando. Se trata del primer parámetro de la operación aritmética que desea realizar el cliente.
   - **¿Quién lo envía?** El cliente.
   - **¿Cuándo se envía?** Justo después del mensaje 1, para indicar el primer operando de la operación.
   - **¿Qué contiene?** Un número entero, por ejemplo 567.
3. Mensaje 3: segundo operando. Se trata del segundo parámetro de la operación aritmética que desea realizar el cliente.
   - **¿Quién lo envía?** El cliente.
   - **¿Cuándo se envía?** Justo después del mensaje 2, para indicar el segundo operando de la operación.
   - **¿Qué contiene?** Un número entero, por ejemplo 12.
4. Mensaje 4: resultado de la operación. Se trata de la respuesta que envía el servidor al cliente, con el resultado de la operación aritmética.
   - **¿Quién lo envía?** El servidor.
   - **¿Cuándo se envía?** Después de recibir el mensaje 3, como respuesta a un cliente que ha realizado una petición.
   - **¿Qué contiene?** Un número entero con el resultado, por ejemplo 2.387.

4.2.3.1 Protocolos, sesión y estado

En una aplicación cliente/servidor, se llama *sesión* a la secuencia de mensajes que se intercambian entre cliente y servidor desde que se establece la conexión entre ellos hasta que se cierra. En algunos servicios (como el del Ejemplo 4.9), la interacción entre ambas partes se realiza de manera rápida, con un único intercambio de tipo pregunta-respuesta. En estos casos, el control de la sesión no es importante, ya que la interacción es sencilla. En otro (como el del Ejemplo 4.10), esta interacción es más compleja, ya que requiere de múltiples mensajes entre ambas partes. En estos casos, la sesión es mucho más importante, ya que la secuencia en la que se envían y reciben los mensajes es clave. El protocolo de nivel de aplicación refleja esto, y en este sentido se pueden distinguir dos tipos de protocolos:

- **Protocolos sin estado** (en inglés *stateless*): son aquellos en los que la secuencia concreta en la que se reciben los mensajes no es importante, ya que no afecta al resultado de estos. El servidor se limita a recibir las peticiones del cliente y a responderlas una por una, de forma independiente.
- **Protocolos con estado** (en inglés *stateful*): son aquellos en los que la secuencia concreta en la que se reciben los mensajes es importante, ya que afecta al resultado final de las peticiones. En estos casos, el servidor debe almacenar información intermedia durante la sesión, denominada el *estado de la sesión*. Conocer este estado es imprescindible para poder resolver las peticiones de manera correcta.

4.2.4 IMPLEMENTACIÓN

Una vez se han especificado las características del servidor, seleccionado la tecnología de comunicaciones y definido el protocolo de nivel de aplicación ya solo queda implementar la aplicación.

EJEMPLO 4.11

Continuando con el Ejemplo 4.9, esta es la implementación del servidor de nuestra aplicación de servicio horario:

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.net.SocketException;
import java.util.Date;

public class HoraServer {

    public static void main(String[] args) {

         System.out.println("Arrancando servidor de hora.");
```

EJEMPLO 4.11 (cont.)

```
            DatagramSocket datagramSocket = null;
            try {
                 InetSocketAddress addr = new InetSocketAddress("localhost", 5555);
                 datagramSocket = new DatagramSocket(addr);
            } catch (SocketException e) {
                 e.printStackTrace();
            }

            while (datagramSocket != null) {
                 try {
                      System.out.println("Esperando mensaje");

                      byte[] buffer = new byte[4];
                      DatagramPacket datagrama1 = new DatagramPacket(buffer, 4);
                      datagramSocket.receive(datagrama1);

                      String mensaje = new String(datagrama1.getData());

                      InetAddress clientAddr = datagrama1.getAddress();
                      int clientPort = datagrama1.getPort();

                      System.out.println("Mensaje recibido: desde " +
                      clientAddr + ", puerto " + clientPort);
                      System.out.println("Contenido del mensaje: " + mensaje);

                      if (mensaje.equals("hora")) {

                           System.out.println("Enviando respuesta");

                           Date d = new Date(System.currentTimeMillis());
                           byte[] respuesta = d.toString().getBytes();
                           DatagramPacket datagrama2 =
                                new DatagramPacket(respuesta, respuesta.length,
                                                   clientAddr, clientPort);
                           datagramSocket.send(datagrama2);

                           System.out.println("Mensaje enviado");
                      } else {
                           System.out.println("Mensaje recibido no reconocido");
                      }

                 } catch (IOException e) {
                      e.printStackTrace();
                 }
            }
            System.out.println("Terminado");
       }
  }
```

EJEMPLO 4.12

Ejemplo de cliente para nuestro servidor de consulta horaria (Ejemplo 4.11):

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class HoraClient {

    public static void main(String[] args) {
        try {
            System.out.println("Creando socket datagrama");

            DatagramSocket datagramSocket = new DatagramSocket();

            System.out.println("Enviando petición al servidor");

            String mensaje = new String("hora");

            InetAddress serverAddr = InetAddress.getByName("localhost");
            DatagramPacket datagrama1 =
                new DatagramPacket(mensaje.getBytes(),
                                   mensaje.getBytes().length,
                                   serverAddr, 5555);
            datagramSocket.send(datagrama1);

            System.out.println("Mensaje enviado");

            System.out.println("Recibiendo respuesta");

            byte[] respuesta = new byte[100];
            DatagramPacket datagrama2 =
                new DatagramPacket(respuesta, respuesta.length);
            datagramSocket.receive(datagrama2);

            System.out.println("Mensaje recibido: "+new String(respuesta));

            System.out.println("Cerrando el socket datagrama");

            datagramSocket.close();

            System.out.println("Terminado");

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

ACTIVIDADES 4.3

- Observa el Ejemplo 4.9, que define el protocolo de nivel de aplicación para el servidor de consultas horarias. ¿Qué modificaciones habría que introducir para que diese también la fecha, dependiendo de lo que solicitase el cliente?
- Basándote en lo desarrollado en la actividad anterior, modifica el Ejemplo 4.11 para que acepte peticiones de hora o fecha.

4.2.4.1 Utilización de hilos en aplicaciones cliente/servidor

La mayor parte de servidores están pensados para atender a múltiples clientes simultáneamente. Un servidor moderno que proporcione servicio a muchos usuarios, como el que da soporte al correo electrónico de GMail, por ejemplo, debe ser capaz de antender a miles de clientes simultáneamente. Esto quiere decir que, si al servidor le llega una nueva petición mientras está atendiendo a un cliente, se deben cumplir las siguientes condiciones:

- La nueva petición no debe interferir con la que está en curso. Dicho de otra forma, los clientes deben percibir que operan con el servidor ellos solos, independientemente de cuántos clientes haya.
- La nueva petición debe ser atendida lo antes posible, incluso de manera simultánea a la que está ya en curso. Esto evita que unos clientes tengan que esperar por otros.

Para que estas condiciones se cumplan, la mayoría de servidores optan por un implementar un enfoque *multihilo*. Un **servidor multihilo** es un servidor en el que a cada cliente se le atiende en un hilo de ejecución independiente. Cuando un nuevo cliente envía una petición, el servidor arranca un hilo específico para atender las peticiones de este cliente. Este hilo se dedica exclusivamente a interactuar con el cliente en cuestión. Mientras tanto, el hilo principal del servidor queda libre, esperando a nuevos clientes.

Los *sockets stream* son especialmente adecuados para implementar servidores multihilo. Cada vez que el *socket* servidor recibe un intento de conexión, la operación *accept* crea un nuevo *socket*, conectado con el cliente. Al implementar nuestro servidor, el procedimiento típico consiste en arrancar entonces un nuevo hilo, y darle a este el nuevo *socket* que se acaba de crear. De esta forma, el nuevo hilo dispone de un canal de comunicación ya abierto con el cliente, a través del cual pude responder a sus peticiones. El hilo principal y el *socket* servidor permanecen libres, realizando la operación *accept* a la espera de nuevas conexiones.

EJEMPLO 4.13

Continuando con el Ejemplo 4.10, esta es la implementación del servidor de nuestra aplicación de calculadora. Fíjate en que hace uso de múltiples hilos para poder atender a varios clientes de manera simultánea:

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.net.Socket;
```

EJEMPLO 4.13 (cont.)

```
public class CalcServer extends Thread {

    private Socket clientSocket;

    public CalcServer(Socket socket) {
         clientSocket = socket;
    }

    public void run() {

         try {

              System.out.println("Arrancando hilo");

              InputStream is = clientSocket.getInputStream();
              OutputStream os = clientSocket.getOutputStream();

              System.out.println("Esperando mensaje de operación");

              byte[] buffer = new byte[1];
              is.read(buffer);
              String operacion = new String(buffer);

              System.out.println("Operaciùn recibida: " + new String(operacion));

              if (operacion.equals("+") || operacion.equals("-") ||
                   operacion.equals("*") || operacion.equals("/")) {

                   System.out.println("Esperando primer operador");

                   int op1 = is.read();

                   System.out.println("Primer operador: " + op1);

                   System.out.println("Esperando segundo operador");

                   int op2 = is.read();

                   System.out.println("Segundo operador: " + op2);

                   System.out.println("Calculando resultado");

                   int result = Integer.MIN_VALUE;

                   if (operacion.equals("+")) {
                        result = op1 + op2;
```

EJEMPLO 4.13 (cont.)

```
                    } else if (operacion.equals("-")) {
                         result = op1 - op2;
                    } else if (operacion.equals("*")) {
                         result = op1 * op2;
                    } else if (operacion.equals("/")) {
                         result = op1 / op2;
                    }

                    System.out.println("Enviando resultado");

                    os.write(result);

               } else {
                    System.out.println("Operaciùon no reconocida");
               }

          } catch (Exception e) {
               e.printStackTrace();
          }

          System.out.println("Hilo terminado");
     }

     public static void main(String[] args) {

          System.out.println("Creando socket servidor");

          ServerSocket serverSocket = null;

          try {
               serverSocket = new ServerSocket();

               System.out.println("Realizando el bind");

               InetSocketAddress addr = new InetSocketAddress("localhost", 5555);
               serverSocket.bind(addr);

          } catch (IOException e) {
               e.printStackTrace();
          }

          System.out.println("Aceptando conexiones");

          while (serverSocket != null) {
               try {
                    Socket newSocket = serverSocket.accept();
```

EJEMPLO 4.13 (cont.)

```
                    System.out.println("Conexión recibida");

                    CalcServer hilo = new CalcServer(newSocket);
                    hilo.start();

               } catch (IOException e) {
                    e.printStackTrace();
               }
          }

          System.out.println("Terminado");
     }
}
```

EJEMPLO 4.14

Ejemplo de cliente para nuestro servidor de calculadora (Ejemplo 4.13):

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.net.Socket;


public class CalcClient {

     public static void main(String[] args) {
          try {

          System.out.println("Creando socket cliente");

          Socket clientSocket = new Socket();

          System.out.println("Estableciendo la conexión");

          InetSocketAddress addr = new InetSocketAddress("localhost", 5555);
          clientSocket.connect(addr);

          InputStream is = clientSocket.getInputStream();
          OutputStream os = clientSocket.getOutputStream();

          System.out.println("Enviando petición de suma");
```

EJEMPLO 4.14 (cont.)

```
            os.write("+".getBytes());

            System.out.println("Enviando primer operando");

            os.write(59);

            System.out.println("Enviando segundo operando");

            os.write(130);

            System.out.println("Recibiendo resultado");

            int result = is.read();

            System.out.println("Resultado de la suma: "+result);

            System.out.println("Cerrando el socket cliente");

            clientSocket.close();

            System.out.println("Terminado");

            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

ACTIVIDADES 4.4

- Observa los Ejemplos 4.13 y 4.14, con la implementación de la aplicación cliente/servidor de calculadora. A la vista del código del servidor, ¿cómo habría que modificar el cliente para que, a continuación, solicitase una operación de división (10/2, por ejemplo)? ¿Le basta al cliente con enviar nuevos mensajes o debe reiniciar la conexión? ¿Por qué?

4.3 PROTOCOLOS ESTÁNDAR DE NIVEL DE APLICACIÓN

En la actualidad existen multitud de aplicaciones distribuidas de uso generalizado. Las páginas web o el correo electrónico son ejemplos típicos de sistemas estándar que siguen el modelo cliente/servidor para organizar sus comunicaciones. En la mayoría de estos casos, el cliente y el servidor de cualquiera de estos sistemas son aplicaciones independientes, desarrolladas por diferentes personas/compañías y muchas veces ni siquiera implementadas usando el mismo lenguaje de programación. La definición exhaustiva de un protocolo de nivel de aplicación es fundamental en estos casos, para garantizar que clientes y servidores pueden comunicarse de manera efectiva. Para la mayoría de aplicaciones distribuidas comunes se han definido, a lo largo de los años, protocolos de nivel de aplicación estándar, que facilitan la tarea de desarrollar servidores y clientes para ellas. A continuación se estudiarán algunos de los protocolos estándar de nivel de aplicación más importantes.

¿SABÍAS QUE...?

La *World Wide Web* es probablemente el conjunto de aplicaciones distribuidas más usado en el mundo. Actualmente en Internet se pueden encontrar muchas aplicaciones servidoras de páginas web distintas, como Apache, Internet Information Service o Google Web Server. A su vez, hay una gran variedad de clientes web (comúnmente llamados "navegadores"), como Firefox, Opera, Safari, Internet Explorer, etc. Gracias al protocolo estándar de nivel de aplicación HTTP, cualquier servidor puede comunicarse con cualquier navegador.

¿SABÍAS QUE...?

Una interacción entre un cliente web y un servidor es un ejemplo típico de una aplicación distribuida en la que cada elemento ha sido desarrollado por una entidad distinta, usando lenguajes y tecnologías diferentes. La aplicación servidora más usada, Apache, está escrita en lenguaje C y ha sido desarrollada por el proyecto Apache. El navegador web Firefox, desarrollado por Mozilla, está escrito en C++ y Javascript UI. El navegador web de Microsoft, Internet Explorer, está escrito también en C++, aunque versiones anteriores fueron desarrolladas en Visual Basic.

4.3.1 TELNET

El **Telnet** es un protocolo de nivel de aplicación diseñado para proporcionar comunicación bidireccional basada en texto plano (caracteres ASCII) entre dos elementos de una red. Este protocolo simula una conexión virtual a una terminal de texto, como las antiguas terminales que se usaban para conectarse a los grandes ordenadores en los centros de datos. Telnet forma parte de la pila de protocolos IP, y utiliza el protocolo de nivel de transporte TCP (*sockets stream*) para intercambiar mensajes entre ambos extremos de la comunicación.

Tradicionalmente, el Telnet se ha usado para conectarse remotamente a máquinas, creando una sesión de línea de mandatos como las *Shell* de los sistemas operativos UNIX o el "símbolo de sistema" de Windows (C:\>). Usando un cliente Telnet, un usuario se puede conectar a una máquina que esté en otro punto de la red y hacer uso de ella de manera similar a si estuviese físicamente junto a ella. En una interacción típica por Telnet, el usuario escribe las órdenes usando su teclado, y el cliente las envía a la máquina destino como cadenas de texto por un *socket stream*. Al llegar al receptor, el servidor de Telnet recoge las cadenas de texto recibidas y las ejecuta como órdenes en la terminal. Posteriormente envía por el mismo *socket stream* la salida que ha producido la ejecución de la orden, para que el usuario pueda verla en su extremo. Telnet es un protocolo con estado (*stateful*). El número de puerto por defecto para servidores Telnet es el 23.

¿SABÍAS QUE...?

Para acceder al símbolo de sistema en Windows 7 basta con abrir el menú de inicio y seleccionar **Todos los programas**. Dentro de Accesorios hay un acceso directo a este.

Figura 4.2. *Símbolo de sistema en Windows 7*

El principal problema del protocolo Telnet es que las cadenas de texto se envían de forma clara por la red, sin ningún tipo de protección o cifrado. Esto hace que cualquier aplicación que monitorice las comunicaciones pueda leer perfectamente los mensajes que se intercambian cliente y servidor. Esto puede suponer un problema de seguridad grave, cuando se transfiere información sensible como contraseñas, etc. Por esta razón, el uso de este protocolo está desaconsejado en la actualidad, salvo en redes locales y entornos controlados en los que existen mecanismos de seguridad adicionales.

¿SABÍAS QUE...?

El protocolo de nivel de aplicación estándar Telnet fue definido en 1973 y ampliamente usado hasta la década de los 90, cuando fue sustituido por SSH (que se verá a continuación).

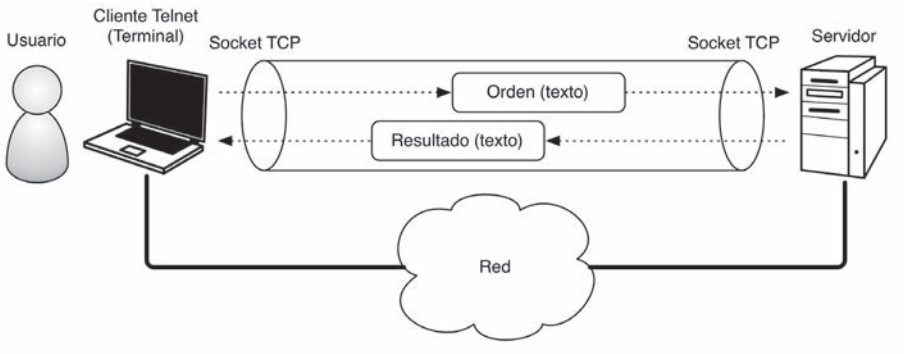

Figura 4.3. *Comunicación usando el protocolo de nivel de aplicación Telnet*

4.3.2 SSH (*SECURE SHELL*)

SSH es un protocolo de nivel de aplicación muy similar a Telnet. Sus características básicas y funcionamiento son esencialmente iguales, y fue diseñado con el mismo propósito. Se trata, en cambio, de un protocolo mucho más moderno (la primera versión es de 1995), que fue desarrollado precisamente para suplir las carencias de Telnet. La principal característica de SSH que lo diferencia de Telnet (y su principal virtud) es que la información transferida entre cliente y servidor está cifrada. Esto aumenta enormemente la seguridad de las comunicaciones ya que, aunque se monitorice el tráfico, no se puede acceder de forma directa a los mensajes, pues se envían codificados. SSH es el protocolo de nivel de aplicación recomendado en la actualidad para realizar sesiones de líneas de mandatos en máquinas remotas, especialmente si la comunicación se realiza a través de Internet u otra red no segura. SSH es un protocolo con estado (*stateful*). El número de puerto por defecto para servidores SSH es el 22.

ACTIVIDADES 4.5

- Busca en la Web más información sobre los protocolos Telnet y SSH. ¿Qué ventajas adicionales ofrece el segundo frente al primero?

4.3.3 FTP (*FILE TRANSFER PROTOCOL*)

FTP es un protocolo de nivel de aplicación diseñado para la transferencia de archivos a través de una red de comunicaciones. Al igual que Telnet y SSH, FTP utiliza el protocolo de transporte TCP (*sockets stream*) para establecer un canal de comunicación entre cliente y servidor y realizar la transferencia de datos. FTP establece un par de conexiones simultáneas entre cliente y servidor, que usa de forma distinta. La primera (conexión de control) se usa para enviar órdenes al servidor y obtener información. La segunda (conexión de datos) se utiliza para transferir los archivos. FTP permite tanto la descarga de archivos (del servidor al cliente) como la subida de estos (del cliente al servidor). Además, incluye mecanismos para listar y navegar por el árbol de directorios, cambiar las propiedades de los archivos (como el nombre o los permisos de acceso), etc. FTP es un protocolo con estado (*stateful*). El número de puerto por defecto para servidores FTP es el 21.

¿SABÍAS QUE...?

Al igual que ocurre con Telnet, el protocolo de nivel de aplicación FTP no está diseñado para ser seguro, y las comunicaciones se realizan sin protección. Esto puede causar toda clase de problemas de seguridad, sobre todo en redes públicas como Internet. Existe una extensión al protocolo, llamada FTPS (*FTP Secure*), que incorpora añadidos que lo hacen más seguro. Además existe otra alternativa segura, llamada SFTP. A diferencia de FTPS, SFTP no es una extensión de FTP, es un protocolo de transferencia de archivos distinto, basado a su vez en SSH.

ACTIVIDADES 4.6

- Uno de los clientes de FTP más usados en Windows es FileZilla (*https://filezilla-project.org/*). Descárgalo y úsalo para conectarte a un servidor de FTP. Comprueba que puedes navegar por los directorios, descargar archivos, etc.

4.3.4 HTTP (*HYPERTEXT TRANSFER PROTOCOL*)

HTTP es, probablemente, el protocolo de nivel de aplicación más importante de la actualidad. La mayoría del tráfico que se realiza en la *World Wide Web*, la parte más visible de la red Internet, utiliza este protocolo para controlar la transferencia de información entre las diferentes aplicaciones implicadas (fundamentalmente navegadores y servidores web).

¿SABÍAS QUE...?

La mayoría de la gente considera los términos "Internet" y "*World Wide Web*" (o simplemente "Web") como sinónimos. Esto se debe a que la gran mayoría de usuarios de la red Internet utiliza casi de forma exclusiva las páginas web para acceder a ella. Esto, por supuesto, es incorrecto. Internet es una red global de comunicaciones que utiliza la pila IP como arquitectura fundamental de comunicaciones. La web es el conjunto de aplicaciones distribuidas que, sobre Internet, proporcionan el servicio de acceso a documentos de *hipertexto*, comúnmente llamados "páginas web".

Según su definición, la función de este protocolo es facilitar la transmisión de documentos de *hipertexto* entre diferentes aplicaciones ubicadas dentro de una red. Un documento de *hipertexto* es un texto expresado en un código especial, diseñado para ser mostrado en un dispositivo electrónico como un ordenador o un *e-book*. El código usado por el documento de *hipertexto* suele contener mecanismos que permiten formatear su contenido de manera sofisticada, permitiendo organizarlo en columnas, insertar imágenes, etc. La característica más importante de un documento de *hipertexto* es que este contiene *hiperenlaces* (en inglés *hiperlinks*) a otros documentos. Un usuario que esté visualizando el documento debe poder acceder a estos *hiperenlaces* de forma sencilla, haciendo clic en ellos con el ratón, por ejemplo. Las páginas web son un caso típico de documentos de *hipertexto*.

¿SABÍAS QUE...?

El protocolo de nivel de aplicación HTTP se ocupa de la transferencia de documentos de *hipertexto*, pero no especifica cómo debe estar organizado el contenido dentro de ellos. En la Web se utiliza principalmente el lenguaje de marcas HTML (*HyperText Markup Language*) para codificar las páginas.

Como el resto de protocolos de nivel de aplicación estándar vistos hasta ahora, HTTP sigue el modelo de comunicaciones cliente/servidor, y gestiona el paso de mensajes por el mecanismo petición-respuesta. HTTP forma parte de la pila de protocolos IP y está diseñado para utilizar un protocolo de nivel de transporte fiable y orientado a conexión. Aunque normalmente funciona sobre TCP, no restringe su uso a este, y puede funcionar también sobre alternativas de transporte menos fiables, como UDP. El número de puerto por defecto para servidores HTTP es el 80.

¿SABÍAS QUE...?

Al igual que Telnet y FTP, el protocolo de nivel de aplicación HTTP no está diseñado para ser seguro, y la transferencia de información se realiza sin ningún tipo de protección. Existe una extensión de HTTP, denominada HTTPS (*HTTP Secure*), que incorpora mecanismos de cifrado de las comunicaciones. HTTPS es la base del comercio electrónico a través de la Web, ya que permite transferir información sensible (contraseñas, números de tarjeta de crédito, etc.) de manera confidencial.

A diferencia de otros protocolos vistos anteriormente, como Telnet y SSH, HTTP es un protocolo sin estado (*stateless*). No obstante, algunos servidores HTTP implementan mecanismos para almacenar el estado de la sesión. El método más usado para conseguir este efecto consiste en almacenar pequeños fragmentos de información en la aplicación cliente (normalmente el navegador web). A estos fragmentos de información se les conoce habitualmente como *cookies*.

4.3.4.1 Sesiones, recursos y peticiones HTTP

Se conoce como una **sesión HTTP** a una secuencia de intercambios petición-respuesta entre un cliente y un servidor. Normalmente, el cliente inicia una sesión estableciendo una conexión TCP (*socket stream*) con el puerto 80 del servidor. Una vez establecida la conexión, el servidor espera la llegada de peticiones HTTP por parte del cliente, y las responde. Las respuestas del servidor contienen información sobre el estado de realización de la petición (si se ha

podido llevar a cabo con éxito o no, etc.). Pueden además incluir contenido adicional (si se ha solicitado), como páginas web. Este contenido se almacena en lo que se conoce como *cuerpo* del mensaje (en inglés *message body*).

Un servidor HTTP organiza la información que contiene (fundamentalmente documentos de *hipertexto*, es decir, páginas web) usando *recursos*. Un recurso puede ser cualquier documento que contenga el servidor y suele estar identificado por una clave única, que se usa en los *hiperenlaces* que se refieren a él. En la Web, estas claves se llaman *Uniform Resource Locators*[7] o URL, comúnmente conocidas como *direcciones web*.

HTTP define toda una serie de peticiones que los clientes pueden realizar durante una sesión. En la versión 1.1 de HTTP (la utilizada actualmente en la Web) son las siguientes:

- **GET**: se utiliza para solicitar recursos, como páginas web identificadas por su URL. Esta es la primera petición que suele realizar un navegador web cuando se conecta a un servidor.
- **HEAD**: similar a GET, pero la respuesta nunca contiene cuerpo del mensaje. Esta petición no permite obtener recursos, pero es útil para obtener información sobre el estado del servidor, sin tener que descargar un documento completo.
- **POST**: se utiliza para solicitar al servidor la incorporación de nuevo contenido a un recurso existente, identificado por su URL. Esta petición se usa de forma habitual en páginas web que permiten a los usuarios agregar contenido, como foros o las secciones de comentarios de muchos artículos en periódicos *on-line* y blogs.
- **PUT**: se utiliza para solicitar la incorporación de un nuevo recurso al servidor, identificado por una nueva URL (proporcionada como parte de la petición). Si la URL ya existe en el servidor, este sustituirá el recurso existente por el nuevo enviado en el PUT.
- **DELETE**: se utiliza para solicitar la eliminación de un recurso.
- **OPTIONS**: se utiliza para solicitar una lista de las peticiones (GET, PUT, POST, etc.) que el servidor acepta sobre una determinada URL.
- **TRACE**: se utiliza para solicitar al servidor que le devuelva la petición que acaba de recibir del mismo cliente, a modo de eco. Esto es útil para detectar posibles modificaciones de la petición realizadas por elementos intermedios de la comunicación entre el cliente y el servidor.
- **CONNECT**: se utiliza para convertir la conexión entre el cliente y servidor en un túnel TCP/IP. Esto facilita la transmisión de datos cifrados.
- **PATCH**: se utiliza para realizar modificaciones parciales a un recurso.

No todos los servidores HTTP están obligados a aceptar todas las posibles peticiones. Dependiendo del recurso, determinadas peticiones pueden no ser aceptables, como POST o DELETE en páginas web que no se pueden modificar. Para poder operar correctamente, un servidor debería ser capaz al menos de aceptar peticiones GET y HEAD. También se recomienda que acepte la operación OPTIONS, cuando sea posible.

7 Se podría traducir como "localizadores uniformes de recursos".

ACTIVIDADES 4.7

- Busca información sobre la estructura de las peticiones HTTP. ¿Cómo es una cabecera HTTP? ¿Qué formato tiene una petición GET? ¿Se pueden leer de manera más o menos cómoda?

4.3.4.2 Códigos de estado

Como ya se ha explicado, todas las respuestas HTTP contienen información sobre el estado de realización de la petición. La primera línea de texto del mensaje de respuesta, llamada *línea de estado* (en inglés *status line*) contiene un código numérico de estado y una pequeña frase explicativa. El estándar HTTP en su versión 1.1 define multitud de códigos de estado, abarcando todas las posibles respuestas a las diferentes peticiones. Estos códigos son normalmente números de tres dígitos, y se dividen en 5 categorías:

- **Información**: son los códigos que empiezan por 1, y se usan para informar al cliente de aspectos diversos. El código 100 (*continue*), por ejemplo, se utiliza para solicitar al cliente que continúe enviando información, como parte de una petición de POST, por ejemplo.
- **Éxito**: son los códigos que empiezan por 2, y se usan para indicar al cliente que su petición ha sido recibida y procesada correctamente. Ejemplos típicos de esta categoría son el 200 (*OK*), el mensaje de éxito estándar, o el 202 (*accepted*), que indica que la petición ha sido recibida y aceptada, pero aún no ha sido procesada.
- **Redirección**: son los códigos que empiezan por 3, y se usan para indicar al cliente que debe realizar operaciones adicionales para completar su petición. Un ejemplo es el código 303 (*see other*), que se utiliza para redirigir al cliente a una nueva URL.
- **Error del cliente**: son los códigos que empiezan por 4, y se usan para indicar al cliente que ha cometido un error. Ejemplos típicos de esta categoría son el código 400 (*bad request*), que indica que la petición es incorrecta, o el 403 (*forbidden*), que indica que la petición no está permitida (se ha intentado hacer un POST sobre un recurso de solo lectura, por ejemplo). La mayoría de códigos de estado del estándar HTTP 1.1 pertenecen a esta categoría.
- **Error del servidor**: son los códigos que empiezan por 5, y se usan para indicar al cliente que el servidor ha experimentado un error y no puede completar la petición. Ejemplos de esta categoría son el 500 (*internal server error*) y el 503 (*service unavailable*).

¿SABÍAS QUE...?

El código de estado HTTP más reconocido por la mayor parte de usuarios de la Web es el 404 (*not found*). Este código indica al cliente que el recurso solicitado no se encuentra en el servidor, y en la mayoría de situaciones se origina por el uso de un *hiperenlace* antiguo, que contiene una URL que ya no existe. Prácticamente todas las personas que usan la Web de manera frecuente se han encontrado en algún momento con un mensaje que contenía este código de estado.

ACTIVIDADES 4.8

- Busca en la Web más información sobre el protocolo HTTP. ¿Existe (o ha existido) alguna vez alguna alternativa a este para la transferencia de páginas web?

4.3.5 POP3 (*POST OFFICE PROTOCOL*, VERSIÓN 3)

El protocolo de nivel de aplicación POP está diseñado para que las aplicaciones clientes de *e-mail*, como Thuderbird o Outlook, accedan los mensajes alojados en los servidores de correo electrónico. La mayoría de servidores comerciales de *e-mail*, como el GMail de Google o el Hotmail de Microsoft, soportan este protocolo. La versión actual y más extendida de POP es la 3, conocida como POP3.

POP3 se basa en el protocolo de transporte TCP (*sockets stream*) y proporciona peticiones básicas para acceso, descarga y borrado de mensajes. POP3 es un protocolo sin estado (*stateless*). El número de puerto por defecto para servidores POP3 es el 110.

¿SABÍAS QUE...?

POP3 es uno de los dos protocolos de nivel de aplicación estándar para clientes de correo electrónico. La alternativa a POP3 es IMAP (*Internet Message Access Protocol*). En muchos aspectos, IMAP es un protocolo mucho más sofisticado que POP3, aunque sus funciones principales son parecidas. Todos los clientes y servidores de correo electrónico comúnmente usados en la actualidad soportan ambos protocolos.

ACTIVIDADES 4.9

- Busca información sobre el protocolo IMAP ¿Qué características adicionales ofrece frente a POP3?
- Considera dos posibles escenarios: 1) acceso al correo electrónico desde el interior de una red corporativa, y 2) acceso al correo electrónico a través de Internet. ¿Qué protocolo de acceso (POP3 o IMAP) te parece más adecuado en cada caso?
- Hoy en día, la mayoría de proveedores de *e-mail*, como GMail o Hotmail, proporcionan aplicaciones para acceder al correo electrónico a través de una página web (cliente de correo web). En un contexto como este, ¿qué ventajas aporta usar un cliente de correo convencional, que implemente el protocolo POP3?
- Busca en la página web de tu proveedor de correo (GMail, Hotmail, etc.) cómo habilitar el acceso POP3 desde un cliente como Thunderbird. Descarga Thunderbird y configúralo para acceder a tu correo por POP3.

4.3.6 SMTP (*SIMPLE MAIL TRANSFER PROTOCOL*)

SMTP es el protocolo de nivel de aplicación estándar para el envío de mensajes de correo electrónico en Internet. Todos los *e-mail* que circulan por la red lo hacen gracias a este protocolo. Además, SMTP es el protocolo que usan los clientes de correo electrónico, como Thunderbird o Outlook, para entregar los mensajes que se desea enviar a los servidores que proporcionan el servicio de correo, como GMail o Yahoo! Mail.

SMTP se basa en el protocolo de transporte TCP (*sockets stream*) y proporciona peticiones para el envío de mensajes de correo electrónico entre un emisor y un receptor. SMTP es un protocolo sin estado (*stateless*). El número de puerto por defecto para servidores SMTP es el 587.

¿SABÍAS QUE...?

Ninguno de los protocolos que se usan habitualmente para la transferencia de correos electrónicos a través de Internet (POP3, SMTP, etc.) incorpora mecanismos de seguridad como el cifrado de las comunicaciones. En el caso del protocolo POP3, existe una extensión a este denominada STARTTLS, que permite cifrar las comunicaciones entre cliente y servidor. El caso de SMTP es mucho más complejo, ya que incorporar mecanismos de seguridad implicaría sustituir todos los servidores de correo electrónico del mundo para que todo el tráfico de mensajes por la red fuese seguro. En la práctica esto no ha ocurrido, por lo que los mensajes se envían casi siempre en claro a través de Internet. Esto es muy importante, ya que quiere decir que **el correo electrónico es un método de comunicación no seguro**, y nunca se debería usar para enviar información confidencial como números de tarjeta de crédito, etc. Si un usuario desea que sus correos electrónicos vayan cifrados, debe realizar esta tarea por sí mismo y asegurarse de que los destinatarios de dichos mensajes tengan los medios para descifrarlos.

4.3.7 OTROS PROTOCOLOS DE NIVEL DE APLICACIÓN IMPORTANTES

Además de los vistos hasta ahora, existen muchos otros protocolos de nivel de aplicación ampliamente utilizados por las aplicaciones distribuidas actuales. Algunos ejemplos de estos protocolos son:

- **DHCP** (*Dynamic Host Configuration Protocol*): es un protocolo que se usa para la configuración dinámica de dispositivos dentro de una red. Su uso más habitual es la asignación dinámica de direcciones IP.
- **DNS** (*Domain Name System*): es un protocolo que se utiliza para la resolución de nombres simbólicos de máquinas en la red. Usando DNS se puede obtener la dirección IP de una máquina, a partir de su nombre.
- **NTP** (*Network Time Protocol*): se trata de un protocolo diseñado para la sincronización de relojes entre máquinas de una red.
- **TLS** (*Transport Layer Security*): es un protocolo diseñado para incorporar características criptográficas a los protocolos de nivel de transporte, como TCP. TLS y su predecesor, SSL (*Secure Sockets Layer*) son la base de la mayoría de extensiones de seguridad de muchos protocolos de nivel de aplicación como el FTPS o el HTTPS.

¿SABÍAS QUE...?

En la mayoría de sistemas operativos (Windows, Linux, Mac OS X, etc.) existe una herramienta llamada *nslookup* que sirve para realizar consultas al sistema de resolución de nombre simbólicos DNS. Por ejemplo, en Mac OS X podemos resolver el nombre simbólico *www.google.es* de la siguiente forma:

```
$ nslookup www.google.es

Server:  8.8.8.8
Address: 8.8.8.8#53

Non-authoritative answer:
Name:    www.google.es
Address: 173.194.41.223
Name:    www.google.es
Address: 173.194.41.215
Name:    www.google.es
Address: 173.194.41.216
```

ACTIVIDADES 4.10

- Busca en la Web información sobre el protocolo NTP. ¿En qué se basa? ¿Cuál es su principal uso?
- DHCP es un protocolo que se usa mucho en redes de área local, para la asignación de direcciones IP locales. ¿Es este el único uso que se hace de este protocolo? Busca información sobre él en la Web e identifica sus funciones principales.

4.4 TÉCNICAS AVANZADAS DE PROGRAMACIÓN DE APLICACIONES DISTRIBUIDAS

Hasta ahora se han visto los fundamentos del desarrollo de servicios en red, las técnicas básicas de implementación de estos y algunos de los más importantes protocolos de nivel de aplicación de la actualidad. Todos los servicios y protocolos de nivel de aplicación vistos se basan en la pila IP para llevar a cabo las comunicaciones y utilizan protocolos de transporte como TCP y UDP para gestionar el envío de sus mensajes. En el capítulo anterior se ha visto en detalle la utilización de *sockets stream* y *sockets datagram*, que son las herramientas básicas para hacer uso de dichos protocolos. No obstante, desde el punto de vista del desarrollo de aplicaciones de alto nivel, las interfaces de programación que normalmente se usan para operar con *sockets* (como la que ofrece Java que se estudió en el capítulo previo) resultan poco prácticas, ya que requieren el formateo completo de los mensajes y el control detallado de las comunicaciones.

Existen multitud de alternativas al uso directo de *sockets* a la hora de programar aplicaciones distribuidas. Cada una de estas alternativas ofrece ventajas distintas, por lo que se deben analizar y considerar en detalle antes de empezar a desarrollar una aplicación de alto nivel. La mayoría de estas tecnologías avanzadas de comunicación son, en realidad, una capa extra de software que se coloca entre el nivel de transporte y el nivel de aplicación de la pila de protocolos. Por debajo, la comunicación sigue ocurriendo mediante *sockets stream* o *sockets datagram*, pero su uso se abstrae al programador para facilitar su trabajo y proporcionar funcionalidades de comunicación complejas de manera cómoda y eficiente.

4.4.1 INVOCACIÓN DE MÉTODOS REMOTOS

La primera de las técnicas avanzadas de comunicación que se van a estudiar es la invocación de métodos remotos. Esta tecnología se basa en la idea de que en un programa orientado a objetos la invocación de métodos es, desde un punto de vista teórico, un proceso de comunicación. Dicho de otra forma, cuando en un programa orientado a objetos el objeto A invoca un método del objeto B, esto se puede analizar como si el objeto A enviase un mensaje a B, B realizase una serie de operaciones, y, finalmente, devolviese el resultado a A mediante otro mensaje. Una invocación de un método remoto implica que el objeto B se encuentra en un lugar distinto de la red que el objeto A y, por lo tanto, para poder invocar su método debe ocurrir un intercambio de mensajes a través de la red. En el lenguaje Java, la invocación a métodos remotos se conoce por sus siglas en inglés: **RMI** (*Remote Method Invocation*).

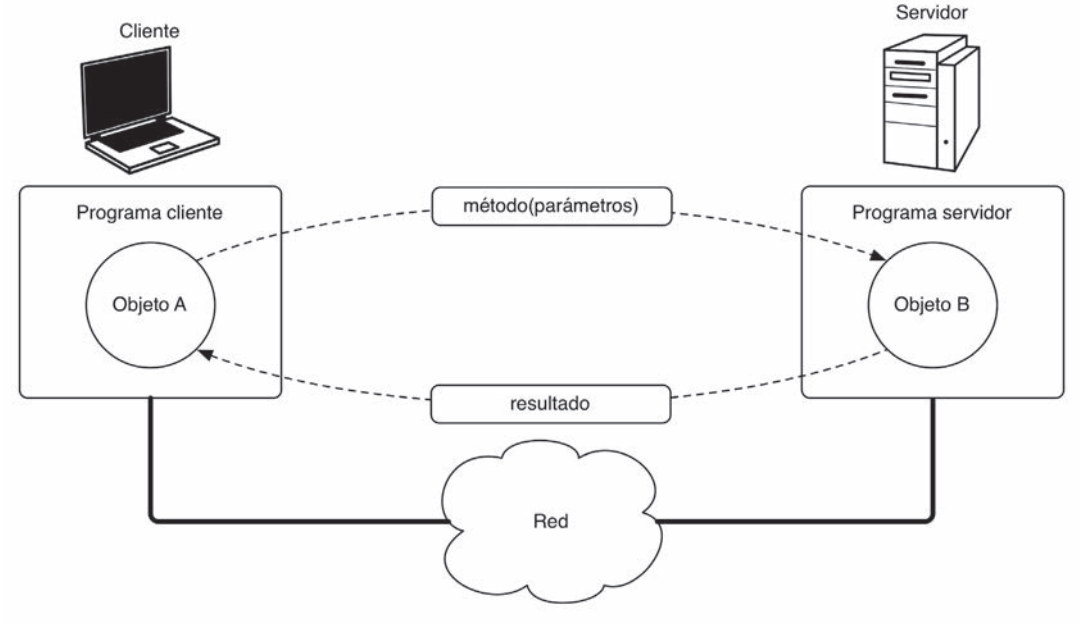

Figura 4.4. *Ejemplo de invocación de un método remoto*

¿SABÍAS QUE...?

Existe otra técnica de comunicación entre aplicaciones muy similar a la invocación de métodos remotos, denominada *llamada a procedimiento remoto* o RPC (en inglés *Remote Procedure Call*). Las RMI y las RPC son tecnologías muy parecidas. La diferencia clave entre ellas es el paradigma de programación en el que se basan. RMI está basado en la programación orientada a objetos, y RPC en la programación estructurada, y de ahí la diferencia de nomenclatura.

En una invocación a un método remoto existen cuatro componentes fundamentales: **objeto servidor**, **objeto cliente**, **método invocado** y **valor de retorno**.

4.4.1.1 Objeto servidor

Se trata del objeto cuyo método es invocado. Se llama **objeto servidor** porque es el que recibe la petición (llamada al método) y la procesa. Al objeto servidor se le llama habitualmente también **objeto remoto**, ya que es al que se accede de forma remota.

4.4.1.2 Objeto cliente

Se trata del objeto que invoca el método. Para ello, envía una petición al objeto servidor a través de la red, indicando el método invocado y sus parámetros. Una vez finalizada la invocación recibe un mensaje del objeto servidor con el resultado obtenido.

4.4.1.3 Método invocado

Para que se pueda realizar la invocación del método, este y sus parámetros se convierten en un mensaje, que se envía por la red al objeto servidor, a modo de petición.

4.4.1.4 Valor del retorno

Una vez se ha concluido la ejecución del método, el resultado de este se convierte en un mensaje que se envía al objeto cliente.

4.4.1.5 Infraestructura de comunicaciones para la invocación de métodos remotos

Para poder llevar a cabo la invocación de métodos remotos se necesita una capa de software adicional entre la aplicación de alto nivel y el nivel de transporte de la pila IP. La función de esta capa es traducir las llamadas a métodos remotos y valores de retorno de estos a mensajes que se envíen por *sockets* y, de forma simétrica, traducir los mensajes recibidos a valores de retorno y llamadas a métodos. Esa capa de software debe incluir además mecanismos para poder crear objetos remotos y localizarlos desde otros puntos de la red, de forma que se pueda operar con ellos desde las aplicaciones de alto nivel. El objetivo final de esta herramienta es que la programación de aplicaciones usando invocación de métodos remotos sea lo más parecida posible al desarrollo de aplicaciones sin comunicaciones por red.

Los componentes básicos de esta capa de software son:

- **Stubs**: los *stubs* son objetos que sustituyen a los objetos remotos dentro del programa donde se encuentra el objeto cliente. Tienen métodos similares al objeto servidor, pero no realizan las mismas funciones. En su lugar, cuando *stub* recibe la invocación de un método, lo que hace es construir un mensaje y enviarlo por la red al objeto servidor. Una vez hecho esto, espera hasta que el objeto remoto responda. Cuando recibe respuesta la devuelve, como si él mismo hubiese realizado la operación. De esta forma, el objeto cliente no tiene que realizar ninguna operación especial para invocar un método remoto. Tan solo invocar el método del objeto *stub*.
- **Registro de objetos remotos**: el registro de objetos remotos es un servicio de nivel de aplicación cuya función consiste en controlar todos los objetos remotos que existen en el sistema. Cuando un programa crea un objeto remoto, el primer paso que debe realizar es apuntarlo en el registro. De esta forma, cuando un objeto cliente necesite contactar con un objeto servidor, lo buscará en el registro. En el momento en que un objeto *stub* quiera configurarse para conectarse a un objeto remoto, el registro le dará la información necesaria.

ACTIVIDADES 4.11

- Como se ha explicado, la invocación de métodos remotos (RMI) está íntimamente relacionada con la llamada a procedimientos remotos (RPC). Busca en la Web información sobre la segunda (RPC) e identifica los elementos que intervienen en la comunicación en esta tecnología. ¿Qué diferencias observas con respecto a RMI? ¿Existe alguna correspondencia entre los componentes básicos de RMI y RPC?

4.4.1.6 Invocación de métodos remotos: proceso detallado

Para poder realizar invocación de métodos remotos se debe seguir una serie de pasos. La secuencia de pasos completa (incluyendo servidor y cliente) es:

1. (En el servidor) Arranque del registro de objetos remotos. Si no está arrancado ya, este debe ser siempre el primer paso.
2. (En el servidor) Creación del objeto servidor. El objeto se crea, como cualquier otro.
3. (En el servidor) Inscripción del objeto en el registro de objetos remotos. Esto se suele hacer indicando un nombre identificador para el objeto. Al realizar esta operación, el objeto servidor se convierte en un objeto remoto.
4. (En el cliente) Localización del objeto remoto en el registro. Para esto se debe conocer la dirección y el puerto del registro. El cliente debe conectarse a él y solicitar la información de conexión con el objeto servidor. Este proceso suele incluir directamente la creación del objeto *stub*.
5. (En el cliente) Invocación de métodos del objeto *stub*. Desde el punto de vista del programa cliente, estas son llamadas normales, pero dentro de ellas se produce la comunicación con el objeto remoto.
6. (Entre servidor y cliente) Intercambio de mensajes entre *stub* y objeto servidor.

7 (En el cliente) Obtención del valor de retorno de la invocación al método remoto.

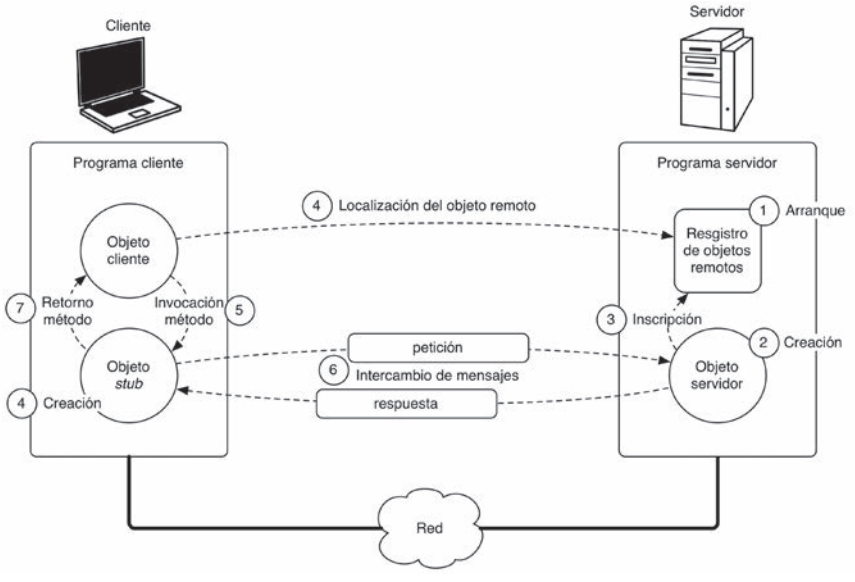

Figura 4.5. *Proceso detallado de invocación de un método remoto*

4.4.1.7 Programación de aplicaciones cliente/servidor basadas en invocación de métodos remotos

La biblioteca estándar de Java proporciona una serie de clases predefinidas que sirven para desarrollar aplicaciones distribuidas basadas en el modelo de invocación a métodos remotos (RMI). Estas clases se encuentran dentro del paquete *java.rmi*, y a continuación se verá cómo usar las más importantes.

Para programar una aplicación cliente/servidor usando Java RMI, se deben seguir los mismos pasos que para cualquier otra aplicación de este tipo:

1 **Definición de las funciones servidor**. Esto se hace de forma similar a como ya se ha visto, determinando sus características fundamentales y propiedades del servicio proporcionado.

2 **Selección de la tecnología de comunicaciones**. En este caso, esta será la invocación de métodos remotos (RMI).

3 **Definición del protocolo de nivel de aplicación**. En este caso, los mensajes que se intercambian entre cliente y servidor van englobados dentro de las llamadas a métodos remotos. Por tanto, esta tarea implica la especificación de dichos métodos. En Java RMI esto significa codificar la interfaz que deberá implementar la clase del objeto remoto. A esto se le llama la **interfaz remota**.

4 **Implementación de la clase del objeto servidor**, incluyendo la implementación de los métodos remotos.

5 **Implementación de la clase del objeto cliente**.

4.4.1.7.1 Interfaz remota

La interfaz remota es una interfaz Java que contiene los métodos que el cliente invocará de forma remota. Al ser una interfaz y no una clase, los métodos no están implementados, solo definidos. Esta interfaz debe extender a su vez de la interfaz Java *Remote* (*java.rmi.Remote*). Además, los métodos remotos definidos deben lanzar la excepción *RemoteException* (*java.rmi.RemoteException*).

EJEMPLO 4.15

Se va a retomar el Ejemplo 4.6, en el que se planteaba el desarrollo de una aplicación cliente/servidor con funciones de calculadora sencilla. En este caso, se va implementar una aplicación de las mismas características, pero usando ahora Java RMI como tecnología de comunicaciones. Como se ha visto, el primer paso es implementar la interfaz remota del objeto servidor:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RMICalcInterface extends Remote {

 public int suma(int a, int b) throws RemoteException;
 public int resta(int a, int b) throws RemoteException;
 public int multip(int a, int b) throws RemoteException;
 public int div(int a, int b) throws RemoteException;
}
```

ACTIVIDADES 4.12

- Copia el Ejemplo 4.15 en un editor de texto y modifícalo para que la interfaz opere con números reales en vez de enteros. Agrega además a la interfaz la operación potencia (a^b).

4.4.1.7.2 Implementación del objeto servidor

Una vez se dispone de la interfaz remota, se debe implementar la clase del objeto servidor. Este debe incluir una implementación de todos los métodos definidos en la interfaz. Además, ya sea en la misma clase o en otra, se deben implementar los mecanismos para poner en funcionamiento el registro de objetos remotos e inscribir el objeto servidor en él.

EJEMPLO 4.16

Continuando con el ejemplo de la aplicación calculadora, este es el código del servidor:

```
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
```

EJEMPLO 4.16 (cont.)

```
public class RMICalcServer implements RMICalcInterface {

    public int suma(int a, int b) throws RemoteException {
         System.out.println("Objeto remoto: Sumando "+a+" y "+b+"...");
         return (a + b);
    }

    public int resta(int a, int b) throws RemoteException {
         System.out.println("Objeto remoto: Restando "+a+" y "+b+"...");
         return (a - b);
    }

    public int multip(int a, int b) throws RemoteException {
         System.out.println("Objeto remoto: Multiplicando "+a+" por "+b+"...");
         return (a * b);
    }

    public int div(int a, int b) throws RemoteException {
         System.out.println("Objeto remoto: Dividiendo "+a+" entre "+b+"...");
         return (a / b);
    }

    public static void main(String[] args) {

         System.out.println("Creando el registro de objetos remotos...");

         Registry reg = null;
         try {
              reg = LocateRegistry.createRegistry(5555);
         } catch (Exception e) {
              System.out.println("ERROR: No se ha podido crear el registro");
              e.printStackTrace();
         }

         System.out.println("Creando el objeto servidor e inscribiéndolo en el registro...");
         RMICalcServer serverObject = new RMICalcServer();

         try {
              reg.rebind("Calculadora",
              (RMICalcInterface) UnicastRemoteObject.exportObject(serverObject, 0));
         } catch (Exception e) {
              System.out.println("ERROR: No se ha podido inscribir el objeto servidor.");
              e.printStackTrace();
         }
    }
}
```

El Ejemplo 4.16 muestra un código típico de un programa servidor usando Java RMI. En este ejemplo se debe prestar atención a los siguientes aspectos:

- La clase del objeto remoto implementa la interfaz remota. Esto hace que implemente todos los métodos definidos en dicha interfaz, que son la representación del protocolo de nivel de aplicación.
- El método *main* crea el registro de objetos, escuchando por el puerto 5555. Como ya se ha explicado, el registro de objetos remotos es un servicio que se usa para localizar y gestionar objetos remotos. Las clases *java.rmi.registry.Registry* y *java.rmi.registry.LocateRegistry* permiten realizar tareas como crear un registro, localizar un registro existente, etc., desde dentro del programa.
- Una vez arrancado el registro, el método *main* crea un objeto servidor y lo inscribe en este. Para ello hace uso de la clase *java.rmi.server.UnicastRemoteObject*, que sirve para representar objetos remotos y operar con ellos. El método *rebind()* de la clase *Registry* permite realizar esta inscripción. Cuando se inscribe un objeto remoto en el registro, se le debe dar un nombre único para identificarlo. En el Ejemplo 4.16 este nombre es "Calculadora".

ACTIVIDADES 4.13

- Copia el Ejemplo 4.16 en un editor de texto y modifícalo para que implemente la interfaz definida en la Actividad 4.8, incluyendo la operación con números reales y el método potencia.

4.4.1.8 Implementación del objeto cliente

Por ultimo está la clase del objeto cliente, que contiene la localización del objeto remoto y la invocación de sus métodos.

EJEMPLO 4.17

Continuando con el ejemplo de la aplicación calculadora, este es el código del cliente:

```
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class RMICalcClient {

    public static void main(String[] args) {

        RMICalcInterface calc = null;
        try {
            System.out.println("Localizando registro de objetos remotos...");
            Registry registry = LocateRegistry.getRegistry("localhost", 5555);
```

EJEMPLO 4.17 (cont.)

```
                System.out.println("Obteniendo el stub del objeto remoto...");
                calc = (RMICalcInterface) registry.lookup("Calculadora");

            } catch (Exception e) {
                e.printStackTrace();
            }

            if (calc != null) {
                System.out.println("Realizando operaciones con el objeto remoto...");

                try {
                    System.out.println("2 + 2 = " + calc.suma(2, 2));
                    System.out.println("99 - 45 = " + calc.resta(99, 45));
                    System.out.println("125 * 3 = " + calc.multip(125, 3));
                    System.out.println("1250 / 5 = " + calc.div(1250, 5));
                } catch (RemoteException e) {
                    e.printStackTrace();
                }
                System.out.println("Terminado");
            }
        }
    }
```

De forma análoga al caso anterior, el Ejemplo 4.17 muestra un código típico de un programa cliente usando Java RMI. En este ejemplo se debe prestar atención a los siguientes aspectos:

- El método *main* hace uso de las clases *java.rmi.registry.Registry* y *java.rmi.registry.LocateRegistry* para localizar el registro de objetos remotos. La llamada al método *lookup()* de la clase *Registry* devuelve el objeto *stub* que el cliente utilizará para invocar remotamente a los métodos del objeto servidor.
- Para localizar el objeto remoto utiliza los datos de conexión del registro (*host*/dirección IP y número de puerto) y el nombre con el que se inscribió el objeto servidor en el registro (en el Ejemplo 4.17 ese nombre es "Calculadora").
- Una vez obtenido el *stub*, el cliente invoca los métodos remotos como si se tratase de métodos en un objeto local.

ACTIVIDADES 4.14

- Extiende el Ejemplo 4.17 para que utilice el método remoto potencia definido en las actividades anteriores y opere con números reales en lugar de enteros.

4.4.1.9 Invocación de métodos remotos y otros modelos de comunicaciones

Hasta ahora se ha visto el uso de la invocación de métodos remotos dentro del marco de las aplicaciones que siguen el modelo cliente/servidor. Esto no es solo debido al hecho de que el modelo cliente/servidor sea el más usado. Debido a la forma en que está diseñada, esta tecnología de comunicaciones resulta especialmente idónea para este modelo.

No obstante, es importante recordar que la invocación de métodos remotos en el fondo no es más que intercambio de mensajes, pero abstraído dentro de la interacción típica entre los objetos de un programa. Nada impide utilizar esta tecnología para implementar modelos de comunicación distintos, como la comunicación en grupo, o enfoques avanzados o híbridos.

ACTIVIDADES 4.15

- ¿Cómo implementarías el modelo de comunicaciones en grupo usando invocación de métodos remotos? ¿Qué ventajas e inconvenientes podría haber?

4.4.2 SERVICIOS WEB

Una de las técnicas de comunicaciones avanzadas más usadas de la actualidad son los llamados *servicios web*. Un servicio web es una aplicación distribuida, basada en el modelo de comunicaciones cliente/servidor, que utiliza el protocolo de nivel de aplicación HTTP para la transferencia de mensajes. El uso de este protocolo, el que se usa en la Web como ya se ha explicado, facilita la interoperabilidad entre clientes y servidores.

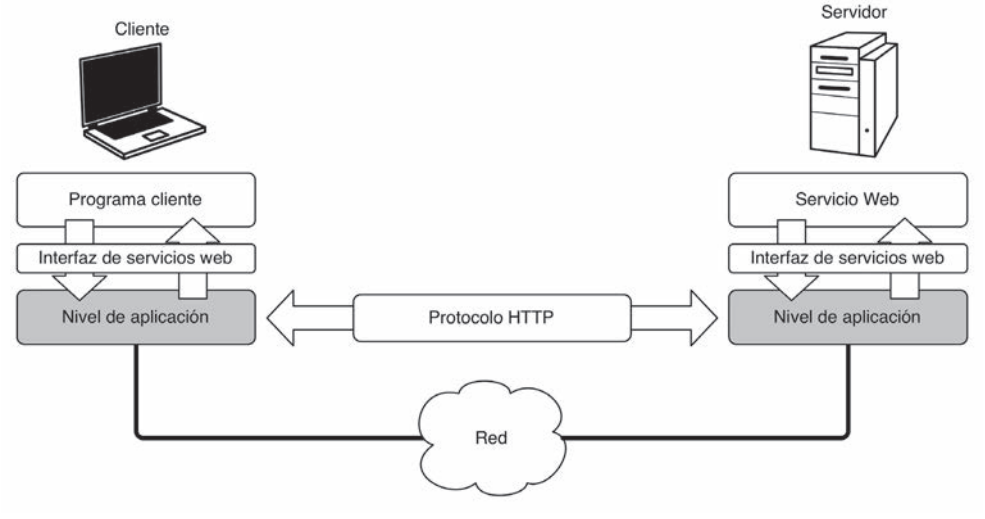

Figura 4.6. *Comunicación con servicios web*

¿SABÍAS QUE...?

Debido a la cantidad de amenazas de seguridad que existen en la red Internet, la mayoría de proveedores de acceso y empresas encargadas de gestionar el tráfico a través de la Red implementan políticas de control que restringen los protocolos de comunicaciones permitidos en determinados entornos. El protocolo HTTP es el que se utiliza para el tráfico web, y raramente se ve restringido. Los servicios web se aprovechan de esto, lo que los vuelve más accesibles y fiables.

El hecho de que los servicios web utilicen HTTP para enviar y recibir mensajes implica que los mensajes del protocolo de nivel de aplicación de un servicio web están, a su vez, encapsulados dentro de peticiones y respuestas HTTP. Dicho de otro modo, cuando un cliente de un servicio web realiza una petición a un servidor, esta se envía en el interior de un mensaje HTTP, como un GET, POST, o similar. A su vez, debemos recordar que HTTP utiliza habitualmente TCP como protocolo de transporte, por lo que estos mensajes irán también dentro de un mensaje TCP. Existen dos tipos de servicios web fundamentales, clasificados según sus características principales y los mecanismos que utilizan para representar los mensajes del protocolo de nivel de aplicación: **servicios web SOAP** y **servicios web REST**.

4.4.2.1 Servicios web SOAP

El primer tipo de servicios web que existieron fueron los llamados “servicios web SOAP”. Reciben este nombre debido al formato en que representan los mensajes, que sigue el estándar SOAP (en inglés *Simple Object Access Protocol*). En SOAP se utiliza el lenguaje XML para definir tanto el protocolo de mensajes como el contenido de estos. Esto hace que los mensajes sean fácilmente procesables de forma automática, a la vez que permite una alta flexibilidad a la hora de diseñarlos. La facilidad de procesamiento de los mensajes XML facilita además el desarrollo separado de servidores y clientes, y la fácil adaptación de las aplicaciones existentes, independientemente del lenguaje de programación en el que estuviesen desarrolladas. La descripción de interfaz de servicio de un servicio web SOAP se realiza usando un lenguaje basado en XML llamado WDSL (en inglés *Web Services Description Language*).

Un servicio web basado en SOAP debe cumplir los siguientes requisitos:

- Se debe expresar de manera formal y pública la interfaz del servicio. Esto se suele hacer usando WSDL.
- La arquitectura del servicio debe ser capaz de soportar realización y procesamiento de peticiones de forma asíncrona.
- Los servicios web basados en SOAP pueden o no tener estado (*stateful* o *stateless*).

La biblioteca de Java incluye un paquete de clases especiales para el desarrollo de servicios web basados en SOAP, denominado JAX-WS.

¿SABÍAS QUE...?

Como se ha explicado, SOAP establece que los mensajes entre cliente y servidor se codifiquen en el lenguaje XML. Esta decisión incorpora numerosas ventajas, como facilitar el procesamiento automático, la interoperabilidad, etc. No obstante, esto fuerza a que los mensajes se envíen siempre como documentos de texto (documentos XML), lo que puede suponer un mayor consumo de los recursos de red. Por esta razón, los servicios web basados en SOAP no están aconsejados para proporcionar servicios que requieran de la transferencia de una gran cantidad de información, como acceso a ficheros remotos.

4.4.2.2 Servicios web REST

Como alternativa a los servicios basados en SOAP, existen los servicios web REST. Estos servicios web no siguen el estándar SOAP, y por tanto no están forzados a utilizar XML para representar sus mensajes y su interfaz. En su lugar, utilizan REST (en inglés *Representational State Transfer*), un formato mucho más ligero y flexible, aunque sin las ventajas de interoperabilidad y facilidad en el procesamiento de XML.

ACTIVIDADES 4.16

- Busca en la Web información sobre el estándar WSRF (*Web Services Resource Framework*). ¿Qué relación tiene con los servicios web basados en SOAP?
- Busca información sobre el formato XML de los mensajes en SOAP. ¿Qué estructura tiene? ¿Resulta fácil de leer para una persona?
- Busca en la Web la descripción del lenguaje WSDL. ¿Qué elementos de un servicio web se definen usándolo?
- Busca en la Web más información sobre los servicios REST. ¿Por qué surgieron? ¿En qué situaciones se recomienda su uso frente a los servicios basados en SOAP?

4.5 CASO PRÁCTICO

Se desea programar una aplicación distribuida de buzón, siguiendo el modelo cliente/servidor. La idea consiste en que los clientes se conecten a un servidor, indicando su nombre. Una vez indicado su nombre, el cliente podrá consultar si tienen mensajes para él/ella, o dejar mensajes para otros usuarios. El servidor almacenará los mensajes entregados hasta que el usuario al que van dirigidos los consulte. Se pide:

- Especificar las funciones del servidor.
- Seleccionar la tecnología de comunicación adecuada (*sockets stream*, *sockets datagram*, llamada a métodos remotos, etc.).
- Diseñar el protocolo de nivel de aplicación.

- Implementar en Java el servidor.
- Implementar en Java un programa cliente que sirva para que diferentes usuarios consulten sus mensajes y dejen mensajes nuevos en el buzón.

Para simplificar el problema, se podrá asumir que los usuarios se identifican solo usando su nombre. No es necesario implementar gestión de contraseñas u otros mecanismos de seguridad.

RESUMEN DEL CAPÍTULO

Todo sistema consta de una estructura (los componentes que lo forman) y una función (aquello para lo que está destinado). El mecanismo específico mediante el cual un sistema realiza su función se denomina el *servicio* del sistema. Este servicio se proporciona mediante una interfaz de servicio. Los diferentes niveles de la pila IP se pueden considerar sistemas, que porporcionan servicios unos a otros. En el nivel más alto de la pila se encuentran los servicios del nivel de aplicación. El protocolo de nivel de aplicación es el protocolo que emplean las aplicaciones distribuidas.

Desarrollar una aplicación siguiendo el modelo de comunicaciones cliente/servidor requiere definir tres aspectos fundamentales: 1) la función del servidor, 2) la tecnología de comunicaciones a emplear, y 3) el protocolo de nivel de aplicación. Un servicio y su protocolo pueden ser sin estado (*stateless*), si el resultado del servicio no depende de la secuencia concreta de interacciones con el servidor, o con estado (*statefull*) en caso contrario.

Existen multitud de protocolos de nivel de aplicación considerados estándar, ya que los usan muchas aplicaciones distribuidas. Ejemplos de estos son Telnet y SSH (para establecer sesiones remotas), FTP (para transferir archivos) y HTTP, que es el protocolo de nivel de aplicación de la *World Wide Web*.

Aunque la base de las comunicaciones entre aplicaciones son los *sockets*, existen técnicas avanzadas para la programación de aplicaciones distribuidas. Todas estas se basan en los *sockets*, pero añaden una serie de mecanismos de abstracción sobre estos que hacen más fácil su desarrollo y añaden nuevas características. Uno de los ejemplos más importantes de estas técnicas son las llamadas a métodos remotos. Con ellas, la comunicación entre objetos de una aplicación distribuida se realiza como si se tratase de un invocación de métodos en local. El lenguaje Java incorpora un conjunto de herramientas para realizar este tipo de comunicación, llamadas RMI. La pieza clave de este conjunto es la interfaz *Remote*.

Por último, existen otras técnicas de comunicación aún más sofisticadas y potentes que RMI. La más importante de ellas son los servicios web, que hacen uso de protocolo HTTP para el intercambio de mensajes.

EJERCICIOS PROPUESTOS

1. Implementa una aplicación cliente/servidor que sirva documentos de texto. El servidor deberá almacenar documentos de texto, en ficheros *txt*. Cuando un cliente se conecte al servidor, especificará el número de fichero que desea obtener y el servidor le enviará su contenido, carácter a carácter. Usa para ello *sockets stream*.

2. Implementa un servicio de identificación de direcciones IP usando *sockets datagram*. Los clientes enviarán peticiones de identificación al servidor, que contestará con un mensaje que contenga la dirección IP del cliente como una cadena de texto.

3. Se pretende crear una aplicación cliente/servidor que almacene una agenda de contactos. Los clientes pueden conectarse al servidor y subir su información de contacto (nombre, dirección y teléfono), o modificarla si ya existe. Especifica las funciones del servidor, escoge la tecnología de comunicaciones adecuada y diseña (de forma conceptual) el protocolo de nivel de aplicación de este servicio.

4. Implementa un servicio de publicación de mensajes. Los clientes enviarán mensajes de texto al servidor, que los presentará por pantalla. Utiliza para ello invocación de métodos remotos.

5. Implementa un servicio de control de estado de procesos remotos mediante *heartbeat*. La función de este servicio es controlar que sus clientes están activos, y detectar cuándo se han desactivado. Para ello, los clientes deben registrarse en el servidor cuando arrancan. Una vez se han registrado, deben enviar un mensaje especial (latido del corazón o *heartbeat*) al servidor a intervalos de tiempo regulares (cada 10 segundos). Si al cabo de 20 segundos el servidor no ha recibido ningún mensaje por parte de un cliente, lo considerará muerto y avisará por pantalla. Implementar este servicio usando *sockets datagram*.

6. Repite el ejercicio anterior, introduciendo las siguientes modificaciones:

- La tecnología de comunicaciones empleada debe ser llamadas a métodos remotos.
- La duración del intervalo de tiempo del *heartbeat* se indicará como parámetro cuando el cliente se registre. De esta forma, el intervalo de un cliente puede ser de 10 s, otro de 5 s, otro de 20 s, etc.
- El tiempo que debe pasar antes de que el servidor declare muerto ha un cliente será el doble del intervalo de *heartbeat* especificado.
- El servidor debe poder detectar "resurrecciones", esto es, si un cliente que ha sido dado por muerto vuelve a enviar un mensaje de *heartbeat*, el servidor debe darse cuenta y notificarlo por su salida estándar.

7. Implementa un servicio de "operador telefónico". Un "operador telefónico" es un servicio que mantiene comunicación con un grupo de clientes, y que ayuda a que los clientes puedan comunicarse entre sí sin necesidad de conocer su dirección IP. Para ello, cuando un cliente se conecta al servidor indica su identificador personal. Después, cuando envía un mensaje, indica además el identificador personal del cliente al que va destinado. El servidor lo recoge y lo hace llegar al destinatario correcto. Implementa este servicio usando *sockets stream*.

TEST DE CONOCIMIENTOS

1 ¿Cuál de los siguientes NO es una parte fundamental de todo sistema?

a) Función.
b) Modelo de comunicaciones.
c) Estructura.
d) Ninguna de las anteriores.

2 En el contexto de la computación distribuida, ¿qué es un servicio?

a) El conjunto de procedimientos que debe llevar a cabo el cliente para acceder al servidor.
b) Los componentes hardware y software que forman el sistema.
c) El conjunto de mecanismos concretos que hacen posible el acceso a la función del sistema.
d) El protocolo de nivel de aplicación.

3 ¿Cuál de las siguientes afirmaciones es FALSA?

a) Un servicio en red es cualquier servicio que se ubique en cualquier nivel de la pila IP.
b) Un protocolo de nivel de aplicación es el conjunto de reglas que gobiernan la interacción entre los diferentes elementos de una aplicación distribuida.
c) La pila de protocolos IP es un conjunto de sistemas independientes, montados unos sobre otros para realizar una tarea compleja.
d) Todos los niveles de la pila IP ofrecen una interfaz de acceso al siguiente nivel, salvo el nivel de aplicación, por ser el último.

4 ¿Cuál de las siguientes tareas NO es un paso fundamental a la hora de programar un servidor?

a) Seleccionar la tecnología de comunicaciones adecuada.
b) Definir las funciones básicas del servidor.
c) Escoger el lenguaje de programación en el que se deberán implementar los clientes.
d) Diseñar el protocolo de nivel de aplicación.

5 ¿Cuál de las siguientes afirmaciones es FALSA?

a) Los protocolos sin estado no almacenan información sobre la evolución de la sesión, por lo que la secuencia en la que se realizan las peticiones no influye en su resultado.
b) Los protocolos con estado necesitan conocer en detalle la sesión, ya que los resultados de las peticiones dependen de cómo se haya desarrollado esta.
c) El hecho de que un protocolo tenga o no estado es independiente de la tecnología de comunicaciones que se use (*sockets stream*, *sockets datagram*, invocación de métodos remotos, etc.).
d) Ninguna de las anteriores.

6 ¿Cuál es la solución más habitual cuando se desea implementar aplicaciones cliente/servidor en las que el servidor pueda atender a muchos clientes de forma simultánea?

a) Usar *sockets datagram* y diseñar un protocolo de nivel de aplicación que incluya pocos mensajes.
b) Implementar un servidor multihilo, probablemente usando *sockets stream*.
c) Paralelizar el código del servidor usando MPI.
d) Ninguna de las anteriores.

7 ¿Cuál de las siguientes afirmaciones es FALSA?

a) El protocolo SSH ofrece comunicaciones seguras.
b) SFTP es una extensión del protocolo FTP que incorpora seguridad en las comunicaciones, gracias la a la incorporación de TLS.
c) Telnet es un protocolo con estado (*stateful*).
d) Ninguna de las anteriores.

8 ¿Cuál de las siguientes afirmaciones es FALSA sobre el protocolo HTTP?

a) Es el protocolo que se usa en la *World Wide Web*.
b) Es un protocolo sin estado (*stateless*).
c) La versión 1.1 define, entre otras, las peticiones GET, PUT y RESTART.
d) Los códigos de estado se agrupan en 5 categorías, dependiendo de su significado.

9 ¿Cuál de los siguientes pasos NO OCURRE durante la invocación de un método remoto?

a) Invocación del método del objeto *stub*.
b) Envío de la petición al objeto servidor.
c) Retorno del valor por parte del objeto *stub*.
d) Destrucción del objeto *stub*.

10 ¿Cuál de las siguientes afirmaciones sobre los servicios web es FALSA?

a) Existen dos tipos de servicios web, en función de si están basados en SOAP o en REST.
b) Basan sus comunicaciones en el protocolo HTTP.
c) Los servicios web REST representan los mensajes en lenguaje XML.
d) Un servicio web se puede expresar de manera formal usando WSDL.