

Introducción

El lenguaje de programación Java se diseñó partiendo de la premisa de que sería la herramienta más importante para la conexión de computadoras a través de Internet y de intranets corporativas. Y así fue. Se sorprenderá de lo sencillo que resulta realizar estas operaciones en Java.

Vamos a comenzar este capítulo hablando un poco sobre los conceptos básicos del trabajo en red. El resto del capítulo versará sobre las dificultades que puede planteársenos a la hora de llevar a cabo trabajos complicados en la Red con Java. Por ejemplo, la programación Java en el lado del servidor. En particular, veremos la forma de combinar un applet y un servlet para obtener información de Internet. En la primera parte asumimos que no tiene ninguna experiencia de programación en red. Hacia el final del capítulo, el código se vuelve más complejo.

Conexión con un servidor

Antes de escribir nuestro primer programa en red, vamos a aprender algo acerca de una gran herramienta de depuración para la programación en red: telnet. Muchos sistemas (incluidos UNIX y Windows) ya incorporan telnet. Sin embargo, es opcional en algunas instalaciones. Si no puede ejecutar telnet desde una ventana de comandos, deberá instalarse desde sus discos de instalación del sistema operativo.

Puede que ya haya empleado telnet para conectarse con una computadora remota y comprobar su correo electrónico, aunque también puede usarse para comunicar con otros servicios ofrecidos por los servidores de Internet. Aquí tiene un ejemplo de lo que puede hacer. En la línea de comandos escriba:

```
telnet tirne-A.tirnefreq.blrdoc.gov 13
```

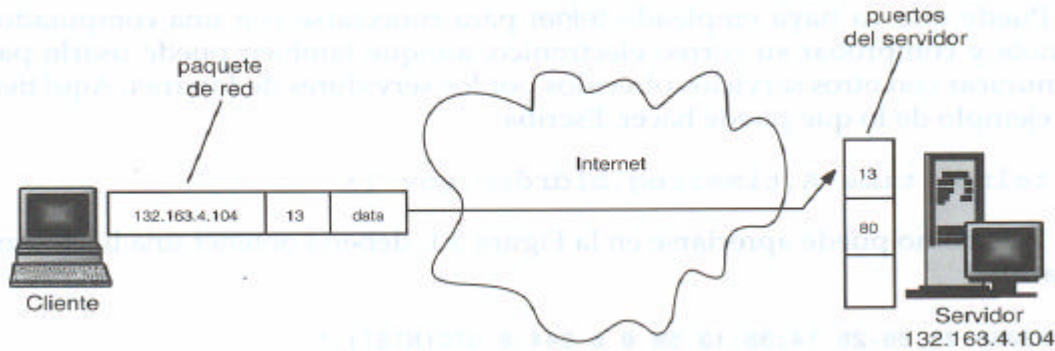
Y la respuesta es:

```
53923 06-07-07 10:25:24 50 0 0 0.0 UTC (NIST) *
```

```
Se ha perdido la conexión con el host
```

¿Qué ha pasado? Pues que ha conectado con el servidor del National Institute of Standards and Technology en Boulder, Colorado, y lo que ha obtenido es la medida de un reloj atómico de Cesio (desde luego, la hora no está totalmente actualizada debido a los retrasos de la red y además, si estamos en Argentina, hay un defasaje de 3 hs, el reloj de la PC está indicando 7:25:xx). Por convenio, el servicio "hora del día" siempre está unido al puerto 13.

En lenguaje de red, un puerto no es un dispositivo físico, sino una abstracción para facilitar la comunicación entre un servidor y un cliente



Lo que ocurre es que el software del servidor está ejecutándose continuamente en la máquina remota, esperando cualquier tráfico de red que dialogue con el puerto 13. Cuando el sistema operativo de este servidor recupera un paquete de red que contiene una petición para conectar con el puerto 13, activa el servicio de escucha del servidor y establece la conexión, que permanece activa hasta que es finalizada por alguna de las dos partes.

Cuando se inicia una sesión telnet con un `time-A.timefreq.bldrdoc.gov` en el puerto 13, una parte del software de red transforma la cadena `"time-A.timefreq.bldrdoc.gov"` a su dirección IP (Protocolo Internet) correcta, `132.163.4.102`, se envía una petición de conexión a dicho servidor, solicitándole la entrada al puerto 13. Una vez establecida, el programa remoto devuelve una línea de datos y cierra la conexión.

Sigue un programa en Java que hace lo mismo que hicimos antes usando telnet (conectar a un puerto y mostrar lo que encuentre).

```
import java.io.*;
import java.net.*;

/**
 * Este programa realiza una conexión Socket a un reloj atómico en
 * Boulder, Colorado, y muestra la hora que el servidor envía
 */
public class SocketTest
{
    public static void main(String[] args)
    {
        try
        {
            Socket s = new Socket("time-A.timefreq.bldrdoc.gov",13);
            // Instanciamos un objeto Socket s;
            // parametros: direccion remota y puerto

            BufferedReader in = new BufferedReader
                (new InputStreamReader(s.getInputStream()));
            // Instanciamos un objeto in, clase BufferedReader
            // Usando el objeto Socket s activamos getInputStream()
            // de la clase InputStreamReader.
            // El retorno de la expresion anterior es parametro del
            // constructor de BufferedReader, quien instancia in

            boolean more = true;
            while (more) // Ciclo de lectura del flujo in
            {
                String line = in.readLine();
                if (line == null)
```

```

        more = false;
    else
        System.out.println(line);
    }

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

```

53923 06-07-07 11:03:31 50 0 0 431.6 UTC(NIST) *

Este programa es muy simple, observe que hemos importado el paquete `java.net` y que hemos capturado cualquier error de entrada/salida que se produzca (el código está encerrado en un bloque `try/catch`), ya que pueden ocurrir muchas cosas en una conexión de red, la mayor parte de los métodos relacionados con ella "amenazan" con lanzar errores de este tipo, por lo que es necesario capturarlos para que el código pueda compilarse.

El proceso continúa hasta que finaliza el flujo y el servidor desconecta. Esto se sabe cuando el método `readLine` devuelva una cadena `null`.

Este programa sólo funciona con servidores muy simples, como un servicio de "hora del día". En programas de red más complejos, el cliente envía peticiones de datos al servidor, y puede que éste no realice la desconexión inmediatamente después del final de una de estas peticiones. Veremos cómo implementar este comportamiento a lo largo de los ejemplos de este capítulo.

En esencia, la clase `Socket` es agradable y fácil de usar porque la tecnología Java oculta las complejidades derivadas del establecimiento de la conexión de red y del envío de datos a través de ella. En esencia, el paquete `java.net` le proporciona la misma interfaz de programación que podría utilizar en el trabajo con un archivo.

Implementación de servidores

Ahora que ya tenemos implementado un cliente de red básico que recibe datos a través de la Red, vamos a desarrollar un sencillo servidor que pueda enviar información. Una vez iniciado el programa de servidor, espera hasta que algún cliente conecta con su puerto. Hemos elegido el 8189 porque no se utiliza por ninguno de los servicios estándar. La clase `ServerSocket` se utiliza para establecer un socket. En nuestro caso, el comando:

```
ServerSocket s = new ServerSocket(8189);
```

establece un servidor que monitoriza el puerto 8189. El comando:

```
Socket incoming = s.accept();
```

indica al programa que debe esperar indefinidamente hasta que algún cliente conecte con ese puerto. Una vez que alguien establezca una conexión válida a través de la red, este método devuelve un objeto `Socket` que representa la conexión que se ha establecido. Puede usar este objeto para obtener un lector

(input reader) y un escritor (output writer) para ese socket, tal y como puede verse en el siguiente fragmento de código:

```
BufferedReader in = new BufferedReader (new
    InputStreamReader(incoming.getInputStream()));

PrintWriter out = new PrintWriter
    (incoming.getOutputStream(), true /* autoFlush */);
```

Todo lo que el servidor envía a través de su flujo de salida se convierte en la entrada del programa del cliente, y todo el flujo de salida de éste acaba en el flujo de entrada del servidor.

En todos los ejemplos de este capítulo, transmitiremos texto a través de sockets. Por tanto, podemos alternar los streams entre lectores y escritores. A continuación, usamos el método `readLine` (definido en `BufferedReader`, no en `InputStream`) y `print` (definido en `PrintWriter`, no en `OutputStream`). Si quisiéramos transmitir datos binarios, deberíamos cambiar los streams por `DataInputStream` y `DataOutputStreams`. Para trabajar con objetos serializados, usaremos `ObjectInputStream` y `ObjectOutputStreams`.

Vamos a enviar un saludo al cliente:

```
out.println("Hola! Typee Adios para salir ...");
```

Cuando utilice telnet para conectar con este programa de servidor en el puerto 8189, verá el mensaje anterior en la pantalla.

En este ejemplo sólo leemos la entrada del cliente (una cada vez) y la repetimos. Esto demuestra que el programa toma la entrada del cliente.

```
String line = in.readLine();
if (line != null){
    out.println('Echo: ' + line);
    if (line.trim().equals("Adios")) done = true;
}
else done = true;
```

Por último, cerramos el socket de entrada.

```
incoming.close();
```

y esto es todo. Cada programa servidor, como un servidor web HTTP, continúa realizando este bucle:

1. Recupera un comando del cliente (" dame esta información") a través de un flujo de datos de entrada.
2. Éste, de algún modo, obtiene esa información.
3. Información que es devuelta al cliente a través del flujo de datos de salida.

Veamos el programa java

```
import java.io.*;
import java.net.*;

/**
 * Este programa implementa un servidor simple que escucha al puerto 8189
 * y devuelve como un eco el texto del cliente. Finaliza cuando el cliente
```

```

* tipea Adios
*/
public class EchoServer{
    public static void main(String[] args ){
        try{
            // instanciamos un servidos ServerSocket que escucha al puerto 8189
            ServerSocket s = new ServerSocket(8189);

            Socket incoming = s.accept( );
            // Le indicamos que espere indefinidamente

            BufferedReader in = new BufferedReader
                (new InputStreamReader(incoming.getInputStream()));
            PrintWriter out = new PrintWriter
                (incoming.getOutputStream(), true /* autoFlush */);
            // instanciamos flujos de entrada y salida

            out.println( "Hola! Tipee Adios para salir..." );

            // echo client input
            boolean done = false;
            while (!done){
                String line = in.readLine();
                if (line == null) done = true;
                else
                {
                    out.println("Echo: " + line);

                    if (line.trim().equals("Adios"))
                        done = true;
                }
            }
            incoming.close();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}

```

Ejecutamos, pero por ahora me dice:

```

java.net.BindException: Address in use: JVM_Bind
    at java.net.PlainSocketImpl.socketBind(Native Method)
    at java.net.PlainSocketImpl.bind(PlainSocketImpl.java:405)
    at java.net.ServerSocket.<init>(ServerSocket.java:170)
    at java.net.ServerSocket.<init>(ServerSocket.java:82)
    at EchoServer.main(EchoServer.java:16)

```

ya lo solucionaremos ...

Para verlo en funcionamiento, debe compilar y ejecutar este programa. A continuación, utilice telnet para conectar con el siguiente servidor y puerto:

```

Server: 127.0.0.1
Port: 8189

```

La dirección IP 127.0.0.1, llamada dirección local del bucle de vuelta (loopback), es especial porque hace referencia a la máquina local. Puesto que está ejecutando un servidor en local, ésta es la forma de conectar.

Observación: Si utiliza una conexión por marcación, necesita tenerlo en funcionamiento para llevar a cabo este experimento. Incluso aunque esté trabajando localmente en su máquina, el software de red debe estar cargado.

A partir de este momento, cualquier persona en el mundo podrá acceder al eco de su servidor, siempre y cuando conozcan su dirección IP y el número mágico del puerto.

Servicio a varios clientes

Existe un problema con el ejemplo del servidor que vimos en la sección anterior. Suponga que queremos permitir que varios usuarios se conecten a la vez. Lo normal es que un servidor esté ejecutándose constantemente en una computadora, y que los usuarios de Internet de cualquier parte del mundo se conecten simultáneamente a la misma. Rechazando las conexiones múltiples se consigue que ningún cliente monopolice el servicio cuando se conecta durante mucho tiempo. Todo esto podemos hacerlo muy bien a través de (claro!!)los threads.

Cada vez que sepamos que el programa ha establecido una nueva conexión socket; es decir, siempre que la petición tenga éxito, lanzaremos un nuevo thread que será el encargado de cuidar la conexión entre el servidor y ese cliente. El programa principal sólo se encargará de seguir esperando nuevas conexiones. Para que esto suceda, el bucle principal del servidor debería parecerse a esto:

```
while (true){
    Socket incoming = s.accept();
    Thread t = new ThreadedEchoHandler(incoming);
    t . start();
}
```

La clase ThreadedEchoHandler deriva de Thread y contiene el bucle de comunicación entre el cliente y su método run.

```
class ThreadedEchoHandler extends Thread{
    public void run(){
        try{
            BufferedReader in = new BufferedRead
                (new InputStreamReader(incoming.getInputStream()));
            PrintWriter out = new PrintWriter
                (incoming.getOutputStream(), true /* autoFlush */);

            String line;
            while ((line = in.readLine()) != null){
                procesar línea
            } // while
            incoming.close();

        }catch(Exception e){manipular la excepción}
    } // run
} // class
```

Ya que cada conexión inicia un nuevo thread, se permite que varios clientes conecten con el servidor al mismo tiempo. Este hecho se puede comprobar muy fácilmente. Compile y ejecute el programa de servidor. Abra varias ventanas telnet. Observará cómo puede comunicarse a través de todas ellas de manera

simultánea. El programa del servidor nunca muere. A menos que lo mate UD, (Ctrl+C).

```
import java.io.*;
import java.net.*;

/**
 * Este programa hace lo mismo que EchoServer, solo que con tantos clientes
 * como se conecten. Usa el mismo puerto 8189.
 */
public class ThreadedEchoServer{
    public static void main(String[] args ){
        try{
            int i = 1;
            ServerSocket s = new ServerSocket(8189);
            for (;;){
                Socket incoming = s.accept( );
                System.out.println("Spawning " + i);
                Thread t = new ThreadedEchoHandler(incoming, i);
                t.start();
                i++;
            }
        }
        catch (Exception e){e.printStackTrace();}
    }
}

/**
 * La clase siguiente, ThreadedEchoHandler, manipula la entrada del cliente
 * mediante una coneccion Socket del servidor.
 */
class ThreadedEchoHandler extends Thread{
    /**
     * Construimos un manipulador (handler).
     * @el parametro i es el socket de entrada
     * @el parametro es el contador de manipuladores (usado en el prompts)
     */
    public ThreadedEchoHandler(Socket i, int c){
        incoming = i; counter = c;
    }

    public void run(){
        try{
            BufferedReader in = new BufferedReader
                (new InputStreamReader(incoming.getInputStream()));
            PrintWriter out = new PrintWriter
                (incoming.getOutputStream(), true /* autoFlush */);

            out.println( "Hola! Tipee Adios para salir ..." );

            boolean done = false;
            while (!done){
                String str = in.readLine();
                if (str == null) done = true;
                else
                {
                    out.println("Echo (" + counter + "): " + str);

                    if (str.trim().equals("Adios"))
                        done = true;
                }
            }
        }
    }
}
```

```

        } // else
    } // while
    incoming.close();
} catch (Exception e){e.printStackTrace();}
}

private Socket incoming;
private int counter;
} // class

```

Documentando Java.Net.ServerSocket

- **ServerSocket(int puerto) throws IOException**

Crea un socket de servidor que monitoriza un puerto.

Parámetros: puerto (número de puerto a monitorizar).

- **Socket accept() throws IOException**

Espera una conexión. Este método bloqueará el thread actual hasta que la conexión se haya establecido. El método devuelve un objeto Socket a través del cual el programa puede comunicar con el cliente conectado.

- **void close() throws IOException**

Cierra el socket del servidor.

Envío de correo electrónico

En esta sección vamos a ver un ejemplo práctico de programación socket: un programa que envía correos electrónicos a un sitio remoto.

Para ello, establecemos una conexión socket al puerto 25, que es el puerto SMTP. SMTP es la abreviatura de Protocolo sencillo de transporte de correo, el cual describe el formato de los mensajes de correo electrónico. Puede conectar con cualquier servidor que ejecute un servicio SMTP. En máquinas UNIX, este servicio suele estar implementado por sendmail. Sin embargo, el servidor debe estar dispuesto para aceptar su petición. Solía ser frecuente que los servidores sendmail enrutaran gustosamente los correos electrónicos de cualquiera, pero en estos tiempos en los que los spams fluyen a diario por la Red, muchos servidores están configurados para aceptar sólo las peticiones de los usuarios, dominios o rangos de direcciones IP en los que confían.

Una vez establecida la conexión con el servidor, envíe una cabecera de correo electrónico (en formato SMTP, que es fácil de generar), seguida del cuerpo del mensaje.

Aquí tiene los detalles:

1. Abra un socket en su servidor.

```

Socket s = new Socket("mail.yourserver.com", 25); // 25 es el "puerto SMTP
PrintWriter out = new PrintWriter(s.getOutputStream());

```

2. Envíe la siguiente información al flujo de impresión:

```

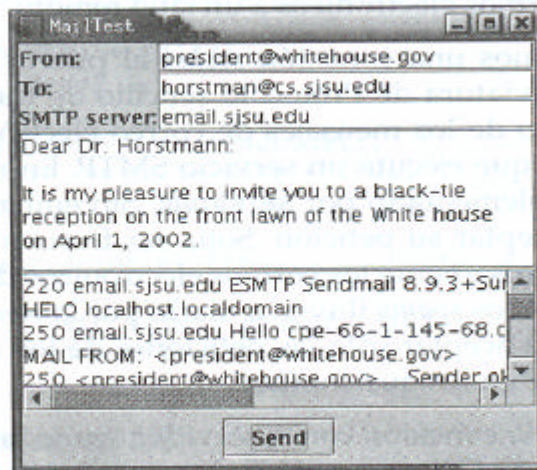
HELO host de envío
MAIL FROM: <dirección email del emisor>
RCPT TO: <dirección email del receptor>
DATA mensaje del correo
    (cualquier número de líneas)
.
QUIT

```


La especificación SMTP (RFC 821) detalla que las líneas deben estar finalizadas con un carácter \r seguido de un \n.

Ojo! (Aunque esto es obvio ...)

Muchos servidores SMTP no comprueban la veracidad de la información: debe estar dispuesto a responder a cualquiera que lo solicite (la próxima vez que reciba un mensaje de correo electrónico procedente de `president@whitehouse.gov` invitándole a una fiesta de etiqueta en el jardín delantero, tenga en cuenta que cualquiera podría haberse conectado a un servidor SMTP y haber creado este mensaje falso). Si hacemos las cosas bien, al ejecutar deberíamos ver la siguiente algo así:



El programa sólo envía la secuencia de comandos que ya hemos comentado antes. Muestra esos comandos que ha enviado al servidor SMTP y las respuestas que recibe. Observe que la comunicación con el servidor de correo se produce en un thread aparte para que el thread de la interfaz de usuario no se bloquee mientras intenta conectar con dicho servidor.

Y el programa es:

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.net.*;
import java.io.*;
import javax.swing.*;

/**
 * Como enviar mensajes de correo usando sockets
 */
public class MailTest{
    public static void main(String[] args){
        JFrame frame = new MailTestFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.show();
    }
}

/**
 * Sigue la interfaz anterior de la imagen anterior ...
 */
class MailTestFrame extends JFrame{
    public MailTestFrame(){
        setSize(WIDTH, HEIGHT);
        setTitle("MailTest");
    }
}
```

```

getContentPane().setLayout(new GridBagLayout());

GridBagConstraints gbc = new GridBagConstraints();
gbc.fill = GridBagConstraints.HORIZONTAL;
gbc.weightx = 0;
gbc.weighty = 0;

gbc.weightx = 0;
add(new JLabel("From:"), gbc, 0, 0, 1, 1);
gbc.weightx = 100;
from = new JTextField(20);
add(from, gbc, 1, 0, 1, 1);

gbc.weightx = 0;
add(new JLabel("To:"), gbc, 0, 1, 1, 1);
gbc.weightx = 100;
to = new JTextField(20);
add(to, gbc, 1, 1, 1, 1);

gbc.weightx = 0;
add(new JLabel("SMTP server:"), gbc, 0, 2, 1, 1);
gbc.weightx = 100;
smtpServer = new JTextField(20);
add(smtpServer, gbc, 1, 2, 1, 1);

gbc.fill = GridBagConstraints.BOTH;
gbc.weighty = 100;
message = new JTextArea();
add(new JScrollPane(message), gbc, 0, 3, 2, 1);

communication = new JTextArea();
add(new JScrollPane(communication), gbc, 0, 4, 2, 1);

gbc.weighty = 0;
JButton sendButton = new JButton("Send");
sendButton.addActionListener(new
    ActionListener(){
        public void actionPerformed(ActionEvent evt){
            new Thread(){
                public void run(){sendMail();}}.start();
            } // actionPerformed
        } // ActionListener()
    }); // new ActionListener()

JPanel buttonPanel = new JPanel();
buttonPanel.add(sendButton);
add(buttonPanel, gbc, 0, 5, 2, 1);
} // constructor public MailTestFrame

/**
Añade componentes a este marco.
@parametro c es el componente a añadir
@parametro gbc la GridBagConstraints
@parametro x es la columna en la rejilla (frame)
@parametro y es la fila idem
@parametro w es la cantidad de columnas de la rejilla
@parametro h idem, filas
*/
private void add(Component c, GridBagConstraints gbc,

```

```

        int x, int y, int w, int h)
    {
        gbc.gridx = x;
        gbc.gridy = y;
        gbc.gridwidth = w;
        gbc.gridheight = h;
        getContentPane().add(c, gbc);
    }

    public void sendMail(){// Envia texto introducido en la interfaz
        try{
            Socket s = new Socket(smtpServer.getText(), 25);

            out = new PrintWriter(s.getOutputStream());
            in = new BufferedReader(new
                InputStreamReader(s.getInputStream()));

            String hostName
                = InetAddress.getLocalHost().getHostName();

            receive();
            send("HELO " + hostName);
            receive();
            send("MAIL FROM: <" + from.getText() + ">");
            receive();
            send("RCPT TO: <" + to.getText() + ">");
            receive();
            send("DATA");
            receive();
            StringTokenizer tokenizer = new StringTokenizer(
                message.getText(), "\n");
            while (tokenizer.hasMoreTokens())
                send(tokenizer.nextToken());
            send(".");
            receive();
            s.close();
        }
        catch (IOException exception)
        {
            communication.append("Error: " + exception);
        }
    }

    /**
     * Envia una cadena al socket y la repite
     * en el area de texto de la comunicacion.
     * @parametro s es la cadena a enviar.
     */
    public void send(String s) throws IOException{
        communication.append(s);
        communication.append("\n");
        out.print(s);
        out.print("\r\n");
        out.flush();
    }

    /**
     * recibe una cadena desde el socket y la muestra en la JTextArea
     */

```

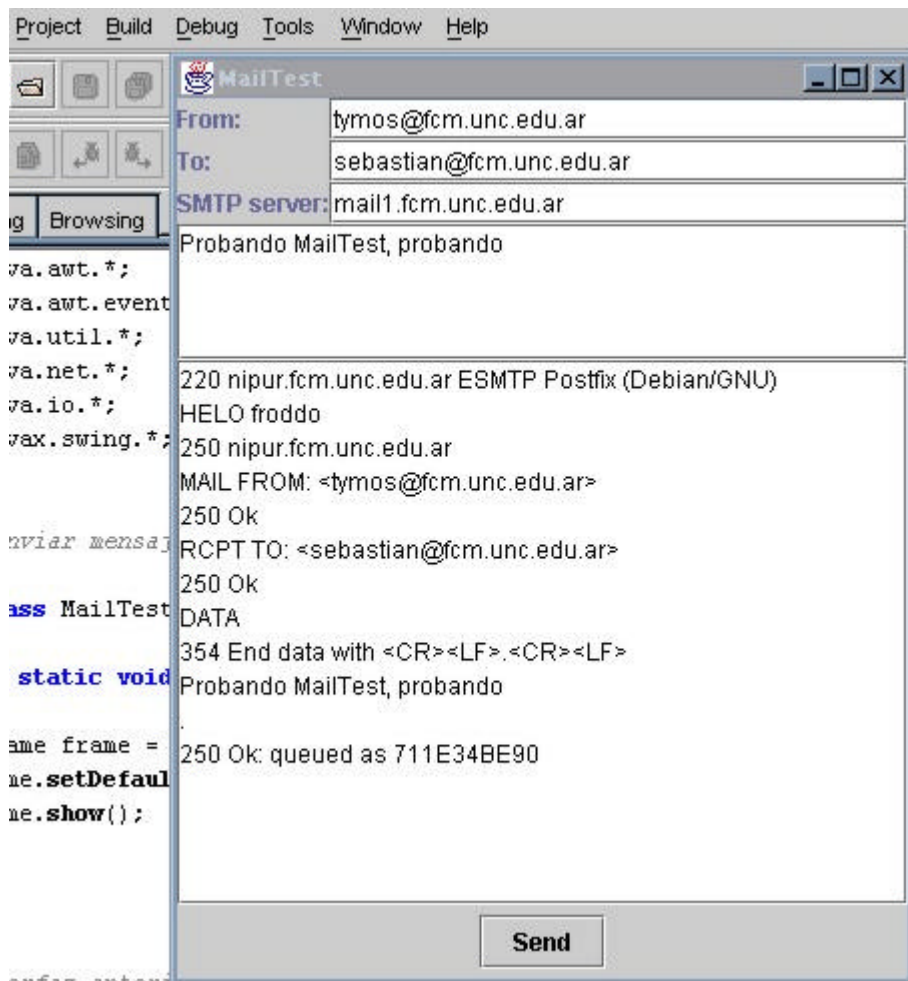
```
public void receive() throws IOException{
    String line = in.readLine();
    if (line != null){
        communication.append(line);
        communication.append("\n");
    }
}

private BufferedReader in;
private PrintWriter out;
private JTextField from;
private JTextField to;
private JTextField smtpServer;
private JTextArea message;
private JTextArea communication;

public static final int WIDTH = 300;
public static final int HEIGHT = 300;
}
```

Un capture de una ejecución de MailTest.java en el "mundo real", en la pg 11
En el area de texto inferior podemos seguir que ocurrió al clicar el botton
Send. Detallamos esto por línea.

- 1 - Mi PC se conectó al puerto 25 de nipur (Nuestro servidor de correo) del cual
 maill es un alias.
- 2 - Mi PC saluda "Hello", ella se llama FRODO, (Como en El Señor de los Anillos)
- 3 - El acknowledgement (reconocimiento, acuse de recibo) del Servidor
- 4 - Mi PC informa envio de correo
- 5 - Servidor dice que lo recibe OK
- 6 - Se indica al destinatario del mail
- 7 - Destinatario dice ok
- 8 - Siguen datos
- 9 - Especificaciones técnicas (HTML)
- 10 - El texto de mi mail
- 11 - . (punto) generado por mi PC, informando fin del texto del mail.
- 12 - Respuesta de nipur informandome mail puesto en cola.



Programación avanzada de sockets

Tiempos muertos del socket

En programas de la vida real, no esperará a leer desde un socket, ya que los métodos de lectura se bloquearán hasta que los datos estén disponibles. Si no se puede alcanzar el servidor, su aplicación esperará durante mucho tiempo, y se encontrará a merced del tiempo muerto que el sistema operativo tenga establecido.

Esto no es bueno, debería ser usted el que decidiera qué tiempo muerto es razonable para una aplicación concreta para, después, llamar al método `setSoTimeout` y establecer dicho valor (en milisegundos).

```
Socket s = new Socket(. . .);
s.setSoTimeout(10000); // tiempo muerto tras 10 segundos
```

Si se ha establecido el valor del tiempo muerto para un socket, todas las operaciones posteriores de lectura lanzarán una `InterruptedException` cuando venza dicho tiempo muerto antes de que el flujo de entrada se haya establecido. En este caso se puede capturar la excepción y reaccionar ante dicho tiempo muerto.

```
try{
    String line
    while ((String line = in.readLine()) != null){código a procesar}
}
```

```
catch (InterruptedException exception){realizar algo ante un tiempo muerto}
```

Existe una utilidad adicional del tiempo muerto que debe conocer. El constructor:

```
Socket(String servidor, int puerto)
```

puede realizar un bloqueo indefinido hasta que se haya establecido una conexión con el servidor.

A partir del SDK 1.4, se puede superar este problema construyendo primero un socket sin conexión y conectándolo con una especificación de tiempo muerto:

```
Socket s = new Socket();
s.connect(new InetSocketAddress(servidor, puerto), timeout);
```

Cierre parcial

Cuando un programa de cliente envía una petición al servidor, éste debe ser capaz de determinar cuándo se produce el final de la petición. Por esta razón, muchos protocolos de Internet (como SMTP) están orientados a la línea. Otros contienen cabeceras que especifican el tamaño de la petición de datos. Y otro grupo es tan duro que escribe el final de esta petición de datos en un archivo. En este caso bastará con cerrar ese archivo al final de los datos. Pero si cierra el socket, se producirá la desconexión inmediata del servidor.

El cierre parcial resuelve este problema. Se puede cerrar el flujo de salida de un socket, indicando de este modo al servidor que se ha producido el final de una petición de datos pero que deseamos mantener abierto el flujo de entrada para recibir la respuesta.

La parte del cliente tendría este aspecto:

```
Socket socket = new Socket(host, port);
BufferedReader reader = new BufferedReader( new
    InputStreamReader(socket.getInputStream()));
PrintWriter writer = new PrintWriter(
    socket.getOutputStream());
// envía la petición de datos
writer.print(. . .);
writer.flush();
socket.shutdownOutput();
// ahora se cierra la mitad del socket
// lee la respuesta
String line;
while ((line = reader.readLine()) != null)
    ...
socket.close();
```

El lado del servidor sólo lee la entrada hasta que se alcanza el final del flujo.

Este protocolo sólo resulta útil para servicios en un solo sentido, como el HTTP en el que un cliente conecta, realiza su petición, toma la respuesta y después desconecta.

Direcciones Internet

Habitualmente, no tendrá que preocuparse por las direcciones Internet (la dirección numérica de un servidor consta de cuatro bytes, o seis en el caso de

IPv6; por ejemplo, 132.163.4.102). Sin embargo, puede usar la clase `InetAddress` si necesita convertir nombres de servidor en direcciones Internet.

A partir del SDK 1.4, el paquete `java.net` soporta direcciones Internet IPv6, ofreciendo lo que hace el sistema operativo del servidor. Por ejemplo, podrá hacer uso de las direcciones IPv6 en Solaris, pero no en Windows (o tal vez ya si, el tiempo pasa ...).

El método estático `getByName` devuelve un objeto `InetAddress` de un servidor. Por ejemplo:

```
InetAddress address = InetAddress.getByName("time-A.timefreq.bldrdoc.gov");
```

devuelve un objeto `InetAddress` que encapsula la secuencia de cuatro bytes 132.163.4.102. Se puede acceder a los bytes a través del método `getAddress`.

```
byte[] addressBytes = address.getAddress();
```

Para facilitar el balanceo de la carga, los nombres de servidores con mucho tráfico suelen corresponderse con varias direcciones Internet. Interesante saber con cuantas cuenta actualmente el servidor `java.sun.com`. Cuando se accede al servidor, se selecciona una de ellas de forma aleatoria. Se puede llamar a todos los servidores mediante el método `getAllByName`.

```
InetAddress[] addresses = InetAddress.getAllByName(host);
```

Por último, hay veces en las que se necesita la dirección del servidor local. Si sólo se pregunta por el valor de `localhost`, siempre se obtendrá la dirección 127.0.0.1, lo que no resulta especialmente útil. En su lugar, mejor usar el método estático `getLocalHost`.

```
InetAddress address = InetAddress.getLocalHost();
```

El ejemplo que sigue es un sencillo programa que muestra la dirección Internet de su servidor local, en el caso de que no especifique nada en la línea de entrada, o las correspondientes al servidor especificado, si informa.

```
import java.net.*;
public class InetAddressTest{
    public void direccion(String dominio){
        try{
            if (dominio.length() > 0){
                String host = dominio;
                InetAddress[] addresses
                    = InetAddress.getAllByName(host);
                for (int i = 0; i < addresses.length; i++)
                    System.out.println("Veamos ... lo tengo! " + addresses[i]);
            }
            else{
                InetAddress localhostAddress
                    = InetAddress.getLocalHost();
                System.out.println(localhostAddress);
            }
        }
        catch (Exception e){e.printStackTrace();}
    }
    // direccion

    public static void main(String[] args){
        String dominio = null, auxText;
        InetAddressTest iAT = new InetAddressTest();
```

```

        System.out.println("Que dominio le interesa? ");
        dominio = In.readLine();
        if(dominio.length() == 0)
            auxText = "Me interesa mi dominio ";
        else
            auxText = "Me interesa el de ";
        System.out.println(auxText+dominio+", por favor");
        iAT.direccion(dominio);
        System.out.println("Demo finished!");
    }
}

```

Probemos

```

Que dominio le interesa?
Me interesa el de java.sun.com, por favor
Veamos ... lo tengo! java.sun.com/72.5.124.55
Demo finished!

```

```

Que dominio le interesa?
Me interesa mi dominio , por favor
tymoschuk/xxx.xx.xxx.x
Demo finished!

```

El metodo informa mi direccion IP; la he censurado, en prevención de alumnos crackers ...

```

Que dominio le interesa?
Me interesa el de java.net.*, por favor
java.net.UnknownHostException: java.net.*
    at java.net.InetAddress.getAllByName0(InetAddress.java:571)
    at java.net.InetAddress.getAllByName0(InetAddress.java:540)
    at java.net.InetAddress.getAllByName(InetAddress.java:533)
    at InetAddressTest.direccion(InetAddressTest.java:7)
    at InetAddressTest.main(InetAddressTest.java:31)
Demo finished!

```

Por supuesto, java.net.* no es ningún dominio. Observe que como tratamiento de la excepción que se produce hemos pedido la impresión de la pila de llamadas, esto es lo que hace `e.printStackTrace()`.

En esta Unidad sólo cubrimos el protocolo de red TCP (Transmission Control Protocol, Protocolo para el control de la transmisión), el cual establece una conexión fiable entre dos computadoras. Java también soporta el protocolo UDP (User Datagram Protocol, Protocolo de datagrama de usuario), que puede emplearse para enviar paquetes de datos (también llamados datagramas) con más carga de la necesaria para TCP. El inconveniente es que dichos paquetes pueden ser entregados de forma aleatoria, o, incluso, completamente descartados. Es obligación del receptor ponerlos en el orden correcto y solicitar la retransmisión de aquellos que no hayan llegado. UDP está mejor adaptado para las aplicaciones en las que se pueden tolerar la desaparición de paquetes, como, por ejemplo, en flujos de audio o vídeo, o en mediciones continuas.

Documentando clase **java.net.Socket**

- **Socket(String servidor, int puerto)**

Crea un socket y lo conecta con un puerto de un servidor remoto.

Parámetros: servidor Es el nombre del servidor.

 Puerto Es el número de puerto.

- **Socket()**
Crea un socket que todavía no tiene establecida la conexión.
- **void connect(SocketAddress dirección)**
Conecta este socket con la dirección dada (a partir del SDK 1.4).
- **void connect(SocketAddress dirección, int timeout)**

Conecta este socket con la dirección dada, o retorna en el caso de que el intervalo de tiempo haya expirado (a partir del SDK 1.4).
Parámetros: dirección dirección remota.
 Timeout retraso en milisegundos.
- **boolean isConnected()**
Devuelve true si el socket está conectado (a partir del SDK 1.4).
- **void close()**
Cierra el socket.
- **boolean isClosed()**
Devuelve true si el socket está cerrado (a partir del SDK 1.4).
- **InputStream getInputStream()**
Define el flujo de entrada para leer desde el socket.
- **OutputStream getOutputStream()**
Define el flujo de salida para escribir en el socket.
- **void setSoTimeout(int timeout)**
Establece el tiempo de bloqueo para peticiones de lectura en este Socket. Si se alcanza el tiempo muerto especificado, se lanza una `InterruptedException`.
Parámetros: timeout tiempo muerto en milisegundos (0 para que dicho tiempo muerto sea infinito).
- **void shutdownOutput()**
Posiciona el flujo de salida a "final del flujo".
- **void shutdownInput()**
Establece el flujo de entrada a "final del flujo".
- **boolean isOutputShutdown**
Devuelve true si la salida ha sido desconectada (a partir del SDK 1.4).
- **boolean isInputShutdown**
Devuelve true si la salida ha sido desconectada (a partir del SDK 1.4).

Documentando clase **Java.net.InetAddress**

- **static InetAddress getByName(String servidor)**
- **static InetAddress[] getAllByName(String servidor)**
Estos métodos construyen un `InetAddress`, que es un array de todas las direcciones de Internet del servidor especificado.
- **static InetAddress getLocalHost()**
Construye un `InetAddress` para el servidor local.
- **byte[] getAddress()**
Devuelve un array de bytes que contiene las direcciones numéricas.
- **String getHostAddress()**
Devuelve una cadena con número decimales separados por puntos, como, por ejemplo, "200.16.4.102".

- **String getHostName()**
Devuelve el nombre del servidor.

Documentando clase **java.net.InetSocketAddress**

A partir del SDK 1.4:

- **InetSocketAddress(String servidor, int puerto)**
Construye un objeto de dirección con el servidor y puerto facilitados, resolviendo el nombre del servidor durante la construcción. Si dicha resolución no puede llevarse a cabo, la propiedad `unresolved` del objeto se ajusta a `true`.
- **boolean isUnresolved()**
Devuelve `true` si el objeto de dirección no pudo ser resuelto.

Conexiones URL

Acabamos de ver cómo usar la programación a nivel de socket para conectar con un servidor SMTP y enviar un mensaje de correo electrónico. Sin embargo, si está planificando una aplicación que incorpore este servicio, lo más seguro es que prefiera trabajar a un nivel superior y usar una librería que encapsule los detalles del protocolo. Por ejemplo, Sun Microsystems ha desarrollado la API `JavaMail` como una extensión estándar de la plataforma Java. En dicha API, basta con efectuar una llamada como la siguiente:

```
Transport.send(mensaje);
```

para enviar un mensaje. La librería es la que se encarga de preocuparse por dicho mensaje, por la recepción múltiple, la manipulación de los adjuntos (attachments), etcétera.

Para el resto de este capítulo nos centraremos en los servicios de alto nivel que proporciona la edición estándar de la plataforma Java. Desde luego, la librería de ejecución emplea sockets para implementar todos estos servicios. Pero no tendrá que preocuparse por los detalles del protocolo cuando use estos servicios de alto nivel.

URL y URI

Las clases `URL` y `URLConnection` encapsulan una gran parte de la complejidad asociada a la recuperación de datos desde un servidor remoto. Aquí tiene la forma de especificar un URL.

```
URL url = new URL(cadenaURL);
```

La plataforma Java 2 soporta los recursos HTTP y FTP.

Si desea obtener el contenido del recurso, puede usar el método `openStream` de la clase `URL`. Este método produce un objeto `InputStream` a través del cual se puede leer fácilmente dicho contenido.

```
InputStream uin = url.openStream();  
BufferedReader in = new BufferedReader(new InputStreamReader(uin));  
String line;  
while ((line = in.readLine()) != null){código a procesar;}
```

El paquete `java.net` distingue entre un URL (Uniform Resource Locator) y un URI (Uniform Resource Identifier).

Un URI es una construcción puramente sintáctica que especifica las distintas partes de la cadena que definen un recurso web. Un **URL es un tipo especial de URI** que dispone de la suficiente información como para localizar el recurso.

Otros URI, como: `mailto:cay@horstmann.com`

no son localizadores (no hay datos que localizar en este identificador). En este caso, dichos URI reciben el nombre de URN (Uniform Resource Name).

En la librería Java, la clase URI no dispone de métodos para acceder al recurso que especifica el identificador (su único propósito es analizar). Por el contrario, la clase URL puede abrir un flujo para el recurso. Por esta razón, dicha clase sólo trabaja con los identificadores que la librería Java sabe cómo manipular, como un URL FTP o HTTP.

Para comprender por qué es necesaria una clase URI, consideremos cómo puede ser un URL complejo. Por ejemplo:

```
http://maps.yahoo.com/py/maps.py?csz=Cupertino+CA
ftp://username:password@ftp.yourserver.com/pub/file.txt
```

La especificación URI ofrece las reglas a las que deben ajustarse estos identificadores. La sintaxis de un URI es la siguiente:

```
[esquema:jparteEspecíficaEsquema[#fragmento]
```

En esta descripción, los corchetes [. . .] denotan una parte opcional, y los dos puntos (:) y la almohadilla (#) están incluidos literalmente en el identificador.

Si la parte esquema: se encuentra presente, el URI recibe el nombre de absoluto. En caso de no estado, es relativo.

Un URI absoluto es opaco si la parteEspecíficaEsquema no viene con el carácter /:

```
mailto:cay@horstmann.com
```

Todos los URI absolutos y no opacos, junto con los relativos, son jerárquicos. Aquí tiene algunos ejemplos:

```
http://java.sun.com/index.html
../../../../java/net/Socket.html#Socket()
```

La parteEspecíficaEsquema de un URI jerárquico tiene la siguiente estructura:

```
[//autoridad] [ruta] [?consulta]
```

donde de nuevo lo que está encerrado entre corchetes es una parte opcional. Para los VRI basados en servidor, la autoridad tiene la siguiente forma:

```
[información-usuario@]servidor[:puerto]
```

El puerto debe ser un entero.

La RFC 2396, que es la encargada de estandarizar los URI, también soporta un mecanismo basado en registros donde la autoridad tiene un formato diferente, aunque su uso no es muy frecuente.

Uno de los objetivos de la clase URI es analizar un identificador y dividido en sus distintos componentes, los cuales pueden recuperarse mediante los métodos:

```
getScheme getSchemeSpecificPart getAuthority getUserInfo getHost getport getpath
getauery getFragment
```

El otro propósito de esta clase es manipular los identificadores absolutos y relativos. Si tiene un URI absoluto como éste:

```
http://docs.mycompany.com/api/java/net/ServerSocket.html
```

y un URI relativo con este aspecto:

```
../../java/net/Socket.html#Socket()
```

es posible combinar ambas en un único URI absoluto.

```
http://docs.mycompany.com/api/java/net/Socket.html#Socket()
```

Este proceso recibe el nombre de resolución de un VRL relativo.

El proceso contrario se llama relativización. Por ejemplo, supongamos que disponemos del URI base:

```
http://docs.mycompany.com/api
```

y del URI:

```
http://docs.mycompany.com/api/java/lang/String.html
```

Por tanto, el URI relativo es:

```
java/lang/String.html
```

La clase URI soporta ambas operaciones:

```
relativa = base.relativeize(combinada);
combinada = base.resolve(relativa);
```

La clase URL posee comportamiento para detectar sus componentes. Veamos algunos ejemplos.

```
import java.net.*;
import java.io.*;
public class ParseURL{
    public static void main(String[] args){
        String urlAux;
        try{
            System.out.println("Que URL desea Ud investigar?");
            urlAux = In.readLine();
            System.out.println("Investigamos "+urlAux);
            URL aURL = new URL(urlAux);
            System.out.println("protocol = " + aURL.getProtocol());
            System.out.println("host = " + aURL.getHost());
            System.out.println("filename = " + aURL.getFile());
            System.out.println("port = " + aURL.getPort());
            System.out.println("ref = " + aURL.getRef());
            System.out.println("-----");
        }
        catch (IOException exception){exception.printStackTrace();}
```

```

    }
}

/* Un ejemplo a investigar
"http://java.sun.com:80/docs/books/"
    + "tutorial/index.html#DOWNLOADING" */

Que URL desea Ud investigar?
Investigamos http://java.sun.com:80/docs/books/tutorial/index.html#DOWNLOADING.
protocol = http
host = java.sun.com
filename = /docs/books/tutorial/index.html
port = 80
ref = DOWNLOADING.
-----

// otro ejemplo
http://docs.mycompany.com/api/java/lang/String.html

Que URL desea Ud investigar?
Investigamos http://docs.mycompany.com/api/java/lang/String.html
protocol = http
host = docs.mycompany.com
filename = /api/java/lang/String.html
port = -1
ref = null
-----

```

Uso de URLConnection para recuperar información

Si desea información adicional acerca de un recurso web, debe utilizar la clase URLConnection, la cual le ofrece un control mayor que la clase URI básica.

Cuando trabaje con un objeto URLConnection, debe seguir estos pasos con mucho cuidado:

1. Llame al método openConnection de la clase URL para obtener el objeto URLConnection:

```
URLConnection connection = url.openConnection();
```

2. Establezca cualquier propiedad de la petición mediante los métodos

```
setDoInput,          setDoOutput,          setIfModifiedSince,          setUseCaches,
setAllowUserInteraction, setRequestProperty. Pronto veremos su uso.
```

3. Conecte con el recurso remoto usando al método connect

```
connection.connect();
```

Además de facilitar una conexión socket con el servidor, este método también le pregunta a éste la información de cabecera.

4. Vna vez establecida la conexión con el servidor, puede solicitar la información de cabecera. Hay dos métodos, getHeaderFieldKey y getHeaderField, que enumeran todos los campos de dicha cabecera. Al igual que en el SDK 1.4, existe el método getHeaderFields que recupera un objeto Map estándar con esos campos. Para su interés, los siguientes métodos solicitan los campos estándar:

getContentType, getContentLength, getContentEncoding, getDate, getExpiration y getLastModified

5. Por último, puede acceder a los datos del recurso. Use el método `getInputStream` para obtener un flujo de entrada para la lectura de la información (este flujo es el mismo que devuelve el método `openStream` de la clase `URL`).

Vamos a ver ahora algunos métodos de `URLConnection` con más detalle.

Existen varios de ellos que establecen propiedades de la conexión antes de efectuar el enlace con el servidor. Dos de los más importantes son `setDoInput` y `setDoOutput`. Por defecto, la conexión produce un flujo de entrada para leer los datos desde el servidor, pero no uno de salida para la escritura en él. En caso de necesitar un flujo de salida (por ejemplo, para enviar datos a un servidor web) debe llamar a:

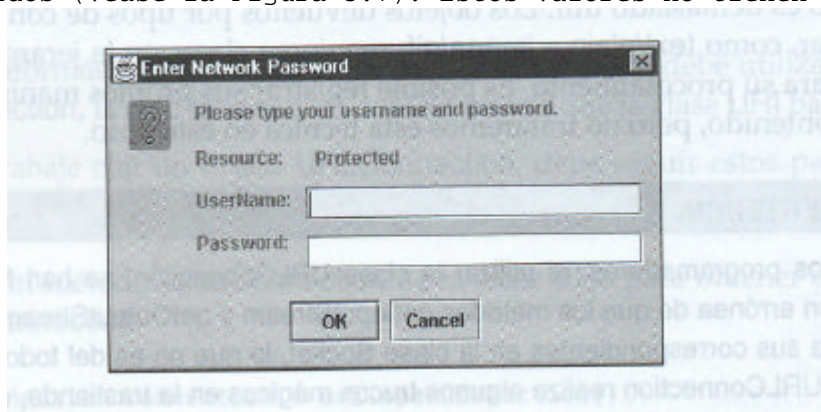
```
connection.setDoOutput(true);
```

A continuación, puede que quiera establecer algunas de las cabeceras de la petición. Dichas cabeceras se envían al servidor junto con el comando de petición. Aquí tiene un ejemplo:

```
GET www.server.com/index.html HTTP/1.0
Referer: http://www.somewhere.com/links.html
```

```
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/4.76 (Windows ME; U) Opera 5.11 [en]
Host: www.server.com
Accept: text/html, image/gif, image/jpeg, image/png, */*
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Cookie: orangemilano=192218887821987
```

El método `setIfModifiedSince` le dice a la conexión que sólo estamos interesados en aquellos datos que han sido modificados a partir de una fecha dada. `setUseCaches` y `setAllowUserInteraction` sólo se emplean dentro de applets. El método `setUseCaches` obliga al navegador a que primero compruebe su caché. `setAllowUserInteraction` permite que una aplicación muestre un cuadro emergente para solicitar el nombre de usuario y la contraseña necesarios para acceder a recursos protegidos (véase la Figura 3.7). Estos valores no tienen efectos fuera



de los applets.

Un cuadro de diálogo para solicitar una contraseña.

Por último, el método `setRequestProperty` le permite establecer cualquier pareja de nombre/valor necesaria para un protocolo concreto. Si necesita obtener el formato de las cabeceras de petición HTTP, consulte la RFC 2616.

Por ejemplo, si desea acceder a una página protegida por contraseña, debe hacer lo siguiente:

1. Cree una cadena concatenando el nombre de usuario, un carácter dos puntos (:) y la contraseña.

```
String input = username + ":" + password;
```

2. Convierta la cadena resultante en otra ASCII apta para su transmisión por Internet.

```
String encoding = URLEncoder.encode(input);
```

3. Llame al método `setRequestProperty` con un nombre "Authorization" y un valor "Basic " + encoding:

```
connection.setRequestProperty("Authorization", "Basic " + encoding);
```

Para acceder a un archivo protegido a través de FTP, el proceso cambia. Basta con construir un URL en la forma:

```
ftp://username:password@ftp.yourserver.com/pub/file.txt
```

Una vez que se ha llamado al método `connect`, puede solicitar la información de cabecera de respuesta. En primer lugar, vamos a ver cómo obtener todos los campos de dicha cabecera. Los desarrolladores de esta clase sintieron la necesidad de expresar su individualidad introduciendo otro protocolo de iteración. La llamada:

```
String key = connection.getHeaderFieldKey(n);
```

toma la clave nesima de la cabecera de respuesta, donde n comienza en 1. Devuelve null si n es cero o contiene un valor mayor que el número total de campo. No hay forma de obtener el número de campos, así que la única solución es llamar sucesivamente a `getHeaderFieldKey` hasta obtener null. De forma análoga, la llamada:

```
String value = connection.getHeaderField(n);
```

 devuelve el nesimo valor.

Afortunadamente, a partir del SDK 1.4 disponemos del método `getHeaderFields` que devuelve un Map de los campos de la cabecera y a los que podemos acceder de la forma que ya vimos en el Capítulo 2.

```
Map headerFields = connection.getHeaderFields();
```

Aquí puede ver un conjunto de campos de cabecera de respuesta obtenido de una petición HTTP normal.

```
Date: Wed, 29 Aug 2001 00:15:48 GMT
Server: Apache/1.3.3 (Unix)
Last-Modified: Sun, 24 Jun 2001 20:53:38 GMT
ETag: "28094e-12cd-37729ad2"
Accept-Ranges: bytes
Content-Length: 4813
Connection: close
Content-Type: text/html
```

Existen seis métodos que obtienen los valores de los campos más habituales y los convierten a tipos numéricos cuando sea necesario. Los métodos que devuelven un

tipo long envían el número de segundo transcurridos desde el 1 de enero de 1970 GMT.

Métodos para la obtención de valores de la cabecera de respuesta

Nombre clave	Nombre del método	Tipo devuelto
Date	getDate	long
Expires	getExpiration	long
Last-Modified	getLastModified	long
Content-Length	getContentLength	int
Content-Type	getContentType	String
Content-Encoding	getContentEncoding	String

El programa del ejemplo que vemos a continuación le permite experimentar con conexiones URL. Al ejecutarlo, suministre un URL y, opcionalmente, un nombre de usuario y una contraseña, por ejemplo:

```
java URLConnectionTest http://www.yourserver.com user pw
```

El programa muestra lo siguiente:

- todas las claves y valores de la cabecera;
- los valores devueltos por los seis métodos de la tabla anterior;
- las diez primeras líneas del recurso solicitado.

Vamos con un programa ejemplo

```
import java.io.*;
import java.net.*;
import java.util.*;

/**
 * Este programa conecta con el URL solicitado por el operador, o con
 * java.sun.com si no se informa nada.
 * Utiliza todos los metodos vistos para exhibir datos de la cabecera,
 * y las diez primeras lineas del archivo de definición del URL.
 * Opcionalmente puede informar usuario y password
 */
public class URLConnectionTest{
    public static void main(String[] args){
        String urlName = null;
        String username = null;
        String password = null;
        try{
            System.out.println("Que URL desea Ud investigar?");
            urlName = In.readLine();
            if (urlName == null)urlName = "http://java.sun.com";
            System.out.println("Investigamos "+urlName);

            URL url = new URL(urlName);
            URLConnection connection = url.openConnection();

            System.out.println("Si lo desea, puede identificarse ahora");
            username = In.readLine();

            System.out.println("Puede ser util disponer de su Password");
            password = In.readLine();

            // set username, password if specified on command line
```



```

    if (username != null && password != null){
        String input = username + ":" + password;
        String encoding = URLEncoder.encode(input);
        connection.setRequestProperty("Authorization",
            "Basic " + encoding);
    }

    connection.connect();

    // Mostrar campos de la cabecera

    int n = 1;
    String key;
    while ((key = connection.getHeaderFieldKey(n)) != null)
    {
        String value = connection.getHeaderField(n);
        System.out.println(key + ": " + value);
        n++;
    }

    // imprimir campos fundamentales

    System.out.println("-----");
    System.out.println("getContentType: "
        + connection.getContentType());
    System.out.println("getContentLength: "
        + connection.getContentLength());
    System.out.println("getContentEncoding: "
        + connection.getContentEncoding());
    System.out.println("getDate: "
        + connection.getDate());
    System.out.println("getExpiration: "
        + connection.getExpiration());
    System.out.println("getLastModified: "
        + connection.getLastModified());
    System.out.println("-----");

    BufferedReader in = new BufferedReader(new
        InputStreamReader(connection.getInputStream()));

    // imprimir las 10 primeras lineas del archivo de definicion de la
    pagina

    String line;
    n = 1;
    while ((line = in.readLine()) != null && n <= 10)
    {
        System.out.println(line);
        n++;
    }
    if (line != null) System.out.println(". . .");
}
catch (IOException exception)
{
    exception.printStackTrace();
}
}
}

```

Una primera ejecución:

```

Que URL desea Ud investigar?
Investigamos http://www.frc.utn.edu.ar
Si lo desea, puede identificarse ahora
Puede ser util disponer de su Password
Server: Microsoft-IIS/5.0
Date: Mon, 10 Jul 2006 21:25:59 GMT
pragma: no-cache
cache-control: private
Content-Length: 29264
Content-Type: text/html
Expires: Sun, 09 Jul 2006 21:25:58 GMT
Set-Cookie: ASPSESSIONIDSAATSCRR=HBFNDLGIBAEENCHPGPCMDIOH; path=/
Cache-control: no-cache
-----
getContentType: text/html
getContentLength: 29264
getContentEncoding: null
getDate: 1152566759000
getExpiration: 1152480358000
getLastModified: 0
-----

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html><!-- InstanceBegin template="/Templates/Principal.dwt.asp"
codeOutsideHTMLOutsideHTMLIsLocked="false" -->
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<meta name="keywords" content="UTN-FRC, UTN, ARGENTINA, FRC, Universidad
Tecnológica Nacional, Universidad Tecnológica Nacional - Facultad Regional
Córdoba, Facultad Regional Córdoba, U.T.N., F.R.C, UTN - Facultad Regional
Córdoba, U.T.N. - Facultad Regional Córdoba, ARGENTINE, CIUDAD DE CORDOBA,
CORDOBA, CORDOBA, CIUDAD DE CORDOBA" />
<meta name="description" content="UTN FRC, casa de altos estudios de la ciudad
de Córdoba. P&aacute;gina Web de la Universidad Tecnológica Nacional - Facultad
Regional Córdoba" />
<!-- InstanceBeginEditable name="doctitle" -->
<title>Universidad Tecnol&oacute;gica Nacional - Facultad Regional
C&oacute;rdoba</title>
<!-- InstanceEndEditable -->
. . .

```

Una segunda prueba

```

Que URL desea Ud investigar?
Investigamos http://www.lanacion.com.ar
Si lo desea, puede identificarse ahora
Puede ser util disponer de su Password
Connection: close
Date: Mon, 10 Jul 2006 21:43:46 GMT
Server: Microsoft-IIS/6.0
Content-Type: text/html
Expires: Mon, 10 Jul 2006 21:43:46 GMT
Set-Cookie: geoid=AR; expires=Tue, 10-Jul-2007 03:00:00 GMT;
domain=.lanacion.com.ar; path=/
Set-Cookie: NacionHost%5FId=%7B68CE4741%2DF2DF%2D4361%2DADFE%2D24E21C5857B8%7D;
expires=Thu, 07-Jul-2016 03:00:00 GMT; domain=.lanacion.com.ar; path=/
Set-Cookie: ASPSESSIONIDAQTDCQBS=BBHPJCLAIIOGIDOPDJMNDA; path=/
Cache-control: private
-----

```

```

getContentType: text/html
getContentLength: -1
getContentEncoding: null
getDate: 1152567826000
getExpiration: 1152567826000
getLastModified: 0
-----

```

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
"http://www.w3.org/TR/REC-html40/loose.dtd">
<html lang="es" xml:lang="es">
  <head>
    <META NAME="Copyright" CONTENT="2006 LA NACION LINE">
      <META NAME="Copyright" CONTENT="2006 LANACION.com">
      <title>LANACION.com</title>

  <!--ONLINE-->
  . . .

```

Una pregunta habitual es si la plataforma Java soporta acceso a páginas web seguras. Si dentro de un applet abre una conexión a un URL https, dicho applet accederá a la página utilizando las ventajas de la implementación SSL del navegador. Sin embargo, antes del SDK 1.4, no existía soporte para un URL https. Necesitaba instalar una librería de extensión separada y el servicio criptográfico suministrado por el algoritmo RSA. A partir del SDK 1.4, el SSL se incluye como parte de la librería estándar.

java.net.URL

- **InputStream openStream()** Abre un flujo de entrada para la lectura de los datos del recurso.
- **URLConnection openConnection();**
Devuelve un objeto URLConnection que gestiona la conexión con el recurso.

java.net.URLConection

- **void setDoInput(boolean doInput)**

Si doInput es true implica que el usuario puede recibir datos desde este URLConnection.

- **void setDoOutput(boolean doOutput)**

Si doOutput es true implica que el usuario puede enviar datos a este URLConnection.

- **void setIfModifiedSince(long time)**

Configura este URLConnection para obtener sólo aquellos datos que se han modificado a partir de una fecha. Time se informa en segundos contados a partir del 1 de enero de 1970, GMT.

- **void setUseCaches(boolean useCaches)**

Si useCaches es true, los datos pueden recuperarse desde una caché local. Hay que indicar que el objeto URLConnection por sí mismo no mantiene nada parecido a una caché. Esta utilidad debe ser suministrada por un programa externo como un navegador.

- **void setAllowUserInteraction(boolean allowUserInteraction)**

Si `allowUserInteraction` es `true`, se puede solicitar una contraseña al usuario. El objeto `URLConnection` no facilita por sí mismo ningún sistema para realizar esta solicitud, la cual debe ser llevada a cabo por un navegador o un plug-in.

- **`void setRequestProperty(String clave, String valor)`**

Establece un campo de cabecera de la solicitud.

- **`Map getRequestProperties()`**

Devuelve un mapa de las parejas clave/valor de la cabecera de la petición. Este método fue incorporado en el SDK 1.4.

- **`void connect()`**

Conecta con el recurso remoto y recupera la información de la cabecera de respuesta.

- **`Map getHeaderFields()`**

Devuelve un mapa de las parejas clave/valor de la cabecera de la respuesta. Este método fue incorporado en el SDK 1.4.

- **`String getHeaderFieldKey(int n)`**

Obtiene el enesimo campo clave de la cabecera de respuesta, o `null` si `n` es `<= 0`, o mayor que el número de campo existentes en dicha cabecera.

- **`String getHeaderField(int n)`**

Recupera el enesimo valor de la cabecera de respuesta, o `null` si `n` es `<= 0`, o mayor que el número de campo existentes en dicha cabecera.

- **`int getContentLength()`**

Recupera el tamaño del contenido, en caso de estar disponible, o `-1` si no se conoce.

- **`String getContentType()`**

Recupera el tipo de contenido, como `text/plain` o `image/gif`.

- **`String getContentEncoding()`**

Recupera la codificación del contenido, ejemplo: `gzip`.

- **`long getDate()`**

- **`long getExpiration()`**

- **`long getLastModified()`**

Estos métodos recuperan la fecha de creación, de expiración y de la última modificación del recurso. Dichas fechas están especificadas en segundos transcurridos desde el 1 de enero de 1970, GMT.

- **`InputStream openInputStream()`**

- `OutputStream openOutputStream()`

Estos métodos devuelven, respectivamente, un flujo para leer desde el recurso o para escribir en él.

- `Object getObject()`

Selecciona el manipulador de contenido adecuado para leer los datos del recurso y convertirlo en un objeto.

Envío de formularios de datos

En la sección anterior vimos cómo leer datos desde un servidor web. Ahora le mostraremos cómo pueden sus aplicaciones enviar datos a dichos servidores y los programas que invocan éstos mediante los mecanismos CGI y servlet.

Scripts CGI y servlets

Antes de la aparición de la tecnología Java, ya existían mecanismos para la escritura de aplicaciones web interactivas. Para enviar información desde un navegador a un servidor, un usuario podía codificar un formulario usando HTML. Suponiendo que conocemos este lenguaje, lo hacemos y guardamos el archivo como `formulario.html`, con la codificación que aparece a continuación.

```
<HTML>
  <HEAD> <TITLE> Envíe su opinión </TITLE> </HEAD>
  <BODY>
    <H2>por favor, envíenos su opinion acerca de este sitio web</H2>
    <FORM ACTION action = "http://labsys.frc.utn.edu.ar:8080
      /servlet/servletopinion"METHOD="POST">
      Nombre:      <INPUT TYPE="TEXT" NAME="Nombres.." SIZE=15><BR>
      Apellidos:   <INPUT TYPE="TEXT" NAME="Apellidos" SIZE=30> <P>
      Opinión que le ha merecido este sitio web<BR>
      <INPUT TYPE="RADIO" CHECKED NAME="opinion"
                                VALUE="Buena">Buena<BR>
      <INPUT TYPE="RADIO"
                                NAME="opinion"
                                VALUE="Regular">Regular<BR>
      <INPUT TYPE="RADIO"
                                NAME="opinion"
                                VALUE="Mala">Mala<p>
      Comentarios <BR>
      <TEXTAREA NAME="comentarios" ROWS=6 COLS=40>
      </TEXTAREA><P>
      <INPUT TYPE="SUBMIT" NAME="botonEnviar" VALUE="Enviar">
      <INPUT TYPE="RESET" NAME="botonLimpiar" VALUE="Limpiar">
    </FORM>
  </BODY>
</HTML>
```

Usando un navegador local nos posicionamos sobre `formulario.html` y <<click>>, el formulario aparece bajo el entorno de nuestro navegador web. Lo que vemos:

por favor, envíenos su opinion acerca de este sitio web

Nombre:

Apellidos:

Opinión que le ha merecido este sitio web



Buena

- ☐ Regular
- ☐ Mala

Comentarios

El formulario pretende evaluar la opinión de los visitantes de un sitio Web. Contiene dos campos de tipo TEXT donde el visitante introducirá su nombre y apellidos. A continuación, deberá indicar la opinión que le merece la página visitada eligiendo una entre tres posibles (Buena, Regular y Mala). Por último, se ofrece al usuario la posibilidad escribir un comentario si así lo considera oportuno.

Es necesario asignar un identificador único (es decir, un valor de la propiedad NAME) a cada uno de los campos del formulario, ya que la información que reciba el servlet estará organizada en forma de pares de valores, donde uno de los elementos de dicho par será un String que contendrá el nombre del campo. Así, por ejemplo, si se introdujera como nombre del visitante "Mikel", el servlet recibiría del browser el par nombre=Mikel, que permitirá acceder de una forma sencilla al nombre introducido mediante el método `getParameter()`. Por este motivo es importante no utilizar nombres duplicados en los elementos de los formularios.

Por otra parte puede observarse que en el tag `<FORM>` se han utilizado las propiedades ACTION y METHOD.

El método (METHOD) utilizado para la transmisión de datos es el método HTTP POST. También se podría haber utilizado el método HTTP GET, pero este método tiene algunas limitaciones en cuanto al volumen de datos transmisible, por lo que es recomendable utilizar el método POST. Mediante la propiedad ACTION deberá especificarse el URL del servlet que debe procesar los datos. Este URL contiene, en el ejemplo presentado, las siguientes características:

El servlet se encuentra situado en un servidor cuyo nombre es `miServidor`. Este nombre dependerá del ordenador que proporcione los servicios de red. La forma de saber cuál es el nombre de dicho ordenador y su dirección IP es acudiendo al Panel de Control (Control Panel) de Windows. Clickeando en Red (Network), se accede a las propiedades de la red (obviamente, si se quiere utilizar servlets, será necesario tener instalados los drivers de red, en concreto los drivers de TCP/IP que vienen con Windows 95/98/NT). Dentro de la lengüeta Protocolos (Protocols) se escoge TCP/IP y se clickea en Propiedades (Properties). Aparecerá un nuevo cuadro de diálogo en el que habrá que seleccionar la lengüeta DNS (Domain Name System). Allí se encuentra definido el nombre del ordenador (Host Name) así como su dominio (Domain Name).

En cualquier caso, para poder hacer pruebas, se puede utilizar el como nombre de servidor el host local o `localhost`. cuyo número IP es `127.0.0.1`. Por ejemplo, se podría haber escrito (aunque estos servicios no serían accesibles desde el exterior de la Regional Córdoba):

```
<FORM ACTION="http://localhost:8080/servlet/Servletopinion" METHOD="POST">
```

El servidor HTTP está "escuchando" por el puerto el puerto 8080. Todas las llamadas utilizando dicho puerto serán procesadas por el módulo del servidor encargado de la gestión de los servlets.

En principio es factible la utilización de cualquier puerto libre del sistema siempre que se indique al servidor HTTP cuál va a ser el puerto utilizado para dichas llamadas. Esto último debe ser configurado por el administrador del sistema.

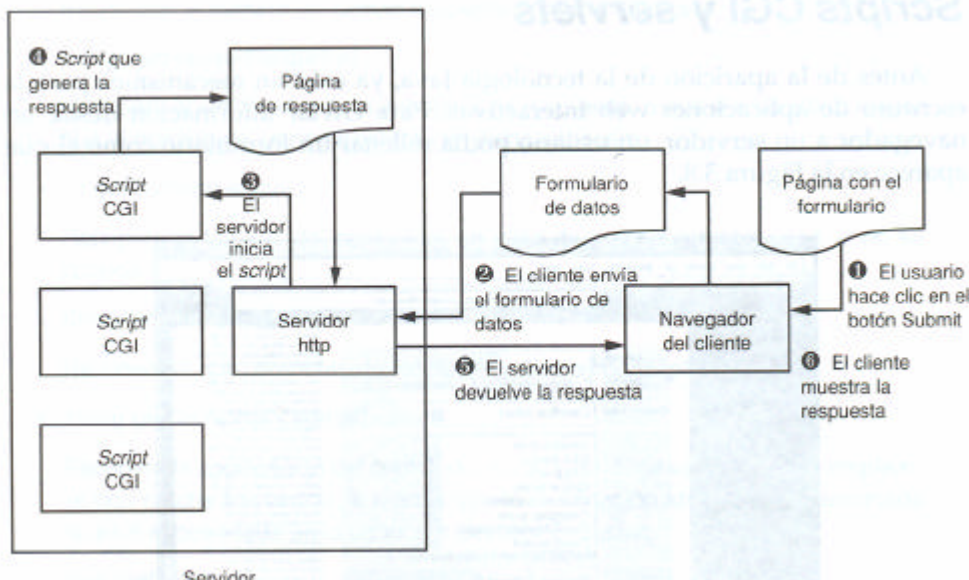
El servlet se encuentra situado en un subdirectorio (virtual) del servidor llamado `servlet`. Este nombre es en principio opcional, aunque la mayoría de los servidores lo utilizan por defecto.

El nombre del servlet empleado es `ServletOpinion`, y es éste el que recibirá la información enviada por el cliente al servidor (el formulario en este caso), y quien se encargará de diseñar la respuesta, que pasará al servidor para que este a su vez la envíe de vuelta al cliente.

En forma muy semejante este formulario podría haber sido procesado por un script CGI (Common Gateway Interface). Ampliemos esto un poco.

Una vez que el usuario ha hecho click en el botón enviar, el texto contenido en los campos y las opciones marcadas en las casillas de verificación y botones de radio se envían al servidor para ser procesados por un script CGI (CGI significa Common Gateway Interface, Interfaz de pasarela común). El script a emplear viene definido por el atributo `ACTION` de la etiqueta `FORM`.

El script CGI es un programa que reside en la computadora servidor. Normalmente, suele haber muchos de estos scripts en un servidor, y, por lo general, residiendo en el directorio `cgi-bin`. El servidor web lanza el script CGI y le suministra el formulario de datos, formulario que es procesado por aquél para producir otra página HTML que el servidor devuelve al navegador del cliente. Esta secuencia se ilustra en la Figura a continuación. La página de respuesta puede contener información nueva (como, por ejemplo, el resultado de una búsqueda en una base de datos) o sólo un reconocimiento de una operación realizada.



Flujo de datos durante la ejecución de un script CGI.

Los programas CGI suelen estar escritos en Perl, aunque también pueden estarlo en cualquier otro lenguaje que pueda leer desde una entrada estándar y escribir en una salida del mismo tipo.

Los CGI son bastante estables y los administradores del sistema suelen estar familiarizados con ellos. Pero tienen una gran desventaja. Cada petición genera un nuevo proceso y no un nuevo thread. Además, es complicado controlar la seguridad de estos scripts. La nueva tecnología servlet supera ambas

desventajas. Los motores servlets usan Java para iniciar cada uno de ellos en un thread aparte y para controlar sus privilegios de seguridad. Primero vemos algo de tecnología CGI y luego ejemplos de Servlets, más adelante en este capítulo.

Envío de datos a un servidor web

Cuando los datos se han enviado a un servidor web, **no importa si dichos datos son procesados por un script CGI o por un servlet**. El cliente los envía en un formato estándar, y es el servidor web el que debe encargarse de pasados al programa que debe generar la respuesta.

Existen dos métodos de pasar datos a un servidor web: GET y POST.

En el método GET, los parámetros se adjuntan al final del URL, que tendrá entonces el siguiente aspecto:

```
http://servidor/script?parámetros
```

Todos los espacios en blanco están sustituidos con un signo +, y los caracteres no alfanuméricos por un signo % seguido de un número hexadecimal de dos dígitos. Por ejemplo, para enviar el nombre de la calle S. Main, debe usar S%2e+Main, ya que el carácter "." tiene el código ASCII 2e (o 46 en decimal). Esta codificación evita que cualquier programa intermedio pueda complicar su ejecución a la hora de procesar los espacios en blanco y cualquier otro carácter especial. Este esquema de codificación suele recibir el nombre de codificación URL. Ya que estamos en esto, digamos que esta codificación ya la hemos hecho, mejor dicho, la ha hecho el método

```
String encoding = URLEncoder.encode(input);
En public class URLConnectionTest, (pg 22)
```

Por ejemplo, para obtener un mapa de 1 Infinite Loop, Cupertino, CA, basta con escribir el siguiente URL:

<http://maps.yahoo.com/py/maps.py?addr=1+Infinite+Loop&csz=Cupertino+CA>

Si lo ejecutamos en un browser web, efectivamente aparece un mapa de Cupertino. Hay una calle en forma de óvalo, supongo esa debe ser el loop infinito, clickeando sobre un icono en forma de estrella se nos ofrece mas información ... muchas gracias.

El método GET es simple. Sin embargo muchos navegadores están limitados en el número de caracteres que se pueden incluir en una petición GET.

En el método POST, los parámetros no se adjuntan al URL. En su lugar, se obtiene un flujo de salida desde el URLConnection y se escriben en él las parejas nombre/valor. La codificación URL de los valores sigue siendo necesaria, así como la separación de las parejas por caracteres &.

Vamos a ver este proceso con más detalle. Para "pegar" datos a un script, lo primero que hay que hacer es establecer un URLConnection.

```
URL url = new URL(nhttp://servidor/script");
URLConnection connection = url.openConnection();
```

A continuación, llame al método setDoOutput para configurar la salida de la conexión.

```
connection.setDoOutput(true);
```


Después, hay que invocar a `getOutputStream` para obtener el flujo a través del cual se enviarán los datos al servidor. Si está mandando texto, es conveniente envolver dicho flujo en un `PrintWriter`.

```
PrintWriter out = new PrintWriter(connection.getOutputStream());
```

Ahora, ya está preparado para enviar los datos al servidor:

```
out.print(name1 + "=" + URLEncoder.encode(value1) + "&");
out.print(name2 + "=" + URLEncoder.encode(value2));
```

Por último, cierre el flujo: `out.close()`;

La respuesta del servidor se lee del modo habitual.

```
BufferedReader in = new BufferedReader(new
InputStreamReader(connection.getInputStream()));
String line;
while (line = in.readLine()) != null){código a procesar}^
```

Vamos a ver un ejemplo práctico, algo que la cátedra tiene funcionando desde 2003. Los alumnos, usando una interface gráfica, se comunican con un programa cgi instalado en el sitio labsys. Los alumnos establecen una comunicación post con un programa gateway cgi quien graba en un `RandomAccessFile` los datos, y responde a los alumnos enviandoles una página. Posteriormente el profe, usando otro juego de programas, da por aprobado el práctico a los alumnos que registraron su entrada. Vamos con el programa cliente.

```
// Cliente del Gateway Prac001
```

```
// Autor: Tymoschuk, Jorge
```

```
import java.io.*;
import java.net.*;
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.Graphics;
import java.security.*;
public class ClieGate extends JFrame{
    boolean trace;
    String parametros;
    String textTip, textDec;
    JLabel etiqueta;
    JTextField curso, legajo, nombres;
    JTextArea frase;
    JButton procesar, salir;
    int x, y, w, h;
    public ClieGate(boolean trazar){
        x=10; y=240;    // Coordenadas de la etiqueta para report
        w=380; h=20;    // Tamaño de la idem
        trace = trazar;
        Graphics g;
    }

    public void iniciar() {
        try { // usando camiseta Windows
            javax.swing.UIManager.setLookAndFeel(
                "com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
        }
    }
}
```

```
} catch (Exception e) {}

// elimina la distribución automática de componentes
this.getContentPane().setLayout(null);
this.setSize(420,520);
// creación y configuración de etiquetas y campos de texto
etiqueta = new JLabel("Practico de Entorno de Red");
etiqueta.setSize(300, 20);
etiqueta.setLocation(100, 10);
this.getContentPane().add(etiqueta);

etiqueta = new JLabel("Curso");
etiqueta.setSize(50, 20);
etiqueta.setLocation(100, 50);
this.getContentPane().add(etiqueta);

curso = new JTextField(3);
curso.setSize(30, 20);
curso.setLocation(150, 50);
this.getContentPane().add(curso);

etiqueta = new JLabel("Legajo");
etiqueta.setSize(60, 20);
etiqueta.setLocation(220, 50);
this.getContentPane().add(etiqueta);

legajo = new JTextField(5);
legajo.setSize(45, 20);
legajo.setLocation(270, 50);
this.getContentPane().add(legajo);

etiqueta = new JLabel("Apellido, Nombres");
etiqueta.setSize(120, 20);
etiqueta.setLocation(50, 80);
this.getContentPane().add(etiqueta);

nombres = new JTextField(20);
nombres.setSize(270, 20);
nombres.setLocation(50,100);
this.getContentPane().add(nombres);

etiqueta = new JLabel("Algo que quiero decir");
etiqueta.setSize(120, 20);
etiqueta.setLocation(50, 130);
getContentPane().add(etiqueta);

frase = new JTextArea(3,40);
frase.setSize(270, 50);
frase.setLocation(50, 150);
frase.setLineWrap(true);
this.getContentPane().add(frase);

ActionListener escuchaAccion = new Eventos();

// creación y configuración del boton Procesar
procesar = new JButton("Procesar");
procesar.setSize(90, 27);
procesar.setLocation(50,220);
this.getContentPane().add(procesar);
procesar.addActionListener(escuchaAccion);
```

```

// creación y configuración del boton salir
salir = new JButton("Salir");
salir.setSize(60, 27);
salir.setLocation(260,220);
this.getContentPane().add(salir);
salir.addActionListener(escuchaAccion);
show();
} // iniciar()

void leerTodo(){
    parametros = " "+"tymos"+
        " "+"regPra01"+
        " "+"curso.getText()+
        " "+"legajo.getText()+
        " "+"nombres.getText();

    parametros = URLEncoder.encode(parametros);
    if (trace) report(parametros);
    textTip = frase.getText();
    textDec = URLEncoder.encode(textTip);
    if (trace) report(textDec);
}

void procFrase(){
    if (trace) report("117 - En tramite de conexion ...");
    try {
        URL urlObj=new URL("http://labsys.frc.utn.edu.ar/cgi-
            bin/java.cgi?Prac001"+parametros);
        URLConnection urlObjConnect = urlObj.openConnection();
        urlObjConnect.setDoOutput(true);
        if (trace) report("122 - Conexion establecida ...");
        PrintWriter out = new PrintWriter(urlObjConnect.getOutputStream());
        if (trace) report("124 - Objeto out instanciado...");
        out.println("string=" + textDec);
        if (trace) report("126 - Mensaje enviado, paciencia por favor ...");
        out.close();
        if (trace) report("128 - Flujo de salida out cerrado");
        // vamos a definir el flujo de retorno paso a paso
        InputStream inStrm = urlObjConnect.getInputStream();
        if (trace) report("131 - Objeto flujo de entrada inStrm
            instanciado");
        InputStreamReader inStrmRd =new InputStreamReader(inStrm);
        if (trace) report("133 - Objeto InputStreamReader inStrmRd
            instanciado");
        BufferedReader in = new BufferedReader(inStrmRd);
        if (trace) report("135 - BufferedReader in instanciado, ya llega
            ...");
        /* Definimos un objeto in, de tipo BufferedReader cuya mision es
            posibilitar la lectura de la salida textual generada por
            cgi-bin/Prac001 */
        String inputLine;
        if (trace) report("140 - Preparandonos a la recepcion ...");
        while ((inputLine = in.readLine()) != null)
            report(inputLine);
        if (trace) report("=== 143 - Recepcion de datos concluida ===");
        in.close();
    }catch (MalformedURLException exce){
        report("Lamento, MalformedURLException");
    }catch (IOException exce){

```

```

        report("URLConnection, IOException");
        report("PrintWriter out, IOException");
    } catch (AccessControlException exce){
        report("Lamento, AccessControlException");
        report("Tema de socket, controlado en java.sun.com ...");
    }
} // void procFrase()

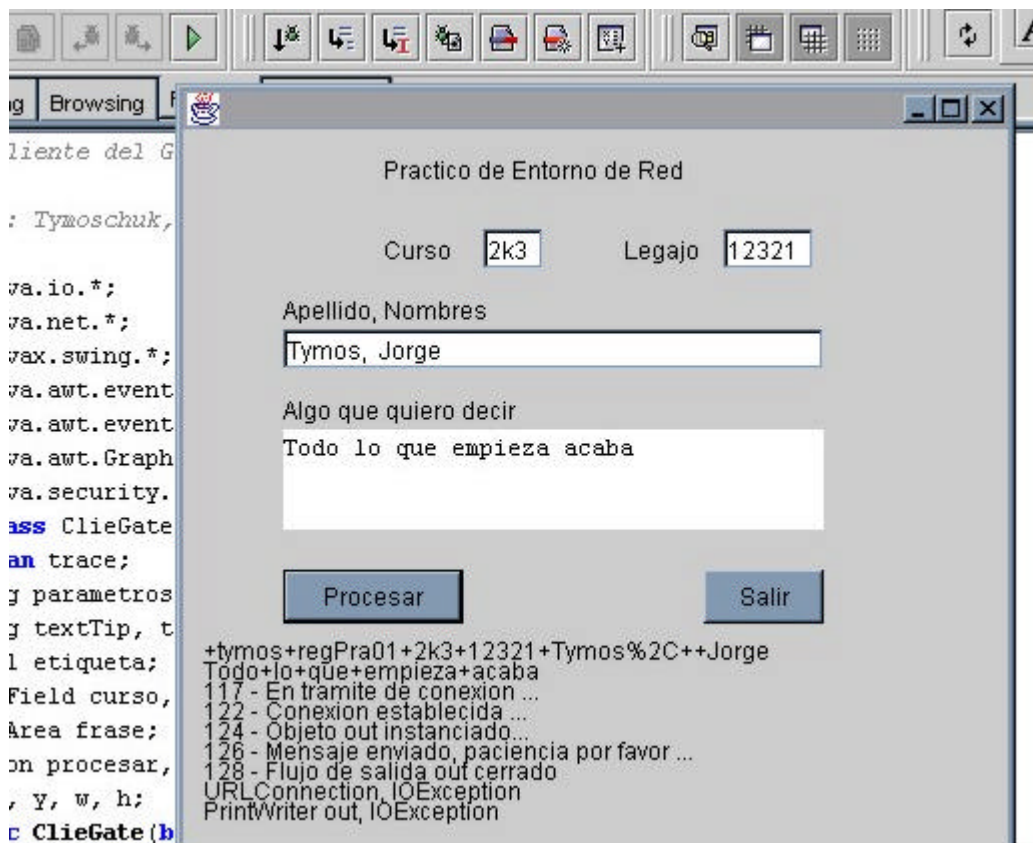
void report(String msge){
    etiqueta = new JLabel(msge);
    etiqueta.setSize(w,h);
    etiqueta.setLocation(x,y+=10);
    this.getContentPane().add(etiqueta);
    repaint();
}

class Eventos implements ActionListener{
    public void actionPerformed(ActionEvent e){
        if(e.getSource() == procesar){
            leerTodo();
            try{
                procFrase();
            } catch (Exception exce){report("Lamento, some Exception ...");}
        } // if (e.getSource())
        else System.exit(1);
    } // actionPerformed
} // class Eventos

public static void main(String args[]) {
    ClieGate clieGate=new ClieGate(true);
    clieGate.iniciar();
} // main()
} // class Clie001

```

Capturemos la ejecución



Por lo que vemos, ha fracasado la URLConection.
 Comencemos por investigar si en la carpeta cgi-bin del servidor
 labsys.frc.utn.edu.ar todavía existe un gateway java Parc001.class
 Mandamos mail al nacho. Luego les contamos que pasó ...

El gateway, o sea el programa instalado en la carpeta cgi-bin del servidor del Laboratorio de sistemas sigue a continuación. Nuevamente hemos necesitado hacer un par de toques en la codificación por estarlo probando localmente, con un sistema operativo Windows XP, mientras que en el servidor del labo funciona bajo Linux.

```
// Gateway para el primer practico de Trabajo en Red

// Author: Tymoschuk, Jorge

import java.io.*;
import java.net.*;
import java.util.StringTokenizer;

public class PracGate{ //
    private File      file;
    private RandomAccessFile ioFile;
    private String legLei, curLei, aNomLei;
    private String algoDecir="";
    private String direct, archivo, curso, legajo, apeNoms;
    public PracGate(String[] args){
        direct   = args[0];    // Directorio (tymos)
        archivo  = args[1];    // Archivo      (regPra01)
        curso    = args[2];    // Donde cursas ?
        legajo    = args[3];    // Legajo UTN
        apeNoms   = args[4];    // Apellido, Nombres
    }

    boolean errorParm(String[] args) throws IOException{
        String msgeAux = null;
        if ( args.length != 5){ // Cantidad de parametros difiere
            System.out.println("Content-Type: text/html"); // 1st line
            System.out.println(); // empty line
            System.out.println("<html><H1> Error en parametros </H1>");
            msgeAux = "cantidad recibida " + args.length;
            System.out.println("<P> msgeAux<P></html>");
            System.out.println("<P>difiere de la esperada (5)<P></html>");
            return true;
        }
        if ( !args[0].equals("tymos")){ // Directorio mal informado

            System.out.println("Content-Type: text/html"); // 1st line
            System.out.println(); // empty line
            System.out.println("<html><H1> Error en parametros </H1>");
            System.out.println("<P>args[0] distinto de tymos <P></html>");
            return true;
        }
        if ( !args[1].equals("regPra01")){ // Archivo mal informado
            System.out.println("Content-Type: text/html"); // 1st line
            System.out.println(); // empty line
            System.out.println("<html><H1> Error en parametros </H1>");
            System.out.println("<P>args[1] difiere de regPra01 <P></html>");
            return true;
        }
    }
}
```

```

        return false;
    }

    boolean openRAF(){
        // file = new File("./"+direct+"/"+archivo);
        /*  Comentarizamos la linea anterior, ocurre que el servidor
         *   labsys en una maquina Linux, y ahora estamos probando
         *   generar el archivo en la misma carpeta de la clase
         *   en una PC Windows XP                                     */
        file = new File(archivo+".dat");
        try    {iofile = new RandomAccessFile(file,"rw");
                return true;
            }catch (IOException e){
                System.out.println("Content-Type: text/html"); // 1st line
                System.out.println(); // empty line
                System.out.println("<html><H1> Error en generacion del RAF </H1>");
                System.out.println("<P>openRAF(), IOException<P></html>");
                return false;
            }
    } //    openRAF()

    boolean existeLeg(){
        if(file.length()==0) return false; // primera vez
        String legLei="00.000";
        try{
            while(true){
                if(legLei.equals(legajo)){ // Encontre igual
                    System.out.println("Content-Type: text/html");
                    // 1st line
                    System.out.println();
                    // empty line
                    System.out.println("<html><H1> Existes !!</H1>");
                    System.out.println("<P>"+apeNoms+"<P></html>");
                    return true;
                }
                legLei= iofile.readUTF(); // legajo
                curLei= iofile.readUTF(); // curso
                aNomLei=iofile.readUTF(); // nombres
                iofile.readUTF(); // algo que quiero decir
            }
        }catch(EOFException e){return false;} // No tengo legajo
        catch(IOException e){
            System.out.println("Content-Type: text/html"); // 1st line
            System.out.println(); // empty line
            System.out.println("<html><H1> Error de lectura (?) </H1>");
            System.out.println("<P>existeLeg(), IOException<P></html>");
            System.exit(1);
        }
        return true;
    } //    existeLeg()

    void grabRAF(){
        try{
            iofile.writeUTF(legajo);
            iofile.writeUTF(curso);
            iofile.writeUTF(apeNoms);
            // Ahora leemos el flujo enviado por el cliente.
            int c;
            while ((c = System.in.read() ) > -1 )
                algoDecir+=c;
        }
    }

```

```

        iofile.writeUTF(algoDecir);
        iofile.close();
    } catch (IOException e){
        System.out.println("Content-Type: text/html"); // 1st line
        System.out.println(); // empty line
        System.out.println("<html><H1> Error de grabacion </H1>");
        System.out.println("<P>"+apeNoms+"<P></html>");
    }
} // grabRAF()

void practOk(){
    // confirmacion del exito logrado
    System.out.println("Content-Type: text/html"); // 1st line
    System.out.println(); // empty line
    System.out.println("<html><H1> Practico exitoso !!! </H1>");
    System.out.println("<P>"+apeNoms+"<P></html>");
}

public static void main (String[] args) throws IOException{
    // PracGate gtw = new PracGate(args);
    /* Como no sabemos que pasa en labsys, necesitamos probar localmente.
     * Entonces comentamos la linea anterior, construimos un array de
     * Strings con lo que vendria por el flujo out del cliente ClieGate
     */

    String[] argsAux = {"tymos","regPra01","2k3","12321","Tymos, Jorge"};

    PracGate gtw = new PracGate(argsAux);
    // if (gtw.errorParm(args))return;

    if (gtw.errorParm(argsAux))return;

    // Consistencia de argumentos fue correcta, seguimos
    if (gtw.openRAF()) // Consigo abrir Random Access File
        if(!gtw.existeLeg()){ // No tengo legajo, darlo de alta
            gtw.grabRAF();
            gtw.practOk();
        }
    } // main()
} // class

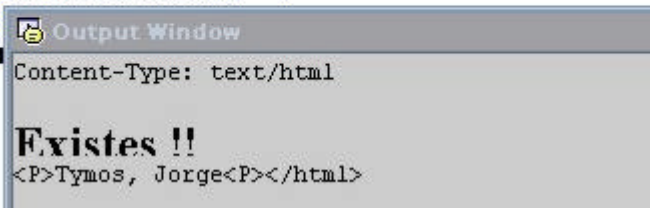
```

Veamos un capture de la segunda vez que estoy ejecutando PractGate:

```

class PracGate{ //
    private File file;
    private RandomAccessFile iofile;
    private String legLei, curLei, aNomLei;
    private String algoDecir="";
    private
    public

```



Es correcto, en la primera ejecución me registró.
 En la segunda me encuentra y me informa.
 La idea es que el alumno pueda verificar el éxito de su operación.

Recuperación de información desde un servidor

El último ejemplo mostró la forma de leer datos desde un servidor web. Pero Internet contiene mucha información, de la cual una parte puede interesarnos y otra no. Y es esta falta de control en la información de lo que se quejan muchos usuarios web. Una de las promesas de la tecnología Java es que podía ayudar a poner algo de orden en este caos: se pueden emplear applets para recuperar la información y presentada al usuario en un formato atractivo.

Existen muchos posibles usos. Por ejemplo:

- Un applet puede inspeccionar todas las páginas web que el usuario haya especificado como interesantes y buscar cuál de ellas ha cambiado recientemente.
- Un applet puede visitar las páginas web de todas las líneas aéreas para buscar aquella que ofrezca las mejores ofertas.
- Los applets pueden recopilar y mostrar cotizaciones recientes de acciones, tasas de cambio de moneda y otro tipo de información financiera.
- Los applets pueden buscar en una FAQ, notas de prensa, artículos, etc. y devolver el texto que contenga ciertas palabras clave.

Vamos a ver un sencillo ejemplo de applet. El National Weather Service almacena los partes meteorológicos en distintos archivos dentro del servidor <http://iwin.nws.noaa.gov>. Por ejemplo, se almacena una previsión por horas de California en:

<http://iwin.nws.noaa.gov/iwin/ca/hourly.html>

Otros directorios contienen información similar.

Nuestro applet presenta al usuario dos listas, una con los Estados y otra con los tipos de informes. Si hace clic en Get report, el applet toma el informe correspondiente y lo coloca en un área de texto (ver figura).

El código del applet es sencillo, aunque un poco extenso. La única acción interesante ocurre en el método `getWeather`. Dicho método construye primero la petición. Obtiene el URL base (<http://iwin.nws.noaa.gov/iwin/>) a través del parámetro `queryBase` de la etiqueta `applet`. Después, añade el estado, el tipo de informe y la extensión `.html`:

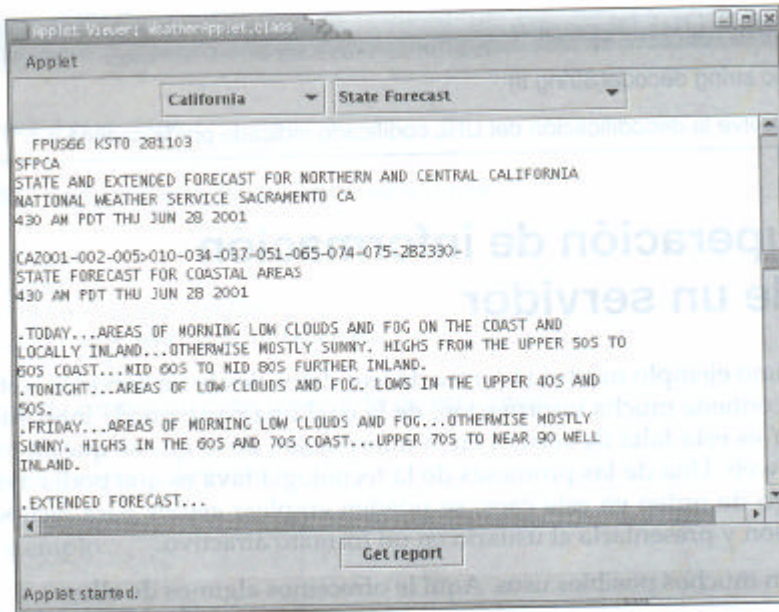
```
String queryBase = getParameter("queryBase");
String query = queryBase + state + "/" + report + ".html";
```

El siguiente paso consiste en crear un objeto URL y llamar a `openStream`.

```
URL url = new URL(query);
BufferedReader in = new BufferedReader(new InputStreamReader(url.openStream()));
```

El resto del método simplemente lee el archivo, elimina las etiquetas HTML y coloca dicho archivo en el área de texto.

```
String line;
while ((line = in.readLine()) != null)
    weather.append(removeTags (line) + '\n');
```

El código de WeatherApplet.java

```
import java.net.*;
import java.io.*;
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import javax.swing.*;

/**
 * Este applet exhibe datos meteorológicos obtenidos del servidor NOA,
 * En formato texto.
 */
public class WeatherApplet extends JApplet{
    public void init(){
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());

        // Prepara listas de estados y tipos de informe
        JPanel comboPanel = new JPanel();
        state = makeCombo(states, comboPanel);
        report = makeCombo(reports, comboPanel);
        contentPane.add(comboPanel, BorderLayout.NORTH);

        // Incorpora el area de texto para exhibir datos
        weather = new JTextArea(20, 80);
        weather.setFont(new Font("Courier", Font.PLAIN, 12));

        // Incorpora boton del evento informe
        contentPane.add(new JScrollPane(weather),
            BorderLayout.CENTER);
        JPanel buttonPanel = new JPanel();
        JButton reportButton = new JButton("Get report");
        reportButton.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent evt)
```

```

        {
            weather.setText("");
            new
                Thread()
            {
                public void run()
                {
                    getWeather(getItem(state, states),
                                getItem(report, reports));
                }
            }.start();
        }
    });

    buttonPanel.add(reportButton);
    contentPane.add(buttonPanel, BorderLayout.SOUTH);
}

/**
 * Construyendo la primer lista desplegable
 * @param items es un array de cadenas. Cada fila contiene
 * dos elementos:
 * . Entrada tipo texto con el nombre del estado
 * . petición para satisfacer la entrada
 * @param parent es el objeto padre al cual se añade la lista
 */
public JComboBox makeCombo(String[][] items, Container parent){
    JComboBox combo = new JComboBox();
    for (int i = 0; i < items.length; i++)
        combo.addItem(items[i][0]);
    parent.add(combo);
    return combo;
}

/**
 * Toma la petición desde la lista de estados.
 * @param box es la lista desplegable
 * @param items es el array de cadenas detallado en el pto anterior
 * @return una cadena con los datos que satisfacen la petición.
 */
public String getItem(JComboBox box, String[][] items){
    return items[box.getSelectedIndex()][1];
}

/**
 * Une el URL de la petición y exhibe la cadena de datos del pto
 * Anterior en el area de datos.
 * @param state es el estado que nos interesa
 * @param report es el informe respectivo
 */
public void getWeather(String state, String report){
    String r = new String();
    try
    {
        String queryBase = getParameter("queryBase");
        String query
            = queryBase + state + "/" + report + ".html";
        URL url = new URL(query);
        BufferedReader in = new BufferedReader(new
            InputStreamReader(url.openStream()));
    }
}

```

```

        String line;
        while ((line = in.readLine()) != null)
            weather.append(removeTags(line) + "\n");
    }
    catch(IOException e)
    {
        showStatus("Error " + e);
    }
}

/**
    Elimina todas las etiquetas HTML de la cadena retornada en pto anterior
    @param s es la cadena retornada
    @return s la misma cadena, sin las etiquetas HTML
*/
public static String removeTags(String s){
    while (true){
        int lb = s.indexOf('<');
        if (lb < 0) return s;
        int rb = s.indexOf('>', lb);
        if (rb < 0) return s;
        s = s.substring(0, lb) + " " + s.substring(rb + 1);
    }
}

private JTextArea weather;
private JComboBox state;
private JComboBox report;

private String[][] states = {
    { "Alabama", "al" },
    { "Alaska", "ak" },
    { "Arizona", "az" },
    { "Arkansas", "ar" },
    { "California", "ca" },
    { "Colorado", "co" },
    { "Connecticut", "ct" },
    { "Delaware", "de" },
    { "Florida", "fl" },
    { "Georgia", "ga" },
    { "Hawaii", "hi" },
    { "Idaho", "id" },
    { "Illinois", "il" },
    { "Indiana", "in" },
    { "Iowa", "ia" },
    { "Kansas", "ks" },
    { "Kentucky", "ky" },
    { "Louisiana", "la" },
    { "Maine", "me" },
    { "Maryland", "md" },
    { "Massachusetts", "ma" },
    { "Michigan", "mi" },
    { "Minnesota", "mn" },
    { "Mississippi", "ms" },
    { "Missouri", "mo" },
    { "Montana", "mt" },
    { "Nebraska", "ne" },
    { "Nevada", "nv" },
    { "New Hampshire", "nh" },

```

```

        { "New Jersey", "nj" },
        { "New Mexico", "nm" },
        { "New York", "ny" },
        { "North Carolina", "nc" },
        { "North Dakota", "nd" },
        { "Ohio", "oh" },
        { "Oklahoma", "ok" },
        { "Oregon", "or" },
        { "Pennsylvania", "pa" },
        { "Rhode Island", "ri" },
        { "South Carolina", "sc" },
        { "South Dakota", "sd" },
        { "Tennessee", "tn" },
        { "Texas", "tx" },
        { "Utah", "ut" },
        { "Vermont", "vt" },
        { "Virginia", "va" },
        { "Washington", "wa" },
        { "West Virginia", "wv" },
        { "Wisconsin", "wi" },
        { "Wyoming", "wy" }
    };

    private String[][] reports = {
        { "Hourly (State Weather Roundup)", "hourly" },
        { "State Forecast", "state" },
        { "Zone Forecast", "zone" },
        { "Short Term (NOWCASTS)", "shortterm" },
        { "Forecast Discussion", "discussion" },
        { "Weather Summary", "summary" },
        { "Public Information", "public" },
        { "Climate Data", "climate" },
        { "Hydrological Data", "hydro" },
        { "Watches", "watches" },
        { "Special Weather Statements", "special" },
        { "Warnings and Advisories", "allwarnings" }
    };
}

```

<HTML>

<HEAD>

<TITLE>Weather Report Applet</TITLE>

</HEAD>

<BODY>

<H1>A Weather Report Applet </H1>

<P>Selecione un estado y tipo de reporte y clickee boton <Get report>;

Ud obtendra un reporte actualizado, cortesía de la
National Oceanic and Atmospheric Administration. </P>

<P>En este applet no se ha puesto énfasis en su "belleza", sino en demostrar
como un applet puede recuperar información desde otro servidor Web</P>

<HR>

<APPLET CODE="WeatherApplet.class" WIDTH="600" HEIGHT="400">

```

        <PARAM NAME="queryBase" VALUE="http://iwin.nws.noaa.gov/iwin/">
    </APPLET>
    <HR>
    <UL>
        <LI><A HREF="WeatherApplet.java">The source.</A> </LI>
        <LI><A HREF="ProxySrv.java">The source for the proxy server.</A> </LI>
    </UL>
</BODY>
</HTML>

```

Puede que la ejecución de este applet le decepcione. Tanto el "AppletViewer" (visor de applets) como el navegador se niegan a ejecutarlo. Siempre que haga <clic> en Get report, se genera una violación de seguridad. Veamos como forzar al visor de applets a que lo ejecute.

Primero, cree un archivo de texto con el siguiente contenido:

```

Grant{
    permission java.net.SocketPermission
        "iwin.nws.noaa.gov:80", ".connect";
};

```

Invoke al WeatherApplet.policy. Este simplemente concede permiso al applet para que se conecte al servidor iwin.nws.noaa.gov por el puerto 80, el que corresponde a HTTP. El porqué de este permiso es el tema de la siguiente sección.

Iniciemos el visor de applets con el siguiente comando

```

appletviewer -J-D java"security.policy=WeatherApplet.policy
    WeatherApplet.html

```

La opción -J del appletviewer pasa los argumentos de la línea de comandos al intérprete Java de bytecodes. La opción -D de dicho intérprete ajusta el valor de una propiedad del sistema, en este caso, la correspondiente a java.security.policy con el nombre de archivo de políticas que contiene los permisos para este programa.

Si quiere ejecutar este applet en un navegador, primero debe firmarlo. A partir de entonces usted, y lo que es más importante, nadie más que quiera ejecutarlo, podrá llamar al navegador:

- Para verificar la firma (a menos que esté garantizado por una entidad certificadora segura).
- Para estar de acuerdo con los privilegios solicitados.

En este material no hemos incluido el tema de la firma de applets. Puede verlo en el capítulo 9 de java 2, vol 2, de cay C. Hortman/ Gary Cornell, que trata del tema de seguridad (permisos, certificados, encriptación,...)

Seguridad de los applets

Como ya ha visto, su applet tiene prohibida una operación tan simple como la de conectar con el puerto 80 de un servidor web. Veamos la razón de esta prohibición y describiremos cómo desarrollar el applet sin que sus usuarios tengan que preocuparse por los certificados.

Los navegadores web sólo permiten que un applet lea y escriba datos en el servidor que suministra dicho applet, y sólo pueden conectar con sockets en la computadora desde la que provienen. Se suele decir que los " applets sólo pueden llamar a casa".

Al principio, esta restricción no tenía mucho sentido. ¿Por qué negarle al applet una operación que el navegador está cansado de hacer continuamente como es la recuperación de datos desde un servidor web? Visualizar los tres servidores involucrados le ayudará a entender el motivo. Véase en la siguiente figura

- El servidor " originador" (su computadora, que entrega la página web y los applets Java a los clientes);
- El servidor local (la máquina del usuario que ejecuta su applet);
- La tercera parte es el repositorio de datos que su applet quiere ver.

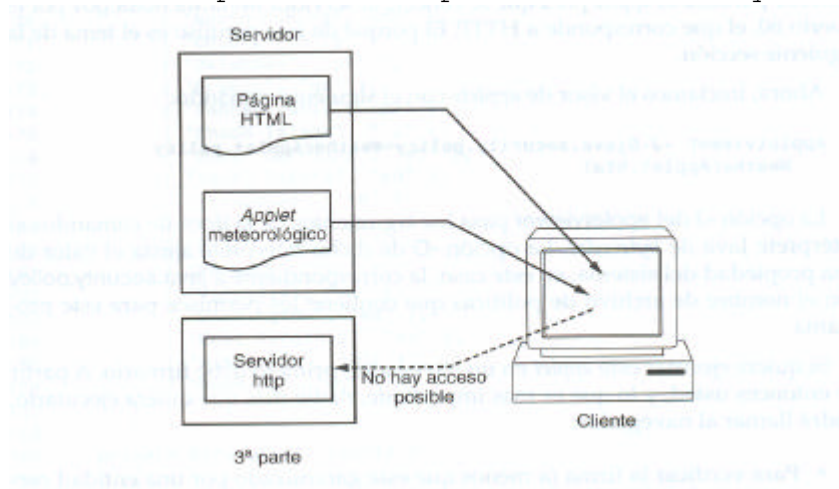


Figura. La seguridad de un applet deniega la conexión con una tercera parte.

La regla de seguridad de los applets dice que éstos sólo pueden leer y escribir datos en los servidores que los originaron. Desde luego, tiene sentido el que un applet no pueda escribir en la máquina local. Si pudiera, podría infectarla con algún virus o alterar archivos importantes. Después de todo, el applet se ejecuta inmediatamente una vez que el usuario se topa con nuestra página web, y éste debe estar protegido contra cualquier daño producido por applets maliciosos o mal diseñados.

También tiene sentido prohibir que un applet lea datos de la máquina local. De otro modo, podría obtener información sensible de los archivos de la computadora local (como números de tarjetas de crédito), abrir una conexión socket con el servidor que alberga el applet y escribir dicha información. Así, usted podría abrir una atractiva página web e interactuar con un applet que haga algo divertido o útil sin tener ni idea de lo que dicho applet puede estar haciendo en otros threads.

Por tanto, el navegador deniega por completo cualquier acceso de un applet a los archivos de su computadora.

Pero, ¿por qué no puede leer un applet archivos de la Web? ¿No es Internet un inmenso medio de información disponible públicamente para que cualquier persona la lea? Si navega por la Web desde casa a través de un proveedor de acceso, no hay duda de que ésta es la situación. Pero la cosa cambia ligeramente cuando dicha navegación se realiza en la oficina (buscando, por supuesto, información que sólo es interesante para nuestro trabajo). Muchas empresas tienen sus máquinas colocadas tras un firewall.

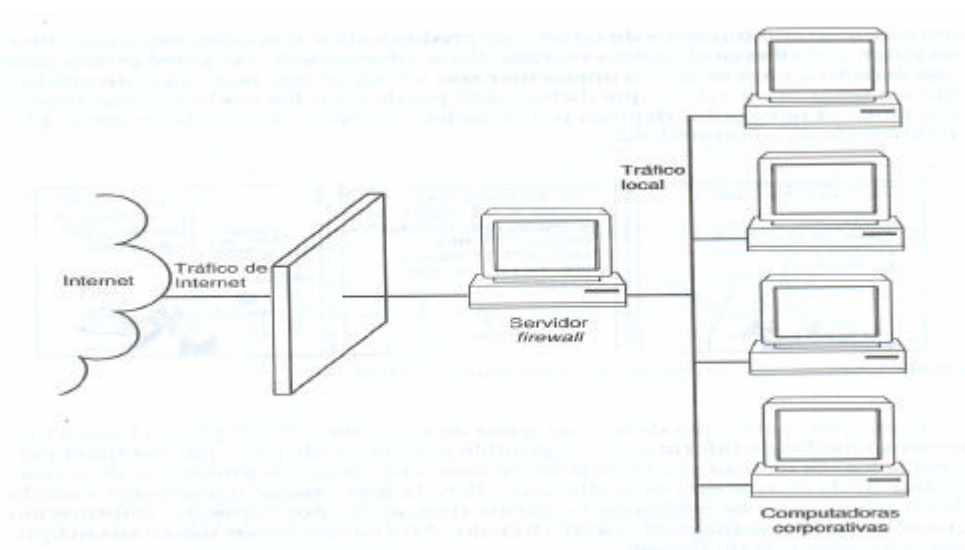
Un firewall es una computadora que filtra el tráfico que entra y sale de una red corporativa, denegando los intentos de acceso a aquellos servicios prohibidos. Por ejemplo, existen agujeros de seguridad conocidos en muchas implementaciones FTP. El firewall podría, simplemente, desactivar las peticiones FTP anónimas, o desviadas a un servidor FTP aislado. También podría denegar una petición de acceso al puerto del correo a todas las máquinas excepto al servidor de correo. Según la filosofía de seguridad que se tenga, el firewall (siguiente figura) puede también aplicar reglas de filtrado al tráfico entre la red corporativa e Internet (si le interesa este tema, consulte *Firewalls and Internet Security: Repelling the Wily Hacker*, de William R. Cheswick y Steven M. Bellovin, editorial Addison-Wesley).

Mediante un firewall, se puede hacer que una compañía utilice la Web para distribuir información interna interesante para los empleados pero que no debe estar accesible desde fuera de la misma. Para ello, basta con configurar un servidor web, difundir la dirección sólo entre los empleados y programar el firewall para que deniegue cualquier petición de acceso a dicho servidor desde el exterior.

Los empleados pueden ver entonces esa información interna con las mismas herramientas web que están acostumbrados a usar.

Si un empleado visita su página web, el applet, tras haber traspasado el firewall, se descarga en la computadora de dicho empleado y se ejecuta aquí. Si ese applet pudiera leer todas las páginas web como hace el navegador, tendría acceso a información corporativa. Después, no resultaría complicado establecer una conexión con el servidor del que proviene y enviar toda esta información privada. Esto, obviamente, es del todo inseguro. Ya que el navegador no tiene idea de qué páginas son públicas y cuáles confidenciales, corta por lo sano y desactiva el acceso a todas ellas.

Esto es una lástima (no es recomendable escribir un applet que vaya por la Web, recopile información, la procese y formatee y después se la presente al usuario). Por ejemplo, nuestro applet con el informe meteorológico no quiere escribir ningún dato de su servidor. Por tanto, ¿por qué no hacer que el navegador llegue a un trato con ese applet? Si éste promete no escribir nada, debería ser capaz de leer desde cualquier sitio. De esta forma, sería un mero recolector y procesador, mostrando un efímero resultado en la pantalla del usuario.



El problema es que el navegador no puede distinguir las peticiones de **lectura** de las de **escritura**. Cuando se solicita la apertura de un flujo en un URL, obviamente se trata de una petición de lectura. O puede que no. El URL podría ser así:

`http://www.rogue.com/cgi-bin/cracker.pl?Garys+password+is+Sicily`

Aquí, el culpable es el mecanismo del CGI, que está diseñado para tomar argumentos de forma arbitraria y procesados. El script que manipula la petición puede, y con frecuencia lo hace, almacenar los datos solicitados. Es sencillo ocultar información en el texto de una petición CGI (un flujo de datos que contenga información oculta suele conocerse como canal secreto en los círculos de la seguridad).

Por tanto, ¿debería desactivar el navegador todas las peticiones CGI y permitir sólo el acceso a páginas web planas? La solución no es tan sencilla. El navegador no tiene forma de saber que el servidor con el que se está conectando en el puerto es un servidor HTTP estándar. Podría ser una shell que almacenara todas las peticiones en un archivo y devolviera una página HTML que dijera: "Lo siento, la información solicitada no está disponible". Después, el applet podría transmitir la información con la pretensión de leer desde el URL:

`http://www.rogue.com/Garys/password/is/Sicily`

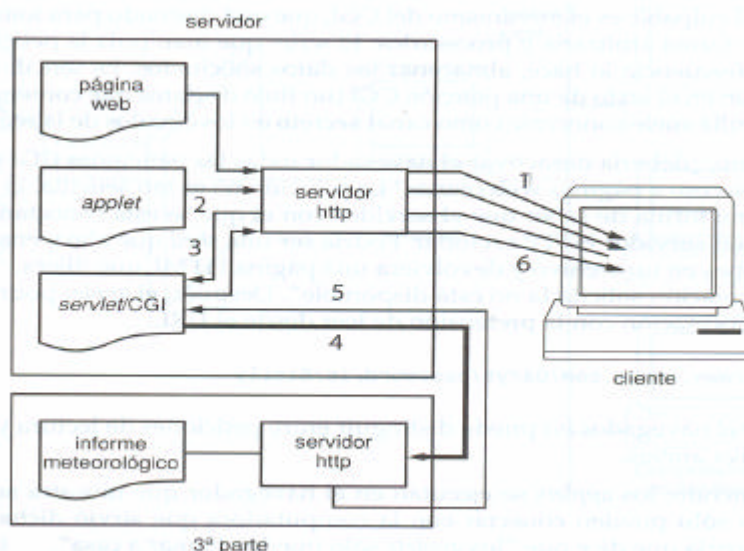
Ya que el navegador no puede distinguir entre peticiones de lectura y escritura, desactiva ambas.

Resumiendo: los applets se ejecutan en el navegador que muestra su página web, pero sólo pueden conectar con la computadora que sirvió dicha página. Éste es el significado de la regla que dice que "los applets sólo pueden llamar a casa".

Servidores proxy

Entonces, ¿cómo distribuir un applet que recolecte información para sus usuarios? Por suerte, existe una forma de proporcionar datos reales al applet: instalar un servidor proxy en su servidor HTML. Dicho proxy es un servicio que almacena peticiones desde la Web y las envía a cualquiera que las pida. Por ejemplo, suponga que su applet realiza la siguiente petición GET:

<http://www.yourserver.com/proxysvr?URL=http://iwin.nws.noaa.gov/iwin/CA/ourly.html>



al proxy que reside en el mismo servidor del código del applet. Dicho proxy recupera la página web correspondiente a la petición y la devuelve como resultado de la petición GET. Ver figura

Los servidores proxy para este propósito deben existir, en este caso será el ProxiSvr.java. Se implementa el proxy como un servlet, un programa que es inicializado por el motor servlet. Muchos servidores web pueden ejecutar servlets directamente, aunque otros deben ser actualizados para ello.

Los servlets no forman parte de la edición estándar de la plataforma Java, (Necesitamos de la J2EE, Enterprise Edition) y un tratamiento en profundidad de los mismos excede ampliamente este capítulo. (Mas detalles, consulte el libro Core Java Web Server, de Chris Taylor y Tim Kirmmet, editorial Prentice may) Sin embargo, nuestro ProxiSvr.java es muy sencillo. Va una breve explicación de su funcionamiento.

Cuando el servidor web recibe una petición GET procedente del servlet, llama al método doGet. El parámetro HttpServletRequest contiene los parámetros de la petición. El servlet utiliza el método getParameter para recuperar el valor del URL. En cualquier servlet, se debe enviar la respuesta en el PrintStream que el método getWriter de la clase HttpServletResponse devuelve. El servlet simplemente conecta con el recurso proporcionado por el parámetro URL, lee una línea de datos cada vez y la envía al flujo de respuesta. Para concluir, es necesario estudiar el método sendError para ver cómo devuelve información de errores.

En la siguiente sección le indicamos cómo configurar el servlet en el caso de que utilice el Apache Tomcat Servlet Container. Este software dispone de un pequeño servidor web que permite ejecutar localmente servlets. Es muy útil poder prescindir de tener que tener el servlet en otra máquina servidora, en la etapa de prueba y depuración. Bajo este punto de vista, Tomcat cumple las funciones del applet viewer, que nos evita salir del IDE para probar un applet.d

No es imprescindible utilizar servlets para implementar un servidor proxy. Se puede hacerlo en C o en Perl.

Vamos con el código

```
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;

/**
 * Un sencillo servlet implementando un servidor proxy.
 * El parámetro URL de la petición especifica el recurso a obtener.
 * El servlet otiene la información solicitada y la retorna.
 */
public class ProxiSvr extends HttpServlet{
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException{
        String query = null;

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        query = request.getParameter("URL");
        if (query == null){
```

```

        response.sendError(HttpServletResponse.SC_BAD_REQUEST,
            "Missing URL parameter");
        return;
    }

    try{
        query = URLDecoder.decode(query);
    }
    catch(Exception exception){
        response.sendError(HttpServletResponse.SC_BAD_REQUEST,
            "URL decode error " + exception);
        return;
    }

    try{
        URL url = new URL(query);
        BufferedReader in = new BufferedReader(new
            InputStreamReader(url.openStream()));

        String line;
        while ((line = in.readLine()) != null)
            out.println(line);
        out.flush();
    }
    catch(IOException exception){
        response.sendError(HttpServletResponse.SC_NOT_FOUND,
            "Exception: " + exception);
    }
}
}

```

Si se compara la codificación de esta Proxy en los lenguajes C y Perl, (El alumno puede acceder a los fuentes si baja el .zip citado en el índice de esta unidad), hay fundamento para recomendar firmemente el uso de servlets a la hora del procesamiento en el lado del servidor. La programación en red y la manipulación de cadenas en la plataforma Java es muchísimo mejor que la que pueda realizar C, y el código Java es mucho más "descifrable" y "manejable" que su equivalente en Perl.

Comprobación de WeatherApplet.java

En esta sección vamos a mostrarle cómo verificar el applet del informe meteorológico. Necesita un servidor web que pueda ejecutar servlets o scripts CGI.

Vamos a suministrarle instrucciones detalladas acerca del contenedor de servlets de Apache Tomcat. Si emplea otro servidor web, deberá hacer los ajustes pertinentes.

1. Descargue Tomcat desde <http://jakarta.apache.org/tomcat/index.html>.
2. Instale Tomcat. Para mayor sencillez, vamos a asumir que dicha instalación se hace en el directorio /tomcat. En caso de emplear otro, modifique las instrucciones en consonancia.

3. Compile el código del servlet:

```
javac -classpath/tomcat/lib/servlet.jar:. ProxySvr.java
```

4. Copie el archivo de clases ProxySvr.class al directorio /tomcat/webapps/examples/WEB-INF/classes.

5. Edite el archivo `/tomcat/webapps/examples/WEB-INF/web.xml` y añada las siguientes líneas:

```
<servlet>
    <servlet-name>
        proxySvr
    </servlet-name>
    <servlet-class>
        proxySvr
    </servlet-class>
</servlet>
```

y

```
<servlet-mapping>
    <servlet-name>
        proxySvr
    </servlet-name>
    <url-pattern>
        /proxySvr
    </url-pattern>
</servlet-mapping>
```

en los lugares adecuados.

6. Copie los archivos:

```
WeatherApplet.html
WeatherApplet*.class
```

en el directorio `/tomcat/webapps/examples`. Observe que hay varios archivos de clases internas que es necesario copiar.

7. Edite el archivo `WeatherApplet.html` y cambie la etiqueta `PARAM` por:

```
<PARAM NAME="queryBase" VALUE=
    "http://localhost:8080/examples/ProxySvr?URL=http://
    iwin.nws.noaa.gov/iwin/">
```

8. Inicie Tomcat.

9. Lance el applet ejecutando:

```
appletviewer http://localhost:8080/examples/weatherApplet.html
```

Alternativamente, puede usar el navegador para ver el URL. Para ello, es necesario que éste sea capaz de manejar applets de Java 2 (como ocurre con Netscape 6 u Opera). No se requiere un archivo de políticas de seguridad para ello.

En este punto suponemos que ya ha visto el applet en acción, vamos a ver por qué un servidor proxy resuelve el problema de seguridad que tiene. Cuando un usuario selecciona un estado y un tipo de informe meteorológico, el applet solicita la información al servidor proxy de su servidor local. El proxy va entonces al National Weather Service, recupera los datos y los devuelve al applet.

Esto da la sensación de crear más problemas de los que resuelve, pero evita el riesgo de violación de la seguridad. El applet sólo habla con el servidor proxy, y éste no puede atisbar en los documentos que pueden estar accesibles desde la máquina en la que está funcionando el applet.

Se puede usar esta técnica siempre que quiera desplegar un applet que recopile información de otros sitios web. En efecto, de esta forma puede recopilar lo que quiera del servidor y utilizar el applet para presentar los resultados.

Ahora, el problema de la seguridad está sobre su espalda. Al instalar un proxy en su servidor web, está permitiendo que cualquiera acceda a él y descargue cualquier archivo visible desde su servidor. Es entonces cuando debe configurar su red para que el servidor proxy no pueda tomar ninguno de sus archivos confidenciales. Alternativamente, puede implementar el proxy para que sólo recupere los URL que tengan una determinada forma. En nuestro caso, una restricción útil podría ser rechazar cualquier petición cuyo URL comience por `http://iwin.nws.noaa.gov`.

¿Tendría sentido disponer de un servidor que sólo registrara la información y la reflejara en el applet? En este caso, sí, pero en general, lo más lógico sería almacenar dicha información en una caché, lo que mejoraría el rendimiento a la hora de que se realizaran varias peticiones o para preprocesar esa información. Esto es lo que típicamente hacen los servidores Proxy, pero podemos ajustar sus "responsabilidades".

En este ejemplo usamos procesamiento en el servidor para evitar los problemas de seguridad. En otras muchas situaciones, es mejor procesar toda la información que sea posible en el servidor, ya que su software es más fácil de mantener y controlar. Los applets aún conservan su lugar (interactuar con el usuario y presentar los resultados). En una típica aplicación a tres niveles (o a n niveles), el applet podría estar emparejado con el servlet, o con cualquier otro mecanismo de programación en el lado del servidor que realice la "tarea dura".

A continuación, una colaboración de un ayudante de la Cátedra. Alguna parte del material está un poco repetida con lo expuesto hasta ahora, mientras que otra es nueva. Es siempre una visión adicional, encuentro útil incorporarla al apunte.

Programación con Sockets en Java - (Colaboración Salvador Celia)

Objetivos. En éste Capítulo vamos a aprender los fundamentos de la programación con "Sockets" en Java. Es decir, vamos a comprender cómo es posible utilizar las facilidades proporcionadas por los sockets de Java para desarrollar aplicaciones distribuidas, en las que uno o más ordenadores intercambian información a través de una red de comunicaciones. En particular, nos vamos a centrar en comunicaciones/servicios que utilizan el protocolo TCP en el nivel de transporte (El Protocolo de Control de Transmisión "TCP" en sus siglas en inglés, Transmission Control Protocol que fue creado entre los años 1973 - 1974 por [Vint Cerf](#) y [Robert Kahn](#) es uno de los protocolos fundamentales en Internet. Muchos programas dentro de una red de datos compuesta por ordenadores pueden usar TCP para crear *conexiones* entre ellos a través de las cuales enviarse datos. El protocolo garantiza que los datos serán entregados en su destino sin errores y en el mismo orden en que se transmitieron. También proporciona un mecanismo para distinguir distintas aplicaciones dentro de una misma máquina, a través del concepto de puerto. TCP soporta muchas de las aplicaciones más populares de Internet, incluidas [HTTP](#), [SMTP](#) y [SSH](#)). Pero TCP no es el único tipo que existe, vayamos primero a la introducción y luego veremos.

Introducción. Los sockets son puntos finales de enlaces de comunicaciones entre procesos. Los procesos los tratan como descriptores de ficheros, de forma que se pueden intercambiar datos con otros procesos transmitiendo y recibiendo a través de sockets.

El tipo de sockets describe la forma en la que se transfiere información a través de ese socket.

Sockets Stream (TCP, Transport Control Protocol)

Son un servicio orientado a conexión donde los datos se transfieren sin encuadrarlos en registros o bloques. Si se rompe la conexión entre los procesos, éstos serán informados.

El protocolo de comunicaciones con streams es un protocolo orientado a conexión, ya que para establecer una comunicación utilizando el protocolo TCP, hay que establecer en primer lugar una conexión entre un par de sockets. Mientras uno de los sockets atiende peticiones de conexión (servidor), el otro solicita una conexión (cliente). Una vez que los dos sockets estén conectados, se pueden utilizar para transmitir datos en ambas direcciones.

Sockets Datagrama (UDP, User Datagram Protocol)

Son un servicio de transporte sin conexión. Son más eficientes que TCP, pero no está garantizada la fiabilidad. Los datos se envían y reciben en paquetes, cuya entrega no está garantizada. Los paquetes pueden ser duplicados, perdidos o llegar en un orden diferente al que se envió.

El protocolo de comunicaciones con datagramas es un protocolo sin conexión, es decir, cada vez que se envíen datagramas es necesario enviar el descriptor del socket local y la dirección del socket que debe recibir el datagrama. Como se puede ver, hay que enviar datos adicionales cada vez que se realice una comunicación.

Sockets Raw

Son sockets que dan acceso directo a la capa de software de red subyacente o a protocolos de más bajo nivel. Se utilizan sobre todo para la depuración del código de los protocolos.

Diferencias entre Sockets Stream y Datagrama

Ahora se nos presenta un problema, ¿qué protocolo, o tipo de sockets, debemos usar - UDP o TCP? La decisión depende de la aplicación cliente/servidor que estemos escribiendo. Vamos a ver algunas diferencias entre los protocolos para ayudar en la decisión.

En UDP, cada vez que se envía un datagrama, hay que enviar también el descriptor del socket local y la dirección del socket que va a recibir el datagrama, luego éstos son más grandes que los TCP. Como el protocolo TCP está orientado a conexión, tenemos que establecer esta conexión entre los dos sockets antes de nada, lo que implica un cierto tiempo empleado en el establecimiento de la conexión, que no existe en UDP.

En UDP hay un límite de tamaño de los datagramas, establecido en 64 kilobytes, que se pueden enviar a una localización determinada, mientras que TCP no tiene límite; una vez que se ha establecido la conexión, el par de sockets funciona como los streams: todos los datos se leen inmediatamente, en el mismo orden en que se van recibiendo.

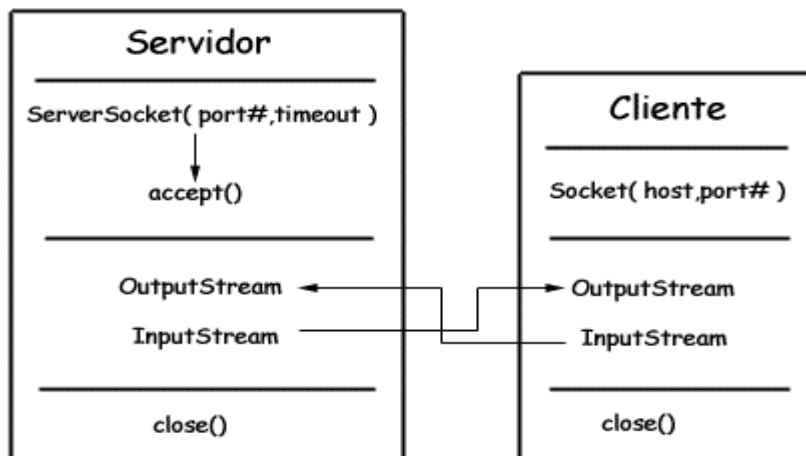
UDP es un protocolo desordenado, no garantiza que los datagramas que se hayan enviado sean recibidos en el mismo orden por el socket de recepción. Al contrario, TCP es un protocolo ordenado, garantiza que todos los paquetes que se envíen serán recibidos en el socket destino en el mismo orden en que se han enviado.

Los datagramas son bloques de información del tipo lanzar y olvidar. Para la mayoría de los programas que utilicen la red, el usar un flujo TCP en vez de un datagrama UDP es más sencillo y hay menos posibilidades de tener problemas. Sin embargo, cuando se requiere un rendimiento óptimo, y está justificado el tiempo adicional que supone realizar la verificación de los datos, los datagramas son un mecanismo realmente útil.

En resumen, TCP parece más indicado para la implementación de servicios de red como un control remoto (rlogin, telnet) y transmisión de ficheros (ftp); que necesitan transmitir datos de longitud indefinida. UDP es menos complejo y tiene una menor sobrecarga sobre la conexión; esto hace que sea el indicado en la implementación de aplicaciones cliente/servidor en sistemas distribuidos montados sobre redes de área local.

Modelo de Comunicación: Cliente/Servidor

En Java, crear una conexión socket TCP/IP se realiza directamente con el paquete `java.net`. A continuación mostramos un diagrama de lo que ocurre en el lado del cliente y del servidor:



El modelo de sockets más simple es:

- El servidor establece un puerto y espera durante un cierto tiempo (timeout segundos), a que el cliente establezca la conexión. Cuando el cliente solicite una conexión, el servidor abrirá la conexión socket con el método `accept()`.
- El cliente establece una conexión con la máquina host a través del puerto que se designe en `puerto#`
- El cliente y el servidor se comunican con manejadores `InputStream` y `OutputStream`

Para entender mejor esto, podemos decir que un socket es un "punto terminal" (extremo) de una comunicación entre dos procesos dentro de una red de ordenadores. Los sockets actúan como una "puerta" por la que los datos pueden entrar desde la red hacia el proceso y por la que pueden salir desde el proceso hacia la red. Comprender los detalles que como esta entrada/salida de datos tiene lugar en el marco del lenguaje Java es el objetivo de esta práctica, sin embargo, antes de poder estudiarlo es necesario que sepamos el concepto básico que está en el corazón del funcionamiento de los sockets: el modelo cliente/servidor.

En el modelo cliente/servidor, uno de los sockets actúa como extremo servidor. En ese caso, este socket se "liga" a un puerto y queda a la espera de recibir peticiones por parte de otro socket, que es el que actúa como cliente. Los detalles particulares de cómo se realizan estas operaciones los estudiaremos en breve. Por ahora, lo importante es observar que efectivamente existen dos "tipos" de sockets con finalidades y comportamientos diferentes. Por este motivo, estudiaremos cada uno de ellos por separado.

Pero antes de entrar de lleno a la programación haremos una síntesis de lo que tenemos que hacer para implementar el "Modelo Cliente/Servidor".

Apertura de Sockets

Si estamos programando un cliente, el socket se abre de la forma:

```
Socket miCliente;  
miCliente = new Socket( "maquina", numeroPuerto );
```

Donde maquina es el nombre de la máquina en donde estamos intentando abrir la conexión y numeroPuerto es el puerto (un número) del servidor que está corriendo sobre el cual nos queremos conectar. Cuando se selecciona un número de puerto, se debe tener en cuenta que los puertos en el rango 0-1023 están reservados para usuarios con muchos privilegios (superusuarios o root). Estos puertos son los que utilizan los servicios estándar del sistema como email, ftp o http. Para las aplicaciones que se desarrollen, asegurarse de seleccionar un puerto por encima del 1023.

En el ejemplo anterior no se usan excepciones; sin embargo, es una gran idea la captura de excepciones cuando se está trabajando con sockets. El mismo ejemplo quedaría como:

```
Socket miCliente;  
try {  
    miCliente = new Socket( "maquina", numeroPuerto );  
} catch( IOException e ) {System.out.println( e );}
```

Si estamos programando un servidor, la forma de apertura del socket es la que muestra el siguiente ejemplo:

```
Socket miServicio;  
try {  
    miServicio = new ServerSocket( numeroPuerto );  
} catch( IOException e ) {System.out.println( e );}
```

A la hora de la implementación de un servidor también necesitamos crear un objeto socket desde el ServerSocket para que esté atento a las conexiones que le puedan realizar clientes potenciales y poder aceptar esas conexiones:

```
Socket socketServicio = null;  
try {  
    socketServicio = miServicio.accept();  
} catch( IOException e ) {System.out.println( e );}
```

Siempre deberemos cerrar los canales de entrada y salida que se hayan abierto durante la ejecución de la aplicación. En la parte del cliente:

```
try {  
    salida.close();  
    entrada.close();  
    miCliente.close();  
} catch( IOException e ) {System.out.println( e );}
```

Y en la parte del servidor:

```
try {  
    salida.close();  
    entrada.close();  
    socketServicio.close();  
    miServicio.close();  
} catch( IOException e ) {System.out.println( e );}
```

Ahora si comencemos a programar**ServerSocket**

La clase *ServerSocket* del paquete *java.net* es la que implementa la funcionalidad de servidor dentro del lenguaje java. Tiene un conjunto numeroso de métodos cuya finalidad puede ser consultada en la documentación sobre la API estándar, para esta asignatura, lo esencial es que comprendamos los siguientes aspectos:

- `ServerSocket` tiene varios constructores, pero el que nosotros utilizaremos es el siguiente:
- `ServerSocket(int port) throws IOException`

Este constructor toma un número de puerto `port`, y "liga" el socket al citado puerto en todas las interfaces de red de las que disponga el host. Cuando especificamos `port = 0`, el `ServerSocket` se atará a cualquier puerto libre que exista en la máquina. Podemos recuperar el puerto concreto al que se ha atado utilizando el método `getLocalPort()`. Cuando hay problemas para crear el `ServerSocket` especificado, el constructor lanzará una `IOException`, que es una excepción `Checked`.

- Una vez que el `ServerSocket` está ligado y escuchando en el puerto especificado, hay que indicarle que se disponga a aceptar solicitudes de conexión procedentes de otros clientes. Esto se logra utilizando el método `accept()`, que bloquea el hilo de ejecución hasta que se reciba una solicitud de conexión.
- Cuando se recibe una solicitud de conexión por parte de un socket remoto, el método `accept()` se desbloquea, crea un nuevo objeto de la clase `Socket` y establece entre este y el socket remoto la conexión. De este modo, el `ServerSocket` queda disponible para volver a invocar `accept()`.
- Una vez que la conexión se ha establecido, es posible enviar/recibir información a/desde la misma utilizando los `Streams` de entrada/salida asociados al socket: `getInputStream()` y `getOutputStream()`.

Observemos el siguiente ejemplo utilizando la funcionalidad descrita para el `ServerSocket`.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class EjemploServerSocket {

    private static final int puerto = 2345;
    private static int numPeticiones = 0;

    public static void main(String[] args) {
        try{
            ServerSocket server = new ServerSocket(puerto);
            System.out.println("El ServerSocket se ha ligado con éxito
                               al puerto: " + puerto);

            while(true){
                Socket conn = server.accept();

                BufferedReader entrada = new BufferedReader(new
                    InputStreamReader(conn.getInputStream()));

                String line;
                while( (line = entrada.readLine()) != null){
                    System.out.println(line);
                    if(line.equals(""))
                        break;
                }
            }
        }
```



```
/*Aquí no solo podemos enviar un mensaje, sino para mayor utilidad
podríamos enviar una pagina html que fue solicitada por el Cliente*/
```

```
PrintWriter salida = new PrintWriter(conn.getOutputStream());
    salida.print("La Información fue recibida con éxito");
    salida.close();
    entrada.close();
}
} catch(IOException e) {e.printStackTrace();}
}
}
```

Prueba a compilar y ejecutar este ejemplo. Modifica el programa de modo que sea la propia clase `ServerSocket` la que elija un puerto libre donde atarse. Imprime dicho puerto por pantalla para poder conectarte al mismo con tu navegador.

Implementación del cliente

La clase `Socket`, al igual que la `ServerSocket`, forma parte del paquete `java.net`. Para ver los constructores y métodos de esta clase podemos consultar la documentación sobre la API estándar de Java. Para esta asignatura, los siguientes aspectos son los más relevantes:

- Para crear un `Socket` que actúe como cliente en una conexión TCP, lo más sencillo es utilizar el constructor siguiente:

```
Socket(String host, int port) throws UnknownHostException,
                                   IOException;
```

Al utilizar este constructor, especificamos el nombre del host servidor y el puerto en el que el servidor se encuentra escuchando. Si la llamada tiene éxito, el `Socket` creado estará conectado. Si la llamada no tiene éxito, se lanzará una `UnknownHostException`, si no se puede obtener una dirección IP válida para el nombre de host especificado, o una `IOException`, si la conexión no puede establecerse por algún otro motivo.

- Una vez que el `Socket` está conectado, se puede recuperar un `Stream` de entrada/salida invocando el método `getInputStream()/getOutputStream()`.
- Una vez que se han enviado/recibido los datos, la conexión se puede cerrar invocando el método `close()` de la clase `Socket`. Sin embargo, en la práctica, la conexión suele cerrarse invocando `close()` sobre los `Streams`.

Siguiendo estas indicaciones trata de compilar y ejecutar el ejemplo siguiente, que funciona junto con el servidor que hemos puesto a punto en el apartado anterior:

Fichero `EjemploSocket.java`:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

public class EjemploSocket{
    public static void main(String[] args) {
        if(args.length != 2){
            System.out.println("Especifique host, puerto servidor");
            System.exit(-1);
        }
        String hostServidor = args[0];
```

```

int puertoServidor = Integer.parseInt(args[1]);

try{
    Socket socket = new Socket(hostServidor, puertoServidor);
    System.out.println("Conexión OK a " + hostServidor + ":" +
                        +puertoServidor);
    PrintWriter salida = new PrintWriter(socket.getOutputStream()
                                          Stream() );
    salida.print("GET /index.html HTTP/1.0\r\n");
    salida.println("\r\n");
    salida.flush();

    BufferedReader entrada = new BufferedReader(new
        InputStreamReader(socket.getInputStream()));

    String line;
    while( (line = entrada.readLine()) != null){
        System.out.println(line);
    }
    entrada.close();
    salida.close();
} catch(IOException e){e.printStackTrace();}
}

```

Clases útiles

Vamos a exponer otras clases que resultan útiles cuando estamos desarrollando programas de comunicaciones, aparte de las que ya se han visto:

Socket

Es el objeto básico en toda comunicación a través de Internet, bajo el protocolo TCP. Esta clase proporciona métodos para la entrada/salida a través de streams que hacen la lectura y escritura a través de sockets muy sencilla.

ServerSocket

Es un objeto utilizado en las aplicaciones servidor para escuchar las peticiones que realicen los clientes conectados a ese servidor. Este objeto no realiza el servicio, sino que crea un objeto Socket en función del cliente para realizar toda la comunicación a través de él.

DatagramSocket

La clase de sockets datagrama puede ser utilizada para implementar datagramas no fiables (sockets UDP), no ordenados. Aunque la comunicación por estos sockets es muy rápida porque no hay que perder tiempo estableciendo la conexión entre cliente y servidor.

DatagramPacket

Clase que representa un paquete datagrama conteniendo información de paquete, longitud de paquete, direcciones Internet y números de puerto.

MulticastSocket

Clase utilizada para crear una versión multicast de las clase socket datagrama. Múltiples clientes/servidores pueden transmitir a un grupo multicast (un grupo de direcciones IP compartiendo el mismo número de puerto).

NetworkServer

Una clase creada para implementar métodos y variables utilizadas en la creación de un servidor TCP/IP.

NetworkClient

Una clase creada para implementar métodos y variables utilizadas en la creación de un cliente TCP/IP.

SocketImpl

Es un Interface que nos permite crearnos nuestro propio modelo de comunicación. Tendremos que implementar sus métodos cuando la usemos. Si vamos a desarrollar una aplicación con requerimientos especiales de comunicaciones, como pueden ser la implementación de un cortafuegos (TCP es un protocolo no seguro), o acceder a equipos especiales (como un lector de código de barras o un GPS diferencial), necesitaremos nuestra propia clase **Socket**.

Programación de servidores multithread

Un thread es un hilo de ejecución secuencial dentro de un programa. El concepto de thread en Java es similar al concepto de tarea (task) en Ada. Habitualmente, los servidores de red se programan con múltiples hilos de ejecución, esto permite, al mismo tiempo, que el desarrollador tenga una visión secuencial de su código y que las peticiones se traten con eficiencia.

En Java, crear un nuevo thread es relativamente sencillo. Existen dos mecanismos básicos para hacerlo:

- Crear una nueva clase que extienda la clase *Thread* del paquete *java.lang*. Esta clase debe redefinir el método *run()* que se ejecutará cuando se lance el nuevo thread.
- Crear una nueva clase que implemente el interfaz *Runnable* del paquete *java.lang*. Esta clase debe implementar el método *run()*, definido en la interfaz.
- Una vez que se tiene una instancia de la clase así definida, el nuevo thread (hilo de ejecución) se lanza invocando el método *start()* sobre el objeto, cuando su clase hereda directamente de *Thread*.
- En caso de que la clase que hayamos definido no herede de *Thread*, sino que implemente *Runnable*, para lanzar el nuevo hilo debemos crear un objeto de la clase *Thread* utilizando el constructor que toma un *Runnable* a la entrada. Una vez que tengamos esa instancia de *Thread*, ejecutamos *start()* sobre la misma.

Observa el siguiente ejemplo en el que definimos un servidor web multithread en el que cada petición es procesada por un nuevo hilo de ejecución.

```
Fichero ProcesadorHttp.java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
public class ProcesadorHttp extends Thread{
    private Socket conn;
    public ProcesadorHttp(Socket conn){
        this.conn = conn;
    }

    public void run(){
        try{
            BufferedReader entrada = new BufferedReader(new
                InputStreamReader(conn.getInputStream()));
            String line;
            while( (line = entrada.readLine()) != null){
```

```

        System.out.println(line);
        if(line.equals(""))
            break;
    }

    StringBuilder page = new StringBuilder("");
    page.append("<html>");
    page.append("<head>");
    page.append("<title>PAGINA DE PRUEBA</title>");
    page.append("</head>");
    page.append("<body>");
    page.append("<hr><p align=\"center\"> Hola, has solicitado
                la pagina " +
                ServidorMultithread.getNumPeticiones()+ "
                veces</p><hr>");

    ServidorMultithread.incrNumPeticiones();
    page.append("</body>");
    page.append("</html>");

    PrintWriter salida = new
        PrintWriter(conn.getOutputStream());
    salida.print("HTTP/1.0 200 OK\r\n");
    salida.print("Content-type: text/html\r\n");
    salida.print("Content-length: " + page.length() + "\r\n");
    salida.print("\r\n");
    salida.print(page);
    salida.close();
    entrada.close();
} catch (IOException e) {e.printStackTrace();}
}
}

```

Fichero ServidorMultithread.java

```

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
public class ServidorMultithread {
    private final static int puerto = 6543;
    private static int numPeticiones = 0;
    public static synchronized int getNumPeticiones(){
        return numPeticiones;
    }
    public static synchronized void incrNumPeticiones(){
        numPeticiones++;
    }
    public static void main(String[] args){
        try{
            ServerSocket serverSocket = new ServerSocket(puerto);
            System.out.println("El ServerSocket se ha atado con éxito
                                al puerto " + puerto);

            while(true){
                Socket conn = serverSocket.accept();
                ProcesadorHttp p = new ProcesadorHttp(conn);
                p.start();
            }
        } catch(IOException e){
            e.printStackTrace();
        }
    }
}

```

Compila este ejemplo y ejecútalo. Observa que funciona de manera apropiada lanzando un Thread por cada petición HTTP que se recibe en el servidor. Observa también un detalle adicional, los métodos `getNumPeticiones()` e `incrNumPeticiones()` están precedidos por la palabra clave *synchronized*. Esto es debido a que múltiples threads pueden estar accediendo de manera simultánea a la variable `numPeticiones`, unos escribiendo y otros leyendo. Si la modificación de esta variable no es atómica, este acceso concurrente puede producir escrituras incompletas y lecturas erróneas. La palabra *synchronized* indica que, para evitar este tipo de problemas, el acceso a estos métodos se encuentra sincronizado.

Envío de objetos serializados a través de un socket

En la Cátedra de "Algoritmos y Estructura de Datos" aprendimos que es posible enviar/recibir un objeto serializable a través de un `ObjectInputStream/ObjectOutputStream`. Esta funcionalidad es muy útil a hora de realizar aplicaciones distribuidas puesto que el desarrollador no tiene que preocuparse por ningún aspecto relativo al aplanamiento/desaplanamiento de datos ni a los formatos de representación. Para que se pueda observar todo el potencial de esta funcionalidad, te proponemos que realices el siguiente ejercicio:

- Escribe una aplicación cliente servidor de envío de mensajes
- El cliente es un programa que lee de la entrada estándar el asunto y el cuerpo del mensaje. Una vez que ha construido el mensaje, se lo envía al servidor
- El servidor es un programa que está a la espera de recibir conexiones de red. Sobre cada conexión establecida, el servidor recibe un mensaje e imprime por pantalla los contenidos del mismo. Tras eso contesta con un mensaje de respuesta que incluye un código de estado.
- Para realizar la aplicación, utiliza las siguientes clases:

Fichero `Peticion.java`

```
import java.io.Serializable;

public class Peticion implements Serializable {
    private String emisor, asunto, cuerpo;
    public Peticion(String emisor, String asunto, String cuerpo){
        this.emisor = emisor; this.asunto = asunto;
        this.cuerpo = cuerpo;
    }
    public String getEmisor(){return emisor;}
    public String getAsunto(){return asunto;}
    public String getCuerpo(){return cuerpo;}
}
```

Fichero `Respuesta.java`

```
import java.io.Serializable;

public class Respuesta implements Serializable {
    public static int OK = 0;
    public static int ERROR = -1;
    private int codigo;
    private String descripcion;
    public Respuesta(int codigo, String descripcion){
        this.codigo = codigo;
        this.descripcion = descripcion;
    }
    public int getCodigo(){return codigo;}
    public String getDescripcion(){return descripcion;}
}
```

