

## Chapter 17

### Network Programming

En este capítulo, discutiremos algunas características intrínsecas de J2SE, que proveen soporte a la programación en redes. Las redes de computadoras son tan comunes que UD. puede tener una en su propia casa y al menos con certeza en su propio lugar de trabajo. Entonces Internet, puede ser descrita como una red global de computadoras. Por esta razón, la programación en redes es fundamental y parte esencial de cualquier lenguaje de programación. Java es un lenguaje nuevo, que tiene muchas clases intrínsecas e interfaces para la programación de redes. Iniciaremos con un breve repaso de redes y de los dos protocolos más comunes: TCP y UDP. Luego, mostraremos como conectar dos computadoras usando sockets, los cuales permiten realizar comunicaciones TCP/IP. Luego discutiremos Java Secure Sockets Extensión (JSSE), lo cual permite comunicaciones seguras. Luego veremos como enviar y recibir paquetes de datagramas UDP, y como usar la clase URLConnection para comunicarse con un URL.

### An Overview of Network Programming

El término programación de redes se refiere a escribir programas que se ejecutan en múltiples dispositivos (computadoras), en los cuales estos dispositivos están todos conectados unos a otros usando una red. El paquete java.net del API de J2SE contiene una colección de clases e interfaces que proveen los detalles de comunicación a bajo nivel, permitiendo que se escriban programas que se enfocan en la solución de problemas **at hand**. El paquete java.net provee el soporte para dos protocolos de redes comunes.

**TCP.** TCP (Transmission Control Protocol), permite comunicaciones seguras entre dos aplicaciones. TCP es típicamente usado sobre el Protocolo de Internet el cual es conocido como TCP/IP.

**UDP.** UDP (User Datagram Protocol), es un protocolo no orientado a conexión que permite que los paquetes de datos sean transmitidos entre las aplicaciones.

En las siguientes secciones echaremos una mirada a la manera en que estos dos protocolos se comparan.

### Transmission Control Protocol

TCP frecuentemente es comparado con una llamada telefónica. Si UD. Desea llamar a alguien por teléfono, necesita un número telefónico y necesita esperar por una llamada entrante. Después de que la persona que UD. llama responde el teléfono, UD. tiene un flujo confiable de comunicación en dos sentidos, que permite a ambas personas hablar el uno al otro (siempre al mismo tiempo). Si una de las personas cuelga el teléfono la comunicación termina.

Con una comunicación de red TCP, la computadora cliente es similar a la persona que hace una llamada telefónica y la computadora servidor es similar a la persona que espera una llamada. Cuando un cliente intenta conectarse al servidor, el servidor necesita estar en ejecución, necesita esperar por una llamada entrante de conexión a un puerto. Cuando la conexión TCP se establece, el cliente y el servidor tienen una manera de comunicación de flujos en dos vías que permite la transmisión de los datos en ambas direcciones. Las dos computadoras pueden comunicarse hasta que la conexión se cierra o se pierde.

Las clases `java.net.ServerSocket` y `java.net.Socket` son las únicas dos clases que UD. probablemente siempre necesite para crear una conexión TCP/IP entre dos computadoras a menos que requiera una conexión segura en la cual UD. usaría las clases `SSLServerSocket` y `SSLSocket` del paquete `javax.net.ssl`. Discutiremos ambas de estas técnicas mas tarde en este capítulo.

## User Datagram Protocol

El UDP (User Datagram Protocol) provee un protocolo para enviar paquetes de datos llamados datagramas entre las aplicaciones. Si TCP es similar a hacer una llamada telefónica, UDP puede ser comparado con enviar una carta a alguien. El datagrama empacado es como una carta en donde el cliente envía un datagrama a un servidor sin estar actualmente conectado al servidor. Esto hace a UDP un protocolo de comunicación no confiable cuando se compara con TCP, en donde el cliente y el servidor están directamente conectados.

Si envío por correo dos cartas el mismo día estas pueden ser entregadas el mismo día a su destinatario; pero esto es difícilmente garantizado. De hecho, no hay garantía que las dos cartas lleguen el mismo día, o que una de ellas no llegue durante el transcurso de dos semanas. Lo mismo es cierto para los paquetes de datagramas. UDP no garantiza que el paquete va a ser recibido en el orden en que fue enviado o que de todas maneras siempre llegará.

Si este tipo de comunicación no confiable es inaceptable para los programas que desarrolla entonces UD. debe usar TCP. Sin embargo si UD. desarrolla una aplicación de red en la cual la confiabilidad de la comunicación no es esencial para la aplicación, UDP es probablemente una mejor opción por que esta no causa la preocupación de TCP.

Las clases `java.net.DatagramPacket` y `java.net.DatagramSocket` son usadas para enviar y recibir paquetes de datagramas; yo le mostraré cómo esto se hace en la siguiente sección *Overview of Datagram Packets*.

## Classroom Q & A

**Q:** ¿Cómo una computadora encuentra a otra en una red ?

**A:** Cualquier computadora en la red tiene un único valor el cual se conoce como dirección IP. Cada computadora probablemente tiene un nombre, esto hace fácil que otra

computadora pueda localizarla, especialmente si esta dirección IP cambia en la red pero su nombre no.

**Q:** ¿ Pueden dos computadoras tener múltiples conexiones TCP entre ellas ?

**A:** Ciertamente. De hecho, es frecuente el caso en que dos computadoras tienen múltiples aplicaciones corriendo y existen comunicaciones entre ellas, unas con otra. Los servidores que tienen aplicaciones como HTTP, FTP, todas corren al mismo tiempo.

**Q:** ¿Así, si dos computadoras tienen múltiples conexiones, como UD. distingue con cual aplicación UD. desea comunicarse ?

**A:** Bien, cuando los datos son enviados desde una aplicación hacia otra, los datos tienen dos valores asociados con esto: la dirección IP de la computadora y el número del puerto. La dirección IP denota a cual computadora los datos son enviados y el puerto denota con que aplicación se desea comunicar.

**Q:** ¿Ud. necesita asociar un número de puerto con todos los programas de red?

**A:** Si, Ud. encontrará que los números de puertos son usados a través del API de JAVA, especialmente en los constructores cuando las aplicaciones se inician.

**Q:** ¿Sólo debo usar un número de puerto ?

**A:** Seguro. Un número de puerto puede ser cualquier entero de 16 – bit. (entre 0 y 65535). Sin embargo UD. debe usar sólo números de puertos mayores de 1024 debido a que los puertos con números menores son reservados para los protocolos comunes, tales como HTTP, FTP, Telnet y otros. Frecuentemente un puerto necesita ser asignado por un administrador de red. Sin embargo, si sólo estoy probando las aplicaciones o escribiendo programas para redes pequeñas, yo sólo escojo números grandes. Los puertos mayores a 10,000 se ven seguros porque muchas aplicaciones comunes como las de los servidores Web usan los puertos hasta el rango 9000.

**Q:** ¿Qué sucede si escojo un número de puerto que ya ha sido usado?

**A:** Ud. obtendrá una Exception en su programa; en este caso, debe manejar la excepción y tratar con otro puerto hasta que ocurra satisfactoriamente. En muchos métodos que requieren un número de puerto UD. puede pasar 0 y dejar que la JVM encuentre un puerto por UD.

Yo ahora estoy listo para mostrarle cómo hacer alguna programación de redes, así iniciemos con la creación de una conexión TCP usando los sockets.

## Using Sockets

Si TCP es similar a realizar una llamada telefónica un socket es el teléfono. Los sockets proveen el mecanismo de comunicación entre dos computadoras usando TCP. El programa cliente crea un socket en su final de la comunicación e intenta conectar este socket a un servidor. Cuando la comunicación se hace, el servidor crea un objeto socket en su final de comunicación. El cliente y el servidor pueden ahora comunicarse escribiendo y leyendo desde el socket.

El paquete `java.net` contiene las clases que proveen todas las comunicaciones de bajo nivel. Por ejemplo, la clase `java.net.Socket` representa un socket y la clase `java.net.ServerSocket` provee un mecanismo para que el programa del servidor pueda escuchar a los clientes y establecer las conexiones con estos. Los siguientes pasos ocurren cuando establecemos una conexión TCP entre dos computadoras usando sockets:

1. El servidor instancia un objeto `ServerSocket` denotando con cual número de puerto debe hacerse la comunicación.
2. El servidor invoca al método `accept()` de la clase `ServerSocket`. Este método espera hasta que el cliente se conecte al servidor en el puerto dado.
3. Después que el servidor esta esperando, un cliente instancia un objeto `Socket`, especificando el nombre del servidor y el número del puerto al cual se va a conectar.
4. El constructor de la clase `Socket` intenta conectar el cliente al servidor especificado y el número de puerto. Si la comunicación se establece, el cliente tiene ahora un objeto `Socket` capaz de comunicarse con el servidor.
5. Del lado del servidor, el método `accept()` retorna una referencia a un nuevo socket en el servidor que es conectado a un socket cliente.

**Nota** Cuando un cliente establece una conexión socket hacia un servidor, el cliente necesita especificar un número de puerto. Este número de puerto denota el puerto que el cliente esta escuchando. Sin embargo, después de que el cliente y el servidor están conectados usando sockets, sus conexiones tienen lugar actualmente en puertos diferentes. Esto le permite al servidor que continúe escuchando en el puerto original para otros clientes en un hilo separado. Todo esto tiene lugar detrás de escenario y no afecta su código pero esta es una golosina importante que debemos entender.

Después de que las conexiones han sido establecidas, las comunicaciones pueden ocurrir usando flujos I/O. Cada socket tiene ambos `OutputStream` y un `InputStream`. El `OutputStream` del cliente se conecta al `InputStream` del servidor, y el `InputStream` cliente

es conectado al OutputStream del servidor. TCP es un protocolo de comunicación en ambos sentidos, por eso los datos pueden ser enviados al mismo tiempo por ambos sentidos del flujo.

Nota:

Los flujos socket son flujos de I/O de bajo nivel: InputStream y OutputStream. Por eso estos pueden ser encadenados juntos con los filtros buffer, y con los otros flujos de alto nivel para permitir cualquier tipo avanzado de I/O que UD. necesite realizar. Esta es la razón por la que vimos el paquete java.io antes de la programación de redes. UD. tiene que encontrar que la creación de la conexión es la parte fácil y mucho de su trabajo en la programación de redes involucra la transmisión actual de los datos hacia delante y hacia atrás. Por su puesto así es como la programación de redes debe ser, permitiéndole a UD. enfocarse en el problema que será resuelto y no se preocupe acerca del bajo nivel y los detalles del protocolo. Esto es tan popular para un gran número de programadores de redes.

Veamos un ejemplo usando los sockets. Yo iniciaré con un programa que corre en el servidor, escuchando en un puerto por una petición del cliente. Entonces le mostraré como escribir el código cliente que se conecta a una aplicación del cliente.

### The ServerSocket Class

La clase java.net.ServerSocket es usada para las aplicaciones de servidor para obtener el puerto y escuchar las peticiones del cliente. La clase ServerSocket tiene cuatro constructores:

**public ServerSocket( int port ) throws IOException.** Intenta crear un socket servidor ceñido a un puerto especificado. Una excepción ocurre si el puerto esta ya usado por otra aplicación. El parámetro del puerto puede ser 0, el cual crea un socket en cualquier puerto libre.

**public ServerSocket( int port, int backlog ) throws IOException.** Similar al constructor previo, el parámetro backlog especifica cuantos clientes entrantes son almacenados en una cola para que puedan esperar. Si la cola esta llena los clientes que intentan conectarse a este puerto recibirán una excepción. Si el valor es 0, el tamaño por defecto de la cola será el tamaño nativo de la plataforma que se usa.

**public ServerSocket( int port, int backlog, InetAddress address ) throws IOException.** Similar al constructor previo, el parámetro InetAddress especifica la dirección IP local a la cual se debe ceñir. La InetAddress es usada por los servidores que pueden tener múltiples direcciones IP, permitiéndole al servidor especificar cual de estas direcciones IP será aceptada por las peticiones de los clientes.

**public ServerSocket( ) throws IOException.** Crea un socket de servidor independiente. Cuando usamos este constructor, use el método bind() cuando esta listo para ceñir el socket del servidor.

Note que cada uno de estos constructores lanza una `IOException` cuando algo esta mal. Sin embargo, si el constructor `ServerSocket` no lanza una excepción, esto significa que su aplicación se ha ceñido al puerto satisfactoriamente y esta listo para recibir las peticiones del cliente. Veamos algunos métodos de la clase `ServerSocket`:

**`public int getLocalPort()`.** Retorna el puerto por el que esta escuchando el servidor del socket. Este método es útil si UD. pasa un cero como número del puerto en un constructor y permite que el servidor encuentre el puerto por UD.

**`public Socket accept() throws IOException`.** Espera por los clientes entrantes. Este método bloquea hasta que un cliente se conecta al servidor en un puerto específico o el tiempo de caducidad del socket termina, asumiendo que el valor del tiempo de caducidad (time-out) ha sido asignado usando el método `setTimeout()`. De otra forma este método bloquea indefinidamente.

**`public void setSoTimeout( int timeout )`.** Asigna el valor del tiempo (time-out) por cuánto tiempo el servidor de sockets espera por un cliente durante el `accept()`.

**`public void bind( SocketAddress host, int backlog )`.** Ciñe el socket a un servidor y puerto especificado en el objeto `SocketAddress`. Use este método si UD. instancia el `ServerSocket` usando un constructor sin argumento.

El método `accept()` es uno de los que deseo que UD. se enfoque por que esta es la forma como el servidor escucha las peticiones entrantes. Cuando el `ServerSocket` invoca `accept()`, el método no retorna hasta que un cliente se conecte ( Asumiendo que ningún valor del tiempo de caducidad ha sido asignado). Después de que el cliente se conecta, el `ServerSocket` crea un nuevo `Socket` en un puerto no especificado (diferente del puerto que ha estado escuchando) y retorna una referencia hacia este nuevo `Socket`. Ahora existe una conexión TCP entre el cliente y el servidor y la comunicación puede iniciar.

Nota:

Si UD. esta escribiendo una aplicación de servidor que permite múltiples clientes, UD. desea que su servidor de socket siempre este invocando `accept()`, esperando por clientes. Cuando un cliente se conecta, un truco estándar consiste en iniciar un nuevo hilo para la comunicación con un nuevo cliente, permitiéndole al thread actual inmediatamente invocar nuevamente a `accept()`. Por ejemplo, si UD tiene 50 clientes conectados a un servidor el programa servidor tendría 51 hilos: 50 hilos para la comunicación con los 50 clientes y un hilos adicional esperando por un nuevo cliente por la vía del método `accept()`.

El siguiente programa `SimpleServer` es un ejemplo de una aplicación que usa la clase `ServerSocket` para escuchar los clientes en un número de puerto especificado en la línea de comando. Note que el servidor no puede hacer mucho con el cliente, pero el programa demuestra como se hace la conexión con el cliente. El valor retornado por `accept()` es un `Socket` así necesito discutir la clase `Socket` antes de que podamos, hacer algo excitante

con la conexión. Estudie el programa SimpleServer y trate de determinar cual será su salida:

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

public class SimpleServer extends Thread
{
    private ServerSocket serverSocket;

    public SimpleServer(int port) throws IOException
    {
        serverSocket = new ServerSocket(port);
        serverSocket.setSoTimeout(10000);
    }

    public void run()
    {
        while(true)
        {
            try
            {
                System.out.println("Waiting for client on port " +
                                   serverSocket.getLocalPort() + "...");
                Socket client = serverSocket.accept();

                System.out.println("Just connected to " +
                                   client.getRemoteSocketAddress());
                client.close();
            } catch(SocketTimeoutException s)
            {
                System.out.println("Socket timed out!");
                break;
            } catch(IOException e)
            {
                e.printStackTrace();
                break;
            }
        }
    }
}
```

```

public static void main(String [] args)
{
    int port = Integer.parseInt(args[0]);

    try
    {
        Thread t = new SimpleServer(port);
        t.start();
    } catch (IOException e)
    {
        e.printStackTrace();
    }
}
}

```

La Figura 17.1 muestra la salida del programa SimpleServer cuando en la línea de comando se ingresa 5001. El programa del servidor esta esperando por un cliente, el cual deseo mostrarle a UD. como crearlo próximamente. Debido a que ningún cliente viene solo y se conecta, el método accept() bloquea por 10 segundos ( ¿por qué? ) y una excepción de tipo SocketTimeoutException ocurre, causando al hilo que corra completamente y el programa termine.

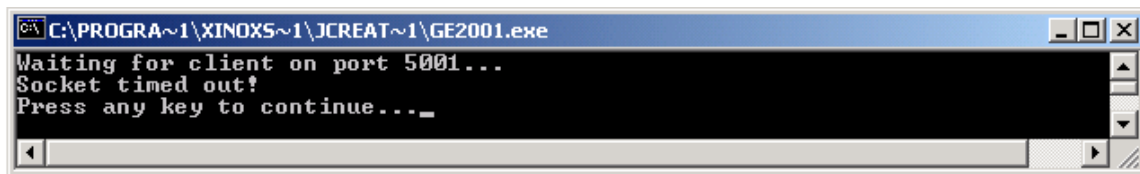


Figura 17.1 Salida del programa: java SimpleServer 5001

## Socket Class

La clase java.net.Socket representa al cliente y al servidor para comunicarse uno con el otro. El cliente obtiene un objeto Socket instanciando uno, luego el Server obtiene un objeto Socket del valor retornado por el método accept(). La clase Socket tiene cinco constructores para conectarse al servidor:

**public Socket(String host, int port ) throws UnknowHostException, IOException.** Intenta conectarse a un servidor en un puerto especificado. Si este constructor no lanza la excepción, la conexión ha ocurrido satisfactoriamente y el cliente esta conectado al servidor. Este es un constructor simplista para usarlo cuando nos conectamos al servidor.



**public Socket(InetAddress host, int port ) throws IOException.** Idéntico al constructor previo, excepto que tiene un servidor (host) denotado por el objeto InetAddress.

**public Socket(String host, int port, InetAddress localAddress, int localPort ) throws IOException.** Se conecta al servidor y puerto especificado, creando un Socket en el servidor local en la dirección y puerto especificado. Esto es útil para los clientes los cuales tienen múltiples dirección IP o desean conectar el socket a un puerto especificado.

**public Socket(InetAddress host, int port, InetAddress localAddress, int localPort ) throws IOException.** Idéntico al constructor previo, excepto que el servidor es denotado por un objeto InetAddress en lugar de una cadena.

**public Socket( ).** Crea un socket sin conexión. Usa el método connect() para conectar este socket al servidor.

Cuando el constructor del Socket retorna; este no simplemente instancia un objeto Socket. Dentro del constructor, este actualmente intenta conectarse al servidor y al puerto especificado. Si el constructor retorna satisfactoriamente, el cliente tiene una conexión TCP al servidor!.

Algunos métodos de interés en la clase Socket son listados aquí. Note que ambos, el cliente y el servidor tienen un objeto Socket, así estos métodos pueden ser invocados por ambos, el cliente y el servidor.

**public void connect(SocketAddress host, int timeout ) throws IOException.** Conecta el Socket al servidor especificado. Este método es necesario sólo cuando UD. instancia un Socket usando el constructor sin argumento.

**public InetAddress getInetAddress().** Retorna la dirección de la otra computadora al cual este socket esta conectado.

**public int getPort().** Retorna el puerto al cual esta ceñido el socket en la computadora remota.

**public int getLocalPort().** Retorna el puerto al cual esta ceñido el socket en la computadora local.

**public SocketAddress getRemoteSocketAddress().** Retorna la dirección del socket remoto.

**public InputStream getInputStream() throws IOException.** Retorna el flujo de entrada de un socket. El flujo de entrada esta conectado al flujo de salida del socket remoto.

**public OutputStream getOutputStream() throws IOException.** Retorna el flujo de salida de un socket. El flujo de salida esta conectado al flujo de entrada del socket remoto.

**public void close() throws IOException.** Cierra el socket, el cual hace que este objeto Socket no sea capaz de conectarse nuevamente a ningún servidor.

La clase Socket contienen muchos más métodos, así verifique la documentación para ver la lista completa. UD. notará que muchos métodos en la clase Socket involucran el acceso y el cambio de varias propiedades TCP de una conexión, tales como la configuración del valor del tiempo de caducidad o el mensaje para mantener la conexión activa (keep-alive). De todos los métodos en la clase Socket probablemente los dos más importantes son `getInputStream()` y `getOutputStream()`, los cuales discutiremos ahora en detalle.

### Communicating between Sockets.

Los atributos `InputStream` y `OutputStream` de un `Socket` es la forma en que dos computadoras se comunican una con otra. Por ejemplo, si el servidor desea enviar datos hacia el cliente, el servidor necesita escribir al `OutputStream` de este socket, el cual luego es leído por el `InputStream` del socket cliente. Similarmente, los datos pueden ser enviados desde el cliente al servidor usando el `OutputStream` del cliente y el `InputStream` del servidor.

El siguiente programa `GreetingClient.java` es un programa cliente que se conecta al servidor usando un socket y envía un saludo y luego espera por una respuesta.

Estudie el programa y trate de determinar exactamente como el cliente acopla esto.

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

public class GreetingClient
{
    public static void main(String [] args)
    {
        String serverName = args[0];
        int port = Integer.parseInt(args[1]);

        try
        {
            System.out.println("Connecting to " + serverName + " on port " + port);
            Socket client = new Socket(serverName, port);

            System.out.println("Just connected to " + client.getRemoteSocketAddress());
```

```

        OutputStream outToServer = client.getOutputStream();
        DataOutputStream out = new DataOutputStream(outToServer);
        out.writeUTF("Hello from " + client.getLocalSocketAddress());

        InputStream inFromServer = client.getInputStream();
        DataInputStream in = new DataInputStream(inFromServer);
        System.out.println("Server says " + in.readUTF());

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

```

El siguiente código de la clase GreetingServer es idéntico a la clase SimpleServer discutida antes, excepto que esta lee una cadena desde el cliente y envía un mensaje hacia atrás al cliente. El servidor entonces cierra el socket e invoca accept() nuevamente para el próximo cliente que venga. Estudie el programa cuidadosamente y vea como estos son acoplados:

```

System.out.println("Waiting for client on port " + serverSocket.getLocalPort() + "...");
Socket server = serverSocket.accept();

System.out.println("Just connected to " + server.getRemoteSocketAddress());
DataInputStream in = new DataInputStream(server.getInputStream());
System.out.println(in.readUTF());

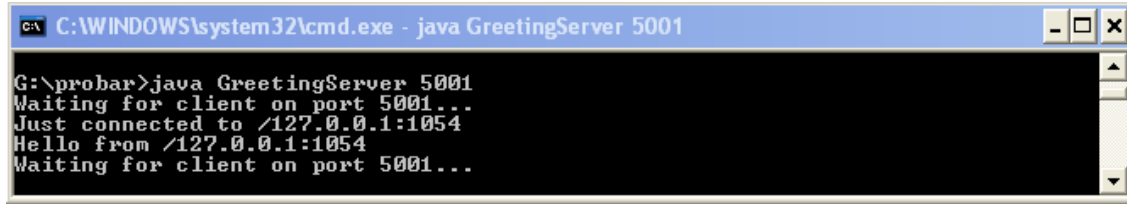
DataOutputStream out = new DataOutputStream(server.getOutputStream());
out.writeUTF("Thank you for connecting to " + server.getLocalSocketAddress() +
            "\nGoodbye!");

server.close();

```

Nota Los programas GreetingServer y GreetingClient pueden ejecutarse en dos computadoras diferentes que estan en la misma red, o UD. puede ejecutarlos ambos en la misma computadora. Cuando el cliente y el servidor están en la misma computadora el cliente puede usar localhost como el nombre del servidor para conectarse. Note que al correr los dos programas en la misma computadora se requiere que tengamos abierta dos terminales con el prompt de comandos.

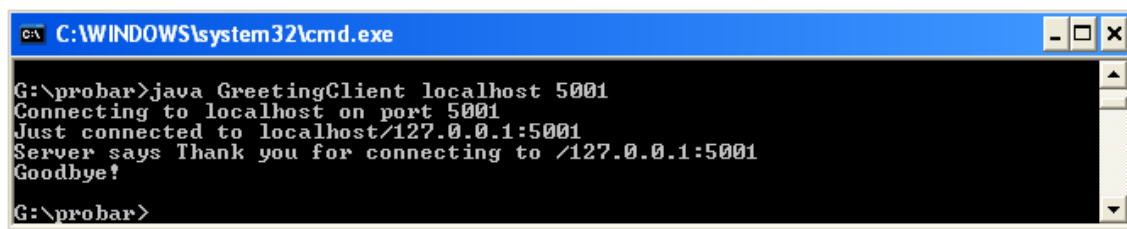
La Figura 17.2 muestra la salida del programa GreetingServer, este necesita ser ejecutado primero.

A screenshot of a Windows command prompt window. The title bar reads "C:\WINDOWS\system32\cmd.exe - java GreetingServer 5001". The command prompt shows the following output:

```
G:\probar>java GreetingServer 5001
Waiting for client on port 5001...
Just connected to /127.0.0.1:1054
Hello from /127.0.0.1:1054
Waiting for client on port 5001...
```

Figura 17.2 Output of the GreetingServer program.

La Figura 17.3 muestra la salida del programa GreetingClient. Note que el programa GreetingServer no termina debido a que invoca accept() en un bucle infinito. UD. puede ejecutar el cliente nuevamente y nuevamente sin tener que reiniciar el programa servidor.

A screenshot of a Windows command prompt window. The title bar reads "C:\WINDOWS\system32\cmd.exe". The command prompt shows the following output:

```
G:\probar>java GreetingClient localhost 5001
Connecting to localhost on port 5001
Just connected to localhost/127.0.0.1:5001
Server says Thank you for connecting to /127.0.0.1:5001
Goodbye!
G:\probar>
```

Figura 17.3 Output of the Greeting program.

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

public class GreetingServer extends Thread
{
    private ServerSocket serverSocket;

    public GreetingServer(int port) throws IOException
    {
        serverSocket = new ServerSocket(port);
    }

    public void run()
    {
        while(true)
        {
            try
            {
                System.out.println("Waiting for client on port " + serverSocket.getLocalPort() + "...");
                Socket server = serverSocket.accept();

                System.out.println("Just connected to " + server.getRemoteSocketAddress());
```

```

DataInputStream in = new DataInputStream(server.getInputStream());
System.out.println(in.readUTF());

DataOutputStream out = new DataOutputStream(server.getOutputStream());
out.writeUTF("Thank you for connecting to " + server.getLocalSocketAddress() +
"\nGoodbye!");

server.close();
    }catch(SocketTimeoutException s)
    {
        System.out.println("Socket timed out!");
        break;
    }catch(IOException e)
    {
        e.printStackTrace();
        break;
    }
}
}

public static void main(String [] args)
{
    int port = Integer.parseInt(args[0]);

    try
    {
        Thread t = new GreetingServer(port);
        t.start();
    }catch(IOException e)
    {
        e.printStackTrace();
    }
}
}

```

## Java Secure Socket Extension (JSSE)

Una conexión socket puede hacerse usando el protocolo Secure Sockets Layers (SSL), el cual provee una conexión segura entre el cliente y el servidor. Usando SSL se asegura un nivel alto de seguridad en lo que respecta a los datos que se envían entre dos computadoras. Cuando UD. compra un artículo en línea y envía el número de su tarjeta de crédito en Internet, SSL es un método seguro el cual es usado para proteger su número de tarjeta de crédito de que sea visto y usado maliciosamente.

La versión 1.4 de J2SE ha introducido un nuevo soporte para usar SSL y los sockets con Java Secure Socket Extensión (JSSE). Las clases involucradas con la creación de una conexión segura con JSSE, se encuentran en los paquetes `javax.net` y `javax.net.ssl` e incluyen las siguientes:

**`javax.net.ssl.SSLServerSocket`.** Es usada por los servidores de aplicaciones para aceptar conexiones de clientes y crea un `SSLSocket`. Compare esta clase con la clase `java.net.ServerSocket`.

**`javax.net.ssl.SSLSocket`.** Representa un socket seguro en el cliente y en el servidor. Compare esta clase con la clase `java.net.Socket`.

**`javax.net.ssl.SSLServerSocketFactory`.** El programa servidor usa esta clase para obtener un objeto `SSLServerSocket`.

**`javax.net.ssl.SSLSocketFactory`.** El programa cliente usa esta clase para obtener un objeto `SSLSocket`.

Los pasos para la creación de una conexión de socket segura son ligeramente diferentes de aquellos en los que la conexión no es segura (usando las clases `Socket` y `SocketServer` del paquete `java.net.package`), así que vamos a ir a través de los detalles.

## Secure Server Socket

Las aplicaciones de servidores realizan los siguientes pasos para crear un secure server socket:

1. El programa servidor inicia con un objeto `SSLServerSocketFactory`, el cual no se instancia usando la palabra clave `new`, en su lugar se obtiene el objeto usando el siguiente método static de la clase `SSLServerSocketFactory`:

```
public static ServerSocketFactory getDeFault()
```

Nota: El método `getDeFault()` retorna el servidor socket factory por defecto. Si UD. tiene que escribir su propio factory, puede sobrescribir el factory por defecto denotando el nombre de la clase usando la propiedad `ssl.ServerSocketFactory.provider`.

2. Use el server socket factory para crear un objeto `SSLServerSocket`, el cual escucha en el puerto especificado por peticiones del cliente.
3. Inicialice el socket servidor con cualquier configuración de seguridad necesaria.
4. El `SSLServerSocket` invoca el método `accept()` para bloquear y esperar por las conexiones del cliente.

El primer paso es localizar un servidor socket factory usando el método `getDeFault()`. Observe que el tipo de retorno del método `getDeFault()` es `javax.net.ServerSocketFactory`, la cual es la clase padre de `SSLServerSocketFactory`. La clase `ServerSocketFactory` tiene cuatro métodos para crear un `ServerSocket`:

**`public ServerSocket createServerSocket( int port ) throws IOException.`** Crea un `ServerSocket` en el puerto dado.

**`public ServerSocket createServerSocket( int port , int backlog) throws IOException.`** Crea un `ServerSocket` en el puerto dado con el backlog especificado, el cual determina la dimensión de la cola que contiene a los clientes que esperan por una conexión.

**`public ServerSocket createServerSocket( int port , int backlog, InetAddress address ) throws IOException.`** Crea un `ServerSocket` en el puerto usando la dirección local especificada, la cual es útil cuando el servidor tiene múltiples direcciones IP.

**`public ServerSocket createServerSocket( ) throws IOException.`** Crea un `ServerSocket` independiente al cual se le puede hacer dependiente utilizando el método `bind()` de la clase `ServerSocket`.

Nota: Los métodos `createServerSocket()` de la clase `ServerSocketFactory` tienen parámetros idénticos en los constructores en la clase `java.net.ServerSocket` discutidos antes en este capítulo. Cuando UD. no esta usando SSL, UD. también puede crear un objeto `ServerSocket` usando uno de estos constructores ( la técnica que se le ha mostrado en el programa `SimpleServer.java` o UD. puede usar la clase `ServerSocketFactory` y uno de los métodos `createServerSocket()`. Cuando use SSL, UD. debe usar un socket factory para obtener un socket servidor.

Cual técnica UD. usa para los sockets no seguros es una cuestión de elección, pero esto es evidente desde mi experiencia con otras APIs de Java que el uso la clase factory tiene sus ventajas, especialmente en hacer su código fácil de implementar en diferentes plataformas y en la configuración de varias propiedades de la factory. Por ejemplo, usando un `SocketFactory` le permite a UD. determinar las propiedades de un `ServerSocket`, tal como su valor de tiempo de caducidad (time-out), sin tener código difícil en su programa.

Después de que UD. ha obtenido un objeto `SSLServerSocket` usando a factory, UD. ahora esta listo para inicializar un socket servidor con cualquier configuración de seguridad especificada. La clase `SSLServerSocket` extiende a la clase `java.net.ServerSocket`, así que UD. puede invocar estos métodos discutidos antes, tales como `accept()` y `close()`, además de los métodos de `SSLServerSocket`, los cuales incluyen los siguientes:

**`public void setEnabledCipherSuites(String[] suites).`** Configura el cipher disponible para esta conexión. Un cipher suites define los algoritmos de seguridad para la autenticación, encriptación y las llaves de acuerdos, un proceso conocido como handshaking. El programa entrante `SSLServerDemo` muestra los valores del cipher suites comunes.

**`public String [] getSupportedCipherSuites().`** Retorna un arreglo que contiene el cipher suites que puede estar permitido para esta conexión particular de socket.

**`public String [] getSupportedProtocols().`** Retorna un arreglo que contiene los protocolos de seguridad que pueden estar permitidos para esta conexión particular de socket. UD. puede esperar que sea soportada tanto SSL como TLS, la Capa de Seguridad de Transporte (Transport Layer Security).

**`public void setNeedClientAuth(boolean flag).`** Denota que los clientes que se conecten a este socket servidor deben ser autenticados. Por defecto, las conexiones de los clientes no requieren autenticación. Si se tornan “on”, los clientes deben autenticarse o la conexión se cerrará inmediatamente después de que el servidor acepta la conexión del cliente.

**`public void setWantClientAuth(boolean flag).`** Denota que los clientes que se conectan con este socket servidor deben ser autenticados. Los clientes deben proveer la información de autenticación, pero la conexión aún se mantiene si este no se autentica.



Un programa para los socket SSL usa un factory para crear una toma (socket) de servidor segura. El siguiente programa SSLServerDemo.java demuestra los pasos que el servidor hace para aceptar una conexión segura del cliente. Estudie el programa y su salida cuidadosamente. Figura 17.A

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

public class SSLServerDemo
{
    public static void main(String [] args)
    {
        int port = Integer.parseInt(args[0]);

        try
        {
            System.out.println("Locating server socket factory for SSL...");

            SSLServerSocketFactory factory = (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();

            System.out.println("Creating a server socket on port " + port);

            SSLServerSocket serverSocket = (SSLServerSocket) factory.createServerSocket(port);

            String [] suites = serverSocket.getSupportedCipherSuites();

            System.out.println("Support cipher suites are:");

            for(int i = 0; i < suites.length; i++)
            {
                System.out.println(suites[i]);
            }

            serverSocket.setEnabledCipherSuites(suites);

            System.out.println("Support protocols are:");

            String [] protocols = serverSocket.getSupportedProtocols();

            for(int i = 0; i < protocols.length; i++)
            {
                System.out.println(protocols[i]);
            }

            System.out.println("Waiting for client...");
```

```

        SSLSocket socket = (SSLSocket) serverSocket.accept();

        System.out.println("Starting handshake...");

        socket.startHandshake();

        System.out.println("Just connected to " + socket.getRemoteSocketAddress());

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

```

```

C:\PROGRA~1\XINOX5~1\JCREAT~1\GE2001.exe
Locating server socket factory for SSL...
Creating a server socket on port 5002
Support cipher suites are:
SSL_RSA_WITH_RC4_128_MD5
SSL_RSA_WITH_RC4_128_SHA
TLS_RSA_WITH_AES_128_CBC_SHA
TLS_DHE_RSA_WITH_AES_128_CBC_SHA
TLS_DHE_DSS_WITH_AES_128_CBC_SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
SSL_RSA_WITH_DES_CBC_SHA
SSL_DHE_RSA_WITH_DES_CBC_SHA
SSL_DHE_DSS_WITH_DES_CBC_SHA
SSL_RSA_EXPORT_WITH_RC4_40_MD5
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
SSL_RSA_WITH_NULL_MD5
SSL_RSA_WITH_NULL_SHA
SSL_DH_anon_WITH_RC4_128_MD5
TLS_DH_anon_WITH_AES_128_CBC_SHA
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
SSL_DH_anon_WITH_DES_CBC_SHA
SSL_DH_anon_EXPORT_WITH_RC4_40_MD5
SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA
TLS_KRB5_WITH_RC4_128_SHA
TLS_KRB5_WITH_RC4_128_MD5
TLS_KRB5_WITH_3DES_EDE_CBC_SHA
TLS_KRB5_WITH_3DES_EDE_CBC_MD5
TLS_KRB5_WITH_DES_CBC_SHA
TLS_KRB5_WITH_DES_CBC_MD5
TLS_KRB5_EXPORT_WITH_RC4_40_SHA
TLS_KRB5_EXPORT_WITH_RC4_40_MD5
TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA
TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5
Support protocols are:
SSLv2Hello
SSLv3
TLSv1
Waiting for client...

```

Figura 17.4 Salida del Programa SSLServerDemo.java

Deseo hacer algunos comentarios acerca del programa SSLServerDemo:

- El método getDefault() declara que este retorna un ServerSocketFactory, pero el tipo de dato actual es SSLServerSocketFactory, así hemos moldeado a este tipo.
- Similarmente, el método createServerSocket() declara que este retorna una referencia. Sin embargo, debido a que usamos a SSLServerSocketFactory, el retorno actual es de tipo SSLServerSocket, así que se tiene que moldear el valor de este de todas maneras.
- Este programa despliega el cipher soportado y los protocolos para el entorno en el cual es ejecutado. La salida en la Figura 17.4 fue generada usando la implementación J2SE 1.4.
- He dispuesto todos los cipher suites que son soportados en esta plataforma con la siguiente sentencia:

```
serverSocket.setEnabledCipherSuites(suites);
```

- Cuando no he permitido el cipher suites soportado, obtengo una excepción que se ve así:

```
Javax.net.ssl.SSLException: No available certificate corresponds to the SSL  
cipher suites which are enabled at  
com.sun.net.ssl.internal.ssl.SSLServerSocketImpl.a(DashoA6275)  
at  
com.sun.net.ssl.internal.ssl.SSLServerSocketImpl.accept(DasoA6275)  
at SSLServerDemo.main(SSLServerDemo.java: 35)
```

- Si UD. obtiene una excepción, trate de permitir por lo menos un cipher suite soportado. En un entorno del mundo real, UD. probablemente no permita que a todos los cipher suites como lo hice en el programa SSLDemo, pero en su lugar UD. sólo permitirá a aquellos que fueron usados por el mecanismo de seguridad de su plataforma.
- El programa despliega los protocolos soportados de esta implementación, los cuales son SSL versión 3 y la versión 2Hello y también TLS versión 1.
- El programa se bloquea debido a que el socket servidor esta esperando por una conexión cliente, la cual se muestra como se crea a continuación.

## Secure Client Socket

Un cliente que desea conectarse a un `SSLServerSocket` debe usar un objeto `SSLSocket`, el cual es acoplado realizando los siguientes pasos:

1. El cliente inicia con un objeto `SSLSocketFactory`, el cual no es instanciado usando la palabra clave `new`, pero en su lugar este es obtenido usando el siguiente método estático que se encuentra en la clase `SSLSocketFactory`:

```
public static SocketFactory getDefault()
```

2. Use el socket factory para crear un objeto `SSLSocket`, el cual escucha en un puerto especificado por la petición de un cliente.

Después de que una conexión segura se ha realizado, el cliente y el servidor tienen un objeto `SSLSocket` para manejar la comunicación. La clase `SSLSocket` extiende a `java.net.Socket`, así el cliente y el servidor pueden invocar los métodos discutidos antes de la clase `Socket`, tales como `getOutputStream()` y `getInputStream()`. Existen también los métodos en `SSLSocket` únicos para obtener conexiones seguras de socket incluyendo los siguientes:

**public void startHandshake() throws IOException.** Inicia un SSL handshake para esta conexión, el cual establece la seguridad y la protección de la conexión. El handshaking es satisfactorio sólo si ambos clientes y servidor tienen un cipher suite común. Un handshaking es iniciado automáticamente cuando se hace un intento de leer y escribir desde un socket, pero UD. puede iniciar el handshake explícitamente con este método.

**public void setEnabledCipherSuites(String[] suites).** Configura el cipher suite disponible para esta conexión. El cliente y el servidor necesitan compartir por lo menos un cipher suite común antes de que el handshaking pueda ocurrir.

**public void addHandshakeCompleteListener(HandshakeCompletedListener h).** Agrega un escucha especificado a la conexión. Cuando el handshake se ha completado satisfactoriamente, el escucha es notificado vía el método `handshakeCompleted()` en la interface `HandshakeCompletedListener`.

**public SSLSession getSession().** Obtiene la sesión de esta conexión SSL, la cual es creada después de que ocurra el handshaking. Este objeto obtiene la información acerca de la conexión, por ejemplo, cual es el cipher suite y el protocolo que esta siendo usado, así como la identidad del cliente y servidor.

Note que la clase `SSLServer` es fuente de un evento `HandshakeCompletedEvent`. Esto le permite a UD. determinar la información acerca de los parámetros de seguridad usados para establecer la conexión, tal como el cipher suite usado o cualquier certificado de

seguridad usado. La siguiente clase MyHandshakeListener demuestra la codificación de un escucha para este evento.

```
import javax.net.ssl.*;

public class MyHandshakeListener implements HandshakeCompletedListener
{
    public void handshakeCompleted(HandshakeCompletedEvent e)
    {
        System.out.println("Handshake succesful!");
        System.out.println("Using cipher suite: " + e.getCipherSuite());
    }
}
```

El siguiente programa SSLClientDemo muestra a un cliente creando una conexión SSL hacia la aplicación SSLServerDemo usando la clase SSLSocket. Estudie el programa y trate de determinar que sucede con la salida que se ilustra en la Figura 17.5. No olvide correr primero el programa SSLServerDemo.

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

public class SSLClientDemo
{
    public static void main(String [] args)
    {
        String host = args[0];
        int port = Integer.parseInt(args[1]);

        try
        {
            System.out.println("Locating socket factory for SSL...");

            SSLSocketFactory factory = (SSLSocketFactory) SSLSocketFactory.getDefault();

            System.out.println("Creating secure socket to " + host + ":" + port);
            SSLSocket socket = (SSLSocket) factory.createSocket(host, port);

            System.out.println("Enabling all available cipher suites...");
            String [] suites = socket.getSupportedCipherSuites();
            socket.setEnabledCipherSuites(suites);

            System.out.println("Registering a handshake listener...");
            socket.addHandshakeCompletedListener(new MyHandshakeListener());
        }
    }
}
```

```

        System.out.println("Starting handshaking...");
        socket.startHandshake();

        System.out.println("Just connected to " + socket.getRemoteSocketAddress());

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

```

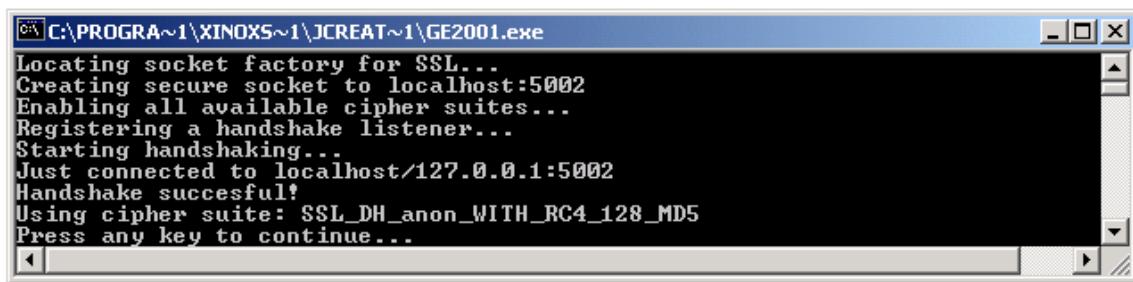


Figura 17.5 Output of the SSLClientDemo program, which starts a hanshake with the server to ensure that secure communication can occur successfully.

Hagamos algunos comentarios acerca del programa SSLClientDemo:

- El método getDefault() de SSLSocketFactory es usado para obtener la referencia al secure socket factory por defecto.
- El socket factory es usado para crear un SSLSocket en el puerto 5002 del localhost. En este ejemplo el programa servidor es el ejemplo SSLServerDemo discutido anteriormente.
- El cliente y el servidor necesitan un cipher suite común, así el cliente necesita permitir por lo menos un cipher suite que el servidor comprenda. Sólo he permitido a todos estos en el ejemplo y he dejado la implementación subyacente escoja un cipher suite.
- Un objeto MyHandshakeListener es registrado para que reciba la notificación de cuando el handshake se completa.

- El cliente específicamente inicia un handshake con el servidor el cual sucede debido a que el escucha fue notificado. El escucha despliega el cipher suite que fue usado para establecer esta conexión segura.

## Communicating over Secure Socket

El JSSE es usado para establecer una conexión de socket segura entre dos computadoras, como ha sido demostrado por los programas `SSLServerDemo` y `SSLClientDemo` discutidos en este capítulo. Después de que UD. ha hecho una conexión segura, la comunicación puede ocurrir similar a una con sockets no seguros. ( Un socket no seguro es aquel creado con las clases `java.net.SocketServer` y `java.net.Socket`.)

Por ejemplo, suponga que UD. necesita una aplicación para procesar las órdenes de tarjetas de créditos de sus clientes. Ud desea asegurar su conexión de tal forma que los datos pasados entre las computadoras no pueden ser interceptados por aplicaciones maliciosas. Deseo mostrarle un ejemplo simple pero útil de cómo esto puede acoplarse uniendo sockets seguro y serialización ( tal como fue discutido en el capítulo 16 “Input and Output”).

Supongamos que la orden es representada por una clase serializable con el nombre de `CustomOrder` que contenga los campos para los nombres del cliente, el número de tarjeta de crédito y la cantidad de la orden.

```
public class CustomerOrder implements java.io.Serializable
{
    public long creditCardNumber;
    public int expMonth;
    public int expYear;
    public double amountOfOrder;

    public CustomerOrder(long c, int m, int y, double a)
    {
        creditCardNumber = c;
        expMonth = m;
        expYear = y;
        amountOfOrder = a;
    }
}
```

El siguiente código de la clase `OrderHandler` corre en un hilo que espera por la conexión segura de un cliente, lee en un objeto sencillo `CustomerOrder`, procesa la orden y entonces cierra la conexión y espera por una nueva orden entrante:

```

import java.net.*;
import javax.net.ssl.*;
import java.io.*;

public class OrderHandler extends Thread
{
    private SSLServerSocket serverSocket;

    public OrderHandler(int port) throws IOException
    {
        SSLServerSocketFactory factory = (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();

        serverSocket = (SSLServerSocket) factory.createServerSocket(port);

        String [] suites = serverSocket.getSupportedCipherSuites();
        serverSocket.setEnabledCipherSuites(suites);
    }

    public void run()
    {
        while(true)
        {
            try
            {
                System.out.println("Waiting for order...");
                SSLSocket socket = (SSLSocket) serverSocket.accept();
                socket.startHandshake();

                ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
                CustomerOrder order = (CustomerOrder) in.readObject();

                System.out.println("*** Processing order ***");
                System.out.println("Amount: " + order.amountOfOrder);
                System.out.println("Card info: " + order.creditCardNumber + " " +
                                   order.expMonth + "/" + order.expYear);
                socket.close();
            }
            catch(Exception e)
            {
                e.printStackTrace();
                break;
            }
        }
    }
}

```



```

public static void main(String [] args)
{
    int port = Integer.parseInt(args[0]);

        try
        {
            Thread t = new OrderHandler(port);
            t.start();

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

```

El siguiente código del programa PurchaseDemo simula una orden que es enviada hacia una aplicación. Estudie el código y trate de determinar la salida de ambos programas y PurchaseDemo y OrderHandler, que se muestra en la Figura 17.6 y 17.7, respectivamente.

```

import java.net.*;
import javax.net.ssl.*;
import java.io.*;

public class PurchaseDemo
{
    public static void main(String [] args)
    {
        String host = args[0];
        int port = Integer.parseInt(args[1]);

        try
        {
            System.out.println("Locating socket factory for SSL...");
            SSLSocketFactory factory = (SSLSocketFactory) SSLSocketFactory.getDefault();

            System.out.println("Creating secure socket to " + host + ":" + port);
            SSLSocket socket = (SSLSocket) factory.createSocket(host, port);

            System.out.println("Enabling all available cipher suites...");
            String [] suites = socket.getSupportedCipherSuites();
            socket.setEnabledCipherSuites(suites);

            ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());

```

```

        System.out.println("Sending order...");
        CustomerOrder order = new CustomerOrder(1111222233334444L, 1, 2010, 853.79);
        out.writeObject(order);

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

```

Al correr el programa OrderHandler en la línea de comando java OrderHandler 5003, obtenemos la salida:

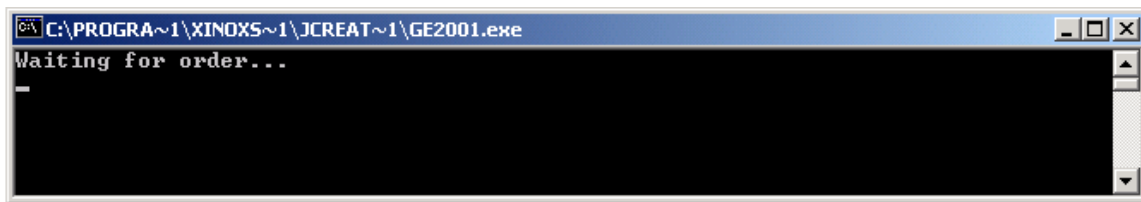


Figura 17.7A Salida inicial del programa OrderHandler.

Al correr el programa PurchaseDemo, el cual al correrlo en la línea de comando java PurchaseDemo localhost 5003 obtenemos la salida:

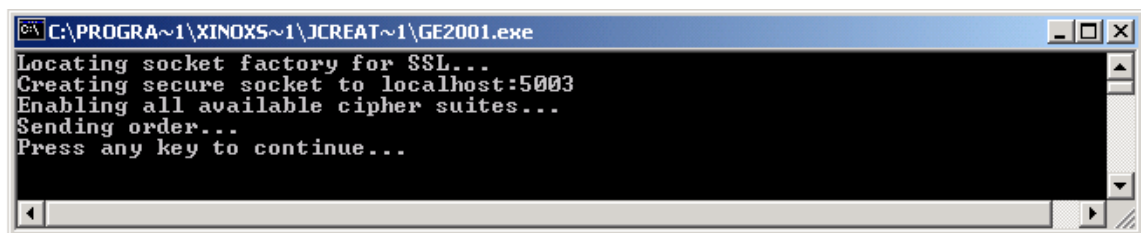


Figura 17.6 Salida del programa PurchaseDemo, este envía un objeto sencillo CustomerOrder al programa OrderHandler sobre un socket seguro.

Luego en la salida del programa OrderHandler se visualiza:

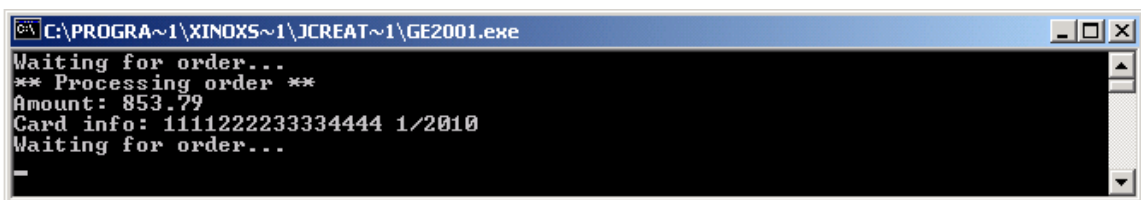


Figura 17.7B Salida final del programa OrderHandler, el cual deserializa el objeto CustomerOrder y despliega la información.

Observe que en la Figura 17.7 que OrderHandler espera por una orden, procesa la orden y luego inmediatamente espera por otra orden.

También note que el cliente no inicia con un handshake explícitamente usando el método `startHandshake()`. Sin embargo, un handshake ocurre automáticamente cuando un cliente intenta escribir un objeto `CustomerOrder` al flujo de salida del socket.

## Overview of Datagram Packets

El User Datagram Protocol (UDP) es un protocolo para enviar datos binarios desde una computadora a otra. Los datos son referidos como un paquete de datagrama el cual también contiene el servidor destino y el número del puerto al cual el dato debe ser entregado. El que envía el mensaje usa un sockets datagrama para enviar un paquete y un recipiente usa un socket datagrama para recibir un mensaje. Cuando un mensaje es enviado, el recipiente no necesita estar disponible. Similarmente, cuando un mensaje es recibido, el que envía el mensaje no necesita mantenerse disponible.

### DatagramSocket Class

La clase `java.net.DatagramSocket` es usada por ambos el que envía y el que recibe el paquete de datagrama para enviar y recibir un paquete respectivamente. La clase `DatagramSocket` tiene cuatro constructores públicos:

**`public DatagramSocket( int port ) throws SocketException.`** Crea un socket datagrama en el localhost de la computadora en el puerto especificado.

**`public DatagramSocket( int port, InetAddress address ) throws SocketException.`** Crea un socket datagrama en el puerto especificado y en la dirección local, la cual es útil si el computador tiene múltiples direcciones.

**`public DatagramSocket( SocketAddress address ) throws SocketException.`** Crea un socket datagrama en la `SocketAddress` especificada, la cual encapsula el nombre del servidor y el número de puerto.

**`public DatagramSocket( ) throws SocketException.`** Crea un socket datagrama que no esta ceñido. Use el método `bind()` de la clase `DatagramSocket` para ceñir el socket a un puerto.

He aquí algunos métodos de interés de la clase `DatagramSocket`:

**`public void send( DatagramPacket packet ) throws IOException.`** Envía el paquete de datagrama especificado. El objeto `DatagramPacket` contiene la información del destino del paquete.

**`public void receive( DatagramPacket packet ) throws IOException.`** Recibe un paquete de datagrama almacenado en su argumento. Este método se bloquea y no retorna hasta que el paquete datagrama es recibido o el socket se desconecta debido al vencimiento del tiempo de caducidad. Si el tiempo de caducidad del socket ocurre se lanza una excepción del tipo `SocketTimeoutException`.

**`public void setTimeout( int timeout) throws SocketTimeoutException.`** Asigna el valor del tiempo de caducidad del socket, el cual determina el número de milisegundos que el método `receive()` se bloquearía.

**public void connect( InetAddress address, int port).** Aun que UDP es un protocolo no orientado a conexión, este método ha sido agregado a la clase DatagramSocket, desde J2SE 1.4. Los paquetes aun se envían y reciben usando los métodos send() y receive(); pero el conectar dos sockets de datagrama mejora la realización de la entrega ya que la verificación de seguridad solo se realiza una vez.

**public void disconnect( ).** Desconecta cualquier conexión actual.

### **DatagramPacket Class**

Note que los métodos send() y receive() de la clase DatagramSocket tiene un parámetro de la clase DatagramPacket. La clase DatagramPacket representa a un paquete de datagrama y ( al igual que DatagramSocket ) este es usado por ambos, el que envía y el que recibe un paquete. La clase DatagramPacket tiene seis constructores: dos para los que reciben y cuatro para los que envían.

Los siguientes dos constructores de la clase DatagramPacket son usados para recibir un paquete de datagrama:

**public DatagramPacket( byte[] buffer, int length ).** Crea un paquete de datagrama para recibir del tamaño especificado. El buffer contendrá el paquete entrante.

**public DatagramPacket( byte[] buffer, int offset, int length ).** Igual que el constructor previo, excepto que los datos del paquete entrante son almacenados en la posición del arreglo de byte especificado por el parámetro offset.

El arreglo de bytes pasado en estos dos constructores es usado para contener los datos del paquete entrante y típicamente son arreglos vacíos. Si estos no están vacíos, entonces el paquete de datagrama entrante sobre escribe los datos en el arreglo.

Los siguientes cuatro constructores son usados para enviar un paquete de datagrama.

**public DatagramPacket( byte[] buffer, int length , InetAddress address, int port).** Crea un paquete de datagrama para enviar un paquete de tamaño específico. El buffer contiene los datos del paquete y la dirección y el puerto denota el recipiente.

**public DatagramPacket( byte[] buffer, int length, SocketAddress address ).** Similar al constructor previo, excepto que el nombre y el número del puerto del recipiente están contenidos en el argumento SocketAddress.

**public DatagramPacket( byte[] buffer, int offset, int length, InetAddress address, int port ).** Le permite a UD. denotar el parámetro offset en un arreglo, (sólo con el argumento length) el cual determina a un subconjunto de arreglos de bytes que representan los datos.

**public DatagramPacket( byte[] buffer, int offset, int length, SocketAddress address).** Similar al constructor previo, excepto que el nombre y el número de puerto del recipiente están contenido en el argumento SocketAddress.

Note que cada uno de los seis constructores toma un arreglo de bytes. Cuando se recibe un paquete, el arreglo inicia vacío y es llenado con el paquete datagrama entrante. Cuando se esta enviando un paquete, el arreglo de byte contiene los datos del paquete que será enviado.

La clase DatagramPacket contiene los métodos accesoros y mutadores con varios atributos del paquete de datagrama:

**public byte [] getData( ).** Retorna los datos del buffer.

**public void setData( byte [] buffer ).** Asigana los datos del paquete.

**public int getLength( ).** Retorna la longitud de los datos que son enviados o recibidos.

**public SocketAddress getSocketAddress( ).** Retorna la dirección del anfitrión (host) remoto cuando el mensaje ha sido enviado o recibido desde el anfitrión remoto.

**public void setSocketAddress( SocketAddress address).** Asigna la dirección del anfitrión remoto desde donde el mensaje haya sido enviado o recibido desde el anfitrión.

Ahora veamos un ejemplo de cómo usar estas clases para enviar o recibir un paquete de datagrama usando UDP.

## Receiving a Datagram Packet

Para recibir un paquete de datagrama se realizan los siguientes pasos:

1. Se Crea un arreglo de bytes lo suficientemente grande para que contenga los datos del paquete entrante.
2. Un objeto Datagram es instanciado usando un arreglo de bytes.
3. Un objeto DatagramSocket es instanciado y a este le es especificado con que puerto ( y la dirección específica del localhost, si es necesaria) en el localhost el socket ha de ceñirse.
4. El método receive() de la clase DatagramSocket es invocado, pasando un objeto DatagramPacket. Esto causa que el hilo se bloquee hasta que un paquete de datagrama es recibido o el tiempo de caducidad se venza.

Después de que el método receive() retorna, un nuevo paquete acaba de llegar satisfactoriamente. ( Note que si el tiempo de caducidad se vence, el método receive() no retorna, pero en su lugar lanza una excepción.) El método getData() de la clase

DatagramPacket puede ser usado para obtener del arreglo de bytes, los datos que contiene este paquete.

El siguiente programa PacketReceiver demuestra los pasos involucrados en el proceso de recibimiento de un paquete de datagrama:

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

public class PacketReceiver
{
    public static void main(String [] args)
    {
        try
        {
            byte [] buffer = new byte[1024];
            DatagramPacket packet = new DatagramPacket(buffer, buffer.length);

            DatagramSocket socket = new DatagramSocket(5002);

            System.out.println("Waiting for a packet...");
            socket.receive(packet);

            System.out.println("Just received packet from " + packet.getSocketAddress());

            buffer = packet.getData();

            System.out.println(new String(buffer));

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

## **Sending a Datagram Packet**

Para recibir un paquete de datagrama, se realizan los siguientes pasos:

1. Se crea un arreglo de bytes lo suficientemente grande para que pueda contener los datos del paquete que se va a enviar y llenar el arreglo con los datos.
2. Se crea un nuevo objeto DatagramPacket que contiene un arreglo de bytes, así como el nombre del servidor y el número del puerto del recipiente.
3. Un objeto del tipo DatagramSocket es creado y le es especificado con cual puerto ( y especificado la dirección del localhost, si es necesaria) en el localhost el socket se ceñirá.
4. El método send() de la clase DatagramSocket es invocado, pasándole un objeto DatagramPacket.

La siguiente clase PacketSender envía un paquete que contiene una cadena. Note que un objeto de tipo String es convertido a un arreglo de bytes usando el método getBytes() de la clase String.

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

public class PacketSender
{
    public static void main(String [] args)
    {
        try
        {
            String data = "You have just received a packet of data sent using UDP";
            byte [] buffer = data.getBytes();

            DatagramPacket packet = new DatagramPacket(buffer, buffer.length,
                                                         new InetSocketAddress("localhost", 5002));

            DatagramSocket socket = new DatagramSocket(5003);

            System.out.println("Sending a packet...");
            socket.send(packet);
        } catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```



El programa PacketSender envía un paquete al programa PacketReceiver discutido antes. La Figura 17.8 muestra la salida del programa PacketSender.



Figura 17.8 PacketSender envía un paquete de datagrama que contiene una cadena.

La Figura 17.9 muestra la salida generada por el programa PacketReceiver cuando el paquete ha sido entregado. Note que el espacio en blanco en la Figura 17.9 es causado por el arreglo de bytes que no esta completamente lleno con el datagrama entrante.

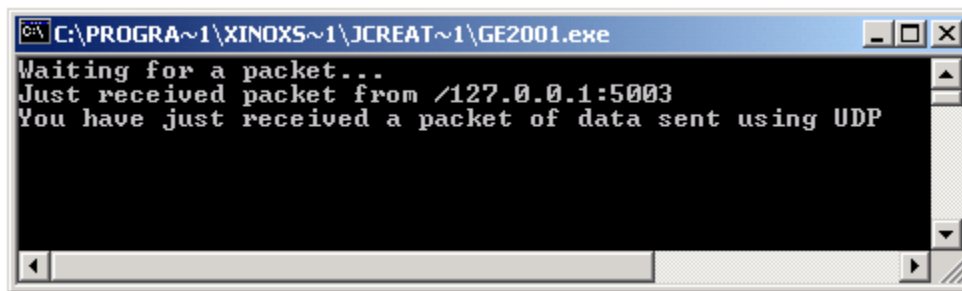


Figura 17.9 El método receive() invocado dentro de Packetreceiver retorna después de que el paquete de datagrama ha sido recibido.

## Working with URLs

Hemos discutido los sockets y los paquetes de datagramas como las opciones para crear aplicaciones de redes en Java. En esta sección le mostraremos a UD. cómo escribir programas en Java que se comunican con un URL. URL, el cual sirve para Uniform Resource Locator, representa un recurso en el Word Wide Web, tal como una página Web o un directorio FTP.

Nota: Un URL es actualmente un tipo de URI , Uniform Resource Identifier. Un URI identifica un recurso, pero no contiene información acerca de cómo acceder el recurso. Un URI identifica un recurso y a un protocolo para acceder el recurso. Un URI está representado en Java usando la clase `java.net.URI`.

Un URI puede ser dividido en partes, así:

`protocol://host:port/path?query#ref`

El path también se conoce el nombre del archivo: filename y el anfitrión (host) es también llamado la autoridad. Ejemplos de protocolos incluyen: HTTP, HTTPS, FTP y File. Por ejemplo, el siguiente es un URL a una página Web cuyo protocolo es http:

<http://www.javalicense.com/training/index.html?language=en#j2se>

Note que este URL no especifica el puerto, en tal caso es usado el puerto por defecto. Con HTTP el puerto por defecto es el 80.

La clase `java.net.URL` es la clase que representa un URL. La clase URL tiene varios constructores para la creación de URL, incluyendo:

**`public URL( String protocol, String host, int port, String file ) throws MalformedURLException.`** Crea un URL juntando las partes dadas.

**`public URL( String protocol, String host, String file ) throws MalformedURLException.`** Idéntico al constructor previo excepto que es usado el puerto por defecto para el protocolo dado.

**`public URL( String url ) throws MalformedURLException.`** Crea un URL de la cadena dada.

**`public URL( URL context, String url ) throws MalformedURLException.`** Crea un URL analizando juntos los argumentos del URL y la cadena.

La clase URL contiene muchos métodos para acceder las partes de cómo un URL es representado. Algunos de los métodos en la clase URL incluyen los siguientes:

**public String getPath().** Retorna el path del URL

**public String getQuery().** Retorna la parte de la petición (query) del URL.

**public String getAuthority().** Retorna la autoridad del URL.

**public int getPort().** Retorna el Puerto de URL.

**public int getDefaultPort().** Retorna el Puerto por defecto del protocolo URL.

**public String getProtocol().** Retorna el protocolo del URL.

**public String getHost().** Retorna el anfitrión (host) del URL.

**public String getFile().** Retorna el nombre del archivo (filename) del URL.

**public String getRef().** Retorna la parte de la referencia de URL.

**public URLConnection openConnection() throws IOException.** Abre una conexión hacia un URL, permitiendo a un cliente comunicarse con el recurso.

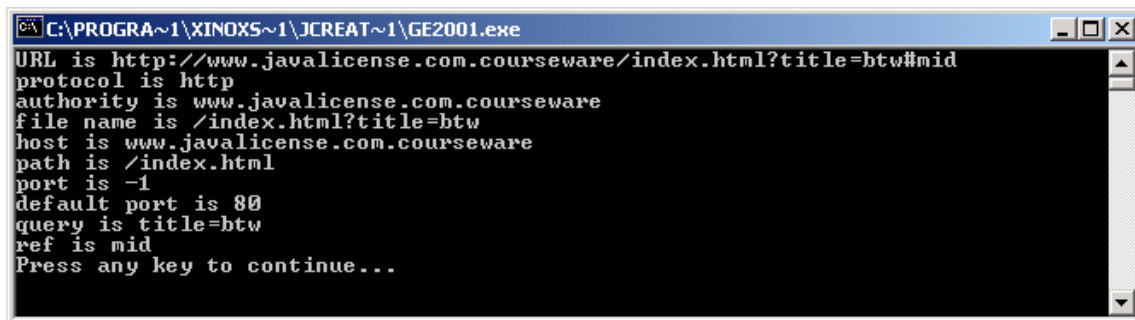
El siguiente programa URLDemo demuestra que hacen estos métodos y también demuestra varias partes de un URL. Un URL es ingresado en la línea de comando y la salida del programa URLDemo de cada parte del URL dado. Estudie el programa y trate de determinar la salida cuando el argumento en la línea de comando es:

<http://www.javalicense.com.courseware/index.html?title=btw#mid>

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

public class URLDemo
{
    public static void main(String [] args)
    {
        try
        {
            URL url = new URL(args[0]);
            System.out.println("URL is " + url.toString());
            System.out.println("protocol is " + url.getProtocol());
            System.out.println("authority is " + url.getAuthority());
            System.out.println("file name is " + url.getFile());
            System.out.println("host is " + url.getHost());
            System.out.println("path is " + url.getPath());
            System.out.println("port is " + url.getPort());
            System.out.println("default port is " + url.getDefaultPort());
            System.out.println("query is " + url.getQuery());
            System.out.println("ref is " + url.getRef());
        } catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

La salida del programa URLDemo con este URL es mostrada en la Figura 17.10.



```
C:\PROGRA~1\XINOX5~1\JCREAT~1\GE2001.exe
URL is http://www.javalicense.com.courseware/index.html?title=btw#mid
protocol is http
authority is www.javalicense.com.courseware
file name is /index.html?title=btw
host is www.javalicense.com.courseware
path is /index.html
port is -1
default port is 80
query is title=btw
ref is mid
Press any key to continue...
```

Figura 17.10 La clase URLDemo despliega cada parte de un URL.

## URL Connections

Usando el método `openConnection()` de la clase `URL`, UD. puede conectarse a un URL y comunicarse con el recurso. El método `openConnection()` retorna un `java.net.URLConnection`, una clase abstracta cuya subclases representan a varios tipos de conexiones URL. Por ejemplo, si UD. se conecta con un URL cuyo protocolo es `http`, el método `openConnection()` retorna un objeto `HttpURLConnection`. Si UD. se conecta con un URL que representa a un archivo JAR, el método `openConnection()` retorna un objeto `JarURLConnection`.

La clase `URLConnection` tiene muchos métodos para la configuración o determinación de la información acerca de la conexión, incluyendo los siguientes:

**`public void setDoInput( boolean input )`**. El pase con `true` para denotar que la conexión será usada para entrada (input). El valor por defecto es `true` por que los clientes típicamente leen desde una conexión `URLConnection`.

**`public void setDoOutput( boolean output )`**. El pase con `true` para denotar que la conexión será usada para salida (output). El valor por defecto es `true` por que muchos tipos de URL no soportan que se les escriba.

**`public InputStream getInputStream( ) throws IOException`**. Retorna el flujo de entrada de una conexión URL para leer desde la fuente.

**`public OutputStream getOutputStream() throws IOException`**. Retorna el flujo de salida de una conexión URL para escribir en un recurso.

**`public URL getURL() .`** Retorna el URL del objeto `URLConnection` al cual esta conectado.

La clase `URLConnection` también contiene métodos para acceder la información del encabezado de la conexión permitiéndole a UD. determinar el tipo y la longitud del contenido del URL, el día de la última modificación del contenido, el contenido codificado. Asegúrese de verificar la documentación para la lista de todos los métodos en la clase `URLConnection`.

El siguiente programa `URLConnection` se conecta a un URL ingresado en la línea de comando. Si el URL representa un recurso HTTP, la conexión es moldeada (cast) hacia un `HttpURLConnection` y los datos en este recurso son leídos una línea a la vez. La Figura 17.11 muestra la salida del programa para el URL <http://www.javalice.com>.

```

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

public class URLConnectionDemo
{
    public static void main(String [] args)
    {
        try
        {
            URL url = new URL(args[0]);
            URLConnection urlConnection = url.openConnection();

            HttpURLConnection connection = null;

            if(urlConnection instanceof HttpURLConnection)
            {
                connection = (HttpURLConnection) urlConnection;
            }
            else
            {
                System.out.println("Please enter an HTTP url");
                return;
            }

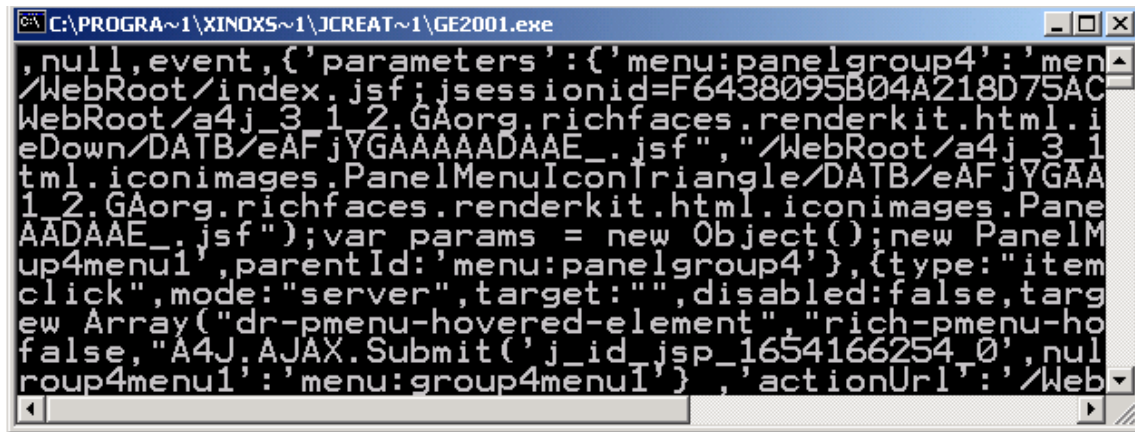
            BufferedReader in = new BufferedReader(new
InputStreamReader(connection.getInputStream()));

            String urlString = "";
            String current;
            while((current = in.readLine()) != null)
            {
                urlString += current;
            }

            System.out.println(urlString);

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

```



```
,null,event,{ 'parameters': { 'menu:panelgroup4': 'men
/WebRoot/index.jsf;jsessionid=F6438095B04A218D75AC
WebRoot/a4j_3_1_2.GAorg.richfaces.renderkit.html.i
eDown/DATB/eAFjYGAAAAADAAE_.jsf", "/WebRoot/a4j_3_1
tml.iconimages.PanelMenuIconTriangle/DATB/eAFjYGAA
1_2.GAorg.richfaces.renderkit.html.iconimages.Pane
AADAAE_.jsf");var params = new Object();new PanelM
up4menu1',parentId:'menu:panelgroup4'}, {type:"item
click",mode:"server",target:"",disabled:false,targ
ew Array("dr-pmenu-hovered-element","rich-pmenu-ho
false,"A4J.AJAX.Submit('j_id_jsp_1654166254_0',nul
roup4menu1':'menu:group4menu1'} , 'actionUrl':'/Web
```

Figura 17.11 El programa URLConnectionDemo, muestra los contenidos de una página HTML. Esta figura muestra las partes de la salida del URL <http://www.javalicense.com>.

## Lab 17.1 Using Sockets

El propósito de este laboratorio es familiarizarse con el trabajo con los sockets:

UD. escribirá programas de redes clientes/servidor que permiten una conversación simple que tiene lugar entre dos computadoras.

1. Escriba un programa llamado TalkServer que usa un objeto ServerSocket para escuchar los clientes por el puerto 5050.
2. Cuando el cliente se conecta, el programa TalkServer tiene que leer una cadena (String) desde el cliente usando el método readUTF() de la clase DataInputStream. UD. necesitará crear un nuevo DataInputStream usando un flujo de entrada del socket. Imprima la cadena (String) que ha sido leída.
3. El programa TalkServer debe enviar un String al cliente usando el método writeUTF() de la clase DataOutputStream, enviando la cadena (String) al socket del cliente. La cadena que es enviada al cliente debe ser ingresada usando el teclado.
4. Este proceso debe continuar hasta que la conexión se haya perdido de alguna forma. En otras palabras, el TalkServer lee un String, la despliega y luego ingresa una String desde el teclado y envía esta al cliente.
5. Escriba el programa llamado TalClient que use la clase Socket para conectarse al servidor socket del programa TalkServer.
6. El cliente debe enviar un String de entrada desde el teclado al servidor usando el método writeUTF(). Entonces el TalkClient debe leer un String desde el Servidor usando el método readUTF(), desplegando la cadena (String).
7. Este proceso debe continuar hasta que la conexión se pierda. El cliente y el servidor están ahora comunicados uno con el otro en una conversación simple.

La salida inicial del servidor debe ser la cadena (String) leída desde el cliente. La salida inicial en el cliente debe ser debe solicitarle a este que ingrese un mensaje para ser enviado al servidor. Cuando un cliente ingresa un mensaje y oprime Enter, el mensaje debe ser desplegado en el servidor. Similarmente cuando el usuario oprime Enter, el mensaje debe desplegarse en el cliente.



## Lab 17.2: Using Datagram Packets

El propósito de este laboratorio es familiarizarse con el envío y recibo de paquetes de datagramas. UD. escribirá un programa que simula el proceso de actualización del estado del tiempo, en el cual el estado del tiempo se actualiza y se envía a un escucha cada 15 segundos usando paquetes de datagramas.

1. Escriba una clase llamada `WeatherUpdater` que extienda a `TimerTask`. Agregue un campo de tipo `DatagramSocket`, un campo de tipo `String` con el nombre `recipientName` y un campo de tipo entero con el nombre `recipientPort`. Agregue un constructor que tome un `String` y un entero para estos dos campos y también inicialice un objeto `DatagramScket` en el constructor usando cualquier puerto disponible.
2. Dentro del método `run()` de la clase `WeatherUpdater` cree un `String` que se vea como: "Current temperatura: 40 Farenheit", donde 40 es un número aleatorio generado entre 0 y 100 que cambia cada vez que el método es invocado.
3. Deentro del método `run()`, instancia un objeto `DatagramPacket` apropiado para enviar un paquete. El arreglo de bytes debe ser el `String` en el caso previo convertido a bytes. Envíelo al recipiente denotado por los campos `recipientName` y `recipientPort` de la clase.
4. Agregue el método `main()` a su clase `WeatherUpdater`. Dentro del método `main()`, instancie un nuevo objeto `WeatherUpdater`, usando en la línea de comandos en línea los primeros dos argumentos que son campos de la clase, `recipientName` y el `recipientPort`.
5. Dentro del método `main()`, instancia un objeto `Timer` y use este objeto para establecer el itinerario (`schedule`) de `WeatherUpdater` con un `fixed-rate-schedule` para cada 15 segundos.
6. Guarde y compile la clase `WeatherUpdater`.
7. Escriba una clase llamada `WeatherWatcher` que contenga el `main()`.
8. Dentro del `main()`, instancie un objeto `DatagramSocket` en el puerto 4444. También instancia un objeto `DatagramPacket` apropiado para recibir, usando un arreglo de 128 bytes de tamaño.
9. Invoque el método `receive()` dentro del `main()` de tal manera que el hilo bloquee hasta que el paquete datagrama sea entregado.

10. Después de que el paquete ha sido entregado, despliegue su contenido. Cree un bucle tal que el WeatherWatcher entonces invoque al método receive() nuevamente, para estar listo para que el nuevo paquete sea entregado.
11. Guarde, compile y ejecute el programa WeatherWatcher.
12. Corra el programa WeatherUpdate, pasando el nombre apropiado en el anfitrión (localhost si ambos corren en la misma computadora) y el número de puerto 4444.

Probablemente el WeatherUpdater no desplegará ninguna salida. Sin embargo en el programa WeatherWatcher UD. debería ver la sentencia: “Current temperatura: N Fahrenheit” cada 15 segundos con la temperatura cambiando aleatoriamente.

### Lab 17.3: The InstantMessage server

Este laboratorio es una continuación del proyecto InstantMessaging de los capítulos previos. En este laboratorio, Ud. escribirá una aplicación que recibe un objeto InstantMessage desde el que envía y envía el mensaje a su correspondiente recipiente.

1. Escriba una clase llamada IMServer que implemente a Runnable. Agregue un campo que contenga a todos los objetos Participant que actualmente están en el sistema.
2. Dentro del método run(), cree un ServerSocket en el puerto 5555. Invoque el método accept() y espere que un cliente se conecte.
3. Cuando un cliente se conecte, ellos enviarán un String usando el método writeUTF() de la clase DataOutputStream. Lea en este String usando el método readUTF() de la clase DataInputStream, el cual será el nombre del usuario. Entonces, instancia a un nuevo Participant, pasándole el socket y su nombre de usuario en el constructor.
4. Debido a que Participant es un Thread, inícielo. Realice estos pasos en un bucle tal que accept() sea invocado en el ServerSocket después de que un nuevo hilo Participant es iniciado.
5. Guarde, compile y ejecute el programa Inservir.

No habrá ninguna salida hasta que escribamos el programa cliente en el próximo laboratorio. Por ahora, su programa IMServer debe estar bloqueado esperando que un cliente se conecte.

## Lab 17.4 Finishing the InstantMessage Program

En este laboratorio UD. Terminará el programa InstantMessage.

1. Para hacer que su aplicación InstantMessage se comunice con el servidor, la única modificación necesita aparecer en el constructor de la clase InstantMessageDialog. Abra esta clase en su editor de texto.
2. Comente o remueva cualquier código que involucre tuberías.
3. Dentro del constructor de InstantMessageDialog, cree una conexión Socket al programa IMServer. Use la entrada y salida de flujos de este socket para instanciar los objetos SendMessage y Participant.
4. Guarde, compile y ejecute la clase InstantMessageDialog.

Su programa puede ahora ser ejecutado en dos computadoras diferentes en una red o simplemente ejecútelos dos veces en su computadora. Cuando un mensaje es enviado este debe aparecer en la ventana de mensaje instantánea del amigo apropiado.

### Lab 17.5: Connecting to URLs

El propósito de este laboratorio es familiarizarse con la conexión a los URL. En este laboratorio UD. creará una interfase gráfica GUI con Swing, la cual puede ser usada para ver el código fuente de un documento HTML. Le daremos una guía para la aplicación la cual le permite a UD. como diseñarlo y escribirlo.

1. Cree un JFrame que contiene un texto en el centro para ver una página HTML y un campo de texto en el sur para ingresar el URL.
2. Cuando el usuario ingrese un URL en el campo de texto, el programa se debe conectar al URL, leer su contenido y desplegarlo en el área de texto.
3. Use la clase `URLConnection` para leer el contenido del URL.
4. Verifique el paquete `javax.swing.text.html` para las clases que puedan ser de interés, notablemente la clase `HTMLEditorKit`.
5. Para hacer la clase mas funcional, considere el agregar un menú con menú ítems que le permita al usuario guardar el contenido del área de texto en un archivo con extensión `.html` en el disco duro. La clase `javax.swing.JFileChooser` puede ser conveniente aquí.

Cuando el programa es ejecutado, el usuario debe poder escribir un URL en el campo de texto, y luego se puede ver el código fuentes de ese HTML en el área de texto.

## Summary

- Java contiene API para el desarrollo de aplicaciones en red que usan los protocolos TCP/IP y UDP.
- Las clases `java.net.ServerSocket` y `java.net.Socket` son usadas para crear conexiones TCP/IP entre dos aplicaciones en Java. La comunicación es realizada usando las clases `java.io`.
- El java Secure Socket Extension (JSSE) permite conexiones seguras entre dos computadoras, usando el protocolo Secure Sockets Layer (SSL). Esto se realiza usando las clases `javax.net.ssl.SSLServerSocket` y `javax.net.ssl.SSLServerSocket`.
- La clase `java.net.DatagramSocket` es usada para enviar y recibir los paquetes de datagramas. La clase `java.net.DatagramPacket` es usada para representar los datos en el paquete.
- La clase `java.net.URL` es usada para conectar y leer datos desde un URL.

## Review Questions

## Answer to Review Questions