JAVAWORLD
SOLUTIONS FOR JAVA DEVELOPERS

This story appeared on JavaWorld at
http://www.javaworld.com/javaworld/jw-12-1996/jw-12-sockets.html

Sockets programming in Java: A tutorial

Writing your own client/server applications can be done seamlessly using Java

By Qusay H. Mahmoud, JavaWorld.com, 12/11/96

This tutorial presents an introduction to sockets programming over TCP/IP networks and shows how to write client/server applications in Java.

A bit of history

The Unix input/output (I/O) system follows a paradigm usually referred to as *Open-Read-Write-Close*. Before a user process can perform I/O operations, it calls *Open* to specify and obtain permissions for the file or device to be used. Once an object has been opened, the user process makes one or more calls to *Read* or *Write* data. *Read* reads data from the object and transfers it to the user process, while *Write* transfers data from the user process to the object. After all transfer operations are complete, the user process calls *Close* to inform the operating system that it has finished using that object.

When facilities for InterProcess Communication (IPC) and networking were added to Unix, the idea was to make the interface to IPC similar to that of file I/O. In Unix, a process has a set of I/O descriptors that one reads from and writes to. These descriptors may refer to files, devices, or communication channels (sockets). The lifetime of a descriptor is made up of three phases: creation (open socket), reading and writing (receive and send to socket), and destruction (close socket).

The IPC interface in BSD-like versions of Unix is implemented as a layer over the network TCP and UDP protocols. Message destinations are specified as socket addresses; each socket address is a communication identifier that consists of a port number and an Internet address.

The IPC operations are based on socket pairs, one belonging to a communication process. IPC is done by exchanging some data through transmitting that data in a message between a socket in one process and another socket in another process. When messages are sent, the messages are queued at the sending socket until the underlying network protocol has transmitted them. When they arrive, the messages are queued at the receiving socket until the receiving process makes the necessary calls to receive them.

TCP/IP and UDP/IP communications

There are two communication protocols that one can use for socket programming: datagram communication and stream communication.

Datagram communication:
The datagram communication protocol, known as UDP (user datagram protocol), is a connectionless protocol, meaning that each time you send datagrams, you also need to send the local socket descriptor and the receiving socket's address. As you can tell, additional data must be sent each time a communication is made.

Stream communication:
The stream communication protocol is known as TCP (transfer control protocol). Unlike UDP, TCP is a connection-oriented protocol. In order to do communication over the TCP protocol, a connection must first be established between the pair of sockets. While one of the sockets listens for a connection request (server), the other asks for a connection (client). Once two sockets have been connected, they can be used to transmit data in both (or either one of the) directions.

Now, you might ask what protocol you should use -- UDP or TCP? This depends on the client/server application you are writing. The following discussion shows the differences between the UDP and TCP protocols; this might help you decide which protocol you should use.

In UDP, as you have read above, every time you send a datagram, you have to send the local descriptor and the socket address of the receiving socket along with it. Since TCP is a connection-oriented protocol, on the other hand, a connection must be established before communications between the pair of sockets start. So there is a connection setup time in TCP.

In UDP, there is a size limit of 64 kilobytes on datagrams you can send to a specified location, while in TCP there is no limit. Once a connection is established, the pair of sockets behaves like streams: All available data are read immediately in the same order in which they are received.

UDP is an unreliable protocol -- there is no guarantee that the datagrams you have sent will be received in the same order by the receiving socket. On the other hand, TCP is a reliable protocol; it is guaranteed that the packets you send will be received in the order in which they were sent.

In short, TCP is useful for implementing network services -- such as remote login (rlogin, telnet) and file transfer (FTP) -- which require data of indefinite length to be transferred. UDP is less complex and incurs fewer overheads. It is often used in implementing client/server applications in distributed systems built over local area networks.

Programming sockets in Java

In this section we will answer the most frequently asked questions about programming sockets in Java. Then we will show some examples of how to write client and server applications.

Note: *In this tutorial we will show how to program sockets in Java using the TCP/IP protocol only since it is more widely used than UDP/IP.* ***Also:*** *All the classes related to sockets are in the* java.net *package, so make sure to import that package when you program sockets.*

How do I open a socket?

If you are programming a client, then you would open a socket like this:

```
Socket MyClient;
MyClient = new Socket("Machine name", PortNumber);
```

Where Machine name is the machine you are trying to open a connection to, and PortNumber is the port (a number) on which the server you are trying to connect to is running. When selecting a port number, you should note that port numbers between 0 and 1,023 are reserved for privileged users (that is, super user or root). These

port numbers are reserved for standard services, such as email, FTP, and HTTP. When selecting a port number for your server, select one that is greater than 1,023!

In the example above, we didn't make use of exception handling, however, it is a good idea to handle exceptions. (From now on, all our code will handle exceptions!) The above can be written as:

```
Socket MyClient;
try {
       MyClient = new Socket("Machine name", PortNumber);
}
catch (IOException e) {
    System.out.println(e);
}
```

If you are programming a server, then this is how you open a socket:

```
ServerSocket MyService;
try {
   MyServerice = new ServerSocket(PortNumber);
    }
    catch (IOException e) {
       System.out.println(e);
    }
```

When implementing a server you also need to create a socket object from the `ServerSocket` in order to listen for and accept connections from clients.

```
Socket clientSocket = null;
try {
   serviceSocket = MyService.accept();
    }
catch (IOException e) {
   System.out.println(e);
}
```

How do I create an input stream?

On the client side, you can use the `DataInputStream` class to create an input stream to receive response from the server:

```
DataInputStream input;
try {
   input = new DataInputStream(MyClient.getInputStream());
}
catch (IOException e) {
   System.out.println(e);
}
```

The class `DataInputStream` allows you to read lines of text and Java primitive data types in a portable way. It has methods such as `read`, `readChar`, `readInt`, `readDouble`, and `readLine`,. Use whichever function you

think suits your needs depending on the type of data that you receive from the server.

On the server side, you can use `DataInputStream` to receive input from the client:

```
DataInputStream input;
try {
   input = new DataInputStream(serviceSocket.getInputStream());
}
catch (IOException e) {
   System.out.println(e);
}
```

How do I create an output stream?

On the client side, you can create an output stream to send information to the server socket using the class `PrintStream` or `DataOutputStream` of java.io:

```
PrintStream output;
try {
   output = new PrintStream(MyClient.getOutputStream());
}
catch (IOException e) {
   System.out.println(e);
}
```

The class PrintStream has methods for displaying textual representation of Java primitive data types. Its Write and println methods are important here. Also, you may want to use the `DataOutputStream`:

```
DataOutputStream output;
try {
   output = new DataOutputStream(MyClient.getOutputStream());
}
catch (IOException e) {
   System.out.println(e);
}
```

The class `DataOutputStream` allows you to write Java primitive data types; many of its methods write a single Java primitive type to the output stream. The method `writeBytes` is a useful one.

On the server side, you can use the class `PrintStream` to send information to the client.

```
PrintStream output;
try {
   output = new PrintStream(serviceSocket.getOutputStream());
}
catch (IOException e) {
   System.out.println(e);
}
```

Note: You can use the class `DataOutputStream` as mentioned above.

How do I close sockets?

You should always close the output and input stream before you close the socket.

On the client side:

```
    try {
           output.close();
           input.close();
       MyClient.close();
    }
    catch (IOException e) {
       System.out.println(e);
    }
```

On the server side:

```
    try {
       output.close();
       input.close();
       serviceSocket.close();
       MyService.close();
    }
    catch (IOException e) {
       System.out.println(e);
    }
```

Examples

In this section we will write two applications: a simple SMTP (simple mail transfer protocol) client, and a simple echo server.

1. SMTP client

Let's write an SMTP (simple mail transfer protocol) client -- one so simple that we have all the data encapsulated within the program. You may change the code around to suit your needs. An interesting modification would be to change it so that you accept the data from the command-line argument and also get the input (the body of the message) from standard input. Try to modify it so that it behaves the same as the mail program that comes with Unix.

```
import java.io.*;
import java.net.*;
public class smtpClient {
    public static void main(String[] args) {
// declaration section:
// smtpClient: our client socket
// os: output stream
// is: input stream
        Socket smtpSocket = null;
        DataOutputStream os = null;
        DataInputStream is = null;
// Initialization section:
// Try to open a socket on port 25
// Try to open input and output streams
        try {
```

```
            smtpSocket = new Socket("hostname", 25);
            os = new DataOutputStream(smtpSocket.getOutputStream());
            is = new DataInputStream(smtpSocket.getInputStream());
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host: hostname");
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to: hostname");
        }
// If everything has been initialized then we want to write some data
// to the socket we have opened a connection to on port 25
    if (smtpSocket != null && os != null && is != null) {
            try {
// The capital string before each colon has a special meaning to SMTP
// you may want to read the SMTP specification, RFC1822/3
        os.writeBytes("HELO\n");
                os.writeBytes("MAIL From: k3is@fundy.csd.unbsj.ca\n");
                os.writeBytes("RCPT To: k3is@fundy.csd.unbsj.ca\n");
                os.writeBytes("DATA\n");
                os.writeBytes("From: k3is@fundy.csd.unbsj.ca\n");
                os.writeBytes("Subject: testing\n");
                os.writeBytes("Hi there\n"); // message body
                os.writeBytes("\n.\n");
        os.writeBytes("QUIT");
// keep on reading from/to the socket till we receive the "Ok" from SMTP,
// once we received that then we want to break.
                String responseLine;
                while ((responseLine = is.readLine()) != null) {
                    System.out.println("Server: " + responseLine);
                    if (responseLine.indexOf("Ok") != -1) {
                      break;
                    }
                }
// clean up:
// close the output stream
// close the input stream
// close the socket
        os.close();
                is.close();
                smtpSocket.close();
            } catch (UnknownHostException e) {
                System.err.println("Trying to connect to unknown host: " + e);
            } catch (IOException e) {
                System.err.println("IOException:  " + e);
            }
        }
    }
}
```

When programming a client, you must follow these four steps:

- Open a socket.
- Open an input and output stream to the socket.
- Read from and write to the socket according to the server's protocol.
- Clean up.

These steps are pretty much the same for all clients. The only step that varies is step three, since it depends on the server you are talking to.

2. Echo server

Now let's write a server. This server is very similar to the echo server running on port 7. Basically, the echo server receives text from the client and then sends that exact text back to the client. This is just about the simplest server you can write. Note that this server handles only one client. Try to modify it to handle multiple clients using threads.

```
import java.io.*;
import java.net.*;
public class echo3 {
    public static void main(String args[]) {
// declaration section:
// declare a server socket and a client socket for the server
// declare an input and an output stream
        ServerSocket echoServer = null;
        String line;
        DataInputStream is;
        PrintStream os;
        Socket clientSocket = null;
// Try to open a server socket on port 9999
// Note that we can't choose a port less than 1023 if we are not
// privileged users (root)
        try {
           echoServer = new ServerSocket(9999);
        }
        catch (IOException e) {
           System.out.println(e);
        }
// Create a socket object from the ServerSocket to listen and accept
// connections.
// Open input and output streams
    try {
           clientSocket = echoServer.accept();
           is = new DataInputStream(clientSocket.getInputStream());
           os = new PrintStream(clientSocket.getOutputStream());
// As long as we receive data, echo that data back to the client.
           while (true) {
             line = is.readLine();
             os.println(line);
           }
        }
    catch (IOException e) {
           System.out.println(e);
        }
    }
}
```

Conclusion

Programming client/server applications is challenging and fun, and programming this kind of application in Java is easier than doing it in other languages, such as C. Socket programming in Java is seamless.

The java.net package provides a powerful and flexibile infrastructure for network programming, so you are encouraged to refer to that package if you would like to know the classes that are provided.

Sun.* packages have some good classes for networking, however you are not encouraged to use those classes at the moment because they may change in the next release. Also, some of the classes are not portable across all platforms.

About the author

Qusay H. Mahmoud is a graduate student in computer science at the University of New Brunswick, Saint John campus, Canada. He is currently working on his master's thesis as well as teaching a computer science course at the University. His thesis concentrates on the Web and Java, and the results of his thesis will be available online as soon as he is finished. He has been working with Java since it was released to the public.