Java Programming: Sockets in Java

Manuel Oriol

May 10, 2007

1 Introduction

Network programming is probably one of the features that is most used in the current world. As soon as people want to send or receive data over a network in a program, you need to use sockets. Java is a language born after the advent of the Internet and hence has very good integration of sockets in the standard API. Sockets-related classes are located in the `java.net` package. In this chapter, we explain how sockets work and show examples that make use of sockets. Section 2 gives some background on networking. Section 3 shows how to use and build TCP sockets. Section 4 shows how to use and build UDP sockets.

2 Networking Background

The traditional approach of networking is the client-server approach. In this model there is one server talks to arbitrarily many clients to answer their requests.

There are two fundamental ways to communicate over a network:

Connected Mode: The connected mode is the most natural way of handling network communications. It correspond closely to how people are used to communicate say over a phone line. In this mode the two parties share a (virtual) connection that they send data over. The connection first needs to be opened, then data can be sent and eventually the connection is closed. Data sent in such a way has a low chance of getting lost and will always arrive in the order it was sent. Both the traditional telephone network and the TCP for the Internet work in this way.

Disconnected Mode: The disconnected mode is a way of communication in which no connection needs to be established between the parties communicating. In this mode information is broadcast. Since there is no notion of connection it cannot be detected if information has not been picked up by a receiver or information might arrive out of order. Radio and television as well as the UDP for the Internet work in this mode.

Traditionally, the code required to make use of sockets has been cumbersome to write and was quite low level. The following gives an example of how a socket is created and bound in C[1]:

[1]Code borrowed from http://www.cs.rpi.edu/courses/sysprog/sockets/sock.html

```
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;

    char buffer[256];
    if (argc < 3) {
       fprintf(stderr,"usage %s hostname port\n", argv[0]);
       exit(0);
    }
    portno = atoi(argv[2]);
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");
    server = gethostbyname(argv[1]);
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(0);
    }
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,
         (char *)&serv_addr.sin_addr.s_addr,
         server->h_length);
    serv_addr.sin_port = htons(portno);
    if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0)
        error("ERROR connecting");
    printf("Please enter the message: ");
    bzero(buffer,256);
```

Such code would be reused by copy, paste and modifying it slightly whenever needed. The equivalent code in Java, which is much more compact and high level is shown in Figure 1.

```
try {
   Socket s = new Socket(args[1],Integer.parseInt(args[2]));
}catch (Exception e){
   System.out.println(e);
   System.exit(0);
}
```

Figure 1: Opening a TCP socket in Java

Ports are a means to distinguish various services on a machine. Ports are divided into the following groups:

ports 0-1023: system ports (admin rights on Unix) or *well-known ports.*

ports 1024-49151: *registered ports* can be used explicitly.

ports 49152-65535: *dynamic ports* or *private ports.*

In the following sections we show how to use sockets in Java in more detail.

3 TCP

As shown in the example, this type of sockets is very simple to build. It relies on an API that is as complete as possible and still as simple to use as possible.

3.1 TCP client sockets

The `Socket` (see part of the APIs in Table 1) is the default representative of the implementations for sockets. In the current version of the JDK, `Socket` should only be used to build conected-mode sockets.

Table 1: Socket methods

java.net.Socket	
Socket()	Creates an unconnected socket.
Socket(InetAddress address, int port)	Opens a socket with the given InetAddress.
Socket(InetAddress address, int port, InetAddress localAddr, int localPort)	Creates a socket and specifies the local port.
protected Socket(SocketImpl impl)	Opens a socket and provide a different implementation.
Socket(String host, int port)	Opens a socket on the host with a server listening on port.
Socket(String host, int port, InetAddress localAddr, int localPort)	Creates a socket and specifies the local port.
InputStream getInputStream()	Returns an InputStream on the socket.
int getLocalPort()	Returns the local port.
OutputStream getOutputStream()	Returns an OutputStream on this socket.
int getSoTimeout()	Gets the timeout fot the socket.
void setSoTimeout(int timeout)	Sets the socket timeout to a value in ms.
String toString()	Returns a string representation of this socket.

Thus opening a TCP socket can be done in several ways, the simplest way is to bind it at creation time as shown in figure 1.

3.2 Example of Use

As a small example, let consider a program that connects on a port and transmits to it characters read on the keyboard. It actually is a telnet-like client. Please note the use of eexceptions and threads.

```
import java.io.*;
import java.net.*;
public class SocketInteractor extends Thread{
   InputStream is;
```

```
/**
 * Creates an instance with the input stream to
 * redirect to the keyboard
 */
public SocketInteractor(InputStream is){
  this.is=is;
}
/**
 * Creates a new Thread and redirect a stream
 * on the keyboard
 */
public void run(){
  try{
   int a;
   // reads from the socket and prints on the terminal
   // as long as the socket is open.
   while(true){
      a=is.read();
      if (a==-1) throw new Exception("Socket closed.");
      System.out.write(a);
   }
  } catch (Exception E){
     System.out.println("socket closed.");
     System.exit(0);
   }
}
/**
 * Prints the usage and exits.
 */
public static void usage(){
   System.out.println("Usage: java SocketInteractor host port_number");
   System.out.println("connects to a socket and
    receive/send information through it");
   System.exit(0);
}
public static void main(String[] args) {
   OutputStream out=null;
   try{
      // checks the arguments
      if (args.length!=2)
       throw new Exception("Bad number of arguments.");
      // creates the socket
      Socket s=new Socket(args[0],Integer.parseInt(args[1]));
      out= s.getOutputStream();
      // starts the new thread
      (new SocketInteractor(s.getInputStream())).start();
   } catch (Exception E){
      usage();
   }
   try{
      // reads on the terminal, outputs on the socket
      while(true){
         out.write(System.in.read());
      }
      } catch (Exception E){
        System.out.println("socket closed.");
        System.exit(0);
      }
   }
}
```

3.3 SSL sockets

Using secure sockets in Java is not much more difficult than to use regular sockets. The class `SSLSocket` can be used as the class `Socket`. Due to the per-country basis for restriction on cryptography secure sockets are not included in the SDK but abstract classes and the infrastructure.

3.4 Server Sockets

The class `ServerSocket` is meant to be used to build server programs on the TCP protocol.

Table 2: ServerSocket methods

java.net.ServerSocket	
ServerSocket()	Creates a server socket.
ServerSocket(int port)	Opens a server socket on a port.
ServerSocket(int port, int backlog)	Opens a server socket (`backlog` simultaneous applications)
Socket accept()	Accept a new connection. Returns the socket.
void close()	Closes the socket.
int getSoTimeout()	Gets the socket's timeout
void setSoTimeout(int timeout)	Sets the socket's timeout

As an example of use, the following code is meant to be used as a server program. It is built by replacing the text of the main method from the previous complete example by:

```
  public static void main(String[] args) {
      OutputStream out=null;
      ServerSocket servs=null;
      try{
         // checks the arguments
         if (args.length!=1)
          throw new Exception("Bad number of arguments.");

         // creates the socket
         servs=new ServerSocket(Integer.parseInt(args[0]));
         Socket s=servs.accept();
         System.out.println("Connection accepted from "+
          s.getRemoteSocketAddress());
         servs.close();
         out= s.getOutputStream();

         // starts the new thread
         (new SocketInteractor(s.getInputStream())).start();
      } catch (Exception E){
         usage();
      }

      try{
         // reads on the terminal, outputs on the socket
         while(true){
            out.write(System.in.read());
         }
      } catch (Exception E){
        System.out.println("socket closed.");
        System.exit(0);
      }

   }
```

4 Datagram sockets

By default, UDP sockets are made using `DatagramSocket`. The idea behind datagram sockets is that the packets contain the information about

Table 3: Datagram Socket methods

java.net.DatagramSocket	
DatagramSocket()	Creates a datagram sockets and binds it to any free UDP port in the system.
DatagramSocket(int port)	Creates a datagram socket and binds to the `port`.
void receive(DatagramPacket p)	Receives a packet.
void send(DatagramPacket p)	Sends a packet.

4.1 Example of Use

As an example, we try to send a datagram packet to a given socket. The datagram includes a test string. The receiver should look like this:

```
import java.io.*;
import java.net.*;
public class UDPReceiver {
    /**
    * Prints the usage and exits.
    */
   public static void usage(){
        System.out.println("Usage: java UDPReceiver port_number\n
        this program reads a Datagram received through a UDP socket
        bound to specified port.");
        System.exit(0);
    }
    public static void main(String[] args) {
        OutputStream out=null;
        try{
            String text="test";
            byte[] b = new byte[100];
            DatagramPacket dp=new DatagramPacket(b,100);
            // checks the arguments
            if (args.length!=1) throw new Exception("Bad number of arguments.");
            // creates the socket
            DatagramSocket s=new DatagramSocket(Integer.parseInt(args[0]));
            s.receive(dp);
            System.out.println(new String(dp.getData()));
        } catch (Exception E){
            usage();
        }
    }
}
```

The sender could look like this:

```
import java.io.*;
import java.net.*;
public class UDPTest {
     /**
     * Prints the usage and exits.
     */
    public static void usage(){
        System.out.println("Usage: java SocketInteractor local_port host remote_port\n this
        System.exit(0);
    }
    public static void main(String[] args) {
        OutputStream out=null;
        try{
            String text="test";
            // checks the arguments
            if (args.length!=3) throw new Exception("Bad number of arguments.");

            // creates the socket
            DatagramSocket s=new DatagramSocket(Integer.parseInt(args[0]));
            s.connect(InetAddress.getByName(args[1]),Integer.parseInt(args[2]));
            s.send(new DatagramPacket(text.getBytes(),text.length()));

        } catch (Exception E){
            usage();
        }
    }
}
```

5 Exercise

1. Read the APIs for `MulticastSocket` and try to use it. As indicated, it is a `DatagramSocket`.
2. Use the example and code a minimalistic Web server.
3. Use the example and code a minimalisstic FTP client.
4. Try to program a UDP based webserver. What do you think of the approach? What could be the interest?