A Tutorial on Socket Programming in Java

Natarajan Meghanathan
Professor of Computer Science
Jackson State University
Jackson, MS 39217, USA
Phone: 1-601-979-3661; Fax: 1-601-979-2478
E-mail: natarajan.meghanathan@jsums.edu

Abstract

We present a tutorial on socket programming in Java. This tutorial illustrates several examples on the two types of socket APIs: connectionless datagram sockets and connection-oriented stream-mode sockets. With datagram sockets, communication occurs in the form of discrete messages sent from the sender to receiver; whereas with stream-mode sockets, data is transferred using the concept of a continuous data stream flowing from a source to a destination. For both kinds of sockets, we illustrate examples for simplex (one-way) and duplex (bi-directional) communication. We also explain in detail the difference between a concurrent server and iterative server and show example programs on how to develop the two types of servers, along with an example. The last section of the tutorial describes the Multicast Socket API in Java and illustrates examples for multicast communication. We conclude by providing descriptions for several practice programming exercises that could be attempted at after reading the tutorial.

1. Introduction

Video Link: http://www.youtube.com/watch?v=O9hWfruFtR4

Interprocess communication (IPC) is the backbone of distributed computing. Processes are runtime representations of a program. IPC refers to the ability for separate, independent processes to communicate among themselves to collaborate on a task. The following figure illustrates basic IPC:

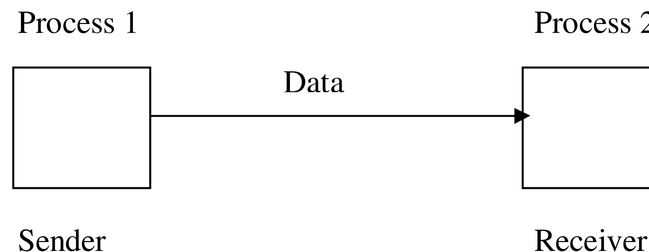

Figure 1: Interprocess Communication

Two or more processes engage in a protocol – a set of rules to be observed by the participants for data communication. A process can be a sender at some instant of the communication and can be a receiver of the data at another instant of the communication. When data is sent from one process to another single process, the communication is said to be unicast. When data is sent from one process to more than one process at the same time, the communication is said to be multicast. Note that multicast is not multiple unicasts.

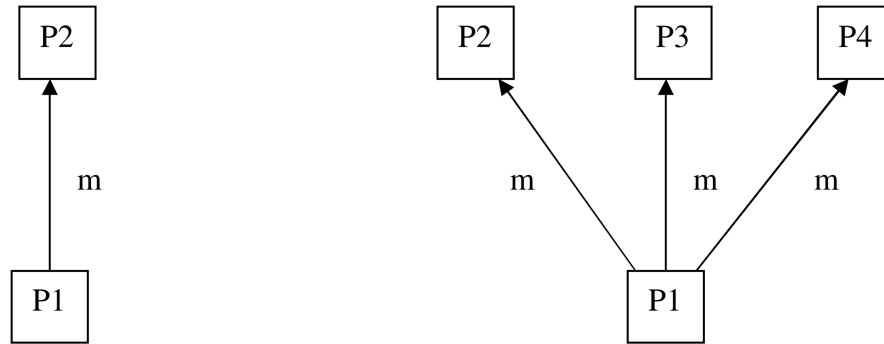

Figure 2: Unicast Vs. Multicast

Most of the operating systems (OS) like UNIX and Windows provide facilities for IPC. The system-level IPC facilities include message queues, semaphores and shared memory. It is possible to directly develop network software using these system-level facilities. Examples are network device drivers and system evaluation programs. On the other hand, the complexities of the applications require the use of some form of abstraction to spare the programmer of the system-level details. An IPC application programming interface (API) abstracts the details and intricacies of the system-level facilities and allows the programmer to concentrate on the application logic.

The Socket API is a low-level programming facility for implementing IPC. The upper-layer facilities are built on top of the operations provided by the Socket API. The Socket API was originally provided as part of the Berkeley UNIX OS, but has been later ported to all operating systems including Sun Solaris and Windows systems. The Socket API provides a programming construct called a “socket”. A process wishing to communicate with another process must create an instance or instantiate a socket. Two processes wishing to communicate can instantiate sockets and then issue operations provided by the API to send and receive data. Note that in network parlance, a packet is the unit of data transmitted over the network. Each packet contains the data (payload) and some control information (header) that includes the destination address.

A socket is uniquely identified by the IP address of the machine and the port number at which the socket is opened (i.e. bound to). Port numbers are allocated 16 bits in the packet headers and thus can be at most 66535. Well-known processes like FTP, HTTP and etc., have their sockets opened on dedicated port numbers (less than or equal to 1024). Hence, sockets corresponding to user-defined processes have to be run on port numbers greater than 1024.

In this chapter, we will discuss two types of sockets – “connectionless” and “connection-oriented” for unicast communication, multicast sockets and several programming examples to illustrate different types of communication using these sockets. All of the programming examples are illustrated in Java.

2. Types of Sockets

The User Datagram Protocol (UDP) transports packets in a connectionless manner [1]. In a connectionless communication, each data packet (also called datagram) is addressed and routed individually and may arrive at the receiver in any order. For example, if process 1 on host A sends datagrams m1 and m2 successively to process 2 on host B, the datagrams may be

transported on the network through different routes and may arrive at the destination in any of the two orders: m1, m2 or m2, m1.

The Transmission Control Protocol (TCP) is connection-oriented and transports a stream of data over a logical connection established between the sender and the receiver [1]. As a result, data sent from a sender to a receiver is guaranteed to be received in the order they were sent. In the above example, messages m1 and m2 are delivered to process 2 on host B in the same order they were sent from process 1 on host A.

A socket programming construct can use either UDP or TCP transport protocols. Sockets that use UDP for transport of packets are called "datagram" sockets and sockets that use TCP for transport are called "stream" sockets.

3. The Connectionless Datagram Socket

In Java, two classes are provided for the datagram socket API: (a) The DatagramSocket class for the sockets (b) The DatagramPacket class for the packets exchanged. A process wishing to send or receive data using the datagram socket API must instantiate a DatagramSocket object, which is bound to a UDP port of the machine and local to the process.

To send a datagram to another process, the sender process must instantiate a DatagramPacket object that carries the following information: (1) a reference to a byte array that contains the payload data and (2) the destination address (the host ID and port number to which the receiver process' DatagramSocket object is bound).

At the receiving process, a DatagramSocket object must be instantiated and bound to a local port – this port should correspond to the port number carried in the datagram packet of the sender. To receive datagrams sent to the socket, the receiving process must instantiate a DatagramPacket object that references a byte array and call the receive method of the DatagramSocket object, specifying as argument, a reference to the DatagramPacket object. The program flow in the sender and receiver process is illustrated in Figure 3 and the key methods used for communication using connectionless sockets are summarized in Table 1.

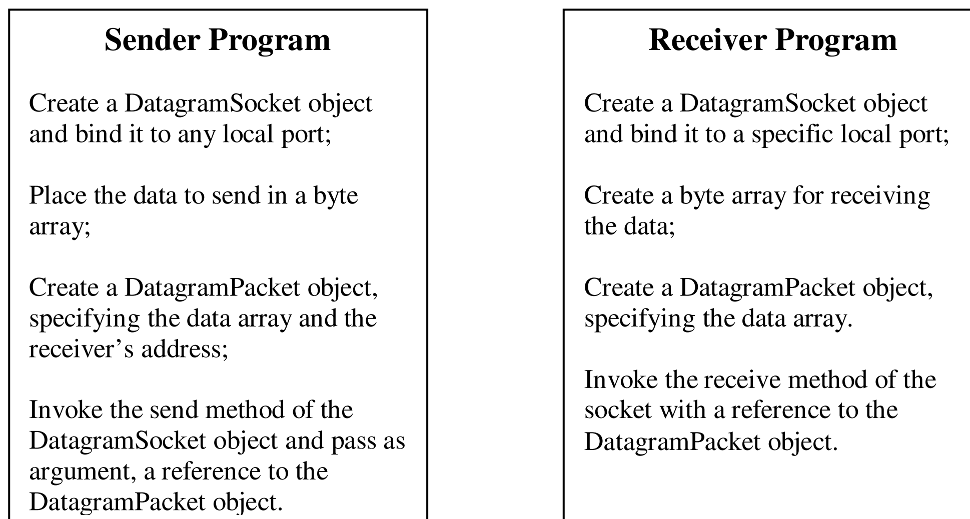

Figure 3: Program flow in the sender and receiver process (adapted from [2])

Table 1: Key Commonly Used Methods of the DatagramSocket API (adapted from [3])

No.	Constructor/ Method	Description
DatagramSocket class		
1	DatagramSocket( )	Constructs an object of class DatagramSocket and binds the object to any available port on the local host machine
2	DatagramSocket(int port)	Constructs an object of class DatagramSocket and binds it to the specified port on the local host machine
3	DatagramSocket (int port, InetAddress addr)	Constructs an object of class DatagramSocket and binds it to the specified local address and port
4	void close( )	Closes the datagram socket
5	void connect(InetAddress address, int port)	Connects the datagram socket to the specified remote address and port number on the machine with that address
6	InetAddress getLocalAddress( )	Returns the local InetAddress to which the socket is connected.
7	int getLocalPort( )	Returns the port number on the local host to which the datagram socket is bound
8	InetAddress getInetAddress( )	Returns the IP address to which the datagram socket is connected to at the remote side.
9	int getPort( )	Returns the port number at the remote side of the socket
10	void receive(DatagramPacket packet)	Receives a datagram packet object from this socket
11	void send(DatagramPacket packet)	Sends a datagram packet object from this socket
12	void setSoTimeout(int timeout)	Set the timeout value for the socket, in milliseconds
DatagramPacket class		
13	DatagramPacket(byte[ ] buf, int length, InetAddress, int port)	Constructs a datagram packet object with the contents stored in a byte array, buf, of specified length to a machine with the specified IP address and port number
14	InetAddress getAddress( )	Returns the IP address of the machine at the remote side to which the datagram is being sent or from which the datagram was received
15	byte [ ] getData( )	Returns the data buffer stored in the packet as a byte array
16	int getLength( )	Returns the length of the data buffer in the datagram packet sent or received
17	int getPort( )	Returns the port number to which the datagram socket is bound to which the datagram is being sent or from which the datagram is received
18	void setData(byte [ ])	Sets the data buffer for the datagram packet
19	void setAddress(InetAddress iaddr)	Sets the datagram packet with the IP address of the remote machine to which the packet is being sent
20	void setPort(int port)	Sets the datagram packet with the port number of the datagram socket at the remote host to which the packet is sent

With connectionless sockets, a DatagramSocket object bound to a process can be used to send datagrams to different destinations. Also, multiple processes can simultaneously send datagrams to the same socket bound to a receiving process. In such a situation, the order of the arrival of the datagrams may not be consistent with the order they were sent from the different processes. Note that in connection-oriented or connectionless Socket APIs, the send operations are non-blocking and the receive operations are blocking. A process continues its execution after the issuance of a *send* method call. On the other hand, once a process calls the *receive* method on a socket, the process is suspended until a datagram is received. To avoid indefinite blocking, the *setSoTimeout* method can be called on the DatagramSocket object.

We now present several sample programs to illustrate the use of the DatagramSocket and DatagramPacket API. Note that in all these exercises, the receiver programs should be started first before starting the sender program. This is analogous to the fact that in any conversation, a receiver should be tuned and willing to hear and receive the information spoken (sent) by the sender. If the receiver is not turned on, then whatever the message was sent will be dropped at the receiving side. The following code segments illustrate the code to send to a datagram packet from one host IP address and port number and receive the same packet at another IP address and port number. Though the sender and receiver programs are normally run at two different hosts, sometimes one can test the correctness of their code by running the two programs on the same host using '*localhost*' as the name of the host at remote side. This is the approach we use in this book chapter. For all socket programs, the package java.net should be imported; and very often we need to also import the java.io package to do any input/output with the sockets. Of course, for any file access, we also need to import the java.io package. Also, since many of the methods (for both the Connectionless and Stream-mode API) could raise exceptions, it is recommended to put the entire code inside a *try-catch* block.

3.1 *Example Program to Send and Receive a Message using Connectionless Sockets*

Video Link: http://www.youtube.com/watch?v=fQrANY-REZQ

The datagram receiver (datagramReceiver.java) program illustrated below can receive a datagram packet of size at most 40 bytes. As explained before, the receive( ) method call on the

---

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

class datagramReceiver{
  public static void main(String[ ] args){
    try{
      int MAX_LEN = 40;
      int localPortNum = Integer.parseInt(args[0]);
      DatagramSocket mySocket  = new DatagramSocket(localPortNum);
      byte[] buffer = new byte[MAX_LEN];
      DatagramPacket packet = new DatagramPacket(buffer, MAX_LEN);
      mySocket.receive(packet);
      String message = new String(buffer);
      System.out.println(message);
      mySocket.close( );
     }
    catch(Exception e){e.printStackTrace( );}
   }
}
```

Figure 4: Program to Receive a Single Datagram Packet

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

```
class datagramSender{
  public static void main(String[ ] args){
    try{
        InetAddress receiverHost = InetAddress.getByName(args[0]);
        int receiverPort = Integer.parseInt(args[1]);
        String message = args[2];
        DatagramSocket mySocket = new DatagramSocket( );
        byte[] buffer = message.getBytes( );
        DatagramPacket packet = new DatagramPacket(buffer, buffer.length, receiverHost,
                                                                           receiverPort);
        mySocket.send(packet);
        mySocket.close( );
     }
    catch(Exception e){ e.printStackTrace( ); }
   }
}
```

Figure 5: Program to Send a Single Datagram Packet

DatagramSocket is a binding call. Once a datagram packet arrives at the host at the specified local port number at which the socket is opened, the receiver program will proceed further – extract the bytes stored in the datagram packet and prints the contents as a String. The local port number at which the receiver should open its datagram socket is passed as an input command-line parameter and this port number should also be known to the sender so that the message can be sent to the same port number. The datagram sender (datagramSender.java) program creates a DatagramPacket object and sets its destination IP address to the IP address of the remote host, the port number at which the message is expected to receive and the actual message. The sender program has nothing much to do after sending the message and hence the socket is closed. Similarly, the socket at the receiver side is also closed after receiving and printing the message. Figure 6 is a screenshot of the execution and output obtained for the code segments illustrated in Figures 4 and 5. Note that the datagramReceiver program should be started first. The maximum size of the message that could be received is 40 bytes (characters) as set by the datagramReceiver.

```
C:\res\tutorial\sockets\sender>javac datagramSender.java

C:\res\tutorial\sockets\sender>java datagramSender localhost 2389 "Hi, How are y
ou?"

C:\res\tutorial\sockets\sender>
```

```
C:\res\tutorial\sockets\receiver>javac datagramReceiver.java

C:\res\tutorial\sockets\receiver>java datagramReceiver 2389
Hi, How are you?

C:\res\tutorial\sockets\receiver>
```

Figure 6: Screenshots of the Execution of datagramSender.java and datagramReceiver.java

3.2 *Example Program to Send and Receive a Message in both Directions (Duplex Communication) using Connectionless Sockets*

Video Link: http://www.youtube.com/watch?v=09LRS5fBFII

The program illustrated in this example is an extension of the program in Section 3.1. Here, we describe two programs – datagramSenderReceiver.java (refer Figure 7) and datagramReceiverSender.java (refer Figure 8). The sender-receiver program will first send a message and then wait for a response for the message. Accordingly, the first half of the sender-receiver program would be to send a message and the second half of the program would be to receive a response. Note that to get the response, the sender-receiver program should invoke the receive( ) method on the same DatagramSocket object and port number that were used to send the message. The receiver-sender program will have to first receive the message and then respond to the sender of the message. It extracts the sender information from the Datagram Packet object received and uses the sender IP address and port number retrieved from the Datagram Packet received as the destination IP address and port number for the response Datagram Packet sent. This is analogous to replying to a mail or an email using the sender information in the mail (i.e., reply to the same address from which the message was sent). The above logic could be used to develop chatting programs using connectionless sockets. The maximum size of the messages that could be sent and received is 60 bytes in this example.

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

class datagramSenderReceiver{
  public static void main(String[ ] args){
    try{
      InetAddress receiverHost = InetAddress.getByName(args[0]);
      int receiverPort = Integer.parseInt(args[1]);
      String message = args[2];

      DatagramSocket mySocket = new DatagramSocket( );
      byte[] sendBuffer = message.getBytes( );
      DatagramPacket packet = new DatagramPacket(sendBuffer, sendBuffer.length,
                                                   receiverHost, receiverPort);
      mySocket.send(packet);

      // to receive a message

      int MESSAGE_LEN = 60;
      byte[ ] recvBuffer = new byte[MESSAGE_LEN];

      DatagramPacket datagram = new DatagramPacket(recvBuffer, MESSAGE_LEN);
      mySocket.receive(datagram);
      String recvdString = new String(recvBuffer);
      System.out.println("\n"+recvdString);

      mySocket.close( );
```

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

Figure 7: Code for the Datagram Sender and Receiver (Sends First, Receives Next) Program

---

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

class datagramReceiverSender{
  public static void main(String[ ] args){
    try{
      int MAX_LEN = 60;
      int localPortNum = Integer.parseInt(args[0]);

      DatagramSocket mySocket  = new DatagramSocket(Integer.parseInt(localPortNum);
      byte[ ] recvBuffer = new byte[MAX_LEN];
      DatagramPacket packet = new DatagramPacket(recvBuffer, MAX_LEN);
      mySocket.receive(packet);
      String message = new String(recvBuffer);

      System.out.println("\n"+message);

      // to reply back to sender
      InetAddress senderAddress = packet.getAddress( );
      int senderPort = packet.getPort( );

     // String messageToSend = args[1];
        Scanner inputScanner = new Scanner(System.in);
        String messageToSend = inputScanner.nextLine();
      byte[ ] sendBuffer = messageToSend.getBytes();

      DatagramPacket datagram = new DatagramPacket(sendBuffer, sendBuffer.length,
                                                   senderAddress, senderPort);
      mySocket.send(datagram);
      mySocket.close( );
    }
   catch(Exception e){e.printStackTrace( );}
  }
}
```

Figure 8: Code for the Datagram Receiver and Sender (Receives First, Sends Next) Program

```
C:\res\tutorial\sockets\sender>javac datagramSenderReceiver.java

C:\res\tutorial\sockets\sender>java datagramSenderReceiver localhost 4567 "Tryin
g to Connect"

Welcome to the world of Java Socket Programming

C:\res\tutorial\sockets\sender>_
```

```
C:\res\tutorial\sockets\receiver>javac datagramReceiverSender.java

C:\res\tutorial\sockets\receiver>java datagramReceiverSender 4567 "Welcome to th
e world of Java Socket Programming"

Trying to Connect

C:\res\tutorial\sockets\receiver>
```

Figure 9: Screenshots of the Execution of datagramSenderReceiver.java and datagramReceiverSender.java

In the above example, the datagramReceiverSender.java program is waiting at a local port number of 4567 to greet with a message “Welcome to the world of Java Socket Programming” for an incoming connection request message from any host. When the datagramSenderReceiver.java program attempts to connect at port number 4567 with a “Trying to Connect” request, it gets the welcome message as the response from the other end.

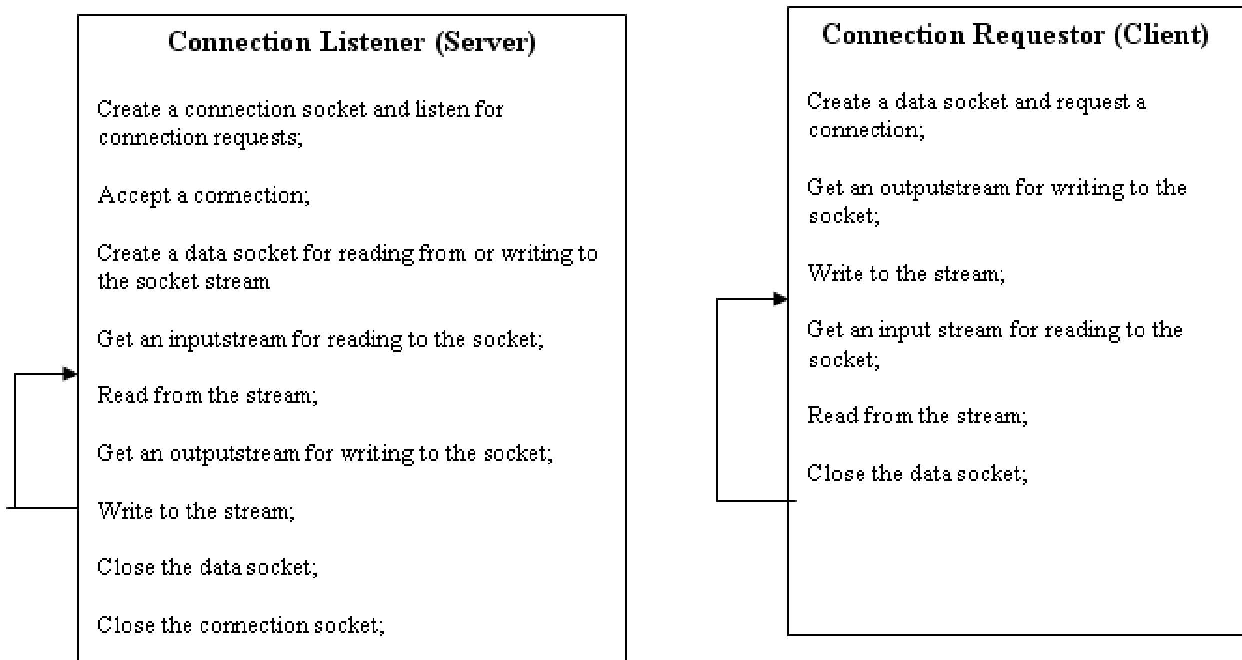

Figure 10: Program Flow in the Connection Listener (Server) and Connection Requester (Client) Processes – adapted from [2]

4 The Connection-Oriented (Stream-Mode) Socket

The Connection-Oriented Sockets are based on the stream-mode I/O model of the UNIX Operating System – data is transferred using the concept of a continuous data stream flowing

from a source to a destination. Data is inserted into the stream by a sender process usually called the server and is extracted from the stream by the receiver process usually called the client. The server process establishes a connection socket and then listens for connection requests from other processes. Connection requests are accepted one at a time. When the connection request is accepted, a data socket is created using which the server process can write or read from/to the data stream. When the communication session between the two processes is over, the data socket is closed and the server process is free to accept another connection request. Note that the server process is blocked when it is listening or waiting for incoming connection requests. This problem could be alleviated by spawning threads, one for each incoming client connection request and a thread would then individually handle the particular client.

Table 2: Key Commonly Used Methods of the Stream-Mode Socket API (adapted from [3])

No.	Constructor/ Method	Description
		ServerSocket class
1	ServerSocket(int port)	Constructs an object of class ServerSocket and binds the object to the specified port – to which all clients attempt to connect
2	accept( )	This is a blocking method call – the server listens (waits) for any incoming client connection request and cannot proceed further unless contacted by a client. When a client contacts, the method is unblocked and returns a Socket object to the server program to communicate with the client.
3	close( )	Closes the ServerSocket object
4	void setSoTimeout(int timeout)	The ServerSocket object is set to listen for an incoming client request, under a particular invocation of the accept ( ) method on the object, for at most the milliseconds specified in “timeout”. When the timeout expires, a java.net.SocketTimeoutException is raised. The timeout value must be > 0; a timeout value of 0 indicates infinite timeout.
		Socket class
5	Socket(InetAddress host, int port)	Creates a stream socket and connects it to the specified port number at the specified IP address
6	InetAddress getInetAddress( )	Returns the IP address at the remote side of the socket
7	InetAddress getLocalAddress( )	Returns the IP address of the local machine to which this socket is bound
8	int getPort( )	Returns the remote port number to which this socket is connected
9	int getLocalPort( )	Returns the local port number to which this socket is bound
10	InputStream getInputStream( )	Returns an input stream for this socket to read data sent from the other end of the connection
11	OutputStream getOutputStream( )	Returns an output stream for this socket to send data to the other end of the connection
12	close( )	Closes this socket
13	void setSoTimeout(int timeout)	Sets a timeout value to block on any read( ) call on the InputStream associated with this socket object. When the timeout expires, a java.net.SocketTimeoutException is raised. The timeout value must be > 0; a timeout value of 0 indicates infinite timeout.

In the next few sections, we illustrate examples to illustrate the use of Stream-mode sockets to send and receive messages. These sockets are typically used for connection-oriented communication during which a sequence of bytes (not discrete messages) need to be transferred in one or both directions. The connection listener program (Server program) should be started

first and ServerSocket object should be the first to be created to accept an incoming client connection request. The connection requester program (Client program) should be then started.

4.1 *Example Program to Send a Message from Server to Client when Contacted*

Video Link: http://www.youtube.com/watch?v=RKtNAp_N4B4

Here, the connectionServer.java program creates a ServerSocket object bound to port 3456 and waits for an incoming client connection request. When a client contacts the server program, the accept( ) method is unblocked and returns a Socket object for the server to communicate with the particular client that contacted. The server program then creates a PrintStream object through the output stream extracted from this socket and uses it to send a welcome message to the contacting client. The client program runs as follows: The client creates a Socket object to connect to the server running at the specified IP address or hostname and at the port number 3456. The client creates a BufferedReader object through the input stream extracted from this socket and waits for an incoming line of message from the other end. The readLine( ) method of the BufferedReader object blocks the client from proceeding further unless a line of message is received. The purpose of the flush( ) method of the PrintStream class is to write any buffered output bytes to the underlying output stream and then flush that stream to send out the bytes. Note that the server program in our example sends a welcome message to an incoming client request and then stops.

```
import java.net.*;
import java.io.*;
class connectionServer{
   public static void main(String[ ] args){
   try{
     String message = args[0];
     int serverPortNumber = Integer.parseInt(args[1]);
     ServerSocket connectionSocket = new ServerSocket(serverPortNumber);
     Socket dataSocket = connectionSocket.accept();
     PrintStream socketOutput = new PrintStream(dataSocket.getOutputStream());
     socketOutput.println(message);
     System.out.println("sent response to client…");
     socketOutput.flush( );
     dataSocket.close( );
     connectionSocket.close( );
    }
    catch(Exception e){
            e.printStackTrace( );
    }
   }
 }
```

Figure 11: Code for the Connection Server Program to Respond to a Connecting Client

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

```
import java.io.*;

class connectionClient{
  public static void main(String[ ] args){
    try{
       InetAddress acceptorHost = InetAddress.getByName(args[0]);
       int serverPortNum = Integer.parseInt(args[1]);
       Socket clientSocket = new Socket(acceptorHost, serverPortNum);
       BufferedReader br = new BufferedReader(new
                                         InputStreamReader(clientSocket.getInputStream( )));
       System.out.println(br.readLine( ));
       clientSocket.close();
      }
     catch(Exception e){
         e.printStackTrace( );
        }
    }
  }
```

Figure 12: Code for the Connection Client Program to Connect to a Server Program

```
C:\res\tutorial\sockets\server>javac connectionServer.java

C:\res\tutorial\sockets\server>java connectionServer "Welcome to Java Socket Pro
gramming" 3456
sent response to client...

C:\res\tutorial\sockets\server>
```

```
C:\res\tutorial\sockets\client>javac connectionClient.java

C:\res\tutorial\sockets\client>java connectionClient localhost 3456
Welcome to Java Socket Programming

C:\res\tutorial\sockets\client>
```

Figure 13: Screenshots of the Execution of the Programs to Send a Message to a Client from a Server when Contacted

---

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

class connectionClient{
  public static void main(String[ ] args){
    try{
       InetAddress acceptorHost = InetAddress.getByName(args[0]);
       int serverPortNum = Integer.parseInt(args[1]);
       Socket clientSocket = new Socket(acceptorHost, serverPortNum);
       BufferedReader br = new BufferedReader(new
```

```
                                        InputStreamReader(clientSocket.getInputStream( )));
    System.out.println(br.readLine( ));
    PrintStream ps = new PrintStream(clientSocket.getOutputStream( ));
    ps.println("received your message.. Thanks");
    ps.flush( );
    clientSocket.close( );
   }
  catch(Exception e){e.printStackTrace( );}
 }
}
```

Figure 14: Code for the Client Program for Duplex Connection Communication

4.2 *Example Program to Illustrate Duplex Nature of Stream-mode Socket Connections*

Video Link: http://www.youtube.com/watch?v=gM3zHe6QYho

This program is an extension of the program illustrated in Section 4.1. Here, we illustrate that the communication using stream-mode sockets could occur in both directions. When the client receives a response for its connection request from the server, the client responds back with an acknowledgement that the server response was received. The server waits to receive such a response from the client. All of these communication occur using the Socket object returned by the accept( ) method call on the ServerSocket object at the server side and using the Socket object originally created by the client program to connect to the server. It is always recommended to close the Socket objects at both sides after their use. Even though server programs typically run without being stopped, our example server program terminates after one duplex communication with the client. Before the server program terminates, the ServerSocket object should be closed.

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

class connectionServer{
 public static void main(String[ ] args){
  try{
    String message = args[0];
    int serverPortNum = Integer.parseInt(args[1]);
    ServerSocket connectionSocket = new ServerSocket(serverPortNum);
    Socket dataSocket = connectionSocket.accept( );
    PrintStream socketOutput = new PrintStream(dataSocket.getOutputStream( ));
    socketOutput.println(message);
    socketOutput.flush( );
    BufferedReader br = new BufferedReader(new
                                        InputStreamReader(dataSocket.getInputStream( )));
    System.out.println(br.readLine( ));
    dataSocket.close();
    connectionSocket.close();
```

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

Figure 15: Code for the Server Program for Duplex Connection Communication

```
C:\res\tutorial\sockets\server>javac connectionServerDuplex.java

C:\res\tutorial\sockets\server>java connectionServer "Welcome to Java Socket Pro
gramming" 4589
received your message.. Thanks

C:\res\tutorial\sockets\server>
```

```
C:\res\tutorial\sockets\client>javac connectionClientDuplex.java

C:\res\tutorial\sockets\client>java connectionClient localhost 4589
Welcome to Java Socket Programming

C:\res\tutorial\sockets\client>
```

Figure 16: Screenshots of the Execution of the Programs for Duplex Communication

4.3 *Example Program to Illustrate the Server can Run in Infinite Loop handling Multiple Client Requests, one at a time*

Video Link: http://www.youtube.com/watch?v=gFHFnJiolzY

In this example, we illustrate a server program (an iterative server) that can service multiple clients, though, one at a time. The server waits for incoming client requests. When a connection request is received, the accept ( ) method returns a Socket object that will be used to handle all the communication with the client. During this time, if any connection requests from any other client reach the server, these requests have to wait before the server has completed its communication with the current client. To stop a server program that runs in an infinite loop, we press Ctrl+C. This terminates the program as well as closes the ServerSocket object.

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

class connectionServer{
 public static void main(String[ ] args){
  try{
    String message = args[0];
    int serverPortNum = Integer.parseInt(args[1]);
    ServerSocket connectionSocket = new ServerSocket(serverPortNum);

    while (true){
      Socket dataSocket = connectionSocket.accept( );
      PrintStream socketOutput = new PrintStream(dataSocket.getOutputStream( ));
      socketOutput.println(message);
```

```
        socketOutput.flush( );
        dataSocket.close( );
      }
    }
   catch(Exception e){e.printStackTrace( );}
  }
}
```

Figure 17: Code for an Iterative Server Program to Send a Message to each of its Clients

```
C:\res\tutorial\sockets\server>javac connectionServerIterative.java

C:\res\tutorial\sockets\server>java connectionServer "Welcome to the Java World!
!" 3210
```

```
C:\res\tutorial\sockets\client>javac connectionClient.java

C:\res\tutorial\sockets\client>java connectionClient localhost 3210
Welcome to the Java World!!

C:\res\tutorial\sockets\client>java connectionClient localhost 3210
Welcome to the Java World!!

C:\res\tutorial\sockets\client>java connectionClient localhost 3210
Welcome to the Java World!!

C:\res\tutorial\sockets\client>
```

Figure 18: Screenshots of the Execution of an Iterative Server along with its Multiple Clients

Note that the client code need not be modified to communicate with the iterative server. The same client code that was used in Section 4.1 or 4.2 could be used here, depending on the case. In this example (illustrated in Figure 17), the iterative server just responds back with a welcome message and does not wait for the client response. Hence, one would have to use a client program, like the one shown in Section 4.1, to communicate with this iterative server program.

4.4 *Example Program to Send Objects of User-defined Classes using Stream-mode Sockets*

Video Link: http://www.youtube.com/watch?v=iWAL-HgOR4k

In this example, we illustrate how objects of user-defined classes could be sent using stream-mode sockets. An important requirement of classes whose objects needs to be transmitted across sockets is that these classes should implement the *Serializable* interface defined in the java.io. package. Figure 19 shows the code for an Employee class (that has three member variables – ID, Name and Salary), implementing the Serializable interface. An object of the Employee class, with all the member variables set by obtaining inputs from the user, is being sent by the client program (code in Figure 20) to a server program (code in Figure 21) that extracts the object from the socket and prints the values of its member variables. To write an object to a socket, we use the ObjectOutputStream and to extract an object from a socket, we use the ObjectInputStream.

```
import java.io.*;
class EmployeeData implements Serializable{
```

```
  int empID;
  String empName;
  double empSalary;
  void setID(int id){      empID = id;    }
  void setName(String name){      empName = name;     }
  void setSalary(double salary){       empSalary = salary;     }
  int getID( ){      return empID;     }
  String getName( ){      return empName;     }
  double getSalary( ){      return empSalary;     }
}
```

Figure 19: Code for the Employee Class, Implementing the *Serializable* Interface

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

class connectionClient{
  public static void main(String[] args){
    try{
      InetAddress serverHost = InetAddress.getByName(args[0]);
      int serverPortNum = Integer.parseInt(args[1]);
      Socket clientSocket = new Socket(serverHost, serverPortNum);
      EmployeeData empData = new EmployeeData( );
      Scanner input = new Scanner(System.in);
      System.out.print("Enter employee id: ");
      int id = input.nextInt( );
      input.nextLine( );
      System.out.print("Enter employee name: ");
      String name = input.nextLine( );
      System.out.print("Enter employee salary: ");
      double salary = input.nextDouble( );
      empData.setID(id);
      empData.setName(name);
      empData.setSalary(salary);
     ObjectOutputStream oos = new ObjectOutputStream(clientSocket.getOutputStream( ));
     oos.writeObject(empData);
     oos.flush( );
     clientSocket.close( );
    }
   catch(Exception e){e.printStackTrace( );}
  }
}
```

Figure 20: Client Program to Send an Object of User-defined Class across Stream-mode Socket

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

class connectionServer{
  public static void main(String[] args){
    try{
      int serverListenPortNum = Integer.parseInt(args[0]);
      ServerSocket connectionSocket = new ServerSocket(serverListenPortNum);
      Socket dataSocket = connectionSocket.accept( );
      ObjectInputStream ois = new ObjectInputStream(dataSocket.getInputStream( ));
      EmployeeData eData = (EmployeeData) ois.readObject( );
      System.out.println("Employee id : "+eData.getID( ));
      System.out.println("Employee name : "+eData.getName( ));
      System.out.println("Employee salary : "+eData.getSalary( ));
      dataSocket.close( );
      connectionSocket.close( );
     }
   catch(Exception e){e.printStackTrace( );}
  }
}
----------------------------------------------------------------------------------------------------------------
```

Figure 21: Server to Receive an Object of User-defined Class across a Stream-mode Socket

```
C:\res\tutorial\sockets\server>javac connectionServerObject.java

C:\res\tutorial\sockets\server>java connectionServer 8904
Employee id : 234
Employee name : ABC
Employee salary : 12000.0

C:\res\tutorial\sockets\server>
```

```
C:\res\tutorial\sockets\client>javac connectionClientObject.java

C:\res\tutorial\sockets\client>java connectionClient localhost 8904
Enter employee id: 234
Enter employee name: ABC
Enter employee salary: 12000

C:\res\tutorial\sockets\client>
```

Figure 22: Screenshots of the Client-Server Program to Send and Receive Object of User-defined Class across Stream-mode Sockets

4.5 *Example Program to Illustrate Sending and Receiving of Integers across a Stream-mode Socket*

Video Link: http://www.youtube.com/watch?v=rDAK0KDuYPQ

In this example program, we illustrate the sending and receiving of integers across a stream-mode socket. The client program sends two integers using the PrintStream object; the server program receives them, computes and prints their sum.

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

class connectionClient{
  public static void main(String[ ] args){
   try{
     InetAddress serverHost = InetAddress.getByName(args[0]);
     int serverPortNum = Integer.parseInt(args[1]);
     Socket clientSocket = new Socket(serverHost, serverPortNum);
     PrintStream ps = new PrintStream(clientSocket.getOutputStream());
     ps.println(2);
     ps.flush( );
     ps.println(3);
     ps.flush( );
     clientSocket.close( );
     }
catch(Exception e){e.printStackTrace( );}
   }
}
```

Figure 23: Code for a Client Program to Send Two Integers across a Stream-mode Socket

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

class connectionServer{
   public static void main(String[] args){
    try{
       int serverListenPortNum = Integer.parseInt(args[0]);
       ServerSocket connectionSocket = new ServerSocket(serverListenPortNum);
       Socket dataSocket = connectionSocket.accept( );
       BufferedReader br = new BufferedReader(new
                                   InputStreamReader(dataSocket.getInputStream( )));
       int num1 = Integer.parseInt(br.readLine( ));
       int num2 = Integer.parseInt(br.readLine( ));
       System.out.println(num1+" + "+num2+" = "+(num1+num2));
       dataSocket.close( );
       connectionSocket.close( );
      }
    catch(Exception e){e.printStackTrace( );}
   }
}
```

Figure 23: Code for a Server to Receive Integers across a Stream-mode Socket

4.6 *Iterative Server vs. Concurrent Server*

4.6.1 Iterative Server

Video Link: http://www.youtube.com/watch?v=LMRzvTdjR7A

As illustrated in Section 4.3, an iterative server is a server program that handles one client at a time. If one or more client connection requests reach the server while the latter is in communication with a client, these requests have to wait for the existing communication to be completed. The pending client connection requests are handled on a First-in-First-Serve basis. However, such a design is not efficient. Clients may have to sometime wait for excessive amount of time for the requests ahead of theirs in the waiting queue to be processed. When the client requests differ in the amount of time they take to be handled by the server, it would then lead to a situation where a client with a lower execution time for its request at the server may have to wait for the requests (ahead in the queue) that have a relatively longer execution time to be completed first. The code in Figure 25 illustrates one such example of an iterative server that has to add integers from 1 to a "count" value (i.e., 1+2+…+count) sent by a client program (Figure 24) and return the sum of these integers to the requesting client. In order to simulate the effect of time-consuming client requests, we make the server program to sleep for 200 milliseconds after performing each addition. As iterative servers are single-threaded programs, the whole program sleeps when we invoke the sleep( ) static function of the Thread class. The execution screenshots illustrated in Figure 26 show that a client with a request to add integers from 1 to 5 will have to wait for 19500 milliseconds (i.e., 19.5 seconds) as the client's request reached the server while the latter was processing a request from another client to add integers from 1 to 100, which takes 20031 milliseconds (i.e., 20.031 seconds).

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

class summationClient{
   public static void main(String[ ] args){
     try{
        InetAddress serverHost = InetAddress.getByName(args[0]);
        int serverPort = Integer.parseInt(args[1]);
        long startTime = System.currentTimeMillis( );
        int count = Integer.parseInt(args[2]);

        Socket clientSocket = new Socket(serverHost, serverPort);
        PrintStream ps = new PrintStream(clientSocket.getOutputStream());
        ps.println(count);
        ps.flush( );

        BufferedReader br = new BufferedReader(new
                  InputStreamReader(clientSocket.getInputStream()));

        int sum = Integer.parseInt(br.readLine());
        System.out.println(" sum = "+sum);

        long endTime = System.currentTimeMillis();
```

```
        System.out.println(" Time consumed for receiving the feedback from the server:
                                            "+(endTime-startTime)+" milliseconds");
        clientSocket.close( );
        }
      catch(Exception e){e.printStackTrace( );}
    }
}
```

Figure 24: Code for a Client Program that Requests a Server to Add Integers (from 1 to a Count value) and Return the Sum

---

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

class summationServer{
    public static void main(String[] args){
      try{
        int serverPort = Integer.parseInt(args[0]);
        ServerSocket calcServer = new ServerSocket(serverPort);
        while (true){
        Socket clientSocket = calcServer.accept( );
        BufferedReader br = new BufferedReader(new
                                    InputStreamReader(clientSocket.getInputStream( )));
        int count = Integer.parseInt(br.readLine( ));

        int sum = 0;
        for (int ctr = 1; ctr <= count; ctr++){
          sum += ctr;
          Thread.sleep(200);
        }

        PrintStream ps = new PrintStream(clientSocket.getOutputStream( ));
        ps.println(sum);
        ps.flush( );
        clientSocket.close( );

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

Figure 25: Code for an Iterative Server that Adds (from 1 to a Count value) and Returns the Sum

```
C:\res\tutorial\sockets\server>javac summationServerIterative.java

C:\res\tutorial\sockets\server>java summationServer 3456
```

```
C:\res\tutorial\sockets\client>java summationClient localhost 3456 100
 sum = 5050
 Time consumed for receiving the feedback from the server: 20031 milliseconds

C:\res\tutorial\sockets\client>
```

```
C:\res\tutorial\sockets\client>java summationClient localhost 3456 5
 sum = 15
 Time consumed for receiving the feedback from the server: 19500 milliseconds

C:\res\tutorial\sockets\client>_
```

Figure 26: Screenshots of Execution of an Iterative Summation Server and its Clients

4.6.2 Concurrent Server

Video Link: http://www.youtube.com/watch?v=ozHR0V_RwrQ

An alternative design is the idea of using a concurrent server, especially to process client requests with variable service time. When a client request is received, the server process spawns a separate thread, which is exclusively meant to handle the particular client. So, if a program has to sleep after each addition, it would be the particular thread (doing the addition) that will sleep and not the whole server process, which was the case with an iterative server. While a thread of a process is sleeping, the operating system could schedule the other threads of this process to run. With such a design, the waiting time of client requests, especially for those with a relatively shorter processing time, could be significantly reduced. Note that the code for the client program is independent of the design choice for the server. In other words, one should be able to use the same client program with either an iterative server or a concurrent server.

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

class summationThread extends Thread{
   Socket clientSocket;
   summationThread(Socket cs){      clientSocket = cs;      }

   public void run( ){
     try{
       BufferedReader br = new BufferedReader(new
                                   InputStreamReader(clientSocket.getInputStream( )));
       int count = Integer.parseInt(br.readLine( ));
       int sum = 0;
       for (int ctr = 1; ctr <= count; ctr++){
         sum += ctr;
         Thread.sleep(200);
```

```
        }

        PrintStream ps = new PrintStream(clientSocket.getOutputStream( ));
        ps.println(sum);
        ps.flush( );
        clientSocket.close( );
      }
      catch(Exception e){e.printStackTrace( );}
     }
}

class summationServer{
    public static void main(String[ ] args){
      try{
        int serverPort = Integer.parseInt(args[0]);
        ServerSocket calcServer = new ServerSocket(serverPort);
        while (true){
           Socket clientSocket = calcServer.accept( );
           summationThread thread = new summationThread(clientSocket);
           thread.start( );
         }
       }
      catch(Exception e){e.printStackTrace( );}
   }
}
```

Figure 27: Concurrent Server Program and the Implementation of a Summation Thread

```
C:\res\tutorial\sockets\server>javac summationServerConcurrent.java

C:\res\tutorial\sockets\server>java summationServer 3456
```

```
C:\res\tutorial\sockets\client>java summationClient localhost 3456 100
 sum = 5050
 Time consumed for receiving the feedback from the server: 20031 milliseconds

C:\res\tutorial\sockets\client>
```

```
C:\res\tutorial\sockets\client>java summationClient localhost 3456 5
 sum = 15
 Time consumed for receiving the feedback from the server: 1000 milliseconds

C:\res\tutorial\sockets\client>
```

Figure 28: Screenshots of Execution of a Concurrent Summation Server and its Clients

Figure 27 presents the code for a concurrent server. We implement the addition module inside the run( ) function of the SummationThread class, an object of which is spawned (i.e., a server thread) for each incoming client connection requests. From then on, the server thread handles the communication with the particular client through the Socket object returned by the accept( )

method of the ServerSocket class and passed as an argument to the constructor of the SummationThread class. The effectiveness of using a concurrent server (illustrated in Figure 28) can be validated through the reduction in the processing time for a client whose request to add integers from 1 to 5 sent to the server after the latter started to process a client request to add integers from 1 to 100. The result at the client that requests to add integers 1 to 5 (it takes only 1000 milliseconds – i.e., 1 second) would actually appear well ahead of that at the other client (it takes 20031 milliseconds – i.e. 20.031 seconds).

5 Multicast Sockets

Multicasting is the process of sending messages from one process to several other processes concurrently. It supports one-to-many Inter Process Communication (IPC). Multicasting is useful for applications like groupware, online conferences and interactive learning, etc. In applications or network services that make use of multicasting, we have a set of processes that form a group called multicast group. Each process in the group can send and receive messages. A message sent by any process in the group will be received by each participating process in the group.

A Multicast API should support the following primitive operations:
(1) Join – allows a process to join a specific multicast group. A process may be a member of one or more multicast groups at the same time.
(2) Leave – allows the process to stop participating in a multicast group.
(3) Send – allows a process to send a message to all the processes of a multicast group.
(4) Receive – allows a process to receive messages sent to a multicast group.

At the transport layer, the basic multicast supported by Java is an extension of UDP, which is connectionless and unreliable. There are four major classes in the API: (1) InetAddress (2) DatagramPacket (3) DatagramSocket and (4) MulticastSocket. The MulticastSocket class is an extension of the DatagramSocket class and provides capabilities for joining and leaving a multicast group. The constructor of the MulticastSocket class takes an integer argument that corresponds to the port number to which the object of this class would be bound to. The receive( ) method of the MulticastSocket class is a blocking-method, which when invoked on a MulticastSocket object will block the execution of the receiver until a message arrives to the port to which the object is bound to.

An IP multicast datagram must be received by all the processes that are currently members of a particular multicast group. Hence, each multicast datagram needs to be addressed to a specific multicast group instead of an individual process. In IPv4, a multicast group is specified by a class D IP address combined with a standard port number. The 224.x.x.x range is used for various static multicast addresses; so, we will avoid using multicast addresses starting with 224. In this chapter, we will use the address 225.4.5.6 in our examples.

5.1 *Example Program to Illustrate an Application in which a Message Sent by a Process Reaches all the Processes Constituting the Multicast Group*

Video Link: http://www.youtube.com/watch?v=vP9UbXDxUyY

In this example, we illustrate an application wherein there are two types of processes: (i) multicast sender – that can only send a message to a multicast group and (ii) multicast receiver – that can only receive a message sent to the multicast group. Similar to the case of connection-oriented and connectionless sockets, the multicast receiver (Figure 30) should be started first and

should be ready to receive messages sent to the port number to which the multicast socket is bound to. We then start the multicast sender (Figure 29).

To keep it simple, the multicast sender program stops after sending one message to the multicast group and the multicast receiver program stops after receiving one message for the group. The maximum size of the message that can be received in this example is 100 bytes.

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

class multicastSender{
  public static void main(String[ ] args){
   try{
     InetAddress group = InetAddress.getByName("225.4.5.6");
     MulticastSocket multicastSock = new MulticastSocket( );
     String msg = "Hello How are you?";
     DatagramPacket packet = new DatagramPacket(msg.getBytes( ), msg.length( ), group,
                                                                                3456);
     multicastSock.send(packet);
     multicastSock.close( );
    }
   catch(Exception e){e.printStackTrace( );}
  }
}
```

Figure 29: Code for Multicast Sender Program

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

class multicastReceiver{
  public static void main(String[ ] args){
    try{
      InetAddress group = InetAddress.getByName("225.4.5.6");
      MulticastSocket multicastSock = new MulticastSocket(3456);
      multicastSock.joinGroup(group);
      byte[ ] buffer = new byte[100];
      DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
      multicastSock.receive(packet);
      System.out.println(new String(buffer));
      multicastSock.close( );
    }
   catch(Exception e){e.printStackTrace( );}
  }
}
```

Figure 30: Code for Multicast Receiver Program

```
C:\3300-laptop\res\book-chapter\sockets\multicast>java multicastReceiver
Hello How are you?

C:\3300-laptop\res\book-chapter\sockets\multicast>

C:\Windows\system32\cmd.exe

C:\3300-laptop\res\book-chapter\sockets\multicast>java multicastReceiver
Hello How are you?

C:\3300-laptop\res\book-chapter\sockets\multicast>

C:\Windows\system32\cmd.exe
C:\3300-laptop\res\book-chapter\sockets\multicast>java multicastSender

C:\3300-laptop\res\book-chapter\sockets\multicast>
```

Figure 31: Screenshots of Execution of a Multicast Sender and Receiver Program

5.2 *Example Program to Illustrate an Application in which each Process of the Multicast Group Sends a Message that is Received by all the Processes Constituting the Group*

Video Link: http://www.youtube.com/watch?v=64or8VY2gGw

In this example, each process should be both a multicast sender as well as a receiver such that the process can send only one message (to the multicast group); but, should be able to receive several messages. Since a process can send only one message, the number of messages received by a process would equal the number of processes that are part of the multicast group. Since a process should have both the sending and receiving functionality built-in to its code, we implement the relatively simpler sending module in the main( ) function; whereas, the receiving functionality is implemented as a thread (readThread class in Figure 32). The readThread object is spawned and starts to run before the sending module begins its execution. In order to facilitate this, the code in Figure 32 will require all the processes to be started first. After all the processes have begun to run (i.e., the readThread has been spawned), we then press the Enter-key in each process command window. This will trigger the sending of a message by each process. The readThread displays the received message (refer to Figure 33).

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

class readThread extends Thread{
  InetAddress group;
  int multicastPort;
  int MAX_MSG_LEN = 100;

  readThread(InetAddress g, int port){
    group = g;
```

```
     multicastPort = port;
    }

    public void run(){

     try{
       MulticastSocket readSocket = new MulticastSocket(multicastPort);
       readSocket.joinGroup(group);

       while (true){
          byte[] message = new byte[MAX_MSG_LEN];
          DatagramPacket packet = new DatagramPacket(message, message.length, group,
                                                                     multicastPort);
          readSocket.receive(packet);
          String msg = new String(packet.getData());
          System.out.println(msg);
        }
      }
     catch(Exception e){ e.printStackTrace( );}
    }
}


class multicastSenderReceiver{
  public static void main(String[] args){

   try{
     int multicastPort = 3456;
     InetAddress group = InetAddress.getByName("225.4.5.6");
     MulticastSocket sendSocket = new MulticastSocket( );

     readThread rt = new readThread(group, multicastPort);
     rt.start();

     int processID = Integer.parseInt(args[0]);
     String message = args[1];
     String multicastMessage = "Message from Process # "+processID+" : "+message;
     byte[] msg = multicastMessage.getBytes();
     DatagramPacket packet = new DatagramPacket(msg, msg.length, group,
                                                                     multicastPort);

     System.out.print("Hit return to send message\n\n");
     BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
     br.readLine( );

     sendSocket.send(packet);
```

```
    sendSocket.close();
   }
  catch(Exception e){ e.printStackTrace( );}

   }
}
```

Figure 32: Code for the Multicast Sender and Receiver Program that can both Send and Receive

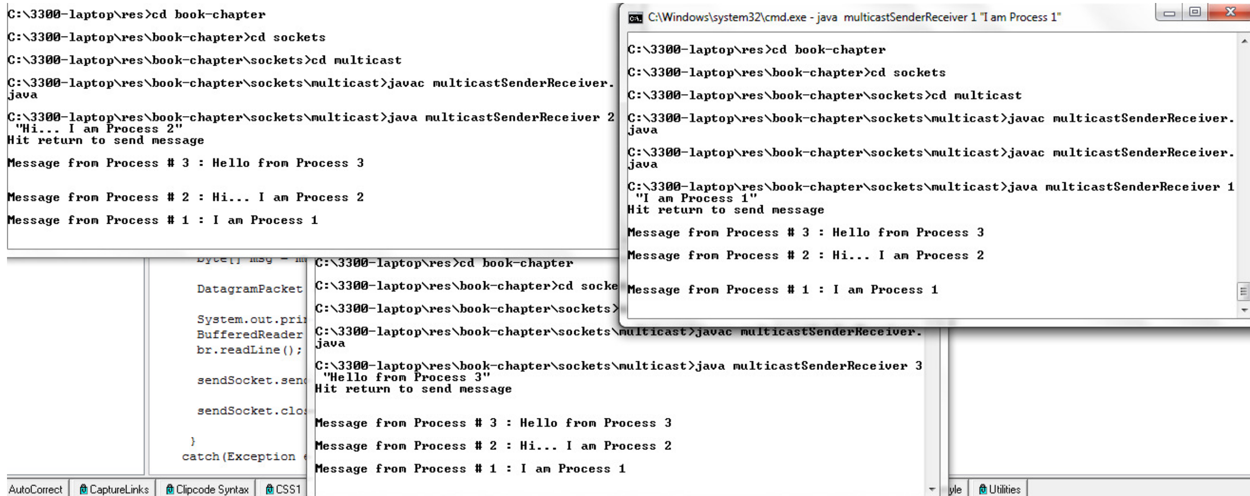

Figure 33: Execution of Multicast Sender and Receiver Program that can both Send and Receive

6 Exercises

1. Implement a simple file transfer protocol (FTP) using connection-oriented and connectionless sockets. The connection-oriented FTP works as follows: At the client side, the file to be transferred is divided into units of 100 bytes (and may be less than 100 bytes for the last unit depending on the size of the file). The client transfers each unit of the file to the server and expects an acknowledgment from the server. Only after receiving an acknowledgment from the server, the client transmits the next unit of the file. If the acknowledgment is not received within a timeout period (choose your own value depending on your network delay), the client retransmits the unit. The above process is repeated until all the contents of the file are transferred. The connectionless FTP works simply as follows: The file is broken down into lines and the client sends one line at a time as a datagram packet to the server. There is no acknowledgment required from the server side.
2. Develop a concurrent file server that spawns several threads, one for each client requesting a specific file. The client program sends the name of the file to be downloaded to the server. The server creates the thread by passing the name of the file as the argument for the thread constructor. From then on, the server thread is responsible for transferring the contents of the requested file. Use connection-oriented sockets (let the transfer size be at most 1000 bytes per flush operation). After a flush operation, the server thread sleeps for 200 milliseconds.

3. Develop a “Remote Calculator” application that works as follows: The client program inputs two integers and an arithmetic operation (‘*’,’/’,’%’,’+’,’-‘) from the user and sends these three values to the server side. The server does the binary operation on the two integers and sends backs the result of the operation to the client. The client displays the result to the user.
4. Develop a streaming client and server application using connectionless sockets that works as follows: The streaming client contacts the streaming server requesting a multi-media file (could be an audio or video file) to be sent. The server then reads the contents of the requested multi-media file in size randomly distributed between 1000 and 2000 bytes and sends the contents read to the client as a datagram packet. The last datagram packet that will be transmitted could be of size less than 1000 bytes, if required. The client reads the bytes, datagram packets, sent from the server. As soon as a reasonable number of bytes are received at the client side, the user working at the client side should be able to launch a media player and view/hear the portions of the received multi-media file while the downloading is in progress.
5. Develop a simple chatting application using (i) Connection-oriented and (ii) Connectionless sockets. In each case, when the user presses the “Enter” key, whatever characters have been typed by the user until then are transferred to the other end. You can also assume that for every message entered from one end, a reply must come from the other end, before another message could be sent. In other words, more than one message cannot be sent from a side before receiving a response from the other side. For connectionless communication, assume the maximum number of characters that can be transferred in a message to be 1000. The chat will be stopped by pressing Ctrl+C on both sides.
6. Extend the single client – single server chatting application developed in Q5 using connection-oriented sockets to a multiple client – single server chatting application. The single server should be able to chat simultaneously with multiple clients. In order to do this, you will have to implement the server program using threads. Once a client program contacts a server, the server process spawns a thread that will handle the client. The communication between a client and its server thread will be like a single client-single server chatting application.
7. Develop a multicast chatting tool that will be used to communicate among a group of processes (four processes for this project). Each process should be able to send and receive any number of messages. The chat tool should have the following functionalities:
    1) Get the message from the user and send it to all the other processes belonging to the group. The message from a process should be numbered (for example, the 14th message from process 1 can be sent like this: Message #14 from Process #1: We are going to Dallas today). A process can receive a copy of its own message.
    2) Read the messages sent by any other process and display the message to the user.
8. Develop an election vote casting application as follows: There are two candidates A and B contesting an election. There are five electorates (processes) and each electorate can cast their vote only once and for only one of the two candidates (A or B). The vote cast by an electorate is a character ‘A’ or ‘B’, sent as a multicast message to all the other electorates. The winner is the candidate who gets the maximum number of votes. After casting the vote and also receiving the vote messages from all other electorates, each electorate should be able to independently determine the winner and display it.

References

[1] D. E. Comer, “Computer Networks and Internets,” 5th Edition, Prentice Hall, 2008.
[2] M. L. Liu, “Distributed Computing: Principles and Applications,” Addison Wesley, 2004.
[3] Java API: http://download.oracle.com/javase/1.4.2/docs/api/