**Institute of Technology, Tralee**

**Computing Department**

Distributed Computing

Lab 1 – Sockets

**It is best to do the following exercises completely from the command line.**

**Keep a log book of your exercises and your answers. Do the exercises in this lab and write the answers in your log book. It will be a good aid in revising for the exam**

Note: You can use Notepad++ to write your code. Save the files as .java. Compile using javac and run using java

**Exercise 1 Datagram Sockets:**

The following code illustrates the code for two programs that use datagram (connectionless) sockets to exchange a single data string. By design the logic of the program is as simple as possible to highlight the basic syntax for interprocess communications. Note that the sender creates a datagram packet that contains a destination address (31 – 33) while the receiver’s datagram packet does not carry a destination address. Note also that the sender’s socket is bound to an unspecified port number (line 28) so that the sender may specify this port number in its datagram (line 33) as the destination. It should also be mentioned that for simplicity the sample programs use rudimentary syntax to handle exceptions.

**The Sender code:**

import java.net.\*;

import java.io.\*;

/\*\*

\* This example illustrates the basic method calls for connectionless

\* datagram socket.

\* @author M. L. Liu

\*/

public class Example1Sender {

// An application which sends a message using connectionless

// datagram socket.

// Three command line arguments are expected, in order:

// <domain name or IP address of the receiver>

// <port number of the receiver's socket>

// <message, a string, to send>

public static void main(String[] args) {

if (args.length != 3)

System.out.println

("This program requires three command line arguments");

else {

try {

InetAddress receiverHost = InetAddress.getByName(args[0]);

System.out.println("Host Name: " + receiverHost.getHostName());

System.out.println("Host Address: " +receiverHost.getHostAddress());

//System.out.println("canonical: " +receiverHost.getCanonicalHostName());

System.out.println("to String: " +receiverHost.toString());

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

String message = args[2];

// instantiates a datagram socket for sending the data

DatagramSocket mySocket = new DatagramSocket(); // bound to any available port

System.out.println("Bound to port: " + mySocket.getLocalPort());

byte[ ] buffer = message.getBytes( );

DatagramPacket datagram =

new DatagramPacket(buffer, buffer.length,

receiverHost, receiverPort);

mySocket.send(datagram);

mySocket.close( );

} // end try

catch (Exception ex) {

ex.printStackTrace( );

}

} // end else

} // end main

} // end class

**The Receiver Code:**

import java.net.\*;

import java.io.\*;

/\*\*

\* This example illustrates the basic method calls for connectionless

\* datagram socket.

\* @author M. L. Liu

\*/

public class Example1Receiver {

// An application which receives a message using connectionless

// datagram socket.

// A command line argument is expected, in order:

// <port number of the receiver's socket>

// Note: the same port number should be specified in the

// command-line arguments for the sender.

public static void main(String[] args) {

if (args.length != 1)

System.out.println

("This program requires a command line argument.");

else {

int port = Integer.parseInt(args[0]);

final int MAX\_LEN = 10;

// This is the assumed maximum byte length of the

// datagram to be received.

try {

DatagramSocket mySocket = new DatagramSocket(port);

// instantiates a datagram socket for receiving the data

byte[ ] buffer = new byte[MAX\_LEN];

DatagramPacket datagram =

new DatagramPacket(buffer, MAX\_LEN);

mySocket.receive(datagram); // blocking

String message = new String(buffer);

System.out.println(message);

mySocket.close( );

} // end try

catch (Exception ex) {

ex.printStackTrace( );

}

} // end else

} // end main

} // end class

* 1. Compile the .java files. Then run the two programs by (i) executing the receiver, then (ii) executing the sender, taking care to specify the appropriate command-line arguments in each case. The message should not exceed the maximum length allowed in the receiver (10 characters). Describe the outcome of the run – execute each application in a separate window on screen.

**Exercise 2**

In exercise 1 above, the datagram socket code, the communication is simplex; that is, it is one way, from the sender to the receiver. It is possible to make the communication duplex, or bidirectional. To do so, **Example1Sender** will need to bind its socket to a specific address so that **Example1Receiver** can send datagrams to that address.

2.1 Write ***Example2SenderReceiver*** and ***Example2ReceiverSender***. For code modularity, a class called **MyDatagramSocket** is shown below. This class is a subclass of **DatagramSocket**, with two instance methods for sending and receiving a message respectively. Use this class in your solution.

import java.net.\*;

import java.io.\*;

/\*\*

\* A subclass of DatagramSocket which contains

\* methods for sending and receiving messages

\* @author M. L. Liu

\*/

public class MyDatagramSocket extends DatagramSocket {

static final int MAX\_LEN = 100;

MyDatagramSocket(int portNo) throws SocketException{

super(portNo);

}

public void sendMessage(InetAddress receiverHost, int receiverPort,

String message) throws IOException {

byte[ ] sendBuffer = message.getBytes( );

DatagramPacket datagram =

new DatagramPacket(sendBuffer, sendBuffer.length,

receiverHost, receiverPort);

this.send(datagram);

} // end sendMessage

public String receiveMessage()

throws IOException {

byte[ ] receiveBuffer = new byte[MAX\_LEN];

DatagramPacket datagram =

new DatagramPacket(receiveBuffer, MAX\_LEN);

this.receive(datagram);

String message = new String(receiveBuffer);

return message;

} //end receiveMessage

} //end class

**Exercise 3: Stream Sockets**

The next example illustrates the basic syntax for stream-mode sockets (i.e. connection-oriented communication)

Whereas the datagram socket API supports the exchange of discrete units of data, the stream-mode socket API provides a model of data transfer based on stream-mode I/O of the Unix operating system. Data is transferred using the concept of a continuous data stream from a source to a destination.

For event synchronization the following operations are blocking:

* **Accept**. If no request is waiting, the server process will be suspended until a request for connection arrives.
* **Reading from the input stream associated with a data soc**ket. If the amount of data requested is not currently present in the data stream, the reading process will be blocked until a sufficient amount of data has been written into the stream

**The Receiver** accepts connections by establishing a ServerSocket object at the specified port (say 12345).The **Sender** creates a Socket object, specifying as arguments the host name and port number (12345 in this case) of the Receiver, Once the connection has been accepted by the Receiver, a message is written to the socket’s data stream by the Receiver. At the Sender, the message is read from the data stream and displayed.

**The Receiver pseudocode:**

* Create a connection socket and listen for connection requests
* Accept a connection
* Create a data socket for reading from or writing to the socket stream
* Read from the stream
* Get an output stream for writing to the socket
* Write to the stream
* Close the data socket
* Close the connection

**The Sender pseudocode**

* Create a data socket and request a connection
* Get an output stream for writing to the socket
* Write to the stream
* Get an input stream for reading to the socket
* Read from the stream
* Close the data socket.

**The Receiver**

import java.net.\*;

import java.io.\*;

/\*\*

\* This example illustrates the basic syntax for stream-mode

\* socket.

\* @author M. L. Liu

\*/

public class Example2Receiver {

// An application that accepts a connection and receives a message

// using stream-mode socket.

// Two command line arguments are expected, in order:

// <port number for the the Server socket used in this process>

// <message, a string, to send>

public static void main(String[] args) {

if (args.length != 2)

System.out.println

("This program requires two command line arguments");

else {

try {

int portNo = Integer.parseInt(args[0]);

String message = args[1];

// instantiates a socket for accepting connection

ServerSocket connectionSocket = new ServerSocket(portNo);

/\*\*/ System.out.println("now ready accept a connection on port: " + portNo);

// wait to accept a connecion request, at which

// time a data socket is created

Socket dataSocket = connectionSocket.accept();

/\*\*/ System.out.println("connection accepted, new data socket on port: " + dataSocket.getLocalPort());

// get a output stream for writing to the data socket

OutputStream outStream = dataSocket.getOutputStream();

// create a PrinterWriter object for character-mode output

PrintWriter socketOutput =

new PrintWriter(new OutputStreamWriter(outStream));

// write a message into the data stream

socketOutput.println(message);

//The ensuing flush method call is necessary for the data to

// be written to the socket data stream before the

// socket is closed.

socketOutput.flush();

/\*\*/ System.out.println("message sent");

dataSocket.close( );

/\*\*/ System.out.println("data socket closed");

connectionSocket.close( );

/\*\*/ System.out.println("connection socket closed");

} // end try

catch (Exception ex) {

ex.printStackTrace( );

} //end catch

} // end else

} // end main

} // end class

**The Sender**

import java.net.\*;

import java.io.\*;

/\*\*

\* This example illustrates the basic syntax for stream-mode

\* socket.

\* @author M. L. Liu

\*/

public class Example2Sender {

// An application that requests a connection and

// sends a message using stream-mode socket.

// Two command line arguments are expected:

//

// <host name of the connection accceptor>

// <port number of the connection accceptor>

public static void main(String[] args) {

if (args.length != 2)

System.out.println

("This program requires two command line arguments");

else {

try {

InetAddress acceptorHost = InetAddress.getByName(args[0]);

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

// instantiates a data socket

Socket mySocket = new Socket(acceptorHost, acceptorPort);

/\*\*/ System.out.println("Connection request granted to port: " + acceptorPort);

// get an input stream for reading from the data socket

InputStream inStream = mySocket.getInputStream();

// create a BufferedReader object for text line input

BufferedReader socketInput =

new BufferedReader(new InputStreamReader(inStream));

/\*\*/ System.out.println("waiting to read");

// read a line from the data stream

String message = socketInput.readLine( );

/\*\*/ System.out.println("Message received:");

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

mySocket.close( );

/\*\*/ System.out.println("data socket closed");

} // end try

catch (Exception ex) {

ex.printStackTrace( );

} //end catch

} // end else

} // end main

} // end class

The Receiver process starts execution first. The process is suspended when the blocking accept method is called, then unsuspended when it receives the connection request from the Sender. Upon resuming execution, the Receiver writes a message to the socket before closing both the data socket and connection socket.

The execution of the Sender process proceeds as follows: A Socket object is instantiated, and an implicit connect request is issued to the Receiver. Although the connect request is nonblocking, data exchange via the connection cannot proceed until the connection is accepted by the process at the other end. Once the connection is accepted, the process invokes a read operation to read a message from the socket. Because the read operation is blocking, the process is suspended once again until the data for the message is received, whereupon the process closes the socket and processes the data.

3.1 Compile and run the program. Start the Receiver first, then the Sender. Sample commands are:

java Example2Receiver 12345 Hello

java Example2Sender localhost 12345

Describe and explain the outcome.

Notes:

1. Because we are dealing with a data stream, we can use Java’s ***PrintWriter*** class for reading from a stream.
2. The roles of Sender as sender of data and Receiver as receiver of data can be reversed. In fact, either process can both read from and write to the stream
3. Although the example reads and writes one line at a time, it is also possible to read and write part of a line (read(), write()), though it is the norm to read/write one line at a time for text based protocols.

**Exercise 4:**

To allow the separation of the application logic and service logic in the programs, we can employ a subclass that hides the details of the data socket. Modify the code in exercise 2 the Stream Sockets example, to use the ***MyStreamSocket*** class instead of the Java class Socket.

import java.net.\*;

import java.io.\*;

/\*\*

\* A wrapper class of Socket which contains

\* methods for sending and receiving messages

\* @author M. L. Liu

\*/

public class MyStreamSocket extends Socket {

private Socket socket;

private BufferedReader input;

private PrintWriter output;

MyStreamSocket(String acceptorHost,

int acceptorPort ) throws SocketException,

IOException{

socket = new Socket(acceptorHost, acceptorPort );

setStreams( );

}

MyStreamSocket(Socket socket) throws IOException {

this.socket = socket;

setStreams( );

}

private void setStreams( ) throws IOException{

// get an input stream for reading from the data socket

InputStream inStream = socket.getInputStream();

input =

new BufferedReader(new InputStreamReader(inStream));

OutputStream outStream = socket.getOutputStream();

// create a PrinterWriter object for character-mode output

output =

new PrintWriter(new OutputStreamWriter(outStream));

}

public void sendMessage(String message)

throws IOException {

output.print(message + "\n");

//The ensuing flush method call is necessary for the data to

// be written to the socket data stream before the

// socket is closed.

output.flush();

} // end sendMessage

public String receiveMessage( )

throws IOException {

// read a line from the data stream

String message = input.readLine( );

return message;

} //end receiveMessage

public void close( )

throws IOException {

socket.close( );

}

} //end class