Institute of Information & Communication Technology
University of Sindh Jamshoro

Distributed Computing (Lab.)

LAB - 01

Laboratory Guide 1

1. Working with `java.net.InetAddress`
2. Client/Server communication using TCP/IP with `ServerSocket` and `Socket`
3. Client/Server communication using UDP with `DatagramSocket`, `DatagramPacket`, etc.

1. IP Addresses and `java.net.InetAddress`

The InetAddress class in the package java.net handles Internet addresses both as host names and as IP addresses. An object of InetAddress abstracts the details of an IP Address. The static method getByName of this class uses DNS (Domain Name System) to return the Internet address of a specified host name as an InetAddress object. Since method getByName throws the checked exception UnknownHostException if the host name is not recognised, we must either throw this exception or (preferably) handle it with a catch clause. Another static method, getLocalHost of the InetAddress class returns the Internet address of the local machine.

Task A: Demostrate the use of getByName() static method of InetAddress class.

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

public class IPFinder
{
    public static void main(String[] args)
    {
        String host;
        Scanner input = new Scanner(System.in);

        System.out.print("\n\nEnter host name: ");
        host = input.next();
        try
        {
            InetAddress address =
                            InetAddress.getByName(host);
            System.out.println("IP address: "
                                   + address.toString());
        }
        catch (UnknownHostException uhEx)
        {
            System.out.println("Could not find " + host);
        }
    }
}
```

Task B: Using a similar style of coverage, demonstrate the use of the static method getLocalHost() of class InetAddress.

Task C: Explore the class java.net.InetAddress and list some of the other static methods with a brief description of their purpose.

2. Client/Server communication using TCP/IP

Different processes (programs) can communicate with each other using sockets. A socket represents one end of the link between the communicating programs. TCP/IP is connection-oriented protocol in the sense that once a connection between two communicating devices has been established it is kept alive for as long as the dialogue is running. Java supports this communication using TCP/IP protocol by providing Socket and ServerSocket classes in the java.net package.

Setting up a server process requires five steps:

1. ***Create a ServerSocket object***
   The ServerSocket constructor requires a port number (1024-65535, for non-reserved ones) as an argument. For example:

```
ServerSocket servSock = new ServerSocket(1234);
```

2. ***Put the server into a waiting state***
   The server waits indefinitely for a client to connect. It does this by calling method accept of class ServerSocket, which returns a Socket object when a connection is made. For example:

```
Socket link = servSock.accept();
```

3. ***Set up input and output streams***
   Methods getInputStream and getOutputStream of class Socket are used to get references to streams associated with the socket returned in the previous step. These streams will be used for communication with the client that has just made connection. For example:

```
// For input
Scanner input = new Scanner(link.getInputStream());

// For output
PrintWriter output = new PrintWriter(link.getOutputStream());
```

   Or, in case of versions of JDK older than 1.5:

```
// For input
BufferedReader input =
      new BufferedReader(new InputStreamReader(
      link.getOutputStream()));
```

4. ***Send and receive data***

   Having set up our Scanner (or BufferedReader) and PrintWriter objects, sending and receiving data is very straightforward. We simply use method nextLine for receiving data and method println for sending data, just as we might do for console I/O. For example:

```
output.println("Awaiting  data...");
String msg = input.nextLine();      // input is Scanner
```

   Or, if BufferedReader is used:

```
String msg = input.readLine();   // input is BufferedReader
```

5. ***Close the connection to the client (when the dialogue is over)***

   Close the connection (after completion of the dialogue). This is achieved via method close of class Socket. For example:

```
link.close();
```

The process of setting up a client for very similar to the above process. It involves the following four steps:

1. ***Establish a connection to the server***

   We create a Socket object, supplying its constructor with the following two arguments:
   - the server's IP address (of type InetAddress)
   - the appropriate port number for the service

   (The port number for server and client programs must be the same, of course!)

   For simplicity's sake, we shall place client and server on the same host, which will allow us to retrieve the IP address by calling static method getLocalHost of class InetAddress. For example:

```
Socket link = new Socket(InetAddress.getLocalHost(),1234);
```

2. ***Open input and output streams***

   This process is the same as it was in the server program. Use the getInputStream() and getOutputStream() static methods.

3. ***Send and receive data***

   Same as described in the last section. The input stream on the client's side will read data sent out by the output stream on the server's side; and the output stream on the client's side will send out data that will be read in by the input stream on the server's side.

4. ***Close the connection***
   This too is same as for the server process. We use the close() method of the socket to terminate the connection.

Here is an example program that works as an echo server – it simply echoes back to the client whatever message it receives from the client.

```
import java.io.*;
import java.net.*;
import java.util.*;
public class TCPEchoServer
{
   private static ServerSocket servSock;
   private static final int PORT = 1234;
   public static void main(String[] args) {
      System.out.println("Opening port...\n");
      try {
         servSock = new ServerSocket(PORT);    //Step 1.
      }
      catch(IOException ioEx) {
         System.out.println("Unable  to  attach  to  port!");
         System.exit(1);
      }
      do {
         handleClient();
      }while (true);
   }

   private static void handleClient()
   {
      Socket link = null;                      //Step 2.
      try
      {
         link = servSock.accept();             //Step 2.
         Scanner input =
     new  Scanner(link.getInputStream());//Step 3.
         PrintWriter output =
             new PrintWriter(
                 link.getOutputStream(),true); //Step 3.
         int numMessages = 0;
         String message = input.nextLine();    //Step 4.
         while (!message.equals("***CLOSE***"))
         {
            System.out.println("Message received.");
            numMessages++;
            output.println("Message " + numMessages
                             + ": " + message); //Step 4.
            message = input.nextLine();
         }
         output.println(numMessages
      +  "  messages  received.");//Step 4.
  }
  catch(IOException  ioEx)
  {
   ioEx.printStackTrace();
  }
  finally
  {
   try
   {
    System.out.println(
       "\n*  Closing  connection...  *");
    link.close();             //Step 5.
   }
   catch(IOException  ioEx)
   {
    System.out.println(
         "Unable  to  disconnect!");
    System.exit(1);
   }
  }
 }
}
```

And, here is the associate client program. The messages sent by the client, while the server is running, are echoed back to the client. The client sends the message “***CLOSE***” to indicate that it wishes to terminate the active dialogue:

```
import java.io.*;
import java.net.*;
import java.util.*;
public class TCPEchoClient
{
  private static InetAddress host;
  private static final int PORT = 1234;
  public static void main(String[] args)
 {
  try
  {
   host  =  InetAddress.getLocalHost();
  }
  catch(UnknownHostException  uhEx)
  {
      System.out.println("Host ID not found!");
   System.exit(1);
  }
  accessServer();
 }
  private static void accessServer()
 {
    Socket link = null;  //Step 1.
  try
  {
      link = new Socket(host,PORT);  //Step 1.
      Scanner input =
     new  Scanner(link.getInputStream());  //Step 2.
   PrintWriter  output  =
        new PrintWriter(
    link.getOutputStream(),true);  //Step 2.
      //Set up stream for keyboard entry...
      Scanner userEntry = new Scanner(System.in);
   String  message,  response;
   do
   {
    System.out.print("Enter  message:  ");
    message  =  userEntry.nextLine();
    output.println(message);    //Step 3.
    response  =  input.nextLine();  //Step 3.
    System.out.println("\nSERVER>  "+response);
   }while  (!message.equals("***CLOSE***"));
  }
  catch(IOException  ioEx)
  {
   ioEx.printStackTrace();
  }
  finally
  {
   try
   {
    System.out.println(
       "\n*  Closing  connection...  *");
    link.close();  //Step 4.
   }
   catch(IOException  ioEx)
   {
    System.out.println(
         "Unable  to  disconnect!");
    System.exit(1);
   }
  }
 }
}
```