JOBSHEET 2

Introduction Programming Socket

I. PREFACE

Welcome to this learning guide on Inter-Process Communication (IPC) using TCP! This guide has been designed to help you understand the fundamentals of IPC and how to develop networked applications using TCP sockets. Whether you are a beginner or an experienced programmer, this guide will provide you with valuable insights, practical tips, and resources to enhance your learning experience. By following this guide, you will be able to develop applications that facilitate communication between different processes using TCP sockets and explore the vast possibilities of IPC.

II. LEARNING OBJECTIVES

Upon completing this learning guide, you will be able to:

- Understand the basics of Inter-Process Communication (IPC)
- Develop applications that utilize TCP sockets for IPC
- Implement client-server communication using TCP sockets
- Handle errors and exceptions in IPC with TCP
- Explore advanced concepts of IPC, such as process synchronization and message passing

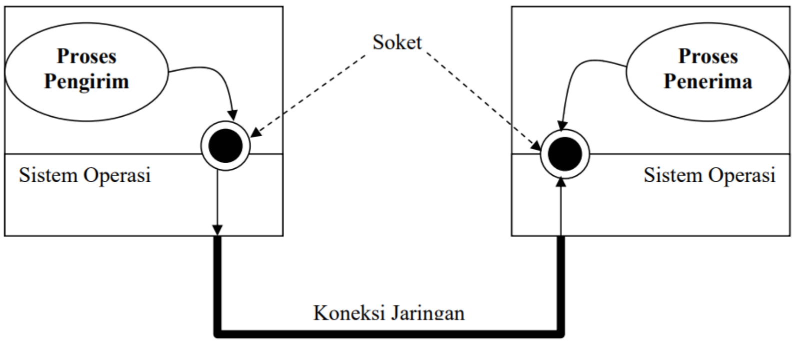
III. TOOLS AND MATERIALS

To follow this learning guide, you will need the following tools and materials:

- A computer with a programming language such as Python, Java, or C/C++ installed
- A text editor or an Integrated Development Environment (IDE)
- A basic understanding of programming concepts such as variables, data types, functions, and control structures
- Access to the internet to download and install necessary libraries and modules.

I. INTER PROCESS COMMUNICATION CONCEPT

In such a system, there is communication between processes located on different computers. What is a Socket? A Socket is an IPC (Inter-Process Communication) facility for network applications.



To enable a socket to communicate with another socket, it needs to be given a unique address as an identification. The socket address consists of an IP Address and a Port Number. An example of a socket address is 192.168.29.30:3000, where the number 3000 is the port number. The IP address can use a Local Area Network (LAN) or internet address. Therefore, a socket can be used for IPC on both LAN and the internet.

Why is a port number needed? Is the IP address of the target computer alone not enough? A port number is needed because a computer typically runs more than one process. Therefore, additional information is needed to identify the process to be contacted. If the computer's IP address is likened to a company's telephone number, the port number is its extension number. A process that wants to communicate with another process through the socket mechanism must bind itself to one of the ports on its computer. This self-binding process is called binding.

Types of Socket Communication Generally, there are two types of communication using sockets: stream communication and datagram communication. Stream communication is also known as connection-oriented communication, while datagram communication is known as connectionless communication. The standard protocol for stream communication is known as TCP (Transmission Control Protocol), while the standard protocol for datagram communication is known as UDP (User Datagram Protocol).

In UDP, every time a data packet is sent, information about the sender socket and the destination socket address is also sent. This is not required by TCP because TCP will first set up a connection with the target socket. Once the connection is established, it is not necessary to send the sender socket information every time data is sent. This is because the target process will identify every arriving data at the destination socket as data from the sender process. The connection established by TCP is bidirectional.

Another difference is that UDP has a maximum datagram (data packet) size limit of 64 kb, while TCP does not have this limit because data is sent as a stream of data. TCP will actually break up large data into several small packets and assign them a sequential number. At the receiver socket side, these data packets are stored, reordered, and finally combined into large data again.

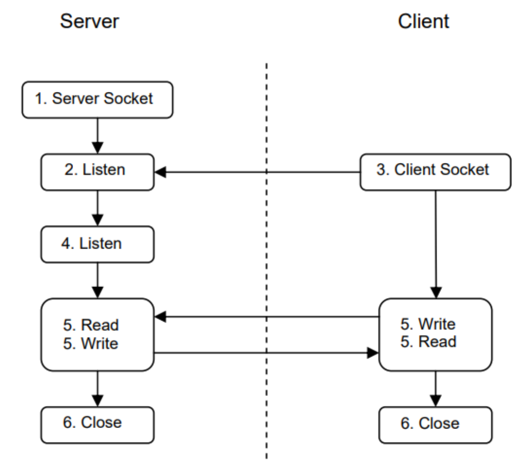
Another difference is that UDP is an unreliable protocol. When data packets are sent, UDP does not check whether the sent data has reached the destination. So, with UDP, there is no certainty for the sender side that its data has arrived at the destination in good condition. In contrast, TCP is a reliable protocol that always waits for confirmation from the receiver socket. If necessary, lost data packets will be sent again. The consequence is that TCP creates a higher network traffic overhead than UDP.

Client-Server Application Model The application model that uses socket communication with the TCP protocol is illustrated in Figure 2. The socket objects on the client and server sides differ slightly. On the server application side, a server socket is created (1) and the listen operation is performed (2). Essentially, this operation waits for connection requests from the client side. On the client side, a regular socket is created.

When the client socket (3) passes the server socket address as an argument, the client socket will automatically try to request a connection to the server socket. When the client connection request reaches the server, the server will create a regular socket. Thi

s socket will communicate with the socket on the client side. After that, the server socket can go back to listen (4) to wait for new connection requests.

After creating a connection between the client and server, then the two can be mutually exchanging messages (5). Either or both of them can then end the communication by closing the socket (6).



II. SOCKET PROGRAMMING IN JAVA

In Java, Socket and ServerSocket objects are provided for TCP socket communication. ServerSocket is used on the server-side of the application, while Socket is used on both the server and client-side. The following are the steps for creating socket communication in Java:

- **Open sockets**

  - On the client, this is done as follows:

```
Socket myKlien = null;
try {
  myKlien = new Socket("host", NomorPort);
} catch (UnknownHostException uhe) {
  uhe.printStackTrace();
} catch (IOException e) {
  e.printStackTrace();
}
```

  - On servers:

```
public static final int NomorPort = 1234;
ServerSocket Layanan = null;
try {
  Layanan = new ServerSocket(NomorPort);
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

  - In addition, the server must also create a socket object of the ServerSocket class to listen and receive connections from clients, as follows:

```
Socket layananSocket = null;
try {
  layananSocket = Layanan.accept();
} catch (IOException iex) {
  iex.printStackTrace();
}
```

- **Create data input streams**

  - To create an input stream on the client, you can use the BufferedReader class to receive a response from the server.

```
BufferedReader is = null;
try {
  is              =              new            BufferedReader(new
InputStreamReader(myKlien.getInputStream()));
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

  - On the server you can also use BufferedReader to receive input from the client.

```
BufferedReader is = null;
try {
  is              =              new            BufferedReader(new
InputStreamReader(Layanan.getInputStream()));
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

- **Creating a data output stream**

  - On the client, you can use the DataOutputStream class to send data to the socket server.

```
DataOutputStream os = null;
try {
  os = new DataOutputStream(myKlien.getOutputStream());
} catch (IOException ix) {
  ix.printStackTrace();
}
```

  - On servers:

```
DataOutputStream os = null;
try {
  os = new DataOutputStream(Layanan.getOutputStream());
} catch (IOException ie) {
  ie.printStackTrace();
}
```

- **Send and receive messages**

  - To send messages using the DataOutput Stream that has been formed and connected to the socket output data buffer.

```
os.writeBytes(dataOutput);
```

  - To receive messages using the BufferedReader that has been created andconnected to the socket input data buffer.

```
dataInput=Is.readLine();
```

- **Close the socket**

  - On clients:

```
try {
  os.close();
  is.close();
  myKlien.close();
} catch (IOException io) {
  io.printStackTrace();
}
```

➢ On servers:

```
try {
  os.close();
  is.close();
  layananSocket.close();
} catch (IOException ic) {
  ic.printStackTrace();
}
```

III. BUILDING A SIMPLE TCP CLIENT-SERVER APPLICATION

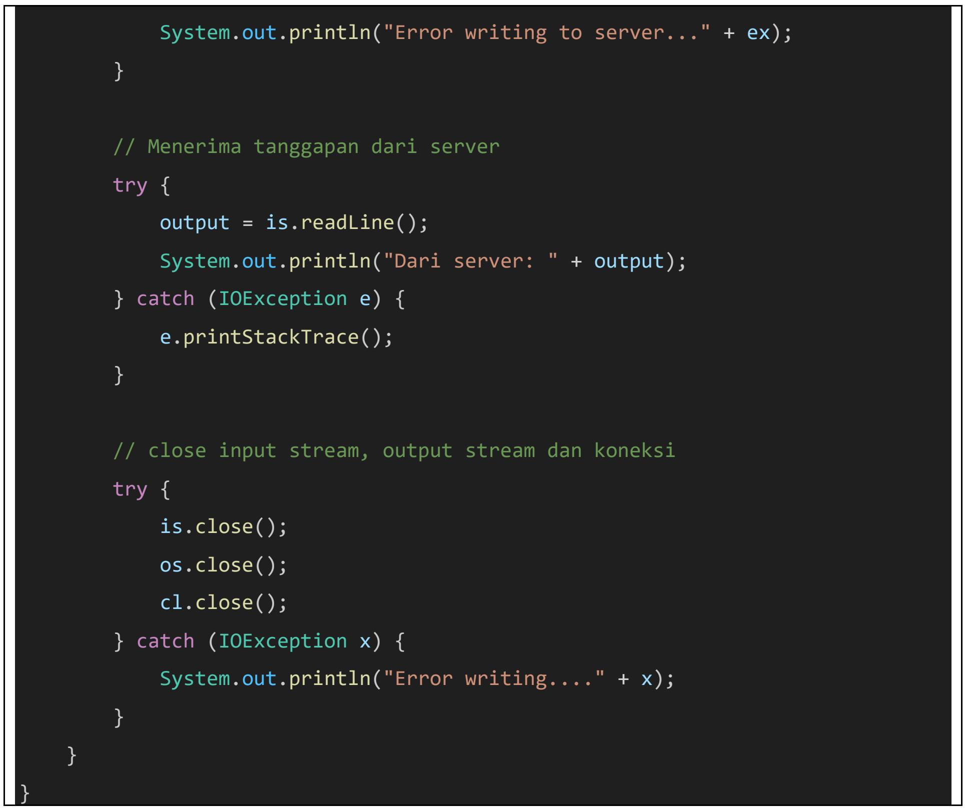
- **Practicum 5 - R**

➢ Create the server program below, save it with the name simpleServer.java:

```
import java.io.*;
import java.net.*;
public class SimpleServer {

    public final static int TESTPORT = 5000;
    public static void main(String args[]) {
        ServerSocket checkServer = null;
        String line;
        BufferedReader is = null;
        DataOutputStream os = null;
        Socket clientSocket = null;

        try {
            checkServer = new ServerSocket(TESTPORT);
            System.out.println("Aplikasi Server hidup ...");
        } catch (IOException e) {
            System.out.println(e);
        }

        try {
            clientSocket = checkServer.accept();
            is = new BufferedReader(new
            InputStreamReader(clientSocket.getInputStream()));
            os = new DataOutputStream(clientSocket.getOutputStream());
        } catch (Exception ei) {
            ei.printStackTrace();
        }

        try {
            line = is.readLine();
            System.out.println("Terima : " + line);

            if (line.compareTo("salam") == 0) {
                os.writeBytes("salam juga");
            } else {
                os.writeBytes("Maaf, saya tidak mengerti");
        }
        } catch (IOException e) {
            System.out.println(e);
        }

        try {
            os.close();
            is.close();
            clientSocket.close();
        } catch (IOException ic) {
            ic.printStackTrace();
        }
    }
}
```

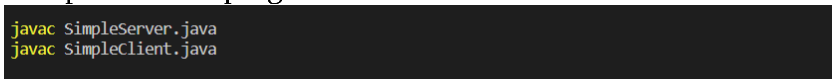
- Create the client program below, save it with the name simpleClient.java:

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

public class SimpleClient {
    public final static int REMOTE_PORT = 5000;
    public static void main(String args[]) throws Exception {
        Socket cl = null;
        BufferedReader is = null;
        DataOutputStream os = null;
        BufferedReader stdin = new BufferedReader(new
        InputStreamReader(System.in));
        String userInput = null;
        String output = null;

        // Membuka koneksi ke server pada port REMOTE_PORT
        try {
            cl = new Socket(args[0], REMOTE_PORT);
            is = new BufferedReader(new
            InputStreamReader(cl.getInputStream()));
            os = new DataOutputStream(cl.getOutputStream());
        } catch(UnknownHostException e1) {
            System.out.println("Unknown Host: " + e1);
        } catch (IOException e2) {
            System.out.println("Erorr io: " + e2);
        }

        // Menulis ke server
        try {
            System.out.print("Masukkan kata kunci: ");
            userInput = stdin.readLine();
            os.writeBytes(userInput + "\n");
        } catch (IOException ex) {
```



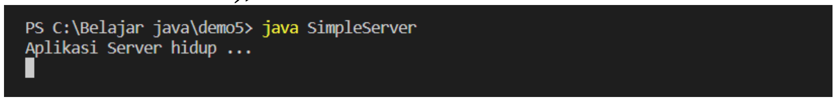
```
            System.out.println("Error writing to server..." + ex);
        }

        // Menerima tanggapan dari server
        try {
            output = is.readLine();
            System.out.println("Dari server: " + output);
        } catch (IOException e) {
            e.printStackTrace();
        }

        // close input stream, output stream dan koneksi
        try {
            is.close();
            os.close();
            cl.close();
        } catch (IOException x) {
            System.out.println("Error writing...." + x);
        }
    }
}
```

- Compile the two programs above with:



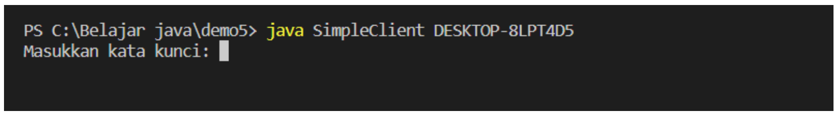
```
javac SimpleServer.java
javac SimpleClient.java
```

- Run both programs on your computer. First run the server, (open console shell window first), and wait for the client connection:



```
PS C:\Belajar java\demo5> java SimpleServer
Aplikasi Server hidup ...
```

- To run the client program, open a new console shell window and type “java SimpleClient <hostname>”



```
PS C:\Belajar java\demo5> java SimpleClient DESKTOP-8LPT4D5
Masukkan kata kunci:
```

- In the client application, enter the requested keyword, namely "salam". Take note what happens next on the server and client applications. Try you to enter other words.
- Perform steps 5 and 6 by running the client and server applications on different computer.

<u>TASK</u>

Develop client server applications on the D1-1 demo so that applications and servers can exchanging messages continuously. Every time the client application sends a message, then the server will reverse the message and send it back to the client. Each the app always displays the messages it receives. Client applications are always asking the user enters a message to be sent to the server. Both applications are complete if the application The client sends an "exit" message. Name the program files comServer.java and comClient.java.

➢ Display on the client side:

```
Aplikasi Client hidup ...
Masukkan pesan: salam
Balasan: malas
Masukkan pesan: hello
Balasan: olleh
Masukkan pesan: exit
Aplikasi Client selesai ...
```

➢ Display on the server side:

```
Aplikasi Server hidup ...
Pesan masuk: salam
Pesan masuk: hello
Pesan masuk: exit
Aplikasi Server selesai ...
```

- **Practicum 5 - R**

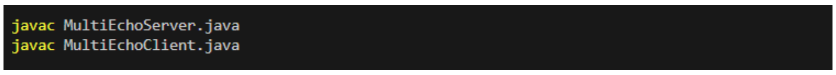
➢ Create the server program below, save it with the name MultiEchoServer.java:

```java
import java.io.*;
import java.net.*;
public class MultiEchoServer{
private static ServerSocket servSock;
private static final int PORT = 1234 ;
    public static void main(String args[]) throws IOException{
    System.out.println("Opening Port.....\n");
        try{
            servSock = new ServerSocket(PORT);
        }catch(IOException e){
            System.out.println("Unable to attach to port");
            System.exit(1);
        }
        do{
            Socket client = servSock.accept();
            ClientHandler handler = new ClientHandler(client);
            handler.start();
        }while(true);
    }
}

class ClientHandler extends Thread{
    private Socket client ;
    private BufferedReader in ;
    private PrintWriter out ;
    public ClientHandler(Socket socket){
    client = socket ;
        try{
            in = new BufferedReader(new
            InputStreamReader(client.getInputStream()));
            out = new PrintWriter(client.getOutputStream(),true);
        }catch(IOException e){
            e.printStackTrace();
        }
    }
        public void run(){
            try{
                String received ;
                do{
                    received = in.readLine();
                    System.out.println(received);
                    out.println("ECHO : " + received);
                }while(!received.equals("QUIT"));
            }catch(IOException e){
                e.printStackTrace();
            }
        finally{
            try{
                if (client != null){
                    System.out.println("Closing down connection");
                    client.close();
                }
            }catch(IOException e){
                e.printStackTrace();
            }
        }
    }
}
```

➢ Create the client program below, save it with the name MultiEchoClient.java:

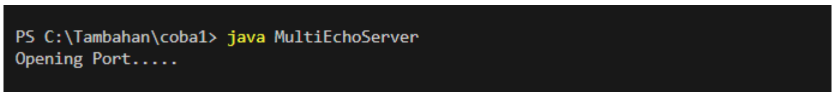
```
import java.io.*;
import java.net.*;
public class MultiEchoClient{
private static InetAddress host ;
private static final int PORT = 1234;
private static Socket link ;
private static BufferedReader in ;
private static PrintWriter out ;
private static BufferedReader keyboard ;
public static void main(String args[]){
    try{
        host = InetAddress.getLocalHost();
        link = new Socket(host,PORT);
        in = new BufferedReader(new InputStreamReader(link.getInputStream()));
        out = new PrintWriter(link.getOutputStream(),true);
        keyboard = new BufferedReader(new InputStreamReader(System.in));
        String message, response;

        do{
            System.out.print("Enter message(QUIT to exit)");
            message = keyboard.readLine();
            out.println(message);
            response = in.readLine();
            System.out.println(response);
        }while(!message.equals("QUIT"));
        }catch(UnknownHostException e){
            System.out.println("Host ID not found!");
    }
    catch(IOException e){
        e.printStackTrace();
    }
        finally{
            try{
                if (link != null){
                    System.out.println("Closing down connection");
                    link.close();
                }
            }
            catch(IOException e){
                e.printStackTrace();
            }
        }
    }
}
```

- Compile the two programs above with:



```
javac MultiEchoServer.java
javac MultiEchoClient.java
```

- Run both programs on your computer. First run the server, (open console shell window first), and wait for the client connection:



```
PS C:\Tambahan\coba1> java MultiEchoServer
Opening Port.....
```