Lab 1: Communication Paradigms. Interprocess communication: **Sockets**

Objectives:

- To understand the concept of sockets:
  - To learn how to send and receive data through sockets
  - To implement network clients and servers
- To implement multithreaded client-server application using sockets

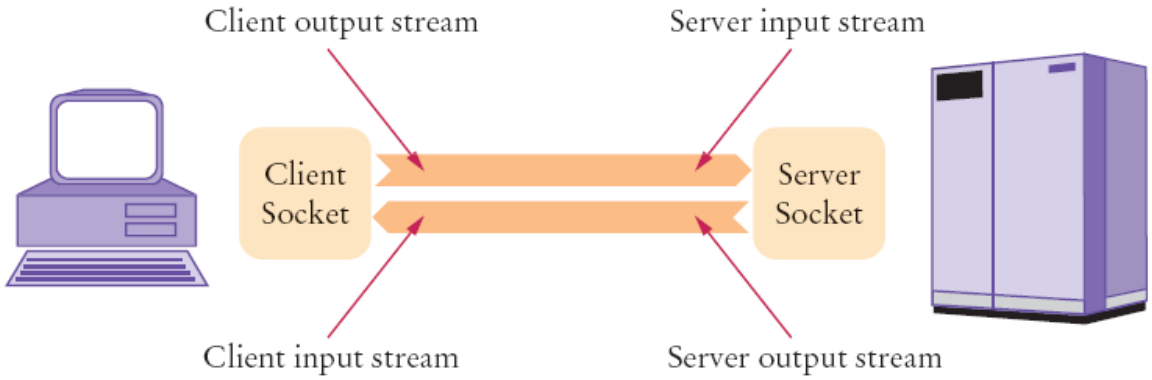
PART 1. Single Socket

Objective

- To understand the concepts of socket programming
- To implement simple client-server applications using sockets
- To explore different socket types and their use cases.

Prerequisites

- Basic understanding of network protocols (TCP/IP, UDP)
- Familiarity with programming concepts in your chosen language (e.g., Python, Java, C++)



Procedure

1. Set up the Environment:

- Ensure that your programming environment is configured correctly with the necessary libraries or modules for socket programming.
- For example, in Python, you might need to install the socket module.

2. Create a Simple Server:

- Create a Python script (or in your preferred language) to establish a server socket.
- Listen for incoming connections on a specified port.
- Accept incoming connections and handle them in a separate thread or process.
- Send and receive data over the socket.

3. Create a Client:

- Create another Python script (or in your preferred language) to establish a client socket.
- Connect to the server on the specified port.
- Send data to the server.
- Receive data from the server.

4. Test the Communication:

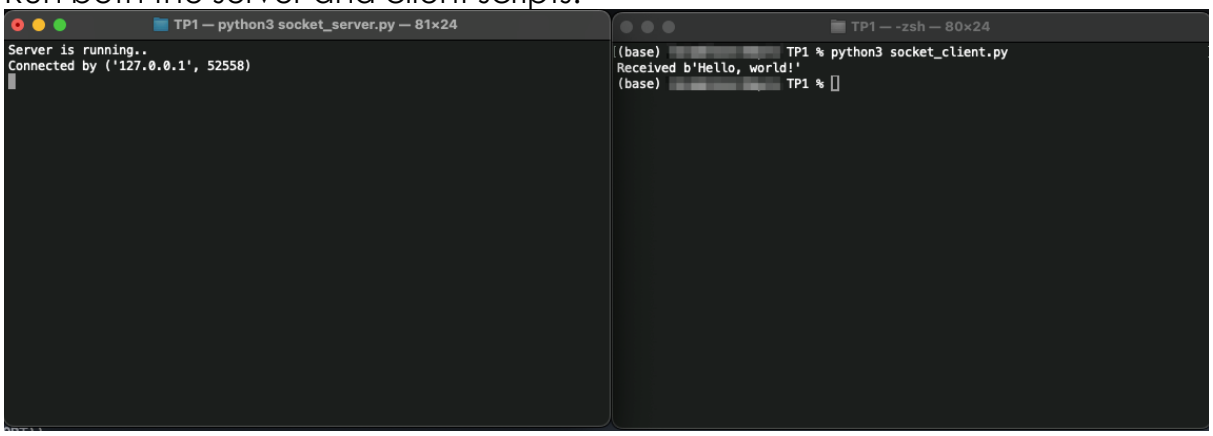
- Run both the server and client scripts.
- Verify that they can communicate successfully.
- Test different scenarios, such as sending and receiving various types of data.

Python Socket Programming Example

```python
# Server
import socket

def handle_client(conn, addr):
    while True:
        data = conn.recv(1024)
        if not data:
            break
        conn.sendall(data)
    conn.close()

if __name__ == "__main__":
    HOST = '127.0.0.1'
    PORT = 65432

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind((HOST, PORT))
        s.listen()
        while True:
            conn, addr = s.accept()
            print('Connected by', addr)
            handle_client(conn, addr)
```

```python
# Client
import socket

if __name__ == "__main__":
    HOST = '127.0.0.1'
    PORT = 65432
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((HOST, PORT))
        s.sendall(b'Hello, world!')
        data = s.recv(1024)
        print('Received', repr(data))
```

Run both the server and client scripts.



Java Socket Programming Example

`Public InputStream getInputStream()`	Returns the InputStream attached with this socket
`Public OutputStream getOutputStream()`	Returns the OutputStream attached with this socket
`Public synchronized void close()`	Closes this socket
`Public Socket accept()`	Returns the socket and establish a connection between client & server
`Public synchronized void close()`	Closes the server socket

Server-side programm:

- The server waits for clients to connect on a certain port (`8080`)
- To listen for incoming connections, use a server socket

- To construct a server socket, provide the port number
  `ServerSocket server = new ServerSocket(8080);`
- Use the accept method to wait for client connection and obtain a socket
  `Socket s = server.accept();`
- Close the connection when the client disconnects

```
package server;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class SocketServer {
   public static void main(String[] args) throws Exception {
      ServerSocket ss = new ServerSocket(8080);
      System.out.println("Server listening on port 8080...");

      while (true) {
         Socket s = ss.accept();
         System.out.println("A new client is connected");

         BufferedReader in = new BufferedReader(new
InputStreamReader(s.getInputStream()));
         PrintWriter out = new PrintWriter(s.getOutputStream(), true);

         String inputLine;
         while ((inputLine = in.readLine()) != null) {
            System.out.println("Server received: " + inputLine);
            out.println("Server response: Hello from the server!");
         }

         s.close();
      }
   }
}
```

Client-side programm:

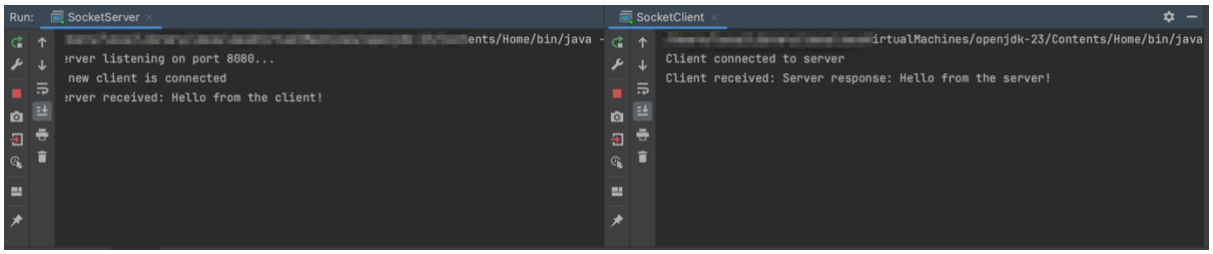
- Create a `Socket` to connect to the server on port `8080`
  ```
  Socket s = new Socket(hostname, portnumber);
  ```
- Code to connect to the HTTP port of server, "localhost"
  ```
  final int HTTP_PORT = 8080;
  Socket s = new Socket("localhost", HTTP_PORT);
  ```
- If it can't find the host, the Socket constructor throws an `UnknownHostException`

- Use the input and output streams attached to the socket to communicate with the other endpoint
- Code to obtain the input and output streams
  ```
  InputStream instream = s.getInputStream();
  OutputStream outstream = s.getOutputStream();
  ```
- When you send data to outstream, the socket forwards them to the server
- The socket catches the server's response and you can read it through instream
- When you are done communicating with the server, close the socket
  ```
  s.close();
  ```

```
package client;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

public class SocketClient {
   public static void main(String[] args) throws Exception {

      final int HTTP_PORT = 8080;
      Socket s = new Socket("localhost", HTTP_PORT);
      System.out.println("Client connected to server");

      BufferedReader in = new BufferedReader(new
InputStreamReader(s.getInputStream()));
      PrintWriter out = new PrintWriter(s.getOutputStream(), true);

      out.println("Hello from the client!");
      String inputLine;
      while ((inputLine = in.readLine()) != null) {
         System.out.println("Client received: " + inputLine);
```

```
        }

        s.close();
    }
}
```

Test the Communication: Run both the server and client applications



Task to do

Implement the advanced features: Extend this basic example of single-socket programming with error hadling and recovery.

PART 2. MultiThreaded Client-Server communication using TCP sockets

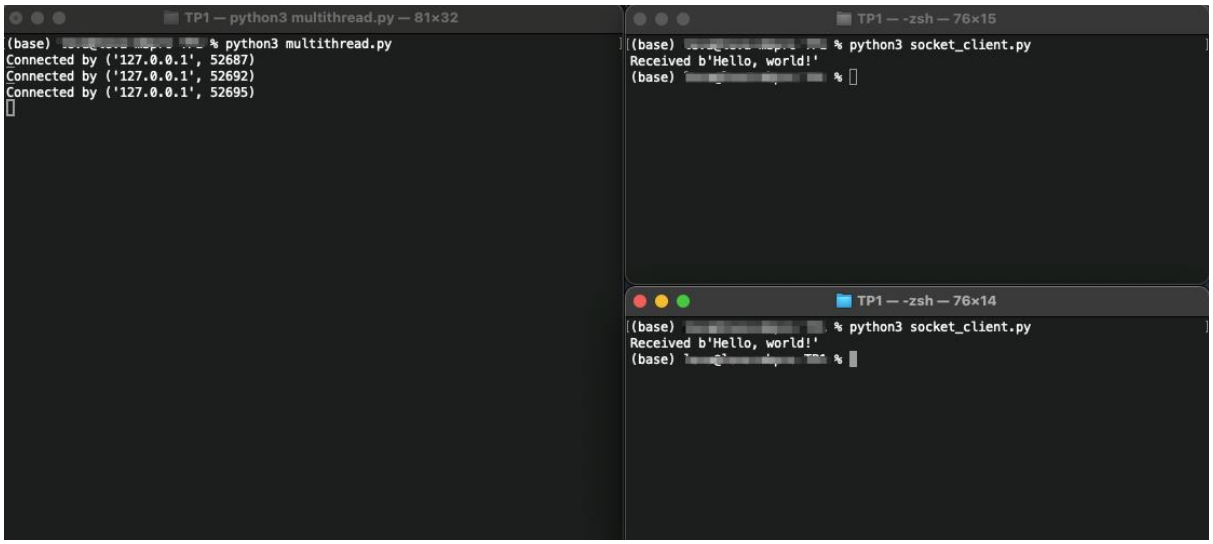
- Multithreading allows a single process to execute multiple tasks concurrently.
- In TCP socket programming, it's often used to handle multiple client connections simultaneously without blocking the main thread.
- This improves responsiveness and scalability.

Modify the code from part 1 to support multi-threading. For each incoming connection, create a separate **thread**. Choose the language and method for the implementation:

- Python methods:
  - `threading.Thread` class in Python https://docs.python.org/3/library/threading.html
  - `concurrent.futures.ThreadPoolExecutor` creates a thread pool executor https://docs.python.org/3/library/concurrent.futures.html

- Java methods:
  - `ExecutorService` - an interface that provides methods for managing a pool of threads https://docs.oracle.com/javase/8/docs/api/index.html?java/util/concurrent/ExecutorService.html
  - `Thread(Runnable target)` constructs a new thread that executes the specified runnable https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

Run the server and multiple clients o test the code.



Part 3(Bonus). Multi-threaded group chat

Implementing group chat using low-level sockets

1 - Implement a basic client application for a chat. Use and *complete* the sample code in the attached file "`mychat.java`":

- Creates a GUI for a chat client.
- Allows users to enter a username (login) and connect to a server on `localhost` port `5555`.
- Users can type messages in the text field and send them to the server upon clicking the "`Send`" button.

2 - Creating the chat server that receives messages from clients and sends them (broadcasts) to all connected clients

Deliverables for the lab

Submit to Moodle:

- Report with screenshots of the output and an explanation of each part (pdf)
- The source code (zip)


isep
École d'ingénieurs du numérique