FR. CONCEICAO RODRIGUES COLLEGE OF ENGINEERING

Department of Computer Engineering

Course, Subject & Experiment Details

Practical No:	1
Title:	Multithreaded Server and Clients
Name of the Student:	Warren Fernandes
Roll No:	8940
Date of Performance:	27/01/2023
Date of Submission:	03/02/2023

Evaluation:

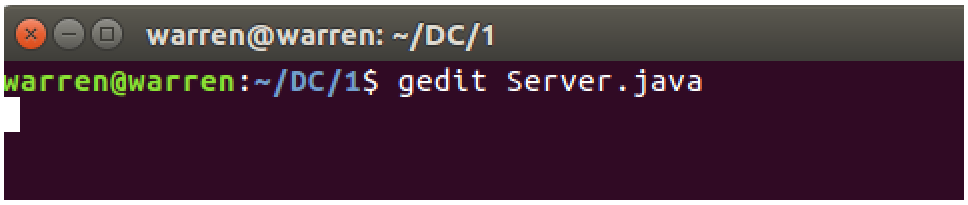
Sr. No.	Rubric	Grade
1	Timeliness (1)	
2	Documentation (2)	
3	Preparation (3)	
4	Performance (4)	

Signature of the Teacher

Aim: The aim of a practical on multi-threaded server and clients is to provide hands-on experience to learners in developing and implementing a client-server architecture that utilizes multiple threads to handle concurrent connections. Through this practical, learners will gain a deeper understanding of the principles of multi-threading and how it can be applied to improve the performance and scalability of networked applications. Additionally, learners will gain experience in writing code for both the server and client-side of a multi-threaded application, and in troubleshooting common issues that can arise in such systems. The practical will enable learners to design, implement, and test a multi-threaded client-server architecture, and to evaluate its performance and scalability under varying workloads. Ultimately, the practical will prepare learners to apply their knowledge of multi-threading in real-world scenarios, where efficient and responsive networked applications are critical.

Steps:

We can use gedit text editor to create a file named Server.java for Java programming



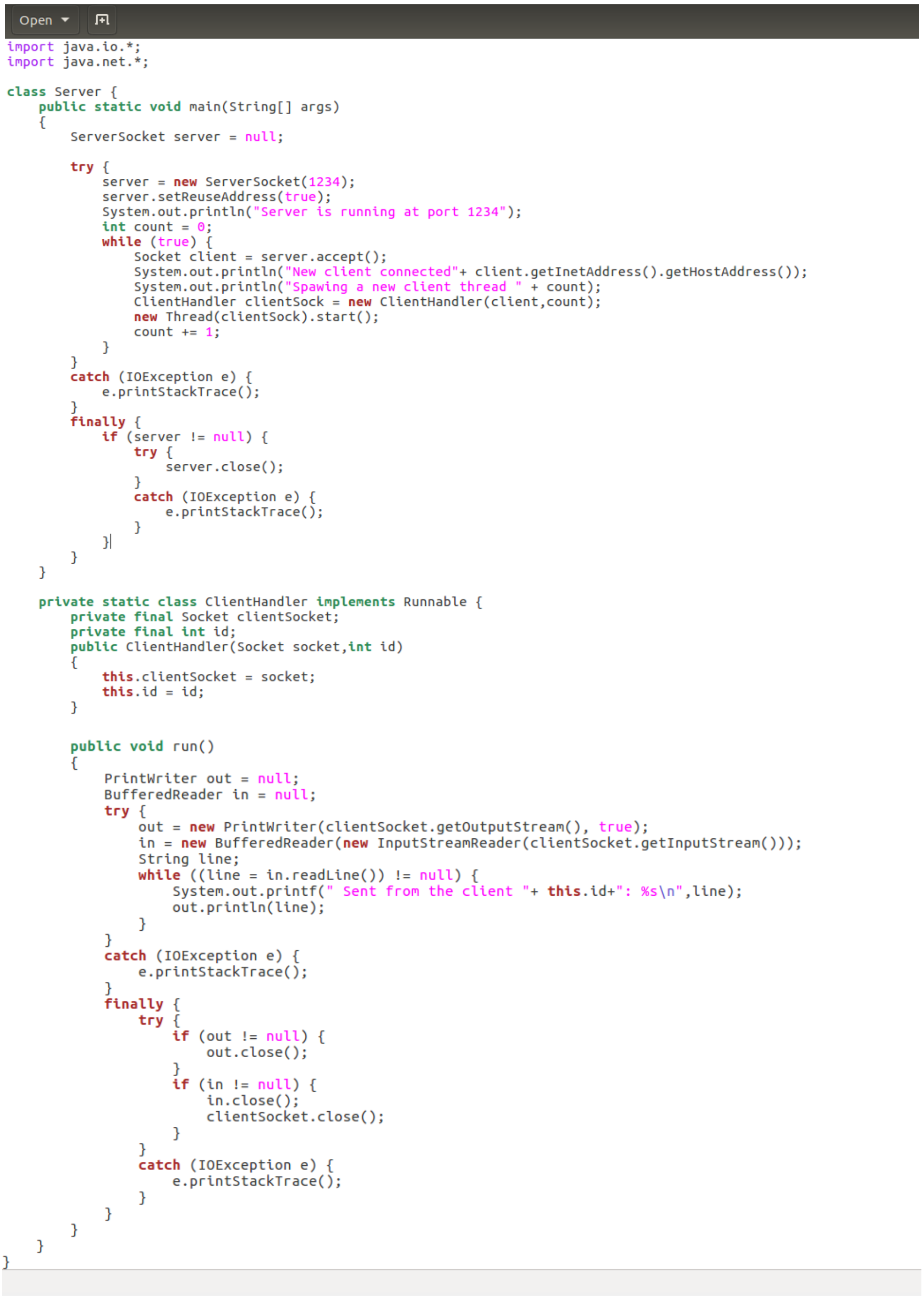
```
warren@warren: ~/DC/1
warren@warren:~/DC/1$ gedit Server.java
```

Server.java

A Java program that uses ClientHandler and ServerSocket can create a multi-threaded server that is capable of handling multiple client connections simultaneously. The ServerSocket class provides a way for the server to listen for incoming client connections, while the ClientHandler class is responsible for processing incoming client requests and sending responses back to the client.

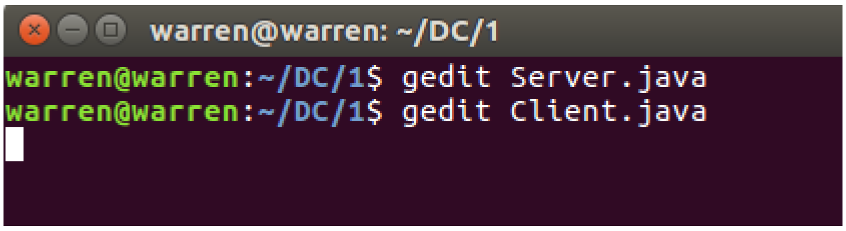
To create a multi-threaded server, the program typically uses a loop to continuously listen for incoming client connections using the ServerSocket. Once a connection is established, the program creates a new thread to handle the client request using a separate instance of the ClientHandler class. This allows the server to process multiple client requests concurrently, without blocking any other clients from connecting or receiving a response.

The use of multi-threading can greatly improve the performance and scalability of the server, enabling it to handle a larger number of client connections and requests with minimal delay or downtime. However, it is important to ensure that the program is designed and implemented correctly to avoid potential issues such as thread synchronization problems or resource contention. Proper error handling and exception handling are also critical to ensure the stability and reliability of the server program.



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

class Server {
    public static void main(String[] args)
    {
        ServerSocket server = null;

        try {
            server = new ServerSocket(1234);
            server.setReuseAddress(true);
            System.out.println("Server is running at port 1234");
            int count = 0;
            while (true) {
                Socket client = server.accept();
                System.out.println("New client connected"+ client.getInetAddress().getHostAddress());
                System.out.println("Spawing a new client thread " + count);
                ClientHandler clientSock = new ClientHandler(client,count);
                new Thread(clientSock).start();
                count += 1;
            }
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        finally {
            if (server != null) {
                try {
                    server.close();
                }
                catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    private static class ClientHandler implements Runnable {
        private final Socket clientSocket;
        private final int id;
        public ClientHandler(Socket socket,int id)
        {
            this.clientSocket = socket;
            this.id = id;
        }

        public void run()
        {
            PrintWriter out = null;
            BufferedReader in = null;
            try {
                out = new PrintWriter(clientSocket.getOutputStream(), true);
                in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
                String line;
                while ((line = in.readLine()) != null) {
                    System.out.printf(" Sent from the client "+ this.id+": %s\n",line);
                    out.println(line);
                }
            }
            catch (IOException e) {
                e.printStackTrace();
            }
            finally {
                try {
                    if (out != null) {
                        out.close();
                    }
                    if (in != null) {
                        in.close();
                        clientSocket.close();
                    }
                }
                catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

We can use gedit text editor to create a file named Client.java for Java programming



```
warren@warren:~/DC/1$ gedit Server.java
warren@warren:~/DC/1$ gedit Client.java
```

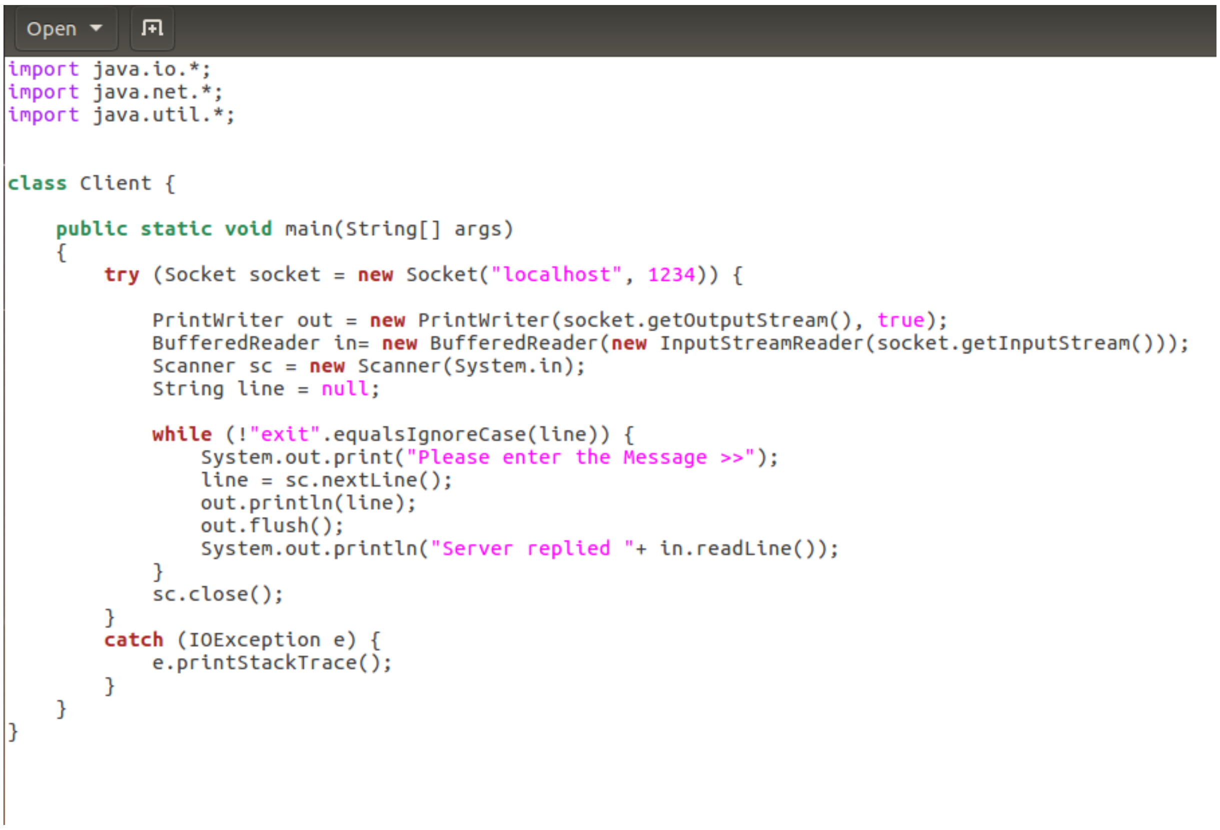
Client.java

A Java program that uses Socket can create a client that can connect to a server over a network. The Socket class provides a way for the client to establish a connection to the server using the server's IP address and port number.

To create a client, the program typically creates a new instance of the Socket class, passing the server's IP address and port number as arguments. Once the connection is established, the program can then use the Socket's input and output streams to send and receive data to and from the server.

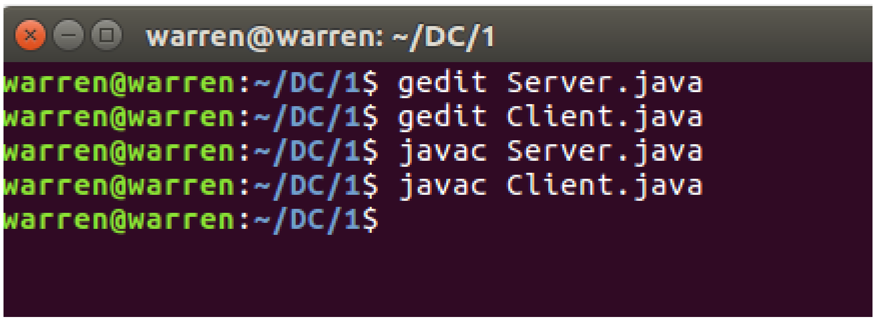
The use of Socket enables the client to communicate with the server over a network, enabling a wide range of applications such as chat programs, file transfer applications, and more. However, it is important to ensure that the program is designed and implemented correctly to avoid potential issues such as connection failures, data corruption, or security vulnerabilities.

Proper error handling, exception handling, and network security measures are critical to ensure the stability, reliability, and security of the client program. Additionally, the use of multi-threading in the client program can further enhance its performance and responsiveness, enabling it to handle multiple tasks and network connections simultaneously.



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


class Client {

    public static void main(String[] args)
    {
        try (Socket socket = new Socket("localhost", 1234)) {

            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
            BufferedReader in= new BufferedReader(new InputStreamReader(socket.getInputStream()));
            Scanner sc = new Scanner(System.in);
            String line = null;

            while (!"exit".equalsIgnoreCase(line)) {
                System.out.print("Please enter the Message >>");
                line = sc.nextLine();
                out.println(line);
                out.flush();
                System.out.println("Server replied "+ in.readLine());
            }
            sc.close();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

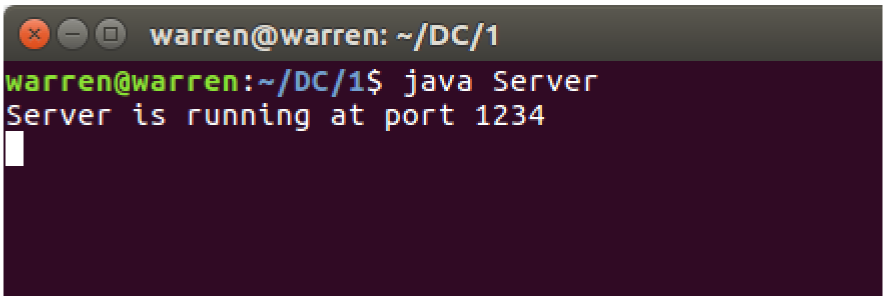
You can use javac to compile both the server and client Java files, which will generate bytecode files that can be executed using the Java Virtual Machine (JVM).



```
warren@warren: ~/DC/1
warren@warren:~/DC/1$ gedit Server.java
warren@warren:~/DC/1$ gedit Client.java
warren@warren:~/DC/1$ javac Server.java
warren@warren:~/DC/1$ javac Client.java
warren@warren:~/DC/1$
```

To run the server on port 1234, you can execute the compiled Server.class file using the following command in the terminal or command prompt:

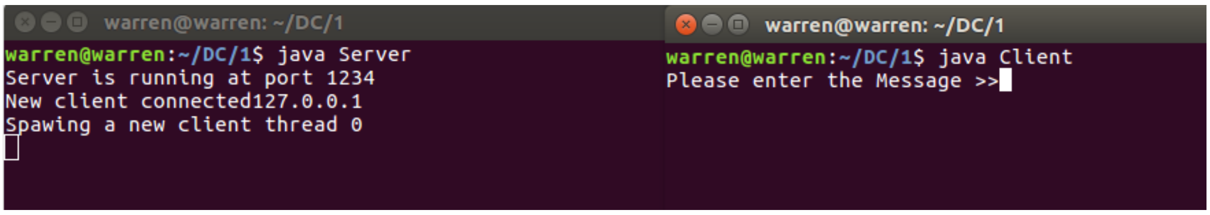
This will start the server program and bind it to port 1234, allowing it to listen for incoming client connections on that port.



```
warren@warren: ~/DC/1
warren@warren:~/DC/1$ java Server
Server is running at port 1234
```

To run the client and automatically bind to the server running on port 1234, you can execute the compiled Client.class file using the following command in the terminal or command prompt:

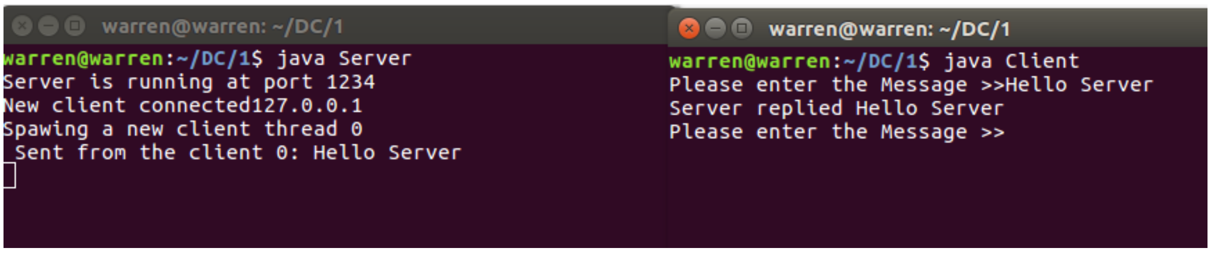
This will start the client program and attempt to establish a connection with the server running on the same machine (localhost) and port 1234. Once the connection is established, the client program can send and receive data to and from the server.



```
warren@warren: ~/DC/1
warren@warren:~/DC/1$ java Server
Server is running at port 1234
New client connected127.0.0.1
Spawing a new client thread 0
```

```
warren@warren: ~/DC/1
warren@warren:~/DC/1$ java Client
Please enter the Message >>
```

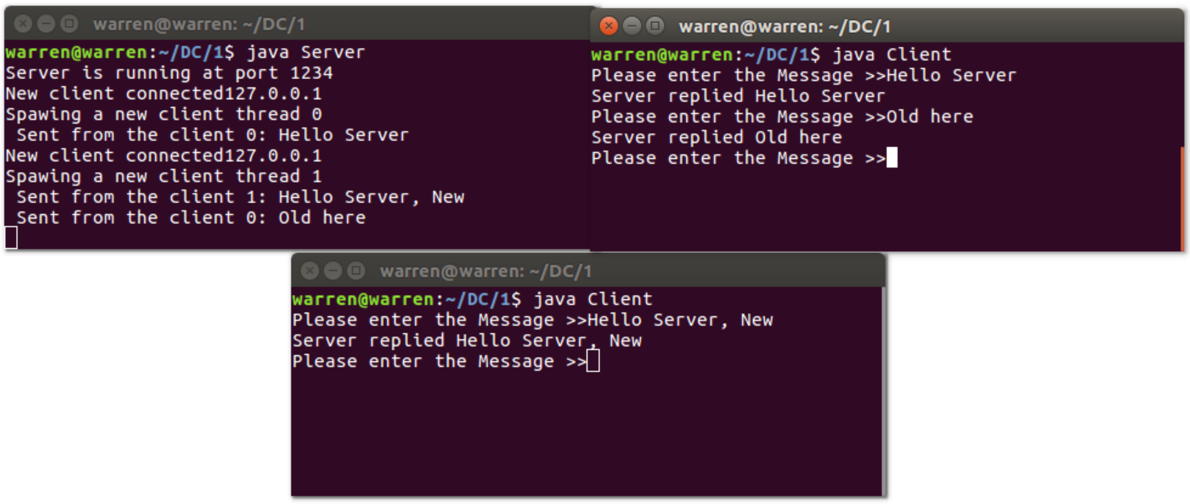
Now we will send a message from the client to the Server



```
warren@warren: ~/DC/1
warren@warren:~/DC/1$ java Server
Server is running at port 1234
New client connected127.0.0.1
Spawing a new client thread 0
 Sent from the client 0: Hello Server
```

```
warren@warren: ~/DC/1
warren@warren:~/DC/1$ java Client
Please enter the Message >>Hello Server
Server replied Hello Server
Please enter the Message >>
```

Once a new client is connected to the server, the server can create a new thread to handle its input and output streams independently of the other clients. This allows multiple clients to communicate with the server simultaneously without blocking each other. Each thread would handle its own input and output streams, and would be responsible for receiving messages from its client and broadcasting them to all connected clients, including the sender. The clients can also receive messages from the server through their own input streams, and can send messages to the server through their output streams.



```
warren@warren:~/DC/1$ java Server
Server is running at port 1234
New client connected127.0.0.1
Spawing a new client thread 0
 Sent from the client 0: Hello Server
New client connected127.0.0.1
Spawing a new client thread 1
 Sent from the client 1: Hello Server, New
 Sent from the client 0: Old here
```

```
warren@warren:~/DC/1$ java Client
Please enter the Message >>Hello Server
Server replied Hello Server
Please enter the Message >>Old here
Server replied Old here
Please enter the Message >>
```

```
warren@warren:~/DC/1$ java Client
Please enter the Message >>Hello Server, New
Server replied Hello Server, New
Please enter the Message >>
```

Conclusion

In conclusion, working with multi-threaded server and clients in Java is a valuable and practical experiment for learning about network programming and concurrency. Through this experiment, we can gain hands-on experience in creating networked applications that can handle multiple connections and requests simultaneously. We can also learn how to use Java's built-in concurrency features, such as threads and synchronization, to ensure that our application is safe and responsive. By using sockets and input/output streams, we can communicate over the network in a standardized and platform-independent way. Overall, the multi-threaded server and clients experiment provides an excellent opportunity to develop practical skills and gain a deeper understanding of network programming and concurrency in Java.