Multi-threaded Server Model

FSAD/SWW2 Week 11

Ana Stroescu

Contents

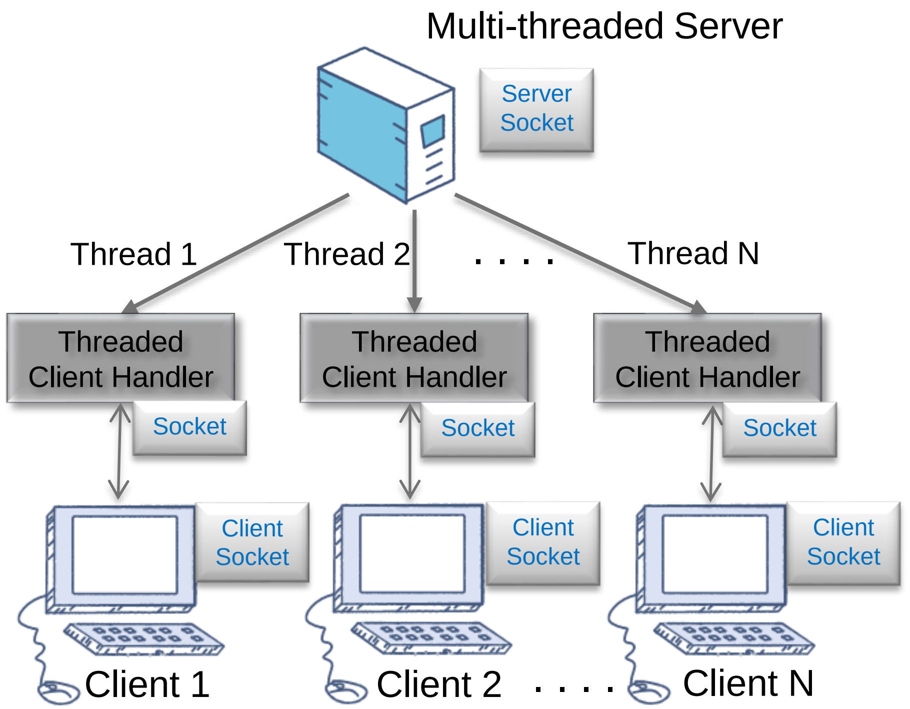
UNIVERSITY OF BIRMINGHAM

Multi-threaded server architecture

- A server must have the capacity to service many clients and many requests at the same time.

- The way to do this is to create a new socket for every new client and service that client's requests on a different thread.

- A Client Handler is needed for handling clients using multithreading.



Multi-threaded server

- Advantages:
  - It can respond fast and efficiently to the client queries.
  - A new thread is generated for each client, hence threads are independent of each other.
  - If an error occurs in a thread, the other threads are not affected so other client processes keep running normally.
  - The waiting time for user decreases since the requests are handled in parallel.
  - The same client could disconnect and reconnect again, without getting a connection refused exception or a connection reset on the server.

- Drawbacks:
  - Difficulty level in writing a program.
  - Complex debugging and testing.

Threads in Java – recap

- There are two methods for creating threads in Java:

1. By extending the `Thread` class

```
public class threadExample extends Thread {
   public static void main(String[] args) {
      threadExample thread = new threadExample();
      thread.start();
   }
   public void run() {
      System.out.println("This code is running in a thread");
   }
}
```

Create an instance of the class and call the `start()` method

2. By implementing the `Runnable` interface

```
public class threadExample implements Runnable {
   public static void main(String[] args) {
      threadExample obj = new threadExample();
      Thread thread = new Thread(obj);
      thread.start();
   }
   public void run() {
      System.out.println("This code is running in a thread");
   }
}
```

Create an instance of the class, pass the instance to the Thread's constructor and call the `start()` method

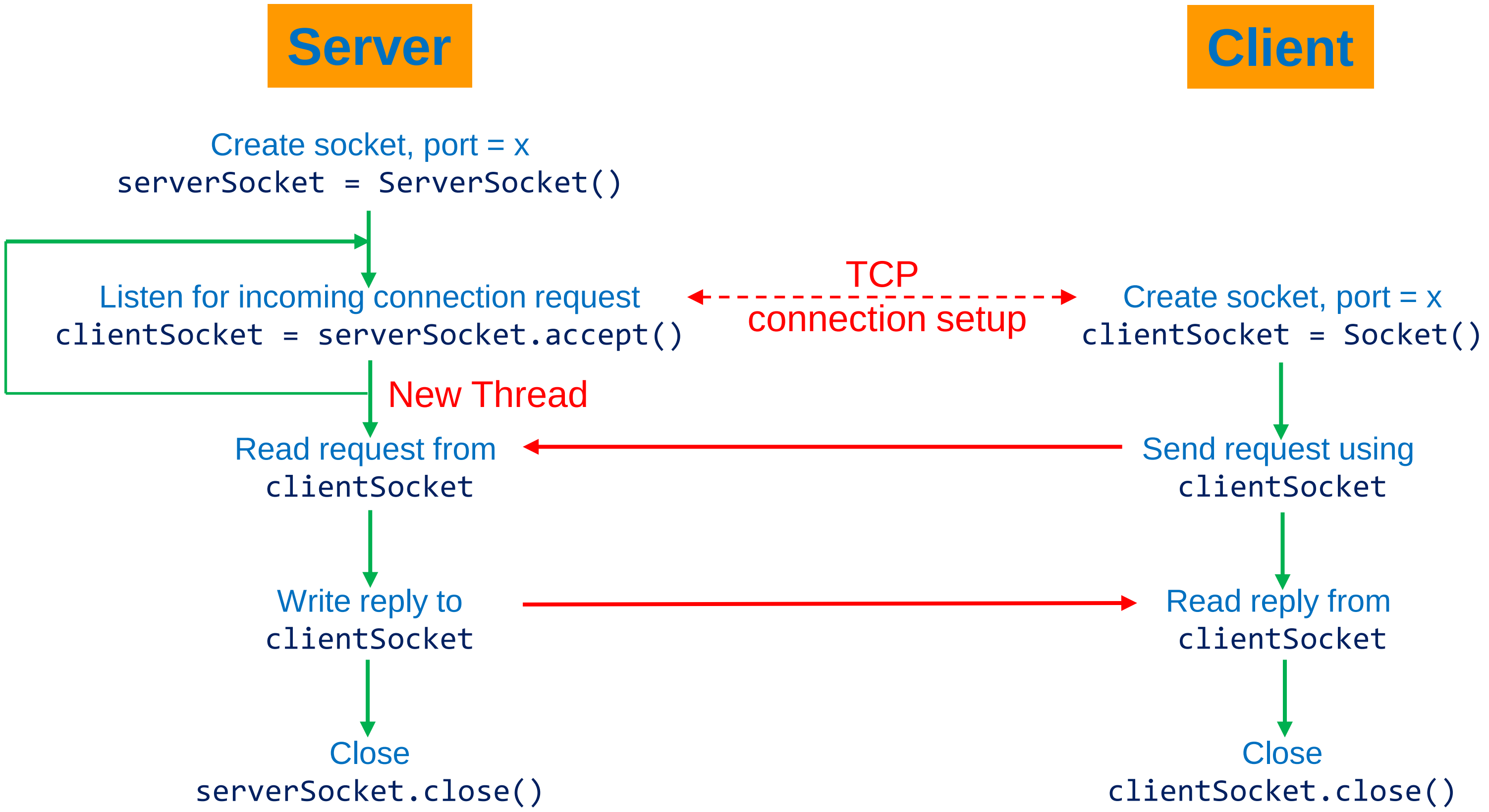
Multi-threaded server in Java

The basic flow of logic of a multi-threaded server is:

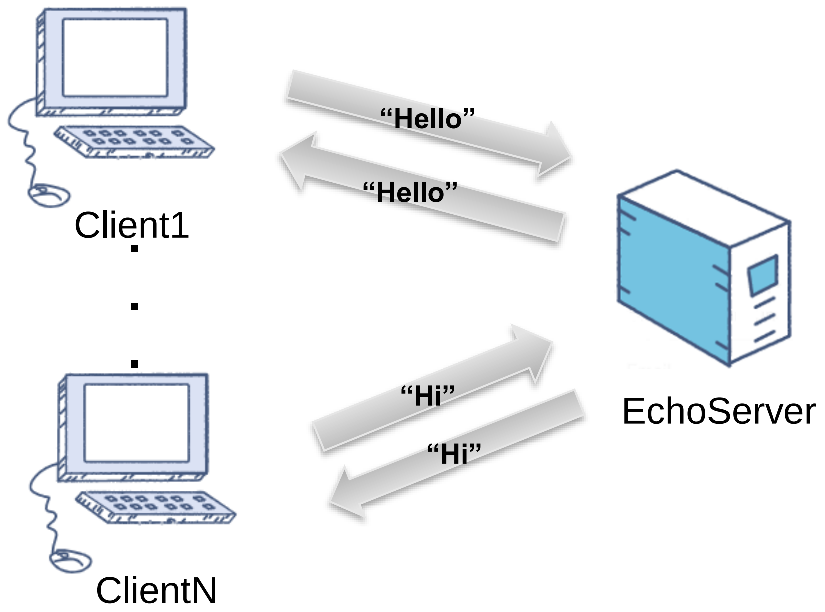
```
while (true)
{ //accept a client connection;
  //create a new thread for the client;
}
```

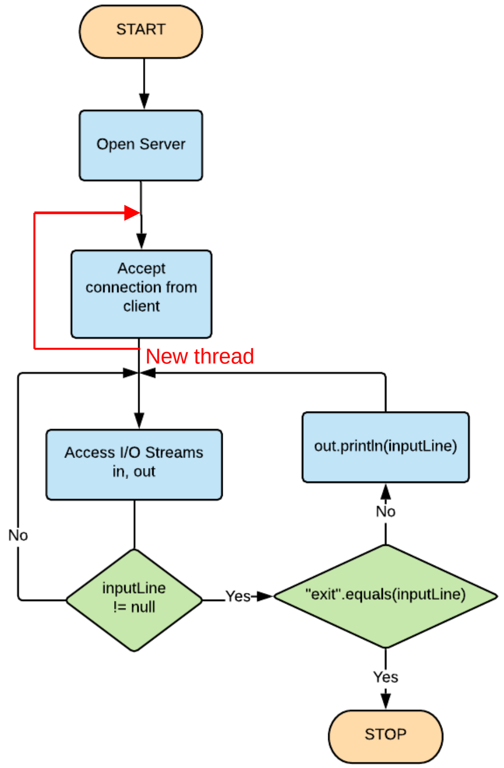
- The main thread is running a while loop as it listens for new connections.
- An additional class is needed to handle the client threads. This is called a Client Handler, which implements the `Runnable` interface and takes care of multiple connections from clients.

TCP Multi Client-Server socket interaction



Application: Echo Multi-Server





Server Side

- The server is designed so that for each service request submitted to a main controller/communications thread, a separate service thread is created to process that request and communicate the results back to the client.
- The server socket can have many connections. Each iteration of the `while` loop creates a new connection.
- The Server class opens a new server socket and continuously listens for client connections.
- Whenever a new client connects, the server creates a new handler for the client and goes back to listening.
- Previous steps 4-7 are now executed in the `run()` method of the `EchoClientHandler` class.

EchoMultiServer.java

1. Import

2. Open the server socket

3. Infinite loop for client requests

4. Accept a connection

5. Create a new EchoClientHandler obejct

6. Start the execution of the thread

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

class EchoMultiServer {

    public static void main(String args[]) throws IOException {

        ServerSocket serverSocket = new ServerSocket(80);
        System.out.println("Server is running" );
        int counter = 0;

        while (true) {
           Socket clientSocket = serverSocket.accept();
            counter ++;
            System.out.println("Client " + counter + " connected with IP " + clientSocket.getInetAddress().getHostAddress());

            EchoClientHandler clientHandler = new EchoClientHandler(clientSocket, counter);

            new Thread(clientHandler).start();
        }
    }
}
```

Client Handler

- This class implements Runnable interface so that each object acts as a Runnable target for a new thread.
- The constructor takes a Socket parameter which uniquely identifies an incoming client request.
- Some of the functionality of the server is now implemented in the `run()` method of this class: open I/O streams, read the client message, reply to the client.

EchoClientHandler.java

1. Import

2: I/O Streams

3: Read message from client

4: Send message to client

5: Close I/O streams and socket

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

public class EchoClientHandler implements Runnable {
   Socket clientSocket;
   int clientNo;

   public EchoClientHandler (Socket socket, int counter) {
      clientSocket = socket;
      clientNo = counter;
   }
   public void run() {
      try {

          BufferedReader inFromClient = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
          PrintWriter outToClient = new PrintWriter(clientSocket.getOutputStream(), true);
          String clientMessage;

          while(!(clientMessage = inFromClient.readLine()).equals("exit"))
             outToClient.println(clientMessage);

          System.out.println("Client " + clientNo + " has disconnected");
          outToClient.println("Connection closed, Goodbye!");

          inFromClient.close();
          outToClient.close();
          clientSocket.close();

      } catch (IOException e) {
          e.printStackTrace();
      }
   }
}
```

Client Side

- The Client class remains unchanged.
- We can distinguish the same basic 7 steps as in our previous applications.

EchoMultiClient.java

1: Import

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

public class EchoMultiClient {
   public static void main(String args[]) throws IOException {
      String message, serverMessage;

      Socket clientSocket = new Socket("127.0.0.1", 80);
      System.out.println("Client is running");

      PrintWriter outToServer = new PrintWriter(clientSocket.getOutputStream(), true);
      BufferedReader inFromServer = new BufferedReader (new InputStreamReader(clientSocket.getInputStream()));
      BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));

      while (true) {
         System.out.println("CLIENT MESSAGE: ");
         message = inFromUser.readLine();

         outToServer.println(message);

         serverMessage = inFromServer.readLine();
         System.out.println("SERVER MESSAGE: " + serverMessage);
         if (message.equals("exit"))
            break;
      }

      inFromServer.close();
      inFromUser.close();
      outToServer.close();
      clientSocket.close();
   }
}
```

2: Open the client socket

Connect to the server on localhost IP address and port 80

3: I/O Streams

4: Continuously read messages from user

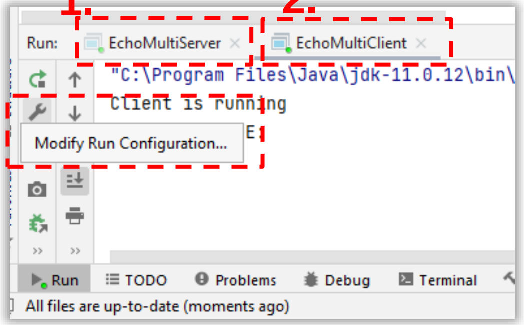
5: Send message to the server

6: Read server response

7: Close I/O streams and socket

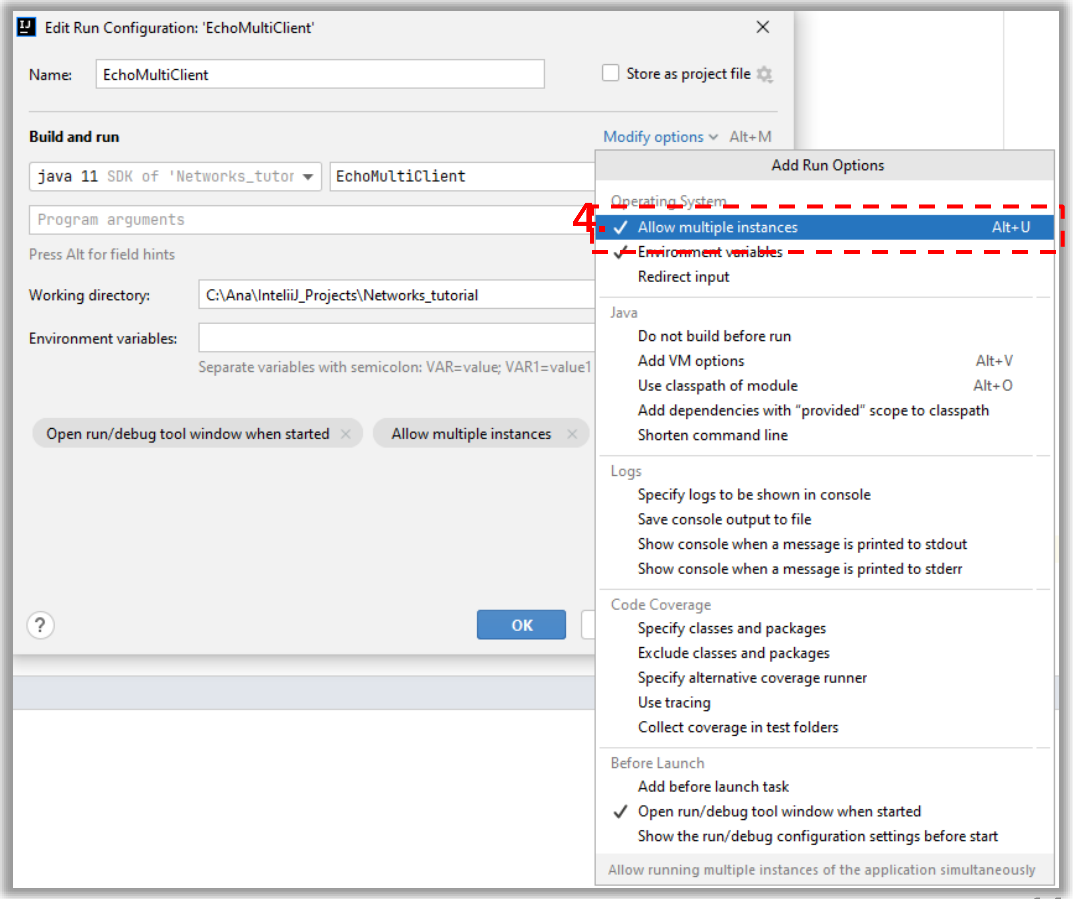
How to open multiple connections in IntelliJ

1. Open the server
2. Open a client connection
3. Modify Run Configuration
4. Modify Options | Allow multiple instances
5. Now you can open multiple client connections



```
"C:\Program Files\Java\jdk-11.0.12\bin\
Client is running
```

All files are up-to-date (moments ago)



Edit Run Configuration: 'EchoMultiClient'

Name: EchoMultiClient ☐ Store as project file

Build and run

java 11 SDK of 'Networks_tutor | EchoMultiClient

Program arguments

Press Alt for field hints

Working directory: C:\Ana\IntelliJ_Projects\Networks_tutorial

Environment variables:

Separate variables with semicolon: VAR=value; VAR1=value1

Open run/debug tool window when started × | Allow multiple instances ×

Operating System
- ✓ Allow multiple instances Alt+U
- Environment variables
- Redirect input

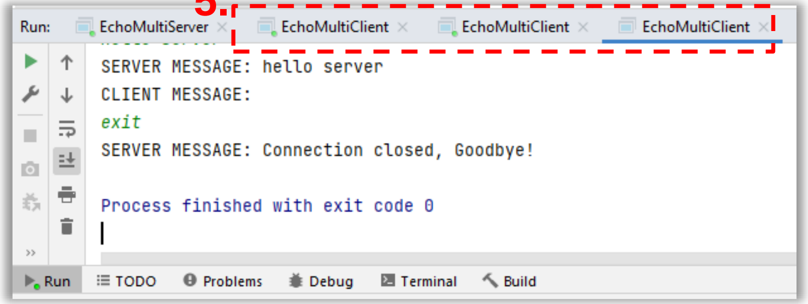
Java
- Do not build before run
- Add VM options Alt+V
- Use classpath of module Alt+O
- Add dependencies with "provided" scope to classpath
- Shorten command line

Logs
- Specify logs to be shown in console
- Save console output to file
- Show console when a message is printed to stdout
- Show console when a message is printed to stderr

Code Coverage
- Specify classes and packages
- Exclude classes and packages
- Specify alternative coverage runner
- Use tracing
- Collect coverage in test folders

Before Launch
- Add before launch task
- ✓ Open run/debug tool window when started
- Show the run/debug configuration settings before start

Allow running multiple instances of the application simultaneously



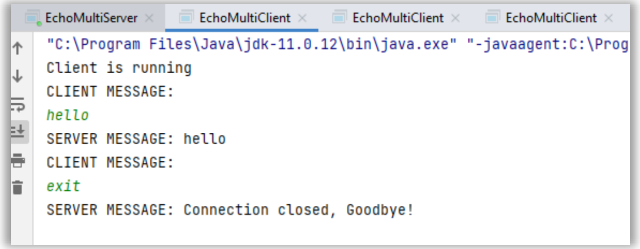
```
SERVER MESSAGE: hello server
CLIENT MESSAGE:
exit
SERVER MESSAGE: Connection closed, Goodbye!

Process finished with exit code 0
```

Echo Multi-Server - sample output

Console output for **`EchoMultiClient.java`**

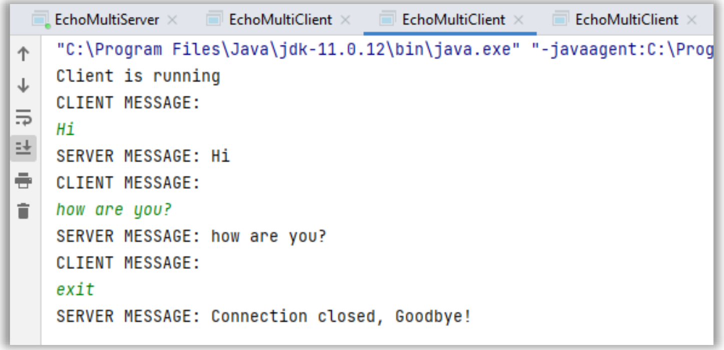
Client 1



```
"C:\Program Files\Java\jdk-11.0.12\bin\java.exe" "-javaagent:C:\Prog
Client is running
CLIENT MESSAGE:
hello
SERVER MESSAGE: hello
CLIENT MESSAGE:
exit
SERVER MESSAGE: Connection closed, Goodbye!
```

Console output for **`EchoMultiClient.java`**

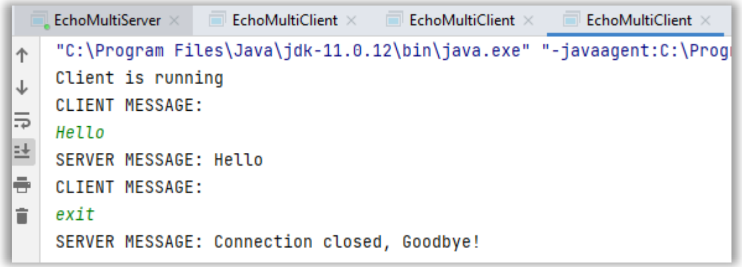
Client 2



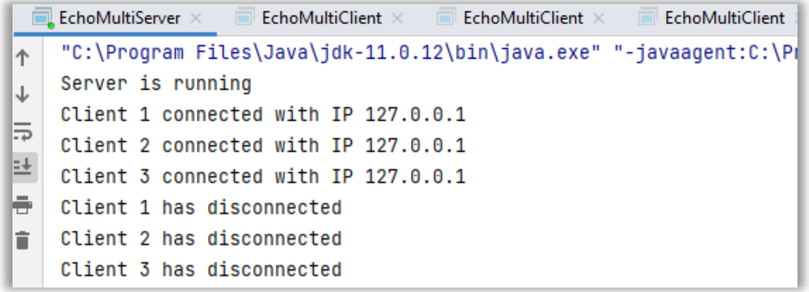
```
"C:\Program Files\Java\jdk-11.0.12\bin\java.exe" "-javaagent:C:\Prog
Client is running
CLIENT MESSAGE:
Hi
SERVER MESSAGE: Hi
CLIENT MESSAGE:
how are you?
SERVER MESSAGE: how are you?
CLIENT MESSAGE:
exit
SERVER MESSAGE: Connection closed, Goodbye!
```

Console output for **`EchoMultiClient.java`**

Client 3



```
"C:\Program Files\Java\jdk-11.0.12\bin\java.exe" "-javaagent:C:\Prog
Client is running
CLIENT MESSAGE:
Hello
SERVER MESSAGE: Hello
CLIENT MESSAGE:
exit
SERVER MESSAGE: Connection closed, Goodbye!
```

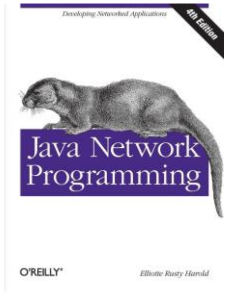


```
"C:\Program Files\Java\jdk-11.0.12\bin\java.exe" "-javaagent:C:\Pr
Server is running
Client 1 connected with IP 127.0.0.1
Client 2 connected with IP 127.0.0.1
Client 3 connected with IP 127.0.0.1
Client 1 has disconnected
Client 2 has disconnected
Client 3 has disconnected
```

Console output for **`EchoMultiServer.java`**

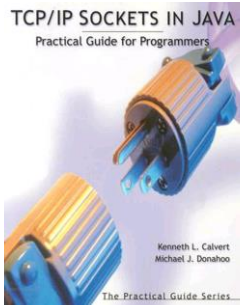
Additional reading

- **Java Network Programming,** by Elliotte Rusty Harold, O'Reilly Media, 4th edition



E-book

- **TCP/IP Sockets in Java: practical guide for programmers,** by Kenneth L. Calvert and Michael J. Donahoo – **Chapter 4**



E-book