Java Socket Programming

Single-Threaded, Multi-Threaded & Thread Pool Servers

---

1. Introduction

Socket programming enables communication between two computers (or processes) using a network. In Java, this is achieved using the `java.net` package.

A **client-server model** consists of:

- **Client**: Initiates the request
- **Server**: Listens on a port and responds to requests

---

2. Core Networking Concepts

2.1 Socket

A **Socket** represents one endpoint of a two-way communication channel.

- Client socket → connects to server
- Server socket → listens for connections

2.2 Port

A **port** uniquely identifies a service on a machine.

- Example: `8010`
- Multiple clients can connect to the **same port**

2.3 Blocking I/O

Methods like:

- `ServerSocket.accept()`

- `BufferedReader.readLine()`

are **blocking**, meaning the thread waits until data is available.

---

3. Single-Threaded Server

3.1 Concept

A **single-threaded server** handles **one client at a time**.

- Only one execution flow
- Other clients must wait
- Simple but inefficient

3.2 Execution Flow

1. Server starts and listens on a port
2. Client connects
3. Server processes request
4. Server closes connection
5. Next client is accepted

3.3 Single-Threaded Client Code

```
package SingleThreaded;

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

public class Client {

    public void run() throws Exception {
        int port = 8010;
        InetAddress ip = InetAddress.getLocalHost();
        Socket socket = new Socket(ip, port);

        PrintWriter toSocket = new PrintWriter(socket.getOutputStream(),
        BufferedReader fromSocket = new BufferedReader(
                new InputStreamReader(socket.getInputStream())
        );

        toSocket.println("Hello from the client");
        String response = fromSocket.readLine();
        System.out.println("Response from the socket is: " + response);

        socket.close();
    }

    public static void main(String[] args) {
        try {
            new Client().run();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

3.4 Single-Threaded Server Code

```
package SingleThreaded;

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

public class Server {

    public void run() {
        int port = 8010;
        try {
            ServerSocket serverSocket = new ServerSocket(port);
            serverSocket.setSoTimeout(10000);

            while (true) {
                System.out.println("Server listening on port " + port);
                Socket socket = serverSocket.accept();
                PrintWriter toClient = new PrintWriter(socket.getOutputSt
                BufferedReader fromClient = new BufferedReader(
                        new InputStreamReader(socket.getInputStream())
                );

                String msg = fromClient.readLine();
                System.out.println("Client says: " + msg);

                toClient.println("hello from my server!");
                socket.close();
            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    public static void main(String[] args) {
        new Server().run();
    }
}
```

3.5 Limitations

- Poor scalability
- Blocking behavior
- Not suitable for real-world servers

4. Multi-Threaded Server (Thread per Client)

4.1 Concept

Each client request is handled by a **new thread**.

- Clients run in parallel
- Better responsiveness
- High overhead if many clients connect

4.2 Execution Flow

1. Server accepts connection
2. New thread created
3. Thread handles client
4. Main thread continues listening

4.3 Multi-Threaded Client Code

```
package MultiThreaded;

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

public class Client {

    public Runnable getRunnable() {
        return () -> {
            try {
                Socket socket = new Socket(InetAddress.getLocalHost(), 8(
                PrintWriter toSocket = new PrintWriter(socket.getOutputSt
                BufferedReader fromSocket = new BufferedReader(
                        new InputStreamReader(socket.getInputStream())
                );

                toSocket.println("Hello from the client");
                System.out.println(fromSocket.readLine());
                socket.close();
            } catch (Exception e) {
                System.out.println(e.getMessage());
            }
        };
    }

    public static void main(String[] args) {
        Client client = new Client();
        for (int i = 0; i < 100; i++) {
            new Thread(client.getRunnable()).start();
        }
    }
}
```

4.4 Multi-Threaded Server Code

```
package MultiThreaded;

import java.io.IOException;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.function.Consumer;

public class Server {

    public Consumer<Socket> getConsumer() {
        return socket -> {
            try {
                PrintWriter toClient = new PrintWriter(socket.getOutputSt
                toClient.println("Hello from the server");
                socket.close();
            } catch (IOException e) {
                System.out.println(e.getMessage());
            }
        };
    }

    public static void main(String[] args) {
        Server server = new Server();
        try {
            ServerSocket serverSocket = new ServerSocket(8010);
            while (true) {
                Socket socket = serverSocket.accept();
                new Thread(() -> server.getConsumer().accept(socket)).sta
            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

4.5 Drawbacks

- Too many threads = memory exhaustion
- Heavy context switching
- No control over thread count

---

5. Thread Pool Server (Best Practice)

5.1 Concept

Uses a **fixed number of reusable threads** managed by `ExecutorService`.

- Efficient
- Scalable
- Production-ready

5.2 Why Thread Pool?

- Threads are expensive
- Reuse improves performance
- Prevents server crash due to overload

5.3 Thread Pool Server Code

```
package ThreadPool;

import java.io.IOException;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Server {

    private final ExecutorService threadPool;

    public Server(int poolSize) {
        this.threadPool = Executors.newFixedThreadPool(poolSize);
    }
    public void handleClient(Socket socket) {
        try (PrintWriter out = new PrintWriter(socket.getOutputStream(),
            out.println("Hello from server " + socket.getInetAddress());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        Server server = new Server(10);

        try {
            ServerSocket serverSocket = new ServerSocket(8010);
            while (true) {
                Socket client = serverSocket.accept();
                server.threadPool.execute(() -> server.handleClient(clier
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

---

6. Comparison Table

Model	Concurrency	Scalability	Resource Control
Single-Threaded	❌ No	❌ Poor	✅ Simple
Multi-Threaded	✅ Yes	⚠️ Limited	❌ Uncontrolled
Thread Pool	✅ Yes	✅ High	✅ Controlled

---

7. Conclusion

- **Single-threaded** servers are for learning only
- **Multi-threaded** servers improve concurrency but don't scale well
- **Thread pools** are industry standard for backend systems

This implementation builds a strong foundation for:

- Backend development

- Distributed systems
- Java concurrency
- Interview preparation
