Assignment 2: MathApp

COM577

[Introduction 2](#_Toc36572565)

[Question 1: Socket-based Solution 3](#_Toc36572566)

[Client 3](#_Toc36572567)

[Iterative Server 4](#_Toc36572568)

[Concurrent Server 4](#_Toc36572569)

[Testing and Evaluation 5](#_Toc36572570)

[Question 2: HTTP-based Solution 6](#_Toc36572571)

[Server 6](#_Toc36572572)

[Testing 6](#_Toc36572573)

[Client 6](#_Toc36572574)

[Testing 6](#_Toc36572575)

# Introduction

This assignment is concerned with the development of several types of client-server system. For both the client and server, generic start-points are used for convenience and ease of testing, the server starting point asks the user which type of server to start: iterative, concurrent or HTTP. Likewise, the client starting point asks the user do they want to start a socket client or an HTTP client. The overall architecture is shown in the following diagram (figure 1).

Technically, the socket client connects to the concurrent server via a ServerSocket, but in practice the ServerThread handles the socket connection to the client. This is depicted in detail later in figure 3.

This architecture enables all the common code to be shared across the different clients and servers. The project structure across all client and server types implemented in this assignment is given below.

────mathapp

│ Client.java

│ Server.java

│

├───common

│ ClientBase.java

│ ServerBase.java

│ Colors.java

│ Constants.java

│ Logger.java

│ MathService.java

│ Params.java

│ Response.java

│ ResponseType.java

│

├───http

│ ├───client

│ │ HttpClient.java

│ │

│ └───server

│ HTTPServer.java

│

└───socket

│ IOSocket.java

│

├───client

│ SocketClient.java

│

└───server

│ Request.java

│ ServerConnection.java

│ ServerConnectionLog.java

│

├───concurrent

│ ConcurrentServer.java

│ ServerThread.java

│ ThreadManager.java

│

└───iterative

IterativeServer.java

On the client side, the assignment requirement to *“close the communication with the server after receiving one result”* has been interpreted as a user option so that should the user wish to perform several calculations before closing the client; then they can. If only one calculation is needed then the user can choose to close the client at that point. The way in which the user expresses the calculation they wish to have performed has not been specified in the requirements of the assignment and so an intuitive scheme has been adopted. The protocol for conveying the calculation between client and server follows the precise specification given in the assignment requirements e.g. +:6.7:3.2 .

With reference to tree structure above, appendix A contains a full listing of all classes in the mathapp root and common packages. Appendix B contains listings for the socket solution with appendix B.1 containing listings for the iterative server and B.2 containing listings for the concurrent server. Appendix B.3 contains the socket-based client which is shared between both iterative and concurrent servers. Appendix C contains source code for the HTTP server and client.

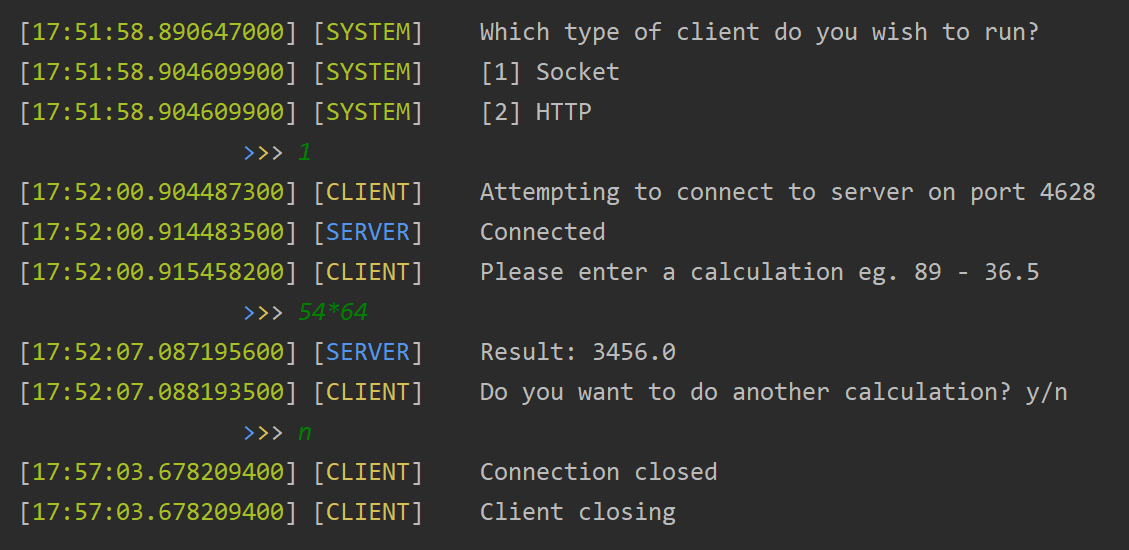
# Question 1: Socket-based Solution

In the following sections reference is made to “IOSocket”; this is a class that wraps a Socket and provides functionality to send and receive strings of text.

## Client

When the client starts, it instantiates a new IOSocket using a port number in common with the server it wishes to connect to. When the server accepts the connection, initially a confirmation of connection is received from the server. The client manages dialog with the user to obtain the maths calculation details, it builds a Params object which contains the maths calculation required and sends a stringified version of the calculation to the server via the socket.

A significant amount of validation of the user input takes place in the client, this is achieved through the getValidInput() and getYesNo() methods inherited from ClientBase.

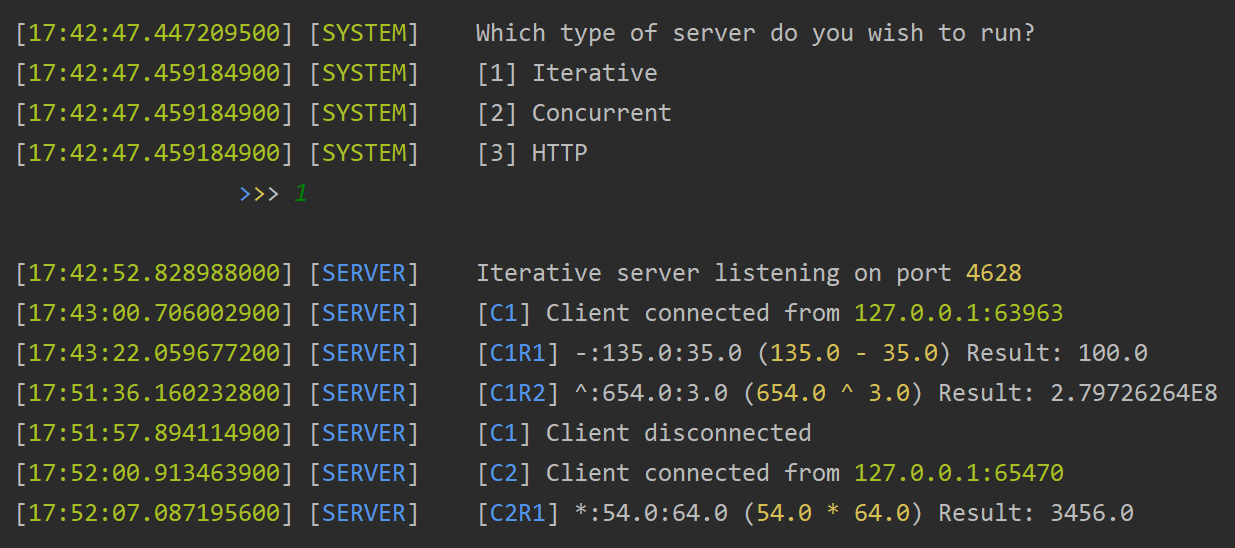
The client waits for the calculation result to be returned and then displays this on the console for the user. At this point the user is prompted if they would like to perform another calculation and if not, the connection is closed and the client closes.

This client is used to connect to both the iterative and concurrent servers. The socket client’s source code is given in appendix B.3. The client-side message sequence for connecting to a server is depicted in both figure 2 (iterative server) and figure 3 (concurrent server) since the same client logic is used in both of these servers.

## Iterative Server

The iterative server (appendix B.1) handles both connection requests and transactions from a client in a simple manner which is depicted in the UML sequence diagram below (figure 2).

When the server starts, it establishes a ServerSocket bound to a port and listens for an incoming client connection. When a client connects; the server sends an initial connection confirmation message to the client and then the client can make maths requests to server. The server makes use of the MathService class to perform the necessary calculations and return the results. During this period no further connections are accepted. When the client disconnects then the server goes back to listening for another incoming client connection. If two or more clients attempt to connect, the ServerSocket handles the queue of requests (up to 50 by default). When the current client disconnects, the next client connection request on the socket is accepted. This makes the iterative server a bottleneck if there are many clients requesting to connect and therefore limits its value.

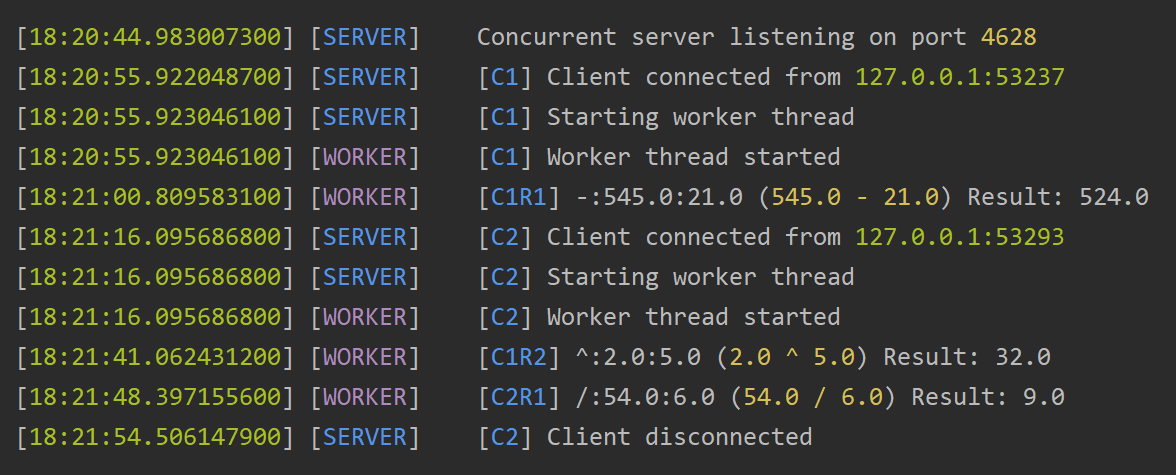
The screenshot on the left depicts two clients sequentially connecting to the same iterative server, the first client can be seen making two calculation requests, with the second client only making one.

## Concurrent Server

The concurrent server gets arounds the limitations of the iterative server by delegating responsibility for servicing a client’s needs to a dedicated child server thread (ServerThread class). So, when the concurrent server starts up, it dedicates itself to listening for client connection requests on the ServerSocket. When a new client connection is received by the server, the server spins up a new ServerThread instance and passes the socket reference for the current client connection to it. From then on, the ServerThread instance handles all of that client’s requests. The concurrent server then immediately goes back to listening for new connections.

The UML sequence diagram below (figure 3) depicts the interaction sequence for a client connecting to the concurrent server.

The ServerThread is responsible for all communication with the client. It sends a connection confirmation, receives the data from client, calls the MathService and returns the calculation result back to the client. When a client disconnects, the thread interrupts itself, causing it to terminate.

The screenshot on the left depicts two clients connecting to the server. Separate worker threads are created to service each individual client and the screen shows the two clients making concurrent requests.

## Testing and Evaluation

This section provides a set of test cases used to verify the iterative and concurrent servers. The iterative and concurrent servers share common code for the maths calculations and the validation of user input, and so only one set of tests is performed on each of these categories. The tests have therefore been split into four categories:

1. iterative server tests
2. concurrent server tests
3. maths calculation tests
4. user input validation tests

Test results where appropriate are provided in a following section.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No. | Description | Expected Result | Actual Result | Pass/Fail |
| Iterative Server Tests | | | | |
| A1 | Client A connects to server | Server log indicates client has connected; client log also indicates it is connected |  |  |
| A2 | Client A disconnects from server | Server log indicates client has disconnected |  |  |
| A3 | While client A is connected to server, client B attempts to connect | Client B should suspend waiting for client A to disconnect, when client A disconnects, client B should immediately connect to the server |  |  |
| Concurrent Server Tests | | | | |
| B1 | Client A connects to server | Server log indicates client has connected; client log also indicates it is connected |  |  |
| B2 | Client A disconnects from server | Server log indicates client has disconnected |  |  |
| B3 | While client A is connected to server, client B attempts to connect | Client B should immediately connect so that both clients A and B are simultaneously connected to the server |  |  |
| B4 | While clients A and B are connected to the server, client B can disconnect and client A can continue to make requests | Client A can continue to make requests after client B disconnects |  |  |
| Maths Calculation Tests | | | | |
| C1 | Valid calculation using + symbol  127 + 16 | Result = 143 |  |  |
| C2 | Valid calculation using – symbol  743 – 287 | Result = 456 |  |  |
| C3 | Valid calculation using \* symbol  41 \* 59 | Result = 2419 |  |  |
| C4 | Valid calculation using / symbol  540 / 90 | Result = 6 |  |  |
| C5 | Valid calculation using ^ symbol  2 ^ 8 | Result = 256 |  |  |
| User Input Validation Tests | | | | |
| D1 | Selection of “yes” to perform another calculation | User is prompted to enter another calculation |  |  |
| D2 | Selection of “no” to not perform another calculation | Client should close |  |  |
| D3 | User enters “t” when prompted for y/n | Any response other than y/n will result in the user being re-prompted |  |  |
| D4 | Invalid calculation – user inputs “a \* 4” | User should be warned no alphabetical characters are permitted |  |  |
| D5 | Missing argument – user inputs “500 \*” | User should be warned they did not enter a valid calculation |  |  |
| D6 | Missing operator – user inputs “123 123” | User should be warned they have not entered an operator |  |  |
| D7 | Invalid operator – user inputs “12 % 2” | User should be warned they have entered an invalid operator |  |  |
| D8 | Duplicate operator – user inputs “123 ++ 456” | User should be warned they entered more than one operator |  |  |
| D9 | Invalid calculation – user inputs “+ 123 123” | User should be warned there is something wrong |  |  |

Test Results

# Question 2: HTTP-based Solution

## Server

### Testing

## Client

### Testing