PROGRAMMING ASSIGNMENT 1

COP - 5615 DISTRIBUTED OPERATING SYSTEM PRINCIPLES

Fall 2023

Instructor: Dr Jonathan Kavalan

Group Number: 36

AUTHORS

Sr No.	Name	UFID	Email ID
1	Charithesh Puppireddy	56562390	cpuppireddy@ufl.edu
2	Jaya Chandra Kanth Reddy Cheemarla	79350130	cheemarla.j@ufl.edu
3	Rohan Reddy Jakkam	29711741	jakkamrohanreddy@ufl.edu
4	Dhanush Paruchuri	31893822	dhanushparuchuri@ufl.edu

TABLE OF CONTENTS

Sr No.	Content	Page
1	Overview and description of the application	3
2	Compilation and Execution Instructions	4
3	Code Structure	5
4	Execution Results	7
5	Exeception Handling Results	8
6	Results Discussion	9
7	Individual Contributions to the Project	10

1. Overview and description of the application

This programming assignment entails the creation of a concurrent client/server implementation using F# for socket communication over TCP/IP. The server program is designed to continuously listen for incoming client requests and handle multiple clients simultaneously by spawning asynchronous tasks. Upon establishing a connection, the server responds with a "#Hello!" message. Clients, once connected, have the capability to send arithmetic calculation commands, such as addition, subtraction, and multiplication, along with 2 to 4 input parameters. The server processes these commands, performs the calculations, and returns the results to the clients. In the event of errors, such as invalid commands or insufficient inputs, the server transmits corresponding error codes. Clients receive and display these results or error messages.

Exception handling plays a crucial role in this implementation, as the server generates error codes for various issues. Additionally, graceful termination is required, with clients exiting upon receiving a "#bye" command, and both clients and the server exiting upon a "#terminate" command. It is of utmost importance to ensure that no lingering processes remain after termination. Overall, this assignment focuses on socket programming, concurrent communication, error handling, and graceful termination within the context of a distributed system.

Compilation and Execution Instructions

Prerequisites

- Ensure Visual Studio Code is installed.
- Install the Ionide-fsharp extension for F# support in VS Code.
- Ensure .NET SDK is installed on your machine.

File Structure in "Team_36.zip"

- server.fsx
- client.fsx

Steps to Compile and Run the Code

Compiling and Running the Server Code (server.fsx)

1. **Extract and Open the Project:**
    - Extract the "PA1_Team 36.zip" file.
    - Open Visual Studio Code.
    - Navigate to **File - > Open Folder** and select the folder where the extracted files are located.
2. **Exploring the Code:**
    - In the Explorer panel, find and click on the "**server.fsx**" file to view the code.
3. **Running the Server Code:**
    - Open the terminal in VS Code by navigating to **Terminal -> New Terminal**.
    - Navigate to the directory containing "**server.fsx**" using the “**cd your_directory_path”** command.
    - Run the server code by typing the following command in the terminal: **dotnet fsi server.fsx**
    - You should see the output "Server is running and listening on port 9001", indicating that the server is active and awaiting client connections.

Compiling and Running the Client Code (client.fsx)

1. **Exploring the Client Code:**
    - In the same project in VS Code, find and click on the "**client.fsx**" file to view the code.
2. **Running the Client Code:**
    - Open a **new terminal** window in VS Code (ensure it is separate from the server terminal to allow them to run concurrently).
    - Navigate to the directory containing "client.fsx" using the "**cd your_directory_path"** command.
    - Run the client code by typing the following command in the terminal: **dotnet fsi client.fsx**
    - The client should connect to the server, and you should see the "Hello Client, How may I help you?" message from the server. Now, you can start sending commands to the server as per the project requirements.

Code Structure

1. Server Program

Functionality:

- Manages incoming client requests.
- Performs arithmetic operations (addition, subtraction, multiplication).
- Sends computation results or error codes to the client.

Concurrent Client Handling:

- Utilizes asynchronous tasks for each client to manage concurrent connections.
- Ensures non-blocking communication and operation processing.

Communication:

- Establishes a communication stream with connected clients.
- Sends a greeting message upon a new client connection.

2. Client Program

User Interaction:

- Accepts and sends operation commands to the server.
- Displays server responses, including results and error messages.

Server Communication:

- Establishes and maintains a communication stream with the server.
- Manages sending commands and receiving responses.
- Graceful Termination:
- Closes the connection and ends the process upon receiving a termination signal from the server.

3. Exception Handling

Server-side Management:

- Validates incoming operation commands and parameters.
- Generates and sends error codes for various exceptions:
    - -1: Incorrect operation command.
    - -2: Insufficient number of inputs.
    - -3: Excessive number of inputs.
    - -4: Non-numeric input detected.

4. **Termination**

Client Termination (bye):

- Server closes the respective client's socket.
- Continues to manage other active client connections.
- Server and Client Termination (terminate):
- Server closes all active client sockets and then shuts down.
- Sends termination signals to all clients to initiate their shutdown.

5. Note on Run-away Processes for Graceful Termination

Ensuring No Runaway Processes:

- All tasks, threads, and processes are ensured to be closed during the termination process.
- Sockets are closed, and memory is freed to avoid resource leaks.

Managing Resource Effectively:

- Attention to releasing resources and closing processes to ensure optimal resource management.
- Ensuring the robustness and reliability of the application by avoiding post-termination issues.

Execution Results

Server inputs and outputs (Server Terminal):

```
Server is running and listening on port 9001
Received: add 3 4
Responding to client 660 with result: 7
Received: subtract 20 7
Responding to client 1616 with result: 13
Received: add d 7
Responding to client 1828 with result: -4
Received: multiply 2 3 4
Responding to client 1852 with result: 24
Received: add 5 6 7 8
Responding to client 1824 with result: 26
Received: bye
Responding to client 660 with result: -5
Received: terminate
Responding to client 1852 with result: -5
Press any key to continue . . .
```

Client 660 inputs and outputs (Client 660 Terminal):

```
Hello Client, How may I help you?
Sending command: add 3 4
Server response: 7
Sending command: bye
Server response: -5
exit
Press any key to continue . . .
```

Client 1616 inputs and outputs (Client 1616 Terminal):

```
Hello Client, How may I help you?
Sending command: subtract 20 7
Server response: 13
Sending command:
```

Client 1828 inputs and outputs (Client 1828 Terminal):

```
Hello Client, How may I help you?
Sending command: add d 7
Server response: one or more of the inputs contain(s) non-number(s)
Sending command:
```

Client 1852 inputs and outputs (Client 1852 Terminal):

```
Hello Client, How may I help you?
Sending command: multiply 2 3 4
Server response: 24
Sending command: terminate
Server response: -5
exit
Press any key to continue . . .
```

Client 1824 inputs and outputs (Client 1824 Terminal):

```
Hello Client, How may I help you?
Sending command: add 5 6 7 8
Server response: 26
Sending command:
```

Exception Results

```
Hello Client, How may I help you?
Sending command: a 1
Server response: incorrect operation command
Sending command: add 1
Server response: number of inputs is less than two
Sending command: add 1 2 3 4 5
Server response: number of inputs is more than four
Sending command: add d f
Server response: one or more of the inputs contain(s) non-number(s)
Sending command: bye
Server response: -5
exit
```

The terminal outputs present a snapshot of the interaction between the client and server during the execution of the application. Below are the detailed scenarios, which also reflect the effectiveness of the implemented logic in the code

Results Discussion

1. Basic Arithmetic Operations

Addition

- **Client:** Sent command **add 3 4**.
- **Server:** Calculated the sum and responded with **7**.

Subtraction

- **Client:** Sent command **subtract 20 7**.
- **Server:** Calculated the difference and responded with **13**.

Multiplication

- **Client:** Sent command **multiply 2 3 4**.
- **Server:** Calculated the product and responded with **24**.

---

2. Handling Invalid Inputs

Non-numeric Input

- **Client:** Sent command **add d 7**.
- **Server:** Identified a non-numeric input and responded with error code **-4**.
- **Client:** Displayed the error message **one or more of the inputs contain(s) non-number(s)**.

---

3. Multiple Input Handling

Addition with Multiple Inputs

- **Client:** Sent command **add 5 6 7 8**.
- **Server:** Calculated the sum of all four numbers and responded with **26**.

---

4. Termination Commands

Client-initiated Termination (bye)

- **Client:** Sent command **bye**.

- **Server:** Responded with termination code **-5** and closed the client's connection.
- **Client:** Displayed message **exit** and terminated.

Server and All Clients Termination (terminate)

- **Client:** Sent command **terminate**.
- **Server:** Responded with termination code **-5** to the client and initiated the shutdown of all client connections and itself.
- **Client:** Displayed message **exit** and terminated.

Individual Contributions to the Project

Member 1:

Full Name: Rohan Reddy Jakkam

Contributions:

Server Logic Development: Implemented the core logic for arithmetic operations on the server-side, ensuring accurate calculations and responses.

Error Handling: Developed error-handling mechanisms on the server-side to manage erroneous client inputs and provide appropriate error codes.

Testing: Conducted initial rounds of testing, especially focusing on server responses and error management.

Member 2:

Full Name: Charithesh Puppireddy

Contributions:

Client Logic Development: Worked on developing the client-side logic to send commands to the server and process the responses received.

User Input Management: Implemented user input retrieval and sending functionality, ensuring smooth communication with the server.

Testing: Conducted robust testing on the client-side, focusing on user inputs and server response handling.

Member 3:

Full Name: Jaya Chandra Kanth Reddy Cheemarla

Contributions:

Asynchronous Task Management: Implemented asynchronous tasks on the server to manage concurrent client connections effectively.

Client-Server Communication: Developed the communication logic to ensure seamless data transfer between clients and the server.

Documentation: Took a lead role in documenting the code and creating initial drafts of the project report.

Member 4:

Full Name: Dhanush Paruchuri

Contributions:

Code Optimization: Worked on optimizing the code for both client and server, ensuring efficient execution and resource utilization.

User Experience: Ensured the client provided a user-friendly interaction environment and clear communication of server responses.

Report Finalization: Managed the finalization of the report, ensuring clarity, coherence, and comprehensive representation of the project and its functionalities.