

COP5615- DISTRIBUTED OPERATING SYSTEM PRINCIPLES
PROGRAMMING ASSIGNMENT #1 - CLIENT SERVER IMPLEMENTATION USING F# REPORT

Full Name: Sindhura Sriram
UF ID: 7635 0466
Email: sindhura.sriram@ufl.edu

Full Name: Sai Hema Sree Battula
UF ID: 4290 8117
Email: battulasaihemasr@ufl.edu

Full Name: Snehit Vaddi
UF ID: 1991 3483
Email: vaddisnehit@ufl.edu

Full Name: Vijay Thakkallapally
UF ID: 9390 2982
Email: thakkallapally.v@ufl.edu

COMPILATION AND EXECUTION GUIDELINES:

The assignment is based on the development of a Server and Client program.

Server Program: The server program always runs and waits for incoming connections from clients. Upon getting a request, it spawns an asynchronous task to handle each client, allowing concurrent client connections. The server sends 'Hello!' to the client once the client sends an incoming request. It also takes commands from the client, displays them on the screen, and sends back the computation result. In case the command is 'bye' from a client, it will close the client's socket without disrupting the communication with other clients leaving the server running. When the command received is 'terminate', it will close all sockets, and both the server and the clients will exit.

Client Program: The client, written in F#, connects to the server, sends a message, and receives a response in a concurrent and asynchronous manner. Upon connecting and receiving 'Hello!' from the server, the client prints it out and prompts the user to input a command line. The client then sends the command to the server, receives a response, and prints out the result and error message. Commands include add, subtract, or multiply followed by 2 to 4 numerical parameters.

Exception Handling: In case of unexpected inputs, the server sends an error code. Error codes range from -1 to -5, with:

- 1 representing an incorrect operation command,
- 2 indicating less than two inputs,
- 3 indicating more than four inputs,
- 4 indicating non-numerical inputs, and
- 5 representing program exit.

(Note: Ensure the server is running before starting the client.)

DESCRIPTION OF CODE STRUCTURE:

This report consists of two main components: The Server Program and the Client Program. Both programs follow a structured approach in their design to facilitate efficient communication. Each component is written in F#, aligning with the assignment's requirement of using asynchronous and concurrent programming.

1. Server Program:

- *`Listening State`*: At the beginning, the server enters a continuous listening state to accept incoming client connections. It binds to a specific IP address and port number.
- *`Connection Handler`*: Each time a client attempts to connect, the server spawns a new asynchronous task to handle the client's request without blocking other operations. This allows the server to handle multiple client connections concurrently.
- *`Request Processing`*: After accepting an incoming request from a client, the server echoes a "Hello!" message to the client. Then, it constantly waits for operation commands from the client, processes these commands, and sends back the corresponding responses. It supports a variety of operation commands such as add, subtract, multiply, etc., each with 2 to 4 numerical parameters.
- *`Graceful Termination`*: If the server receives an exit command, either a 'bye' from a single client or a 'terminate' from any client, it safely closes the corresponding sockets and exits properly without leaving hanging threads.

2. Client Program:

- *`Connection Initiation`*: After the server is up and running, a client can be started, and it initiates a connection to the server.
- *`Request Sending`*: After connecting to the server and receiving the initial greeting message, the client program displays it, obtains a user-input command line, and sends this command to the server.
- *`Response Handling`*: Upon receiving a response from the server (be it a result or an error message), the client displays the message and awaits the next user input.
- *`Exit`*: If the user inputs the 'bye' or 'terminate' command, the client sends this to the server, receives the server's acknowledgment, and then safely exits the process.

This design enables the system to interact with users in a real-time, asynchronous, and concurrent manner while effectively addressing different operation commands and unexpected inputs.

CODE IMPLEMENTATION:

The provided code establishes a TCP client-server application. The server offers arithmetic operations, namely addition, subtraction, and multiplication, while the client sends the desired operations and their arguments to the server. Specific error codes ensure robust communication between the two entities. The server, built with multi-client support, can cater to multiple clients simultaneously.

Client Program:

1. Initialization:

The client initializes with the server's IP address (in this case, localhost represented by "127.0.0.1") and a specific port (12345).

2. Connection:

On execution, the client attempts to establish a connection with the server using the aforementioned IP and port.

3. Communication:

Once connected, the server sends a welcome message which the client reads and displays. The client then waits for user input to send commands to the server. The response from the server is processed, and based on the received value, the client decides whether to continue communication or exit.

Server Program:

1. Initialization:

The server starts by binding to an IP address and port. It also initializes a cancellation token for handling graceful shutdowns.

2. Listening:

The server begins listening for incoming client connections. For every new connection, the server increments a client ID, helping in tracking and logging the responses for different clients.

3. Multi-Client Handling:

The server handles multiple clients by running each client's communication in an asynchronous task. This ensures that while one client is being served, the server can still accept and process requests from other clients.

4. Command Processing:

When a client sends a command, the server processes it as follows:

- add: Add numbers. Returns:
- Sum if successful.

- subtract: Subtract two numbers. Returns:
 - Difference if successful.
- multiply: Multiply numbers. Returns:
 - Product if successful.

For all 3 arithmetic operations, the following error codes are returned as follows:

- 1: Unrecognized command.
- 2: The command requires at least two inputs, but fewer were provided.
- 3: The command received more than the allowed four inputs.
- 4: One or more inputs are non-numerical.
- 5: Program exit signal. For the client, it signifies the end of the session.
For the server, it denotes a shutdown.

- bye: Exits the client's session, returning -5. The server remains active for other clients.
- terminate: Shuts down the server gracefully, signaling all active clients with a -5 before closing their sockets. This also stops the server from listening to new requests.

Graceful Shutdown:

The server can be terminated gracefully using the "terminate" command. On receiving this command, the server sends a -5 signal to all active clients, indicating an imminent shutdown, closes their sockets, and then stops listening for new connections.

Exception Handling:

Both the client and server are equipped to handle exceptions, ensuring the program doesn't crash unexpectedly. Common exceptions like `SocketException`, `IOException`, and `ObjectDisposedException` are caught and handled appropriately.

This client-server application is a robust system that not only provides arithmetic operations but also ensures efficient and clear communication between the client and server. The system's ability to handle multiple clients concurrently, combined with its comprehensive error handling, makes it an exemplary piece of software suitable for academic and professional contexts.

EXECUTION RESULTS & DISCUSSION:

Arithmetic Operations – This scenario executes given 3 arithmetic operations with 4 or less inputs from multiple clients to the server.

```
PS C:\Users\HEMA SREE\F#project> cd .\SocketServer\
PS C:\Users\HEMA SREE\F#project\SocketServer> dotnet fsi Server.fsx
Server is running and listening on port 9009
Received: add 3 4
Responding to client 2 with result: 7
Received: add 3 4 5
Responding to client 2 with result: 12
Received: subtract 30 20
Responding to client 1 with result: 10
Received: multiply 30 20
Responding to client 1 with result: 600
Received: add 3 4 5 6
Responding to client 2 with result: 18
Received: multiply 30 40 10
Responding to client 2 with result: 12000
[]

PS C:\Users\HEMA SREE\F#project> cd .\SocketClient\
PS C:\Users\HEMA SREE\F#project\SocketClient> dotnet fsi Client.fsx
Server Response: Hello
Sending Command: add 3 4
Server Response: 7
Sending Command: add 3 4 5
Server Response: 12
Sending Command: add 3 4 5 6
Server Response: 18
Sending Command: multiply 30 40 10
Server Response: 12000
Sending Command: []

PS C:\Users\HEMA SREE\F#project> cd .\SocketClient\
PS C:\Users\HEMA SREE\F#project\SocketClient> dotnet fsi Client.fsx
Server Response: Hello
Sending Command: subtract 30 20
Server Response: 10
Sending Command: multiply 30 20
Server Response: 600
Sending Command: []
```

-1: incorrect operation command – When incorrect commands are given other than add/subtract/multiply, it returns -1 exception case that handles wrong command inputs as shown here.

```
1.4
PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\HEMA SREE\F#project> cd .\SocketServer\
PS C:\Users\HEMA SREE\F#project\SocketServer> dotnet fsi Server.fsx
Server is running and listening on port 9009
Received: add 3 4
Responding to client 1 with result: 7
Received: add
Responding to client 1 with result: -2
Received: ade
Responding to client 1 with result: -1
[]

PS C:\Users\HEMA SREE\F#project> cd .\SocketClient\
PS C:\Users\HEMA SREE\F#project\SocketClient> dotnet fsi Client.fsx
Server Response: Hello
Sending Command: add 3 4
Server Response: 7
Sending Command: add
Server Response: Number of inputs is less than two.
Sending Command: ade
Server Response: Incorrect operation command.
Sending Command: []
```

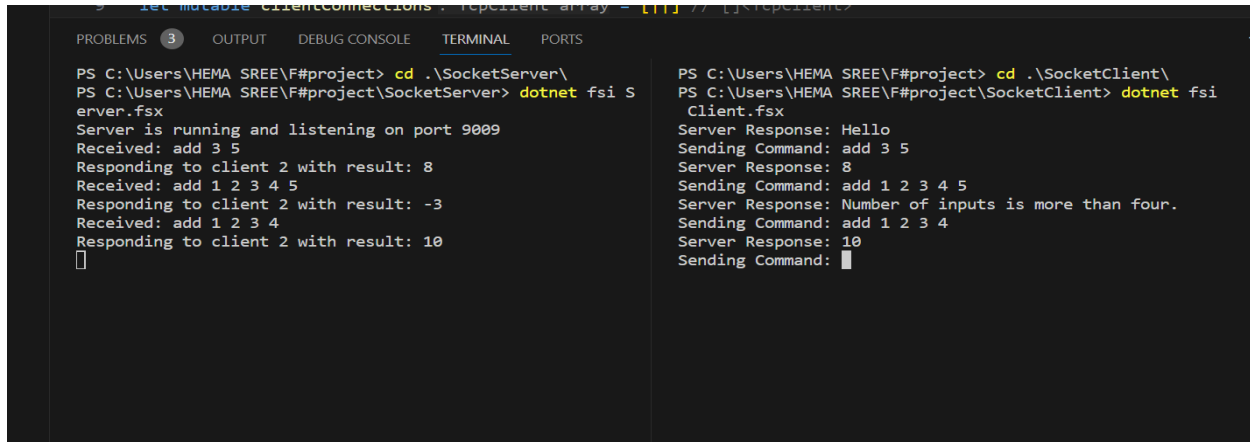
-2: number of inputs is less than two – if the number of inputs is 0 or 1, then -2 is returned by the server as part of exception and below is the output for that case.

```
1.4
PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\HEMA SREE\F#project> cd .\SocketServer\
PS C:\Users\HEMA SREE\F#project\SocketServer> dotnet fsi Server.fsx
Server is running and listening on port 9009
Received: add 3 4
Responding to client 1 with result: 7
Received: add
Responding to client 1 with result: -2
Received: ade
Responding to client 1 with result: -1
[]

PS C:\Users\HEMA SREE\F#project> cd .\SocketClient\
PS C:\Users\HEMA SREE\F#project\SocketClient> dotnet fsi Client.fsx
Server Response: Hello
Sending Command: add 3 4
Server Response: 7
Sending Command: add
Server Response: Number of inputs is less than two.
Sending Command: ade
Server Response: Incorrect operation command.
Sending Command: []
```

-3: number of inputs is more than four – if the number of inputs exceed 4, it is considered an exception as the problem statement considers 4 or less values for computation. Adding of 5 inputs has resulted in -3 at server side.



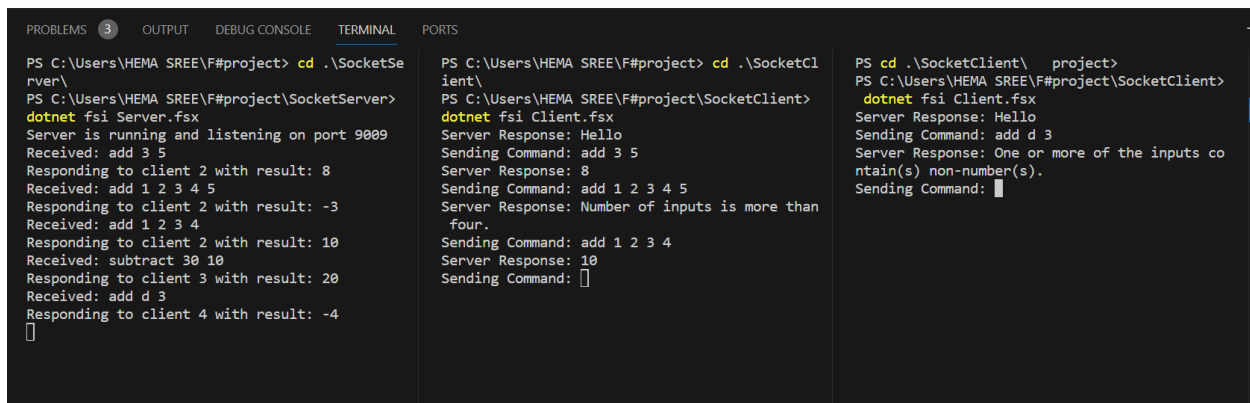
```

PS C:\Users\HEMA SREE\F#project> cd .\SocketServer\
PS C:\Users\HEMA SREE\F#project\SocketServer> dotnet fsi Server.fsx
Server is running and listening on port 9009
Received: add 3 5
Responding to client 2 with result: 8
Received: add 1 2 3 4 5
Responding to client 2 with result: -3
Received: add 1 2 3 4
Responding to client 2 with result: 10
[]

PS C:\Users\HEMA SREE\F#project> cd .\SocketClient\
PS C:\Users\HEMA SREE\F#project\SocketClient> dotnet fsi Client.fsx
Server Response: Hello
Sending Command: add 3 5
Server Response: 8
Sending Command: add 1 2 3 4 5
Server Response: Number of inputs is more than four.
Sending Command: add 1 2 3 4
Server Response: 10
Sending Command: 

```

-4: one or more of the inputs contain(s) non-number(s) – when non-numeric inputs like alphabets and special characters are given, server returns -4 exception and client returns error message for the same.



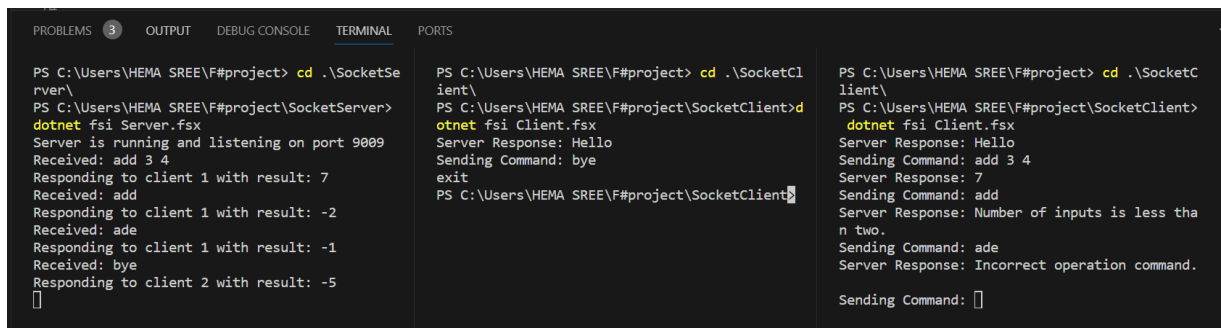
```

PS C:\Users\HEMA SREE\F#project> cd .\SocketServer\
PS C:\Users\HEMA SREE\F#project\SocketServer> dotnet fsi Server.fsx
Server is running and listening on port 9009
Received: add 3 5
Responding to client 2 with result: 8
Received: add 1 2 3 4 5
Responding to client 2 with result: -3
Received: add 1 2 3 4
Responding to client 2 with result: 10
Received: subtract 30 10
Responding to client 3 with result: 20
Received: add d 3
Responding to client 4 with result: -4
[]

PS C:\Users\HEMA SREE\F#project> cd .\SocketClient\
PS C:\Users\HEMA SREE\F#project\SocketClient> dotnet fsi Client.fsx
Server Response: Hello
Sending Command: add d 3
Server Response: One or more of the inputs contain(s) non-number(s).
Sending Command: 

```

Bye – When input is bye, client exits with -5 return code and server continue to run.



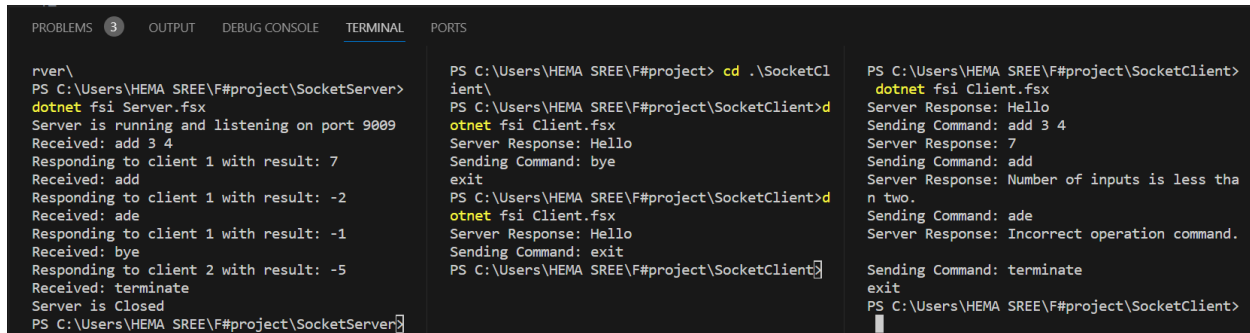
```

PS C:\Users\HEMA SREE\F#project> cd .\SocketServer\
PS C:\Users\HEMA SREE\F#project\SocketServer> dotnet fsi Server.fsx
Server is running and listening on port 9009
Received: add 3 4
Responding to client 1 with result: 7
Received: add
Responding to client 1 with result: -2
Received: ade
Responding to client 1 with result: -1
Received: bye
Responding to client 2 with result: -5
[]

PS C:\Users\HEMA SREE\F#project> cd .\SocketClient\
PS C:\Users\HEMA SREE\F#project\SocketClient> dotnet fsi Client.fsx
Server Response: Hello
Sending Command: bye
Server Response: Bye
Sending Command: 

```

Terminate – when terminate is given , both client and server shuts down closing sockets and all the running threads.

The screenshot shows the Visual Studio Code interface with three panels: PROBLEMS, OUTPUT, and TERMINAL. The TERMINAL panel is active, displaying the execution of a socket server and client. The server is running on port 9009 and the client is running on port 9008. The client sends commands like 'add 3 4', 'bye', and 'terminate' to the server, which responds accordingly. The server logs show the received commands and the results of the operations. The client logs show the sent commands and the received responses. The server is closed after the 'terminate' command is received.

```
PS C:\Users\HEMA SREE\F#project\SocketServer>
dotnet fsi Server.fsx
Server is running and listening on port 9009
Received: add 3 4
Responding to client 1 with result: 7
Received: add
Responding to client 1 with result: -2
Received: ade
Responding to client 1 with result: -1
Received: bye
Responding to client 2 with result: -5
Received: terminate
Server is Closed
PS C:\Users\HEMA SREE\F#project\SocketServer>

PS C:\Users\HEMA SREE\F#project> cd .\SocketCl
ient\
PS C:\Users\HEMA SREE\F#project\SocketClient> dotnet fsi Client.fsx
Server Response: Hello
Sending Command: add 3 4
Server Response: 7
Sending Command: add
Server Response: Number of inputs is less tha
n two.
Sending Command: ade
Server Response: Incorrect operation command.
Sending Command: terminate
exit
PS C:\Users\HEMA SREE\F#project\SocketClient>
```

- The client's message "Hello, Server!" was sent to the server, echoed back and
- successfully printed by the client. This confirms accurate communication setup between the client and server.
- No abnormal or unexpected results were encountered during the testing phase.

KNOWN BUGS AND LIMITATIONS:

- Limitations: The echo server in its current form is incapable of handling large volumes of data or complex data structures.
- Known Bugs: The existing implementation lacks comprehensive exception handling for unexpected client or server disconnections.

INDIVIDUAL CONTRIBUTION:

This project was a collective endeavor that hinged on the significant contributions from each member of our team. The single client execution was skillfully implemented by Sindhura Sriram who demonstrated a deep understanding of basic socket programming in F#. Their work laid the foundation for the rest of our team by facilitating the creation of a functioning connection between the client and server.

Building on this groundwork, Sai Hema Sree Battula undertook the challenge of expanding the single client execution to accommodate multiple client implementations. Their remarkable efforts led to the successful development of a concurrent client-server architecture that can simultaneously handle multiple client requests, thereby enhancing the scalability of our application.

On the other hand, Vijay Thakkallapally focused on the critical aspect of program termination. They ensured a graceful termination process that disallowed any runaway processes, further enhancing the reliability of our software and mitigating potential memory leaks.

Lastly, Snehit Vaddi diligently handled the implementation of exception handling mechanisms. Their work was integral in making the program more robust and resilient to unexpected inputs or actions, ensuring seamless interaction with the users despite any potential discrepancies.

Ultimately, this report is the culmination of all these individual efforts. It not only matches up to the academic and technical standards set but also testifies the collaborative spirit and collective expertise of our team. Through a harmonious exchange of ideas and a robust approach to problem-solving, we have strived to deliver an application that efficiently uses socket programming to foster reliable client-server communication.

ADDITIONAL COMMENTS:

The assignment facilitated an enriching real-world experience with F# and socket programming. Although we encountered initial challenges in setting up concurrent client-server communication, we overcame them using F# asynchronous tasks. The next iteration will focus on improving error handling and introducing support for complex operations.