COP5615- Distributed Operating System Principles
Fall 2023

Programming Assignment #1

Team ID: **PA1_Team 21**

Team Members:

Name	UFID	Email
Lahari Kethinedi	29493112	kethinedi.lahari@ufl.edu
Harish Thimmapuram	95473909	harish.thimmapur@ufl.edu
Shivani Sanjay Patil	53549503	patil.s@ufl.edu
Sheel Taskar	27534463	sheel.taskar@ufl.edu

Steps to Compile and Run:

Prerequisites:

- Ensure you have an environment that supports F# (e.g., .NET with F# installed) on your computer.

Open Terminals:

- Open two separate terminal windows or sessions on your computer.

Compile and Run the Server:

- In one of the terminal windows, navigate to the directory containing server.fsx.
- Execute the following command to compile and run the server:
  ```
  dotnet fsi server.fsx <portno>
  ```
  Note that <portno> is optional, with the default port number being 8888.

Compile and Run the Client:

- In another terminal window, navigate to the directory containing client.fsx.
- Execute the following command to compile and run the client:
  ```
  dotnet fsi client.fsx <portno>
  ```
  Note that <portno> is optional, with the default port number being 8888.

Multiple Clients:

- If you want to test with multiple client connections, open additional terminal windows and repeat step 3 to run multiple instances of the client.
- Each client will connect to the server independently.

Code Structure:

The assignment implements the client-server model in F#. The model is implemented using sockets operating on the TCP/IP protocol. The program enables a server to run on a specific port, continuously listening for communication requests. Multiple clients can simultaneously connect to the server and request it to perform arithmetic operations, such as addition, subtraction, and multiplication. The server executes these operations and returns the results or error codes to the respective clients. The code for implementing the server and client is written in two separate files: server.fsx and client.fsx.

server.fsx

A server implementation that listens for client connections and performs basic arithmetic operations based on client requests. Below is an explanation of the code structure:

ClientInfo Type:

- ClientInfo is a record type that represents information about a connected client. It contains two fields: ClientId (an identifier for the client) and Socket (the client's socket connection).
- Code enumerates ClientId every time a new client is connected to the server.

handleClient Function:

- This function is responsible for handling communication with a single connected client. It runs in an infinite loop (while true) to continuously listen for client requests.
- It receives data from the client, parses the input, and performs arithmetic operations based on the client's request.
- The parsed input is split into a command ("add", "subtract", “multiply”) and a list of numbers.
- Depending on the command, the function performs addition, subtraction, or multiplication on the numbers.
- If the command is not recognized, it assigns an error code of -1, indicating an invalid command.
- If the input does not have at least two parts (command and operands), it assigns an error code of -2, indicating an invalid input format.
- If there are too many operands (more than four), it assigns an error code of -3, indicating too many operands.
- If there's an error parsing operands as integers, it assigns an error code of -4, indicating an invalid operand format.
- If the input is” bye”, it assigns an error code of -5 and terminates the client socket.
- If the input is “terminate”, it assigns an error code of -5 and terminates the server as well as all the clients.
- It sends the result or an error code back to the client and logs the actions.

Main Function:

- The main function is the entry point of the program.
- It initializes a TCP listener on port 8888 to accept incoming client connections.
- It maintains a list of connected clients (connectedClients) as ClientInfo records.
- Inside an infinite loop (while true), the server accepts incoming client connections using listener.AcceptSocket().
- For each connected client, it generates a unique ClientId and creates a ClientInfo record with the client's socket.
- It then queues the handleClient function to run asynchronously in a separate thread, passing the ClientInfo as the state.
- The server continues to accept new connections and spawn threads to handle them concurrently.

ThreadPool Usage:

- The ThreadPool.QueueUserWorkItem function is used to enqueue the handleClient function to be executed concurrently on a separate thread. This allows the server to handle multiple clients simultaneously without blocking.

Printing and Logging:

- The code includes printfn statements for printing information to the console, such as receiving client requests and responses sent to clients. These are used for debugging and logging purposes.

client.fsx

A client program that connects to the server, allowing users to send arithmetic commands to the server and receive responses. Here's an explanation of the code structure.

Client Connection Setup:
- The code begins by establishing a connection to the server using a TcpClient. It connects to the IP address (localhost) and port 8888, assuming the server is running on the same machine.

User Interaction Loop:
- The program enters an infinite loop (while true) to repeatedly interact with the server.

User Input and Sending Requests:
- Inside the loop, the user is prompted to enter a command, such as "add," "subtract," or "multiply."
- The entered command is sent to the server over the established network stream.

Receiving and Displaying Server Responses:
- After sending a command to the server, the client reads the server's response from the network stream.
- It checks the response for various predefined error codes (e.g., -1, -2, -3) or the computed result from the server.
- It displays the appropriate message based on the response received from the server.

Checking Server Status:
- Added a thread that continuously monitors if the server is up and running.
- If the server is down, possibly due to the terminate command from the other client, this thread will detect it and shut down the client.

Multiple Clients:
- The code structure allows you to run multiple instances of the client program simultaneously, each connecting to the same server on port 8888.
- Each client operates independently and can send requests to the server concurrently.
- You can open multiple terminal windows or run the client program in separate terminal sessions to simulate multiple clients interacting with the server concurrently.

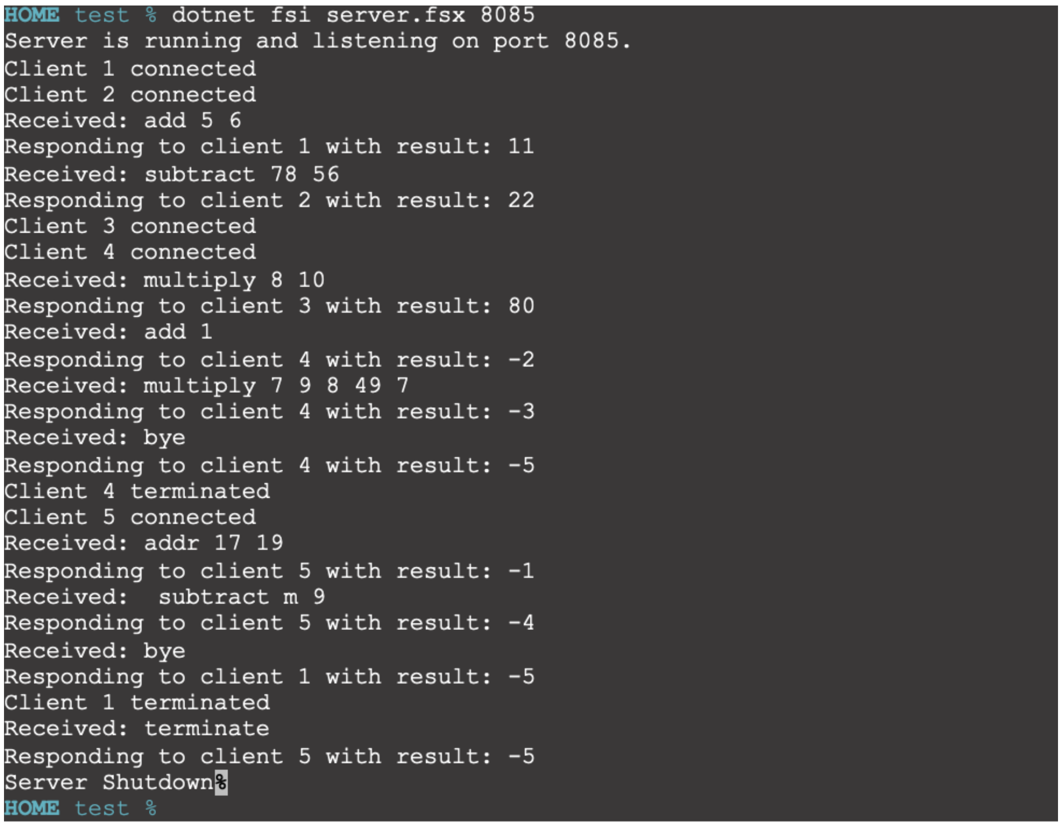
Execution and Results:

Below are some commands that we ran on the client side. The images show the client requests and server responses.

add 5 6
subtract 78 56
multiply 8 10
add 1
multiply 7 9 8 49 7
addr 17 19
subtract m 9
bye
terminate

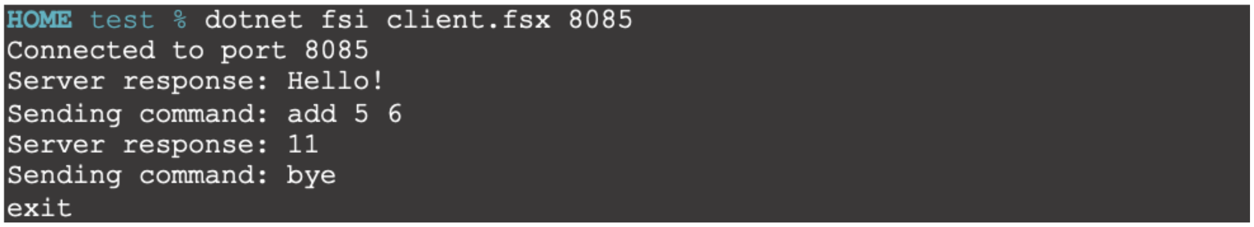
Multiple clients' examples:

Server



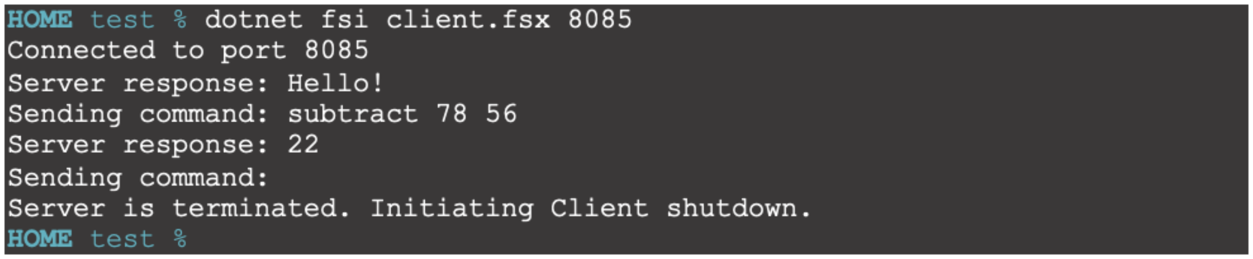
```
HOME test % dotnet fsi server.fsx 8085
Server is running and listening on port 8085.
Client 1 connected
Client 2 connected
Received: add 5 6
Responding to client 1 with result: 11
Received: subtract 78 56
Responding to client 2 with result: 22
Client 3 connected
Client 4 connected
Received: multiply 8 10
Responding to client 3 with result: 80
Received: add 1
Responding to client 4 with result: -2
Received: multiply 7 9 8 49 7
Responding to client 4 with result: -3
Received: bye
Responding to client 4 with result: -5
Client 4 terminated
Client 5 connected
Received: addr 17 19
Responding to client 5 with result: -1
Received:  subtract m 9
Responding to client 5 with result: -4
Received: bye
Responding to client 1 with result: -5
Client 1 terminated
Received: terminate
Responding to client 5 with result: -5
Server Shutdown%
HOME test %
```

Client 1



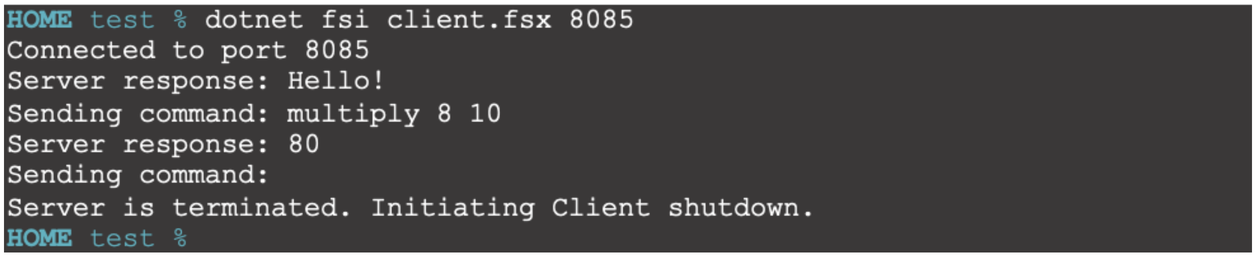
```
HOME test % dotnet fsi client.fsx 8085
Connected to port 8085
Server response: Hello!
Sending command: add 5 6
Server response: 11
Sending command: bye
exit
```

Client 2



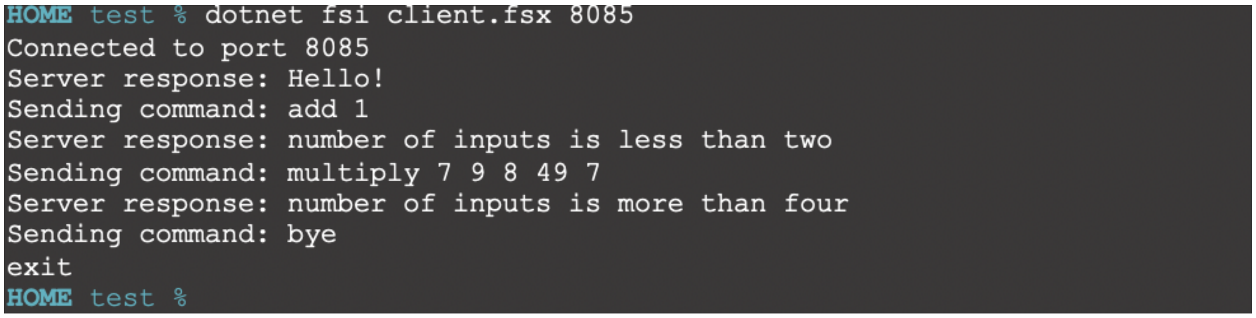
```
HOME test % dotnet fsi client.fsx 8085
Connected to port 8085
Server response: Hello!
Sending command: subtract 78 56
Server response: 22
Sending command:
Server is terminated. Initiating Client shutdown.
HOME test %
```

Client 3



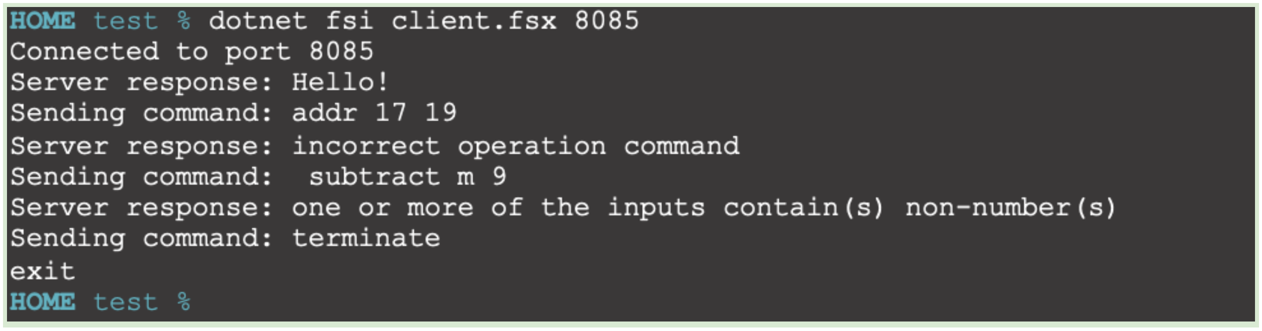
```
HOME test % dotnet fsi client.fsx 8085
Connected to port 8085
Server response: Hello!
Sending command: multiply 8 10
Server response: 80
Sending command:
Server is terminated. Initiating Client shutdown.
HOME test %
```

Client 4



```
HOME test % dotnet fsi client.fsx 8085
Connected to port 8085
Server response: Hello!
Sending command: add 1
Server response: number of inputs is less than two
Sending command: multiply 7 9 8 49 7
Server response: number of inputs is more than four
Sending command: bye
exit
HOME test %
```

Client 5



```
HOME test % dotnet fsi client.fsx 8085
Connected to port 8085
Server response: Hello!
Sending command: addr 17 19
Server response: incorrect operation command
Sending command:  subtract m 9
Server response: one or more of the inputs contain(s) non-number(s)
Sending command: terminate
exit
HOME test %
```

Above are the screenshots from the tests we ran in the system. The results are as expected with no bugs and all the exceptions handled. The code handles all the input error codes mentioned in the assignment. The program terminates gracefully on the “bye” as well as the “terminate” command. No abnormal results were detected in our thorough testing.

Bugs and limitations of the program:

Bugs:

- Encountered a bug, when a client presses CTRL + C, the client process terminates, but a runtime exception occurs on the server side, causing the server to stop. However, the remaining clients are unaware that the server has been shut down.
- **Resolved this bug** by enabling the server to be notified of an unexpected client termination without disrupting server functionality.

Limitations:

- **Lack of Client Timeout:** The code does not implement a timeout mechanism for client connections. If a client hangs or disconnects unexpectedly, the server may continue waiting indefinitely. Implementing timeouts and connection management would make code more robust.

- **Single-Threaded Server:** The server handles each client connection in a separate thread, but it uses a single thread for processing client requests. In scenarios with high concurrency and heavy computation, this design may not be scalable. A multi-threaded or asynchronous server design would be more efficient.

- **No Authentication or Security:** The code does not include any form of authentication or security measures. Anyone with access to the server's IP address and port can connect and send commands. Adding authentication and encryption mechanisms would be necessary for securing real-world applications.

- **No Data Persistence:** The server does not store any data or maintain a state between client connections. For certain applications, data persistence or state management might be necessary.

Contribution:

1. **Lahari Kethinedi:**

- Established a successful connection between the server and client, implementing the TCP/IP protocol suite and utilizing sockets as communication endpoints.
- Imported necessary packages for TCP/IP implementation. Sockets, analogous to files, were bound to specific port numbers for reliable data transmission. The project exclusively used TCP/IP for communication reliability.

- Developed a well-structured code ensuring the server actively listened to client requests and responded appropriately.
- Implemented a robust code for handling client-server communication, closing sockets as necessary for efficient operation.
- Created fundamental functions for arithmetic operations, such as addition, subtraction, and multiplication, accepting multiple parameters. Commands like "add 12 7" were processed, and the server responded with the calculated result, e.g., 19.
- Implemented multithreading in the program, allowing for concurrent execution of tasks and enhancing efficiency.
- Ensured the code snippet supported multiple clients simultaneously, enabling asynchronous communication and enhancing overall responsiveness.
- Developed a structured report format by incorporating essential elements to enhance clarity and precision.

2. **Harish Thimmapuram:**

- Implemented the setup and configuration of the development environment and ensured that the necessary environment variables were correctly configured to enable seamless execution of .NET applications.
- Implemented a robust code for error handling in client and server program, and fixed glitches that were identified while developing the code
- Contributed to the code that detects termination command "bye" initiated by the clients and appropriately responds with error code "-5" to signify the termination signal. Ensured that clients display "exit" messages on the screen and perform the exit process gracefully in response to the termination signal effectively avoiding the risk of runaway processes.
- Implemented a mechanism that allowed the server to continue accepting other connection requests even after responding to a clients "bye" command and extended the termination logic to cover the "terminate" command
- Ensured a clean and reliable termination process, guaranteeing that no unresolved threads remained after termination
- Worked on the documentation by adding essential components and conducted a comprehensive review and provided concise summaries of the program's execution results.

3. **Sheel Taskar:**

- Implemented graceful program termination by implementing an additional thread for each client. The function of this thread is to continuously monitor the server to check its status and whether it’s up and running. If the server is down, the thread will close the socket, close the other thread, and terminate the process.
- Implemented logic such that whenever any of the clients send a “terminate” request to the server, the server responds with error code #-5 and signals termination. The server gracefully exits, preventing any lingering threads. Once the server terminates, the threads in every client are notified of the server's shutdown. Consequently, all clients proceed to terminate gracefully, displaying the #exit message before concluding their processes.

- Implemented the logic to allow client and server programs to be executed on dynamically specified port numbers, which users can provide as input through the terminal. This flexibility ensures that users can select the port according to their requirements.
- Focused on comprehensive documentation, execution prerequisites, and step-by-step instructions, covering code structure, architecture, function explanations, and program limitations for both client and server programs.
- Added comments for improved code readability, maintenance, and understanding of functionality and logic flow.
- Resolved bugs encountered during code development. (e.g. Implementation of multiple threads, execution of a program on specific port no.)
- Fixed the issue that caused the code to work on MacOS but not on Windows due to variations in string handling between the two operating systems.

4. **Shivani Patil**

- Implemented logic of client ID association which enables the server to uniquely identify and differentiate between multiple connected clients, this facilitates targeted communication and data handling.
- Implemented robust error handling in the server program. This ensures that the server can handle unexpected inputs effectively. This includes checks for unrecognized/incorrect operation commands, an insufficient or excessive number of inputs, and non-numeric values in the input.
- Integrated a system so that it can generate appropriate error codes on the server side, as suggested in the assignment (-1 to -5) for different error scenarios. It allows the server to communicate error information to the client.
- Implemented client program to interpret and display corresponding error messages based on the error codes received from the server.
- Incorporated a functionality where the server sends a "Hello" message to the client, upon initiation of every client program and successful establishment of a connection with the server.
- Worked on compiling execution results of the program. Reviewed and summarized the program's execution outcomes.
- Tested the code in different environments (Windows and MacOS).