COP5615- Distributed Operating System Principles
Fall 2023

Programming Assignment #1

Date due: October 5th, 2023 **(9:35 AM Eastern Time)**

NO LATE submission will be accepted for grading!

How to submit: Please complete the group submission via CANVAS system

Introduction

In this programming assignment, you will develop a concurrent client/server implementation for socket communication using F#. It is intended for the students to experience socket programming with proper exception handling.

The implementation will have a server program and multiple client programs running with simultaneous requests which should be written in F#. They communicate with each other using socket communication on the TCP/IP protocol suite. The process is like this: After initialization, the server process listens and responds to the clients' requests. Multiple clients will try to connect to the server and send out a message containing a command and inputs. The command will be arithmetic calculation commands of adding, subtracting and multiplying and is followed by inputs of one or more numbers (e.g. add 12 7). Upon receiving the command and inputs, the server will respond to the clients with the calculating result (e.g. 19). More details will be explained later in the following sections.

Prerequisite

Socket

A socket is one endpoint of a two-way communication link between two programs running on the network. It is bound to a port number so that the TCP layer can identify the application that data is destined to. A socket is analogous to a file (an example of transparency) and is used very much like a file. Its send/receive primitives correspond to the write/read operations in a file system. In this project, we will use sockets over TCP/IP (not UDP datagram) in order ensure the reliability of the communication.

Specification

Server Program

A server program is an always-running program that accepts the clients' requests. This program starts earlier than all clients and waits for the in-coming connection. Your server program needs to handle concurrent requests by several clients, i.e., simultaneous requests using F#. *The server should listen for incoming connections and spawn a new asynchronous task to handle each client, thus allowing for concurrent client connections.* After accepting an incoming request from a client, it responds with a message "Hello!" to the client. Then, whenever it receives an operation command from the client, it prints out the command on screen and then returns with the calculation result (e.g. for the message "add 4 5" from client, the server will send a message "9" to the client). Maximum number of inputs

following the operator should be four, and the server returns an error code for exceeding or insufficient number of inputs. Note that the returning message to the client should only include a number (calculation result or error code. *If the server receives a command "bye" from a client, it closes the corresponding socket, but can still be communicating with other clients. However, if the server receives command "terminate" from any client, it closes all the sockets and exit. And all clients exit too.*

Client Program

Clients are started on remote machines after a server is running. The client written in F# should connect to a server, send a message, and receive a response in a concurrent and asynchronous manner. After connecting to the server and receiving "Hello!" message, client program prints it out and lets the user input a command line. Then it will send the command to the server (clients **do not** handle invalid input) and receive a response. After printing out the result or error message (to be specified later), it will let the user input again. The following are the operation commands used in this project:

add number1 number2 …

subtract number1 number2 …

multiply number1 number2 …

Please note that there should be 2 to 4 input parameters after each operator.

Exception Handling

Your **server** program should be able to handle unexpected inputs. For example, consider a command "add d 7". This operation cannot be done. In this program assignment, the server needs to send an error code instead of the wrong result. Error codes are negative numbers. To avoid confusion, TA will only test your program with **non-negative natural numbers**, and the result should also be non-negative (e.g. we will not test "subtract 3 5"). Therefore, negative numbers are reserved for error code.

Your server program (NOT your client!) needs to generate an error code for incorrect command (e.g. "add 4.2" or "3 2"). The following is the code list:

-1: incorrect operation command.
-2: number of inputs is less than two.
-3: number of inputs is more than four.
-4: one or more of the inputs contain(s) non-number(s).
-5: exit.

When there is more than one error in the message, the error code on the top of the error codes list precedes other error codes. (e.g. "5 4" will have two errors: incorrect operation command & number of inputs is less than two. As the error code -1 precedes the other error codes, the server should return -1). The client program should print out the corresponding error message after receiving an error code.

Termination

Graceful termination is required in this project. If there are runaway processes, points will be deducted. The detailed process is: when user types the command "**bye**" and the client send it to the server, server will reply with error code "-5". Upon receiving "-5", the client process will print out "exit" on screen and exit. But the server program will keep accepting other connection requests. If user types

“**terminate**” command and sends to server, server will also reply “-5” but both client and server will exit after that. After termination, there should not be any dangling thread.

Note on Run-away Processes for the Graceful Termination:

Your program should terminate gracefully. While testing your programs, run-away processes might exist. However, these run-away processes should be killed. Please check frequently if there are any remaining processes after termination.

Execution format

Execute server program with a port number
Execute client program with a port number

Program should follow the sample output format.
Example:

Server inputs and outputs (Server Terminal):

Server is running and listening on port 12345.
Received: add 5 8
Responding to client 1 with result: 13
Received: subtract 20 7
Responding to client 2 with result: 13
Received: add d 7
Responding to client 3 with result: -4
Received: bye
Responding to client 1 with result: -5
Received: multiply 2 3 4
Responding to client 4 with result: 24
Received: add 5 6 7 8
Responding to client 5 with result: 26
Received: terminate
Responding to client 4 with result: -5

Client 1 inputs and outputs (Client 1 Terminal):

Sending command: add 5 8
Server response: 13
Sending command: bye
exit

Client 2 inputs and outputs (Client 2 Terminal):

Sending command: subtract 20 7
Server response: 13

Client 3 inputs and outputs (Client 3 Terminal):

Sending command: add d 7
Server response: one or more of the inputs contain(s) non-number(s)

Client 4 inputs and outputs (Client 4 Terminal):

Sending command: multiply 2 3 4
Server response: 24
Sending command: terminate

Client 5 inputs and outputs (Client 5 Terminal):

Sending command: add 5 6 7 8
Server response: 26

Your screen should show results like this.

Port number
You have to be careful when using a port number, i.e., some port numbers are already reserved for special purposes, e.g., 20:FTP, 23:Telnet, 80:HTTP, etc. Keep in mind that the port number must be less than 2^{16} (=65,536).

Getting the most points

1. Please state your testing commands and results along with screenshots in your report.
2. Please try to complete the socket communication part first. After testing that, your program will be able to correctly send a message to the server and receive a response, and then you can move on to the next step.
3. *Complete simple client/server implementation of socket programming and then move to handling concurrent requests by several clients.*
4. *Try to finish the simplest operations first* (e.g. add 3 4). Exception handling can follow next.
5. *Have backups whenever your program can do more.* That way, you can avoid the situation where your program used to work but not anymore (e.g. after trying to implement exception handling). If your program cannot be compiled, your will receive zero point. Therefore, it is very important for you to keep the working versions for submission. If your program does not handle exceptions properly, points will be deducted but this is much better than receiving no point.

Reminder

Using blocks of code from other people or any other resources is strictly prohibited and is regarded as a violating of UF honor code! Please refer to the UF honor code if you are unfamiliar with it (http://itl.chem.ufl.edu/honor.html).

Report

Submitted report file should be named report.pdf or .txt and include the following:
• Your team information: Full name, UF ID, and Email of each member
• How to compile and run your code under which environment.
- **You will receive zero points if the submitted program cannot be compiled successfully.**
• Description of your code structure.
• Show some of the execution results.
- Discuss the results you got with your program.
- Discuss any abnormal results.

- Explain about the bugs, missing items and limitations of the program if there is any. - Honesty is valued here.
- Any additional comments.

Submission Guidelines:

1. The name of a source code for the server program should be named “server” and the client program “client”.
2. Only submit your source code files and report, do not include any other files or executable files in your submission.
3. Include “makefile” if you have one. (Not required)
4. Include the report in .pdf or .txt format. Name it report.pdf/txt.
5. Zip all your files into a packet: Team_ID.zip
6. Upload the zip packet as attachment in CANVAS before deadline.

Grading Criteria:

1. Correct Implementation / Outputs	60%
2. Graceful Termination / Exception handling	20%
3. Report	15%
4. Readability / Comments / Code structure	5%
Total:	100%