Network Programming, ID1212

Catherine Rakama|

Introduction

The tasks required for this project include:

- Client and server to communicate by sending messages over a TCP connection, using blocking TCP sockets.
- The client must not store any data. All data entered by the user must be sent to the server for processing, and all data displayed to the user must be received from the server.
- The client must have a responsive user interface, which means it must be multithreaded.
- The user must be able to give commands, for example to quit the program, even if the client is waiting for a message from the server.
- The server must be able to handle multiple clients playing concurrently, which means it must be multithreaded.
- The user interface must be informative. The current state of the program must be clear to the user, and the user must understand what to do next.

The assignment was done alone.

Literature Study

Literature Read:

- Lecture Videos on TCP Sockets and Streams
- Communication protocol in distributed application.
- YouTube Videos and online articles on sockets, concurrency and blocking in distributed systems
- Basic use of Observer Design pattern with layered architecture

Summary of Concepts learned:

Servers that use TCP communication have ServerSockets that usually waits for clients request to connect on a specific port number that is listening from. Clients on the other hand create socket objects that connects to a host listening to a specific port number.

A chosen application-specific communication protocol is then used to define the type of messages sent between the client and server.

Multithreading is a good way of achieving concurrency and take care of some problems experienced in DS such latency, but other factors must be considered too to make sure that the threads are not used to cause more harm than good. For instance;

> **Method of implementation and/or server/systems capability**.
> By method of Implementation I mean use of pools vs creating single threads per request. By capability I refer to the number of cores to be used. Processes have limitations in terms of the number of threads that can be created. Creating more threads that exceeds capacity of a system would result to OutofMemoryError and the program will either crush, drop jobs or become slow.
> If the program creates a new thread per client connection and the number of connected clients increases, the system cores may have to do context switching to manage all threads and this can be expensive in terms of time resource.
> If Thread pool is used instead, the incoming jobs get queued before being processed by one of the threads in a pool. This approach works well with small number of clients that are accessing the server concurrently but fails for larger number of clients because some clients will get connected and but not get serviced because clients who connected ealier have their jobs already in the front of the queue. This means this client requests will block until the queue is offloaded.

Layered architecture used with a good design pattern is good when designing distributed client server applications since it helps in ensuring that classes and subsystems are loosely coupled but have high cohesion within the sub components of the class. This is a good practice for purpose of maintaining, scaling and even debugging the system.

In a layered architecture each class should have a clear separation of responsibilities. Dependency among important classes/or packages on client side should be from top layer downwards. Upward depency should be avoided, a design pattern like observer pattern which provides interfaces to help other classes in lower layers access objects of classes at the top layer without necessarily using upward dependency to pass message/results back to the application layer.

Method

Approach

Read goals of the assignments and searched for helpful resources around these topics to understand the concepts well. Since it was a second attempt to do the assignment it was clear what was technically required so more time was spent on getting the main concepts right and trying to implement it in the solution.

N/B After having solid understanding of the concepts behind this whole exercise, the previous code was discarded as it had not so well organised code and some flow of logic didn't make sense ant thus resulted to challenges as discussed in Discussion section.

Evaluation that a requirement has been met

Client and server to communicate by sending messages over a TCP connection, using blocking TCP sockets.
Implemented a client that accepts host IP address and port number then creates a *Socket* object using these parameters. Implemented communication protocol, basically a messaging mechanism that can be understood by both server and client.

Implemented a server application that when started, it creates a *ServerSocket* object that listens on the same port as that of its clients, if connections requests come in, it accepts them.

The client must not store any data. All data entered by the user must be sent to the server for processing, and all data displayed to the user must be received from the server.
Used *BufferedBeader* object to pick clients commands from the command line and immediately passed that information to the lower layers of the application to write and send it to the server for processing.

Implemented a server that kept listening for clients' requests at the lower layer of the application and read from streams data sent by client, pass it to other layers to handle operation and write back to the stream for clients to read and display on user screen.

The client must have a responsive user interface, which means it must be multithreaded. The user must be able to give commands, for example to quit the program, even if the client is waiting for a message from the server.
Implemented a client with main thread to handle user input from terminal, interprets it and hands it to appropriate method in middle layer and comes back to listen for more instruction from the user or top layer application.

Meanwhile, the thread pool in the middle layer communicates with lower layer to get message sent to server. Lower layer also creates a listening thread to handle responses from the server.

The server must be able to handle multiple clients playing concurrently, which means it must be multithreaded.
Implemented the server with main(Listening) thread which creates another thread, operational thread to handle clients' requests.

The user interface must be informative. The current state of the program must be clear to the user, and the user must understand what to do next.
The server has a model and service layers which handlers file operations and handles game operations such as keeping game state respectively and as a result generate updated information that is sent back to client.

Result

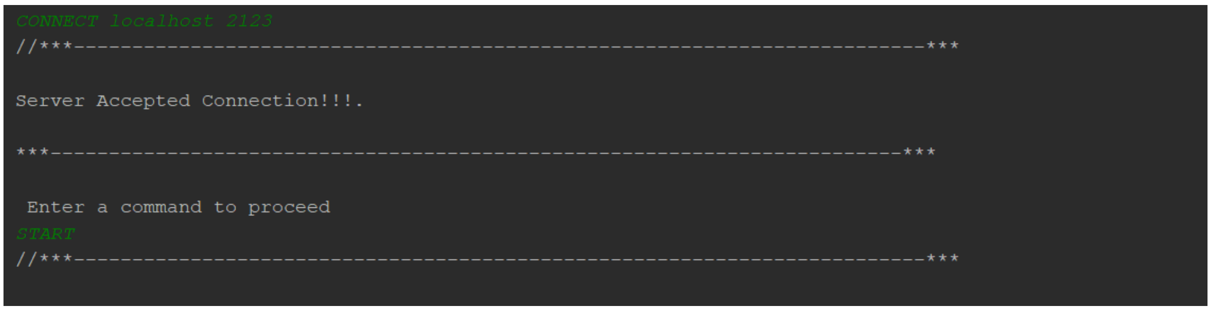
Requirements that were met.

Client and server to communicate by sending messages over a TCP connection, using blocking TCP sockets.

The client handles TCP communication in ***net* layer** which is its lower layer the app. At the top layer, It uses buffered reader to capture IP address and port number entered by the user using a buffered reader on line 59 of requestHandler(), in CmdInterpreter class. These parameters are passed to a lower layer ServerCommHandler class. This class uses them to create an object of client Socket on line 22 which will make the client connect to a server with same IP address as the one passed in *host* variable and listens to a socket passed in *port* variable.

To keep conversation going, the client starts a listening thread to keep reading any data sent by the server as shown on line 24 of *connect* method, in ServerCommHandler class. To communicate effectively, the client and server use a pre-defined application specific communication protocol as shown in MsgType class. This class outlines types of messages that the two can send to each other for better interpretation. Both client and server have access to common layer where the protocol is defined. MsgProtocol serialises message objects before they are written to stream. A good example is a *sendResponse()* method in ClientCommHandler class on line 20 which receives server state in a *response* variable and passes it to line 21 where the server creates MsgProtocol object by setting server state as body and MsgType as protocol to be read by client on listening thread. Current implementation does not involve client interpreting message protocol because of size.

The server handles TCP communication under ***net*** layer, a layer responsible for reading and writing to socket streams for transmission. ServerSocket object is created when the server is initially started as shown on line 24 of ThreadedBlockingServer class using port number passed to it as a parameter through *DEFAULT_PORT* variable. The server will then block while as it waits for clients' connection, upon receipt of a connection, line 26 of the same class is executed to whereby the server accepts clients' connection and creates an instance of this client object which will be referenced by other classes in the server whenever update of the state needs to be sent to that specific client. The final product of this is shown in *figure 1: Server Connection.*



```
CONNECT localhost 2123
//***------------------------------------------------------------------------***

Server Accepted Connection!!!.

***------------------------------------------------------------------------***

 Enter a command to proceed
START
//***------------------------------------------------------------------------***
```

Figure 1:Server Connection

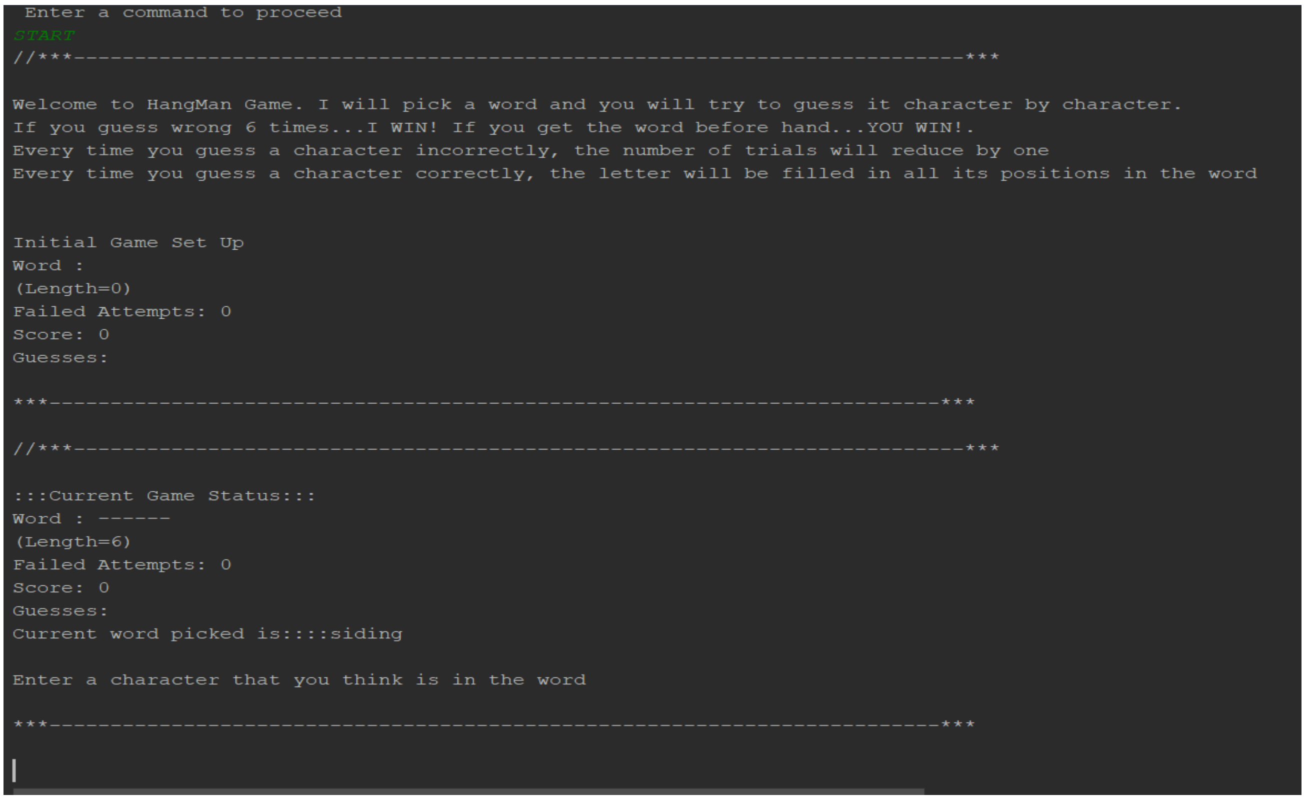
The client must not store any data. All data entered by the user must be sent to the server for processing, and all data displayed to the user must be received from the server.

Upon receipt of *START* command, host name and port number, the client establishes a connection to the server as previously explained, it then starts a new thread to listen for incoming communication from the server as shown on line 24 of ServerCommHandler class. When a new state has been received on line 56 of the same class, this Thread reads server state as stream object and coverts it to a *MsgProtocol* object and uses an instance of the object to retrieve message body which it then uses an interface object on net layer to get information sent to top layer(view) for display as shown on line 57. Interface object contains all the methods that net layer classes can access whenever they want to communicate with the top layer and is implemented on line 67 of CmdInterpreter view class at the top layer which is responsible for information displayed on GUI.

On sending data to server, user input is picked from command line using a buffered reader on line 59 of requestHandler(), in CmdInterpreter class. This data traverses through middle layer and to net layer where it is sent to the server by *sendMsg()* method found on line 35 of ServerCommHandler class, it has the same functionality as previously described *sendResponse()* method, the only difference is that it takes user input(guesses) as message body and a *GUESS* message type.

The client must have a responsive user interface, which means it must be multithreaded. The user must be able to give commands, for example to quit the program, even if the client is waiting for a message from the server.

The thread handling user interface is started when a new object of the thread class is created by the main starter class. This UI thread(CmdInterpreter class) interprets user commands redirects it to appropriate method in the controller class(Controller). All methods defined in this controller class implements *CompletableFuture* which is responsible for creating thread pools. After delivering work to the thread pool, the UI thread(CmdInterpreter class) is freed and ready to interpret more commands from the user. A good example is when the UI thread (CmdInterpreter class) receives *START* command on line 38 and calls *initialiseGame*() method of Controller class, to deliver work to thread pool as shown on line 26 in Controller class. Upon return, since the program uses a fall through mechanism inside the switch statement, UI thread automatically executes next case(PLAY), whenever its free as shown on line 44 of the UI thread(CmdInterpreter class), where the thread can process next command even though response for the first request hasn't arrived yet. As shown in *figure 2: Multithreading on Client,* responses for the two requests (Initial Game Status & Current Game Status) arrive from the server by user only issuing one command(*START*).

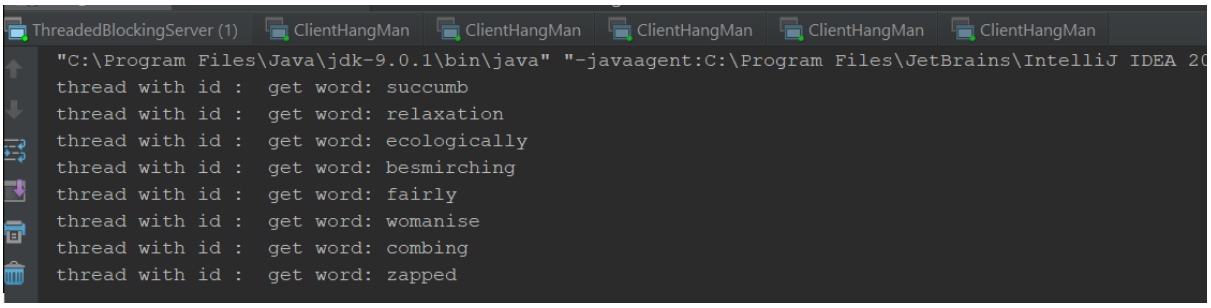


```
 Enter a command to proceed
START
//***---------------------------------------------------------------------------***

Welcome to HangMan Game. I will pick a word and you will try to guess it character by character.
If you guess wrong 6 times...I WIN! If you get the word before hand...YOU WIN!.
Every time you guess a character incorrectly, the number of trials will reduce by one
Every time you guess a character correctly, the letter will be filled in all its positions in the word


Initial Game Set Up
Word :
(Length=0)
Failed Attempts: 0
Score: 0
Guesses:

***---------------------------------------------------------------------------***

//***---------------------------------------------------------------------------***

:::Current Game Status:::
Word : ------
(Length=6)
Failed Attempts: 0
Score: 0
Guesses:
Current word picked is::::siding

Enter a character that you think is in the word

***---------------------------------------------------------------------------***
```

Figure 2:Multithreading on Client

The server must be able to handle multiple clients playing concurrently, which means it must be multithreaded.

The thread handling multiple clients is started in network layer on line 38 of server entry class. RequestHandler is a runnable class object responsible for handling server operations and is first created on line 37, an instance of this object contains information about a network layer class(ClientCommHandler) which handles sending information to client as previously discussed and an object of currently connected client socket, which had been passed as parameters in *clientCommHandler and s* variables respectively. An instance of this object is passed as a parameter to the server worker thread on line 38. The code *work* on line 39 executes this runnable class and depending on data read from stream on line 24 of RequestHandler class, this worker thread takes appropriate action to serve the client connected. The same mechanism is used to have more clients connected as shown in *figure 3: Multiple Clients where the server is connected to and is serving 5 clients concurrently.*



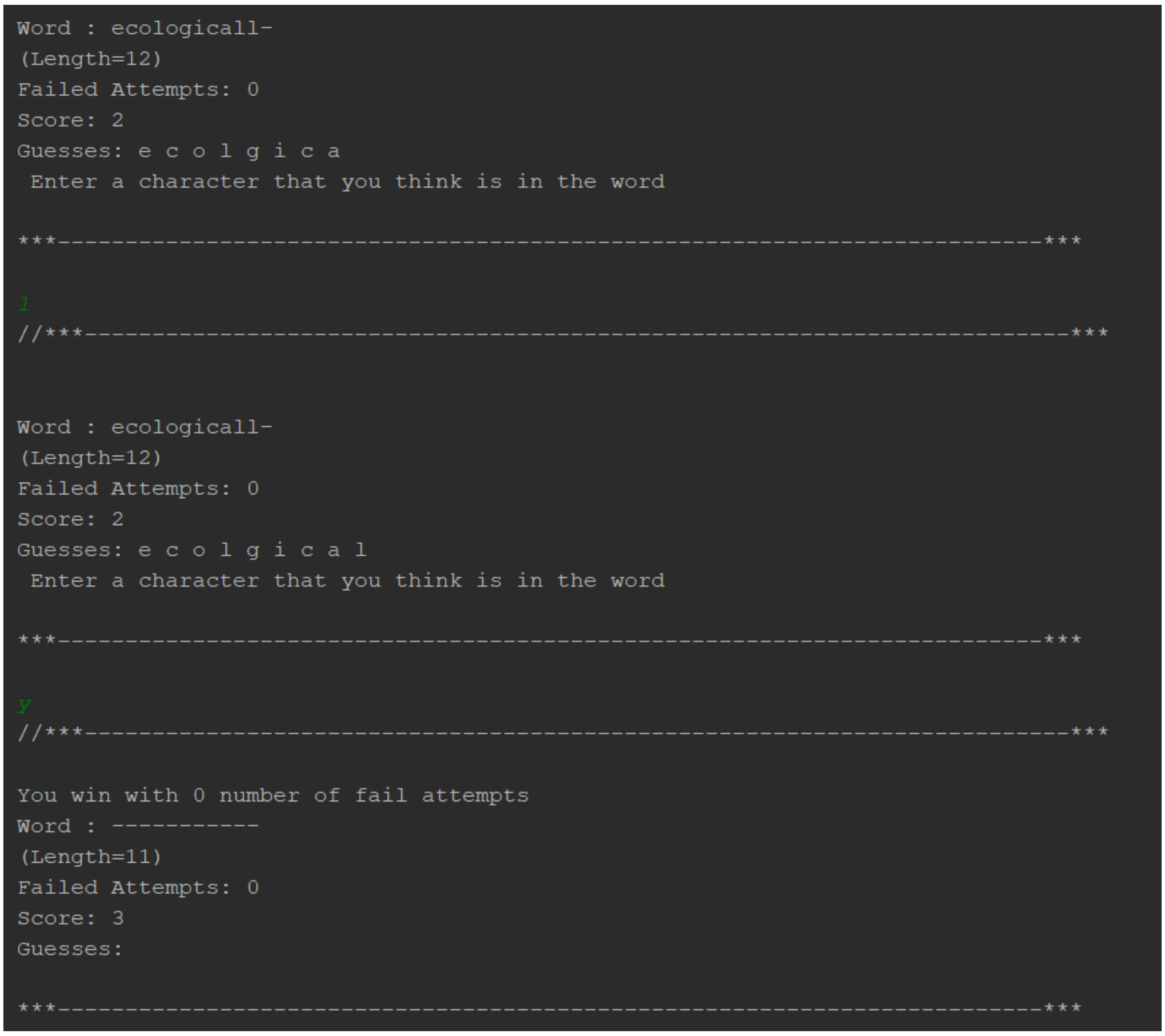
```
"C:\Program Files\Java\jdk-9.0.1\bin\java" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 20
thread with id :  get word: succumb
thread with id :  get word: relaxation
thread with id :  get word: ecologically
thread with id :  get word: besmirching
thread with id :  get word: fairly
thread with id :  get word: womanise
thread with id :  get word: combing
thread with id :  get word: zapped
```

Figure 3:Multiple Clients

The user interface must be informative. The current state of the program must be clear to the user, and the user must understand what to do next.

The program uses a command line interface to display status of the game as shown in Figure 4: Informative UI

A server state is managed in the service layer which basically consists of an interface and its implementation class . This implementation class (ServerInterfaceImpl) contains *informationMessage()* and *generateNewWord* and *playGame* methods on line 126, 101 and 41 respectively which takes user input and updates server status. At the beginning and end of the game it references FileModel class in the model layer to read from file using a buffered reader. Updated status is then send to the user.



```
Word : ecologicall-
(Length=12)
Failed Attempts: 0
Score: 2
Guesses: e c o l g i c a
 Enter a character that you think is in the word

***-------------------------------------------------------------------------***

l
//***-------------------------------------------------------------------------***


Word : ecologicall-
(Length=12)
Failed Attempts: 0
Score: 2
Guesses: e c o l g i c a l
 Enter a character that you think is in the word

***-------------------------------------------------------------------------***

y
//***-------------------------------------------------------------------------***

You win with 0 number of fail attempts
Word : -----------
(Length=11)
Failed Attempts: 0
Score: 3
Guesses:

***-------------------------------------------------------------------------***
```

Figure 5: Informative UI

Discussion

Requirements Analysis

To develop an interactive system which uses a communication protocol and TCP sockets as a communication paradigm.

This requirement was met as previously discussed the client and server were able to communicate using TCP sockets and customized application specific protocol. what I could have done differently is use web sockets as communication protocol which still utilizes TCP sockets, as opposed to using plain TCP sockets with my customized communication protocol. This would have saved me time to design my own protocol by utilizing a protocol already designed by professionals, although I haven't run benchmarks to test performance of this implementation, but it could be the case that its less efficient that web sockets.

My communication protocol worked well with the server since it was small and used local loop back for connection but if the clients were to connect to HTTP server that listen on port 80 and accepts only HTTP requests websockets would have been better choice since it provides interoperability with proxies and other network infrastructure. This is because web sockets usually initiate a HTTP connection request with a custom header where the client sets a custom header that indicates that the it wishes to *upgrade* to the webSocket protocol. If the server agrees to the *upgrade*,and the two exchange security keys then they both switch from HTTP communication protocol to webSocket which utilizes TCP sockets.

An implementation that uses web sockets will also be good if the system was to communicate with other peers that already use websockets. If the game was to be a Web App, then web sockets would be a good option since browsers only support plain http requests or websocket requests.

Both server and client should be multithreaded

This requirement was met by ensuring that the client remains interactive and responsive to the user even if it still waits for server's responses.
Threading on the client side is used to hide network related problems in distributed systems such as communication latency.

On the server side, threading helped with stability by ensuring that more clients can connect to the server and get served and performance by ensuring that the server is able to handle many clients concurrently without failing.

A blocking Server creating threads per client connection can be resource expensive especially if the server has less cores than the number of threads being created per process, as a result this could lead to out-of-memory-errors or jobs being dropped since many threads are created and as they execute long operations with large network latency the client gets blocked and cannot call the thread to do more requests, unless if each operation thread has a queue where jobs are queued but even with queues if the thread is not fast enough to empty it, the queue might get full, so use of thread pools which are queue based is not entirely a solution.

One article argued that for non-blocking socket solutions the number of bytes sent may be less than the size of the buffer and a course material said that this problem is solved by non-blocking IO so that's something that needs to be clear when I finish Non-Blocking IO assignment is complete.

Problems faced:

TCP Sockets: Major problems faced during development involved around dealing with sockets in distributed systems, knowing when to open or close the objects and how other components relate to each other. I would encounter stream corruption errors, connection reset, invalid stream headers. The solution to this was to ensure that stream read, and writes are written and read by same object types; magic value of a stream is not read by two different objects not using a new Output object to write to an existing Input stream.

Threading in distribute systems:
The first implementation of this solution had threads, but it was not clear which task are allocated to which thread. The client implemented multiple threads, but they weren't fully optimized to split tasks and leverage the benefits that come with use of multithreading. This second version implements multithreading and ensures that work is well split among.

Layered architecture and design pattern: It was tricky to learn at first even with the explanations but once the concept was understood, it was easier to scale the system in terms of layers and functionalities. Although this solution is different from my first implementation since I refactored large portion of the code because it is adding more functionality(Task) would introduce new problems that I could not find answers for, that's when I knew my architecture wasn't so good and needed some refactoring to change how different components relate to each other.

Comments About the Course

For the first attempt, many concepts were not familiar, so it took weeks before I could understand the whole concept.

For the second attempt, I had better understand the concepts behind various implementations, design pattern and layers architecture, sockets and communication very well and it took me 3 days to have the program up and running.