25/03/2018

[INFO0010-4] Introduction to Computer Networking

Project 1 : Mastermind

Adrien Minne

1. Software Architecture

(I'll refer to a client request or a server response as a "message" to be shorter.)

In this project, I've implemented the Mastermind Protocol (MP) as 3 enums and an interface.

The "MPColors" enum implements simply the different colors that our protocol can support and different useful methods to use them, such as methods to convert a String into a color byte value, a byte color value to a String, …

The "MP" class is an interface defining the principal constants of the MP (the header length, the number of colors in a combination, the protocol version,…) and the abstract methods getValue() and getMsgLenght() which return respectively the byte value in the MP and the length of a message (excluding the header) of a given type of message. It also defines a static method which takes as argument a type of message to set up correctly a header in the Mastermind Protocol.

The "MPSrv" enum implements the MP interface and defines all the type of server response and their byte value in addition to overriding the getValue() and getMsgLenght() methods to define them to server responses under the Mastermind Protocol. It's worth noting that I defined the length of a server response to a list request as the maximal length that this answer can have, but I'll take more about that in the "limits" part of the report.

The "MPClt" enum, in addition to, like the "MPSrv" enum, defining all the client request and their byte value, and implementing the methods from the "MP" interface, also provide a method to convert a byte value to a client request, and methods to find which type of server response is expected when the client send a given request.

The main class for the server, the "MastermindServer" class is a class that implements "Runnable". The main have an infinite loop listening to the port 2340, and creating new sockets and new threads, which are themselves simply instances of the "MastermindServer" class. The "run" method will read a MP header from the InputStream, then, after checking if the protocol version it's running is the same as the client, I've implemented a simple switch block for every type of client request that will create the server answer adapted to the request. After this switch block, all is left is to write the answer in the OutputStream and flush it. Obviously, if the protocol version or the client request type were incorrect, an error message (14) will be send to the client.

The main in "MastermindClient" class only initialize an instance of the "MastermindClient" class and call its method run() (the class neither implements Runnable or extends Thread, I just named it that way because it's the main loop of the program). The "run" method will first call the connectToServer() method which will try to connect to the server. This method will loop until it connects to the server or is forcefully closed. It will then send a new game request to the server and read the expected "new game created" answer. After the connection was successfully established, we will go back on the "run" method where we will ask the user for what he wants to do. After the input was successfully received, we will parse it to a byte array according to the MP protocol, send it to the server, and read the answer it gives us. If the answer is what we expected, we will parse it for the user. Then, after rapidly checking if the game is either won or lost, we're back to asking the user for input and looping until the game ends.

2 .Multi-thread coordination

The server is multi-threaded, but the threads don't have any shared variable or any type of interaction, so there is no need for synchronization mechanisms.

What we're going to talk about is the coordination between the client and the server. Since we are using the TCP protocol which is stream-oriented, we don't need to pay attention to potential data losses, but we need to be careful about stream buffering. The way I implemented that is that, in both "MastermindClient" and "MastermindServer" classes, I implemented a read() method taking as argument the length of the message to read, that will try to read on the InputStream until it has read the full desired length. In our Mastermind Protocol, we know the length of the header, and from the header we can also know the length of the following message, so we can always know the length that we have to read on the stream. If it waits for too long, I implemented a timeout on the socket that will then close itself automatically. I set the timeout at several minutes in the server, and only 30 seconds in the client, because the server always must wait for message, whereas the client only has to wait when he sends a message and actually expect an answer.

The handling of what happens if a side of the connection closes is implemented by java that will throw Exceptions. If an exception occurs, I suppose the connection closed for whatever reason, so on the server side I close the thread, and on the client side I try to establish a new connection to the server and launch a new game.

3 .Limits

I believe my code to be not too bad in terms of modifiability of the Mastermind Protocol. I tried to manage most of the protocol checks and translations in the enums forming the MP. Even if it isn't perfect, a good part of the protocol is easily modifiable directly from the enums, without touching to the main client and server classes. You can for example modify the byte value of a client request, add a color or use combinations of 6 colors easily. What can't be done easily is for example add a new type of client request, as it would mean modifying code in the protocol files, but also in the client and the serve files.

Another problem of my program is, as I said in the first part of the report, I define the length of a server response to a request to list the proposed combinations to a constant of the maximum size of the message, whereas it is a variable length. By doing that, I can simplify my code, because all the server request can thus be managed the same way, but at the cost of memory space and more date to send from the server to the client. But I found the increase needed so small (less than 100 bytes) compared to the speed of current internet connection (the slowest still are still measured in hundreds of **kilo**bytes/second), that I privileged the simplification of the code to this increase of performance.

The way I handle exception is also far from perfect, as most of the time I just reset the connection and create a new game. It's something that should be improved, but I didn't really see how, at least not with the protocol that was imposed.

4 .Possible improvement

There is two big improvement that I feel the Mastermind Protocol could use, and one improvement that is more on the level of player-friendliness.

First, the errors. There is no way for the client to signal an error to the client, which really impact the possibilities in terms of error management and recovery from an error/incorrect state. In the server side, *yes* there is an error message, but it could use some improvement. When the client receives an error, he just knows "*hey, there's an error"*, but that's all we can say. Is it using a wrong protocol version? Or is it the message he sent incorrect? That's why I just reset the connection whenever an error occurred, in hope that the next time will be better.

Secondly, we're missing a message to announce that we want to close the connection. Closing the connection of one side means that the socket will close on the other side with the TCP protocol, but we could use something implemented inside our own protocol, to not have to close abruptly after catching an Exception.

Thirdly, it would be player-friendly to have a client request to ask the server for the winning combination. Even if it goes a bit against the whole interest of the protocol (if the client side have the correct combination, why bother communicate with the server?), it would be nice to display the correct combination to the user when he loses (We *could* also develop an algorithm to search for the right answer by asking a lot of combinations to the server, but I feel like it would be a tad bit overkill and mostly a waste of time).