Reza Hassanpour

Ahmad Omar

Afshin Amighi

Andrea Minuto

Networking Assignment

Retake Assignment

The networking assignment requires to develop programs that will establish communication over THE INTERNET. This communication involves a scenario in which clients and servers exchange data.
The assignment simulates a **distributed library application**.
The *client* sends a request about a book to the *server* (librarian). The server will establish communications with the book helper to satisfy these requests. The helper (database book helper) will communicate with the server to satisfy the requests made from the client.

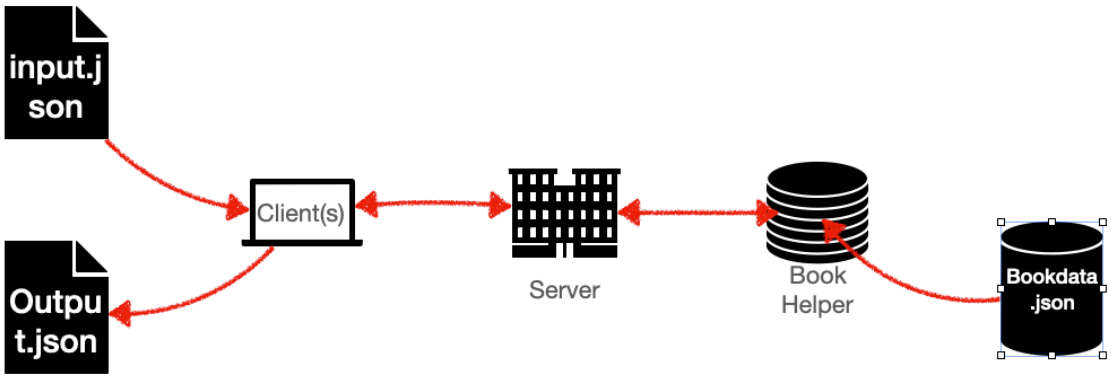

Figure 1: Topology of communications

General information for the implementation

The assignment code consist of three projects. Each project represent one part of the distributed application: *client, server, book helper.* In each project there exists one abstract class, that specifies abstracts methods to be overridden. The blueprint of a concrete class to override those methods is given in the same file. It is expected that your implementation respect the given structure and method

signature, but it is fine to add more methods when needed. Additionally, you can find some comments/ suggestions on the implementation and expected behaviour in the code.

Message exchange between the three applications:

Any message communicated between programs is a serialised json of the **Message** type (see the picture below):

```
public class Message
{
    public MessageType type { get; set; }
    public string content { get; set; }
}
```

The *type* can take one of the values defined in **MessageType**:

```
public enum MessageType
{
    Hello,
    Welcome,
    BookInquiry,
    BookInquiryReply,
    EndCommunication,
    Error,
    NotFound
}
```

Hello:
who? Client sends to the server at the first connection.
when? At the first connection to the server.
Content: *client_id*

Welcome:
who? Server sends to the client
when? After receiving Hello from client as a reply to first connection
Content: *null*

BookInquiry (scenario 1):
who?

- Client sends it to server to request data about a book,

when?

- After receiving a welcome message from the server

Content: *Title of the book*

BookInquiry (scenario 2):
who?

- Server sends it to book helper to request data from it

when?

- When server receives *BookInquiry* message from a client

Content: *Title of the book*

BookInquiryReply (scenario 1):

who?

- Book helper sends it to server,

when?

- After receiving a *BookInquiry* message from the server and searching for the title inside its book list

Content: book information from as given in the *Bookdata.json*

BookInquiryReply (scenario 2):

who?

- Server sends it to the client

when?

- After receiving a reply from book helper to forward the message of the book to the client

Content: book information from as given in the *Bookdata.json*

NotFound

who?

- Book helper sends it to the server, and server forward it to the client

when?

- When looking for a book which not found inside the json file.

Content: Book title

Error:

who?

- Both server and book helper can send this message

when?

- In case of any error. For example, if a server is running, but not able to connect to the book helper. The client would receive a message of type *Error* unavailable resource.

Content: For example, given the when scenario the content could be “Server has no access to resource”

Client:

Each *Client* program needs to read a book title from a JSON-file called “*LibInput.json*”. For each entry in this file, a client sends a query to the server asking for the *status of the book*.
After it receives the book *information* from the server it writes the book title and its status in another JSON-file called “*LibOutput.json*”. In case the clients finishes sending all the request it stops running.

For example:


```
[
 {
        "Client_id": "Client 1",
        "BookName": "Harry Potter",
        "Status": "Borrowed",
        "Error": null,
        "BorrowerName": "user-2",
        "ReturnDate": "20-09-2021"
    },
    {
        "Client_id": "Client 2",
        "BookName": "Lord of the Rings",
        "Status": "Available",
        "Error": null,
        "BorrowerName": null,
        "ReturnDate": null
    },
    {
        "Client_id": "Client 3",
        "BookName": "Unknown book",
        "Status": "Not Found",
        "Error": null,
        "BorrowerName": null,
        "ReturnDate": null
    }
```


```
public class BookData
{
    // the name of the book
    public string Title { get; set; }
    // the author of the book
    public string Author { get; set; }
    // the availability of the book: can be either Available or Borrowed
    public string Status { get; set; }
    //the user id of the person who borrowed the book, otherwise null if the book is available.
    public string BorrowedBy { get; set; }
    // return date of a book if it is borrowed, otherwise null.
    public string ReturnDate { get; set; }
}
```

Server:

Starting the server:

The server tries to connect to the book helper and listen to clients. In case the book helper is not available the server attempts to connect to the book helper (maximum 3 times). Between each attempt the server must wait for 3 seconds. If the server fails to connect to the book helper it continues to run and listen to client request. The server can be in one of the followings states:

- **State 1:** Book helper is running.
  In this case messages of the client are processed and forwarded to the book helper. Additionally, messages of the book helper are processed and forwarded to the client. If the server replies to all the requests it stays running and waiting for new client requests.

- **State 2:** Book helper is not running
  In this case messages of the client are processed, but are not forwarded to the book helper. The server sends for each *BookInquiry* an error message(See Error explanation)

- **State 3:** Book helper starts in the middle of connecting attempts with the server.
  In this case the server connects to the book helper. It continues like **State 1**

Book Helper:

The book helper starts and listen to the requests from the server. It processes the messages of the server and replies accordingly to the server. If the book helper replies to all the requests it stays running and waiting for new server requests.

Protocol:
ALL the communications should be done through ***TCP sockets*** between the client and the server, and two other **TCP sockets** between the server and the helper servers. **Helpers and clients CANNOT have direct access if not through the server (see Figure 1).** Each message has *a type* and *a content*. The type of the message defines what kind of message it is, and the content field is used to carry additional information.

Requesting book data

1. Each client is instantiated with a *client_id* and a *bookName*.
2. As soon as clients start their communications, they send a message with type *Hello* and content as their *client_id*.
3. The server responds with a message typed as *Welcome*. Content is not important in this message.
4. As soon as the client receives the *Welcome* message from the server, it sends its book request: message type is *BookInquiry* with *bookName* as the content.

5. The server forwards the *book* request to the BookHelper.
6. The BookHelper replies with *BookInqReply* message type. The content of this message must be the complete information extracted from given book information as a json file.
7. The server forwards the book information to the client.
8. The client reads the state of the reply about a the book, and builds an object of Output.

Book is not found

1. Each client is instantiated with a *client_id* and a *bookName*.
2. As soon as clients start their communications, they send a message with type *Hello* and content as their *client_id*.
3. The server responds with a message typed as *Welcome*. Content is not important in this message.
1. As soon as the client receives the *Welcome* message from the server, it sends its book request: message type is *BookInquiry* with *bookName* as the content.
2. The server forwards the book request to the book helper.
3. The book helper tries to find the requested book but it is not available in the given book information. Therefore the BookHelper *replies* with *NotFound* message type.
4. The server forwards the *NotFound* message to the client.
5. The client builds an object of Output with the name of the proper values that indicates the result of the request.

Assignment Contents:

- **Download provided codes and files from MS Teams**
- There will be 3 C# projects, one configuration file, json files for input, books, and one sample output file.
- Each project contains required json files, defined classes for: BookData, Message and MessageType.
- All the projects are based on a setting json file that contains configuration parameters.
- Students must follow instructions provided in the projects to implement their assignment.
- The assignment can be done in a group of maximum two students.

Grading requirements:
Any missing, imprecise, partial or incomplete requirement will be graded as a fail.
Other requirements details can be extracted from the description.

- The program should be written in .Net 5.0.*x*.
- Mandatory libraries to implement sockets and de-/ serialization of JSON are: System.Text.Json; System.Net.Sockets;
- Other built in libraries such as System, System.IO are allowed as long as it's not conflicting with the mandatory ones
- You need to use our project **template** to implement your application. Do not change any file-names in the provided template AND FILL IT APPROPRIATELY.
- When running the application there must be no errors/warnings. Apply proper exception handling where needed (failure to do so will result in a FAIL).

- All the communication between the programs must be carried with the proper socket kind with appropriate settings.
- The list of books will be provided as json files named *BooksData.json in appropriate format*. Your implementation must use this file. The client program should get the names of the books to be enquired as arguments from the *LibInput.json*.
- The structure of messages, *BookData*, *Output* and interface programs (main programs and clients simulator) MUST NOT CHANGE. Otherwise, your submissions will not work with the testing programs developed by the teachers.
- Your program will be tested using **different values** therefore, make sure that the data files are read from the same location as the program running, and their names are as specified here. The path/location should be handled independently from the running operating system (Mac/ Windows/Linux), error handling should be included. Your implementation should have proper exception handeling, such as *NullPointerException* and *SocketException.*
- The *LibOutput.json* should be properly formatted and respect the object data structure.
- The solution should be the **original** work of the submitting group.
- The solution should include the **names** of the participants (and students numbers) as comment in the ***Client.cs*** file
- **Use and respect the setting file provided in the project, FAILING TO DO SO WILL RESULT IN A NEGATIVE ASSESSMENT.**

The delivery of the assignment is on Microsoft form: linked below

The zip file must be renamed as ".dot" file (see also instruction in the delivery form).

The delivery is expected before **Friday 1 February at 11:00 pm**.

Submission form: TODO

In case something is unclear on the description/code/delivery, ask per time. Asking later will not be useful. Feel free to give feedback if you think there is any imperfection or doubt (please do so in the appropriate channel).

CLARIFICATION NOTE:

This note is meant to clarify a point raised by few students during the lesson regarding socket communication between the different provided projects:

- Client -> the provided code in the project for the client enforces the following behaviour: for each book record an instance of a socket client is created that must establish a socket connection, send and receive multiple messages related to one book inquiry then closing the socket connection.
- Server -> the provided code gives you some freedom about how to handle a socket connection. But the expected implementation should be you create a socket for that client connection. After the connection has been established you process the message as described in the document. How does the server understand if the client has closed connection? One solution would be to check "connected" property. Check details here:https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.socket.connected?view=netcore-3.1 .Then the socket connection is closed for that client. All the messages of a single client can be handled using the same socket connection.

To understand further how to close a socket connection properly using a connection oriented protocol have a look at this resource (read the remark section) https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.socket.close?view=netcore-3.1