1. Work Methodology and Tools Used

Team Members: Ivan, Daniel, German, and Juan

Project Overview: Our project involves implementing HTTP practices at a low-level specification using Java Sockets. To manage our project efficiently and ensure a smooth workflow, we adopted the Scrum methodology and utilised both GitHub and Trello.

Our Trello Board: https://trello.com/b/RvijUP3N

1.1. Scrum Methodology

We chose Scrum because it provides a flexible and collaborative framework that is well-suited for iterative and incremental development. Here's how we implemented Scrum in our project:

1. **Sprint Planning**: At the beginning of each sprint, we held a planning meeting to define the goals for the sprint. We broke down the project into manageable tasks and prioritized them based on importance and urgency.
2. **Daily Stand-Ups**: We conducted short daily meetings to discuss our progress, any roadblocks we encountered, and our plans for the day. This helped us stay aligned and promptly address any issues.
3. **Sprint Review and Retrospective**: At the end of each sprint, we reviewed our completed tasks, demonstrated the new functionalities, and gathered feedback. We also held a retrospective meeting to reflect on what went well and what could be improved for the next sprint.

1.2. Why We Used GitHub

GitHub was our choice for version control and collaboration for several reasons:

- **Version Control**: GitHub allowed us to track changes to our codebase efficiently. Each team member could work on their own branch, and we could merge changes seamlessly, minimizing conflicts.
- **Collaboration**: Using GitHub, we could review each other's code through pull requests, ensuring code quality and consistency. This collaborative approach also facilitated knowledge sharing among team members.
- **Continuous Integration (CI)**: We integrated CI into our GitHub repository to automatically run tests and build our project whenever new changes were pushed. This ensured that our code was always in a deployable state and helped us catch bugs early.

Our GitHub repository: https://github.com/dbsDevelops/HTTP-Project-Networks-II.git

1.3. Why We Used Trello

Trello was our tool of choice for implementing Scrum and managing our tasks:

- **Task Management**: Trello's board system allowed us to create and organize tasks into columns such as "To Do", "In Progress", and "Done". This visual representation helped us easily track the status of each task.

- **Collaboration**: Each team member could update the status of tasks, add comments, and attach relevant documents. This transparency ensured that everyone was on the same page regarding project progress.
- **Flexibility**: Trello's drag-and-drop interface made it simple to move tasks across different stages, adjust priorities, and reassign tasks as needed.

2. Group Task Distribution

Our group, composed of Ivan (IG), Daniel (DS), German (G), and Juan (JO), followed a structured approach to task distribution to efficiently manage and complete our project on implementing HTTP practices using Java Sockets. The following is a detailed overview of how we distributed and accomplished our tasks based on the images provided from our Trello board.

2.1. Task Distribution and Accomplishments

Ivan (IG)

- **Logging**: Implementing logging of server requests and responses, ensuring all actions were logged.
- **TLS Implementation**: Worked on the Client Hello, Server Hello, encryption (Cypher with TLS), and decryption with TLS.
- **Server Deployment**: Implemented real server deployment inside a Docker image.
- **Static Resources**: Implemented static resources on the server and detection of index.html files if they exist.
- **Miscellaneous Research**: Conducted research on configuring the server, displaying responses, and server request handling.
- **Persistent Connections**: Implemented part of persistent connections to maintain long-term client-server communication.

Juan (JO)

- **HTTP Verbs Implementation**: Focused on implementing various HTTP verbs like GET, HEAD, POST, PUT, and DELETE on both client and server sides.
- **TLS Features**: Worked alongside Ivan on implementing TLS features such as Client Hello, Server Hello, and ensuring proper TLS communication.
- **Cookies Logic**: Implemented the logic for handling cookies within the server.
- **General Research**: Researched multiple aspects including how to handle HTTP requests and responses and managing multiple requests concurrently.

German (G)

- **Advanced CRUD Operations**: Took charge of advanced CRUD operations, ensuring robust database interaction.
- **Conditional GET**: Implemented the conditional GET feature to enhance server-client efficiency along with the cache system.
- **API Development**: Worked on developing advanced API systems and ensuring the implementation of design patterns within the API.
- **Research Tasks**: Conducted research on modifying and deleting resources from the server, returning static content, and how to return appropriate error codes.

- **API Features**: Implemented the API for all HTTP verbs and advanced API system functionalities.
- **Basic Response from the Server:** Basic Response functionality to be used with the API.

Daniel (DS)

- **Client and Server Research**: Carried out extensive research on how to add headers to requests, specify the body of requests, and other client-server interaction mechanisms.
- **Concurrency**: Researched and implemented how to manage multiple requests concurrently.
- **CI Integration**: Automated Continuous Integration (CI) on Gradle for building and testing Java code.
- **Persistent Connections**: Implemented part of persistent connections to maintain long-term client-server communication.
- **Dynamic Features**: Focused on making the server more dynamic, including selecting ports and managing server configurations from scratch.
- **Test Infrastructure**: Set up the test infrastructure for the GreetClient class.

***Note**: It is important to note that we have all done the javadocs and documentation for the project.*

2.2. Summary of Completed Tasks

Our Trello board shows a comprehensive list of tasks marked as "Done". These tasks include:

- **Mandatory Features**: Ensured all fundamental requirements were met.
- **Persistent Connection**: Implemented to keep the connection alive between client and server.
- **Client GUI**: Developed a graphical user interface for the client.
- **Multimedia Messages**: Added support for multimedia message handling.
- **Cookies Management**: Implemented logic for managing cookies within the server.
- **Conditional GET**: Improved efficiency with conditional GET requests.
- **Advanced CRUD**: Enabled advanced Create, Read, Update, and Delete operations.
- **Folder System**: Created a folder system for organizing HTML files.
- **Docker**: Containerized the application using Docker.
- **Client CLI**: Developed a command-line interface for the client.
- **TLS**: Implemented Transport Layer Security for secure communications.
- **Logging**: Implemented logging for tracking actions.
- **Javadocs**: Implemented javadocs comment and generate field with gradle

3. Our Project Features

In our project we were able to build the following mandatory and extra features.

3.1. Mandatory Features (+6 pts)

In this section, we are going to explain how we have implemented the mandatory features for the client and server, as well as for the API which manages the requests the server receives.

3.1.1. Client

We can choose which URL to send the request to thanks to the Request class in the 'request' pacakage in which we can specify the method used, the URL, the protocol version (the 1.1 because we also manage persistence), the headers we want to send, the body type and the body content.

We are able to use any of the HTTP verbs thanks to the enumeration created named 'Verbs' in the 'utils' package and by providing it in the previous Request constructor.

We automatically added five arbitrary headers each time a request is sent thanks to the RequestHeaders constructor by calling five times the addHeaderToHeaders method which adds the headers to our list of headers. In addition, thanks to our GUI, the CLI and by code we can add arbitrary headers.

We can specify a body content thanks to the body content field in the Request class, as well as doing it manually through the GUI.

The application is able to receive the response through the console and the GUI thanks to the getResponseString() method in the GreetClient class in the 'client' package. This class previously has a readAndProcessResponse which depending on the port we have stablished the connection with (80 or 443), will manage the response in different manners.

The request status is informed through the console or response panel in the GUI and is managed thanks to the classes in 'handle_requests' explained later. In addition, we can send successive requests through our GUI by clicking the "Send" button multiple times without restarting the program.

3.1.2. Server

We have two threads in ServerApp. One is running HTTP, and the other is running HTTPS. This is because the server Socket only accepts one connection, so it's important to divide it in threads and instantiate two instances of GreetServer (one in each thread).

After starting the client app, we have the GreetServer class that is going to manage all the general logic of our server. At first is going to receive (this class) the static files path and the port that is going to run the server (80 for HTTP and 443 for HTTPS). When the greet server starts, it also creates 3 initial cookies for the client. Also instantiates the logger for the logs.

Then in the ServerApp threads are going to start the server with the initServer() method.

In here the serverSocket is going to be initialized with the specified port (80 or 443) and if there are no problems is going to handle in an infinite loop (the supposed client can send 1 or n requests and the server can support them concurrently).

In the handleClientConnection(), first we are going to make all the server TLS handshake part if the port used is 443 (HTTPS). The server accepts the connection of the client (if not is waiting until it accepts it). For a persistent connection to the client the setKeepAlive is set on true to enable it. Then we have an instance of ServerThread to handle each request individually. We have used this techique of envolving the client request´s in different threads to manage these requests concurrently avoiding the blocking of the main thread.

In the ServerThread class, overriding the run method, it is shown all the logic of processing an individual request. It enters in a while(keepAlive) to manage a persistent conexion with the client.

Inside is going to receive requests from server.readRequest() (a GreetServer method) and decrypting it if it´s running on TLS. If it´s empty this string, the client has closed the connection and is not sending any request to the server, so the connection has finished.

After having the request, we also look in the header “keep-alive” of the client request and if it´s there the connection still persists (if it´s not there it closes the connection). After that the server sends a response to the client with server.response() (using the GreetServer instance).

The server thread will be running with the persistence, looking if has that “keep-alive” header in the client request and looking if the client is sending to the server requests.

The Response class is encapsulating the status code, description, headers, and body. It features two constructors: one that initializes the status and body with default headers, and another that allows for custom headers. The constructors ensure that if a body is provided, the headers include the appropriate content type and length. Additionally, the static parse method constructs a Response object from a raw HTTP response string by validating and extracting the status line, headers, and body, while determining the correct body type.

It also provides methods to obtain its data as headers info and body data.

3.1.3. API

The Teachers API is a tool designed to streamline the management of teacher and project data in the server. At its core is the APITeachers class, responsible for orchestrating various operations. It facilitates tasks such as initializing mock data, handling HTTP requests and being the “visible face” of the data stored.

Through the APITeachers class, users can seamlessly interact with the system, using HTTP methods to perform actions like retrieving, creating, updating, and deleting teacher and project information. Additionally, the API ensures proper handling of HTTP requests by normalizing and processing paths, responding appropriately to different methods, and safeguarding against unauthorized actions with method not allowed responses.

Under the functionality of the API are the TeachersClass, Teacher, and Project classes. The TeachersClass serves as a container for managing collections of teachers and projects, it also stores the current data in the server’s RAM. It offers methods for CRUD (Create, Read, Update, Delete) operations, retrieval of teacher and project data, and maintenance of last modified timestamps.

Meanwhile, the Teacher and Project classes represent individual entities. They encapsulate essential attributes such as names, pass rates, descriptions, grades, and statuses. These classes also provide mechanisms for tracking the last modified time, ensuring data integrity and making possible to cache an old value even if other value of the same type has been modified.

3.1.4. API Request Commands

In the context of handling HTTP requests, the Command design pattern provides a robust structure by encapsulating each request as an object. This design pattern is particularly useful for handling HTTP verbs, as it allows each verb's logic to be implemented in separate classes while providing a uniform interface for executing these requests. By implementing the Command pattern, the system can handle various types of HTTP requests (GET, POST, PUT, DELETE and HEAD) in a decoupled and maintainable way.

For HTTP GET requests, the RequestGET class acts as the command. It encapsulates the logic required to handle a GET request within its execute() method. This method processes the request path, retrieves the necessary data, and generates a corresponding response. By implementing the RequestCommand interface, RequestGET provides a consistent method signature for execution, allowing clients to invoke GET requests without needing to understand the underlying retrieval mechanisms.

Similarly, HTTP POST requests are handled by the RequestPOST class, which also implements the RequestCommand interface. The RequestPOST class encapsulates the logic for creating new resources. Its execute() method processes the request body, validates the input, and adds new entries to the system. By isolating the creation logic in the RequestPOST class, the system adheres to the single responsibility principle, ensuring that each class has a clear and focused purpose.

For updating existing resources, the RequestPUT class serves as the command for HTTP PUT requests. Like the other command classes, it implements the RequestCommand interface. The RequestPUT class's execute() method handles the processing of the request body, updates existing resources based on provided data, and ensures the system's state is appropriately modified.

The RequestDELETE class represents the command for handling HTTP DELETE requests. It encapsulates the logic for removing resources in its execute() method. This class ensures that the correct resources are identified and deleted, providing appropriate responses based on whether the resource existed or not.

Additionally, the HTTP HEAD verb is managed by the RequestHEAD class. The RequestHEAD class's execute() method processes the request path and returns only the headers of the requested resource. This is useful for clients that need to check resource metadata, such as modification dates or content types, without downloading the entire resource.

Each command class focuses on a specific type of request, providing clear separation of concerns and adhering to the single responsibility principle. This approach simplifies the addition of new modifications, as changes to one type of request do not impact others, promoting a robust and scalable architecture.

3.2. Javadocs (+? pts)

For this optional section, few changes had to be made as everything is done automatically by the Java API, which is public and available on the web: https://docs.oracle.com/javase/8/docs/api/

. The only thing we had to modify was the gradle.build file inside the project's app folder, which indicates how we want gradle to generate the javadocs:

```
javadoc {
    source = sourceSets.main.allJava
    classpath += configurations.runtimeClasspath
    destinationDir = file("src/main/java/http/project/networks/ii/static_resources/javadocs")
    options.addStringOption('Xdoclint:-missing', '-quiet')
    options.memberLevel = JavadocMemberLevel.PUBLIC
    options.author = true
    options.version = true
    options.links("https://docs.oracle.com/javase/8/docs/api/")
}
```

As might be clear by now, we have generated the javadocs inside the default static folder of our server, so accessing them can be done by deploying the same. To build the javadocs you must first execute the "./gradlew build" command at the root project directory level, and then the "./gradlew javadoc" command (or just execute our Docker image, that does it by default).

3.3. Persistent Connections (+2 pts)

In the 'server' package, thanks to the ServerThread class we can manage persistent connections. In the GreetServer class, when we call the handleClientConnection() method it starts the ServerThread after handling the TLS and cookies. In our constructor we must specify the GreetServer and the socket, which will be the client's socket. We have a Boolean named 'keepAlive' to check if the connection must be persisted or not. It is initialised as true from the beginning.

When running the thread, we get the input from the client's socket and we check if the connection should be kept alive which by default is going to be true. Now we check if the client socket is open and if it is, with TLS we must decrypt the message or just read the request if it's the port 80. If the request read is null we exit and if not we check if the request is kept alive by checking the "Connection" header and looking if its values is "keep-alive" thanks to the isConnectionAlive method in Request.

Finally, we manage the response of the client and if we received that the connection must not be kept alive we stop the persistence.

3.4. Multimedia Messages (+1.3 pts)

For this section, we have made use of methods in the HTTPUtils class, the HttpBody class, and reference 1, to determine if the file existed within the path we are serving locally from static, and then determine (once we know the file exists, if not, null body is returned), if it is a file that we have to send in binary or not.

The first problem we encountered when we started doing this section was to determine whether a file was binary or not, since reading an html document is not the same as reading an audio/video file. To do this, we used a HashMap, that is initiated statically at HTTPUtils, with the extension of a file as the key and our HttpBodyType class as a value. To determine what the content of the body would be, we defined a function "determineBodyType", which parses the extension of the file and returns the type it is, being RAW if it does not find the extension in the HashMap. Then, a function that returns true if it's binary type and false if not, allows us to properly read the file, is called "isBinaryType". At the end, the function "createRequestBodyFromFile" returns a constructed body, with its type and content, to which the "Content-Type" and

“Content-Length” headers, required to determine the body information, are previously added when constructing the response class.

Finally, the last problem we encountered is that we had to send, depending on whether it was a binary file or not, the answer in one part or in two (answer string and the binary content in order not to damage it). To do this, with an if sentence we determine whether or not it has a binary body, and if it has a binary body (we store the binary files in a byte[] at the body and files that are not binary in a String), we use a function that adds the bytes of the response to the bytes of the binary file, to finally send it through the socket.

3.5. Cookies (+0.9 pts)

For this section I have put an extra of expired Cookies. I have a method isExpiredCookie(Cookie cookie) in HTTPUtils to compare the dates with a simple substraction and then look if the Cookie has exceeded the time or not.

The flow of the Cookies is that first the server is going to send 3 cookies to the client (after client sends the first request to our server). These cookies have a random time of expiration (between 20 sec and 60 sec). The server is going to use the header “Set-Cookie” to send it, and according to the standard of the cookies, first I must send the Cookie ID (an Int that starts in 0 and is automatically incremented when new cookies are created). When the client receives the server response, in GreetClient, is going to add the server cookies to our client. In the next request the client add the cookies in the request for the server and when the server receives the request is going to look if the cookies that the client has sent (that are the same cookies that the server has) are expired or not. So, it´s going to look if every cookie of the server cookies list is expired. If one is expired it will remove it and add a new one. Then the client is going to receive the new cookies in the response and will remake the cookies again (to have the same cookeis as the server).

All the cookies implementation is in GreetClient, GreetServer (where I have my cookies for client and server respectively), a Cookie class (where I do the parser cookie and i control the id of the cookies) and two boolean methods in HTTPUtils: *isExpiredCookie()* and *existServerCookie()* to extract some of the logic to our utils library.

3.6. Conditional GET (+1.3 pts)

The Conditional Get Class handles HTTP GET requests with support for the "If-Modified-Since" header, which allows clients to conditionally request resources based on their last modified date. In the execute method, it first checks for the presence of the "If-Modified-Since" header, if this header is found, the method compares it with the last modified date of the requested resource. If the resource has been modified since the specified date, it proceeds to return the resource using a regular GET request, otherwise, it returns a 304 Not Modified response, making the client search its cache for the item. If no conditional headers are present, the resource is fetched and returned as usual.

The class uses helper methods to find specific headers and determine the last modified date of resources, now implemented in the teachers API. The methods allow the client to search for specific data and retrieve only if it has been changed, avoiding fetching already cached data and allowing to cache individual items.

3.7. Advanced CRUD (+0.6 pts)

- Create (POST)

RequestPOST: Handles HTTP POST requests for creating new resources. It processes the request body, validates the input data, and adds new entries to the system (e.g., new teachers or projects). It ensures proper creation with appropriate response statuses and error handling.

- Read (GET)

RequestGET: Manages HTTP GET requests to retrieve resources. It processes the request path to fetch data for teachers, projects, or specific sub-resources. It returns the requested data in JSON format, supporting comprehensive data retrieval.

- Update (PUT)

RequestPUT: Handles HTTP PUT requests for updating existing resources. It processes the request body to update resources such as teachers or projects. It ensures the data is correctly updated and provides feedback on the success or failure of the update operation.

- Delete (DELETE)

RequestDELETE: Manages HTTP DELETE requests for removing resources. It processes the request path to identify and delete the specified resources (e.g., a specific teacher or project). It ensures proper deletion with accurate status responses.

3.8. HTML Directory Listing (+1 pts)

For this section, we have based ourselves on how the apache2 or python3 service displays directories. To do this, within the function “createRequestBodyFromFile” of HTTPUtils, we determine whether the path that the user passes us is a directory or not, using the isDirectory() method of the File class. If the method determines that it is a directory, it will enter the “buildDirectoryHtml” method of the same class, which will enumerate the directory entries and auto-generate a list of all entries.

Here, we had a very big problem, and it was that you had to be careful with the “/” slashes, because if you specified the wrong path, then the client was not able to solve well the location of the files required by the possible page you clicked on (if any).

Although at first it seemed crazy, it was as easy as separating the logic of how we managed the path; before fixing the error, we passed all the path together, that is to say, local path where you serve the files, plus the path that you obtained from the url; now, passing it separately using the path class and then the normalize method before enumerating the things in the directory, so that whatever you did, it always managed the path resolution issue well, since the Path class of Java manages to create a valid Path.

At the end, the method returns an auto-generated html that is then put into a body inside the “createRequestBodyFromFile” method, which is then returned.

3.9. Docker Deployment (+1 pts)

To deploy to a Docker machine, what you do is by creating a custom Docker image (possible thanks to the Dockerfile), load the most current version of the server using the git command and then build the project, generating a jar called "ServerDocker", which is the one that must be run to serve the server automatically. What this container does for you is to serve the server through its port 80 to the port you bind on your machine with the docker run -p parameter (if you don't bind port it's as if nothing happens).

One of the biggest problems we had was that we were initially trying to configure the docker image with an Ubuntu container, which gave us problems finding the Java jdk. So doing some research, we found that there was a pre-assembled image that came with java configured: openjdk. So starting from this image and installing the rest of the necessary programs, we were able to make the image work.

The other big part of the challenge is that we had to generate the .jar at the time of building the project, since gradle by default, only generates one, and we needed two (later in ClientCLI). So with a little help from artificial intelligence (which also did not have much idea), we managed to build the following function within the build.gradle file:

```
task jarServerDocker(type: Jar) {
    archiveBaseName.set('ServerDocker')
    manifest {
        attributes(
            'Main-Class': 'http.project.networks.ii.server.ServerDockerApp'
        )
    }
    from sourceSets.main.output
    from {
        configurations.runtimeClasspath.collect { it.isDirectory() ? it : zipTree(it) }
    }
    duplicatesStrategy = DuplicatesStrategy.EXCLUDE
}
```

This creates a task, which you could later call, and it generates the .jar required by the Docker image to work. Then, by adding at the end of the build.gradle file the line ‘build.dependsOn jarServerDocker’, you tell gradle that when it is built it has to compulsorily execute that task, so you do two necessary things in one: compile the project and generate the custom jar in the relative path of the project “app/build/libs/”.

The generated jar (compiled ServerDockerApp Main class) takes by default the path of the project where we are serving the statics, but there is an option that allows, through an environment variable (STATIC_FILES_DIR), to take a custom path inside the container, which gives more freedom to create your own web page and test how it looks/works.

3.10. Client CLI (+1.3 pts)

At the beginning it was what made us doubt the most, as it generated doubts such as how we were going to interpret the parameters both with one script and with two, how we were going to be able to pass the arguments in the order you want, how we were going to get the hostname right, which does not require parameters.

As we all know this is not like python, which is made for scripting and includes its own argument parser. But thinking about the python parser made us think, until we finally found out if there was an argument parser for java (there isn't), but we did find one from an external library: jpotsimple (Reference 2). So, researching inside their own examples page, we were finally able to implement

a very good parser that was able to do everything we doubted about, so we got down to work and created the ClientCLI class, which contains all the logic to walk the arguments correctly.

By using our existing GreetClient class and using all the methods contained in it, we can communicate with any server in a more modular and interactive way; and thanks to the jpotsimple library, we have managed to save many more lines of code than if we had implemented our own argument parser or made an interactive menu.

Again, we re-created a task in the same way as we did in the Docker section, and added it to the final line I mentioned earlier:

```
build.dependsOn jarServerDocker, jarClientCLI
```

So, in building the project, we not only generated the Docker compiles and jar, but now we are also generating the Client CLI jar file. Try running it and see what happens (don't worry, it has a help menu if you don't put arguments :)).

3.11. TLS (+3.3 pts)

3.11.1. TLS Handshake

You have sysout commented to print all the TLS handshake if you want to see it

In this part we have created a TLS package with a class ClientHello that will be our TLS client and the class ServerHello that will be our TLS server. Greet client will have an instance of ClientHello and GreetServer will have an instance of ServerHello.

At first Client Hello start with the TLS handshake doing a ClientHello. Here we use the DataOutputStream (dos) and the DataIputStream (dis) to write data to the server using the socket of the client and to read the data from the server using the socket. In the ClientHello we send the version of TLS 3.3, we send a random number of 32 bytes, and we send a cipher suite that is going to be used for the encryption.

After that starts the Server that reads all that information, saving the clientRandom and looking if suports the ciperSuite sent by the client. If the server suports the cipher suite (always supports, it because we always send the same cipher suite) the ServerHello will start. In the ServerHello we send another random (the serverRandom), we send the same TLS version, and we send a real certificate to the client (created with the tool OpenSSL). Our certificate uses the type “X.509” and has an expiration of 10 years and we have also put the private_key.key that is the private key used to create this certificate.

Then the client receives the serverRandom and the Certificate (the server sends the Certificate bytes to the server and the client transforms those bytes into a certificate, using the java class Certificate). After having it, we validate the certificate in the client with the methods checkValidity() and verify(publicKey) of the X509Certificate class (using the public key that the certificate provides to us). If is not validated, an exception is going to be thrown and TLS handshake will stop. After the validation of the certificate, the client sends the Premaster Secret.

For the premaster secret the client generate 48 random bytes and it will be sent to the server **ciphered with the public key of the certificate**. Then the client has the clientRandom,

serverRandom and premasterSecret to build the symetric key, so it will build the symetric key using the TlsShared class.

TlsShared class explanation:

1. For building the symetricKey first we concatenate the clientRandom and serverRandom generating a seed for symetric key.
2. Then we use the PBKDF2 (Password-Based Key Derivation Function 2) algorithm with HMAC-SHA256 as the pseudo-random function to generate a password-derived key. We must pass it the premaster, the previous seed and the key length and iterations to generate the symmetric key with PBKDF2 algorithm. This derived key will be the symmetric key used to transmit the data between client and server.

The server is going to receive that premaster secret, but it needs to decipher the information with the private key of the certificate. For that we need to convert the private_key.key (that is in PEM format) to a DER format (java Cipher class only supports that format). Is the same private key of the certificate, but in two different formats. We use the private_key.der to **decipher the premaster secret** ciphered by the client with the certificate public key and when we deciphered it we generate the same symmetric key (using again TlsShared class) in the server using the clientHello, serverHello and the premaster secret obtained in the previous steps.

Now the client and the server have the same symmetric key to transmit the information.

3.11.2. Symetric ciphering between client and server

First, before encrypting (since the process of creating the request is the same), the client determines if the port defined in the GreetClient class corresponds to the port defined for HTTPS in HTTPUtils, if this corresponds, it will encrypt the message using the method defined in the HTTPUtils class called “encryptMessage”, which is in charge of taking an array of bytes and first encrypt it with the generated symmetric key, then from the array of bytes that is already encrypted, pass it to a base64 string, which will generate a single line, and pass it through the Socket.

For both encryption and decryption, we use more or less the same methodology, for this, we use the Java Cipher class, which is the one that helps us to encrypt/decrypt the messages; for this, we use an instance of “AES/ECB/PKCS5Padding” cipher. Then, by creating the secret key with the SecretKeySpec class, we indicate the symmetric key we are going to use and then the algorithm, which in this case is “AES”. Finally, depending on the operation that we are going to do, an encryption/decryption operation will be performed according to the function that is called.

Then, when the key arrives at the server, again, we determine whether the port on which the server is serving is the HTTPS port or not, and if it is the HTTPS port, we read the incoming string over the socket and then decrypt it with our method. Here, we had a small problem, and it is that when we read a simple HTTP request, we read several lines, but when we read an HTTPS request, we read only one line in base64, so the methods are incompatible; therefore, we defined a new method called “readBase64String”, which is responsible for reading a single line from the socket, corresponding to the encrypted line that comes from the client. Again, the request interpretation logic is the same as with HTTP.

Then, and finally, again, in the "response" method of the GreetServer class, we have the same logic as before to send the responses through the socket, only that again we change a little when sending the message through the Socket. To do this, we define a method called “handleResponse” within the GreetServer class, which determines whether or not the defined HTTPS port is being used; if it is being used, an array of encrypted bytes is returned, and if it is not being used, the same array of bytes that has been entered is returned; subsequently, the response is sent through the Socket so that the client receives it.

Now, this is where we had the most problems, and that is that when we tried to generalise the method of reading the response, we had interpretation problems, as the method was adapted to read the response in plain text and was executed at read time every time it read a line. So the generalisation of the method consisted of making it read the whole answer, and then proceed to its subsequent interpretation. To do this we created a parser method of the response, to pass it after having read it to the method that processes it, called “processResponse”, which is responsible for passing it and then interpret its contents.

This is where we had the next problem, and that is that once we tested the method with HTTPS (which mysteriously worked), we realised that for HTTP it didn't work, the input buffer got stuck once it read the entire request; This was only because the buffer was not ready to continue reading data, and did not return null, so by using the “ready()” method of the BufferedReader class we were able to get out of that reading loop, because if the buffer was not ready to continue reading, it was out of the loop and went on to process the response.

3.12. Logging (+0.6 pts)

For this additional feature we have implemented a class named ‘Logger’ which when initialised with its constructor in the ‘GreetServer’ class entering the desired logName (in our case it will always be “server_log_”) and calling its log(String message, int type) method, it will automatically generate a log appending to the file name the current date with underscores and a .txt file extension.

The desired output would have been a file in the style of *server_log_2024/06/09_16:33:00.log* but due to issues with Windows not being able to open files with ‘:’ or ‘.log’ we had to do the previous compromise. In addition, we needed to add a hidden file, ‘novaesto’, to push the folder into the Github repository whenever no logs are in the /logger/logs directory.

3.13. Client GUI (+1.3 pts)

In this section, we created a Graphical User Interface for the user to send any type of request to the server. It is possible the host (text field), the port (text field), the method (dropdown), the body type (dropdown), the body content (text area) and additional headers (dialog). It will try to send the body type and its content even if the method is a GET, HEAD or DELETE but won’t be successful due to how we have managed GreetClient and GreetServer. Here is the initial aspect when running the ClientGui.java class in the ‘gui’ package.

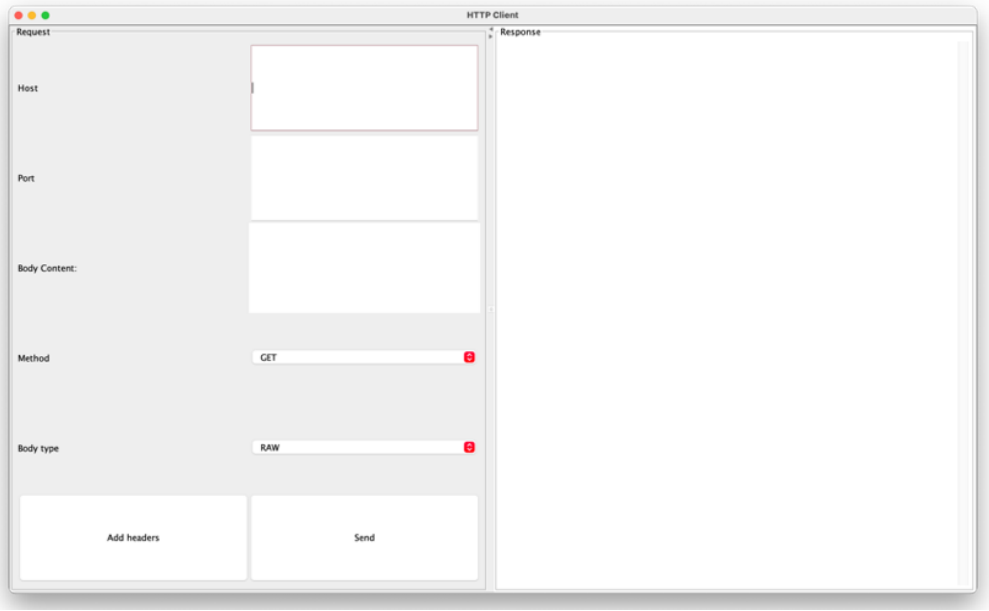

Figure 21: Screenshot of initial aspect of the GUI

In the next figure we can show the aspect of the response panel when sending a request.

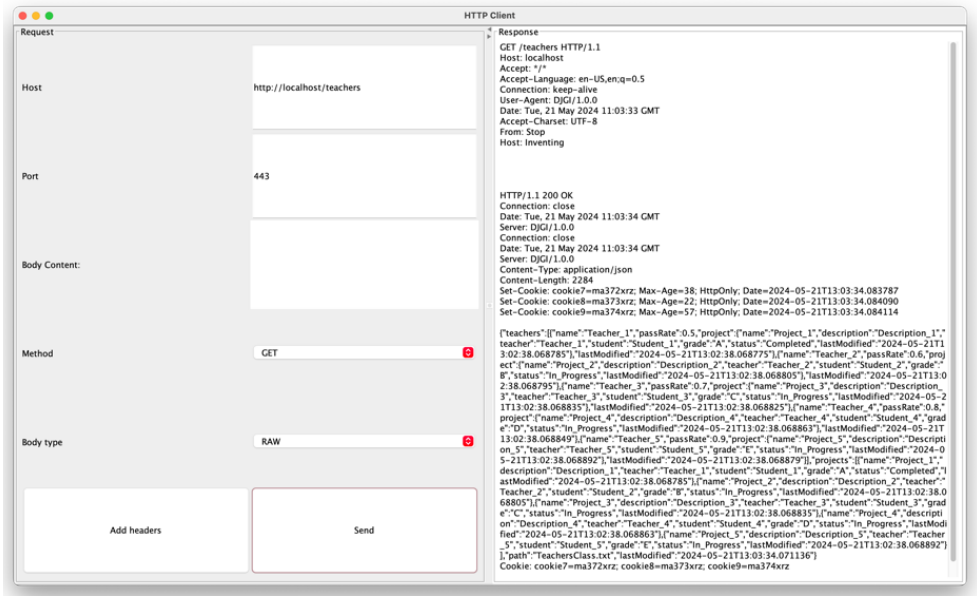

Figure 2: Example of response

Furthermore, we can run the GUI from a .jar file without needing to use our project. *Note: it will be necessary to build the project first before to generate the ClientGUI.jar file found in app/build/libs, like S4vitar would say ¿vale?.*

4. References

[1]. https://developer.mozilla.org/es/docs/Web/HTTP/Basics_of_HTTP/MIME_types
[2]. https://jopt-simple.github.io/jopt-simple/index.html
[3]. https://www.catchpoint.com/blog/wireshark-tls-handshake
[4]. https://docs.oracle.com/javase/tutorial/uiswing/
[5]. https://refactoring.guru/design-patterns/command