COMP416: Computer Networks
Project 1
NFTNet: CoinGecko-API BasedApplication Layer Protocol

Emirhan Kartal 76130

INTRODUCTION

This project presents the development of an application layer protocol, aptly named NFTNet, which leverages the comprehensive CoinGecko API to facilitate the exchange of Non-Fungible Token (NFT) data. Operating on the classic client/server model, the NFTNet server establishes a TCP connection to ensure reliable communication with its clients. Also, server is using a "Multi-Thread" architecture for the utilization of serving multiple clients at one socket. All computations are unique for the one client and does not affect the other clients. The server acts as the intermediary between the clients and the CoinGecko API web server, thereby centralizing the data retrieval process. This design ensures that clients are abstracted from direct interaction with the CoinGecko API, receiving pertinent NFT information through a streamlined and specialized protocol.

Server Implementation

My codes are modified versions from the sample projects that uploaded on course site. Under the ServerThread.java file, I modified the "public void run()" function to suit my input, output, and data fetching protocols. Since under this function, all code runs at a parallel fashion, enables the multi-client serving.

```
public void run()
{
    try
    {
        is = new BufferedReader(new InputStreamReader(s.getInputStream()));
        os = new PrintWriter(s.getOutputStream());
        s.setSoTimeout(DEFAULT_TIMEOUT_SET);
        line = is.readLine();
        while (line.compareTo("QUIT") != 0)
        {
            System.out.println("Client " + s.getRemoteSocketAddress() + " sent :  " + line);
            s.setSoTimeout(DEFAULT_TIMEOUT_SET);
            ArrayList<String> outArray = getRequest(line);
            for (String data : outArray) {
                os.println(data);
                os.flush();
            }

            line = is.readLine();
        }
    }
    catch(SocketTimeoutException e)
    {
        os.println("timeout");
        os.flush();
        System.out.println("Socket timed out!");
    }
```

I am starting a timeout after a connection and before any prompt is given for preventing empty connections, which means a client is connecting but there are no inputs of any kind. After that I am taking a string from the is.readLine() and starting the while() loop. This loop stays on until there is a “QUIT” message from the client. After taking a valid input, I give info about the client and its prompt to the server terminal. Then, the input is given into the “getRequest” function to create a String Array List which the details are explained below. After getting the String Array List, a for loop outputs each line to the client. After the list ends gets the input from the client again. All these steps are unique for an individual client. Also, there is an exception case for socket timeout which sends an information line to client and server itself before closing the socket for that client.

```
private ArrayList<String> getRequest(String inputLine) throws IOException {

    ArrayList<String> finalOut = new ArrayList<>();
    int listFlag = 10;
    String inline = "";
    int maxPage = 14;
    if ("list".equals(inputLine)) {
        listFlag = 1;
        String pageContent = "";
        String checkCondition = "";
        for (int i = 1; i <= maxPage; i++) {
            System.out.println(Integer.toString(i));
            pageContent = fullList(i);
            if("Error".equals(pageContent)) {
                finalOut.add("Error while getting the data. Try again later.");
                listFlag = 5;
                break;
            }
                if (i != 1) {
                    pageContent = pageContent.substring( beginIndex: 1);
                }
                if (i != maxPage) {
                    pageContent = pageContent.substring(0, pageContent.length() - 1);
                }
                // Append a comma between pages if not the last page
                if (i < maxPage) {
                    pageContent += ",";
                }
            inline += pageContent;
        }
    }else{
        inline = getSearchedNFT(inputLine);
        listFlag = 0;
        //System.out.println(inline);
        if (inline.equals("Error")) {
            listFlag = 10;
        }
    }

    if (listFlag == 0) {
        //System.out.println(inline);
        JSONObject jsonObject = new JSONObject(inline);
        name = jsonObject.getString( key: "name");
        assetPlatformId = jsonObject.getString( key: "asset_platform_id");
        floorPriceUSD = jsonObject.getJSONObject( key: "floor_price").getBigInteger( key: "usd").toString();
        finalOut.add( index: 0, element: "Name: " +  name + " | Asset Platform ID: " + assetPlatformId + " | Floor Price USD$"  + floorPriceUSD);
    }

    if (listFlag == 1) {
        JSONArray jsonArray = new JSONArray(inline);
        for (int i = 0; i < jsonArray.length(); i++) {
            id = jsonArray.getJSONObject(i).getString( key: "id");
            symbol = jsonArray.getJSONObject(i).getString( key: "symbol");
            name = jsonArray.getJSONObject(i).getString( key: "name");
            assetPlatformId = jsonArray.getJSONObject(i).getString( key: "asset_platform_id");
            if (!jsonArray.getJSONObject(i).isNull( key: "contract_address")) {
                contractAddress = jsonArray.getJSONObject(i).getString( key: "contract_address");
                finalOut.add(i,  element: "|Symbol: " + symbol + "  |Name: " + name + "  |ID: " + id + "  |Asset PlatformID: " + assetPlatformId + "  |Contract Address: " + contractAddress);
            } else {
                finalOut.add(i,  element: "|Symbol: " + symbol + "  |Name: " + name + "  |ID: " + id + "  |Asset PlatformID: " + assetPlatformId + "  |Contract Address: Null");
            }
        }
    }

    if (listFlag==10){
        finalOut.add("There is an error while getting the data. Check your command or wait for a while to restore requests.");
    }

    finalOut.add("done");

    //System.out.println(finalOut);

    return finalOut;
```

This is the “getRequest” function that returns the requested string array list at the run() function mentioned earlier. Firstly, the inputLine (clients’ prompt) is checked if it is a “list” or a specific NFT_ID. If the prompt is exactly “list”, a for loop starts to get each page from the GeckoAPI. Before every iteration of for loop, “fullList” function is called to save all the data to a string and then with if checks of the pages, parsing (removing) the first, last or both of characters from this list to remove square brackets. This is to create one large string file with all the JSON data inside and ready for a conversion after to an array. This gives an output string that starts with a “[“ and ends with a “]”. If the inputLine (clients’ prompt) is not “list”, another function named “getSearchedNFT” called to get the specific JSON data for an NFT. This also returns a string which is a JSON object. For both of these checks, there is a flag variable that is used to keep the track of the selected events. This listFlag is set to 10 because this is the number that I chose as an error number. At the end of the function, this variable is checked so that can send an error message if none of the fetching functions worked. Also, for the non-existing NFT search URLs, the “getSearchedNFT” function returns “Error” string and then this condition is checked. If it is the case, the listFlag variable is set to 10 for error output. Additionally, for the next JSON parsing, object parsing or array parsing, 0 and 1 is assigned to listFlag. If the client sent a correct NFT id to fetch, listFlag is set to 0 and the if condition below is fulfilled. Then the “inline” string

that contains all the fetched information is used to create a JSON object with the imported library "JSONObject". With ".getJSONObject" and ".getString" functions, the requested data is saved to a temporary string variable. After combining and formatting the string in a usable manner, added to the String Array List named" finalOut". After these steps finished, as a last change, string "done" is added at the end of the Array List. Which will be used at the client side after for proofing. Otherwise, if the client sends "list", 1 is assigned to listFlag which means the parsing for listing will be used next. Getting the data and adding to the finalOut array part is nearly the same except this time, for every JSONObject inside the JSONArray is parsed to get the required string values. The for loop is to get the data for I number of objects which in this case it is around 3382 JSONObjects. After the formatting and creating the array, the "done" string is added at the end for correction.

```
private String fullList(int page) throws IOException {
    String BASE_URL = "https://api.coingecko.com/api/v3/nfts/";
    String LIST_ENDPOINT = "list?per_page=250&page=" + Integer.toString(page);
    String urlString = BASE_URL + LIST_ENDPOINT;

    URL url = new URL(urlString);

    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setRequestMethod("GET");
    conn.connect();

    String inLine = "";
    Scanner scanner = new Scanner(url.openStream());

    while (scanner.hasNext()) {
        inLine += (scanner.nextLine());
    }

    scanner.close();
    System.out.println(inLine);
    return inLine;
}
```

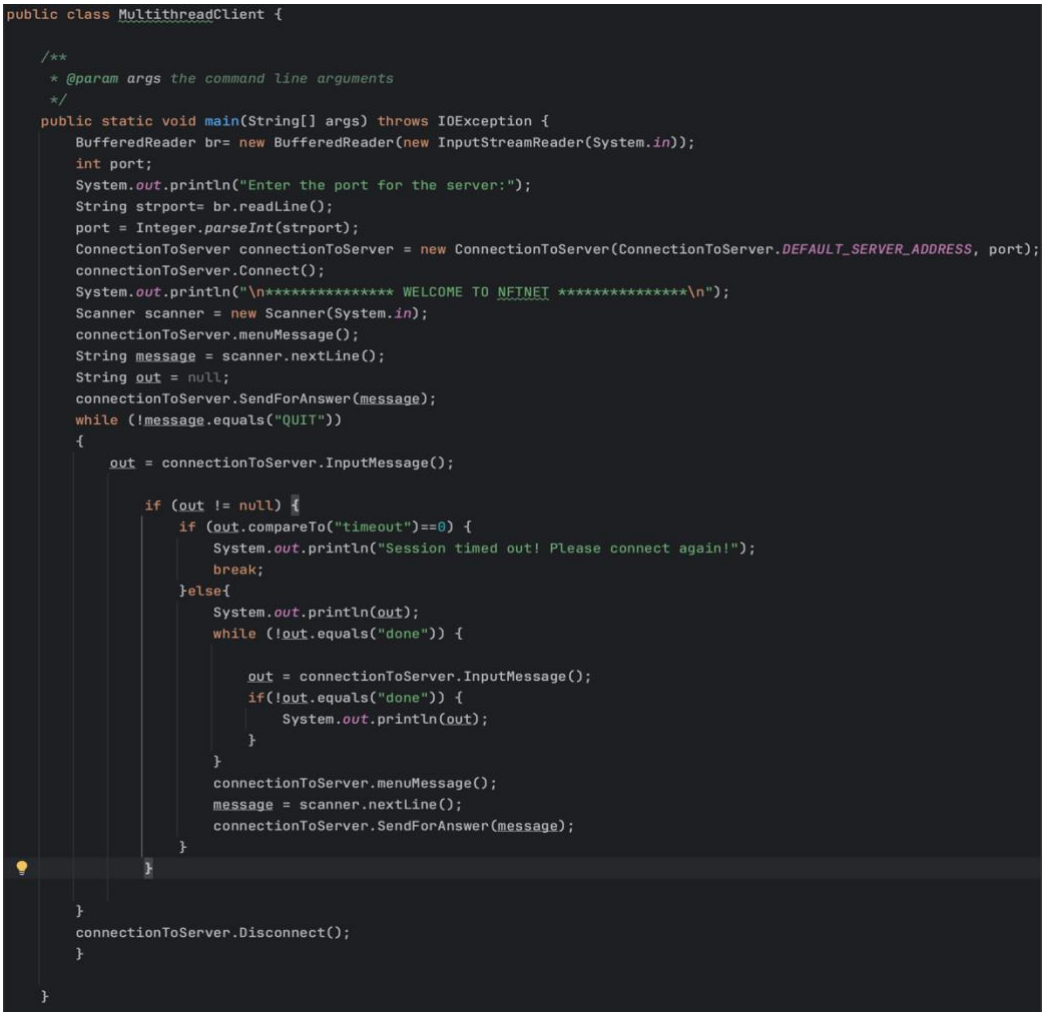
The API and fetching side are these two functions that mentioned above. fullList function is getting an integer input for the page number and fetches that page number with per_page value of 250. There are 14 pages with this setup and that is why the for loop at the "getRequest" function have a maxPage integer set to 14. With GET HTTP tool, it gets all the data and saves it to the "inLine" string which gets returned at the end of the function. This function is called 14 times for every list request.

```
1 usage
private String getSearchedNFT(String nftName) throws IOException{
    String BASE_URL = "https://api.coingecko.com/api/v3/nfts/";
    String urlString = BASE_URL + nftName;
    StringBuilder inLine = new StringBuilder();
    try {
        URL url = new URL(urlString);
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod("GET");
        conn.connect();

        int status = conn.getResponseCode();
        if (status != HttpURLConnection.HTTP_OK) {
            return "Error";
        }

        Scanner scanner = new Scanner(url.openStream());
        while (scanner.hasNext()) {
            inLine.append(scanner.nextLine());
        }
        scanner.close();
        //System.out.println(inLine.toString());
    } catch (IOException e) {
        return "Error";
    }
    return inLine.toString();
}
```

The getSearchedNFT function gets a string input and returns a string variable. The input is the name of the NFT that client is searching for. The data collection is same as the "fullList" but there is an error handling with the ".HTTP_OK". That line checks if there is a valid HTTP connection with the searched NFTs' URL. If not, immediately returns a string which is "Error". This is checked at the "getRequest" function part and if the return equals to "Error" the listFlag is set to 10, send an error message to client. The error computation without sending terminal error massages to client helps to get much smoother connection and communication for the client.

Client Implementation

For the client side, I modified the given example code just like the server side. The code asks for the port at first to connect to the server. Then there is a scanner function to initialize the getting input from client. Then there is a menu message that I created to guide the client what are the expected prompts and their functionalities. Then with the scanner, clients' prompt is saved and sent to the server with the "SendForAnswer" function that I modified in the connectionToServer class. After that a while loop starts until the client writes "QUIT". Until there is a QUIT input, the connection stays open with the help of the while loop. At first in the while loop, "InputMessage" function is called. The first line of the server output is saved in the out variable. Then the check for the time out begins, if the input string that comes from the server equals to "timeout", a session timeout error is displayed, and the message string is set to "QUIT" to close the connection without extra functions. If the first input from the server is not "timeout", another condition check is started again which is the null check of the input stream. If not, the message is displayed, and another loop starts for the list function. Until the last string came from the server is "done", every new line is printed out to client. Just for a cleaner look, there is an additional check for if the string is "done", do not print it out section.



```
public class MultithreadClient {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) throws IOException {
        BufferedReader br= new BufferedReader(new InputStreamReader(System.in));
        int port;
        System.out.println("Enter the port for the server:");
        String strport= br.readLine();
        port = Integer.parseInt(strport);
        ConnectionToServer connectionToServer = new ConnectionToServer(ConnectionToServer.DEFAULT_SERVER_ADDRESS, port);
        connectionToServer.Connect();
        System.out.println("\n*************** WELCOME TO NETNET **************\n");
        Scanner scanner = new Scanner(System.in);
        connectionToServer.menuMessage();
        String message = scanner.nextLine();
        String out = null;
        connectionToServer.SendForAnswer(message);
        while (!message.equals("QUIT"))
        {
            out = connectionToServer.InputMessage();

                if (out != null) {
                    if (out.compareTo("timeout")==0) {
                        System.out.println("Session timed out! Please connect again!");
                        break;
                    }else{
                        System.out.println(out);
                        while (!out.equals("done")) {

                            out = connectionToServer.InputMessage();
                            if(!out.equals("done")) {
                                System.out.println(out);
                            }
                        }
                        connectionToServer.menuMessage();
                        message = scanner.nextLine();
                        connectionToServer.SendForAnswer(message);
                    }
                }

        }
        connectionToServer.Disconnect();
        }

    }
```

```java
public void SendForAnswer(String message) {
        /*
        Sends the message to the server via PrintWriter
         */
        os.println(message);
        os.flush();
}

2 usages
public String InputMessage() throws IOException {
    String response = new String();
    response = is.readLine();

    return response;
}

2 usages
public void menuMessage(){
    System.out.println("\nIf you want to see the list of NFTs please just type 'list', if you want" +
            " a specific NFT's info, just write the 'id'.\n");
}
```

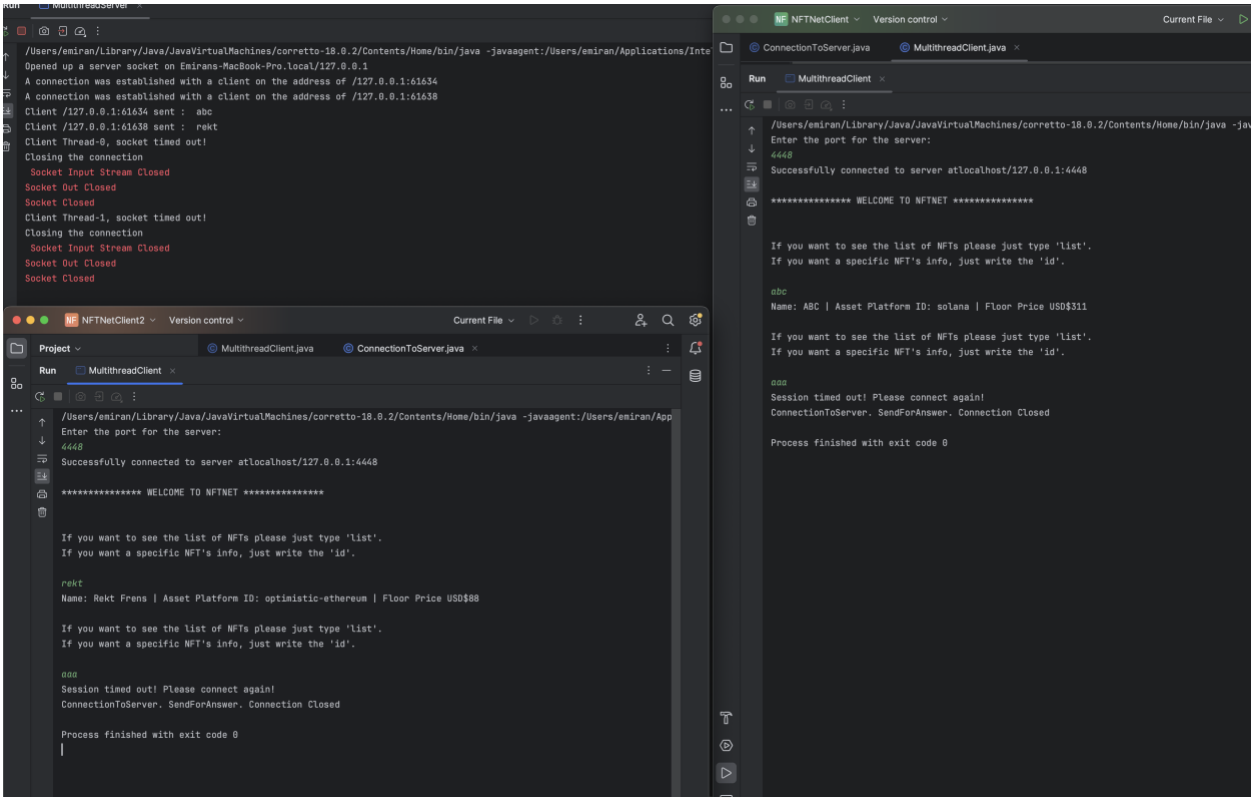
This is the changed and added functions from the “ConnectionToServer” class that shared in the example project files. Again, I used the main connection and disconnect parts since this is a simple TCP connection for the client side. The multi-threading part is at the server only. “sendForAnswer” function is changed just to sending the clients’ prompt to the server and there is an “InputMessage” function that reads the input stream and returns the line. The menu message is just to be informative about the prompts and the general functionality of the application.

Test Cases and Outputs

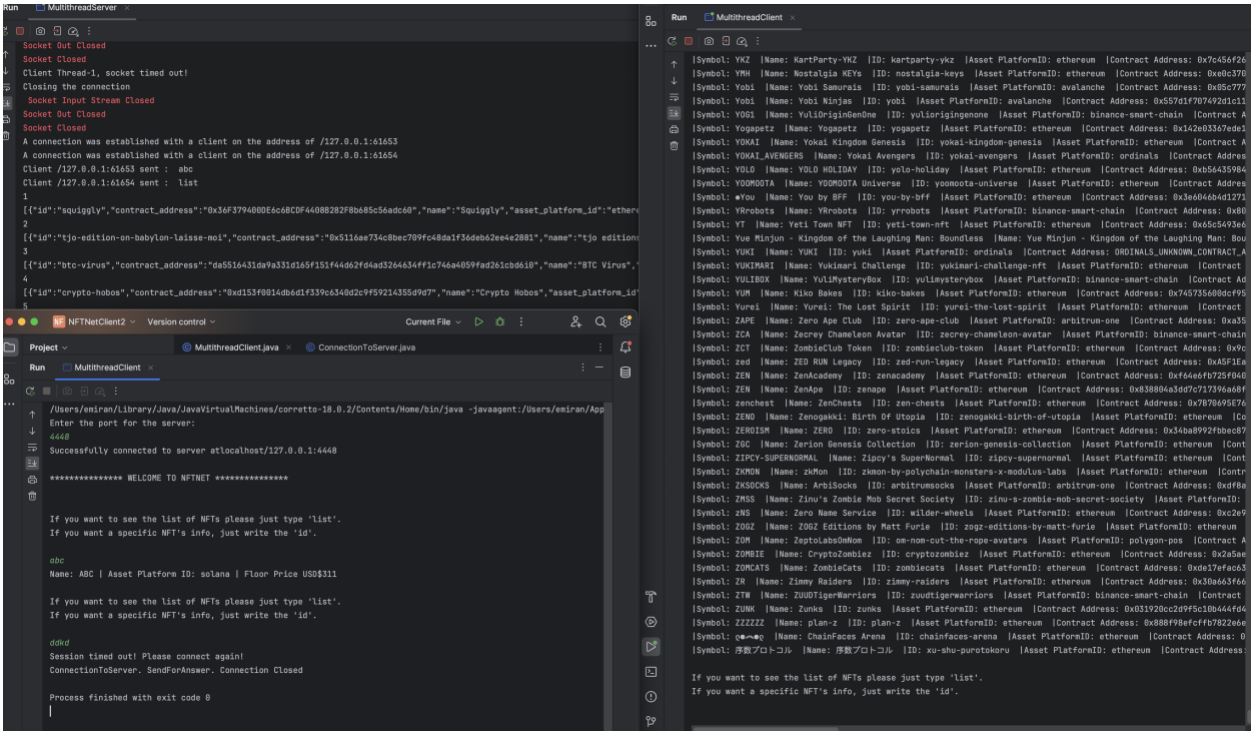
For testing the project, I created two identical client projects that runs separately to simulate multiple client connection at the same time, testing the multi-thread server application.

Some Case Examples:

No input line, timeout test with 2 different clients. Clients are trying to send but “timeout” line and the error is printed.



One client timed-out, other one still getting data on the same port. Also, the API fetching limit is reached so trying to get NFT id is giving the wait error line.



Invalid NFT ID requesting and QUIT function. Requesting multiple NFT ID’s and waiting until the timeout to get timeout error.