COMP 416/ELEC 416

Computer Networks

Prof. Öznur Özkasap

Project 1 – WeatNet

By Cansu Doğanay, Oğuz Bayhun, Murat Erdoğan

Objective: In this project we implemented a weather reporting network application by interacting with the application layer abstract programming interface (API) of OpenWeatherMap (openweathermap.org) using the knowledge we have learned in COMP416 such as, application layer software development, client/server protocol, application layer protocol principles, socket programming, and multithreading.

Our WeatNet application has 3 main parts:

· ∗ Server by Murat Erdoğan (Student 2 according to the suggested task distribution)

· ∗ Client by Cansu Doğanay (Student 1 according to the suggested task distribution)

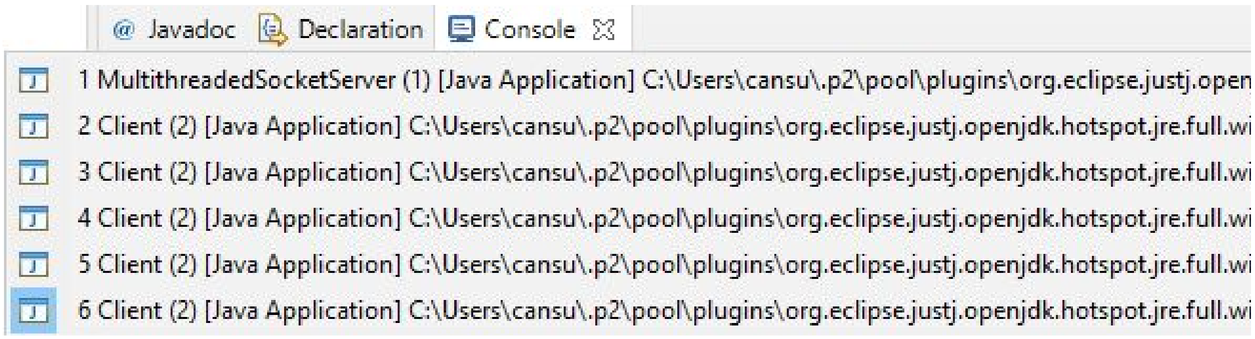
· ∗ API by Oğuz Bayhun (Student 3 according to the suggested task distribution)

Throughout the whole project, we worked as a great team. We always took care of our issues and tried to solve all problems, errors together.

Server Part

a. Class MultiThreadedSocketServer
In this class we created a ServerSocket with the port number 8888.
When we open the server, it will continuously listen for a connection to be made to this socket and accepts it. Then creates a thread with the given counter number and that socket. For debug purposes we also have an arraylist of these threads.

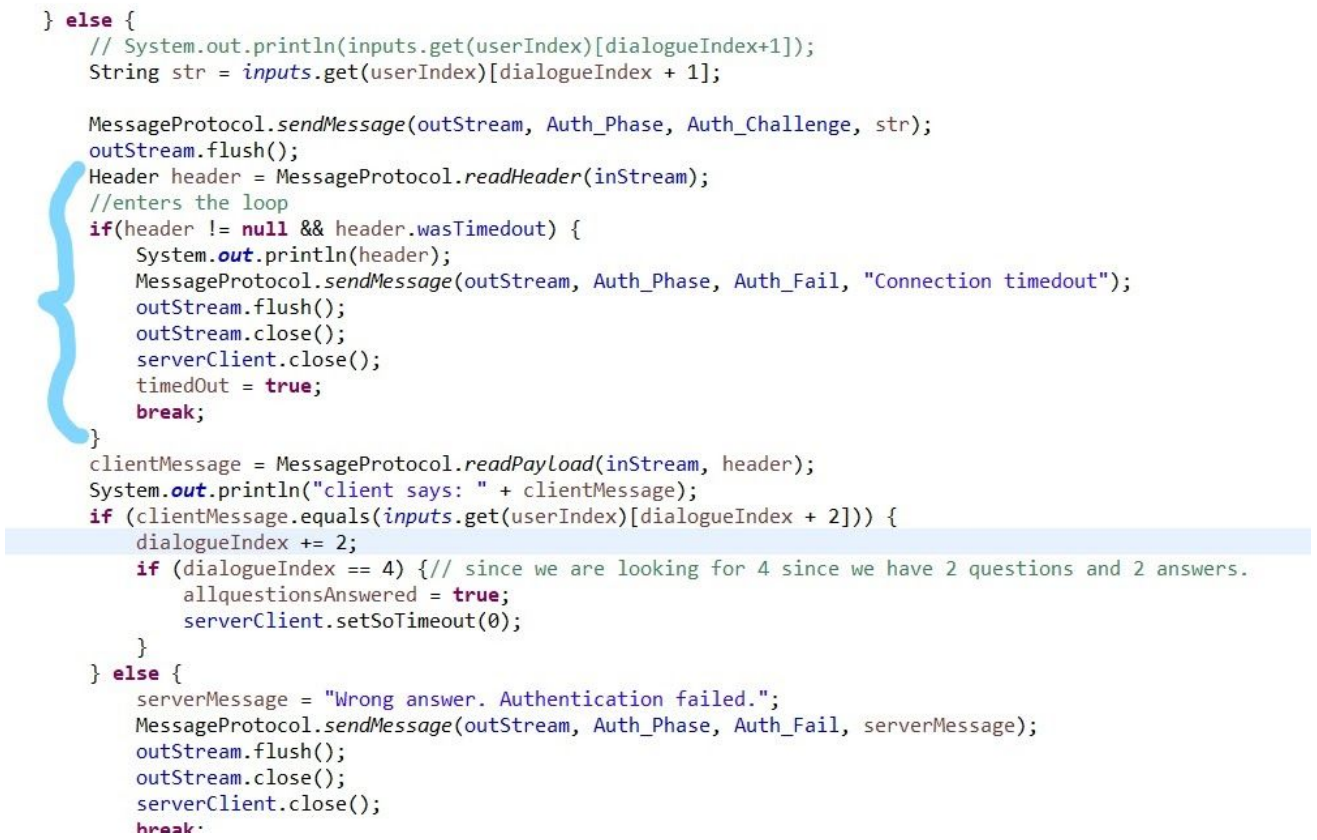
b. Class ServerClientThread
This class allows the server to handle more than a single client. We are able to create 5 and more clients at the same time. (Since the project expects 5 clients in maximum we created 5 user information.) As you can see in the figure below 5 different client applications can connect to our server



```
1 MultithreadedSocketServer (1) [Java Application] C:\Users\cansu\.p2\pool\plugins\org.eclipse.justj.open
2 Client (2) [Java Application] C:\Users\cansu\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.wi
3 Client (2) [Java Application] C:\Users\cansu\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.wi
4 Client (2) [Java Application] C:\Users\cansu\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.wi
5 Client (2) [Java Application] C:\Users\cansu\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.wi
6 Client (2) [Java Application] C:\Users\cansu\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.wi
```

First this class gets all the user information in the text file that we created (see client getUsers()). Then checks if the client entered an existing users username. In a failed match the thread disconnects the client. If an existing username is entered. Then the thread starts the questioning phase.

Figure below shows the questioning part in the code



```java
} else {
    // System.out.println(inputs.get(userIndex)[dialogueIndex+1]);
    String str = inputs.get(userIndex)[dialogueIndex + 1];

    MessageProtocol.sendMessage(outStream, Auth_Phase, Auth_Challenge, str);
    outStream.flush();
    Header header = MessageProtocol.readHeader(inStream);
    //enters the loop
    if(header != null && header.wasTimedout) {
        System.out.println(header);
        MessageProtocol.sendMessage(outStream, Auth_Phase, Auth_Fail, "Connection timedout");
        outStream.flush();
        outStream.close();
        serverClient.close();
        timedOut = true;
        break;
    }
    clientMessage = MessageProtocol.readPayload(inStream, header);
    System.out.println("client says: " + clientMessage);
    if (clientMessage.equals(inputs.get(userIndex)[dialogueIndex + 2])) {
        dialogueIndex += 2;
        if (dialogueIndex == 4) {// since we are looking for 4 since we have 2 questions and 2 answers.
            allquestionsAnswered = true;
            serverClient.setSoTimeout(0);
        }
    } else {
        serverMessage = "Wrong answer. Authentication failed.";
        MessageProtocol.sendMessage(outStream, Auth_Phase, Auth_Fail, serverMessage);
        outStream.flush();
        outStream.close();
        serverClient.close();
        break;
```

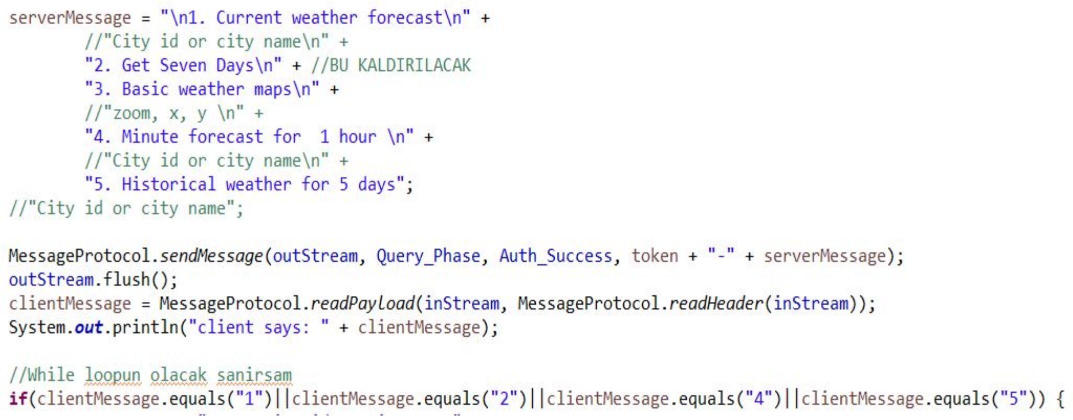
(The timeout checking in the code is marked with blue, if we get a header that is not null but timed out we send an Auth_fail message with the payload “connection timed out”) We set the timeout duration by the setSoTimeout(int); method for the socket which allows us to stop reading the input stream after a certain time.

In this phase there is a timeout duration (we decided to put a 10 second duration) if the timeout duration is expired the server sends a auth_fail message to the client (further info in client part). We have 3 (2 in the picture changed this later) questions for each user in our implementation. If an answer is wrong the server will send an auth_fail message If all the questions are answered correctly the user will move to the query phase.

Query Phase

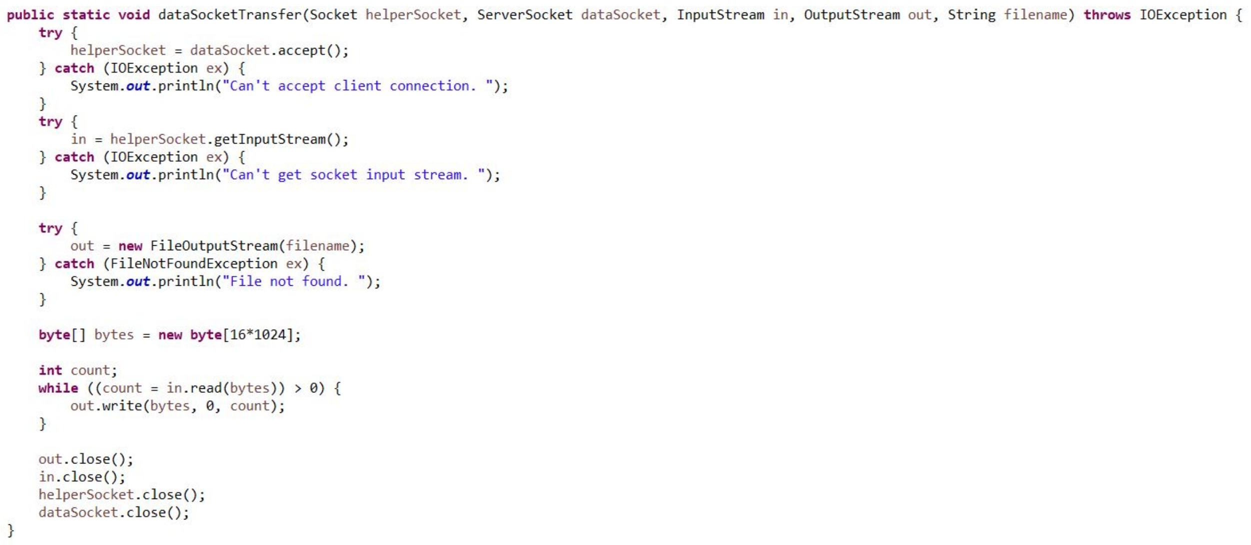
1. First we create a new ServerSocket with the name data socket, we will use this socket for the json file transmission.
2. The server will create an on-the-go token with the combination of the users username and the name of the device. This token will be used to create the requested file names and to check if the current user is authenticated.

3. Client enters its requested type of weather information from a range of numbers 1 to 5



```
serverMessage = "\n1. Current weather forecast\n" +
        //"City id or city name\n" +
        "2. Get Seven Days\n" + //BU KALDIRILACAK
        "3. Basic weather maps\n" +
        //"zoom, x, y \n" +
        "4. Minute forecast for  1 hour \n" +
        //"City id or city name\n" +
        "5. Historical weather for 5 days";
//"City id or city name";

MessageProtocol.sendMessage(outStream, Query_Phase, Auth_Success, token + "-" + serverMessage);
outStream.flush();
clientMessage = MessageProtocol.readPayload(inStream, MessageProtocol.readHeader(inStream));
System.out.println("client says: " + clientMessage);

//While loopun olacak sanirsam
if(clientMessage.equals("1")||clientMessage.equals("2")||clientMessage.equals("4")||clientMessage.equals("5")) {
```

4. Then based on the users input we return the requested weather information from the API using our data socket
   We have a method called dataSocketTransfer which takes a socket, server socket, input stream, output stream and a string as the filename (this filename is created with the token). This method transfers the downloaded file from server to client.

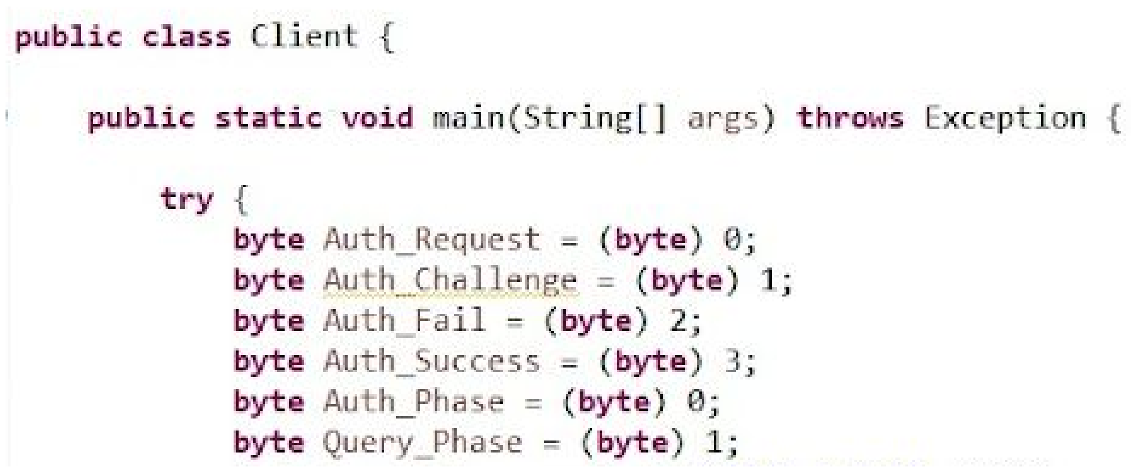


```
public static void dataSocketTransfer(Socket helperSocket, ServerSocket dataSocket, InputStream in, OutputStream out, String filename) throws IOException {
    try {
        helperSocket = dataSocket.accept();
    } catch (IOException ex) {
        System.out.println("Can't accept client connection. ");
    }
    try {
        in = helperSocket.getInputStream();
    } catch (IOException ex) {
        System.out.println("Can't get socket input stream. ");
    }

    try {
        out = new FileOutputStream(filename);
    } catch (FileNotFoundException ex) {
        System.out.println("File not found. ");
    }

    byte[] bytes = new byte[16*1024];

    int count;
    while ((count = in.read(bytes)) > 0) {
        out.write(bytes, 0, count);
    }

    out.close();
    in.close();
    helperSocket.close();
    dataSocket.close();
}
```

Client Part

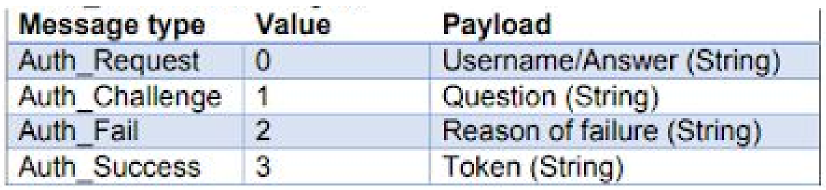
Authentication Phase

a. Message Format
   We have created our own Message Protocol (as a separate class) and use it throughout the project, especially for communication of client(s) and the server.



```
public class Client {

    public static void main(String[] args) throws Exception {

        try {
            byte Auth_Request = (byte) 0;
            byte Auth_Challenge = (byte) 1;
            byte Auth_Fail = (byte) 2;
            byte Auth_Success = (byte) 3;
            byte Auth_Phase = (byte) 0;
            byte Query_Phase = (byte) 1;
```

In Client class, we created the variables for Authentication as stated in the project explanation PDF,

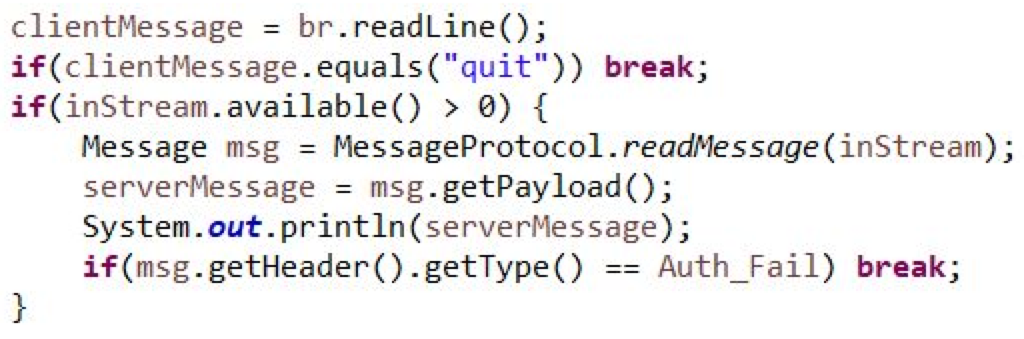


Message type	Value	Payload
Auth_Request	0	Username/Answer (String)
Auth_Challenge	1	Question (String)
Auth_Fail	2	Reason of failure (String)
Auth_Success	3	Token (String)

We have 4 message types and two phases (authentication and querying) as shown in the above figures.

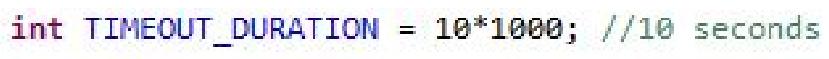
b. Timeout Mechanism

We gave clients 10 seconds to answer the challenges sent by the server. If they cannot, they will receive an Authentication_Fail message. If the client tries to send any messages after they are timed out they are shut down.



```
clientMessage = br.readLine();
if(clientMessage.equals("quit")) break;
if(inStream.available() > 0) {
    Message msg = MessageProtocol.readMessage(inStream);
    serverMessage = msg.getPayload();
    System.out.println(serverMessage);
    if(msg.getHeader().getType() == Auth_Fail) break;
}
```

In ServerClientThread class we specified the timeout duration as 10 seconds



```
int TIMEOUT_DURATION = 10*1000; //10 seconds
```

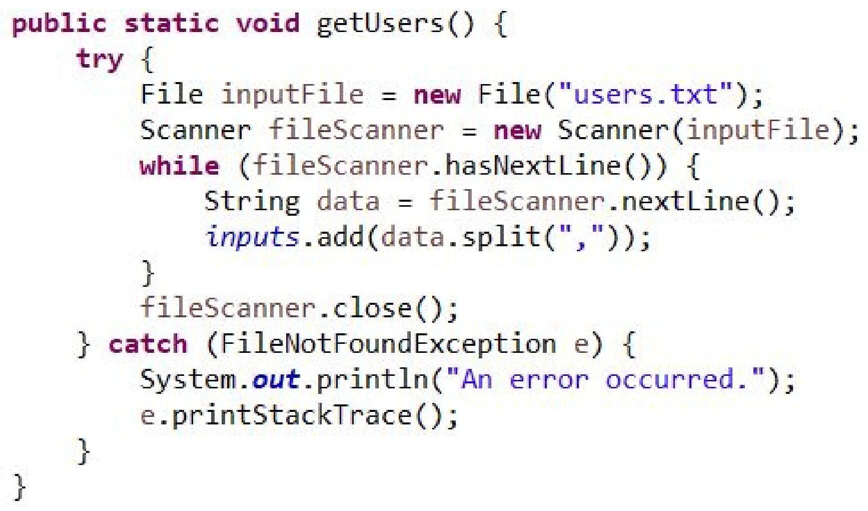
c. Implementation details

Client sends an Authentication request to the Server. First it sends its username. We created a users.txt file which includes 5 users, the first column of the file represents the usernames of the users, if the client sends one of the existing usernames, the server starts to send challenges. If the username is not in the users.txt file, or the client does not reply the challenge sent by the server during the specified time period (10 seconds), or the answer is not correct; then the server sends an Authentication_Fail message to the server according to the reason of the failure, and client is disconnected. Otherwise, it finally gets authenticated.

Helper Methods:

getUsers()

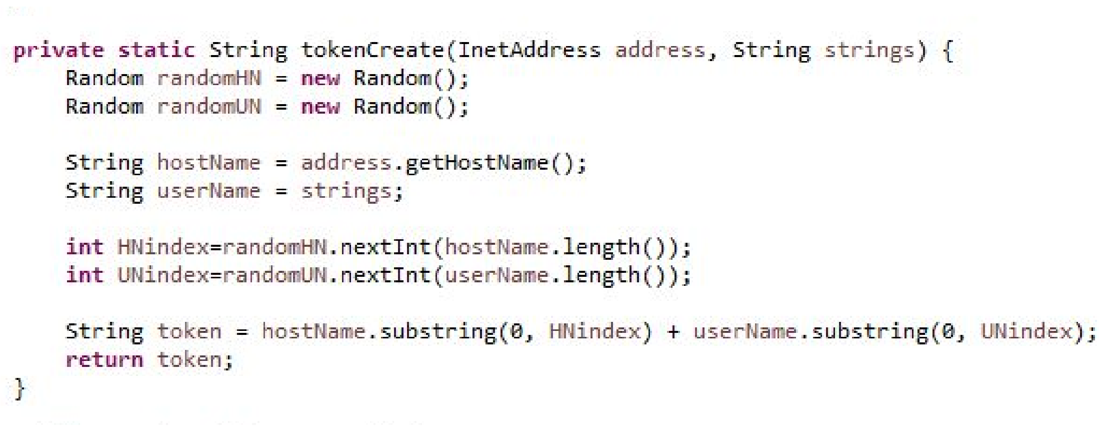
getUsers() method is used to get usernames in the txt file in order to figure out whether he/she is an existing user or not. The first word of each line (e.g., first column of the 2D array) represents the usernames.



```
public static void getUsers() {
    try {
        File inputFile = new File("users.txt");
        Scanner fileScanner = new Scanner(inputFile);
        while (fileScanner.hasNextLine()) {
            String data = fileScanner.nextLine();
            inputs.add(data.split(","));
        }
        fileScanner.close();
    } catch (FileNotFoundException e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
    }
}
```

tokenCreate(InetAddress, String)

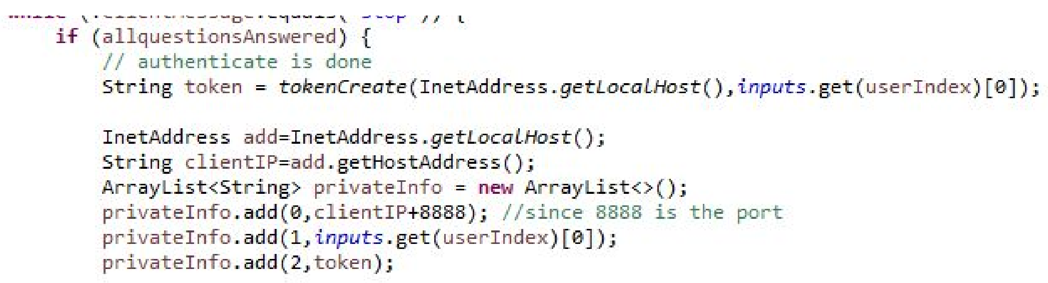
Since we need to create unique tokens for clients each time, we wrote a method that gets usernames and addresses of clients and creates random tokens for them. The randomness is based on the length of the variables.



```java
private static String tokenCreate(InetAddress address, String strings) {
    Random randomHN = new Random();
    Random randomUN = new Random();

    String hostName = address.getHostName();
    String userName = strings;

    int HNindex=randomHN.nextInt(hostName.length());
    int UNindex=randomUN.nextInt(userName.length());

    String token = hostName.substring(0, HNindex) + userName.substring(0, UNindex);
    return token;
}
```

By that way, we guarantee that each client will receive random and unique tokens in each run according to their host and usernames.

When authentication is complete,



```java
    if (allquestionsAnswered) {
        // authenticate is done
        String token = tokenCreate(InetAddress.getLocalHost(),inputs.get(userIndex)[0]);

        InetAddress add=InetAddress.getLocalHost();
        String clientIP=add.getHostAddress();
        ArrayList<String> privateInfo = new ArrayList<>();
        privateInfo.add(0,clientIP+8888); //since 8888 is the port
        privateInfo.add(1,inputs.get(userIndex)[0]);
        privateInfo.add(2,token);
```

We create an ArrayList to keep user information for each Client. The zero index keeps the client's IP address and port number, the first index keeps the username and the second index keeps the token. We can verify the user information by that way, during the API operations

API Part

Helper Methods

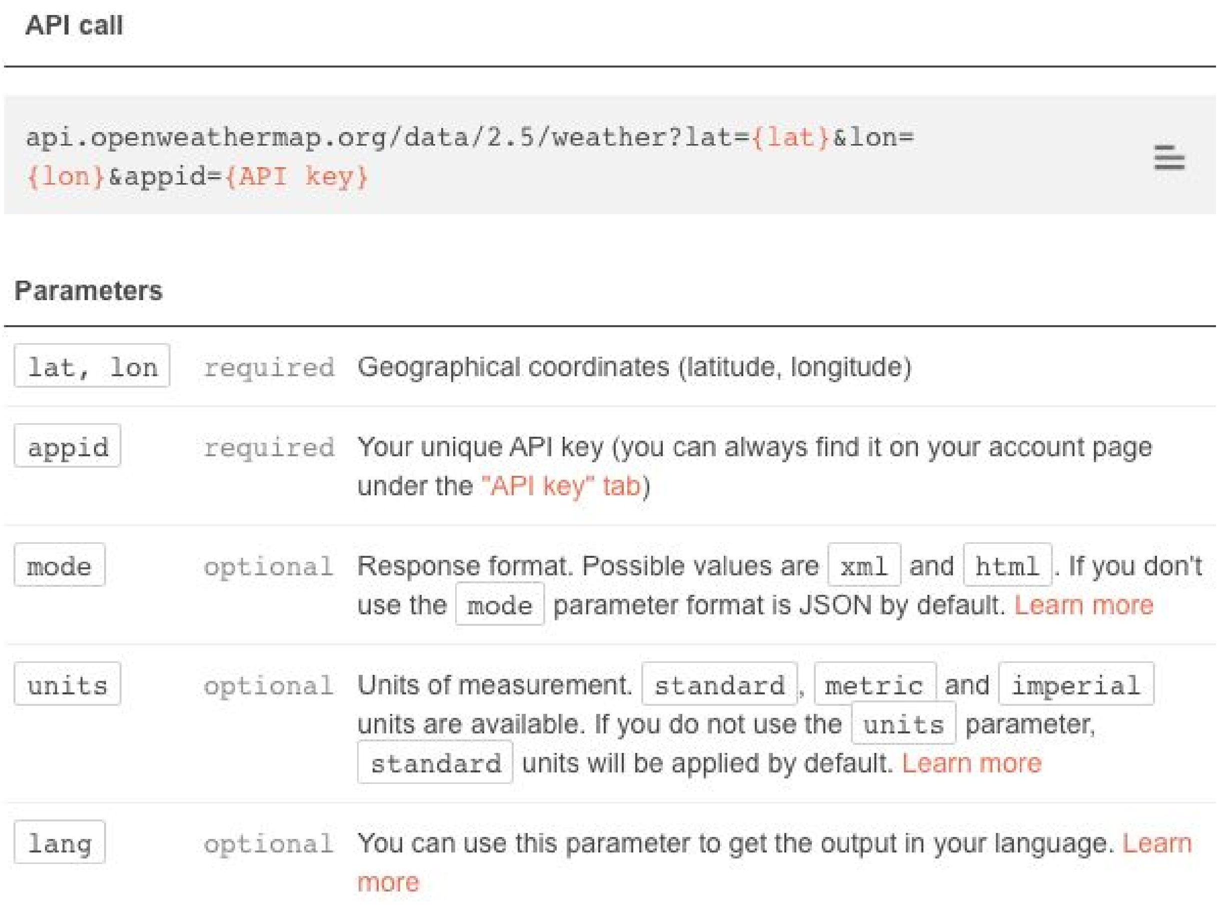
1. readJson

   *Parameters: **String** cityName, **int** cityID*
   This function takes two input names or id of the requested city, scans the cities.json file where we hold given cities in project description, returns the last, lon information of the matching city.
   Returns: {lat: LAT, lon: LON}

Weather Requests

1. getCurrentWeather

   *Parameters: **double** lat, **double** lon, **String** token*
   This function takes 3 parameters; lat, lon information of the city from readJson function and a string to hash file name. Since we received the id matching lat&lon from the cities we sent below call to OWM's current weather APi:



API call

```
api.openweathermap.org/data/2.5/weather?lat={lat}&lon=
{lon}&appid={API key}
```

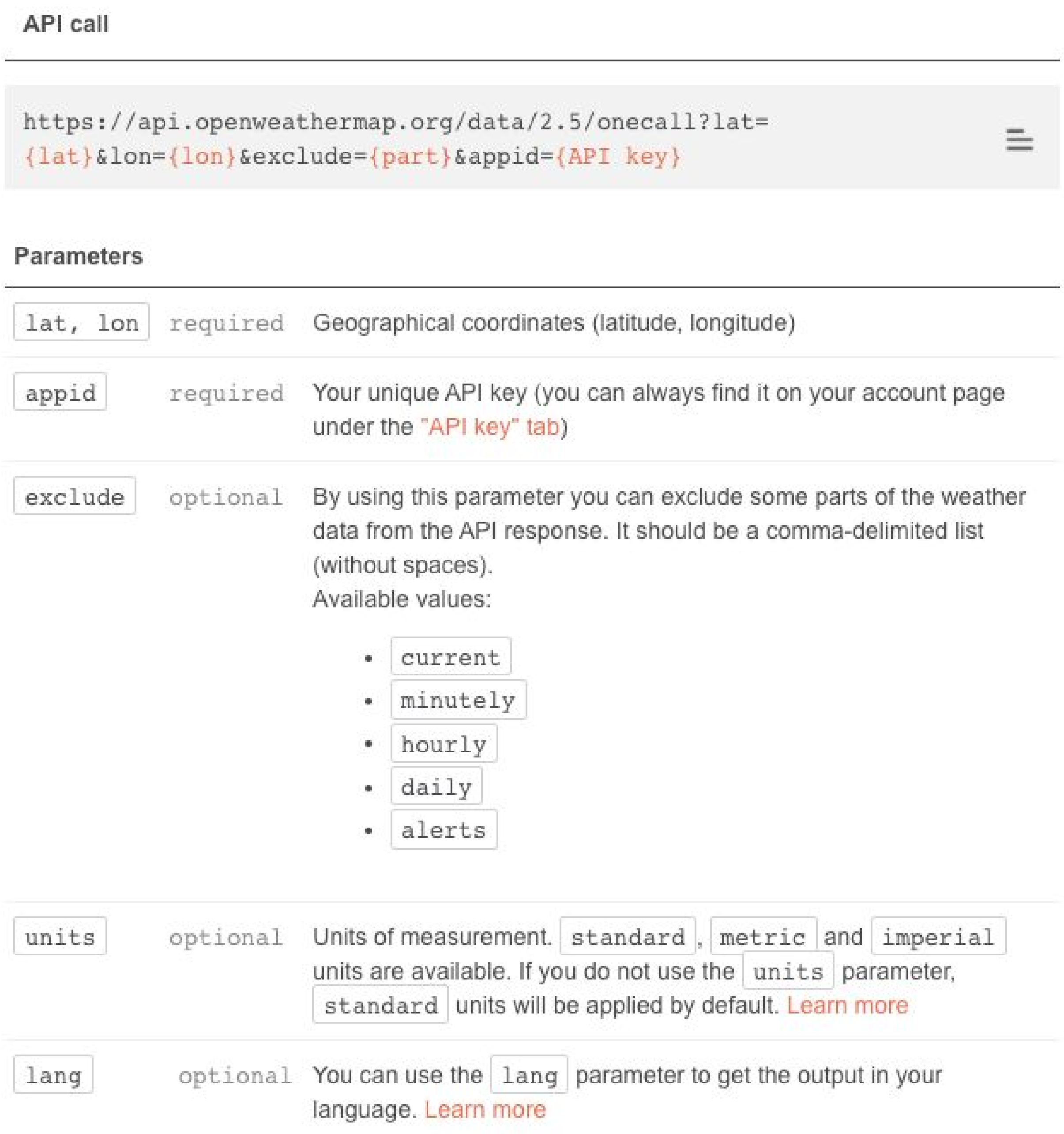
Parameters

`lat, lon`	required	Geographical coordinates (latitude, longitude)
`appid`	required	Your unique API key (you can always find it on your account page under the "API key" tab)
`mode`	optional	Response format. Possible values are `xml` and `html`. If you don't use the `mode` parameter format is JSON by default. Learn more
`units`	optional	Units of measurement. `standard`, `metric` and `imperial` units are available. If you do not use the `units` parameter, `standard` units will be applied by default. Learn more
`lang`	optional	You can use this parameter to get the output in your language. Learn more

   We fill required fields with the data we have and switch units to metric. APi key also added into the url. File writing procedure starts with returned data and stored as current+HashValue.json. Response is returned according to error or success status.

2. getMinuteForecast

   *Parameters: **double** lat, **double** lon, **String** token*
   Structure and parameters are actually same with the above function but this time we use different api endpoint of OMW which is One Call APi;



API call

```
https://api.openweathermap.org/data/2.5/onecall?lat=
{lat}&lon={lon}&exclude={part}&appid={API key}
```

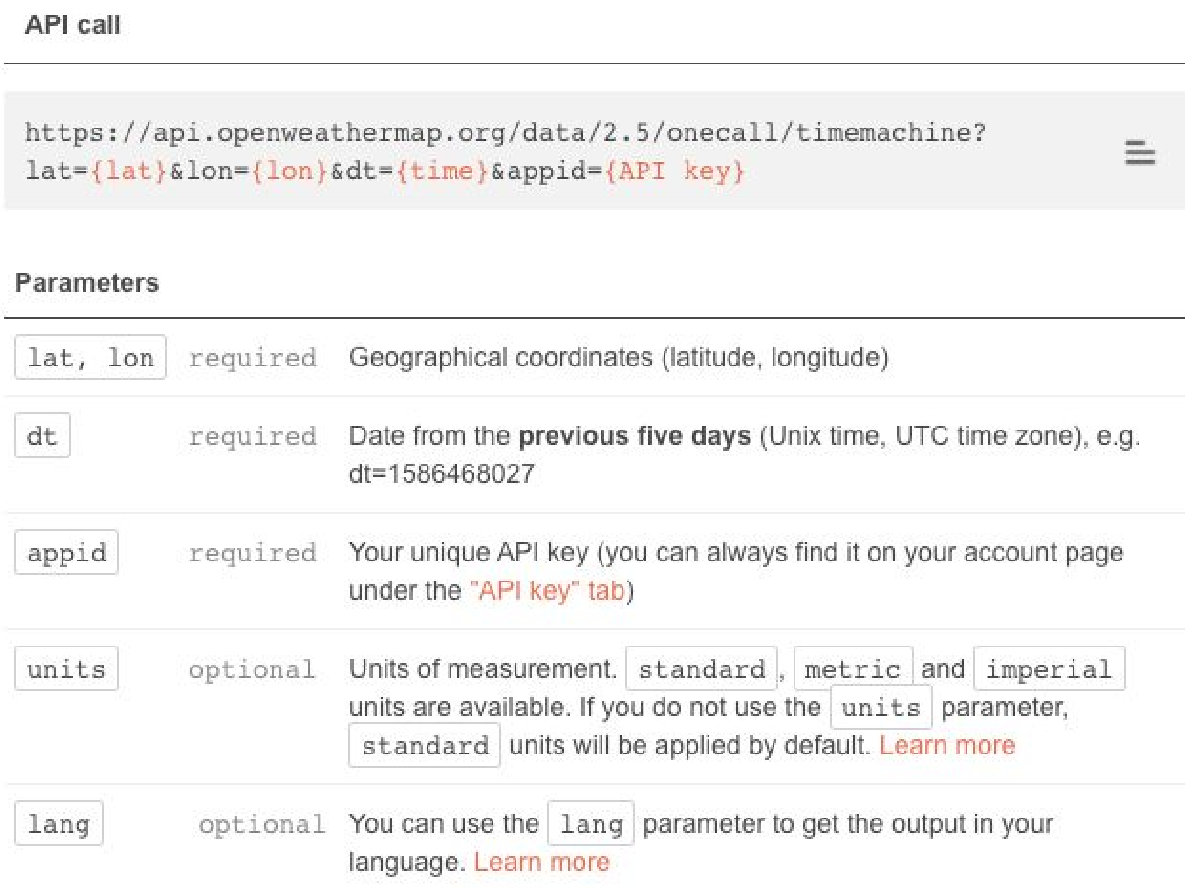
Parameters

`lat, lon`	required	Geographical coordinates (latitude, longitude)
`appid`	required	Your unique API key (you can always find it on your account page under the "API key" tab)
`exclude`	optional	By using this parameter you can exclude some parts of the weather data from the API response. It should be a comma-delimited list (without spaces). Available values: • `current` • `minutely` • `hourly` • `daily` • `alerts`
`units`	optional	Units of measurement. `standard`, `metric` and `imperial` units are available. If you do not use the `units` parameter, `standard` units will be applied by default. Learn more
`lang`	optional	You can use the `lang` parameter to get the output in your language. Learn more

Again we fill the lat, lon field with received data from readJson function. Since our request is for receiving minutely data we set exclude to minituley, and switch units from standard to metric. Response is returned according to both api response check and file creation status. File should be created with a name: *minute_HashValue.json.*

3. get5Days

   *Parameters: **double** lat, **double** lon, **String** token*
   For 5 days of historical data request we again used the provided One Call APi's *Historical Weather Data* section.


API call

```
https://api.openweathermap.org/data/2.5/onecall/timemachine?
lat={lat}&lon={lon}&dt={time}&appid={API key}
```

Parameters

`lat, lon`	required	Geographical coordinates (latitude, longitude)
`dt`	required	Date from the **previous five days** (Unix time, UTC time zone), e.g. dt=1586468027
`appid`	required	Your unique API key (you can always find it on your account page under the "API key" tab)
`units`	optional	Units of measurement. `standard`, `metric` and `imperial` units are available. If you do not use the `units` parameter, `standard` units will be applied by default. Learn more
`lang`	optional	You can use the `lang` parameter to get the output in your language. Learn more

We have few helper lines to send this request due to dt parameter. Dt parameter should be in unix time so function first takes current time in milliseconds and stores on a variable as a Date. Then we divide the variable with 1000 and store it as String(unix time value) in a millis(second variable). Request parameters again filled with the data we received from readJson function and dt parameter set to our millis. Response is returned to the client as json file and message to console. File name generated is : historical_HashValue.json

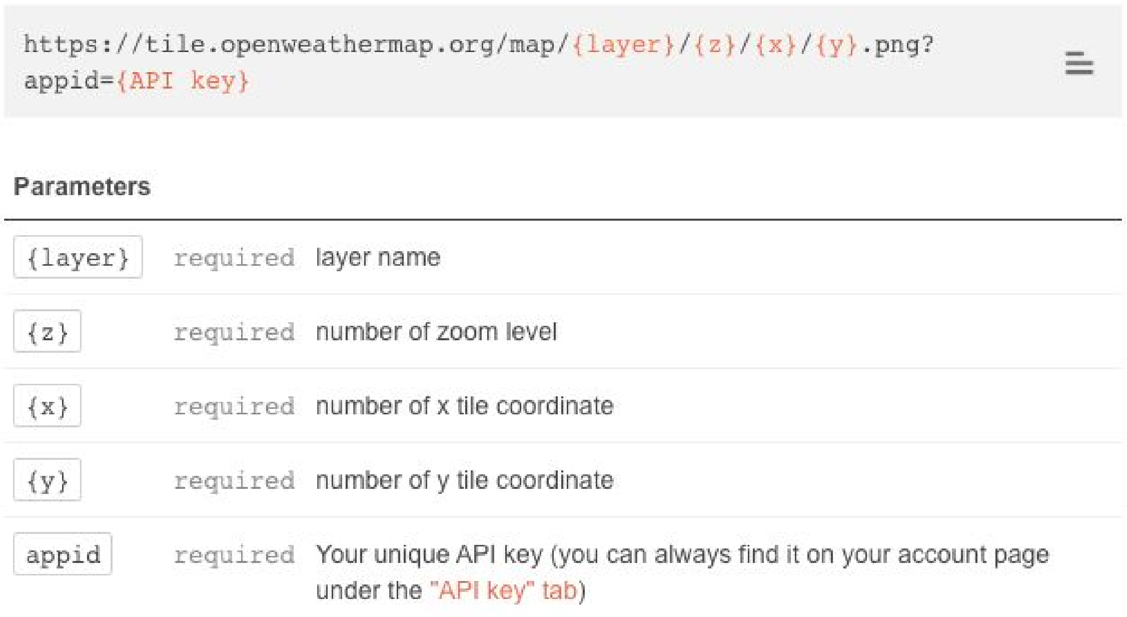
4. getSevenDays

   *Parameters: **double** lat, **double** lon, **String** token*
   This function uses the same api endpoint of OMW One Call, the only difference is exclude parameter value. For receiving seven days data we set the exclude parameter to "daily" as described in OMW's documentation. Response is returned same as previous methods. File name generated is : DailySeven_+HashValue.json

5. getMap

   *Parameters: **String** layerName, **int** z, **int** x **int** y, **String** token*
   This function takes 5 different parameters; layername specifies map type client prefers, z is the zoom level of the map, x and y coordinates are integer and states tile coordinates.

These inputs are asked to the user and the user returns the parameter values separated by comma.
OWM has a different endpoint which is open source Weather Maps 1.0.



```
https://tile.openweathermap.org/map/{layer}/{z}/{x}/{y}.png?
appid={API key}
```

Parameters

`{layer}`	required	layer name
`{z}`	required	number of zoom level
`{x}`	required	number of x tile coordinate
`{y}`	required	number of y tile coordinate
`appid`	required	Your unique API key (you can always find it on your account page under the "API key" tab)

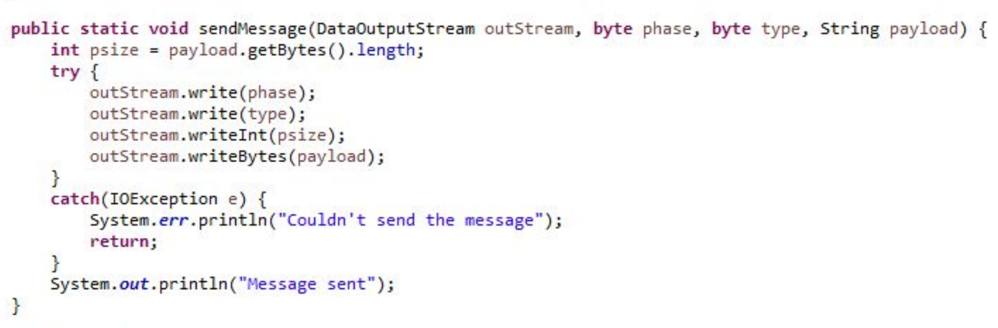
Client has 4 options for layer as mentioned in OWM documentation; clouds_new, temp_new, pressure_new, precipitation_new, wind_new.

API response is downloaded as png file with the name "basic_HashValue.png"

File Transfer Mechanism

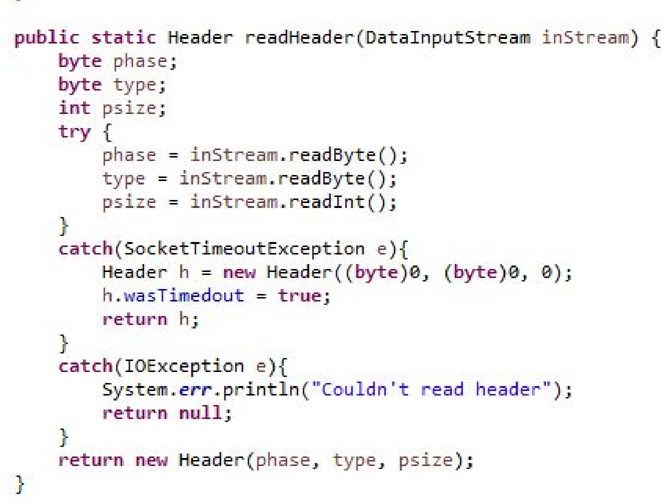
After a client authenticates the system, we ask the client what he/she wants to learn via the Command Socket. Then the Data Socket opens and the file transfer process starts.

Message Protocol



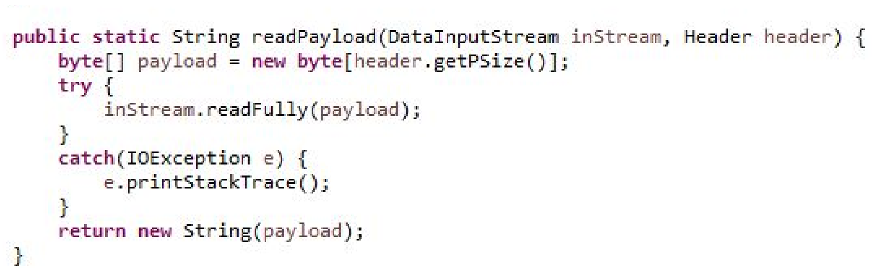
```
public static void sendMessage(DataOutputStream outStream, byte phase, byte type, String payload) {
    int psize = payload.getBytes().length;
    try {
        outStream.write(phase);
        outStream.write(type);
        outStream.writeInt(psize);
        outStream.writeBytes(payload);
    }
    catch(IOException e) {
        System.err.println("Couldn't send the message");
        return;
    }
    System.out.println("Message sent");
}
```

Sends message in the wanted format of 1byte+1byte+int(payload_size_in_bytes)+string payload



```
public static Header readHeader(DataInputStream inStream) {
    byte phase;
    byte type;
    int psize;
    try {
        phase = inStream.readByte();
        type = inStream.readByte();
        psize = inStream.readInt();
    }
    catch(SocketTimeoutException e){
        Header h = new Header((byte)0, (byte)0, 0);
        h.wasTimedout = true;
        return h;
    }
    catch(IOException e){
        System.err.println("Couldn't read header");
        return null;
    }
    return new Header(phase, type, psize);
}
```

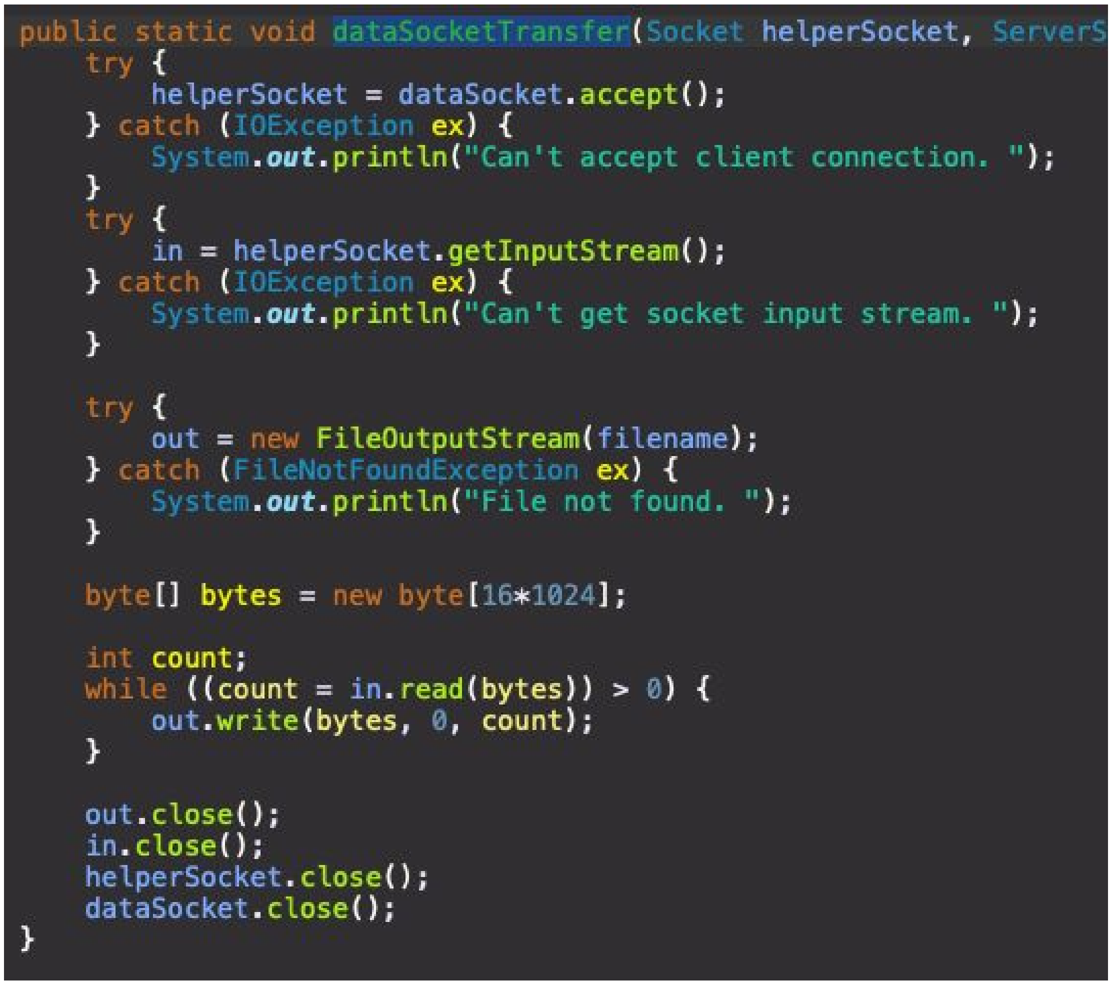
Reads header of the message



```
public static String readPayload(DataInputStream inStream, Header header) {
    byte[] payload = new byte[header.getPSize()];
    try {
        inStream.readFully(payload);
    }
    catch(IOException e) {
        e.printStackTrace();
    }
    return new String(payload);
}
```

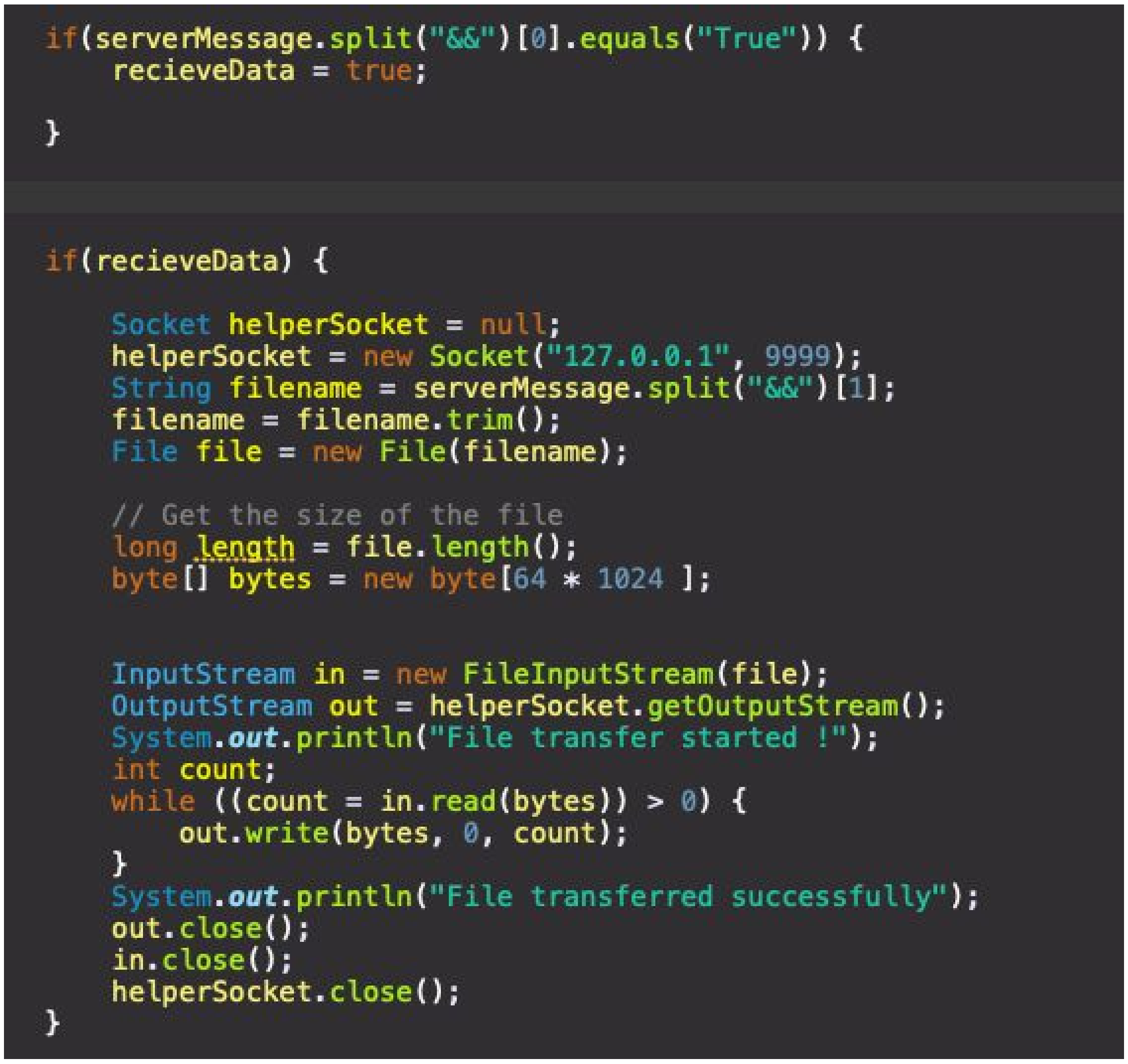
Reads payload of the message

Data Protocol



```
public static void dataSocketTransfer(Socket helperSocket, ServerS
    try {
        helperSocket = dataSocket.accept();
    } catch (IOException ex) {
        System.out.println("Can't accept client connection. ");
    }
    try {
        in = helperSocket.getInputStream();
    } catch (IOException ex) {
        System.out.println("Can't get socket input stream. ");
    }

    try {
        out = new FileOutputStream(filename);
    } catch (FileNotFoundException ex) {
        System.out.println("File not found. ");
    }

    byte[] bytes = new byte[16*1024];

    int count;
    while ((count = in.read(bytes)) > 0) {
        out.write(bytes, 0, count);
    }

    out.close();
    in.close();
    helperSocket.close();
    dataSocket.close();
}
```

When we get enough information from the client to send a request to OWM, we start file transfer after we make sure that response has been received from OMW. This dataSocketTransfer function takes a helper socket which we created after the user is authorized, main Server Socket, current input and output streams and the file name which includes current user's token and starts transferring the file to the client. On client side:



```java
if(serverMessage.split("&&")[0].equals("True")) {
    recieveData = true;

}


if(recieveData) {

    Socket helperSocket = null;
    helperSocket = new Socket("127.0.0.1", 9999);
    String filename = serverMessage.split("&&")[1];
    filename = filename.trim();
    File file = new File(filename);

    // Get the size of the file
    long length = file.length();
    byte[] bytes = new byte[64 * 1024 ];


    InputStream in = new FileInputStream(file);
    OutputStream out = helperSocket.getOutputStream();
    System.out.println("File transfer started !");
    int count;
    while ((count = in.read(bytes)) > 0) {
        out.write(bytes, 0, count);
    }
    System.out.println("File transferred successfully");
    out.close();
    in.close();
    helperSocket.close();
}
```

Since that data socket port should only work during the file transfers; we send a trigger message from server to client saying "TRUE" and when this message is received by the client it sets receiveData boolean to true and client starts receiving the file.

Testing Method

1. Enter username: mustafa → Fail (Invalid user. Authentication failed.)
2. Enter username: cansu → Success
   2.1.a. Enter: red → Fail (Wrong answer. Authentication failed.)
   2.1.b. Enter: blue → Success
   2.1.c. Wait 10 seconds and enter an answer → Fail (Connection timed out.)
   2.2.a. Enter: virgo → Success
   2.3.a. Enter: hamburger → Success
3. Enter: 1-5 → Success
   3.a. Enter: anything rather than 1-5 → Fail
4. If 1-2-4-5 Enter: Malatya or ID given in cities.json → Success

4.a If 3 Enter: temp_new,3,2,2 → Success
4.b Enter: anything else → Fail