COMP 416 – Computer Networks

Project #1

Due: 02 November, 2020 @ 11:59pm (Late submissions will <u>not</u> be accepted).

Submission of the project deliverables is via Blackboard.

Note: This is a group-oriented project, you are encouraged to form groups of 3 students, we suggest you a fair task distribution at the end of this project description.

WeatNet: Development of Weather Reporting Network Application

Introduction and Motivation:

This project work is about the application **layer** of the network protocol stack. It involves application layer software development, client/server protocol, application layer protocol principles, socket programming, and multithreading.

Through this project, you are going to develop a weather reporting network application (WeatNet) by interacting with the application layer abstract programming interface (API) of OpenWeatherMap (openweathermap.org). The project will require you to work with the following APIs for information extraction from the OWM web server:

1. Current Weather forecast
2. Daily forecast for 7 days
3. Basic Weather maps
4. Minute forecast for 1 hour
5. Historical Weather for 5 days

This API provides you access to the weather data which will be subsequently accessed by the clients.

Project Overview:

In this project, you are asked to develop a weather reporting network application based on the client/server model. WeatNet server provides two types of TCP connections to

interact with the clients: One connection for exchanging the protocol commands, and one for data transfers. Fig.1 shows connections for a sample WeatNet server and client interactions. As shown in this figure, the WeatNet server also takes the responsibility of interacting with a OpenWeatherNet web server using the OpenWeatherMap API.

Figure 1. The OWM Client/Server connections.

Implementation Details:

The WeatNet reporting application has three main components:

- Interaction of the OpenWeatherMap (OWM) and the server
- Server side of the application
- Client side of the application

It is pertinent to note that the free subscription for OWM allows for up to 60 API calls per minute and a total of 1 million calls in a month. The developed application and testing including for demonstration must keep these limits in perspective.

Phases:

This project has two phases: Authentication phase, and querying phase. During the authentication phase, the client provides its username and answers a series of secret questions to prove its identity. The protocol (the message flow, message types and their formats) is provided in the “Authentication” section. The groups need only to implement the provided protocol. During the querying phase, the authenticated clients communicate with the server to retrieve weather information conforming to the specifications that will be explained in the following sections. Unlike the authentication phase, we do not provide you a protocol but expect you to design your own. This will be a part of your report. You may use the authentication protocol as a starting point and build from there.

City List:
The WeatNet reporting will be done for the following cities:

1. {'id': 745044, 'name': 'Istanbul', 'state': '', 'country': 'TR', 'coord': {'lon': 28.949659, 'lat': 41.01384}}

2. {'id': 740264, 'name': 'Samsun', 'state': '', 'country': 'TR', 'coord': {'lon': 36.330002, 'lat': 41.286671}}
3. {'id': 315201, 'name': 'Eskişehir', 'state': '', 'country': 'TR', 'coord': {'lon': 31.16667, 'lat': 39.666672}}
4. {'id': 323784, 'name': 'Ankara', 'state': '', 'country': 'TR', 'coord': {'lon': 32.833328, 'lat': 39.916672}}
5. {'id': 304919, 'name': 'Malatya', 'state': '', 'country': 'TR', 'coord': {'lon': 38.0, 'lat': 38.5}}
6. {'id': 750268, 'name': 'Bursa', 'state': '', 'country': 'TR', 'coord': {'lon': 29.08333, 'lat': 40.166672}}
7. {'id': 311044, 'name': 'İzmir', 'state': '', 'country': 'TR', 'coord': {'lon': 27.092291, 'lat': 38.462189}}

You can find the entire list of cities supported by OWM here. Please note that OWM may use multiple coordinates within the same city. The developed application must ensure that even if the names may be duplicated, the coordinates must be for the cities provided for the list. In this case, the city IDs will be useful when using the OWM API.

WeatNet Server side overview:

The server for the WeatNet application should:

1) Establish a connection with the OpenWeatherMap (OWM) web server using the API and download the specified weather metrics for all the cities specified for this project.

2) Authenticate any client before initiating any data exchange.

3) Allow multiple clients to connect using multi-threading with the same functionality.

4) Add a timestamp in each file before sending to any client while providing hashes of the files for error detection purposes..

The metrics categories for which the OWM API will be used are:

1. Current weather forecast
2. Weather triggers
3. Basic weather maps
4. Minute forecast for 1 hour
6. Historical weather for 5 days

All these metrics except the weather maps are to be downloaded in the JSON format as separate files whereas the basic maps will be downloaded as images (.jpg, .png etc.).

API Interactions:

For the interactions to take place, the client will pass the name/ID of the city and/or the weather triggers. All this information shall be conveyed between the client and the

server on the Command socket (the socket allocated to the client after acceptance and validation).

Based on this input, the server will parse the client input and use the API to extract the relevant information from the web server. This information will then be passed to the client in the form of a JSON/image file over the Data Socket. The server will be responsible for generating a hash value of each file before transmitting to the client and this will also be sent to the client for file verification over the Command Socket

The process at server side would be:

1) Create welcoming socket
2) Accept incoming client connections at the welcoming socket, simultaneously if needed using multi-threading.
3) Authenticate any incoming client connection requests **after acceptance**. Create an additional Data socket with the client if authenticated. Terminate connection if authentication fails. (Demonstration should present both scenarios in which clients are validated as well as rejected).
4) Decipher the requests coming in from the clients and download the required files from the web server. (The protocol for forwarding requests has to be developed by the groups themselves and explained in their reports. A similar format is provided in the Authentication part).
5) Generate the hash values of the file requested by the client.
6) Send the hash value of the required file over the Command socket.
7) Send the requisite file over the Data socket.
8) Terminate connection if a “file received” acknowledgment message is received from the client and no other files are requested within timeout duration.

The exact implementation of Data Socket is left to the choice of the groups. It can be a single socket shared by the clients through multi-threading or a dedicated socket for each client. The initiation of the Data Socket and the exact nature of how the parameters are passed to the client are also to be decided by the groups. The parameters of the Data socket are also to be conveyed over the Command socket if required.

Weatnet Client Side Overview:

This weather application envisions multiple clients interacting with a single server. For this application, the clients should:

1) Confirm its authenticity with the server
2) Be able to submit requests to the server.

3) Be able to receive data in form of JSON/image files from the server over the Data socket.

4) Verify the file based on its digital signature.

5) Display the JSON data in tabular form or display the image on the client side

Client-Server Interaction

The client side must take care of the parameters to be passed once requesting any metric.

The process at the client side will follow the given steps:

1) Initiate connection with the server over Command Socket

2) Authenticate based on the server requirements.

3) Receive the parameters for Data Socket and connect to that after authentication.

4) Pass the requests to the server

5) Receive the hash value of the file on the Command socket

6) Receive the files over the Data socket

7) Confirm if the hash value corresponds to the file

8) Request retransmit from the server if mismatch between hash value and file or failure to receive file (Relevant String should be displayed in terminal). A scenario for this step may be specifically designed for demonstration purposes.)

9) Display the files in the appropriate manner.

10) Terminate the connection in case an appropriate file is received and no other request forwarded within the timeout duration. (Appropriate tests for demonstration purposes should be developed.)

Authentication Phase

In our implementation, the server does not share the weather information with everyone. The client needs to be authenticated in order to be able to query the server, which will be done by a series of “challenges” (i.e., secret questions) instead of simple password-based authentication. After the authentication is done, the client will receive a “token” from the server. The token will act as a “proof” of the authenticity of the client. From that point on, the client will need to append this token to its requests, and once the server receives a request, it responds only if the token is valid. Please note that the authenticated clients are authorized to perform all possible weather queries, so we do not distinguish between *authentication* and *authorization* for the sake of simplicity.

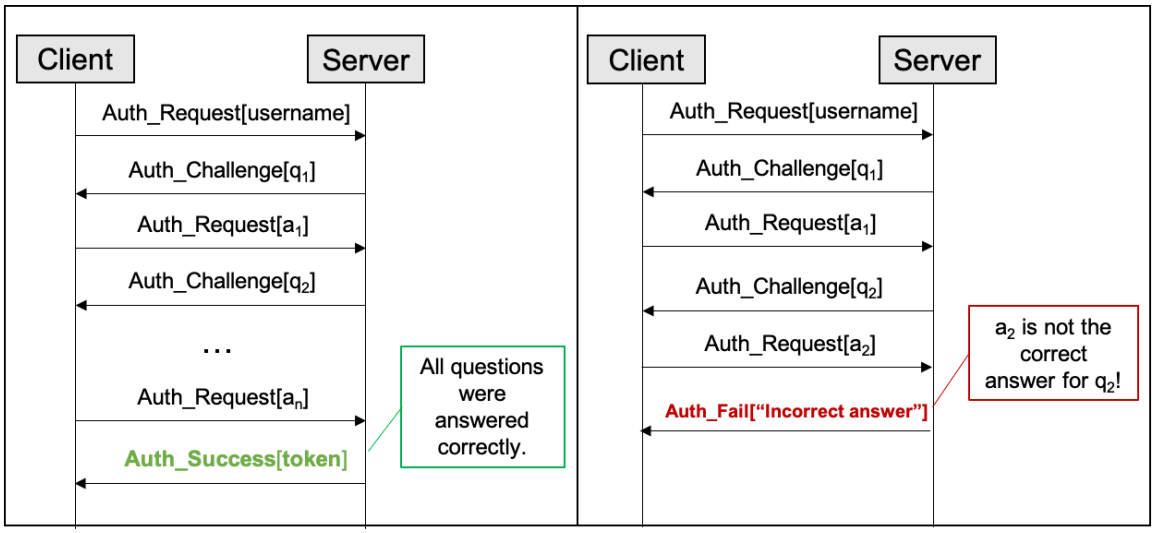

Fig. 2. The message flows for the authentication phase.

First, the client will send its username to request to be authenticated (i.e., acquire a token). Then, the server will authenticate the client by requesting the answers for a series of secret questions that are included in **Auth_Challenge** messages. After receiving a question, the client will prompt the user, the user will enter the answer through standard input, and the client will send the answer by including it into an **Auth_Request** message. Some example questions and answers:

- "What is your favorite color?" – "red"
- "What is the first name of your favorite author?" – "kadri"
- "In which city you were born?" – "istanbul"
- "What is the last name of your best friend?" – "zorlu"
- "What is your goal in this course?" – "to get an A"

The server decides how many questions the client should answer, and the questions will be chosen from a pool of possible questions. The correct answers to these questions for the particular user will be known by the server. If all the questions were answered correctly by the client, the server will send back an **Auth_Success**, including a unique token for the client. If the client answers a question incorrectly, the server will immediately respond with an **Auth_Fail** with the reason of failure message as "Incorrect answer". Please note that the server may also send back an **Auth_Fail** when the first **Auth_Request** includes a nonexistent username. In this case, the reason for failure should be "User does not exist".

Figure 1 illustrates the intended message flow, where q_i denotes the i^{th} question and a_i denotes the client's answer to q_i.

a. Message format

In protocol design, (1) the message types, and (2) the format & the meaning of the values in each type of message must be clearly specified. The client and the server will handle the received messages according to their types. Similarly, they will construct the messages adhering to the protocol.

Message types

During the authentication phase, the client is able to send only **Auth_Request** messages, and the server is able to send **Auth_Challenge**, **Auth_Fail** and **Auth_Success** messages.

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)

Deconstructing the TCP data

The TCP data received from the socket must be deconstructed correctly. The first six bytes are designated as the application header, where the first byte represents the "phase" (either the authentication (0) or querying (1) phase.), and the second byte represents the "type" of the message. If the "phase" byte is set to 0, then the messages will be handled by the authentication module of our implementation. Otherwise, your implementation should hand over the handling of the request to the weather querying module. The remaining four bytes of the header are designated as an integer (4 bytes) denoting the length of the payload in bytes. Your application should use this value to read the correct number of bytes from the TCP stream as the payload, since we do not know the length of the payload beforehand. Figure 2 shows the deconstruction of an authentication message.

Here are some useful links:

https://docs.oracle.com/javase/7/docs/api/java/io/DataInputStream.html
https://docs.oracle.com/javase/7/docs/api/java/io/DataOutputStream.html
https://docs.oracle.com/javase/7/docs/api/java/io/FilterOutputStream.html

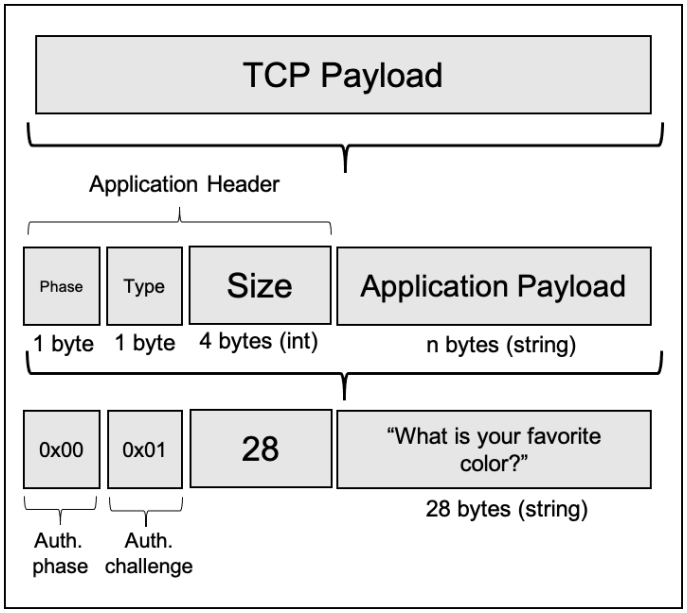

Fig. 3. Deconstructing the TCP data.

b. Timeout mechanism

You also need to handle the cases where the client is unresponsive to a question. After sending a challenge, the server should wait only for a predetermined amount of time (e.g. 10 seconds) before sending an **Auth_Fail** to the client with the appropriate reason message and closing the connection.

c. Implementation details

Storing users, questions, and answers
For simplicity, you may want to keep all the users, questions and answers in an easily parsable text file. Figure 3 illustrates an example of such a file, where we have two users – “ali” and “veli” with different questions and answers.

```
ali
What is your favorite color?
red
What is the first name of your favorite author?
kadri
In which city you were born?
istanbul

veli
What is your favorite color?
blue
What is your goal in this course?
to get an A
```

Fig. 4. An example of a text file storing users, questions and answers.

The token
The token should be unique for each session. Ideally, they should be constructed on-the-fly. You can construct a token by performing a hash on the concatenation of the username and a random number. Then, the token will be the first n (e.g., 6) characters of the output. After that, you should save the token along with the corresponding client IP + port and username, so that the server can verify the token in the future during the querying phase.

Architecture
While architecturing your application, it may be beneficial to separate it into modules/layers so that different group members can independently work on a single module. For example, the authentication module of the server and client would communicate with each other (through the command socket) to agree on a token in the authentication phase. In the querying phase, the only responsibility of the authentication module would be as following:

- **At the client:** Append the token to the message received from the weather module before sending the message through TCP to the server.
- **At the server:** Verify the received token appended to the received message from the TCP before supplying it to the weather module. Then, the weather module could simply focus on communicating with the OWM API.

Figure 4 shows an example of how the modules in your project may interact. The top section represents the authentication phase, while the bottom section represents an authenticated client sending a request to the server.

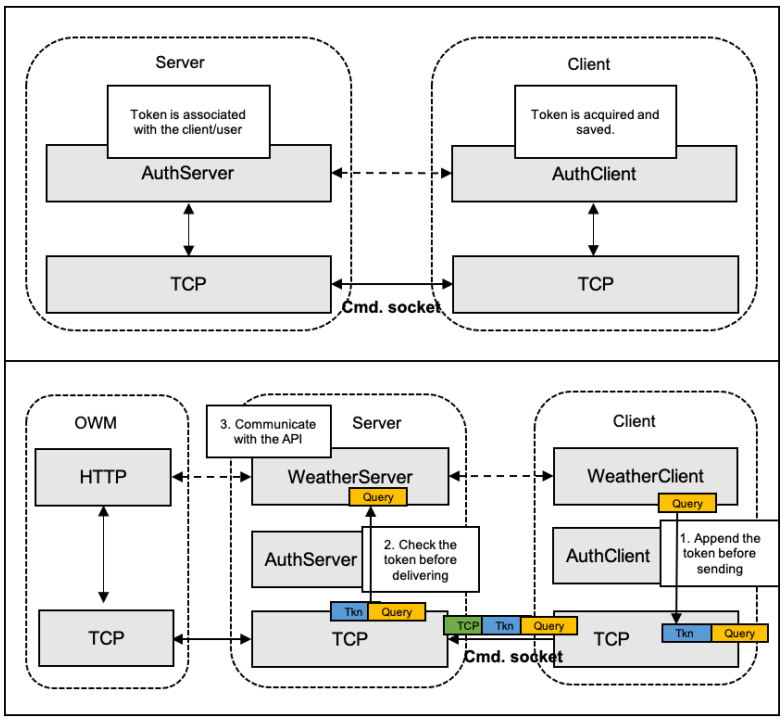

Fig. 5. The interaction of the different modules of WeatNet.

Execution Scenario

Due to the fact that the classes are being conducted online, the client and server are expected to reside on a single machine for simplicity of both execution and demonstration. However, any group preferring connection to a Remote device is free to do so albeit with necessary consideration given to the overall execution and demonstration scenarios.

The project envisions the groups to implement at least five unique clients and one server.

Project deliverables:

You should submit your source code and project report (in a single .rar or .zip file) via Blackboard.

Report

The report should start with a very brief description of the project. This should be followed by an explanation of the philosophies especially post-authentication server-client protocol. This should be followed by an overview of the programming of the client and the server side including the initial authentication, connection with the API, the file transfer mechanism and the file verification. Instead of attaching complete code, it is strongly recommended to attach code Snippets and images of the results of their

execution. **The report should explicitly describe the test routines developed to evaluate the full range of features required for the WeatNet.**

Source Code: A .zip or .rar file that contains your implementation in a single Eclipse or Intellij IDEA. If you aim to implement your project in any IDE rather than the mentioned one, you should first consult with TA and get confirmation.

- Source Code: A .zip or .rar file that contains your implementation in a single Eclipse or Intellij IDEA. If you aim to implement your project in any IDE rather than the mentioned one, you should first consult with TA and get confirmation.

- The **report** is an ***important part of your project*** presentation, which should be submitted as both a .pdf and Word file. Your report should show the step by step OWM configuration and connection with your code. As well as your server-client communication initiation. **Report acts as a proof of work for you to assert your contributions to this project.** Everyone who reads your report should be able to reproduce the parts we asked to document without hard effort and any other external resource. If you need to put the code in your report, segment it as small as possible (i.e. just the parts you need to explain) and clarify each segment before heading to the next segment. For codes, you should be taking ***screenshot*** instead of copy/pasting the direct code. Strictly avoid ***replicating the code*** as whole in the report, or leaving code ***unexplained***. You are expected to **provide detailed explanations** of the following clearly in your report:

Demonstration:

You are required to demonstrate the execution of WeatNet Application for the defined requirements. Your demo sessions will be announced by the TAs. Attending the demo session is required for your project to be graded. ***All group members*** are supposed to be available during the demo session. **The on time attendance of all group members to the demo session is considered as a grading criteria.**

During your demonstration of the authentication phase, you will be asked to demonstrate **two** clients being authenticated at the same time. You can assume that all the clients will present different usernames. You need to make sure that all the users have different correct answers for each of the possible questions. The clients and the servers will be running on the same machine, so different clients should be running at different ports. In this vein, the server will differentiate the clients not only by their IP address, but also with their port number. For the demonstration, the server should send **three** different questions before authenticating the user. First, you will need to show an unsuccessful authentication, then you will need to show a successful authentication.

During the demonstration, the group will be asked to display all the operations of the application starting from initiating client connections and authentication to passing requests for the listed metrics. It is strongly recommended that the appropriate test

routines may be developed to present an effective demonstration and display the full range of features as defined for WeatNet application.

The groups have the creative freedom to present any additional features they have built into their application in any way they deem feasible. However, please note that you will be given 10-15 minutes for your demonstration.

Following is a detailed but not exhaustive list of the test routines the groups should develop to aid them in testing and demonstration:

1. Client creation. The application should be designed with client scalability in mind even though the testing would be done with max 5 clients.
2. Single client connection with the server
3. Multiple clients simultaneously connecting with the server
4. Authentication procedure for a single randomly chosen client
5. The process from initiating a request by the client side to the final verification of the received file at the client side and file display.
6. Process in case of a mismatch between received file and received hash.

Suggested task distribution:

We recommend you work in a group of 3 students, and suggest the following task distribution accordingly:

- Student 1: Client-side programming and Authentication.
- Student 2: Server-side programming and multi-threading.
- Student 3: OpenWeatherMap API programming and file transfer mechanism.
- All members of the group to perform integration, tests and report.

The groups should be clear on the task distribution and the relevant questions will be directed towards the student responsible for the task.

Good Luck!