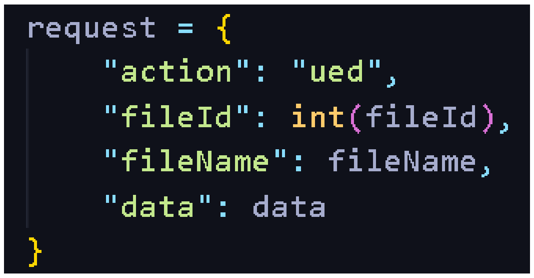
Language version: **Python 3.9.2**

Program Design

A TCP server will be started by server.py, from which it will wait for connections from clients. By using threads, the server has support for multiple concurrent client connections. The client will then communicate with the server via the TCP connection, the server will execute the request and return an appropriate message to indicate the outcome. Active clients can also communicate directly to each other via UDP.

Application layer message format

The client will send requests to the server through dictionaries, which will contain an action and possibly other necessary arguments. These dictionaries will then be converted to JSON objects before being sent to the TCP server. Example:



```
request = {
    "action": "ued",
    "fileId": int(fileId),
    "fileName": fileName,
    "data": data
}
```

Assumptions

- fileIds are unique across all client's files. The implications:
    - When data is generated, it is assumed that a file id will never be used twice (across all users) e.g. if supersmartwatch issues "EDG 1 10", uperwristband will not issue "EDG 1 _"
    - One device can upload a file with id 1 e.g. "supersmartwatch-1.txt" to the server and another device can use DTE 1 to delete it.
- UVF performs a .split('_'), so it is assumed that device names do not contain '_'

Brief Description of program

server.py

- Set max number_of_consecutive_failed_attempts as global variable
- Start TCP server. Wait for connections. Once it accepts a connection from a client, it will start a new thread:

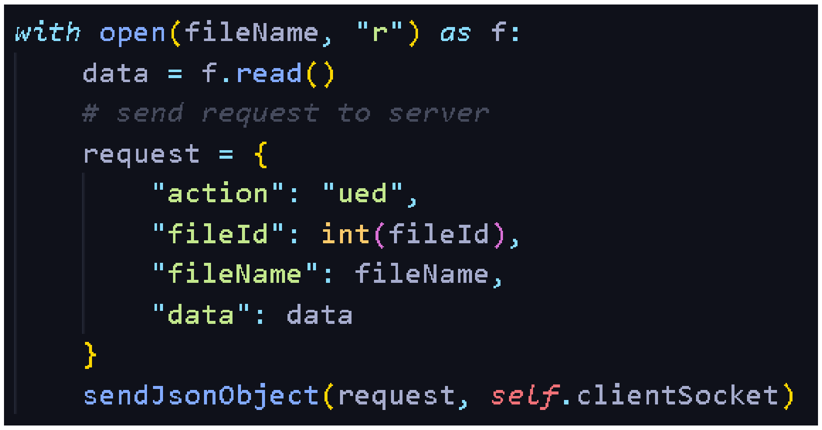
    - While the connection is still alive, the server will wait for requests from the client (see application layer message format)
    - The server will process the request and perform necessary actions before sending the client a message to indicate the outcome
    - The first requisite request from the client will always be login. The server will check if the credentials are correct.
        - If correct: continue
        - If not, the username will be added to a global variable, failedLogins, which is a dictionary containing usernames as its keys and the number of failed login attempts as its values. Once a username's number of failed login attempts == number_of_consecutive_failed_attempts, the server will add the username to another global dictionary, blockedUsers, for 10 seconds.

client.py

- Upon connecting to the server, the client will be prompted to login. If successful, the client will then have access to several commands (EDG, UED, SCS, DTE, AED, OUT, UVF).
    - If the user enters a valid command with valid arguments, the client will store the parameters in a dictionary then send it to the TCP server as a JSON object
- The client will also start a new thread for receiving UDP packets
    - For UVF, the client will first check if the recipient is active on the TCP server
    - If so, the specified file is read in chunks of 1024 bytes and sent in UDP packets to the recipient

Design Trade-offs

UED puts data samples into a string, then into a dictionary as pictured below:



```
with open(fileName, "r") as f:
    data = f.read()
    # send request to server
    request = {
        "action": "ued",
        "fileId": int(fileId),
        "fileName": fileName,
        "data": data
    }
    sendJsonObject(request, self.clientSocket)
```

In the case where large data samples are uploaded (where the JSON object exceeds 1024), the program will fail. To address this, the program should first check whether the data sample can be placed in a single 1024-byte packet. If not, the program should subsequently divide the sample into 1024-byte chunks.

Possible Improvements and Extensions

- While the client sends dictionary-formed responses, the TCP replies purely using strings. Ideally, servers should also be responding to the client with dictionaries, as we will be able to store more information about the outcome of the command.
- The implementation of AED currently uses a global dictionary to track active clients. Consequently, the program tracks active users twice: in the dictionary and in the log file. Thus, we can improve its time efficiency by acquiring the information directly from the active devices log without the need for a dictionary
- Our program could also allow for new users to join the network. To do so, client.py would need to have separate API for a new user to register, which would prompt for information such as username and password to be stored in credentials.txt
    - We could further provide support for emails to be used in place of usernames, by prompting users to enter their email upon registration
- To improve security for our users, their credentials could be stored as hashes. Specifically, we could use a “Bcrypt”-like function which can protect against rainbow table or brute force attacks.

References

- Logic of UDP server inspired by: ATOzTOA (2012) sending/receiving file UDP in python - Stack Overflow , accessed 6 November 2022
- The Python Cryptographic Authority developers (2022) bcrypt · PyPI, accessed 10 November 2022