

BLG 433E Computer Communications Project 1

Due: Tuesday, March 19, 2024

Lecturer: Prof. Abdul Halim Zaim CRN: 21358

Fatih Baskın

150210710

March 18, 2024

Contents

Basics of Socket Programming	3
Socket Connections	3
TCP Socket Connection	3
How Socket Applications Work	3
Comparison Between TCP Socket Communication and HTTP	3
Project Implementation Details	4
Server Runtime Environment	4
Example Folder Structure and Environment Variables	5
Communication Architecture, Request and Response Formats	5
Server Application File Storage, Updates, Event Loop	5
Client Application File Write, Read	6
Example Use Cases	6
Client Read, Write	6
Server Run and Close	8
Possible Improvements	9
Event Handler Thread and Event Queue	9
Client Timeout	9
User Authentication, Security, Validation, Encryption	9

Basics of Socket Programming

Socket Connections

A socket connection establishes a two-way communication channel between two applications on a network. A good analogy would be a phone line between two applications. Similar to phone lines, socket connection allows full-duplex communication, data can flow in both directions simultaneously, like a two-way conversation.

TCP Socket Connection

TCP ensures reliable data delivery by breaking down large messages into smaller packets, sequencing them, and checking for errors. Lost or corrupt packets are retransmitted. A TCP connection requires a handshake process to establish and terminate the connection. This guarantees both parties are ready to communicate.

The properties of TCP make communication both reliable and easy to manage since TCP ensures reliable data delivery by breaking down large messages into smaller packets, sequencing them, and checking for errors. Also, a TCP connection requires a handshake process to establish and terminate the connection. This guarantees both parties are ready to communicate. This additionally contributes to the reliability of the TCP connections.

With this reliability on hand, in my project, TCP socket communication is used. There is also another alternative, UDP socket communication but it does not offer the same reliability as the TCP socket does, therefore I decided to use TCP socket communication.

How Socket Applications Work

- The server application creates a socket and binds it to a specific port, essentially opening a listening port.
- The client application creates a socket and initiates a connection request to the server's IP address and port. The client socket connects to the server's socket, this way a communication channel is established.
- A three-way handshake establishes the connection, confirming both sides are ready.
- Once connected, applications can send and receive data streams reliably.
- When communication is finished, the socket connection is terminated.

Comparison Between TCP Socket Communication and HTTP

- TCP sockets provide a general-purpose communication channel, while HTTP is a specific application-layer protocol for web communication.
- TCP connections are persistent by default, meaning they stay open until closed. HTTP connections can be persistent or non-persistent (closed after each request-response cycle).
- TCP deals with raw byte streams, requiring applications to handle data structure and interpretation. HTTP uses a structured format with headers and a message body for requests and responses.
- TCP sockets offer a reliable, low-level communication channel. HTTP builds upon TCP, adding structure and message exchange specific to web interactions.

Project Implementation Details

Server Runtime Environment

I have implemented both client and server applications in a Linux environment. Applications are not tested on a Windows environment therefore they might not work on a Windows machine.

The server application is configured using a `.env` file to make my project flexible. This `.env` file must be in the same folder as the `server.py` file. In this file, there must be several fields:

- `SERVER_PORT` is an integer value, which is the listening port of the server application.
- `SERVER_HOST` is a string, indicating the address of the server. For running the application locally, you can put `localhost` as a value.
- `MAX_CONNECTION` is an integer value, which is the maximum number of socket connections that the server application can make.
- `UPDATE_TIMER` is an integer value, indicating how often the server application checks the files to update their timestamp periodically. The value indicates seconds, so if the value is `60` then the server application checks the files every 60 seconds.
- `UPDATE_TRESH` is an integer value, which is the update time threshold of the server, given in seconds. This means, that if a file is updated more than `UPDATE_TRESH` seconds ago, the file timestamps are updated. If we want file update periods to be 1 day, then the value should be `86400` since a day consists of 86400 seconds.
- `FILE_PATH` is a string, is the absolute path to the folder that stores client data, meaning the `.txt` files sent by the client.
- `RECORD_PATH` is a string, is the absolute path to the folder where the client file records are held. Since reading all the files takes quite a long time, timestamp checks are done in the record file first. The record file contains the timestamp of each client file. If a file's records are outdated, the file and its record will be updated.
- `RECORD_FILE` is a string, which is the name of the record file. I recommend the file name to be `records.json`. This file is a json file, and since json files are parsed into Python dictionaries, it is easy to keep track of clients.

Since the server's configuration is done using a `.env` file, the python runtime environment that runs the `server.py` must contain `python-dotenv` package, which is used for loading `.env` files.

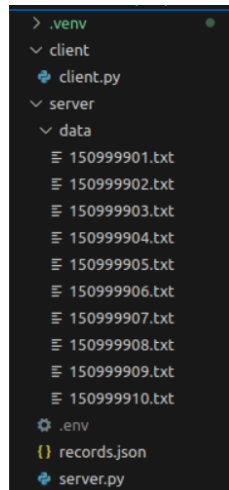
You can easily create a Python virtual environment in the Linux systems, by running `python3 -m venv .venv` which creates a virtual environment, which will be under the `.venv` folder, which is created in the folder this command is called from.

You can run this environment by running this command, `source .venv/bin/activate`, which activates the runtime environment which was created by the previous command.

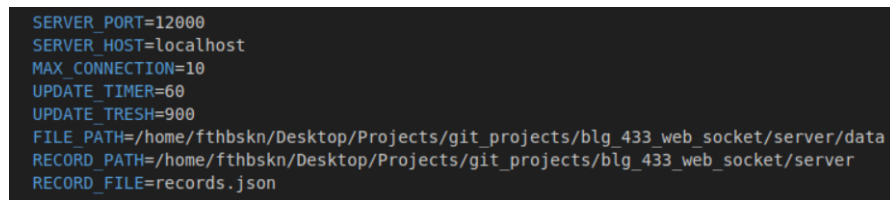
Then, to add packages to this environment, I have used `pip`, to add `python-dotenv` to the runtime environment, `pip install python-dotenv` command must be run when the runtime environment is active.

Example Folder Structure and Environment Variables

This example is from my computer. I have developed both server and client applications to run on Linux environment therefore, there might be some problems in Windows environments.



(a) Folder structure



(b) .env file structure

Figure 1: Overall structure

Communication Architecture, Request and Response Formats

Since both client and server can send messages to each other at any given time, to organize this complicated communication, each message should have a format. I have created a format based on JSON. The communication works in a client-server architecture where clients make requests and the server returns a response. Each message is in a JSON format.

Client messages contain three fields, `client_ID` corresponding to the student ID of the client-side user, `type` corresponding to the request type, and `data` containing the data sent in the request. In my architecture, there are two types of requests: read and write. Read is for downloading client information and write is for adding a new text file.

Server responses also contain three fields, `client_ID` corresponding to the client to which the server is responding, `type` is the response type, and the `data`. There are three types of responses: success, error, and info. Success and error are pretty straightforward, and the info responses are not implemented. They were for hypothetical cases where the client asks for how many connections are available etc.

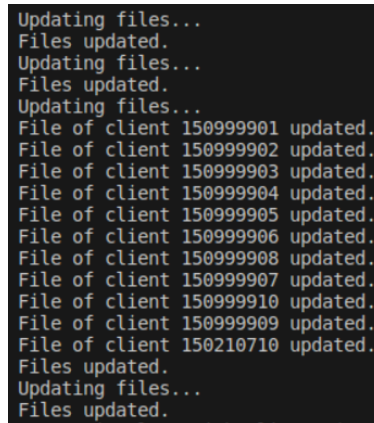
Server Application File Storage, Updates, Event Loop

The server application keeps a record of the files it stores. When a new file is created by a client or an existing file is updated by providing new data to the currently existing file by deleting the old information, the server will store that file and also update the record. File names are the student IDs.

The server also does a periodical file update, in which the update period is determined by the `.env` file. The server will check the record file only since file reading operation is quite expensive. If a file is older than the

specified threshold in the `.env` file, the file record and the file's first line which contains the timestamp are updated with the current date. When such an operation occurs, the server will print a message specifying which files are updated.

Finally, the sockets are being listened to in an event-driven fashion, meaning that the socket events (data received from the client) will raise an interrupt, and with this interrupt, the client request is handled. This interrupt-based event loop is achieved by using `select` library of the Python.



```
Updating files...
Files updated.
Updating files...
Files updated.
Updating files...
File of client 150999901 updated.
File of client 150999902 updated.
File of client 150999903 updated.
File of client 150999904 updated.
File of client 150999905 updated.
File of client 150999906 updated.
File of client 150999908 updated.
File of client 150999907 updated.
File of client 150999910 updated.
File of client 150999909 updated.
File of client 150210710 updated.
Files updated.
Updating files...
Files updated.
```

Figure 2: Periodical file update messages.

Client Application File Write, Read

When the user wants to write a file to the server, the client application will gather the relevant data, which are: Student ID, Name, Date of Birth, City, and Reason which stands for reason why that student has chosen to take BLG 433E Computer Communications. The user will enter this information from the command line and enter the relevant data. Then the client application will add a timestamp on top of this information and will send this information in the client request data format, labeled as write request.

When the user wants to read the information from the server, the client program will ask for the user's student ID, and then the client will send a request with the user (student) ID labeled as read. Then the server will send the txt file in the server response format (in the data field) and the client application will save this `txt` file in the same folder as the `client.py`.

Example Use Cases

Client Read, Write

When the client application is launched, it asks for the server host address and server port, after writing them down, the client establishes a connection with the server.

After the connection is established, the user chooses to write a file into the system. The user selects option 1 (Write to the server), then the client application asks for some information and the client enters them one by one. Then finally, the client's application will add a timestamp to this data and send the data in a request structure. Then client will receive an appropriate response from the server.

```
(.venv) fthbskn@fatihPC:~/Desktop/Projects/git_projects/blg_433_web_socket$ python3 ./client/client.py
Enter the server address: localhost
Enter the server port: 12000
Connected to localhost:12000 address.
Server address set successfully.
1. Write to the server
2. Read from the server
3. Exit
Enter your choice: █
```

(a) Client server connection process from the client side

```
Server shut down.
(.venv) fthbskn@fatihPC:~/Desktop/Projects/git_projects/blg_433_web_socket$ python3 ./server/server.py
The server is ready to receive on port 12000.
Server host: localhost
Server max connections: 10
Press Ctrl+C to quit.
Connection established with ('127.0.0.1', 36330)
Updating files...
█
```

(b) Server accepts the client connection

Figure 3: Client-Server connection establishment from client and server side

```
Enter your choice: 1
Enter your student ID: 150210710
Enter your name: Fatih Baskin
Enter your date of birth: 23/11/2000
Enter your city: Tekirdag
Enter your reason to chose this class: I want to learn about how Internet and WEB works in general and its details.
Server response: File updated successfully.
1. Write to the server
2. Read from the server
3. Exit
Enter your choice: █
```

(a) Client takes the necessary data from the user then adds datestamp to the data and finally sends a write request to the server.

```
Handling write request of client: 150210710
```

(b) Server processes this request. In the console, it writes down which client had made which request.

```
server > data > 150210710.txt
1 Last updated: 2024-03-18 13:14:33
2 Student ID: 150210710
3 Name: Fatih Baskin
4 Date of Birth: 23/11/2000
5 City: Tekirdag
6 Reason: I want to learn about how Internet and WEB works in general and its details.
```

(c) Server writes the incoming file to the data folder

```
150999901": "2024-03-18 13:14:33",
150999902": "2024-03-18 13:14:33",
150999903": "2024-03-18 13:14:33",
150999904": "2024-03-18 13:14:33",
150999905": "2024-03-18 13:14:33",
150999906": "2024-03-18 13:14:33",
150999908": "2024-03-18 13:14:33",
150999907": "2024-03-18 13:14:33",
150999910": "2024-03-18 13:14:33",
150999909": "2024-03-18 13:14:33",
150210710": "2024-03-18 13:14:33"
```

(d) Server logs the written files in the record file.

Figure 4: Sequence of client's write request

After a while, the client wants to read what was written about him on the server, so he decides to download the file. He selects the read from the server (second) option to get his file. He enters his student number and the file is downloaded to the same folder where the `client.py` is located.

```

1. Write to the server
2. Read from the server
3. Exit
Enter your choice: 2
Enter your student ID: 150210710
Written requested file to /home/fthbskn/Desktop/Projects/git_projects/blg_433_web_socket/client/150210710.txt successfully.
1. Write to the server
2. Read from the server
3. Exit
Enter your choice: █

```

(a) Client application makes a read request, first takes the user student number then makes a request to the server. Then the server sends the requested file and the client saves it on the disk.

```

Files updated.
Handling read request of client: 150210710
Updating files...

```

(b) Server takes the read request and handles it.

```

client > E 150210710.txt
1 Last updated: 2024-03-18 13:14:33
2 Student ID: 150210710
3 Name: Fatih Baskin
4 Date of Birth: 23/11/2000
5 City: Tekirdag
6 Reason: I want to learn about how Internet and WEB works in general and its details.

```

(c) The downloaded file from the server

Figure 5: Client file read (download) sequence

The client is done with his actions, so he wants to quit the application. When the client application is closed, It closes the connection from the server. The server's socket will receive a TCP packet with empty data, the server recognizes this as closure of the client's socket so the server will also close its socket from the server side.

```

Written requested file to /home/
1. Write to the server
2. Read from the server
3. Exit
Enter your choice: 3
Connection closed successfully.
Goodbye!
(fatih@fatihPC: ~/Desktop)

Files updated.
Connection lost with client: ('127.0.0.1', 36330)
Updating files...

```

(a) User chooses to close the application, (b) Server recognizes the client closing its socket connection therefore server also closes its side of the connection and close.

Figure 6: Closure of connection between client and server.

Server Run and Close

Server application launches, reads environment variables establishes listening socket, and waits for client socket connections.

```

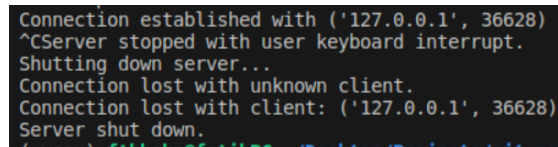
(.venv) fthbskn@fatihPC:~/Desktop/Projects/git_projects/blg_433_web_socket$ python3 ./server/server.py
The server is ready to receive on port 12000.
Server host: localhost
Server max connections: 10
Press Ctrl+C to quit.

```

Figure 7: Message after server application is run

When the server application is going to be closed, the server administrator closes the application using a

keyboard interrupt (**Ctrl+C** in Linux environment), and the server application closes connections with clients running on sockets, one by one, and finally closes the application.

A terminal window with a black background and green text. The text shows the server's state during a shutdown: 'Connection established with ('127.0.0.1', 36628)', '^CServer stopped with user keyboard interrupt.', 'Shutting down server...', 'Connection lost with unknown client.', 'Connection lost with client: ('127.0.0.1', 36628)', and 'Server shut down.'.

```
Connection established with ('127.0.0.1', 36628)
^CServer stopped with user keyboard interrupt.
Shutting down server...
Connection lost with unknown client.
Connection lost with client: ('127.0.0.1', 36628)
Server shut down.
```

Figure 8: Server closes the connection with clients one by one. The unknown connection is the server's listening socket.

Possible Improvements

Event Handler Thread and Event Queue

I have tried to make my application as event-driven as possible, but the problem with my code is, that the application handles client events (socket connected to client receiving data) sequentially. Event handler(s) can be run as a thread when a client event occurs, and to distribute those events, there must be an event queue to distribute those events to threads.

The downside of this multithreaded approach is the possible problems with concurrency, if two client events require the same resource, in my project this would be the `RECORD_FILE` or any client data file, and acquiring-releasing locks would run those threads sequentially if there is such a case. It might be sufficient to run only one event handler thread but these are design choices to be made if this application is made to run with event threads.

Nevertheless, adding parallelism would prevent the main thread from being blocked, this is especially important if a client event is taking a long time, possibly blocking other socket connections.

Client Timeout

Currently, my server application does not have a timeout option. This might be important if this was a big project since a client application might stop running abruptly, or might be idle, therefore to make room for other clients to connect to the limited amount of sockets, a timeout mechanism should be added. Since this project is very rudimentary, I have not implemented client timeout but for a big project, it should be added.

User Authentication, Security, Validation, Encryption

Currently, the server application is running with the assumption that the client application is sending valid data. Since this is a simple project, this is acceptable. But for complicated projects, there should be a validation step to check data sent by the client is valid, and then there should be security checks to ensure that data received is safe to handle. Also, there can be an authentication step, right after the client application establishes a connection. As a final note, these messages are sent in a raw, string form, but for secure communication, the data must be encrypted. These are out of the scope of this project but still, these are important design criteria when designing a big project using socket programming.