Private On-Premise TCP Chat Server

"Archimede" High School - Treviglio

Author:

Zanotti Giovanni

Abstract

Chatting never feels secure whenever we open our messaging tools and send data over the Internet. To overcome this problem, we decided to develop an on-premise solution, which could make it possible to chat between users in the same network without the need of accessing the Internet. To do this Python has been used for its enormous variety of available libraries and ease of use for a fast deploy.

Summary

1.0 Objective

Whenever we use a messaging tool, our information passes through means unknown to us, and therefore without guarantee that the data is truly protected. To escape this danger, we can use a system that operates on our network, without traffic passing through the router to the Internet. Internal communications, in this way, become secure and it is more difficult for someone to intercept and steal valuable data. To do this, we used Python and its libraries, which allow for rapid development for rapid deployment within a business environment.

2.0 Tools and Technologies

To develop this project, different technologies have been used.

2.1 Visual Studio Code

To develop this project, it has been decided to use the Visual Studio Code text editor, which excels for portability and ease of use, including the possibility to install extensions which increase its versatility in multi-language development.

2.2 Python

Python is a programming language, which excels for its syntax simplicity and available libraries, making it the perfect choice for data analysis, machine learning or to apply algorithms without having to focus on the language complexity itself.

2.2.1 Socket

Socket is a Python library which makes it possible for processes to communicate through the operating system and/or through the network. To actually use it, we create a socket object inside the server and give it the ip and source information, and then we make it listen for incoming requests. A client just has to create the socket and connect to the server. When the connection is established, the two machines can start communicating and exchanging messages; in our case, the client would send the string to encode, and then the server would reply with the transformed.

2.2.2 Threading

Threading is a Python library which allows you to make multithreading programs, which means having more than one independent flow of instructions.
The socket library makes it possible to realize communications, but the problem is that it would create a *sequential server*, which means that only one instruction at a

time can be executed, which also translates to being able to only manage one client at a time. To solve this problem we decided to use the threading library. The solution is simple: instead of replying to only one client at a time, we create a thread for every client, which is going to independently handle the client request and reply.

2.2.3 JSON

JSON is a Python library that allows you to manage and convert json files as if they were Python dictionaries, through functions dedicated to transforming from JSON to dictionary and vice versa. This format, which is considered the de facto standard in our project and used to communicate between the different components of the architecture, becomes simpler for developers to use.

2.2.4 SYS and OS

Sys and os are Python libraries that allow you to access files and directories outside of the one in which the current file is in. This means that we can access data, such as libraries developed by ourselves, such as the common "utils" library in the case of our project, from any project directory. How it works is simple: os allows you to navigate (for example by running simple '..' to exit the folder), while sys adds the directory you are in after the os commands, making the files available within it or by specifying the paths starting from it.

2.2.5 Time and Datetime

Time and datetime are Python libraries which make it possible to manage time based on the machine on which the code is executed. In this case they are firstly used for the datetime endpoint, which permits the end-user to know the time configuration of the central server and check if it's different from the one of his own machine. And secondly, they are also used to avoid the "TCP Coalescing" problem, which would merge the "/help" endpoint request with the name sent on registration.

2.2.6 Random

Random is a Python library which permits generating random values. The values are pseudo-random, but it is still a very useful library as it makes it possible to create programs which look non-deterministic, and in our case, to send random requests to the chat server from our "Stressing Bots".

2.3 Docker

Docker is an open source software which makes it possible to realize *containers*, for both Linux and Windows. Containers are “virtual boxes” where we can save the source code and all of its dependencies, making it possible to execute the program on a second machine without having to install all the necessary dependencies.
In this project, Docker has been used with the chat server to make the server easily runnable everywhere at any time.

2.4 Networking and Protocols

Talking about Networking and connection between the different machines, the TCP protocol has been used, part of the TCP/IP stack. This protocol makes it possible to send "secure messages", by sending an "ACK" (which stands for "acknowledgment") every few packets of data, and sending again lost data. This ensures that all messages that get sent are received by the server and by the other users of the platform.

2.5 CI/CD and Version Control

CI/CD and Version Control permit a clean flow for multi-dev projects. In this case, five groups of people have worked on the same project, making it a must to have a way of checking who adds what and if everything can work together. Talking about CI/CD, GitHub actions have been implemented (with an AI generated code) by implementing an automatic script that generates a package (a Docker image) that gets published on the GHCR (GitHub Container Registry), making the Docker image available and ready to be executed on the actual enterprise server.

3.0 Client - Server Architecture

In this solution, we find a complete implementation of the Client - Server architecture, also seen as Star Topology.

3.1 Server

The server script creates a socket and listens to incoming requests. When a request is made, the client handling is given to an independent thread, making it possible to manage multiple clients at once.
Here we can find the client_handler function script:
We can find here the main steps of the client handling algorithm:

1. The server receives the name of the new user.
2. It validates the username uniqueness:

   a. If not it refuses the connection and disconnects the new user
   b. If yes, it is added to the registry.
3. The server generates the Thread to handle the client (user) requests.
4. It receives JSON messages with data about the sender, the expected receiver and the message:
   a. If the receiver name is one of the commands, the server responds with the endpoint function output (discussed later).
   b. If the receiver name is inside the registry, the server sends the message to the end-user..
   c. If the receiver name isn't anywhere, the server replies that the user isn't available.
5. In case of disconnection errors, the server cleans up the local data(the registry).

3.2 Client

The client script generates two threads:

- One is responsible for taking the expected receiver and the message to send to the chat server.
- The other is responsible for catching all incoming messages and displaying them on the terminal.

4.0 Discussion

4.1 JSON Strict Schema

As we said, different groups worked on the same project, with the idea of developing different clients which would connect to the same server. To be able to do this, a standard has been developed by the team leaders. The standard involves the type of messages that would be sent on the network, and the json format has been selected for its simplicity and ease of use.
We can see here the standard JSON message schema:

As we can see in the image, we find a second dictionary inside our schema, which contains the data about the sender (name and ip), and after that the data about the expected receiver and the message sent.

This made it possible for the groups to develop different solutions which could communicate over the same media and to the same server.

```
{
  "from": { "name": "...", "ip": "..." },
  "to": "...",
  "msg": "..."
}
```

4.2 Endpoints

As we have a central server, we may need some data while chatting with our friends, like "how do I send a message to all the active users on the platform?". To do this, we implemented "hand-crafted" endpoints. There are different currently reachable endpoints in the server:

- /help → show the available commands
- /time → show the current server time
- /list → list all active users
- /shout → send a message to everyone (broadcast)

To reach them, the user just has to send a message to the endpoint, just like if it was an active user, for instance: if I type /help when the client asks *To:*, I would receive the list of all available commands.

We can analyze the script of one of the endpoints:

```python
def endpoint_list(conn, requester_name, text):

    with lock:
        users = [name for name in clients if name != requester_name] # everyone but the requester
    if users:
        msg = f"Online users ({len(users)}): " + ", ".join(users)
    else:
        msg = "Your are the only active user!"

    send_server_response(conn, requester_name, msg)
```

In this code, we can find the '/list' endpoints, which replies with the list of all current users. As we can see, the code uses the lock to be sure the list of active clients isn't

accessed by other server threads (as we can find the race condition for example whenever multiple clients register at the same time). After that, the list is created by getting all names inside the *clients* list, except for the one sending the request to the endpoint.

After that:

- If there isn't anyone: the server tells the requester that he is the only active user.
- Else, the server sends the generated list as a formatted string.

And then, the message is sent to the user by using the "send_server_response", which simply formats a json with the server signature and the message passed as parameter, and then sends it to the endpoint requester.

4.3 Stress Tests

As we developed a chat server, we have to know if it is able to manage a large amount of active users at the same time, and this is why we decided to generate testing scripts (*made by AI*). We generated two scripts:

- stress-test: to decide how many bots would connect to the server at the same time
- breakpoint-test: to find if the server would collapse with too many connections.

After our tests, we found the server could manage over 6000 active users at the same time, without losing performance. Testing capped at 6000 active users, primarily due to hardware constraints of the host machine (CPU and RAM usage), rather than software limitations.

5.0 Conclusions

In the end we realized a working solution. This report talks about the work of the Group 3 of the project, which includes the central server and the main repository documentation.
All the endpoints work and the stress-tests have been passed successfully. The project can be improved in the future by adding a GUI, making it easier to use for the actual end-user, as in the current state it can be seen as a *cool way of chatting for developers*.

6.0 Project Links

- GitHub Repository:
  https://github.com/GiZano/multithreaded-python-chat
- Web Presentation:
  https://giovanni-zanotti.is-a.dev/Pages/Works/private-chat.html

7.0 Bibliography

- Visual Studio Code installation: https://code.visualstudio.com/
- Docker documentation: https://docs.docker.com/
- Python documentation: https://docs.python.org/3/