

1. Introduction

The project involves building a chat server in Python that supports multiple client connections, allowing for real-time messaging and a word-jumbling game. The server and clients communicate over TCP/IP sockets, with a focus on multi-threading to handle concurrent connections. This system aims to facilitate an interactive chat environment while adding fun elements such as trivia facts and a word game.

2. Objectives

- **Establish a Multi-Client Chat System:** Develop a chat server that allows multiple clients to connect, send and receive messages, and manage user interactions in a networked environment.
 - **Integrate a Word Game:** Add a simple word game where the server sends a jumbled word to the client, and the client must guess the original word.
 - **Provide Fun Facts:** Implement a /fact command that allows clients to request and receive random fun facts.
 - **Utilize Multi-Threading:** Use Python's threading module to handle concurrent client connections without blocking any client's communication.
-

3. Technologies Used

- **Programming Language:** Python (version 3.x)
 - **Libraries:**
 - **socket:** For network communication (TCP/IP).
 - **threading:** For handling multiple client connections concurrently.
 - **random:** For selecting random words, facts, and shuffling words for the game.
 - **Network Protocol:** TCP/IP, providing reliable, ordered, and error-checked delivery of messages.
-

4. System Architecture

The system is divided into two major components:

1. Server Side:

- The server listens for incoming client connections on a specific IP address and port.
- It spawns a new thread for each client to handle communication independently and concurrently.
- The server can send a random fact or initiate the word jumbling game.
- It also handles broadcasting messages from one client to all other connected clients.

2. Client Side:

- The client connects to the server by specifying the server IP and port.
 - It sends messages, listens for incoming messages, and can request facts or initiate a word jumble game.
 - The client sends guesses for the word jumble game and receives feedback from the server.
-

5. Features and Functionalities

1. Multi-Client Chat:

- Multiple clients can connect to the server and send messages to each other.
- The server broadcasts incoming messages to all other clients.
- Each client is identified by a unique nickname.

2. Word Game:

- Clients can start a word jumble game by sending the /game command.
- The server responds with a jumbled word, and the client attempts to guess the correct word.
- The server provides feedback on whether the guess was correct or incorrect.

3. Fun Facts:

- Clients can request random fun facts by typing the /fact command.

- The server replies with one of the pre-stored facts.

4. Real-Time Communication:

- Clients can send and receive messages in real time.
 - The server ensures that messages are distributed to all clients except the sender.
-

6. Code Explanation

The project is implemented using Python's built-in libraries: socket, threading, and random.

Server Code Overview:

1. **Socket Initialization:** The server creates a TCP socket and binds it to a specific host and port to listen for incoming client connections.
2. **Client Connection Handling:** For each incoming connection, the server creates a new thread using the threading module. This allows the server to manage multiple clients at once.
3. **Jumbled Word Game:**

```
def jumble():
    original = random.choice(words)
    lst = list(original)
    random.shuffle(lst)
    jumbled = ''.join(lst)
    return original, jumbled
```

- A random word is selected from a list, shuffled, and sent to the client. The client then tries to guess the original word.

4. Broadcasting Messages:

```
def sendmessage(message, sender):
    for client in clients:
        if client != sender:
            try:
                client.send(message.encode())
```

```
except:  
  
    client.send("Client Disconnected".encode())  
  
    clients.remove(client)
```

- The server broadcasts messages from one client to all others, except the sender.

Client Code Overview:

1. Connecting to Server:

- The client connects to the server using the specified IP and port.

2. Sending Messages:

- The client can send messages to the server and receive messages from other clients.

3. Game Interaction:

- When the client sends the /game command, the server sends a jumbled word, and the client sends back a guess.

4. Fun Fact Request:

- The client can request a fun fact by typing the /fact command.
-

7. Workflow

1. Client Connection:

- The client connects to the server, which accepts the connection and assigns a new thread for communication.

2. Messaging:

- Clients send messages to the server, and the server broadcasts these messages to all other connected clients.

3. Game:

- The client sends the /game command.
- The server responds with a jumbled word, and the client guesses it.
- The server checks the guess and sends feedback.

4. Fun Facts:

- The client can type /fact to receive a random fun fact from the server.
-

8. Multi-threading Implementation

The server uses threading to handle multiple clients at once:

```
def serverside(conn, add):  
    while True:  
        # Handle client interactions  
        pass  
  
def threadings():  
    while True:  
        conn, add = server.accept()  
        thread = threading.Thread(target=serverside, args=(conn, add)).start()
```

Each client has its own thread, allowing for real-time communication and game play without blocking other clients.

9. Testing and Results

1. Testing Client Connections:

- Multiple clients can connect to the server simultaneously and exchange messages.
- Each client can independently interact with the game and request fun facts.

2. Game Functionality Test:

- When the /game command is sent, the server successfully jumbles a word and checks the client's guess.
- The server responds with correct or incorrect feedback.

3. Fun Facts Test:

- The /fact command returns a random fun fact each time it is invoked.
-

10. Challenges and Solutions

- **Challenge:** Managing multiple client connections.
 - **Solution:** Multi-threading was implemented to allow each client to have their own connection and communication thread.
 - **Challenge:** Synchronizing client data and handling client disconnections gracefully.
 - **Solution:** Used proper exception handling to remove disconnected clients from the list.
 - **Challenge:** Ensuring game interaction is smooth for multiple clients.
 - **Solution:** The game logic was kept simple and clear, ensuring no blocking between clients during gameplay.
-

11. Future Enhancements

- **Graphical User Interface (GUI):**
 - A GUI could be added to the client-side application for a more user-friendly experience.
 - **Persistent Storage:**
 - Store user data and messages in a database to allow for user profiles and message history.
 - **Extended Game Features:**
 - Add more games, such as trivia quizzes or multiplayer games, to increase engagement.
-

12. Conclusion

This project demonstrates how Python's socket, threading, and random libraries can be used to create a robust and interactive multi-client chat server. The system supports real-time messaging, word jumbles, and fun facts, offering both communication and entertainment. It is a solid foundation for further development into more complex networked applications.

13. Appendix

- Client Code:

```
● ○ ●

import socket
import threading

port = 12345
server_ip='192.168.106.138'
client = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
#server_ip=input("Enter the server IP: ")

client.connect((server_ip,port))
def send():
    while 1:
        try:
            message=input()
            client.send(message.encode())
        except:
            print("Failed")
            client.close()
            break
def receive():
    while True:
        try:
            msg = client.recv(1024).decode()
            if msg:
                print(msg)
            else:
                break
        except:
            print("Not connected")
            break

def sendthread():
    threads = threading.Thread(target = send).start()
def recvthread():
    threads=threading.Thread(target=receive).start()

sendthread()
recvthread()
```

- **Server Code:**

```

import socket
import threading
import random
facts = [
    "Honey never spoils. Archaeologists have found pots of honey in ancient Egyptian tombs that are over 3,000 years old and still edible.",
    "Octopuses have three hearts.",
    "Bananas are berries, but strawberries are not.",
    "A day on Venus is longer than a year on Venus.",
    "Sharks existed before trees.",
]
words = ['sad', 'angry', 'excited', 'calm', 'tired', 'hungry', 'thirsty', 'hot', 'cold', 'sun', 'moon', 'stars', 'sky', 'cloud', 'rain', 'snow', 'wind', 'tree', 'flower', 'dog', 'cat', 'bird', 'fish', 'horse', 'cow', 'pig', 'sheep', 'lion', 'tiger', 'red', 'green', 'blue', 'yellow', 'orange', 'purple', 'pink', 'brown', 'black', 'white', 'apple', 'banana', 'orange', 'grape', 'pear', 'strawberry', 'blueberry', 'raspberry', 'cherry', 'peach']

port = 12345
host = socket.gethostname()
host_ip = socket.gethostbyname(host)
print(host_ip)

clients=[]
nicknames=[]

server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
server.bind((host_ip,port)) #binding socket to host and port
server.listen(20) #listening for 20 connection
print("Server is active")
def jumble():
    original = random.choice(words)
    lst=list(original)
    random.shuffle(lst)
    jumbled = ''.join(lst)
    return original,jumbled
#send message
def sendmessage(message, sender):
    for client in clients:
        if client != sender:
            try:
                client.send(message.encode())
            except:
                client.send("Client Disconnected".encode())
                clients.remove(client)
#storing client info
def serverside(conn,add):
    while True:
        try:
            conn.send("Welcome,Please enter your name: ".encode())
            nick = conn.recv(1024).decode()

```



```
if not nick:
    conn.send("Name cannot be empty, bye")
    conn.close()
    continue
nicknames.append(nick)
clients.append(conn)
print(f"Nickname of {add[0]} is {nick}")
for client in clients:
    client.send(f"{nick} has joined the chat".encode())

#receiving messages
while True:

    message = conn.recv(1024).decode()
    if message:
        if message.startswith("/fact"):
            fact= random.choice(facts)
            conn.send(fact.encode())
        elif message.startswith("/game"):
            original,jump = jumble()
            conn.send(f"Server: Guess this word: {jump}".encode())
            userrecv = conn.recv(1024).decode()
            if userrecv == original:
                conn.send("Server: Correct".encode())
            else:
                wrong = "Server: Wrong! Right answer is: "+original
                conn.send(wrong.encode())
        else:
            sendmessage(f"{nick}: {message}", conn)
    else:
        break
except:
    if conn in clients:
        clients.remove(conn)
    if nick in nicknames:
        nicknames.remove(nick)
    print(f"{nick} has disconnected.")
    sendmessage(f"{nick} has left the chat.", conn)
    conn.close()
    break

def threadings():
    while 1:
        conn,add = server.accept()
        print(add[0]," has connected ")
        thread = threading.Thread(target=serverside,args=(conn,add)).start()
threadings()
```

- **References:**

<https://docs.python.org/3/howto/sockets.html>

<https://www.askpython.com/python/examples/create-chatroom-in-python>

<https://pandeyshikha075.medium.com/building-a-chat-server-and-client-in-python-with-socket-programming-c76de52cc1d5>

https://www.tutorialspoint.com/python/python_socket_programming.htm

<https://github.com/IamLucif3r/Chat-On>

<https://www.javatpoint.com/how-to-create-a-simple-chatroom-in-python>