Lecture 5: Create basic chat

Building a simple chat application in Python using TCP sockets. We'll break it down into several steps.

Step 1: Create a Basic Server

In this step, we'll create a basic server that listens for incoming connections from clients and echoes back any message it receives. Save this code in a file named `server.py`.

```python
import socket

# Server configuration
HOST = '127.0.0.1'  # Loopback address for localhost
PORT = 12345       # Port to listen on

# Create a socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to the specified address and port
server_socket.bind((HOST, PORT))

# Listen for incoming connections
server_socket.listen()

print(f"Server is listening on {HOST}:{PORT}")

while True:
    # Accept a connection from a client
    client_socket, client_address = server_socket.accept()
    print(f"Accepted connection from {client_address}")

    # Receive and echo back messages
    while True:
        message = client_socket.recv(1024).decode('utf-8')
        if not message:
            break  # Exit the loop when the client disconnects
        print(f"Received: {message}")
        client_socket.send(message.encode('utf-8'))

    # Close the client socket
    client_socket.close()
```

Step 2: Create a Basic Client

In this step, we'll create a basic client that connects to the server and allows the user to send messages. Save this code in a file named `client.py`.

```python
import socket

# Server configuration
HOST = '127.0.0.1'  # Server's IP address
PORT = 12345       # Server's port

# Create a socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

```python


# Connect to the server
client_socket.connect((HOST, PORT))
print(f"Connected to {HOST}:{PORT}")

while True:
    message = input("Enter a message (or 'exit' to quit): ")

    if message.lower() == 'exit':
        break

    # Send the message to the server
    client_socket.send(message.encode('utf-8'))

    # Receive and display the server's response
    response = client_socket.recv(1024).decode('utf-8')
    print(f"Received: {response}")

# Close the client socket
client_socket.close()
```

Step 3: Create a Chat Room on the Server

To create a chat room on the server, you can extend the server code from Step 1 to handle multiple clients simultaneously. You'll need to use threads or asynchronous programming to achieve this. Here's an example using threads:

```python
import socket
import threading

# ... (Code from Step 1)

# Function to handle a client's messages
def handle_client(client_socket, client_address):
    print(f"Accepted connection from {client_address}")

    while True:
        message = client_socket.recv(1024).decode('utf-8')
        if not message:
            break  # Exit the loop when the client disconnects
        print(f"Received from {client_address}: {message}")

        # Broadcast the message to all clients
        for client in clients:
            # if client != client_socket: # find alternative to this IF
            client.send(message.encode('utf-8'))

    # Remove the client from the list
    clients.remove(client_socket)
    client_socket.close()

clients = []

while True:
    client_socket, client_address = server_socket.accept()
    clients.append(client_socket)

    # Start a thread to handle the client
    client_thread = threading.Thread(target=handle_client, args=(client_socket, client_address))
    client_thread.start()
```

Step 4: Connect Client to the Room

In Step 4, you can use the client code from Step 2 to connect to the chat room. Multiple clients can connect to the server, and messages sent by one client will be broadcasted to all connected clients.

To run the server, open a terminal and execute `server.py`. To run clients, open multiple terminals and execute `client.py` in each of them. Clients can then send and receive messages within the chat room.

Please note that this is a basic example and may not be suitable for production use. In a production scenario, you may want to consider using a more robust framework or library for building chat applications and handling concurrency. Additionally, you should handle errors and edge cases more gracefully for a production-ready application.

Step 5: client receive messages independently of whether the user is typing or not

To make the client receive messages independently of whether the user is typing or not, you can implement a separate thread to continuously receive messages in the background while the main thread handles user input. We can use the `threading` module for this purpose. Here's an updated version of the client code from Step 2 with message reception handled in a separate thread:

```python
import socket
import threading

# Server configuration
HOST = '127.0.0.1'  # Server's IP address
PORT = 12345       # Server's port

# Create a socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)


# Connect to the server
client_socket.connect((HOST, PORT))
print(f"Connected to {HOST}:{PORT}")

# Function to receive and display messages
def receive_messages():
    while True:
        message = client_socket.recv(1024).decode('utf-8')
        if not message:
            break  # Exit the loop when the server disconnects
        print(f"Received: {message}")

# Start the message reception thread
receive_thread = threading.Thread(target=receive_messages)
receive_thread.daemon = True  # Thread will exit when the main program exits
receive_thread.start()

while True:
    message = input("Enter a message (or 'exit' to quit): ")

    if message.lower() == 'exit':
        break

    # Send the message to the server
    client_socket.send(message.encode('utf-8'))

```

```
# Close the client socket when done
client_socket.close()
```

In this updated code, we create a separate thread `receive_thread` that continuously listens for incoming messages from the server. This allows the client to receive messages independently of whether the user is typing or not. The `receive_thread` runs in the background, while the main thread handles user input.

Please note that the `receive_thread` is set as a daemon thread using `receive_thread.daemon = True`. This means that the thread will automatically exit when the main program (the client) exits.

Now, when you run this client code, it will continuously receive messages from the server in the background while allowing you to type and send messages in the main thread.

Step 6: communication protocol using JSON

Define a communication protocol using JSON for both clients and the server, where each client connects with a name and a room name, you can establish a structure for the messages exchanged between the clients and the server. Here's a simple JSON-based protocol:

Client-to-Server Message Structure:

1. **Connect to Server**:
   - Message Type: "connect"
   - Payload:
     - "name": [Client's Name]
     - "room": [Room Name]

Example:

```
{
  "type": "connect",
  "payload": {
    "name": "Alice",
    "room": "General"
  }
}
```

Server-to-Client Message Structure:

1. **Successful Connection Acknowledgment**:
   - Message Type: "connect_ack"
   - Payload:
     - "message": "Connected to the room."

Example:

```
{
  "type": "connect_ack",
  "payload": {
    "message": "Connected to the room."
  }
}
```

2. **Broadcast Message to Room**:
   - Message Type: "message"
   - Payload:
     - "sender": [Sender's Name]
     - "room": [Room Name]
     - "text": [Message Text]

Example:

```
{
  "type": "message",
  "payload": {
    "sender": "Alice",
    "room": "General",
    "text": "Hello, everyone!"
  }
}
```

3. **Server Notification**:
   - Message Type: "notification"
   - Payload:
     - "message": [Notification Message]

Example:

```
{
  "type": "notification",
  "payload": {
    "message": "Alice has joined the room."
  }
}
```

Using this JSON-based protocol, clients can send a "connect" message to the server when they connect, providing their name and the desired room. The server acknowledges the connection with a "connect_ack" message.

When a client sends a message to the room, it uses a "message" message type, and the server broadcasts this message to all clients in the same room.

Additionally, the server can send "notification" messages to inform clients about events like someone joining or leaving the room.

LAB ASIGNMENT: add this protocol to both CLIENT AND SERVER.