CS 6431 Computer Networks
Name: Naitik Shetty
GW id: G26859345
Date – 09/27/2024

Application Programming Interface Design Document for Song Streaming Service

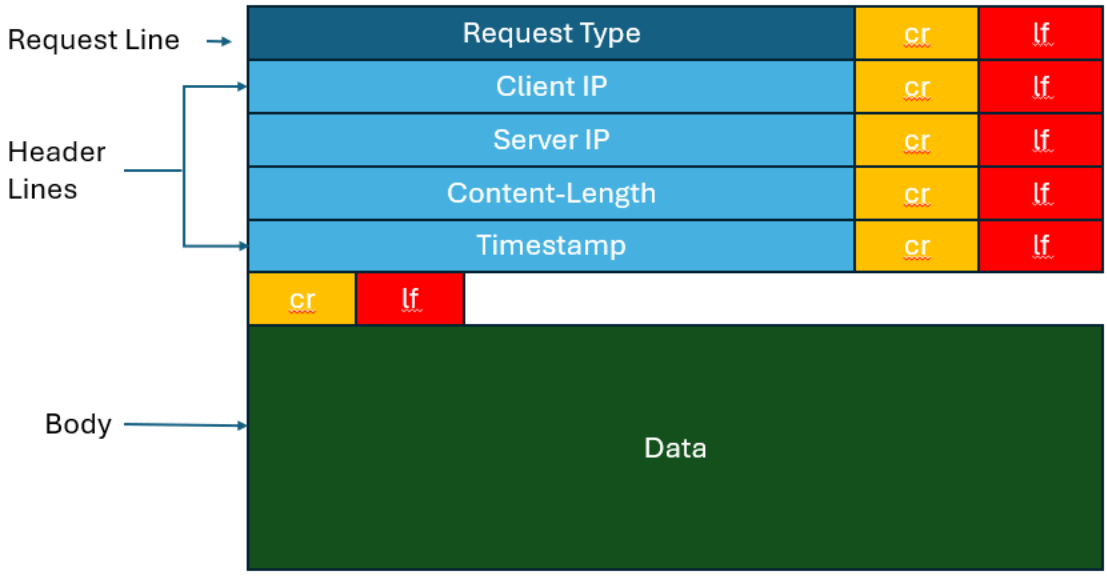
1. Overview:

This document provides the design and implementation of the application programming interface and a custom application protocol for a song streaming application. The application facilitates communication between a client and server to create, manage and stream songs from a playlist. The protocol includes message structures, metadata, communication flow, behaviors triggered by different requests. The documentation provides an in-depth view of the required data structures, behaviors, message structures and flow diagrams to ensure clarity in the system's operation. It also provides the workings of the python files such as Server.py, Client.py, Request.py and Response.py

2. Message Structure:

All the message exchanged between the client and server follow a well-defined structure. including the command, metadata, data. Metadata contains important header information like timestamps, Client IP, Server IP and content length to ensure traceability and reliability of communication.

2.1 Message Structure Diagram:



Name: Naitik Shetty
GW id: G26859345
Date – 09/27/2024

2.2 Sample Request and Response Message Structure:

Request Message:

```
Connected by ('127.0.0.1', 61638)
Received data: Message-Type: GET
Client-IP: 127.0.0.1
Server-IP: 127.0.0.1
Content-Length: 0
Timestamp: 2024-09-29 16:22:34 GMT
```

Response Message:

```
Message-Type: GET_RESPONSE
Client-IP: 127.0.0.1
Server-IP: 127.0.0.1
Content-Length: 311
Timestamp: 2024-09-29 16:22:34 GMT

0 Song 1 Artist 1 Album 1 240
1 Song 2 Artist 1 Album 1 210
2 Song 3 Artist 2 Album 2 420
3 Song 4 Artist 2 Album 2 320
4 Song 5 Artist 3 Album 1 220
5 Song 6 Artist 1 Album 1 250
6 Song 7 Artist 4 Album 2 180
7 Song 8 Artist 4 Album 1 220
8 Song 9 Artist 5 Album 1 260
9 Song 10 Artist 3 Album 1 300
```

2.3 Message Structure Details:

The message structure is defined by 3 components:

1. Request Line: Contains the request or response message that determines the nature of the request made for the appropriate response.
2. Headers: Contains important information or metadata about the request and response such as IP addresses and timestamps.
3. Message Body: Contains data to be sent across during client-server communication.

Name: Naitik Shetty
GW id: G26859345
Date – 09/27/2024

3. Metadata Requirements:

Metadata ensures traceability of messages in communication between the client and server. Each request-response message contains the following metadata in its headers:

- Client IP: The IP address of the client that sends the request or receives the response.
- Server IP: The IP address of the server that processes the request or sends the response.
- Message Length: Message length in bytes to ensure the received data has not been altered in transit ensuring its integrity
- Timestamp: A human-readable timestamp in GMT to synchronize logs.

These fields allow for accurate tracking of communication flow and identification of the parties involved in the exchange.

4. Endline and Delimiting Characters:

In this protocol, message fields are separated by newline characters (\r\n). The newline character is represented by ASCII 0x0D 0x0A, which ensures consistent formatting across platforms.

5. Communication Flow Diagrams:

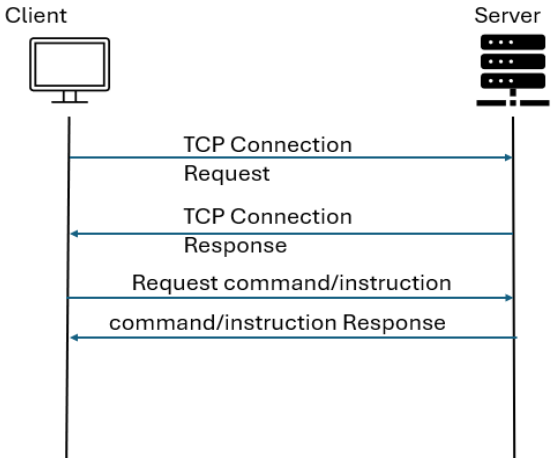
Communication between the client and server occurs in 2 methods.

- Communication for Open Catalog and Design Mode
- Communication for Play Mode.

5.1 Communication for Open Catalog and Design Mode:

This communication uses TCP or Transmission Control Protocol. TCP is a transport layer protocol that ensures reliable data transmission. It ensures this with an initial exchange between client and server before a client request is made. Since the API modes require successful transmission of instructions/data with/without lag hence TCP is the best option for this communication.

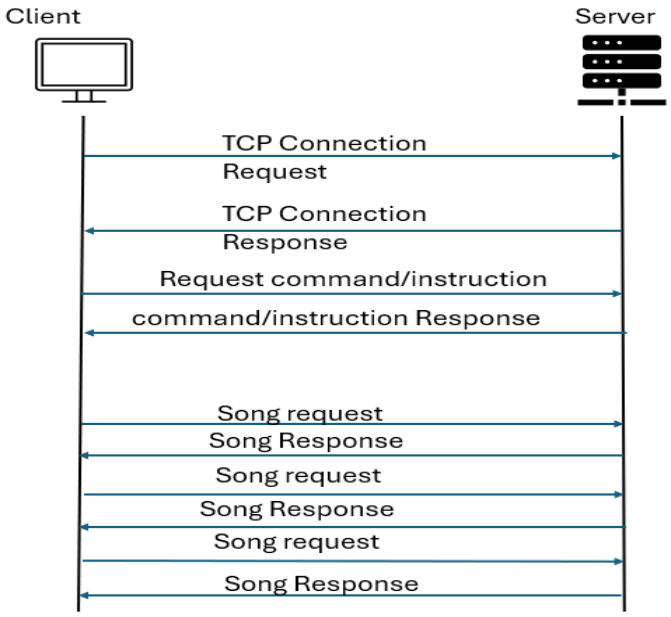
Design Flow Diagram:



5.2 Communication for Play Mode:

This communication uses a combination of both TCP and UDP. While instructions like play song in default, shuffle or loop mode as well as play next song and play last song require TCP connection. To mimic song streaming, the API establishes a UDP connection with the server and exchanges repeated requests and responses till the duration of the song. UDP is used since it is considered best-effort or fastest protocol. Since song streaming needs to ensure no lag in communication as it may ruin the experience, this protocol is best suited to this form of client-server exchange.

Design Flow Diagram:



6. Triggered Behaviors:

6.1 Client-Side Behavior:
Client sends a request and waits for the server to process the response. Depending on the status code:

- **Success**: The client proceeds with the next operation, such as play song or display playlist/catalog.
- **Error**: The client displays an error message and reloads the command menu.

6.2 Server-Side Behavior:
Server receives a request, parses the request action and associated data, performs the corresponding behavior (e.g. play song, add song to playlist, open playlist/catalog etc.), and sends a response with the appropriate response message, metadata and corresponding data to client.

Name: Naitik Shetty
GW id: G26859345
Date – 09/27/2024

7. Database:

The database is a json file called database.json and will act as the database for our song catalog. When the server starts the json file is loaded into a dictionary called catalog. This database follows the following structure:

Field name	Type	Content
id	int	A unique key for each record in the catalog
song_title	string	The title of the song
artist	string	The artist who recorded the song
album_title	string	The album on which the song was released
duration	int	The length of time in seconds of the song

7.1 Objects:

The API communicates in terms of the following objects:

- Catalog – Loads the list of available songs in the database
- Playlist – User created custom playlist
- Now_Playing – Stores the song that is being played when one of the play modes is triggered. Should contain only 1 entry at any given time.
- Playlist_temp – Stores copy of playlist to perform operations.
- Play_Next – Stores the list of songs to be played next.
- Play_Previous – Stores the list of songs that have been played.

These objects follow the same structure as the json file.

8. Python Files operation details:

This section details of how the core components (Client.py, Server.py, Request.py, and Response.py) are structured and interact with each other in this API.

Standard python libraries used:

- Socket – To establish TCP and UDP connection
- Time – To trigger time related function such as time-based loop to mimic song streaming
- Datetime – To get date and time in GMT.
- Random – To shuffle the playlist in random order in Shuffle play mode.
- Json – To import json files on python
- Threading – To handle TCP and UDP connections on the same PORT.

Name: Naitik Shetty
GW id: G26859345
Date – 09/27/2024

8.1 Client.py:

The client is the one that sends requests to the server. It prepares a request message using the send_request function in Request.py and waits for a response from the server. The client handles user inputs for actions like adding a song, playing the playlist, or changing modes.

Functions:

1. Main – runs the run_client function when called

Code snippets:

```
def Main():
   run_client()

Main()
```

2. run_client – When called, it establishes a TCP connection with server and runs Menu function

Code snippets:

```
def run_client():
    # Connect to the server
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as client_socket:
        try:
            client_socket.connect((HOST, PORT))
            client_ip = client_socket.getsockname()[0]
            server_ip = HOST
            print("Connection to server: Successful")
        except Exception as e:
            print(f"Connection to server: Unsuccessful: {e}")
            return
        while True:
            def Menu():
                print("Select:")
                print("1. Open Song Catalog")
```

3. Menu – Displays command line menu and user input to retrieve song catalog, open playlist or quit application.

Code Snippet:

```
    while True:
        def Menu():
            print("Select:")
            print("1. Open Song Catalog")
            print("2. Open Playlist")
            print("3. Quit Application")
            try:
                choice = int(input("Enter your choice: "))
            except ValueError:
                print("Invalid input. Please enter a number.")
            if choice == 1:
                response = Request.send_request(client_socket, "GET", client_ip=client_ip, server_ip=server_ip)
                print(response)
            elif choice == 2:
                Playlist()
            elif choice == 3:
                response = Request.send_request(client_socket, "CLOSE", client_ip=client_ip, server_ip=server_ip)
                print(response)
                client_socket.close()
            else:
                print("Invalid choice. Please enter a number between 1 and 3.")
```

4. Playlist – Displays and provides user input to select Design Mode, Play Mode or return back to Menu
   Code Snippet:

```
def Playlist():
    print("Choose Mode")
    print("1. Design Mode")
    print("2. Play Mode")
    print("3. Return")
    try:
        choice = int(input("Enter your choice: "))
    except ValueError:
        print("Invalid input. Please enter a number.")
    if choice == 1:
        Design_Mode()
    elif choice == 2:
        Play_Mode()
    else:
        Menu()
```

5. Mode_Button – Displays the options of play next song and play last song.
6. Design_Mode – Displays and provides user input for options in the Design Mode. In this mode the user sends requests for retrieving song catalog or creating a playlist by adding or removing songs from loaded song catalog. Option to switch to Play Mode, finding a song in created playlist and retrieving the created playlist are also available.

   Code Snippet:

```
def Design_Mode():
    print("Design Menu:")
    print("1. Open Song Catalog")
    print("2. Open Playlist")
    print("3. Add Song")
    print("4. Remove Song")
    print("5. Find Song in Playlist")
    print("6. Switch to Play Mode")
    print("7. Quit")
    try:
        choice = int(input("Enter your choice: "))
    except ValueError:
        print("Invalid input. Please enter a number.")
        continue
```

7. Play_Mode: Displays and provides user input to select and play a song in 3 submodes: Default,Shuffle,Loop. Default will play the songs one after another as per the originally created playlist. Shuffle will randomly shuffle the created playlist and play songs such that each song in the playlist is played at least once. Loop mode will play the created playlist in a loop. This means that there is no end to the playlist as it will play the first song after last song is reached. After every song is played, the client will ask to skip the next song. If yes is selected, the program skips the song. It plays the song for no.

Code Snippet:

```
def Play_Mode():
    print("Play Menu:")
    print("1. Default")
    print("2. Shuffle")
    print("3. Loop")
    print("4. Stop and Switch to Design Mode")
    print("5. Quit")
    try:
        choice = int(input("Enter your choice: "))
    except ValueError:
        print("Invalid input. Please enter a number.")
```

8. Duration: parses the last characters before space of the retrieved response and converts into integer. This is then sent as input into the time loop to run the loop based on the duration of the song. Sends duration of 1s if characters are not numerical.

8.2 Request.py:
This file defines the structure of a Request message. It contains fields like action(Request type) and metadata(headers). It is responsible for encoding and decoding messages according to the protocol.

Functions:

1. build_header – creates the header file for the request message. Contains metadata like message_type, client ip, server ip, content_length and timestamps as per GMT. Hence this function returns the header to the send_request and send_request_UDP functions.

Code snippet:

```
# Function to build headers with timestamp, IP addresses, and other metadata
def build_header(message_type, client_ip, server_ip, content_length=0):
    timestamp = datetime.datetime.utcnow().strftime('%Y-%m-%d %H:%M:%S GMT')
    header = (f"Message-Type: {message_type}\r\n"f"Client-IP: {client_ip}\r\n"f"Server-IP:
    return header
```

2. send_request – creates the final request message to be sent to the server. This function combines the headers with message body to form the message structure of the application protocol. The length of the message_body is 0. This message body is added to retrieve data from the server.

   Code snippet:

```
# Function to send commands to the server and receive responses
def send_request(client_socket, message_type, message_body="", client_ip="127.0.0.1", server_ip="127.0.0.1"):
    header = build_header(message_type, client_ip, server_ip, len(message_body))
    request = header + "\r\n" + message_body
    try:
        client_socket.send(request.encode())
        response = client_socket.recv(1024).decode()
        return response
    except Exception as e:
        return f"Error: {e}"
```

3. send_requestUDP – Responsible for the same function as send_request but for UDP connection.
   Code snippet:

```
def send_requestUDP(clientSocketUDP, message_type, message_body="", client_ip="127.0.0.1", server_ip="127.0.0.1"):
    header = build_header(message_type, client_ip, server_ip, len(message_body))
    request = header + "\r\n" + message_body
    try:
        clientSocketUDP.sendto(request.encode(),(client_ip,12000))
        response, server_ip = clientSocketUDP.recvfrom(1024)
        response = response.decode()
        return response
    except Exception as e:
        return f"Error: {e}"
```

8.3 Server.py:
The server loads the json file and is listening for TCP/UDP connection to receive client requests. It then sends request to handle_client function in Response.py to process the actions and send back the appropriate response.
Functions:

1. load_catalog – Loads the json file database.json into the dictionary catalog.
   Code snippet:

```
# Loads the song catalog from the JSON file
def load_catalog():
    with open(CATALOG_FILE, 'r') as file:
        catalog = json.load(file)
    return catalog['catalog']
```

2. run_server – Binds the host to port 12000 and listens for TCP requests these request and then sent to handle_client in Response.py for parsing, processing and response.
   Code snippet:

```python
# Main server function
def run_server():
    catalog = load_catalog()  # Load catalog from JSON file
    print("Song catalog loaded.")

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:
        server_socket.bind((HOST, PORT))
        server_socket.listen()

        print(f"TCP Server is listening on {HOST}:{PORT}")
        while True:
            connectionSocket, addr = server_socket.accept()
            Response.handle_client(connectionSocket, addr, catalog)
```

3. Run_serverUDP – Binds the host to port 12000 and UDP requests these request and then sent to handle_clientUDP in Response.py for parsing, processing and response.
   Code snippet:

```python
def run_serverUDP():
    serverSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    serverSocket.bind(('', PORT))
    print( "The UDP server is ready to receive: " )
    while(True):

        Response.handle_clientUDP(serverSocket)
```

4. Main – threads run_server and run_serverUDP to run on the same port.
   Code snippet:

```python
def Main():
    tcp_thread = threading.Thread(target=run_server)
    udp_thread = threading.Thread(target=run_serverUDP)
    tcp_thread.start()
    udp_thread.start()

    tcp_thread.join()
    udp_thread.join()

Main()
```

Name: Naitik Shetty
GW id: G26859345
Date – 09/27/2024

8.4 Response.py:
This file defines the structure of a Response message. It contains fields like action (Request type) and metadata(headers). It is responsible for encoding and decoding messages according to the protocol.

Functions:

1. build_header- creates the header file for the request message. Contains metadata like message_type, client ip, server ip, content_length and timestamps as per GMT. Hence this function returns the header to the handle_client and handle_clientUDP functions.

   Code Snippet:

```
# Function to build headers
def build_header(response_type, client_ip, server_ip, content_length=0):
    timestamp = datetime.datetime.utcnow().strftime('%Y-%m-%d %H:%M:%S GMT')
    header = (f"Message-Type: {response_type}\r\n"f"Client-IP: {client_ip}\r\n"f"Serv
    return header
```

2. handle_client - creates the final response message to be sent to the server. This function combines the headers with message body to form the message structure of the application protocol. It parses and processes the request based on the request_type and send the appropriate response in the message body of the response structure. The length of the message body is updated in the metadata of the header.

   Code Snippet:

```
# Function to handle client requests
def handle_client(connectionSocket, addr, catalog):
    print(f"Connected by {addr}")
    client_ip = addr[0]
    server_ip = HOST


    playlist = []  # Initialize an empty playlist
    playlist_temp = []
    now_playing = []
    play_next = []
    play_previous = []
    while True:
        try:
            data = connectionSocket.recv(1024).decode()
            if not data:
                break

            print(f"Received data: {data}")
            request_lines = data.split("\r\n")
            request_type = request_lines[0].split(": ")[1].strip()  # Extract the message type
```

3. Handle_clientUDP - creates the final response message to be sent to the server. This function combines the headers with message body to form the message structure of the application protocol. It parses the song request, processes the song request and sends the song data as the response in the message body.
   Code Snippet:

```
def handle_clientUDP(serverSocket):
    server_ip = HOST
    while True:
        try:
            data1, client_ip  = serverSocket.recvfrom(2048)
            data1=data1.decode()
            if not data1:
                break

            print(f"Received data: {data1}")
            request_lines1 = data1.split("\r\n")
            request_type1 = request_lines1[0].split(": ")[1].strip()  # Extract the message type
            if request_type1 == "SONG_DATA_UDP":
                response = "song data UDP .... song data UDP"
                header = build_header("SONG_DATA_RESPONSE_UDP", client_ip, server_ip, len(response))
                response = header + " " + response
                serverSocket.sendto(response.encode(),client_ip)

        except Exception as e:
            print(f"Error: {e}")
            break
```

9. Requests and Response:

Server sends different responses based on the message_type/request_type:

1. GET – retrieves the song catalog from the loaded json file and send it as a response
2. GET_LIST – retrieves the user created custom playlist and send it as a response
3. ADD – adds song to the playlist
4. REMOVE – removes song from the playlist based on “id”
5. FINDS – finds song from the playlist based on “id”
6. CLOSE – ends connection with server and closes the application
7. DEFAULT – plays the original playlist
8. NEXT – moves to the next song of the playlist
9. LAST – moves to the previous song of the playlist
10. SHUFFLE – randomizes the playlist such that every song the the playlist is played at least one regardless of repetition
11. SHUFFLE_NEXT – plays the next song in the shuffled playlist
12. SHUFFLE_LAST – plays the previous song in the playlist
13. LOOP – plays the playlist in a loop
14. LOOP_NEXT - plays the next song in the playlist for LOOP Mode
15. LOOP_LAST - plays the previous song in the playlist for LOOP Mode

16. SONG_DATA – sends data continuously every 2 seconds on request for the duration of the song. (The request message for this is hashed since the program uses SONG_DATA_UDP for the same, this is meant as an alternative for song data transmission in TCP.
17. SONG_DATA_UDP - sends data continuously every 2 seconds on request for the duration of the song. (uses UDP protocol)

10. **Requirements:**

- Make sure all files (Client.py, Server.py, Request.py, Response.py and database.json) are present in the same directory.
- Run the Client.py and Server.py from that directory in the command

11. **Conclusion:**

This document provides a comprehensive overview of a song streaming application protocol for this assignment including the workings of the python files Client.py, Server.py, Request.py, and Response.py. The message structures, communication flow, metadata, and handling of newline characters are detailed to ensure that the protocol remains clear, reliable, platform-independent and easy to execute. The protocol has been designed with integrity and accuracy and best effort in mind, with a focus on creating a seamless message exchange between between client and server.

Name: Naitik Shetty
GW id: G26859345
Date – 09/27/2024

12. References:

1. https://docs.python.org/3/library/socket.html
2. https://docs.python.org/3/library/time.html
3. https://www.fortinet.com/resources/cyberglossary/user-datagram-protocol-udp#:~:text=User%20Datagram%20Protocol%20(UDP)%20is,destination%20before%20transferring%20the%20data.
4. https://www.fortinet.com/resources/cyberglossary/tcp-ip#:~:text=2.,data%20in%20digital%20network%20communications.
5. https://docs.python.org/3/library/datetime.html