Documentation for VoIP Client Script

Overview

This Python script implements a VoIP (Voice over IP) client that records, transmits, receives, and plays back audio data in real-time over a network. It uses UDP for audio data transmission and TCP for initial connection and handshakes with a server.

Key Features

- **TCP Handshake**: Establishes a communication channel with the server before sending audio data over UDP.
- **Audio Recording and Transmission**: Captures audio data from the system's microphone, encodes it into packets, and sends it over UDP to a target machine (client/server).
- **Jitter Buffer**: Handles network delays, packet loss, and reordering to ensure smooth audio playback.
- **Audio Playback**: Plays back the received audio data in real-time with synchronization.
- **Heartbeat Messages**: Periodically sends heartbeat messages to the server to verify that the connection is active.
- **End of Transmission (EOT)**: Sends an EOT signal to the server to indicate the end of transmission.
- **Robust Error Handling**: Handles network timeouts, retries, and errors in audio packet transmission and reception.

Requirements

- **Python Libraries**:
    - `sounddevice`: For audio input and output.

- `numpy`: For numerical processing of audio data.
- `socket`: For network communication.
- `struct`: For packing and unpacking binary data.
- `threading`: For concurrent execution of different client functions.
- `time`: For time-related operations (e.g., handling timeouts and intervals).
- `argparse`: For parsing command-line arguments.
- `collections.deque`: For implementing the jitter buffer (used to handle out-of-sequence packets).

Class Descriptions

`JitterBuffer`

The `JitterBuffer` class is responsible for storing and reordering audio packets to ensure smooth playback despite network-induced delays or out-of-sequence packets.

Methods

- **`__init__(self, max_size)`**: Initializes the jitter buffer with a specified maximum size.
- **`add_packet(self, seq_num, audio_data)`**: Adds a new packet to the buffer, ensuring that out-of-sequence packets are handled appropriately.
- **`get_packet(self)`**: Retrieves the next packet from the buffer in sequence.

`Client`

The `Client` class manages the VoIP client's core functionality, including network communication (via TCP and UDP), audio recording and playback, jitter buffer handling, and periodic heartbeat messages.

Initialization

- `__init__(self, udp_port, server_ip, target_ip)`: Initializes the client, setting up UDP communication, jitter buffer, and flags for recording and transmission.

Methods

- `tcp_handshake(self)`: Performs a TCP handshake with the server to initiate communication.
- `send_heartbeat(self)`: Periodically sends heartbeat messages to the server to confirm the connection is active.
- `record_and_send_audio(self)`: Records audio from the system's microphone, splits it into packets, and sends it via UDP to the target client/server.
- `play_audio(self)`: Plays back the audio received in packets, ensuring synchronization and jitter handling.
- `receive_audio(self)`: Receives audio packets from the network and stores them in the jitter buffer.
- `send_eot(self)`: Sends an End of Transmission (EOT) signal to the server to indicate that audio transmission has ended.
- `start(self)`: Initiates the VoIP client's operations, including handshake, recording, receiving, and playback.

Constants

- `SAMPLE_RATE`: Audio sample rate in Hz (default: 44100).
- `CHANNELS`: Number of audio channels (default: 1, i.e., mono).

- **`CHUNK_SIZE`**: Number of audio samples per chunk (default: 882).
- **`BYTES_PER_PACKET`**: Size of each audio packet in bytes (default: 1764 bytes for 16-bit mono audio).
- **`PACKET_SIZE`**: Size of each UDP packet (default: 2200 bytes, including headers).
- **`EOT_SEQ_NUM`**: Special sequence number used to indicate the End of Transmission (EOT) packet.
- **`JITTER_BUFFER_SIZE`**: Maximum number of packets the jitter buffer can hold (default: 8).
- **`PLAYBACK_INTERVAL_MS`**: Time interval (in milliseconds) between audio playback iterations (default: 10ms).
- **`TCP_PORT`**: TCP port used for the initial handshake with the server (default: 8888).
- **`UDP_SERVER_PORT`**: UDP port used for transmitting and receiving audio data (default: 9999).
- **`TIMEOUT`**: Timeout duration (in seconds) for network operations (default: 2s).
- **`MAX_RETRIES`**: Maximum number of retries allowed for the TCP handshake (default: 3).
- **`HEARTBEAT_INTERVAL`**: Interval (in seconds) for sending heartbeat packets to the server (default: 30s).
- **`SILENCE_THRESHOLD`**: Amplitude threshold used to warn if the audio input is too quiet (default: 0.01).

Main Function

`main()`

The entry point of the script. It parses command-line arguments (server IP, UDP port, and target IP) and initiates the client's operations by calling the `Client.start()` method.

Command-Line Arguments

- **`server_ip`**: The IP address of the server to connect to.
- **`udp_port`**: The local UDP port for the client to send and receive audio data.
- **`target_ip`**: The IP address of the target client/server for communication.

Usage

Running the client: After setting up the server, the client can be started with the following command:

bash
CopyEdit
```
python voip_client.py <server_ip> <udp_port> <target_ip>
```

1. Replace `<server_ip>`, `<udp_port>`, and `<target_ip>` with appropriate values.
2. **Stopping the client**: To stop the client, type `quit` in the terminal, or press `Ctrl+C` to terminate the process.

Example

Start the client with the following command:

bash
CopyEdit
```
python voip_client.py 192.168.1.1 8888 192.168.1.2
```

This will connect the client to a server at IP `192.168.1.1` on port `8888` and communicate with another client at IP `192.168.1.2`.

Error Handling

- **TCP Handshake Failure**: If the client fails to establish a connection with the server, it will retry the handshake up to `MAX_RETRIES` times before aborting.
- **UDP Packet Transmission Errors**: If a packet cannot be sent due to network issues, an error message is logged, and the transmission continues.
- **Timeouts**: If the client experiences a network timeout, it will retry the operation up to the configured limit.

Server Program Documentation

Overview

This Python server program is designed to handle both UDP and TCP communication. It is intended for use in network-based applications, particularly those involving audio data transmission. The server handles client registration, manages timeouts, forwards UDP audio packets to target clients, and monitors client heartbeats to ensure active connections.

Key Features:

- **TCP Communication**: Handles client registration, heartbeat monitoring, and disconnection over TCP.
- **UDP Communication**: Receives and forwards UDP audio packets to registered clients.
- **Heartbeat System**: Monitors and removes clients that fail to send a heartbeat within the timeout period.
- **Client Management**: Supports a limited number of clients, with mechanisms to manage their registration and disconnection.

Dependencies

This program requires the following Python modules:

- `socket`: For creating network connections (both TCP and UDP).

- `threading`: For concurrent execution of different tasks (e.g., TCP client handling, heartbeat monitoring).
- `time`: For managing timeouts and heartbeats.
- `select`: For handling non-blocking socket operations.
- `struct`: For packing and unpacking binary data (e.g., extracting client ports).

These modules are part of the Python standard library and do not require additional installation.

Constants

Constant	Description
`TCP_IP`	IP address for the TCP server to bind to (set to `"0.0.0.0"` for all interfaces).
`TCP_PORT`	Port number for the TCP server to listen on (default is `8888`).
`UDP_IP`	IP address for the UDP server to bind to (set to `"0.0.0.0"` for all interfaces).
`UDP_PORT`	Port number for the UDP server to listen on (default is `9999`).
`BUFFER_SIZE`	Size of the buffer for receiving UDP data (set to `2200` bytes).
`MAX_CLIENTS`	Maximum number of clients allowed to register with the server (default is `10`).

`TCP_TIMEOUT`	Timeout for TCP client connections in seconds (default is `30` seconds).
`HEARTBEAT_TIMEOUT`	Timeout for heartbeat messages in seconds, after which the client is considered dead (default is `120` seconds).
`HEARTBEAT_INTERVAL`	Interval in seconds between heartbeat checks (default is `10` seconds).

Data Structures

- **`clients`**: A dictionary mapping client IPs to a tuple of `(listen_port, target_ip, target_port)`. It stores the client's registration details, including the target IP and port for forwarding UDP audio packets.
- **`last_heartbeat`**: A dictionary mapping client IPs to the last time they sent a heartbeat message. Used to monitor client activity.
- **`clients_lock`**: A threading lock to synchronize access to the `clients` dictionary, preventing race conditions during updates.
- **`heartbeat_lock`**: A threading lock to synchronize access to the `last_heartbeat` dictionary, ensuring safe updates during heartbeat checks.

Functions

`handle_client(data, addr)`

Description: Handles the reception of UDP data and forwards it to the appropriate target client.

Parameters:

- `data`: The UDP data packet received from the client.
- `addr`: The address (IP and port) of the client that sent the data.

Behavior:

- If the client is registered, forwards the received data to the target IP and port.
- If the client is not registered, ignores the data and logs a message.

`receive_audio()`

Description: Listens for incoming UDP audio packets and processes them.

Behavior:

- Sets the UDP socket to non-blocking mode using `select.select()`.
- Continuously listens for UDP packets and forwards them to the appropriate target using the `handle_client()` function.

`handle_tcp_client(tcp_conn, tcp_addr)`

Description: Handles TCP client connections, processes messages like HELLO, HEARTBEAT, and DISCONNECT.

Parameters:

- `tcp_conn`: The TCP connection object for communication with the client.
- `tcp_addr`: The address (IP and port) of the TCP client.

Behavior:

- Handles client registration when the client sends a `HELLO` message.

- Responds to heartbeat messages (`HEARTBEAT`).
- Removes clients from the registration list when they send a `DISCONNECT` message or after a timeout.

`tcp_handshake_listener()`

Description: Listens for incoming TCP connections and handles them by spawning a new thread for each client.

Behavior:

- Listens on the specified `TCP_PORT` for incoming TCP connections.
- Accepts connections and spawns a new thread to handle the client using the `handle_tcp_client()` function.

`heartbeat_monitor()`

Description: Monitors the heartbeat of connected clients and removes clients that have timed out.

Behavior:

- Periodically checks if any client has failed to send a heartbeat within the specified `HEARTBEAT_TIMEOUT`.
- Removes clients that have timed out from the `clients` dictionary.

`main()`

Description: The main entry point of the server program. Initializes the server components.

Behavior:

- Creates and binds a UDP socket for receiving audio packets.
- Starts the TCP handshake listener and heartbeat monitor in separate threads.
- Calls the `receive_audio()` function to listen for and forward UDP packets.

Server Workflow

1. **Client Registration**:
    - A client connects via TCP and sends a `HELLO` message with its port and target IP.
    - The server registers the client, stores its details, and assigns the target IP/port for forwarding UDP packets.
2. **UDP Packet Forwarding**:
    - The server receives UDP audio packets from clients.
    - It forwards the UDP data to the target client based on the registration information.
3. **Heartbeat Monitoring**:
    - Clients must periodically send `HEARTBEAT` messages to the server.
    - The server tracks the last heartbeat for each client. If a client exceeds the `HEARTBEAT_TIMEOUT`, it is considered dead and removed.
4. **Client Disconnection**:
    - A client can disconnect by sending a `DISCONNECT` message.
    - The server removes the client from its registration list and sends a confirmation.

Error Handling

- **TCP Timeouts**: If a TCP connection times out, the server responds with a `TIMEOUT` message and closes the connection.
- **Invalid Data**: The server sends an `INVALID` message to clients that send unrecognized or malformed data.
- **Max Clients Exceeded**: If the server reaches its client limit (`MAX_CLIENTS`), it responds with a `FULL` message to any new clients attempting to connect.

Running the Server

1. Ensure Python is installed and the necessary modules are available (standard library modules).

Run the server script using the command:

bash
CopyEdit
```
python server.py
```

2.
3. The server will start listening for both UDP and TCP connections.

Configuration

You can adjust the following parameters in the script for customization:

- **TCP_PORT**: Port for the TCP server.
- **UDP_PORT**: Port for the UDP server.
- **MAX_CLIENTS**: Maximum number of clients the server will accept.
- **TIMEOUTS**: Adjust `TCP_TIMEOUT`, `HEARTBEAT_TIMEOUT`, and `HEARTBEAT_INTERVAL` as needed