VOICE OVER INTERNET PROTOCOL

Team Members:

1. S Sruthi Priya - 22BLC1069
2. R Sai Kumar - 22BLC1031
3. L Shaneel Reddy - 22BLC1164

Abstract:

This project focuses on the development of a basic Voice Over Internet Protocol (VOIP) system using Python. It facilitates real-time audio communication between a client and a server over a network using the User Datagram Protocol (UDP). The client captures audio input from a microphone, transmits it to the server, which then plays the audio in real-time. The implementation leverages Python libraries such as PyAudio for audio processing, socket programming for networking, and Tkinter for creating a simple GUI. The system emphasises low-latency audio streaming suitable for applications where speed is more critical than guaranteed delivery.

Introduction & Background:

Voice over Internet Protocol (VoIP) is transforming the way we communicate by enabling voice calls over the Internet instead of traditional phone lines. This innovative technology converts your voice into digital data packets, which are transmitted via broadband connections. VoIP not only offers significant cost savings on both domestic calls and international calls but also provides a suite of advanced features such as call forwarding, voicemail-to-email, and video conferencing. With the increasing demand for real-time communication, VOIP technologies have become fundamental for enabling voice communication over the internet. Traditional telecommunication systems rely on circuit-switched networks, which are often expensive and less flexible. VOIP, in contrast, uses packet-switched networks, providing a cost-effective and scalable solution for voice communication.

This project demonstrates a basic VOIP system using Python. It employs UDP for audio data transmission, as it is faster than TCP and better suited for time-sensitive applications like voice and video. While UDP sacrifices reliability (e.g., packets may be dropped or arrive out of order), it ensures minimal latency, which is critical for voice communication.

Objective / Problem Statement

The primary objective of this project is to create a simple VOIP system that:

1. Captures and transmits audio data from a client to a server in real time.
2. Plays back the received audio data on the server with minimal delay.

3. Uses UDP for faster data transmission, accepting trade-offs in packet reliability.
4. Implements a basic graphical user interface (GUI) to control the audio transmission (e.g., start/stop/mute).
5. Demonstrates the core principles of VOIP systems for educational or prototype purposes.

The main challenge lies in achieving low-latency communication while ensuring reasonable audio quality, despite potential packet loss inherent in UDP-based communication.

Methodology

1. **Technology Stack**:
    - **PyAudio**: For capturing and playing audio data.
    - **Socket Programming**: For sending and receiving audio data over a network.
    - **Tkinter**: For creating a simple GUI to control audio transmission.
    - **UDP Protocol**: For lightweight and fast data transfer.
2. **Client Implementation**:
    - Captures audio input using PyAudio.
    - Sends the audio data in chunks to the server via a UDP socket.
    - Provides GUI controls for starting and muting audio transmission using Tkinter.
3. **Server Implementation**:
    - Listens for incoming audio data from the client on a specific UDP port.
    - Plays the received audio data in real-time using PyAudio.
4. **Concurrency**:
    - Utilises Python's threading module to handle real-time audio streaming alongside GUI responsiveness.
5. **System Architecture**:
    - A client-server model where the client captures and transmits audio data, and the server processes and plays it back.

Code and Output:

server.py

```
import pyaudio
import socket
chunk = 1024
pa = pyaudio.PyAudio()

stream = pa.open(format=pyaudio.paInt16,
        channels=1,
        rate=10240,
        output=True)
host = ''
```

```
port = 1234
size = 4096
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((host, port))

print("Server is now running\n======================")

while True:
    data, addr = sock.recvfrom(size)
    print(f"{addr}")
    if data:
        stream.write(data)
sock.close()
stream.close()
pa.terminate()
print("Server has stopped running")
```

This script sets up a basic audio streaming server using UDP (User Datagram Protocol) and the PyAudio library. The server listens for audio data packets sent over the network and plays them in real-time.

How It Works:

1. Audio Setup:
   The PyAudio library initialises an audio stream that plays 16-bit mono audio at a sampling rate of 10,240 samples per second. This stream sends the received audio data to the system's audio output (like speakers or headphones).

2. UDP Server Setup:
   A UDP server is created using Python's socket library. It listens on port 1234 for incoming audio data. UDP is used because it prioritises low-latency communication over reliability (suitable for streaming).

3. Main Process:
   The server continuously waits for incoming UDP packets containing raw audio data. When a packet is received:
   - It reads the data.
   - Plays the audio data in real-time through the PyAudio stream.
   - Prints the sender's address for reference.

4. Closing the Server:
   When stopped, the server cleans up resources, closes the audio stream, and terminates the PyAudio instance.

Explanation:

This is a simple implementation for real-time audio streaming over a network. It could be used in applications like:
- Live voice communication.
- Remote audio monitoring.
- Broadcast systems.

However, it's a basic script and could benefit from improvements like error handling, multi-client support, or audio compression.

client.py
```
import pyaudio
import socket
import threading
from tkinter import *


chunk = 1024
FORMAT = pyaudio.paInt16
CHANNELS = 1
RATE = 10240

p = pyaudio.PyAudio()

stream = p.open(format=FORMAT,
                channels=CHANNELS,
                rate=RATE,
                input=True,
                frames_per_buffer=chunk)

host = '192.168.179.207'
port = 1234
size = 1024
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.connect((host, port))

class VOIP_FRAME(Frame):
    def speakStart(self):
        self.mute = False
        t = threading.Thread(target=self.speak)
        t.start()

    def muteSpeak(self):
        self.mute = True
        print("You are now muted")

    def speak(self):
        while not self.mute:
            data = stream.read(chunk)
            s.send(data)

    def createWidgets(self):
        self.speakb = Button(self, text="Speak", command=self.speakStart)
        self.speakb.pack(side="left")
        self.muteb = Button(self, text="Mute", command=self.muteSpeak)
        self.muteb.pack(side="left")

    def _init_(self, master=None):
        self.mute = True
        Frame._init_(self, master)
        self.pack()
        self.createWidgets()

root = Tk()
app = VOIP_FRAME(master=root)
app.mainloop()
root.destroy()
s.close()
stream.close()
p.terminate()
```

This script is a simple Voice-over-IP (VoIP) application that lets you stream your microphone input to a specified server over a network using UDP. It includes a graphical user interface (GUI) built with tkinter, allowing users to control the audio stream with "Speak" and "Mute" buttons.

Key Features

1. Audio Streaming:
   The script uses the PyAudio library to capture live audio from the microphone. It streams this audio data in real-time to a server using a UDP connection.

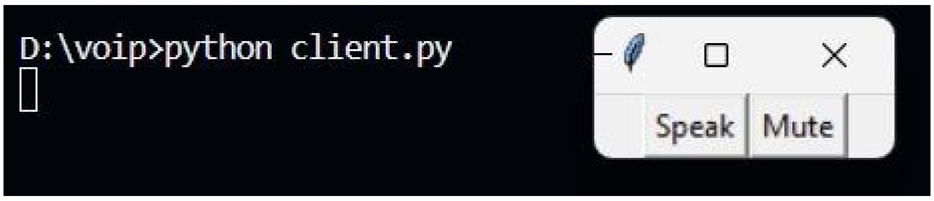
2. GUI for Control:
   - Speak Button: Starts streaming audio to the server.
   - Mute Button: Stops streaming audio.

3. Multithreading:
   Audio streaming runs in a separate thread, ensuring the GUI remains responsive while transmitting data.

4. UDP Connection:
   Audio data is sent over a lightweight and fast UDP connection, suitable for real-time applications where low latency is prioritised.

How It Works

1. The program captures audio from the microphone using PyAudio, with a predefined sampling rate and buffer size.
2. When the "Speak" button is clicked, audio streaming starts in a separate thread. Audio data is sent in chunks to the server's IP address and port.
3. Clicking the "Mute" button stops the audio transmission.
4. The GUI is built using tkinter, providing a simple interface for controlling the streaming process.



```
D:\voip>python client.py
```

Results

1. **Functionality**:
   - The system successfully captures audio from the client, transmits it to the server, and plays it in real-time.
   - The GUI allows users to start and stop the audio transmission effectively.
2. **Performance**:
   - Achieves low latency suitable for real-time communication.
   - Handles minor packet losses gracefully without significant degradation in audio quality.
3. **Limitations**:
   - No error correction or retransmission mechanism for lost packets.
   - Limited to local network communication due to the absence of NAT traversal or encryption.

Conclusion

In conclusion, this project successfully implements a basic VOIP system that enables real-time audio communication between a client and server using UDP. It highlights the fundamental principles of VOIP, including fast, low-latency data transmission and the use of simple audio processing and GUI controls. While the system meets its objectives, it also reveals areas for improvement, such as handling packet loss, enhancing audio quality, and implementing security measures. Overall, this project provides a strong foundation for understanding VOIP systems and serves as a starting point for developing more advanced and feature-rich communication solutions.

Future Work Possibilities

1. **Error Handling and Quality Improvements**:
    - Implement mechanisms for handling packet loss and jitter (e.g., buffering or error correction).
2. **Cross-Network Communication**:
    - Incorporate NAT traversal techniques like STUN/TURN to allow communication across different networks.
3. **Audio Enhancements**:
    - Add noise suppression and echo cancellation to improve audio quality.
    - Support stereo audio by extending to multiple channels.
4. **Advanced GUI**:
    - Enhance the GUI to include features like volume control, participant lists, and connection status indicators.

This project lays the groundwork for exploring these advanced features, offering a stepping stone for building more robust and feature-rich VOIP systems.