

Real-Time Audio Streaming over TCP/IP using C++

Mini Project Report of

Computer Networks Lab (CSE 3162)

Student 1: Aryan Mangla(Reg. no. 210905028)

Student 2: Vaibhav Sandhir(Reg. no. 210905152)

Student 3: Kushala Sarada A V(Reg. no. 210905189)

Abstract

In today's digital age, the significance of efficient and low-latency audio communication systems cannot be overstated. As remote communication becomes the norm, there is a pressing need for robust solutions that can facilitate seamless audio exchanges. This report delves deep into the intricacies of enhancing audio communication through socket programming, with a special focus on a TCP-based audio streaming system.

Our comprehensive study unravels the practical applications of socket programming, with particular emphasis on the TCP protocol, in revolutionizing real-time audio communication. The main objective of this report is to demystify the architecture, implementation, and implications of this cutting-edge technology. By decoding the core concepts and mechanisms underlying the system, we aim to empower organizations and individuals alike to unlock new possibilities in creating more sophisticated and efficient audio communication systems.

The benefits of a TCP-based system, such as its reliability and orderliness in data transmission, are meticulously analyzed. Moreover, we provide actionable insights into addressing potential challenges, including latency and bandwidth constraints.

Through real-world examples and case studies, this report illustrates how diverse sectors, from telemedicine to online gaming, can leverage this technological breakthrough. It is our conviction that the incorporation of socket programming in a TCP-based audio streaming system is not just an enhancement, but an essential step forward in the ongoing evolution of communication technologies. By harnessing the power of this innovation, we can unlock new dimensions in audio communication, paving the way for more immersive and interactive experiences.

Contents

1	Introduction	1
2	Literature Review	1
2.1	Audio Streaming Technologies	1
2.2	Socket Programming	1
2.3	Real-time Communication Systems	1
2.4	Network Protocols for Audio Communication	1
2.5	Challenges and Solutions in Audio Streaming	2
2.6	Conclusion	2
3	Methodology	2
3.1	Hardware and Software Tools	2
3.2	Network Topology and Protocols	2
4	Implementation Details	2
4.1	Audio Capturing and Playback	2
4.2	Network Communication	3
4.3	Network Configuration	3
5	Code Implementation	4
5.1	Client Code	4
5.2	Server Code	6
6	Results and Analysis	9
7	Discussion	10
8	Conclusion	10
9	Future Work	11
10	References	12
A	Appendices	13

1. Introduction

The project focuses on the development of a client-server application that enables real-time audio data transmission over a network. Utilizing the C++ programming language, the system integrates the Winsock library for network communication and the Windows Multimedia System for audio capturing and playback.

The client captures audio from a microphone, encodes it, and transmits it to the server via a TCP socket. On the server side, the received audio data is decoded and played back through the speakers. This setup exemplifies the practical application of socket programming and audio processing in real-time communication systems, potentially laying the groundwork for advancements in VoIP, online gaming, and live streaming platforms.

2. Literature Review

The domain of audio streaming and real-time communication has been extensively studied, leading to a multitude of technological advances aimed at improving user experiences and system performance.

2.1 Audio Streaming Technologies

Significant advancements in audio streaming technologies have paved the way for a variety of platforms and applications. Protocols like RTP (Real-time Transport Protocol) have been instrumental in facilitating audio and video streaming over the internet (Perkins, C., & Westerlund, M., 2006).

2.2 Socket Programming

As a cornerstone of real-time communication, socket programming creates conduits for data transfer across networks. TCP's reliability is ensured through established connections before data transmission, a topic well-documented in the literature (Stevens, W. R., 1990).

2.3 Real-time Communication Systems

Real-time communication systems, like VoIP and online gaming, demand low latency and high fidelity in audio transmission. The significance of codec selection and network management in optimizing audio quality has been well noted in studies (Johnston, A. B., 2004).

2.4 Network Protocols for Audio Communication

A wide range of network protocols for audio communication have been analyzed for their applicability in various scenarios. Despite UDP's low latency benefits, its lack of reliability when compared to TCP has been a subject of discussion (Postel, J., 1980).

2.5 Challenges and Solutions in Audio Streaming

Latency, jitter, and packet loss pose considerable challenges in audio streaming. Adaptive jitter buffering and error correction mechanisms are among the solutions proposed to address these issues (Bolot, J., & Vega-Garcia, A., 1998).

2.6 Conclusion

This project builds upon existing research, applying practical TCP-based socket programming to audio streaming, and aims to overcome its challenges to develop a stable and efficient system.

3. Methodology

The methodology for our Real-Time Audio Streaming over TCP/IP using C++ project involves developing a client-server application to facilitate seamless audio data transmission over a network. This process includes capturing audio from a client's microphone, encoding and transmitting it to the server, which then decodes the data and outputs it through speakers.

3.1 Hardware and Software Tools

- **Hardware:** Microphone and speakers.
- **Software:** C++ programming language, Winsock library for network communication, and Windows Multimedia System for audio capturing and playback.

3.2 Network Topology and Protocols

The project operates within a client-server network topology. The client application captures audio input from a microphone, encodes it, and transmits it over the network. The server receives the audio data, decodes it, and then plays it back through speakers. TCP/IP protocol is employed to ensure reliable transmission of data between the client and server.

4. Implementation Details

4.1 Audio Capturing and Playback

- The client application captures audio from the system's default microphone using the `waveInOpen` and `waveInStart` functions from the Windows Multimedia System.
- The server receives the audio data and plays it back with the `waveOutOpen` and `waveOutWrite` functions, also from the Windows Multimedia System.

4.2 Network Communication

- Network communication between the client and server is facilitated using the Winsock library.
- TCP sockets are used to establish and maintain a reliable connection for audio data transmission.

4.3 Network Configuration

Below is the network diagram illustrating the client-server topology and protocols for the Real-Time Audio Streaming over TCP/IP project:

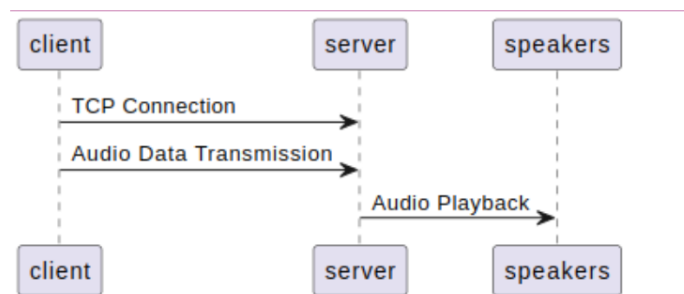


Figure 1: Network Configuration

Configuration Details:

- The **client** captures audio from a microphone, encodes it, and transmits it over the network.
- The **server** receives the audio data, decodes it, and outputs it through speakers.

Hardware and Software Utilization:

- **Microphone:** For capturing audio input on the client side.
- **Speakers:** For outputting audio on the server side.
- **C++ Programming Language:** For developing the client-server application.
- **Winsock Library:** For handling network communication.
- **Windows Multimedia System:** For managing audio capture and playback.

This foundation facilitates the development of real-time communication applications such as VoIP, online gaming, and live streaming platforms, requiring low latency and high-quality audio transmission.

5. Code Implementation

5.1 Client Code

Below is the C++ code snippet for the TCP/IP client side of the application:

```
1  #include <iostream>
2  #include <WS2tcpip.h>
3  #include <Windows.h>
4  #include <mmsystem.h>
5
6
7  #define PORT 6868
8  #define SAMPLE_RATE 44100 // Adjust this to match your audio
   settings
9  #define BUFFER_SIZE 1024 // Adjust the buffer size as needed
10
11 using namespace std;
12 int main() {
13
14     // Initialize winsock WSADATA wsData;
15
16     WORD ver = MAKEWORD(2, 2);
17
18     int wsOk = WSASStartup(ver, &wsData);
19     if (wsOk != 0)
20     {
21         cout << "Can't initialize winsock! Quitting" << endl;
22         return 1;
23     }
24     // Create a socket
25     SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
26     if (sock == INVALID_SOCKET)
27     {
28         cout << "Can't create a socket! Quitting" << endl;
29         return 1;
30     }
31     // Bind the ip address and port to a sockaddr_in structure
       sockaddr_in hint;
32
33     hint.sin_family = AF_INET;
34     hint.sin_port = htons(PORT);
35     hint.sin_addr.s_addr = inet_addr("127.0.0.1");
36
37     // Connect to server
38
39     int connResult = connect(sock, (sockaddr *)&hint,
       sizeof(hint));
40
41     if (connResult == SOCKET_ERROR)
```

```

42 {
43     cout << "Can't connect to server, err #" << WSAGetLastError()
44         << endl; closesocket(sock);
45     WSACleanup();
46     return 1;
47 }
48 // Initialize audio capture
49 HWAVEIN hWaveIn;
50 WAVEFORMATEX waveFormat;
51 waveFormat.wFormatTag = WAVE_FORMAT_PCM;
52 waveFormat.nChannels = 2; // Stereo audio
53 waveFormat.nSamplesPerSec = SAMPLE_RATE;
54 waveFormat.wBitsPerSample = 16;
55 waveFormat.nBlockAlign = (waveFormat.nChannels *
56     waveFormat.wBitsPerSample) / 8; waveFormat.nAvgBytesPerSec =
57     waveFormat.nSamplesPerSec * waveFormat.nBlockAlign;
58 MMRESULT result = waveInOpen(&hWaveIn, WAVE_MAPPER,
59     &waveFormat, 0, 0, CALLBACK_NULL);
60
61 if (result)
62 {
63     cout << "Failed to open waveform input device." << endl;
64     closesocket(sock);
65     WSACleanup();
66     return 1;
67 }
68
69 short waveInBuffer[BUFFER_SIZE]; // Audio buffer
70 WAVEHDR waveHdr;
71
72 ZeroMemory(&waveHdr, sizeof(WAVEHDR));
73 waveHdr.lpData = (LPSTR)waveInBuffer;
74 waveHdr.dwBufferLength = BUFFER_SIZE * sizeof(short);
75 waveHdr.dwFlags = 0L;
76 waveHdr.dwLoops = 0L;
77 waveInPrepareHeader(hWaveIn, &waveHdr, sizeof(WAVEHDR));
78     waveInAddBuffer(hWaveIn, &waveHdr, sizeof(WAVEHDR));
79 waveInStart(hWaveIn);
80 bool running = true;
81
82 while (running) {
83     // Capture audio data and send to server
84     if (waveInUnprepareHeader(hWaveIn, &waveHdr, sizeof(WAVEHDR))
85         == WAVEERR_STILLPLAYING)
86     {
87         continue;
88     }
89 }

```



```

85 int sendResult = send(sock, (char *)waveInBuffer, BUFFER_SIZE
    * sizeof(short), 0);
86 if (sendResult == SOCKET_ERROR)
87 {
88     cout << "Failed to send data to server, err #" <<
        WSAGetLastError() << endl; running = false;
89     break;
90 }
91
92 waveInPrepareHeader(hWaveIn, &waveHdr, sizeof(WAVEHDR));
93 waveInAddBuffer(hWaveIn, &waveHdr, sizeof(WAVEHDR));
94 }
95 // Close the audio capture
96 waveInClose(hWaveIn);
97 // Close the socket
98 closesocket(sock);
99 // Cleanup winsock
100 WSACleanup();
101
102 return 0;
103 }

```

Code extraction 1: C++ code for TCP/IP client

5.2 Server Code

```

1  #include <iostream>
2  #include <WS2tcpip.h>
3  #include <Windows.h>
4  #include <mmsystem.h>
5
6  #define Message(msg) MessageBoxA(GetConsoleWindow(), msg,
    "Server Contact - Error", MB_OK)
7  #define PORT 6868
8  #define CLIENTS 2
9  #define SAMPLE_RATE 44100 // Adjust this to match your audio
    settings
10 #define BUFFER_SIZE 1024 // Adjust the buffer size as needed
11 using namespace std;
12
13 fd_set master;
14 MMRESULT result;
15
16 short ConnectedUsers() {
17     return master.fd_count - 1;
18 }
19
20 int main() {
21     HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
22     // Initialize winsock

```

```

23 WSADATA wsData;
24 WORD ver = MAKEWORD(2, 2);
25 int wsOk = WSStartup(ver, &wsData);
26 if (wsOk != 0) {
27     Message("Can't initialize winsock! Quitting");
28     cout << "Can't initialize winsock! Quitting" << endl
        << endl;
29     return 1;
30 }
31 // Create a socket
32 SOCKET listening = socket(AF_INET, SOCK_STREAM, 0);
33 if (listening == INVALID_SOCKET) {
34     Message("Can't create a socket! Quitting");
35     cout << "Can't create a socket! Quitting" << endl <<
        endl;
36     return 1;
37 }
38
39 // Bind the socket to an IP address and port
40 sockaddr_in hint;
41 hint.sin_family = AF_INET;
42 hint.sin_port = htons(PORT);
43 hint.sin_addr.s_addr = INADDR_ANY; // Use INADDR_ANY to
    bind to all available network interfaces
44 if (bind(listening, (sockaddr *)&hint, sizeof(hint)) ==
    SOCKET_ERROR) {
45     Message("Can't bind socket! Quitting");
46     cout << "Can't bind socket! Quitting" << endl << endl;
47     closesocket(listening);
48     WSACleanup();
49     return 1;
50 }
51 // Tell winsock the socket is for listening
52 listen(listening, SOMAXCONN);
53 FD_ZERO(&master);
54 FD_SET(listening, &master);
55
56 sockaddr_in client;
57 int clientSize = sizeof(client);
58
59 SetConsoleTextAttribute(hConsole, FOREGROUND_RED |
    FOREGROUND_GREEN | FOREGROUND_INTENSITY);
60 cout << "Waiting for incoming connection...\n" << endl;
61 SetConsoleTextAttribute(hConsole, FOREGROUND_GREEN |
    FOREGROUND_BLUE | FOREGROUND_RED);
62
63 short audioBuffer[BUFFER_SIZE]; // Audio buffer
64 // Initialize audio playback
65 WAVEFORMATEX playbackFormat;
66 playbackFormat.wFormatTag = WAVE_FORMAT_PCM;

```

```

67 playbackFormat.nChannels = 2; // Stereo audio
68 playbackFormat.nSamplesPerSec = SAMPLE_RATE;
69 playbackFormat.wBitsPerSample = 16;
70 playbackFormat.nBlockAlign = (playbackFormat.nChannels *
    playbackFormat.wBitsPerSample) / 8;
71 playbackFormat.nAvgBytesPerSec =
    playbackFormat.nSamplesPerSec *
    playbackFormat.nBlockAlign;
72
73 HWAVEOUT hWaveOut;
74 result = waveOutOpen(&hWaveOut, WAVE_MAPPER,
    &playbackFormat, 0, 0, CALLBACK_NULL);
75 if (result) {
76     char fault[256];
77     waveInGetErrorTextA(result, fault, 256);
78     Message(fault);
79     return 1;
80 }
81
82 bool running = true;
83 while (running) {
84     fd_set copy = master;
85     // See who's talking to us
86     int socketCount = select(0, &copy, nullptr, nullptr,
        nullptr);
87
88     // Loop through all the current connections /
        potential connections
89     for (int i = 0; i < socketCount; i++) {
90         // Makes things easy for us doing this assignment
91         SOCKET sock = copy.fd_array[i];
92         // Is it an inbound communication?
93         if (sock == listening) {
94             // Accept a new connection
95             SOCKET clientSocket = accept(listening,
                nullptr, nullptr);
96             // Add the new connection to the list of
                connected clients
97             FD_SET(clientSocket, &master);
98             // Send a welcome message to the connected
                client
99             string welcomeMsg = "Welcome to the Awesome
                Chat Server!";
100             send(clientSocket, welcomeMsg.c_str(),
                welcomeMsg.size() + 1, 0);
101             cout << "Connected users - " <<
                ConnectedUsers() << endl;
102         } else {
103             // It's an inbound audio data

```

```

104         int bytesIn = recv(sock, (char*)audioBuffer,
105                             BUFFER_SIZE * sizeof(short), 0);
106         if (bytesIn <= 0) {
107             // Drop the client
108             closesocket(sock);
109             FD_CLR(sock, &master);
110             cout << "Connected users - " <<
111                 ConnectedUsers() << endl;
112         } else {
113             // Process and play back the received
114             // audio data
115             if (bytesIn > 0) {
116                 // Write the audio data to the
117                 // playback buffer
118                 WAVEHDR playbackHdr;
119                 ZeroMemory(&playbackHdr,
120                             sizeof(WAVEHDR));
121                 playbackHdr.lpData =
122                     (LPSTR)audioBuffer;
123                 playbackHdr.dwBufferLength = bytesIn;
124                 playbackHdr.dwFlags = 0L;
125                 playbackHdr.dwLoops = 0L;
126                 waveOutPrepareHeader(hWaveOut,
127                                     &playbackHdr, sizeof(WAVEHDR));
128                 waveOutWrite(hWaveOut, &playbackHdr,
129                             sizeof(WAVEHDR));
130                 waveOutUnprepareHeader(hWaveOut,
131                                       &playbackHdr, sizeof(WAVEHDR));
132             }
133         }
134     }
135 }
136
137 // Close the audio playback waveOutClose(hWaveOut);
138 FD_CLR(listening, &master);
139 // Close the socket
140
141 closesocket(listening);
142 // Cleanup winsock WSACleanup();
143 return 0; }

```

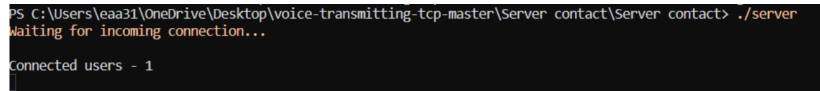
Code extraction 2: C++ code for TCP/IP server

6. Results and Analysis

The following points highlight the key outcomes of the project:

- The maximum number of clients that can concurrently join is 2.
- The sample rate of audio being transferred is 44100 Hz.

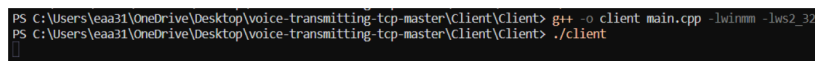
Server Side



```
PS C:\Users\ea31\OneDrive\Desktop\voice-transmitting-tcp-master\Server contact\Server contact> ./server
Waiting for incoming connection...
connected users - 1
```

Figure 2: Server side operation visualization.

Client Side



```
PS C:\Users\ea31\OneDrive\Desktop\voice-transmitting-tcp-master\Client\Client> g++ -o client main.cpp -lws2_32
PS C:\Users\ea31\OneDrive\Desktop\voice-transmitting-tcp-master\Client\Client> ./client
```

Figure 3: Client side interface.

Challenges Encountered: During the project, we faced issues with audio distortion and excessive noise in the transmitted audio. These issues were addressed by making adjustments in the audio processing pipeline.

7. Discussion

In the final output, the server plays back the audio sent by the client. The volume is a little low, but a lot of noise has been reduced.

The issues faced were addressed by making 2 changes:

1. The Portaudio library was initially used for audio capture and sending, but later on, the Windows Multimedia System API was utilized.
2. The buffer size was increased from 1024 bytes to 4096 bytes.

Currently, in the industry for communication, Voice over IP (VoIP) protocol is used which implements both TCP and UDP. Voice over Internet Protocol (VoIP) is a technology that enables voice communication and multimedia sessions over the Internet or other IP-based networks. It allows users to make voice calls, video calls, and send multimedia messages using Internet Protocol (IP) networks rather than traditional telephone networks.

8. Conclusion

The server-client audio transmission project aimed to establish a real-time audio communication system between a client and a server over a network. By utilizing Winsock for network communication and the Windows Multimedia System (MMSYSTEM) for

audio capture and playback, we developed a functional system that showcases the feasibility of live audio streaming over a local network.

Key Findings

1. **Audio Handling:** The MMSYSTEM library allowed for efficient capture of audio from the client and playback on the server side, ensuring synchronized audio transmission.
2. **Client-Server Interaction:** The system facilitated simultaneous audio streaming from multiple clients to the server, making it suitable for applications requiring multi-user audio conferencing.
3. **TCP/IP vs. UDP:** The use of TCP sockets (as demonstrated in the project) provides reliability and data integrity but may introduce some delay. UDP could be considered for applications that prioritize real-time performance over reliability.

Contribution to Computer Networks Field

1. **Real-Time Multimedia Communication:** The project showcases the feasibility of real-time multimedia communication over a local network, relevant to various fields such as video conferencing, online gaming, and audio chat applications.
2. **Scalability:** The demonstrated server-client architecture can be scaled to support more users, making it applicable to education, business, and social interaction.
3. **Potential for Further Research:** This project opens up avenues for research into optimizing network performance, reducing latency, and enhancing audio quality in real-time multimedia applications.

9. Future Work

The project demonstrates the feasibility of real-time multimedia communication over a local network, which has applications in fields such as video conferencing, online gaming, and audio chat applications.

The server-client architecture used in the project could be scaled to support a greater number of users, which would be beneficial for applications in education, business, and social interaction. Moreover, there is potential for further research into optimizing network performance, reducing latency, and enhancing audio quality in real-time multimedia applications.

10. References

1. Perkins, C., & Westerlund, M. (2006). *RTP: Audio and Video for the Internet*. Addison-Wesley.
2. Stevens, W. R. (1990). *UNIX Network Programming*. Prentice Hall.
3. Bolot, J., & Vega-Garcia, A. (1998). The case for FEC-based error control for packet audio in the Internet. *ACM Transactions on Networking*, 6(1), 1-14.

A. Appendices

A. Network Configuration Details

The following table summarizes the network configuration used for the project:

Parameter	Value
Protocol	TCP
IP Address	127.0.0.1
Port	6868
Buffer Size	1024 bytes
Sample Rate	44100 Hz

Table 1: Network Configuration Summary

B. Audio Data Handling

The process of handling audio data involves several steps detailed below:

- **Encoding:** The audio captured from the microphone is encoded using the PCM format.
- **Transmission:** The encoded audio data is transmitted over a TCP connection.
- **Reception:** The server receives the audio data and buffers it for playback.
- **Decoding:** The server decodes the PCM data and sends it to the speaker for output.

The buffer size and sample rate are crucial for maintaining audio quality:

- **Buffer Size:** Initially set to 1024 bytes, later increased to 4096 bytes to reduce noise.
- **Sample Rate:** Set to 44100 Hz to ensure high-quality audio playback.

Bibliography