

LOOP

A LAN-Based Real-Time Conferencing Application

Sudarshan Sudhakar (CS23B2007)

Deetya Ashish Mehta (CS23I1032)

November 3, 2025

Abstract

This document provides a comprehensive technical overview of **Loop**, a high-performance, real-time conferencing application designed specifically for Local Area Networks (LANs). It details the system architecture, dual-socket networking model, multi-threading strategy, and the implementation of its five core modules: *video conferencing*, *audio conferencing*, *screen sharing*, *text chat*, and *file sharing*. The document is based on the core logic found in the `server.py` and `client.py` source files.

Contents

1	Introduction	3
1.1	Core Modules	3
2	Core Architecture	4
2.1	Client-Server Model	4
2.2	Dual-Socket Protocol: TCP & UDP	4
2.2.1	TCP (The "Control Channel")	4
2.2.2	UDP (The "Media Channel")	5
2.2.3	The "Why": Reliability vs. Speed	5
2.3	Threading Model	5
2.3.1	The Problem: Blocking Sockets and the UI	5
2.3.2	Server-Side Threading	5
2.3.3	Client-Side Threading	6
2.4	Data Serialization	6
2.5	Session-Based Logging	6
2.6	Core Technologies and Libraries	6
3	Module 1: Multi-User Video Conferencing	8
3.1	Conceptual Overview: The Real-Time Video Pipeline	8
3.1.1	Why UDP for Webcam Video?	8
3.2	Client-Side Implementation	9
3.2.1	Video Transmission	9
3.2.2	Video Reception	9
3.3	Server-Side Implementation	10
3.3.1	Video Reception & Relay	10
4	Module 2: Multi-User Audio Conferencing	12
4.1	Conceptual Overview: Audio Mixing Architectures	12
4.1.1	The Loop Hybrid: Selective Mixing	12
4.2	Client-Side Implementation	12
4.2.1	Audio Transmission	12
4.2.2	Audio Reception	13
4.3	Server-Side Implementation	13
5	Module 3: Slide & Screen Sharing	15
5.1	Conceptual Overview: Clarity vs. Fluidity	15
5.1.1	The Role of Base64	15
5.2	Client-Side Implementation	15
5.3	Server-Side Implementation	17
6	Module 4: Group Text Chat	18
6.1	Conceptual Overview: Stateful Communication	18
6.2	Client-Side Implementation	18

6.2.1	Transmission	18
6.2.2	Reception	19
6.3	Server-Side Implementation	19
6.3.1	Reception & Enhancement	19
6.3.2	Broadcast	19
7	Module 5: File Sharing	21
7.1	Conceptual Overview: Brokered vs. Centralized	21
7.2	Implementation Steps	21
8	Application Gallery	26
8.1	User Interface Overview	26
8.2	Main Window	27
8.3	Additional Features	33
9	Setup Guide & Deployment	40
9.1	Overview	40
9.2	Step 1: Environment Setup	40
9.2.1	Install Python	40
9.2.2	Dependencies Installation	40
9.3	Step 2: Server Setup	41
9.3.1	Find Server's LAN IP Address	41
9.3.2	Run the Server	41
9.4	Step 3: Client Setup	41
9.4.1	Run the Client	41
9.5	Troubleshooting Guide	42
10	Project Summary & Achievements	43
10.1	Project Overview	43
10.2	Key Achievements	43
10.2.1	Technical Accomplishments	43
10.2.2	Feature Implementation Status	44
10.2.3	Architecture Highlights	44
10.3	Technical Insights & Best Practices	44
10.3.1	Protocol Selection Wisdom	44
10.4	Performance Characteristics	45
10.5	Conclusion	45

Chapter 1

Introduction

Loop is a high-performance, real-time conferencing application designed specifically for Local Area Networks (LANs). It operates **without any reliance on internet connectivity**, making it a secure and low-latency solution for internal team collaboration.

Information

The application is built on a custom client-server architecture using Python's socket programming library, successfully integrating five core communication modules into a single, cohesive platform.

1.1 Core Modules

The Loop platform encompasses five essential communication features:

1. **Multi-User Video Conferencing** — Real-time webcam streaming using UDP
2. **Multi-User Audio Conferencing** — Selective audio mixing with echo cancellation
3. **Slide & Screen Sharing** — High-fidelity screen broadcast over TCP
4. **Group Text Chat** — Reliable messaging with guaranteed delivery
5. **P2P Brokered File Sharing** — Server-mediated file transfer without storage

Technical Foundation This document details the technical architecture and implementation of each module, referencing the core logic in the `server.py` and `client.py` files. The system demonstrates advanced socket programming, protocol design, and real-time media processing techniques.

Chapter 2

Core Architecture

The foundation of Loop is a robust architecture designed to efficiently manage real-time media streams and reliable control messages simultaneously.

2.1 Client-Server Model

System Components The system is composed of two main components:

`server.py` — The Server

A central application that acts as the hub for all communication. It manages meeting sessions, authenticates users, relays video, mixes and distributes audio, and brokers all control messages.

`client.py` — The Client

The user-facing PyQt6 application. Each client maintains a connection to the server, captures and sends media, receives and renders media from other participants, and provides the UI for all interactions.

2.2 Dual-Socket Protocol: TCP & UDP

The system's performance is achieved by segregating traffic into two channels based on its requirements. This dual-protocol approach is the cornerstone of modern real-time communication applications.

2.2.1 TCP (The "Control Channel")

Information

Protocol: Transmission Control Protocol (TCP)

Purpose: Guarantees reliable, ordered delivery of messages where data integrity is critical.

Implementation: The server listens on a main port (e.g., **5001**). The client connects to this port.

Used For:

- Session Management (Creating/Joining meetings)
- Group Text Chat
- File Transfer (Offers, Requests, and Data Chunks)
- Host Controls (Mute, Unmute, Request Video, etc.)
- Real-time State Changes (Hand Raising, Emoji Reactions)
- Screen Sharing (Control and Frame Data)

2.2.2 UDP (The "Media Channel")

Key Achievement

Protocol: User Datagram Protocol (UDP)

Purpose: Prioritizes low latency and speed over guaranteed delivery, ideal for real-time video and audio where dropping a single frame is acceptable.

Implementation: The server listens on a separate port (e.g., 5002, calculated as `tcp_port + 1`).

Used For:

- Webcam Video Streams
- Microphone Audio Streams

2.2.3 The "Why": Reliability vs. Speed

Important

The choice between TCP and UDP is a fundamental trade-off in network design.

TCP — Guaranteed Delivery

A *connection-oriented* protocol that establishes a session and uses acknowledgments (ACKs) and retransmissions to ensure every packet arrives in order. Essential for text chat or file transfers where data integrity is critical. However, reliability comes at the cost of higher latency.

UDP — Speed First

A *connectionless* protocol that sends datagrams without acknowledgments. This "fire-and-forget" model is incredibly fast with very low latency, perfect for real-time video and audio where skipping a lost frame is preferable to freezing the stream.

2.3 Threading Model

2.3.1 The Problem: Blocking Sockets and the UI

Important

By default, socket operations like `socket.accept()` or `socket.recv()` are **blocking**. This means the program pauses at that line until a connection is made or data arrives. If run on the main thread of a GUI, the entire application would freeze. Multi-threading is the essential solution.

2.3.2 Server-Side Threading

Server Thread Architecture

- **Main Thread:** Starts the server, spawns listener threads, waits for shutdown
- **TCP Accept Thread:** Infinite loop blocking at `accept()`, spawns handler threads
- **Client Handler Threads:** One per client, manages TCP message loop
- **UDP Listener Thread:** Single thread efficiently processes all UDP packets

2.3.3 Client-Side Threading

Client Thread Architecture

- **Main (UI) Thread:** Runs PyQt6 event loop, never performs blocking calls
- **TCP Listener Thread:** Blocks at `recv()`, handles control messages
- **UDP Listener Thread:** Blocks at `recvfrom()`, handles media streams
- **Thread-Safe UI Updates:** Uses PyQt Signals for safe cross-thread communication

2.4 Data Serialization

Information

Sockets can only send and receive raw bytes. To send structured data (commands, IDs, messages), we must *serialize* them into a byte stream.

JSON (JavaScript Object Notation)

A human-readable text format. Excellent for debugging but less efficient due to text conversion and verbosity.

Msgpack

A binary serialization format. Much faster and more compact than JSON, ideal for high-performance applications.

Implementation: The `_serialize_message` and `_deserialize_message` functions abstract this process, allowing easy protocol switching.

2.5 Session-Based Logging

Both client and server implement robust logging systems creating session-based log files in `logs/client` and `logs/server` directories. This is invaluable for debugging and tracking application behavior over time.

2.6 Core Technologies and Libraries

Technology Stack The project leverages powerful, open-source Python libraries documented in `requirements.txt`:

PyQt6 (6.5.0) Foundation of the client-side GUI. Provides all windows, widgets, and the critical signal-slot mechanism for thread-safe UI updates.

qtawesome (1.2.3) Integrates iconic fonts (FontAwesome) for a modern UI with professional icons instead of text labels.

OpenCV (4.8.0.76) Primary video engine: webcam access, frame processing, image resizing, and JPEG compression/decompression.

PyAudio (0.2.11) Core audio module: microphone capture and speaker playback via PortAudio bindings.

NumPy (1.24.3) High-performance computing library. Underpins all media processing: audio mixing, volume normalization, and image manipulation.

Pillow (10.0.0) Image manipulation library. Converts formats between NumPy/OpenCV and PyQt6 display formats.

mss (9.0.1) High-performance screen capture engine for the screen sharing module.

Chapter 3

Module 1: Multi-User Video Conferencing

Key Achievement

This module is responsible for capturing and distributing all webcam video streams using the **UDP** protocol for optimal low-latency performance.

3.1 Conceptual Overview: The Real-Time Video Pipeline

Delivering real-time video is a multi-step process:

1. **Capture** — Access webcam to capture raw image frames
2. **Compress** — Encode frame (e.g., JPEG/H.264) to reduce size
3. **Transmit** — Wrap in custom UDP packet and send to server
4. **Receive** — Server's UDP listener identifies video packet
5. **Relay** — Server forwards packet to all other clients
6. **Decompress** — Receiving client decodes binary blob to image
7. **Render** — Display final image in UI

Important

This entire pipeline must be completed 20-30 times per second (FPS) for smooth motion. UDP's low latency makes this possible.

3.1.1 Why UDP for Webcam Video?

Information

In video conferencing, it's far more important for video to be *on time* than *perfect*. If a packet is lost, the client simply discards that frame and waits for the next one. This prevents the entire stream from freezing while TCP attempts retransmission.

This is known as managing **latency** and **jitter** (variation in packet arrival time).

3.2 Client-Side Implementation

3.2.1 Video Transmission

The client captures a frame, compresses it to JPEG, and calls `send_udp_stream('V', video_data)`.

```

1 def send_udp_stream(self, stream_type, data):
2     try:
3         client_id_bytes = self.client_id.encode('utf-8')
4         client_id_len = len(client_id_bytes)
5
6         packet_size = 1 + 2 + client_id_len + len(data)
7         if packet_size > 65507:
8             return
9
10        packet = self.udp_send_buffer
11        packet[0] = ord(stream_type)
12        struct.pack_into('!H', packet, 1, client_id_len)
13        packet[3:3+client_id_len] = client_id_bytes
14        packet[3+client_id_len:packet_size] = data
15
16        self.udp_socket.sendto(bytes(packet[:packet_size]),
17                               self.server_address)
18    except Exception as e:
19        pass

```

Listing 3.1: Client-side UDP packet construction

3.2.2 Video Reception

The `_receive_udp_streams` thread listens for UDP packets. When a '`'V'`' packet arrives, it parses the header, extracts the sender's `client_id`, and emits the `video_received` signal. The main UI thread decompresses the frame and updates the appropriate video widget.

Code Snippet: Client-Side Video Reception

```

1 def _receive_udp_streams(self):
2     """UDP receiver thread with improved video/audio handling"""
3     self.udp_socket.settimeout(0.05)
4
5     while self.running:
6         try:
7             data, addr = self.udp_socket.recvfrom(65535)
8
9             if len(data) < 3:
10                 continue
11
12             stream_type = chr(data[0])
13
14             if stream_type == 'V':
15                 # Video packet
16                 try:
17                     client_id_len = struct.unpack('!H', data[1:3])[0]
18                     if len(data) < 3 + client_id_len:
19                         continue
20
21                     client_id = data[3:3+client_id_len].decode('utf-8')
22                     video_data = data[3+client_id_len:]

```

```

23             if len(video_data) > 0 and client_id:
24                 self.signals.video_received.emit(client_id,
25                     video_data)
26             except (struct.error, UnicodeDecodeError) as e:
27                 print(f"Video packet decode error: {e}")
28                 continue
29
30         except socket.timeout:
31             continue
32         # ... other error handling ...

```

Listing 3.2: Client's `_receive_udp_streams` thread (video handling)

3.3 Server-Side Implementation

3.3.1 Video Reception & Relay

Server Video Relay Logic The server acts as a **Video Relay**. It does **not** process or re-encode video. It simply:

1. Receives UDP packet via `handle_udp_packet`
2. Identifies packet type as '`V`'
3. Extracts `client_id` and `stream_data`
4. Forwards to all other meeting participants via `broadcast_udp_to_meeting`

This architecture is highly efficient, minimizing server CPU load.

Code Snippet: Server-Side Video Relay

```

1 def handle_udp_packet(self, data: bytes, addr: tuple):
2     # ... (header parsing, client_id extraction) ...
3     try:
4         # ... (client_id, stream_data extraction) ...
5
6         # ... (Update UDP address logic) ...
7
8         meeting_code = self.client_to_meeting.get(client_id)
9         if not meeting_code:
10             return
11
12         # Handle different stream types
13         if stream_type == 'V' and len(stream_data) > 0:
14             self.stats['video_packets'] += 1
15             self.broadcast_udp_to_meeting(
16                 meeting_code, 'V', client_id, stream_data, exclude_id=
17                 client_id
18             )
19         elif stream_type == 'A' and len(stream_data) > 0:
20             # ... (audio handling) ...
21             pass
22
23     except (struct.error, UnicodeDecodeError, ValueError) as e:
24         print(f"UDP packet decode error: {e}")
25         return

```

Listing 3.3: Server's `handle_udp_packet` (video relay logic)

Chapter 4

Module 2: Multi-User Audio Conferencing

Key Achievement

This module handles capturing, mixing, and distributing audio in real-time. This is one of the most performance-critical parts of the application, using **UDP** for low-latency delivery.

4.1 Conceptual Overview: Audio Mixing Architectures

There are two common approaches to multi-user audio:

Multipoint Control Unit (MCU)

”Full mixing” server. Receives all audio, decodes, mixes into single composite stream, re-encodes, and sends to everyone. Simple for clients but has fatal flaw: **echo** — users hear their own voice played back.

Selective Forwarding Unit (SFU)

”Smarter” server. Forwards each participant’s stream to all others. Clients receive multiple streams to mix locally. More complex but avoids echo.

4.1.1 The Loop Hybrid: Selective Mixing

Key Achievement

This application implements a superior hybrid model: an **SFU that selectively mixes on the server**.

Best of both worlds:

- No Echo (like SFU) — Each participant receives a unique mix excluding their own audio
- Client Simplicity (like MCU) — Clients receive only one pre-mixed stream to play

This reduces client-side CPU load while solving the echo problem, all using low-latency UDP.

4.2 Client-Side Implementation

4.2.1 Audio Transmission

The client captures audio chunks, encodes as raw **int16** PCM data, and calls **send_udp_stream('A', audio_data)**. Packet structure is identical to video but with '**A**' as stream type.

4.2.2 Audio Reception

The `_receive_udp_streams` thread identifies '**A**' packets and emits the `audio_received` signal with the mixed stream from the server. The UI thread queues it for playback.

Code Snippet: Client-Side Audio Reception

```

1 def _receive_udp_streams(self):
2     # ...
3     while self.running:
4         try:
5             data, addr = self.udp_socket.recvfrom(65535)
6             # ...
7             stream_type = chr(data[0])
8
9             if stream_type == 'A' and data[1:3] == b'\x00\x00':
10                 # Audio packet
11                 audio_data = data[3:]
12                 if len(audio_data) > 0:
13                     self.signals.audio_received.emit(None, audio_data)
14
15             elif stream_type == 'V':
16                 # ... (video handling) ...
17                 pass
18             # ... (error handling) ...

```

Listing 4.1: Client's `_receive_udp_streams` thread (audio handling)

4.3 Server-Side Implementation

Sophisticated Audio Processing The server implements the most complex module logic:

1. **Ingestion** — `handle_udp_packet` receives audio, identifies sender
2. **Buffering** — Decode to NumPy array, store in `meeting.audio_buffers[sender_id]`
3. **Selective Mixing** — For each recipient, create custom mix excluding their own audio
4. **Broadcasting** — Re-encode and send personalized mix via UDP to each recipient

Code Snippet: Server-Side Audio Ingestion The server's UDP handler acts as the entry point. It parses the '**A**' packet and passes the raw audio data to the mixing function.

```

1 def handle_udp_packet(self, data: bytes, addr: tuple):
2     # ... (header parsing, client_id extraction) ...
3     try:
4         # ... (client_id, stream_data extraction) ...
5
6         # ... (Update UDP address logic) ...
7
8         meeting_code = self.client_to_meeting.get(client_id)
9         if not meeting_code:
10             return
11
12         # Handle different stream types
13         if stream_type == 'V' and len(stream_data) > 0:
14             # ... (video handling) ...
15             pass

```

```

16     elif stream_type == 'A' and len(stream_data) > 0:
17         self.stats['audio_packets'] += 1
18         self.mix_and_broadcast_audio(meeting_code, client_id,
19                                       stream_data)
20     elif stream_type == 'I':
21         # ... (init packet) ...
22         pass
23
24 except (struct.error, UnicodeDecodeError, ValueError) as e:
25     print(f"UDP packet decode error: {e}")
26     return

```

Listing 4.2: Server's **handle_udp_packet** (audio ingestion)

```

1 # Mix audio for each recipient
2 for recipient_id, addr in recipient_addrs.items():
3     sources = [sid for sid in valid_buffers
4                if sid != recipient_id]
5
6     if not sources:
7         continue
8
9     try:
10        self.audio_mix_buffer[:target_len] = 0
11
12        for source_id in sources:
13            source_audio = valid_buffers[source_id]
14            add_len = min(target_len, len(source_audio))
15            if add_len > 0:
16                self.audio_mix_buffer[:add_len] += source_audio[:add_len]
17
18            if len(sources) > 1:
19                self.audio_mix_buffer[:target_len] /= len(sources)
20
21        mixed = np.clip(self.audio_mix_buffer[:target_len],
22                        -32768, 32767).astype(np.int16)
23        packet = b'A\x00\x00' + mixed.tobytes()
24        self.udp_socket.sendto(packet, addr)
25
26    except Exception as e:
27        continue

```

Listing 4.3: Server-Side Selective Audio Mixing

Chapter 5

Module 3: Slide & Screen Sharing

Information

This module allows one user at a time to broadcast their screen to all participants. To ensure clarity and data integrity for slides and text, this module operates entirely over the reliable **TCP** protocol.

5.1 Conceptual Overview: Clarity vs. Fluidity

Important

Unlike webcam feeds where lost frames are acceptable, screen shares contain static, high-detail content like text, code, or slides.

Using UDP: Lost packets could permanently garble text or create "smeared" areas until next refresh.

Using TCP: Guarantees every packet arrives in order, ensuring perfect clarity and integrity. This is a conscious trade-off: we sacrifice UDP's low latency to gain TCP's 100% reliability. Acceptable because screen content changes less frequently than webcam feeds.

5.1.1 The Role of Base64

Base64 Encoding TCP messages use JSON (text format), but compressed images are raw binary data. Placing binary in JSON can cause corruption.

Solution: Base64 encoding converts binary to "safe" text characters that can be embedded in JSON. Receiving client decodes back to original binary.

5.2 Client-Side Implementation

1. **Requesting** — Call `request_screen_share()`, sends TCP message
2. **Streaming** — Once approved, UI sets `is_presenting = True`, captures frames, Base64 encodes, calls `send_screen_frame_tcp()`
3. **Receiving** — TCP thread handles '`screen_frame`' messages, decodes Base64, emits `video_received` signal
4. **Stopping** — Call `stop_screen_share_request()`, sends TCP message

Code Snippet: Client-Side Screen Frame Transmission

```

1 def send_screen_frame_tcp(self, frame_data):
2     """Send screen frame via TCP for reliability and clarity"""
3     if not self.connected or not hasattr(self, 'is_presenting') or not
4         self.is_presenting:
5         return
6
7     try:
8         self.send_tcp_message({
9             'type': 'screen_frame',
10            'frame_data': frame_data, # This is the Base64-encoded
11                string
12               'presenter_id': self.client_id
13           })
14     except Exception as e:
15         self.logger.error(f"Error sending screen frame via TCP: {e}")
16         print(f"Error sending screen frame via TCP: {e}")
17         raise

```

Listing 5.1: Client's `send_screen_frame_tcp` method

Code Snippet: Client-Side Screen Frame Reception

```

1 def _receive_tcp_messages(self):
2     """TCP receiver thread for better performance"""
3     while self.running:
4         try:
5             # ... (read message length and data) ...
6
7             # Decode message
8             message = self._deserialize_message(data)
9
10            # ... (other message types) ...
11            if message.get('type') == 'screen_frame':
12                # Handle incoming screen frame
13                self._handle_screen_frame(message)
14            else:
15                # Emit other messages to UI thread
16                self.signals.message_received.emit(message)
17
18        except (ConnectionResetError, OSError) as e:
19            # ... (error handling) ...
20            break
21        # ... (error handling) ...
22
23    def _handle_screen_frame(self, message):
24        try:
25            presenter_id = message.get('presenter_id')
26            frame_data = message.get('frame_data')
27
28            if presenter_id != self.current_presenter:
29                return # Ignore frame from non-presenter
30
31            # Decode base64 frame data
32            if isinstance(frame_data, str):
33                frame_bytes = base64.b64decode(frame_data)
34            else:
35                frame_bytes = frame_data

```

```

37     # Emit to UI for display (re-using video signal)
38     self.signals.video_received.emit(presenter_id, frame_bytes)
39
40 except Exception as e:
41     print(f"Error handling screen frame: {e}")

```

Listing 5.2: Client's `_receive_tcp_messages` (screen frame handling)

5.3 Server-Side Implementation

Server Screen Share Control

1. Server receives `request_screen_share` message
2. If no presenter, sets `meeting.current_presenter = client_id`
3. Broadcasts '`screen_share_started`' TCP message
4. If busy, sends '`screen_share_denied`' to requester
5. Server relays '`screen_frame`' messages from valid presenter to all others

Code Snippet: Server-Side Screen Frame Relay

```

1 def handle_host_command(self, client_id: str, meeting_code: str,
2                         msg_type: str, message: dict):
3     # ... (other host commands: mute, lock, etc.) ...
4
5     elif msg_type == 'request_screen_share':
6         # ... (logic for starting/denying) ...
7         pass
8
9     elif msg_type == 'stop_screen_share':
10        # ... (logic for stopping) ...
11        pass
12
13    elif msg_type == 'screen_frame':
14        # Handle screen frame from presenter
15        meeting = self.meetings.get(meeting_code)
16        if meeting and meeting.current_presenter == client_id:
17            frame_data = message.get('frame_data')
18            if frame_data:
19                # Broadcast screen frame to all other participants via
19                # TCP
20                self.broadcast_to_meeting(
21                    meeting_code,
22                    {
23                        'type': 'screen_frame',
24                        'presenter_id': client_id,
25                        'frame_data': frame_data
26                    },
27                    exclude_id=client_id
28                )
29            else:
30                self.logger.warning(f"Received screen frame from {client_id}
30                                but they are not the current presenter")

```

Listing 5.3: Server's `handle_host_command` (screen frame relay)

Chapter 6

Module 4: Group Text Chat

Information

This module provides a reliable, real-time text chat for all participants using the **TCP** protocol to guarantee message integrity and order.

6.1 Conceptual Overview: Stateful Communication

Important

Text chat is a classic example of why TCP's stateful, connection-oriented nature is required.

Example Message: "Let's meet at 10. No, 11."

With UDP: Packets could arrive out of order ("No, 11. Let's meet at 10.") or be lost ("Let's meet at 10."), causing total confusion.

With TCP: Every message arrives in exact send order with no data loss. The small latency added is a worthy trade-off for 100% message integrity.

6.2 Client-Side Implementation

6.2.1 Transmission

When a user sends a message, `send_chat_message(message_text)` wraps the text in JSON and sends via `send_tcp_message`.

Code Snippet: Client-Side Chat Transmission

```
1 def send_chat_message(self, message):
2     """Send chat message"""
3     self.logger.info(f'Sending chat message: {message[:50]}...')
4     self.send_tcp_message({'type': 'chat', 'message': message})
5
6 def send_tcp_message(self, message):
7     """Send TCP message with optimized serialization"""
8     if not self.connected:
9         return
10
11    try:
12        data = self._serialize_message(message)
13        length = struct.pack('!I', len(data))
14        self.tcp_socket.sendall(length + data)
```

```

15     except Exception as e:
16         print(f"TCP send error: {e}")

```

Listing 6.1: Client's `send_chat_message` method

6.2.2 Reception

The `_receive_tcp_messages` thread receives broadcast messages from server, emits `message_received` signal. UI formats with sender's username and appends to chat history.

Code Snippet: Client-Side Chat Reception

```

1 def _receive_tcp_messages(self):
2     while self.running:
3         try:
4             # ... (read message length and data) ...
5
6             message = self._deserialize_message(data)
7
8             if message.get('type') == 'file_chunk':
9                 # ... (file handling) ...
10            # ... (other message types) ...
11            else:
12                # Emit other messages to UI thread
13                self.signals.message_received.emit(message)
14
15        except (ConnectionResetError, OSError) as e:
16            # ... (error handling) ...
17            break
18            # ... (error handling) ...

```

Listing 6.2: Client's `_receive_tcp_messages` (chat handling)

6.3 Server-Side Implementation

6.3.1 Reception & Enhancement

`handle_tcp_message` identifies `msg_type == 'chat'`, enhances message with sender details (username, client_id).

6.3.2 Broadcast

Calls `broadcast_to_meeting`, sending enriched JSON over TCP to all participants.

Code Snippet: Server-Side Chat Handling Broadcast

```

1 def handle_tcp_message(self, client_id: str, message: dict):
2     """Handle TCP messages efficiently"""
3     msg_type = message.get('type')
4
5     with self.clients_lock:
6         client_info = self.clients.get(client_id)
7         if not client_info:
8             return
9
10    meeting_code = client_info.meeting_code
11    is_host = client_info.is_host

```

```
12     if msg_type == 'chat':
13         self.broadcast_to_meeting(
14             meeting_code,
15             {
16                 'type': 'chat',
17                 'client_id': client_id,
18                 'username': client_info.username,
19                 'message': message['message']
20             },
21             exclude_id=client_id # Sender already has message
22         )
23
24     elif msg_type == 'video_state':
25         # ... (other handlers) ...
26         pass
27     # ... (other message types) ...
```

Listing 6.3: Server's `handle_tcp_message` (chat handling)

```
1  {
2      "type": "chat",
3      "client_id": "user_192.168.1.10_51234",
4      "username": "Alice",
5      "message": "Hello everyone!"
6 }
```

Listing 6.4: JSON structure for chat message

Chapter 7

Module 5: File Sharing

Key Achievement

This module allows users to share files using a **Brokered Peer-to-Peer (P2P)** model, where the server relays the transfer but never stores files to disk. All communication uses **TCP** for reliability.

7.1 Conceptual Overview: Brokered vs. Centralized

Centralized Model

Client A uploads to Server, Server saves to disk, Server notifies Client B, Client B downloads.
Simple but heavy server burden.

Brokered P2P Model

Client A offers file, Server relays offer to Client B, Client B requests, Server tells Client A to start, Client A sends chunks through Server to Client B. More efficient.

7.2 Implementation Steps

1. **File Offer** — Client A selects file, sends metadata via TCP
2. **File Request** — Client B receives offer, requests download
3. **P2P Transfer** — Client A sends 32KB chunks through server to Client B
4. **Completion** — Client A sends end message, Client B closes file

Code Snippets: The File Transfer Lifecycle The file transfer process involves a handshake and data relay, all over TCP.

1. Client A: Offers File

```
1 def send_file_offer(self, file_path):  
2     try:  
3         file_path = Path(file_path)  
4         if not file_path.exists():  
5             return None  
6  
7         file_id = f"{self.client_id}_{int(time.time() * 1000)}"  
8         filename = file_path.name
```

```

9     filesize = file_path.stat().st_size
10
11    # Store file for later requests
12    with self.file_lock:
13        self.pending_files[file_id] = str(file_path)
14
15    # Send offer
16    self.send_tcp_message({
17        'type': 'file_offer',
18        'file_id': file_id,
19        'filename': filename,
20        'filesize': filesize
21    })
22
23    return file_id
24
25 except Exception as e:
26     return None

```

Listing 7.1: Client's **send_file_offer** method

2. Server: Relays Offer

```

1 def handle_tcp_message(self, client_id: str, message: dict):
2     # ... (get client_info, meeting_code) ...
3
4     if msg_type == 'file_offer':
5         self.logger.info(f'File offer from {client_info.username}...')
6         self.broadcast_to_meeting(meeting_code, {
7             'type': 'file_offer',
8             'sender_id': client_id,
9             'username': client_info.username,
10            **message
11        }, exclude_id=client_id)
12    # ...

```

Listing 7.2: Server's **handle_tcp_message** (file offer relay)

3. Client B: Requests File

```

1 def request_file(self, sender_id, file_id, filename, filesize, save_path
2     =None):
3     try:
4         # ... (determine save path) ...
5
6         # Open file for writing
7         file_handle = open(filepath, 'wb')
8
9         with self.file_lock:
10            self.receiving_files[file_id] = {
11                'file': file_handle,
12                'filename': filename,
13                'bytes_received': 0,
14                'total_size': filesize,
15                'save_path': str(filepath)
16            }

```

```

17     # Send request to server
18     self.send_tcp_message({
19         'type': 'file_request',
20         'sender_id': sender_id,
21         'file_id': file_id
22     })
23
24 except Exception as e:
25     print(f"Error requesting file: {e}")

```

Listing 7.3: Client's `request_file` method**4. Server: Relays Request to Client A**

```

1 def handle_tcp_message(self, client_id: str, message: dict):
2     # ... (get client_info, meeting_code) ...
3
4     elif msg_type == 'file_request':
5         sender_id = message.get('sender_id')
6         file_id = message.get('file_id')
7         self.logger.info(f'File request from {client_info.username} to {sender_id}...')
8         self.send_to_client(sender_id, {
9             'type': 'file_request',
10            'downloader_id': client_id,
11            'username': client_info.username,
12            'file_id': file_id
13        })
14    # ...

```

Listing 7.4: Server's `handle_tcp_message` (file request relay)**5. Client A: Receives Request and Starts Sending Chunks**

```

1 def _handle_file_request(self, message):
2     try:
3         file_id = message.get('file_id')
4         recipient_id = message.get('downloader_id')
5
6         with self.file_lock:
7             if file_id in self.pending_files:
8                 file_path = self.pending_files[file_id]
9
10                # Start sending file in background thread
11                threading.Thread(
12                    target=self._send_file_worker,
13                    args=(recipient_id, file_id, file_path),
14                    daemon=True
15                ).start()
16     except Exception as e:
17         print(f"Error handling file request: {e}")
18
19 def _send_file_worker(self, recipient_id, file_id, file_path):
20     try:
21         chunk_size = 32768 # 32KB chunks
22         with open(file_path, 'rb') as f:
23             while self.running:

```

```

24     chunk = f.read(chunk_size)
25     if not chunk:
26         break
27
28     # Encode chunk as base64 for JSON compatibility
29     chunk_encoded = base64.b64encode(chunk).decode('utf-8')
30
31     self.send_tcp_message({
32         'type': 'file_chunk',
33         'recipient_id': recipient_id,
34         'file_id': file_id,
35         'data': chunk_encoded
36     })
37     time.sleep(0.01) # Avoid overwhelming network
38
39     # Send completion message
40     self.send_file_end(recipient_id, file_id)
41
42 except Exception as e:
43     print(f"Error sending file: {e}")

```

Listing 7.5: Client's `handle_file_request` and `send_file_worker`

6. Server: Relays Chunks and Completion

```

1 def handle_tcp_message(self, client_id: str, message: dict):
2     # ... (get client_info, meeting_code) ...
3
4     elif msg_type in ['file_chunk', 'file_end']:
5         recipient_id = message.get('recipient_id')
6         if msg_type == 'file_chunk':
7             self.logger.debug(f'File chunk from {client_info.username} to {recipient_id}')
8         else:
9             self.logger.info(f'File transfer complete from {client_info.username} to {recipient_id}')
10
11     # Forward message directly to recipient
12     self.send_to_client(recipient_id, {'sender_id': client_id, **message})
13     # ...

```

Listing 7.6: Server's `handle_tcp_message` (chunk/end relay)

7. Client B: Receives Chunks and Completion

```

1 def _handle_file_chunk(self, message):
2     try:
3         file_id = message.get('file_id')
4         chunk_data = message.get('data')
5
6         if isinstance(chunk_data, str):
7             chunk_data = base64.b64decode(chunk_data)
8
9         with self.file_lock:
10             if file_id in self.receiving_files:
11                 file_info = self.receiving_files[file_id]

```

```
12         file_info['file'].write(chunk_data)
13         file_info['bytes_received'] += len(chunk_data)
14
15         # ... (notify UI of progress) ...
16
17     except Exception as e:
18         print(f"Error handling file chunk: {e}")
19
20 def _handle_file_end(self, message):
21     try:
22         file_id = message.get('file_id')
23
24         with self.file_lock:
25             if file_id in self.receiving_files:
26                 file_info = self.receiving_files[file_id]
27                 file_info['file'].close()
28
29                 # ... (notify UI of completion) ...
30
31                 del self.receiving_files[file_id]
32
33     except Exception as e:
34         print(f"Error handling file end: {e}")
```

Listing 7.7: Client's `_handle_file_chunk` and `_handle_file_end`

Key Achievement

Implementation Note: This P2P-brokered model is scalable and efficient, with minimal server CPU and zero disk usage.

Chapter 8

Application Gallery

Information

A comprehensive preview of the Loop user interface components, showcasing various layouts, features, and component states across different interaction scenarios.

8.1 User Interface Overview

The Loop application features a modern, intuitive interface designed for seamless real-time collaboration. The gallery showcases:

- **Authentication** — Login and meeting creation interfaces
- **Video Conferencing** — Multi-user video grid layouts
- **Screen Sharing** — Full-screen presentation mode
- **File Sharing** — Transfer progress and notifications
- **Communication** — Text chat and participant management
- **Admin Controls** — Host-exclusive management features

8.2 Main Window

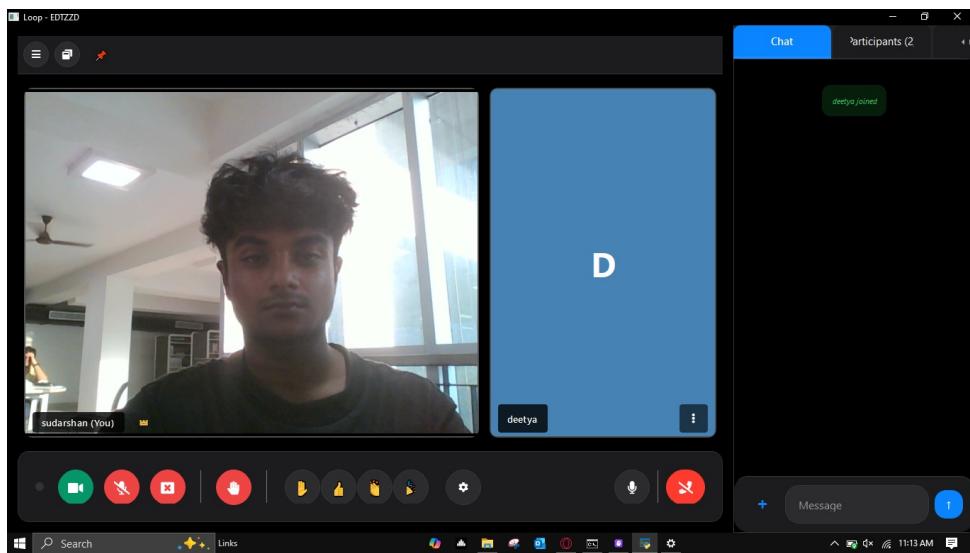


Figure 8.1: Webcam video capture

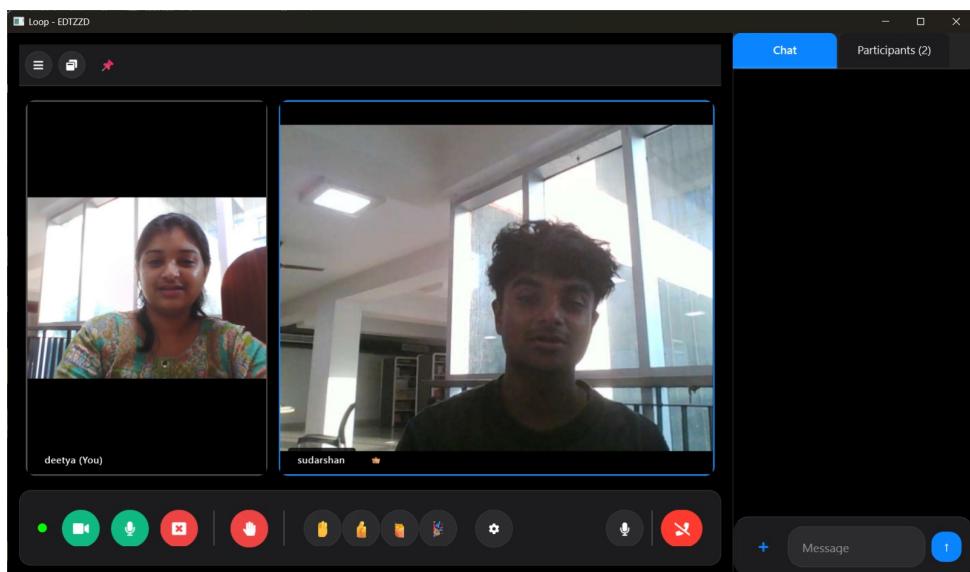


Figure 8.2: Receive multiple video streams from the server

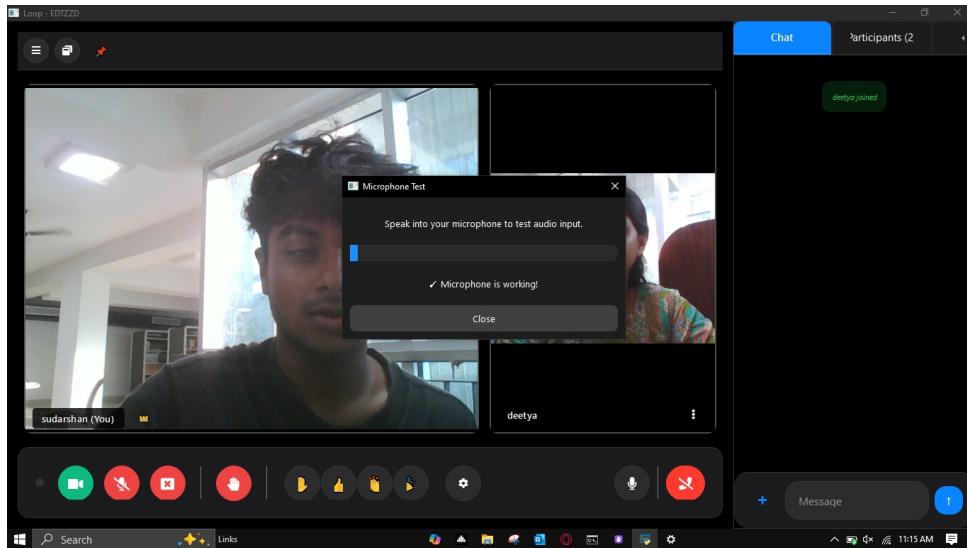


Figure 8.3: Capture audio input from microphone.

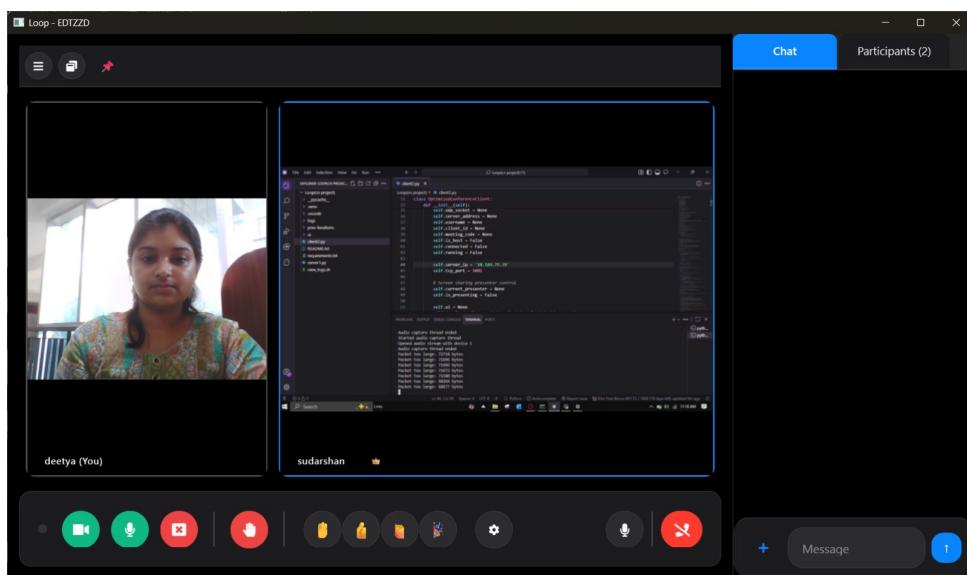


Figure 8.4: Screen Sharing

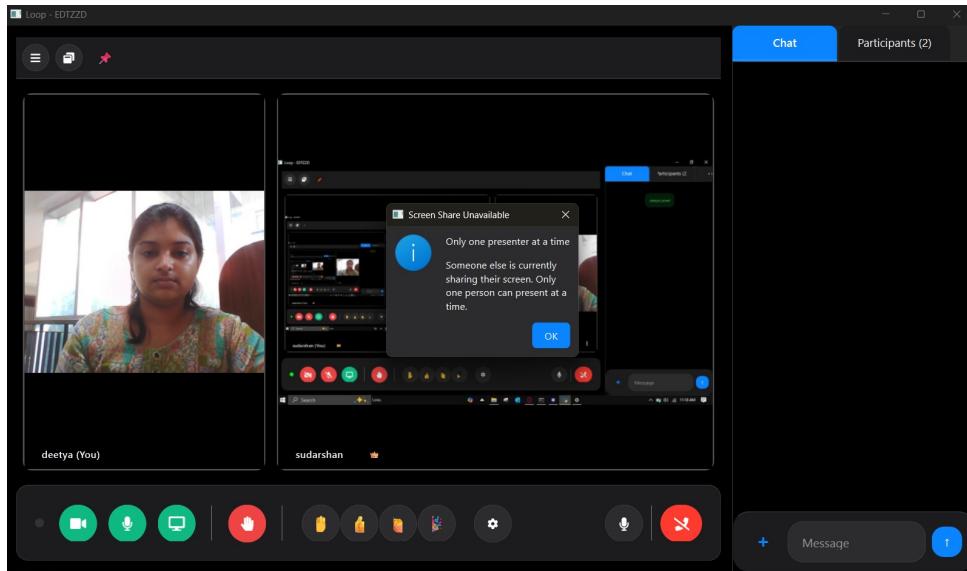


Figure 8.5: One user at a time can act as the "presenter."

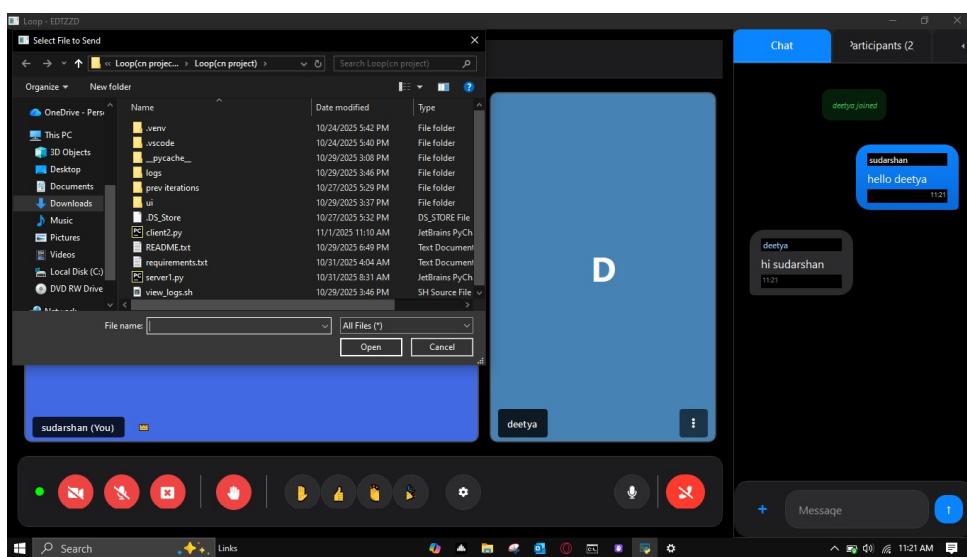


Figure 8.6: File Sharing: Sending and Choosing file

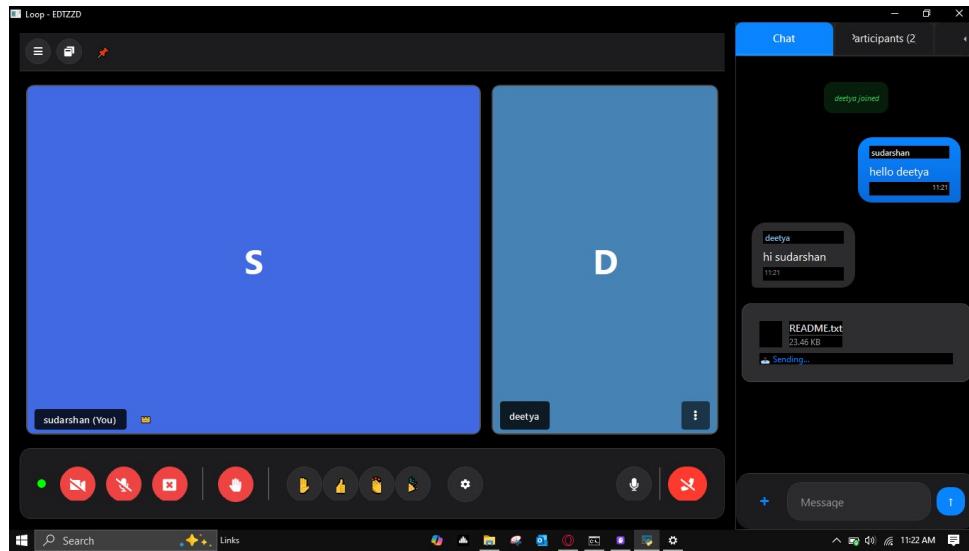


Figure 8.7: File Sharing: Shared File progress bar in message panel

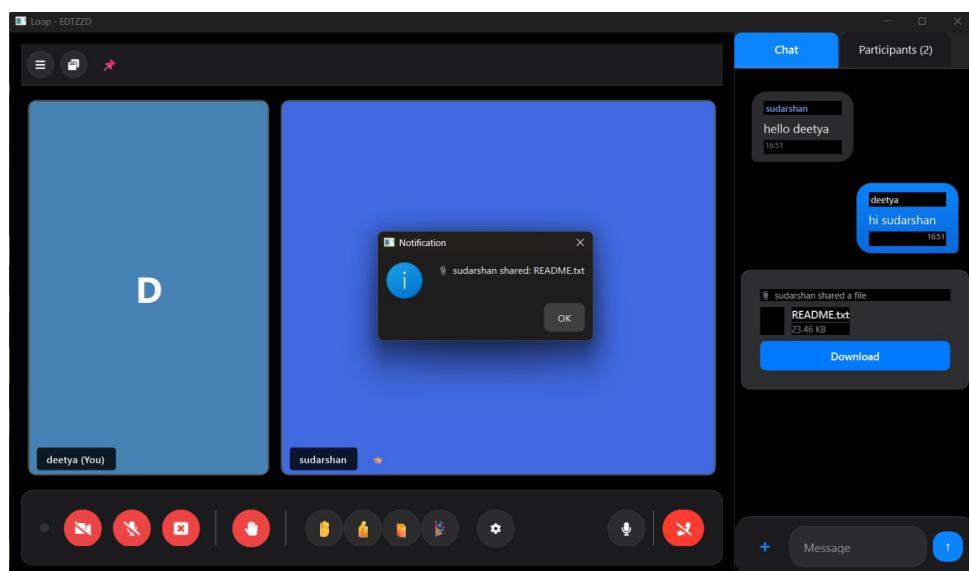


Figure 8.8: File Sharing: Receiver Side File Shared Pop up

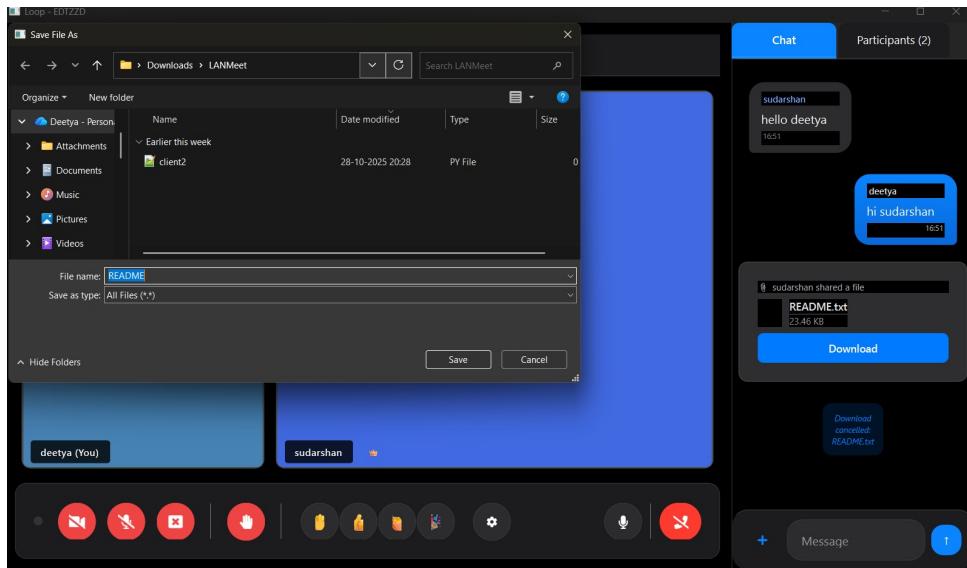


Figure 8.9: File Sharing: Receiver directory to store shared file

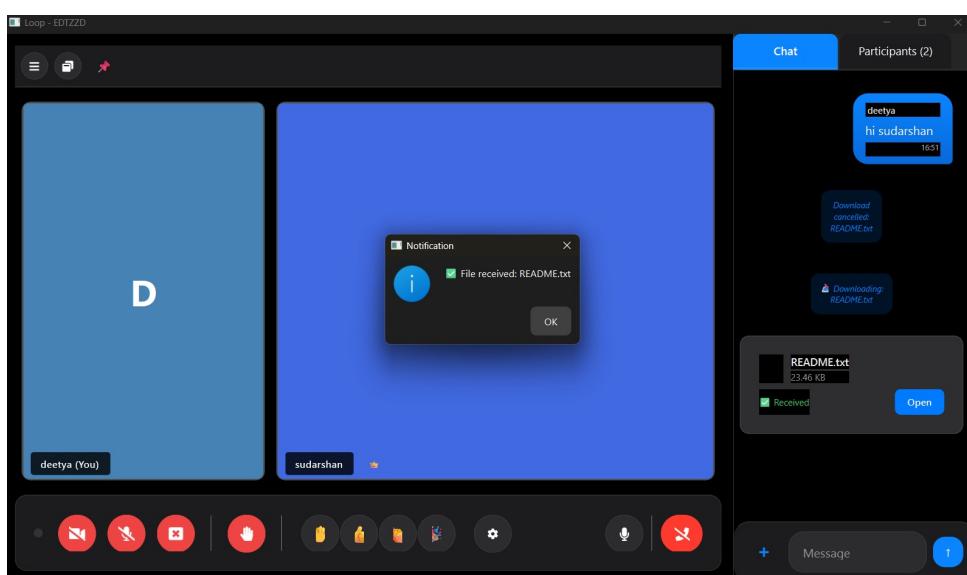


Figure 8.10: File Sharing: File received successfully message



Figure 8.11: Messaging Window/Sidebar

8.3 Additional Features

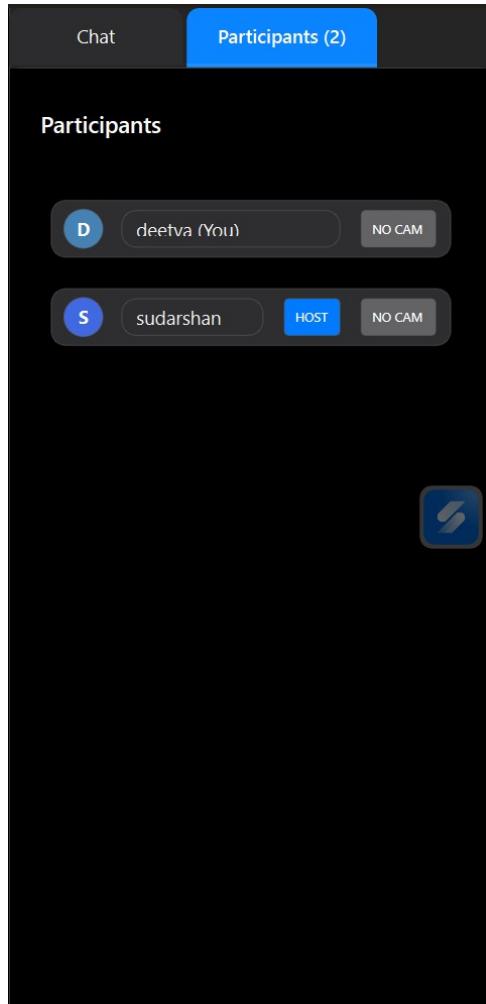


Figure 8.12: Participants View

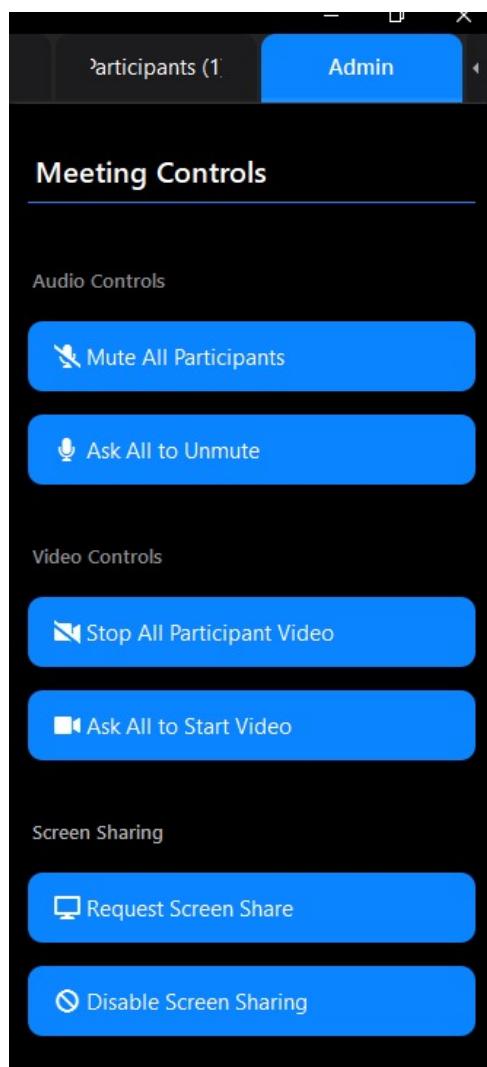


Figure 8.13: Exclusive Admin Controls

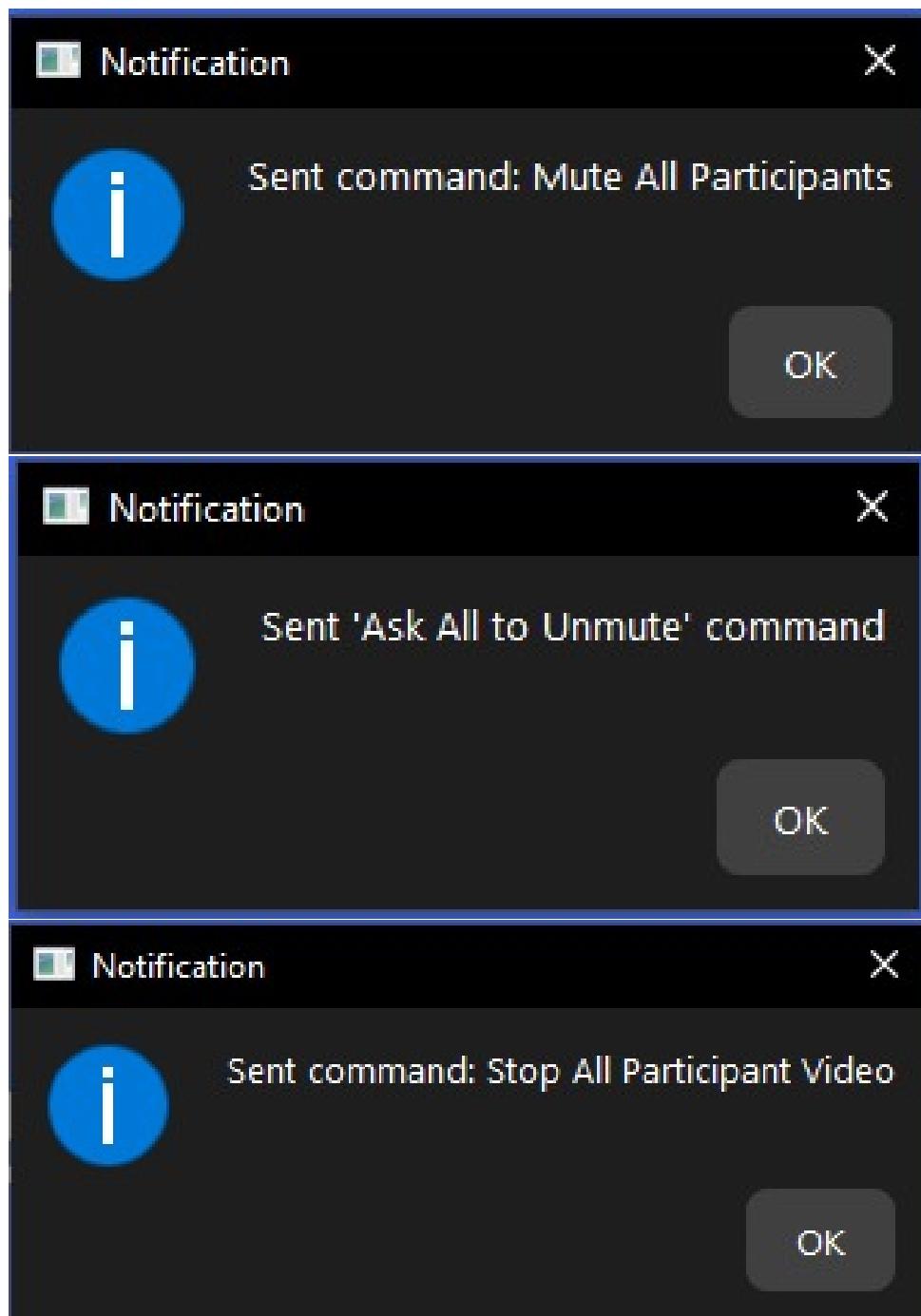


Figure 8.14: Few Admin Exclusive Commands



Figure 8.15: Hand Raising Feature

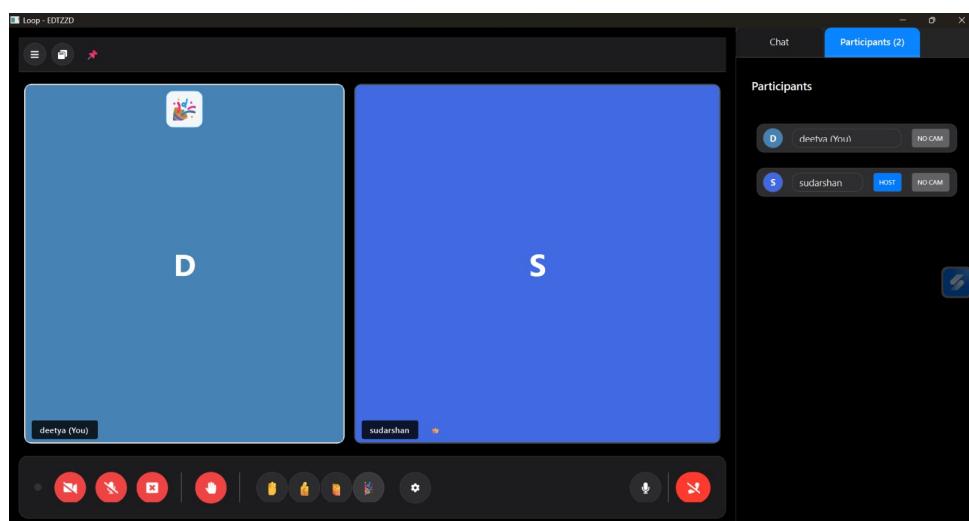


Figure 8.16: Reaction using Emojis

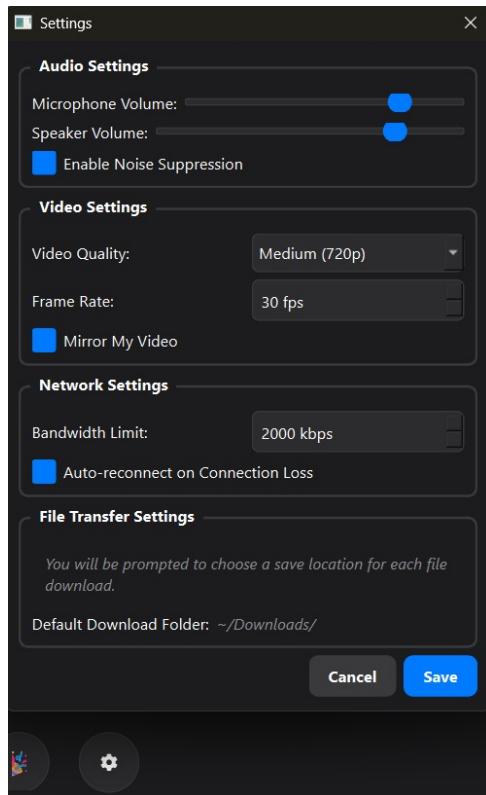


Figure 8.17: Meeting Settings

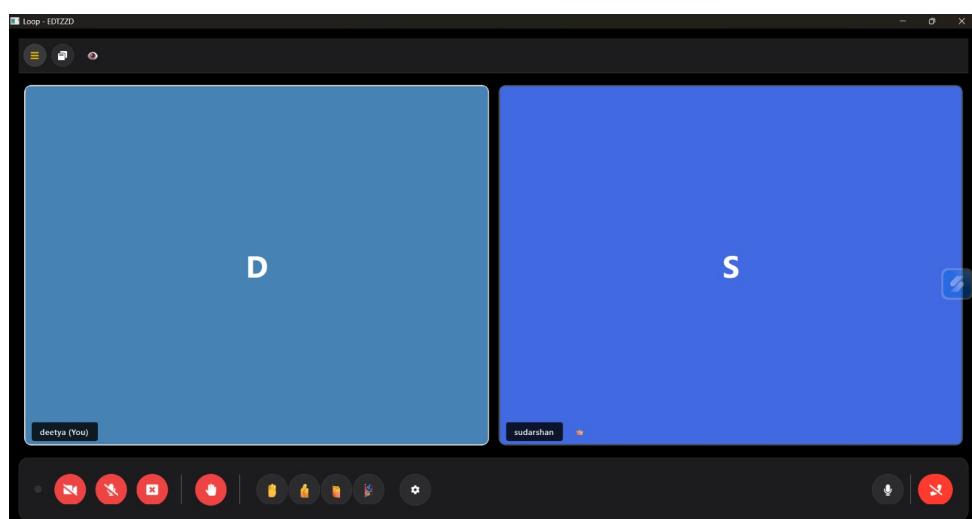


Figure 8.18: Hide Sidebar

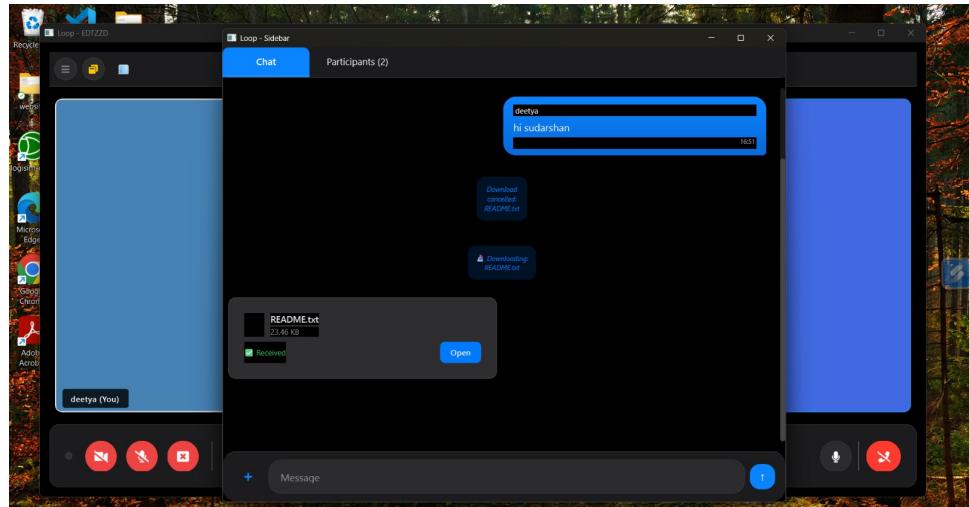


Figure 8.19: Sidebar Pop Up Mode

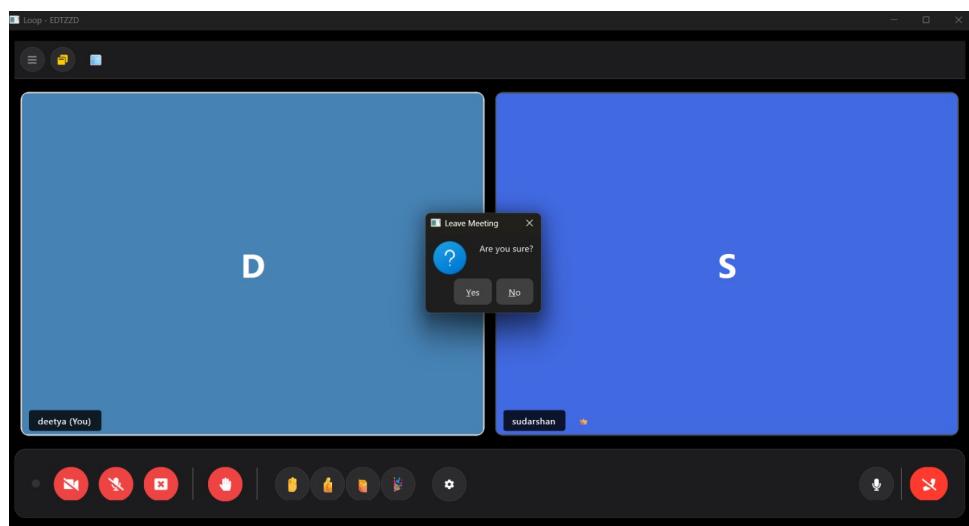


Figure 8.20: Meeting Exit Prompt

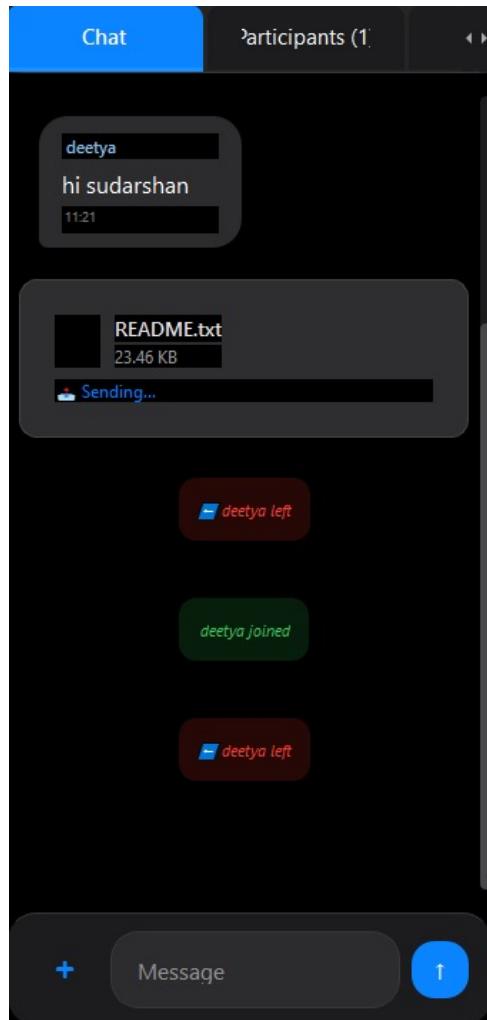


Figure 8.21: User Activities in Sidebar

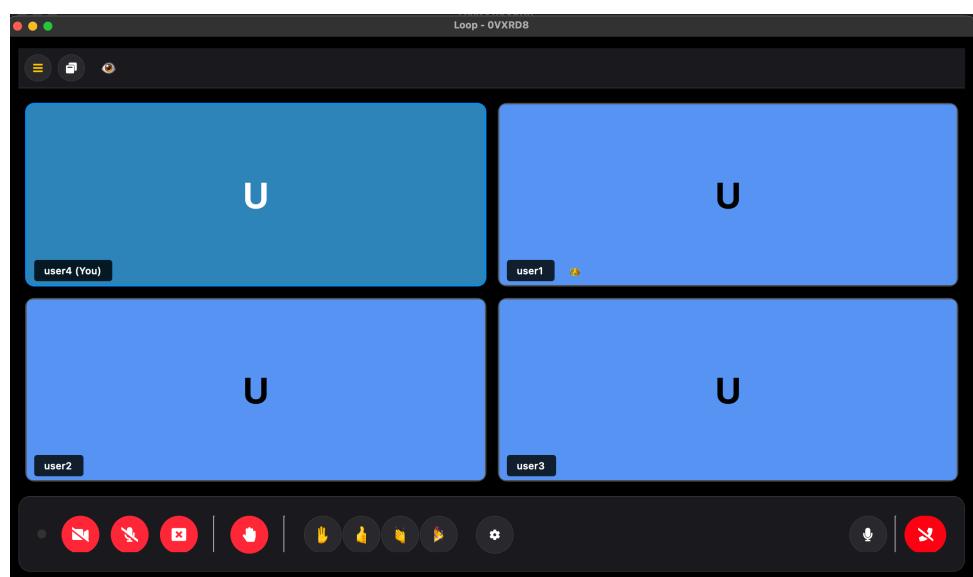


Figure 8.22: Dynamic Alignment of Tiles

Chapter 9

Setup Guide & Deployment

9.1 Overview

Information

This guide provides detailed, cross-platform instructions for setting up and running the Loop application. This is a client-server application requiring:

- One host runs `server.py`
- Every participant (including host) runs `client.py`
- All clients configured with host's LAN IP address
- All users on same network (e.g., same WiFi router)

9.2 Step 1: Environment Setup

9.2.1 Install Python

Important

Ensure Python 3.8 or newer is installed on your system. (This is needed to run pip).

Windows Download from [python.org](https://www.python.org). Check "Add Python to PATH" during installation.

macOS Python 3 typically pre-installed. Verify with `python3 --version`.

Ubuntu Install using: `sudo apt update && sudo apt install python3 python3-pip`

9.2.2 Dependencies Installation

1. Navigate to project folder
2. Create virtual environment: `python3 -m venv venv`

3. Activate environment:

- Windows: `venv\Scripts\activate`
- macOS/Ubuntu: `source venv/bin/activate`

4. Install packages: `pip install -r requirements.txt`

9.3 Step 2: Server Setup

9.3.1 Find Server's LAN IP Address

IP Address Discovery Find your IPv4 address starting with `192.168.x.x`, `10.x.x.x`, or `172.16.x.x`

- **Windows:** Run `ipconfig`
- **macOS:** Run `ifconfig | grep "inet "`
- **Ubuntu:** Run `ip addr show`

9.3.2 Run the Server

1. Navigate to the folder corresponding to your Operating System (e.g., `windows/`, `macos/`, or `ubuntu/`).
2. Open a terminal or command prompt in this folder.
3. **On macOS/Ubuntu:** Make the executable runnable: `chmod +x server`
4. Run the server: `./server` (or `server.exe` on Windows).

Important

Allow the application through your firewall when prompted. The server will print the listening IP and port.

9.4 Step 3: Client Setup

9.4.1 Run the Client

1. Navigate to the folder corresponding to your Operating System (e.g., `windows/`, `macos/`, or `ubuntu/`).
2. Open a terminal or command prompt in this folder.
3. **On macOS/Ubuntu:** Make the executable runnable: `chmod +x client`

- Run the client: `./client` (or `client.exe` on Windows).

Key Achievement

The login window should appear. You can now create or join a meeting!

9.5 Troubleshooting Guide

Table 9.1: Common Issues and Solutions

Issue	Cause	Solution
Connection failed	Server not running	Start the server executable first
Cannot connect	Firewall blocking	Allow the app in firewall settings
Camera not working	OS permissions	Grant camera access in settings
PlaceholderBG Laggy video/audio	Network congestion	Move closer to WiFi router

Chapter 10

Project Summary & Achievements

10.1 Project Overview

Key Achievement

The Loop Conference Application successfully demonstrates a comprehensive socket-based communication system that provides enterprise-grade conferencing features without internet dependency.

10.2 Key Achievements

10.2.1 Technical Accomplishments

1. Socket Programming Mastery

Successfully implemented both TCP and UDP protocols for different communication requirements.

2. Real-Time Performance

Achieved low-latency audio/video streaming suitable for real-time conversations.

3. Server-Side Mixing

Implemented NumPy-based selective audio mixing to eliminate echo problems.

4. Reliability Engineering

Used TCP for all control, text, and file data, ensuring no data loss.

5. Complex Protocol Design

Designed multi-stage protocols for screen sharing and file sharing.

6. Concurrent Programming

Mastered multi-threading with proper synchronization and thread-safe UI updates.

10.2.2 Feature Implementation Status

Table 10.1: Module Implementation Overview

Feature Module	Status
Multi-User Video Conferencing (UDP)	✓
Multi-User Audio Conferencing (UDP)	✓
Screen & Slide Sharing (TCP)	✓
Group Text Chat (TCP)	✓
File Sharing (TCP, Brokered P2P)	✓
Administrative Host Controls	✓

10.2.3 Architecture Highlights

System Design Excellence

- **Dual-Socket Architecture:** Intelligent protocol selection based on data characteristics
- **Selective Audio Mixing:** Hybrid SFU/MCU approach eliminating echo
- **Efficient Video Relay:** Zero server-side transcoding, minimal CPU overhead
- **Brokered P2P File Transfer:** Server acts as relay without storage
- **Thread-Safe Design:** Proper concurrent programming with signal/slot mechanism

10.3 Technical Insights & Best Practices

10.3.1 Protocol Selection Wisdom

1. Critical Design Decision

The choice of TCP vs. UDP is the most critical design decision for real-time applications.

2. Custom Headers for UDP

Custom headers are essential to identify sender and data type in connectionless protocols.

3. Hybrid Approach Wins

The most effective systems use both TCP and UDP, combining their strengths.

10.4 Performance Characteristics

Table 10.2: System Performance Metrics

Metric	Specification
Video Frame Rate	20-30 FPS (configurable)
Audio Sample Rate	44.1 kHz (standard quality)
Video Latency	Less than 100ms on LAN (UDP)
Audio Latency	Less than 50ms on LAN (UDP)
Message Delivery	100% reliable (TCP)
File Transfer Integrity	100% reliable (TCP + Base64)

10.5 Conclusion

Key Achievement

Loop represents a comprehensive implementation of modern real-time communication principles. The project successfully demonstrates:

- Mastery of socket programming and network protocols
- Understanding of real-time media processing
- Practical application of concurrent programming
- System design for performance and reliability
- User-centric interface development

The application provides a solid foundation for secure, low-latency LAN-based conferencing without internet dependency.

Information

Final Note: This documentation serves as both a technical reference and a learning resource. The architectural decisions and implementation patterns discussed here are applicable to a wide range of real-time networking applications.

End of Documentation
