

Video Conferencing Application over LAN

A Robust, Standalone Multi-User Communication System

Dhage Pratik Bhishmacharya (Roll No. CS23B1047)
Saumadeep Sardar (Roll No. CS23B1049)

Developed for Computer Networks Course

November 5, 2025

Contents

1	Abstract	3
2	Introduction	4
2.1	Project Overview	4
2.2	Objectives	4
2.3	Technology Stack	4
2.4	Computer Networks Context	5
3	System Architecture	6
3.1	Network Model	6
3.2	Key Components	8
3.3	Protocol Design	8
3.4	Component Interactions	8
4	Implementation Details	11
4.1	Socket Programming Fundamentals	11
4.2	Server Implementation	11
4.2.1	Client Management	11
4.2.2	Media Broadcasting	11
4.2.3	Audio Mixing	12
4.2.4	File Handling	12
4.3	Client Implementation	12
4.3.1	GUI Framework	12
4.3.2	Stream Capture and Transmission	13
4.3.3	Error Resilience	13
4.3.4	Screen Sharing	13
4.4	Constants and Utilities	13
5	Features and Usage	14
5.1	Video Conferencing	14
5.2	Audio Conferencing	14
5.3	Screen/Slide Sharing	14
5.4	Group Text Chat	15
5.4.1	Chat and Multicast	16
5.4.2	File Sharing	17
5.5	Installation and Setup	18
5.6	Usage Workflow	20
6	Challenges and Future Work	21
6.1	Challenges	21

6.2	Limitations	21
6.3	Future Enhancements	21
7	Conclusion	22

Chapter 1

Abstract

This document provides a comprehensive overview and in-depth analysis of the Video Conferencing Application, a client-server system designed for real-time, multi-user collaboration over Local Area Networks (LANs). The application integrates multi-user video and audio conferencing, screen/slide sharing, group text chat, and secure file sharing, all without requiring internet connectivity. Built using Python with PyQt6 for the GUI, OpenCV for video processing, and PyAudio for audio, the system ensures low-latency communication suitable for team environments with restricted or unreliable networks.

Emphasizing practical networking principles, it leverages socket programming for UDP-based media streams and TCP for reliable control, with server-side optimizations like audio mixing and file integrity checks. Key features include dynamic video grid rendering, presenter-controlled screen sharing with window selection, timestamped direct/group chat, and store-and-forward file distribution with MD5 hashing and progress tracking. Cross-platform compatibility (Windows, Linux, macOS) is achieved through fallback mechanisms. This report details the architecture, implementation, usage, performance, and future considerations, balancing technical depth with user-centric design.

Chapter 2

Introduction

2.1 Project Overview

The Video Conferencing Application addresses the need for offline collaboration tools in scenarios such as corporate intranets, educational institutions, or remote sites with limited connectivity. It operates exclusively on LANs using TCP/IP sockets and features cross-platform support for both Windows and Ubuntu environments. Supporting up to 10 concurrent users with optimized bandwidth usage, it focuses on transport and application layers to enable seamless interaction without external dependencies, making it ideal for secure or isolated environments.

2.2 Objectives

- Enable real-time video and audio streams with low latency via UDP/TCP integration.
- Provide secure, one-at-a-time screen sharing with window selection and integrity checks.
- Facilitate group chat and direct messaging with timestamps and multicast routing.
- Implement reliable file sharing with progress tracking, validation, and store-and-forward model.
- Ensure cross-platform deployment, graceful session management, and intuitive GUI.

2.3 Technology Stack

Component	Technologies
Networking	Socket Programming (TCP/UDP)
GUI	PyQt6
Video Capture	OpenCV, MSS/pyscreenshot
Audio	PyAudio, NumPy (mixing)
File Handling	Hashlib (MD5 integrity)
Cross-Platform	pygetwindow (window selection)

Table 2.1: Technology Stack

2.4 Computer Networks Context

This project illustrates foundational concepts without overwhelming complexity:

- **Transport Layer:** TCP for reliable chat/files, UDP for real-time media.
- **Application Layer:** Custom serialized messages for efficient routing.
- **Network Layer:** LAN broadcasting via IP binding.
- **Performance Aspects:** Compression for congestion avoidance.

Networking Layer	Implementation Focus
Physical/Data Link	Ethernet LAN assumptions
Network	IP addressing (hostname resolution)
Transport	TCP/UDP sockets with heartbeats
Application	Message serialization (pickle)

Table 2.2: OSI Layer Mapping

Chapter 3

System Architecture

3.1 Network Model

The system follows a client-server architecture:

- **Server:** Central hub for connection management, media broadcasting, audio mixing, and file storage.
- **Clients:** Handle local capture, rendering, and user interactions.

Communication protocols:

- TCP (Port 53530): Main connection for chat, files, and control messages.
- UDP (Ports 53531/53532): Video/audio streams for low latency.

The star topology simplifies session management on LANs, with server binding to all interfaces for easy discovery.

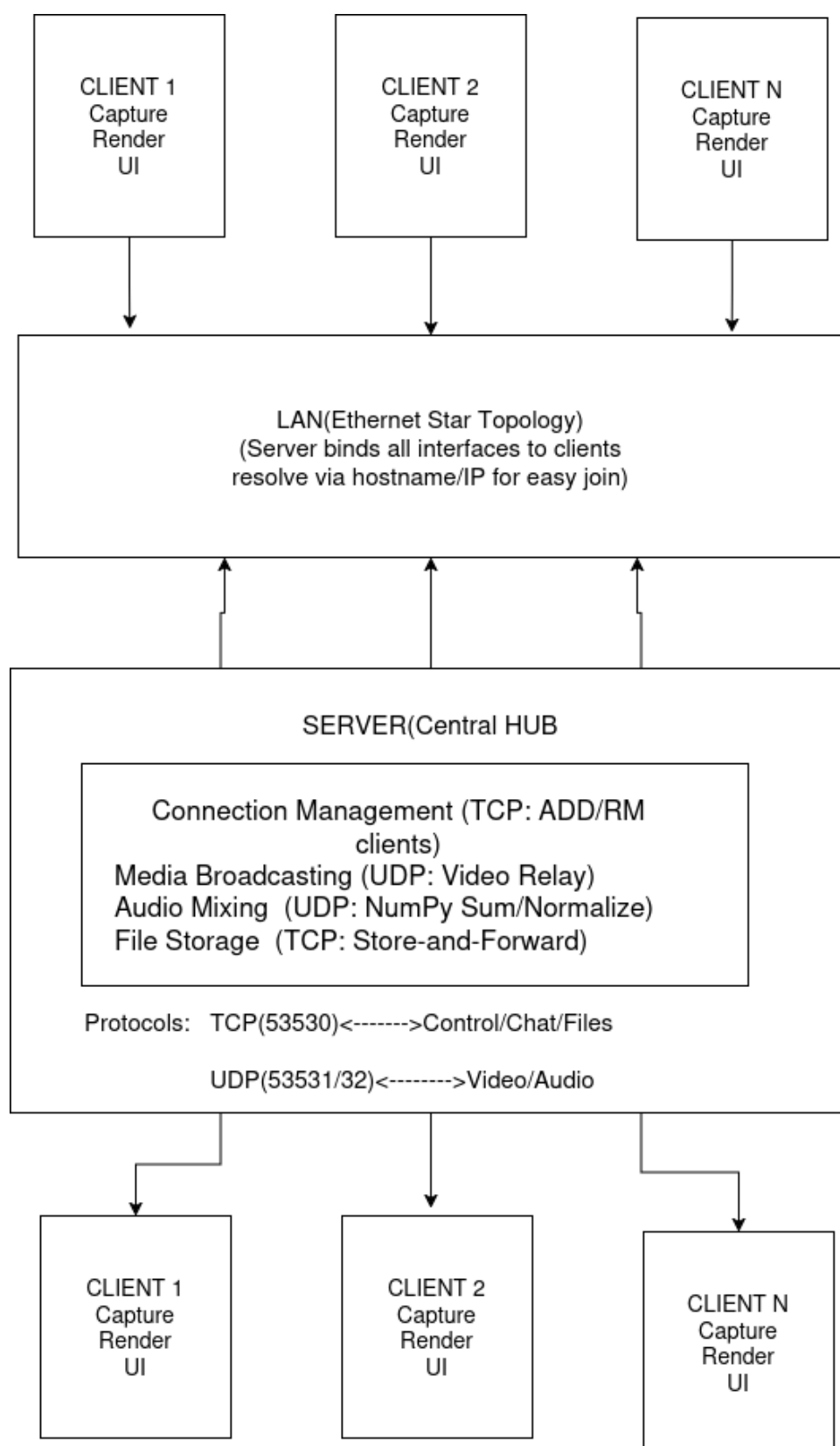


Figure 3.1: System Architecture Diagram

3.2 Key Components

- **Server Components:** Client registry, media servers (video/audio), audio mixer thread, file storage directory.
- **Client Components:** GUI (MainWindow, VideoWidget), capture threads (Camera, Microphone, ScreenCapturer), connection handler.

3.3 Protocol Design

Messages are dataclasses serialized via pickle, prefixed with length headers for TCP. UDP uses direct sends for speed.

Algorithm 1 Message Transmission Protocol

Require: Sender socket s , message m

```
1:  $bytes \leftarrow \text{pickle.dumps}(m)$ 
2:  $header \leftarrow \text{pack}(>I, \text{len}(bytes))$  {TCP only}
3: if  $s$  is TCP then
4:    $s.\text{sendall}(header + bytes)$ 
5: else
6:   {UDP}
7:    $s.\text{sendto}(bytes, \text{addr})$ 
8: end if
```

Rationale: Balances reliability and speed for different data types.

3.4 Component Interactions

Server uses threading for concurrency; clients employ QThreads for async operations.

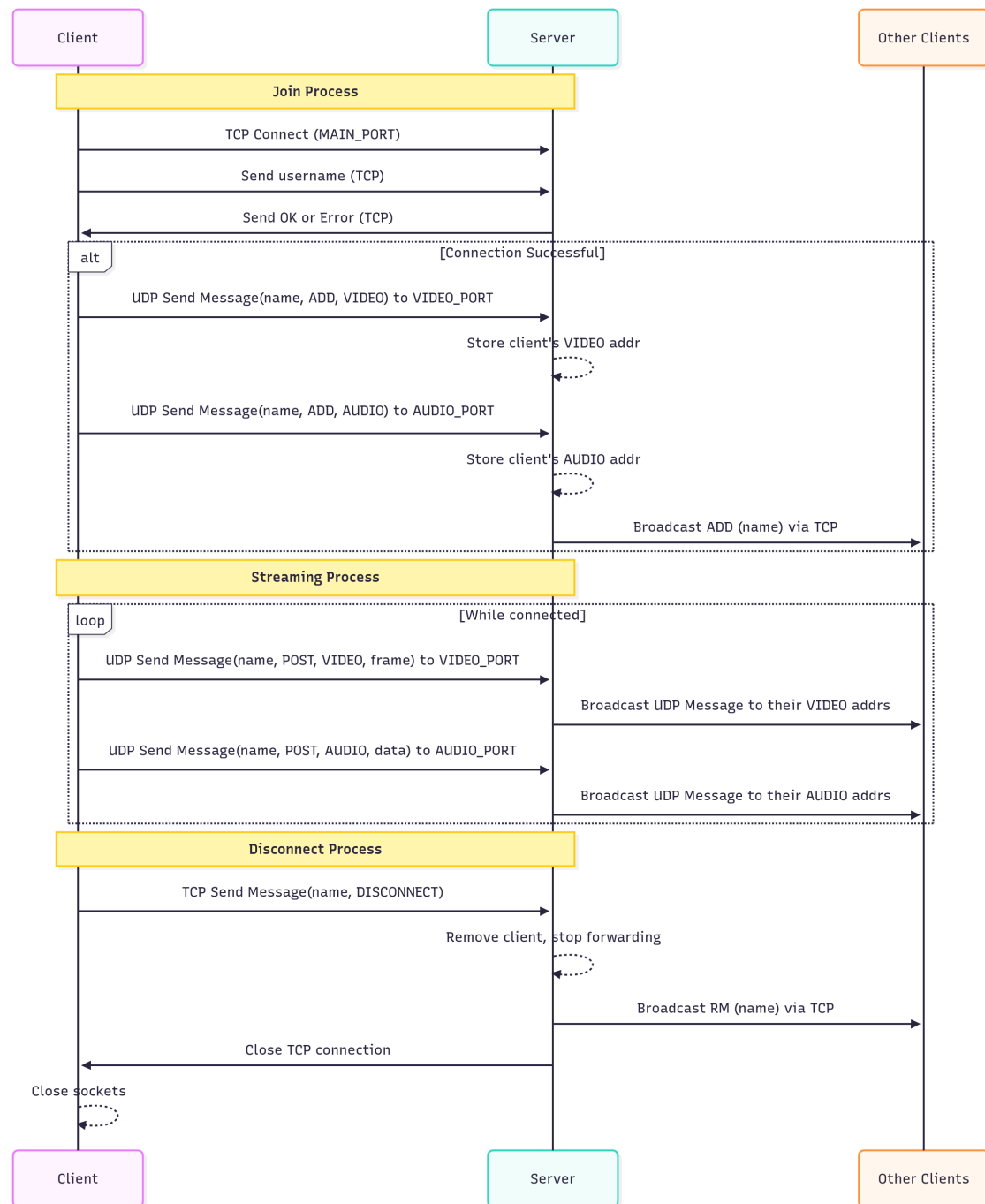


Figure 3.2: Component Interaction Flow

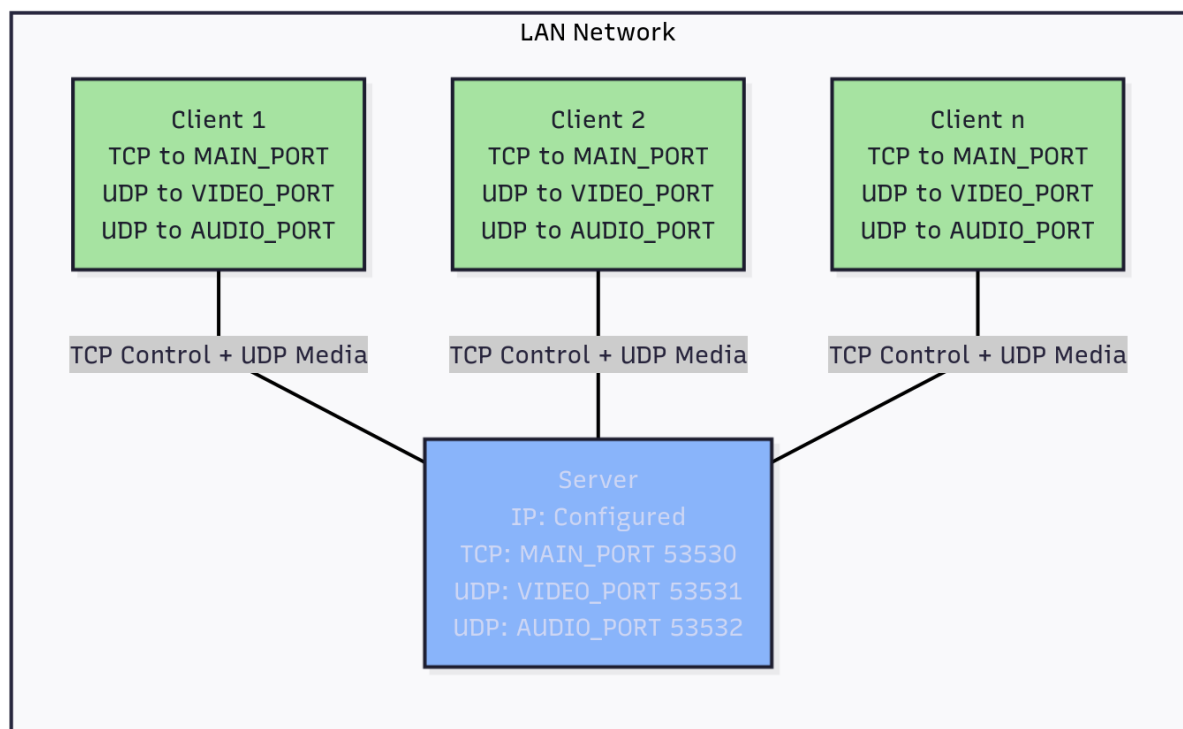


Figure 3.3: LAN Star Topology

Chapter 4

Implementation Details

4.1 Socket Programming Fundamentals

Sockets are extended for framed messaging, handling partial receives.

```
1 def send_bytes(self, msg):
2     msg = struct.pack('>I', len(msg)) + msg
3     self.sendall(msg)
4
5 def recv_bytes(self):
6     raw_msglen = self.recvall(4)
7     if not raw_msglen: return b''
8     msglen = struct.unpack('>I', raw_msglen)[0]
9     return self.recvall(msglen)
```

Listing 4.1: Socket Extensions for Reliability

4.2 Server Implementation

The server (`server.py`) manages sessions with threading.

4.2.1 Client Management

Dictionary-based registry with heartbeats for liveness.

4.2.2 Media Broadcasting

UDP listeners map client addresses; video relayed, audio mixed.

```
1 def media_server(media: str, port: int):
2     conn.bind((IP, port))
3     while True:
4         msg_bytes, addr = conn.recvfrom(MEDIA_SIZE[media])
5         msg = pickle.loads(msg_bytes)
6         if msg.request == ADD:
7             client.media_addrs[media] = addr
8         elif media == AUDIO:
9             audio_buffers[msg.from_name].append(msg.data)
10        else:
11            broadcast_msg(msg.from_name, msg.request, msg.data_type,
                           msg.data)
```

Listing 4.2: UDP Media Server

4.2.3 Audio Mixing

NumPy sums/normalizes buffers every 33ms.

```

1 def audio_mixer():
2     while True:
3         time.sleep(MIX_INTERVAL)
4         with mixer_lock:
5             mix_data = np.zeros(BLOCK_SIZE, dtype=np.int16)
6             active_count = 0
7             for name, buf in audio_buffers.items():
8                 if buf and clients.get(name, {}).connected:
9                     latest = buf[-1] if isinstance(buf[-1], np.ndarray)
10                else np.frombuffer(buf[-1], dtype=np.int16)
11                    mix_data += latest.astype(np.int64)
12                    active_count += 1
13            if active_count > 0:
14                mix_data = (mix_data / active_count).astype(np.int16)
15                # Broadcast mixed audio

```

Listing 4.3: Audio Mixing Snippet

Algorithm 2 Audio Mixing

Require: Buffers per client, BLOCK_SIZE=2048

```

1: mix ← zeros(BLOCK_SIZE, int16)
2: count ← 0
3: for each active client do
4:   latest ← buffer[-1].asarray(int16)
5:   mix += latest.astype(int64)
6:   count++
7: end for
8: if count > 0 then
9:   mix = (mix / count).astype(int16)
10:  Broadcast mix.tobytes() to all
11: end if

```

4.2.4 File Handling

Stored in server_files/ with MD5; multicast availability.

4.3 Client Implementation

The client (client.py) uses QThreads for non-blocking I/O.

4.3.1 GUI Framework

PyQt6: Docks for chat, grids for videos.

```

1 def handle_msg(self, msg: Message):
2     # ... (video/audio updates)
3     elif msg.data_type == FILE:
4         # Chunk assembly and hash verification
5         full_data = b''.join(self.pending_downloads[file_id]['chunks'])

```

```

6         if hashlib.md5(full_data).hexdigest() != expected_hash:
7             print("[ERROR] File integrity fail")

```

Listing 4.4: Message Handling Snippet

4.3.2 Stream Capture and Transmission

Video: OpenCV compress, 30 FPS UDP. Audio: PyAudio raw bytes.

4.3.3 Error Resilience

Try-except for resets; reconnects limited to 3.

```

1 def handle_conn(self, conn: socket.socket, media: str):
2     while self.connected:
3         try:
4             if media in [VIDEO, AUDIO]:
5                 msg_bytes, _ = conn.recvfrom(MEDIA_SIZE[media])
6             else:
7                 msg_bytes = conn.recv_bytes()
8             # ... load and handle
9         except (OSError, ConnectionResetError):
10            self.connected = False

```

Listing 4.5: Client Recv with Fallback

4.3.4 Screen Sharing

The screen sharing feature allows one user at a time to broadcast their entire desktop (or primary monitor) to all other participants in the conference. It is built on top of the existing client-server architecture using:

- TCP for control signaling
- UDP for media broadcasting
- Server-mediated distribution (star topology)
- JPEG compression for efficient bandwidth usage

4.4 Constants and Utilities

`constants.py`: Ports, Message dataclass, socket helpers.

Algorithm 3 File Transfer Sequence

- 1: Client uploads chunks + EOF to server (TCP)
 - 2: Server: assemble, hash, store; multicast `FILE_AVAILABLE(meta)`
 - 3: Receiver: on available, send `GET_FILE(id)`
 - 4: Server streams chunks + EOF
 - 5: Client: verify hash, save
-

Chapter 5

Features and Usage

5.1 Video Conferencing

Capture: Webcam at 360p, resizable. Rendering: Dynamic grid.

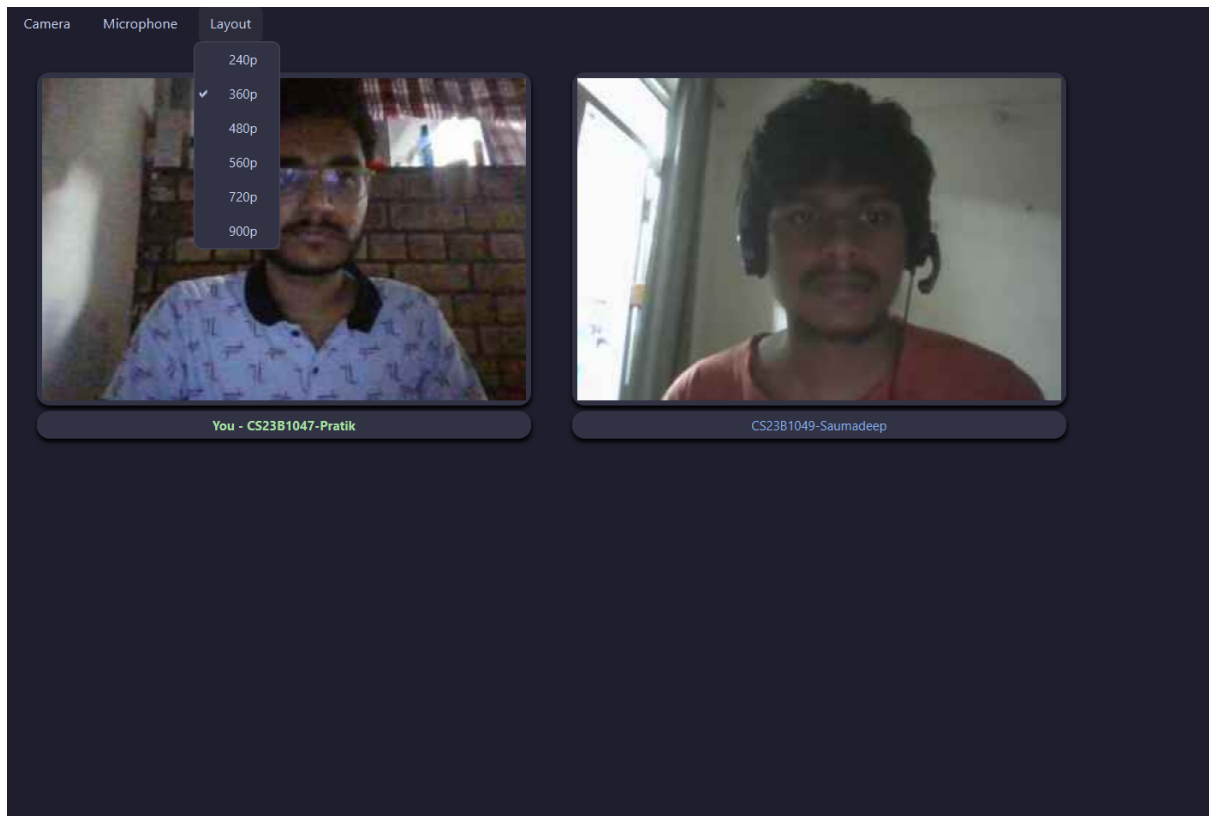


Figure 5.1: Video Grid Interface

5.2 Audio Conferencing

Server mixes to single stream; client threads for playback.

5.3 Screen/Slide Sharing

One presenter; JPEG compression, window menu.

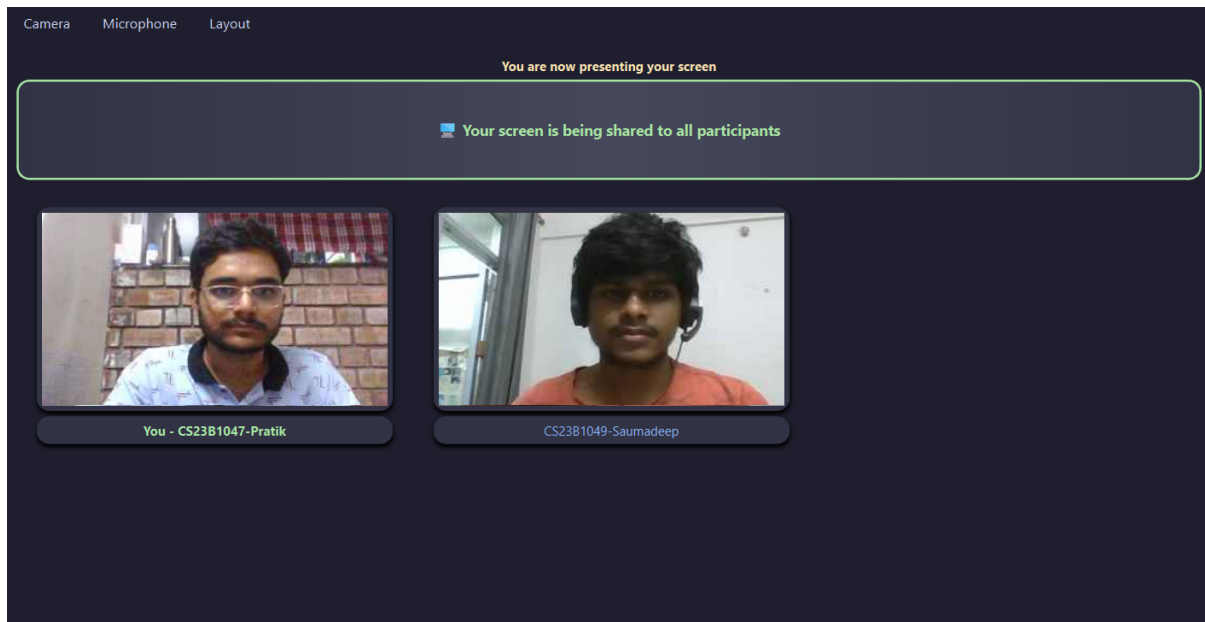


Figure 5.2: Screen Share Widget (Presenter's side)

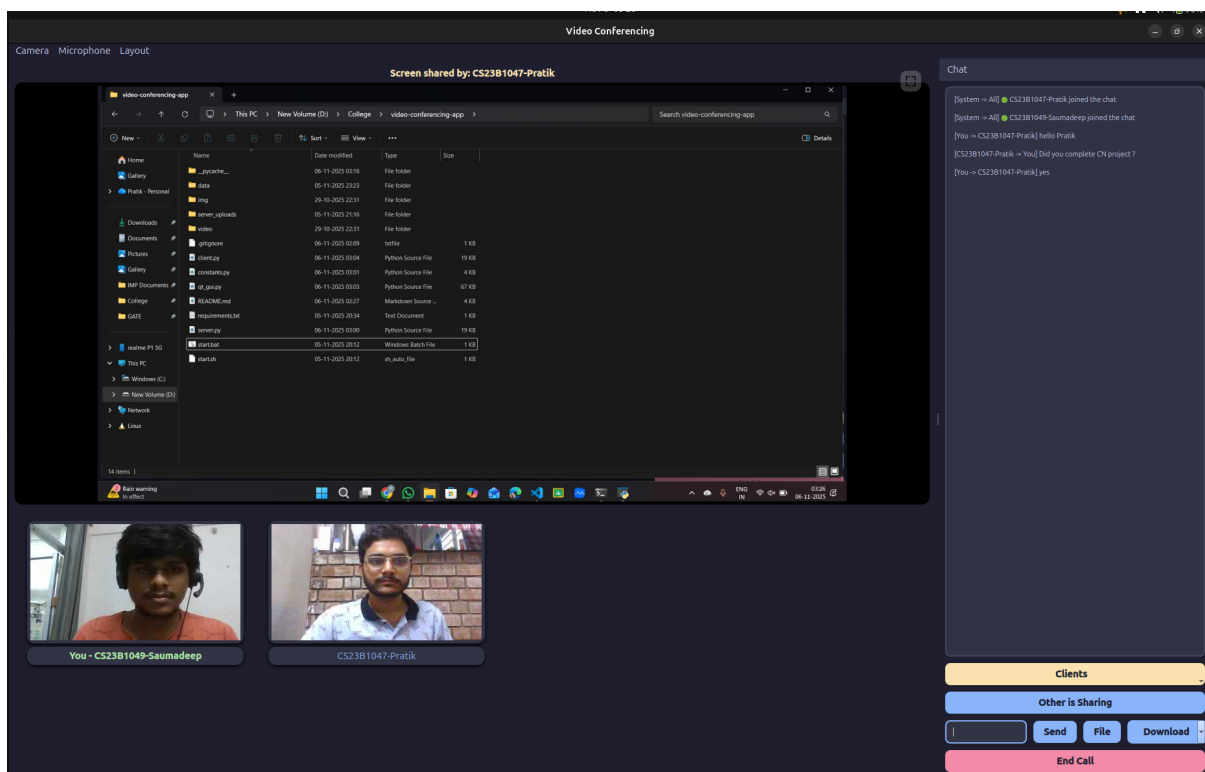


Figure 5.3: Screen Share Widget (Viewer's side)

5.4 Group Text Chat

Timestamps, DM tabs, multicast to selected.

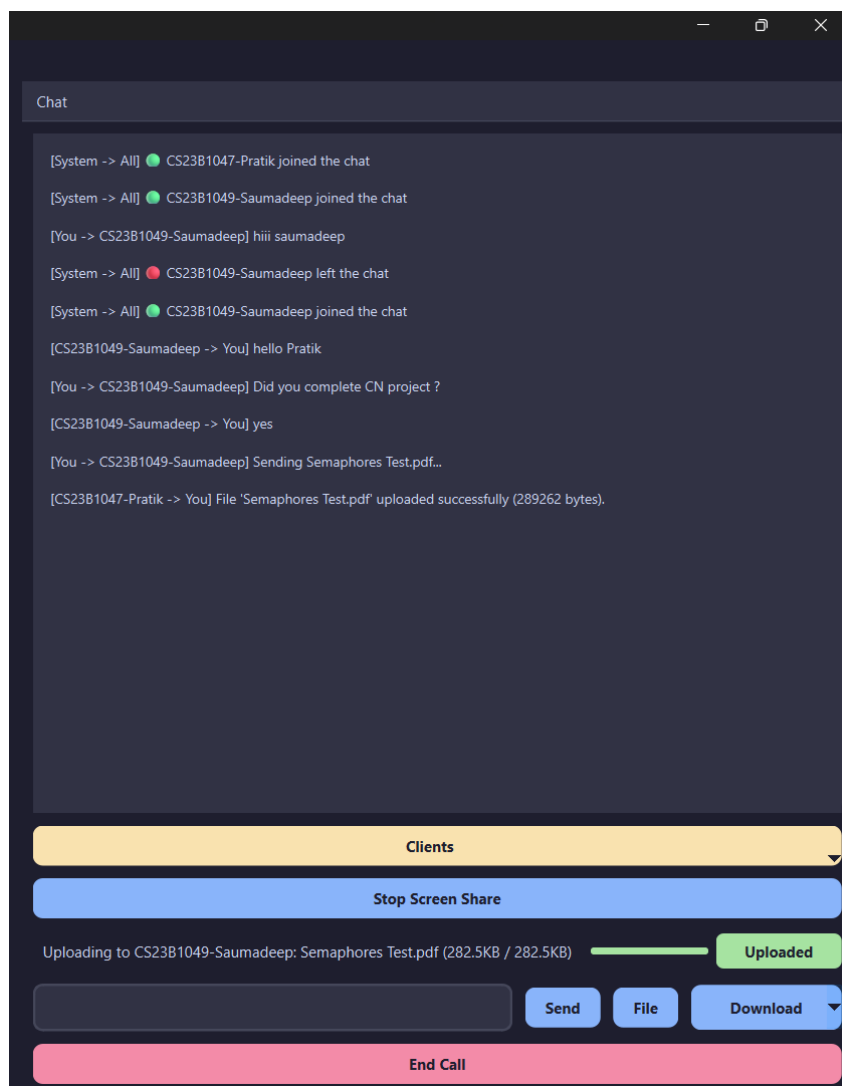


Figure 5.4: Chat Dock with Tabs

5.4.1 Chat and Multicast

Chat messages are sent via TCP as `Message(data_type=TEXT, ...)` objects. The client specifies `to_names` (empty for group chat). The server routes messages using `multicast_msg`:

```

1 def multicast_msg(msg: Message, exclude=None):
2     for name in (msg.to_names or [n for n in clients if n != msg.from_name]):
3         if name != exclude and name in clients:
4             clients[name].conn.send_bytes(pickle.dumps(msg))

```

Direct messages create dynamic tabs in the GUI via `chat_tab_signal`. Timestamps are added client-side upon receipt.

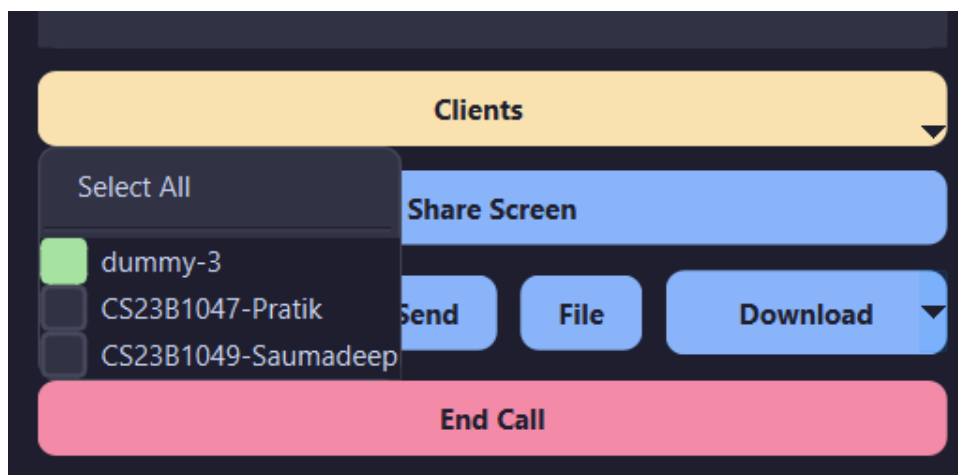


Figure 5.5: Selecting client for file and chat sending

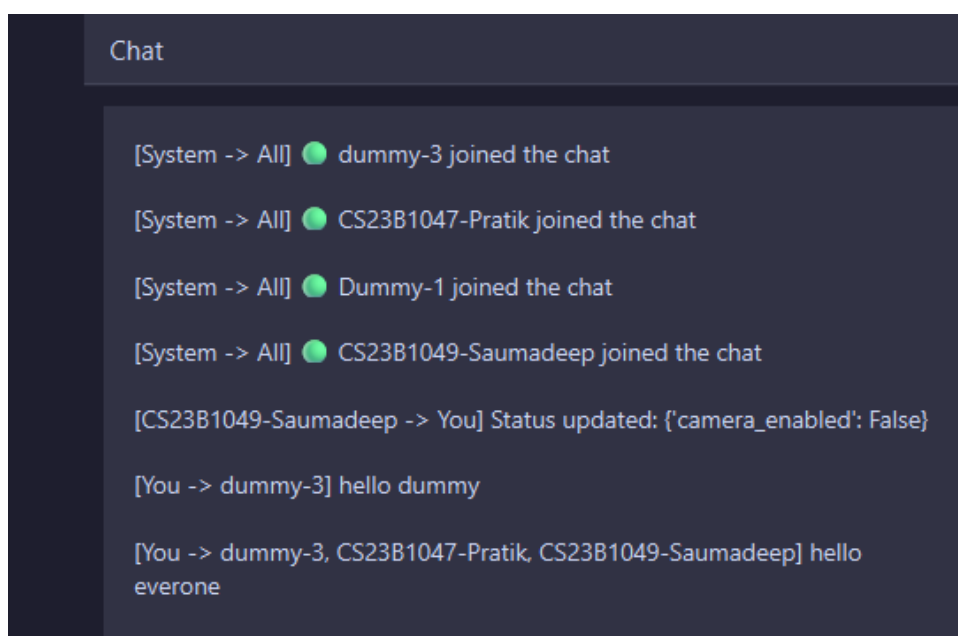


Figure 5.6: Sending messages to single client vs group

5.4.2 File Sharing

Files are transferred in chunks over TCP using a store-and-forward technique. An upload begins with the filename, followed by data chunks, and ends with a `None` marker. The server stores each file per recipient inside `data/` and tracks them using `files_index`.

```

1 # Client upload (send_file)
2 self.send_msg(Message(FILE, filename, to_names))
3 for chunk in file_chunks:
4     self.send_msg(Message(FILE, chunk, to_names))
5 self.send_msg(Message(FILE, None, to_names))
6
7 # Server handles this via handle_file_post (fixed version)

```

Clients poll `GET_FILES` to list available files. On `DOWNLOAD_FILE(id)`, the server streams stored chunks. Progress updates are emitted using `download_progress_signal`. File integrity is verified through MD5:

```

1 full_data = b''.join(chunks)
2 if hashlib.md5(full_data).hexdigest() != expected_hash:
3     print("[ERROR] File integrity fail")

```

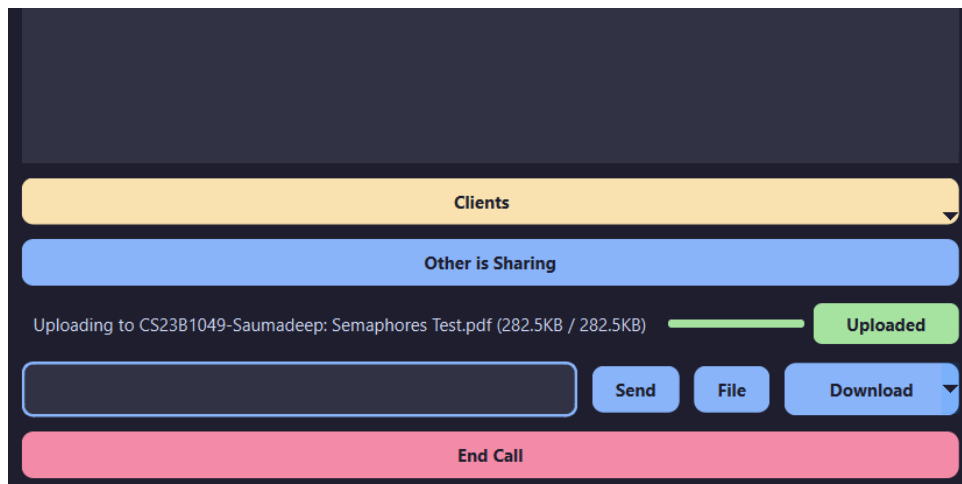


Figure 5.7: File Upload Queue

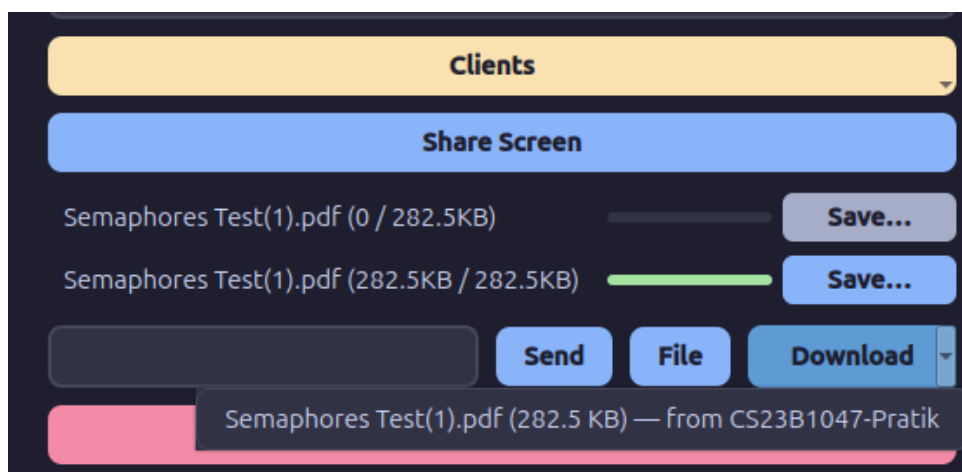


Figure 5.8: File Download Queue

5.5 Installation and Setup

1. Git Clone: `https://github.com/saumadeepsardar/video-conferencing-app.git`
2. Select Folder: `cd video-conferencing-app`
3. Create venv: `python3 -m venv venv`
4. Activate venv: `source venv/bin/activate` for ubuntu
`venv/Sources/activate` for windows
5. Install: `pip install -r requirements.txt`
6. Server: `python server.py`

7. Clients: `python client.py`
8. Login with server IP and unique username.

Other Way of Installation:

1. Git Clone: `https://github.com/saumadeepsardar/video-conferencing-app.git`
2. Select Folder: `cd video-conferencing-app`
3. Create venv: `python3 -m venv venv`
4. Activate venv: `source venv/bin/activate` for ubuntu
`venv/Sources/activate` for windows
5. Install: `pip install -r requirements.txt`
6. Executable: Double click `start.bat` for windows run `./start.sh`
7. Select Role: select client or server
8. Login with server IP and unique username.

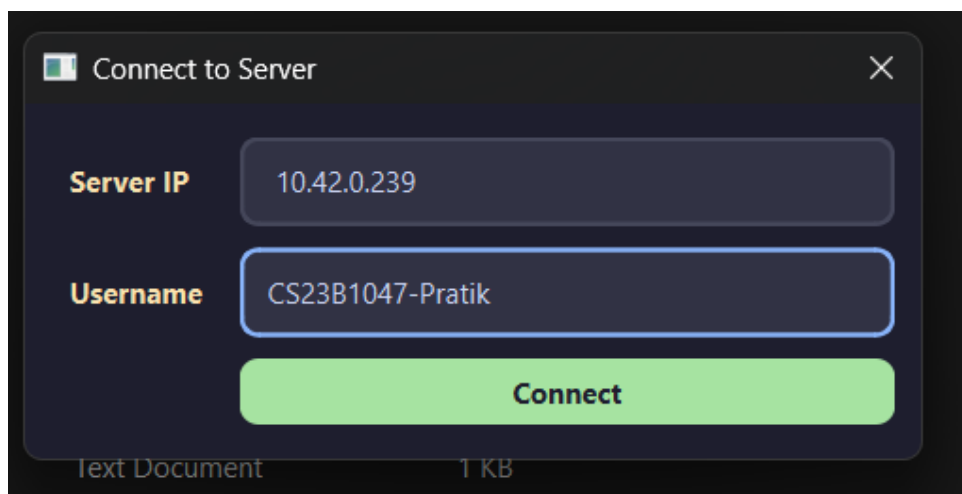


Figure 5.9: Client Login Interface

5.6 Usage Workflow

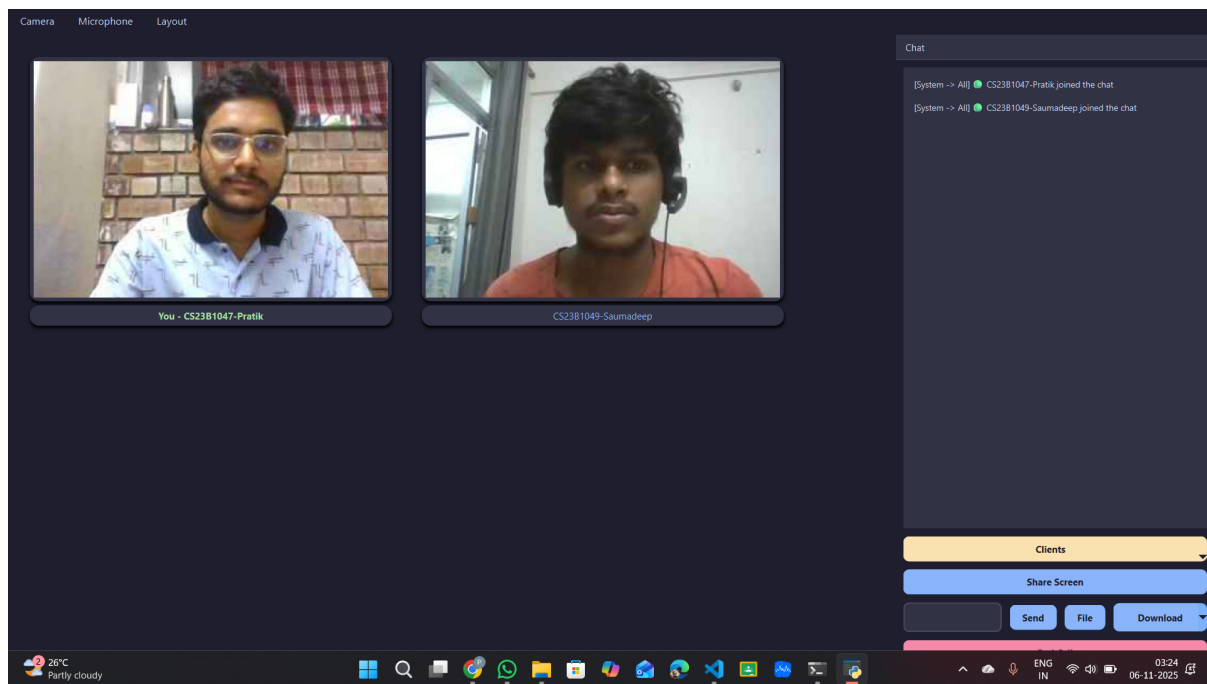


Figure 5.10: Complete GUI Layout

Chapter 6

Challenges and Future Work

6.1 Challenges

- Cross-platform perms (try-except resolution).
- UDP loss (compression mitigation).
- Platform capture errors (fallbacks).

6.2 Limitations

- No QoS prioritization.
- Flat LAN assumption.
- Threading limits scalability.

6.3 Future Enhancements

- Multi-room with passwords.
- AES encryption for media.
- Mobile Qt support.
- RTP/QUIC for advanced networking.
- Hierarchical servers for scale.

Chapter 7

Conclusion

This project successfully delivered a robust, standalone, and server-based communication application operating exclusively on a Local Area Network. Using fundamental socket programming, the system provides a complete, internet-independent collaboration suite.

All five core functional requirements were fully integrated. The application strategically uses UDP for low-latency real-time video and audio, while leveraging TCP for the reliable, error-free transmission of screen sharing, group chat, and file transfers.

The client-server architecture proved to be a stable foundation, efficiently managing multiple users and mixed-protocol data streams. While future enhancements like end-to-end encryption are possible, the application in its current form fully meets the project's objectives, delivering a powerful and self-contained tool for real-time local collaboration.