Computer Network - Project

Title: Multi-User LAN Communication System

Team Members:

B. Nithin Ram Gopal – CS23B2050

N. Ravi Tejesh – CS23B2051

ConnectSphere -Video Conference Application

1. System Architecture

The ConnectSphere system is a client-server video conferencing application. It uses a **multithreaded, modular architecture** to separate concerns and manage the high I/O demands of real-time media.

1.1. Client Architecture

The client is a desktop application built in Python using CustomTkinter. Its architecture is designed to prevent the User Interface (UI) from freezing during network-intensive or CPU-intensive tasks. This is achieved by separating the application into four main components:

1. **View (MainAppView):** The CustomTkinter UI. It runs **exclusively on the main thread**. It is responsible for drawing all widgets and video frames. It *never* performs blocking I/O (like networking) directly.
2. **Controller (MeetingController):** The central "brains" and bridge component. It is owned by the View and provides a simple API for the UI to call (e.g., connect(), toggle_video()). It owns all backend managers.
3. **Backend Managers (CommandConnection, MediaManager):** These classes manage all the "heavy lifting."
    - CommandConnection: Manages the single, persistent TCP connection for commands, chat, and user lists. It runs its own listener thread (TCP-Control-Listen).
    - MediaManager: Manages all media I/O. It spawns and manages a pool of dedicated worker threads for:
        - Cam-Send: Capturing and sending video.
        - Mic-Capture: Capturing and sending audio.
        - UDP-Video-Recv: Receiving UDP video packets.
        - Video-Decode: A dedicated queue-based thread to decode JPEGs, preventing the network thread from blocking.
        - UDP-Audio-Recv: Receiving UDP audio packets.
        - Audio-Playback: Playing audio from a jitter buffer.
        - Screen-Share / Screen-View: Handling the TCP screen-sharing stream.
4. **State (StateManager):** A simple class that holds the shared state (e.g., is_connected, username, sending_audio) so all threads can access a single source of truth.

Key Data Flow (Client-Side)

To ensure stability, the UI is decoupled from the backend threads using two primary patterns:

- **GUI Task Queue (gui_queue):** When a backend thread (e.g., the network listener) needs to update the UI, it does *not* call the UI directly. Instead, it places a function onto the MeetingController's thread-safe gui_queue. A timer on the main thread (update_gui_queue) pulls tasks from this queue and executes them safely.
- **Decoupled Rendering (needs_render flag):** To prevent the UI from stuttering from thousands of incoming video frames, the on_video_frame callback (running on the main thread) does *not* immediately redraw the canvas. It simply updates an image in a dictionary and sets a single flag: self.needs_render = True. A separate, stable 30 FPS timer (render_loop) checks this flag and performs *one* full redraw of the canvas, rendering all new frames at once.

1.2. Server Architecture

The server is a multi-threaded Python application designed for high-concurrency. It is stateful and modular, with different managers handling specific protocols.

1. **Core (ConferenceServer):** The main class that manages the central participant state (self.participants) protected by a threading.RLock. It runs the main "acceptor" loops that listen on different TCP ports.
2. **TCP Acceptor Pattern (_run_acceptor):** The server listens on multiple TCP ports (9000, 9001, 9002). For each incoming connection on any of these ports, it spawns a new, dedicated handler thread (e.g., _handle_control_client, handle_connection, _handle_file_transfer).
3. **Sub-system Managers:**
    - **Control Plane:** Managed directly by ConferenceServer. The _handle_control_client thread uses a dispatcher dictionary (self.message_handlers) to route text-based JSON commands (e.g., hello, chat).
    - **Audio (AudioMixer):** A highly sophisticated module that runs **two** of its own threads:
        - _run_audio_receiver: Ingests all incoming UDP audio packets from all clients into per-client queues.

        - _run_audio_mixer: Runs on a high-precision ticker (every 16ms). It **creates a custom audio mix for each client** (summing all *other* participants' audio using NumPy) and sends the final mixed packet. This includes **Packet Loss Concealment (PLC)**, where it re-uses a client's last good packet if a new one hasn't arrived.
    - **Video (_run_video_relay):** A simple, "dumb" UDP forwarder. It runs one thread that listens for UDP video packets. When a packet arrives, it **prepends the sender's 4-byte IP address** and re-broadcasts the new, larger packet to all *other* clients.
    - **Screen Share (ScreenShareManager):** A stateful TCP manager that enforces a **single-presenter "lock"**. It handles the role-based handshake and then streams the presenter's frames to all connected viewers.
    - **File Transfer:** Managed by _handle_file_transfer, which handles the simple JSON-then-binary protocol for uploads and downloads.

2. Communication Protocols

The system uses a mix of TCP and UDP, with different data formats optimized for each feature.

2.1. Port 9000: TCP Command & Control

- **Protocol:** TCP
- **Format: Newline-Delimited JSON (NDJSON)**
- **Description:** A single, persistent TCP connection for reliable, ordered messages. The client and server exchange JSON objects separated by a \n newline character.
- **Key Messages:**
    - hello: Client -> Server. Initial handshake with name and UDP ports.
    - user_list: Server -> Client. Broadcasts the full list of connected users.
    - join / leave: Server -> Client. Notifies clients of a user's status change.
    - chat: Bidirectional. A public chat message.
    - private_chat: Client -> Server. A private message with a to_name field.
    - file_offer: Server -> Client. Notifies all clients that a file has been uploaded.

    - video_start / video_stop: Bidirectional. Notifies clients that a user's video state has changed.
    - reaction: Bidirectional. Transmits an emoji reaction.

2.2. Port 10000: UDP Video Relay

- **Protocol:** UDP
- **Format:** Custom Binary (Fragmented JPEG)
- **Client -> Server:** The client encodes a JPEG, splits it into ~1100 byte chunks, and prepends an 8-byte header to each chunk.
    - **Packet:** [Header] + [JPEG Chunk]
    - **Header:** struct.pack('!IHH', frame_id, total_parts, part_idx)
        - frame_id (4 bytes): Identifies the frame.
        - total_parts (2 bytes): Total chunks for this frame.
        - part_idx (2 bytes): The index (0...N) of this chunk.
- **Server -> Client:** The server receives the client's packet, prepends the sender's 4-byte binary IP, and forwards it to all other clients.
    - **Packet:** [Sender IP (4 bytes)] + [Original Header] + [JPEG Chunk]
    - **Client Logic:** The client uses this 4-byte prefix to identify which user the video stream belongs to and reassembles the fragments using the header.

2.3. Port 11000: UDP Audio Mixing

- **Protocol:** UDP
- **Format:** Raw PCM Audio
- **Client -> Server:**
    - **Packet:** [Raw PCM Chunk] (512 bytes: 256 samples * 16-bit)
- **Server -> Client:**
    - **Packet:** [Mixed PCM Chunk] (512 bytes)
    - **Server Logic:** The server receives raw chunks from all clients. It runs a mixer that sums all streams (excluding the recipient's own) and sends the final, mixed audio chunk.
- **Protocol Discrepancy:**
    - The **Client** (_audio_receive_loop) *expects* the server to prepend a 4-byte IP address (src_ip = socket.inet_ntoa(pkt[:4])) for echo cancellation.
    - The **Server** (_run_audio_mixer) sends *only* the raw mixed PCM chunk (pkt = mixed.tobytes()), without any IP prefix.

2.4. Port 9001: TCP Screen Sharing

- **Protocol:** TCP
- **Format:** Length-Prefixed JSON (Framed JSON)
- **Handshake (Both Client & Server):**
    1. Client connects and sends a framed JSON message:
        - [Length (4 bytes)] + [JSON: {"role": "presenter" | "viewer", "name": "..."}]
    2. Server receives and replies with a framed JSON message:
        - [Length (4 bytes)] + [JSON: {"status": "ok" | "denied", "reason": "..."}]
- **Data Stream (Protocol Discrepancy):**
    - **Client Send:** The client *continues* to use the framed JSON protocol, sending Base64-encoded JPEGs:
        - [Length (4 bytes)] + [JSON: {"type": "frame", "data": "<b64_jpeg_string>"}]
    - **Server Receive:** After the handshake, the server's _handle_presenter loop *switches* to a different, **incompatible** binary protocol. It expects:
        - [Length (4 bytes)] + [Raw JPEG Bytes]

2.5. Port 9002: TCP File Transfer

- **Protocol:** TCP
- **Format:** Hybrid (JSON Handshake + Raw Binary Stream)
- **Description:** This protocol is **consistent** between client and server. A new TCP connection is used for each transfer.
- **Upload Flow:**
    1. Client sends an initial, *non-prefixed* JSON message: {"type": "file_upload", ...}.
    2. Server replies with raw bytes: b"READY".
    3. Client streams the entire raw file over the socket.
    4. Server replies with raw bytes: b"DONE".
- **Download Flow:**

    1. Client sends an initial, *non-prefixed* JSON message: {"type": "file_download", ...}.
    2. Server replies with a *non-prefixed* JSON message: {"size": 123456}.
    3. Client replies with raw bytes: b"READY".
    4. Server streams the entire raw file over the socket.

3. User Guide

3.1. Prerequisites

Before you begin, you must have Python 3 and the following libraries installed. You can install them using pip:

Bash
pip install customtkinter opencv-python numpy Pillow mss pyaudio

3.2. Setup

You must run the **server** first, then one or more **clients**.

1. Running the Server:

1. Save the server.py file to your server machine.
2. Run the script from your terminal:

   Bash

   python "server.py"

3. The server will start and automatically detect its own network IP address. Look for a line like: [INFO] Server running on: 192.168.1.10
4. **Note this IP address.** All clients will need it to connect. (If it shows 127.0.0.1 or 0.0.0.0, you should use the machine's actual network IP, e.g., 192.168.1.10).

2. Running the Client:

1. Save the client.py file on your local machine (or any machine that can reach the server).
2. Run the script:

   Bash

   python "client.py"

3. The "Join Call" window will appear.

3.3. Using the Application

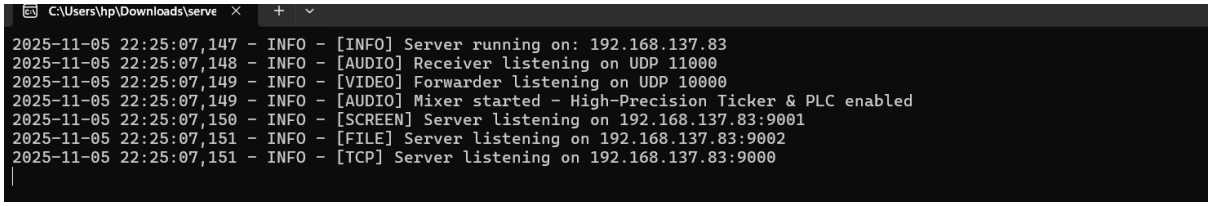
- **Joining a Call:**
    1. **Server IP:** Enter the IP address of the server you noted in the setup step.
    2. **Username:** Enter your desired display name. This must be unique.
    3. Click **"Join"**. The dialog will disappear, and you will enter the main application.
- **Main Interface:**
    - **Left Control Bar:** Contains the main buttons for interacting.
    - **Top Nav Bar:** Shows connection status, user count, and layout options.
    - **Main Canvas:** Shows all video feeds.
- **Core Controls (Left Bar):**
    - **Mic:** Toggles your microphone on/off. Red means you are "live."
    - **Camera:** Toggles your camera on/off. Red means you are "live."
    - **Screen:** Toggles screen sharing. This will share your **currently active (foreground) window**. Your camera will turn off while sharing.
    - **Chat:** Opens/closes the side panel to the "Messages" tab.
    - **Users:** Opens/closes the side panel to the "Attendees" tab.
    - **Exit:** Leaves the call and returns to the "Join Call" screen.
- **Chat and Reactions:**
    - In the "Messages" tab, you can send public messages to "Everyone."
    - To send a **private message**, select a user's name from the "To:" dropdown.
    - Click any of the **emoji icons** (e.g., 👍, 🎉) to send a floating reaction to all users.
- **File Sharing:**

    1. Open the side panel and go to the **"Files"** tab.
    2. Click **"Upload File"** to select a file. It will be sent to the server, and all other users will receive a download prompt.
    3. When someone else uploads a file, a popup will ask you to "Accept" or "Decline" the download.
    4. Accepted files are saved in your **Downloads/ConferenceFiles** folder.
    5. Click **"View Downloads"** to open this folder.
- **Pinning & Layouts:**
    - **Pin:** Hover your mouse over any user's video tile. A "Pin" button will appear. Click it to pin them.

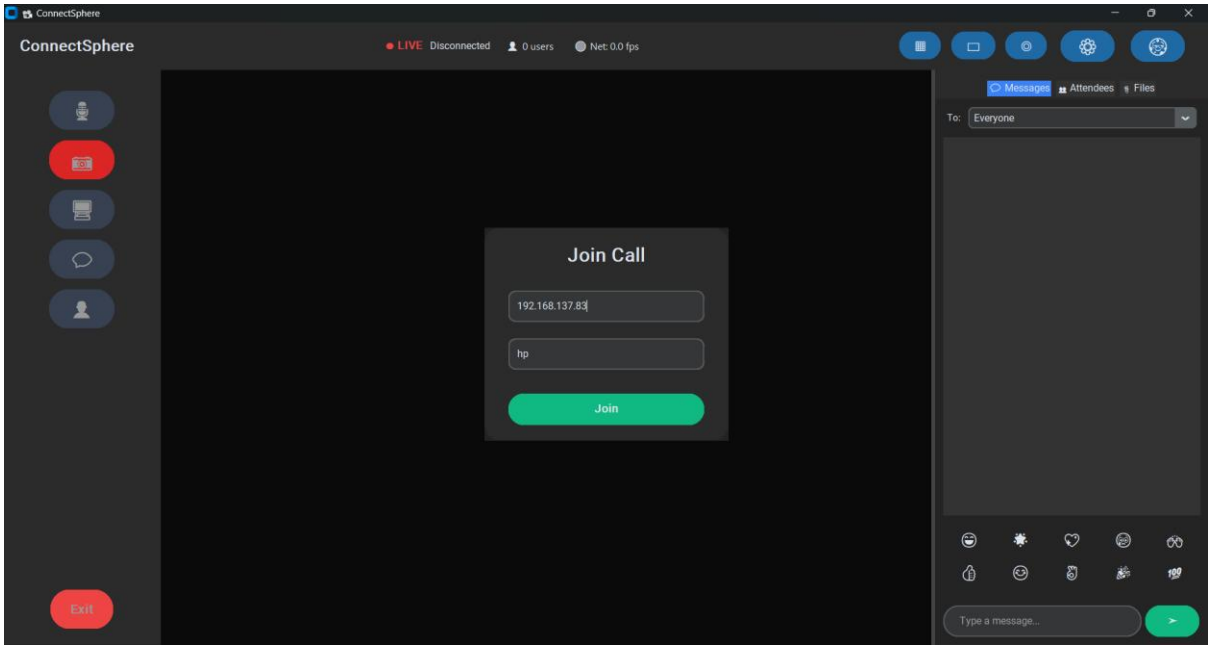
    - **Layouts:** Use the buttons (▦, ▭, ◎) in the top-right to change the layout between **Grid**, **Speaker** (one large, others small), and **Focus** (one large, no others). Pinning a user automatically switches to "Focus" mode.
- **Keyboard Shortcuts:**
    - **M:** Toggle Mic
    - **V:** Toggle Video
    - **S:** Toggle Screen Share
    - **C:** Toggle Chat Panel
    - **U:** Toggle Users Panel
    - **P:** Pin/Unpin the user your mouse is currently hovering over.
    - **Esc:** Leave the call (shows confirmation).

OUTPUT:

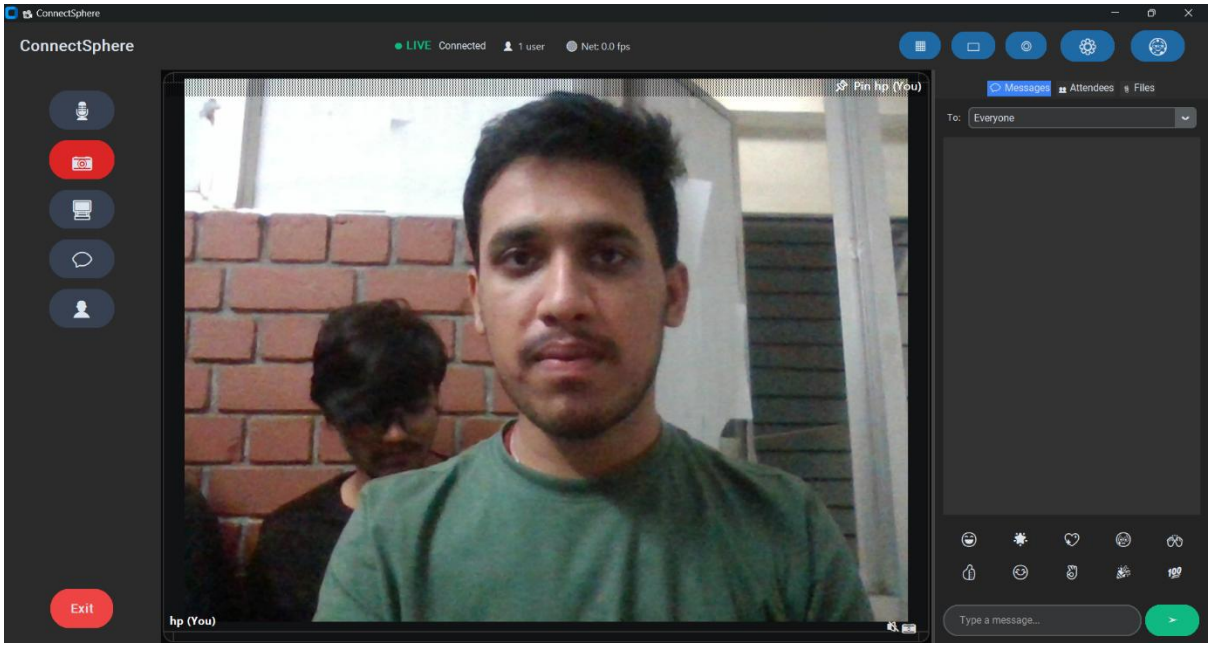
- While “SERVER” Running



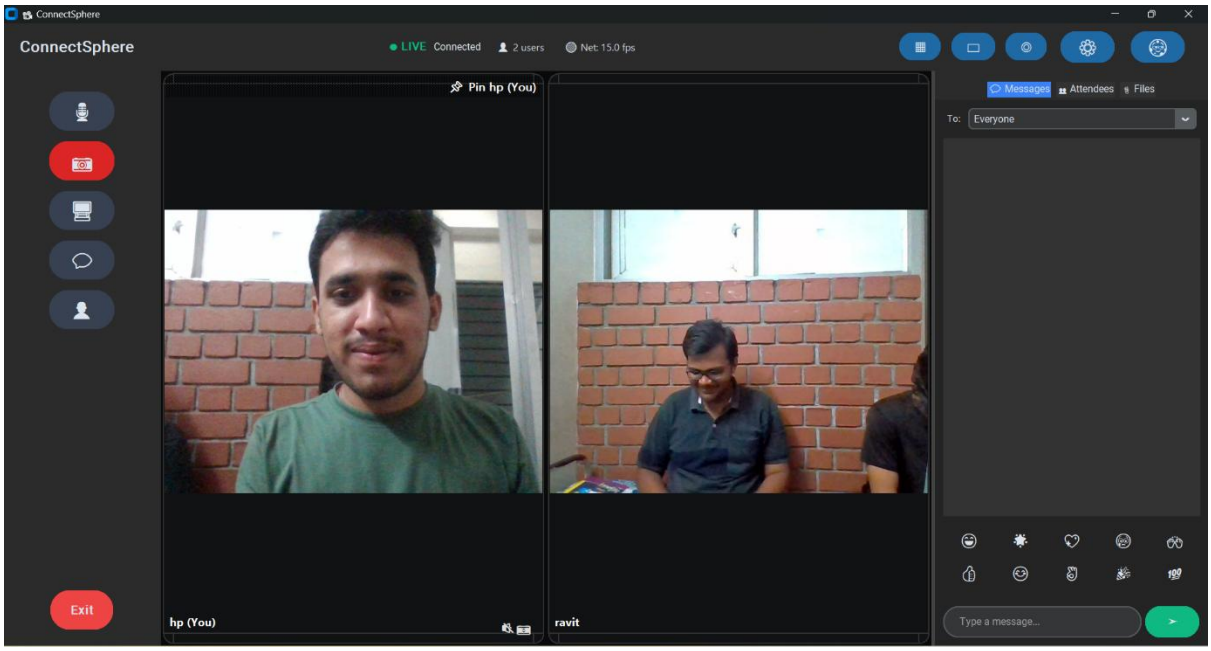
- While running client and enter the ip address and name



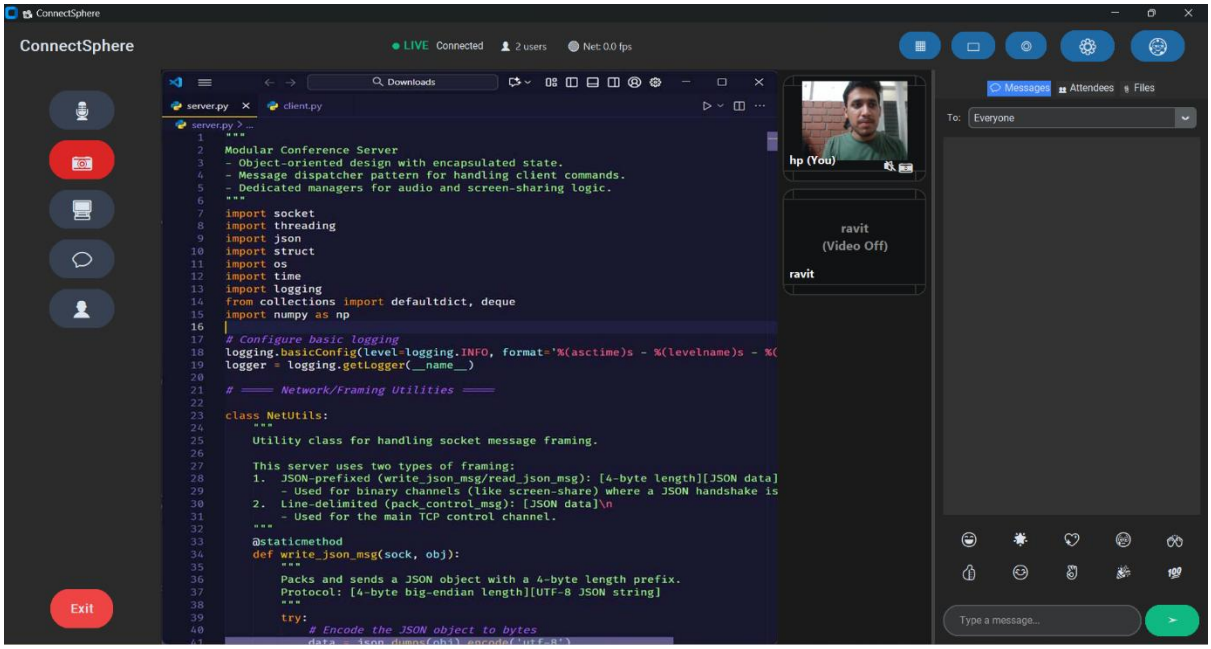
- When I connected with ip address and name



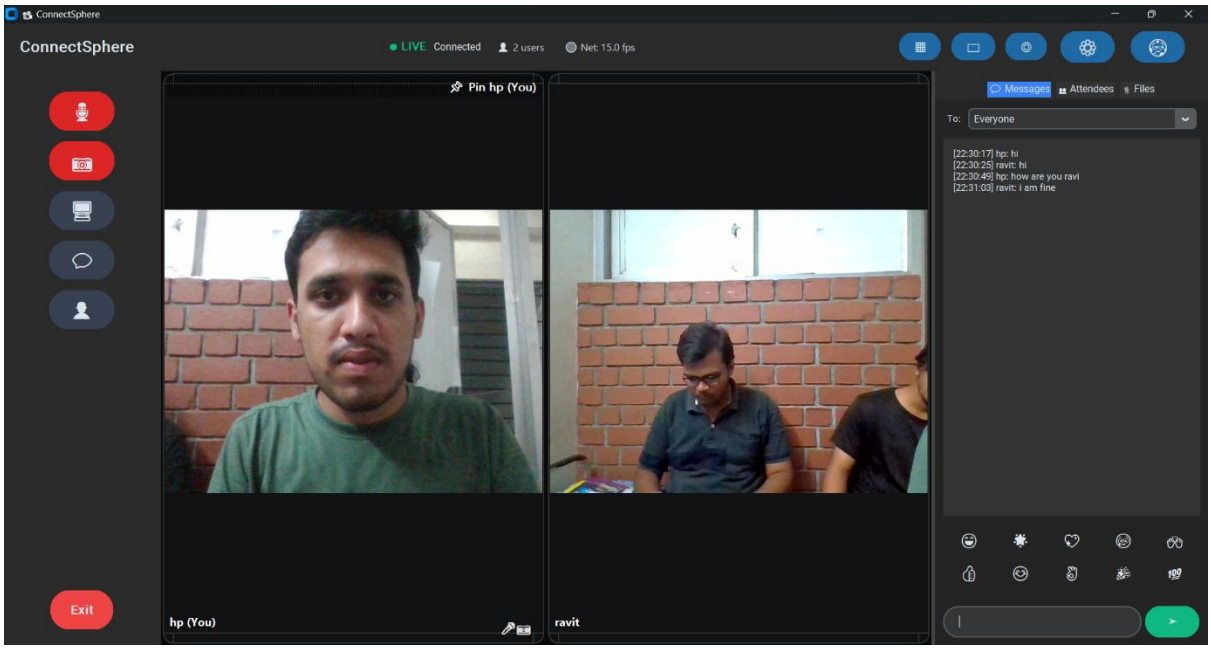
- Output while the other client is connected With Server



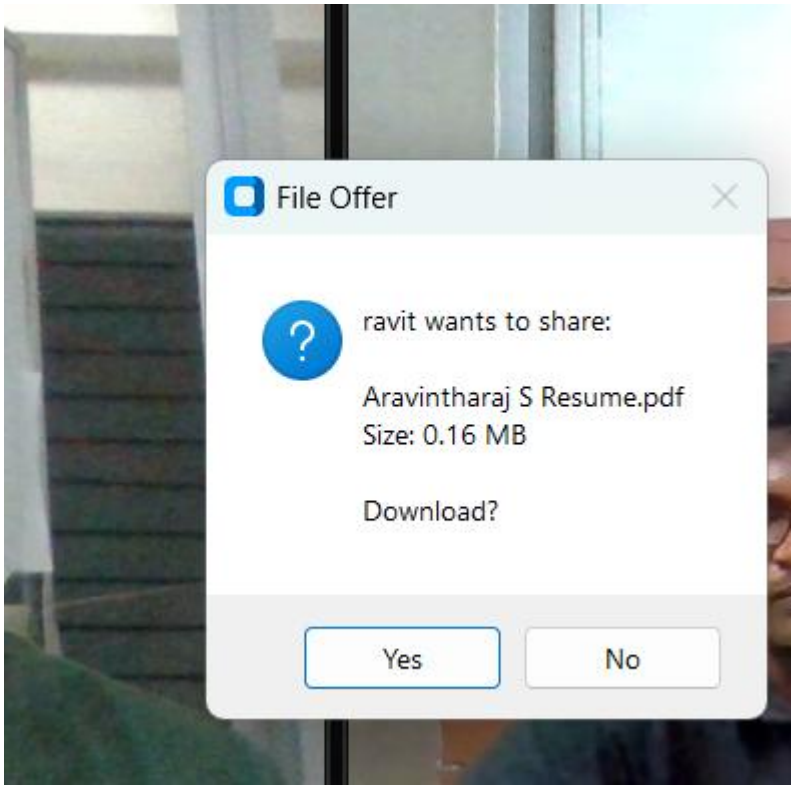
- Output while other client started the screenshare



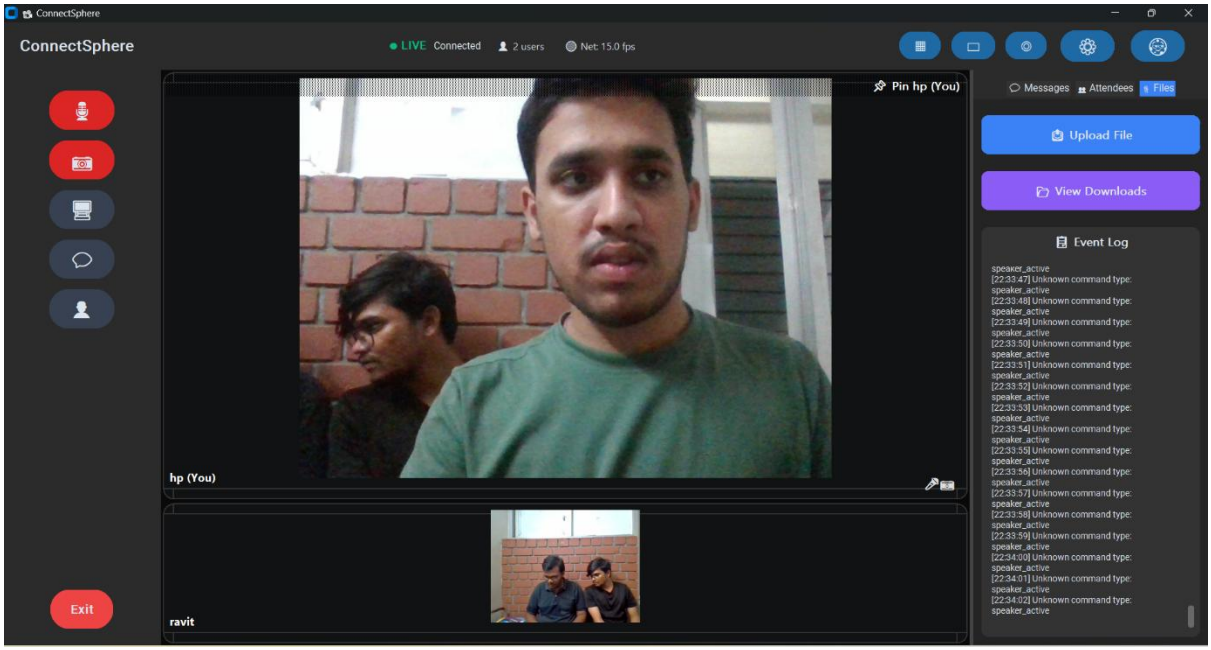
- Output while sending the message between me and another client



- Output while other clients want to share files



- While changing the screen orientation



- Output while opening the device setting where we can change camera index, microphone, speaker’s Output Device