LAN Collaboration Suite

A Computer Networks Project Report

Submitted by:

Srihariram Asuvathraman (CS23B1063) Vijay Pranav (CS23B1073)

Course: Computer Networks Instructor:

Noor Mahammad

Date: November 6, 2025

Department of Computer Science
Your University Name

Academic Year: 2025

Contents

1 Introduction

The LAN Collaboration Suite is a local network communication system that integrates chat, file sharing, screen sharing, and live video/audio conferencing. It is designed using a client-server model, employing TCP for reliable data transmission (e.g., chat, files, control messages) and UDP for real-time streaming of audio and video. The goal of this project is to demonstrate key networking concepts—such as socket programming, concurrency, and protocol design—while building a practical, real-world collaboration tool for LAN environments.

2 System Architecture

The system follows a centralized client-server model:

- The server acts as the hub, managing TCP connections and UDP data forwarding.
- Clients connect to the server, exchange messages, and register their UDP ports for streaming.

TCP ensures reliable, ordered delivery for chat and file data, while UDP supports fast, lightweight transport for video and audio frames.

Threading Model: Both the server and clients use multithreading to handle si- multaneous network operations. The server spawns a thread for each TCP client, while clients run separate threads for listening to TCP and UDP streams, ensuring smooth performance and GUI responsiveness.

3 Communication Protocols

Communication between server and clients is structured using custom message types over TCP and UDP. Each message includes:

- A one-byte message type identifier.
- A four-byte payload length.
- A payload, formatted as JSON (for TCP) or raw bytes (for UDP).

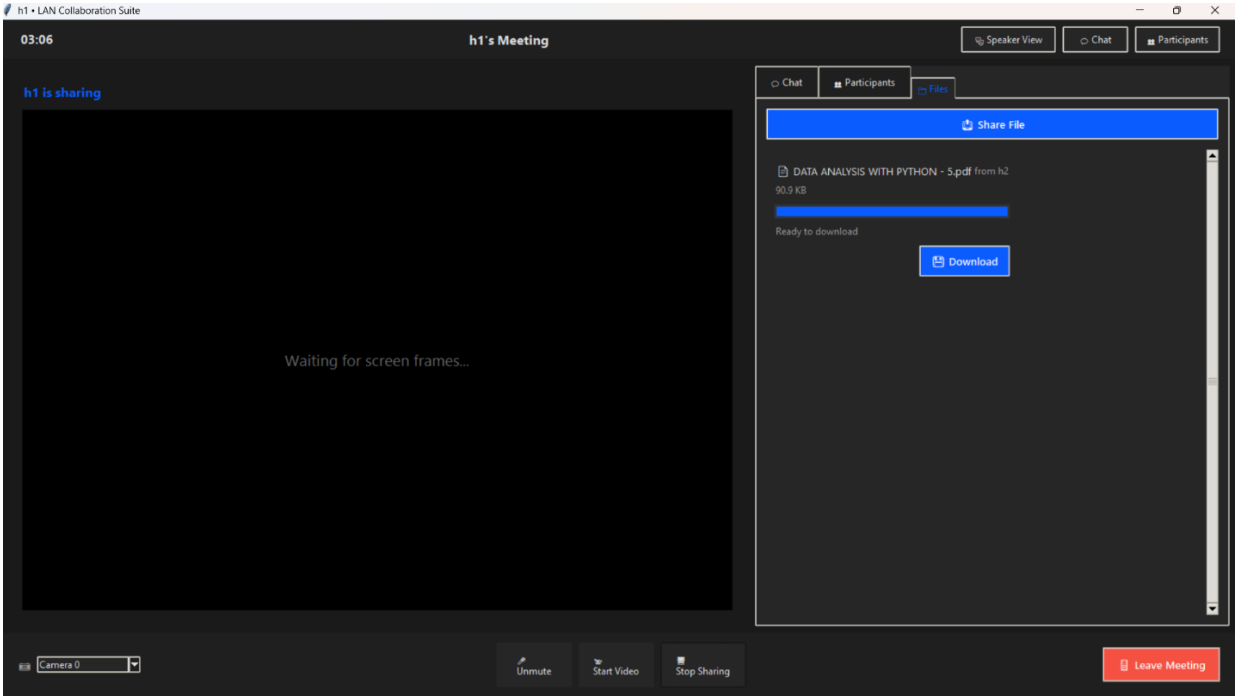
4 Main Features

4.1 Text Chat

Clients exchange messages using TCP for reliability. The GUI displays messages in a chat panel, distinguishing between user messages and system notifications.

4.2 File Sharing

Users can select and share files with all participants. Files are split into chunks, encoded, and transferred over TCP to ensure data integrity. Recipients reconstruct the file and can download it through the GUI.



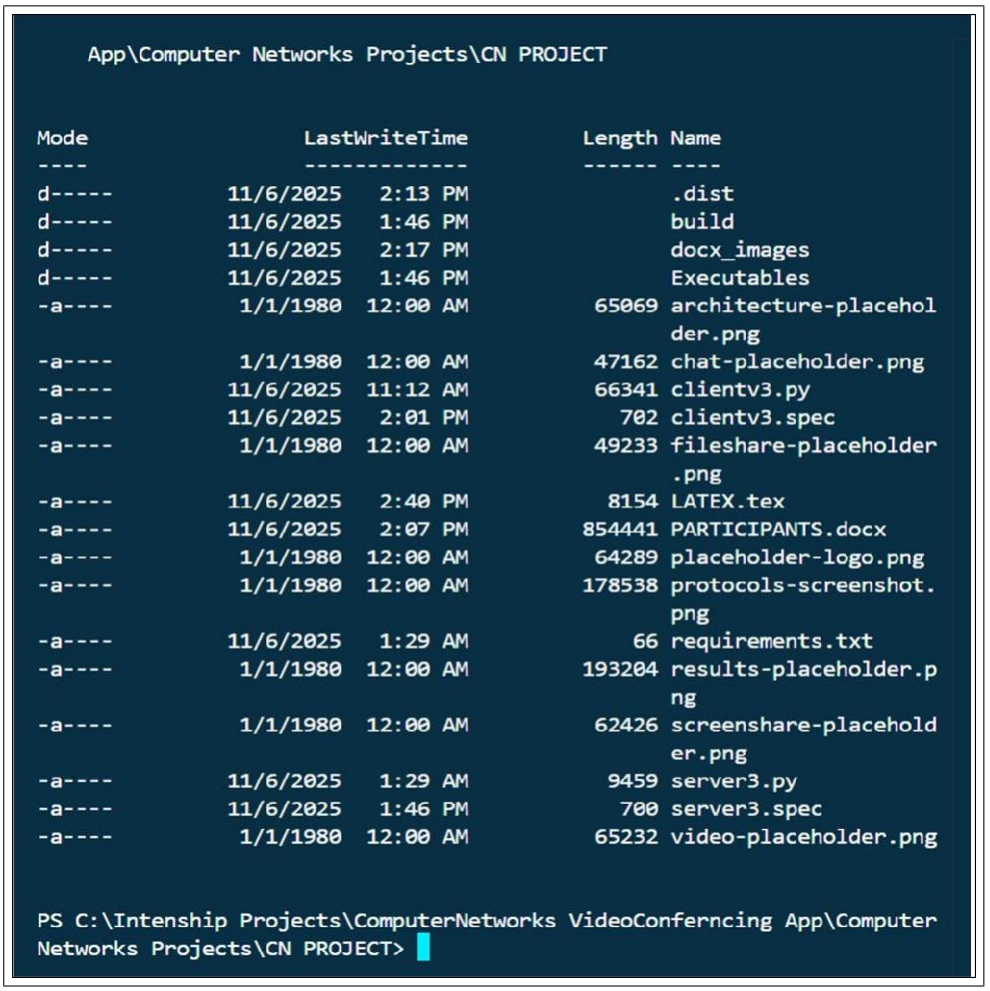
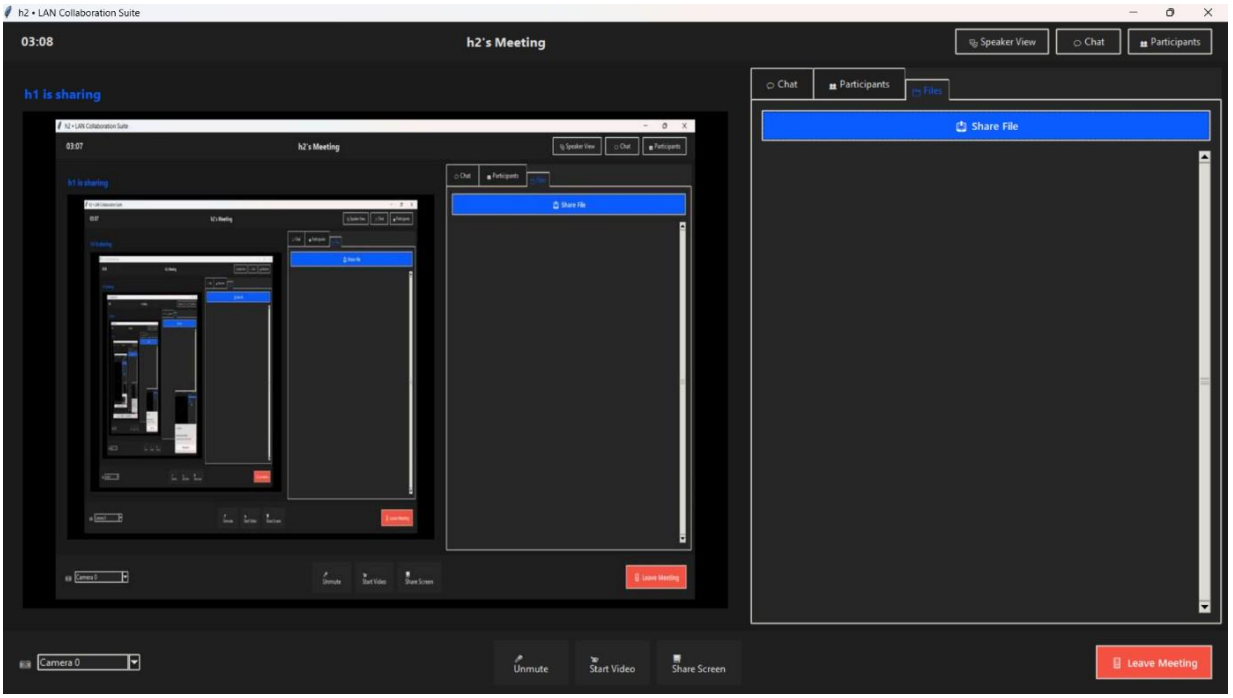


Figure 1: System Architecture Diagram

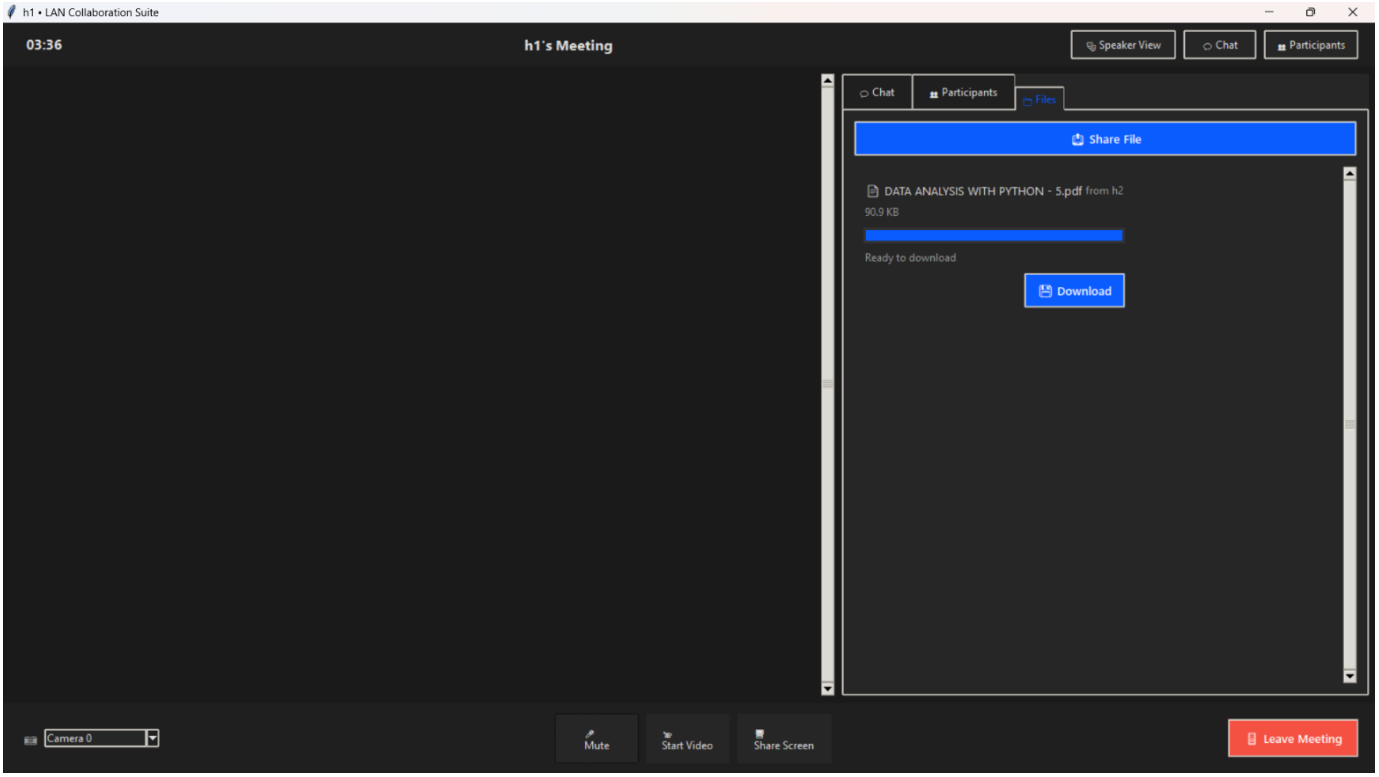
4.3 Screen Sharing

When enabled, the client captures periodic screenshots of the user's desktop and transmits them as compressed JPEG images over TCP. Other clients display the presenter's screen in real time.

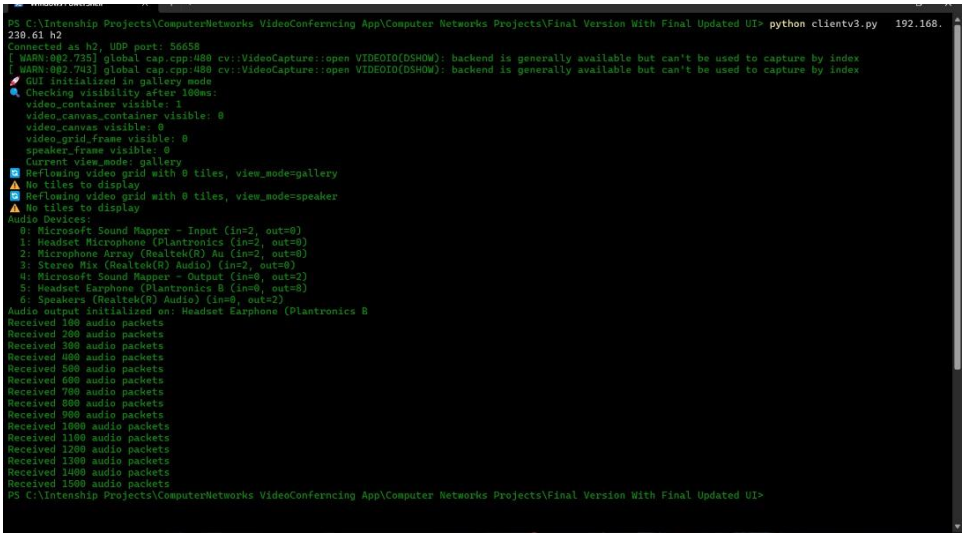


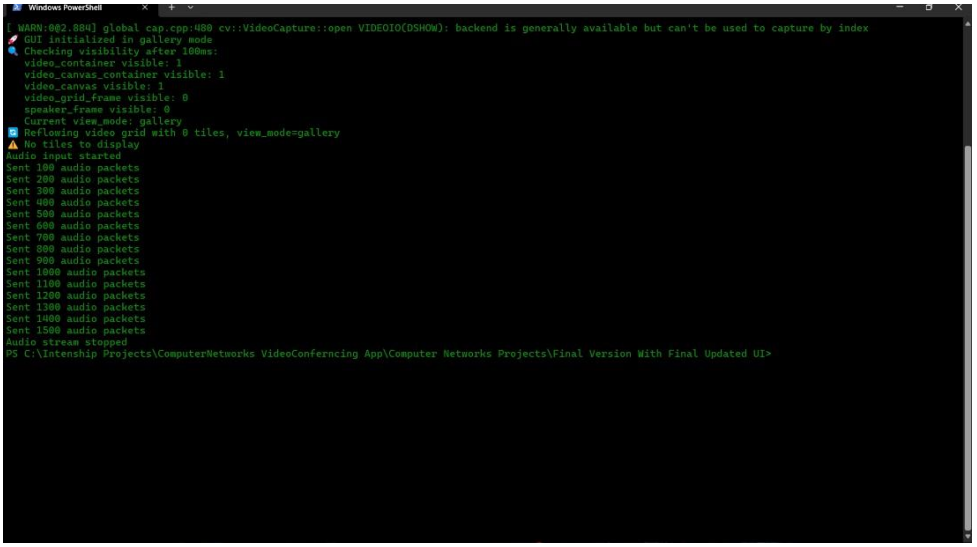
4.4 Audio and Video Streaming

Audio (captured via PyAudio) and video (captured via OpenCV) are transmitted using UDP for low-latency streaming. Each client receives streams from others and plays or displays them live.



Audio Transmissions In Terminals





5 Installation and Setup

Requirements

- Python 3.8 or higher
- Required libraries:opencv-python, Pillow, numpy, pyaudio
- Local network connectivity (server and clients must be on the same LAN)

Code	Name	Transport		Payload format
1	chat	TCP	JSON: {"user": string, "msg": string}	Text chat message broadcast to all clients.
2	file_notify	TCP	JSON: {"user": string, "filename": string, "size": int}	Announces a pending file transfer (name + size).
3	file_request	TCP	JSON: {"user": string, "filename": string}	(Optional) Request or acceptance for a file.
4	file_chunk	TCP	JSON: {"filename": string, "chunk_id": int, "data": base64}	Base64-encoded chunk of file bytes; recipients reassemble by chunk id.
5	screen_start	TCP	JSON: {"user": string}	Notifies others that the user started screen sharing.
6	screen_image	TCP	JSON: {"user": string, "image": base64, "seq": int}	Compressed JPEG image frame (base64). seq helps ordering.
7	screen_stop	TCP	JSON: {"user": string}	Notifies end of screen sharing.
8	user_join	TCP	JSON: {"user": string}	Server notification: a new user connected.
9	user_leave	TCP	JSON: {"user": string}	Server notification: a user disconnected.
10	video_stream	UDP	Raw bytes (framed): euser-name len(1)eeusernameeeejpeg bytes..f.reames sent over UDP; or: header + raw JPEG	Low-latency webcam header contains sender ID/username length + username (so receivers know source).
11	audio_stream	UDP	Raw bytes (framed): euser-name len(1)eeusernameeeepcm bytes..c.ompressed) sent over UDP for low latency.	Audio PCM frames (or
12	udp_register	TCP	JSON: {"user": string, "udp_port": int}	Client tells server its UDP receiving port so server can forward UDP streams.

Table 1: Corrected list of message ("lex") codes, transport, payload format and notes. Each TCP message uses framing: 1 byte type + 4- byte payload length (network/big- endian) + payload bytes. UDP messages are sent in single UDP datagrams and use a small header (e.g. username length) when necessary.

Installation Steps

1. Install dependencies:

```
pip install -r requirements.txt
```

2. Start the server:

```
python
server3.py
```

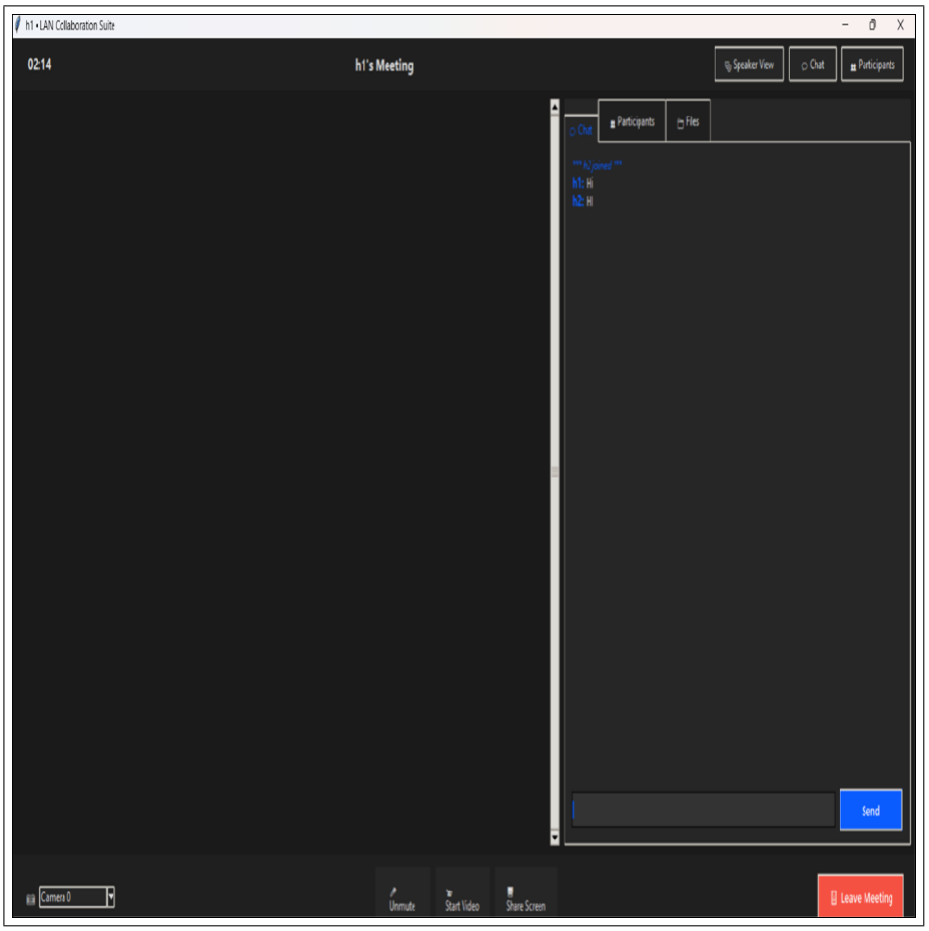


Figure 2: Chat interface

3. Run the client on any LAN device:

   python clientv3.py <server_ip> <username>

6 User Guide

- Start Meeting: Launch the client and connect to the server. Your name appears in the participant list.
- Chat: Type messages and press Enter.
- File Sharing: Click "Share File" to upload files to all participants.
- Video and Audio: Enable your camera and microphone using the toolbar buttons.
- Screen Sharing: Click "Share Screen" to present your display to others.
- End Session: Click "Leave Meeting" to disconnect safely.

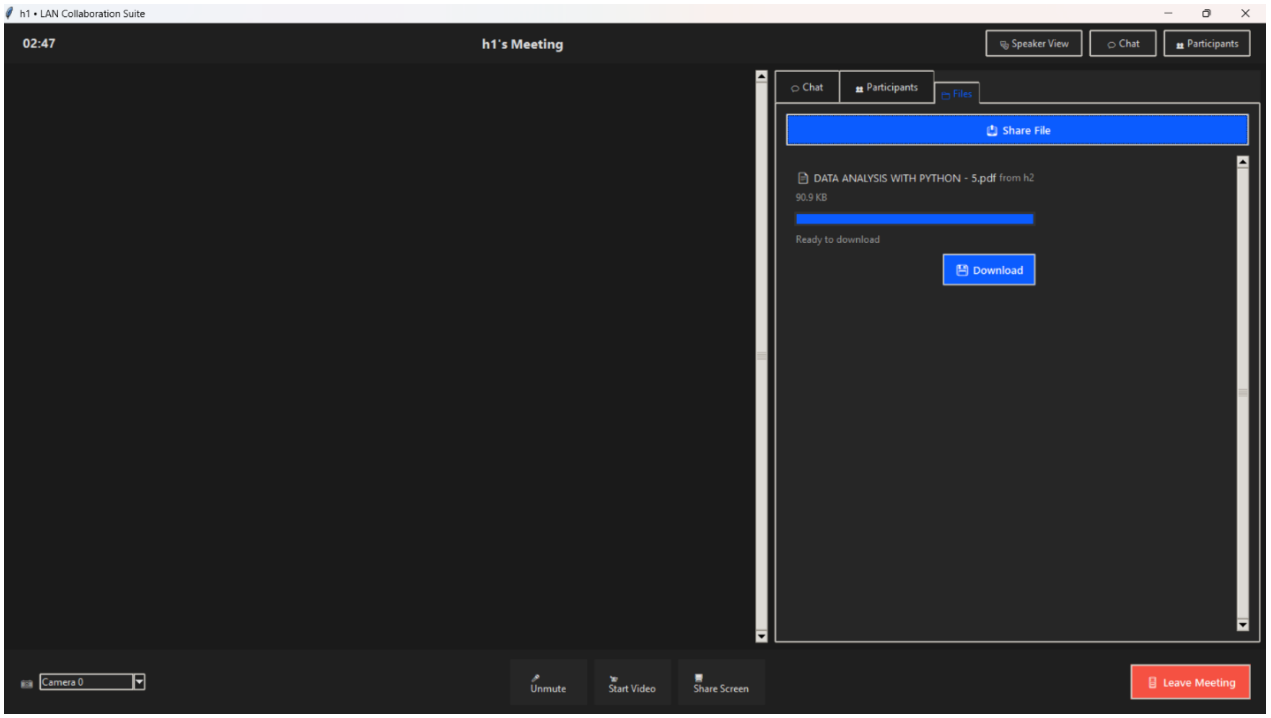


Figure 3: File sharing in progress

7 Testing and Results

The system was tested across multiple machines connected via the same local network. The following results were observed:

- Low-latency video and audio transmission over UDP.
- Reliable file transfer and chat delivery over TCP.
- Screen sharing worked smoothly with moderate frame rates.
- The interface remained responsive due to multithreading.

8 Future Enhancements

- Add user authentication and encryption for secure communication.
- Implement bandwidth optimization for larger meetings.
- Add private chat rooms and selective video streams.
- Extend compatibility to mobile devices.

Further Detailed Explanation of the Code and Code Snipptes

Project Report — LAN Collaboration Suite

High-level summary

- **Purpose:** simple LAN-based collaboration app with chat, file sharing, screen sharing, live video (camera) and audio streaming.
- **Architecture:** single TCP control channel (reliable) for signalling, chat, files and screen images; UDP channel for low-latency real-time media (camera frames and microphone audio). The server routes/broadcasts TCP messages and rewraps UDP media with sender identification for distribution to other clients.

Part A — server3.py (server)

File-level constants & types:

```
from enum import Enum
class MessageType(Enum):
    CHAT = 1
    FILE_NOTIFY = 2
    FILE_REQUEST = 3
    FILE_CHUNK = 4
    SCREEN_START = 5
    SCREEN_IMAGE = 6
    SCREEN_STOP = 7
    USER_JOIN = 8
    USER_LEAVE = 9
    VIDEO_STREAM = 10
    AUDIO_STREAM = 11
    UDP_REGISTER = 12  # NEW: Client registers UDP port
```

Used for message type IDs on the wire (1 byte).

server3

Core constants:

```
HOST = '0.0.0.0'
TCP_PORT = 5000
UDP_PORT = 5001
BUFFER_SIZE = 65536
```

Large BUFFER_SIZE to accommodate images.

Server.__init__(self)

Key code (summary):

```
self.tcp_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
self.udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
self.tcp_socket.bind((HOST, TCP_PORT))
self.udp_socket.bind((HOST, UDP_PORT))
self.tcp_socket.listen(5)
self.clients = {}
self.clients_lock = threading.Lock()
```

What it does

- Creates and binds TCP (control) and UDP (media) sockets.

- Prepares self.clients mapping to track connected clients: { username: {'tcp': sock, 'udp_addr': (ip,port), 'tcp_addr': addr} }.
- Uses a lock for thread-safe access to self.clients.
- Begins listening for TCP connections.

Side effects

- Binds to ports and blocks those ports on the machine until server process stops.

Notes

- SO_REUSEADDR is set to allow quick restarts.

Server.start(self)

Key behavior:

```
threading.Thread(target=self.handle_tcp_connections,
    daemon=True).start()
threading.Thread(target=self.handle_udp_packets, daemon=True).start()
# main loop waits (KeyboardInterrupt stops server)
```

What it does

- Spawns two daemon threads:
  - TCP accept / per-client handling thread pool (handle_tcp_connections)
  - UDP packet handling thread (handle_udp_packets)
- Keeps the main thread alive so the server runs until Ctrl+C.

Notes

- Threads are daemonized so process exits if main thread ends.

Server.handle_tcp_connections(self)

Behavior:

```
while True:
    client_sock, addr = self.tcp_socket.accept()
    threading.Thread(target=self.handle_client_tcp, args=(client_sock,
      addr), daemon=True).start()
```

What it does

- Accepts incoming TCP connections, spawns a handler thread per client.

Edge cases

- Accept errors are printed; connection acceptance continues.

Server.handle_client_tcp(self, client_sock, addr)

This is the per-client TCP loop. Key steps:

1. **Receive username (first message):**
2. username = client_sock.recv(1024).decode().strip()
   - If missing, connection closed.
3. **Make username unique:**
   - If username already in self.clients, append _1, _2, etc., until unique. Then insert into self.clients with 'udp_addr': None.

server3

4. **Send confirmation back:**
5. client_sock.send(username.encode())
   - Client replaces its chosen username with the confirmed username.
6. **Broadcast USER_JOIN:**
7. self.broadcast_tcp(MessageType.USER_JOIN, {"user": username}, exclude_username=username)
8. **Main message loop (framed messages):**

- Uses the same TCP framing as client: 1 byte type + 4 byte big-endian length + JSON payload.
- Reads data into buffer, and processes complete messages; calls self.handle_tcp_message(msg_type, payload, username).

9. On disconnect, call handle_disconnect.

Important details

- BUFFER_SIZE large enough to hold whole messages.
- Framing ensures multiple messages can be buffered or partial messages handled.

Server.handle_tcp_message(self, msg_type, payload, username)

Dispatch logic (selected branches):

- **CHAT:**
- payload['user'] = username
- self.broadcast_tcp(msg_type, payload)
  - Adds the username and broadcasts the chat JSON to all clients.
- **UDP_REGISTER:**
- udp_port = payload.get('port')
- self.clients[username]['udp_addr'] = (client_ip, udp_port)
  - Stores client's UDP address (IP from TCP addr; port supplied by client) so server can send UDP to that client later.

server3

- **FILE_NOTIFY / FILE_CHUNK / SCREEN_START / SCREEN_IMAGE / SCREEN_STOP:**
  - Attaches 'user' = username and broadcasts over TCP to other clients. For file chunks, server blindly relays the chunk messages.

What it implies

- Server acts as relay / broadcaster for signalling and non-real-time data. It does no heavy processing of files/screens beyond retransmit.

Server.handle_udp_packets(self)

Key code (simplified):

```
data, addr = self.udp_socket.recvfrom(BUFFER_SIZE)
msg_type = MessageType(data[0])
length = struct.unpack('!I', data[1:5])[0]
payload = data[5:5+length]

# Identify sender by matching addr to a stored udp_addr
for username, info in self.clients.items():
    if info['udp_addr'] == addr:
        sender_username = username
        break

# For VIDEO_STREAM/AUDIO_STREAM: prepend username and rebroadcast
    to other clients
username_bytes = sender_username.encode()
new_payload = struct.pack('!I', len(username_bytes)) + username_bytes +
    payload
new_data = struct.pack('!BI', msg_type.value, len(new_payload)) +
    new_payload

for each other client with info['udp_addr']:
    udp_socket.sendto(new_data, info['udp_addr'])
```

Explanation

- The server receives raw UDP packets from clients (camera frames or audio).
- It looks up which username corresponds to the sender UDP (ip,port) pair.
- It constructs a new UDP payload that **prefixes sender username length + username** so receivers can know who sent it, then forwards to all other clients' recorded UDP addresses.

server3

Important

- The server **does not** try to decode or modify media payloads; it forwards them verbatim (with username wrapper).
- If the client hasn't registered its UDP port (via UDP_REGISTER), the server will not know how to map incoming UDP packets => will ignore them. So client must register its UDP port after TCP connect (client does this).

Server.broadcast_tcp(self, msg_type, payload, exclude_username=None) and pack_message

- pack_message creates the 1+4 header plus JSON payload; broadcast_tcp iterates self.clients and sends to their TCP sockets (except optional excluded user).
- If a TCP send fails, it prints an error but continues.

Wire format (TCP):

[1 byte msg_type][4 bytes big-endian length][JSON payload bytes]

This format is used by both server and client.

Server.handle_disconnect(self, username)

- Removes client entry from self.clients, closes TCP socket, prints disconnect, and broadcasts USER_LEAVE.

Part B — clientv3.py (client)

Top-level imports and feature flags:

- Uses tkinter for GUI, cv2 for camera capture, pyaudio for audio, PIL for screen capture and image display, numpy for decoding frames. Detects OS to pick the best OpenCV backend. Attempts to import VirtualCamera as fallback.

clientv3

Key constants:

```
TCP_PORT = 5000
UDP_PORT = 5001
BUFFER_SIZE = 65536
CHUNK_SIZE = 1024
```

Matches server.

Helper functions

list_audio_devices(p) & get_default_output_device_index(p)

- list_audio_devices(p): prints host API devices for debugging (useful if audio not playing).

- get_default_output_device_index(p): returns index if default output exists, otherwise None.
- Used to pick correct audio output device when playing received audio.

Client.__init__(self, server_ip, username)

Key sequence (abbreviated):

```
self.tcp_socket = socket.socket(...)
self.tcp_socket.connect((server_ip, TCP_PORT))
self.tcp_socket.send(username.encode())
self.username = self.tcp_socket.recv(1024).decode()

self.udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
self.udp_socket.bind(('', 0))
udp_port = self.udp_socket.getsockname()[1]
self.send_tcp(MessageType.UDP_REGISTER, {'port': udp_port})

self.gui = GUI(self)

threading.Thread(target=self.handle_tcp_messages, daemon=True).start()
threading.Thread(target=self.handle_udp_receives, daemon=True).start()

self.gui.root.protocol("WM_DELETE_WINDOW", self.on_closing)
self.gui.root.mainloop()
```

Explanation, step by step

1. Open TCP connection and send the desired username as a raw initial message. Server may modify to ensure uniqueness; client receives confirmed username and replaces self.username.
   clientv3
2. Create and bind a UDP socket to an ephemeral local port; register that port with server by sending UDP_REGISTER over TCP so the server knows where to send UDP packets back.
   clientv3
3. Build GUI (GUI(self)) before starting network threads — GUI instance is used immediately by networking threads for callbacks.
4. Start two background threads:
   - handle_tcp_messages — processes incoming framed TCP JSON messages and forwards them to GUI safely (via root.after).
   - handle_udp_receives — listens to UDP packets (media) and dispatches frames/audio to GUI or audio playback.
5. Start Tk main loop on the main thread.

Side effects

- Blocking call mainloop() keeps process alive. Threads are daemonized so they don't prevent exit.

Client.on_closing(self) & cleanup(self)

Purpose:

- Set self.running = False, call cleanup() to release/close camera, audio streams, and sockets, then destroy GUI.

cleanup():

- Releases OpenCV capture, stops/closes PyAudio streams, terminates PyAudio instance, closes TCP & UDP sockets. Exceptions are caught and ignored.

Note

- Threads that are blocking in recv() will see socket closure and exit when they receive an error.

TCP receive loop — Client.handle_tcp_messages(self)

Key code pattern:

```
buffer = b''
while self.running:
    data = self.tcp_socket.recv(BUFFER_SIZE)
    buffer += data
    while len(buffer) >= 5:
        msg_type_byte = buffer[0]
        length = struct.unpack('!I', buffer[1:5])[0]
        if len(buffer) < 5 + length: break
        msg_type = MessageType(msg_type_byte)
        payload_bytes = buffer[5:5+length]
        buffer = buffer[5+length:]
        payload = json.loads(payload_bytes.decode())
        self.gui.root.after(0, self.gui.handle_message, msg_type, payload)
```

Explanation

- Implements TCP framing matching the server: 1 byte type + 4 byte length + JSON bytes.
- Buffers data because recv() may return partial messages or multiples.
- On parsing, uses root.after(0, ...) to schedule GUI updates on the Tk main thread (thread-safe mechanism).

clientv3

Edge cases & robustness

- If JSON decode fails, prints error and continues.
- If recv returns empty (peer closed), loop ends and cleanup occurs.

UDP receive loop — Client.handle_udp_receives(self)

Key steps:

```
data, addr = self.udp_socket.recvfrom(BUFFER_SIZE)
msg_type = MessageType(data[0])
length = struct.unpack('!I', data[1:5])[0]
full_payload = data[5:5+length]

# Extract username: first 4 bytes indicate username length
username_len = struct.unpack('!I', full_payload[:4])[0]
sender_username = full_payload[4:4+username_len].decode()
payload = full_payload[4+username_len:]

if msg_type == MessageType.VIDEO_STREAM:
    self.gui.root.after(0, self.gui.update_video, payload, sender_username)
elif msg_type == MessageType.AUDIO_STREAM:
    self.play_audio(payload)
    self.gui.root.after(0, self.gui.update_speaker, sender_username)
```

Explanation

- Expects UDP payloads to be **already wrapped by server** with the sender username length + username. This is consistent with the server which prepends the username when re-broadcasting UDP.
  server3
- For video frames, dispatch to GUI.update_video on the main thread.
- For audio frames, call play_audio on the client thread (audio playback need not run on GUI thread) and update speaker UI.

Gotcha

- The client's own send_udp() does not add username — the server must do so; if server does not wrap, client will fail to parse. This server does implement wrapping.

Client.send_tcp(self, msg_type, payload) and pack_message

- pack_message:
- payload_bytes = json.dumps(payload).encode()
- return struct.pack('!BI', msg_type.value, length) + payload_bytes
- send_tcp uses tcp_socket.send.

Wire format matches server expectation.

Client.send_udp(self, msg_type, payload) and pack_udp_message

- pack_udp_message builds struct.pack('!BI', msg_type.value, length) + payload.
- Sends to self.udp_socket.sendto(data, (server_ip, UDP_PORT)).

Important design detail

- Client sends raw media bytes (no username); server tags them and forwards.

Video capture & streaming

Client.start_video_stream(self, device_index=None)

Key code:

- Chooses backend depending on OS:
  - Windows → cv2.CAP_DSHOW
  - Linux → cv2.CAP_V4L2
- If fail, tries default backend; if still fails and VirtualCamera exists, uses it. Otherwise shows GUI warning.
- Tests reading a frame; sets capture properties: width=640, height=480, fps=15.
- Spawns video_stream_loop thread.

clientv3

Edge cases

- If camera opens but fails to read frames, shows warnings and refuses to start.

Client.video_stream_loop(self)

Key logic:

```
while self.running and self.cap and self.cap.isOpened():
    ret, frame = self.cap.read()
    frame = cv2.resize(frame, (320,240))
    _, buffer = cv2.imencode('.jpg', frame, [cv2.IMWRITE_JPEG_QUALITY, 65])
    payload = buffer.tobytes()
    self.send_udp(MessageType.VIDEO_STREAM, payload)
    time.sleep(1/12)  # ~12 FPS
```

What it does

- Continuously captures frames, resizes to 320×240, JPEG encodes at quality 65, sends over UDP to server.
- Uses a short sleep to target ~12 FPS for stability and bandwidth control.
- On repeated read failures (consecutive_failures threshold), stops the video stream and warns user.

Notes

- JPEG encoding compresses frames but still may be expensive on CPU and bandwidth; client doesn't perform additional optimizations or retransmissions.

Client.stop_video_stream(self)

- Releases self.cap and sets to None. GUI updates its own state accordingly.

Audio capture & playback

Client.start_audio_stream(self)

- Initializes pyaudio.PyAudio() if needed.
- Opens input stream with format paInt16, mono, rate 44100, frames per buffer CHUNK_SIZE (1024).
- Spawns audio_stream_loop.

Notes

- 44100 Hz chosen for better quality; increases bandwidth vs 16k. There is no compression (raw PCM sent), which can be bandwidth heavy.

Client.audio_stream_loop(self)

Loop:

```
while self.running and self.input_stream and self.input_stream.is_active():
    data = self.input_stream.read(CHUNK_SIZE,
      exception_on_overflow=False)
    self.send_udp(MessageType.AUDIO_STREAM, data)
    time.sleep(0.002)  # small throttle
```

What it does

- Reads frames from microphone and forwards them over UDP to the server (server will prepend username and broadcast to others).

clientv3

Limitations

- No jitter buffer, no packet loss concealment, no compression (Opus recommended for production).

Client.play_audio(self, audio_bytes)

Behavior:

- Lazily initializes output pyaudio stream with matching format (paInt16, 44100, mono).
- Writes audio_bytes to output stream via output_stream.write(audio_bytes).
- Lists audio devices once for debugging (list_audio_devices) to help choose the correct output device.

clientv3

Notes

- Blocking write; if many packets arrive, playback may block the UDP handling thread briefly. Could be improved with a separate audio thread and jitter buffer.

Screen sharing

Client.share_screen(self) and screen_share_loop(self)

Key code:

```
self.is_sharing = True
self.send_tcp(MessageType.SCREEN_START, {"user": self.username})
threading.Thread(target=self.screen_share_loop, daemon=True).start()
screen_share_loop:
img = ImageGrab.grab()
img = img.resize((800,600), Image.Resampling.LANCZOS)
buffer = io.BytesIO()
img.save(buffer, format='JPEG', quality=60)
img_data = base64.b64encode(buffer.getvalue()).decode()
self.send_tcp(MessageType.SCREEN_IMAGE, {"image": img_data, "user":
    self.username})
time.sleep(0.5)  # ~2 FPS
```

What it does

- Captures the screen using PIL ImageGrab, resizes and JPEG-encodes it, base64 encodes that bytes and sends as JSON over TCP with SCREEN_IMAGE.
- Uses TCP (reliable) for screen images, not UDP—this prioritizes reliable delivery at cost of latency/bandwidth.

Notes & platform caveats

- ImageGrab.grab() works reliably on Windows/macOS; Linux may require X11 and may not work on Wayland without additional setup.
- Base64+JSON has significant overhead; good for low-FPS screen share, not for high-frame interactive screen share.

Chat & File sharing

Client.send_chat(self, message)

- Packs {"msg": message} and sends as MessageType.CHAT over TCP. Server will add 'user' and broadcast.

Client.share_file(self, filepath)

High-level flow:

1. Notify server of filename & size via FILE_NOTIFY TCP JSON: {"filename": filename, "size": filesize}.
2. Break the file into **4096 byte** chunks, base64-encode each chunk, and send each as a FILE_CHUNK TCP message:
3. {"filename": filename, "chunk_id": chunk_id, "data": chunk_data}
4. GUI progress callbacks are scheduled via root.after to update progress bar.

Receiver side (GUI)

- The GUI reconstructs files by collecting chunks in memory (list of (chunk_id, data)), counting received bytes; when total bytes >= declared size, the Download button is enabled.

Important considerations

- File chunks are base64-encoded and sent over TCP JSON — reliable but memory intensive (both base64 expansion and keeping chunks in memory). For large files, use streaming to disk (temp file) and/or send binary on separate channel.

Part C — GUI class (from clientv3.py)

The GUI class is sizable — I'll summarize main functions and behaviour, focusing on how it ties into client actions and received messages. All GUI callbacks are designed to be invoked on main Tk thread (either directly or via root.after from networking threads).

GUI.__init__(self, client)

- Builds a Zoom-like UI: header, main video area (speaker + gallery), sidebar with tabs (Chat, Participants, Files), and bottom control bar with camera/mic/share/leave buttons.
- Initializes internal state dicts: video_tiles, participants, file_chunks, incoming_files_meta, etc.
- Calls self.add_participant(self.client.username, is_self=True) to add local user to participants list.

Design detail

- GUI keeps a mapping video_tiles[username] to hold per-user video widgets and images; reflow_video_grid() arranges tiles in a grid.

---

build_layout(self) & setup_styles(self)

- build_layout constructs all Tk widgets (Canvas, frames, text box for chat, treeview for participants, file list area, control buttons).
- setup_styles defines a dark theme and styles for ttk elements (colors, fonts, hover states).

Why important

- Provides a polished UI; interaction handling (button commands) are wired here (e.g., self.video_btn.config(command=self.toggle_video)).

Interaction functions (UI → Client)

- toggle_video(self): asks permission (one-time dialog), calls client.start_video_stream() with selected camera index (if available). Updates local buttons and participant state.
  clientv3
- toggle_mic(self): asks permission, calls client.start_audio_stream() or stops audio. Updates UI.
- toggle_share(self): asks permission, calls client.share_screen() or stops sharing.
- share_file_cb(self): file dialog → client.share_file(filepath) executed in a

background thread.

- send_chat_cb(self): reads chat entry and calls client.send_chat(message); server will broadcast back to clients including self (so GUI does not echo immediately).

Message handling — handle_message(self, msg_type, payload)
This is the central dispatcher used by Client.handle_tcp_messages:

- MessageType.CHAT → add_chat_message(payload['user'], payload['msg'])
- SCREEN_START → set_presenter(presenter) + chat system message
- SCREEN_IMAGE → update_presenter_image(payload.get('image'), payload.get('user'))
- SCREEN_STOP → clear_presenter
- FILE_NOTIFY → create file row + allocate structures for incoming file
- FILE_CHUNK → decode base64 chunk, append to file_chunks[filename], update received bytes and progress
- USER_JOIN / USER_LEAVE → add_participant / remove_participant

Thread-safety

- All calls into GUI originate from root.after in the client network threads so they execute on main Tk thread.

Video rendering — update_video(self, frame_bytes, sender_username)
Key steps:

1. Convert bytes into NumPy array and decode JPEG:
2. nparr = np.frombuffer(frame_bytes, np.uint8)
3. frame = cv2.imdecode(nparr, cv2.IMREAD_COLOR)
4. frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
5. img = Image.fromarray(frame)
6. img = img.resize((320,240), Image.Resampling.LANCZOS)
7. photo = ImageTk.PhotoImage(image=img)
8. If sender_username has no tile, create one via _create_video_tile.
9. Update tile label with label.config(image=photo, text='', bg='#000000') and keep a reference label.image = photo so Tk doesn't garbage collect the image.

Subtleties

- Must keep reference to PhotoImage (the code does this).
- Layout (reflow_video_grid) is called when new tiles are created or removed.

File UI behavior

- create_file_row creates visual widgets for incoming file with progress bar and disabled Download button.
- update_file_row_progress updates progress bar and when fully received enables Download.
- download_file sorts chunks by chunk_id, concatenates, then shows Save As dialog to write to disk.

Memory concern

- Incoming file chunks are stored in memory in self.file_chunks[filename] — for large files, this can spike memory usage. Consider streaming to disk (temp file) as chunks arrive.

Participant list functions

- add_participant(self, username, is_self=False) inserts username into Treeview and updates counts.
- remove_participant(self, username) deletes participant and destroys video tile if present.
- update_participant_status(self, username, field, value) updates mic/camera icons.

Part D — Protocol summaries (wire formats)

TCP (control channel)

- All TCP control messages use framing:
  [1 byte: msg_type][4 bytes: big-endian unsigned int length][length bytes: JSON payload (utf-8)]
  Examples: CHAT, FILE_NOTIFY, FILE_CHUNK, SCREEN_IMAGE (base64 image string), SCREEN_START, USER_JOIN, etc.

UDP (real-time media channel)

- Client sends (to server):
  [1 byte msg_type][4 bytes length][payload bytes]
  where payload bytes are raw JPEG frame bytes (video) or raw PCM audio bytes (audio). The **client does not include the username** when sending.
- Server receives the above, then *forwards* to other clients **after** prepending [4 bytes username_len][username bytes] to payload, and repacking header:
  [1 byte msg_type][4 bytes length_of_username+payload][4 bytes username_len][username bytes][payload bytes]
  This allows receiving clients to extract the sender username and the actual media payload.

Part E — End-to-end example (Start camera → other clients see frames)

1. Client A calls Client.start_video_stream() which begins video_stream_loop that sends JPEG frames via UDP to server.
2. Server handle_udp_packets sees packet from A's (ip,port), finds sender_username='A', wraps payload with username length & username, repacks and sendto to other clients B, C.
3. Client B handle_udp_receives receives UDP, extracts sender_username and payload and calls GUI.update_video(payload, sender_username) on main thread. GUI decodes JPEG and updates B's display tile for A.

Part F — Important notes, limitations & suggested improvements

Security

- No encryption or authentication (plaintext TCP/UDP). For production, add TLS for TCP and DTLS for UDP or use WebRTC.

server3

File transfer

- Currently base64 in TCP JSON and stored in memory on receivers. For large files:
  - Stream file chunks directly to a temporary file on disk on the receiver instead of storing in RAM.
  - Consider using a binary TCP stream (no base64) or chunked file transfer protocol.

Audio/Video

- Audio is raw PCM at 44.1 kHz mono; heavy on bandwidth. Use Opus (codec) or lower sample rate. Add jitter buffer and small sequence numbers/timestamps.
- Video uses JPEG frames over UDP (simple). Consider RTP or WebRTC for adaptive bitrate, NAT traversal, and better resiliency.

Robustness

- When closing, background threads may be blocked in recv(); closing sockets triggers exceptions and threads exit — current code handles this but monitor logs for socket errors on shutdown.

clientv3

Cross-platform

- ImageGrab.grab() (screen capture) may not work on Linux Wayland without extra tools; camera backends differ across OS — code detects OS and selects appropriate backend but test on target OS.

clientv3

Performance

- JPEG compression quality 65 is reasonable for balance; you can tune resolution, fps and JPEG quality to manage CPU/bandwidth tradeoffs.

Conclusion

The LAN Collaboration Suite demonstrates a fully functional real-time communica- tion system that integrates key networking concepts such as socket programming, con- currency, and media streaming. Through this project, the principles of TCP and UDP communication were applied to create a practical and interactive local collaboration tool