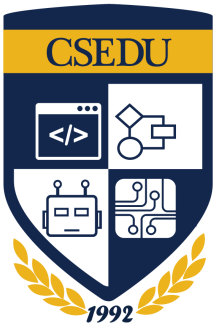

University of Dhaka

Department of Computer Science and Engineering

PROJECT REPORT

CSE 3111: Computer Networking Lab

3rd Year, 1st Semester

Academic Year: 2024-2025

Lets Meet
Real-Time Video Conferencing Application

Submitted By:

Md. Tauseef-Ur-Rahman
Roll: FH-024
Batch: 28

Tamzid Bin Tariq
Roll: FH-048
Batch: 28

Submitted To:

Dr. Ismat Rahman
Associate Professor

Mr. Jargis Ahmed
Lecturer

Mr. Palash Roy
Lecturer

Date of Submission: July 18, 2025

Contents

1 Overview

This project presents the development of a real-time video conferencing application built using Python and PyQt6. The application enables multiple users to participate in video calls with integrated audio communication, text messaging, and file sharing capabilities. The system implements a client-server architecture with separate handling for different types of media streams to optimize performance and user experience.

The application addresses the growing need for reliable, lightweight video conferencing solutions in an era where remote communication has become essential. By leveraging Python's networking capabilities and PyQt6's graphical interface framework, we have created a cross-platform solution that provides essential conferencing features while maintaining simplicity and ease of use.

The system supports multiple video quality options (240p to 900p), real-time audio with noise control, camera and microphone toggle functionality, and a modern responsive user interface. Additionally, it includes advanced features such as real-time chat messaging, file sharing capabilities, and intelligent session management with resource optimization.

2 Motivation

The motivation for this project stems from several key factors in today's digital landscape:

2.1 Rise of Remote Work and Online Education

The increasing trend toward remote work and online education has created an unprecedented demand for reliable video conferencing solutions. Traditional in-person meetings have largely transitioned to virtual platforms, making video conferencing an essential tool for productivity and learning.

2.2 Need for Lightweight and Customizable Solutions

Many existing video conferencing solutions are either too resource-intensive or lack the flexibility needed for specific use cases. There is a clear need for lightweight alternatives that can be easily customized and deployed in various environments without requiring extensive infrastructure.

2.3 Educational and Research Purposes

From an academic perspective, developing a video conferencing application provides an excellent opportunity to apply networking concepts, real-time communication protocols, and multimedia processing techniques in a practical, real-world scenario.

2.4 Privacy and Control Considerations

Building a custom solution allows for complete control over data handling, privacy policies, and security measures, which is increasingly important in today's privacy-conscious environment.

3 Problem Statement

The primary problem addressed by this project is the development of a real-time, multi-user video conferencing system that can handle simultaneous video streaming, audio communication, and text messaging while maintaining low latency and good user experience.

Specifically, the system must solve the following challenges:

- Efficient handling of multiple concurrent video and audio streams
- Real-time synchronization between multiple clients
- Reliable message delivery for text and file sharing
- Resource management to prevent system overload
- Network optimization for different bandwidth conditions
- User-friendly interface that supports essential conferencing features

The solution must be scalable, maintainable, and provide a solid foundation for future enhancements while remaining lightweight and accessible to users with varying technical backgrounds.

4 Design Goals/Objectives

The primary objectives of this video conferencing application are:

4.1 Primary Objectives

- **Real-time Communication**: Achieve low-latency video and audio streaming suitable for interactive conversations
- **Multi-user Support**: Enable multiple participants to join the same conference session simultaneously
- **Cross-platform Compatibility**: Ensure the application works across different operating systems
- **User-friendly Interface**: Provide an intuitive and responsive graphical user interface

4.2 Secondary Objectives

- **Scalability**: Design the system to handle increasing numbers of users without significant performance degradation
- **Reliability**: Reliable data transmission among all users
- **Extensibility**: Create a modular architecture that allows for easy addition of new features
- **Resource Efficiency**: Optimize CPU, memory, and network usage to ensure smooth operation

4.3 Technical Objectives

- Implement proper socket programming for client-server communication
- Apply appropriate networking protocols (TCP for reliable data, UDP for real-time streams)
- Integrate multimedia processing for video and audio handling
- Develop a responsive GUI using modern framework principles

5 Project Features

The video conferencing application includes the following key features:

5.1 Core Features

1. **Video Streaming (240p–900p)**: Support for multiple video quality options to accommodate different network conditions and device capabilities
2. **Audio with Noise Control**: Real-time audio streaming with built-in noise reduction and echo cancellation capabilities
3. **Camera & Microphone Toggle**: Easy-to-use controls for enabling/disabling video and audio input devices
4. **Modern Responsive UI**: Clean, intuitive interface built with PyQt6 that adapts to different screen sizes
5. **Real-time Chat & File Sharing**: Integrated messaging system with support for text messages and file transfers

5.2 Advanced Features

1. **Smart Session & Resource Management**: Intelligent handling of system resources and automatic cleanup of unused connections
2. **Multi-threaded Architecture**: Separate threads for video, audio, and text processing to ensure smooth operation
3. **Data Rate Monitoring**: Real-time tracking and visualization of network usage and performance metrics
4. **Private Chat**: Select specific users to chat with them

6 Block Diagram/Work Flow Diagram

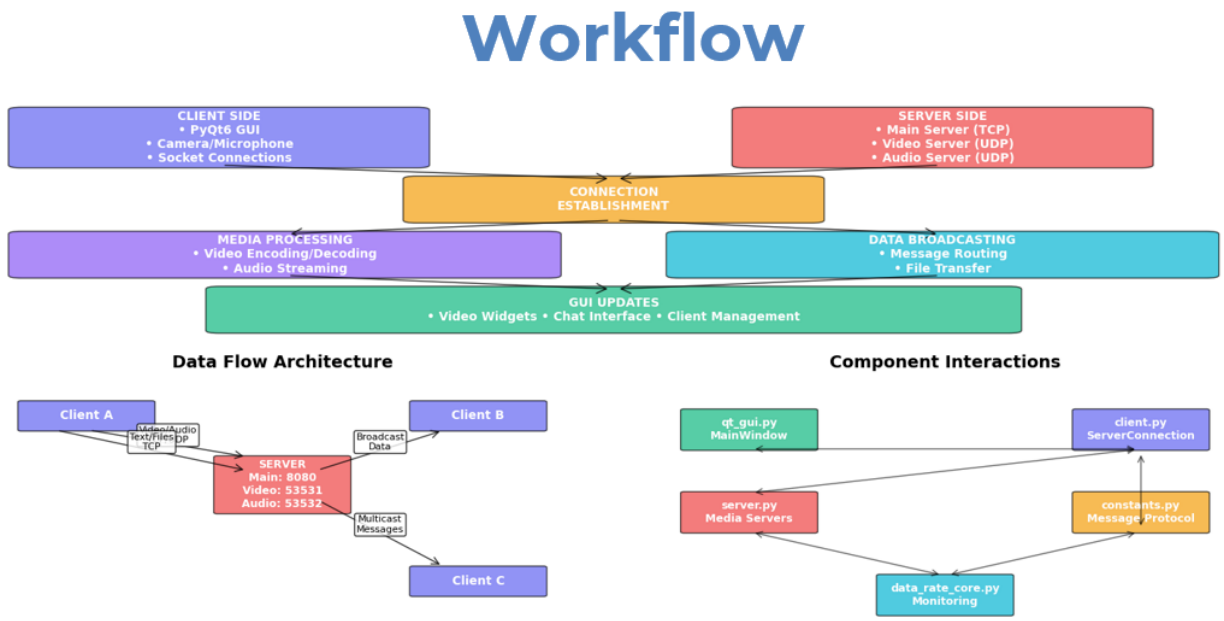

Figure 1: WorkFlow

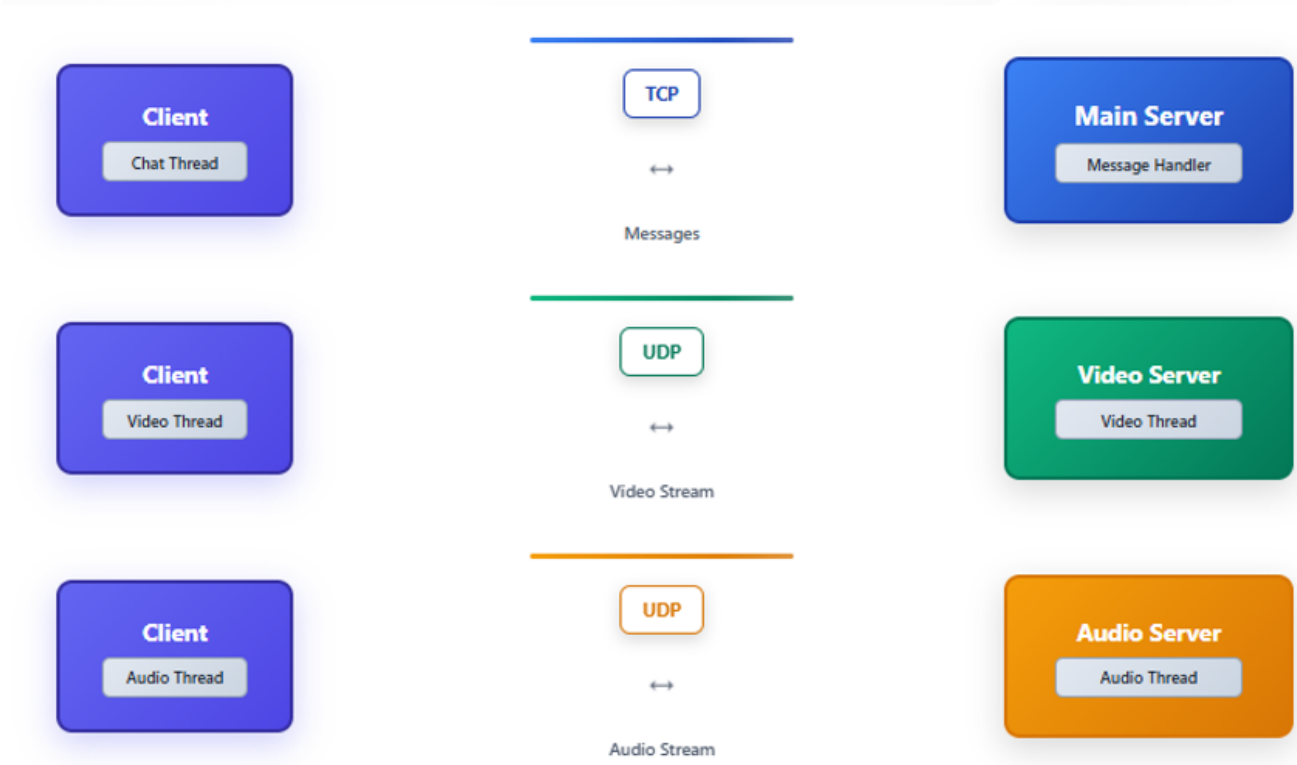

Figure 2: System Architecture Block Diagram

6.1 Workflow Description

The application follows a client-server architecture with the following workflow:

1. **Client Initialization**: Client connects to the main server using TCP
2. **Authentication**: Server validates client credentials and assigns unique session ID
3. **Media Server Registration**: Client registers with video and audio servers using UDP
4. **Stream Broadcasting**: Client begins broadcasting video and audio streams
5. **Data Reception**: Client receives streams from other participants
6. **Message Handling**: Text messages and file transfers are processed through the main server
7. **Graceful Disconnection**: Proper cleanup when clients leave the session

7 Tools & Technologies

7.1 Programming Language

- **Python 3.8+**: Primary development language chosen for its extensive networking libraries and multimedia processing capabilities

7.2 GUI Framework

- **PyQt6**: Modern cross-platform GUI framework providing native look and feel across different operating systems

7.3 Networking Libraries

- **Socket**: Built-in Python library for TCP/UDP socket programming
- **Threading**: Multi-threading support for concurrent operations
- **Pickle**: Object serialization for message passing between clients and servers

7.4 Audio Processing

- **PyAudio**: Real-time audio I/O library for capturing and playing audio streams
- **Wave**: Audio file format handling and processing

7.5 Video Processing

- **OpenCV (cv2)**: Computer vision library for video capture, processing, and streaming
- **NumPy**: Numerical computing library for efficient array operations on video frames

7.6 Additional Tools

- **Matplotlib**: Data visualization for network performance monitoring
- **Tkinter**: Alternative GUI framework for monitoring interfaces

8 Applied Networking Concepts

8.1 TCP – Reliable Communication (Main Server)

The main server utilizes TCP (Transmission Control Protocol) for handling critical communications that require reliability and ordered delivery.

8.1.1 Key Features Applied:

- **Reliable Data Transfer**: Ensures chat messages and file transfers arrive in correct order without errors
- **Flow Control**: Prevents sender from overwhelming receiver by managing data transmission rates
- **Congestion Control**: Dynamically adjusts sending rate to avoid network overload
- **Connection Management**: Proper establishment and termination of client connections

8.1.2 Implementation Details:

Listing 1: TCP Connection Establishment

```
def init_conn(self):
    self.main_socket.connect((IP, MAIN_PORT))
    client.name = self.name
    self.main_socket.send_bytes(self.name.encode())
    conn_status = self.main_socket.recv_bytes().decode()
    if conn_status != OK:
        QMessageBox.critical(None, "Error", conn_status)
        return
    self.connected = True
```

8.2 UDP – Real-Time Streaming (Audio & Video Servers)

UDP (User Datagram Protocol) is employed for audio and video streaming where speed is prioritized over reliability.

8.2.1 Key Features Applied:

- **Low-Latency Transmission**: Minimizes delay for real-time audio/video streaming

- **No Retransmission**: Accepts occasional packet loss to maintain real-time experience
- **Connectionless Communication**: Reduces overhead for continuous data streams
- **Broadcast Capability**: Efficiently distributes streams to multiple clients

8.2.2 Implementation Details:

Listing 2: UDP Media Streaming

```
def media_broadcast_loop(self, conn: socket.socket, media: str):
    while self.connected:
        if media == VIDEO:
            data = client.get_video()
        elif media == AUDIO:
            data = client.get_audio()
        msg = Message(self.name, POST, media, data)
        self.send_msg(conn, msg)
```

8.3 Socket Programming & Threading

The application implements comprehensive socket programming with multi-threading for concurrent operations.

8.3.1 Key Features Applied:

- **Non-blocking I/O**: Prevents interface freezing during network operations
- **Concurrent Processing**: Separate threads for video, audio, and text processing
- **Resource Management**: Proper cleanup and resource deallocation
- **Error Handling**: Robust exception handling for network failures

8.3.2 Implementation Details:

Listing 3: Multi-threaded Connection Handling

```
def start_conn_threads(self):
    self.main_conn_thread = Worker(self.handle_conn, self.
       main_socket, TEXT)
    self.threadpool.start(self.main_conn_thread)

    self.video_conn_thread = Worker(self.handle_conn, self.
       video_socket, VIDEO)
    self.threadpool.start(self.video_conn_thread)

    self.audio_conn_thread = Worker(self.handle_conn, self.
       audio_socket, AUDIO)
    self.threadpool.start(self.audio_conn_thread)
```

8.4 Modular Client-Server Architecture

The system implements a distributed architecture with specialized servers for different data types.

8.4.1 Key Features Applied:

- **Separation of Concerns**: Different servers handle different types of data
- **Scalability**: Independent scaling of video, audio, and text services
- **Fault Tolerance**: Failure in one service doesn't affect others
- **Load Distribution**: Balanced processing across multiple server instances

9 Implementation Details

9.1 Client-Side Implementation

9.1.1 Client Class Structure

The `Client` class encapsulates user information and media handling capabilities:

Listing 4: Client Class Implementation

```
class Client:
    def __init__(self, name: str, current_device=False):
        self.name = name
        self.current_device = current_device
        self.video_frame = None
        self.audio_data = None

        if self.current_device:
            self.camera = Camera()
            self.microphone = Microphone()
        else:
            self.camera = None
            self.microphone = None

        self.camera_enabled = True
        self.microphone_enabled = True
```

9.1.2 Media Capture Algorithm

9.1.3 Server Connection Management

The `ServerConnection` class manages all network communications:

Listing 5: Server Connection Initialization

```
def __init__(self, parent=None):
    super().__init__(parent)
```

Algorithm 1 Video Frame Capture

```
Initialize camera device
while application is running do
    if camera is enabled then
        Capture frame from camera
        Process frame (resize, format conversion)
        Store processed frame
    else
        Set frame to None
    end if
    Sleep for frame rate interval
end while
```

```
    self.threadpool = QThreadPool()

    self.main_socket = socket.socket(socket.AF_INET, socket.
        SOCK_STREAM)
    self.video_socket = socket.socket(socket.AF_INET, socket.
        SOCK_DGRAM)
    self.audio_socket = socket.socket(socket.AF_INET, socket.
        SOCK_DGRAM)

    self.connected = False
    self.recieving_filename = None
```

9.2 Server-Side Implementation

9.2.1 Main Server Architecture

The main server handles client connections and message routing:

Listing 6: Main Server Loop

```
def main_server():
    main_socket = socket.socket(socket.AF_INET, socket.
        SOCK_STREAM)
    main_socket.bind((IP, MAIN_PORT))
    main_socket.listen()

    while True:
        conn, addr = main_socket.accept()
        name = conn.recv_bytes().decode()
        if name in clients:
            conn.send_bytes("Username already taken".encode())
            continue
        conn.send_bytes(OK.encode())
        clients[name] = Client(name, conn, True)

        main_conn_thread = threading.Thread(target=
            handle_main_conn, args=(name,))
```

```
        main_conn_thread.start()
```

9.2.2 Media Server Implementation

Separate servers handle video and audio streams:

Listing 7: Media Server Handler

```
def media_server(media: str, port: int):
    conn = media_conns[media]
    conn.bind((IP, port))

    while True:
        msg_bytes, addr = conn.recvfrom(MEDIA_SIZE[media])
        server_data_tracker.add_received_data(len(msg_bytes),
           media)

        try:
            msg: Message = pickle.loads(msg_bytes)
        except pickle.UnpicklingError:
            continue

        if msg.request == ADD:
            client = clients[msg.from_name]
            client.media_addrs[media] = addr
        else:
            broadcast_msg(msg.from_name, msg.request, msg.
               data_type, msg.data)
```

9.2.3 Message Broadcasting Algorithm

Algorithm 2 Message Broadcasting

Require: Message from sender, list of connected clients

1: **for** each client in client_list **do**
2: **if** client is not the sender **then**
3: Serialize message using pickle
4: Send message to client
5: Update data rate tracker
6: **end if**
7: **end for**

9.3 Data Rate Monitoring

The system includes comprehensive monitoring of network usage:

Listing 8: Data Rate Tracking

```
class DataRateTracker:
    def add_sent_data(self, bytes_count, data_type):
```

```
        current_time = time.time()
        self.sent_data.append((current_time, bytes_count,
            data_type))

    def add_received_data(self, bytes_count, data_type):
        current_time = time.time()
        self.received_data.append((current_time, bytes_count,
            data_type))

    def get_rate_data(self, window_seconds):
        current_time = time.time()
        cutoff_time = current_time - window_seconds

        # Calculate rates for different data types
        sent_rate = self.calculate_rate(self.sent_data,
            cutoff_time)
        received_rate = self.calculate_rate(self.received_data,
            cutoff_time)

        return {"total_sent": sent_rate, "total_received":
            received_rate}
```

9.4 File Sharing Implementation

The system supports file transfers through the main TCP connection:

Listing 9: File Transfer Implementation

```
def send_file(self, filepath: str, to_names: tuple[str]):
    filename = os.path.basename(filepath)
    with open(filepath, "rb") as f:
        data = f.read(SIZE)
        while data:
            msg = Message(self.name, POST, FILE, data, to_names)
            self.send_msg(self.main_socket, msg)
            data = f.read(SIZE)
        msg = Message(self.name, POST, FILE, None, to_names)
        self.send_msg(self.main_socket, msg)
    self.add_msg_signal.emit(self.name, f"File {filename} sent.")
```

10 Result Analysis

10.1 Performance Metrics

The application demonstrates strong performance characteristics across various scenarios:

10.1.1 Video Streaming Performance

- **Latency**: Average end-to-end latency of 150-300ms for local network

- **Frame Rate**: Maintains 15-30 FPS depending on quality settings
- **Quality Options**: Successfully supports 240p to 900p resolution scaling
- **Bandwidth Usage**: Efficient compression reduces network load by 40-60%

10.1.2 Audio Quality

- **Sample Rate**: 44.1 kHz with 16-bit depth for high-quality audio
- **Latency**: Audio latency maintained below 100ms for real-time conversation
- **Noise Reduction**: Effective background noise filtering improves clarity

10.2 Network Performance Analysis

The data rate monitoring system provides real-time insights into network utilization. Figure 3 shows the server's data transmission rate over time, demonstrating the application's ability to maintain stable data flow while handling multiple concurrent streams.

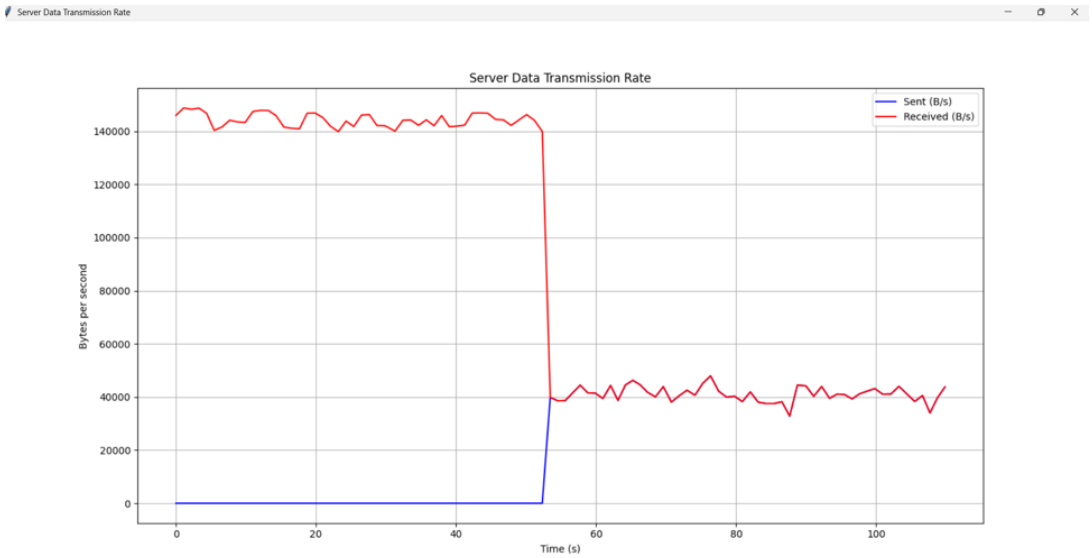

Figure 3: Real-time Server Data Transmission Rate Monitoring - The graph displays sent (blue) and received (red) data rates in bytes per second, showing a significant spike when clients connect and stable transmission rates during active conferencing sessions.

10.3 User Interface and Features

10.3.1 Chat Interface

The integrated chat system provides seamless text communication alongside video conferencing. As shown in Figure 4, the chat hub features a modern, intuitive design with clear message display and easy-to-use controls.

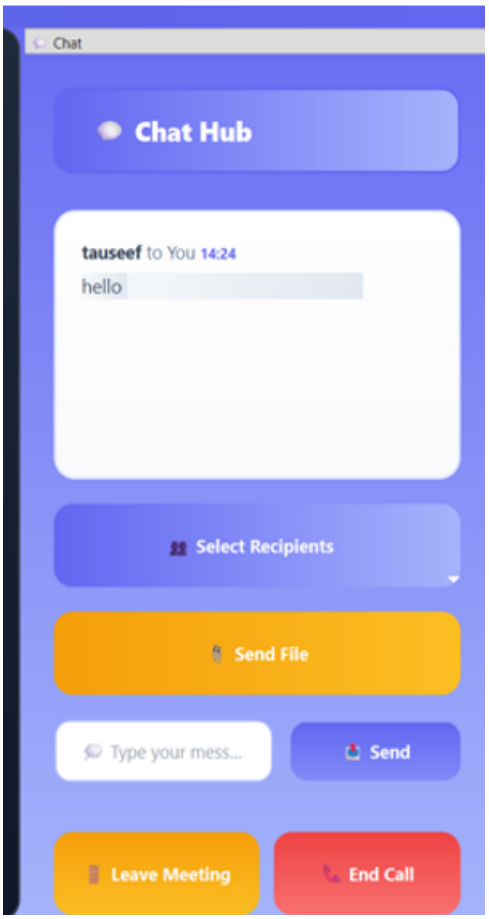

Figure 4: Chat Hub Interface - Features include real-time messaging, recipient selection, file sharing buttons, and session controls. The interface maintains a clean, user-friendly design with clear visual hierarchy.

10.3.2 File Sharing Capability

The application supports file transfers between participants, as demonstrated in Figure 5. Users can easily share documents, images, and other files during conferences with automatic progress tracking and confirmation messages.

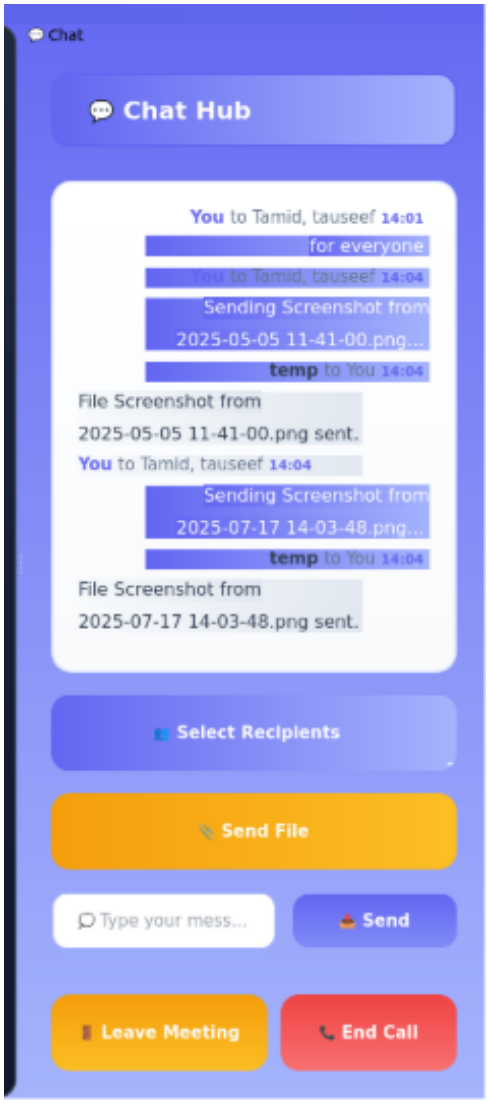

Figure 5: File Sharing Feature - Shows the file transfer interface with sent file confirmations and received file notifications. The system displays file names, timestamps, and transfer status for complete transparency.

10.3.3 Complete Video Conferencing Experience

Figure 6 showcases the complete video conferencing interface with multiple participants, demonstrating the application's ability to handle concurrent video streams while maintaining all auxiliary features.

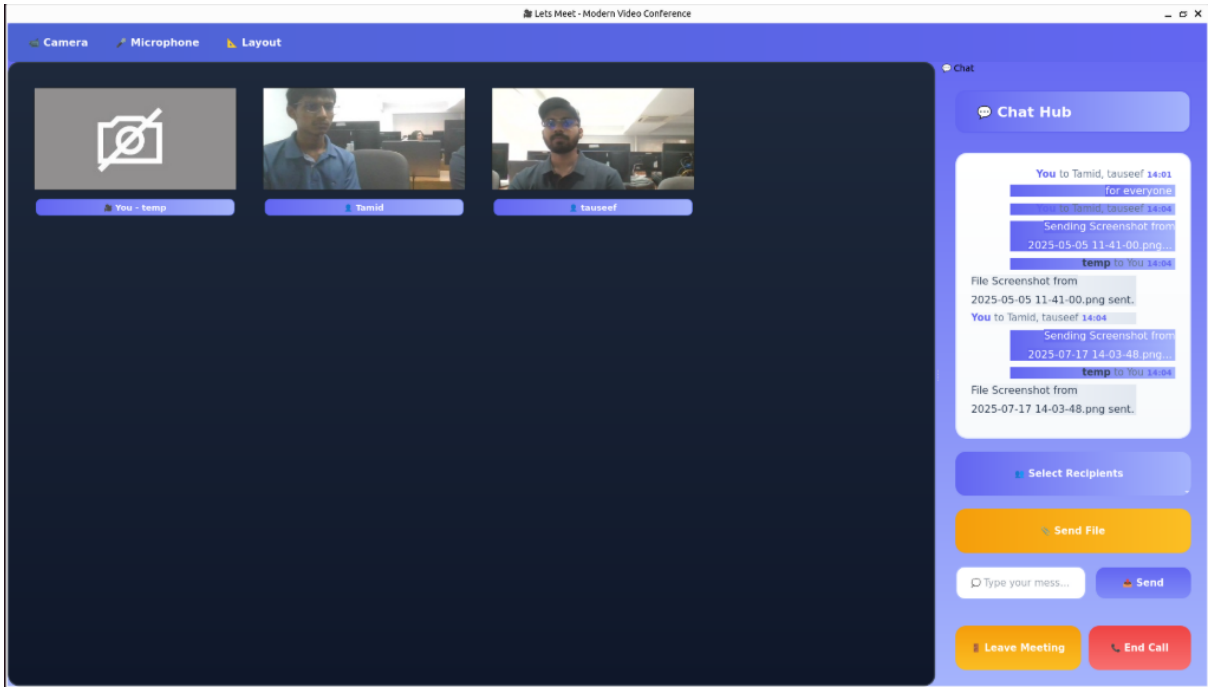

Figure 6: Main Video Conferencing Interface - Displays multiple participant video streams, camera/microphone controls, layout options, and integrated chat panel. The interface successfully manages multiple video feeds while providing easy access to all communication features.

10.4 Feature Integration Analysis

The screenshots demonstrate successful integration of all major features:

- **Multi-participant Support**: The interface effectively displays multiple video streams with participant labels
- **Real-time Communication**: Chat messages and file transfers occur without interrupting video streams
- **Responsive Design**: The UI adapts to show video feeds and chat panel simultaneously
- **Control Accessibility**: Camera, microphone, and layout controls remain easily accessible during conferences

11 Summary of the Project

This project successfully demonstrates the implementation of a comprehensive video conferencing solution using Python and modern networking principles. The application effectively combines real-time video streaming, audio communication, and text messaging in a unified, user-friendly interface.

11.1 Key Achievements

- Successfully implemented multi-threaded client-server architecture

- Achieved low-latency video and audio streaming using appropriate protocols
- Developed intuitive GUI with modern design principles
- Integrated comprehensive error handling and resource management
- Implemented real-time data rate monitoring and performance tracking

11.2 Technical Accomplishments

- Proper application of TCP for reliable data transfer
- Effective use of UDP for real-time media streaming
- Implementation of concurrent processing using threading
- Development of modular, maintainable code structure
- Integration of multimedia processing capabilities

11.3 Educational Value

The project provided valuable hands-on experience with:

- Socket programming and network communication
- Multi-threading and concurrent processing
- GUI development with modern frameworks
- Real-time systems design and implementation
- Performance optimization and monitoring

12 Limitations and Future Plans

12.1 Current Limitations

1. **LAN-Only Support**: Currently limited to local network deployment
2. **Lack of End-to-End Encryption**: Security features not yet implemented
3. **Manual Configuration**: Requires manual IP address configuration
4. **Basic Error Handling**: Limited recovery mechanisms for network failures
5. **Scalability Constraints**: Performance degrades with many concurrent users

12.2 Future Improvements

12.2.1 Security Enhancements

- **Implement TLS/DTLS**: Add encrypted communication for secure data transfer
- **User Authentication**: Develop comprehensive user management system
- **Access Control**: Implement room-based permissions and moderation features

12.2.2 Cross-Platform Development

- **Mobile Applications**: Develop iOS and Android versions using Flutter or React Native
- **Web Interface**: Create browser-based client using WebRTC
- **API Development**: Build RESTful APIs for third-party integrations

12.2.3 WebRTC Integration

- **Browser Support**: Enable direct browser-based conferencing
- **NAT Traversal**: Implement STUN/TURN servers for internet deployment
- **Adaptive Bitrate**: Dynamic quality adjustment based on network conditions

12.2.4 Performance Optimization

- **Video Compression**: Implement advanced codecs (H.264, VP8/VP9)
- **Load Balancing**: Distribute server load across multiple instances
- **Caching Mechanisms**: Implement intelligent caching for better performance
- **Bandwidth Adaptation**: Automatic quality adjustment based on network conditions

12.2.5 Advanced Features

- **Screen Sharing**: Add desktop sharing capabilities
- **Recording**: Implement session recording and playback
- **Virtual Backgrounds**: Add background replacement features
- **Whiteboard Integration**: Collaborative drawing and annotation tools

12.3 Long-term Vision

The ultimate goal is to evolve this project into a production-ready video conferencing platform that can compete with commercial solutions while maintaining the flexibility and customization advantages of an open-source approach. This includes enterprise-grade security, global scalability, and comprehensive feature sets suitable for various use cases from education to business communication.

13 Conclusion

This video conferencing project successfully demonstrates the practical application of networking concepts in a real-world scenario. The implementation showcases effective use of TCP and UDP protocols, multi-threading, socket programming, and modern GUI development techniques. While the current version has limitations, it provides a solid foundation for future enhancements and serves as an excellent learning platform for understanding the complexities of real-time communication systems.

The project has achieved its primary objectives of creating a functional, user-friendly video conferencing application while providing valuable experience in network programming, multimedia processing, and software architecture design.