Development of Application for Frame-Based Image Transmission and Reception System

TABLE OF CONTENT

1.INTRODUCTION

Real-time image transmission is a fundamental requirement for modern applications such as messaging, security, and remote monitoring. This project introduces a robust and interactive platform that enables the transmission of high-resolution images over a local network in frame-wise chunks. The system segments compressed images into manageable byte-sized frames, each embedded with a header that encodes essential metadata for efficient reconstruction. The application supports both manual and automated operations, including timer-based sending, continuous streaming, and error-resilient receiving. With a modular and scalable architecture, this software prototype serves as a versatile foundation for various engineering and real-time communication systems.

2.TOOLS AND TECHNIQUES

This project was implemented using Python 3.10, integrating key libraries such as OpenCV for image compression and transmission, NumPy for efficient data handling, and Tkinter with ttkbootstrap for building modern desktop GUIs. Socket programming was employed to facilitate real-time communication between the transmitter and receiver, ensuring structured and secure data exchange over TCP/IP. PAGE was used for GUI layout design, enhancing the visual interface and user experience. Additional tools such as PIL, threading, and multiprocessing enabled smooth execution and responsiveness.

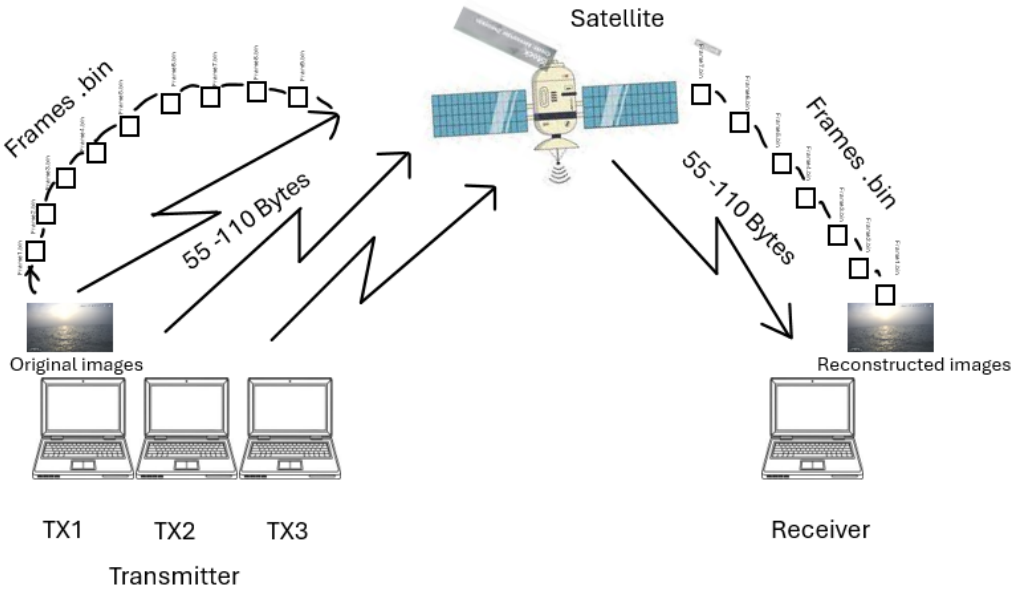
- Programming Language: Python 3.10+
- Core Libraries: OpenCV, NumPy, PIL, socket
- GUI Frameworks: Tkinter, ttkbootstrap, PAGE
- Communication Protocol: TCP (via socket)
- Supporting Modules: threading, multiprocessing
- Hardware Used: USB/Web Cameras, LAN
- IDE Used: Visual Studio Code

Protocol Design:

- 10-byte custom frame header for index, coordinates, and total count
- Fixed-size data chunks per frame (e.g., 90 bytes)
- TCP/IP for reliable transport

3. SYSTEM ARCHITECTURE OVERVIEW

The proposed system architecture comprises two main functional modules that operate independently: the Transmitter Module and the Receiver Module. These modules communicate over a local network using a reliable TCP/IP socket-based protocol, enabling seamless image transmission and reconstruction. Both modules are designed with scalability, modularity, and ease of use in mind.

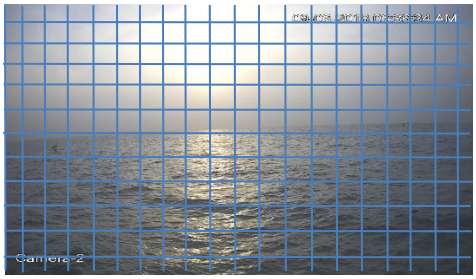


Architecture Diagram

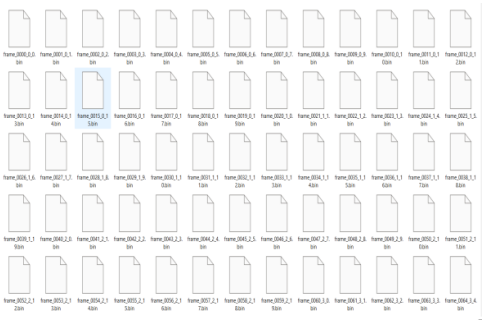
Transmitter Module:

- Captures live images through a connected camera or allows users to select existing image files from the system.
- JPEG compression applies to reduce the image size while maintaining sufficient visual quality (fixed quality level set at 40).
- Splits the compressed image into equal-sized data chunks and prepends each chunk with a 10-byte header containing metadata such as frame index, row/column identifiers, and total frame count.
- Establishes a socket connection with the receiver and sequentially sends each frame along with the necessary metadata for accurate reconstruction.
- Provides a graphical user interface (GUI) that supports live camera preview, frame-size adjustment, automated capture modes (timer/continuous), and event logging for user awareness.

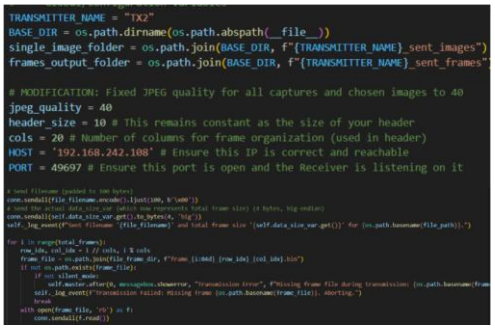
Input image



Splitted Frames



Transmitted Frames



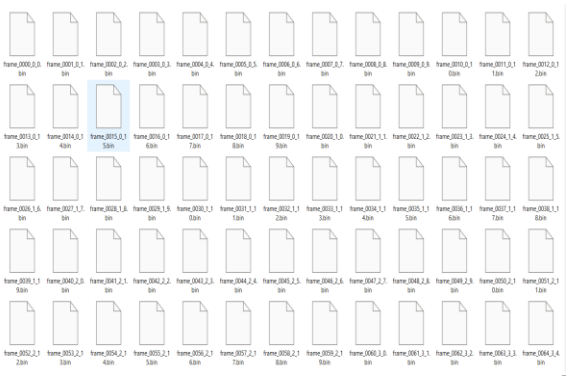
```
TRANSMITTER_NAME = "TX2"
BASE_DIR = os.path.dirname(os.path.abspath(__file__))
single_image_folder = os.path.join(BASE_DIR, f"{TRANSMITTER_NAME}_sent_images")
frames_output_folder = os.path.join(BASE_DIR, f"{TRANSMITTER_NAME}_sent_frames")

# MODIFICATION: Fixed JPEG quality for all captures and chosen images to 40
jpeg_quality = 40
header_size = 10 # This remains constant as the size of your header
cols = 20 # Number of columns for frame organization (used in header)
HOST = '192.168.242.108' # Ensure this IP is correct and reachable
PORT = 49697 # Ensure this port is open and the Receiver is listening on it
```

Frames converted

Receiver Module:

- Initializes a TCP server socket that listens for incoming connections on a predefined IP address and port, waiting to accept transmissions from any authorized transmitter.
- Once connected, it receives frame-by-frame data packets and saves them temporarily in an organized manner for further processing.
- Extracts metadata from the frame headers to map each chunk into a structured grid based on its row and column positions for reassembly.
- Reconstructs the original image by concatenating received frame chunks in the correct order and decodes the final image using OpenCV functions.
- Updates the GUI in real time with the received image, logs into the reception activity, and maintains an accessible history of images grouped by transmitter ID.



Received Frames

Reconstruct image

4.TRANSMITTER-SIDE DESIGN AND PROTOCOL

The transmitter-side operation begins with the selection or live capture of an image using a connected camera device or file explorer interface. Once an image is selected, the system initiates compression using OpenCV's JPEG encoding with a fixed quality setting of 40% to ensure a balance between file size and image clarity.

The compressed image is converted into a byte stream, which is then divided into multiple frames. Each frame consists of a 3-row × 11-column pixel matrix (i.e., 33 pixels per frame). This pixel block is then encoded into a data chunk with a size close to 90 bytes, keeping the total frame size within 100 bytes including the 10-byte header.

Each frame is constructed by appending the following header metadata:

- 2 bytes – Frame Index: Indicates the sequence number of the frame.
- 2 bytes – Row Index: Specifies the vertical row location in the image matrix.
- 2 bytes – Column Index: Specifies the horizontal column location.
- 2 bytes – Total Frame Count: Total number of frames required to transmit the image.
- 2 bytes – Reserved Padding: Reserved for alignment or additional control data.

After the header is attached, the frames are saved in binary format (.bin files), each containing the header followed by the image data. The transmitter then opens a socket connection and sends each frame sequentially to the receiver.

The transmitter supports three operational modes to accommodate different use cases:

- Manual Mode: Allows the user to capture and transmit images manually on demand.
- Timer-Based Mode: Automates the capture and transmission process in defined cycles (e.g., 15 minutes active transmission followed by 15 minutes idle).
- Continuous Mode: Enables real-time streaming by continuously capturing and transmitting images in succession.

```
def generate_frames_from_file(self, file_path):
    file_filename = os.path.basename(file_path)
    try:
        with open(file_path, 'rb') as f:
            file_bytes = f.read()
    except FileNotFoundError:
        self._log_event(f"ERROR: File not found for frame generation: {file_path}")
        return None, 0, None

    total_frame_size = self.data_size_var.get()
    chunk_size = total_frame_size - header_size

    # This check is still valid and important
    if chunk_size <= 0:
        self._log_event(f"ERROR: Calculated chunk size ({chunk_size} bytes) is zero or negative (total frame size {total_frame_size} - header size {header_size}). Abo
        self.master.after(0, messagebox.showerror, "Configuration Error", f"Calculated image data chunk size is too small or negative. Please set 'Data Size (Bytes pe
        return None, 0, None

    total_frames = (len(file_bytes) + chunk_size - 1) // chunk_size
    file_frame_dir = os.path.join(frames_output_folder, os.path.splitext(file_filename)[0] + "_" + datetime.datetime.now().strftime("%d%m%S_%f"))
    os.makedirs(file_frame_dir, exist_ok=True)

    self._log_event(f"Generating frames: Total Frame Size requested: {total_frame_size} bytes (Header: {header_size} bytes, Image Data Chunk: {chunk_size} bytes).")

    for i in range(total_frames):
        chunk_start = i * chunk_size
        chunk_end = (i + 1) * chunk_size
        raw_chunk = file_bytes[chunk_start:chunk_end]
        chunk = raw_chunk + b'\x00' * (chunk_size - len(raw_chunk))

        row_idx, col_idx = i // cols, i % cols
        header = (
            i.to_bytes(2, 'big') +
            row_idx.to_bytes(2, 'big') +
```

Generate Frames Format

5.RECEIVER-SIDE HANDLING AND RECONSTRUCTION

The Receiver accepts frames via socket and processes as follows:

- Extracts metadata from the frame headers.
- Maps each chunk to its respective (row, column) position based on the transmitted index.
- Tracks and identifies any missing frames and logs their absence.
- Arranges the received frames in the same order and matrix alignment as the transmitted side, ensuring structural accuracy.
- If any frame is missing, replace it with a black placeholder to maintain visual and positional consistency.
- Reconstructs the full image byte stream by stitching frames in sequence.
- Decodes the reconstructed stream using OpenCV (cv2.imdecode).
- Displays the final image in the GUI and stores it in categorized folders.

The receiver uses consistent IP and port configuration for secure and predictable communication with the transmitter. The structure is designed to be scalable and can be adapted for satellite-based data reception using similar frame formats and protocols in future implementations.

```python
def _recv_all(self, sock, n):
    data = b''
    while len(data) < n:
        packet = sock.recv(n - len(data))
        if not packet:
            return None
        data += packet
    return data

def stop_server(self):
    if not self.server_running:
        self._log_event("Server already off request received.")
        return

    self._log_event("Stopping server...")
    self.server_running = False
    try:
        if self.server_socket:
            self.server_socket.shutdown(socket.SHUT_RDWR)
            self.server_socket.close()
            self.server_socket = None
    except Exception as e:
        self._log_event(f"Error closing server socket: {e}")

    if self.server_thread and self.server_thread.is_alive():
        self._log_event("Waiting for server thread to terminate...")
        self.server_thread.join(timeout=2)
        if self.server_thread.is_alive():
            self._log_event("Server thread did not terminate gracefully.")

    for thread in list(self.active_connections):
        if thread.is_alive():
            self._log_event(f"Attempting to join client handler thread {thread.name}...")
            thread.join(timeout=1)
```

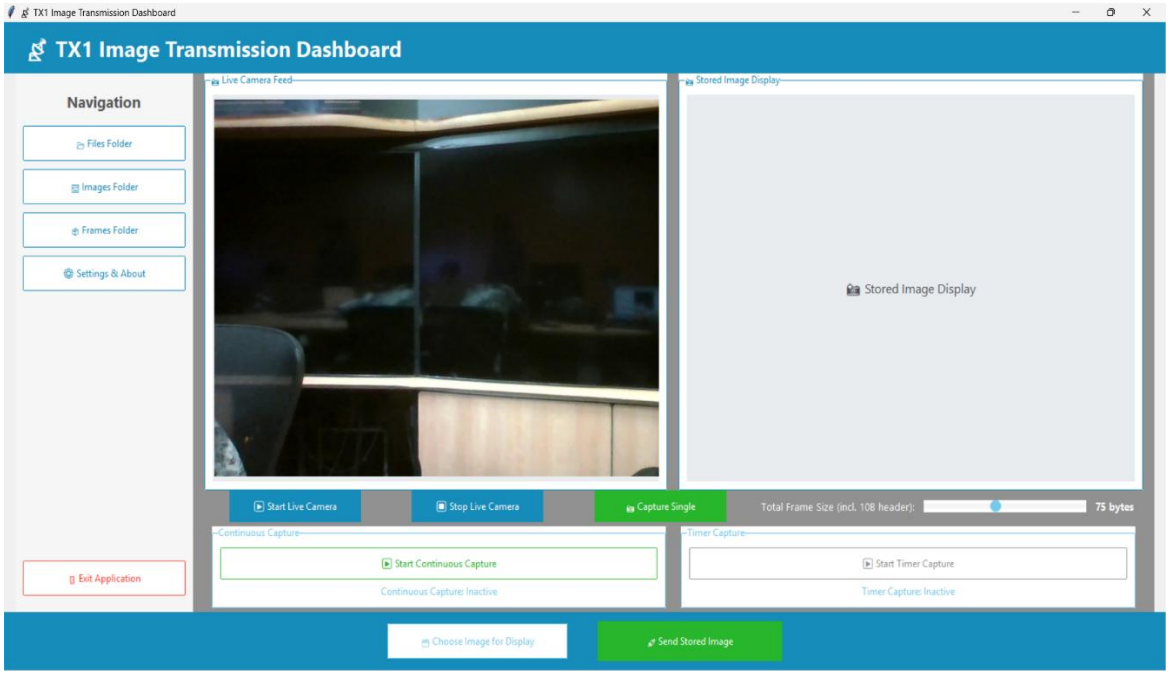
Receiver Side Reconstruct images

6.GRAPHICAL INTERFACE (GUI) CAPABILITIES

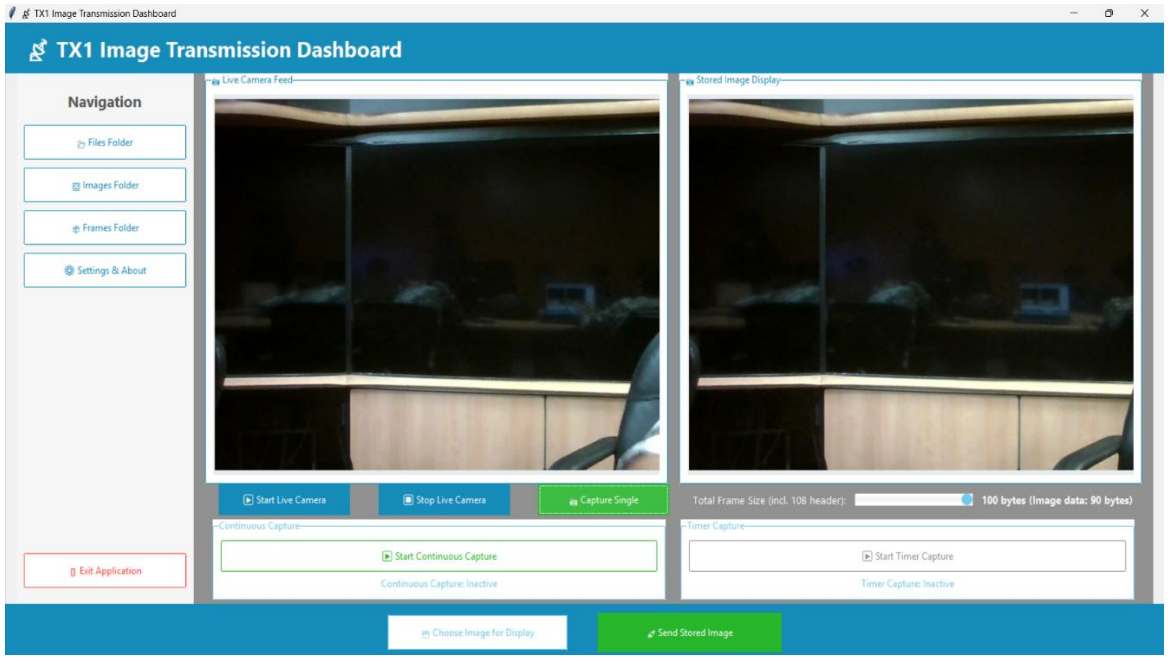
The system includes intuitive graphical interfaces for both the transmitter and receiver modules, making the application interactive and user-friendly.

Transmitter Interface:

- **Live Camera Preview**: Displays the real-time video feed from the connected camera, allowing the user to position the object or environment before capturing.
- **Capture Image / Choose File / Auto-Capture**: Offers the ability to either manually capture an image, browse and select an image file, or automatically capture images at intervals.
- **Frame Size Slider (80–100 bytes)**: Allows users to fine-tune the data payload of each frame to optimize transmission speed and quality.
- **Transmission Logs**: Keeps a real-time record of all frames sent, including timestamps, statuses, and any errors encountered.
- **Timer and Continuous Mode Controls**: Lets users enable timed captures or continuous streaming depending on application needs.
- **Progress Bar Display**: Visually indicates the percentage of the transmission completed, ensuring feedback on task status.



Transmitter Side GUI



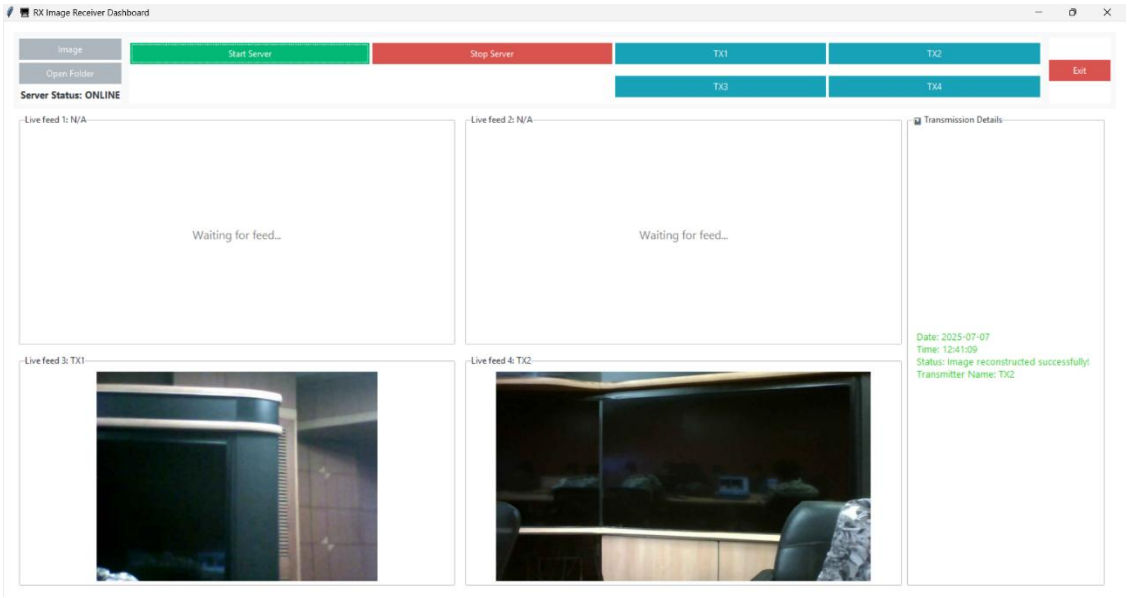
Transmitter Capture image

Receiver Interface:

- **Start/Stop Server Controls**: Provides a simple interface to initiate or terminate the socket server for receiving data.
- **Image History Panel (by Transmitter)**: A scrollable pane showing thumbnails of previously received images, sorted by transmitter ID.
- **Live Preview of Last Received Frame**: Shows the most recent image received in full size or preview mode.

- **Transmission Statistics Viewer**: Displays metadata like number of frames received, any missing frames, and total image size.
- **Real-Time Logs and Alerts**: Notifies users of missing frames, connection errors, or successful reception events in a separate logging window.

All GUI elements are styled using ttkbootstrap for a clean and modern look, and layout was generated using the PAGE tool for rapid interface design.



Receiver side GUI

7.EXPERIMENTAL RESULTS AND ANALYSIS

Testing Setup: The system was tested on a real-time local network environment involving a laptop (as Transmitter) and a Desktop (as Receiver), both connected via Wi-Fi and Ethernet for comparison. Images of approximately 120 KB (JPEG compressed) were used, and each image was divided into frames of 100 bytes (90 bytes of image data + 10 bytes of header).

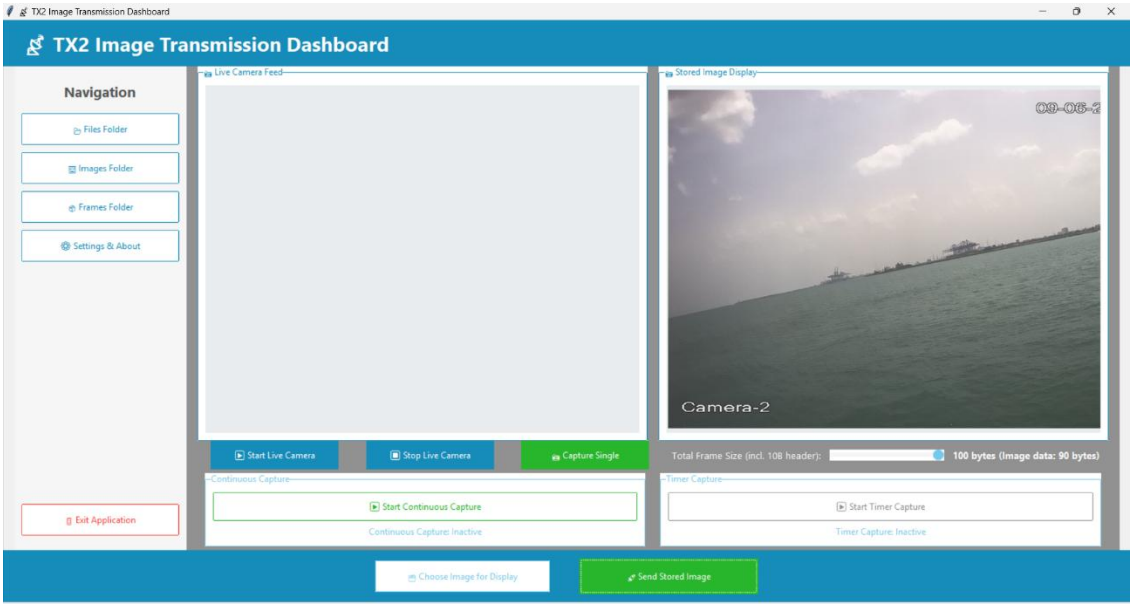
Performance Metrics:

- **Average Transmission Time:** Approximately **2.3 seconds per image** under standard Wi-Fi conditions. The time was marginally lower over Ethernet, confirming the system's responsiveness in wired networks.
- **Decoding Success Rate:** Achieved **100% decoding accuracy** when all frames were received, with OpenCV successfully reconstructing the full image byte stream.
- **With Missing Frames:** The receiver effectively filled missing segments with black blocks, preserving spatial consistency and clearly indicating data loss regions. This fallback mechanism ensured the integrity of reconstructed visuals.
- **GUI Latency:** Minimal lag was observed between sending and GUI updates, with

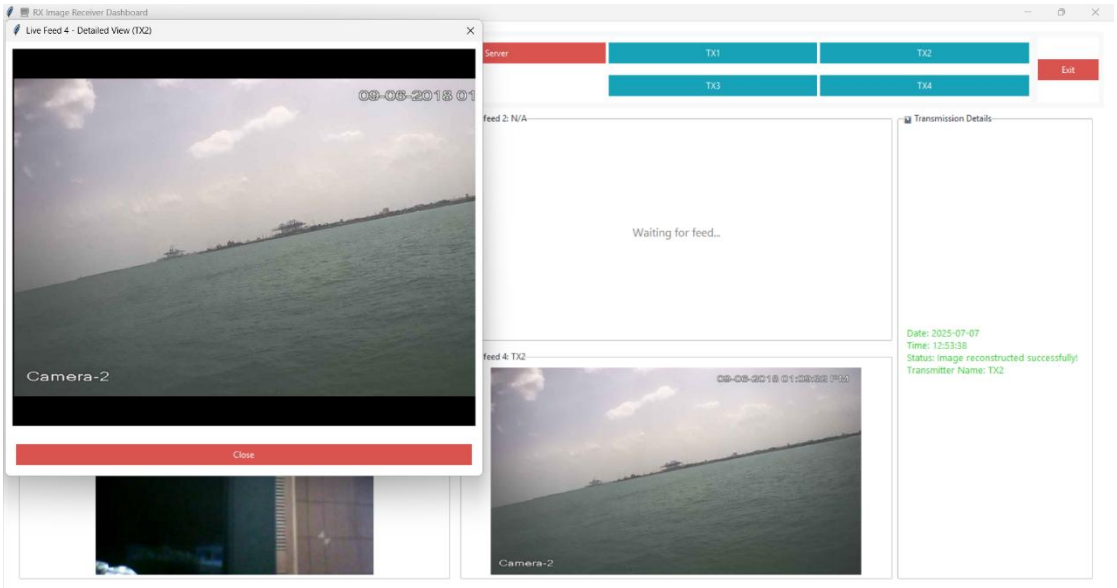
frame previews and status indicators updating in near real-time even under continuous mode.

- **Auto Modes:** Both timer-based and continuous capture modes functioned without performance drop or data mismatch. Scheduled image transmissions worked reliably and maintained synchronization between sender and receiver.

The system's modular handling of frames and metadata proved effective in managing asynchronous transfers and network variability. Moreover, the structured folder organization per transmitter enhanced post-analysis of image logs. This implementation lays a scalable foundation for more complex real-time communication systems.



Transmitter GUI



Receiver GUI

8.CHALLENGES AND LIMITATIONS

- Frame Loss and Tolerance: Frame loss due to network delays was a significant challenge. This was mitigated by implementing receiver-side logic to detect missing frames and visually patch them using black placeholders. However, accurate data restoration in lossy networks remains an area for improvement.
- Fixed JPEG Quality: The current implementation uses a fixed JPEG compression quality to simplify transmission. While effective, adaptive encoding based on network speed or image content could improve efficiency and visual fidelity.
- Still Image Only: The system is currently designed for still image transmission only. Extending the same concept to support live video streaming will require real-time encoding, buffering, and synchronization logic.
- GUI Performance: As the number of received images grows, the receiver GUI may experience performance slowdowns. Features like pagination, lazy loading, or batch rendering could be added to improve scalability.
- Limited to Static IP Addressing: The project currently uses IP-based communication suitable for LAN setups. However, future implementations are aimed at satellite communication systems where traditional IP addressing may not be feasible. Instead, communication must adhere to satellite telemetry constraints, and the system must operate under byte-limited frame protocols.
- Satellite Communication Byte Limitation: In satellite systems, the frame size is typically constrained to a narrow bandwidth, usually between 55 to 110 bytes per frame. The existing system must be adapted to comply with these size restrictions to support reliable frame-based image transmission and reception in satellite environments such as remote sensing or low-orbit relay systems**.**

9. DEVELOPMENT OF THE APPLICATION

The development of the application focused on creating a timer-based autonomous system in which the transmitter and receiver programs operate seamlessly after setup. A camera is centrally placed (such as at an entrance or monitoring zone), and the transmitter is configured to capture a live image at regular time intervals. At each scheduled time, the system captures a fresh image, compresses it, segments it into frames, and immediately transmits it to the receiver—without accessing or sending any previously stored images.

On the receiver side, the program continuously listens for incoming frames, reconstructs the received image in real-time, and displays it through the user interface. This ensures that only newly captured images are transmitted and visualized, offering a real-time snapshot of the monitored area at each interval.

The system is designed for portability, with the complete software stored on a USB drive or microcontroller-based storage. Upon insertion into any compatible device, the appropriate transmitter or receiver interface auto-launches based on predefined configurations. All operational parameters—including IP, port, capture interval, and display mode—are preloaded in config files, allowing plug-and-play use in the field without requiring manual script execution. This timed, live-capture transmission model closely simulates real-world scenarios such as periodic monitoring or scheduled surveillance, enabling efficient and intelligent image handling without user intervention between captures.

10. ANALYSIS AND RESULT

This work delivers a robust, modular, and extensible image transmission system. With real-time interaction, error handling, and visual feedback, the system mirrors practical use cases like WhatsApp image sharing, but at a protocol-level implementation. During testing, the transmitter and receiver demonstrated accurate transmission of segmented image frames, smooth GUI operation, and tolerance to minor data loss by filling missing frames with black placeholders.

All functionalities including live camera preview, timed transmission, and continuous mode worked effectively under both Wi-Fi and Ethernet. Results indicate high accuracy in image reconstruction, minimal GUI latency, and seamless handling of frames across multiple sender identities.

Output Image

11. CONCLUSION AND FUTURE SCOPE

Scope for Future Enhancements:

- Live video stream slicing and transmission
- AES encryption for secure transport
- Cloud storage of received images
- Mobile app integration
- WebSocket or MQTT protocol support

The system has potential for deployment in real-world scenarios, especially for applications involving offline image collection, satellite communication, and underwater observation. By enhancing adaptability and security, this tool could become a lightweight and powerful transmission engine for embedded and mobile systems.