Reducing paper waste and Electricity consumption with the help of E-Paper Displays and Banners in public spaces

By: Sameed Ahmed

USN: 2VX22CB043

Abstract:

The growing number of digital displays and billboards, along with adverts in the form of flyers and posters have resulted in lots of paper wastage as well as greater electricity consumption, especially in densely populated regions, most notably public hotspots like bus-stands and train stations

In this rapidly evolving landscape of urban communication, this project presents a pioneering approach to public information dissemination through E-paper display technology. By leveraging a distributed Internet of Things (IoT) infrastructure, the system replaces traditional LED displays and static posters with dynamic, energy-efficient, and environmentally sustainable digital signage. The solution integrates NodeJS web interfaces, MQTT communication protocols, Raspberry Pi edge computing, ESP32 microcontrollers, and Seeed Studio E-paper breakout boards to create a scalable, low-power public information network that can be easily deployed and maintained across urban and rural environments.

Introduction

The E-paper Display System discussed in this paper incorporates the technology of e-paper displays and a set of modern tools to produce a public sign display system. There is a constant, growing trend for public display systems in places such as airports, shopping centers, offices, and transport centers, where information changes constantly. The combined inclusion of e-paper displays gives a low power consumption, attractive and flexible solution to such settings. Here, we develop a cloud-based e-paper display system which is integrated with IoT functionalities in such a way that content can be uploaded, processed and sent out to many e-paper displays over a wireless broadcast system, i.e. MQTT, with minimal setup, effort and extensive control through a comprehensive web interface.

Literature review

Historical context of public information displays

The landscape and architecture of public information system has experienced several significant transformations over the past few decades, most significant of which include traditional methods, like printed posters and banners, LED displays and other methods such as static signs and pamphlets.

However, as effective these methods are, they contribute highly to our carbon footprint on the world. This huge carbon footprint originated from multiple factors such as paper processing and waste, high electricity consumption and e-waste toxicity, etc. all these factors being the major disadvantages of such conventional methods of public information, among many others.

E-Paper Displays:

This is where E-Paper displays come into play as the redeeming quality.

E-Paper (also known as e-ink) technology has emerged as a groundbreaking alternative to these conventional display systems. Unlike traditional displays such as LED and OLED, E-paper closely mimics the appearance, feel and thickness of ink on an ordinary paper, hence the name.

This system in and of itself provides multiple critical advantages over the traditional displays:

1. **Low Power Consumption**: E=paper displays consume minimal electricity, as much as 95% lower than traditional LED displays; given they only consume electricity when updating, and do not require electricity to maintain the image on the screen for any amount of time, rather than constant illumination.
2. **Visibility in varying lighting conditions:** E-Paper displays maintain excellent visibility even in bright lighting conditions, such as under bright sunlight, as compared to other types of displays that lose effectiveness when placed under bright light.
3. **Reduced Environmental Impact:** By reducing the amount of physical paper printing and wastage, as well as completely overhauling the need for high consumption of electricity, and further being highly recyclable (almost 99% recyclability), they greatly reduce the environmental impact, representing a sustainable communication medium.

IoT and Distributed Display Systems

The integration of Internet of Things (IoT) technologies has transformed display infrastructure from boring static systems to dynamic, intelligent systems. Key developments include:

- Distributed communication protocols enabling real-time content updates.
- Edge computing approaches for efficient data processing
- Low-power, effective wireless communication technologies

Technical Architecture Breakdown:

```
 [Web Upload Interface (NodeJS)]
                ↓
      [Raspberry Pi 5 Server]
        ├── MQTT Broker (Mosquitto)
        ├── Image Processing (Python)
        └── Content Distribution
                ↓
     [ESP32 Microcontrollers]
                ↓
         [E-Paper Displays]
```

Detailed Component Analysis:

Detailed Component Analysis:

1. **Web Upload Interface (NodeJS)**

In this project, for the purpose of deploying a quick, efficient, simple and user-friendly user interface for uploading image and data files, NodeJS was taken into picture.

Nodejs is a JavaScript-based web-development framework that makes full use of the available HTTP protocols to deliver a functioning web interface in a short amount of time. As it is, one can either connect to the website, select an image and then click upload to upload the image to the server, or, for those who want to either make life difficult for themselves or want to operate everything through their terminals (because that is supposedly "faster and more efficient"), one can also send a post request to the website at https://websiteurl/post.

This effectively uploads the image to the website, triggering scripts to process the uploaded image for transmission to the display

Advantages:

- Provides user-friendly image upload mechanism
- Handles initial content ingestion
- Validates and prepares images for processing

2. Raspberry Pi 5 Server

For the scope of this project, I have utilised a Raspberry Pi 5, running its trademark Bookwork Debian OS, to effectively handle all the process to be run on the server.

Further, I also have a ngrok instance running a SSH tunnel, for remote access and maintenance of the server, for cases of failure or just regular updates and cleanup.

Also there also exists a layer for MQTT communications, via a software called Mosquitto

Advantages:

- Centralized computing and distribution hub
- Runs **Debian Bookworm** for stability
- Manages:
    - MQTT broker (Mosquitto)
    - Image processing pipeline
    - Content distribution logic

3. Image Processing Pipeline (Python)

Here, Python was used for the main scripts due to its fast nature and its simplicity.

The python part of the project mainly comprises of two scripts. One of them handles image processing, getting it ready for presenting on the E-Paper display. This mainly Uses the Pillow Imaging library and NumPy for resizing and converting the image into a black-and-white bitmap (for displaying on the Black-and-white E-Paper Display).

The second python script consists of an algorithm to periodically broadcast the processed data over to the esp32 microcontrollers (every 5-7 seconds for now), using the paho-mqtt library.

Advantages:

- Utilizes Pillow and NumPy for image transformation
- Responsibilities:
    - Resize images to display specifications
    - Convert images to bitmap format
    - Prepare images for transmission

4. Communication Layer (MQTT)

The Communication layer is built on the standard MQTT protocol. MQTT is a protocol/service that user 2.4 GHz and 5 GHz WIFI channels for broadcasting structured messages over the network.

The server side MQTT client is handled by the Mosquitto service program; subsequently, the client-side MQTT client is paho-mqtt on the server (for broadcasting) and Adafruit's Mini-mqtt (for receiving)

Advantages:

- Uses lightweight publish-subscribe messaging protocol
- Enables efficient, low-overhead content distribution
- Supports multiple simultaneous display updates

5. Edge Nodes (ESP32 Microcontrollers)

The currently used microcontrollers have ESP32 processors, to effectively communicate with the MQTT server by utilizing Adafruit's mini-mqtt library and hence receiving and forwarding the image data after processing to the display

Advantages:

- Low-power wireless microcontrollers
- Running CircuitPython firmware
- Receives and renders display content
- Utilizes Mini-MQTT for communication

6. **Display Hardware (Seeed Studio E-Paper Breakout)**

The main and only available part to connect the microcontrollers directly to the E-Paper display. It provides an 8-pin interface to the microcontroller and receives data over SPI Interface (SCK and MOSI pins) and a 20-pin cable interface for connecting to the display

Advantages:

- Provides direct interface between microcontrollers and e-paper displays
- Supports efficient refresh and update mechanisms

7. **Tunnelling software (ngrok):**

Tunnelling software provides an interfacing option to access http and ssh ports outside of the local network, usually when one does not have access to the port forwarding options of a router's interface.

This project utilises 2 tunnels, namely SSH and HTTP tunnels, for remote access to the imaging website and ssh access for remote maintenance, cleanup, and updates

Advantages:

- **Pros:**
  - Remote access to http and ssh ports
- **Specific to Project:** Simple, free and easy to use

Technology Selection Rationale

Why These Specific Technologies?

NodeJS

- **Pros:**
  - Rapid development
  - Excellent for web interfaces
  - Large ecosystem of packages
- **Specific to Project:** Simple and easy image upload and form handling

Raspberry Pi 5

- **Pros:**
  - Cost-effective edge computing
  - Low power consumption
  - Robust Linux environment
  - Multi-threaded/cored functionality
- **Specific to Project:** Centralized processing and MQTT brokering

Python (Image Processing)

- **Pros:**
  - Rich image processing libraries - Pillow
  - NumPy for efficient array manipulations
  - Quick script development
- **Specific to Project:** Bitmap conversion and image preparation

MQTT Protocol

- **Pros:**
  - Lightweight messaging
  - Low bandwidth requirements
  - Lower power consumption
  - Support for multiple clients
- **Specific to Project:** Efficient display content distribution

ESP32 with CircuitPython

- **Pros:**
  - Low-power microcontroller
  - Wireless connectivity
  - Python-based programming
  - Cost-effective
- **Specific to Project:** Efficient display node implementation

Seeed Studio E-Paper Breakout

- **Pros:**
  - Direct microcontroller integration
  - Support for multiple e-paper display types
  - Robust hardware design
- **Specific to Project**: Reliable display interface

Comparative Alternatives Considered

While the current stack offers an optimal solution, alternative approaches were evaluated:

1. **Raspberry Pi Alternative:** Arduino Uno Rev4, NVIDIA Jetson Nano, Particle Tachyon
2. **Communication Protocols:** WebSockets, HTTP polling, SPI interfacing
3. **Microcontrollers:** Arduino, STM32, ESP8266, RP2040

Cause of selection of current technologies over alternatives:

1. **Raspberry Pi:** Cost Parameters and community support for faster deployment and error hotfixes.
2. **Communication Protocols (MQTT):** Simplicity, security, cost-effective scalability, and lower maintenance.
3. **Microcontrollers (ESP32):** Ready wireless connectivity, cost and community support for faster development and easier maintenance and long-term support.

The selected technologies provide the best balance of:

- Performance
- Cost-effectiveness
- Energy efficiency
- Development speed
- Scalability
- Long-term maintenance

Comparative Analysis of Current Public Display Technologies:

The evolution of public information display technologies reflects a broader narrative of technological transformation, where each generation of systems attempts to address the limitations of its predecessors. Traditional display technologies have long been constrained by significant trade-offs between visual performance, energy consumption, and content flexibility.

LED display systems, which were once the epitome of public communication infrastructure, are a technology that consumes a lot of energy and is not very adaptable. The LED display system captures the attention of people with bright, glowing displays but is very costly to operate. An LED display consumes a huge amount of electricity, which means it generates a large amount of heat and requires constant maintenance. Their main strength is in being visible during low-light conditions, but this is at the cost of environmental sustainability and long-term operational efficiency.

LCD public displays came as a highly dynamic alternative, with the scope of electronic content update possibilities and multimedia capabilities. Being an intermediate technological solution, it still stands between static printed media on one side and fully interactive digital platforms on the other. However, LCD has significant drawbacks, such as low performance in extreme temperature conditions, possible screen burn-in, and moderate or high power consumption. Additionally, due to its sophistication and sensitivity to environmental changes, it is not as dependable for steady implementation in public infrastructure.

Printed posters and traditional banners, while outdated, have maintained relevance due to their simplicity and low initial investment. These analog communication methods are characterized by their zero electrical consumption and straightforward production process. Yet, they fundamentally represent a static communication model, requiring manual replacement and generating significant paper waste. Their inability to provide real-time, adaptive information makes them increasingly obsolete in a rapidly changing urban landscape.

The proposed E-paper display system is a transformative solution that addresses multiple limitations of existing technologies. This infrastructure combines IoT principles with e-paper display technology to offer an unprecedented combination of energy efficiency, content flexibility, and environmental sustainability. The architecture of the system, which leverages NodeJS web interfaces, MQTT communication protocols, Raspberry Pi edge computing, and ESP32 microcontrollers, represents a comprehensive approach to public information dissemination.

Its main reason for being so appealing, as an E-paper solution is that it has almost-zero power consumption when on the display panel. Unlike LED and LCDs, which constantly use electrical power, e-paper-based displays only use electricity at the time of updating of content. This makes all the difference in operational expenditure and carbon footprint reduction to a great extent. And the displays have high-quality visibility under a variety of lighting conditions, including low-light environments, an imperative requirement for public infrastructure.

Furthermore, the distributed IoT architecture allows for immediate, wireless content updates across multiple display nodes. This is a change in thinking from traditional display systems, which involved content modification that was time-consuming and logistically complex. The image processing pipeline, which is based on Python, and the MQTT communication protocol ensure low-overhead, efficient content distribution, thus making large-scale deployment technically feasible and economically viable.

While the initial investment in infrastructure is moderate compared to traditional display systems, the long-term benefits are quite significant. The proposed system has a projected lifespan of 7-10 years, with very minimal **maintenance requirements.** Such durability and low operational costs make technology an attractive solution for urban communication infrastructure, public transportation systems, educational institutions, and corporate environments.

The technological ecosystem underlying this solution—combining Raspberry Pi 5's computational efficiency, ESP32 microcontrollers' low-power characteristics, and Seeed Studio's e-paper breakout boards—demonstrates a carefully curated approach to system design. Each component is selected not just for individual performance but for its synergistic potential within the broader technological framework.

Challenges remain: color display limitations at the current moment, and complex setup of initial systems. These limitations aside, there is potential in this system for continuous innovation. Further enhancements could come in terms of multi-language support, advanced content analytics, and adaptive display mechanisms responsive to contextual factors, such as time, location, and environmental conditions.

From an environmental perspective, this approach represents more than a technological solution—it is a statement about sustainable urban communication. The system dramatically reduces electronic waste, minimizes power consumption, and provides a flexible, updateable display infrastructure that aligns with broader goals of digital transformation and environmental responsibility.

Future Enhancements:

Technological Evolution and Framework Migration

The current project presents an extensive opportunity for technological advancement, with two promising web interface frameworks offering distinct advantages:

SvelteKit Migration

- **Performance Benefits:**
  - Compile-time framework with minimal runtime overhead
  - Built-in server-side rendering capabilities
  - Lightweight and highly efficient client-side interactions
- **Potential Improvements:**
  - Reduced load times
  - More responsive user interfaces
  - Enhanced developer experience with component-based architecture

OR FastAPI Integration

- **Backend Optimization:**
  - High-performance Python web framework
  - Automatic API documentation generation
  - Built-in type checking and validation

- **Scalability Features:**
    - Supports async programming models
    - Low overhead for high-concurrency scenarios
    - Easy integration with existing Python image processing scripts

Intelligent Content Management

- **AI-Powered Content Optimization**
    - Automatic content formatting
    - Contextual information insertion
    - Accessibility feature generation
- **Analytics and Usage Tracking**
    - Display interaction metrics
    - Content engagement analysis
    - Predictive maintenance algorithms

Communication Protocol Improvements

- **Enhanced MQTT Architecture**
    - Improved encryption mechanisms
    - More efficient bandwidth utilization
    - Support for larger payload transmissions

Further, for an even more enhanced performance, we can completely migrate the backend over to a Rust-based system, providing greater security and faster execution.

This is highly feasible, as ESP32 microcontroller also have integrations and support Rust, leading to a completely Rust-based system.

However, this comes with a downside. It greatly complicates the system, making it harder to manage.

Potential Research and Directions for Development

- Colour E-Paper Technology Integration
- LoRaWAN for extended wireless communication
- Edge AI capabilities for intelligent content management

Project Source

Server Structure

```
Server Root
    |
    ├ [node_modules] (nodejs package cache)
    ├ [processed] (save processed data)
    ├ [uploads] (save uploaded data)
    ├ mqtt-broadcast.py
    ├ package.json
    ├ process-script.py
    └ uploader.js
```

mqtt-broadcast.py:

```
from os import listdir
from os.path import isfile, join
import paho.mqtt.client as mqtt
import time

client = mqtt.Client("rpi_pubclient1")
flagger = 0

def on_connect(client, userdata, flags, rc):
    print("Connected to MQTT Server!")
    print("Beginning Message Broadcast")

def on_disconnect(client, userdata, rc):
    print("Disconnected from MQTT server")

def on_publish(client, userdata, mid):
    print(f"Message with mid : {mid} broadcasted successfully")

client.on_connect = on_connect
client.on_disconnect = on_disconnect
client.on_publish = on_publish

mypath = "processed"

file_list = [f for f in listdir(mypath) if isfile(join(mypath, f))]

print(file_list)

def send_img(path):
    print(f'Sending image {path}')

    f = open(f"{mypath}/{path}.jpg", "rb")  # 3.7kiB in same folder
    fileContent = f.read()
    byteArr = bytearray(fileContent)

    try:
        pubMsg = client.publish(
            topic='esp32/board1',
            payload=byteArr,
            qos=0,
        )
        pubMsg.wait_for_publish()
    except Exception as e:
        print(e)

while True :
    for i in file_list:
        client.connect("192.168.43.221", 1883)
        time.sleep(1)
        # Code to publish Message
        send_img(i)
        time.sleep(1)
        client.disconnect()
        time.sleep(5)
    temp_filelist = [f for f in listdir(mypath) if isfile(join(mypath, f))]
    file_list = temp_filelist if file_list != temp_filelist else file_list
    time.sleep(1)
```

process-script.py

```
import sys
import os
from PIL import Image
import numpy as np

display_size = (480, 800)

def process_uploaded_file(file_path):
    """
    Process the uploaded file with correct path handling
    """
    try:
        # Ensure the file path uses forward slashes and is absolute
        file_path = file_path.replace('\\', '/')

        # Basic file information logging
        print(f"Full uploaded file path: {file_path}")
        print(f"File name: {os.path.basename(file_path)}")
        print(f"File size: {os.path.getsize(file_path)} bytes")

        # Open the image using the full path
        img = Image.open(file_path)

        # Example processing - you can replace this with your specific logic
        print(f"Image size: {img.size}")
        print(f"Image format: {img.format}")
        print(f"Image mode: {img.mode}")

        print("beginning Image processing...")
        if img.format == "PNG":
            img = img.convert("RGB")

        ary = np.array(img)

        # Split the three channels
        r, g, b = np.split(ary, 3, axis=2)
        r = r.reshape(-1)
        g = r.reshape(-1)
        b = r.reshape(-1)

        # Standard RGB to grayscale
        bitmap = list(map(lambda x: 0.299 * x[0] + 0.587 * x[1] + 0.114 * x[2],
                          zip(r, g, b)))
        bitmap = np.array(bitmap).reshape([ary.shape[0], ary.shape[1]])
        bitmap = np.dot((bitmap > 128).astype(float), 255)
        im = Image.fromarray(bitmap.astype(np.uint8))

        print(f"Successfully converted image file into bitmap")

        resized_img = im.resize(display_size, Image.LANCZOS)

        print("Succesfully resized image")


resized_img.save(f'processed/{os.path.basename(file_path).split('.')[0]}.bmp')

        print(f"Successfully saved image as
{os.path.basename(file_path).split('.')[0]}.bmp")

        # Additional processing can go here
        # For example:
```

```
        # - Resize image
        # - Convert image format
        # - Apply filters
        # - Generate thumbnails

    except Exception as e:
        print(f"Error processing file: {e}")
        sys.exit(1)


if __name__ == "__main__":
    # Check if file path is provided as an argument
    if len(sys.argv) < 2:
        print("No file path provided")
        sys.exit(1)

    file_path = sys.argv[1]
    process_uploaded_file(file_path)
```

uploader.js

```
var http = require('http');
var formidable = require('formidable');
var fs = require('fs');
var path = require('path');
var { spawn } = require('child_process');

http.createServer(function (req, res) {
  if (req.url == '/fileupload' && req.method.toLowerCase() === 'post') {
    var form = new formidable.IncomingForm({
      uploadDir: 'C:/Users/basta/Documents/fileSite/uploads/',
      keepExtensions: true,
      maxFileSize: 50 * 1024 * 1024, // 50MB
      filter: function({name, originalFilename, mimetype}) {
        // Server-side image type validation
        const allowedMimeTypes = [
          'image/jpeg',
          'image/png',
          'image/gif',
          'image/webp',
          'image/bmp'
        ];
        return allowedMimeTypes.includes(mimetype);
      }
    });

    form.parse(req, function (err, fields, files) {
      if (err) {
        res.writeHead(500, {'Content-Type': 'text/html'});
        res.end(`
          <!DOCTYPE html>
          <html>
            <head>
              <title>Upload Error</title>
              <style>${getCSS()}</style>
            </head>
            <body>
              <div class="container error">
                <h1>Upload Failed</h1>
                <p>${err.message}</p>
                <a href="/" class="btn">Try Again</a>
              </div>
            </body>
          </html>
        `);
        return;
      }

      const uploadedFile = files.filetoupload || files['filetoupload[]'];
      const customFilename = fields.customFilename[0] || '';

      if (!uploadedFile) {
        res.writeHead(400, {'Content-Type': 'text/html'});
        res.end(`
          <!DOCTYPE html>
          <html>
            <head>
              <title>No File Uploaded</title>
              <style>${getCSS()}</style>
            </head>
            <body>
              <div class="container error">
```

```
                <h1>No File Selected</h1>
                <p>Please choose an image file to upload.</p>
                <a href="/" class="btn">Back to Upload</a>
              </div>
            </body>
          </html>
        `);
        return;
      }

      const fileToMove = Array.isArray(uploadedFile) ? uploadedFile[0] :
uploadedFile;

      if (!fileToMove.filepath) {
        res.writeHead(400, {'Content-Type': 'text/html'});
        res.end(`
          <!DOCTYPE html>
          <html>
            <head>
              <title>Upload Error</title>
              <style>${getCSS()}</style>
            </head>
            <body>
              <div class="container error">
                <h1>Upload Failed</h1>
                <p>Unable to process the file.</p>
                <a href="/" class="btn">Try Again</a>
              </div>
            </body>
          </html>
        `);
        return;
      }

      // Determine the final filename
      const originalExtension = path.extname(fileToMove.originalFilename ||
'');
      const finalFilename = customFilename
        ? (customFilename.endsWith(originalExtension)
          ? customFilename
          : customFilename + originalExtension)
        : fileToMove.originalFilename;

      const newFilename =
path.join('C:/Users/basta/Documents/fileSite/uploads/', finalFilename ||
'uploadedImage');

      fs.rename(fileToMove.filepath, newFilename, function (err) {
        if (err) {
          res.writeHead(500, {'Content-Type': 'text/html'});
          res.end(`
            <!DOCTYPE html>
            <html>
              <head>
                <title>Move Error</title>
                <style>${getCSS()}</style>
              </head>
              <body>
                <div class="container error">
                  <h1>File Move Failed</h1>
                  <p>${err.message}</p>
                  <a href="/" class="btn">Try Again</a>
                </div>
              </body>
            </html>
          `);
          return;
        }

        // Execute Python script after successful file upload
        const pythonProcess = spawn('python', [
          'process-script.py',
          newFilename  // Pass the full path of the uploaded file to the Python
script
        ]);

        // Optional: Handle Python script output
        pythonProcess.stdout.on('data', (data) => {
          console.log(`Python Script Output: ${data}`);
        });

        pythonProcess.stderr.on('data', (data) => {
          console.error(`Python Script Error: ${data}`);
        });

        pythonProcess.on('close', (code) => {
          console.log(`Python script exited with code ${code}`);
        });

        res.writeHead(200, {'Content-Type': 'text/html'});
        res.end(`
          <!DOCTYPE html>
          <html>
            <head>
              <title>Upload Success</title>
              <style>${getCSS()}</style>
            </head>
            <body>
              <div class="container success">
                <h1>Upload Successful!</h1>
                <p>Image saved as "${finalFilename}".</p>
                <a href="/" class="btn">Upload Another Image</a>
              </div>
            </body>
          </html>
        `);
      });
    });
  } else {
    // Serve the upload form
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.end(`
      <!DOCTYPE html>
      <html>
        <head>
          <title>Image Uploader</title>
          <style>${getCSS()}</style>
        </head>
        <body>
          <div class="container">
            <div class="upload-box">
              <h1>E-Paper Display System Image Uploader</h1>
              <form action="/fileupload" method="post" enctype="multipart/form-
  data">
                <div class="file-input-wrapper">
                  <input type="file" id="fileInput" name="filetoupload"
class="file-input"
```

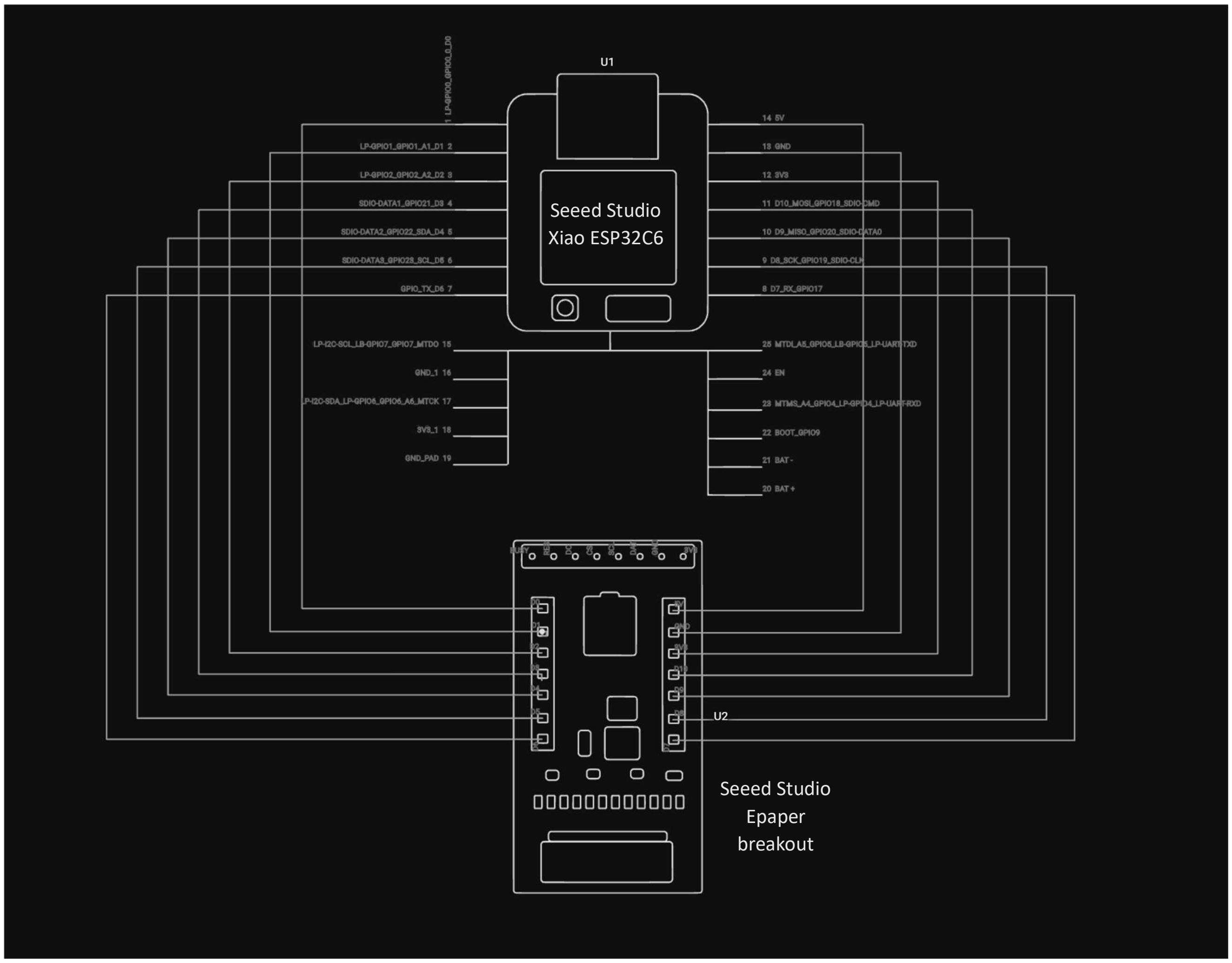
```
                         accept="image/jpeg, image/png, image/gif, image/webp,
image/bmp" multiple>
                  <label for="fileInput" class="file-input-label">
                    <span class="file-input-text">Choose Image</span>
                    <span class="file-input-btn">Browse</span>
                  </label>
                </div>
                <div id="file-chosen">No image chosen</div>

                <div id="image-preview-container" class="image-preview-
container">
                  <img id="image-preview" src="#" alt="Image preview"
class="image-preview" style="display:none;">
                </div>

                <div class="custom-filename-wrapper">
                  <label for="customFilename">Custom Filename
(optional):</label>
                  <input type="text" id="customFilename" name="customFilename"
class="custom-filename-input" placeholder="Enter custom filename">
                  <small>Leave blank to use original filename. Only images
allowed.</small>
                </div>
                <button type="submit" class="btn upload-btn" id="upload-btn"
disabled>Upload Image</button>
              </form>
            </div>
          </div>
          <script>
            const fileInput = document.getElementById('fileInput');
            const fileChosen = document.getElementById('file-chosen');
            const imagePreview = document.getElementById('image-preview');
            const uploadBtn = document.getElementById('upload-btn');

            fileInput.addEventListener('change', function(){
              const file = this.files[0];
              if(file) {
                // Validate file type
                const allowedTypes = ['image/jpeg', 'image/png', 'image/gif',
'image/webp', 'image/bmp'];
                if (!allowedTypes.includes(file.type)) {
                  alert('Please select a valid image file (JPEG, PNG, GIF,
WebP, BMP)');
                  this.value = ''; // Clear the input
                  fileChosen.textContent = 'No image chosen';
                  imagePreview.style.display = 'none';
                  uploadBtn.disabled = true;
                  return;
                }

                // File name and preview
                fileChosen.textContent = file.name;

                // Image preview
                const reader = new FileReader();
                reader.onload = function(e) {
                  imagePreview.src = e.target.result;
                  imagePreview.style.display = 'block';
                }
                reader.readAsDataURL(file);

                // Enable upload button
                uploadBtn.disabled = false;
              } else {
                fileChosen.textContent = 'No image chosen';
                imagePreview.style.display = 'none';
                uploadBtn.disabled = true;
              }
            });
          </script>
        </body>
      </html>
    `);
  }
}).listen(8080, () => {
  console.log('Server running on http://localhost:8080');
});

// getCSS() function remains unchanged from the previous code
function getCSS() {
  return `
    * {
      margin: 0;
      padding: 0;
      box-sizing: border-box;
    }
    body {
      font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto,
Oxygen, Ubuntu, Cantarell, 'Open Sans', 'Helvetica Neue', sans-serif;
      background-color: #f4f4f4;
      display: flex;
      justify-content: center;
      align-items: center;
      min-height: 100vh;
      line-height: 1.6;
    }
    .container {
      background-color: white;
      border-radius: 10px;
      box-shadow: 0 4px 6px rgba(0,0,0,0.1);
      width: 100%;
      max-width: 500px;
      padding: 30px;
      text-align: center;
    }
    .upload-box {
      display: flex;
      flex-direction: column;
      align-items: center;
    }
    h1 {
      margin-bottom: 20px;
      color: #333;
    }
    .file-input-wrapper {
      position: relative;
      width: 100%;
      margin-bottom: 15px;
    }
    .file-input {
      position: absolute;
      width: 1px;
      height: 1px;
      padding: 0;
      margin: -1px;
      overflow: hidden;
      clip: rect(0,0,0,0);
      white-space: nowrap;
      border: 0;
    }
    .file-input-label {
      display: flex;
      justify-content: space-between;
      align-items: center;
      width: 100%;
      padding: 10px 15px;
      border: 2px dashed #3498db;
      border-radius: 5px;
      cursor: pointer;
      transition: all 0.3s;
    }
    .file-input-label:hover {
      border-color: #2980b9;
    }
    .file-input-text {
      color: #7f8c8d;
    }
    .file-input-btn {
      background-color: #3498db;
      color: white;
      padding: 5px 10px;
      border-radius: 3px;
    }
    #file-chosen {
      margin-bottom: 15px;
      color: #7f8c8d;
    }
    .image-preview-container {
      margin-bottom: 15px;
      max-height: 250px;
      display: flex;
      justify-content: center;
    }
    .image-preview {
      max-width: 100%;
      max-height: 250px;
      object-fit: contain;
      border-radius: 5px;
      box-shadow: 0 2px 4px rgba(0,0,0,0.1);
    }
    .btn {
      display: inline-block;
      background-color: #3498db;
      color: white;
      padding: 10px 20px;
      text-decoration: none;
      border-radius: 5px;
      transition: background-color 0.3s;
      border: none;
      cursor: pointer;
      margin-top: 15px;
    }
    .btn:disabled {
      background-color: #bdc3c7;
      cursor: not-allowed;
    }
    .btn:hover:not(:disabled) {
      background-color: #2980b9;
    }
    .upload-btn {
      width: 100%;
    }
```

```
    .custom-filename-wrapper {
      margin-bottom: 15px;
      text-align: left;
    }
    .custom-filename-input {
      width: 100%;
      padding: 10px;
      margin-top: 5px;
      border: 1px solid #ddd;
      border-radius: 4px;
    }
    .custom-filename-wrapper small {
      display: block;
      color: #7f8c8d;
      margin-top: 5px;
      font-size: 0.8em;
    }
  `;
}
```

ESP32 Client Structure

```
Client Root
    |
    ├ [lib] (CircuitPython packages)
    └ code.py
```

Client-side microcontroller structure (pcb diagram) :


U1
1 LP-GPIO0_GPIO0_0_D0
LP-GPIO1_GPIO1_A1_D1 2
LP-GPIO2_GPIO2_A2_D2 3
SDIO-DATA1_GPIO21_D3 4
SDIO-DATA2_GPIO22_SDA_D4 5
SDIO-DATA3_GPIO23_SCL_D5 6
GPIO_TX_D6 7
14 5V
13 GND
12 3V3
11 D10_MOSI_GPIO18_SDIO-CMD
10 D9_MISO_GPIO20_SDIO-DATA0
9 D8_SCK_GPIO19_SDIO-CLK
8 D7_RX_GPIO17
Seeed Studio Xiao ESP32C6
LP-I2C-SCL_LB-GPIO7_GPIO7_MTDO 15
GND_1 16
LP-I2C-SDA_LP-GPIO6_GPIO6_A6_MTCK 17
3V3_1 18
GND_PAD 19
25 MTDI_A5_GPIO5_LB-GPIO5_LP-UART-TXD
24 EN
23 MTMS_A4_GPIO4_LP-GPIO4_LP-UART-RXD
22 BOOT_GPIO9
21 BAT -
20 BAT +
BUSY
RES
DC
CS
SCL
DAT
GND
3V3
U2
Seeed Studio Epaper breakout

Code.py:

```
import time
import board
import displayio
import neopixel
import os
import ssl
import socketpool
import wifi
import adafruit_minimqtt.adafruit_minimqtt as MQTT

try:
    from i2cdisplaybus import I2CDisplayBus
except ImportError:
    from displayio import I2CDisplay as I2CDisplayBus

import terminalio

# can try import bitmap_label below for alternative
from adafruit_display_text import label
import adafruit_displayio_sh1107

displayio.release_displays()
# epaper_reset = board.D9

# Use for I2C
i2c = board.I2C()  # uses board.SCL and board.SDA
display_bus = I2CDisplayBus(i2c, device_address=0x3C)

pixel = neopixel.NeoPixel(board.NEOPIXEL, 1)

main_feed = "esp32/board1"

WIDTH = 128
HEIGHT = 64
BORDER = 2

display = adafruit_displayio_sh1107.SH1107(
    display_bus, width=WIDTH, height=HEIGHT, rotation=0
)

time.sleep(1)

# Make the display context
splash = displayio.Group()
display.root_group = splash

color_bitmap = displayio.Bitmap(WIDTH, HEIGHT, 1)
color_palette = displayio.Palette(1)
color_palette[0] = 0xFFFFFF  # White

bg_sprite = displayio.TileGrid(color_bitmap, pixel_shader=color_palette, x=0,
y=0)
splash.append(bg_sprite)

# Draw a smaller inner rectangle in black
inner_bitmap = displayio.Bitmap(WIDTH - BORDER * 2, HEIGHT - BORDER * 2, 1)
inner_palette = displayio.Palette(1)
inner_palette[0] = 0x000000  # Black
inner_sprite = displayio.TileGrid(
    inner_bitmap, pixel_shader=inner_palette, x=BORDER, y=BORDER
)
splash.append(inner_sprite)

text2 = " "
text_area2 = label.Label(
    terminalio.FONT, text=text2, scale=1, color=0xFFFFFF, x=6, y=15
)
splash.append(text_area2)



# Define callback methods which are called when events occur
# pylint: disable=unused-argument, redefined-outer-name
def connected(client, userdata, flags, rc):
    # This function will be called when the client is connected
    # successfully to the broker.
    print(f"Connected to MQTT Server! Listening for topic changes on
{main_feed}")
    # Subscribe to all changes on the onoff_feed.
    client.subscribe(main_feed)


def disconnected(client, userdata, rc):
    # This method is called when the client is disconnected
    print("Disconnected from mqtt!")


def message(client, topic, message):
    # This method is called when a topic the client is subscribed to
    # has a new message.

    text_area2.text = message.format(time.monotonic())
    print("recieved message on esp/board1 : {0}".format(message))
    pixel.fill((0, 0, 255))
    time.sleep(1)
    pixel.fill((0,0,0))


# Create a socket pool
pool = socketpool.SocketPool(wifi.radio)
ssl_context = ssl.create_default_context()

mqtt_client = MQTT.MQTT(
    broker="192.168.43.221",
    port=1883,
    socket_pool=pool,
    ssl_context=ssl_context,
)

# Setup the callback methods above
mqtt_client.on_connect = connected
mqtt_client.on_disconnect = disconnected
mqtt_client.on_message = message


# Connect the client to the MQTT broker.
print("Connecting to MQTT Server")
mqtt_client.connect()



while True:
    mqtt_client.loop(timeout=1)
```