VIETNAM ACADEMY OF SCIENCE AND TECHNOLOGY
UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI

Course: Network Programming

FINAL REPORT TERM PROJECT

DISTRIBUTED IMAGE PROCESSING SYSTEM

Project leader: Nguyen Thi Minh Nguyet - 22BA13241

Class: B2

Major: Cyber Security

Team members: Nguyen Quynh Trang - 22BA13303

Nguyen Ky Khai - 22BA13168

Vu Gia Bach - 22BA13039

Vuong Ngoc Duy - 22BA13104

Tran Anh Dung - 22BA13090

Nguyen Hai An - 22BA13003

Hanoi, May 2024

A. Introduction:

This project presents a Distributed Image Processing System designed to convert high-resolution color images into grayscale using a parallel and networked approach. The system leverages multiple worker clients to perform image processing tasks simultaneously, showcasing the benefits of distributed computing.

- **Input**: A high-resolution color image in .jpg or .png format, uploaded via the user interface.
- **Output**: A grayscale version of the same image with identical resolution, processed in a distributed manner using multiple clients.

B. Overview

I. Workflow

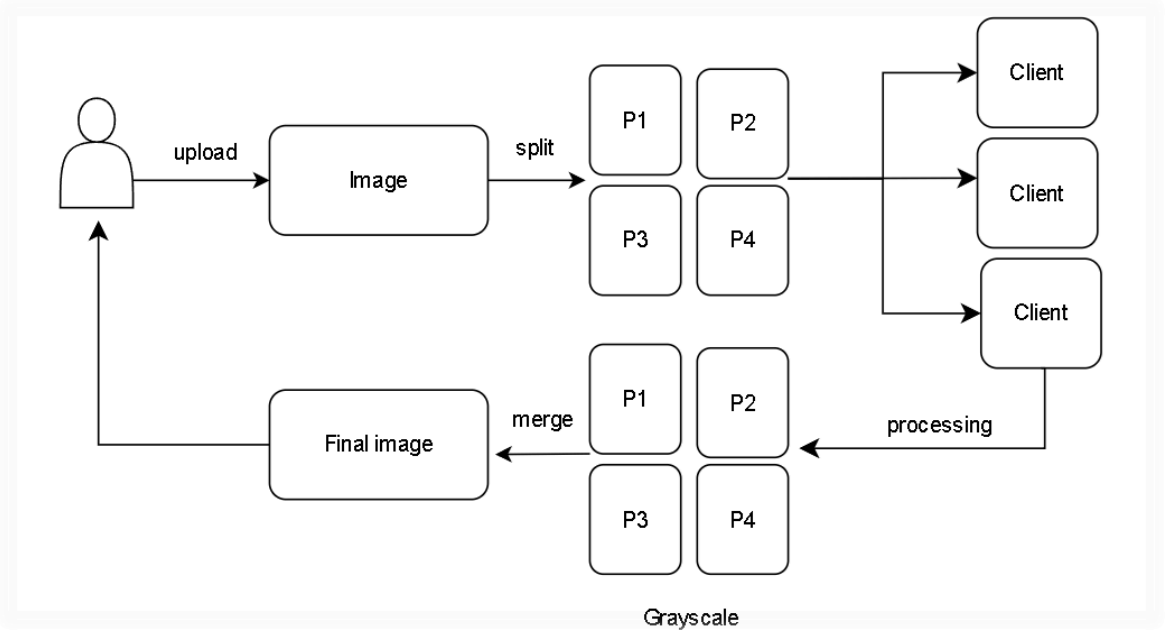

Figure 1: Multi-Client Grayscale Processing Workflow

The system receives a high-resolution color image from the user, then splits it into smaller segments. Each segment is sent to a separate client for grayscale processing in parallel. After processing, the grayscale parts are merged to form the final image, which is returned to the user. This workflow demonstrates how image processing can be distributed across multiple clients using socket-based communication.

II. Technology

1. Python

Main programming language used to implement all backend logic, including socket communication, image processing (via PIL), threading, and orchestration logic.

2. Flask

Lightweight web framework used to build the REST API and serve as the bridge between front-end and back-end.

3. Sockets

Used for TCP communication between the Master Server and Worker Clients.

4. Docker

Can be used to containerize the Master, Worker, and Flask services for ease of deployment and scalability.

III. Implementation

1. Architecture

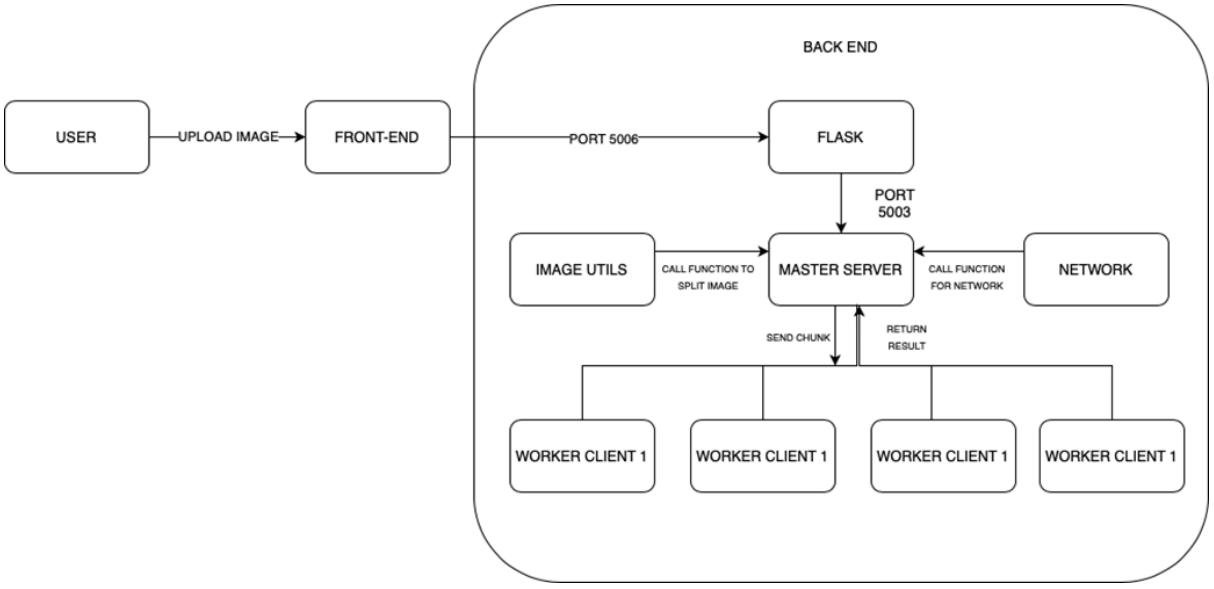

Figure 2: System Architecture of the Distributed Image Processing Platform

The user uploads a high-resolution image via the frontend, which is handled by the Flask backend through port 5006. The master server coordinates the processing task by calling utility functions to split the image and distribute chunks to multiple worker clients via socket communication on port 5003. Each worker processes its assigned segment and returns the result for merging into the final grayscale image. The system

demonstrates modular design, socket-based networking, and parallel image processing.

2. Component details

User:

The user serves as the entry point of the system by initiating the image processing workflow through an upload action. Once submitted, the image is transmitted directly to the front-end for further handling.

Front-end:

The front-end functions as the bridge between the user and the back-end system. It communicates with the Flask server over **port 5006**, facilitating data exchange between the two layers. This interface is implemented in the app.py file, which defines the Flask application responsible for handling incoming requests and forwarding them to the back-end logic.

Flask (Back-end):

The Flask server acts as the primary back-end component. It listens for image uploads on the /convert endpoint. Upon receiving an image, Flask passes it to the Master Server for distributed processing. Once all processing is completed, the Flask server collects the grayscale result and returns it to the front-end for display to the user.

Master Server:

```python
def process_image_distributed_bytes(img_bytes: bytes) -> bytes:
    """
    Distribute image processing across workers via master-server:
    1) Convert bytes→Image
    2) Split image into segments
    3) Send each segment to master-server; collect processed segments
    4) Reconstruct full grayscale image and return bytes
    """
    # 1) Bytes to PIL Image
    img = bytes_to_image(img_bytes)

    # 2) Split into segments
    segments = split_image(img, NUM_WORKERS)

    gray_segments: List[bytes] = []
    # 3) Exchange segments with master-server
    for seg_bytes in segments:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect((HOST, MASTER_PORT))
            send_bytes(s, seg_bytes)
            processed = recv_bytes(s)
            gray_segments.append(processed)

    # 4) Reconstruct full image
    result_bytes = reconstruct_image(gray_segments)
    return result_bytes
```

Figure 3: Master Server Code for Coordination and Distribution

The Master Server coordinates the distributed image processing task.

- It listens for connections from multiple Worker Clients and manages the distribution of work.
- It calls the **split_image** function from the image utilities module to divide the original image into segments. These segments are then dispatched to worker clients for processing. Once the grayscale segments are returned, the Master Server invokes a reconstruction function—also from **image_utils**—to merge the results into a final grayscale image.

Worker Client:

```
while True:
    try:
        # Kết nối về Master-Server
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect((MASTER_HOST, MASTER_PORT))
            # Nhận segment
            data = recv_bytes(s)
            # Chuyển PIL image sang gray
            img = bytes_to_image(data).convert('L')
            out = image_to_bytes(img)
            # Gửi lại kết quả
            send_bytes(s, out)
    except ConnectionRefusedError:
        # Master chưa sẵn sàng → chờ rồi thử lại
        time.sleep(1)
    except Exception as e:
        print("Worker error:", e)
        time.sleep(1)
```

Figure 4: Worker Client Code for Grayscale Chunk Processing

Each Worker Client is responsible for processing a single image segment. Upon receiving a chunk from the Master Server, the worker applies a grayscale transformation to the image data and returns the result back to the server. Each worker handles exactly one chunk per connection session.

Network Communication and Protocol Design:

- Communication between Master and Workers is handled via **TCP sockets**.
- A length-prefixed format is used to ensure accurate message boundaries for binary data.

- The Master acts as a sender of image chunks, uses a thread-per-connection model to support concurrency and scalability.

```python
def send_bytes(conn: socket.socket, data: bytes) -> None:
    """
    Gửi data đã cho qua socket kèm header 8 byte biểu diễn độ dài dữ liệu.
    """
    length = len(data)
    conn.sendall(length.to_bytes(8, 'big'))
    conn.sendall(data)
```

Figure 5: Master - send_bytes function

- Each Worker acts as the receiver first, then sender to establish a short-lived, bidirectional connection per task.

```python
def recv_bytes(conn: socket.socket) -> bytes:
    """
    Nhận dữ liệu được gửi kèm header 8 byte độ dài.
    """
    length_bytes = _recvall(conn, 8)
    length = int.from_bytes(length_bytes, 'big')
    return _recvall(conn, length)
```

Figure 6: Worker - recv_bytes function

3. Image Utilities

Image Splitting Function Overview

```python
def split_image(img: Image.Image, num_parts: int) -> List[bytes]:
    """
    Split the input image into `num_parts` horizontal stripes,
    returning a list of raw PNG bytes for each segment.
    """
    width, height = img.size
    part_height = height // num_parts
    segments: List[bytes] = []
    for i in range(num_parts):
        top = i * part_height
        bottom = (i + 1) * part_height if i < num_parts - 1 else height
        box = (0, top, width, bottom)
        segment = img.crop(box)
        segments.append(image_to_bytes(segment))
    return segments
```

Figure 7: Image Splitting Function

The function split_image(image: Image.Image, num_parts: int) -> List[Image.Image] is designed to divide a given image into multiple equal horizontal segments. This is particularly useful in distributed image processing systems, where each segment can be independently processed by a worker node in parallel.

This approach eliminates inter-segment dependency, allowing full utilization of distributed or multi-threaded infrastructure, and significantly reduces overall image processing time.

Function Definition

```
def split_image(img: Image.Image, num_parts: int) -> List[bytes]:
```

Figure 8: Definition for the split_image function

- Parameters:

  image: A PIL Image object representing the original input image.

  num_parts: The number of horizontal segments to divide the image into.

- Returns:

  A list of PIL.Image segments, each representing a horizontal slice of the original image.

- Step-by-Step Operation:
  1. Retrieve Image Dimensions: Get the width and height of the original image.

```
width, height = img.size
part_height = height // num_parts
```

Figure 9: Retrieve Image Dimensions

  2. Initialize segment list: Create an empty list to store the output segments.

```
segments: List[bytes] = []
```

Figure 10: Initialize segment list

  3. Iterate and Slice Image: Loop through the number of parts and use .crop() to extract each horizontal segment. Append each segment to the list.

```python
for i in range(num_parts):
    top = i * part_height
    bottom = (i + 1) * part_height if i < num_parts - 1 else height
    box = (0, top, width, bottom)
    segment = img.crop(box)
    segments.append(image_to_bytes(segment))
return segments
```

Figure 11: Iterate and Slice Image

4. Socket communication

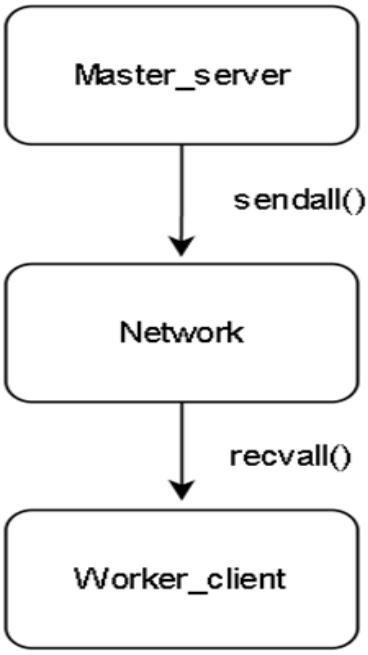

Figure 12: Data Transmission Flow

- Each Worker establishes a bidirectional TCP socket with the Master.
- Communication follows a length-prefixed protocol: **send_bytes()** and **recv_bytes()** functions ensure accurate message boundaries.
- Workers handle one chunk per session and reconnect for new tasks.
- The system uses thread-per-connection on the Master Server to support concurrency and scalability.

C. Result

I. Image Comparison

Original Image (Before)

Grayscale Image (After)

The image on the left shows the original high-resolution input in color, while the image on the right displays the result after distributed grayscale processing. The system is designed to support large image resolutions, including 4K and 8K, by dividing the image into smaller segments and distributing the workload across multiple worker clients. This architecture enables efficient parallel processing and ensures scalability for high-resolution inputs.

II. Performance metrics

N.Client	Avg Send (s)	Avg Proc (s)	Avg Recv (s)	Speedup (×)	Efficiency (%)
1	0.3329	1.8943	3.1962	1.00	100
4	0.1456	1.8282	3.3011	1.03	25.7
8	0.0429	1.9610	3.5336	0.98	12.2

Table: Performance Comparison of Image Processing with Varying Client Count

Observation

- System shows minimal speedup with 4 clients and regresses with 8.
- Overhead due to networking and synchronization limits scalability.
- Efficiency drops sharply as more clients are added.

D. Conclusion

This project successfully demonstrates a distributed approach to image processing. The system architecture promotes modularity and clarity, enabling:

- Parallel processing of image segments.
- Scalability through a stateless worker design.
- Robust network communication via the TCP protocol.

However, performance evaluation indicates that increased parallelism does not always guarantee improved performance, primarily due to communication latency and coordination overhead.

To enhance system efficiency, the following improvements are recommended:

- Batching images or optimizing chunk sizes to minimize synchronization costs.
- Exploring alternative transport protocols, such as UDP combined with a reliability layer, to reduce overhead.
- Compressing or downscaling images before transmission to lower bandwidth and processing load.