Distributed System for Batch Image Processing

Escalona, Jose Miguel
College of Computer Science
De La Salle University
2401 Taft Avenue, Manila, Philippines
jose_miguel_escalona@dlsu.edu.ph

Estebal, Eidrene Glena
College of Computer Science
De La Salle University
2401 Taft Avenue, Manila, Philippines
eidrene_glena_estebal@dlsu.edu.ph

Fortiz, Patrick Ryan
College of Computer Science
De La Salle University
2401 Taft Avenue, Manila, Philippines
patrick_ryan_fortiz@dlsu.edu.ph

***Abstract*— This project focuses on developing and demonstrating distributed systems principles by creating a scalable server application. The application acts as a bulk image processor, enhancing images through brightness, contrast, and sharpness adjustments. The distributed system allows flexibility in setting the number of machines for execution and employs RabbitMQ as a message broker for client-server communication. Utilizing OpenCV and pika for image processing and messaging, the system adopts a MIMD architecture, introducing complexity for enhanced efficiency. Communication follows a Publish/Subscribe model for client-to-server and a Topic Exchange model for server-to-client. The project addresses optimizations, limitations, and issues, such as dedicated connections, multiprocessing, file size constraints, and startup challenges. Benchmarking with 'Tokyo' and 'Crops' datasets reveals that the two-machine configuration outperforms the one-machine setup, showing a 19.91%-time reduction for the 'Tokyo' dataset and 15.55% for 'Crops.' These results underscore the system's scalability and improved efficiency, showcasing its potential for diverse image processing tasks in a distributed environment.**

Keywords—distributed systems, image processing, rabbitMQ, OpenCV, pika

I. Introduction

The project aims to build and demonstrate the concepts of distributed systems by means of building a scalable server application which has the supposed benefit of improved performance while maintaining reliability and data integrity. The requirement of the project is to build a bulk image processor where it takes in a set of images which are processed accordingly with adjustments in brightness, contrast, and sharpness. Being a scalable distributed system, it is also optional for the distributed system to be able to set the number of machines that it can use in its execution, otherwise it can simply execute on-scale on all systems available.

II. Program Implementation

A. System Specifications

Following the project objectives, the system specifications, which follows the project requirements, are set as follows:

- Input
  - Input & Output Paths
  - Brightness, Contrast, and Sharpness Factors
- Process: Image Processing
- Output:
  - Processed Files
  - Report File

The report file contains the necessary information about the program's runtime and performance such as the Input and Output Paths, the Number of Images Processed, and the Number of Machines Used. The

B. System Dependencies and Middleware

The program uses a Message Broker called RabbitMQ as a middleware between the client and the server. RabbitMQ was selected as the primary middleware since it was determined that it is already sufficient for the needs of client-to-server communication (i.e., message passing), as well as job coordination and synchronization.

For the Python libraries, both Client and Server applications use OpenCV and pika for image processing and message passing through RabbitMQ respectively.

C. System Design

The overall System Design follows the MIMD architecture which allows for full-efficiency on a per-machine basis. However, this introduces complexity as there are different threads running at the same time which require coordination with each other. Nevertheless, the advantages of such an implementation outweigh the disadvantages.

1) Client

The Client is designed to run on at least four (4) threads, each having its own function. It is responsible for sending the raw images alongside other metadata needed for processing (adjustment factors & output path) and origin (i.e., Client's UUID). It is also responsible for processing the messages it receives from the Topic Exchange that is addressed to it. In order to allow for selective consumption of the messages, the Client binds its Client's UUID as the 'routing_key' or Topic of to the Topic Message Exchange of RabbitMQ.

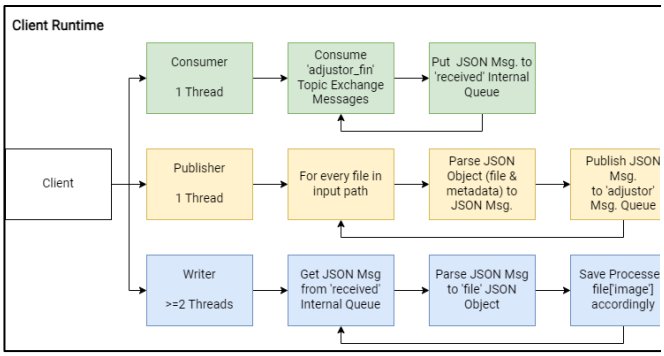

Fig. 1 Client Runtime Design

The following are the specific details of the components of the Client.

a) Consumer

The Consumer component of the Client listens and consumes any message that the 'adjustor_fin' Topic Exchange has for the client (one that matches the Client's UUID) which is then queued on the 'received' queue of the Client for further processing.

b) Publisher

The Publisher component of the Client parses the image and metadata to a JSON message and sends it accordingly to the server via the Message Broker's 'adjustor' Queue.

c) Writer

The Writer component of the Client monitors the 'received' internal queue of the Client where it will continually get available queued messages which will be parsed from a JSON message into an OpenCV image object and metadata which will be accordingly written to the disk via the specified output path.

2) Server

The Server is designed to run on at least n+2 threads (n being the total available threads in the system), each having its own function. It is responsible for consuming and parsing the messages from the client, processing the image (according to the adjustment factors included in the message), and sending the processed image back to the appropriate Client through its UUID included in the received message.

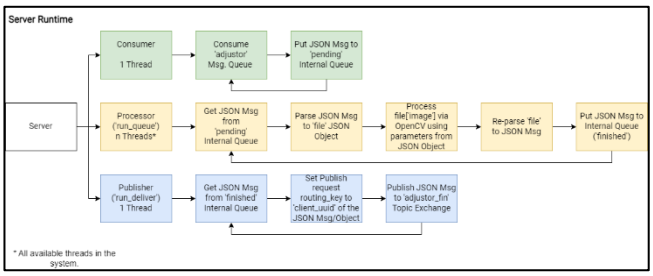

Fig. 2 Server Runtime Design

The following are the specific details of the components of the Server.

a) Consumer

The Consumer component of the Server listens and consumes any messages that are available to be consumed from the Message Broker. It then queues the received/consumed message from the Message Broker to the 'pending' queue of the Server which will be used by other components of the Server application later on.

b) Processor

The Processor component of the Server monitors the 'pending' queue and gets every available entry in the queue (by FIFO pattern) for parsing to JSON object which will process the image accordingly. The metadata and the processed image will then be parsed back to a JSON String which will be queued in the 'finished' queue in the Server application to be sent back to its origin Client instance (via Client UUID). The Processor component of the Server is also the most heavily threaded component of the Server application, as it runs on all available threads in the system for increased efficiency.

c) Publisher

The Publisher component of the Server monitors the 'finished' queue and delivers any messages available from the 'finished' queue to the appropriate Client through the Message Broker.

D. Message Passing Design

It was determined during development that the Message Passing Design will be different for each direction. While both designs use a Message Broker (i.e., RabbitMQ) for its message passing, it does so with differences depending on the direction of the communication (i.e., either Client-to-Server or Server-to-Client).

1) Client-to-Server

Client-to-Server communication uses the Publish/Subscribe Queue model of RabbitMQ where it uses the concept of queue data structures [1,6]. This allows for multiple Clients to simply publish or queue their messages while at the same time the Servers are able to retrieve or consume these messages in an asynchronous manner. This asynchronous implementation allows for the Client to feel more responsive, thus allowing for the capability of bulk processing.

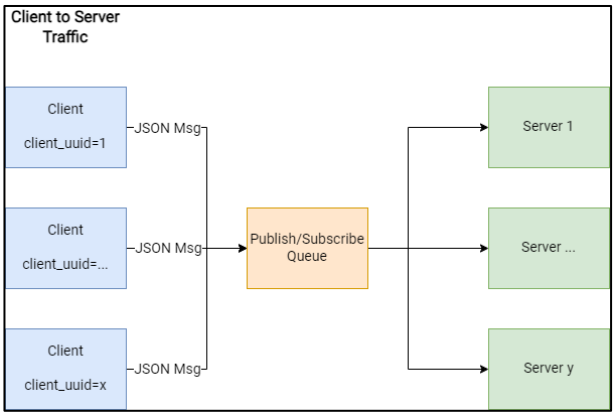

Fig. 3 Client to Server Message Delivery

2) Server-to-Client

Server-to-Client communication uses the Topic Exchange model of RabbitMQ where it allows for a more selective sending of messages to the appropriate Client [1-4]. The 'routing_key' or the Topic used will be set as the Client UUID where the Client will simply consume messages that match that of its own Client UUID.

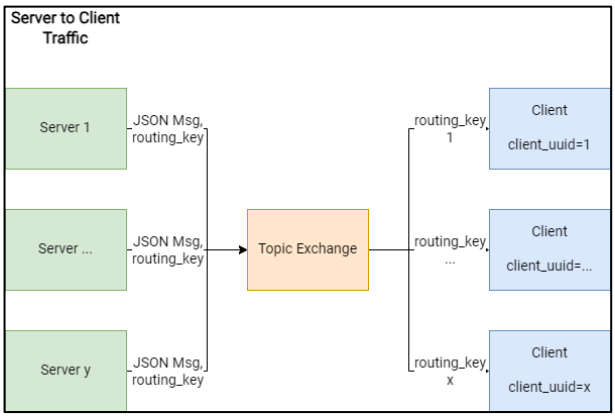

Fig. 4 Server to Client Message Delivery

E. Optimizations, Limitations, & Issues

1) Dedicated Connections and Channels

Due to the nature of both the Client and the Server applications of needing for two-way communications, the application was implemented to have a dedicated send and receive connection and channel to accommodate each direction of communication.

This was due to performance and reliability issues found on early implementations of the distributed system where there is a bottleneck during transmission (either send or receive) between the Client or Server and the Message Broker which oftentimes result to timeouts. It was attributed to the fact that the connection itself uses a Blocking Connection which as the name suggests is not asynchronous despite the asynchronous design (for both send and receive) of the distributed system.

2) Multiprocessing

Due to the single-threaded nature of OpenCV, it was decided to increase the per-Server scalability [5,13] of the Server application by implementing Queues and Multiprocessing to allow for multiple instances of OpenCV to process images it receives through the Server's 'pending' queue. However, this has the disadvantage of being difficult to containerize as the Server application was designed and implemented to utilize all available threads in the system. A similar mechanism was also implemented in the Client application for file writing of received messages.

3) File Size Limitations

The distributed system was added with a limitation to only accept files under or equal to 5MB, which is common in most online bulk image processing services. This limitation was introduced due to message passing-related issues associated with large data where the Message Broker is limited in handling such sizes of data (as per distributed system's configuration) which oftentimes result to corrupted messages to be delivered in both directions.

4) Start-up Issues

It was determined that the initial instances of the Server will encounter issues on a newly launched RabbitMQ instance which means that it will take a while (i.e., a 'warmup') before the entire distributed system to be reliably functional when being under load by a Client. It can be remedied by restarting the instance Server instance from where other instances can also be run (i.e., to increase scalability).

F. JSON Object Structure

The JSON message (called 'file') used throughout the entire process is structured as shown in the table below.

TABLE I. JSON Object Structure

Key	Raw Data Type	Description
input	String	Input folder path
filename	String	Filename of the image
output	String	Output folder path
brightness	Integer	Brightness factor (0-100)
contrast	Integer	Contrast factor (0-100)
sharpness	Integer	Sharpness (0-100)
image	OpenCV Mat Class Parsed to String via pickle.dumps() → Base64 Encode → ASCII Decode	String-parsed image data
client_uuid	String	Client UUID

III. Result

A. Benchmarking

1) Benchmark Datasets

The two datasets used for benchmarking the distributed system, 'Tokyo' and 'Crops'[12]. 'Tokyo' dataset represents heavy or large image files which are common for photography use cases. 'Crops' datasets represent light or small image files which are common for computer vision uses cases. Further details about the datasets are as shown below.

TABLE II. Benchmark Datasets

Attribute	Tokyo	Crops
File Size Range	367KB to 12.9MB	2.9KB to 1.8MB
Total File Size	134.2MB	76.8MB
Total Count	55 Images	773 Images
Acceptable File Count (File Sizes <= 5MB)	49 Images (89.09%)	773 Images (100.00%)

2) Benchmark Setup

The computer setup used for benchmarking is as shown in the table below. The host system allows for up to two (2) VMs to be running at the same time, hence the benchmarking process allows for the minimum of two (2) instances of the Server application running in accordance with the requirements given for this project.

TABLE III Benchmark Setup

Component	Server (Guest VM)	Client (Host)	RabbitMQ (Host)
CPU Threads	4	8	
RAM	8192MB	2048MB	

Benchmarking was also done in multiple configurations with multiple runs. The two (2) configurations that were used for the benchmarking are one (1) and two (2) machines which represent the scalability of the distributed system in terms of performance. To isolate cases of negative performance fluctuations, the benchmark will be executed in four (4) runs for each config on both datasets where the average will be used to determine the overall performance of the distributed system.

B. Benchmark Results

Table IV presents the raw benchmark results for the distributed system. It may be seen that the time it takes to process images from the 'Tokyo' dataset is much longer than the 'Crops' dataset. As the image file size for each sample in the 'Tokyo' dataset is considerably larger than those from 'Crops', and these require more data to be transferred between both client and server. On the server side, larger images may also take longer to load and process through OpenCV.

TABLE IV - Benchmark Results

Number of Machines	Run	Time (s) Tokyo Dataset	Time (s) Crops Dataset
1	1	67.8779	38.1426
	2	45.0550	34.7252
	3	45.8793	33.6548
	4	48.2554	35.0385
2	1	39.5424	29.7509
	2	39.7682	29.6472
	3	46.0564	30.4623
	4	40.4689	29.4173

Table V averages each run instance found in Table IV for the one and two machine configurations. The one-machine configuration showed an average time of 51.7669 seconds for the 'Tokyo' dataset, and 35.3093 seconds for the 'Crops' dataset. The two-machine configuration on the other hand, displayed an average time of 41.4590 seconds for the 'Tokyo' dataset, and 29.8194 seconds for the 'Crops' dataset.

TABLE V Average Time

No. of Machines	Average Time (s) Tokyo Dataset	Average Time (s) Crops Dataset
1 (4 Threads Total)	51.7669	35.3093
2 (8 Threads Total)	41.4590	29.8194

Table VI shows how much of the average time is reduced by the two-machine configuration compared to the one-machine configuration. The time reduction for the 'Tokyo' dataset was 19.91% and 15.55% for the 'Crops' dataset. A time reduction value above 0 for both datasets confirms that an increase in performance is evident as more computational resources become available (i.e., adding more server machines to process client requests).

TABLE VI Time Reduction

Time Reduction Tokyo Dataset	Time Reduction Crops Dataset
19.91%	15.55%

IV. Conclusion

Distributed techniques were used to create a scalable server application for bulk image processing. Threading, multiprocessing, and optimized message passing designs were utilized to enhance scalability and efficiency. Despite file size limitations and start-up issues, results demonstrated a reduction in average processing time when transitioning from a one-machine to a two-machine configuration, confirming the effectiveness of distributed techniques in optimizing the performance of the bulk image processing system.

References

[1] "AMQP 0-9-1 Model Explained — RabbitMQ," RabbitMQ. https://www.rabbitmq.com/tutorials/amqp-concepts.html (accessed Nov. 19, 2023).

[2] "Part 4: RabbitMQ Exchanges, routing keys and bindings," CloudAMQP, 2019. https://www.cloudamqp.com/blog/part4-rabbitmq-for-beginners-exchanges-routing-keys-bindings.html (accessed Nov. 19, 2023).

[3] "RabbitMQ Topic Exchange Explained," CloudAMQP, 2022. https://www.cloudamqp.com/blog/rabbitmq-topic-exchange-explained.html (accessed Nov. 19, 2023).

[4] "RabbitMQ tutorial," Topics — RabbitMQ. https://www.rabbitmq.com/tutorials/tutorial-five-python.html (accessed Nov. 19, 2023).

[5] Pankaj, "Python Multiprocessing Example," DigitalOcean, Aug. 03, 2022. Accessed: Nov. 19, 2023. [Online]. Available: https://www.digitalocean.com/community/tutorials/python-multiprocessing-example

[6] "RabbitMQ tutorial," Work Queues — RabbitMQ. https://www.rabbitmq.com/tutorials/tutorial-two-python.html (accessed Nov. 19, 2023).

[7] K41F4r, "Sending OpenCV image in JSON," Stack Overflow. https://stackoverflow.com/questions/55892362/sending-opencv-image-in-json/55900422#55900422 (accessed Nov. 19, 2023).

[8] pika, “pika/examples/basic_consumer_threaded.py at 1.0.1 · pika/pika,” GitHub. https://github.com/pika/pika/blob/1.0.1/examples/basic_consumer_threaded.py (accessed Nov. 19, 2023).

[9] S. A. Khan, “How to change the contrast and brightness of an image using OpenCV in Python?,” Tutorialspoint, 2023. https://www.tutorialspoint.com/how-to-change-the-contrast-and-brightness-of-an-image-using-opencv-in-python (accessed Nov. 19, 2023).

[10] “How to Sharpen an Image with OpenCV,” How to use OpenCV, 2023. https://www.opencvhelp.org/tutorials/image-processing/how-to-sharpen-image/ (accessed Nov. 19, 2023).

[11] “Python OpenCV cv2.imwrite method,” GeeksforGeeks, Aug. 05, 2019. Accessed: Nov. 19, 2023. [Online]. Available: https://www.geeksforgeeks.org/python-opencv-cv2-imwrite-method/

[12] M. W. Azam, “Agricultural crops image classification,” Kaggle. https://www.kaggle.com/datasets/mdwaquarazam/agricultural-crops-image-classification (accessed Nov. 19, 2023)

[13] “Difference Between Multithreading vs Multiprocessing in Python,” GeeksforGeeks, Apr. 24, 2020. Accessed: Nov. 23, 2023. [Online]. Available: https://www.geeksforgeeks.org/difference-between-multithreading-vs-multiprocessing-in-python/