Technical Report

Distributed Project

Table of Contents

Project Overview

This project is a Peer-to-Peer Instagram-like application based on a distributed cloud for encryption in Rust. It is developed based on a literature surveying of various algorithms, class discussions, and implementation of distributed systems principles such as distributed election, multithreading, interprocess communication, etc. In the following report, our group is going to illustrate the system architecture and engineering decisions that we made during the development of this project.

System Features

The system comprises two main components: the cloud servers and the P2P client communication.

2.1 Cloud Servers Feature

- Load balancing in the cloud based on distributed election.
- Fault tolerance in case of server failure.
- Messages cashing of offline clients in the cloud and their delivery once the client goes back online.
- Image encryption.
- UDP communication.
- Keeping track of online and offline clients.

2.2 Clients Communication

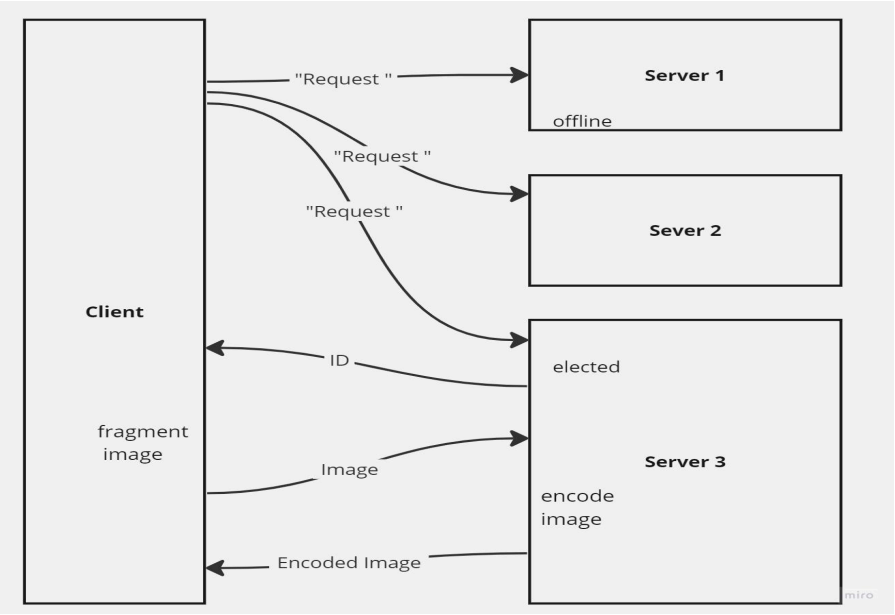
- Peer-to-Peer client communication.
- Sending images in low resolution in the initial client's communication.
- Decryption of images between clients.

- Changing the image access rights from the owner to the receiving clients.

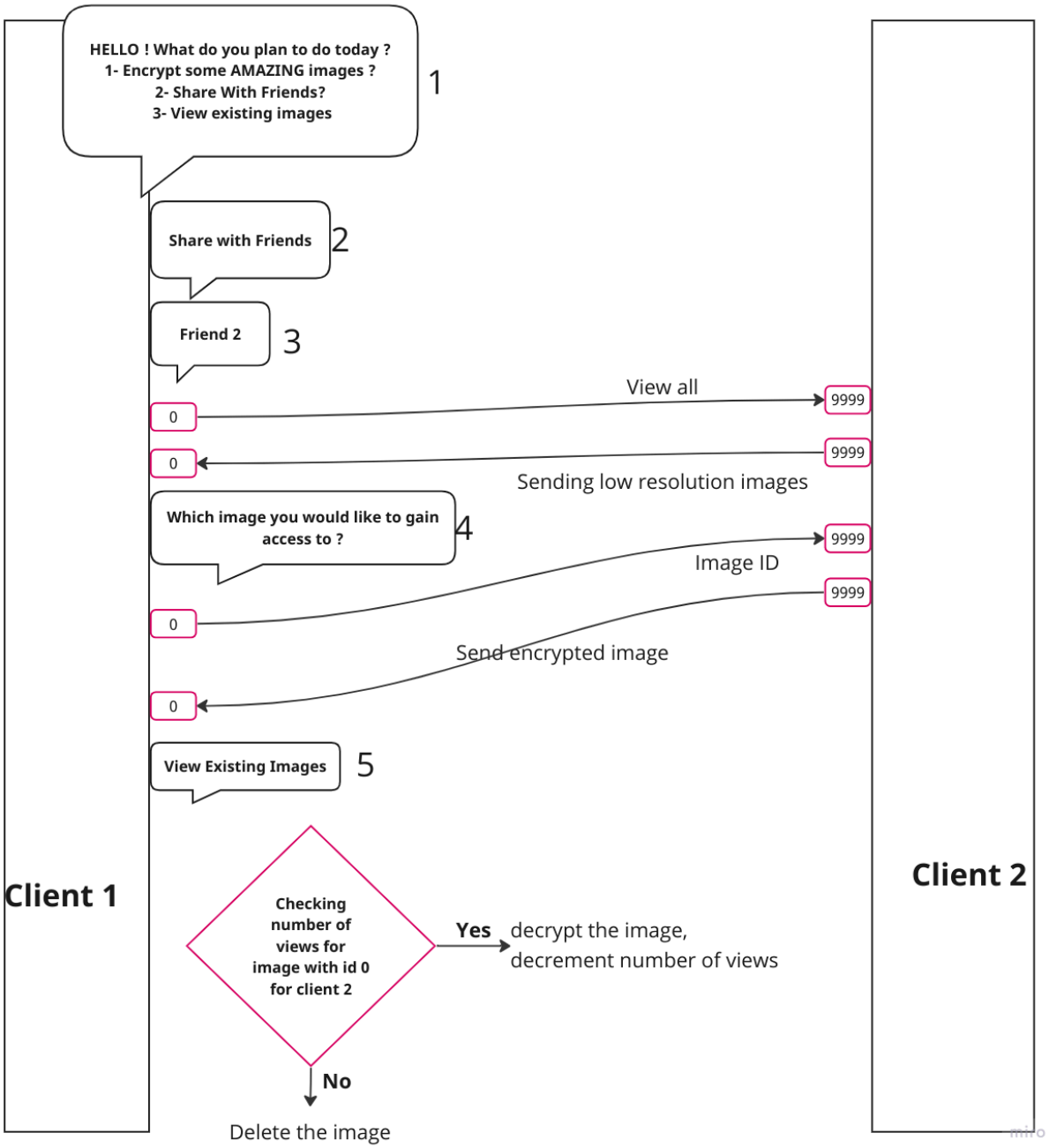
System Architecture

3.1 Image Encryption

When a client logs in and chooses to encrypt the images, it multicasts the message “Request” to all servers waiting for a replay with the ID of the elected server. Meanwhile, when servers receive this message they start electing the server that will handle this request. If the elected server fails to acknowledge handling the request, another server will be elected as part of the fault tolerance algorithm. The elected server will then reply to the client with its ID, which will be used to get the server’s IP address from the servers list in the client code. After this election is completed, the communication will only be between the client and the elected server. The client will start fragmenting the image. After sending each chunk, the client waits for an acknowledgment -to ensure reliability - before sending the next chunk. After sending the final chunk the client will send an “END OF TRANSMISSION ” message to the server to indicate that the server can start decoding. Once the server receives this message, it will start by reassembling these fragments and then uploading the cover image, and encoding them using steganography. Once the image is encoded, the server will send the encoded image back to the client using the same sending technique used by the client.



3.2 P2P Communication



Program Setup

4.1 File Structure

The project is of the common lib file which contains all the main functions and dependencies used by both client and server. Therefore it should be in the same folder as the server/client project folder. In addition, the program is made up of 3 files/cargo packets:

1. P2P: this file contains the client code, the images that will be shared, and the cover image used in encryption. It is the same code that works with all clients
2. Server: this file contains the server's code. It works on all servers except one.
3. Server with token: this code is identical to the server code, but the server contains the token once the program starts. Thus it runs only on 1 server.

4.2 Needed Libraries

All these libraries are added under dependencies in the cargo.toml found in the common lib file.

1. queues = "1.0.2"
2. steganography = "1.0.2"
3. image = "0.21.0"
4. photon-rs = "0.3.2"
5. tokio = { version = "1", features = ["full"] }
6. serde = { version = "1.0", features = ["derive"] }
7. serde_json = "1.0"

4.3 Compilation Procedures

Before compilation, the IPs in the clients and servers need to be updated. Also, we need to confirm that each server has a unique ID. Then we simply open the terminal, navigate to the src file of the code, and type “cargo run ”.

Engineering Design Decisions

5.1 UDP Vs TCP

We have implemented the sending and receiving using transmission control protocol (TCP) and user datagram protocol (UDP). There are advantages and disadvantages for each method. However, we decided to implement our algorithm using UDP because it is more efficient for our design requirements. Here are the reasons for choosing UDP over TCP :

- Low Latency: Compared to TCP, UDP has reduced overhead and is connectionless. As a result, there is less latency, which makes it appropriate for real-time communication where prompt responses are essential. Your app's users will value quicker interactions in features like texting and exchanging images.
- Real-time Features: UDP can be useful for real-time functionality like instant messaging. In comparison to TCP, it is more responsive for real-time communication since it enables the faster delivery of tiny packets.
- Loss Tolerance: Images can withstand loss better than other data kinds like text. UDP is a good choice when periodic packet loss is tolerated because little losses in picture data might not be as obvious or significant.
- Simplicity and Speed: Faster data transmission can be achieved through UDP's lack of connection setup and teardown procedures, which can enhance the efficiency of the image-sharing process.

However, there are also some disadvantages to using UDP:

- No guaranteed delivery: Packet loss can occur and there is no retransmission in UDP.
- Ordering Not Guaranteed: UDP does not ensure the order of packet delivery.

5.2 Election Algorithm

Queues:

During the literature survey research phase, we found out that there is a modified version of the bully algorithm, which is called the "Well-Organized Bully Algorithm", and since this algorithm had a best-case and worst-case complexity O(1) we decided to build our election design based on it. This algorithm is preference-based, in other words, the leader is not elected based on the highest or lowest ID number. The main structure that is used in this algorithm is the queue-linked list. This made selecting a leader and inserting a new node simple. At the initialization phase of the algorithm, all servers are given a unique ID number and added to an empty unsorted linked list. Whenever a new request is multicast by a client to all servers, the server with the ID on top of the list is directly elected to handle this request. The queues are updated in all servers after each request in an attempt to achieve synchronization among all servers. Also, for fault tolerance simulation, the server that is elected to crash was dropped out of the list and was enqueued at the bottom of the list when recovered.

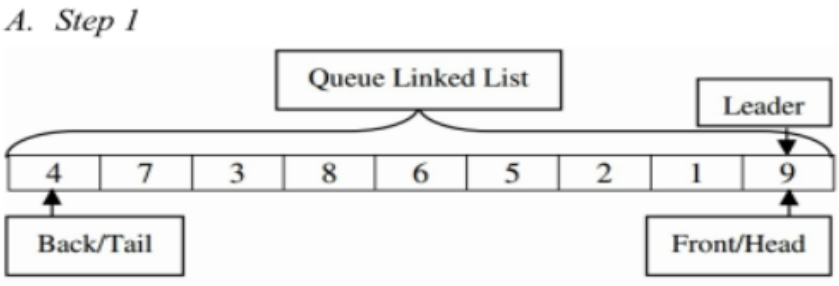


Fig. 8. Assign unique IDs and select a leader

B. Step 2

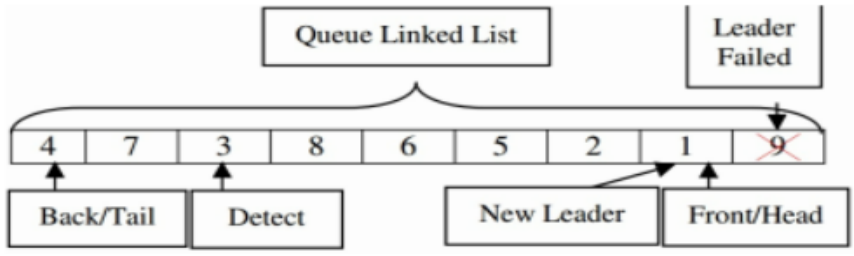


Fig. 9. Leader failed and select new leader

C. Step 3

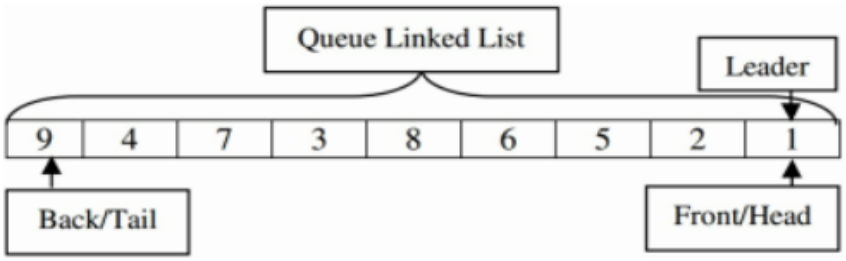


Fig. 10. Recover previous leader

Load balancing optimization of the queue:

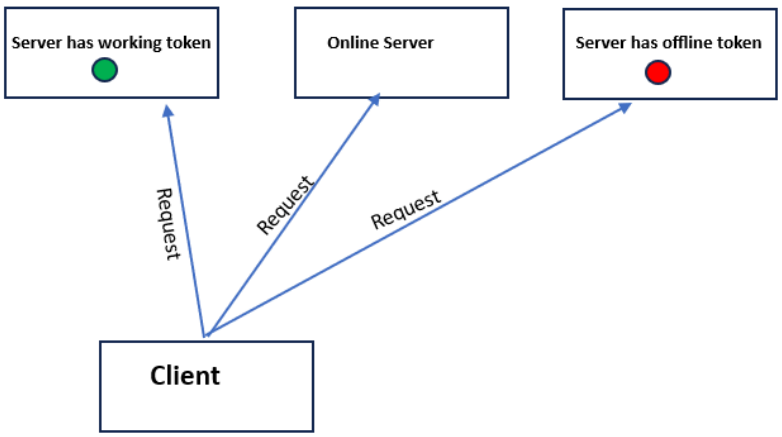
The above optimization conducted in the literature review that we adopted for our queue implementation included some synchronization issues as there is a repetitive queue on each machine. This approach caused inefficient request handling as there were some requests handled multiple times and some requests were dropped because of the unsynchronization of the top of the queue in each server. Hence, we optimized this implementation by converting it to a round robin approach where each server knows the next one to hand the token to and the one after in case if the next server was online, then the token is handed to the one after. This implementation

showed better performance in terms of packet loss as compared to the previous unsynchronized queues implementation.

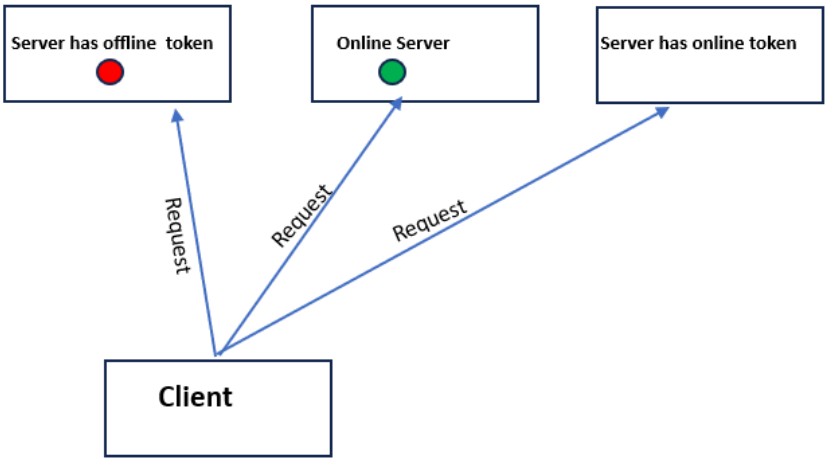
5.3 Fault Tolerance Algorithm

For implementing the fault tolerance algorithm, we had a working token. When the working token is with a certain server, it means that the server is handling the requests at the current time. After the time passes, the server passes that token to the next server in the queue. If a certain server is offline, it sends that it is offline and the next server in the queue handles the message. We have a token that is called an offline token to simulate when a server is offline. The server has the offline token, sleeps for some time then passes the token to the next server.

Below are simple diagrams to illustrate the algorithm:



In this diagram, There is an online server that has the working token (green one) and it is the first in the queue so it is going to handle the incoming request from the client.



In this diagram, the first server in the queue is offline. It sends a message to the other servers that it is offline and the request is handled by the coming server in the queue.

Failure transparency:

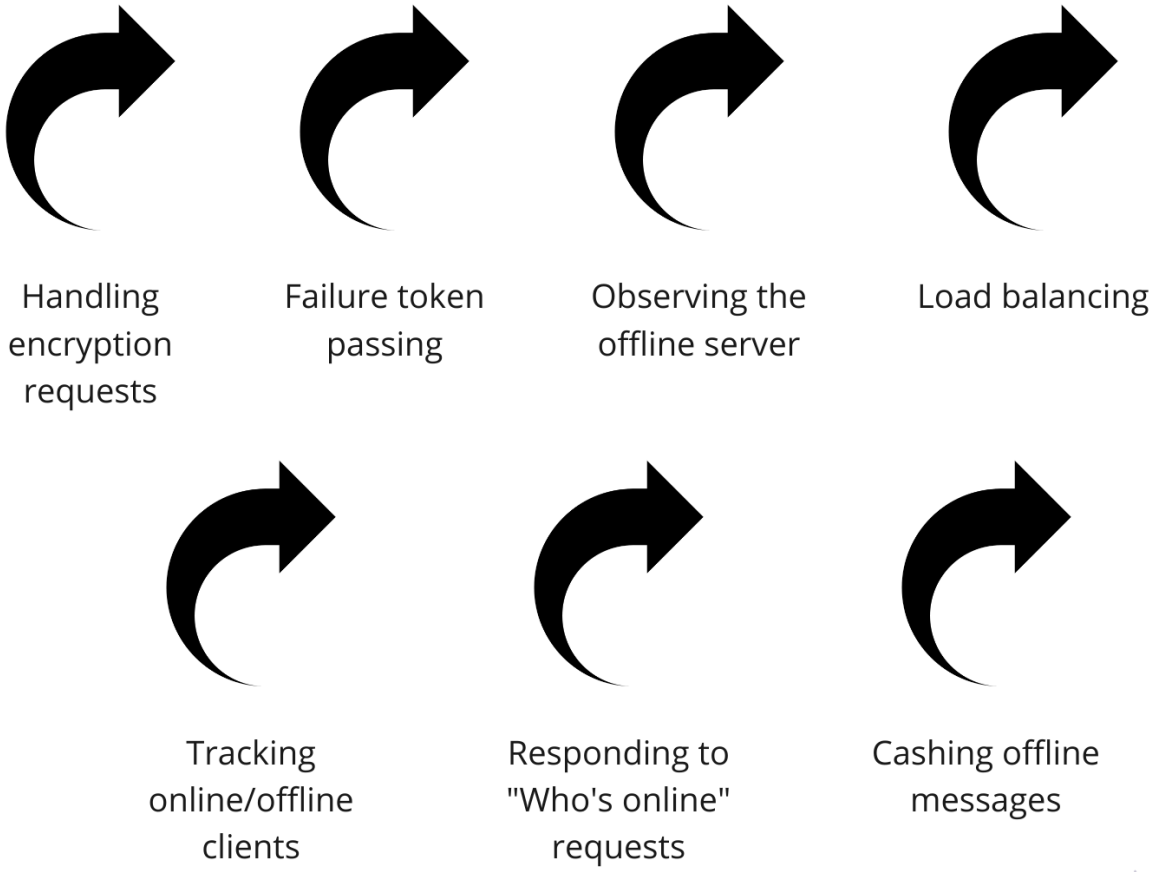
Implementing this mechanism ensures the cloud failure transparency to the clients as it is handled in the cloud itself shielding the details from the client. Therefore, In our system, the clients are not aware of servers failures because the system continues to function correctly despite them by passing the workload to the following server in the queue. The goal of failure transparency is to provide a seamless experience, where the cloud's operation appears uninterrupted and consistent, even when some of its components have failed.

5.4 Concurrency

In order to optimize the distributed system performance and achieve the required features, concurrency has to be embedded in our system in both the cloud and client architectures. This is because concurrency and multithreading play crucial roles in the design and operation of distributed systems. Concurrency helps in managing the complexity inherent in our distributed systems. It allows modularity in designing, where different components operate independently

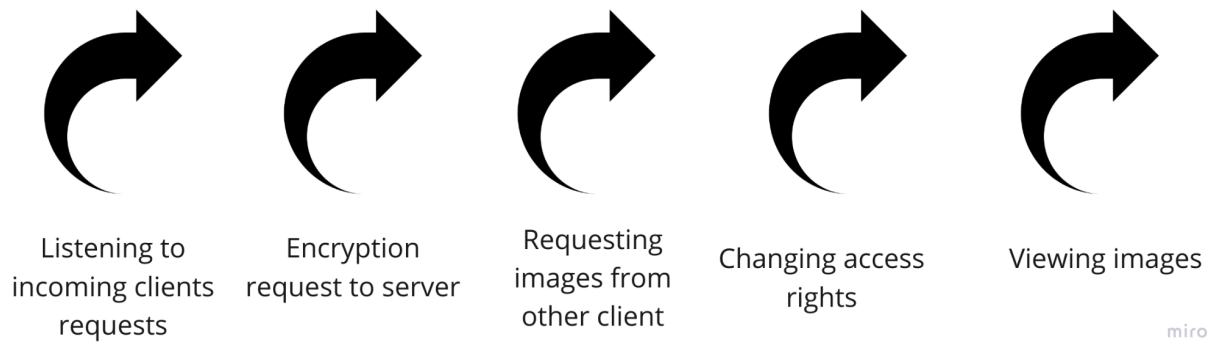
and concurrently. For example, on the cloud side, the following threads were used to handle different tasks done by the cloud.

Threads in the cloud:



The above threads combine the communication between the client and the cloud also the communication in the cloud itself.

Threads in the client:



The above threads illustrate the concurrent threads running in each client machine that communicate with each other or with the server.

5.5 Resources synchronization

In our multithreaded system, synchronizing resource sharing is of paramount importance to ensure system stability and efficiency. As illustrated there are multiple threads operate concurrently in both the clients side and cloud side, they often need to access shared resources such as data structures. Without proper synchronization, this can lead to conflicts and unpredictable behavior, commonly known as race conditions, where the outcome depends on the sequence or timing of the threads' execution. In our system we were keen on using the mutex (mutual exclusion) mechanism to tackle this critical problem. It works by allowing only one thread at a time to access a shared resource. When a thread needs to use a resource, it locks the mutex associated with that resource. If the mutex is already locked by another thread, the requesting thread is blocked until the mutex becomes available. This ensures that only one thread can modify the shared resource at any given time, preventing concurrent access issues. Mutexes are a fundamental tool in our parallel programed system, providing a straightforward yet powerful way to maintain data integrity and prevent the complexities that arise from uncontrolled resource access in multi-threaded environments.

The main shared resources that we had in our system involved:

In the peers side:

- Maps that keeps track of the image ID of each sender and its total access rights: This structure is accessed when a client wants to open a certain image, it has to check its access rights and the thread that updates the access rights when issued by the owner.

In the cloud side:

- Fault tolerance token: boolean flag to indicate if the server is down
- Load balancing token: boolean flag to indicate the server turn in the queue.
- Directory of service: map of online and offline clients.
- Message cashing: map to cache changing of access rights for the offline clients.

Sending Images

In our system, the transmission of images between clients and servers leverages the User Datagram Protocol (UDP). UDP affords us direct control over data transmission. This is particularly beneficial for our application's need for rapid and concurrent data exchanges.

6.1 Marshalling and Image Preparation

Initially, images are prepared — potentially involving encoding or resizing, akin to data preparation in RPC systems but with a focus on the data itself rather than on procedure calls. The prepared images are then marshaled, a process analogous to serialization in RPC. Here, marshaling involves converting the image data into a series of bytes (a byte stream) suitable for transmission over the network, a crucial step given the binary nature of images and network communication.

6.2 Image Chunking and Sequence Information

Due to the packet size limitations of UDP, large images are segmented into smaller data chunks. Each chunk is carefully prepared, including a header with sequence information. This process is

integral to marshaling in our system, ensuring that each segment of the image can be correctly reassembled at the destination. The inclusion of sequence numbers in each packet's header is particularly crucial, given that UDP, unlike TCP, does not guarantee the order of packet delivery.

6.3 Acknowledgment Mechanism

To address UDP's lack of delivery guarantees, which is typically managed within the RPC framework, we implement a custom acknowledgment mechanism. For each transmitted data chunk, the receiver sends back an acknowledgment signal. If the sender does not receive this acknowledgment within a predetermined timeout period, it indicates packet loss, prompting a retransmission. This system of acknowledgment is a key aspect where our approach diverges from RPC, which inherently manages reliable communication.

6.4 Handling Timeouts and Reliability

The acknowledgment process involves careful management of timeouts, a critical aspect of our system's reliability. This approach, while more manually intensive than RPC, allows us to ensure the integrity of the data being transmitted over an inherently unreliable network. Setting the timeout threshold is a delicate balance that affects the efficiency and reliability of the transmission process.

6.5 Reassembly and Decoding

After successfully receiving all the image chunks, the recipient undertakes the task of reassembling the image using the sequence numbers, akin to the deserialization process in RPC. If the image was encoded or altered before transmission, it undergoes a decoding process, much like how RPC systems handle responses and return values.

In conclusion, our UDP-based approach with explicit marshaling, chunking, acknowledgment, and timeout handling provides an efficient and reliable means of image transmission in our distributed system. This methodology effectively addresses our unique

challenges like network reliability and data integrity. It underscores the system's design considerations and adaptability to the specific requirements of our application, demonstrating a customized solution for efficient data transmission in a distributed environment.

6.6 Optimizing Image Sending Time

When sending a large image, we noticed that it took a long time to be fragmented, sent, reassembled, encoded, and saved, which could have been more efficient. Thus, to optimize the sending time, we decided to fragment the image based on the chunk size rather than the number of chunks. To do so, we started by finding the maximum packet size that we can send. Through trial and error, we concluded that the maximum packet size is 60,516 bytes, 8 bytes of which are reserved for the packet header, and the rest is the chunk size. This packet size is known to both the client and the server since it is used in fragmentation and reassembling. By doing this, the image sending now happens in the following order :

1- Check the image size

2- Fragment the image into x fragments, each with a size of 60,508 bytes

3- Send fragment, wait for acknowledgment, send next fragment after that

Interprocess Communication

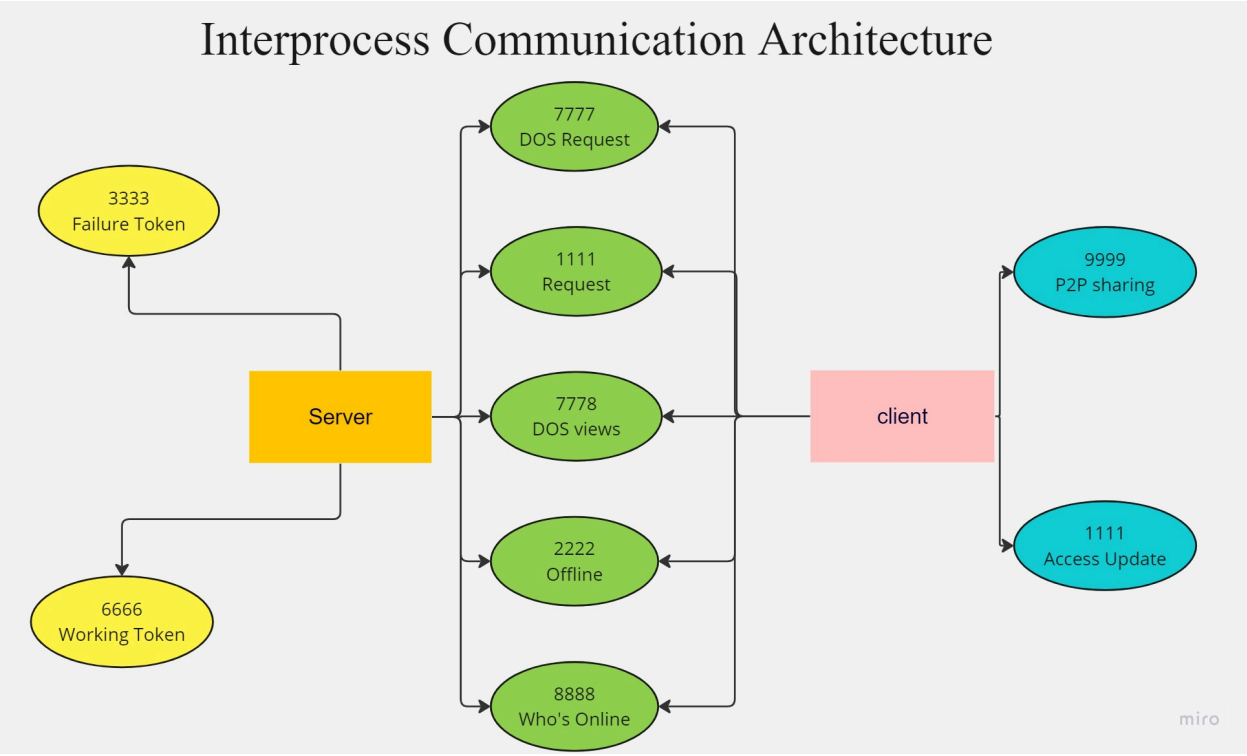
Managing Interprocess Communication (IPC) is a critical component of our architecture in our distributed system. To enhance efficiency, organization, and scalability, we have strategically utilized multiple sockets, each dedicated to handling specific types of communication. This approach offers numerous advantages, including ease of management, improved performance, and better scalability.

Our system employs different sockets for distinct communication tasks. For instance, we designate a specific port for sending images. This port handles the high data throughput required for image transmission, optimized for speed and efficiency. Another port handles client requests, including registering new clients or processing image-sharing requests. This separation ensures that the heavy traffic of image data does not interfere with the relatively lighter request-handling process. We also employ a separate port dedicated to handling failures, such as re-establishing connections or managing retries in case of dropped packets. This is crucial for maintaining system robustness and ensuring data integrity, especially in a UDP-based communication environment where packet delivery isn't guaranteed.

Additionally, in the context of our P2P architecture, different ports are used for peer discovery and direct peer-to-peer communication, allowing clients to discover each other and then communicate directly for image sharing. The use of distinct ports for these different aspects of communication not only aids in load balancing but also enhances the system's capability to handle multiple tasks concurrently without performance degradation.

A significant challenge we faced in our multi-socket environment was maintaining synchronization and preventing data races. To address this, we implemented mutexes (mutual exclusion locks) across shared resources. Mutexes ensure that only one thread accesses a critical section of code at a time, thereby preserving data consistency and preventing concurrency-related issues. This synchronization mechanism is particularly important in our system, where different threads may simultaneously read or write to shared data structures, such as socket buffers or data queues. For example, in our image transmission process, mutexes are

used to synchronize access to the shared buffer that stores image chunks waiting to be sent. This ensures that while one thread is adding data to the buffer, another thread can safely retrieve data for transmission without any overlap or data corruption.



Performance Analysis

In order to measure the performance of our system, we calculated the failure rate to be the percentage of unhandled requests from the total requests sent. We have implemented many threads that send many requests and the below analysis is based on sending text messages. We had around 200 threads each sending 5 requests and there was a delay of 100 milliseconds between each request.

7.1 Without load balancing

Initially, we tested our system without load balancing. In that case, our system had a higher failure rate due to the uneven distribution of requests across servers. This led to some servers being overwhelmed while others remained underutilized. Consequently, the system's overall ability to handle incoming requests efficiently was compromised, resulting in a noticeable increase in request-handling failures. In this case, the failure rate was 42% and the majority of packets were dropped.

7.2 With load balancing and Without Fault Tolerance

Implementing load balancing significantly improved our system's performance. By evenly distributing the requests among all available servers, we observed a substantial reduction in the failure rate. This approach ensured no single server became a bottleneck, leading to more efficient request processing and a more stable system overall. Therefore, the failure rate in this case decreased and became around 29.5%.

7.3 With Fault Tolerance

Integrating fault tolerance dramatically enhanced system resilience and reliability. The failure rate decreased as the system could now handle server outages effectively by rerouting requests to operational servers. This capability ensured continuous processing and handling of requests, even in the face of server failures. However, the processing time of handling the requests increased and the failure rate did not change dramatically as it was around 31%.

7.4 After Queue Optimization With Fault Tolerance

Post queue optimization, coupled with fault tolerance, the system's performance peaked. The optimized queuing system efficiently managed the influx of requests, reducing processing delays. Combined with fault tolerance, this resulted in a notably low failure rate, showcasing the system's enhanced capability to handle high loads and server failures seamlessly. With the

combination of all elements in the cloud (load balancing and fault tolerance), the failure rate was significantly decreased and became only 17%.