Cloud P2P Application for Image Sharing

Fall 2024

CSCE 4411 - Fund. of Distributed Systems

The American University in Cairo

Ahmed Amin	**900202813**
Amena Hassan	**900205225**
Mario Ghaly	**900202178**
Hamza Elgobba	**900202728**

Submitted to:

Dr. Amr El Kadi

1. Project Overview

1.1 Introduction

In today's interconnected world, distributed systems are essential for delivering reliable and efficient services. This document details the Spring 2024 Distributed Systems Course Project under the supervision of Dr. Amr El-Kadi, outlining the project requirements, design decisions, system performance, and user experience flow.

1.2 Project Description

This project aims to implement a cloud-based peer-to-peer(P2P) environment for image encryption and sharing, emphasizing transparency, load balancing, fault tolerance, and P2P communication. The cloud simply consists of 3 interconnected servers communicating with each other P2P to support leader election, load balancing, and fault tolerance in handling client requests. Moreover, the cloud is associated with a user-oriented discovery service to keep track of active users and the images they offer for sharing. On the other hand, the client side's high-level objective is simply to control image exchange with other clients through ownership and viewer rights. The following sections will extensively explain the requirements of each component in the project to provide a detailed understanding of the project at hand.

1.3 Project Requirements

For the sake of clarity, the environment is divided into distinct components, outlining the functionalities and objectives of each component at a time.

1. **Service Side**
    a. **Cloud of Servers**
        i. Three interconnected servers can talk to and listen from each other through P2P architecture
        ii. Communication between servers is via reliable User Datagram Protocol(UDP)

iii. The cloud uses an election algorithm to pick the server to handle a client request
iv. The election algorithm is used for load balancing as it should choose the server with the least workload
v. The cloud is failure tolerant; that is, if one or two servers fail or disconnect, the remaining servers can detect the failure and carry on their operations normally without affecting the user experience
vi. Each server can implement image encryption using Steganography
vii. Each server can receive an unencrypted image from a user and send the encrypted image back to the same user over UDP
viii. The three servers periodically simulate failure via an election algorithm

b. Directory of Service(DoS)

i. The DoS registers users and keeps track of each user's essential information, which includes their current status(online or offline), and how other clients can reach them(IP address & port number)
ii. The DoS returns information about the active peers of a user and how to reach them
iii. The DoS receives from owners access rights modifications for offline clients and enforces them whenever the clients become online again
iv. The DoS should be synchronized and consistent among all servers
v. Servers reviving after failure should be updated with the latest version of the DoS

2. Client Side

a. Client Application

i. Each client is expected to own image(s)
ii. The client application supports encrypting its images
iii. The client application supports requesting images from its peers with a requested number of views
iv. The client application can accept or reject a request to view its images

v. The client application supports sharing its images with other peers
vi. The client application can modify the access rights given to the images it has shared with other peers

b. Client Middleware

i. The client's middleware multicasts requests to the three cloud servers
ii. The client's middleware can encrypt access rights to a specific user
iii. The client's middleware can send the two-level encrypted image to the requester via UDP and can receive one
iv. The client's middleware allows the user to view only their images and images they have access to.
v. The client's middleware decrements the number of permitted views within the image whenever the user views the image
vi. The client's middleware denies access to an image when the number of permitted views is consumed, and it shows a default image instead
vii. The client middleware implements a method for enforcing access modification when the target user is online or offline

2. Design Choices

2.1 Leader Election

2.1.1 Election Algorithm Choice

A leader election survey was conducted to carefully choose the most suitable algorithm for this project [Distributed Election Survey - Group 4]. The final decision was to use Gholipour's improved Bully algorithm [1]. In this modification, the leader is aware of the highest priorities and saves them in an ordered coordination group. This group shall be passed to all other nodes and periodically updated via heartbeats. This modification gives advantage of reducing message communication from $O(n^2)$ to $O(n)$. This is because the modification eliminates the need for a global election for each client request since every server has the coordination table and knows which server will handle it without need for extra communication.

2.1.2 System Load Balancing

System load balancing may involve several parameters such as the current server load or communication delay. In the context of our project, we have 3 identical servers with the same CPU power at the same place, so it is not logical to assign static IDs or priorities to them, but instead priorities should be assigned real-time. Our project heavily depends on the threading capabilities of the server. Thus, we chose the main parameter to be the current server load, which in implementation is the CPU load average metric over the past minute. CPU load is defined as the number of processes using or waiting to use one core at a single point in time. The CPU load is fetched by querying the OS structure to get the average load. The server with the lowest load should be assigned the highest priority via the following formula $Priority = \frac{1}{Average\ CPU\ Load}$. In case of equal priorities, the IP address, and the port number used for communication are the tiebreakers. The servers periodically exchange their updated priority to update the coordination table order. This mechanism ensures that the load is fairly distributed among the 3 servers. An illustration image for the priorities exchange can be shown in the Use Cases section [3.1].

Surely, future modifications could include the network I/O bandwidth as this project heavily relies on network communication between servers and clients. However, the CPU load was quite sufficient for our use cases and experiments.

2.1.3 System Fault Tolerance

The final crucial step in leader election algorithms is failure tolerance without affecting the user experience as much as possible. Our implementation supports reliability since servers can detect the failure of its peer servers. This feature works by recording the timestamp when a server receives a heartbeat message from its peer. This timestamp is the "*last alive time*" to track when the peer was last active or reachable. When the server gets its turn to send a heartbeat, it will check the difference between current time and each of its peers' "*last alive time*". If this difference exceeds a specified threshold, the server detects the failure of its peer, and its local coordination table is updated accordingly. After detecting a server failure, the next step is to manage any client requests that were mistakenly assigned to the failed server. This process works as follows:

1. **Leader Notification:**
   The leader server sends a notification message to all its peer servers, declaring its responsibility for specific client requests.
2. **Storing Request IDs:**
   Each server temporarily stores the IDs of all multicast client requests it receives. This ensures that every request is tracked.
3. **Timeout and Reassignment:**
   If a server does not receive a notification from the leader within a predefined timeout period, it assumes that the leader has failed. In this case, the new leader takes over and handles the unprocessed client requests with all the normal steps.

2.2 Failure Simulation

Decision choice of simulating servers failures is a token passed between them in a round robin fashion. Once the token is received, the server goes down for 20 seconds. This selected server then ignores communication with other servers or clients during this period.

2.3 Directory of Service (DoS)

The directory of service -- also called the discovery service -- plays a crucial role in this project by identifying and tracking active peers. The implementation choices for the DoS were either to be decentralized locally on the servers via a replicated data store or to be in a centralized manner. The problems with the shared memory are mainly due to two main challenges:

1. **Synchronization & Ensuring Consistency:**
   Coulouris et al. states the following, "synchronization is via normal constructs for shared-memory programming such as locks and semaphores (although these require different implementations in the distributed memory environment)" [2]. Maintaining consistency across replicated data stores introduces complexity, as preventing race conditions across multiple servers requires robust synchronization mechanisms tailored to ensure replica coherence.
2. **Heavy Message Communication:**
   Coulouris et al. discuss how message passing is inevitable in distributed environments to ensure that each server updates its local replica of recently modified items [2]. This results in significant communication overhead, which can impact system performance and scalability. While replication enhances fault tolerance and reliability by maintaining multiple copies of the directory service across servers, it introduces challenges such as increased communication overhead and data redundancy.

In summary, implementing synchronization primitives in a distributed setting can be error-prone and resource-intensive. On the other hand, having a centralized DoS would be efficient because there are only 3 servers at hand. This means that at the worst case scenario, there would be 3 simultaneous accesses to the directory of service, which would not be an issue for centrality. Thus, our decision was to use a centralized MySQL database hosted on Aiven. This decision choice eliminates all synchronization issues and limitations and simplifies the DoS implementation. This is also because MySQL database supports concurrency control.

Regarding implementation details of the database, we created 3 tables. The first one is "*clients*" to store client UUID(the primary key), its status: online or offline, and its IP address concatenated with the port number(socket address) it would listen on for P2P image requests.

The client UUID is generated by the database by the auto increment feature to ensure uniqueness among clients. Secondly, we decided to store the encrypted images IDs of each user as a business model to advertise themselves. Each image ID is uniquely identified through concatenating the client UUID with the image name stored on their application side. This way we achieved unique IDs for all images and at the same time made the image ID representative as an advertisement so that other peers would want to view it. Thus, the second table is "*resources*", storing the client ID and the image ID both as a composite primary key. Finally, the DoS is involved in best-effort policy in modifying the access rights of an offline user. Hence, we needed to create a third table named "*access_rights*" that has viewer ID, the image ID, and the relative change in the number of views. The primary key of the third table is a composition of the viewer and the image IDs. To make things simpler, the following table would show all the DoS use cases and their equivalent database queries:

Action	**Query**
Register	Insert a row in "*clients*" table and returns the generated client ID
Sign-in	● If the client ID exists, make its status online and change its IP address and port number ● The DoS checks in "*access_rights*" table if this client ID has any pending access modifications. If so, return them and remove the row(s)
Encrypt Image	Insert a row with the image ID and its client ID in "*resources*" table
Who is up?	Returns a list with the online peers' socket addresses and their image IDs
Shutdown	Change client status to be offline in "*clients*" table

Table 1 : DoS Operations

2.4 Defining Client Requests Types

Our cloud P2P environment system enables two primary types of communication: client-server communication and peer-to-peer (P2P) communication. Each mode of communication supports distinct request types tailored to specific functionalities within the system.

The client-server communication forms the backbone of user management and cloud-based interactions. It facilitates essential tasks such as user registration, authentication, and querying cloud resources. Initially, a user interacts with the cloud to register or sign in. Once authenticated, the user gains access to a variety of features, including:

- Requesting image encryption through the cloud.
- Inquiring about active users and their available images.
- Retrieving the IP and port information for communication with active peers.

Complementing this, P2P communication operates between authenticated users. This mode primarily supports requests for accessing images stored on peers and modifying access rights, such as adjusting the number of allowed views or granting specific permissions.

2.4.1 Request Formats and Serialization Mechanisms:

A. JSON-based Requests

Most of the requests in our system, except for image transfers, are JSON-based. These requests undergo a process of serialization before transmission and deserialization upon receipt, ensuring efficient and structured data exchange. Key examples include:

1. Client Registration: Requests include user details to register on the cloud, with responses indicating success or failure.
2. User Authentication: Involves user credentials and associated metadata to validate access.
3. Active User Inquiries: Queries to retrieve a list of online users, their available images, and the corresponding IP and port information.

Each JSON request contains a "type" field indicating the nature of the request, and the responses include an acknowledgment (ack) with the request's status and any additional relevant data.

B. Image Transfer Request

Image transfer between clients and the cloud or peers involves dividing images into data chunks. Each chunk is enriched with metadata to ensure reliable and ordered transmission:

- Chunk Metadata:
  - image_id: Identifies the image associated with the chunk.
  - sequence_number: Specifies the order of the chunk within the image.
  - ack_number: Confirms receipt of chunks from the receiver.
  - termination_signal: Indicates the end of the transfer.
- Reliability Mechanisms:
  - Reordering chunks based on sequence_number to reconstruct the image correctly.
  - Ensuring acknowledgment (ack) for each chunk to confirm successful receipt.
  - Utilizing a termination signal to indicate the completion of the image transfer.

2.5 Data Marshalling and Serialization

In distributed systems, data marshaling and serialization are critical for enabling communication between processes running on different machines. These techniques ensure that structured data, such as JSON objects or image chunks, is converted into a transmittable format and reconstructed accurately at the receiving end.

Serialization is essential for interoperability between heterogeneous systems and plays a pivotal role in achieving reliable and efficient communication in distributed environments.

2.5.1 Serialization and Communication in the Cloud P2P Environment

In our cloud P2P environment, serialization occurs at multiple stages of communication to handle various types of requests and data payloads.

1. **JSON-Based Requests**:
    - JSON is used to structure requests and responses for tasks such as client registration, authentication, and querying active users or images. These JSON payloads are relatively compact, typically containing key-value pairs to convey essential information.
    - **Packet Size**: JSON-based requests are small in size, typically ranging from a few bytes to a few hundred bytes, depending on the request type. This efficiency makes them ideal for metadata exchange.
2. **Image Transfer**:
    - Images are divided into fixed-size **chunks** of 1024 bytes, which represent the maximum packet size in the system. Each chunk includes additional metadata:
        - image_id to associate the chunk with a specific image.
        - sequence_number for ordering.
        - ack_number for reliability.
        - termination_signal to denote the end of the transfer.
    - The metadata ensures that even large images can be transmitted reliably across peers in a distributed network.
    - **Packet Size**: While each chunk is 1024 bytes, including metadata, the payload size is considerably larger due to the image content. This results in a higher communication overhead compared to JSON-based requests.

2.5.2 Comparison of Packet Sizes and Efficiency

Request Type	Typical Packet Size	Content
JSON-Based Request	Few bytes to 1 KB	Compact key-value pairs for metadata and request/response data.
Image Chunk (P2P)	1024 bytes (max)	Includes image data, chunk-specific metadata, and reliability markers (e.g., sequence, ack).

JSON Efficiency: JSON requests are lightweight and efficient for operations that require minimal data, such as user authentication or metadata inquiries.

Image Chunk Overhead: Image transfer introduces significant overhead due to its size and the need for reliability mechanisms. However, this trade-off is necessary to ensure complete and accurate image delivery in the P2P network.

The marshaling and serialization processes occur at critical points in the communication flow:

- **Client-Server Communication**:
    - Registration, authentication, and metadata queries utilize JSON serialization for compact and efficient communication.
- **Peer-to-Peer Communication**:
    - Image requests and access right changes rely on image chunking and metadata serialization to enable reliable and ordered delivery between peers.

2.6 Coding Framework : Tokio Crate

The User Datagram Protocol (UDP) is a lightweight and connectionless protocol widely used in distributed systems where low-latency and high-throughput communication is essential. Unlike TCP, UDP does not guarantee reliable delivery, order, or error correction, making it faster and more suitable for applications that can handle these responsibilities at the application level.

In our cloud P2P environment, UDP serves as the foundation for communication between clients, the server, and peers. By utilizing the Tokio crate in Rust, the system takes full advantage of non-blocking I/O operations, asynchronous communication, and low overhead, all while maintaining the flexibility to implement custom reliability mechanisms. This approach ensures efficient and reliable interactions within the system.

Benefits of Using Tokio with UDP in Rust

1. **Asynchronous Communication**:
    - Tokio provides powerful abstractions for asynchronous programming, enabling the system to handle multiple client requests and peer communications

concurrently. This is particularly beneficial in the P2P network, where several peers may simultaneously interact.

2. **Non-blocking I/O**:
    - Tokio's event-driven architecture ensures that the system remains highly responsive, even under heavy workloads. Non-blocking UDP sockets allow the system to send and receive packets without stalling for other tasks.
3. **Low Latency and Overhead**:
    - UDP inherently avoids the handshake, congestion control, and session management mechanisms of TCP, resulting in reduced latency and lower overhead. The Tokio crate amplifies this advantage by providing efficient task scheduling and resource management.
4. **Custom Reliability Mechanisms**:
    - While UDP does not provide built-in reliability, this lack of overhead allows the system to build its own reliability features tailored to its needs. Using sequence numbers, acknowledgments, and termination signals, we ensure ordered and reliable data transfers, especially for image chunks.
5. **Scalability**:
    - The combination of UDP's lightweight nature and Tokio's asynchronous execution allows the system to scale effectively, handling a growing number of peers and requests without significant performance degradation.
6. **Seamless Rust Integration**:
    - By leveraging Rust's ownership and type safety features, the Tokio crate minimizes the risk of common programming errors, such as data races or invalid memory access, ensuring robust communication.
7. **System-Specific Optimization**:
    - Using Tokio enabled the system to achieve low-latency communication while retaining control over custom reliability mechanisms. This flexibility allows for efficient data transfer in scenarios that demand reliability, such as image chunking, without compromising on performance.

By using UDP with Tokio, our system achieves a fine balance between speed, efficiency, and reliability. This approach ensures that communication within the distributed environment meets

the stringent demands of low latency and scalability while providing the flexibility to tailor reliability mechanisms to specific system requirements.

2.7 Interprocess Communication

Interprocess communication (IPC) within the system is a cornerstone for enabling collaboration across its distributed components. In our system, IPC occurs through UDP and the Tokio crate, which facilitates non-blocking, asynchronous communication. As discussed in Section 2.5, this approach ensures high efficiency, scalability, and low latency. The communication architecture in the system can be classified into three types:

1. **Server-to-Server Communication**
2. **Client-to-Server Communication**
3. **Client-to-Client Communication**

2.7.1 Server-to-Server Communication

Server-to-server communication primarily occurs over port 8085 and is responsible for maintaining synchronization among servers. This communication channel handles the heartbeat messages essential for the election algorithm. Heartbeat messages allow servers to stay updated on the status of peers and determine which server has the highest priority for handling incoming requests. The selected server takes on the role of the primary handler.

This server-to-server communication is asynchronous and operates in a non-blocking manner, ensuring minimal interference with other system processes. It uses a single port (8085) to exchange messages efficiently, reducing resource usage while maintaining consistency and availability across the servers.

2.7.2 Client-to-Server Communication

Client-to-server communication is more complex due to the high load and multi-directional nature of the interactions. This type of communication follows a one-to-many-to-one model,

where the initial request is broadcasted to all servers, and a single server is eventually selected to handle the client's request. Here's a breakdown of the process:

1. **Multicast Request**:
    - When a client initiates a request, it broadcasts a **multicast message** to all servers on a predetermined parent port (8081).
    - The servers receiving this multicast request execute the election algorithm to determine which server will handle the client's request. The election considers factors such as server availability, load, and priority.
2. **Port Allocation**:
    - Once a server is selected, it allocates a **worker port** through the parent port (8081) and sends this port number back to the client.
    - The server starts listening on this allocated worker port asynchronously.
3. **Client-Server Communication**:
    - After receiving the worker port number, the client initiates communication by sending the required data, such as image chunks or specific requests, to the server.
    - The server processes the request on the worker port.
4. **Termination Signal**:
    - When the client has finished communication, it sends a **termination signal** to the server on the worker port. This allows the server to unbind and free the port, making it available for future use.
    - If the server does not receive a termination signal or any communication on the worker port within a predefined timeout period, the port is automatically unbound and freed.

This dynamic port allocation ensures efficient resource management, as worker ports are only engaged for the duration of active communication. The parent port (8081) manages the allocation and reuse of worker ports, as depicted in the figure (1).

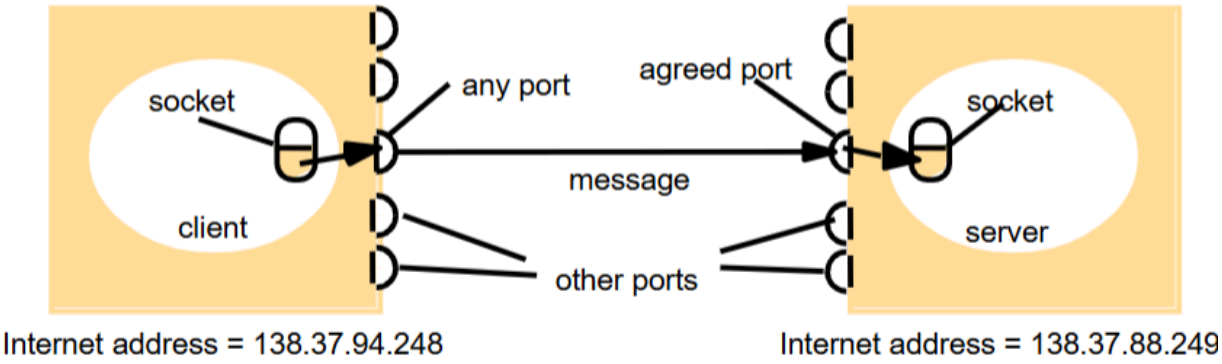

Figure 1: Dynamic Port Allocation within the Server and the Client [2]

For the system to be scalable, we set a maximum number of threads that are dynamically allocated. Upon reaching this number, we shift to a threading architecture shown in the figure (2) where all the requests are queued. Once a thread completes its task, it picks up the next request from the queue. If no request is found in the queue, the thread waits for a certain period before terminating. This approach ensures efficient resource utilization and avoids excessive thread creation while maintaining scalability.

In this scenario the pool of worker threads created will be dealing with the incoming tasks (e.g., I/O operations or processing). Tasks are queued, and workers pick them up as they become available.

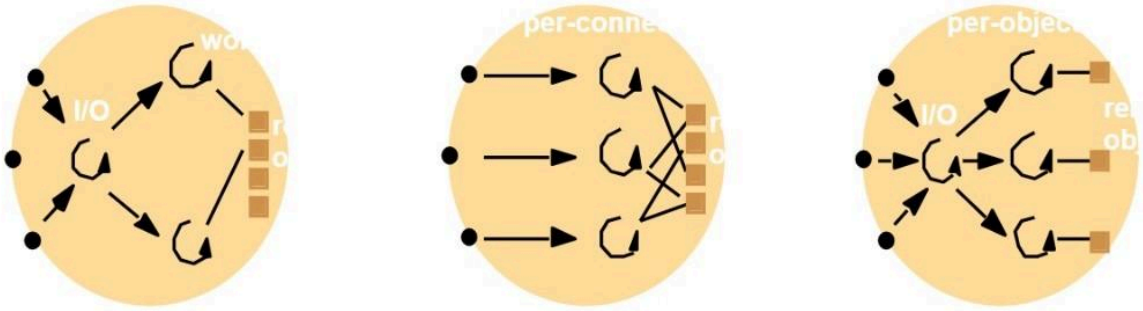

Figure 2: Different threading and concurrency models [2]

2.7.3 Client-to-Client Communication

Client-to-client communication leverages a decentralized approach, where each client allocates a port specifically for peer-to-peer (P2P) communication. The process is as follows:

1. **Port Allocation and Directory Registration**:
    - When a client starts up, it allocates a parent port for P2P communication and registers its information (e.g., IP address and port) with the Directory of Services (DoS).
    - The DoS marks the client as online and updates its entry, making the client available for interactions with other peers.
2. **Listening and Request Handling**:
    - The allocated parent port listens asynchronously for incoming requests from other clients.
    - Upon receiving a communication request, the parent port dynamically allocates a **worker port** to handle the specific interaction.
3. **Communication and Termination**:
    - The worker port manages the communication session between the clients, such as transferring image data or updating access rights.
    - Once the communication is complete, a termination signal is sent, after which the worker port is freed and unbound.

This architecture ensures that clients are available for P2P interactions while managing resources efficiently. The system's use of asynchronous communication through UDP allows it to handle multiple concurrent requests without blocking.It balances scalability, resource management, and responsiveness, reliability, enabling seamless interactions between servers and clients in real time.

2.8 Encryption

Steganography is the practice of hiding information within another medium, such as embedding hidden images or data within cover images. Unlike cryptography, which focuses on encrypting

data to prevent unauthorized access, steganography conceals the existence of the data, ensuring it remains unnoticed. This technique provides an additional layer of security and is widely used for secure communication in distributed systems.

In our cloud P2P environment, steganography is employed at two critical points:

1. Encoding a hidden image within a cover image during communication between the client and the server.
2. Encoding access rights (such as the number of views remaining) into the last row of an image on the client side.

2.8.1 Image Encoding (Client-Server)

The first application of steganography in our system occurs during communication between the client and the server, where a hidden image is embedded within a cover image. This process ensures the secure transmission of sensitive data, such as images, without drawing attention. The process begins by loading the cover and hidden images, both of which are converted to RGBA format for processing. To ensure compatibility, the dimensions of the hidden image are validated to confirm that it fits within the cover image. Once validated, the hidden image is serialized into a byte array to prepare it for embedding.

The embedding process leverages the Encoder from the steganography crate. The serialized hidden image data is securely embedded into the alpha channel of the cover image using the encode_alpha method. This technique seamlessly integrates the hidden data into the cover image without affecting its visible content. The resulting image, now containing the hidden data, is then saved to the specified output path. This method enables the client and server to exchange encrypted images effectively while maintaining the integrity and confidentiality of the data.

2.8.2 Access Rights Encoding (Client-Client)

The second application of steganography is on the client side, where it is used to encode access rights, such as the number of remaining views, into the last row of an image. This dynamic encoding integrates access control metadata directly into the image, ensuring that the image itself

remains self-contained and tamper-evident. To encode the access rights, the number of remaining views (num_views) is converted into a byte array. A new row is then appended to the image, with the bytes of num_views embedded into the alpha channel of the pixels in the added row. The modified image, now containing the encoded access rights, is saved for further use.

Decoding the access rights involves reading the last row of the image to retrieve the embedded bytes representing num_views. These bytes are then converted back into a u16 integer to restore the access rights metadata. To reconstruct the original image, the last row containing the access rights is removed, ensuring the image remains visually unaltered. If the access rights need to be updated, such as decrementing the number of views, the encoding process is repeated to embed the revised rights.

3. Use Cases

3.1 Exchanging Heartbeat Messages

Servers exchange heartbeat messages with their peers to update priorities, which are calculated based on CPU utilization over the past minute. The priority is determined using the formula: $Priority = \frac{1}{Average\ CPU\ Load}$

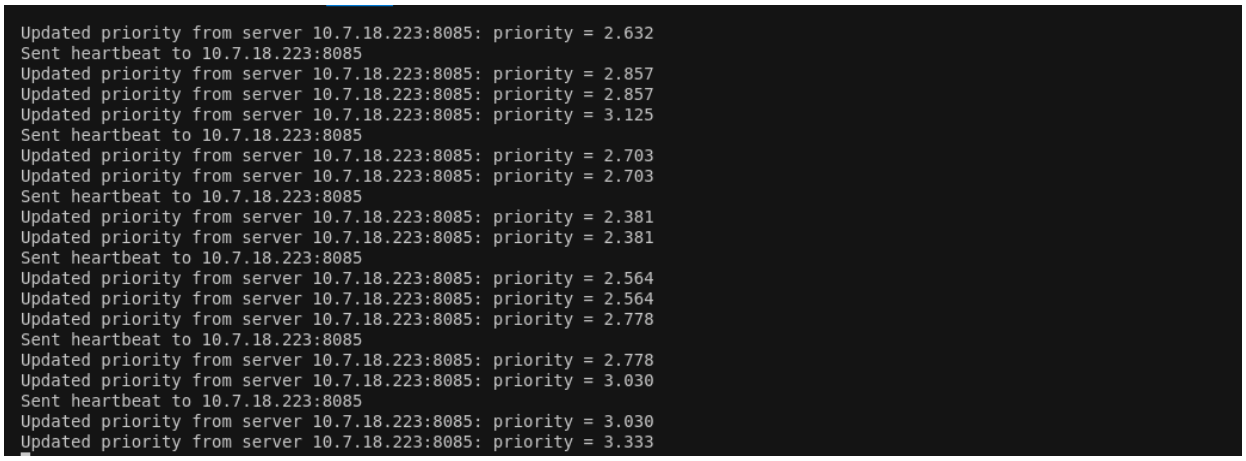

Figure 3: Servers Exchanging their Priorities

Priorities are used in elections to select the server responsible for handling client requests. For example, a server may be elected as the coordinator for processing a client registration request. This election process applies to all types of client requests.

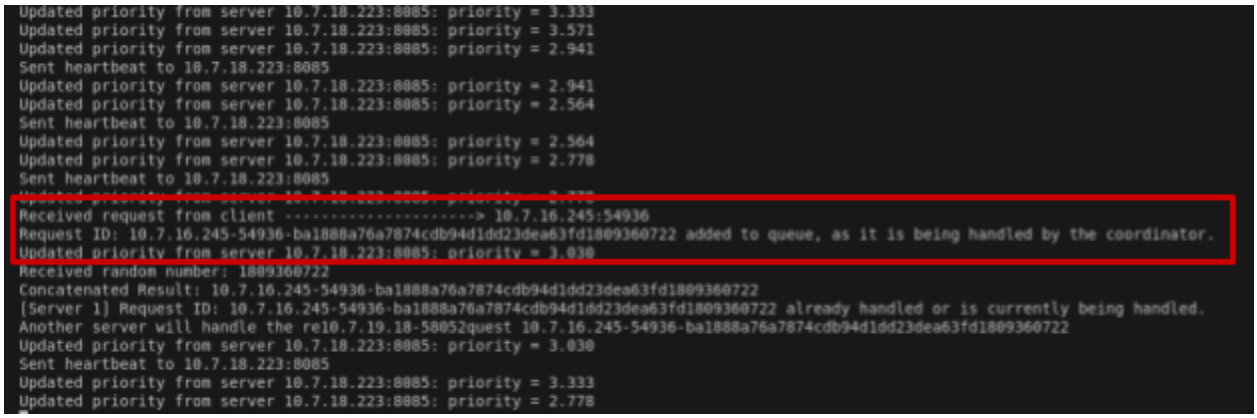

Figure 4: Election Algorithm Initiation Upon Receiving Request From Clients

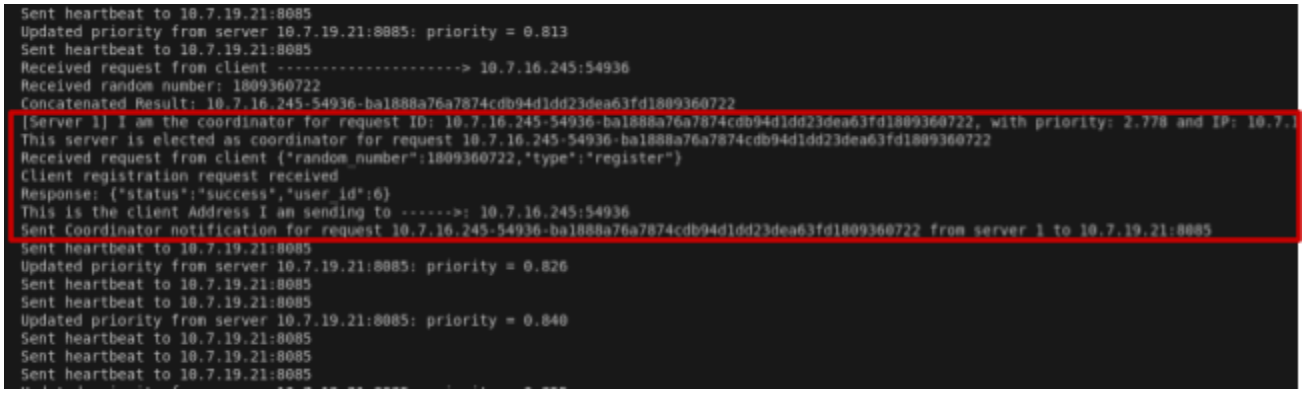

Figure 5: Coordinator Selection Based on the Priority

3.2 Registration and Sign-In

First, the client selects the option to register and receives a globally unique client ID.

Figure 6 : User Registration Process

The client can then sign in to access the main menu. After signing in, The message "No resources available" upon sign-in indicates that there were no messages sent to the client during their period of inactivity.

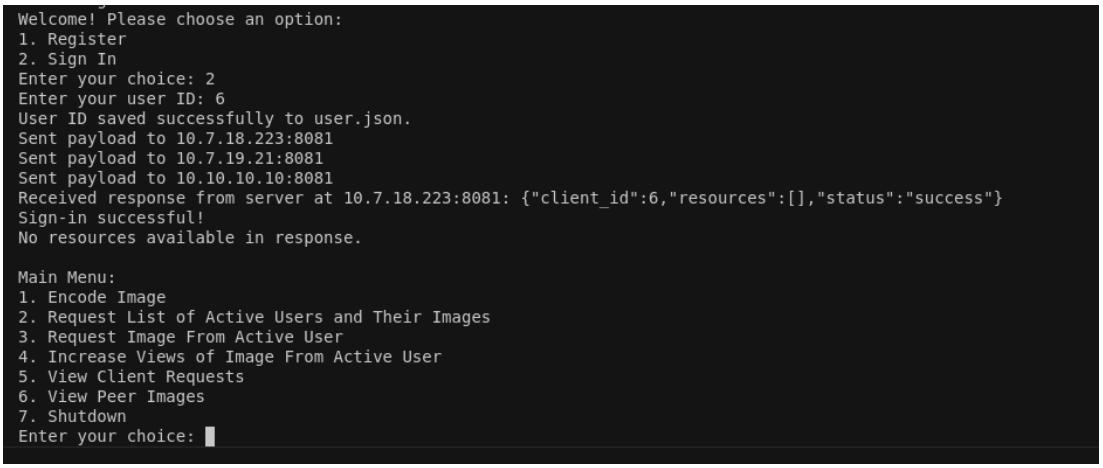

Figure 7 : User Sign in Process

3.3 Encrypting an Image

The user selects option 1, **Encode Image**, and is prompted to enter the path of the image to be encrypted.

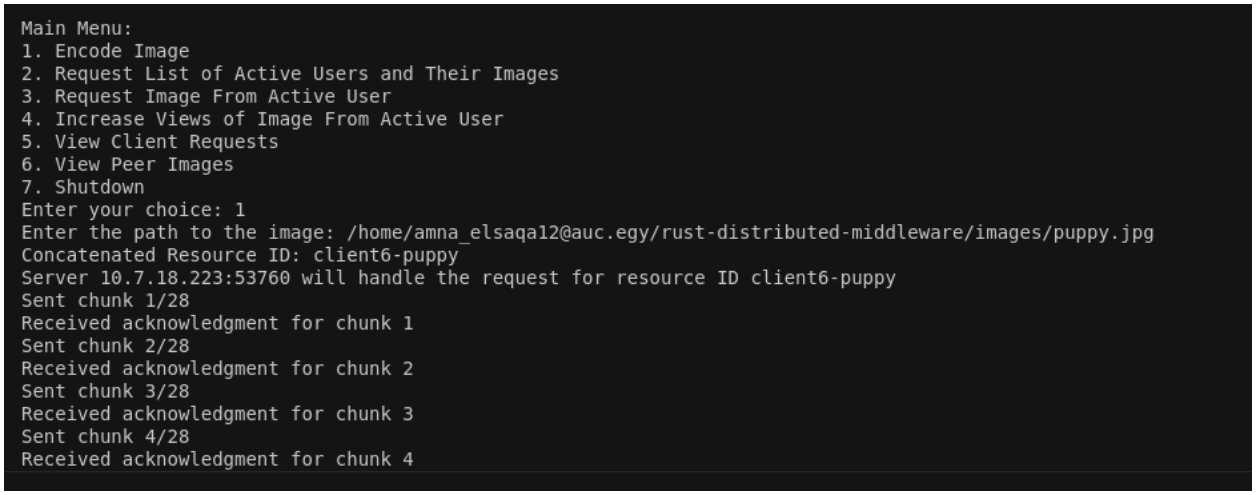

Figure 8 : User Initiate Image Encoding Request

The following images are the original image being sent by the user in this example and the encrypted image received to the user.

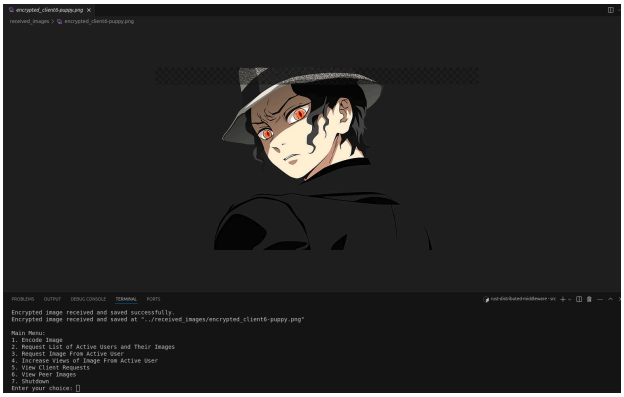

Figure 9 : Encrypted Image

Figure 10 : Original Image

3.3 Requesting Client Image

The user selects option 3, **Request Image From Active User**, and is prompted to choose the user to request from and specify the requested image ID. The requesting client then enters the number of desired views.

Figure 11 : User Requests Active Users from the Cloud

The client receives the request, which can be accessed through option 5, **View Client Requests**. The client can then choose to approve or reject the request. If approved, the image is sent.

```
3
Request Queue:
1. [10.7.17.127:41698] {"image_id":"client6-puppy","type":"image_request","views":5}
Enter the number of the request to handle, or '0' to go back:
1
Selected request: [10.7.17.127:41698] {"image_id":"client6-puppy","type":"image_request","views":5}
Options:
1. Approve
2. Reject
3. Go back
Enter your choice: 1
Approved request from 10.7.17.127:41698: {"image_id":"client6-puppy","type":"image_request","views":5}
Processing 'image_request' for image_id: client6-puppy, views: 5
Responder bound to new socket at 0.0.0.0:54892
IN ENCODE: num_views = 5
In respond_to_request: 10.7.17.127:41698
Sent chunk 1/279
Received acknowledgment for chunk 1
Sent chunk 2/279
Received acknowledgment for chunk 2
Sent chunk 3/279
Received acknowledgment for chunk 3
```

Figure 12 : User Action Upon Receiving Image Request From Active User

The requesting client receives the image encrypted with the access rights encoded within it.

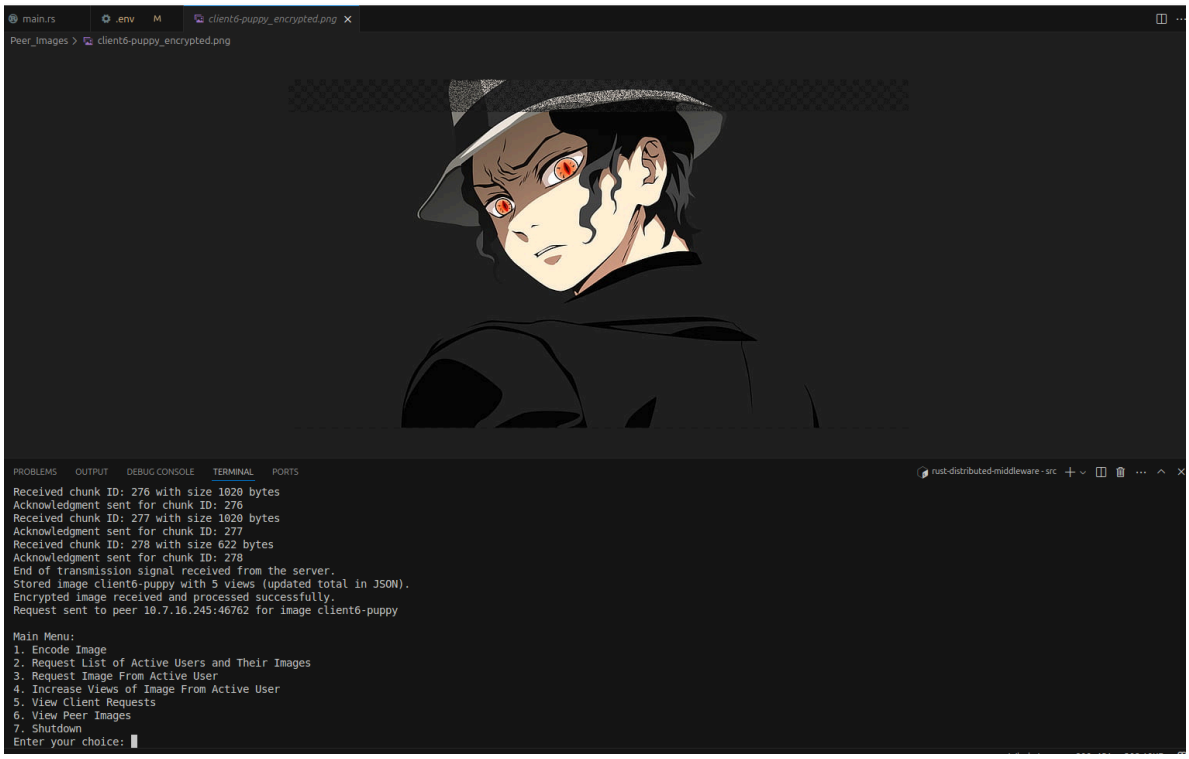

Figure 13 : Encoded Image Received By the Requester With the Access Rights embedded

3.4 Viewing the Image

The user selects option 6, **View Peer Images**, and is presented with the available peer images along with the number of views remaining for each. After entering the ID of the image to view, it is displayed. Once the Image Viewer is closed, the number of views for that image is reduced by one.

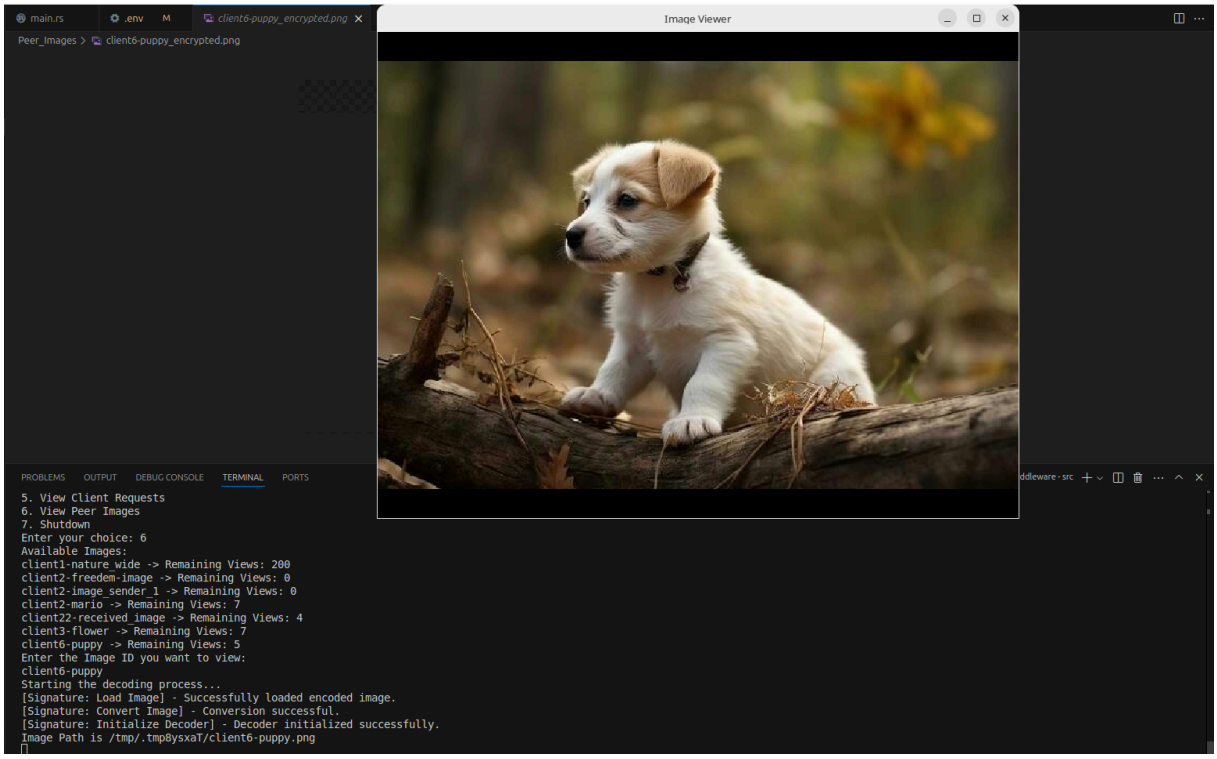

Figure 14 : User Viewing the Image Requested from Other Another User

Upon viewing the image, The number of views changes from 5 to 4 as shown in the below figure:

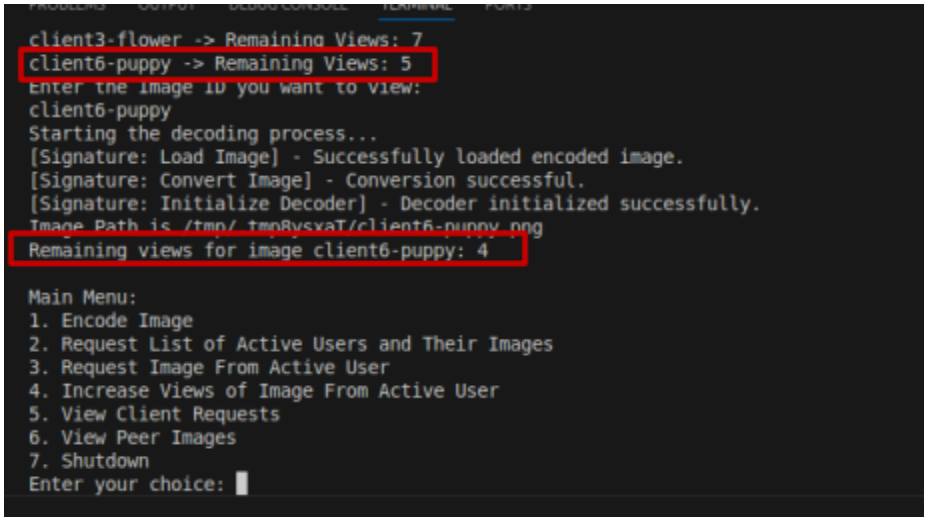

Figure 15 : Encoding the new Access Rights

When the number of views becomes 0, the default image is displayed upon requesting to view the image.

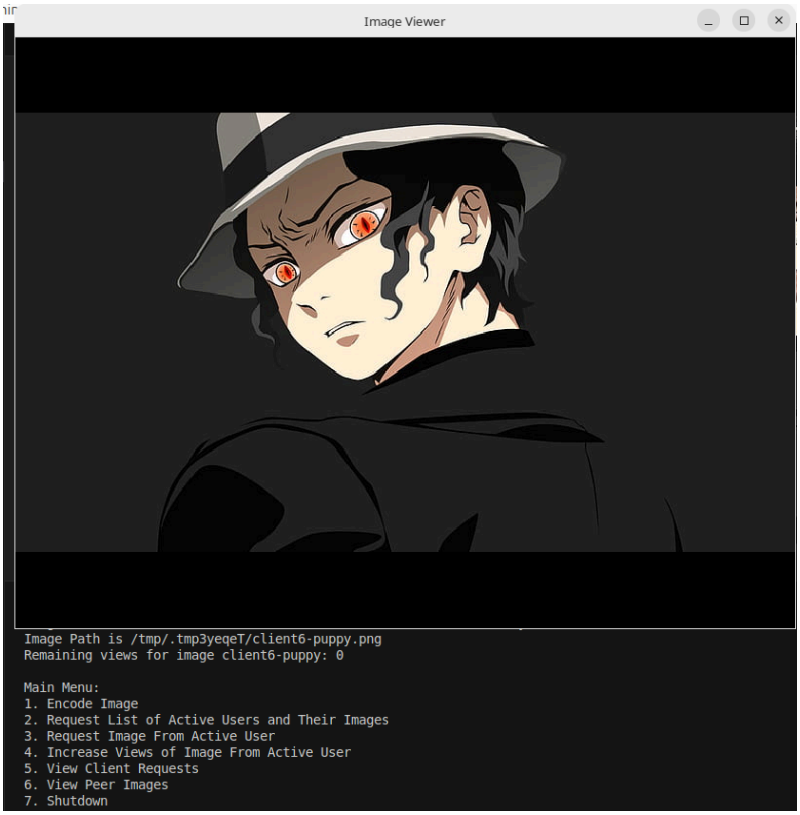

Figure 16 : Displaying the Default Image After Number View Reaches Zero

3.5 Requesting Additional Views

To request additional views for a peer image, the user selects option 4, **Increase Views of Image From Active User**, and is presented with a list of active clients and their images. The user must then select an image ID from a client they have previously requested access from and already received approval.

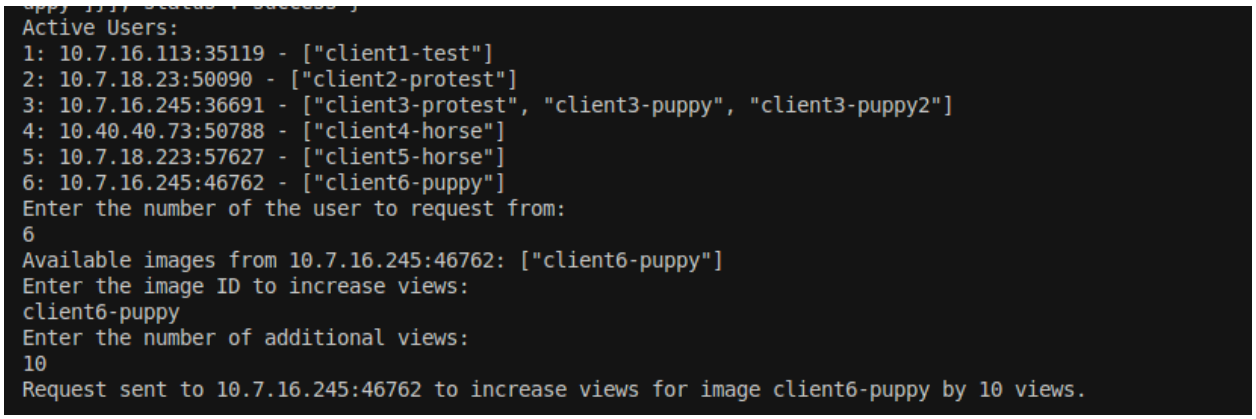

Figure 17 : User Requesting Additional Views from the Image Owner

The client receiving the request can also choose to accept or reject requests for additional views.

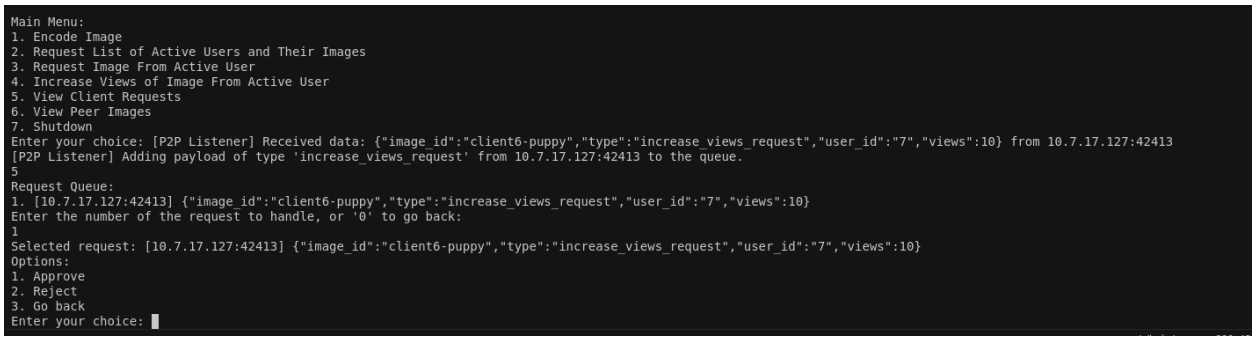

Figure 18 : Image Owner Receiving the Increase View Request

Number of views increased (from 0 to 10) after approval:

Figure 19 : Changing Access Rights Upon Receiving the Approval from the User

3.6 Requesting Additional Views with Failure Handling

The client requests additional views but then fails or shutdowns

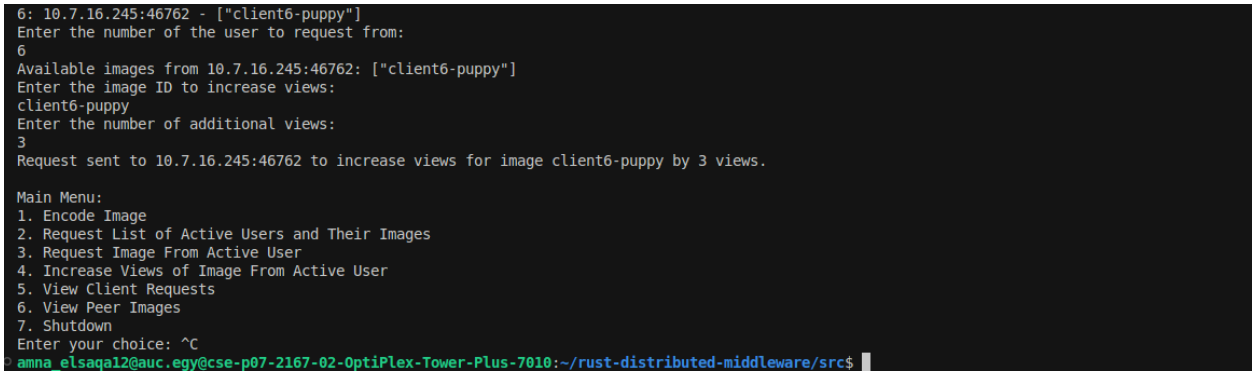

Figure 20 : User Shutdowns abnormally

If the peer responds to the request but does not receive an acknowledgment from the requesting client, it forwards its response to the servers. The DoS then ensures the response is delivered to the requesting client when they sign in.

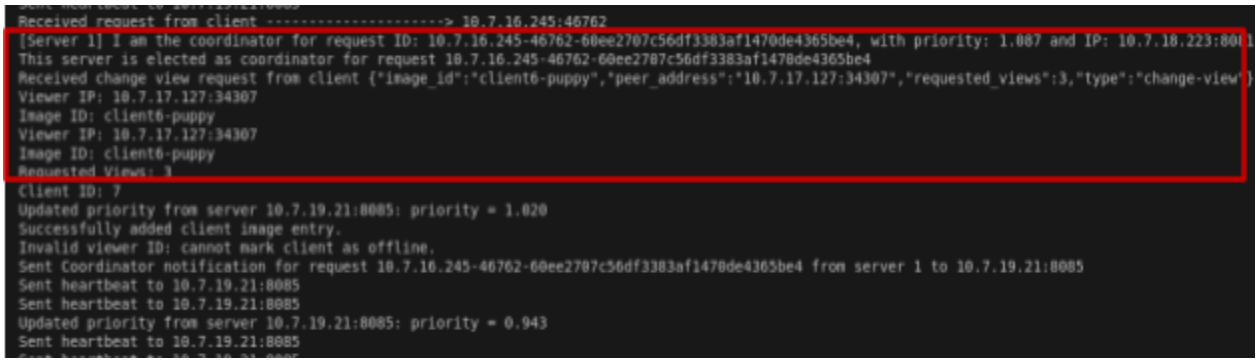

Figure 21 : DoS Receives the New access Rights to be Sent to the User Once Being Active

Upon signing in, the client receives the response and immediately updates the number of views for the image.

Figure 22 : User Receiving from the DoS the New Access Rights Upon Sign In

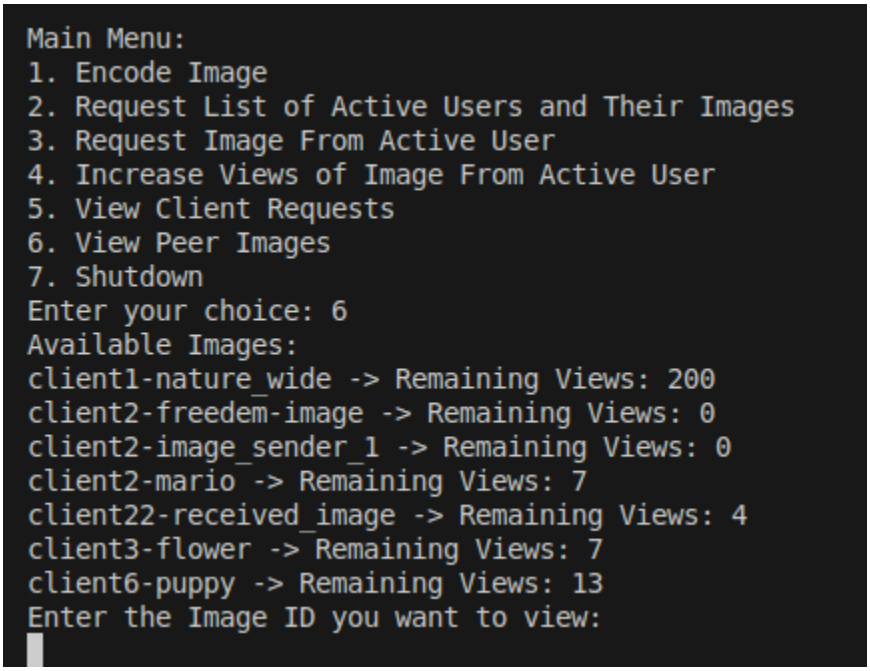

Figure 23 : Access Rights Updated

3.7 Client Controls Access Rights

To manage access rights, a client selects option 7, **Update Access Rights**, and is presented with a list of users they have shared images with, along with the shared images. The client can then choose to increase or decrease access rights for a specific shared image.

Figure 24 : Image Owner Update the Access Rights for certain client

The number of views decreased from 13 to 9:

Figure 25 : Client Application Middleware Embeds the New Access Rights

4. System Performance

4.1 Performance Metrics

Our performance evaluation centers around the following key metrics:

- **Response Time:** The time taken by the system to respond to client requests.
  This metric is computed as the time elapsed from the initiation of the request sent to the cloud until the complete reception of the encrypted image.
- **Failure Rate:** The proportion of requests that result in system failures or errors.
  This metric is computed as the number dropped/failed requests.

4.2 Testing Environment Setup

Stress testing is a very important process to test a distributed system's reliability and affinity to scale. In this experiment, we wanted to measure how our election algorithm, the modified bully, gives the user a better experience, in terms of average response time and total response time. In this experiment, we had a set up of three servers and three clients. We sent, simultaneously, N copies of a $10\,KB$ image to the servers for encryption. The simultaneous sending was implemented via tokio::spawn that creates multiple asynchronous tasks. Multiple trials were conducted to evaluate the system performance with changing the value N each trial, beginning at $N = 100$, increasing to $N = 1000$, and finally scaling up to $N = 10,000$. This incremental approach allowed us to systematically assess the system's handling of increasing workloads.

Figure 26 : Image Used for Testing the System Performance

It is important to mention that a non-transparent failure occurred when N was larger than 100, which was that some threads overflowed their stacks, which is why we set a cap to the number of threads that can run concurrently to a maximum of 50 threads running at the same time. This tweak reduced our failure rate to 0% across all experiments, meaning that no matter how many times we looped over the directory and sent its contents, and no matter which load balancing method we chose, all images were encrypted and sent back to the clients successfully. Once a thread at the client side was created, a timer would start, and then stop right before the thread was terminated. The elapsed time was recorded by each thread in a text file called times.txt. The times were then transferred to an excel file where the averages were computed and the plots were created.

4.3 Testing Results & Analysis

The plot below shows the average response time per request in milliseconds plotted against the number of requests sent for the modified bully (load balancing) and the round robin.

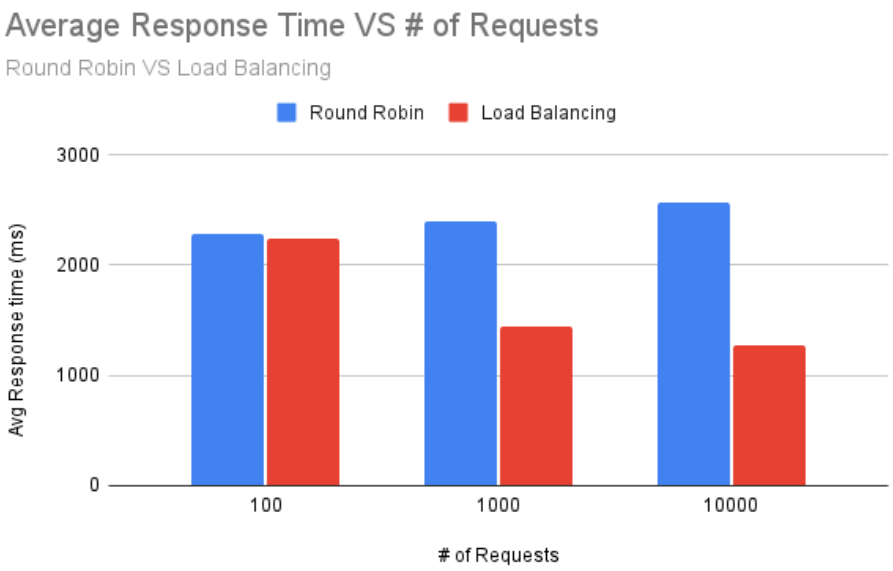

Figure 27 : Average Response Time With Respect to The Number of Requests Received.

We can see that for the round robin, as the number of requests increase, the average response time increases as well going from 2.2 to 2.6 seconds, while with the load balancing, the average

response time actually decreases as the number of requests increase going from 2.2 to 1.3 seconds.

The next plot compares the total response time for the entire batch of requests when using load balancing as opposed to using round robin.

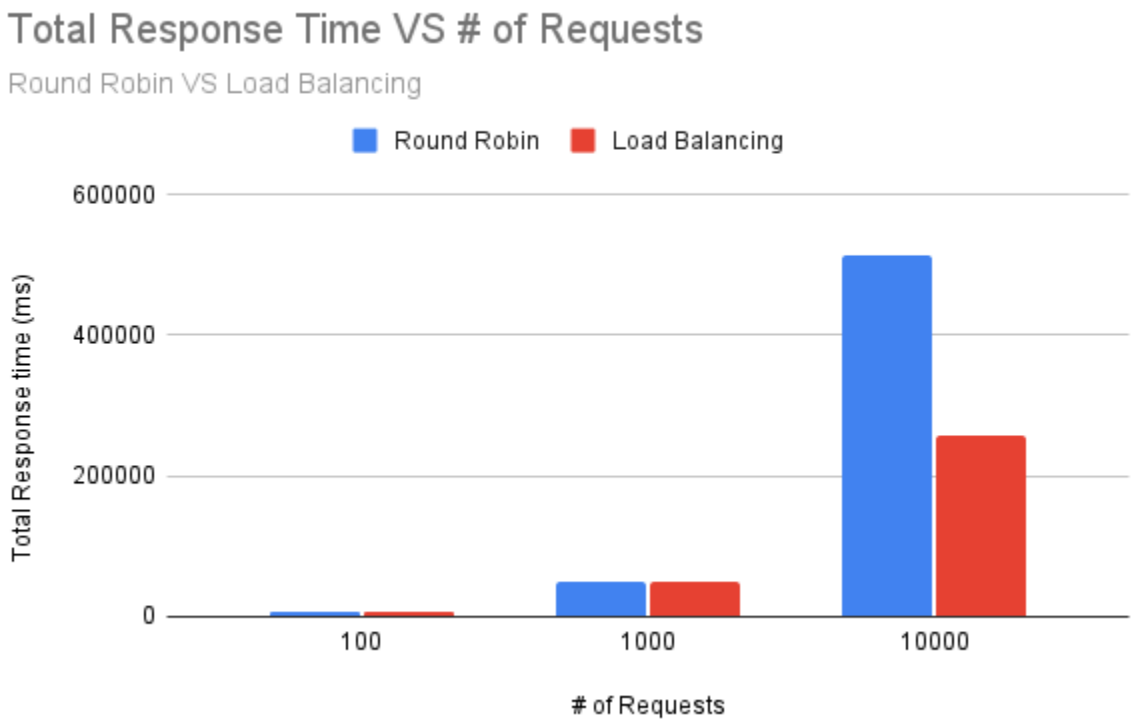

Figure 28 : Total Response Time With Respect to The Number of Requests Received.

It is clear from the plot that for sending 100 and 1,000 pictures, the two algorithms are almost identical, however, the superiority of our load balancing algorithm becomes evident as we send 10,000 pictures. The round robin algorithm took 515 seconds to respond to all 10k requests, while the modified bully only took 256 seconds finishing in almost half the time.

When It came to simulating failure with our election algorithm. We conducted the same experiment with three clients and three servers and looping over the image directory for 100,

1,000 and 10,000 times but the difference was that we shut down one server midway, and then recorded the success rate i.e. how many sent requests were handled. The results are in the table below.

# of requests	Success rate	# of failed requests
100	100%	0
1,000	99.90%	10
10,000	99.98%	2

Table 2 : Server Side Failure Effect on the Success Rates

As it can be seen here, the amount of failed requests does not increase with the amount of requests, which means that the success rate increases with the amount of requests. The explanation here is that when abrupt failure occurs, the system goes briefly into a state of instability where the other servers are not updated with the failure right away and a non-increasing number of requests fall in the middle. This number totally depends on when the failure happens and to which server. If it happens to a server that is taking requests, the number might increase a little. If it happens to a server that is not the coordinator, the number decreases.

5. Contributions

Member	Role
Ahmed Amin	● Worked on interprocess communication, developing the module responsible for facilitating communication between servers and clients, as well as client-to-client communication. ● Built and integrated client-side operations with server-side functionality, including the following: ○ User Registration and Sign-In: Developed mechanisms for clients to register and authenticate with the server securely. ○ Encoding Image Requests: Enabled clients to request image encoding using steganography, with feedback on the process's success or failure. ○ Viewing Active Users: Implemented a feature for clients to query the server for active users and their availability. ○ Sending P2P Requests for Images: Facilitated peer-to-peer requests, allowing clients to retrieve images directly from other clients using dynamically allocated ports. ● Improved and resolved issues related to steganography, including refining the image resizing process to support any image dimensions without restrictions. ● Developed and implemented fragmentation, serialization, and deserialization mechanisms for efficient data transfer. ● Designed and implemented various request types to support diverse system functionalities. ● Collaborated on the image viewing process by integrating decoding modules (for the access rights and the hidden image) into its workflow. ● Implemented the failure handling in the Directory of Service
Amena Hassan	● Worked on initial implementation for the election algorithm, which was later improved by Mario. ● Implemented the encryption module where an image is hidden (encoded) inside another default image using

	steganography. ● Implemented a decryption module that takes an encoded image (a default image with a hidden image encoded in it), and extracts the hidden image. ● Implemented the encoding and decoding of access rights. ● Handled all request types between clients for P2P communication. ● Implemented the following client functionalities: ○ Requesting peer images. ○ Receiving client requests and adding them to requests queue. ○ The option to view and approve or reject client requests. ○ Transmitting an encrypted image over UDP with reliability ensured through acknowledgements. ○ Sending requests to increase the number of views for a peer image. ○ Handling requests to increase views, allowing a client to approve or reject the request. ○ The option to view the list of images for which a client has access and the number of views (access rights) for each image. ○ An option enabling a client to forcibly update access rights for a peer with whom they have shared images. ● Implemented failure handling (client side, did not implement handling in DoS side) in case of approval or rejection of access rights changes or a forced modification of access rights ● Collaborated on the image viewing process by integrating decoding modules (for the access rights and the hidden image) into its workflow.
Mario Ghaly	● Added on Amena initial implementation of normal Bully algorithm and moved it to the modified Bully algorithm: ○ Solved Parallel elections problem ○ Added heartbeats implementation (sending and receiving) ○ Supported load balancing by adding priorities assigning based on CPU load, then IP address as a

	tiebreaker ○ Ensured only one server handles a client request ● Supported Failure Tolerance in the election algorithm ● Solved issues on the server side in handling multiple threads at the same time ● Created the MySQL tables of the DoS with Hamza ● Implemented all the DoS functionalities(SQL queries) on the server side with error handling: ○ User registration ○ User sign-in ○ Storing access rights modifications of an offline user ○ Enforcing access rights modifications from the DoS on the client when signing in ○ Image Encryption(adding its ID to user images list) ○ Returning list of active users and images they offer ○ User shutdown ● Handled Server-to-Client deserialization of received requests to correctly categorize the request type ● In the image viewing workflow: ○ Implemented the logic of decrypting the image with each view, saving it in a temporary folder, and deleting it when closing the image ○ Integrated Imageviewer handling ● In stress testing: ○ Did the first implementation of parallelizing images sending on the client side
Hamza Elgobba	● Worked with Ahmad on setting up the initial communication between senders and receivers ● Migrated to tokio UDP to ensure better reliability ● Modified parallelization of image sending from the client side to tackle errors in stress testing. ● Parallelized the encryption process on the server side by creating threads for each client request. ● Stress tested the system for image encryption for adequate performance ● Tested the system for fault tolerance ● Created a mySQL database that serves as a directory of service with Mario

	● Integrated the methods that invoked the DoS with the main code base. ● Implemented the p2p communication mechanism that involved p2p designated sockets with Amena ● Implemented the code that updates the DOS with the peer sockets ● Engineered the image viewing workflow in which a user gets the desired image decodes the access rights and stores them in a json file, then stores the decrypted image in a hidden folder, then views the image on demand and decrements the access rights, then deletes the decrypted image from the hidden file once viewing rights expire and displays the default image.

6. References

1. M. Gholipour, M. S. Kordafshari, M. Jahanshahi and A. M. Rahmani, "A New Approach for Election Algorithm in Distributed Systems," 2009 Second International Conference on Communication Theory, Reliability, and Quality of Service, Colmar, France, 2009, pp. 70-74, doi: 10.1109/CTRQ.2009.32.

2. Coulouris, George, et al. *Distributed Systems: Concepts and Design*. 5th ed., Addison-Wesley, 2011.