School of Sciences and Engineering

FALL 2023

CSCE 4411 Fund. of Distributed Systems

Project Report: A Cloud P2P Environment for Controlled Sharing of Images

Submitted by:

Abdullah AbdelAziz	900196083
Hashem Elmaleeh	900192812
Amr Sallam	900196011
Kareem Amr Talaat	900192903

Submitted to:

Dr. Amr El Kadi

Table of Contents

1 Introduction

This document explains in great detail the requirements, the design choices made, the results obtained through testing the system, and the user interface of the Fall 2023 Distributed Systems Course Project. In the first section we talk about the requirements of the project. The requirements are categorized into three categories. Requirements related to the servers, the clients, and the way encrypted images are accessed. The second section is the core part of the report and is about the design choices done throughout the development of the system. These include major design choices such as the choice of the election algorithm, fragmentation and flow control, and interprocess communication. In addition, we also talk about the optimizations done to improve the performance of the system. Examples of these optimizations are related to the choice of the default image, and the choice of the serialization technique. We also talk about different challenges such as the identification of client requests and images that are shared between clients. Then we discuss the use cases of the system and the results of testing the system. The penultimate section illustrates the compilation process for both the server code and the client application code. Moreover, we show the libraries used and their versions. Finally, in the last section we document the roles of each member of the team.

1.1 Project Description

One goal of the project is to have a service of encryption. Clients of the service send images to be encrypted. The images are encrypted and then sent back to them. No copies of the images are kept by the service provider.

The service provider in our case is a cluster of 3 servers. The servers, among themselves, are communicating in a peer-to-peer fashion. They are on equal footing; and they use an election algorithm to distribute the workload, i.e. load balancing. They also use the election algorithm to simulate failures. In other words, at random times, a server is elected to go down. The service should be uninterrupted in the case of such failure, and the server should be updated with the state of the system whenever it recovers from failure.

Another goal is to provide a directory of services that shows active clients. Using the directory of services, clients can connect together and share images in a peer to peer communication paradigm. The owner of the picture can constraint the shared image with a certain number of views. This number can be adjusted after the image is shared. A mechanism will be implemented to allow the enforcement of these adjustments once the client with which the images are shared is online.

1.2 Requirement list

Note: Distinction should be made between the application layer and the middleware layer. The application layer is infrastructure agnostic meaning that it knows nothing about the underlying network topology and connection details.

1.2.1 Requirements pertaining the Service cloud

1. The service has three servers.
2. Each server can communicate with any other server using a bidirectional communication channel.
3. The communication channels are best effort (Udp connections).
4. The three servers implement an election algorithm.
5. The election algorithm is used for load balancing.
6. A variant of the election algorithm is used for failure simulation.
7. The system is fail tolerant (in the case of simulated failures).
8. Recovering servers should be updated with the system state.
9. All servers implement an image encryption algorithm.
10. The servers don't keep copies of the images after the encryption is done.
11. Servers keep a directory of services which has the online users with whom other uses can connect.
12. The servers should share a token that enables connection between different peers.

1.2.2 Requirements pertaining the Client application

1. The client application supports dealing with encrypted images.
2. The client application supports sharing encrypted images with others.
3. The client application supports modifying access permissions of shared images.
4. The client middleware should be able to multicast a request to the servers of the service to encrypt an image.
5. The client middleware should implement a method for enforcing access modification in case of the other client being offline.

6. The client middleware should register the user to the system directory of service upon connection.

7. The client middleware should unregister the user to the system directory of service upon disconnection.

1.2.3 Requirements pertaining the Encrypted image

1. An encrypted image is in essence two images imposed on each other. One image is the data of the user, and the other image is a default image chosen by system designers.

2. The image shared by the user appears when the user shares the image with another client as long as the client did not exceed the maximum number of views of the image.

3. When the access permission expires (i.e. when the user hits the maximum number of views), the default image is shown to the user.

2 Design Choices and Technical Details

2.1 Election Algorithm Choice

In selecting the election algorithm for our distributed system, we carefully evaluated the merits of both the Bully algorithm and the Ring algorithm. After a thorough consideration of our system's requirements and characteristics, we opted for the Bully algorithm due to its superior attributes in addressing fault tolerance, simplicity, and decentralized coordination.

2.1.1 Choice Considerations

1. **Fault Tolerance and Recovery:**
    - **Bully Algorithm:** The Bully algorithm stands out for its robust fault tolerance mechanism. In the event of a node failure, that node will not be involved in the election process, ensuring system continuity. This capability aligns seamlessly with our objective of maintaining high availability and fault tolerance.
    - **Ring Algorithm:** While the Ring algorithm provides a decentralized structure, it does not implement a fault tolerance mechanism inherently and it will need additional modifications and testing to obtain a fault tolerance mechanism.
2. **Message Overhead:**
    - **Bully Algorithm:** The Bully algorithm typically incurs lower average message overhead during the election process; **N - 2**.
    - **Ring Algorithm:** The Ring algorithm may require increased message passing, particularly in larger systems, as information about a node's state must propagate through the entire ring; **3N - 1**.

2.1.2 Technical Details

We decided on using a modified version of the Bully algorithm that works as shown in figure (1).

When a server receives a request from a client, it starts an election. This is done by sending a message to all peer servers, which includes the initiator’s priority. Only receivers of higher priority reply with their priority, then, send an election message to their peers, if they have not already. The process continues until the initiator does not receive any replies, within a specific time period, which means that it is of the highest priority. Each exchanged message has the priority of the replier and the id of the client request that is under consideration at the moment. This allows for multiple election processes to take place at the same time to handle different requests from different clients. We decided on adding a random wait time for each node before calling for an election. This will reduce the chances of having multiple election initialization requests at the same time. This election algorithm inherently implements fault tolerance; any failed server will not participate in the election process.

The following patterns are utilized in the implementation:

1. **Concurrency with Async/Await:**
    - The implementation extensively uses asynchronous programming with `async` and `await` to handle concurrent tasks, especially in network communication.
    - `tokio::spawn` is used to run tasks concurrently.
2. **Mutex for Shared State:**
    - The `ServerStats` struct is wrapped in a `Mutex` to handle concurrent access to shared state across multiple threads.
    - `Arc<Mutex<ServerStats>>` is used to share the server stats among different asynchronous tasks.
3. **Selective Waiting:**
    - `tokio::select!` Is used to wait for either a timeout or a condition to be met, allowing for efficient asynchronous waiting.

2.1.3 Servers Priority Metric Choice

Since our application heavily depends on the threading capabilities of the servers, we decided to use the CPU load average metric over the past 1 minute to decide the next leader. CPU load is defined as the number of processes using or waiting to use one core at a single point in time. We assign the task to the server with the least load. This CPU load is first fetched by querying the OS structures to get the average load. Then the load is subtracted from a big number. The result of the subtraction is used as the priority of the servers. Most of the time the priorities are unique. However, under some conditions, the two machines might have the same load. In this case we use the address of the servers to decide the handler of the request.

As we tested the system, it performed well using the mentioned priority metric. However, the choice of the metric remains open for future exploration. The system heavily depends on the network bandwidth, in addition, memory needs to be taken into account as we store the images received from clients in memory. This is in addition to the default images which are cached in memory to enhance the performance of the servers. Taking these factors into account is left as future work.

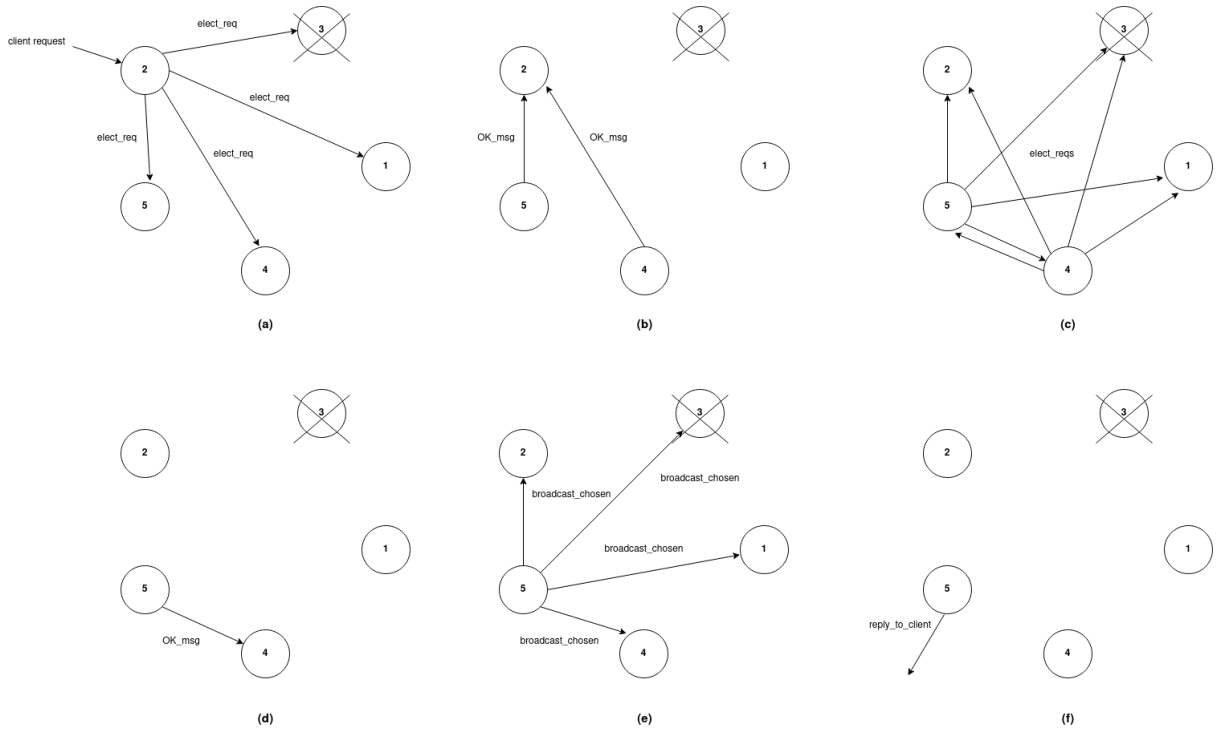

client request
elect_req
elect_req
elect_req
elect_req
OK_msg
OK_msg
elect_reqs
OK_msg
broadcast_chosen
broadcast_chosen
broadcast_chosen
broadcast_chosen
reply_to_client
1
2
3
4
5
(a)
(b)
(c)
(d)
(e)
(f)

figure (1)

2.2 Identifying Client Requests and Images

The election algorithm is done on a request basis, so we had to distinguish requests from one another. The id of a request is composed of the IP address of the client and a request index that is different across requests for a given client. This is done through incrementing an index whenever a new request is made. The index is maintained over sessions to eliminate the case of having requests with identical ids in a short time span. In this context, short is relative to the time a request is cached in the servers' memory.

We also need to identify images that are shared between clients. This is done through giving each image an ID as follows. It starts with the IP address of the owner of the image (source), then the IP of the borrower (destination), and finally the image name as used to store the image in the owner's storage <SOURCE_ADDRESS>&<DESTINATION_ADDRESS>&<IMAGE_NAME>.

2.3 Client Requests Handling

In our system design, the client multicasts its request to all the servers. This starts off the election algorithm to see who will handle the request. But because this piece of information is absent by the time the request arrives to servers, we decided to cache the client's request until a server is chosen to handle the request.

We found this caching to be an opportunity in case the server chosen to handle this request fails. When the server fails, it sends a message informing other servers that it is going down. This starts a new election to decide who will service the client. This is done to make use of the already cached data and to reduce the delay experienced by the client. Even though the data might be big, we think that this is an additional reason to use the cached data instead of requiring that the client sends it again. If the server successfully completes the task, the data is dropped from the cache of servers.

2.4 Failure Simulation

We decided to simulate failures using a token that is passed between servers in a round robin fashion. When a server receives the token it fails for a variable duration of time up to 20 seconds. The failure simulated is a fail-stop. Meaning that the node's behavior is predictable; it will not process any requests from clients, and it will not participate in elections that take place during its failure.

2.5 Listening on Multiple Sockets

We decided to make our cloud servers listen on multiple sockets. A server listens simultaneously on two sockets: one for client communication and the other for communicating with peer servers.

The client communication socket is exposed externally to handle client requests. A server receives client requests from clients through this socket and uses it in responding to clients if this server is the chosen one to handle a given client request. The inner server communication socket is dedicated to communication with peer servers, it allows the server listening on it to share information with other servers, i.e., notifying other servers of its state in case of failure or recovering from a failure or performing a task that requires server-to-server interaction like electing the chosen server to handle a given client request. This decision was taken primarily to enhance the security of our system, as separating the sockets reduces the risk of security breaches between client and inter-server communications. In addition, the way a server handles client communication differs from the way it handles communication between other peer servers, so separating the two communications allows for distinct data handling, control, and monitoring, ensuring a more robust system.

2.6 Marshaling: Serialization Technique

Marshaling is an important step to ensure that the communication between devices that may represent data differently is correct. There are different ways to marshal data, the easiest of which is to serialize data into a textual format. For example, to convert different arguments and data structures into JSON. However, as we experimented with different serialization techniques, we noticed that the textual format increases the message size drastically. In particular, the serialized data was almost three times the size of the unmarshalled data. In addition, different serialization techniques took different time periods to serialize and deserialize objects. The performance of different serialization techniques in terms of size increase and time required for serialization and deserialization is recorded in the following table. Note that this is for the serialization of an object of size 8 KB.

Serialization format	Serialized size (KB)	Serialization time (µs)	Deserialization time (µs)
JSON	28	672.8	905.8
Concise Binary Object Representation (CBOR)	14	283.8	383.5
Rust Message Pack (RMP)	14	638.2	712.0
Bencode	35	1526.6	2307.2

Based on the data provided in the table above, we decided to use CBOR serialization in our communication.

2.7 Default Image Selection

To encrypt images we use steganography which is based on embedding the bytes of the secret (the data we want to hide) in a default image. However, we are limited by the number of pixels of the default image. To support big secret sizes, we need to have bigger images, but then clients who wish to encrypt light weight secrets (in our case images) will experience a big delay that is disproportionate to the images they sent. To handle this problem, we use different default images with different sizes. In particular, we use six images of sizes 484 KB, 997 KB, 2.71 MB, 4.18 MB, 10.4 MB, and 19.9 MB. This allows us to cater for a wide range of client image size while not compromising the speed of encryption and communication.

2.8 Fragmentation and Flow Control

As mentioned in the previous section, we support a variety of image sizes, and to transmit these images to the recipient, we need to fragment the image. This is to avoid packets being dropped because of links' MTUs. In addition, as more and more clients use the system, server buffers' may get full; hence, some messages may be dropped. To avoid this, a flow control mechanism that is based on ACKs is implemented as well. However, receiving an ACK for each message will severely downgrade the performance. Thus, we ACK blocks of fragments.

The fragmentation process goes as follows. First, we need to configure the fragment size as well as the block after which we wait for the acknowledgement of the receiver. Then, the big message is fragmented into fragments of the chosen size. Then a fragment message is constructed with the two extra pieces of information. The first is the total length of the big message, and the second is the index of the fragment in the big message. After sending a block-size of fragments, the sender waits for an ACK from the receiver. If the ACK is not received within a time period of 2 seconds, the block is retransmitted.

The receiver on the other side has the knowledge of the fragment size and the block size. Therefore, when it receives a fragment, it reserves a space in memory for the whole big message. Not only this, but it will place the fragment in its place in the big message based on its index which is provided in the fragment message. Whenever a block-size of fragments arrive, the receiver sends an ACK with the block index.

The figure below shows an example where a message consists of 6 fragments and a block consists of 2 fragments. Note that at each row of the figure the right side represents the sender and the left side represents the state of the receiver before receiving the messages sent by the sender.

At the state captured by the figure, the first two fragments were successfully sent and ACKed by the receiver. Therefore, the sender sends the next block which consists of fragments 3 and 4. However, since UDP is not reliable, message 4 didn't arrive. Because the second block was not received successfully, the receiver doesn't send an ACK. Consequently, the timer at the sender times out and the second block is sent again. This time, the both fragments arrive at the receiver and an ACK is sent back. The sender then sends fragments 5 and 6.

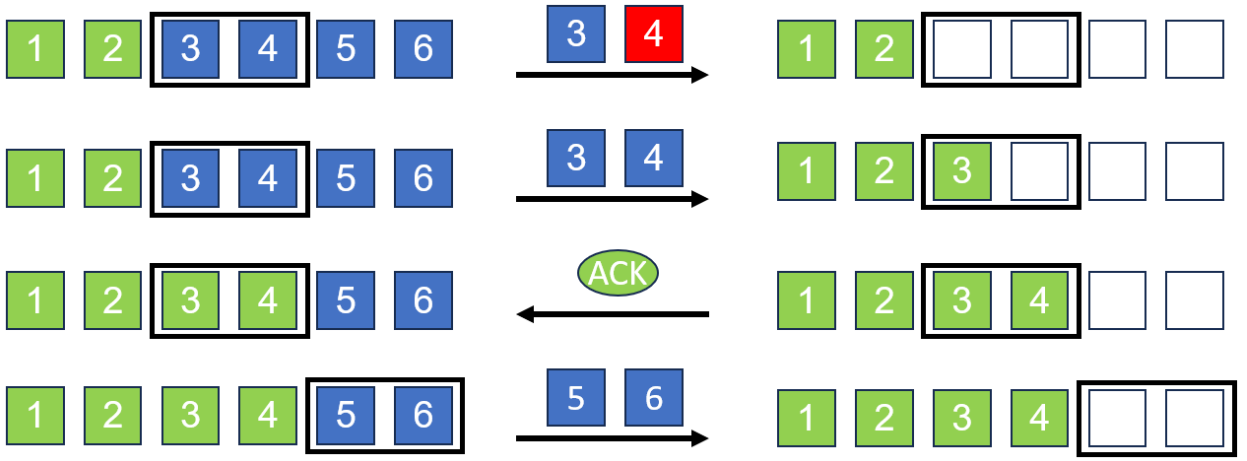


figure (2)

Under this described model, a lot of cases arise that need to be handled carefully. One such case is when the ACK is lost. In this case, it is assumed that the receiver didn't receive the message and the whole block is sent back. Another case is that messages may arrive out of order, but we account for this case since the fragment ID specifies its place in the message. Different cases were handled in the implementation of the described method to ensure correct execution under different cases.

2.9 Coding Framework Choice

We chose to work with the Tokio framework to optimize cpu usage. Tokio is an asynchronous runtime for the Rust programming language. It provides the building blocks needed for writing network applications. At a high level, Tokio provides a multi-threaded runtime for executing asynchronous code. This allows us not to block on operations that we know will take a lot of time and are not necessary to be done at the moment. In addition, we can spawn many Tokio tasks which is equivalent to a user thread. These tasks are multiplexed on a number of system threads. Whenever one tokio thread is waiting for an operation to be done, it can allow another task to run until it finishes. Having the tasks done by the server and the client implemented as tokio tasks allows for fast switching between tasks without the need to do context switching which is required if we are using actual system threads.

Tokio was a suitable choice for most of our tasks in both the server and the client since most of the operations are reading and writing to the disk or a network socket. However, the encryption task is a cpu bound task, and hence, Tokio was not suitable for that task. Because of this, we chose to spawn a system thread using Rayon to do the encryption then return the result to Tokio task[1].

2.10 Interprocess Communication

Interprocess communication (IPC) is a crucial aspect of distributed systems, allowing different processes to exchange information. In our project, IPC is primarily achieved through UDP sockets and Tokio's asynchronous runtime.

UDP (User Datagram Protocol) is a connectionless and lightweight transport layer protocol. The choice of using UDP sockets over TCP ones avoids the overheads associated with TCP.

As mentioned in section 2.5, the cloud servers are listening on multiple sockets: election, service, and send sockets. The cloud servers use the election socket to communicate with each other. On the other hand, the service socket and the send socket are used by the cloud servers to communicate with the clients to receive a client request or to reply to a client, respectively. Each socket is handled by a thread, and communication between different sockets is done using shared memory (mutex) or message passing (channels) based on the type of data to be shared between different threads and how frequently it will be shared.

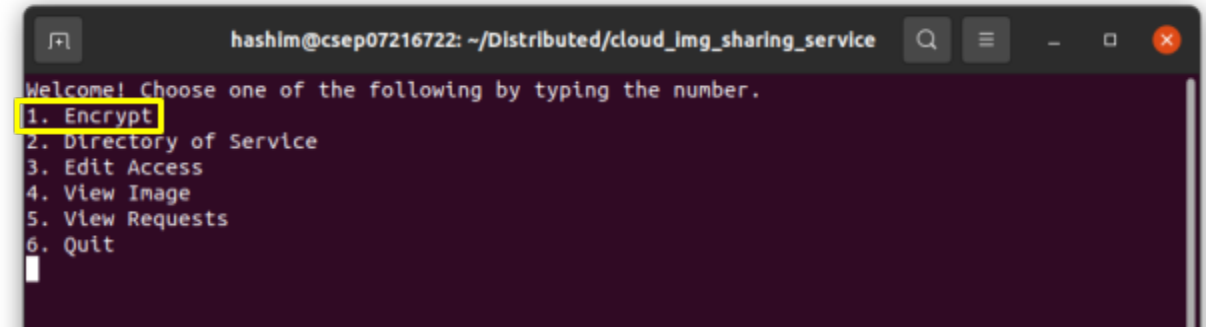
Upon receiving a client request, the thread responsible for the service socket spawns a thread to handle the required service requested by the client and it associates a communication channel between it and the newly spawned thread to handle sending acks back to the request's owner. This allows the cloud servers to send back ACKs to multiple clients at the same time without any conflict or any drop of the acks which will cause a retransmissio for the fragments and hence increases the response time for the corresponding client.

[1] This pattern is recommended by Alice Ryhl one of the maintainers of Tokio on her personal website.

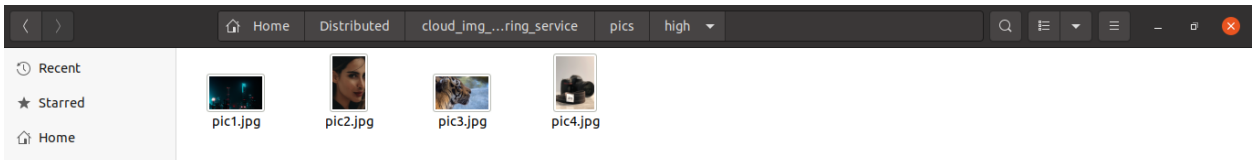
3 Use Cases and Experiments

3.1 Encrypting an Image

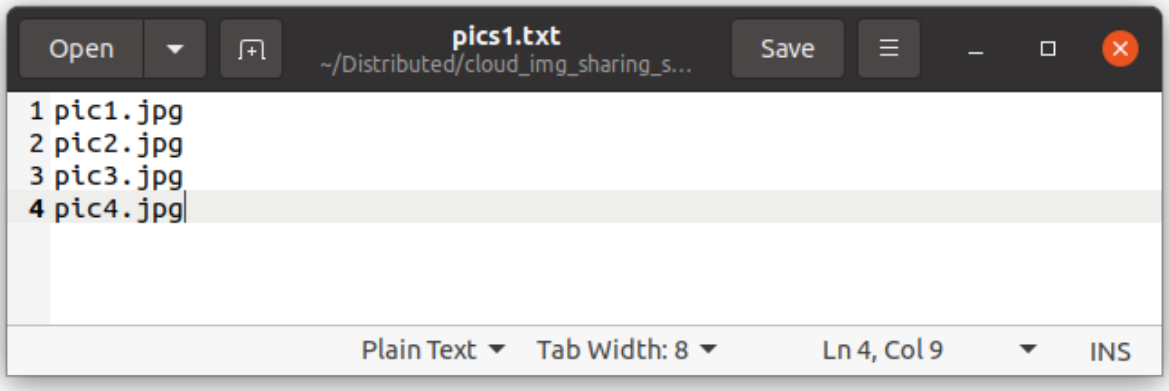
When users open the client application program, they see the following menu. The first option of this menu is to encrypt an image.



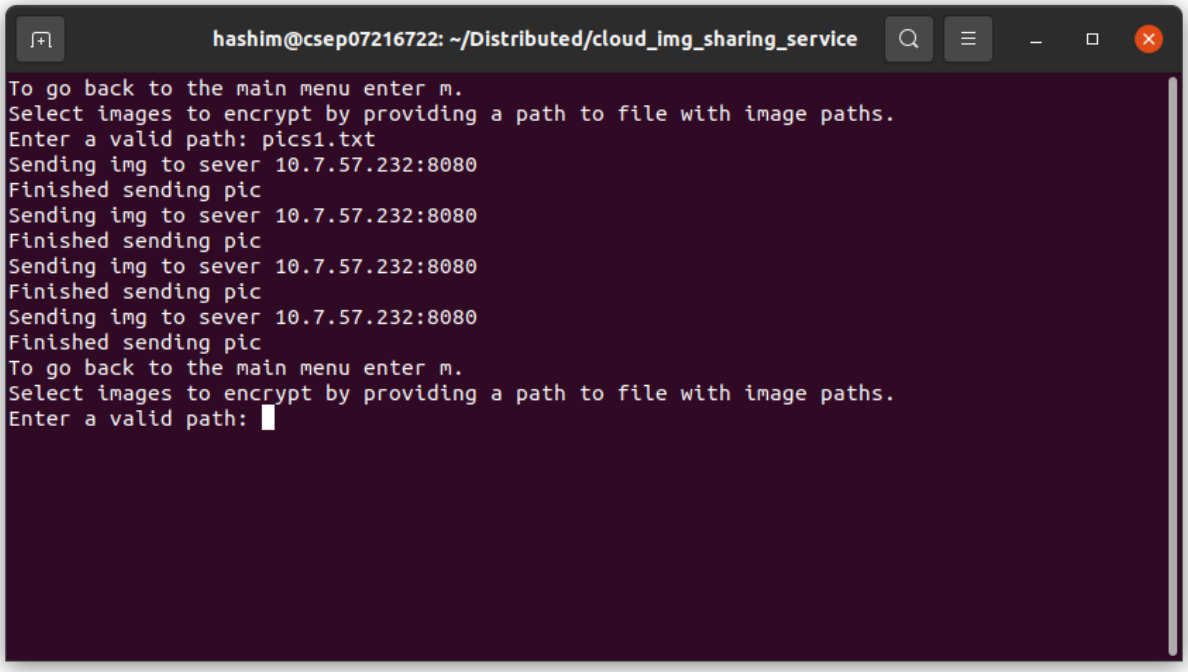
The user can encrypt images that they store in pics/high directory. Below is a screenshot displaying the images in the pics/high directory of a sample client.



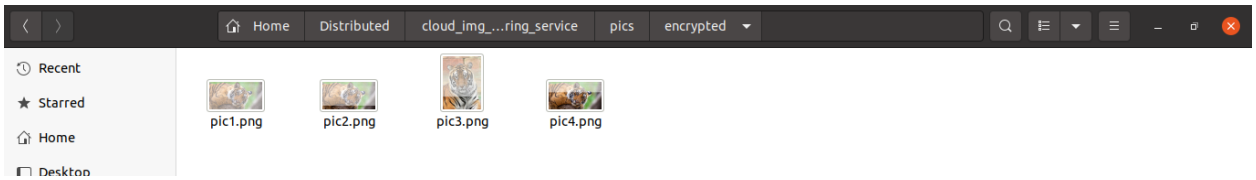
The user can choose which images to encrypt by writing the images names in a text file then input the path to this file to our program. Below is a screenshot showing the content of a text file that will be used to encrypt the images shown above.



When the user chooses to encrypt images by typing 1 in the menu screen, the user is prompted to enter the path of the text file. Then the system sends these images to the server and stores the replies in pics/encrypted directory.

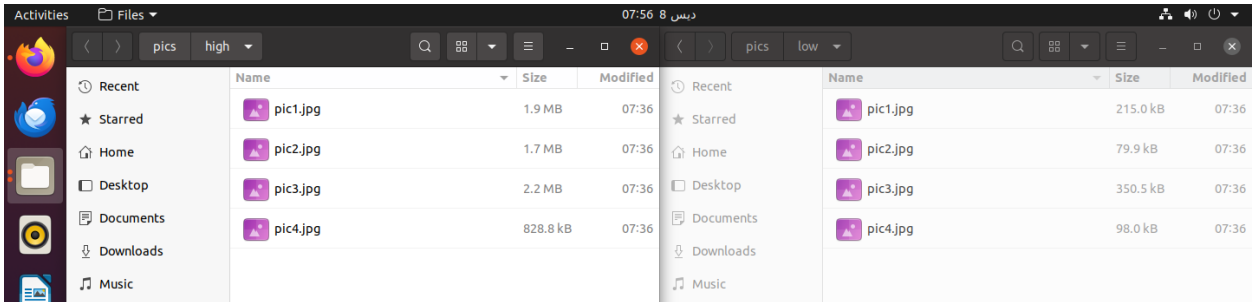


Below is a screenshot of the contents of pics/encrypted folder. Notice how pic3 which is a small image in size is encoded in a default image of small size to reduce the latency of requests. In addition, if any of these images is opened using any image viewer, the default image is shown, and the images cannot be viewed except through our application.

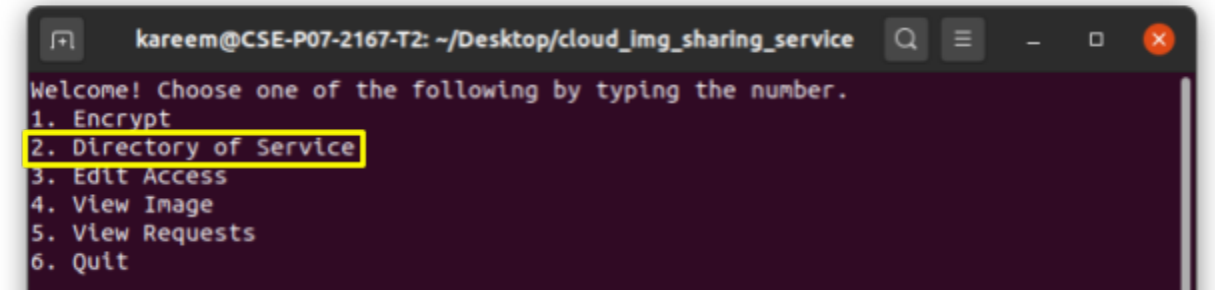


3.2 Directory of Services and Requesting Images

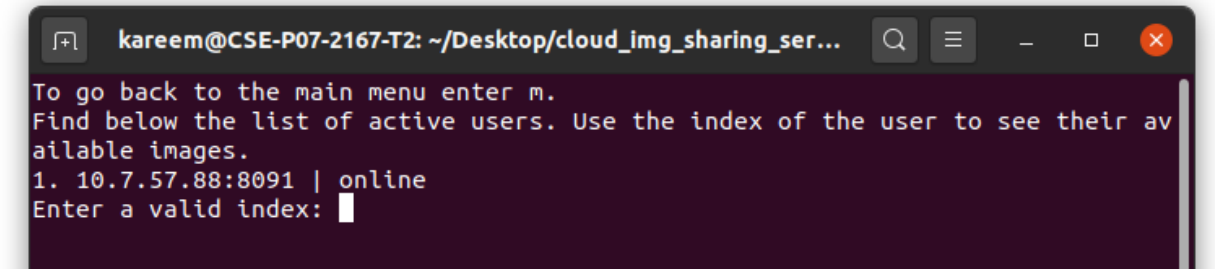
Each user has images that they are willing to share. These images are stored under pics/high and pics/low directories. The images in the first directories are the images with their actual sizes. In the pics/low directory a low resolution copy of each image is stored. Notice that the size of the low resolution images is much less than the original images. This allows for a fast transfer of images between clients.



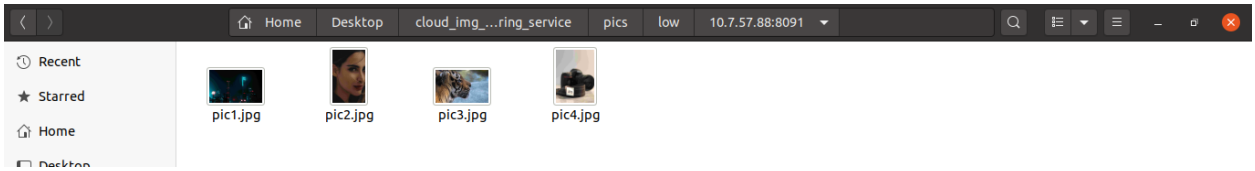
The second option in the main menu is to view the directory of services. In the directory of services, clients can view other online clients.



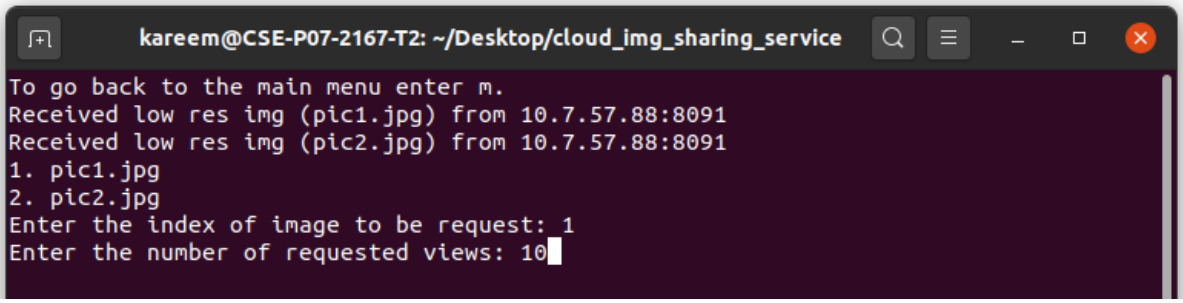
Upon choosing option 2, the user can see a table of active users. From that table clients can choose a particular client to view the images they are offering to share. These images are stored under a directory with the name of the client that is offering the images. When the user types 1 to choose client 10.7.57.88:8091. A directory is created with low resolution images that are stored in the low resolution directory on the device of the other client.



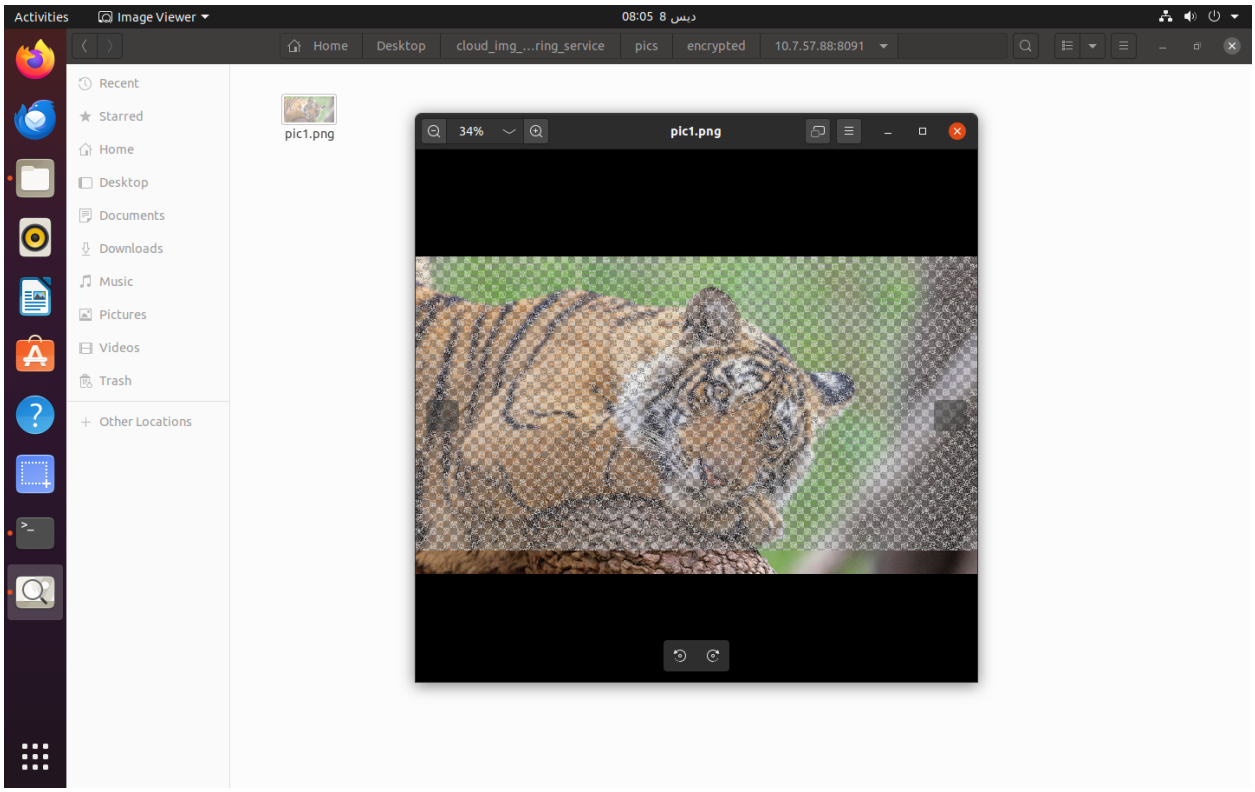
Below is a screenshot of the folder.



Upon viewing images, the user can select one image to be sent with higher resolution. The user can also select the number of views they will be allowed to view the image.

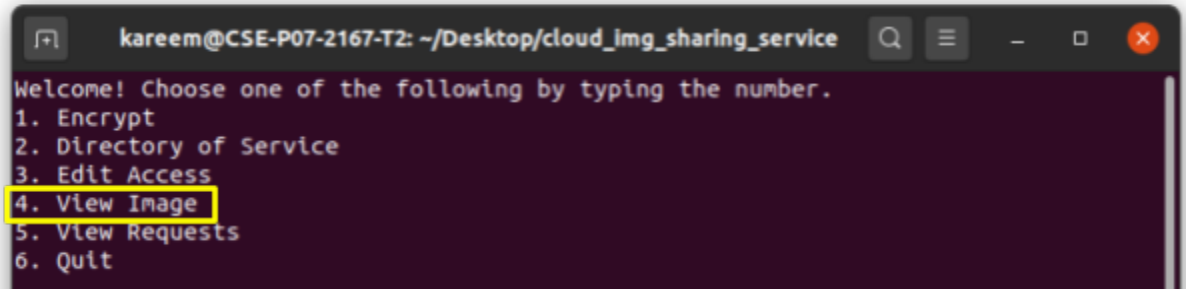


Upon submitting this request, a new file is created under a directory holding the name of the owner client with the encrypted image. As mentioned earlier, opening the image using any image viewer will view the default image.

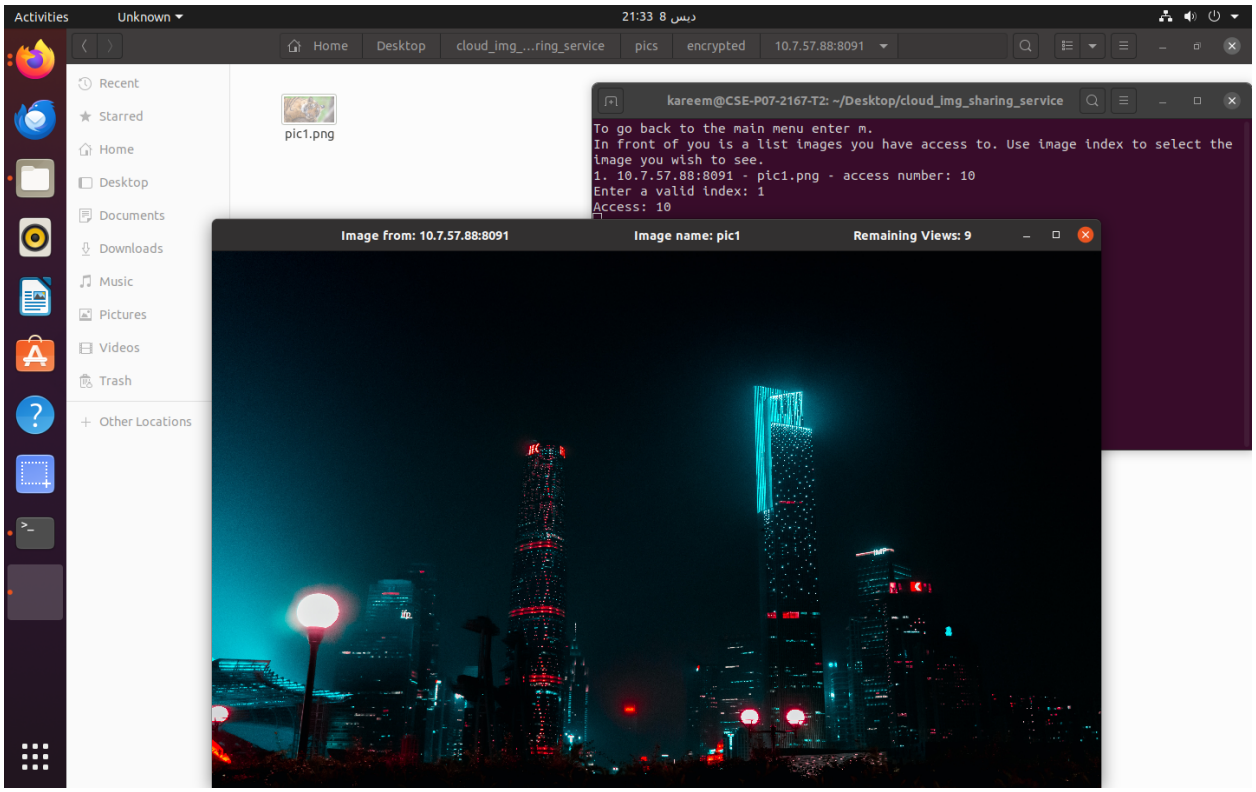


3.3 Viewing Requested Images

A client can view the requested image the allowed number of times. This can be done by selecting option 4 in the main menu.

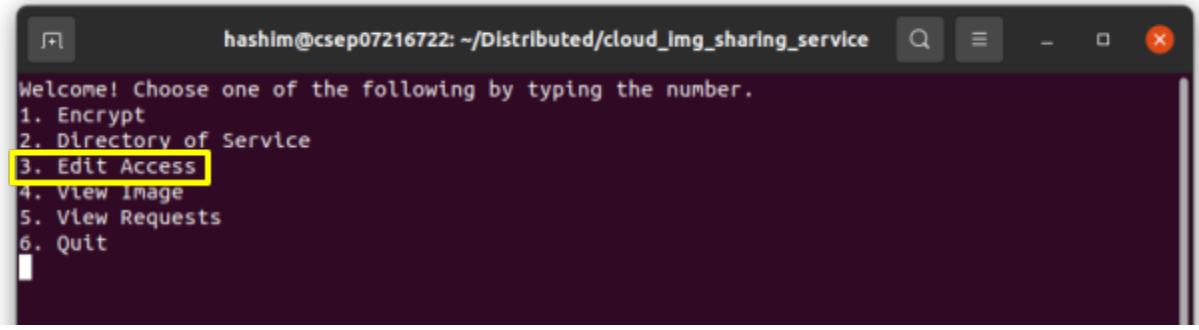


Upon choosing this option, the user can choose the index of the image to view. A pop up window appears and the user can view the image. The number of access is decremented with each access. Below is a screenshot showing the terminal input used to select the image, the image pop up and the encrypted image that the viewed image is extracted from. We believe that this is the most secure way of viewing the images since it can only be done through our software.

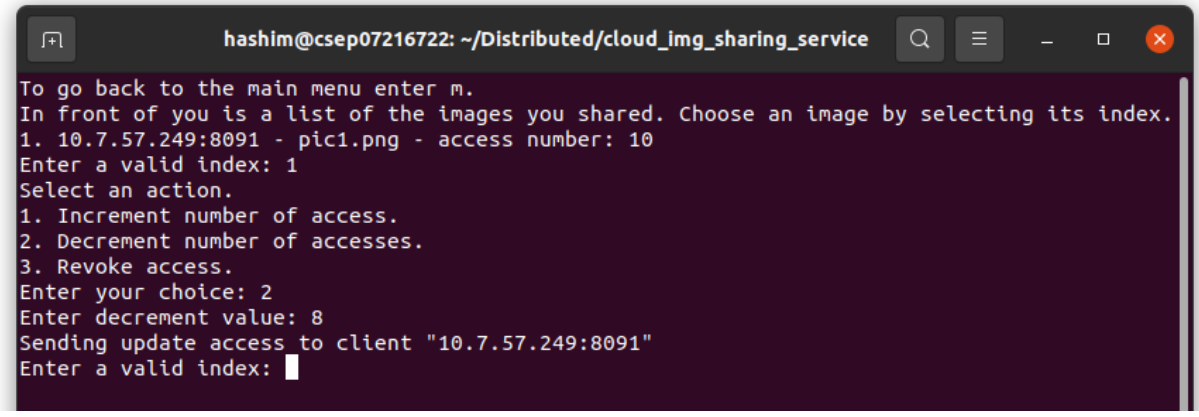


3.4 Online Access Update

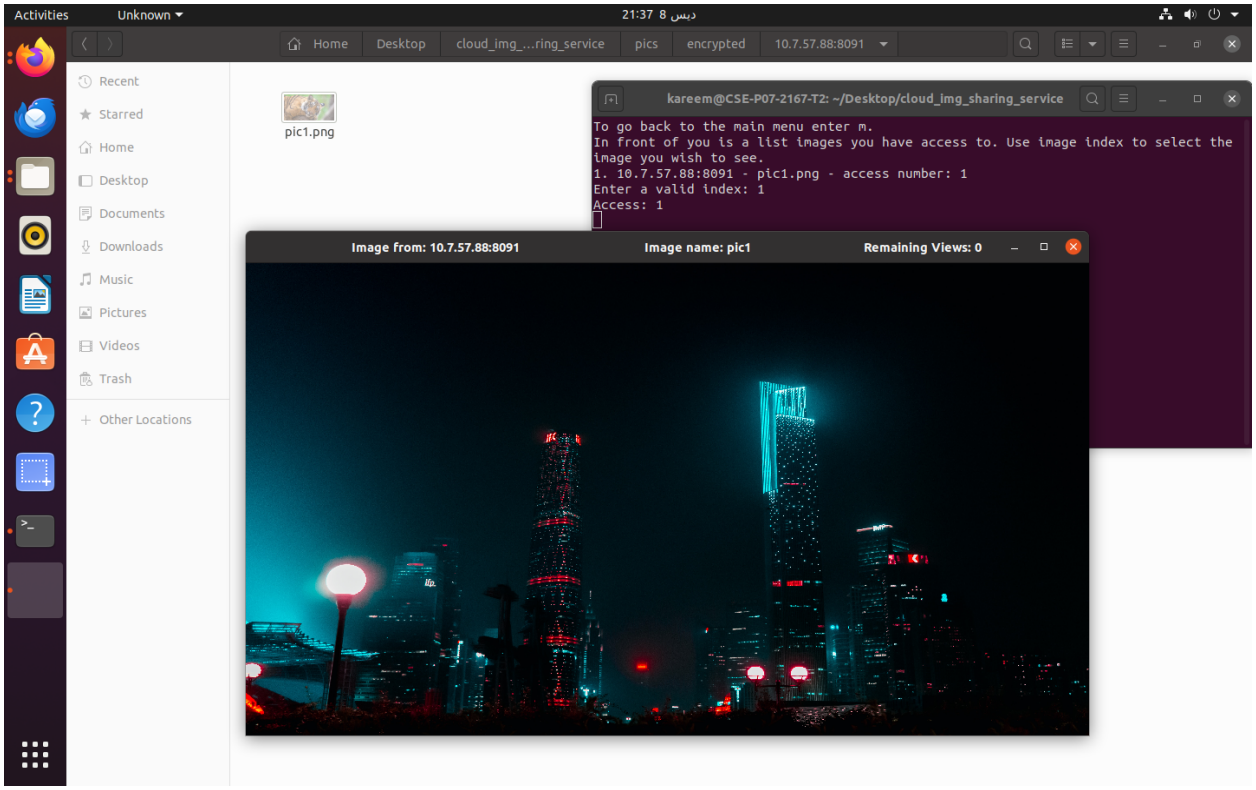
The owner of the image can control the number of accesses the borrower can see. This is done through selecting option 3 in the main menu.



The owner can see a list of images with the images shared with other clients. Then the user can select this image and choose an action to be done to the image. Actions can be incrementing the number of accesses, decrementing the number of accesses or revoking the access of the image. In the screenshot shown below. The owner decremented the number of views by eight. Since the borrower is online, the update action is sent to the borrower.



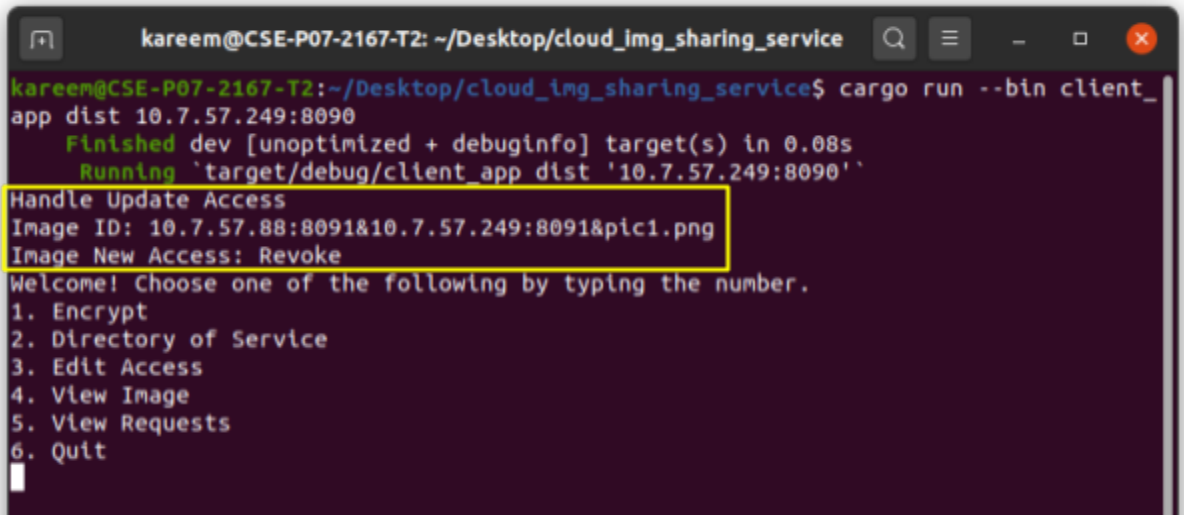
This action is applied to the image immediately. Shown below is a screenshot with the number of views decremented by eight.



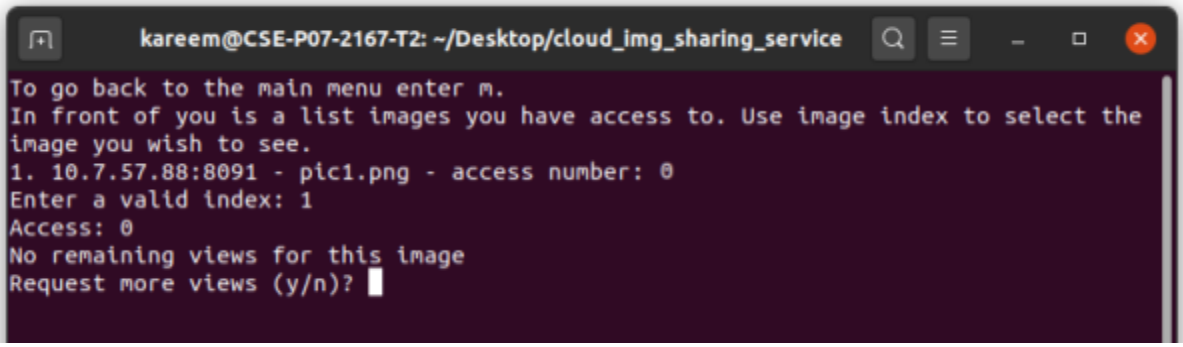
3.5 Offline Access Update

When the borrower is offline, the owner can still change the number of accesses through sending the action and the picture ID associated with the action to the servers. The servers then can enforce these actions whenever the borrower comes online. This can be done through the same screen as the online access.

When the borrower client logs in again, the action is enforced.

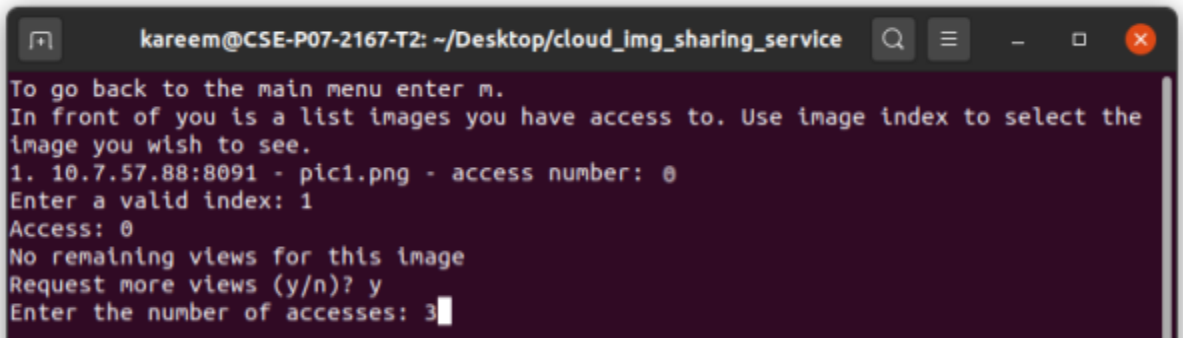


If the borrower tries to view the image, they are shown a message indicating that they have no more accesses.

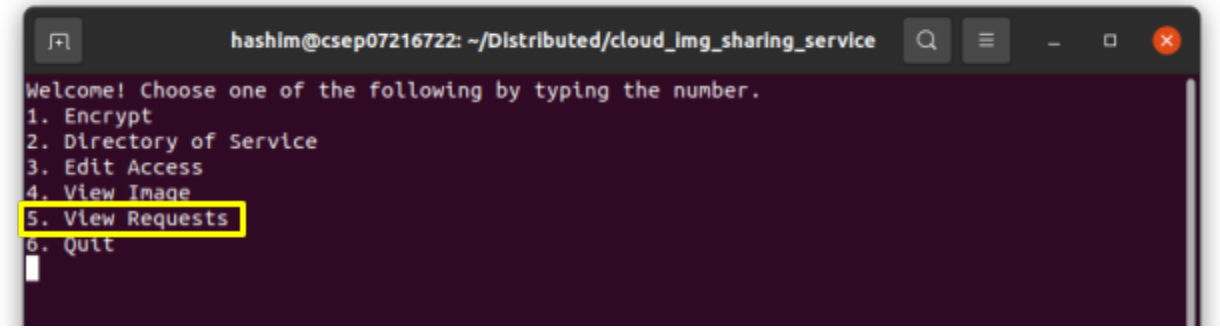


3.6 Requesting More Views

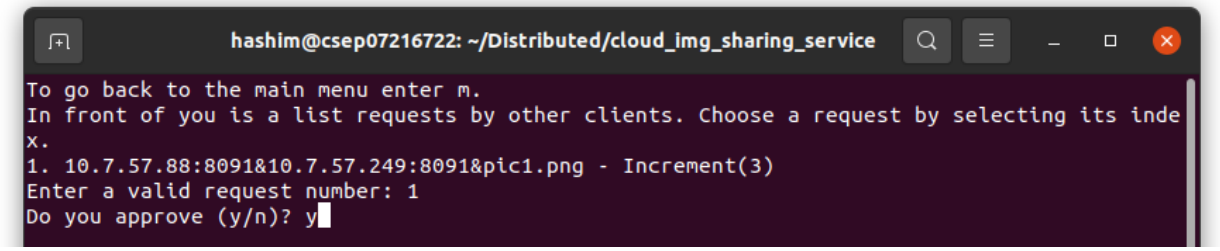
When borrowers run out of accesses, they can request more accesses from the owner. This can be done by responding to the message shown in the previous subsection. The screenshot shows a sample borrower requesting three more views.



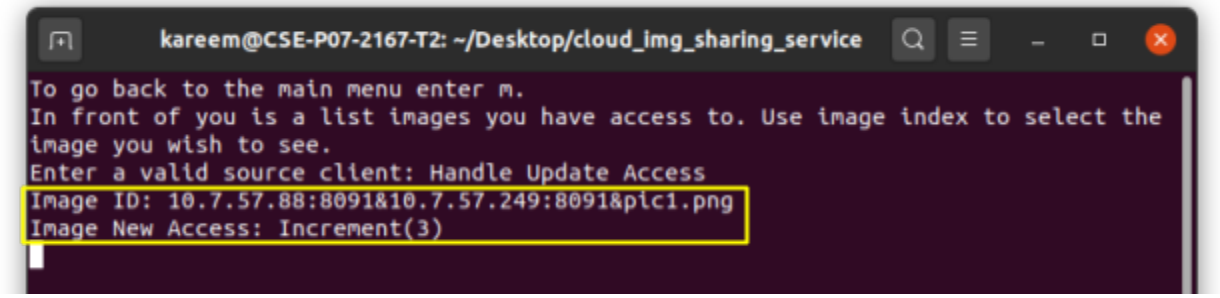
The owner can view requests by different borrowers from screen five from the main menu.



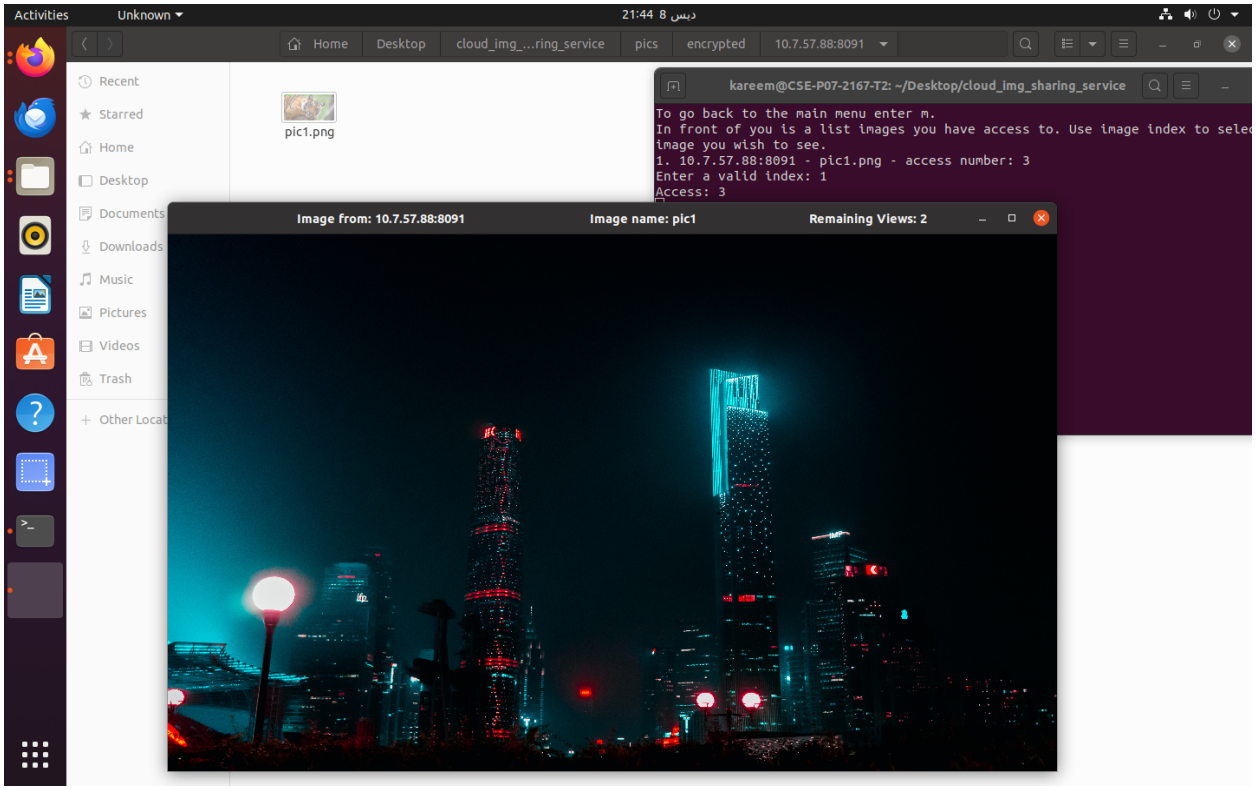
The owner can then choose whether to approve or disapprove the request.



Upon approval, the borrower user gets a notification with the approval of the request.



After this, the borrower can view the image.


Activities
Unknown
21:44 8 ديس
Home
Desktop
cloud_img_...ring_service
pics
encrypted
10.7.57.88:8091
Recent
Starred
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
Trash
Other Locat
pic1.png
kareem@CSE-P07-2167-T2: ~/Desktop/cloud_img_sharing_service
To go back to the main menu enter m.
In front of you is a list images you have access to. Use image index to selec
image you wish to see.
1. 10.7.57.88:8091 - pic1.png - access number: 3
Enter a valid index: 1
Access: 3
Image from: 10.7.57.88:8091
Image name: pic1
Remaining Views: 2

4 System Performance

As we delve into the evaluation of our distributed system's performance, it is imperative to assess its behavior and capabilities, particularly focusing on metrics that define its efficiency and reliability. This section aims to provide a comprehensive overview of the performance evaluation conducted on our system, emphasizing key aspects such as response time and failure rates. The evaluation spans both project phases, aiming to capture the evolution of the system's performance from its initial implementation to the incorporation of load balancing mechanisms.

4.1 Performance Metrics and Goals

4.1.1 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.

4.1.2 Goals

Our goals for system performance are defined as follows:

- **Response Time:** Maintain a response time below a specified threshold to ensure a seamless user experience.
- **Failure Rate:** Minimize the failure rate, striving for robustness and reliability even under heavy loads.

4.2 Evaluation Scenarios

To comprehensively assess the system's performance, we conducted evaluations under various scenarios, including:

- **No Load Balancing Scenarios:**
  - **Normal Load Conditions:** Assessing the system's behavior, without load balancing, under typical operating conditions to establish a baseline performance.
  - **Heavy Load Conditions:** Simulating scenarios with increased traffic and load to evaluate the system's scalability and robustness, in case of no load balancing.

- **Load Balancing Scenarios:**
    - **Normal Load Conditions:** Assessing the system's behavior, with load balancing, under typical operating conditions to evaluate the effect of load balancing .
    - **Heavy Load Conditions:** Simulating scenarios with increased traffic and load to evaluate the system's scalability and robustness, in case of load balancing.

These scenarios allow us to capture a holistic view of the system's performance, providing valuable insights into its strengths and areas for improvement. The subsequent sections delve into detailed analyses of each scenario, shedding light on the nuances of our system's behavior and the effectiveness of load balancing strategies.

4.3 Evaluation

4.3.1 No Load Balancing:

4.3.1.1 Normal Load Conditions:

To assess this particular scenario, a single server will be utilized, and a minimal load of 1 client will be applied with a one second interval between successive requests. 50 images of average 4 Kbs will be used.

Average Response Time per image (s)	Failure Rate (%)
1.06	0

4.3.1.2 Heavy Load Conditions:

To assess this particular scenario, a single server will be utilized, and a load of 1-6 clients will be applied with immediate successive requests. 500 images of average 4 Kb will be used.

	Average Response Time per image (s)	Failure Rate (%)
1 Client	0.99	0
2 Clients	1.43	0
3 Clients	1.71	0
4 Clients	2.34	0
5 Clients	2.92	0
6 Clients	3.18	0

figure (3)

4.3.2 Load Balancing:

4.3.2.1 Normal Load Conditions:

To assess this particular scenario, three servers will be utilized, and a minimal load of 1 client will be applied with a one second interval between successive requests. 50 images of average 4 Kb will be used.

Average Response Time per image (s)	Failure Rate (%)
0.98	0

4.3.2.2 Heavy Load Conditions:

To assess this particular scenario, three servers will be utilized, and a load of 1-6 clients will be applied with immediate successive requests. 500 images of average 4 Kb will be used.

	Average Response Time per image (s)	Failure Rate (%)
1 Client	0.98	0
2 Clients	1.01	0
3 Clients	1.04	0
4 Clients	1.08	0
5 Clients	1.12	0
6 Clients	1.18	0

figure (4)

4.3.3 Discussion:

The impact of load balancing on system performance is evident, markedly reducing the response time by nearly one third. Figures 3 and 4 illustrate a considerably faster growth in response time when load balancing is not employed, indicating that the use of load balancing could further decrease the response time by a significant factor.

It's important to highlight that the failure rate consistently remains at 0 in all scenarios, underscoring the effectiveness of the robust and reliable communication module that has been developed.

4.3.4 Response Time Breakdown:

For the response time breakdown analysis, we consider an example of 2 images with sizes 1.9 MB and 4.8 kB:

	Image 1 (1.9 MB)	**Image 2 (4.8 kB)**
Total Response Time (s)	3.1	0.93
Election result	0.53	0.52
Sending to cloud	0.26	0.002
Encryption on the cloud	0.72	0.17
Sending back to client	1.59	0.24

The primary factor influencing response time is the transmission of the encrypted image, which is directly correlated with the increased size post-encryption. Following this, encryption time

contributes to the overall response duration. Additionally, the time required to obtain the election result falls within the range of 0.5 to 0.53 seconds, influenced by the pre-defined TIMEOUT parameter set at 500 milliseconds for the election algorithm. These observations present potential areas for performance enhancement, particularly concerning TIMEOUT adjustments. Despite these considerations, the current performance is deemed satisfactory, especially when considering the size of the image undergoing encryption.

5 Compilation Procedure

5.1 Server

cargo run –bin server dist <ip>

```
amrmsallam@majskbt:~/Desktop/Fall23/Distributed_Systems/project$ cargo run --bin server dist 10.65.150.17
    Finished dev [unoptimized + debuginfo] target(s) in 0.10s
     Running `target/debug/server dist 10.65.150.17`
[]
Server (service) listening on 10.65.150.17:8080
Server (election) listening on 10.65.150.17:8081
Server (send back) on 10.65.150.17:8082
```

5.2 Client

cargo run –bin client_app dist <ip:port>

```
amrmsallam@majskbt:~/Desktop/Fall23/Distributed_Systems/project$ cargo run --bin client_app dist 10.
65.150.17:9000
    Finished dev [unoptimized + debuginfo] target(s) in 0.09s
     Running `target/debug/client_app dist '10.65.150.17:9000'`
Welcome! Choose one of the following by typing the number.
1. Encrypt
2. Directory of Service
3. Edit Access
4. View Image
5. View Requests
6. Quit
```

5.3 Libraries

```
rand = "0.8.5"
rand_chacha = "0.3.1"
tokio = { version = "1.33.0", features = ["full"] }
serde_cbor = "0.11.2"
image = "0.24.7"
rayon = "1.8.0"
sysinfo = "0.29.10"
minifb = "0.20.0"
log = "0.4.20"
```

6 Roles

Member	Role
Abdallah Abdelaziz	● Implemented the fragmentation part, with reliable communication by using Acks and retransmissions. ● Implemented the client interface which included the menus shown to the client and the flow in each of the screens. ● Implemented the offline update feature to allow for enforcing updates when a client is offline on the server side. ● Implemented recovery from failure mechanism for each server to keep updated with the pending (offline) updates that need to be enforced. ● Optimized the encryption process through the use of rayon thread to run concurrently with other tasks. ● Tested and documented the use cases of the system. ● Added a logging feature to the client for traceability of events and errors. ● Tested different serialization techniques and documented the results. ● Contributed to the testing of the performance through recording the delays experienced by clients and saving them for analysis. ● Contributed to the modularity of the project through the conversion of the client middleware to a struct that keeps track of the status of the client (in collaboration with Amr). ● Integrated with other parts of the project.
Amr Sallam	● Implemented the election algorithm. ● Extensively tested the election algorithm implementation to account for corner cases. ● Implemented the following features in the directory of service: ○ Online access update ○ Join ○ Leave ○ Query (both from a client and from a server after failure recovery) ○ Request low resolution images ○ Part of the sharing images feature (request and handling) ○ Request additional access for a specific image. ● Modified the user interface to be index-based, instead of typing image and folder names, which involved implementing a mechanism to keep track of which images are received, from whom and how many accesses are left for each. ● Thoroughly debugged various unexpected system behaviors, successfully pinpointing and resolving issues, such as unexpected

	behavior in asynchronous functions and optimizing the efficient use of exclusive access primitives like "Mutex." ● Implemented some utility functions: ○ Parse server Ips and return all associated ips (service, election, and sendback). ● Integrated other parts of the project and solved any unexpected incompatibilities while integration.
Hashem Elmaleeh	● Implemented the encryption module in which a secret image is hidden inside another default image using steganography. ● Implemented a decoder that takes an encoded image (a default image carrying the secret image), and extracts the secret image. ● Implemented utilities functions to read and write images from and to disk efficiently and only when needed. ● Extended the encoder and the decoder modules to handle any image extension. ● Designed and implemented a mechanism to handle and serve different client requests in parallel. ● Constructed a mechanism to accommodate for variant image sizes to be encoded by providing a wide range of default images to be used. ● Enhanced the response time per client request by sending images as raw bytes rather than pixel bytes. ● Associated a number of view parameters within the encrypted images to be sent to clients requesting the image. This parameter can be altered only through the owner of the image. Once the permitted number of views are consumed, viewing the image is denied unless the owner allows for more views. ● Implemented the skeleton for the peer to peer client communication. ● Implemented an image viewer which has the following functionalities: ○ Extracting the secret image from the encoded image. ○ Viewing an image in a GUI window which is user friendly. ○ Checking the eligibility of viewing the requested image. ○ Conserving the aspect ratio of the displayed image allowing for window resizing. ○ Decrementing the remaining number of views upon closing the displayed window. A window can be closed through the normal close button or pressing “Esc” key. ● Integrated with other parts of the project.
Kareem Amr Talaat	● Researched Different Election Algorithms: ○ Conducted an in-depth exploration of various election algorithms, providing comprehensive documentation highlighting the strengths and weaknesses of each

	algorithm. ○ Presented findings to the team, facilitating an informed decision-making process for selecting the most suitable election algorithm for the system. ● Integration and Collaboration: ○ Actively participated in the integration process, ensuring seamless collaboration between different components developed by different team members. ○ Worked closely with team members to integrate the client interface with the implemented election algorithm and directory of services, fostering a cohesive system architecture. ● Image Viewer Development: ○ Implemented an initial version of the image viewer, contributing to the project's multimedia functionality. ● Developed some utility functions: ○ Implemented a mechanism to preserve the last request ID for each client, saving the information into a file. ○ Designed functions to create output directories if they do not exist to ensure organization of file structure. ○ Implemented checks for the existence of a given path in the file system to be implemented in other functions such as file reading. ○ Reading server IPs from a text file to further complement the configuration process and the overall scalability. ● Environment Setup: ○ Worked on setting up laboratory PCs.