P2P Cloud Project Technical Report

CSCE4411- Fundamentals of Distributed Systems

Fall 2023

Presented to Dr. Amr El Kadi

Nada Badawi 900201364

Salma Aly 900203182

Amy Samy 900193256

Allaa Elkhouly 900201771

1. **Introduction**

Throughout the semester we have been working on developing a cloud system that is composed of three servers and three clients that communicate together using peer to peer communication. The advantages of our system is that it has an election algorithm that undergoes load balancing and fault tolerance to improve the system performance. There’s a stark difference between running the program with and without load balance–it can be seen from the values provided in section 4 where we bombarded the cloud by sending it multiple images for encryption.

1. **Features Implemented**

This section outlines all the features implemented:

* 1. **DoS:**

The purpose of the directory of service is to keep track of the users that have logged in and out of our application in order to display to the users the current online users so that they would be able to request images for viewership.

The first thing the user is greeted with when they run the program is a prompt to login. No other feature is displayed. The user is requested to enter their username which is then stored in the following struct:

struct User{

username: String,

logged\_in: bool,

dos: HashMap<String, SocketAddr>,

my\_images: Vec<DynamicImage>,

images\_received: HashMap<String, Vec<HashMap<DynamicImage, i32>>>,

images\_sent: HashMap<String, Vec<HashMap<DynamicImage, i32>>>,

}

The username entered is then sent to the cloud alongside a LOGIN message, informing the cloud that the user wants to login and so the leader then pushes an entry into the online\_users hashmap which contains the username and ip address of the user. Once it does so successfully, the server sends a message to the client informing them LOGIN\_SUCCESS in which the client middleware changes the client’s logged\_in boolean to true and sending the clients image for encryption if they are not already encrypted. The send\_after\_login function checks to see if all the images have been encrypted and if there are low res versions of them from previous sessions, if not, it does the operations. After the user is logged in, they are presented with the available functionalities.

* 1. **Access Rights**

The process begins by encoding the number of views a user has into an image through steganography. It converts the view count into bytes and embeds this message into the image's alpha channel, using the steganography crate. Later, to decode this embedded information, the code loads the image, initializes a decoder object, and filters out bytes that don't represent the concealed message, reconstructing these bytes into a readable string. This seamless concealment and extraction of data within the image using steganographic techniques provide a covert means to embed and retrieve information, such as the number of views associated with the image. Instead of sending the image with the number of allocated views with it. We read the views allocated inside the image itself.

The client will have the ability to edit the access rights it gave one of its peers and revoke it completely by sending to the client a EDIT\_ACCESS\_RIGHTS message. If the user client middleware doesn’t respond then the client editing the access rights will assume the client is offline, send to the server the message, and the server will edit the access rights of the receiving client when they go back online.

* 1. **Sending + Displaying Low Res Images**

Part of the client to client communication is requesting to see other user’s images. This doesn’t require permission because they are small, low quality images that the user generates upon logging in via the get\_low\_res\_image function which resizes it to a fraction of its size.

When the user chooses option 2 after logging in, they are prompted to enter the username of the person they want to view the low res images of. The middleware will retrieve the client's IP address from the DOS hashmap in the user struct and send a REQUEST\_LOW\_RES message to them. Subsequently, the middleware on the receiving client will send the images back to the requesting client, who will display them using the Window option in the minifb library.

* 1. **Sending + Displaying High Res Images**

To view the high res images of another user, clients have to get permission from the owner of the image in order to view it. That is why when the user chooses option 3 they are prompted to enter the username and number of times they want to view the image. The middleware then sends a REQUEST\_HIGH\_RES message to the middleware of the image owner. The owner of the images then responds by determining the number of views they wish to give the requesting client and then sends the encrypted image with the access rights embedded inside. The middleware of the client that requested the image then checks if the number of views is greater than 0, if it is then the middleware decrypts the image and displays it the same way the low res images were displayed. The number of views is then decremented.

* 1. **Load balancing + fault tolerance**

Every row of the process status table consists of a server ID along with its CPU utilization. Every server regularly sends its CPU utilization to all other servers and the PST is always sorted by the CPU utilization in ascending order. We then run the election algorithm every 5 seconds to assign a leader. The leader chosen is based on having the lowest CPU utilization. Once a client sends a request, the leader accepts the request. It is also important to note that if a server accepts a request when it is the leader, then a new leader emerges, the original server will still continue handling the request it accepted. That's how we implement load balancing in our project. We divide the work or the requests across all servers so that they balance the load. As for fault tolerance, there is a token that is passed to all servers in order. If the server has the token, it will make sure that it is currently not handling any requests, then it will send an infinite CPU utilization and fail. Otherwise, if the server is busy, then it will just pass the token to the next server.

* 1. **Image Fragmentation**

The images we are dealing with are 4k and their individual sizes are larger than the max UDP socket size and so we have to use image fragmentation to send the images in chunks.

The mechanism starts out by calculating the number of chunks needed based on the image size and a specified chunk size. It then iterates over these chunks, reading each chunk from the image file and sending it individually to the receiving end. There’s a one-second delay between sending each chunk to ensure it is received. On the receiving side, the chunks are received and concatenated into a vector named encrypted\_image\_data. This vector is continuously extended with the data received from each chunk until the total image data is retrieved. The function then loads the concatenated encrypted image data into a DynamicImage using the image::load\_from\_memory function. This process effectively breaks down the image into manageable chunks for transmission over the network and reassembles them on the server side to obtain the complete encrypted image.

**III. Architecture & Design**

1. **Client Side:**

Every user is a struct that consists of a username, a boolean variable for login, a hashmap for the directory of service, a vector of owned images, a hashmap of the images received, and a hashmap of the images sent. All threads communicate with each other via mpsc (multi-producer, single consumer) channels.

1. **Server-Sender Thread:**

The thread is responsible for sending any requests the client wants to make, which is either logging into the system, logging out, or asking to encrypt images.

1. **Server-Listener Thread:**

This thread is responsible for always listening to the servers once a response is received for the image encryption requests made or for any updates regarding the Directory of Service (DoS). If an encrypted image is received, it is saved into the directory to view it once needed. If DoS updates are received, then the cached DoS is updated, accordingly.

1. **Client-Listener Thread:**

This thread listens to all clients for any incoming requests, such as requests for the low resolution of image of the user, or a copy of the encrypted images. This allows a user to be navigating elsewhere in the application and not having to wait for a response to each request made. To illustrate, if the user with username nada sends to another account of username salma a batch of 5 requests for encrypted images, then nada can receive them, even if she exited the requests page. Also, nada will receive them in one batch. Alternatively, if this feature was not divided among two separate threads, then nada will have to send one request for a single image, wait to receive the image from salma, then send the next request, wait to receive the image, and so on… The difference between methods does have a significant influence since the time taken from the moment nada sends a request until she receives back the image depends on human interaction. This is due to the fact that salma will have to accept/decline and adjust the number of views if accepted to grant nada the access. Therefore, the system shouldn’t force nada to wait until she receives the image or a decline message from salma. That is the importance of having a separate listener thread to clients and not only listening when sending a request in the client-sender thread.

1. **Client-Sender/Main Thread:**

This is the thread that the user interacts with in the application. The program displays a menu for the user to choose what they wish to do. First, the user is prompted to log into the system by asking for the username they wish to proceed with. Next, they get prompted with six different options: list online users, request low resolution images of an online user, request encrypted images from an online user, view an encrypted image, edit access rights to a previously sent encrypted image, or logout.

* If the user chooses to list the online users, then the cached DoS is read to display its content as long as the users are marked as online.
* If the user chooses to request low resolution images, which means that they want to view another user’s profile, then the request is made through socket communication. The DoS is used to learn the client’s ip by searching with the username.
* If the user requests for an encrypted image, then socket communication is also used to tell the peer client how many views are requested, and for which image. After that point, the thread does not wait for a response from the peer client as explained earlier in the report.
* If the user wishes to view a previously requested image, then the number of views that are granted now is checked if it is greater than zero, which means that the user is eligible to view the image. Otherwise, the user will receive a message that they do not have access to the image.
* If the user chooses to edit access rights, then they must send the user the new number of views that should be granted, and which image they are referring to, and to whom this change is made. The user sends to the peer the new data. The peer should change the number of views that are stored in the hashmap that corresponds to the image specified and user sender to the new number.
* The final option is logging out, which channels to the server-sender thread to inform the servers that the client is going offline.

1. **Server Side:**
2. **Server-Listener Thread:**

This is a listening thread that is always listening to other servers and willing to receive messages from them. Messages from this thread are sent to the working thread using a channel called lw\_serv\_mess\_tx.

1. **Election Worker Thread:**

This thread has two jobs: It parses the message received from other servers and it constantly updates the PST.

If the first character of the message is 0, then this is a CPU utilization message. This means that one of the other servers is sending its CPU utilization to the current server so that it updates its PST.

If the first character of the message is 1, then this is a token message. This means that the current server will send true over the token\_tx channel so that the sender thread will handle this.

1. **Server-Sender Thread:**

This thread is divided into two threads: CPU utilization and token.

In the CPU utilization thread, the server constantly multicasts its CPU utilization to all other servers.

In the token thread, the server is receiving the token variable through the token\_rx channel. If the value received is true, the server will check the number of requests that they are currently handling. If the number of requests is equal to zero, then the server will send an infinite CPU utilization message to all other servers. This means that the server has now failed. It will then wait for 20 seconds, then revive again.

After it revives, it sets its token to false, then sends it to the next server so that the next server fails.

However, if the server is handling at least one request, then it will not fail and will pass the token to the next server because it is busy.

1. **Client-Listener Thread:**

This is a listening thread that is always listening to clients and willing to receive messages from them. Not only does it listen to clients’ requests, but it also decides whether it will accept the request or not.

If the server is the leader, then it will accept the request and will send an ACK message to the client who sent the request. The ACK message is sent through the sender thread. ACK messages from the listener thread are sent to the sender thread using a channel called send\_accept\_tx.

In order for a server to be the leader, the server has to have the lowest CPU utilization. In other words, it has to be the first row in the PST.

* If the first part of the message is LOGIN, then a client is asking the server to log them in.
* If the first part of the message is LOGOUT, then a client is asking the server to log them out.
* If the first part of the message is CHANGE\_ACCESS\_RIGHTS\_OFFLINE, then a client wants to change another’s client’s access rights. However, the other client seems to be offline, so the client is sending the request to the server to handle it. The server will store the request until the other client returns back online.
* There is a variable named waiting\_img that determines whether the image received is the first chunk of an image or not. This variable is used to tell if the server is receiving a new image or the rest of the image it is currently reconstructing.

1. **Worker Thread (Image Encryption):**

The thread is responsible for taking the image chunks from the blocking multi-producer single-consumer channel end and assembling them into a whole image that should be encrypted. The chunks are pumped into the channel from the thread that listens to clients for new requests. The image crate is made use of to hide one image behind another default cover image. After that is achieved, the encrypted image is fragmented into different chunks of pixels to be sent through a channel where the thread that sends messages to clients is the receiving end.

We use the steganography and base64 crates to embed and extract data within images. In the embedding process, the function embed\_image loads a hidden image and a destination image, encodes the hidden image data into base64, and embeds it within the destination image using alpha channel manipulation. The hidden image is converted into base64 bytes, which are then embedded into the destination image using the Encoder from the steganography crate. The resulting image, containing the embedded message, is saved to an output file specified by the user. Conversely, the extract\_image function retrieves the encoded image, utilizing the Decoder from steganography to extract the embedded data from the alpha channel. It then decodes the base64 message, retrieves the hidden bytes, cleans the buffer by removing specific byte patterns, and saves the decoded image to an output file.

1. **Worker Thread (Directory of Service):**

This thread is responsible for keeping track of the online users in the system. Once a LOGIN request has been received from a client, it is channeled from the listening-to-client thread to this worker thread. All servers will save the client’s username and ip address that will be used for future communication in a hash map. No two clients can have the same username to ensure uniqueness and avoid overwriting ip addresses in the hash map.

The mechanism that is responsible for updating the cached copy of the directory of service (DoS) on the clients’ side can be implemented in multiple ways. One of which is to send the DoS data to the client once logged in. If any user has gone offline, then peer users will not be able to detect their absence unless they try to send the offline user a message. If a user has tried messaging a client for three times and has not received any response from their peer before timing out, then the client will request DoS from the server. This method has its deficiencies when it comes to the experienced user response time. Since the designed GUI is listing the online users and choosing from the list, then it is not the best choice to have an offline user listed as online.

Therefore, it was decided upon that the simplest and yet an efficient way to update the cached directory of service copy on the clients’ side is to send only the updates to all clients. In case a new user logins into the system, the server will send the whole DoS to the new user, and will send its username and ip address to the rest of the clients and ask them to append it to their cached DoS. Similarly, if a client logs out or goes offline, the server will receive the LOGOUT request and will ask all other clients to mark them as offline. Consequently, the now-offline user will not appear in the online-users list for each of the other clients.

Two channels are constructed between this thread and the sending-to-client thread. This is because it depends on the type of message that should be sent to the client. It is either the whole DoS, or the new entry that should be added.

1. **Client-Sender Thread:**

The server uses this thread to send messages to other clients. This includes the ACK message, which is the message sent to the client who is asking for the encryption service from all servers; only the elected server is the one who sends the ACK message. It simulates the idea of telling the client that this server is responsible for handling the request. Servers also send the DoS to the new client and the new entry to the rest of the clients. Moreover, the encrypted images are sent in this thread, where each image is concatenated with an end marker. The end marker is used to indicate the completion of sending a whole image, the last fragment of the whole image has been sent.

**IV. Experiments and Results**

In order to test our cloud’s ability to simulate real world scenarios, we bombarded it with 1000 images from each one of our three clients simultaneously to see how it will perform with and without load balancing. We measure the turnaround time by seeing the time difference between from before the client sends requests to the cloud until it receives and opens the encrypted image. We use the std::time library in order to do this and then we save each entry into a txt file. To check for dropped messages, I created a counter which checks if the client received an acknowledgement message from the server, if not, the counter is incremented. Additionally, if the client isn’t able to open the encrypted image, the counter is incremented again because a chunk might have been dropped indicating an unsuccessful response.

1. **With load balancing:**

|  | Client 1 | Client 2 | Client 3 | Average |
| --- | --- | --- | --- | --- |
| Total Turnaround Time (sec) | 24,850.7057 | 25,081.1489 | 24,763.503 | 24,898 |
| Images Dropped | 12 | 19 | 14 | 15 |
| Images Encrypted Successfully | 988 | 981 | 986 | 985 |

1. **Without load balancing:**

|  | Client 1 | Client 2 | Client 3 | Average |
| --- | --- | --- | --- | --- |
| Total Turnaround Time (sec) | 31,018.7472 | 30,864.7732 | 30,599.6763 | 30,827.73 |
| Images Dropped | 19 | 23 | 18 | 20 |
| Images Encrypted Successfully | 981 | 977 | 982 | 980 |

Our implementation of the election algorithm and load balancing is a successful and effective one as it has led to an average decrease in turnaround time by 19.24% and an 0.5% average decrease in failure rate.

**V. Compilation Details**

1. **Libraries needed**

**Server:**

* **machine-info = "1.0.9" →** we used it to get the CPU utilization of the pcs for the server pst rows
* **termion = "2.0.3" →** a terminal control library for Rust. It's used for handling things like color, mouse events, and keyboard inputs in terminal applications, enabling the creation of complex text-based user interfaces.
* **chrono = "0.4.31" →** a date and time library for Rust. It provides comprehensive support for parsing, formatting, and manipulating dates and times. We used it for the image file naming to differentiate between the generated images in loops.
* **image = "0.24.7" →** a comprehensive library for image processing. It supports opening, manipulating, and saving various image formats.
* **bincode = "1.3.3" →** a crate for encoding and decoding using a binary serializer. It's often used for efficient data storage or network communication.

**Client:**

* **image = "0.24.7"**
* **serde = { version = "1.0.192", features = ["derive"] } →** a serialization/deserialization framework. It's highly flexible and supports various data formats. The `derive` feature enables automatic implementation of `serde` traits for custom structs.
* **bincode = "1.3.3”**
* **termion = "2.0.3"**
* **minifb = "0.25" →** used to display the low res and high res images in a pop up window
* **steganography = "\*" →** used for the encryption + decryption
* **base64 = "0.13.0" →** used for encoding and decoding data in Base64 format, commonly used in data encoding and transmission.

1. **Compilation Procedure** 
   1. The server run by typing the command cargo run 0 0. The first argument is an ID that is assigned to the server and the second argument is to simulate fault tolerance.
   2. The client is run by typing the command cargo run and following the steps of the menu such as logging in and choosing one of the program menu options.

**VI. Conclusion**

In conclusion, we learned so much about distributed systems in this project. We have integrated many concepts and designs while implementing this project such as load balancing, fault tolerance, encryption through steganography, and a server offering a discovery of service to clients. We have used a refactored method of Basu’s election algorithm that makes use of the servers’ CPU utilization to implement load balancing.This ensured effective distribution workload among the three servers. The image encryption through steganography and the discovery of service enables clients to communicate with each other. Users log in to the system through the discovery of service and are able to request images from other clients. Clients also have the option of editing the number of views and revoking access rights from other clients. To sum up, this project successfully combines theoretical design concepts with practical implementations, showcasing a cloud system that does load balancing, fault tolerance, while handling the requests of the clients including image sharing and discovery of service.

**VII. Team members contribution:**

Nada: election algorithm + client peer to peer communication + thread management

Salma: DoS + image fragmentation + thread management + access rights management

Amy: election algorithm + load balancing + fault tolerance + image viewership

Allaa: encryption + decryption + steganography + client peer to peer communication