A Cloud P2P Environment for Controlled Sharing of Images

*

1st Ali Ashraf Ali
Dept. of Computer Science & Engineering
The American University in Cairo
Cairo, Egypt

2nd Maya Hussein
Dept. of Computer Science & Engineering)
The American University in Cairo
Cairo, Egypt

3rd Omar El Sewedy
Dept. of Computer Science & Engineering
The American University in Cairo
Cairo, Egypt

4th Youssef Montasser
Dept. of Computer Science & Engineering
The American University in Cairo
Cairo, Egypt

I. Overview

The project at hand is a comprehensive cloud infrastructure that places a significant emphasis on transparency, load balancing, and fault tolerance. Leveraging distributed election algorithms, the system is designed to support key cloud qualities. The primary services offered by this cloud include image encryption through steganography and a discovery service.

II. Features

A. Image Encryption and Decryption through Steganography

1) User Registration: Users are provided with a registration process, allowing them to encrypt their images using steganography. Users are automatically registered into the cloud when they decide to request an image from another user. The steganographic technique ensures the secure embedding of information within the images. Users, upon registering with the cloud's discovery service, will have to encrypt their images before sharing them with their peers.

2) Gallery Viewing and Image Compression: Upon requesting to view images from a certain client, an entire gallery of compressed low-res images is sent to the user to choose the desired image.

3) Image Encryption and Decryption: A high-resolution copy of the desired image is then sent for encryption at the server's end and sent back to the client, where its access rights are encapsulated. The desired image is then sent to its destination client, where it is decrypted and viewed according to its number of views.

4) Viewing Permissions: Images are accompanied by specified viewing permissions, and users can set the number of permitted views. The cloud ensures that users can only see their images with restricted views. If the user exceeds that number, they have no access to view the decrypted image. Unless the user has newly updated rights, the encrypted image pops up whenever the user requests to view that image (assuming they have completed their number of views).

5) Ownership Control and Offline Operation Support: Owners retain control over their images, being able to update and modify viewing quotas. To support offline operations, the system follows an acknowledgment reply to detect if a client is offline. Whenever a client is sent an image, it has to send to the owner client an acknowledgment back saying that the image has been received. If an acknowledgment has not been received after a specified period, after a couple of attempts, the receiver user is considered offline. This feature aims to provide a robust mechanism for reconstructing permissions in case of failures.

B. Discovery Service

1) Peer Registration and Online Visibility: The discovery service serves as a fundamental component where users register to identify online peers. This registration allows users to understand which peers are currently online and how to establish direct P2P communication with them. The table of online users is consistently maintained within the cloud, promoting reliable and up-to-date peer discovery.

2) P2P Communication: The P2P architecture plays a crucial role in the project, enabling users to communicate directly without relying on the central cloud infrastructure when both parties are online. This design choice enhances the efficiency of communication and reduces reliance on cloud resources, contributing to a more resilient and scalable system.

3) Offline Support for Consistency: Offline support is integrated into the discovery service to handle scenarios where users or image owners are offline. After a client goes offline and the owner has been notified, the cloud is notified, and the offline host is removed from the directory of service. If an owner wants to modify the access rights to any of the offline clients, it sends the updated access rights to the

cloud, the cloud buffers it, and whenever the offline client registers itself once again as active, the cloud directly sends the buffered rights to the designated host. This proactive approach helps mitigate the impact of failures and provides a seamless experience for users.

III. UseCases

A. Case I: Request an Image

1) Client B requests to view Client A's gallery.
2) The server does an IP look-up on Client A's hostname.
3) If it exists, the server sends back the associated IP address. Otherwise, notify Client B that Client A is not registered within the cloud.
4) Client B requests from Client A its gallery and sends back to Client A the index of the desired image.
5) Client A sends the high-resolution image for encryption and sends it to Client B, encapsulated with its access rights.
6) Client B can only view the image any time within its limited number of views. Upon reaching the limit, only the encrypted image appears.

B. Case II: Update Access Permissions

1) Client B has requested to view an image with all the previous steps.
2) Client A has decided to update Client B's access rights.
3) If Client B is online, it automatically receives its newly updated rights

C. Case III: Offline Client

1) If client B goes offline after receiving an encrypted image.
2) Client A is notified and sends to the server to update its directory of service.

D. Update Permissions of an Offline Client

1) If client A wants to update the access rights of a specific image on an offline client.
2) Client A sends the required information along with the new viewing permissions to the cloud
3) The cloud then buffers the request in its servers and waits for the offline client to become online
4) Once the client registers itself onto the cloud, the cloud automatically sends the updated access rights, and the viewing permissions are updated
5) The current online client is able to view the image at any time with its new permissions.

It is good to note that a new leader is always elected based on CPU Usage to handle any request.

IV. Performance Evaluation

It is estimated that each request takes about 18 sec. However, this differs when there is multiple clients requesting at the same time. In the election, it was reduced a lot to finish all client requests. It balanced and handled the requests smoothly compared to when no election happened. The tables below will show 1000 requests from each client and how much time is needed to finish all the requests.

With Balance Loading:

Clients	**Time est to finish 1000 requests**	**Avg Time est by each request**
Client 1	6.03 hours	21.72 sec
Client 2	5.7 hours	20.52 sec
Client 3	5.95 hours	21.42 sec

Without Balance Loading:

Clients	**Time est to finish 1000 requests**	**Avg Time est by each request**
Client 1	8.7 hours	31.32 sec
Client 2	8.4 hours	30.24 sec
Client 3	8.64 hours	31.104 sec

Load balancing proves to be a game-changer in enhancing server performance, as evidenced by a comparative analysis of two scenarios. In the balanced loading setup, where requests are smartly distributed among servers, the system operates more efficiently and responsively. The key difference in processing times between the load-balanced and non-load-balanced scenarios boils down to how server resources are allocated.

When servers share the load more evenly, the system's response to requests is better coordinated, leading to shorter waiting times and overall improved performance. Distributing requests across multiple servers prevents bottlenecks and allows for parallel processing, resulting in a noticeable decrease in the average time it takes to handle each request, as shown in the data.

For example, in the load-balanced configuration, Client 1 estimates 6.03 hours to complete 1000 requests, with an average time of 21.72 seconds per request. Clients 2 and 3 also experience significant time savings and reduced average processing times. This efficiency is a result of smartly balancing the workload among servers.

On the other hand, the non-load-balanced scenario reveals inefficiencies due to uneven request distribution. Some servers may sit idle while others are swamped, leading to longer waiting times and delayed processing. The substantial differences in time estimates and average processing times for each client in the non-load-balanced setup highlight suboptimal resource use.

The performance metrics underscore the positive impact of load balancing on the cloud system. Beyond just speeding up processing times, load balancing ensures fair resource use, making it a vital element for efficiently handling multiple clients and requests. The data presented emphasizes the tangible advantages of load balancing in enhancing overall performance and responsiveness in the cloud infrastructure.

V. Technical Details

We started the project by constructing a simple client that sends to a server, listening on a fixed port. Then, we decided to construct a simple Directory of Service (DoS), on said server that maintains a list of client names and their IP addresses. Accordingly, we expanded our operational scale via adding another client to the picture. We had this client issue a

pseudo-request such that it gets registered in the server's DoS, followed by a lookup request by another client that receives the IP of the client and sends a message. Now, we felt this was the logical approach to our project seeing as it allowed us to establish a stable pipeline that would allow for future growth: be it adding clients, servers, or even features.

A. Architecture

In the beginning, before using multiple clients, each simple client request was handled perfectly by our cloud. However, this was short-lived. As soon as we had two clients sending requests concurrently, one of them would fail to be processed by the server. Hence, it became apparent that we need to utilize the services of multi-threading on the server-side. We had to take multiple decisions. Firstly, we pre-spawn a number of threads during server startup for increased time-efficiency – instead of spawning a thread when needed – creating a thread-pool. Hence, the time required for server-side processing of requests is reduced. This allowed our servers to process concurrent requests seeing as this architecture maximizes thread throughput and allows for concurrent access of resources whereby we protect these resources via mutex-locks to ensure exclusive access. As such we ensure task parallelization and our server processes to concurrent requests.

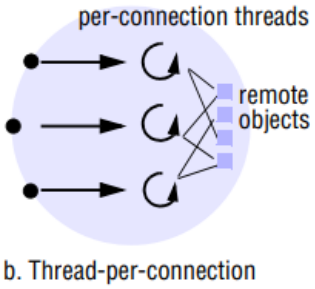

Fig. 1. We implement two listeners: one for client requests, and server-to-server messages. This is very similar to the figure above, as obtained from [7]

B. Distributed Election

Originally, we had proposed the implementation of the Raft election algorithm. However, in hindsight, we realized that this proposal was not feasible given the small-scale nature of our cloud, which consists of a minute set of three servers – not to mention its complex implementation requirements. As such, we decided to go ahead and implement a synchronous election algorithm, one in which each server would have to sleep for one second in order to ensure synchronization - of each server's load - between them. However, after adding another client to the picture we noticed that this technique produced incorrect and inconsistent election results. Due to its synchronous nature, some it would always happen that two servers believe that they are elected and, accordingly, the client receives two concurrent fragmented replies which would collide and invalidate the received packets. After heavy testing, we noticed a consistent pattern: load-exchange was flawed. Servers A and C consistently thought they were elected seeing as they each failed to send each other their respective load and, given that the default load for servers was 999, each server thought it won the election.

Capitalizing on this problem, we realized a solution that incorporates the separation of concerns design spirit. We allocated a dedicated thread, and accordingly a listener, on each server, whereby said threads periodically (every ten seconds) multicast the current load to other servers. Firstly, this sped up the process of election and, subsequently, response to the client by over 40% as we eliminated the one-second sleep that was inherent in the old election mechanism. Not only this, but also, having a dedicated thread and listener served as a deterministic method for ensuring that servers exchange their loads. In addition to that, this eliminated the unnecessary calls for election each time a client request is received. Seeing as it is highly likely that, during a short time span, the least loaded server would remain the least-loaded server and, thus, gets elected again.

C. Fault Tolerance and DoS Replication

In the event of failure, the failing server notifies its peers via sending its updated load as 999. Accordingly, during election, the peers automatically discard said failed server due to its unreasonably high load. Election still takes place perfectly fine. Now, when the server goes back online, it updates its peers with the actual load, signifying that it is back online. When that change happens - from 999 to the actual load - the servers recognize that the server in question was offline and, therefore, update it with the latest Directory of Service.

Now, we chose, from the very beginning, to always register a user that sends a request to a server apriori to election outcome. Seeing as requests are always multicasted to servers, it was imperative to cache these users in the DoS so as to evade server-to-server inquiries about clients; instead, this ensured that, should servers be online, their DoS must be identical.

D. Image Encryption: a Protocol

A critical pillar in our system was the provision of an image encryption feature or, perhaps more particularly, steganography. While we tried doing our manual, bare-metal steganography technique, which failed, we decided to just use a Rust library, unsurprisingly called Steganography. We had to test this library, and other failed ones too, on a separate program and then port it back to our server codebase. Additionally, we tested with differently sized images and implemented UDP message fragmentation. Since the UDP MTU spans 65 KB, we chose to divide our image into chunks of 30 KB so as to give room for our header data (maximum of five KB) and maintain a margin of safety of around 45%. Now, having a ready mechanism for image encryption, it was time to dwell

upon the process of image encryption. This is depicted in the figure below.

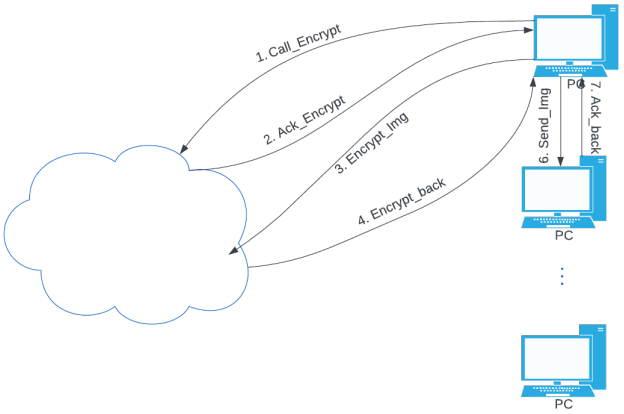

Fig. 2. This illustrates the image encryption flow (Self-captured)

1) Client issues an image encryption request
2) Servers conduct election. The chosen one replies back to the client.
3) Client starts sending the fragmented image to the designated server.
4) The server encrypts the image and sends it back to the client.

The beauty of this design lies in its simplicity and efficiency. It is highly network-efficient seeing as the packets need not propagate to all servers because ultimately only one will get elected. As such, the network does not suffer from over-flooding which can cause latency. Should the number of clients increase, the network would quickly get congested if they fail to adopt our implemented protocol. As such, this serves as a merit for the scalability of our proposed implementation.

E. Access Rights

Having sent a fellow client one's image with a default amount of views, it was necessary to provide the owner with the capacity to update (or revoke) one of his viewer's access rights: mainly number of views. The assumption is, if the viewer is online, then the owner can issue an IP lookup request, gets served with the corresponding IP address of the intended client (as fetched from the DoS) and sends the updated access rights directly to the viewer. The natural question that comes into one's mind is what happens in the case that the viewer is offline. Well, firstly, the owner has to detect that the viewer is offline, which is done via sending the request directly to the viewer and issuing a listen that times out after a given period of ten seconds.

Now, having detected the viewer's offline status, the client had two options: either delegate this responsibility to the itself or the server. In the first scenario, the server would just simply notify the owner of the viewer's now-online status and the owner would then contact the viewer and update its access rights. However, this seemed rather complicated and swarmed with edge cases. For instance, said proposal would fail in the event that the server tries to notify the owner (of the viewer's online status) but the owner goes offline and does not receive the notification. In other words, this would introduce a huge rabbit hole problem.

As such, we decided to move ahead with the second scenario by buffering or caching the owner's access rights update request on the server side, and have the server's notify the viewer on their startup.

VI. Crates and Versions

Client Side:

```
tokio={ version = "1", features = ["full"]}
lazy_static = "1.4.0"
sysinfo = "0.18.0"
base64 = "0.21.5"
image = "0.24.7"
steganography = "1.0.2"
show-image = "0.13.1"
```

Server Side:

```
tokio={ version = "1", features = ["full"]}
lazy_static = "1.4.0"
sysinfo = "0.29.10"
crossbeam-channel = "0.5.1"
steganography = "1.0.2"
base64 = "0.21.5"
image = "0.21.3"
chrono = "0.4.19"
```

VII. Compilation Procedure

To compile the project, follow the steps outlined below: Simply, run the following using Rust V. 1.66.1

```
cargo build
cargo run
```

Make sure to change all the directory image paths as they are the complete paths.

VIII. Conclusion

In conclusion, this project represents a significant advancement in cloud computing, meticulously addressing transparency, load balancing, fault tolerance, and user-centric design. The integration of distributed election algorithms ensures equitable workload distribution, optimizing resource utilization and responsiveness. The fault-tolerant server cluster exhibits resilience through simulated failures, promoting system reliability and consistency. Leveraging steganography for image encryption adds a layer of security, ensuring confidentiality and integrity. The user-centric approach empowers owners with control, offline operation support, and view updates. Looking ahead, potential enhancements include machine learning for predictive load balancing and exploring advanced encryption methods. The collaborative efforts of the development team, coupled with effective communication, have yielded a robust cloud infrastructure that not only meets academic criteria but also demonstrates practical application in addressing real-world challenges.

IX. Roles & Distribution

1) Ali Ashraf: Image Encryption/ Decryption, Message Fragmentation, Directory of Service, Access Rights Update, Client-to-Client Communication
2) Maya Hussein: Image Encryption/ Decryption, Message Fragmentation, Directory of Service, Access Rights Update, Client-to-Client Communication
3) Omar Elsewedy: Distributed Election, Fault-Tolerance, and Server-to-Server Communication, offline access rights updates
4) Youssef Montasser: Distributed Election, Fault-Tolerance, and Server-to-Server Communication, offline access rights updates

References

[1] Rust documentation, https://doc.rust-lang.org/beta/.

[2] "Crates.io: Rust package registry," crates.io: Rust Package Registry, https://crates.io/

[3] Garcia-Molina, "Elections in a Distributed Computing System," in IEEE Transactions on Computers, vol. C-31, no. 1, pp. 48-59, Jan. 1982, doi: 10.1109/TC.1982.1675885. https://ieeexplore.ieee.org/abstract/document/1675885

[4] G. Ngo Hoang, H. Nguyen Chan, K. Nguyen Van, T. Le thi Xuan, T. Nguyen Manh and V. Vu Thanh, "Performance improvement of Chord Distributed Hash Table under high churn rate," 2009 International Conference on Advanced Technologies for Communications, Hai Phong, Vietnam, 2009, pp. 191-196, doi: 10.1109/ATC.2009.5349535. https://ieeexplore.ieee.org/abstract/document/5349535

[5] Ongaro, D., & Ousterhout, J. (2014). In Search of an Understandable Consensus Algorithm. In 2014 USENIX Annual Technical Conference (USENIX ATC 14). USENIX Association. Retrieved from https://www.usenix.org/system/files/conference/atc14/atc14-paper-ongaro.pdf

[6] Towards Data Science. (n.d.). Raft Algorithm Explained. Retrieved from https://towardsdatascience.com/raft-algorithm-explained-a7c856529f40

[7] Coulouris, G., Dollimore, J., & Kindberg, T. (2011). Distributed Systems: Concepts and Design (5th ed.). Addison-Wesley.