Distributed Insta

Course project for cloud computing

Vishal Singh, Vishal Bindal, Chirag Mohapatra

1. Inspiration: Decentralising social media

There are privacy concerns and storage limitations with storing user data in data centres. For storage limitation, we can leverage storage of billions of devices connected to the internet. More replication to nearby nodes will mean reduced latency. This will allow companies to not worry about putting CDN at a lot of locations.

2. Architecture

In our distributed system each node acts as a storage node. Each user holds an encrypt key (private) and a decrypt key (shared with followers only) to ensure privacy of shared images.
We also have a centralised master for facilitating exchange of information.

Sample workflow

Creating a account

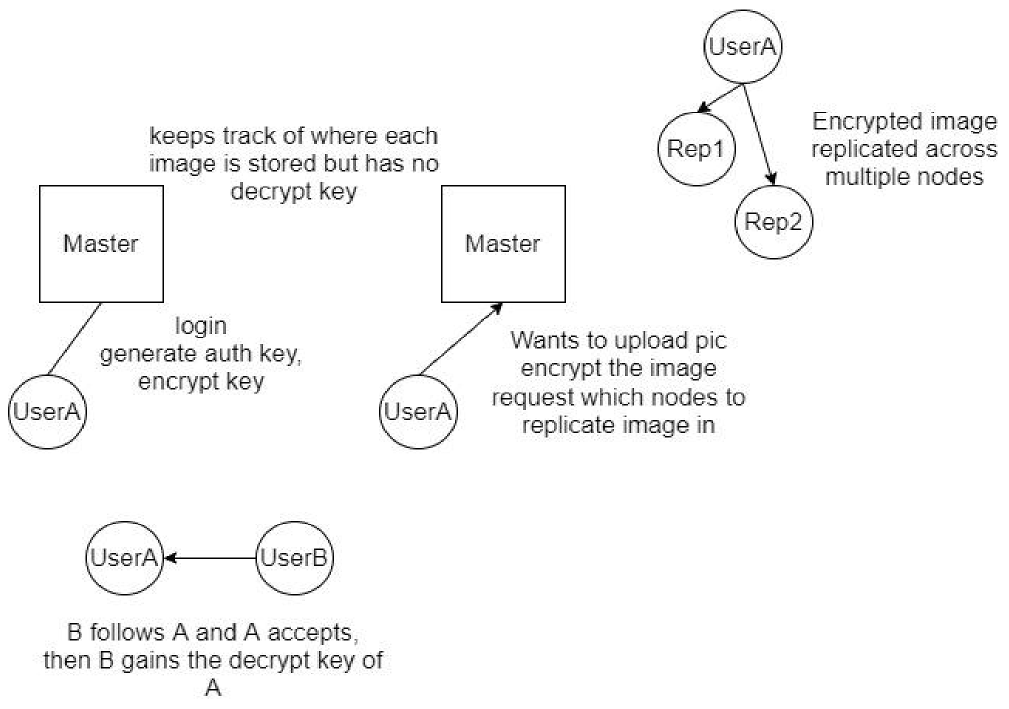
When a user (suppose user A) creates an account on the app, a new m_key is generated for all future communication with the master. Also pairs of keys (public/private) are generated and the encryption key is also stored on master.

Uploading a picture

When a user uploads a picture, it is encrypted using its encrypted key and then replicated across the network. Master keeps track of which nodes this image is present.

Following a account

When a user (say A) wants to follow another user (say B). Then A will send the following request to B. If B accepts the request then B will share its decrypt key to A. Now, if A wants to see B’s profile, it will fetch the list of B’s images and optimal nodes from master. For each image, optimal node allows for fetching that image with less latency from A. Now, A will fetch images and then decrypt them using A's decrypt key.


UserA
Rep1
Rep2
Encrypted image replicated across multiple nodes
keeps track of where each image is stored but has no decrypt key
Master
Master
login
generate auth key,
encrypt key
UserA
UserA
Wants to upload pic
encrypt the image
request which nodes to replicate image in
UserA
UserB
B follows A and A accepts, then B gains the decrypt key of A

A close look at user nodes
Allocate a specific storage at each node for encrypted image data
Assumption: one-one mapping between users and nodes
User nodes interact with master to fetch info regarding image locations, and then with each other to exchange image data
User nodes are expected to frequently go offline

Stored object	Description
m_key	a unique key for auth with master
key_encrypt, key_decrypt (self)	key pair for encrypting/decrypting user's personal profile images
list [key_decrypt] (following)	a list of decrypt keys for all users who are being followed by this user
image data files	Multiple encrypted image files

A close look at master node

- Stores the names of all users and unique mkeys for each user which are used to authorise requests
- Stores the location and timestamp of each user
- Performs periodic k means using the locations and assigns each user to a cluster, we have fixed the number of clusters

- For each image, stores the name of the nodes where the image has been replicated so that any user can request them and if they have the required decrypt key, then they can view the images

Stored object	Description
user -> mkey, hashed password	for auth with each user
user -> ip,location,timestamp	for determining where to upload/download photos
user -> images	list of images in user's profile
image -> nodes	nodes at which the image is stored
user -> followers	followers of each node
other metadata	for cluster info etc

Challenges compared to other distributed systems
- Node locations and inter-node latency is dynamic
- Nodes can go down or offline very frequently and simultaneously
- Available storage scales linearly with number of users, thus storage per user remains constrained
- User privacy is top priority

Design choice 1: Clustering nodes via k-means
The master periodically runs the k-means clustering algorithm on current node locations.
For replicating an image uploaded by a user, some nodes are chosen within the same cluster (followers likely to be nearby), and then some in several different clusters (reduce latency for far-away followers).
In addition, it is ensured that storage across nodes stays roughly uniform.
For downloading an image of a user, a node in the same cluster or a nearby cluster is accessed

Design choice 2: Fault tolerance
Replication of any user data across nodes ensures that data is not lost when a a node goes down (replication across multiple nodes in multiple clusters makes it highly unlikely that any data would be lost).
The application is not affected and all images stay available as long as at least one node containing an image stays up.

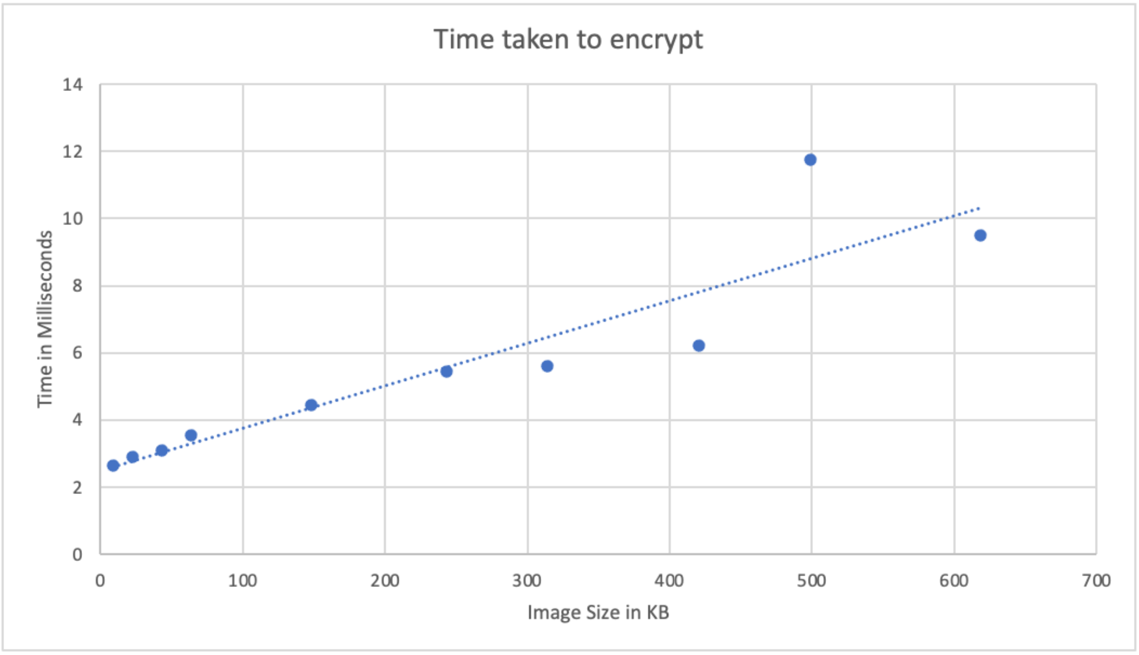
We have used **transactions** for all redis I/O operations at the master node, so that operations like clustering which are executed in background do not cause data inconsistency when executed at the same time as any other API call. Even if the master crashes, transactions ensure that operations done are not incomplete.

Design choice 3: User privacy
Every user has its own unique encrypt and decrypt key. The master does not have access to the decrypt key.
We have used the RSA hash algorithm to generate these keys and any data stored in the other users is encrypted data which cannot be accessed without the associated decrypt keys.
The only nodes which have the decrypt key are the user all his followers.

Encryption
We used RSA to encrypt and decrypt data. But the issue is that RSA can only encrypt a limited amount of input. How much that is depends on the key size of RSA (1024-bit in your case) and the used padding.
So, how did we deal with it ? Well, AES can encrypt data of any length.
So, we used hybrid encryption.

1. Generate a random byte string 16, 24 or 32 bytes to be used as the AES key,
2. Encrypt the actual data with AES using the previously generated key and
3. Encrypt the AES key with RSA.

Decryption is simply decrypting the AES key with rsa public key and then using that key to decrypt the image.



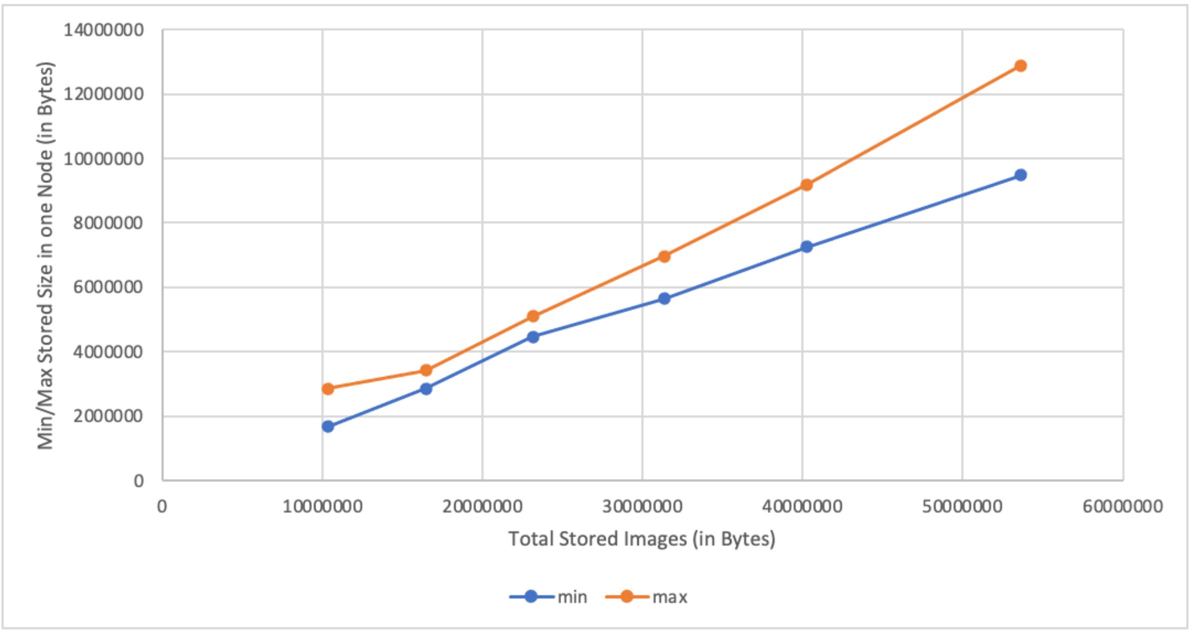
Link to presentation:
https://docs.google.com/presentation/d/1LSh5CP3oIKV-3zTIWxOwlKpE9tI9Z6DH-UE46ILDtP0/edit?usp=sharing

Link to github repo:
https://github.com/vishalBindal/Distributed-instagram

How to set up
The repo readme contains instructions for installing requirements, and starting the master and user node servers.

Experiments

Scalability Experiment



If we do randomised uploads (uploads from almost all nodes) we can see that the differences between data of nodes is not that different showing that we scale linearly storage wise.
Even if we upload unsymmetrically (more from one node) we can see consistent storage among diff nodes.

Fault Tolerant Experiments
As demonstrated in the demo, we first initialised multiple user nodes, and uploaded multiple images by various users. We then tried crashing multiple nodes.
For our case, on crashing the first 2 nodes, photos of a given node are still visible since they are replicated across the network. On crashing

yet another node, the data is now not visible since all replications of the image were only among these crashed nodes.

User Privacy
As demonstrated in the demo, the photos of a user A are only visible to user B only if B follows A, since in this case only would B hold the decrypt key of A. This shows the correctness of the encryption scheme.

Clustering and storage uniformity
As demonstrated in the demo, adding more users to the network updates the clusters, and uploading images replicates them across multiple nodes across clusters, and the storage across nodes remains roughly uniform with time.
To replicate this, simulate creation of users and uploading photos in the network, and view the storage and clustering stats on the master index url.