P2P Image Sharing Documentation

**Project Description:**

Using our GUI based on QT, it shows the peer interfaces which is consisted of a server side and

a client side, in which in the server, we store the online users and the availability for this user to

run, and a user or a peer can login and register as a new user. Furthermore, the user would

register using a unique username and password and ip address, which are automatically stored to

the database system. After logging in, the server verifies the statues of the users as online and

offline users, furthermore, from where is this client logging in from the ip address of the

machine.

Client::Client(string \_username, string \_IPAddress, int \_port) {

username = \_username; IPAddress = \_IPAddress; port = \_port; timeout = 0; } **Flow:**

In order to communicate, each user can view the number of online users in order to deal

communicate and send messages and images to each other. Each client sends its own status

continuously every few seconds to the server indicating that it is online or offline. If the peer did

not update its status , the server considers it offline and removes it from the table.

Each peer in the system can now upload their own images to their client page. The peer chooses

which user to send the picture to and specifies how many views associated with it. The user can

modify the number of views after sending the picture and can even delete the whole request.

Peers can also request to get an image from any other client by sending request to clients using

their user names which can be accepted or denied. A list of the pictures stored at the client is sent

to the client where he can specify which image he/she wants.

**Steganography:**

Images have their receiver and view count embedded using steganography and are then extracted

at the other end after it is received. After every view, the view count is updated in the embedded

text file and can be even see at the QT until the views are 0 and the user cannot more see

anything We used the steghide library for Steganography; basically it hides the number of views

allowed for this image along with the owner ID.This library takes the number of views granted

from the owner to the client through an input field in the UI and increment it with the owner id

inside the image and when the image is received from the server the opposite operation is done

to extract the data from the image. All these images attributes are passed as json objects using the

serialization method. **Limitations while using steghide:** limitations in using images with png

extensions.

**Threads:**

Thread-Per-Connection Architecture

A new worker thread is created for each client and handles all

its requests, it is destroyed when

the client closes the connection. This has lower overhead,

however some clients might be

delayed as their thread is working while other threads are

free.

**Functions:**

Function **updateStatusOnServerTimer** is in charge of keeping the server consistent with its

peers in regard to who is online and who is not.

Function **updateRemoteImagesTimer** manages timer fault tolerance on images that are within

the system and sent from the current peer.

This is the message class that handles the message types **( Reply or Request), the RPCID** that

was attached to each process per clients. It handled the status of each message sent along with its

username . it also handled the registration and the requests sent from clients.

**Classes:**

int Message::RPCIdCounter = 0; // **A counter to each process per client** UDPSocket Message::mySocket = UDPSocket();

Message::Message(MessageType \_messageType, string \_message, int \_RPCId) {

messageType = \_messageType; message = \_message; RPCId = \_RPCId; } void Message::sendToClient(string ip, int port) **// Send to client function that sends the message to the specified username** {

mySocket.sendMessage(message, ip, port); } void Message::processReceivedMessage(string message, string ip, int port) **//Function that process the requests/messages sent whether it is new registration or logging in or updating status of the client**

**Robot:**

Created a robot in which it’s function to roam around every all online users and collect data

which is the list of clients images and which images each client has privilege to view them and

the remaining view quota for each image for each user.

**Database:**

we created database management with updating the tables with the received data from the QT.

This is the Database class that handled the new registration , and logging in of existing users by

comparing them to a data set of clients who we input into the database

bool DatabaseManager::updateUserIP(string username, string IPAddress) {

QSqlQuery query; query.prepare("UPDATE User SET ip\_address = (:ip) WHERE username = (:username)"); query.bindValue(":ip", QString::fromStdString(IPAddress)); query.bindValue(":username", QString::fromStdString(username));

**GUI:**

Using QT, we developed a login/register page appears at first when the user uses the application,

the user is required to register in case of new user, or login the case of an existing account.

Furthermore, the user is taken to a page with different tabs and from which they chose what to

do, either to request views, or the pending request and my images tabs, each image has its name

and it unique id, each user can only see their own images that they uploaded, or images that have

their usernames hidden into them. Also each time a user views an image, the number of views

needs to be updated inside the image and the user is denied view to the image when the number

of views gets consumed. The owner of the image has the right to update the image data by

adding or removing users from the image or changing their viewing quota. Moreover, users can

retrieve a list of their own images and a list of all the images they can view with the remaining

view quota for each image.

**Updated Features from Exercise #2:**

**Fault Tolerance:**

We tested our fault-tolerance capabilities of our system. As we added a timeout in our client program.

This is used in case a message was dropped, or the server has crashed. The client should report on its behavior in these circumstances. Extend Status to allowfor the time out. For example, we tested the program time-out by running the client when the server is

not running.

Testing Fault tolerance when a client issues a request for a client list while centralized server is down

aborted the operation and discarded any received image fragments.

Fault tolerance is supported to protect against packets that are sent out of order and in case of a connection drop using acknowledge messages with unique IDs for each frame sent.

Timeout added if there is no response from the server for a set amount of time or if a client paused sending to the other peer.

Fault tolerance was further verified by having the server not run and the result was the peer program failing to login as it cannot connect to the server for verification.

The class that was made to handle time outs and fault tolerance is timermanager.h and timeoutmanager.h

Function updateStatusOnServerTimer is in charge of keeping the server consistent with its peers in regard to who is online and who is not.

Function updateRemoteImagesTimer manages timer fault tolerance on images that are within the system and sent from the current peer.

**Issues from Exercise #2:**

1. Demo was done on just two machines, **Solution**: now we installed qt on three machines and we can have a demo on the three machines, as we would have 2 peers and 1 central server.
2. The image was send every time a peer requests a view from another peer. **Solution**: now the image is sent only once when first requested and then whenever a new a view is requested. The view is decremented by one, through the attributes json object send by the owner, without actually sending the image every time.
3. It was missing to have a folder with the owners’ image and name. When images are sent from one peer to another a folder will be created on the receivers’ machine with the owners’ name. The folder will have an encrypted version of the image but not the actual one.