Report for phases 2 & 3(Group 1)

The report for phase 2 has been edited and explained better in more details.

For both the server and peer projects, delete the “.pro.user” files.

To run server:

1-Install sqlite:

sudo apt-get update

sudo apt-get install sqlite3 libsqlite3-dev

2-Navigate to project folder in terminal

3-run executable with the server’s ip address and port number as command arguments.

Example:

./Exercise1 192.168.200.1 3000

To run Peer:

Make sure you have the necessary qt tools installed.

Install steghide:

sudo apt-get install steghide

2-Navigate to project folder in terminal

3-run executable with the server’s ip address and port number as command arguments.

Example:

./Exercise2 192.168.200.1 3000

Note that the make files and executables are still named Exercise 1 & 2 but they are actually related to Exercise 3.

**We implemented the peer to peer method.**

**Phase 2**

Database Class:

For the server’s database, we used the SQLite library, creating 3 tables: user, image and allowedviewer. The user table consists of the user’s username, password and token. The image table consists of the image id, image name, ownerusername, owner’s ip address and owner’s port. The allowedviewer table consists of the id, image id, image name, username and viewcount. On initialization of the server, those tables are created in a database file unless the file already exists. We implement many functions inside the database that include select, insert, update and delete queries depending on which records we wish to manipulate. SQlite is thread safe and every time the server runs, it makes sure that we are not overwriting tables using the create method.

Server Class: Check server.cpp in the server project

The Server spawns n threads on initialization that sit on recvfrom in order to handle multiple requests concurrently. When the server receives a request, it checks for the operation id, performs the operation, updates the database and replies with the same rpc id.

Server functions (operations): Check server.cpp

Message \* getRequest();

Continously receives requests from peers

Message \*setviewcount (char \*info)

Changes the number of views for a list of users

Message \*makeimageavailable (Message \*m,string ip);

Uploads image info from a peer

Message \*sendimage (Message \*m);

When a peer sends an image to another

Message \*getlistofimages (Message \*m);

When a user wants to get a list of images which he can view

Message \*createaccount (Message \*m);

When the user wants to create an account

Message \*authenticate (Message \*m,string ip);

When the user logs in

Message\* getlistofusers(Message \*m);

when the user logs in, his gui is updated with the list of users in the system.

Message\* getlistofownedimages(Message \*m);

When a user wants to retrieve the list of images which he owns

Message\* deletelistofimages(Message \*m);

When a user wants to delete a list of images

Message\* getlistofallowedviewers (Message \*m);

When a user wants to retrieve the users that are allowed to see one of his images

Message\* addviewer (Message \*m);

When the user wants to add a viewer to an image

Message\* removeviewer (Message \*m);

When a user wants to remove a viewer from an image

Message\* editviewcount (Message \*m);

When the user wants to edit another user’s viewcount for a certain image

Message \* authenticateuser (Message \*m);

When a peer wants to check if a user is allowed to see a certain image

Message \* decrementview(Message \*m);

When a peer requests that a user’s viewcount be decremented for a certain image

void sendReply (Message \* \_message,struct sockaddr\_in pAddr);

When the server sends a reply to a peer.

Client Class: Check client.cpp

The class is still named “client” but it is actually a peer. When a user logs in or creates an account, his ip address and port is updated in the server database for each image entry (if there are any). The user is also provided a token which is a random integer that he must provide to the server for each request he presents to the server. When the client receives a request from the GUI, it forms a message with the operation id, rpc id and message data and sends it to the server for processing and awaits a reply. If the client wants to view an image, he sends a request to the server, the server then replies with the port, ip and image path of the target peer who has the image.

Functions & operations:

Message \* editallowedviewers (Image I,string usernames);

Message \*createaccount ();

Message \*authenticate ();

Message\* getlistofimages (string username);

Message \* makeimageavailable (Image I,string allowedviewers,string viewcounts);

Message\* getlistofusers();

Message\* getlistofownedimages ();

Message\* deletelistofimages (string imagenames) ;

Message\* getlistofallowedviewers (string imagename);

Message\* addviewer (string imagename,string viewer,int viewcount);

Message\* removeviewer (string imagename,string viewer);

Message\* editviewcount (string imagename,string viewer,int newcount);

Those correspond the server operations

Server Class in Peer: check server.cpp in the client project

The peer also contains a server that awaits image requests from other peers, the peer then checks with the server if the sender is allowed to access the image and if so, sends the image back to the receiver peer.

Functions & operations:

Message\* authenticateuser (string user,string image,string ownerusername);

Message\* decrementview (string username,string image,string ownerusername);

Message \* getRequest();

Message \* doOperation();

Message \* sendImage (Image I,struct sockaddr\_in pAddr,string username,string image,string ownerusername);

void sendReply (Message \* \_message,struct sockaddr\_in pAddr);

Image Class: Check Image.cpp

The image class contains information about an image and contains two functions that marshal and un-marshal the image bytes.

Functions:

Image(string ownerusername,Qstring name,Qstring path);

Image (){}

int getimageID();

int getsize();

string getownerusername();

Qstring getname();

Qstring getpath();

void setownerusername(string);

void setname(Qstring);

void setpath(Qstring);

bool marshal (char\* marshalledinfo,char\* marshalledimage);

Qimage unmarshal (char \* marshalledinfo,char\* marshalledimage);

~Image ();

Message Class: Check message.cpp

The message class contains the rpc id, operation id, message and message size. It also contains two functions that marshal and un-marshal a given message.

Functions:

Message(int operation,void \* p\_message,size\_t p\_message\_size,int p\_rpc\_id);

Message(char \* arshaled\_base64);

void copy (Message\*);

Message(){}

int entiremessagesize;

void marshal(char \*);

void unmarshal(char \* ,int type);

int getOperation ();

int getRPCId();

void \* getMessage();

size\_t getMessageSize();

MessageType getMessageType();

void setOperation (int \_operation);

void setRPCId(int);

void setMessage (void \* message,size\_t message\_size);

void setMessageType (MessageType message\_type);

MessageHandler Class: Check messagehandler.cpp

The MessageHandler Class contains two important functions that split the message into fragments and gather the messages back together.

Messagehandler(){fragmentsize=2048;numberoffragments=0;}

Messagehandler(Message \* message);

void splitmessage (vector <Message> &);

string gathermessages (vector <Message>);

void setmessage (Message \*message);

void setfragmentsize (int fragmentsize);

Message \*getmessage ();

int getfragmentsize ();

int getnumberoffragments ();

~Messagehandler();

Packet loss and re ordering: Refer to client.cpp and server.cpp in the client program

We implemented the stop and wait protocol where a message fragment is sent and no other fragment is sent unless an acknowledgement is returned. If no acknowledgement is returned within a timeout of 2 seconds, the same fragment is sent again and so on until an acknowledgement is returned. This ensures that packets are delivered in the correct sequence and that no packets are dropped.

GUI: check mainwindow.cpp and appwindow.cpp

The user is presented with a login/create account window and is authenticated with the server’s database. If successful, a new window is show with 4 tabs: View image, upload image, remove image and edit viewer data. A user may perform any of those operations. When a user requests an image that is not his, the view count is updated inside the server database and the GUI and once the count reaches zero, the user can no longer view the image. A user may change the view data for any of his images where he can remove a user, add a user or change his view count. When a user uploads an image, he may select a few users that can view it and add a maximum view count for each user. When a user clicks the view image button, a pop up window is shown with the image.

Steganography: check Image.cpp, functions marshal and unmarshal

We used a tool called steghide for steganography. The sender peer embeds the real image into a fake image, adds a random integer password and sends the fake image with the password to the receiver. The receiver then extracts the fake image with the password, gets the real image and displays it. We fixed a mistake from the demo of phase 2 and now when a user’s view count reaches zero, the fake image is shown.

Sender’s end steganography:

string call=”steghide embed –cf “+fakepath+” –ef “+p+” –p “+password+” –f “;

system (call.c\_str());

p is the real path, the password is a generated random number that is sent to the peer.

Receiver’s end “unstegnafying”

string call=”steghide extract –sf “+fakepath.toStdString()+” –p “+\_password+” –f “;

system(call.c\_str());

Peer extracts the image with the password received from the sender

**Phase 3**

Packet loss and re ordering: Refer to client.cpp and server.cpp in the client program

We implemented the stop and wait protocol where a message fragment is sent and no other fragment is sent unless an acknowledgement is returned. If no acknowledgement is returned within a timeout of 2 seconds, the same fragment is sent again and so on until an acknowledgement is returned. This ensures that packets are delivered in the correct sequence and that no packets are dropped.

For phase 3, we added a timeout of 2 seconds for each message request and acknowledgement. A request is sent 10 times and if there is no reply, a label appears on the GUI informing the user that there was no response. Refer to function getImage (string name,string ownerusername,string ip,int port,string path,int &b) in getimage.cpp and function sendImage (Image I,struct sockaddr\_in pAddr,string username,string image,string ownerusername) in server.cpp of the peer project. Also, the sender peer does not inform the server to decrement the receiver’s view count unless he is sure that the entire image was sent. Therefore, the receiver will never get this view count decremented if he does not receive the entire image. Refer to sendImage (Image I,struct sockaddr\_in pAddr,string username,string image,string ownerusername) in server.cpp of the peer project. If a peer or server disconnects during an rpc, the request/reply message is sent 10 times. If the disconnected machine reconnects during this period, all operations continue normally otherwise the user is notified in the GUI that there was no response. We also added a history to the server where the server makes sure that no request is processed twice by checking for each peer the rpc id of the message and making sure that it does not match the last rpc id for the last request; A map is created map<string,int>lastrpc this can be checked in the server.cpp file in the server, an example exists in the function decrementview. To test for the fault tolerance, we tried sending request from peers to the server when the server is offline and after 3 attempts, the peer displays a message stating that it did not receive a response. Also, when one peer requests an image from another peer and that other peer is offline, the peer sends the request for 10 times and if there is no response the peer displays a message stating that there was no response. Also, if a peer disconnects during the period where an image is being sent, the number of views is not decremented for the receiver peer and a message appears. We applied a status to the message to determine whether the request completed successfully or not. Also, if a peer requests an image from a disconnected peer and the disconnected peer reconnects, the reconnected peer will proceed normally and send the image back to that peer.

Thank You.