CSCE 441101 - Fundamentals of Distributed Systems

CSCE 441101 Project 1: Group 2

Final Project Report

Fall 2016

Submitted to: Dr. Amr El Kadi

Submitted on: December 12th 2016

Submitted by:

Hager Rady, 900123209

Hussam El Araby, 900131261

Mohamed Ismail, 900123361

Ola Salem, 900123200

Client and Server Design and RPC-based Middleware

Client Design

General Activities of Client:

- Register on the server
- Create an Image
- Upload an image to the server
- Check whether a certain client exists on the system
- Get a list of all images accessible by the client
- View Image
- Get a list of users that can access a certain image and their number of views
- Update the access list of a certain image
- Delete an owned image
- View a list of users whether online or offline

Server Design

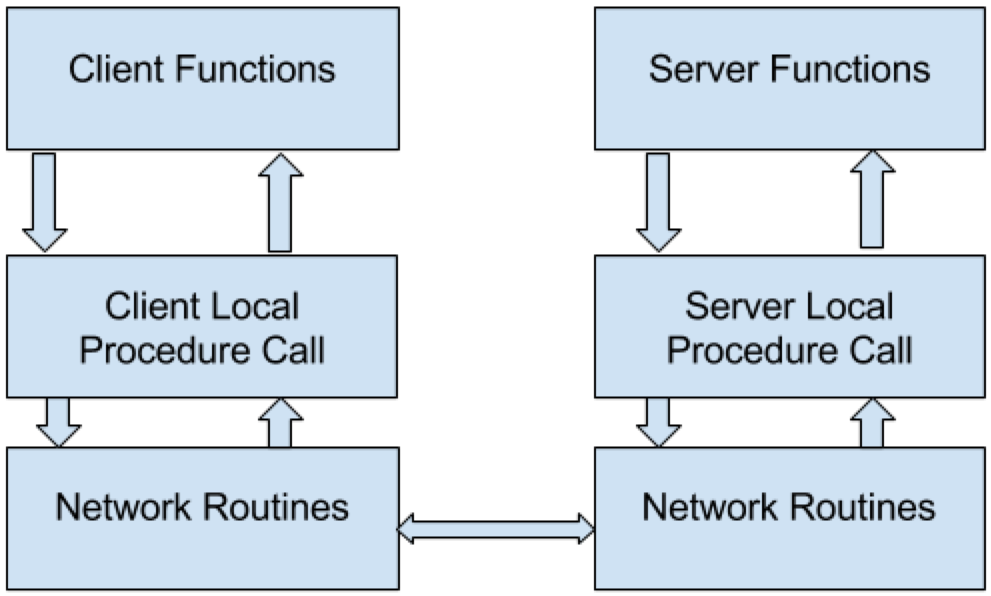
General Activities of Server:

- Register Users for the first time
- Allow users to log in and log out
- Allow registered clients to uploaded images
- Allow owners to delete their registred images
- Allow owners to update access rights of their registred images
- Allow users to view and download images they have access to
- Allow users to view a list of users registered on the server
- Allow users to view the status of other users, whether online or offline

The server is being designed to have a receiving thread running in a *while(true)* loop. The receiving thread sends the received message to a serving function that starts a new thread to serve the matching request operation obtained from the message content.

RPC-based middleware

The RPC components make it easy for clients to call a procedure located in a remote server program. The client and server each have their own address spaces; that is, each has its own memory resource allocated to data used by the procedure. Applications on the client only need to know one transport address: that of the name server process. They don't need to know the port number of each server-side process that they want to contact. The following figure shows the RPC process:



Both client and server make use of the messageType field of the Message class after unmarshaling to test that Requests and Replies are not confused.

The client generates a new RPC Id for each call; and the server copies the requestId from the Request message to the Reply message where the client tests that the replies correspond to the requests.

<u>Message Class</u>

Message ID
Message Type: Request 0 , Reply 1
Message Nature: Idempotent 0, NonIdempotent 1
Message Size
Message Content

<u>Operations</u>

CheckUserOperation 0
- Client sends: username (string)
- Server replies: -2 unexpected problem, 0 username does not exist, 1 username exists

RegisterOperation 1
- Client sends: username (string) and password (string),
- Server replies: -2 unexpected problem, -1 auth problem , 0 success, 1 username exists

UploadOperation 2
- Client sends: username (string) and password (string), access list , QImage
- Server replies: -2 unexpected problem, -1 auth problem , postive interger imageID

DownloadOperation 3
- Client sends: username (string) and password (string), imageid (int) ,
- Server replies: QImage or false image if auth failed or image not found or unexpected error

both embedded with either true list or empty list

GetRevisionCountOperation 4

- Client sends: username (string) and password (string), imageid (int) ,
- Server replies: -2 unexpected problem, -1 auth problem, int revisionCount for that user for that image

UpdateListOperation 5

- Client sends: username (string) and password (string), imageid (int), updated access list
- Server replies: -2 unexpected problem, -1 auth problem, 0 success

DeleteOperation 6

- Client sends: username (string) and password (string), imageid (int),
- Server replies: -2 unexpected problem, -1 auth problem, 0 success

GetImageList 7

- Client sends: username (string) and password (string),
- Server replies: map of images accessible by this user in form of map of int (image id) and pair of int (max views for that user) and int revision count (for that user for that image, for that image) empty map if auth error or unexpected error

GetUsersList 8

- Client sends: username (string) and password (string),
- Server replies: vector <string> of all users

GetUserListOnline 9

- Client sends: username (string) and password (string),
- Server replies: vector <string> of all users online

LoginOperation 10

- Client sends: username (string) and password (string),
- Server replies: -2 unexpected problem, -1 auth problem , 0 success

LogoutOperation 11

- Client sends: username (string) and password (string),
- Server replies: -2 unexpected problem, 0 success

Communication Module

Introduction

The communication module allows the client and server application to communicate over the Internet. The communication module is implemented as the communication class. The client and server each have their own communication object.

Functions of the communication object:

1. Send and receive messages.
2. Send and receive packets.
3. Reordering incoming packets and discarding incoming duplicate or old packets.
4. Implementing a timeout of a couple of seconds on receive for the client and retransmitting the last request message in case packets are lost for a fixed number of times.

Main Data Structures of the communication object:

- *bool isClient;*
    - A boolean value to indicate whether the object that hold this communication object a client or server.

- *bool terminate;*
    - A boolean value to indicate whether stop communication was called or not. Indicating to the sending and receiving threads to return.

- *bool clientReceiveFail;*
    - A boolean value to indicate whether the client failed to receive any message after all the retransmissions were exhausted.

- *bool waitingOnReply;*
    - A boolean value to indicate whether the client is waiting on a reply or not.

- *ofstream logFile;*
    - Stream object for the log file.

- *tuple<int,int,message> lastMessageSent;*
    - A tuple to hold the IP,port last sent to and the message last sent.

- *std::map<std::pair<int,int>,std::deque<packet> > incomingPackets;*
    - A map to that holds a queue of incoming packets for each IP,port.

- *std::queue<std::tuple<int,int,packet> > outgoingPackets;*
    - A queue to hold packets to send out.

- *std::queue<std::tuple<int,int,message> > incomingMessages;*
    - A queue to hold messages received.

- *int lastMessageIDReceived;*
    - Indicates for the client the last message ID received.

- *std::queue<tuple<int,int,int>> repliedQueue;*

    - Queue for the server to hold information about clients that have been replied to. It holds tuples of IP,port, and message ID.

- *std::map<tuple<int,int,int>,message> sentReplies;*
    - Map for the server to hold the reply messages sent to the clients. Acts as history.

- *UDPSocket socket;*
    - The socket object for the communication class.

- *thread sendThread, receiveThread;*
    - Threads to send and receive packets.

- Mutex locks to synchronize access to the above structures

Send and receive messages/packets (Functions 1 and 2)

<u>Sending a message</u>

Sending a message happens on two stages:
1. Dividing the message into packets.
2. Sending the packets one at a time.

Step 1:
The *sendMessage* function
bool sendMessage(message messageToSend, int IP,int port);

Takes a message to send, and IP&Port to send to.
The function divides the messsage into packets using *divideIntoPackets* call of the message class, and pushed that into *outgoingPackets* queue.
And in case of the server, it puts the reply into *repliedQueue* and *sentReplies.* These structures should only store the last 50 replies. *sendMessage* replaces the oldest entry (when 50 replies are stored).

Step 2:
The *sendThreadFunction* function
void sendThreadFunction()

This function is run in a seperate thread. It is an infinitely running loop. In each loop it takes a packet from *outgoingPackets* , marshals and serializes it into a string and finally encodes it in Base64. The result is a string ready to be send to a certain IP,port. The function *sendFromSocket* takes its C style string and calls the function *writeToSocket* from the socket object to actually write the data to the socket.

<u>Receiving a message</u>

Receiving a message happens on two stages:
1. Collecting packets into queues in the map *incomingPackets*
2. On each new packet received, extracting a complete message (if formed) and pushing it into the queue *incomingMessages*
3. *receiveMessage* call to pop the front of the *incomingMessages* queue.

Step 1:
The *receiveThreadFunction* function

void receiveThreadFunction()

This function is run in a separate thread. It is an infinitely running loop. In each loop it should try to extract read a packet from the socket. If successful it checks the string received, tries to deserialize and unmarshal it. If the packet is not corrupt, it checks the value of the packet ID and message ID. It then decides to which queue it belongs. Queue are in a map with the key the pair IP,port. The decisions that will occur here will include: discarding the packet if the messageID on the received packet is older than the message ID the queue is collecting. This indicates that the client gave up on the previous message and expects reply to the newer message. Also when a newer packet arrives the older queue is flushed.
This is exemplified below

```
        if(received_packet.getMessageID() ==
incomingPackets[make_pair(IP,port)].front().getMessageID())
        { //Push received packet
else if(received_packet.getMessageID() > incomingPackets[make_pair(IP,port)].front().getMessageID())
 { //Clears old packet queue and pushes new packet}
else
{ //Discards old packet}
```

Step 2:
On each new packet received, extracting a complete message (if formed) and pushing it into the queue *incomingMessages*
Here we check when if the received packet completes the message of that queue we pushed it in.

```
if (received_packet.getNumOfPackets()==int(incomingPackets[make_pair(IP,port)].size()))
```

If so, it calls the constructor of *message* with the queue of packets passed, the formed message is pushed to *incomingMessage* queue.

Step 3:
receiveMessage call to pop the front of the *incomingMessages* queue and return it to the calling function

Reordering incoming packets and discarding incoming duplicate or old packets (Function 3)

Discussed in the experiments section.

Fault Tolerance Measures (Function 4)

Timeouts and retransmission

The receive for the server is blocking, the receive for the client is also blocking but we have used a timeout. The timeout is set at the initialization of the client using (where TIMEOUT_SEC is in seconds, and TIMEOUT_USEC is in microseconds)

```
  struct timeval tv;
tv.tv_sec = TIMEOUT_SEC;
tv.tv_usec = TIMEOUT_USEC;
setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, (char *)&tv,sizeof(struct timeval));
```

Client and server reporting on their status

First, there is an extensive log for both the communication object and the socket object. If any send, receive, failure, or retransmission happens it will be shown there. The use of the log helped us debug many of our bugs while coding the project. It is a very helpful tool.

The log looks something like this:

For communication module:

```
[7662348334]    Communication log ready.
[7662530039]    Server communication initialized.
[8195447142]    Packet read from socket: Packet ID:1, 37 bytes, Message ID:1, from:
192.168.43.68:41378
[8195473733]    Incoming message now on queue: ID:1, 37 bytes, from: 192.168.43.68:41378
[8195514212]    REQUEST        Message received: ID:1 OpID:registerUser, 37 bytes, from:
192.168.43.68:41378
[8197420193]    REPLY Message put on send queue: ID:1 OpID:registerUser, 1 bytes, to:
192.168.43.68:41378
[8198618979]    Packet written to socket: Packet ID:1, 1 bytes, Message ID:1, to: 192.168.43.68:41378
[2115299749]    Packet read from socket: Packet ID:1, 37 bytes, Message ID:1, from:
192.168.43.68:41378
[2115330047]    Incoming message now on queue: ID:1, 37 bytes, from: 192.168.43.68:41378
[2115383630]    REQUEST        Message received has already been replied to, REPLY resent: ID:1
OpID:registerUser, 37 bytes, to: 192.168.43.68:41378
```

For socket object:

```
[7662444064]    Server socket log ready
[7662510857]    Server socket bound to 192.168.43.209:7891
[7662522893]    Server socket initialized.
[8195365106]    Read: 6268 bytes, from: 192.168.43.68:41378
[8197512299]    Wrote: 6268 bytes, to: 192.168.43.68:41378
[2115214356]    Read: 6268 bytes, from: 192.168.43.68:41378
[2115488245]    Wrote: 6268 bytes, to: 192.168.43.68:41378
[2156057440]    Read: 6268 bytes, from: 192.168.43.68:41378
[2159813364]    Wrote: 6268 bytes, to: 192.168.43.68:41378
[2170290425]    Read: 6268 bytes, from: 192.168.43.68:41378
```

Regarding the client reporting to the user its status, we aimed for a more transparent approach. When the client timeouts on an operation. It retransmits, as discussed above. This is not reported as we do not want to clutter the view for the user, though it is still available in the logs, typically for more technical people. However if all the retransmissions are exhausted and the client permanently fails on the the client

Experiments for Exercise 3

List of experiments done

1. The network fails for a short period of time during normal operation.
2. The network fails permanently.

Objectives

We wanted to test how tolerant our system are to failures. This means that under failure the system should try to operate normally and transparently, either masking or recovering from failures. And when it can't it should report back to the user about.
So the objective was to make sure our system supported this to a reasonable extent.

Experiment 1

Steps taken throughout the experiment:

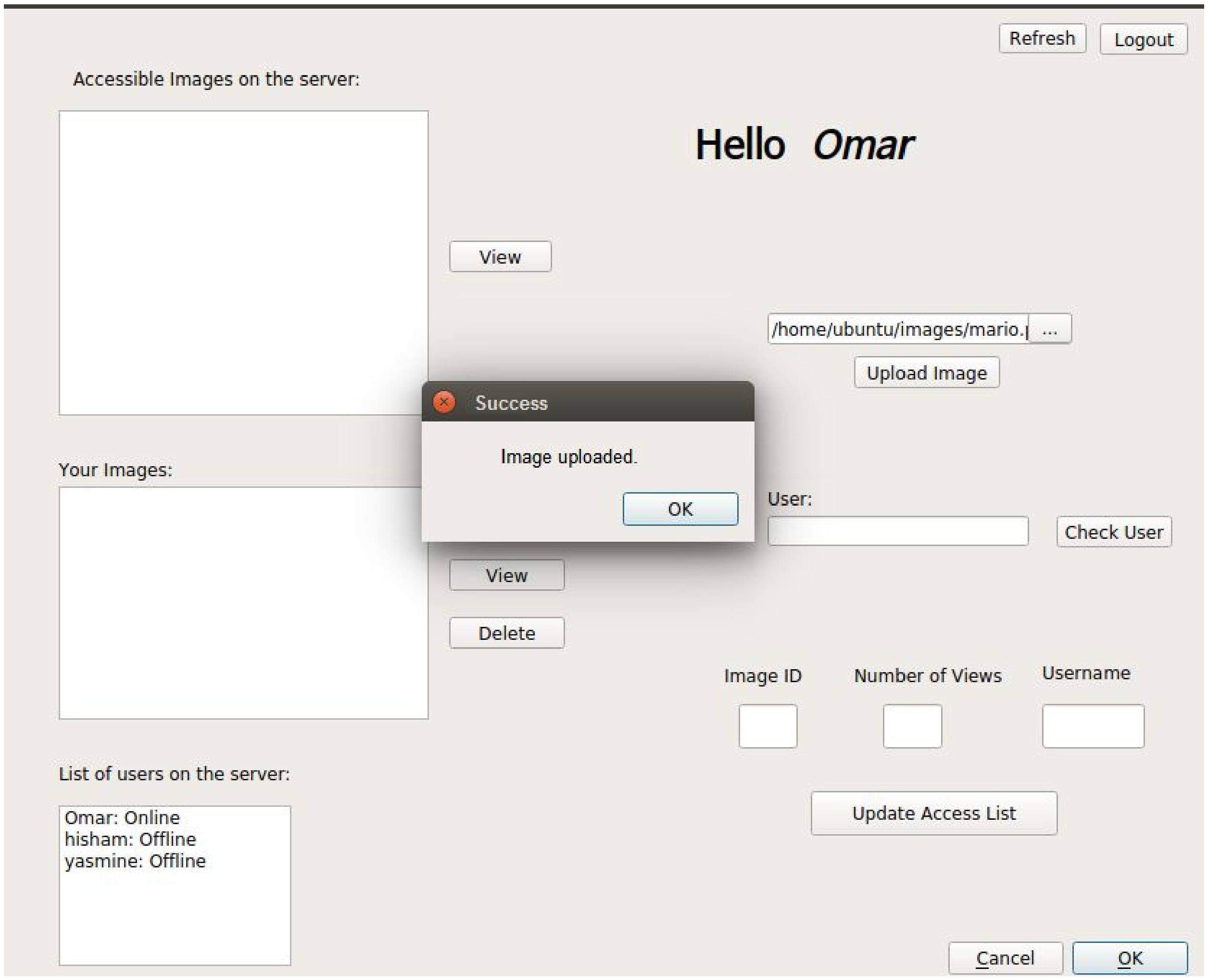
- A user logs into the system.
- We disconnect the server for a short period of time.
- The user then uploads an image to the server or requests any remote operation in general.
- The server is reconnected.

Expected Output:

- As soon as the connection is being established, an acknowledgment is being popped up in the user interface informing the user that the image has been uploaded.

Output and Analysis

As demonstrated in the screenshot below, the expected output was correctly observed on the client side.



Omar: Online
hisham: Offline
yasmine: Offline

We also observe the logs from the communication module and socket of both the client and server.

[9188415790]   Communication log ready.
[9188651541]   Client communication initialized.
[5875988947]   REQUEST   Message put on send queue: ID:1 OpID:LoginOperation, 33 bytes, to: 192.168.43.68:47444
[5876218836]   Packet written to socket: Packet ID:1, 33 bytes, Message ID:1, to: 192.168.43.68:47444
[5939532143]   Packet read from socket: Packet ID:1, 1 bytes, Message ID:2, from: 192.168.43.68:47444
[5939558789]   Incoming message now on queue: ID:2, 1 bytes, from: 192.168.43.68:47444
[5939614339]   REPLY Message received: ID:2 OpID:LoginOperation, 1 bytes, from: 192.168.43.68:47444

…………………...

The login operation is being displayed in the last line, after some automatic operations like refreshing and getting the list of users, the client invokes the upload operation which includes sending the image in a message that consists of packets.

[3869335848]   REQUEST   Message put on send queue: ID:11 OpID:uploadImage, 137698 bytes, to: 192.168.43.68:47444

```
[3869748541]    Packet written to socket: Packet ID:1, 4096 bytes, Message ID:11, to:
192.168.43.68:47444
[3869983219]    Packet written to socket: Packet ID:2, 4096 bytes, Message ID:11, to:
192.168.43.68:47444
[3870211442]    Packet written to socket: Packet ID:3, 4096 bytes, Message ID:11, to:
192.168.43.68:47444
[3870442376]    Packet written to socket: Packet ID:4, 4096 bytes, Message ID:11, to:
192.168.43.68:47444
                                        …………………
```

Once the client recognizes that packets were not received during some time limit (timeout), it displays a fail registry in the log, followed by retransmitting of the last operation invoked.

```
[3892806508]    Packet written to socket: Packet ID:32, 4096 bytes, Message ID:11, to:
192.168.43.68:47444
[3893025637]    Packet written to socket: Packet ID:33, 4096 bytes, Message ID:11, to:
192.168.43.68:47444
[3893183216]    Packet written to socket: Packet ID:34, 2530 bytes, Message ID:11, to:
192.168.43.68:47444
[8865786385]    Packet receive fail.
[8865803073]    Packet receive timed-out. Retransmitting last REQEUST sent.
```

The client keeps retransmitting until the message is being received by the server. This happened because the client managed to reconnect with the server before retransmitting a certain number of times, otherwise, a connection error would be displayed in the client

Log from the server:

```
[8636276865]    Packet read from socket: Packet ID:25, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[8641119852]    Packet read from socket: Packet ID:26, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[8644999316]    Packet read from socket: Packet ID:27, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[8648914256]    Packet read from socket: Packet ID:28, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[8652294989]    Packet read from socket: Packet ID:29, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[8656340988]    Packet read from socket: Packet ID:30, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[8659370672]    Packet read from socket: Packet ID:31, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[8663352807]    Packet read from socket: Packet ID:32, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[8665528199]    Packet read from socket: Packet ID:33, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[8668566766]    Packet read from socket: Packet ID:34, 2530 bytes, Message ID:11, from:
192.168.43.209:58802
[3670355777]    Packet read from socket: Packet ID:1, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3673478873]    Packet read from socket: Packet ID:2, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3678620269]    Packet read from socket: Packet ID:3, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
```

```
[3681011330]    Packet read from socket: Packet ID:4, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3683737277]    Packet read from socket: Packet ID:5, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3687880415]    Packet read from socket: Packet ID:6, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3690971659]    Packet read from socket: Packet ID:7, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3696449215]    Packet read from socket: Packet ID:8, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3700061738]    Packet read from socket: Packet ID:9, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3702987548]    Packet read from socket: Packet ID:10, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3706069778]    Packet read from socket: Packet ID:11, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3710375582]    Packet read from socket: Packet ID:12, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3713364586]    Packet read from socket: Packet ID:13, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3716102093]    Packet read from socket: Packet ID:14, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3718870667]    Packet read from socket: Packet ID:15, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3723906199]    Packet read from socket: Packet ID:16, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3728015521]    Packet read from socket: Packet ID:17, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3733076942]    Packet read from socket: Packet ID:18, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3735554863]    Packet read from socket: Packet ID:19, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3738639115]    Packet read from socket: Packet ID:20, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3742347583]    Packet read from socket: Packet ID:21, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3746294459]    Packet read from socket: Packet ID:22, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3749606146]    Packet read from socket: Packet ID:23, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3754152425]    Packet read from socket: Packet ID:24, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3754164306]    Incoming message now on queue: ID:11, 137698 bytes, from: 192.168.43.209:58802
[3754518950]    REQUEST         Message received: ID:11 OpID:uploadImage, 137698 bytes, from:
192.168.43.209:58802
[3756181465]    Packet read from socket: Packet ID:25, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3760785731]    Packet read from socket: Packet ID:26, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3763683389]    Packet read from socket: Packet ID:27, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3767454417]    Packet read from socket: Packet ID:28, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
```

```
[3865260515]    Packet read from socket: Packet ID:29, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3867062150]    Packet read from socket: Packet ID:30, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3871035635]    Packet read from socket: Packet ID:31, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3874947806]    Packet read from socket: Packet ID:32, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3877385597]    Packet read from socket: Packet ID:33, 4096 bytes, Message ID:11, from:
192.168.43.209:58802
[3880212140]    Packet read from socket: Packet ID:34, 2530 bytes, Message ID:11, from:
192.168.43.209:58802
```

Here, we can see the packet IDs are not ordered, the server sorts them using the packet IDs and since each packet has the total number of packets of the message embedded within it, the server issues that a one whole message is being received when that number of packets is being read.
Packets read after the “message received” entry are being ignored since these are duplicates caused by the retransmission from the client.

We can see in the code regarding packer ordering:

```
if (received_packet.getNumOfPackets()==int(incomingPackets[make_pair(IP,port)].size()))
                {
sort(incomingPackets[make_pair(IP,port)].begin(),incomingPackets[make_pair(IP,port)].end(),sortByPack
etID);
…….
}
```

Here when the number of packets required equal the number we have in the queue we go to sort them using the function *sortByPacket*

```
bool sortByPacketID(packet a, packet b)
{
   return (a.getPacketID()<b.getPacketID());
}
```

Regarding discarding duplicate packets:

```
if (find(incomingPackets[make_pair(IP,port)].begin()
                        ,incomingPackets[make_pair(IP,port)].end(), received_packet)
                       == incomingPackets[make_pair(IP,port)].end())
```

Here we try to find the occurrence of the same packet in the queue.

Finally, the server replies back with the acknowledgment that the image has been uploaded as shown below.

```
[4843139626]    REPLY Message put on send queue: ID:12 OpID:uploadImage, 8 bytes, to:
192.168.43.209:58802
[4843994119]    Packet written to socket: Packet ID:1, 8 bytes, Message ID:12, to: 192.168.43.209:58802
```

The experiment can be cross referenced in the below code which is a part of the receiveThreadFunction in the communication module::

receiveFromSocket function returns whether there is a packet to be read from the socket or not where a sending function invoked by the client precedes it. Hence, if the packet is not being read after a certain time limit (timeout), then communication log would note *"Packet receive fail."* which we can find in the above log. Then the client retransmits the same message for a known number of times. We fixed those times in the experiment as 12 retransmits each of 5 seconds.

```
 bool status = receiveFromSocket(received_string,received_size, IP, port);
     if (isClient && !status)
     {
        int i;
        commLog("Packet receive fail.");
        for (i =0 ; i < TIMEOUT_RETRANSMIT_TIMES; ++i)
        {
           if (terminate)
           {
              return;
           }
           mutexWaitingOnReply.lock();
           if(waitingOnReply == false)
           {
              break;
           }
           else{
              mutexWaitingOnReply.unlock();}
           commLog(string("Packet receive timed-out. Retransmitting last REQEUST sent."));
           sendMessage(get<2>(lastMessageSent),get<0>(lastMessageSent),get<1>(lastMessageSent));
           status = receiveFromSocket(received_string,received_size, IP, port);
           if(status)
           {
              break;
           }
           else
           {
              commLog("Packet receive fail.");
           }
        }
```

Experiment 2

Experiment

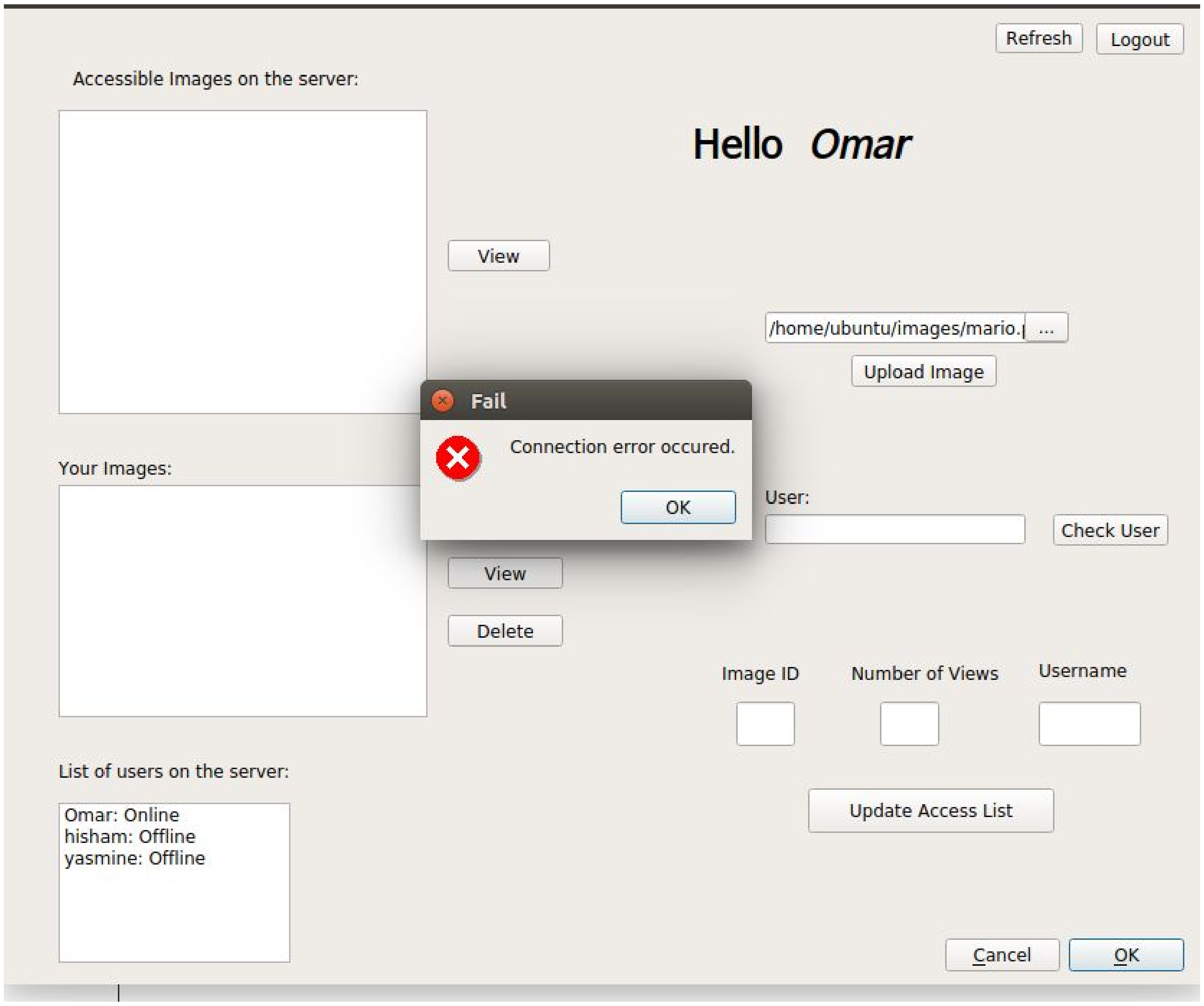
- The network between the server and client is disconnected.
- The client requests an operation that needs to communicate with the server.
- The network is not reconnected and the client is observed.

Expected Output

- After a minute (our maximum wait on receiving a message on client side) the client should report the connection error and it permanently failed to receive any message from the server.

Output and Analysis

In our experiment we requested to upload an image to the server, the expected output was correctly observed (screenshot below).



Here the client application presented the user with the error stating that there was a connection error and the operation failed.

We also observe the logs from the communication module and socket of both the client and server.

On client side we have:

```
[9647654873]    REQUEST        Message put on send queue: ID:5 OpID:uploadImage, 1057213 bytes,
to: 192.168.43.209:753357
[9648448078]    Packet written to socket: Packet ID:1, 4096 bytes, Message ID:5, to:
192.168.43.209:753357
[9649245019]    Packet written to socket: Packet ID:2, 4096 bytes, Message ID:5, to:
192.168.43.209:753357
```

……… many packets written here …..

```
[3765426696]    Packet written to socket: Packet ID:258, 4096 bytes, Message ID:5, to:
192.168.43.209:753357
[3765454913]    Packet written to socket: Packet ID:259, 445 bytes, Message ID:5, to:
192.168.43.209:753357
[4646441932]    Packet receive fail.
[4646446071]    Packet receive timed-out. Retransmitting last REQEUST sent.
```

This pattern repeats for 12 more times, at 5 second intervals. (The number of retransmissions and the timeout for the client receive). We can note that the timestamp here in milliseconds, is almost five seconds. Between the last packet written and the receive fail.
Finally we get to the point of failure.

```
[9646457472]    Permenantly failed to receive REPLY for last REQUEST sent
```

On server side:

It is actually pretty boring the last message we receive is the previous one. After the network is disconnected we don't have any messages received and neither sent.

Pointing out to code:

Again from experiment 1 we have the same code dealing with the looping over retransmissions and breaking out of the loop when the client starts to receive packets. But in experiment 2 this does not ever happen and after the loop we reach this condition

```
if(i == TIMEOUT_RETRANSMIT_TIMES)
{
   mutexWaitingOnReply.lock();
   waitingOnReply = false;
   mutexWaitingOnReply.unlock();
   mutexClientReceiveFail.lock();
   clientReceiveFail = true;
   mutexClientReceiveFail.unlock();
   continue;
}
```

Which tells the cleint that it is no longer waiting for a reply, *waitingOnReply* is false and that it failed to receive the last reply, *clientReceiveFail* is true. Of course the receiving thread will continue and loop again to serve future requests.

But both these changes affect other aspects, when the receive thread loops again. It is blocked by a condition that will not allow it to further timeout while the client is NOT waiting for any replies. This is the condition at the start of the loop in the receive thread.

```
while(1)
{
   mutexWaitingOnReply.lock();
   if (waitingOnReply)
   {
      mutexWaitingOnReply.unlock();
      if (terminate)
      {
         return;
```

```
            }
            break;
        }
        else
        {
            mutexWaitingOnReply.unlock();
            if (terminate)
            {
                return;
            }
        }
    }
```

Also back to the call for receive message that the client had invoked. We can see know that this code will allow receive message to return false to its caller.

```
    if(clientReceiveFail)
    {
        clientReceiveFail = false;
        mutexClientReceiveFail.unlock();
        commLog(string("Permenantly failed to receive REPLY for last REQUEST sent"));
        return false;
    }
    else
    {
        mutexClientReceiveFail.unlock();
    }
```

Conclusions

We were able to demonstrate

- A system based on RPC communication that implements a full fledged application on top of a self contained communication system.
- A responsive GUI that was demonstrated in front of the instructor.
- Extensive logging system.
- A flexible timeout and retransmission based system that allows for interim network disconnectivity without failing the operations.
- Fault tolerance to a reasonable manner that satisfies the requirements.

We were challenged most by the existence of several threads. Many problems we faced were related to syncing the threads.

Work done by each member of the team

- RPC Communication: Hager, Hussam, Ola
- Image Steganography: Ola, Hussam
- Client: Hussam, Hager
- Server: Mohamed, Ola
- GUI: Hager, Mohamed
- Fault tolerance: Hussam, Mohamed
- User database: Ola
- Report: Hussam, Mohamed