DISTRIBUTED SYSTEMS FINAL PROJECT REPORT

Malak Sadek - 900140107,
Mohamed Hegab – 900142811,
Ibrahim Abdelrahman - 900131254,
Baher Mahmoud - 900133218,
Mohamed Badreldin - 900140813

1. Policies and Assumptions Used

a) Usage of RRA and RR Schemes

i) RRA:

The request-reply-acknowledge scheme is only used when the data sent is important and for non-idempotent operations. It is used while sending the fragments of an image, since every single packet needs to arrive correctly for the service to work and because sending the same packet twice would corrupt the image (thus changing the state of the system).

ii) RR:

The request-reply scheme is used for other operations that do not affect the state of the system and can be repeated multiple times with the same effect. This includes asking for a client list from the centralized server, asking for a picture list from a client, and sending and receiving views. It is used for views since the system works by sending a number of views and overwriting the old value of views remaining for an image with the new number, so even if the views are sent multiple times it will just keep overwriting the same value.

b) Usage of Maybe, At-Least-Once, and At-Most-Once Schemes

i) Maybe:

We only use the maybe scheme for requesting an image. If the request is received then packets will automatically start being sent to the server thread of the client who made the request, however if it did not arrive, then nothing will happen and the client will have to request the image again later.

ii) At-Least-Once:

At-Least-Once is used in instances where RR is employed. These include requesting a client list from the centralized server, asking for a picture list from a client, and sending and receiving views. The client waits for a reply and if it does not receive it, then it retransmits the request. These request types are not added to the history of the server thread on the other client and therefore multiple instances might be sent, however this does not affect the state of the system so it is tolerated.

iii) At-Most-Once:

At-Most-Once is used in instanes where RRA is employed. This is for sending the fragments of an image, since packets cannot be dropped and cannot be duplicated without ruining the image, then each one must arrive at most once. This is done through the history and acknowledgement mechanisms.

c) Usage of Thread Models

The centralized server only has one thread that continuously waits for requests. The clients have a client thread and a server thread, each operating on their own socket. The client thread takes requests from the users and displays results, the server thread is responsible for sending the requests to another client or the centralized server and handling requests and replying when received from another client. Since there is only one server thread per client, we do not employ any of the models discussed in class as we felt they would create too much overhead. The server thread processes one request from a client entirely and then moves on to the next one.

d) Usage of Fragmentation

For images, especially after marshalling, the message size exceeds the limit that UDP allows for sending a packet. Therefore, we fragment the data into packets of size 500 characters and send them sequentially. Each packet contains the same header except that the request ID is incremented by one (the request number part of it). This occurs until all packets are sent, which is determined by the total length of data \500. The receiver then concatenates all packets without their header to form the picture again (after unmarshalling).

e) Security

i) Usernames

When a client first starts, they are required to enter a username to send to the centralized server, if another client already uses the same username, then the client is not allowed to use it to prevent them acting as another client.

ii) Marshalling & Unmarshalling

Messages sent are marshalled at the sender and unmarshalled at the receiver so that they cannot be intercepted.

iii) Steganography

The number of views are hidden as a data field in the header and marshalled along with the data so that they are unable to be changed by the users. Additionally, the default image viewed when the number of views remaining for an image equals 0 is handled by the server thread of a client and is inaccessible by the user on their own.

iii) Checking Reply Source

Whenever a client receives a server reply, the request ID of the reply is checked against the request ID of the original request to make sure that this is the actual reply expected.

iv) Checking Client Location

The server checks whether a client making a request is local or remote by comparing their address to their own, it then offers them a different set of services accordingly.

v) Checking View Rights

When a server receives requests for more views, it first checks that the client making the request has received the image before.

2. Failure Handling

a) Timeouts & Retransmissions & Acknowledgement

Clients use timeouts on their sockets in order to retransmit requests messages that the server has not replied to (meaning that the request was lost). Retransmissions are used when a socket timeouts to retransmit the message that was not received. After retransmitting, if a server reply is received, this will be followed by an acknowledgement. Acknowledgements are used to enforce the at-most-once scheme for sending packets.

b) History Table & Message IDs & Duplicates & Obsoletes

A history table is kept by both the centralized client and the server thread of the clients. Whenever a new request is received, its message ID is checked. This message ID is a combination of a unique client ID and the request number, which is incremented with every new request. If the message ID is found in the history table, then this is a duplicate request and is ignored. If it is not found, it is added to the history and a reply is sent. A message ID is deleted from the history table when an acknowledgement is received containing the same message ID. Deleting a message ID also deletes other entries in the table that have the same client ID but smaller request numbers as this indicates that a more recent request is being acknowledged, so the earlier requests must have taken place as well.

3. Flow of Events

1. When a new client runs, it prompts the user to enter their username.

2. A message is sent to the centralized server containing “c-[username]”

3. The centralized server adds the client to its total client list and its currently connected client list

4. The client prompts the user to make a request.

5. If the user quits, the client sends the centralized server a message containing “q”. The server then removes the client from the currently connected client list, but not the total client list.

6. If the client requests a client list, the centralized server will reply.

7. Whenever the centralized server receives a request, it also checks its currently connected clients by sending a message to them and awaits a reply, updating the table as necessary.

8. A client could request an image (without blocking), the server thread of the other client would fragment the image and send the first fragment, the client will acknowledge the first fragment, and so on until the entire image is received.

4. Limitations

These are the known limitations of the system:

- If a client's server is busy serving another client's request, the faster one will be handled, creating a race condition. The client serving will essentially be blocked from serving any other requests until the running request is being handeled. As well as, any requests made while the client is busy will be dropped, and the client requesting will have to issue a new request.
- If the centralized server falls while there are connected clients, the clients can still communicate with each other, but requesting a client list will return an empty list.
- The complexity of the algorithms used is high, requiring heavy computation and leading to larger images taking a longer time to be parsed and transmitted.
- The rate of dropping packets can often be very fast due to the network, leading to buffer overflows, and increasing the buffers past a certain limit will take a lot of memory, a balance must be made between the two.
- No way of referring to peers using their usernames, the port number and IP address must be entered to communicate with them.
- If two different clients have two different pictures with the same name, they will not be both stored on the database as the image name is our unique key.

5. The Usage of RPCId:

As mentioned earlier, the RPCId has many uses within our system. It is used by clients to match a reply to a request made and make sure it is the appropriate reply. Without using this matching, sometimes the client could be requesting an image, and it starts reading its buffer, however it receives another message, such as the "UCC" message used by the centralized server to update its currently connected clients, it would then believe this message is the first packet of the image, which would be the expected reply, ultimately crashing the system.

Another usage of RPCId is in the server's history table. This allows duplicate and obsoletes filtering which is essential when sending packets. Without using a history table, sometimes certain packets are dropped in the process, and sometimes the same packet is sent twice as the server's reply was not sent quickly enough, both these events would cause the final image to be corrupted. Using the history table guarantees that every single packet is received at most once.

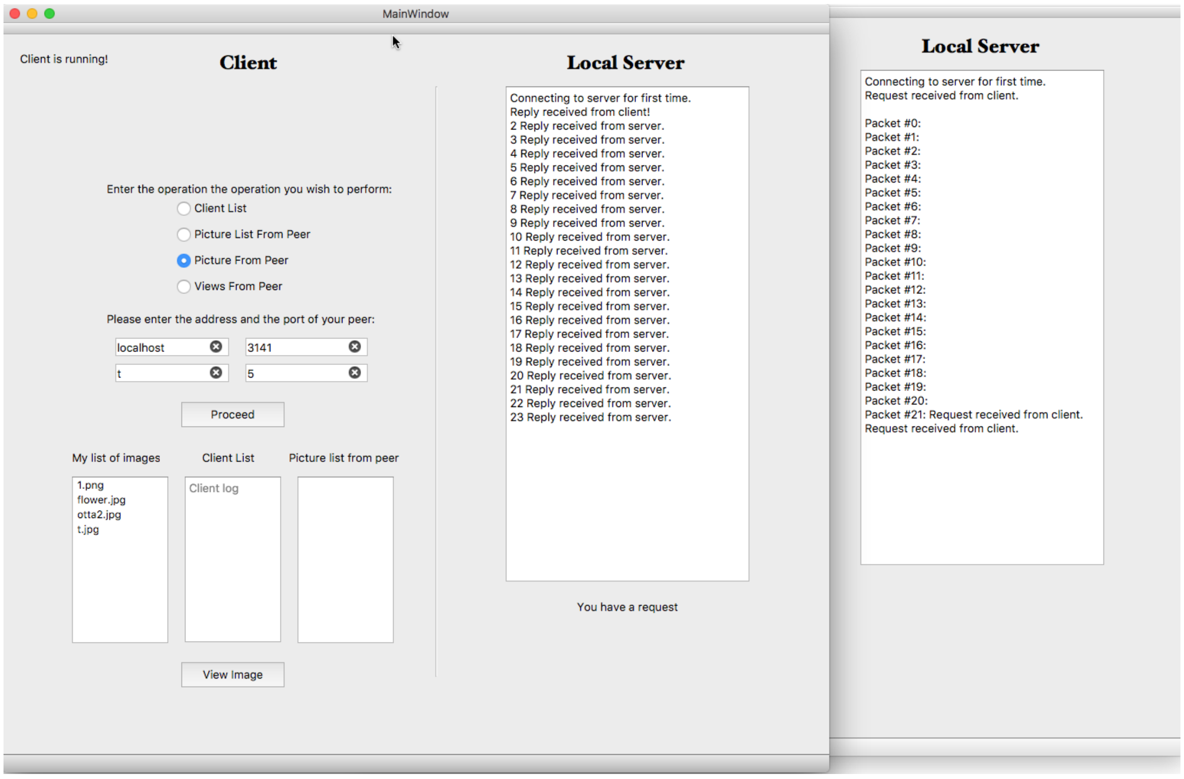
6. The Usage of Qt:

One of the major differences in Qt is that mutex locks are used to pause the GUI when a thread is waiting for input from the user. When we need the peer to reply, we employ signals and slots in Qt, accompanied by mutex locks in order to output in real time and not when the application finishes.

There are two database tables used, one for storing pictures and contains the fields: Picture Name, Owner (me or not me), Address (if not me), Port (if not me), Number of Views, and Data (picture in a marshalled format), and another database for keeping track of the users that a client has sent images to. When a client requests views, this table is used to validate that this client has been sent this picture before.

The pictures owned by a client are stored in Users/Pics, this also contains the database which stores the images they received in a marshalled format so that users are unable to open them outside the application.

The GUI is composed of a window for the centralized server, and one for the client. The client window is divided into its client thread and its local server thread. The client side can make requests and will receive results such as the picture list of another client or the client list from the centralized server, while the server side will show a log of replies received as well as packets and acknowledgements received and other server-side messages. When the client window is closed, this sends a message to the centralized server that the client is disconnecting.

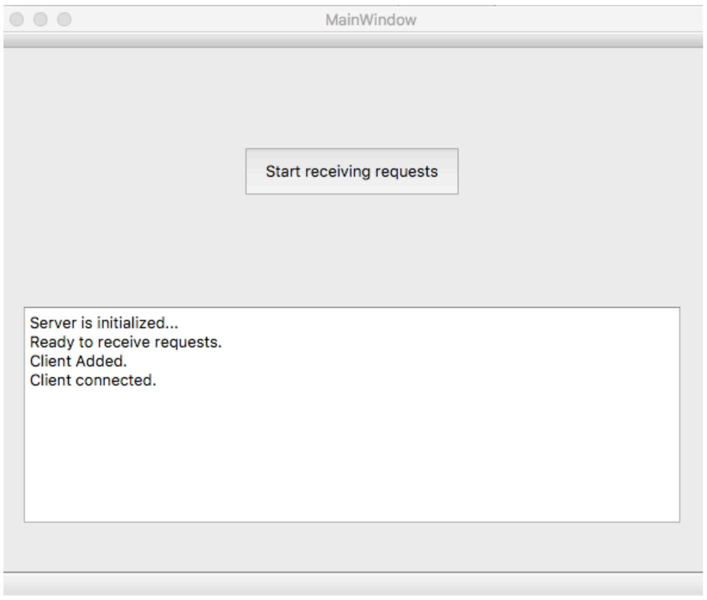


```
1.png
flower.jpg
otta2.jpg
t.jpg
```

```
Connecting to server for first time.
Reply received from client!
2 Reply received from server.
3 Reply received from server.
4 Reply received from server.
5 Reply received from server.
6 Reply received from server.
7 Reply received from server.
8 Reply received from server.
9 Reply received from server.
10 Reply received from server.
11 Reply received from server.
12 Reply received from server.
13 Reply received from server.
14 Reply received from server.
15 Reply received from server.
16 Reply received from server.
17 Reply received from server.
18 Reply received from server.
19 Reply received from server.
20 Reply received from server.
21 Reply received from server.
22 Reply received from server.
23 Reply received from server.
```

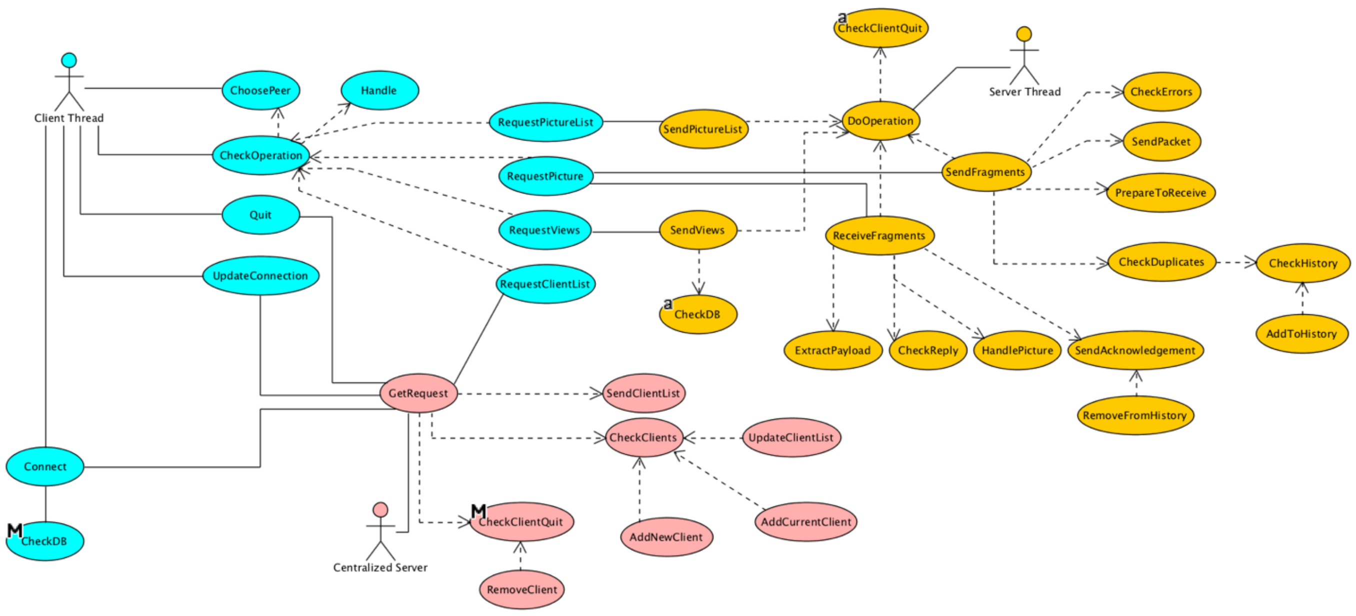
```
Connecting to server for first time.
Request received from client.

Packet #0:
Packet #1:
Packet #2:
Packet #3:
Packet #4:
Packet #5:
Packet #6:
Packet #7:
Packet #8:
Packet #9:
Packet #10:
Packet #11:
Packet #12:
Packet #13:
Packet #14:
Packet #15:
Packet #16:
Packet #17:
Packet #18:
Packet #19:
Packet #20:
Packet #21: Request received from client.
Request received from client.
```



```
Server is initialized...
Ready to receive requests.
Client Added.
Client connected.
```

7. Use Case Diagram



8. Classes Used:

a) Message Class:

i) Message Class Data Members:

int message_type;
int operation;
char* message;
size_t message_size;
int rpc_id;
int fraged;
int frag_count;
int frag_total;
int nOv;
static const std::string base64_chars =
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
"abcdefghijklmnopqrstuvwxyz"
"0123456789+/";

ii) Message Class Functions:

Message(*int message_type, int operation, char* p_message, size_t p_message_size, int p_rpc_id, int fraged, int frag_count, int frag_total, int nOv*)

-Constructor, simply sets all the fields in the Message class, the message itself is unmarshalled

Message(*char* marshalled_base64*)

-Takes an already marshalled string and unmarshals it, setting all the data fields of the class from the information extracted

char* marshal(*int datamarsh*)

-Marshals the data & header or the header only depending on the bool value of datamarsh.

bool isBase64(*char c*)

-Used in marshalling and unmarshalling, decides whether the incoming string can be marshalled or unmarshalled based on the base64_chars variable.

**There are also getters and setters for all the data fields of the class

b) UDPSocket Class:

i) UDPSocket Data Members:

int mysock;
int peersock;
sockaddr_in peerAddr;
sockaddr_in myAddress;
int myPort;
int peerPort;
bool enabled;

ii) UDPSocket Class Functions:

sockaddr_in initializeServer (*int sock, int port, sockaddr_in sa*)

-This is called by the server socket and sets the socket's port, family, address, etc.

sockaddr_in* initializeClient(*int sock, sockaddr_in mSocketAddress, sockaddr_in yourSocketAddress, int port, char computer[50]*)

-This is called by the client socket and sets the socket's port, family, address, etc.

void writeToSocket(*int sock, char reply[1000000], size_t replylength, int x, sockaddr_in aSocketAddress, socklen_t addrlength*)

-This uses the "sendto" method to write to the specified socket

int readFromSocketWithTimeout(*int sock, char message1[1000000], char message2[1000000], size_t message2length, size_t length, int x, sockaddr_in aSocketAddress, int aLength, int timeout*)

-This is used by the client, it employs "recvfrom". It sets the socket to timeout (the duration is received as a parameter) instead of block, if the socket times out, it will retransmit the same message again, repeating this 10 times before giving up entirely.

int ServerReadFromSocketWithTimeout(*int sock, char message1[1000000], size_t length, sockaddr_in aSocketAddress, int aLength, int timeout*)

-This is used by the server, it also set the socket to timeout instead of block, however it does not try to retransmit.

sockaddr_in readFromSocketWithBlock(*int sock, char message1[1000000], size_t length, int x, sockaddr_in aSocketAddress, int aLength*)

-This is used by the server, it blocks and waits for a message to arrive to unblock.

**There are also getters and setters for all the data fields of the class

c) UDPClientSocket & UDPServerSocket Classes:

i) Functions:

Both classes inherit from UDPSocket. UDPClientSocket utilizes initializeClient, writeToSocket, and readFromSocketWithTimeout. UDPServerSocket utilizes initializeServer, writeToSocket, readFromSocketWithBlock and ServerReadFromSocketWithTimeout.

d) Server Class:

i) Server Class Data Members:

UDPServerSocket udpServerSocket;
static int sock;
int port;
int history[100];
struct sockaddr_in connectedClients[5];
std::string connectedUsernames[5];
struct sockaddr_in totalClients[5];
int numberOfConnections;
int totalConnections;

ii) Server Class Functions:

Server*(int _myPort*)

-This initializes the port of the server and initializes all tables.

ssize_t getRequest()

-This awaits a message and then checks whether the client is local or remote, checks for duplicates, checks whether the client is in the appropriate tables or not, and then handles the request (sends the client list).

void serveRequest()

-This is the public version of getRequest() and loops over it indefinitely always waiting for requests, it also periodically calls updateClientList().

void sendReply (*int sock, char* reply, size_t replylength, int x, sockaddr_in aSocketAddress, socklen_t addrlength*)

-This function is what is used to send replies to clients.

----
bool NewClientConnection(*char* message1*)

-If the server receives a message, this function checks if this message starts with a “c-“ which means a client is connecting for the first time, it then adds its username (also in the message) to both the totalClients and connectedClients lists.

bool checkCurrentClientConnections()

-This function checks if an incoming message's source client has connected to the server before

void addClientConnection()

-If checkCurrentClientConnections() is false, then this function adds the incoming message's source client to the currently connected clients list.

----

void updateClientList()

-This is periodically called, it sends a message containing "UCC" to each client on the server's currently connected client list, if they do not reply to the message, this means they are no longer connected and so are removed from the list.

----

void checkClientQuit(*char* message1*)

-If the server receives a message, this function is called to check if this message is "q" which means that a client is quitting, it then removes this client from the list of connected clients.

int checkClientType()

-This checks whether a client is remote or local by comparing the address to its own.

void returnClientList(*int ID*)

-This packages the client list and sends it to the client who requested it.

----

bool duplicateFilter(*int ID*)

-This checks for the request ID of the incoming message in the server's history table. The requestID is a combination of the client's ID and the request number. If it is found in the table, the function returns true.

void addToHistory(*int ID*)

-If duplicateFilter() had returned false, this function adds the request ID of the incoming message to the server's history table.

e) Client Class:

i) Client Class Data Members:

UDPClientSocket udpSocket;
UDPServerSocket udpSSocket;
static int sockc;
static int sockss;
int history[100];
static int socks;
std::string username;

ii) Client Class General Functions:

Client(*char* computer, int port*)

-This initializes the client thread socket and server thread socket, runs the server thread, and initializes the server thread's history.

void runClient(*char* string*)

-This creates the client thread and then joins it to the main thread when it has finished making a request (all inside a while(1) loop).

void runServer()

-This creates the server thread and detaches it from the main thread to continuously service incoming requests.

iii) Client Class Client Thread Functions:

void connect()

-This is automatically called when the client starts and sends "c-[client's username]" to the central server

bool Quit(*char* string*)

-This is called when handle() discovers that the user wants to quit, it informs the server thread that it is quitting.

int handle()

-This asks the client whether they want to quit or do something else, it then runs the client thread with the selected option.

int updateConnection()

-This is called upon receiving the central server's "UCC" message to update its currently connected clients list.

int selectOperation()

-This asks the client which operation they want to perform. They can request a client list from the central server, or request a picture list, picture, or views from another client. Each calls a respective function.

void choosePeer()

-This prompts the client to enter the address and port number of the server they want to send a request to. This does not apply to the centralized server.

void doOperationClient(*char* string*)

-This checks whether the user wants to quit or not, sets the request ID, and then calls the respective function based on the request that the client wants to make.

void checkReplySource(*char* message1, Message originalMessage, Message testMessage*)

-This makes sure that the reply received from any server, whether the centralized or another client's server thread is indeed the reply the client is waiting for. It does this by comparing the request ID of the reply received with the request ID of the request sent (should be the same).

void requestClientList()

-This requests a client list from the centralized server by sending "cl".

void requestPictureList()

-This requests a picture list from the server thread of another client by sending "pic".

void requestPicture()

-This requests a picture from the server thread of another client by sending the picture's name and the number of views wanted.

void requestViews()

-This requests more views for a certain picture by sending the picture's name and the number of views wanted.

iv) Client Class Server Thread Functions:

int clientQuit(*char* string*)

-Checks whether the client is quitting and it sends "q" to the centralized server so that it can be removed from its currently connected clients list.

void duplicateFiltering(*int ID*)

-When an acknowledgement is received, all entries in the server thread's history with the same requestID and with a smaller request number but same client ID are removed from the history.

int checkHistory (*int ID*)

-This checks for the requestID of an incoming message in the server thread's history to know whether the message is a duplicate or not.

void addHistory (*int ID*)

-When a new request is received, if it is not already in the history (a duplicate) then it is added to the history until its reply is acknowledged.

----

char* extractPayload (*int index, char* message*)

-This receives a message and separates the header from the data.

char* errorCheck (*char* temp, char* message, int total, int count*)

-This makes sure the packet about to send does not contain any errors from memory leaks

void prepareToReceive(*int ID, int op*)

-This is called when a client request is received, it sends a message to the client to acknowledge that the request has been received and that it will begin being serviced.

void handlePicture (*std::string fragments*)

-This is called when a picture has been received from the server thread of another client, it puts the fragments together, creates the .jpg file, and outputs it.

Message readFile(*char* pic*)

-This opens an image and reads it to prepare it for being sent.

----

void receiveFragments(*Message M, char* string*)

-This receives all the fragments of an image and stores them in a buffer until they are all received and then hands them to handlePicture().

void sendPacket(*char* temp, int ID, int total, int count, int numberOfViews*)

-This sends packets (containing picture fragments) to the server thread of another client.

void sendAcknowledgement(*int ID*)

-This sends an acknowledgement for each picture fragment received so that the server thread can remove the reply containing that fragment from its history and move onto the next one.

void sendFragments(*Message M2, int ID, size_t size, int nOv*)

-This sends the packets and waits for an acknowledgement and either retransmits the packet, sends the next packet, or aborts the operation.

void sendPictureList(*int ID*)

-This sends a list of the pictures available with the client to send.

void doOperationServer()

-This calls the appropriate function based on the kind of request received, filtered on whether the message is less than 500 characters or not to distinguish whether it is the first packet of an image or another kind of request.