



**THE AMERICAN  
UNIVERSITY IN CAIRO**  
الجامعة الأمريكية بالقاهرة

## **Distributed Systems**

**Fall 2019**

### **Remote Invocation Middleware Project: Exercise #2**

#### **Group Members**

Eslam A. Soliman 900163257

Fadi A. Dawoud 900163212

Ahmed Abouzaid 900163141

Mohamed Abdel Hamed 900163202

## **Build instructions:**

### **1) The Peer**

a) In the directory containing the source files, run the following script:

```
./peerCompileRun
```

The peer process from testMain.cpp will be compiled and run, connecting to the central service broker whose address and port number are given in the Peer constructor in testMain.cpp.

### **2) The Service Broker**

a) In the directory containing the source files, run the following script:

```
./brokerCompileRun
```

The service broker process will start and will serve peer processes who designate it as their service broker.

## **Work distribution and meetings/decisions log.**

### **Exercise 1:**

Oct. 16<sup>th</sup>:

A group meeting to break down the requirement of the first exercise and the whole project's infrastructure into tasks and distribute them to the group.

#### **Tasks assigning:**

- Message (Mohamed)
- UDPSocket (Fadi)
- UDPClientSocket (Ahmed)
- UDPServerSocket (Eslam)
- Client (Ahmed)
- Server (Eslam)
- ServerMain (Eslam)
- ClientMain (Ahmed)
- Bombarder/Jammer (Mohamed)

### **Exercise 2:**

Nov 2<sup>nd</sup>:

A group meeting was held to identify the most important requirements of the second exercise. The following main tasks were identified and assigned to members:

- Central service implementation (Broker class) (Mohamed)
- Peer class model and extension, including the implementation of the various application-level functions (Eslam, Mohamed)
- Handling long messages, packet re-ordering, packet dropping (Fadi)
- Graphical User Interface design (Ahmed)
- Steganography (Fadi)
- Interfacing the GUI and the application-level methods (Mohamed, Ahmed)

## Important Design Decisions for Exercise 2

### 1- The communication paradigm

We had built our UDPSocket class on UDP in the course of exercise 1. Hence, we had a choice of adopting either RPC or RMI for our design. We chose RPC, as we did not see a benefit in partitioning a single peer process into objects. In other words, a peer process can be contained in one object, and hence, there was no need for an overhead of RMI.

We are employing a history table for non-idempotent methods duplicate checking. We also chose RR protocol for our RPC system. R protocol, implying *maybe* semantics, would not be suitable for almost all of our required methods (discussed below). RRA protocol was viewed as overkill, and was replaced with the requester repeating their request.

### 2- Application-level methods

Within the initial planning meeting (and also refined in numerous later meetings) we have identified the following important application level methods. Note that a stub module on the peer process, which masks an RPC to the central service and/or another peer, executes these methods.

Operation ID	Function	Request Arguments	Reply Arguments	Idempotent?
0	Login	Username, password	Session token, or 0 if invalid	No; the session token changes
1	Register	Username, password	0: invalid username OR 1: Registered (success)	No; the server would return 0 if the request is executed more than once because user would be already existent
2	Get user IP	Session token, username	IP address, port number, last	Yes

			active (e.g. 10 seconds ago)	
3	Get previews feed	Session token	List of usernames & their respective preview images with their titles	Yes
4	Get user preview	Session token, username	List of preview images with their titles	Yes
5	Upload image preview	Session token, image title, image thumbnail	1: success 0: failure	Yes
6	Get all image titles of a user	null	List of image titles	Yes
7	Request Image	Image name	0: refused image: granted	Yes
8	Request Image Quota	image title, needed quota	0: denied 1: granted	Yes; the needed quota is assigned, not incremented
9	Set Image Quota	granted quota	No reply	Yes

**Note:** All arguments are comma separated in the same order they appear left to right.

### 3- Sockets and ports

There was a disagreement over whether we should use one or two sockets per peer process. If we use two sockets, one would be a dedicated listener while another would be a dedicated sender. Using one, we would have to utilize the same socket on separate threads for listening and sending.

We eventually opted to use a single port, to facilitate sending replies back to senders. As receivers can only know the port number bound to the sending socket of the sending peer process, they would not know the port number bound to the receiving socket of the sending process, so replies would entail an extra transaction on the central service. Generally, a single address and port number would identify every online member of the system.

#### 4- Matching replies to requesting threads

As two separate threads perform listening to incoming messages and performing application-level methods, a scheme must be developed to facilitate matching the incoming replies to the requesters.

Our first idea was to use two tables, one of which stores reply messages and another stores `condition_variables` which enable sleeping and notifying. When a requester sends out a request, it announces that it is waiting for a reply by placing its condition variable in the appropriate table. A listener, when encountering the reply for this requester, will store it in the appropriate table and use the `condition_variable` to wake the thread up. Upon waking, the thread will check if its reply has arrived. Also, a thread wakes up on its own after a specified timeout.

However, several errors were happening after implementing this scheme. We switched to a different system, in which dedicated threads check for the incoming messages and do the waking up. A requester thread announces that it is waiting for a reply and then terminates. If an incoming message arrives, or timeout happens on a table entry, a new thread is spawned to resume execution.

The problem with this scheme was that, by terminating, the thread could lose its context. For example, in the case of nested RPC's, like requesting a user's address in the process of asking to view a message, losing the context of the inner RPC was not an option.

After close analysis, we found the error in the first scheme: we were sending out the requests, and then announcing that we are waiting for a reply. However, replies came too early, and so, the listeners do not find the entry for the requester in the mutex table and the requesters are never notified of the arrival. The fix to this was simple: populate the mutex table – announcing that you are waiting for a reply – before you send the request.

## 5- Handling long messages & packet dropping and re-ordering

Some messages would contain images, which means their sizes are to exceed the 8KB maximum UDP packet size (actually, the maximum size for a message is found to be 6130 bytes, as marshaling, using a base64 encoding, adds ~30% to the length of the message).

To overcome this, we had to partition the long messages into smaller portions would respect the 8KB limit. This would happen on the sender end. Each packet would be equipped with its order and the total number of packets in the sequence.

At the receiver side, a data structure exists to facilitate receiving partitioned images. When a part is received, it is “shelved” in its right order inside a vector. Once the final packet receives – the message is completely received – a worker concatenates all the packets and produces an intact message transparently to the rest of the receiving process.

This process handles re-ordering – as packets are placed in their right places – duplicates – as each packet is stored once, the rest ignored – and dropping – as a series of packets can be accumulated over many requests from the sender.

## 6- Steganography

As a requirement to the project, we are to transmit images between peers and store them on the local device in a way that protects the quota integrity. If a message is intercepted through a backdoor, or local device copy of it is accessed, or when the quota for viewing the images is exceeded, a default image is displayed instead of the desired image. This requires us to, somehow, conceal the desired image inside the default message.

Another requirement was to store owner and quota information also within the default image, accompanying the desired image.

To achieve this, we first looked at traditional steganography techniques. The most common technique is done by encoding the bits of the desire image into the least-significant bits of the default image (which causes a slight distortion to the image).

However, we abandoned this technique, because of the size limit. For each bit of the desired image, as well as owner and quota info, we need a byte of the default image. Hence, the default image must be huge to contain large sizes of desired images.

We looked for an alternative technique. We decided to use the comment section of JPEG images, which do not affect how they are displayed, to store our info. This way, no distortion happens to the default image, and there's no bound of the sizes of our desired images.



## Discussion of components:

### The UDPSocket class:

The UDPSocket class is the first level of abstraction above the Linux system calls provided by the Linux communication libraries. It is a wrapper around a socket descriptor and abstracts away the details of binding the socket to a socket address, designating self as a client or a server, and writing or reading from the socket with different options. Below are the implemented methods for this class:

- 1) UDPSocket(): The constructor of the class. It creates generates a new socket descriptor for the new object and sets the details to some default value.
- 2) (private) doBind(): it binds the socket to a local socket address, setting a flag that it is bound.
- 3) (private) initializeMyAddr(): it is a private method, whose function is to initialize the socket address object of the UDPSocket. Unless values are specified for the address and port number the socket should be bound to, default values are used (by passing 0 as port number and INADDR\_ANY as an address).
- 4) initializeServer(char \* \_myAddr, int \_myPort): this method designates the UDPSocket as a server with the given address and port number. It binds the socket to the provided socket address. It returns true if the process is successful.
- 5) initializeClient(char \* \_peerAddr, int \_peerPort): this method designates the UDPSocket as a client of the server whose socket address is provided. If the socket is not already bound to a local socket address, it is bound with a socket address of default values. It returns true if the process is successful and false otherwise.
- 6) initializeClient(sockaddr\_in client): same as above, but the socket address of the server is given as struct sockaddr\_in object.
- 7) initializeClient(unsigned int peerAddrCoded, unsigned int peerPort): same as above, but the address and port of the recipient are given as unsigned ints.
- 8) writeToSocket(char\* buffer, int maxBytes): it writes up to *maxBytes* from the *buffer* pointer onto the socket. To invoke this, either one version of initializeClient() or readSocketXX() must be invoked (otherwise it will not do its function and will return -1). The recipient of the message is the latest to be designated with initializeClient() call or the latest to have sent a message received by the socket via readFromXX() call, whichever latest.
- 9) writeToSocketAndWait(char\* buffer, int maxBytes, int microSec): same as above, but the socket waits for a total of *microSec* microseconds before returning.
- 10) readSocketWithBlock(char \* buffer, int maxBytes): the socket blocks until it receives a message, storing in *buffer* a maximum of *maxBytes*. It returns the number of bytes received, and -1 on error.
- 11) readSocketWithTimeout(char\* buffer, int maxBytes, int timeoutSec, int timeoutMilli): same as above, but it returns control after blocking for *timeoutSec* seconds and *timeoutMilli* milliseconds. It returns -1 on timeout.

- 12) `readSocketWithNoBlock(char* buffer, int maxBytes)`: same as (9), but the socket does not block until it receives a message. If no message was found about invoking the method, it returns -1.
- 13) `getMyPort()`: return the port number the `UDPSocket` is bound to. If the socket is not yet bound, it returns -1.
- 14) `getPeerPort()`: number of the latest client/sender to the socket. If no such entity exists, it returns -1.
- 15) `getSocketHandler()`: returns the descriptor of the socket.
- 16) `~UDPSocket()`: the destructor. It closes the socket.

### The Message Class:

From an object-oriented point of view, a `Message` object represents the unit of communication between servers and clients conforming to the fundamental principle in distributed systems that communication is done primarily through message passing. For our class, there are two types of messages, namely, a Request and a Reply, Each message has an operation number, RPC identifier to clearly specify the function performed, and the remote procedure called. The `Message` class has the following methods implemented:

- 1) `Message (int operation, void * p_message, size_t p_message_size, int p_rpc_id)`: The class constructor as perceived and used by the sender. It takes all the info needed for a message such as message data, `rpc_id`, etc.
- 2) `Message (char * marshalled_base64)`: The class constructor as perceived and used by the receiver. It takes a data stream in Base64 format and unmarshals it to its original data contents.
- 3) `marshal()`: converts the message data into a Base64 encoded data stream encapsulating message parameters (e.g. message request, `rpc_id`, etc.) to be transmitted into the network. Base64 was preferred over other encoding systems like UTF-8 because it basically solves the problem of null characters so they don't get misinterpreted by the recipient. Also, they solve the problem of special (or control) characters altogether making no room for unexpected misinterpretations on the recipient side. However, of course, it comes at a cost of memory.
- 4) `getOperation()`: Returns message operation.
- 5) `getRPCId()`: Returns message remote procedure call ID.
- 6) `getMessage()`: Returns a pointer to a copy of the `Message`.
- 7) `getMessageSize()`: Returns message size.
- 8) `getMessageType()`: Returns message type.
- 9) `setOperation (int _operation)`: Sets `Message` operation.
- 10) `setMessage (void * msg, size_t msg_size)`: Sets `Message` contents (without marshalling).
- 11) `setMessageType (MessageType msg_type)`: Sets type of `Message` (i.e. Request or Reply).

- 12) `base64_encode (char* msg, size_t message_size)`: Static private helper method to encode messages in Base64 during the marshalling process.
- 13) `base64_decode (char* encoded_string, size_t message_size)`: Static private helper method to decode marshalled messages in Base64 during the unmarshalling process.
- 14) `~Message()`: Destroys the message and frees its memory.

Message Schema:

Message Type (1 Byte)	RPC_ID (4 Bytes)	Operation (1 Byte)	Message Size (4 Bytes)	Total Number of Parts (4 Bytes)	Part Index (4 Bytes)	Marshaled Base64 Message (The remaining bytes)

### The Server Class:

Once an object is instantiated from the Server class, it uses a public member function to accept exactly one request and, for exercise 1, replies with the same message content as a message of type reply.

- 1) `Message* getRequest()`: used by the class to block listen to its socket waiting for request type message. In the case of returning an error by the server socket or receiving a message that is not of type request, the function will return a null pointer.
- 2) `Message* doOperation(Message* _received)`: used by the class to pass a message of type request and returns a message depending on the operation requested by message “\_received”. For exercise 1, this function always returns a pointer to the same message except in the case of receiving a message with the string content “q”, this will result in the termination of the server.
- 3) `void sendReply(Message* _message)`: a function used by the class to send reply message. It takes the output of `doOperation` and changes its type to reply then attempts to send it using the socket member of the server class. In the case of the socket write function returning an error code the function will report the existence of an error to the console. It uses `sendPartitioned`.
- 4) `Server(char* _listen_hostname, int _listen_port)`: the default and only constructor for the Server class, it takes the IP address with an agreed port number for the server to listen to, both parameters will be used on the client side.
- 5) `serveRequest()`: a void type function that is used to listen and serve exactly one message of type request. If it receives a reply, it assigns it to the thread which declared that it was waiting for this reply.

- 6) bool isRunning(): a function that return a bool which is the state of the server, used to terminated the ServerMain depending on the values sent to the server.
- 7) ~Server(): the destructor of the object, it's only job, currently, is to delete the socket member of the class.
- 8) void sendPartitioned(Message \*\_message): this method performs the writing to the socket. If the send message is too big (see above for details) it is partitioned into packets and are sent one by one.
- 9) Message \*processPart(Message \* msg, unsigned int addr, int port): This is the reconstruction method for the partitioned messages.
- 10) bool idempotent (int operation\_id): returns whether or not the operation denoted by the operation id is idempotent.

### **The Peer class:**

The Peer class encapsulates the peer functionalities, and it also inherits server functionalities (hence, it both sends and receives). It has an instance of the UDPClientSocket class, which implements the UDP client.

- 1) Peer(char \*\_myHostname, int \_myPort, char\* \_shostname, int \_sport): The peer constructor receives the hostname and the port number for the peer process, as well as the hostname and the port number for the central service. (You can add \_myPort = 0 to let the system assign you an empty port number).
- 2) Peer \* execute (Message \*\_message): The execute method receives a message which is transmitted for remote execution, and keeps repeating the send until a reply arrives.
- 3) ~Peer(): Client destructor.
- 4) void login(string username, string password): performs login on the central service.
- 5) void signup(string username, string password): performs signup on the central service.
- 6) void getIP(string otherPeer): retrieves the IP address of a different user.
- 7) void getPreviews(): fetches image previews from the central service.
- 8) void getUserPreviews(string otherpeer): fetches image previews for a particular user
- 9) void uploadImagePreview(string imageName, string imagePath): uploads an image owned by the peer as a preview to the central service.
- 10)void requestImage(string otherpeer, string imageName, int quota): requests an image with a certain quota from its owner.
- 11)void requestImageQuota(string imageName, int quota): requests the owner to set a new quota for the image currently granted.
- 12)void setImageQuota(string otherpeer, string imageName, int quota): sets the value of the quota given to another user on an owned image.

## The broker class (central service)

The broker object is the central service. It runs on a separate, independent process and it serves requests for all peers.

- 1) `Broker(char *_listen_hostname, int _listen_port)`: constructor, takes the hostname and the port number the central service operates on.
- 2) `void loadAuthDB()`: loads the file containing the user credentials.
- 3) `void loadPreviewDB()`: loads the file containing the details of the uploaded preview images.
- 4) `Message *doOperation(Message *_received, IP user_ip, Port user_port)`: this is the central service equivalent of do operations, which satisfies requests from peers.
- 5) `int updateLastActive(Token _token)`: documents the last transaction a peer has done.
- 6) `bool validToken(Token token)`: checks if the provided token was valid.
- 7) `Token Login(string username, string password)`: satisfies login requests, and returns session tokens to peers.
- 8) `int Register(string username, string password)`: satisfies register requests for peers.
- 9) `Address getUserAddr(string username)`: returns the machine address and port number of the user.
- 10) `int uploadPreview(string username, string title, Image* preview)`: manages the uploaded preview by a user.
- 11) `vector<Image*> getUserPreviews(string username)`: returns all the previews uploaded by peers
- 12) `vector<Image*> getPreviewFeed()`: creates a feed of a max 2 preview images per user.

## ParseUtil

It is a library containing useful methods for parsing and unparsing messages into RPC parameters and vice versa.

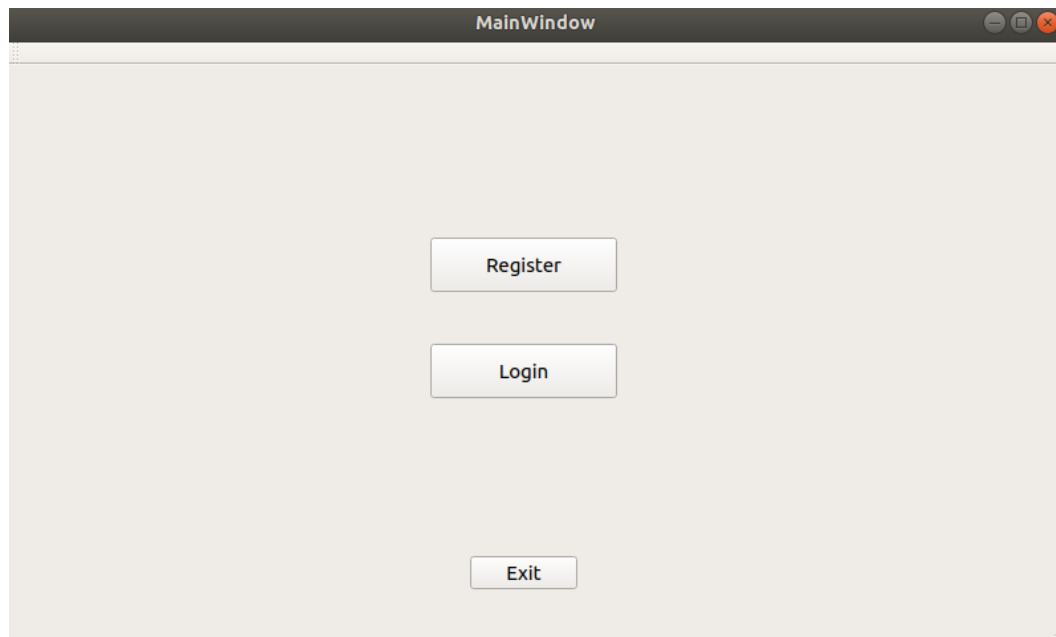
## Image

- 1) `Image(vector<uint8_t> defImage, vector<uint8_t> _content, string _owner, unsigned int _quota)` : takes a default image, desired image, owner and quota, and inserts the latter three into the default image (denoted as codified).
- 2) `Image(vector<uint8_t> _codified)` : it takes a default image and extracts the desired image and owner and quota details.
- 3) `Image(vector<uint8_t> _content, string _owner)` : constructor that does not perform special operations.
- 4) `vector<uint8_t> getContent()` : returns the desired image
- 5) `vector<uint8_t> getCodified()`: returns the default image, with the desired image and the metadata inserted.

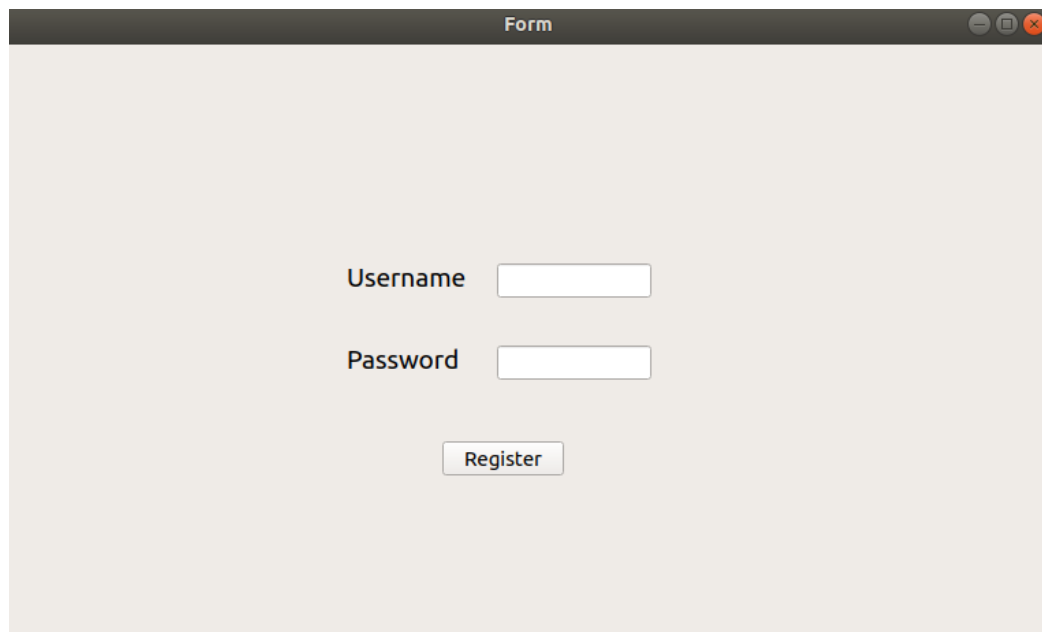
- 6) `string getOwner()`: returns the owner
- 7) `unsigned int getQuota()` : returns the quota
- 8) `static char* vectorToCharPtr (vector<uint8_t> v)` : converts a vector of bytes to char pointer
- 9) `static vector<uint8_t> charPtrToVector (char * c, int len)` : converts a char pointer to a vector of bytes.
- 10) `static vector<uint8_t> stringToVector (string str)` : converts a string to a vector of bytes.
- 11) `static vector<uint8_t> readImage (string fileName, int& sz)` : reads the image file and records its size.
- 12) `static void writeImage( string fileName, vector<uint8_t> bytes )`: writes the image file with the given name.
- 13) `void setTitle(string _title)`: sets the title
- 14) `string getTitle()`: returns the title
- 15) `size_t getSize()`: returns the size

# Graphical User Interface

The GUI for our project consists of multiple screens. The initial page allows the user to either sign up or login, as shown:

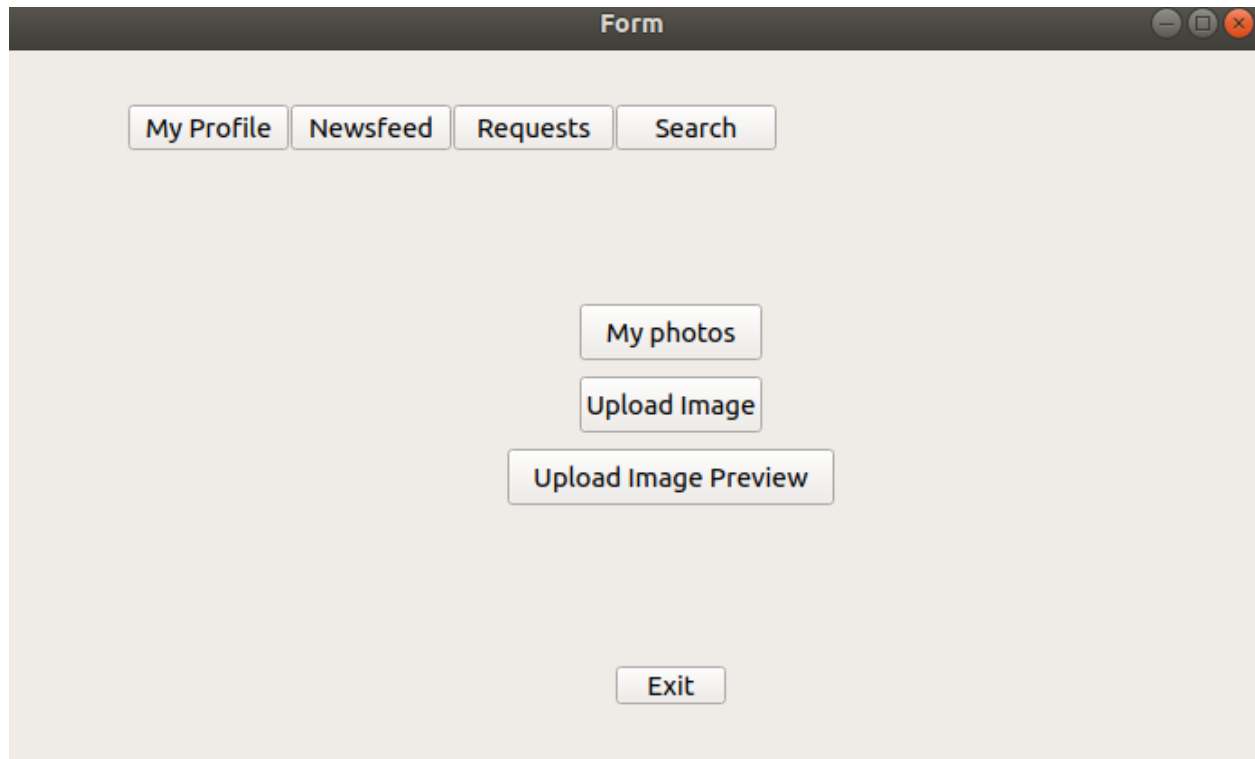


Upon clicking on register:



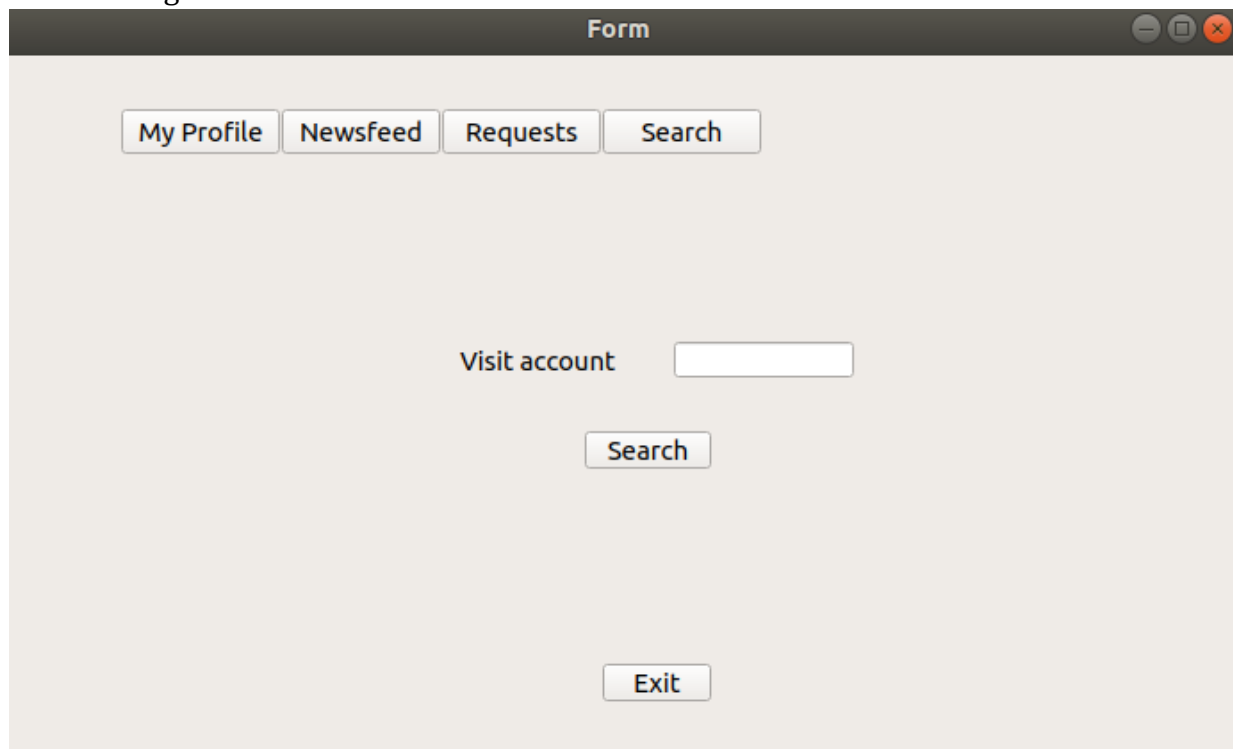
Note: if the user clicked on login, they'd be directed to similar screen, although the functions are different.

On successful login and signup, the dashboard is open, allowing different interfaces as follows:



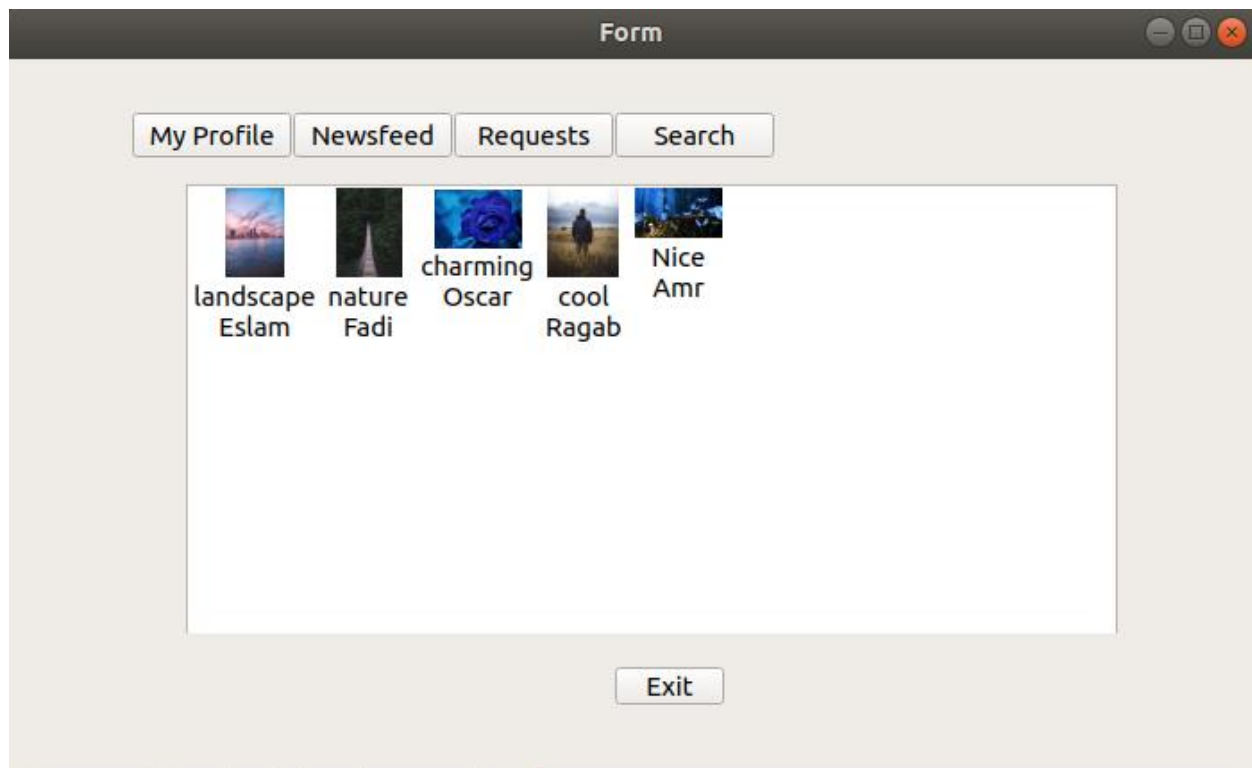


On initiating search:



The screenshot shows a web application window titled "Form". At the top, there are four buttons: "My Profile", "Newsfeed", "Requests", and "Search". Below these, there is a section labeled "Visit account" followed by a text input field. Under the input field is a "Search" button. At the bottom of the form is an "Exit" button.

On viewing the feed, which displays preview images previously uploaded to the central service:



The screenshot shows the same "Form" window, but now displaying a grid of five image thumbnails. Each thumbnail has a caption below it. The thumbnails and their captions are:

- landscape Eslam
- nature Fadi
- charming Oscar
- cool Ragab
- Nice Amr

At the bottom of the grid area is an "Exit" button.