Message Format:

Message is a dictionary with common hierarchical keys “Type” and “Spec”

1. “Type”=“Auth”
    1. “Spec”=“Syn” //Server asks for authentication details
    2. “Spec”=“Ack” //Server responds with valid user
    3. “Spec”=“Nack” //Server responds with inValid user
    4. [No Spec] “Username”=username and “Password”=password// user-details as sent by Client
2. “Type”=“ClientStatus”
    1. “Spec”=“List” //list of online users sent by the server to a newly connected client
    2. “Spec”=“Join” //details of the newly connected client sent by server to rest of the online clients
    3. “Spec”=“Exit” //details of the recently exited client sent by server to rest of the online clients
3. “Type”=“Forward”
    1. [No Spec]
4. “Type”=“Receipts”
    1. “Spec”=“Sent”
    2. “Spec”=“Delivered”
    3. “Spec”=“Read”
5. “Type”=Group”
    1. “Spec”=“Form” //message sent when a group is formed
    2. “Spec”=“Multicast” //messages sent by the users in the group
6. “Type”=“User_Query”
    1. “Spec”=0 //Required User not found
    2. “Spec”=1 //User Found
7. “Type”=“File_Transfer”
    1. “Spec”= Number of chunks the file was divided in to send

Server Side:PYTHON Based

It's just a single python file "server.py" which is responsible for forwarding of messages between clients, authenticating them and storing some of the info useful at the client side.

Please install the following modules to run this:

- socket
- datetime
- threading
- json
- ldap3
- _thread

Functions/Variables Used

- clientList-> List of active users
- buff -> list of messages sent by active users but not yet received by an inactive one
- serversocket-> The socket listening to new users
- Function:send_to_client(client,message)t-> Responsible for sending the message to a given client class Instance
- For each new user a thread is spawned which runs a function handle_all with an instance of class
- Client. The instance just has the clientsocket and its address.
- Function: handle-all-> It first sends a Syn and gets user's credential back. If those check out by the ldap server then it sends an Ack followed by a list of active clients(clientList variable), It also sends to the active clients that this new user has joined. Then it iterates over the messages that were collected in buff for this user and sends it to him.

All messages apart from "Auth" {Syn and Ack} go through process_message function before sending.

- make_message(essence,spec,info) ->
  Responsible for construction of messages whose source is the server:
  "Auth" , Receipts{Sent and Delivered}
- process_message(x)->
  Its sees the type of message and determines what to do with it. For messages of type forward it is seen if the receiver is active or not. Then if the receiver is active a "delivered" receipt needs to be sent to the sender and "sent" otherwise.
  In case of the "sent" one the message is appended to the buff.
  Sending these involves processing them first as mentioned above, So a second case of message type being "Receipts" is made.
  We fix that the "sender" and "receiver" fields of Receipts is same as the one corresponding to the original message. So the receipt is simply forwarded to message["From"] of the Receipt. For that too it is checked if the message["From"] is active or not.
  Messages of type "Group" are same as "Forward" just that the message["To"] is gonna be a list. Same holds for message type file_transfer.

Identify_messages(message)-> Is used to take care of correct parsing of messages by identifying boundaries. Since sockets run over TCP which is a byte oriented protocol we check for each message the number of messages it may have and also check if some message is incomplete. In that case the a prefix is stored for that thread.

Client Side:JavaScript Based

Built on Electron(https://electron.atom.io), which is a platform for building Desktop apps using HTML, CSS and JS

Please install npm (Node.js package manager) to build the client code.

Built instructions:

- Go to src/client
- Do “npm install”
- Add the server IP and Port in client.js file in js directory
- Do “npm start”

Note: This package is just being used for GUI purposes. All implementation pertinent to exchange of messages has been COMPLETELY handled by us.
Upon, “npm start” a window will appear which is being rendered by index.html. This window will take in Username and Password and forward it to the next html page/window where the socket part{the fun :P} begins.
So chat.html will be rendered first with the acquired username and password.
Two scripts are run at this page: client.js, chat.js

Chat.js is more or less just for rendering and display.
Most of the networking part is done in client.js. The username and password entered at the login page were saved using electron framework and are used here.
Please understand that it isn’t possible/fruitful to explain how the GUI is done via this medium.So we will restrict ourselves to client.js only
File_Transfer Implementation is being explained later. More or less it has no overlap with these functions.
The socket instantiated here will connect to the server and wait for a “SYN” upon which it send its authentication details to the server. If it’s rejected (“Nack”) then the login page is rendered with a note otherwise the chat.html page is rendered.
The user now gets a list of active users which is added to the list in the left pane and also gets the messages that were sent to it when it was inactive.
So, Function process_message(message)

We maintain a variable currentUser and another all_messages which signify the friend that the client here is talking to currently and all messages stored as a dict from a friend to a list of exchanged messages. If the message is of type forward its added to all_messages. If at that

point of time the sender is current user a read receipt is sent otherwise the count on the tab of the sender is increased by one followed by a Notification.
If the message is of type “Receipts” the “Status” {a new field absent in the original message} is updated.
All the processing of group messages is exactly same since we represent a group as a name and store a dict from group name to list of actual users{gname_members}.
This list is created when a message of spec “Form” is received . NO Receipts for group messages are sent. The sender is rendered appropriately. handle_new_message function is responsible for the GUI portion.

Function -> resolve_message:
Client side function for handling byte stream nature of TCP, like identify_messages(prefix,message) in the server side

Used to send a completely formed message. If Message is of type forward then it is also added in the all_messages list of the currentUser

Function -> make_message:

This function is responsible for all messages that are sent by the client. {apart from file_transfer} It converts raw messages of different types and specs to fixed defined formats. The function addUser, myFun and updateUserList are just for GUI.

For Auth:
essence=”Auth”
info=Username
info_extra=Password
For CientStatus
essence=”CientStatus”
info=Spec
info_extra=0 //Not Required
For Receipts:
essence=”Receipts”
info=message of which the receipt is created
info_extra=0 //Not Required
For Forward
essence=”Forward”
info=receiver’s name
info_extra=text message to be sent
For Group
essence=”Group”
info=”Form” or “Multicast” depending on the type of message (Refer to the Message Format)
Info_extra = groupid if “Form” message and text message content if “Multicast” message
For UserQuery
essence=”UserQuery”
“Spec”=0/1 depending on knowledge of existence of user

FIle_Transfer:

Process:

For Sending

a) Get the file{File_Selector}
b) Convert it to FIle_URL format with base64 encoding.(send_file}
c) Fix a chunk length{Max size of file excluding informative components}
d) Get the number of chunks
e) Form messages with Spec =number of chunks, an order index{allowing us to accommodate out of order delivery} , From, To and FIle Name

For Receiving:

(a)Make a dict for incomplete files{type:file_name->array of messages}{name:in_process_files}{process_message}

(b) Upon getting a new message of this type get the array corresponding to the file and append the appropriate index. Remaining are set as undefined

(c)Check if number of defined elements are equal to message[“Spec”]{number of chunks}

(d) If yes delete the entry from dict and join the array to get the original file URL.

(e) Render this URL as link, available for download

An exit message is sent before the unload event of window signalling the departure of the user