COMP3234B Computer and Communication Networks
ELEC3443B Computer Networks

Programming Assignment

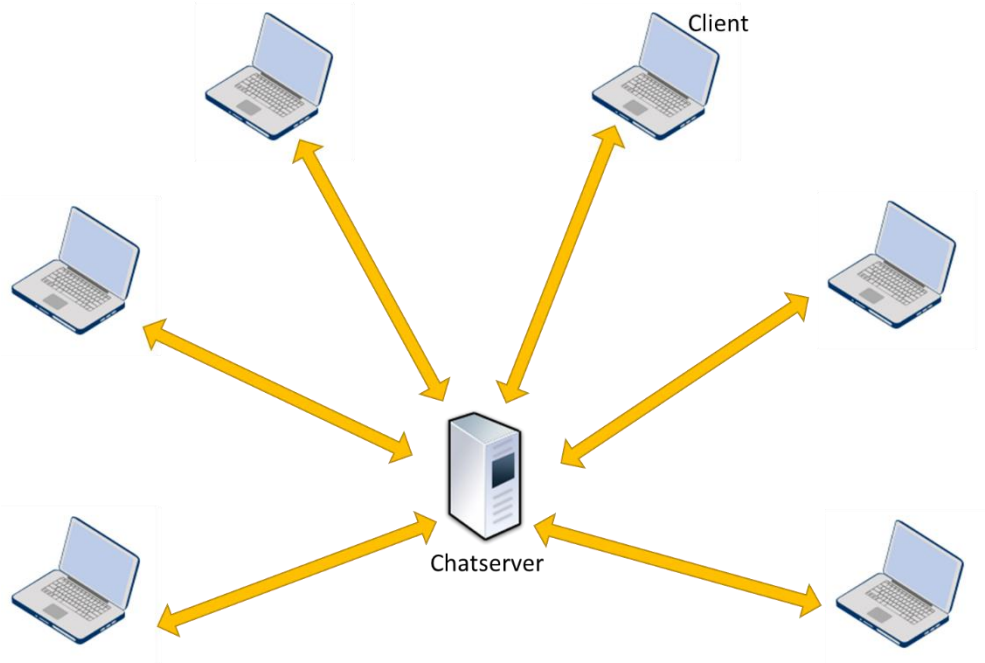
Total 14 points

Due date: 17:00 April 1, 2022

Hand in the assignment via the Moodle System.

Overview

In this assignment, you are going to design and implement a multi-user chat application that supports message exchanges between peers via a Chatserver. To get the list of connected peers in the Chatroom, the chat client program must join the Chatroom managed by the Chatserver. The application supports sending a private message to a connected peer, a message to a group of connected peers, and a broadcast message to all connected peers. Peers may leave or join the Chatroom at any time; the Chatserver must update all connected peers with the updated peer list. To send a private or group or broadcast message, a peer sends the message to the Chatserver and the server sends the message to the target peer(s) accordingly. In this assignment, you will implement a Chat client program - ChatApp.py and the server program - Chatserver.py.



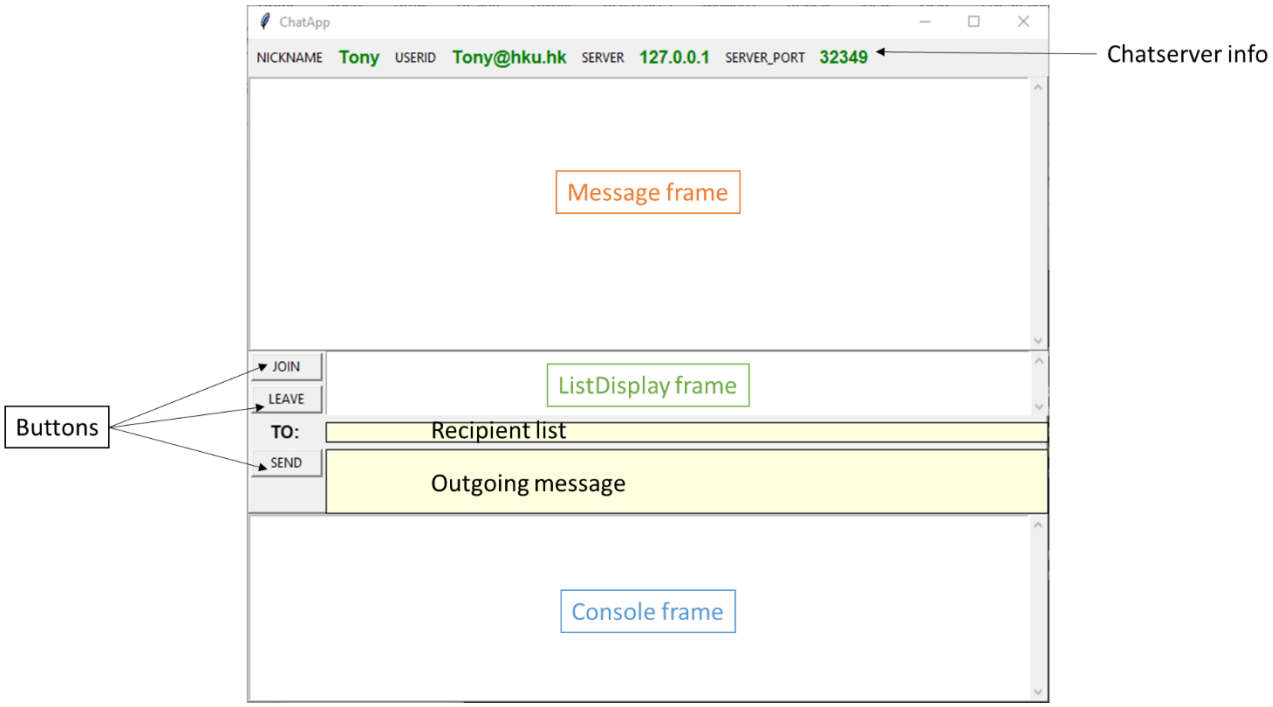
Objectives

1. An assessment task related to ILO4 [Implementation] – "be able to demonstrate knowledge in using Socket Interface to design and implement a network application".

2. A learning activity to support ILO1, ILO2, & ILO4.
3. The goals of this programming assignment are:
    - to get a solid experience in using Socket functions to implement a multi-user chat protocol;
    - to get a good understanding of how a JSON-based networking protocol works as well as how to implement one.

User Interface Design

The ChatApp program makes use of the Tkinter module to implement the UI for handling all user inputs and displaying chat messages. Tkinter is the standard GUI library for Python. You are not required to write the UI, as the UI framework (**ChatApp-UI.py**) will be provided to you. It consists of the necessary initialization Tk code to draw the UI.



Message frame – this is the place where all chat messages are displayed.

- To print a message to the Message frame, the program uses the "chat_print()" method to add the message to **the top** of the Message frame.
- To output the message sent by this client, we use "chat_print(message)"
- To output a received private message from a peer, we use "chat_print(message,'redmsg')"
- To output a group message from a peer, we use "chat_print(message,'greenmsg')"
- To output a broadcast message from a peer, we use "chat_print(message,'bluemsg')

Console frame – this pane is for displaying system/error messages that show what actions/events have happened.

- To print a message to the console frame, the program uses "console_print()" to add it to the top of the console frame, i.e., old contents will be pushed to the bottom of the frame.

ListDisplay frame – this pane is for displaying the current peer list and will be updated by the program whenever the server informs there are changes to the peer list.

- To print the list to the ListDisplay frame, the program uses "list_print()" to update the peer list by replacing the current content with the update content.

Buttons and user input – this is the middle pane with 3 buttons and two text fields for accepting user input.

- Each button has its associated function defined in the UI framework. For example, when the [JOIN] button is pressed, the system runs the "do_Join()" function.
- To read in the input from the "Recipient list" input field, the program uses the "get_tolist()" method, and to read in the input from the "Outgoing message" field, it uses the "get_sendmsg()" method.

Specification of the ChatApp client program

The ChatApp.py program accepts one optional argument:

```
python3 ChatApp.py [config file]
```

If the user does not provide any input argument, the program assumes the use of the default file "config.txt", which should be in the same directory of the ChatApp.py file. The config file contains a JSON string that provides the user's userid and nickname, and the IP address and port number of the Chatserver. Here is an example JSON string:

```
{
  "USERID": "Tony@hku.hk",
  "NICKNAME": "Tony",
  "SERVER": "127.0.0.1",
  "SERVER_PORT": 32349
}
```

The ChatApp client program uses the JSON string to set the user's information and chatserver's addressing information (Note: has already been implemented in ChatApp-UI.py). For this assignment, we assume that each client program has a unique userid and nickname pair.

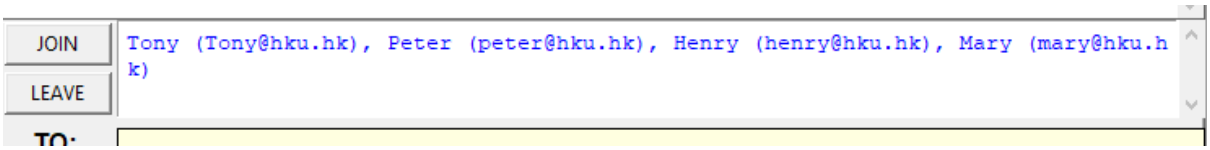
There are in total 3 commands for the user to control the ChatApp client program; their actions are specified as follows.

- [JOIN] button – to join the Chatroom.

  The user cannot send any messages before joining the Chatroom. When no TCP connection exists between the client and Chatserver, the client sends a **JOIN command** to the Chatserver upon the user clicks on the JOIN button. If a TCP connection has already been established, the client should print a message – e.g., "Already connected to server" to the console frame. If the Chatserver is not ready or could not be reached, the client should report to the user **after 2 seconds** by printing an error message to the console frame.

  The Chatserver should respond to the client's JOIN request with the **ACK command**, which indicates whether the request is successful or not. When successfully connected to the Chatserver, the client prints a message to the console frame; otherwise, it prints an error message to the console frame.

Once the client has successfully connected to the Chatserver, it would expect the Chatserver will frequently send the updated peer list to the client by the **LIST command**. Upon receiving an updated peer list, the client displays the information (in the format of NICKNAME (USERID)) of **all peers** in the ListDisplay frame. For example,



More details on the JOIN, ACK, and LIST commands will be covered in the communication protocol section.

- [Send] button – to send a message to the Chatroom.

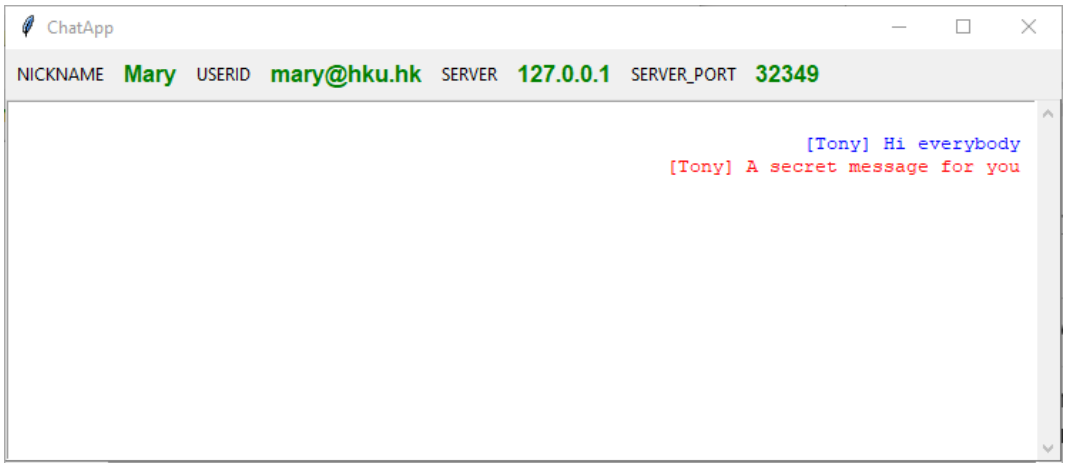
  The client must be connected to the Chatserver before sending a message. When no TCP connection exists between the client and the Chatserver, the client prints a message – e.g. "Connect to the server first before sending messages" to the console frame.

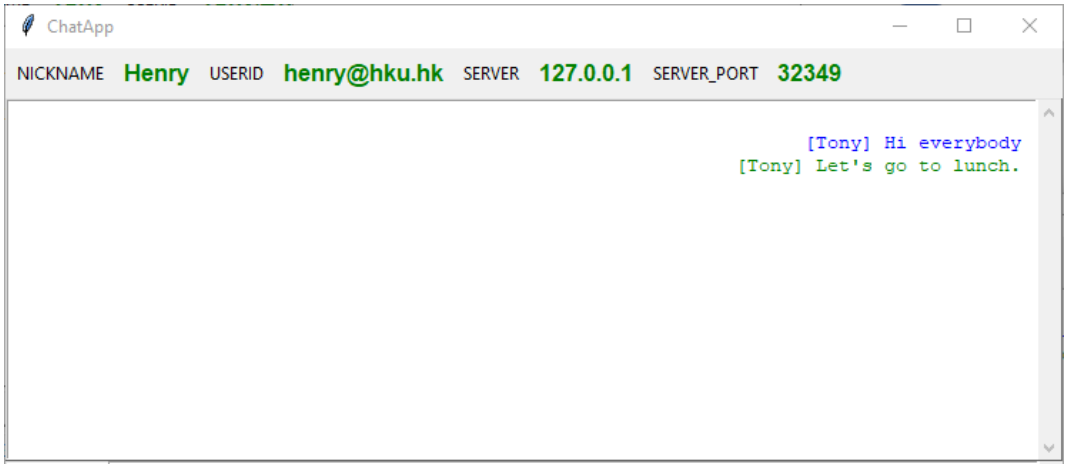
  To send a message, the user must provide the recipient list via the "**TO:**" input field and the to-be-sent message via the message input field. When either input is empty, the client prints an error message to the console frame.

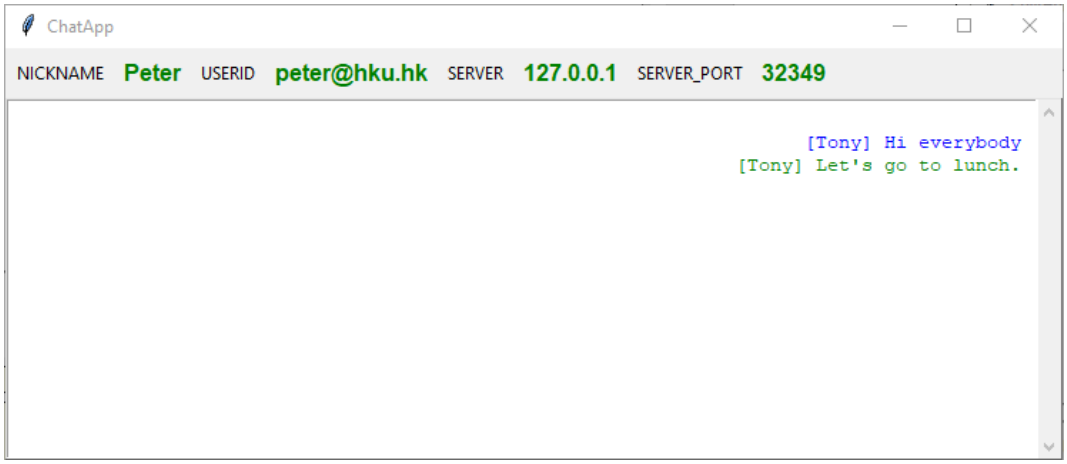
  The user can send a private message, a group message, or a broadcast message. The client uses the **SEND command** to send these messages. To send a private message, the user enters the recipient's nickname into the "TO:" input field; while to send a broadcast message, the user enters the keyword "ALL" into the "TO:" input field. To send a group message, the user enters the list of recipients' nicknames (separated by commas) into the "TO:" field. The program should check whether the inputted nickname is a valid nickname under the corresponding context. For example, if the user tries to send a private message to himself/herself or an unknown peer, the program should print an error message to the console frame. If the user tries to send a group message that includes himself/herself or an unknown peer in the recipient list, the program should remove that user or unknown peer from the recipient list before sending the message to the Chatserver. In addition, it should print an error message to the console frame. To simplify the checking, the inputted names must match the registered nickname of the peers, e.g., tony ≠ Tony, and the keyword "ALL" must be in capital letters.

  After sending the message, the client displays the user's outgoing message to the Message frame. For example, Tony sent a private message to Mary, a group message to Henry & Peter, and a broadcast message. The program adds a header to the message to indicate the nature of the message.

On the other hand, when the client program receives a chat message from the Chatserver (by the **MSG command**), it displays the message to the Message frame. All received chat messages will be aligned to the right with a specific color scheme according to the nature of the message. In addition, the sender's nickname is included in the message. For example,







- [LEAVE] button – to disconnect from the Chatroom.

  Upon the click event, the client closes the TCP connection, releases any resources previously created for this chat session, and prints a message to the console frame. However, if no TCP connection exists, the client does not perform any actions upon the click event.

  The Chatserver would detect the termination of the client connection and remove the corresponding user's information from the peer list. After that, it would send the updated peer list to all connected peers.

  If the user wants to join the Chatroom again, he/she simply clicks on the JOIN button to reconnect to the Chatroom.

Implementation hint

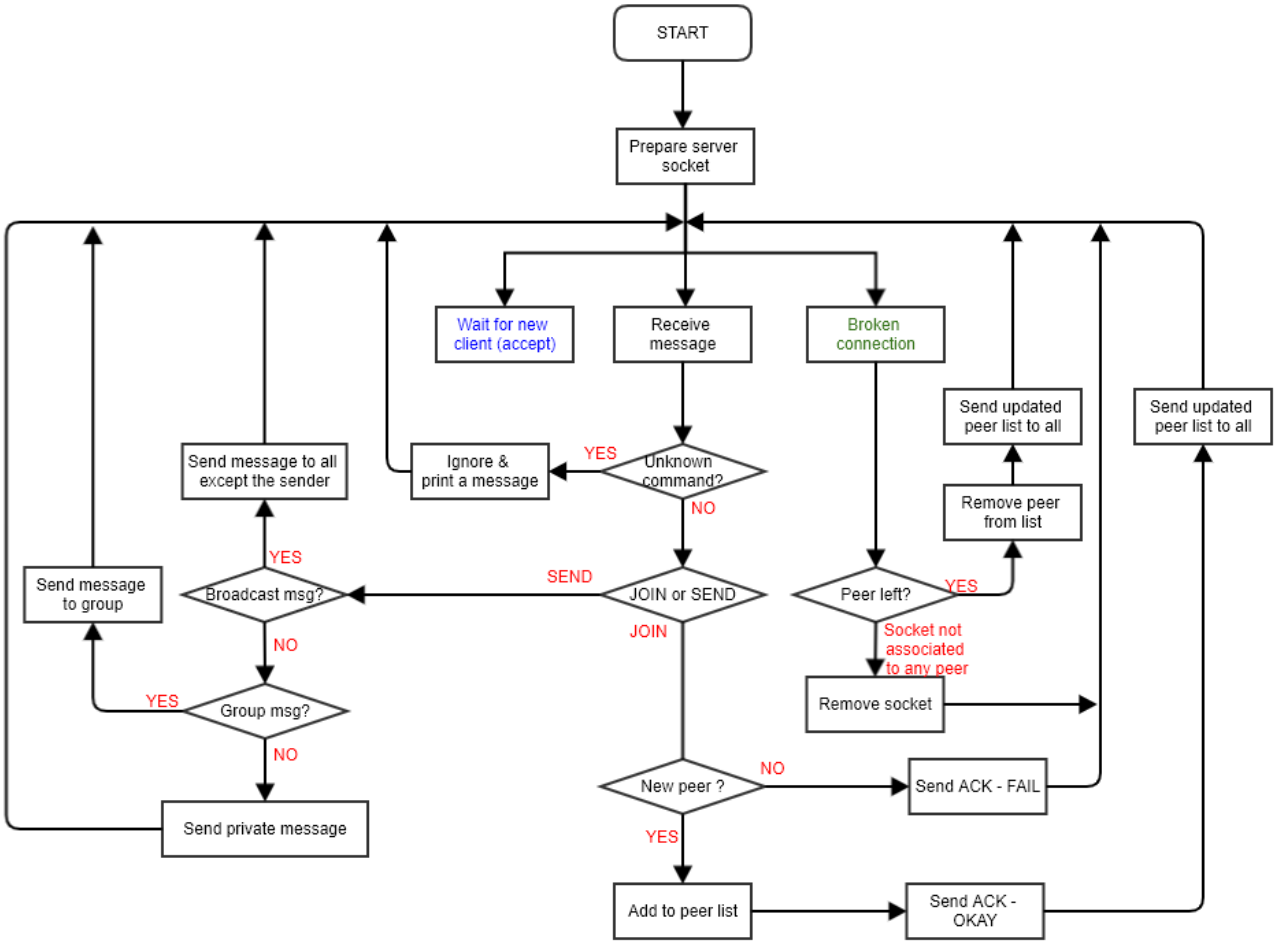
1. The client program uses Tkinter to render the UI and handle all user's inputs. As all Tk python programs, by default, are sequential single-threaded programs. If the client process is working on a blocking I/O operation, e.g., recv(), the whole process will be blocked and the UI will be frozen. Unfortunately, using select() may not be possible here as we do not know which file descriptor(s) the UI is/are using. Therefore, a better way is to use a python thread to handle all asynchronous recv() operations for the client process.

2. There is no specific requirement on the format and the contents of the messages displayed in the console frame. We suggest you print an error/log message for each significant event to the console frame.

Specification of the Chatserver server program

The behavior of your *Chatserver* program MUST match with the behavior of the ChatApp client program. Below shows the flowchart of the *Chatserver* program, which you can use as the blueprint to develop your server program.

The server program accepts one optional input argument – *listen_port_number*. If not providing the argument, the server should use the default port number XXXXX (a port number under your assigned port number range). Once it starts, it runs forever. To terminate the server program, use Ctrl + C (or Ctrl + Break for Windows) to terminate the Chatserver program.



As the server must handle multiple client connections and interact with multiple peers at the same time, you can use a single-threaded process with the select I/O operation or multi-threaded process to handle the concurrency. If you are using a single-threaded process, the sample implementation of Workshop 2 should be a good framework for implementing this Chatserver.

When the server receives a JOIN request, it checks whether the requesting peer has already registered (by the peer's userid) with the server. If this is a duplicated request, the server sends the ACK command with the FAIL type to the client; otherwise, it adds the peer to the peer list and sends the OKAY ACK command to the client. Afterward, it broadcasts the updated peer list to all registered peers (by the LIST command).

The Chatserver never closes any TCP client connections and the termination of TCP connections is always initiated by the clients. When the Chatserver detects a broken TCP connection, it checks whether this connection is associated with a registered peer. It should release the socket, remove the peer from the peer list, and broadcast the updated peer lists to all registered peers. If the connection is not associated with any peer, it simply releases the socket without touching the peer list.

To send the chat messages to the peers, the Chatserver process uses the **MSG command** to deliver those private / group / broadcast messages.

Communication Protocol

All communications happened between ChatApp and Chatserver are in text-based JSON format. To make sure all parties can communicate successfully, ***you are not allowed to modify, add, or change the protocol***.

JavaScript Object Notation (JSON) is a standard text-based format for representing structured data based on JavaScript object syntax. It is commonly used for transmitting data in web applications. Protocol data is converted to JSON string before sending, carried by TCP, and turned back to protocol data format at the other end. Many programming environments (including Python) come with functions to convert information between JSON string and structured data.

To use JSON in Python, you must import the json module to your program. To convert a JSON string x to the Python structured data (which is a Python dictionary), we use `json.loads(x)` for the conversion. To convert a Python object obj (e.g., dictionary, list, tuple) to the JSON string, we use `json.dumps(obj)`. For example,

```
import json

# a JSON string:
x = '{ "name":"John", "age":30, "city":"New York"}'

# parse x:
y = json.loads(x)

# the result is a Python dictionary:
print(y["age"])    # that prints an integer 30
print(y["name"])   # that prints the string John

# a python list
obj = ['apple','orange','mango']

# encode obj:
jstr = json.dumps(obj)

print(jstr)        # that prints ["apple", "orange", "mango"]

# parse jstr:
nobj = json.loads(jstr)

print(nobj[1])     # that prints orange
```

Here is the set of commands exchanges between the ChatApp and the Chatserver. One of the advantages of using JSON is that the order of the fields in the message is not important.

Commands
A client sends a request to **JOIN** the chatroom. The JOIN command consists of three fields: • CMD - should have the value JOIN • UN - user's nickname • UID - user's userid e.g.,

```
{"CMD": "JOIN", "UN": "Tony", "UID": "Tony@hku.hk"}
```

The server should respond with the ACK command

The server responds with an **ACK** command for each JOIN request. It consists of two fields:

- CMD - should have the value ACK
- TYPE - either OKAY or FAIL

For example, the server accepts the JOIN request:

```
{"CMD": "ACK", "TYPE": "OKAY"}
```

Another example, the server rejects the JOIN request:

```
{"CMD": "ACK", "TYPE": "FAIL"}
```

The server sends the updated peer list to all registered peers by the **LIST** command. The LIST command consists of two fields:

- CMD - should have the value LIST
- DATA - is a list data type that keeps the peer list. Each peer is structured as a dictionary type with the UN and UID fields.

For example, this is the LIST command with only one peer in the peer list.

```
{"CMD": "LIST", "DATA": [{"UN": "Mary", "UID": "mary@hku.hk"}]}
```

Another example, this is the LIST command with two peers in the list.

```
{"CMD": "LIST", "DATA": [{"UN": "Mary", "UID": "mary@hku.hk"}, {"UN":
"Henry", "UID": "henry@hku.hk"}]}
```

The peers do not need to respond to the LIST command.

The client uses the **SEND** command to send a private / group / broadcast message to the Chatserver. The SEND command consists of four fields:

- CMD - should have the value SEND
- MSG - contains the chat message
- TO - is a list data type that contains the userids of the recipients.
- FROM - contains the userid of the sender

Here is an example of a private message from Mary to Tony.

{"CMD": "SEND", "MSG": "Where are you?", "TO": ["Tony@hku.hk"], "FROM": "mary@hku.hk"}

Another example shows a group message from Tony to Peter and Mary.

{"CMD": "SEND", "MSG": "Do you know Henry\'s phone number?", "TO": ["peter@hku.hk", "mary@hku.hk"], "FROM": "Tony@hku.hk"}

The last example shows a broadcast message from Henry to ALL. Please note that we use an empty list to represent targeting the message to ALL.

{"CMD": "SEND", "MSG": "Dear all, you can reach me via 91176842", "TO": [], "FROM": "henry@hku.hk"}

The server does not directly respond to the sender; instead, it reacts by sending the MSG command to the target peer(s).

The Chatserver uses the **MSG** command to send a private / group / broadcast message to target peer(s). The MSG command consists of four fields:

- CMD - should be the value MSG
- TYPE - indicates the type of message; either ALL, GROUP, or PRIVATE
- MSG - contains the chat message
- FROM - contains the userid of the sender

Here is the example of a private message from Mary to Tony.

{"CMD": "MSG", "TYPE":"PRIVATE", "MSG": "Where are you?", "FROM": "mary@hku.hk"}

This shows a group message from Tony to Peter and Mary.

{"CMD": "MSG", "TYPE":"GROUP", "MSG": " Do you know Henry\'s phone number?", "FROM": "Tony@hku.hk"}

This shows a broadcast message from Henry to ALL.

{"CMD": "MSG", "TYPE":"ALL", "MSG": " Dear all, you can reach me via 91176842", "FROM": "henry@hku.hk"}

The peers do not need to respond to the MSG command.

To simplify the implementation, we assume all messages between the ChatApp and the Chatserver are shorter than 1000 bytes.

Resources provided

You will be provided with the following files:

1. The template of the ChatApp.py - ChatApp-UI.py. You have to implement the do_Join(), do_Send(), and do_Leave() functions to complete the ChatApp.py program.
2. The sample implementation of Workshop 2 can be a good starting point for the Chatserver.py (if you prefer using select() for the I/O). However, you are welcome to implement the concurrent server by threading.
3. A set of config files - config.txt, config1.txt, config2.txt, & config3.txt. Using these four config files, you can start 4 peers with the specified information.
4. A set of script files - start.ps1, start-linux.sh, start-OSX.sh, & start-OSX-tab.sh. These script files help you to quickly start four peers (with the corresponding config file) and the Chatserver on a specific OS platform.

Computer Platform to Use

For this assignment, you can develop and test your Chat application (ChatApp.py and Chatserver.py) on any platform installed with **Python 3.6 or above**.

Submission

Name the chat client program to ChatApp.py and the server program to Chatserver.py. Submit the programs to the Moodle assignment submission page on or before **April 1, 2022 (Friday) at 17:00**.

The Format for the documentation

1. At the head of the submitted programs, state the
   - Student name and No.:
   - Development platform:
   - Python version:
2. Inline comments (try to be informative so that your code could be understood by others easily)

Grading Policy

As the tutors will check your source code, please write your program with good readability (i.e., with **good code convention and comments**) so that you will not lose marks due to possible confusion.

Documentation	• Include the required program and student's info at the beginning of the program (-0.5 points)

ChatApp program	• [JOIN] button (2 points)
	  ○ Can successfully join the Chatserver and detect a failure in joining
	  ○ Allow user to rejoin the Chatserver if the connection to Chatserver is broken
	  ○ Report to the user when the Chatserver is not reachable/available after 2 seconds
	• [SEND] button (3 points)
	  ○ Check the TO: and message fields
	  ○ Handle private, group, & broadcast messages
	  ○ Display the message
	• [LEAVE] button (0.5 points)
	  ○ Close the TCP connection
	  ○ Allow user to rejoin the Chatserver
	• Display peer list (1 point)
	  ○ Can display the updated peer list upon receiving the LIST command
	• Display received chat messages (1.5 points)
	  ○ Can display those received messages with the correct display setting

Chatserver program	• Connection and peer management (3 points)
	  ○ Accept any TCP client requests
	  ○ Able to detect broken connections
	  ○ Accept/reject peer join request and send updated peer list after accepting the peer
	  ○ Able to remove a specific peer when its TCP connection is broken and send updated peer list to all
	• Handle chat messages (3 points)
	  ○ Handle private, group, & broadcast messages
	  ○ Correctly send the messages to target peer(s)

Plagiarism

Plagiarism is a very serious academic offence. Students should understand what constitutes plagiarism, the consequences of committing an offence of plagiarism, and how to avoid it. **Please note that we may request you to explain to us how your program is functioning as well as we may also make use of software tools to detect software plagiarism.**