# Spring 2023 - CSEE 4119 Computer Networks

# Program ent 1 - Simple Chat Application

f. Gil Zussman

/2023, 23:59PM, EST

## 1 Introduction

The objective of this programming assignment is to implement a *simple* chat application with at least *4* clients and a server using UDP. You are asked to create one program. The program should have two modes of operation, one is the client, and the other is the server. The client instances communicate directly with each other. The server instance is used to set up clients and for book-keeping purposes. The server will also broadcast channel messages to clients within a predefined communication channel (group chat). The functionalities and specification of each program are described in detail below. ***Please start early and read the entire homework before you start!!***

## 2 Functionalities

The complete chat application can be broadly classified into four functions outlined below. Each function involves either the client part or the server part or a combination of the two. The four functions and their respective parts in both the server and the client are explained in the following sections.

### 2.1 Registration

For the registration function, the server has to take in a registration or a subscription request from a client. The server needs to be started before the client. The server maintains a table with the IP addresses, port numbers, and nick-names of all clients. This functionality involves both client and server modes.

**Client mode:**

- The client has to communicate with the server using the IP address and the port number of the server [assume all clients by default know the server information].

  `$ ChatApp <mode> <command-line arguments>` : Start the program for server and client (for example: ChatApp -c for client and ChatApp -s for server). The server mode takes one argument: its listening port. The client mode should take four arguments: client name, server IP address, server's listening port number, and client's listening port number.

  `$ ChatApp -s <port>` : Initiates the server process

  `$ ChatApp -c <name> <server-ip> <server-port> <client-port>` : Initiates client communication to the server. Client name is like a username for this chat client. Server IP address should be given in decimal format and the port number should be an integer value in the range `1024-65535`. For example, if the server IP is `198.123.75.45`, the server port is `1024`, the client's port number for listening is `2000`, then the command will be: `$ ChatApp -c client-name 198.123.75.45 1024 2000`. If arguments are taken in a proper format, a prompt like `'>>>'` should be displayed. The application should also be able to perform basic error checking where the IP addresses are valid numbers, and assigned ports are within the range. Otherwise, an appropriate error message should be displayed.

- Successful registration of the client on the server should also display the status message to the client:
  `$>>> [Welcome, You are registered.]`

- Every client should also maintain a local table with information about all the other clients (name, IP, port number, online-status). Every client should update (overwrite) its local table when the server sends information about all the other clients (further detail on this in upcoming section).

- When the table ha[illegible]ed, the client should display the message :
  `$ >>> [Client tab[illegible]`

  There should be t[illegible]ose' as a client:

- Silent leave: Once [illegible]es, the server will not be notified. You can expect that the client will not register a[illegible]rmation after it exits via Silent leave. To exit or close, a client uses `$ >>> ctrl +` [illegible] window that the client is running on (both actions need to be implemented, and the system should not crash).

- Notified leave: De-registers the client, and the de-registration action will be notified to the server. The client status in the server table should be changed to offline. More detailed information is covered in 2.3.

**Server mode:**

- The server process should maintain a table to hold the names, IP addresses, and port numbers of all the clients.

- When a client sends a registration request, it should add the client information (name, IP address, port number, online-status) to the table.

- When the server updates its table it should print a message to the terminal that the table has updated.

- The server should *broadcast* the complete table of active clients to all the online clients so that they may update their local information. This should happen whenever the server updates its table.

- Once the server is offline, it will not come back online again.

## 2.2 Chatting

Once the clients are set up and registered with the server, the next step is to implement the actual chat functionality. The clients should communicate to each other *directly* and must not use the server to forward chat messages. Since it does not involve the server, only the client part is described for the chat function.

**Client:**

- A client should communicate to another client using the information from its local table (including communicating with itself).

  The client should support the following command for sending messages
  `$ >>> send <name> <message>` : This command should make the client look up the IP address and port number of the recipient client from its local table and send the message to the appropriate client (message length should be variable).

- The client which sends the message has to wait for an *ack* and likewise, the client which receives the message has to send an *ack* once it receives the message.

- If *ack* times out (*500 msecs*) for a message sent to a another client, it means the client at the receiving end is offline, and so the message has not been delivered. The client should notify the server that the recipient client is offline, and both the server and client should update their tables.

  The appropriate status messages also need to be displayed for each scenario:
  ```
  $ >>> [Message received by <receiver nickname>.]
  $ >>> [No ACK from <receiver nickname>, message not delivered]
  ```

- The client should keep track of whether it is in a group chat room (see 2.4 below). If the client is in a group chat room it should not print any private messages while it is in the room. See section 2.5.1 for the expected behavior.

- Once the client successfully receives a message, if it is not in the group chat mode (see below), it should display the message on the screen:
  ```
  $ >>> <sender-nic
  ```

## 2.3 De-registratio

This is a book-keeping i active clients. This functionality involves both client and server parts.

**Server:**

- When the server receives a de-registration request from a client, it has to change the respective client's status to offline in the table (do not close or exit the client to change its status to offline).
- When the server updates its table it should print a message to the terminal that the table has updated.

- It then has to *broadcast* the updated table to all the active (online) clients.
- The server then has to send an *ack* to the client which requested de-registration.

**Client:**

- When a client is about to go offline, it has to send a de-registration request to the server to announce that it is going offline.

- The client has to wait for an *ack* from the server within *500 msecs*. If it does not receive an *ack*, the client should retry for *5* times. If it fails all five times the client should display the message:
  ```
  $ >>> [Server not responding]
  $ >>> [Exiting]
  ```
  and exit.

- All the other active clients, when they receive the table from the server, should update their respective local tables (just overwrite the existing table).

  `$ >>> dereg <nick-name>` : This is a de-registration request to the server from the client to go offline.
  You do not need to consider a case in which another client uses the same information to register while the client is de-registered.

- Successful de-registration from the server should display the following status message in the client:
  ```
  $ >>> [You are Offline. Bye.]
  ```

## 2.4 Group Chat

Besides the basic one-to-one chatting, the chat software should also provide the group chat functionality. That is, clients can create, list, join, leave and chat in one group chat. While the clients participate in the group chat, they do not send private messages until they leave the group. See section 2.5.1 for the expected behavior when a client receives a message while in a group chat. Here are the details of these actions.

### 2.4.1 Create

The client can create a group chat which does not exist yet.

**Client:**

- While client is in [illegible]hich means the client is not in a group chat mode), the client can create a group cha[illegible]
- The following com[illegible] process of sending a request to the server to create a new group chat:

  ```
  $ >>> create_grou[illegible]
  ```

- The client which [illegible]wait for an *ack* from the server within 500 msecs. Once *ack* is received from the [illegible]
  - if the ACKed message indicates the group created successfully, display the following message on the client screen:

    ```
    $ >>> [Group <group-name> created by Server.]
    ```

  - if the ACKed message indicates the group already exists, display the following message on the client screen:

    ```
    $ >>> [Group <group-name> already exists.]
    ```

- If the requesting client does not receive an *ack* response from server within time limit, the client should retry for five times. If it fails all five times the client should display the message and exit:

  ```
  $ >>> [Server not responding.]
  $ >>> [Exiting]
  ```

- To enter the group chat, the client that created the chat must also use the join command (see below).

**Server:**

- Upon receiving the message from a client, the server should check whether the group already exists and send an *ack* back to the sender client.
- If the group does not exist yet, the ACKed message should contain extra information indicating this, and display the following message on the server screen:

  ```
  $ >>> [Client <nick-name> created group <group-name> successfully]
  ```

- If the group already exists, the ACKed message should also contain extra information indicating this case, and display the following message on the server screen:

  ```
  $ >>> [Client <nick-name> creating group <group-name> failed, group already exists]
  ```

- The server must also keep a record of which group chats exists, and which clients are in which groups, and print the list when it updates.

### 2.4.2 List All Group Chats

Clients have to know the available group chats before they join.

**Client:**

- While a client is in the normal mode, it can list all available group chats.
- Send the following command to server to list all group chats:

  ```
  $ >>> list_groups
  ```

- The client which sent the message has to wait for an *ack* from the server within 500 msecs. Once *ack* is received from the server, group names also need to be displayed:

```
$ >>> [Available group chats:]
$ >>> <groupA_name>
$ >>> <groupB_name>
$ >>> <...>
```

- If the requesting cl ck response from the server within the time limit, the client should retry five times. If lient should display the message and exit:

```
$ >>> [Server not
$ >>> [Exiting]
```

**Server:**

- Upon receiving a request from a client, the server should send an *ack* together with a list of all existing group names back to the requesting client. Meanwhile, the server should display the following message on the screen:

```
$ >>> [Client <nick-name> requested listing groups, current groups:]
$ >>> <groupA_name>
$ >>> <groupB_name>
$ >>> <...>
```

### 2.4.3 Join

A client must join a chat group before sending messages to the group. A client can join a group chat while it is in normal mode. The group chat must exist. After entering a group, the client will be in the "group chat mode" which is different from the normal mode.

**Client:**

- In order to request from the server to join a group chat, the client should do as follows:

```
$ >>> join_group <group_name>
```

- The client which sent the message has to wait for an *ack* from the server within 500 msecs. Once *ack* is received from the server, based on the response message, appropriate messages should be displayed:
  - If the the client successfully joins an existing group:

    ```
    $ >>> [Entered group <group_name> successfully]
    ```

  - If the group does not exist:

    ```
    $ >>> [Group <group_name> does not exist]
    ```

- If the requesting client does not receive an *ack* response from server within the time limit, the client should retry for five times. If it fails all five times the client should display the message and exit:

```
$ >>> [Server not responding.]
$ >>> [Exiting]
```

**Server:**

- Upon receiving the message from a client, the server should send an *ack* together with an appropriate message which indicates that the client joined the group.
- When a client successfully joins a group, the server should record the user in the group chat and display the following message:

```
$ >>> [Client <nick-name> joined group <group-name>]
```

- If the group does not exist, the client cannot join the group chat. The server should send *ack* with extra information indicating this and display the following message:
```
$ >>> [Client <nick-name> joining group <group-name> failed, group does not exist]
```

- The server should update its record of group members and display any updates when clients join/leave a group.

### 2.4.4 Chat in the Gro[illegible]

Once the client enters th[illegible] it will be in the group chat mode. In this mode, all messages sent and received by the clie[illegible]ting with the prompt:
```
$ >>> (<group_name>)
```

**Client:**

- A client can send messages in the group chat by the *send_group* command:
```
$ >>> (<group_name>) send_group <message>
```

- The client that sent the message has to wait for an *ack* from the server within 500 msecs. Once *ack* is received from the server, the following message should be displayed:
```
$ >>> (<group_name>) [Message received by Server.]
```

- A client who receives a group message from the server, should send an *ack* back to the server and display the received message:
```
$ >>> (<group_name>) Group_Message <nick name(sender client)>: <message>.
```
Note: 'Group_Message' should be a hard-coded string

- If the client (sender) does not receive an *ack* response from server within time limit, the client should retry for five times. If it fails all five times the client should display the message and exit:
```
$ >>> (<group_name>) [Server not responding.]
$ >>> (<group_name>) [Exiting]
```

**Server:**

- Upon receiving a group chat message from a client, the server should send *ack* back to the sender and broadcast the message to all other clients in the sender's group except the sender.

- The following message should be displayed on the server screen:
```
$ >>> [Client <nick-name> sent group message: <message>]
```

- The server should also expect an *ack* from all the active (online) clients (except the sender client) in this group, The server can wait up to 500 msecs to receive an *ack* from all clients.

- If the server does not receive an *ack* response from a client within time limit, the server should remove the client from the group and display the following message:
```
$ >>> [Client <nick-name> not responsive, removed from group <group name>]
```

### 2.4.5 List Group Members

Clients in group chat mode can list all members in the current group.

**Client:**

- While client is in the group chat mode, it can send the following command to server to list all members in the current group:
```
$ >>> (<group_name>) list_members
```

- The client which sent the message has to wait for an *ack* from the server within 500 msecs along with the information about group chat members. Once *ack* is received from the server, members in the group also need to be displayed:

```
$ >>> (<group_name>) [Members in the group <group_name>:]
$ >>> (<group_name>) <clientA-nick-name>
$ >>> (<group_nam[illegible]
$ >>> (<group_nam[illegible]
```

- If the client (sende[illegible] response from server within the time limit, the client should retry for five times. If it[illegible]ent should display the message and exit:

```
$ >>> (<group_nam[illegible]ng.]
$ >>> (<group_nam[illegible]
```

**Server:**

- Upon receiving the message from a client, server should send an *ack* together with a list of the names of all member in the sender's group back to the sender client. Meanwhile, the server should display the following message on the screen:

```
$ >>> [Client <nick-name> requested listing members of group <group_name>:]
$ >>> <clientA-nick-name>
$ >>> <clientB-nick-name>
$ >>> <...>
```

#### 2.4.6 Leave

Clients can leave the group chat back to the normal chat mode.

**Client:**

- Client can type the following command in order to leave the group (this will initiate a message to the server):

```
$ >>> (<group_name>) leave_group
```

- The client that sent the message has to wait for an *ack* from the server within 500 msecs. Once an *ack* is received from the server, the following message should be displayed:

```
$ >>> [Leave group chat <group_name>]
```

  Note after leaving the group chat, messages should no longer display `<group_name>`.

- If the requesting client does not receive an *ack* response from server within time limit, the client should retry for five times. If it fails all five times the client should display the message and exit:

```
$ >>> [Server not responding.]
$ >>> (<group_name>) [Exiting]
```

**Server:**

- Upon receiving the request from a client, server should send an *ack* back to client and remove the user from the group chat at the same time. The server should also display the following message:

```
$ >>> [Client <nick-name> left group <group_name>]
```

- The server should update its record of group members and display any updates when clients join/leave a group.

### 2.5 Special Notes

Instructions of some special scenarios are list here.

### 2.5.1 Private Messages in Group Chat Mode

If the client is in the group chat, they can only send and receive the messages in the group. They cannot send or display received private messages as they do in the normal mode. If the client receives a private message from another client, it should reply *ack*. However, it will not display the message on the screen immediately; instead it will store the message internally. When the client leaves the group chat, all stored private messages will be displayed. For example:

```
$ >>> (GroupA) leave_gr[illegible]
$ >>> [Leave group chat[illegible]
$ >>> ClientB: hello!
$ >>> ClientC: nice to [illegible]
```

Assume the last two priv[illegible]ed while the client was in the group chat mode.

### 2.5.2 Command Scop[illegible]

Since we have two modes: normal mode and group chat mode. Some commands are valid in one mode but not in another. The valid commands in each mode are:

**Normal Mode:**

- `send`
- `dereg`
- `create_group`
- `list_groups`
- `join_group`

**Group Chat Mode:**

- `send_group`
- `list_members`
- `leave_group`
- `dereg`

The client should verify that the commands issued fit the mode in which the client operates.

`$ >>> [Invalid command]` (Normal Mode)

`$ >>> (<group_name>) [Invalid command]` (Group Chat Mode)

## 3 Testing

Before submitting your work, please do **test your programs thoroughly**. Your chat application should *at least* work with

- *One* instance of the program in server mode.
- *Four* instances of the program in client mode.

To start off with you can assume fixed sizes for the client table and extend your implementation to handle dynamic length if you have time. Full points will be awarded only if you handle dynamic length. You must handle business-logic errors such as a user trying to log in with an already connected nickname.

Three simple example test cases have been provided for you. You should also test your program with your own test cases.

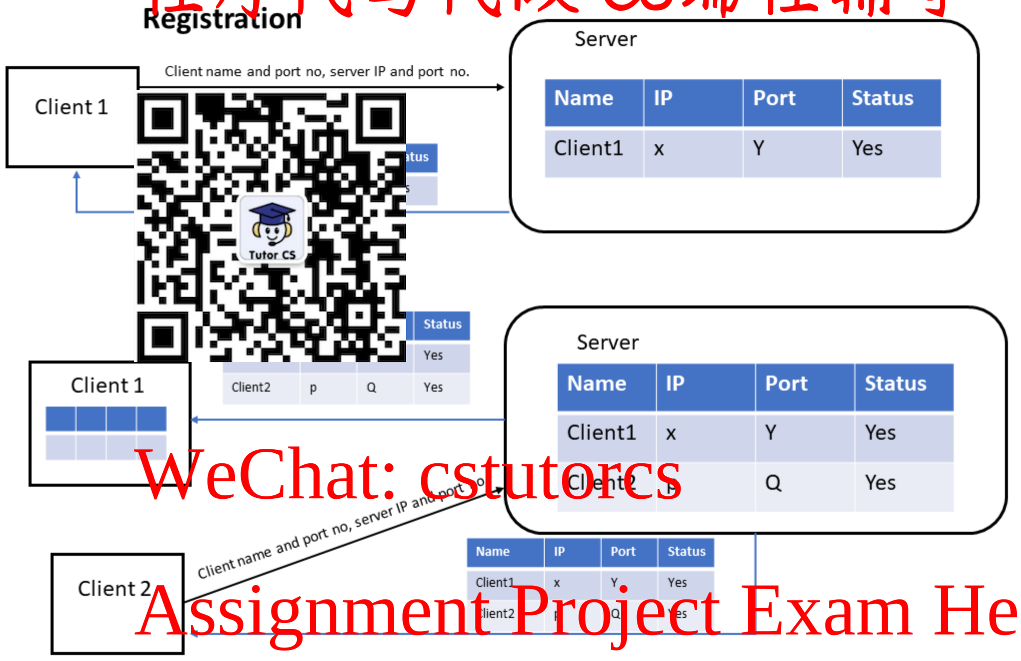
**Test-case 1:**

1. start server
2. start client `x`(the t[illegible]erver to x)
3. start client `y`(the t[illegible]erver to x and y)
4. start client `z`(the t[illegible]erver to x and y and z)
5. chat `x -> y, y->z, ... , x ->z` (All combinations)
6. dereg x (the table should be sent to y, z. x should receive 'ack')
7. chat `y->x` (this should fail, y should display that the message failed)
8. chat `z->x` (same as above)
9. `y, z` : exit

**Test-case 2:**

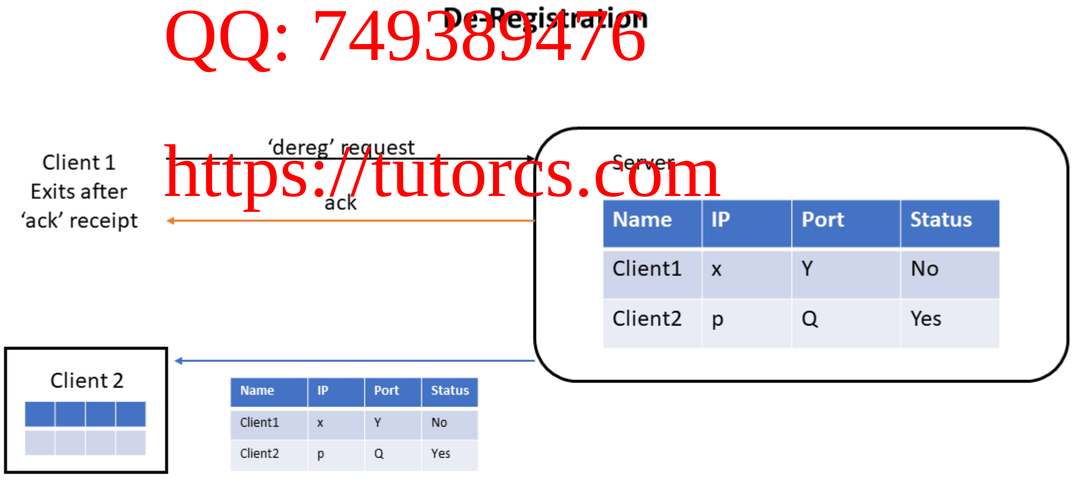
1. start server
2. start client `x` (the table should be sent from server to `x` )
3. start client `y` (the table should be sent from server to `x` and `y`)
4. dereg `y`
5. server exit
6. send message `x-> y` (will fail with both `y` and server, so should make *5* attempts and exit)

The figures below shows the registration process and de-registration process involving two clients. To provide some more *clarity*.


Registration
Client name and port no, server IP and port no.
Client 1
Server
Client 1
Server
Client name and port no, server IP and port no.
Client 2

| Name | IP | Port | Status |
|---|---|---|---|
| Client1 | x | Y | Yes |

| Name | IP | Port | Status |
|---|---|---|---|
| Client1 | x | Y | Yes |
| Client2 | p | Q | Yes |

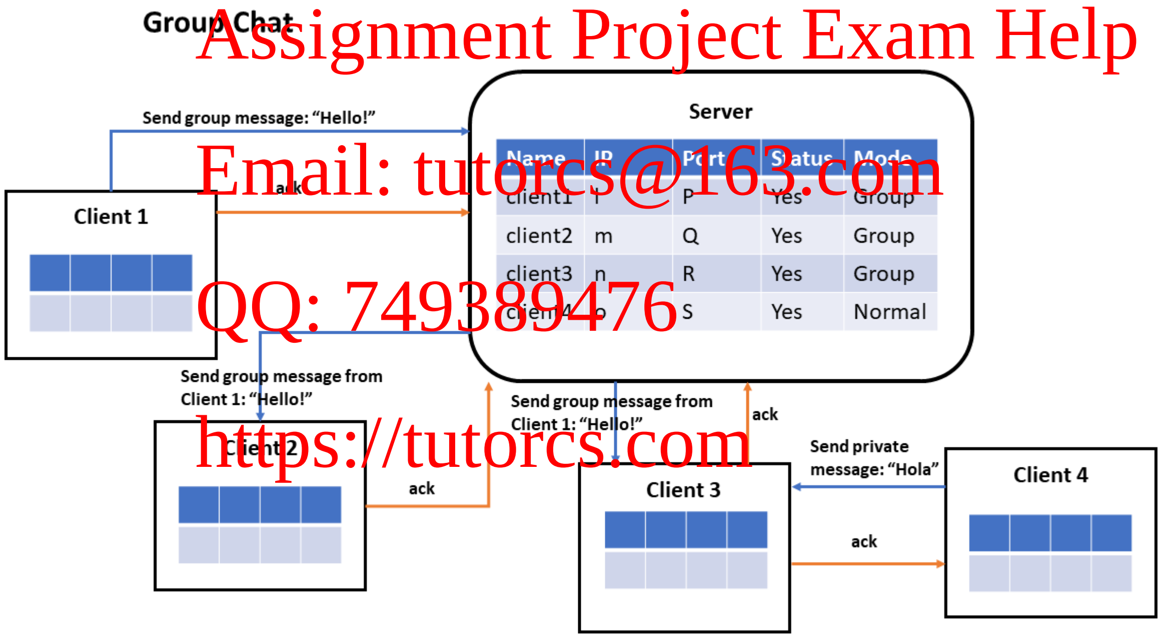


| Name | IP | Port | Status |
|---|---|---|---|
| Client1 | x | Y | No |
| Client2 | p | Q | Yes |

**Test-case 3:**

1. start server
2. start client x (the table should be sent from server to x )
3. start client y (the t [illegible] server to x and y)
4. start client z (the t [illegible] server to x , y and z)
5. start client a (the t [illegible] server to x , y, z, and a)
6. client x sends $ >> [illegible] me> command to the server
7. clients y and z joi [illegible]
8. send group message x-> y, z , but a does not receive the message.
9. send private message a-> z , z stores the message locally and does not display the message until after it exits the chat room.

The figures below shows the group chat involving 4 clients and 1 server. To provide some more *clarity*.


Group Chat
Send group message: "Hello!"
ack
Server
Name | IP | Port | Status | Mode
client1 | l | P | Yes | Group
client2 | m | Q | Yes | Group
client3 | n | R | Yes | Group
client4 | o | S | Yes | Normal
Client 1
Send group message from Client 1: "Hello!"
Client 2
ack
Send group message from Client 1: "Hello!"
ack
Client 3
Send private message: "Hola"
ack
Client 4

## 4 Submission Instructions

You may use either C, Java, or Python for developing the chat application. You should use the version that comes with the standard download of these languages. Your submission package should include the following deliverables.

- `README`: Please put your name and UNI at the top of your README. The next thing in your README should be explicit comma[illegible]ompiling and running your program. The file should also contain basic project docu[illegible]res, brief explanation of algorithms or data structures used, a list of known bugs, a[illegible] additional features/functions you may have implemented (fully optional).
- `Makefile`: This file [illegible]rogram. If you have written the program in C, the output file name should be `ChatApp.`[illegible] name should be `ChatApp.class`. If Python, have your program be called `ChatApp.py`. [illegible]y a Makefile to compile your code if implementing in Python.
- Your source code. Please comment your code well, and use clear and sensible variable names.
- `test.txt`: This file should contain some output samples from the command line on several test cases. This will help others to understand how your programs work in each test scenario. It is optional to include this as a section of your `README` document.

Your submission should be made via Courseworks. Zip all the deliverables mentioned above, and name the zip file as `<last-name>_<your UNI>_PA1.zip` (e.g. `Zussman_gz2136_PA1.zip` for Professor Zussman).

Please do not utilize Windows programming environments including .NET, Visual Studio, VC++, etc. Programs written in C have to be compiled using gcc, not clang or another compiler. All submissions will be compiled, run, and evaluated on the version of Ubuntu that comes standard with your Google Cloud credits. If you have any issues with your environment, please let the TA know early on.

Please comment your code. This not only makes it more likely that you will be awarded partial credit for anything which does not work, but you will thank yourself in six months when you are reviewing your code for a job interview, expanding on it as a personal project, explaining it to your pet fish, etc...

In the grading of your work, we will take the following points into account:

- The documentation clearly describes your work and the test result.
- The program takes command line arguments in the ***exact same format as specified by the assignment***.
- You handle all errors (exceptions, memory management and business-logic) and exit the program gracefully.
- The source code can be compiled and run properly, without errors or warnings.
- The programs run properly, including 1) take appropriate commands and arguments, 2) handle different situations and support required functions, and 3) display correct status messages in given scenarios.

***Happy Coding and Good luck!!***

□