Group members:
Harshit Garg (2018A7PS0218P)
Shreyans Jain (2018A7PS0253P)
Pranav Gupta (2018A7PS0190P)

DESIGN DOCUMENT

P3: Message Queue Messaging System

Problem Statement - Create a basic group messaging system using message queue design that supports the following functions:

1. Login with User ID
2. Create groups
3. List groups
4. Join groups
5. Send private and group messages
6. Receive messages in offline/online mode as and when they arrive
7. Auto Deletion of messages

Message Communication Model

There is a separate message queue associated with between each client and the server for receiving messages, and a common queue between clients as a whole and server for sending messages to the server. There were two reasons to choose this approach. Firstly, a group broadcast message is cumulatively expected to be of size = `size of message * num of users in the group.` This would mean that, in a common queue for all communication, if all group messages are to be queued up in a single queue the message queue size would explode very fast. Secondly, the model for having a separate sending queue for each client and server would be unnecessary since the messages are expected to be sent as and when received by the server to the client (and not be waiting for any party to be alive as in the case of client being offline while receiving message) and hence there is no hogging of queue expected.

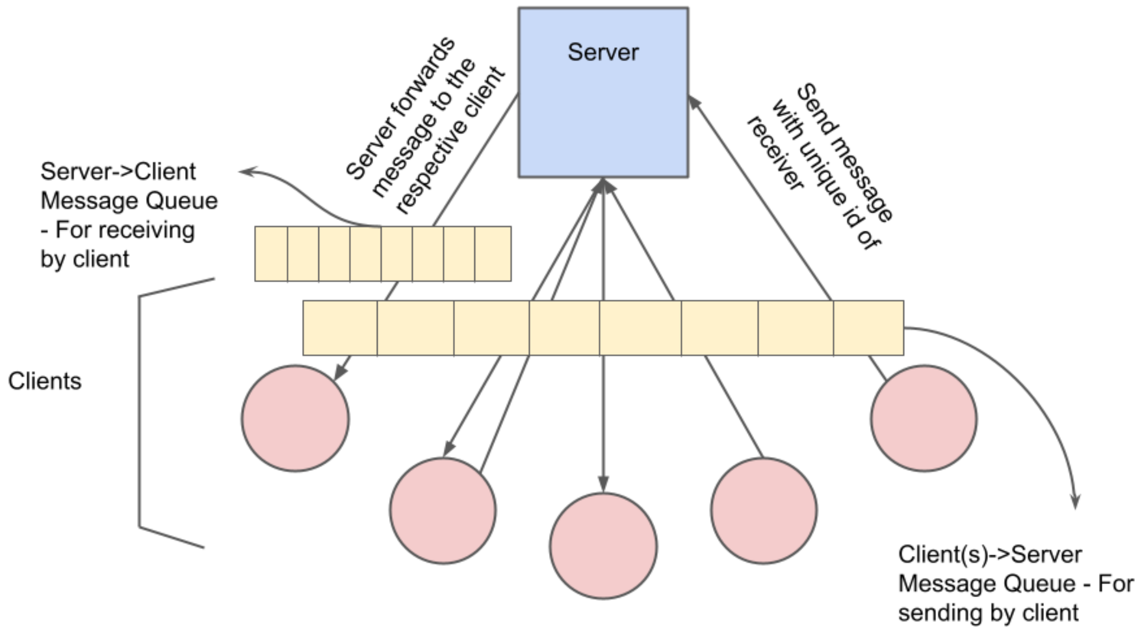
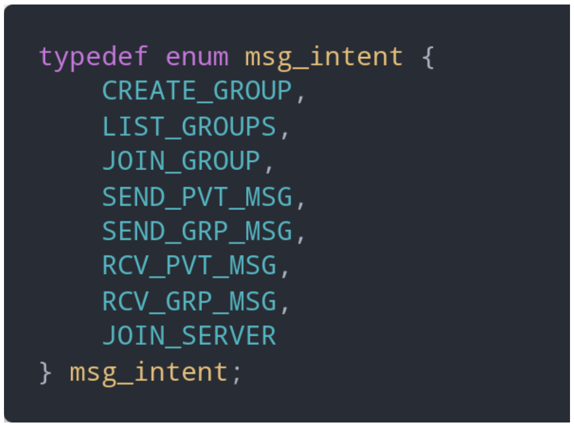

Server
Server forwards message to the respective client
Send message with unique id of receiver
Server->Client Message Queue - For receiving by client
Clients
Client(s)->Server Message Queue - For sending by client

Fig 1. *Graphical representation of the message communication model*

Data Structure

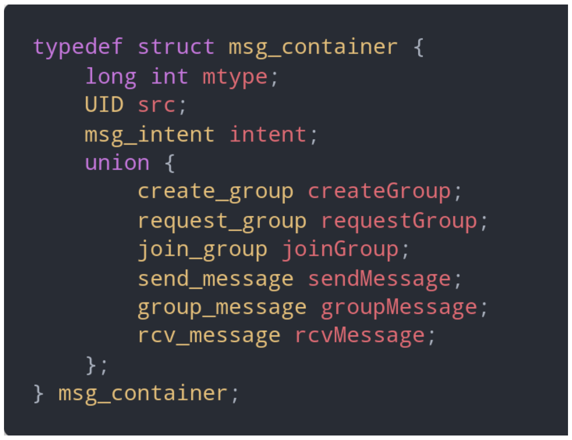
Each message packet has an `intent` variable associated with it that the receiving party uses to decide on which operation must be performed. This is created using an `enum`. The standard `mtype` as a long variable is there, and a `UID` field for source has been created. Further, a `union` of different types of data structures depending on type of message has been created to have a uniform message size and to minimize the size (which otherwise would be manifold if all the fields were entered ad-hoc and left sparse). Individual structures in union have variables as per the need of the message type. The tag for the union is based on the `intent` variable.



```
typedef enum msg_intent {
    CREATE_GROUP,
    LIST_GROUPS,
    JOIN_GROUP,
    SEND_PVT_MSG,
    SEND_GRP_MSG,
    RCV_PVT_MSG,
    RCV_GRP_MSG,
    JOIN_SERVER
} msg_intent;
```

Fig 2. *enum for message intent*

The program in general has been divided into separate functions (although they add some overhead), which makes it easier to make any further changes in their functionality in the future.



```
typedef struct msg_container {
    long int mtype;
    UID src;
    msg_intent intent;
    union {
        create_group createGroup;
        request_group requestGroup;
        join_group joinGroup;
        send_message sendMessage;
        group_message groupMessage;
        rcv_message rcvMessage;
    };
} msg_container;
```

Fig 3. *Message container struct*

Functionalities

1. **Logging in with User ID** Each client has a User ID that is taken as input by the user. This is equivalent to "logging in" to the user account. The user announces the logging in to the server with the intent `JOIN_SERVER`. Any user interactions (joining of groups, sent messages, receiving pending messages etc.) are associated with this UID. The record of joint groups is maintained at the server through a `2D array` of UIDs associated with Group ID (hereafter referred to as gid), and will be stored so long as the server process is running. It is assumed that there is only one server process running.

2. **Create groups** Groups are handled completely server side through arrays. A mapping exists for group ID to the group name. When the user creates a group, the `groupCount` increases and a new value is added to the `groupList array`. Subsequently in the same message call, the user is added to the newly created group.
3. **List groups** This is simply done by the server replying back with the 2D mapping of group ID to group name, and the `groupCount`. The display of group name is done by the `main()` function of the client. It is also displayed if the user is currently a member of the group or not. This helps choose which groups to join.
4. **Join groups** A `2D int array` is maintained for keeping the record of which members exist in a particular group, which is initialized to -1. When a user joins the group, the value is changed to UID.
5. **Send private and group messages** The client reads a message from the user, and sends it to the server through the common queue, with appropriate intent of group or private message. The server then decides whether it's a group or private message. If it's private, it forwards the message to the appropriate UID's message queue. If its group, then the message is sent to each of the members of the corresponding `gid`. The server sends the message through a child process to avoid hogging of the server process. Group message is not sent if the user does not belong in the group.
6. **Receive message** Message is received by a child process of the client at all times, and displayed as soon as it is available. The distinction between sending a message to child or parent is made by the `mtype` variable (in the message queue b/w client and server).
7. **Auto Deletion of messages** It is handled by maintaining a time variable for when the UID last connected and the time when message was sent. The timeout is decided by the user as input for each message. If the time has expired, then the message is not sent to the message queue. The time of joining of the user is obtained from the latest process that has logged in using that id.

Additionally, the server logs all the interactions between the clients and the server for recordkeeping purposes. These logs are displayed on the terminal. A provision has been added to clear all message queues when starting the server to ensure smooth functioning.

Usage

The code can be compiled using the Makefile:

```
make all
make runclient

# In another terminal window in the same directory, run

make runserver
```