Distributed Systems Final Project Design Document

Minqi Ma (JHED: mma17), Jerry Chen (JHED: gchen41)

Communication

- Server's public group, named "server?". It is used for clients to send updates to the server.

#server1#ugrad1

- Client's private group, named "#<username>#ugrad?". It is used for server to send updates to only this client.
- Server-client group, for each connected server and client, named "server?-<username>-ugrad?". It is used to detect client/server connectivity.

#server1#ugrad1 #<username>#ugrad?

- Server-room group, for all clients in this room connected to the same server, named "server?-<roomname>". It is used for the server to send room updates

#user1#ugrad1 #user2#ugrad1 #user1#ugrad2 …...

- Servers group with all 5 servers in it, named "servers". It is used to send messages among servers.

#server1#ugrad1 #server2#ugrad2 #server3#ugrad3 #server4#ugrad4 #server5#ugrad5

Notes

- No hashtag, hyphen and space in username and room name
- Room name cannot be "null"
- Max number of characters in a username, message content and room name is 80
- Some ideas
  - When to save state? Anytime
  - When to clear *logs[server_index]*? If all 5 servers all have some logs up to x
- **Merge steps**
  1. Reconcile on participant list
     - Clear participant list of servers not here
     - Send new participant list of every room to servers group

2. Send “MATRIX <server_index> <25 integers>” to servers group
3. If received expected number of matrices
    - Servers send updated participant list for every server-room group
    - Clear logs up to the lowest index of 5 servers
    - Clear merge *updates* list
    - If I have the highest timestamp and lowest process index for this server, send INIT_SEND_SIZE missing logs to servers group
4. Reconcile on messages
    - Every time get a new “a” update, insert to the messages list in the appropriate order, and inform the server-room group to append new messages
5. Reconcile on likes
    - Wait until get all “l/r” updates (including ones the server already has in logs), keep a list of logs with the highest timestamp+server_index
    - Update the data structures, and inform affected server-room group to update likes on messages
6. Stamp and send updates in buffer to servers group

- Message tags flow at merging
  Membership change → (UPDATE_NORMAL) → PARTICIPANTS_SERVER → MATRIX → PARTICIPANTS_ROOM → UPDATE_MERGE → UPDATE_NORMAL
  There can be another Membership change in the middle of this flow
  There can be another client message at any time

Algorithm

Client

- u <username>
    - Leave server-client, server-room groups if exist
    - Clear data structures, like *messages*, *participants, room_name*
    - Create its new private group with the new username
- c <server_index>
    - Record server index; later the client will send updates to its public group
    - Leave server-client group, if exists
    - Clear data structures, like *messages*, *participants*
    - Join new server-client group
    - Send “CONNECT” message to the server
- j <room_name>
    - Leave previous server-room group, if exists
    - Join new server-room group
    - Send “JOIN <room_name>” message to the server’s public group
- a <content>
    - Send “UPDATE_CLIENT a <room_name> <username> <content>” to the server’s public group
- l <line_number>

    - Find the message's lamport timestamp and server index in the *messages* list
    - Send "UPDATE_CLIENT l <room_name> <timestamp of the liked message> <server_index of the liked message> <username>" to the server's public group
- r <line_number>
    - Find the message's lamport timestamp and server index in the *messages* list
    - Send "UPDATE_CLIENT r <room_name> <timestamp of the liked message> <server_index of the liked message> <username>" to the server's public group
- h
    - Send "HISTORY" to the server's public group
- v
    - Send "VIEW" to the server's public group
- Receive "MESSAGES <num_messages> ..." from server
    - Construct *messages* list accordingly
    - Display
- Receive "HISTORY <creator> <num_likes> <content>" from server
    - Display
- Receive "VIEW <5 numbers 0/1>" from server
    - Display
- Receive "PARTICIPANTS_ROOM <client1> <client2> ..." from server-room group
    - Clear *participants* list
    - Reconstruct the list accordingly
    - Display
- Receive "APPEND <timestamp> <server_index> <username> <content>" from server-room group
    - Append a new message to *messages* list
    - If list size is larger than 25, remove the first message
    - Display
- Receive "LIKES <message's timestamp> <message's server_index> <num_likes>"
    - If the message is in *messages* list
        - Update the likes for that message
- Receive membership change in server-client group
    - JOIN (client itself or server joins the group)
        - Do nothing; client first joins the server, or the server receives "CONNECT" and joins the group
    - DISCONNECT/NETWORK CHANGE (server crashes/daemon crashes)
        - Leave server-room group, if exists
        - Clear data structures, like *messages*, *participants*
        - Leave server-client group
    - LEAVE (should NOT receive this service_type)
    - IS_CAUSED_LEAVE_MESS (client itself leaves the group)
- Receive membership change in server-room group
    - Do nothing; the server will send corresponding updates later
- Timeout

    - If there is only the client itself in server-client group (server crashes and did not respond to CONNECT message)
        - Leave server-client group
        - Prompt the user to connect to another server

Server
Start: ./server <my_server_index>

- Upon the server starts
    - Join servers group
    - Join its public group “server?”
    - If there is state file
        - Reconstruct data structures from state file
        - Retrieve timestamp (counter)
        - Retrieve 5 indices

      else
        - initialize empty data structures, *timestamp* = 0, index = 0
    - For every server,
        - If log file exists, read from the line matching with the corresponding index, insert it to updates list in the order of lamport timestamp + process_index
        - else, initialize empty *logs[server_index]* list
    - Traverse each log in *updates* list
        - Append the update to *logs[server_index]* list
        - Adopt the timestamp if higher
        - Update matrix accordingly
        - Execute the update
- Enter for loop of receiving messages
- Receive “CONNECT” message in the public group
    - Join server-client group “server?#<client_name>?”
- Receive “JOIN <room_name>” message from <client_name> in the public group
    - If in merging state, put update in *buffer* list
    - Search rooms and see if the client is previously in any room
    - Send “ROOMCHANGE <client_name> <old_room> <new_room> <server_index>” to servers group (old_room can be null)
    - Send up to latest 25 messages of this room to the client’s private group
        - Format: “MESSAGES” followed by messages
        - Each message: <timestamp> <server_index> <creator> <num_likes> <content>\n
          4+4+80+4+80 = 172 bytes
          172*25=4300 bytes
- Receives “UPDATE_CLIENT <update>” in the public group
    - If in merging state, put update in *buffer* list
    - Increment lamport timestamp (counter)
    - Increment index

    - Stamp the message with lamport timestamp + index
    - Write the message to log file
    - Append it *logs[my_server_index]* list
    - Update *matrix[my_server_index][my_server_index] to new index*
    - Send “UPDATE_NORMAL <timestamp> <my_server_index> <index> <update>” to servers group
- Receives “HISTORY” from <client_name> in the public group
    - If in merging state, put update in *buffer* list
    - for each message in this room
        - Send “HISTORY <creator> <num_likes> <content>” to client’s private group
- Receives “VIEW” from <client_name> in the public group
    - Send “VIEW <5 numbers 0/1>” to the client’s private group
- Receive “ROOMCHANGE <client_name> <old_room> <new_room> <server_index>” in servers group
    - If in merging state, return
    - If old_room is not null, remove client from *participants[server_index]* in the old room
    - if new_room is not null
        - create the new room if the room does not exist
        - add client to *participants[server_index]* in the new room
    - If *participants[my_server_index]* is not empty for the affected server-room groups
        - Send new participant list to the affected server-room groups, “PARTICIPANTS_ROOM <num_participants> <user1> <user2> ...”
- Receive “PARTICIPANTS_SERVER <room_name> <server_index> <client1> <client2> ...” in servers group
    - (can ignore the one sent from server itself)
    - Create the room if it does exist in the *rooms* list
    - Clear *participants[server_index]* list in this room
    - Construct *participants[server_index]* list accordingly
- Receives “UPDATE_NORMAL <timestamp> <server_index> <index> <update>” in servers group
    - If index is out of order from matrix[my_server_index][server_index], return
    - If the update is not sent by myself
        - Write it to “server[my_server_index]-log[server_index].out” file
        - Append it in *logs[server_index]* list
        - Adopt the lamport timestamp if it is higher
        - Update *matrix[my_server_index][server_index]* to the new index
    - If update is “a <room_name> <username> <content>”
        - Insert the message to *messages* list of the room, in the order of timestamp+server_index
        - If participants[my_server_index] is not empty in this room

            - Send “APPEND <timestamp> <server_index> <username> <content>” to the server-room group
    - If update is “l/r <room_name> <timestamp of the liked message> <server_index of the liked message> <username>”
        - Check if user is the creator, and if the message has been liked by the user.
        - If valid, add/remove username from *liked_by* list
        - If *participants[my_server_index]* is not empty in this room
            - Send “LIKES <message’s timestamp> <message’s server_index> <num_likes>” to the server-room group
    - Save state to state file
- Receives “MATRIX <25 integers>” in servers group
    - Adopt all integers if it is higher, except for line *matrix[my_server_index]*
    - Increment number of matrices received
    - If just received expected number of matrices
        - Clear *updates* list
        - For every room
            - If *participants[my_server_index]* is not empty
                - Send new participant list of this room to my server-room group, in the format of “PARTICIPANTS_ROOM <client1> <client2> …”
        - For each server column
            - Clear *logs[server_index]* list up to the lowest integer
            - Find highest index and lowest index among active servers and put it in *expected_timestamp[server_index]*
            - Calculate *num_updates* expected to received
            - If I have the highest index with the lowest server index
                - Get update in *logs[server_index]* from lowest index+1 to highest index
                - Send INIT_SEND_SIZE updates to servers group in format of “UPDATE_MERGE <timestamp> <server_index> <index> <update>”
                - Record highest sent index in *sent_updates[server_index]*
        - If *num_updates == 0 (i.e no updates to merge)*
            - Mark as out of merging state
            - If *buffer* list is not empty
                - Do what UPDATE_CLIENT or JOIN or HISTORY will do normally
- Receives “UPDATE_MERGE <timestamp> <server_index> <index> <update>” in servers group
    - If not in merging state, print error messages
    - num_updates--
    - If update is sent from myself

        - If *sent_updates[server_index]* has not reached my highest index for this server
            - Send one more update to servers group in format of "UPDATE_MERGE <timestamp> <server_index> <index> <update>"
            - *sent_updates[server_index]*++
    - ~~If update does not exist~~
        - ~~Write it to log file~~
        - ~~Append it to *logs[server_index]* list~~
        - ~~Adopt the lamport timestamp if it is higher~~
        - ~~Update *matrix[my_server_index][server_index]* to the new index~~
    - If update is "a <room_name> <username> <content>" and does not exist
        - Insert it in *updates* list in the order of timestamp+server_index
    - If update is "l/r <room_name> <timestamp of the liked message> <server_index of the liked message> <username>"
        - Insert it in *updates* list in the order of timestamp+server_index
    - If *num_updates == 0*, (i.e just received all missing updates)
        -
        - For every update in *updates* list
            - If update does not exist
                - Write it to log file
                - Append it to *logs[server_index]* list
                - Adopt the lamport timestamp if it is higher
                - Update *matrix[my_server_index][server_index]* to the new index
            - If update is 'a <room_name> <username> <content>'
                - Insert the message to *messages* list of the room, in the order of timestamp+server_index
                - Send "APPEND <timestamp> <server_index> <username> <content>" to the server-room group
            - If update is "l/r <room_name> <timestamp of the liked message> <server_index of the liked message> <username>"
                - Check if user is the creator, and if the message has been liked by the user.
                - If valid, add/remove username from *liked_by* list
                - If *participants[my_server_index]* is not empty in this room
                    - Send "LIKES <message's timestamp> <message's server_index> <num_likes>" to the server-room group
            - (Optional) Save state to state file (save here or after execute the entire *updates* list)
        - Mark as out of merging state
        - If *buffer* list is not empty

- Do what UPDATE_CLIENT or JOIN or HISTORY will do normally

- Receive membership change in server-client group
    - JOIN (server itself joins the group)
    - LEAVE/NETWORK CHANGE/DISCONNECT (i.e client reconnects to another server or crashes)
        - Search all rooms and see if the client is previously in *participants[my_server_index]*
        - If the client is previously in a room, send “ROOMCHANGE <client_name> <old_room> <null> <server_index>” to servers group
        - Leave server-client group
    - IS_CAUSED_LEAVE_MESS (server itself leaves the group, if the client reconnects/crashes)
- Receive membership change in servers group (i.e servers crash/network partition) (Generalize the following events to one merging case)
    - JOIN (a new server just starts, or recovers from crash and rejoin)
    - DISCONNECT/NETWORK CHANGE where OTHER VS sets have no members (a server crashes/daemon crashes/network partition separates away some of the servers; it can happen during a merge)
    - NETWORK CHANGE where OTHER VS sets is not empty (i.e network heals and brings back some servers)
    - LEAVE/IS_CAUSED_LEAVE_MESS (should NOT receive this service_type)

    - Mark as in merging state, record servers in the current network component
    - for every room
        - Clear *participants[server_index]* for servers not in the current network component
        - Send *participants[my_server_index]* to the servers group, “PARTICIPANTS_SERVER <room_name> <server_index> <client1> <client2> ...”
    - Send “MATRIX <25 integers>” to servers group

Tags

- CONNECT: client to server, request to connect with server
- JOIN: client to server, request to join a room
- ROOMCHANGE: server to servers, notify other servers that my client changes room
- PARTICIPANTS_ROOM: server to client, inform current participants for this room
- PARTICIPANTS_SERVER: server to servers, servers reconcile on participant lists at merging
- UPDATE_CLIENT: client to server, send update to server
- UPDATE_NORMAL: server to servers, send client’s update to servers group, not during merging
- UPDATE_MERGE: server to servers, merge logs

- APPEND: server to client, append a new message
- LIKES: server to client, update likes of a message
- MATRIX: server to servers, exchange matrix during merge
- HISTORY
    - client to server, request history of the room
    - server to client, send history of the room
- VIEW
    - client to server, request membership of each server
    - server to client, send servers in the current network component

Data Structures

Client

- struct message
    - *int timestamp*
    - int server_index
    - *char[] creator*: username of the creator
    - *char[] content*: content of the message
    - *int num_likes*
    - *struct message* next*
- *struct message* messages:* up to 25 messages in this room
- *struct participant*
    - *char* name*
    - *struct participant* next*
- *struct participant* participants: list of participants in this room*
- *char* username*
- *char* room_name*
- *int server_index*: connected server

Server

- *struct room*
    - *char[] room_name*
    - *struct participant** participants: array of 5 lists of participants from 5 servers*
    - *struct message *messages*: a list of messages
    - struct room* next;
- *struct room *rooms*: list of all rooms
- *struct participant*
    - *char* name*
    - *struct participant* next*
- *struct message*
    - *int timestamp*
    - int server_index
    - *char[] content*: content of the message
    - *char[] creator*: username of the creator

        - *struct like* liked_by*: list of likes and unlikes of different users
        - struct message* next
- struct like
    - char username[80]
    - bool liked
    - int counter
    - int server_index
    - struct like *next
- *int timestamp*
- *int index*
- *struct log* logs[5]*
- *struct log*
    - *int timestamp*
    - *int server_index*
    - *int index*
    - *char* content*
    - *struct log* next*
- *int[][] matrix*
- *struct log* buffer*: buffered client updates sent in the middle of merging
- *struct log* updates*: updates from log file OR l/r updates sent in merging state
- *bool connected_servers[5]*: index of servers in the current servers group
- *int num_updates*: number of updates expected to receive during merge
- *int sent_updates[5]*: index of updates already sent during merge
- *int num_matrices*: number of matrices expected to receive during merge
- *bool merging*: in merging state or not

Log File Format

- Name: server<server_index>-log<server_index of logs>.out
  e.g server1-log1.out, server1-log2.out
- Line format: <timestamp> <server_index> <index> <log content>
- log content can be
    - a <room_name> <username> <content>
    - l <room_name> <timestamp of the liked message> <server_index of the liked message> <username>
    - r <room_name> <timestamp of the liked message> <server_index of the liked message> <username>

State File Format

- Name: server<server_index>-state.out
  E.g server1-state.out, state2.out
- Format (line by line)
    - timestamp (counter)
    - 5 lamport indices: logs which are executed up to to achieve the state

    - <num_rooms>
    - <room_name> <num_messages>, followed by messages in this room
    - Message:
      <timestamp> <server_index> <creator> <content>
      <liked_by> // it can be an empty line if there is no like; there is a space after each username
- e.g state1.txt
  10
  1 2 3 4 5 ← timestamps
  2 ← 2 rooms
  room1 2
  1 1 user1 hello everyone
  user2
  2 1 user2 hi there

  room2 1
  4 2 user3 what's up
  user1 user2