Trevor Aron
Akshay Srivatsan

Final Design Document

Group structure:

There is a group containing all the servers called “Servers”. This is used for the servers to communicate with each other. Each server is also in a group called “Server#” where # in their server ID. This is for the clients to communicate to the servers. There is a group called “Server#-Client” which contains each server and all of its clients, which is used to monitor the health of the server, so the clients can disconnect if it crashes. Finally, for every group in every server there is a group called “room-Server#” where room is replaced by the room name and # is replaced by the server number. This is the group for a server to multicast updates to clients about a particular group.

Data structures (overview):

Our goal for this project was to abstract most of the logic into complex data structures (a more Object Oriented approach then putting everything in one method :P). These included the following. We end up having a like list, a list of users, a room that has a list of messages(with a list of likes for each) and list of users, like list, a list of rooms, a stack of messages(stored in order of lamport timestamps), and an array of 5 of these stacks (one for each server). There is also an array of size five of user lists. (This is a separate user list to keep track of which users are on which servers.)
When a message gets received by the server, if it is a like, unlike, or message, it will first get lamport time stamped if it is from a client (messages coming from a server already have an associated LTS). Then, we will insert it into our array of stacks, which will automatically put it in the correct stack according to the sender, and then write the message to file.
For every message type (except merge and views), the message will also be passed to the room list, and sent to clients in the corresponding room (based on the group nomenclature). In the room list, the message will be placed in the appropriate room structure, which might mean inserting it somewhere in the middle of the list of messages, and do different things based on its message type: if it is a join or leave, it will update a list of attendees which is stored in each room. If it is a message, it will add it into the room’s list of msgs, in a spot according to its LTS. If it is a like or unlike, it will find the message it is referring to (by lamport time stamp), and then pass the actual like message to the like list associated with that text message. The like list is sorted by the user that sent the like / unlike. When a like is added, if a like / unlike is already there, it gets over written if this new like/unlike has a higher lamport time stamp. This way we only have to maintain the most recent like/unlike action and can avoid issues that might arise during a merge.
On the client side, the client also has a room, although they only have one for the room they have joined (the server by contrast has many rooms). All messages it gets from the server are put into the room (handled by the room’s functions), and then the room prints the most recent 25 messages. We do this by maintaining a pointer in the 25th position, and when an update comes in, we simply iterate through the room starting at the 25th position pointer.

Our basic msg sent over the server
Serv_msg:

- Type (JOIN, MSG, LIKE, UNLIKE, LEAVE, DUMMY, VIEW)
- Lamport time stamp
- Username
- Room

Payload (if Like, will be LTS of msg liked, if msg, will be the msg in the payload)

Room
A linked list of “texts”. These “texts” have a serv_msg (which in the payload will have the message), and a like list of texts. The room also has a list of users. We can pass the room a server message, and it will update the relevant things.

Like List
Linked list of likes. When updating, it will get rid of old likes / unlikes from the same user. What is actually stored is a serv_msg in each node. This like list “tombstones” un likes, to make sure that an old like from a merged server doesn’t kill it. Will store the total like count, and update it in the like list update.

Room List
List of rooms. We can pass a serv_msg to it, and it will update the relevant room.

Msg Stack
Doubling array to store messages. It is called a stack because we only add to the end. (Although we can access the middle to send messages).

Lamp Struct
We pass serv_msg’s to this. It has a msg stack for each server, and can return an array of the highest lamport timestamps it has for each server.

User List
A list of users, with each user having a name, room, and number of occurrences (to hold duplicates).

Merging
On a merge, if a server detects that the new view contains any servers that were not in the previous view, then every server sends out an array containing their most recently received lamport time stamps from each server. The servers use the arrays to determine if they have the most recent message sent by any of the servers (ties are broken by server ID number; lower will send over higher), and then they will multicast all messages from the oldest LTS appearing in at least one server’s array to their most recent. When a server gets these messages, they handle them normally. However, if a server receives a message that has an LTS which is lower than its most recent for that server, it will discard it to avoid duplication.
To keep track of attendees, whenever a server moves to a new view that does not contain a server from the previous view, it will wipe the users which were tied to that server in its room structures. The servers then make “leave” messages for each of these, and pass them to their room list and all the clients. When new servers join a membership, all servers will send a join for each user they have stored (if a user has multiple instances they will send multiple joins, and they will include the room names in the joins). Since this can lead to duplicates among the servers that transition from view to view together, we clear the user lists after sending our own. These joins will be handled by a server only if the server just joined their partition.

Boot

On boot, the server will read blocks of messages from its file (Server#), and handle them just like it handles regular messages (it will give them to the appropriate data structures).

Client

The client will print a menu describing the options to the screen.

U command: The client will "log in". This will disconnect the client from a server (it will leave the group of the server and the room, and send appropriate leave message.).

C command: The client will connect to the server specified by joining the "RoomS#" group. This will then give it a membership message, which it will parse to make sure the server is there. If it is already in a room, it will disconnect from the group and delete its room data structure, and send a leave message to the server. If it is already connected to a server, it will disconnect by leaving the old group

J command: The client will join the "room-Server#" group, and initiate a new room data structure. If it is already in a room, it will leave the old group, delete the room data structure, and send a leave.

L and R commands: The client will run get_lts(room, number), to get the lamport time stamp of the message it is liking/un-liking. This will then be put into the payload of a serv_msg message (that will also include user name and room) and sent to the server.

A command: the text will be put into the payload of a serv_msg message that will be sent to the sterver.

History: Print the room datastructure with recent parameter of 0.

V: The user will send a message of type VIEW to the server.

On receiving msg:

View type: for each char in the payload of the message, if there is a 1 that server is present, and print that.

Other types: hand over to room data structure.

Server

When a server gets a message from itself, it ignores it unless it is a merge type.

For merge type messages, it calculates new min's and max's, and once it has all of them, the server will initiate merges.

For View type messages, the server will look at its history, see which servers were included in the last membership change, and send a serv_msg to the client that sent the message, with a payload that has a 1 in slot X if server X is up.

For likes/unlikes/msgs from the client, the server will stamp them with a lamport time stamp, allocate space for them (since they are stored), write the message to file (Server#), give it to its lamp_struct (the array of msg_stacks), and also give it to its room list. Then it will multicast the message to the corresponding group, and to the other servers.

For likes/unlikes/msgs from servers, the server will allocate space for them (since they are stored), write the message to file (Server#), give it to its lamp_struct (the array of msg_stacks), and also give it to its room list. Then it will multicast the message to the corresponding group specified in the serv_msg it just received.

For joins from clients: The server will go to the specified room in its roomlist, and send the client that sent the join a serv_msg for each text in the list of texts (and one for each like in the list of likes in each text), and a join serv_msg for each user in the list of users (and if a user has multiple instances, send multiple). The server will then multicast the orignal to the servers, and the corresponding group. Then the server will give the message to its room data structure.

For leaves from clients: The server will multicast this leave message to the other servers, and to the relevant group of clients, and give this leave to its room list, and own userlist to update.

For joins/leaves from servers: the server will multicast this to the relevant group of clients. It will then give this to it's room list and relevant userlist (depending on which server it is from) to update.