Trevor Aron
Akshay Srivatsan

Final Project Design Document

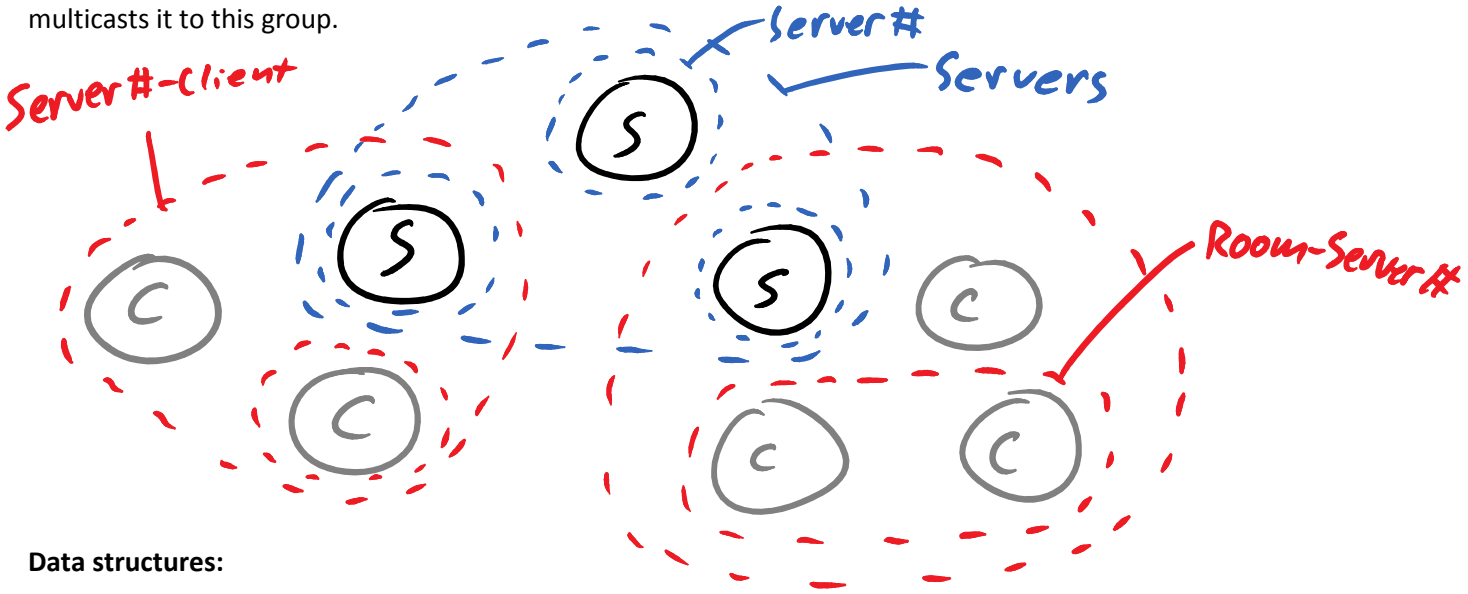
Spread Groups:

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 room and every server there is a group called “Room-Server#” where Room is replaced by the room name and # is replaced by the server number. All clients which are in that room and connected to that server will join this group. When a server receives or creates an update to a room, it multicasts it to this group.



Data structures:

Serv_msg: A struct for messages sent over Spread. It contains a type (JOIN, MSG, LIKE, UNLIKE, LEAVE, DUMMY, MERGE, VIEW), a Lamport time stamp, a username, a room, and a payload (if it’s a like, the LTS of the relevant message, if it’s a message, the string for the message).

LTS: A simple struct for Lamport time stamps, containing an index and a server number. The servers each maintain an LTS for their own internal clocks, and update them appropriately upon sending and receiving messages.

Room: A linked list of texts, which each represent a message in a chatroom. These texts have a serv_msg and a like list (see below). The room also has a list of users who are presently online. We can pass the room an update in the form of a serv_msg. If it is a message, it will create a text and insert it into the list based on its LTS. If it is a like/unlike, it will find the corresponding message and based on the LTS add/remove a like from that user (if the message does not exist it will create a DUMMY message there).

We can print the room by iterating through the texts, starting either at the head, or the message with the 25th most recent LTS.

Room List: Each server has a list of rooms that it has messages for. The room list accepts serv_msgs and updates the corresponding room as described above. If it gets a serv_msg about a room that it doesn't know about, it creates it.

Like List: Linked list of like/unlike serv_msgs, sorted by the user that sent it. The like list will only store the most recent (based on LTS) like/unlike message from any user (we tombstone the unlikes to make sure they aren't removed by an old like message). We also store the total number of likes/unlikes for easy access.

Msg Stack: A doubling array to store serv_msgs. We use it to store all messages that we have received from another server. We use a stack because we only add to the end, as it is not possible to receive a message with a lower timestamp than the most recent one from a particular server. We maintain a head pointer so that we can iterate through the items as well.

Lamp Struct: An array of msg stacks, with one for each server. It can return an array of the highest lamport timestamps it has for each server, which is useful in merging.

User List: A list of users, with each user containing a name, room, and number of logins (to hold duplicates). Each server has an array of user lists, with one for each server.

Client Map: This is a mapping data structure from private group names of clients, to their associate username and room name. The purpose of this is to handle when a client crashes (so it doesn't send a leave message).

Client

The client will print a menu describing the options to the screen, and will process them as described below.

u <username>

The user will be logged in with the given name. If the client is connected to a server, it will leave the Room-Server# group, after sending the appropriate leave message to its Server# group. Then it will clear its room data structure.

c <server name>

The client will connect to the server specified by joining the Server#-Client group. It will then receive a membership message for that group, which it will check to make sure the server is there. If at any point it receives a membership message with a view that does not include the server, it will leave the Server#-Client and Room-Server# groups. If the client is already in a room or connected to a server, it will disconnect from the associated groups and delete its room data structure, after sending a leave message to the server.

j <room>

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 Room-Server# group, delete the room data structure and make a new one, and send a leave message to the Server# group.

l <line #>

r <line #>

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 <message>

The text will be put into the payload of a serv_msg message that will be sent to the server.

h

Print the room data structure, starting at the head pointer (not the 25th position).

v

The user will send a message of type VIEW to the server. When the server receives that message, it will send a message back to the client containing an array of 1/0 indicating whether or not it can see the server with the corresponding index.

q

The user will send a leave message to its Server# group and leave the Server#-Client and Room-Server# groups. It will then close the client program.

The client has a room structure to track the texts its user is interested in. The reason that we designed it this way is because if the user wants to request the full history relatively frequently, it is more efficient to have that information stored locally on the client side. We assume that when a user connects, it will stay connected for a fairly long amount of time before leaving, and that it can tolerate a moderate waiting time upon connection in exchange for faster updates; if we do not store the room structure on the client side, we will have to send the full contents of the last 25 messages upon every update, and the liking system will be more complicated since the client will not have the LTS for each message. When it receives a message from the server, it inserts it into the room, and then prints the room starting at the 25^{th} most recent message pointer.

Server

On boot, the server will read any saved messages from its file (named Server#). Each message is then inserted into the lamp struct and the room list. It will then join the Servers and Server#-Client groups.

When a serv_msg gets received by the server, it will first check if it was sent by itself, and if so ignore it unless it is of type MERGE.

If it is a like, unlike, or message, it will first get assigned an LTS if it is from a client (messages coming from a server already have an associated LTS), based on the LTS maintained by the server. Then, we will insert it into our lamp struct and room list, and then write the message to file. Then we send the message to the appropriate Room-Server# group where it will be received by the clients. If the message was from a client, we also relay it to the Servers group.

More specific handling is described below:

Joins from clients: The server will go to the specified room in its room list, and send the client every message and all of the associated likes in the like list. It then sends a join serv_msg to the client for each user in the list of users for that room (and if a user has multiple instances, it sends multiple messages). The server will then multicast the original join message sent by the client to the Servers group, and the corresponding Room-Server# group. Then the server will give the message to its room data structure and update its user list. We then give the private group name of the client that sent the join, and the server message to our client map 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 user list to update. We then remove the private group name of the client from our client map data structure.

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

For views: Based on which servers that were present in the previous membership message, the server will send a view message back to the client that sent the message, with a payload that has a 1 in slot X if server X is up, and a 0 otherwise.

For merges: The server will check if the LTS sent for each server is either the maximum or minimum LTS received since the start of the merge. If so it stores the values and remembers the sender as the sender of the maximum LTS for that server. If it has received a merge message from every server in the current view since the start of the merge, it will then check for each server who had the maximum LTS. If the server itself has the maximum LTS, it then sends all messages in its lamp struct for the relevant server from the minimum onwards.

For membership messages: If there was a change in the Servers group, the server will look through the target_groups and determine who is in the current view. It will compare this to the previous view, and see if any servers have left. If so, it will empty its user list for that server, and send leave messages to its clients for those users. As always, these leave messages are sent to the Room-Server# group and in multiples based on the number of times that user was logged in. If the server notices that there is a member in the current view who was not in the previous view, it will perform a merge, detailed below. If there was a change in the Server#-Client group, the server will loop through its client map, and make sure they are all still there. If anyone leaves that is still in the client map, then the server will send itself a leave message with a CC in the payload. When the server receives a message, if it is from itself, and has the CC it will handle the message as if it was from a client, and remove the CC.

Merging

On a merge, if a server detects that the new view contains any servers that were not in the previous view, then it will send a serv_msg of type merge containing its most recently received lamport time stamps from each server. Upon receiving a merge message, 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 message from that server, it will discard it to avoid duplication.

To keep track of attendees, when a merge occurs, all servers will send a join for each user connected to them (if a user has multiple instances they will send multiple joins). These joins are sent only to the servers that were not in the previous view. Other servers will receive these messages and handle them as normal joins.