John An
Edward Schembor
Distributed Systems
Final Project Design Document - 337 Level

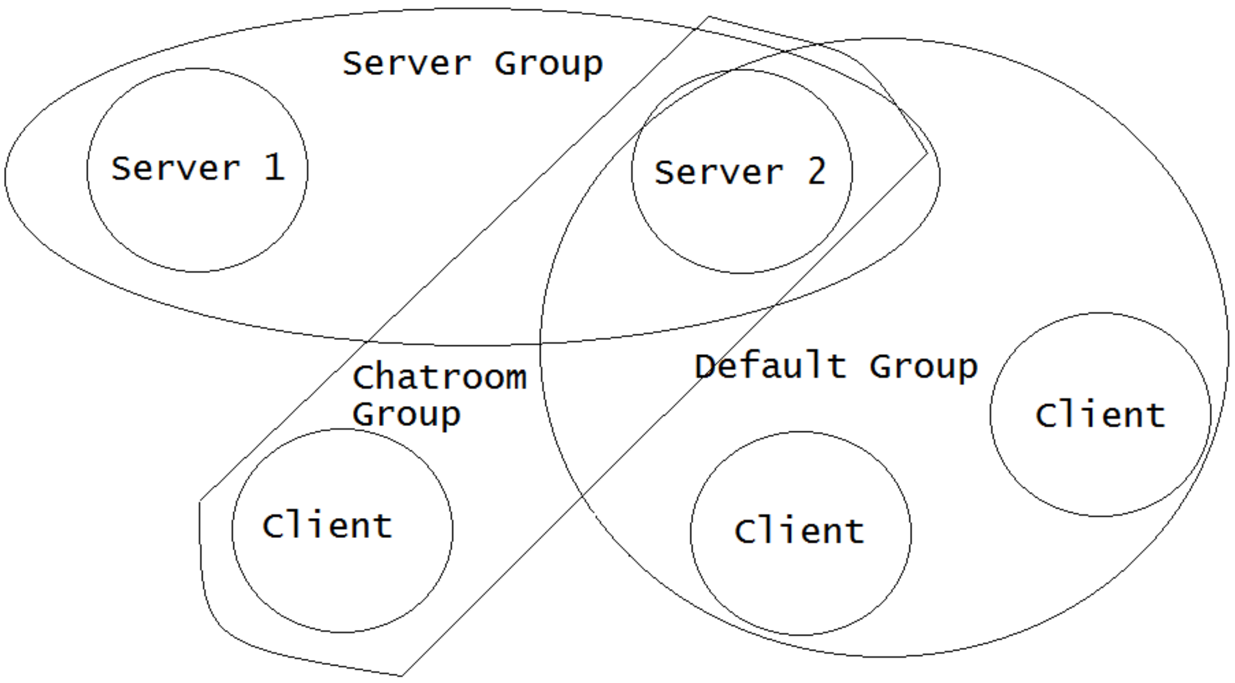
Project Goal: Construct a reliable and efficient chat room service for users over a network of computers.

<u>Spread Architecture</u>

Servers -> Spread group for multicasting between servers
Chat Room(s) -> Spread group(s) with connected server and other clients in the same chatroom

We assign server users names with the form #<machine_index>#ugrad<N>
We assign clients user names by using the local machine time

All servers are in a group together, all clients in a chatroom are in a group with the server they're connected to and grouped by chat room, all clients connected to a server are in a group (for crashes) so that no clients are "floating" while connected to a server.



Server Data Structures and Important Variables

Update Structure:
- int : type
- char[] : message
- Lamport timestamp ---> int stamp and int server_index
- Lamport timestamp ---> timestamp of message liked, if any
- char user[]
- int vector[]
- char chatroom[]

-We know the room name of the update because thats just the name of the Spread group that the client is currently in, but we put it in the message when a client wants to let a server know that it is joining a new chatroom

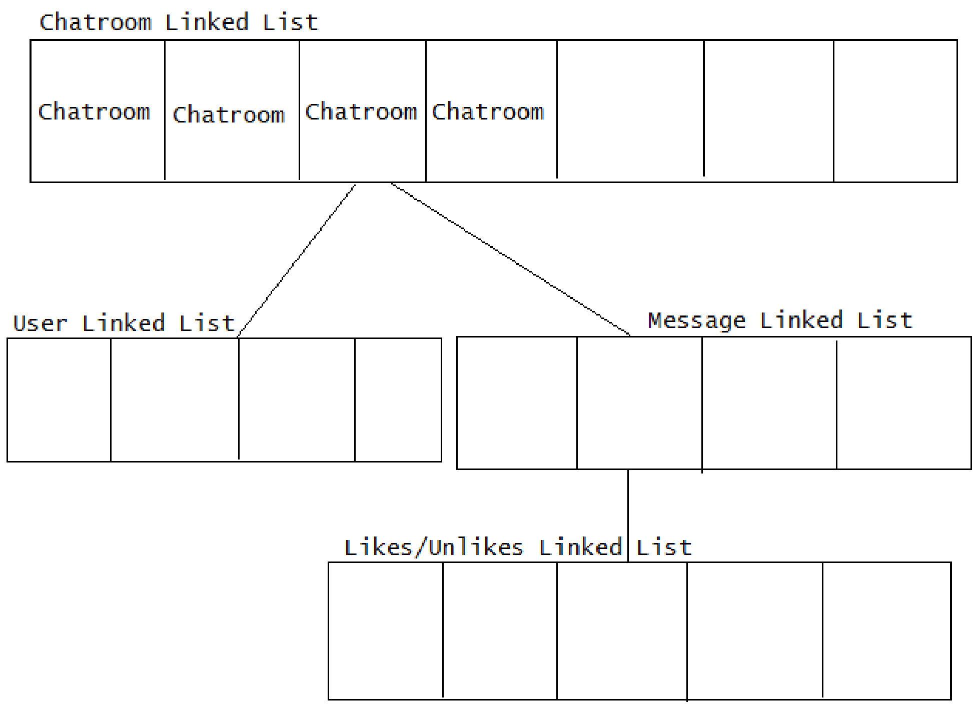
- Structure for Keeping Track of Other Servers aru's: 2D Array Matrix (see Anti-Entropy Method)

- Structure for Holding updates from other servers after a merge: Doubling array for each server's updates

- Structure for Holding all Messages and the main information: Linked list data structure

- Structure for Holding which servers are in the current server's view: Integer Array

Lamport Counter - int
Server Index - int



Chatroom Node: - char* chatroom_name
- user_node* user_list_head
- message_node* mess_head
- chatroom_node* next

Message Node: - lamport_timestamp timestamp
- char message[]
- like_node* like_head
- message_node* next
- char username[]
- int num_likes

Like Node: - lamport_timestamp timestamp
- lamport_timestamp mess_liked_timestamp
- char username[]
- like_node* next
- int type (1 for like, -1 for unlike)

<u>When to Replicate</u>

When a user sends a message to the chat, that message is multicast to all servers in the server's group. This ensures that all the non-partitioned servers maintain the same state.

When a server gets a membership change in which a process was added to the server group, this is when we want to replicate. This is because when there is a membership change which adds a new member, the new process must have been originally partitioned. So, if we now merge the server groups which have diverged, our replication method will bring them back together so that they have the exact same contents.

<u>Replication Method - Anti-Entropy Based Method</u>

When a merge occurs (server(s) join a group), each server will multicast its anti-entropy vector. Then, each process in the new group will have a matrix containing the knowledge of all other processes.

Then, the process with the highest value in each column of the matrix will send the necessary updates to the processes which do not have this value (by sending to the servers' private groups). If two servers have the same highest value, then the server with the higher machine index is the one which will send. So only one process from each group sends messages to those who are missing it - a message needed by x servers is only sent x times. To deal with flow control when sending a high volume of updates, we sends smaller bursts of messages and only send another burst if we have received all of our messages.

Each process has an array of updates for each other server and one for itself. This method is much more efficient for sending missed messaged as opposed to something like a linked list.

When a server receives a message update during replication, it checks the Lamport Timestamp of the message which it just received. The server then orders the messages in the appropriate update array based on the timestamps. This method avoids special cases and maintains a consistent ordering system on all servers which avoids line numbers.

When a server receives a like/unlike update during replication, it likes the proper message by finding the message with the appropriate Lamport Timestamp, avoiding absolute line number issues. If there is a collision (one server's likes to not equal another server's likes). then we must consistently choose a server to use. We choose the server with the higher like count for the message. This avoids the special case where one server has a like and nothing occurred with likes on the other server. If we see a like for a message which we do not yet have in our data structure, we create a placeholder for that message with the like/unlike since we know that the message will exist eventually.

After sending all of its updates to the servers which need it, the servers send out a user join message to all of the new servers in its partition. This can be done by having the "prev_in_group" vector which we already have to keep track of who was previously in the servers group.

Also, by definition of the lamport timestamp protocol, if the lamport timestamp index of a received update is greater than your local timestamp index, set yours to the new, higher one.

If we receive a membership message during a merge which represents another partition, we cease sending out updates to those servers which have become partitioned, process the updates received from them, and continue the process with the servers in the new partition.

If we receive a membership message during a merge which represents another merge, we restart the process with the new server view.

Client Side

Our client side's main purpose is for the user interface and to check that updates are valid before sending them - this avoids sending unnecessary messages which would be ignored by the server.

The client keeps track of the user's name, the name of the current chat room, and the server which the client is currently connected to. When a server is partitioned, its sends who is in its current view to its clients and they can remove users in their user list based on their connected server.

Client Requests

In addition to the primary user input requests (ie. like, append message, history), we have requests to deal with clients joining and leaving chat rooms, which are both sent using update message structs.

Both update contain the chatroom which the client is trying to join, and the client's user name.

We always join the new group before leaving our previous one. This is because we do not want the server to crash in the middle and then have the client connected to no chatroom group.

Client Methods

Login: First, the method sets the 'user' field, which stores the username of the client. Then, if the user is connected to a server, we must send a pair of leave/join chatroom requests in order for the server to update the username of the client on all other servers/clients.

Connect to Server: This method joins the client to a default Spread group with the server which contains all clients on the server which are not currently in a chat room, after checking that the client has set a username. We then check to see if the chat room is non-empty to be sure the server is running and we can join the room. If not, we leave it. If so, we leave our previous group.

Join a Chat: This puts the client in a Spread group, the name of the group being the name of the chat room with the server index of the server it is connected to appended on, after we check that the client has connected to a server. If the given name is too long, it is concatenated. We first send a message to the server telling it we want to join the chatroom and the server joins the proper group. Because of this, we can have the client check that the group contains someone when joining, or it is invalid. The group contains all clients in the same chatroom which are also connected to that server and the server itself. The server looks for the the chatroom in its data structure; if it exists, it unicasts the most recent 25 messages to the newly joined client via private group.

Append to Chat: If the client is in a chatroom, it unicasts the new message update to the server. If the given message is too long, it is concatenated. When the server receives the update, it places it in its update array, sends the update out to the other servers in its partition and applies the update to its data structure. The server then sends the updated message back to the clients in the chatroom group of the original message.

Like/Unlike: If the client is in a chatroom, the client checks is liking/unliking the chosen message is valid (if it is in the line number range, wasn't created by the user, hasn't already been liked/unliked). If the operation is valid, the client unicasts a message with the lamport timestamp of the message attempting to be liked/unliked.

When the server receives the update, it places it in its update array, sends the update out to the other servers in its partition and applies the update to its data structure. The server then sends the updated message back to the clients in the chatroom group of the original message.

Note: In the data structure, we update the messages num_likes parameter by traversing the edited like linked list to check the new amount of likes which the message has.

Print History: If the client is in a chatroom, it sends a request to the server and the server traverses through all of the messages it has for the user's chat room, except for the most recent 25, and unicasts to the client who requested them.

It then sends a message knowing the client know that it is done with sending historical messages. The server then prints the most recent 25, which it already has stored locally. We do not have the server send the most recent 25 because it would interfere with our normal message append logic which is done by lamport. We can do this because we know the client will have a consistent view with the server it is connected to for the last 25 messages at all times. We also can wait for the severs "done sending" message since we are guaranteed that the server will not crash in the middle of the sending process.

Server View: If the client is connected to a server, it sends a message to the server requesting the servers current view in the network. The server then sends its vector which contains which servers it is connected to. From this, the client can print which servers the connected server is in a partition with. When the server receives a membership message from the server group, it checks if the servers in its partition are different; if they are, the server updates its local vector containing who is in its partition.

Client-Server Partitions

When a client and the server it is connected to become partitioned, the client will receive a membership message and see that in the groups parameter the server its was connected to is gone. In this case, the client will leave its chat room, and set its is_connected parameter to 0 since it is no longer connected to a server. If this occurs while the client is waiting on a “history end” message from the server, it ends the history process.