Chas Forelle, Mark Tsao

CS 437 FINAL PROJECT DESIGN DOCUMENT

TABLE OF CONTENTS PAGE

1. OVERVIEW

GROUP ARCHITECTURE

Three public Spread groups are used:

1. A ServerGroup that servers pass updates through.
2. A client_group<server_num> that all clients join when connecting to a server.
3. CR<server number>_<chatroom name>: all clients who have joined <chatroom name> and are connected to server <server number>. This is to isolate chatrooms to each server.

Additionally, servers create a public group server<server_num> in order for clients to be able to send initial connection messages.

TYPES OF MESSAGES USED AND THEIR FORMAT

Each kind of update and membership change is represented by packets with different TYPE constant. Each packet contains the information needed for the clients and servers to understand its source and contents.

DATA STRUCTURES USED IN SERVER AND CLIENT

Both the servers and the clients maintain accurate representations of the rooms they know of:

- Clients only need a single room node with all users in that chatroom, and a single room node with a linked list of messages, where each message has a linked list of users who liked that message.
- Servers need a linked list of rooms, where each room has a linked list of messages, and each message has a linked list of users who liked that message. Another linked list of rooms is stored where each room has a linked list of users in each room.

Servers use an int vector[5] array as an anti-entropy vector. The latest update received from each server is stored here, so that merges can be resolved.

The server can build its data structures by reading each update from its update file. The updates can be read out of order because their order will be resolved as they are inserted into the data structure.

ALGORITHM IN REGULAR CASE

Based on the type of update a client wants to enact upon the chatroom, the appropriate message type will be generated.

When the server receives the message from its client, it will stamp it, write it to file, update its vector and room-list data structure, and send the update to its clients in the chatroom and other servers.

Other servers will adopt the stamp if it’s higher than their local stamp-index, write it to file, update their room-list data structure, and send the message to their clients in the chatroom.

ALGORITHM IN CASE OF MEMBERSHIP CHANGE

When membership changes, a Spread update will be generated.

If it’s a client membership change, where the client is joining/leaving a chatroom on a server, that server will create a join/leave message and send it to all other servers, who will send it to their clients in that chatroom. Everyone who receives the message can update their room-users list accordingly, by either inserting or deleting the user.

If it’s a server membership change, in the case of a server joining the partition, the servers will exchange anti-entropy vectors. For each server i, if the values for server i are not identical across all servers’ vectors, the server with the latest update from server i will send to other servers updates they are missing in the event the source server is not in the partition. Otherwise, the source server i handles its own section of the vector.

2. GROUP ARCHITECTURE

1. ServerGroup: A group for all servers in a partition
   (1 such group per partition)
2. client_group<server number>: A group for each server and all clients connected to that server
   (1 such group per server)
3. CR<server number>_<chatroom name>: A group for all clients connected to a server that are all in the same chatroom
   (1 such group per server per chatroom)

1. ServerGroup: A group for all servers in a partition
   (1 such group per partition)

   Whenever a server joins or leaves a partition, all servers in the left and joined partition(s) receive a membership update from Spread. By being in this Spread group, the servers in every affected partition can know of membership changes and reconcile their updates. In addition, this Spread group is used to synchronise any updates a server receives from any of its clients with the other four servers.

2. client_group<server number>: A group for each server and all clients connected to that server
   (1 such group per server)

   On connecting to a server, clients connect to the server's client_group so that both servers and clients have an easy method of determining when the other crashes or leaves. Clients do not directly connect to any other group that also contains the server, and by utilizing this shared group both parties can leverage the Spread membership updates.

3. CR<server number>_<chatroom name>: A group for all clients connected to a server that are all in the same chatroom
   (1 such group per server per chatroom):

   Because this project is simulating a distributed network of servers on Spread and an indeterminate number of clients that connect directly to specific servers, a method is needed to provide isolation for each server's chatrooms. When clients wish to connect to a chatroom they must connect to their server's "local" copy of the chatroom, even though all clients could theoretically connect to single Spread group chatroom and remove the servers altogether. Clients send their server a chatroom connection message so that servers know when to serve specific local chatrooms with updates.

3. TYPES OF MESSAGES AND THEIR FORMAT

1. TYPE_JOIN_CHATROOM
    a. type: joining a chatroom
    b. source: from a client or server
    c. u_id: user id of user logging in
    d. sp_id: Spread user of client logging in
    e. chatroom: name of chatroom logging into

   When a client joins a chatroom, it will send this type of message to its server. The server will change source to server before sending it to other servers. Other servers will receive this message and see that source is server so it will only send it to clients in that room.

2. TYPE_LEAVE_CHATROOM
    a. type: leaving a chatroom
    b. sp_id: Spread ID of client leaving
    c. u_id: user id of user leaving
    d. room: room that client and user are leaving

   When a client leaves a chatroom, its server will receive a Spread update. The server will send this type of message to other servers and its clients in that room. Other servers will receive this message will only send it to clients in that room.

3. TYPE_SEND_MSG
    a. type: sending a message
    b. source: from client or server origin
    c. lamport timestamp: server ID and counter
    d. u_id: user id of message author
    e. chatroom: chatroom message was said in
    f. mess: content of message

   When a client appends a message to a chatroom, it will send this type of message to its server. The server will change source to SOURCE_SERVER and timestamp it before sending the message to other servers. Other servers will receive this message and see that source is server, so they will only send it to their clients in that room.

4. TYPE_LIKE_MSG
    a. type: liking a message
    b. source: client or server origin
    c. lamport timestamp: server ID and counter
    d. lamport msg_timestamp: timestamp of message being liked
    e. u_id: user doing the liking
    f. chatroom: chatroom message being liked is in
    g. like_state: LIKE or UNLIKE

   When a client likes a message, it will send this type of message to its server. The server will stamp it and change source to server before sending it to other servers. Then, all servers will send the message to their clients in that room.

5. TYPE_PARTITION_STATE
   When a client wants to see the state of the current server partition, it uses this message to signal the server to send it the current members in its ServerGroup.

6. TYPE_HISTORY
   When a client wants to view the entire chatroom structure for debugging purposes, it signals the server to send all messages (and likes for those messages) to it with this message.

4. DATA STRUCTURES USED IN SERVER AND CLIENT

1. Data structures used in server
    a. roomslist: stores a linked list of room_node.
        i. room_node: stores the name of the chatroom and a linked list of msg_node.
            1. msg_node: stores a message message with Lamport timestamp, message content, and author user id, along with a linked list of like_node.
                a. like_node: stores a like message with Lamport timestamp, like state, and user id.

    roomslist is used to keep track of the messages and likes in each chatroom. Message and like/unlike messages can be inserted regardless of the order they are received because they will be ordered by ascending timestamp. In the event of a tie, they will be ordered by ascending server ID.

    b. roomuserslist: stores a linked list of room_users_node.
        i. room_users_node: stores a linked list of user_node.
            1. user_node: stores a user with user id and Spread id

    roomuserslist is used to keep track of which users are in each chatroom. When a client joins a chatroom, a user_node is created and stored in the appropriate room_users_node.

    c. int vector[5]

    vector is used to store the latest message’s timestamp received from every server. These values are used as an anti-entropy vector to resolve merges.

2. Data structures used in client
    a. roomlist: Same as server, except the max amount of rooms is 1.
        i. msg_node: Same as server, except clients enforce the max_messages to be 25.
            1. like_node: stores a like message with Lamport timestamp, like state, and user id.

    b. room_users_node: stores a linked list of user_node.
        i. user_node: stores a user with user id and Spread id

    The data structures in the client are similar to those used by the server (minus the anti-entropy vector), except the client only needs to maintain a single room_node and room_users_node, because a client can only be logged into a single chatroom at a time. When a client logs into a chatroom, the server will look through its data structure and send the client the last 25 messages for that chatroom. The client subsequently destroys its data structures when leaving or joining a chatroom.

5. ALGORITHM IN REGULAR CASE

<u>Client joins a chatroom:</u>

- the client cannot log into a server without a username
- the client cannot log into a chatroom without a username & a server
- the client joins the relevant chatroom Spread group CR<server number>_<chatroom name>
- the client sends the server a TYPE_JOIN_CHATROOM message to register the chatroom as active with the server:
    - the server updates roomuserlist with the new membership, inserting the new user and spread id into the correct room_node of roomuserlist
    - the server sends a TYPE_JOIN_CHATROOM message to CR<server number>_<chatroom name>, and all other servers in its ServerGroup
- when other servers receive the TYPE_JOIN_CHATROOM message from another server, they will:
    - update their roomuserlist by inserting the new user and spread id into the correct room_node of roomuserlist
    - send a TYPE_JOIN_CHATROOM message to CR<server number>_<chatroom name>
- when clients in the Spread groups CR<server number>_<chatroom number> receive the TYPE_JOIN_CHATROOM message from their server, they will:
    - update their room_users_node to reflect this membership update by adding the new user
- the server will find the correct chatroom in roomslist
- the server will send the latest 25 messages and their likes from that chatroom as TYPE_SEND_MSG and TYPE_LIKE_MESSAGE to the client
- the client will update its roomslist with the 25 latest TYPE_SEND_MSG and their corresponding TYPE_LIKE_MESSAGE updates
- all clients in the chatroom will reprint their rooms, because a new user has joined the chatroom, and the joining client also needs to print the last 25 messages and their likes

<u>Client wants to send a message:</u>

- the client sends a TYPE_SEND_MSG to its server
- when the server receives a TYPE_SEND_MSG from a client, it will:
    - increment its timestamp counter and stamps the TYPE_SEND_MSG update
    - write the message update to file
    - insert the message update into the correct room_node in roomslist
    - update its index in its anti-entropy vector, as this is a new message
    - send the TYPE_SEND_MSG update to all their clients in Spread groups CR<server number>_<chatroom name> and all other servers in its partition
- when a server receives the TYPE_SEND_MSG from another server, it will
    - check the index for the server it recieved the message from. If the received timestamp is less than the stamp in the checked index, the TYPE_SEND_MSG update is thrown out. Then, the server will:

    - adopt the counter if it’s greater than the server’s personal counter
    - write the message update to file
    - insert the message update into roomslist
    - update the sending server’s index in its anti-entropy vector, as this is now the latest message received from that server
    - send the TYPE_SEND_MSG update to all their clients in Spread groups CR<server number>_<chatroom name>
- clients in Spread groups CR<server number>_<chatroom name> receive the TYPE_SEND_MSG from their servers, they will
    - update their roomslist by inserting the message
    - reprint the room

TYPE_SEND_MSG update as it would appear in servers’ update files:
<timestamp server ID> <timestamp counter> <type = 1 = TYPE_SEND_MSG> <chatroom name> <author user id> <message content>
example: if a client logged into server 1, in chatroom room1, as user “Yair”, sends the first message of “hi”, it would appear like this:
1 1 1 room1 Yair hi

<u>Client wants to like/unlike a message:</u>

- the client will check its room_node to see if the like/unlike is valid (not liking a message the same user wrote) by converting the integer to lamport timestamp and checking if the user id of the msg_node with the same timestamp is the same as the user currently logged into the client
- the client will check its room_node to see if the like/unlike is nonredundant (not liking/unliking the same message twice) by checking the mesg_node’s like_list for the presence/lack of the user id
- if the user is not trying to like their own message or a message it has already liked or unliked, the client sends a TYPE_LIKE_MSG with the appropriate like_state to its server
- when the server receives a TYPE_LIKE_MSG rom a client, it will:
    - increment its timestamp index and stamps the TYPE_LIKE_MSG update
    - write the TYPE_LIKE_MSG update to file
    - insert the TYPE_LIKE_MSG update into roomslist
    - update its index in its anti-entropy vector with the new timestamp counter value, as this is a new message
    - send the TYPE_LIKE_MSG update to all their clients in Spread groups CR<server number>_<chatroom name> and all other servers in its partition
- when a server receives the TYPE_LIKE_MSG from another server, it will
    - check the index for the server it recieved the message from. If the received timestamp is less than the stamp in the checked index, the TYPE_LIKE_MSG update is thrown out. Else, the server will:
    - adopt the index if it’s greater than the server’s personal timestamp index
    - write the TYPE_LIKE_MSG update to file
    - insert the TYPE_LIKE_MSG update into roomslist

    - update the sending server’s index in its anti-entropy vector, as this is now the latest update received from that server
    - send the TYPE_LIKE_MSG update to all their clients in Spread groups CR<server number>_<chatroom name>
- clients in Spread groups CR<server number>_<chatroom name> receive the TYPE_LIKE_MSG from their servers, they will
    - update their roomslist by inserting the message
    - reprint the room

TYPE_LIKE_MSG update as it would appear in servers’ update files:
<timestamp server ID> <timestamp counter> <type = TYPE_LIKE_MSG> <like_state> <chatroom name> <author user id> <timestamp server ID of of message being liked/unliked> <timestamp counter of message being liked/unlike>

<u>Client requests entire chatroom’s history:</u>

- client sends its server the request with TYPE_PARTITION_REQUEST
- the server finds the correct chatroom in its chatroom messages data structure
- the server iterates through all messages in the chatroom
- for each message, find the number of likes
- for each message, construct a string in the format of <author>: <message> Likes: <# likes>
- send the messages to the client to display

<u>Client requests current partition view</u>

- client sends its server the request with TYPE_HISTORY
- the server looks at other members of its ServerGroup
- the server sends the client the server IDS of all members in its ServerGroup
- the client prints these server IDS

6. RECONCILIATION ALGORITHM IN CASE OF MEMBERSHIP CHANGE

Server starts up/merges:

- server joins the Spread group of servers in its partition
- all servers in the group receive a Spread update of this membership change
- anti-entropy vectors int vector[5] are sent to and received from all other servers in the same Spread group of servers
- for each index 1-5 in the vector, if not all servers have the same value (latest update) for that index (server), the server with the greatest stamp (and in a tie, smallest ID) sends the missing updates to the server(s) without the greatest stamp
- write the received updates to file, update anti-entropy vector
- for each room_users_node, send and receive TYPE_JOIN_CHATROOM messages for each user_node's u_id and spread_id, inserting unique user_nodes that were unknown of beforehand
- construct the roomslist data structure with the contents of the now-updated update file

Client crashes/leaves a chatroom:

- the client's server receives a membership change update from Spread of the client leaving (if the client immediately joins a different chat room, the process described in section 5 will also occur)
- when the server receives the membership update from Spread, it will:
    - update roomuserlist with the new membership by deleting the user with the matching Spread id and user id
    - send a TYPE_LEAVE_CHATROOM message to CR<server number>_<chatroom name>, and all other servers in its Spread group
- when other servers receive the TYPE_LEAVE_CHATROOM message from another server, they will:
    - update their roomuserlist by deleting the user
    - send a TYPE_JOIN_CHATROOM message to CR<server number>_<chatroom name>
- when clients in the Spread groups CR<server number>_<chatroom number> receive the TYPE_LEAVE_CHATROOM message from their server, they will:
    - update their room_users_node to reflect this membership update by deleting the user
- all clients in that chatroom will reprint their rooms, because a user has left

Server crashes:

- all clients connected to that server will receive a Spread notification and be forced to disconnect
- clients will free their roomslist and roomuserlist

anti-entropy vector

Each server maintains an anti entropy vector for all servers, including itself that holds the highest update timestamp received from each server. Pictured below in fig. A is an anti-entropy vector at startup with no messages sent:

Fig. A

Server	1	2	3	4	5
Value	0	0	0	0	0

If all servers are in the same partition and server 1 has sent the only update with timestamp of 1, all 5 servers' vectors should look like fig B:

Fig B.

Server	1	2	3	4	5
Value	1	0	0	0	0

However, if server 1 splits into a partition by itself, and server 1 gets an update with timestamp 2, servers 2-5 will each have the same vector as in fig B, but server A's vector will look like fig C.

Fig C.

Server	1	2	3	4	5
Value	2	0	0	0	0

If these two partitions remain and server 5 sends an update, server 1's vector will remain as in fig C., but servers 2-5 will have vectors as in fig D.

Fig. D

Server	1	2	3	4	5
Value	1	0	0	0	2

If server 1's partition merges with the server 2-5 partition:

- each server 1-5 will send its vector to the others
- Server 1 will see that it is missing all updates stamped by 5 with timestamp counter 0 < x <= 2
- Servers 2-5 will see that they are missing all updates from server 1 with timestamp 1 < y <= 2
- Server 1 will see that it has the update(s) server 2-5 are missing, and send the update stamped with counter 2, server ID 1 to servers 2-5
- Servers 2-5 will see that they have the update(s) server 1 is missing. To avoid the case where 4 servers 2-5 send updates to

server 1, in this case, since server 5 is present, server 5 will send the updates server 1 is missing to server 1.

- In the case where server 5 is not present, only server 2 will send the update stamped with server ID 5 counter 2, server ID 5 because server 2 has the lowest server ID in this case.