Chat System
EN 600.437 Final Project

David Gong, Stephen Hamilton

Abstract—A distributed chat server system that is fault tolerant through the use of a lamport timestamp.

1 Overall Approach

This chat service is composed of 5 servers each hosting their own set of clients. Servers communicate with each other and their clients through spread utilizing multicasting with agreed ordering. A client connects to a server in order to begin utilizing the chat functions. Then the client selects a username which is a unique identifier (meaning other client process running with the same name will be treated as one person). Finally a client joins the the desired chat room to chat with other clients that joined that chatroom.

All operation ordering is done on the server side. The client simply makes requests for its desired information and the server sends it what it needs. Therefore, the client will not see that its request has been fulfilled until the server has completely processed it. There is no computation done on the client side. Rather, the only thing the client must do is maintain the linked list of messages which exist in the chat room of which it is a member.

In order to maintain global consistency across servers, we create a lamport timestamp for each request made by a client on the server they are connected to. By using a lamport timestamp which consists of a process ID and a process sequence number, we can maintain total ordering across all servers that are connected. Furthermore, the lamport timestamp makes it possible to retain consistent and agreed ordering across partitions and merges.

We also use a matrix to maintain knowledge of the most recent messages each server has from every server. Therefore, from the perspective of some given server, it knows what messages other servers are missing from not only itself but also from all other servers.

By using a lamport timestamp along with this matrix of the highest lamport timestamps received, we can acheive an eventually consistent state across all of the servers.

Next, in order to acheive resiliency in the face of crashes, we simply incorporate a file writing system such that after every update received by some server, whether that is from another server or a client, is written to a file (disk). Therefore, when the crashed process is rejuvinated, it opens the written file and and begins processing from its state prior to the crash, requesting any updates required through the matrix it sends.

2 Design

2.1 Assumptions

Below are our assumptions.

- A user is uniquely identified by their username.
- Users cannot communicate until they connect to a server.
- Chatroom names are only comprised of alphabetic letters.

2.2 Group Architecture

Four kinds of groups are created in order to properly manage this chat service:

- One server group for all of the servers.
- One group for each individual server.
- One group for each individual server and the clients connected to that server.
- Five groups per existing chat room, one for each server.

The group for all of the servers is a group that each server joins. The main use is for communication between the servers. By being a part of the group, servers can receive updates from other servers' clients through those servers. Furthermore, when a partition occurs (some set of servers lose communication with the other set of servers), then the servers know which servers it can still communicate with because those servers that remained in the same partition will see that they are members of the same group. This information is kept on each server, and is also used for when the client requests to see which servers are available.

The groups per server are used for communication from client to server. The server alone joins the group and a client will be able to send requests through this group to the server to which the client is connected. The messages passed through this group are text messages, likes, unlikes, display servers online, and request to join a group.

The groups per server and a server's clients are used for determining whether the server is available or not. A client joins this group when it wants to connect to the corresponding server. If the server is also in that group, then the client knows that the server is available. If a server were to disconnect, then clients would be notified that the server they were connected to became unavailable through a membership change notification through the group. At this point the clients are notified that their server is no longer available. Similarly, if a client connects to the server group, and the server is not in the group, the client is immediately notified that the server the user requested is not available.

The five groups per existing chat room, one per each server, represents the membership in a given chat room on a specific server. Therefore, when a client decides to join some chat room, it joins this chat room group associated to the client's server. From this, it becomes evident that it becomes the servers' job to maintain consistent communication within chat rooms across servers. The servers are also a part of these chat rooms in order to allow them to determine which clients are in what groups. This information is utilized to see which usernames belong to a chatroom. Therefore, if a client is suddenly disconnected, the server is notified that the client is gone, and therefore can let other clients know that the user has left the chat room they were in.

2.3 Message Packet

We only have one type of packet we send called a chat_packet. There is a field, "type",on the packet that helps distinguish the role which a given chat_packet fufills. The different types of packets are as follows:

0) Text chat.
1) Like message.
2) Connection to a server.
3) Receive message.
4) Tell client to display all messages after a given lamport timestamp
5) Request to join a group.
6) Tells the client to refresh the screen with updated messages.
7) Unlike message.
8) Display servers online.
9) Send client user list.
10) Send username/private name combination from server to server.
11) Remove username/private name combination from server to server.

Types 0, 1, and 7 are used both for communcation from server to server and between client and server.

These constitute the main actions that a client can perform in a chat room. A client can say something in a chat room or a client can like/unlike a message in a chat room. These different types help the client and server processes distinguish which one of these types of actions a given chat_packet is so that they can treat them accordingly. Specifically, when

a server receives any of these packets, it will insert them into the data structures (which are discussed in Section 4). If it comes directly from a client, then the server will insert them directly but if it comes from a different server, then the server must check the lamport timestamp and compare it to the value for the sending server in its own vector which will determine whether or not the message has already been received. When a client receives a message, it responds based on these types in conjunction with the mess_type that accompanies messages from Spread (determined by the server). These responses may occur in two ways, the client process may immediately update the screen with a new message or a set of new messages, or the client process may not display any changes and simply update the data structure. The immediate update of the screen when there are new messages to display.

Types 2, 5, and 8 are used for communcation between client and server.

These constitute the packets necessary for setting up a client on a server/chat room. A client thus has a way to tell a server that it wants to connect or that it wants to join a chat room on that server. The client will send these types of packets to the server it wants to join after checking if the server is up (by using membership groups as discussed). Then the server will respond with the same types of messages to confirm the membership requests so that the client can begin to initiate chat room actions. Type 8 messages are also related in that they have the specific role for requesting/receiving information on the available servers. A client may ask for the current availability of servers in relation to the one it is connected to by asking that server. The server responds with a type 8 message with the available servers as a string.

Types 6, 8, 9 are used for communcation from server to client.

These constitute the updates that a server must provide a client in order to provide a consistent chat room experience. The server can send updates to the client process that do various things, but in general just keep the client updated on specifically messages and likes from other clients and user changes in a group. Specifically, type 6 messages accompany like updates and new messages that have been inserted in between current messages on the client due to a merge.

Types 10 and 11 are used for communcation from server to server.

These are specifically for maintaining a consistent view of chat room membership across servers. The servers have these types used specifically for communicating changes in users in all of the chat rooms. Specifically, these packets are used after a membership change amongst the servers. When a membership change occurs, the members in a chat room which any given process can communcicate with changes because they are from that point able/unable to talk with them anymore. Type 10 is used to tell connected servers to add some user to the data structure (more on this discussed later). Similarly, type 11 is used to tell them to remove some user.

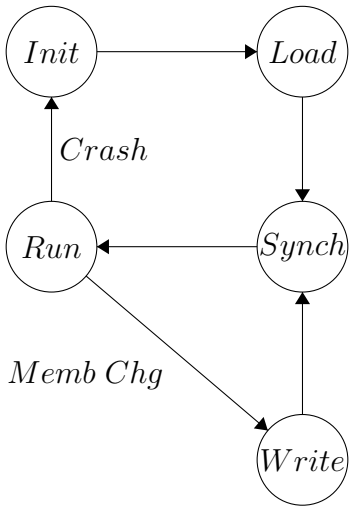
2.4 Matrix of Received Updates

In order to make the lamport timestamp method, each server process maintains a matrix where each of the columns, i, are the latest received "vector" from the corresponding server i. These are communicated between servers after a membership change amongst the servers. That way all of the servers in some partition will share the most possibly up-to-date information all servers in the chat service. Each component, j, in a vector i is a lamport timestamp value for the highest lamport timestamp received from the server j. Therefore, if the 2nd column (the second server's vector), has 84 in the 4th component, that means that, amongst all of the processes in the given partition, the servers know for certain that server 2 has messages from server 4 at least up to a lamport timestamp value of 84.

The point of this is so that after a partition and merge, servers know what information they must share with each other in order to reach a matching state within the partition. Put simply, the servers in a merge check to see what the lowest lamport timestamp in each row is and then the server with, first, the highest

lamport timestamp for that row, and second, the lowest server ID, will send all the messages it has from that lowest value to the highest it has to the server group it is in. More on this is discussed in the discussion of the algorithm later on.

2.4.1 Server State Machine



2.5 Server Data Structures

The server has two primary data structures. An array of linked lists for all user updates and a linked list of all chat rooms each with a linked list of the messages from that chat room in lamport timestamp order.

2.5.1 Array of Linked List

This is an array of length 5, each cell corresponding to a linked list of one server's updates. The updates received from a given server are linked together in no particular order along the designated cell of the array. This data structure is used alongside a matrix which carries the highest lamport timestamp received by each server for each server. This makes it so that when there is a merge or partition, the servers know what updates other servers do not have but need.

The array of linked lists is also useful for achieving fault tolerence. A server can write all of the data in this data structure onto disk and simply come back online by first reading all of the updates and then properly updating its matrix. With this, a server can come back to the exact state with which it crashed.

2.5.2 Linked List of Chat Rooms

A second data structure is used in order to keep track of messages (in order) in every chat room. At the highest level, this is a linked list of chat rooms in no particular order. These chat rooms each contain the name string of the chat room it represents and two linked.

There are two linked lists which have different purposes. First, there is a linked list of names used to keep track of the clients (and their usernames) who are a part of that chat room. Each of these names has a name string, a pointer to the next name, and a pointer to a linked list of private names which are unique to each client process. This linked list of private names is used to keep track of the client processes that share the same username (this goes along with the assumption that usernames are unique). This way, when there are no private usernames left in the list, then it is definite that there are no users in that chat room with the associated username. When this occurrs, the username is removed from the chatroom, and a message is sent to the client with the new memberhsip list. Also a removal message is sent to all other servers with that private name, so that each server can remove it from their attendee lists.

The other linked list at the second level is the actual messages sent on that chat room. This linked list is sorted by increasing order based on lamport timestamp. These message nodes therefore have a chat_packet, a pointer to the next message, and a pointer to a list of likes. The mentioned linked lists of likes from each message contain the name of a user who has liked (or then subsequently unliked) the message, a "like" field which determines whether or not that given user is still liking the message, and a field for the lamport timestamp of the highest like/unlike message from the given user.

The incentive behind having this data structure is that a client process will only ever be in one chat room at a time, therefore it will only want updates that pertain to that chat room. With this three tiered linked list, a server can send a client process the updates for a chat room for a single room without difficulty by

iterating through the correct chat room's linked list of messages. And then when sending updated messages to the clients, the server can count the number of actual likes by traversing a given message's linked list of likes and send that as part of the message. Therefore the client does no computational work, it simply figures out where the received packet goes in its data structure which is discussed later.

2.6 Client Data Structures

The client only has one data structure. A double linked list of the messages that have occured in the chat room to which a client is a member. We utilize a double linked list in order to traverse back to the last 25 message for display purposes. These messages are sorted in lamport timestamp order. Each of the nodes has the number of likes associated with that message as a part of the message data. This makes it very easy for the client process to output the contents of its chat room. One simply iterates through the nodes and prints out the username of the message's author, the text, and the number of likes. The link list is destroyed each time a user leaves the chatroom, and a new one is created when the user joins a new chatroom. When new messages arrive from a server reconciliation, we may get messages that are not in order. These messages are simply inserted into the list based on their lamport timestamp. If they have the same LTS as a message in the linked list, it is simply ignored, however if it is in between two messages in the linked list, it is stitched into the list to maintain complete LTS order for the chatroom.

2.7 Client Operations

The client begins with a menu screen allowing the user to connect to a server or change username. Once the client connects, the menu gives the rest of the options to the user. The user can then connect to a chatroom (if they have set their username), and begin adding messages, liking messages, removing likes, and print history. The user may also view the chat servers available at any time while connected to the server. Each operation sends a request to the server public group, and the response triggers the client screen update. The screen is cleared when connected to the chat room, and it is redrawn upon updates. Line numbers are created for each message in a chat, which allows the user to specify messages to be liked or unliked. The line numbers continue indefinitely, which allows the user to like a message back in history prior to the last 25 that are printed out. We realize that this is not actually suitable for a system that will run over a long period of time, since the numbers will be too long for users to type in, however we wanted to have the functionality to like all messages in the chat history. If we were to program a GUI as the client, the numbers could still exist, and the user would simply click like on the message, which would correspond to the LTS associated with the message. For chatroom attendees, we simplified the client side by displaying text built by the server. We realize that this has the limitation of the text line length. If we had a requirement for how many people could be in a chat room, we might create another data structure on the client, and have the server send a message for each user and the client can maintain a linked list. Therefore no matter how many updates are made to the attendee list due to partitions or servers going down, the client only needs to receive one message with the attendee list to display correctly for the user. As messages arrive, they are placed at the end of the chatroom linked list, unless the mess type is 13, which means it came from a merge and it needs to be placed in correct LTS order.

2.8 Algorithm

2.8.1 Regular Case

In the regular case, each server starts and has no log file. As each server comes online, they share with the server group their matrix utilizing a spread mess type of 10. This is a matrix of lamport timestamps of the latest update from each server. Therefore a server will have the latest timestamps from itself, along with the latest timestamps of messages it has received from other servers. This is key, because if two servers join after a partition, one may know about

messages that were sent from a third server, and thus can send the third servers messages to ensure the latest knowledge is shared within their partition. Although we send the entire matrix, in our implementation we only utilize the vector from each server to send updates. We sent the entire matrix in order to trim messages received by all servers, since we would no longer need to resend these messages. This could be implemented by utilizing this matrix.

Initially these timestamps are zero. As clients join servers, they create messages, and the servers assign each message an LTS, saves it to its log, and sends that message to the chatroom on its server, and also sends the message to the other servers so they can update their respective chatrooms. If a user "likes" a message, a chat message is created with the LTS of the message they like, and that is sent to the server. The server then updates its data structure by finding the message, and either adding a likes struct to that message for that user, or finding one that exists and checks to see if the LTS is greater or not. If it is greater it updates the LTS, and changes the like to one. The same occurs for an unlike, however it just updates the value to 0. Although the server maintains the linked list of names that like a message, when the server updates the clients, it simply updates that message with the number of users that like the message. This makes it simple for the clients to display the number of likes without maintaining a user list for each message.

When a server crashes and is restarted, it begins by reading from the log file. As it reads, it updates its matrix of LTS timestamps. When complete, it joins the server group, which triggers all servers to send their matrix. It will immediately receive updates for all the message it does not have from the server that has the highest LTS of messages in the matrix for each server. It will process these updates, and if any clients connect during this time, they will be given the updates as well by the group they are connected to for their chatroom.

2.8.2 Reconciliation Case

Each time a membership change operation occurs in the server group, each server begins a reconciliation process. If a server leaves the group, each server begins removing all users from the chatroom data structure for that server. Then an update is sent to the client to update the chatrooms with the new membership list of the rooms. Therefore every user will know that the users that were on the server that went down are no longer in the chatroom.

When a server joins the server group, this triggers each server to send their matrix of latest LTS timestamps for each message they now about. Since the membership change contains how many servers are in the group, each server will wait for the correct number of matrices from each other server before sending any updates. Furthermore, the receiving servers will filter through the received matrices. This is done by keeping an array of received matrices and marking ones already received for a given merge. This is done in order to ensure we get a vector from each server in the new group. If there are two merges in a row to a single group, then the servers in the original group will end up sending their matrices twice. They may receive each other's more than once and if we solely rely on the number of matrices received, then we may end up neglecting some servers that joined in the second merge. Once they are all received, each server computes the minimum and maximum LTS for each server. If the server finds that their server ID contains the maximum, it will send all messages after the minimum LTS (regardless if they are the owner of the messages). Servers will disregard any message it receives if it has a lower LTS than what is received. Therefore, each server will now have the latest messages after the join operation.

As each of these update message arrive at the server, they are placed in the server array/linked list structure in order, and in the chatroom linked list by LTS order. In addition, each packet is also sent to the chatroom group on that server that it belongs to. This allows each client to receive the message, and place it in its data structure in LTS order. Once this occurs, it will refresh the screen to display that message and all after it, to inform the user of the changed messages and/or likes.

In the case that a message is liked, but the message that was liked was not yet received by the server, the server will create a message that is marked as not received, and the like is attached to that message. This placeholder message will not be displayed until it is marked received, however it is put in the correct place in the linked list based on the timestamp the like refers to. When the referred to message is received, the data is copied into this placeholder message, it is marked as received, and the like will be on that message. This creates eventual consistency, since likes refer to messages that potentially may not be received due to partitioning.