Final Distributed System Design Document

The overall approach

A chat server and chat client programs.

The chat server could be run in two modes. In the first mode, no parameter is given and program is run by "./server". In the second mode, a single parameter "r" is given and program is run by "./server r". The first mode will start server from scratch. The second mode will recover from a local file. Client program could be run by ./client

net_include.h is all the header files. linkListFunc.c is a c file for all functions.

The group structure

1.Each server has a public group – a client connects to the server by this public group
2.Each server has a private group
(1)When remerging happens, this private group is used to identify servers in the opposite partition
(2) A client will join its server's private group. When the server is down, the client will receive a membership message.
3. For each chat room created, each server has a replicated version of this chat room.
As a result, each chat room corresponds to five groups on servers. For example, chat room 1 corresponds to group "1" "1001", "2001", "3001" and "4001".
4. For all the servers, we have a group. This group is used for anti-entropy algorithm.
5. Each client has a private group. This group is used for command "h" and "v".

The type of messages being used

In our program a uniform packet is used. They can be classified as follows according to their member variable command. The server to client packet and client to server packet structure are the same for the same command.

1. Server keeps track of its user
   1.1 Packet c. Each user is identified by its name and connected server name. When reconnecting, a user's connected server name needs to be reset.
   Contains: char command// identify type of message
   char* userName // used to locate the previous userNode
   int currentChatroomID // used to locate the chat room user is in
   char* serverName // used to create the new userNode
   char* previousServerName // used to locate the previous userNode
   1.2 Packet u. Each user is identified by its name and connected server name. When switching user name, the user name needs to be reset.
   Contains: char command
   char* username
   int currentChatRoomID
   char* switchUserName // used to locate previous userNode
   char* userName // used to create new userNode
2. Server keeps track of chat room
   Packet j. When user enters a chat room, if it does not exist, then it should be created.
   Contains: char command
   int currentChatRoomID // previous chat room, chat room to leave
   int joinChatRoomID //chat room to join

char* userName //used to create new userNode
char* serverName // used to create new userNode

3. Server keeps track of message and likes
   3.1Packet a. A client says a new message
   Contains: char command
   int currentChatRoomID
   int messagePayLoad // the message client says
   char* userName
   3.2 Packet l. A client likes a message
   Contains: char command
   int likeLineNum // line number seen by the client
   char* userName
   int currentChatRoomID
   3.3 Packet r. A client dislikes a message
   Contains: char command
   char* userName
   int likeLineNum
   int currentChatRoomID
4. Clients' requests for history and server view
   4.1Packet v. Client requests for partition info.
   Contains: char command
   4.2 Packet h. Clients requests for past history.
   Contains: char command
   int currentChatRoomID
   int userName
5. AntiEntropy vectors
   Packet p. anti-entropy exchange vectors.
   int antiEntroVector[5];

The data structures

Server side:

It is composed of two main parts: a multi-server updates tracking array and a server link list.

A multi-server updates tracking array contains link list header for 5 link lists, each of which represents a server. Each node in the link list is a packet.

A server link list contains a list of chat rooms. Each chat rooms contain a list of users and messages. Each message node contains a list of users who like this message. The message link list is bi-directional. All other link lists are all one-directional. Each user node contains the name of the user, the server it is connected to and an active flag to show whether it is active. Knowing which server a user is connecting to is useful in case of network partition. By comparing the current partition view with the server a user is connecting to, we can decide whether the user should be displayed in this chat room. If yes, we set the active flag as 1. Otherwise, the active flag will be set to 0. Each like node contains a user name and the lamport stamp of the message being liked. This is especially useful in case of partition remerging. Only associating likes/dislikes with messages by line number is not enough. Since each message has a unique lamport, we could embed this lamport in like node to identify the message to be liked or disliked.

Client side:

The data structures for client side are quite simple. It only needs several state variables such as connected server, current chat room and user name.

The algorithm in the regular case

Packet flows:

1. A server receives a packet from a client

Each time a server receives a 'c','u','a','j','l' and 'r' type packet from its client. The server will first add a lamport stamp, and then multicast these packets to its server peers. Next it will apply the updates to its local data structure. Finally, for 'a','j','l' and 'r' type of message, the server will reply these packets to clients directly connected to it according to the updated local data structure.

Each time a server receives a 'h' and 'v' type packet from its client, the server only needs to reply to the clients' private group directly.

2. A server receives a packet from a server

Each time a server receives a 'c','u','a','j','l' and 'r' type packet from its peer. It will first need to apply the updates to its local structure. Then for 'a','j','l' and 'r' type of message, the server will reply these packets to clients directly connected to it according to the updated local data structure.

BTW, for 'l' and 'r' types of message, they have two lamport stamps: one for themselves and the other for the message they are associated with. The latter one is set in the first time that packet arrived at server side because the first time the line numbers are consistent from both a client and its connected server's view.

For each packet received, the server will write each message to disk by appending them in a hard-coded file. Then when the server restarts, it only needs to apply the updates one by one until finishing reading the file.

The reconciliation algorithm in the case of membership change

Each time a network partition change is detected, the anti-entropy algorithm will be activated. After exchanging lamport vectors, for packet from each server, the server who has the maximum number of packet lamport and lowest server index will be responsible for sending that. According to the previous server view, it can only send packets to those who are not in the same partition before.

To give an agreed order of message being displayed, the message link list is ordered according to the lamport stamp. The like/dislike message will only be applied on the message who have the same lamport stamp.

When a server crash, since all its clients are in server's private group, by looping through the group member list, they will know that server is down.

When network partition happens, we need to make sure that some users should not be displayed when a new user joins this chat room simply because they are in a different network partition. To accomplish this, each time network partition happens, we will reset the active flag in user nodes. As a result, when a new client enters the chat room, the server will only display users who have TRUE active flags. And when partition merges, the users in same partition will be reactivated again.

Similarly, when a server recovers itself from crash, all the users should not be present in the chat room because the connection is lost. This is done by writing a special flag called recoverStates in the recovery file. Each packet obtained from the recovery file will have this special flag. As a result, these packets will be ignored interms of adding users to or removing users from chat room.