Design Document for Final Project

600.337 Distributed Systems

Sihao (Ben) Lu

Yixiao Wu

Final Draft

* Grouping

We create four basic group types.

1. One public group containing all servers in which the servers inter-communicate. The name is like “servers”.
2. Every server has a special public group containing only itself for receiving messages from the clients connected to it. The name is like “server\_x” where x ranges from 1-5.
3. A public group for every server and the clients that are connected to that server. No one send or receive using these groups. They exist for testing memberships. The name is like “server\_x\_clients” where x ranges from 1-5.
4. Every server has a special public group for every chat room in it. This makes it easier to reply to clients in the same chat room. The name is of format “server\_x\_chatroom\_y” where x ranges from 1-5 and y is the name of the chat room.

* Data Structures

1. Note: ArrayList here means the expandable array structure. We wrote a “vector.c” implementation to supply the need.

* Every request that contributes data to the server is called an “update”. Server keeps all of its updates received into a log so that they can resend them if needed.

The data structure of update is followed.

typedef enum {  
 ADD\_LINE, LIKE\_LINE, UNLIKE\_LINE, JOIN\_ROOM,

HISTORY\_REQUEST, VIEW\_REQUEST

} UpdateType;

typedef struct {

type: UpdateType

user: char[MAX\_USER\_NAME]

room: char[MAX\_ROOM\_NAME]

line: char[MAX\_LINE\_LENGTH]

line\_num: int

sender\_index: int

timestamp: int

} Update;

* How do clients compose and send updates
  1. Update Composition

1. Common

There are six types of updates. The first four are “real” updates, meaning they will change the state of the server. For add line, like line, unlike line and join chat room updates, the client sets the type field to the corresponding type, sets user field to its logged in user set by “u” command, sets room field to the chat room name defined by “j” command. The last two does not change the state of server, so we call them requests rather than updates. But for the sake of uniformity, we still wrap them inside the Update packet to give the server a consistent structure of parsing.

Every update sent by the client sets its own sender\_index to 0 and timestamp to 0. The servers reserve these fields.

1. Add line

Fields set: type, user, room, line

1. Like line / Unlike line

Fields set: type, user, room, line\_num

1. Join room

Fields set: type, user, room

* 1. Send Update

After composition, the clients multicasts the update inside “server\_x\_clients” spread group. The client itself doesn’t keep any record of the updates.

Clients send update to their servers and server process them, mutating their own data structure inside, and reply to the clients using the Reply data structure as described below.

typedef enum {  
 ReplyAddLine, ReplyLikeLine, ReplyUnlikeLine, ReplyUserJoinRoom, ReplyPrintHistory, ReplyPrintView

} ReplyType;

typedef struct {

type: ReplyType

item\_count: int

content: char[]

// content layout 1: room\_name, members[], senders[], lines[], like\_counts[], changed\_user, success

// content layout 2: server\_layouts

} Reply;

The Reply is dynamically laid out. Every Reply type except Print View contains the recent history of the chat room. The number of lines is specified in item\_count. In this type of layout, the reply contains the chat room name, the members of the room, the senders of every line, the content of every line, the count of likes of every line. If the reply is of type Join Room, the changed\_user will contain the recently joined user’s name, so the clients can pop up “XXX joined the chat” according to this piece of information. If the reply is of type Like Line or Unlike Line, the success field will contain a single Boolean value to indicate if the operation is successful.

* How do servers process the updates from client and make replies

1. Upon receiving an update, it checks if it comes from the client. If not, it executes some other kinds of mechanism other than this.
2. The server adds line, likes line, unlikes line or join user to the chat room for the user who sent the update (if the update is of the first 4 types). The detailed description of how to mutate the data structures within is discussed at section 2 below. Then it prepares a Reply and multicasts it inside the “server\_x\_chatroom\_y” spread group to make sure every client who are in the same chat room and who connects to that server gets the reply.
3. The server reads the history or reads the view (if the update is of the last two types) and prepares a Reply. Then it sends the reply to the private group of the client who sends it.
4. Servers keep their own data structure to store the current state:

chatrooms: ArrayList<Chatroom>

typedef struct {

name: char[MAX\_ROOM\_NAME]

members: ArrayList<char[MAX\_USER\_NAME]>

lines: ArrayList<Line>

} Chatroom;

typedef struct {

content: char[MAX\_LINE\_LENGTH]

sender: chat[MAX\_USER\_NAME]

likes: ArrayList<char[MAX\_USER\_NAME]>

} Line;

* How do the servers mutate their inner data structures

1. Upon receiving the Join Room update, it finds if the user is already in one of the chat rooms, if so it removes the old user. After that it creates a chat room if no specified chat room exists by allocating new struct of chat room and setting the names, the members and lines to empty array, or it just gets the existing chat room with that name. Then it adds the user to the members list.
2. Upon receiving Add Line update, it finds the room first, and creates a Line struct by specifying the content to the line string, the sender to the user name and likes to empty array. Then it inserts that Line struct into the lines array inside the corresponding chat room.
3. Upon receiving Add Like update, it first finds the chat room, and then it checks if the requested line number is out of range. If the line number is valid, it searches the likes list in the line for the user name. If not found it adds that to it, otherwise it reports an error (the user cannot like a line twice).
4. Similar for Remove Like update, the difference is that the server reports error when it doesn’t find the username (the user cannot unlike a line that is not liked before), and it adds username into the likes list of that line in the chat room.
5. For Print History and Print View requests, no inner data is altered.

* What happens inside when you type a command
  + Environment initialization

The server joins all four groups as discussed in section “Grouping”. When every server receives 5 membership message from “servers”, it begins to run.

* + Set user (u <username>)

The client changes a global variable containing the user name. Does not do anything else.

* + Connect to server (c <server index>)

The client disconnects from old server (if any) by leaving the old server-client group and it joins the new group. If it has already joined a chat room, disconnect from it first.

* + Join a chat room (j <chat room name>)

The client disconnects from the old chat room (if any) by leaving the server-chatroom group and it joins the new group. Then it sends an update of Join Room to the server it connects to.

The server process the update as discussed before and sends the reply back to the clients connected to it. It resends the update to all the fellow servers. Details are discussed in the next section.

* + Add a line (a <line content>)

The client sends an update to the servers. The server processes it, replies to the client and resends the update to all servers.

* + Like / Unlike a line (l/r <line number>)

The routine is similar to the previous one. The difference is that the operation may be unsuccessful so it includes a Boolean to tell the client if it succeeded or not. It only resends to all servers if the operation is successful.

* + Print history (h)

The client sends the request to the server; the server collects all lines in the chat room and replies to the client. No resends happens.

* + Print view (v)

On every membership update of the spread group “servers”, every server keeps a record of the latest group members. It sends the members currently observed to the client.

* Synchronization and history keeping
  + Every server has a Lamport timestamp vector for the lastest timestamp of message received from each server, including itself.
  + Upon receiving an update, if the update is from a server but not the client, the server updates the timestamp vector corresponds to that sender server to the timestamp the update contains only if the timestamp is newer; otherwise it does not process this update.
  + If the update is indeed newer, the server tries to apply the change to its inside data structure. If the update is valid, meaning it does not like some line twice or does similar illegal operation, the server copies it and keeps the copy in the history.
  + If the update is from the client, the server increments its own timestamp in the vector, set the sender\_index of message to the server index of itself, set the timestamp to the timestamp corresponds to itself, and multicasts the update in “servers” spread group. In this way, every server is synchronized.
* Strategy to handle network partition and remerge
  + The server does nothing when the network is partitioned. The clients disconnect from the server if it is partitioned to a different network from that of the server.
  + As we talked about earlier, every server maintains a vector containing the each server’s last update. On remerge, the “servers” group changes the membership, every server sends each other their vector and every server keeps every server’s vector, including the vector of its own.
  + Every server scans through the five vectors from index 1 to 5. For every same index position i of each vector, it finds out the largest one. If the vector that contains the largest index in some position is the server’s own vector, the server is responsible for resending the updates that were once sent by server i. In another words, every server finds out the server that has the most updates sent from server i and resends the updates it kept inside the history if the server that knows the most is itself. i is ranged from 1-5 so this procedure repeats for five times, once each for the updates originally sent from each server.
  + Maintain total order: The new Add Line update received during the remerge needs to be ordered. Thus every line has a Lamport timestamp associated with it. When inserting the new lines the server inserts them in order so that the lines appear exactly the same on every server.
  + Solve “answer comes before question” problem: We have a hashmap that maps from an integer (line number) to an array of strings (likes) for any line that doesn’t exist. When adding such line, it checks if there’s a value in the hashmap. If so, it set the likes to the value described above.