Design Document for Final Project

600.337 Distributed Systems

Sihao (Ben) Lu

Yixiao Wu

Latest Draft

* Grouping

1. One public group containing all servers.
2. Every server has a special public group containing only itself for receiving messages from the clients connected to it.
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.

* Data Structures

1. 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.

Update {

Type: enum {SEND\_LINE, LIKE\_LINE, UNLIKE LINE, etc}

Timestamp: Int[2]

Content: char[]

}

Clients send update to their servers and server process them, mutating their own data structure inside, and reflect to the clients upon request.

1. Servers keep their own data structure to store the current state:

Chatrooms: Chatroom[]

Chatroom {

Lines: String[]

}

Line {

Content: String[]

Sender: User

Likes: String[] // List of names of users

}

* Strategy to sync
  + Upon receiving an update, every server refreshes its own state and resends the update in causal order to every other server. To realize causal order, every update to be resent will contain a Lamport timestamp.
* Strategy to handle network partition
  + Every server maintains a vector containing the each server’s last update. On server group membership change, every server sends each other their vectors and then some server will find some other servers are out of sync. We bring all the servers in to synchronous state by doing the two steps below.
  + If server A receives server B’s vector, by comparing Lamport timestamps it finds out that server B’s last update from server A is older than the last update from server A, it sends all the missing updates (the updates A received by B didn’t) to B.
  + In some situations, some sets of server S in the group when network was partitioned may not be there at the time when network remerges. To solve this, we choose the servers with most knowledge of servers in S to resend the updates to all the other servers that missed updates from S.

First Draft

* The set of messages and likes, etc that clients send to the server are represented as events. Below is the data structure of the event.

Event {  
 type: Int

Timestamp: time

message: char[]

}

Event can have different types: User Join, User Quit, Add Message, Add Like, Remove Like and Remove Message, Add Chatroom, Add Like, Remove Like. The reason we have a Remove Message event even if our program doesn’t contain that function is stated as follows.

* The servers replicate their states using the Paxos replication algorithm. Each server keeps the data structure as follows:

Chatrooms: Chatroom[]

Chatroom {

Lines: Line[]

Users: string[]

Messages: string[]

Likes: Hashmap<Int, Int> // Maps from message index to number of likes

LikeLists: Hashmap<Int, string[]> // Maps from message index to the list of users who liked that message

}

* The servers process requests and commit them by unit of event. Every server **maintains a log of all events** ordered by time. This makes them easier to handle different faults. Say there’s a user Jack. He logged in and joined “haha” chatroom, he send “I am happy” with index 3 and liked the message at index 2. Here’s what events will happen inside the server.

User Join: Jack; Add Chatroom: “haha”; Add Message: “I am happy” from Jack; Add Like: 3 from Jack.

The server that received message from Jack will then commence Paxos protocol to try to sync the changes with other servers, while changing its own data inside its data structure and present to the clients when they need.

* When network replication occurs, servers in each partition of the network continues to function until the network remerges. Then the servers do such things to synchronize themselves:

1. Trace back their log to find the last common event they share, which should be the last event in sync before partition.
2. Undo all the events in each of the server up till that event.
3. Merge the list of events from servers among different network partitions during the time of partition.
4. Redo the merged list of events in every server.

This should fix the asynchronies among different partition and the new state looks as if the partition has never occurred.