Chas Forelle

Final Project Design Document

0. Overview

I chose to utilize a combined linked list data structure for both servers and clients, in addition to an update file for the servers. This allows my servers to build the data structure state to better serve updates to new clients, but is also easy to process when sending updates between merges/partitions. An anti-entropy vector is used by the servers to synchronize updates across merges/partitions.

I. Data Structures

Data Structures for Data Transfer (located in packet.h)

Several simple structs and integer constants are implemented to facilitate the transfer of data between clients and servers. The different types of transfer constants are: **TYPE_JOIN_CHATROOM, TYPE_LEAVE_CHATROOM, TYPE_SEND_MSG, TYPE_LIKE_MSG** and **TYPE_PARTITION_STATE.** Each transfer constant corresponds to a struct that performs data transfer for each function. Each type of transfer struct can be sent from a client or a server, and contains a constant **SOURCE_CLIENT** and **SOURCE_SERVER**.

1. Struct *join_chatroom* contains **u_id**, **sp_id** and **chatroom** fields, in addition to its type and source. **U_id** contains a client's chosen username, **sp_id** contains their spread private group and **chatroom** is the name of the chatroom they are joining.
2. Struct *leave_chatroom* contains **sp_id** and **chatroom** fields, in addition to its type and source. **Sp_id** is the client's private spread name and **chatroom** is the name of the room they are leaving.
3. Struct *lamport* is a lamport timestamp container, which holds a **server_id** and a **index**, and is used to compare messages and likes to determine causality.
4. Struct *typecheck* is used to parse incoming data, and only contains a **type** and **source**.
5. Struct *message* is used to transfer text messages from clients, and contains a **timestamp**, **u_id, chatroom** and **mess**. **Timestamps** are written by servers when received from a client.
6. Struct *like* is used to transfer new likes or unlikes. It contains **timestamp**, **msg_timestamp, u_id, chatroom** and **like_state**. The **timestamp** in this case is the like's timestamp, while **msg_timestamp** is the timestamp of the message the client wishes to affect. **Like_state** is a constant, either **LIKE** or **UNLIKE**.

Data Structures for Local Storage (located in room.h)

The data structure used by both clients and servers is a series of doubly linked lists. The topmost level is a circular double-linked list of **room_node** structs.

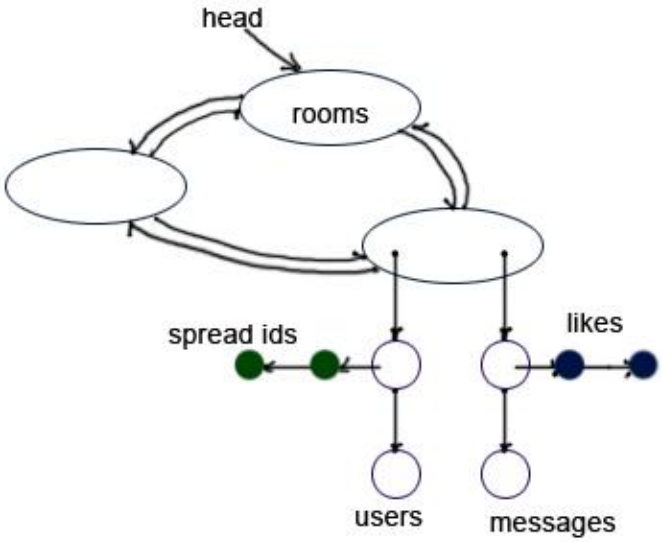
Room_nodes maintain a name, number of messages, number of users, **msg_node*** message list head, **user_node*** user list head, and next and prev pointers.

Every room has a list of **msg_node** that maintain the messages in a chatroom. **Msg_nodes** have a timestamp, mess, u_id, number of likes, **like_node*** list of likes and next/prev pointers.

Every message has a list of **like_nodes**, which maintain if a user has liked or unliked a message. **Like_nodes** contain a state, which may be **LIKE** or **UNLIKE**, a timestamp, the u_id that issed the like, and next/prev pointers.

Every room maintains a list of **user_nodes**, which keep track of any active users in a chatroom. A **user_node** contains a u_id, **sp_id_node** list to keep track of every unique client logged in with the same username, a counter of unique spread id's, and next/prev pointers.

Every **sp_id_node** is a simple wrapper for a **sp_id**, which contains a next/prev pointers in addition to its **sp_id** field.



List of Data Structure Methods

Int cmp_timestamp(stamp1, stamp2): This function is used to compare two timestamps, and will return -1 if the lefthand stamp is smaller, 0 if they are equal and 1 if the righthand stamp is smaller.

Get_num_rooms() returns the number of active rooms (has either users or messages) in the data structure.

Get_head returns the head node of the room_nodes.

Add_room(name) adds a new room if it does not already exist.

Rm_room(name) removes a room if it exists.

Find_room(name) Searches the top ring and returns a pointer to the matching room_node if it finds it.

Add_user(u_id,sp_id,room) adds a user to room.

Find_user(room_node*, u_id) returns a pointer to the user_node containing u_id, contained within room_node.

Rm_user(sp_id) searches all sp_id nodes and attempts to remove a unique client instance from the data structure. It collapses any parent user_nodes that might be empty as a result.

Add_msg(**timestamp, u_id, room, message**) adds a message into room with author u_id and timestamp that is generated by a server.

Find_msg(room, timestamp) returns a pointer to a message node that matches timestamps if exists.

Get_timestamp_from_index(room,line number) is used by the client to determine the corresponding message from a user-inputted line number.

Rm_id(user_node target, sp_id) searches a user node for the given spread id and removes it if found.

Add_like(state, u_id, timestamp, parent-message) adds a like onto the parent message with value of state.

Find_like(parent message, timestamp) searches a message node for a like with the same timestamp. Returns a pointer to the like node if found.

Find_like_by_uname(parent message, u_id) searches a message node for a like with the same username. Returns a pointer to the like node if found.

II. Spread Groups

Clients

Upon joining a server, a client connects to the server's client_group#. This allows the client to receive membership updates in the event the server crashes.

On joining a chatroom, a client connects to the server's CR#_<chatroom>. Each server maintains its own instance of each chatroom, in order to simulate clients that don't have access to a Spread network.

Clients send updates to the server's personal group, and the server then pushes the updates down to the chatroom group.

Servers

Servers create or join three groups on start-up: a ServerGroup, a client_group# and a server# group. The ServerGroup is utilized by the servers to pass updates between each other, and to receive membership updates when the network partitions and merges.

The client_group# is utilized by servers and clients, in order to receive membership updates when one of them crashes. Because a server is not in the chatroom groups, it would otherwise have no indication that a client had crashed and would not know if it was safe to remove them from the data structure.

The server# group receives updates from connected clients. Only the servers are included in these groups, and they function as a pseudo-private group that clients know the name of prior to runtime.

III. Client Algorithm

The client interfaces with the user via a command prompt interface. A user may set their username, connect to a server, join a chatroom and like/unlike messages with various commands. Additionally, diagnostic network status and chatroom history options are available. A user may not join a server without a username, and they may not join a chatroom without having connected to a server. Once a user has connected to a chatroom they may send messages and likes/unlikes. A user may view the full chat history once connected to a chatroom, and they may view network partition status once they have connected to a server.

User changes username

A user may change their username at any time, but if they are in a chatroom they will be disconnected from it.

User changes server

A user may change servers, but will be first disconnected from the one they are logged into. The client creates a LEAVE_CHATROOM update and sends it to the server before the local data structure is purged in preparation for a new server connection. The server pushes the LEAVE_CHATROOM update to the other servers and the local chatroom.

Server crashes

When a server crashes, clients receive membership change via the client_group. They purge the data structure and reset the chatroom, and ask the user to reconnect to a different server.

User writes message/like/unlike

When a user creates a new content update, the client wraps the data in the appropriate transfer structure and sends it to the server's 'private' group. The server then pushes it down to the chatroom and onto the server group.

User changes chatroom

A user may change chatrooms by inputting a new chatroom to join. The client disconnects from the prior chatroom and sends the server a LEAVE_CHATROOM update. It then purges the data structure and creates a JOIN_CHATROOM update for the server. The server pushes these updates to the server group and local chatroom, and serves the client with the latest 25 messages in the new chatroom.

IV. Server Algorithm

For normal operation the server either gets an update from a client or another server. If the update is from its client, it must send the update to the other servers. When it receives the update from a server, it processes the update like normal but does not send it over the ServerGroup.

Startup

On startup, the server attempts to rebuild its data structure from file if available. Otherwise, it initializes variables and connects to Spread.

User joins chatroom

When a user joins a chatroom, the client sends the server a JOIN_CHATROOM update. The server updates its local data structure and pushes the update out to the other servers, and the local chatroom the user is trying to join.

User leaves chatroom

When a user leaves a chatroom voluntarily, the client sends the server a LEAVE_CHATROOM update. The server updates its data structure and pushes the update out to other servers, and the local chatroom the user just left.

Message/like/unlike is received

When a user creates a content update they send it to a server's 'private' update-receiving group. The server then stamps the update, writes it to file and sends it through the ServerGroup. Then it updates its local data structure, chatroom and anti-entropy vector. It is important to do the steps in this order, because even if the server crashes in the middle it will still be able to recover the update from the file on reboot. Periodically each server sends out its anti-entropy vector to the ServerGroup, so that all servers are highly likely to be synchronized.

Network partitions

When the network partitions the servers will receive a membership change update. When this happens the servers will exchange anti-entropy vectors to determine the minimum value for each server. If the server is in the partition it is in charge of serving its own updates. If the server is not in the partition, the server with the highest vector value will be in charge of serving the updates if possible. If there are multiple servers with the max vector value, the lowest server id takes charge of sending the updates.

Additionally, the servers will send out their user room participation data so that they can all determine which clients had been cut off by the partition.

Network merges

When the network merges the servers will receive a membership update. When this happens the servers exchange anti-entropy vectors to determine the minimum value for each server. Just like in a partition, servers are responsible for their own value, their max value, and their max where they are the lowest server index (in that order). If a merge is interrupted by another merge part of the way through sending updates, the servers stop and re-send their vectors out to determine if the new partition is farther behind or not. This avoids sending too many updates twice.