Final Project Report

Wei Huo

The core of this project is to have weakly consistent replications from various servers such that no matter from which server the user log into the chat room, the chat history is as consistent as possible even with network partition. We can use the anti-entropy method to achieve this goal.

1 Algorithm

Each server needs to maintain write logs for each server in the group with the file name "<server_{1,2,3,4,5}>#{1,2,3,4,5}.log". The first segment denotes the server which maintains the files. The last segment denotes the server where the log comes from. The content in each write log is the write history received by the corresponding server and sent by the client which connects to the corresponding server.

When a server first receives a write from a client application, the server assigns a monotonically increasing accept-stamp = (lamport timestamp, server_Id) to the write. As it propagates via anti-entropy, each write carries its accept-stamp. Accept stamps define global order over all writes by all servers.

Each server R needs to maintain a version vector R.V such that R.V(X) is the largest timestamp of any write sent by the clients connected to X, accepted by X and known to R.

The anti-entropy method runs in the following manner. Each server will multicast its version vector to all the other servers in the same network component when a new server joined the group. Whenever a server S gets R.V from all the other servers R in the same network component, it compares the version vectors, figure out the minimum timestamp, traverses the write logs and send the missed writes to the receiving server. Whenever the receivers have received the writes, they update their version vectors and append the missed writes to the corresponding logs maintained by the receiving server.

Whenever the version vector is updated, the server needs to update the chat history accordingly. In our case, each server might have its own accepted writes which differ among servers. The write logs corresponding to each server are guaranteed to be properly ordered. However, the server needs to collect all the writes from all the servers and sort based on the accept stamp to make sure the chat history is properly ordered. The reason why we need to sort on each write is because a client might start the same chat room in a different network component, the chat history in both components need to be merged properly once the two components somehow merge into one. Since the server might not be informed of the older chat history, it has to keep the global order of the history by combining its own history with the older history and sort the writes to generate the globally correct history.

In order to record the number of likes for each message, we can do it in the similar way as regular writes, but tagged with the corresponding title. (Refer to the details in the data structure).

To update the chat history with the number of likes, the regular messages are tagged with the user name who likes it. In our case, all the relevant information is maintained in the doubly linked list. Once we know who likes the messages, it should be easier to avoid duplicate likes and get the number of likes.

Since we are using doubly linked list to maintain the data, it suffices to use the node as the head of the most recent 25 messages. The head is updated when the client first logged into the chat room or use the command “h” to acquire the most recent 25 chat messages by backward traversing from the tail of the list.

2 Data structure

The write log is in the following format:

```
<lamport timestamp, server_ID > chat_room {regular, like, unlike
} user_name: message
```

The message can only be liked by non-creators.

We are not writing the entire chat history to the disk since the chat history could be recomposed by reading from the write logs and sort accordingly in the memory. This is a simplification because the chat history might be so large such that the entire history can not be stored in the memory. In the realistic scenario, the chat history might need to be written to the disk. The oldest part of the history will be the part that is always correctly ordered and doesn’t need to be updated ever.

The version vector is an array of lamport timestamps for each server in the group.

We keep the entire history in the memory with doubly linked list.

```
struct Node{
    enum Type type;
    char stamp[MAX_MESSAGE_SIZE];
    char chat_room[MAX_MESSAGE_SIZE];
    char user_name[MAX_MESSAGE_SIZE];
    char message[MAX_MESSAGE_SIZE];
    char user_liked[MAX_MESSAGE_SIZE][MAX_MESSAGE_SIZE];
    struct Node *next;
    struct Node *prev;
};
```

We use various packet structure to helper various kinds of server-to-server and client-to-server communication.

2.1 `struct` Packet

for client-server communication.

```
struct Packet{
    char chat_room[MAX_MESSAGE_SIZE];
    char user_name[MAX_MESSAGE_SIZE];
    char message[MAX_MESSAGE_SIZE];
    enum Type type;
    enum ChangeType ctype;
    int status;
};
```

2.2 `struct PropPacket`

for propagating the content of the regular packets `struct Packet` to other servers.

```
struct PropPacket{
    enum Type type;
    char chat_room[MAX_MESSAGE_SIZE];
    char user_name[MAX_MESSAGE_SIZE];
    char message[MAX_MESSAGE_SIZE];
    char from_server[MAX_MESSAGE_SIZE];
    int from_server_id;
    unsigned long lamport;
};
```

2.3 `struct GroupPacket`

for propagating the version vectors and participants to the other servers

```
struct GroupPacket{
    unsigned long version_vector[5];
    struct Participant participants[MAX_PARTICIPANTS];
    int num_participants;
    int from_server_id;
};
```

2.4 `struct UserNamePacket`

for propagating the change of the participants to the other servers

```
struct UserNamePacket{
    char chat_room[50];
    char user_name[50];
    int from_server_id;
    enum UserNameType type;
};
```

3 Group architecture

We are implementing the communications based on group management in Spread.

3.1 All the servers need to be in the same group to propagate the messages.

The servers need to propagate the message when new servers joined the group, when the network partition happens, when the participant information are being exchanged among the servers, etc.

3.2 The client is in the group with the server it is connected to.

Since we need to present the current participants in the chat room, the client and the server it is connected to need to be in the same group. When the participant joined, left the group, the server will be alerted such that necessary operations will be conducted. The change of the user name could also be processed with this group management. Once the user name is changed, the older user name will be logged out and the new user name will be logged in. In my case, if the server or the user name changed, the client needs to send the request to the server first. The client cannot leave the group until the server left first and told him to leave. This is because the server needs to check for duplicate user name first. If anything wrong happened in this process, the state won't change and error message will be sent to the client so that the client knows what happens on the server side and respond properly.

3.3 The server itself is in a group.

The client needs to communicate with the server it is connected, however, they are not supposed to be in the same group, otherwise, the communication between one client and the server will be visible to the other clients.

3.4 All the clients in the same chat room are in the same group.

The server need to send message to all the participants in the chat room, thus, all the chat room participants need to be in the same group.

4 How to run the software.

Use the command ./client to start the client.
Use the command ./server server_id[1-5] to start the server. For example, if running the server on ugrad1, the command should be ./server 1.