Project 1 Report

Yuxuan Zhang, Yuhang Wang

Email: YUZ276@pitt.edu, YUW199@pitt.edu

Contribution: Both members have the same contribution to the project.

Overall design

- **Language: Go**
- We implemented a **bi-directional streaming** chat server based on **gRPC.**
  - `rpc ChatRoom(stream ClientRequest) returns (stream ServerResponse){}`
- We defined a **broadcast** function to send messages to **multiple clients** on every update.
  - On every **broadcast**, the server will send the **latest 10 messages** to every client in the group.
- All functions and features are tested and work properly.
- The client program automatically connects to the server without inputing `c localhost:12000`

gRPC Message and Service

Sercive

```
service Chat {
  rpc ChatRoom(stream ClientRequest) returns (stream ServerResponse){}
}

func (sm *Server) ChatRoom(ct Chat_ChatRoomServer) error{} // our service function
```

- After several attempts and careful consideration, we have changed our service design from multiple rpc sercive to a **single one** to support easier broadcasting funciton.
- On the **Client side**, the program will **read from the user input** and set the corresponding `Request` field inside the **client message** `ClientRequest` sent to the server.
- On the **Server side**, the program will execute different functions given the `Request` field.

Message

- We have to use **one message type** for different chat service (join group, login, like.....) and put **all field required** for the chat room in it.
- Therefore, **not all field will be set** for every message sent, we **only set and read** the **certain fields** for

certain function.

```
message ClientRequest{
  string request = 1;
  int32 user_id = 2;
  string userName = 3;
  string ori_group_name = 4;
  string new_group_name = 5;
  int32 like_number = 6;
  string Content = 7;
}
message ServerResponse{
  string request = 1;
  string username = 2;
  string content = 3;
  int32 like = 4;
  int32  msg_id = 5;
  int32 user_id = 6;
  string goodbye = 7;
  int32 local_message = 8;
}
```

Data Structure

Our data structure for storing the whole chat room data is complicated. We design a complex **linked list**.

Please refer to the `data.go` file for the whole data structure.

Server struct

```
type Server struct {
    UnimplementedChatServer
    GroupList *List
    TotalUser *List
    mu        sync.Mutex
    UserCount int32
}
```

- `*List` is a pointer to a linked list.
- We kept two linked list for the chat room:
  - `GroupList` to store different **chat room**
  - `TotalUser` to store all **connected client**, no matter which group they belong to.

Linked List definition

For the above `*List*` type, here are our definitions:

```
type Node struct {
    // below for message or username
    ObjectName string // the string of username or group name
    Content    string // for message
    Like       int32
    ObjectId   int32 // can be user ObjectId or group ObjectId
    MessageId  int32
    Next       *Node // Next node
    // below for group
    UserList    *List
    MessageList *List
    ct          Chat_ChatRoomServer
}

type List struct {
    Head     *Node
    TenStart *Node
    Count    int32
}
```

- Every `node` of the linked list `GroupList` we mentioned in the previous section represents a group. Inside every group:
  - we store a **linked list** `MessageList` , for every `node` inside the `MessageList`:
    - We store every message inside `Content`
    - `Objectname` for the **owner of the message**.
    - `MessageId` for the message ID
    - `Like` for the number of total like.
    - `UserList` is a **linked list** to store every client who likes this message
    - Etc...
  - We store other information for the group, like `ObjectName` for group name.....
- Every `node` of the linked list `TotalUser` we mentioned in the previous section represents a user.

You can notice that **for all of the linked list being a node of another linked list**, we all use the same `Node` type . `ObjectName` can represent a username or a groupname **in different linked list**. Hence, our data structure may look a little bit of complicated,but it is convenient to use.

Latest 10 Msg

Our message stored as a `Node` in our linked list. Inside the list, `TenStart` is a pointer pointing to the start node of the lastest 10 message. We monitor the number of total message in the group, moving the `TenStart` to the `Next` node inside the list.

```
if the_group.MessageList.Count >= 10 {
        println("moving latest ten message start point")
        the_group.MessageList.TenStart = the_group.MessageList.TenStart.Next
}
```

Communication between Server and Client

With our proto file, the gRPC has provided a good interface for us to use.

```
type Chat_ChatRoomClient interface {
    Send(*ClientRequest) error
    Recv() (*ServerResponse, error)
    grpc.ClientStream
}

type Chat_ChatRoomServer interface {
    Send(*ServerResponse) error
    Recv() (*ClientRequest, error)
    grpc.ServerStream
}
```

We use the `Send()` and `Recv()` on both side to communicate bewtween each other.

Client Logic

- In general, we have a never stop go rountine `receiving` to receive **broadcast** or other message from the server and print it to the screen.
- We also have a infinte for loop to send request until you press `q` to quit the program.

```
var wg sync.WaitGroup
func main(){
    // set a go rountine for receiving
    wg.Add(1)
    go receiving()
    // infinte loop for reading input and sending request
    for{
        // read from keyboard
        switch input{  // send different request content
            case "u":
                stream.Send()
            case "j":
                ...
                ...
                ...
            case "q":
                stream.Send()
                // quit the program
                wg.Wait()  // wait for receiving to stop once quit message is received
        }
    }
    conn.close()
}

func receiving(){
    for{  // infinte loop
        stream.Recv()
        println() // print receiving message
        if quit response received:
            wg.Done()
            return
    }

}
```

concurrency handling

After you press `q` to quit the program, the client will send a **quit message** to the server and waiting for response. The sending part will then be **blocked** and **wait** for the receiving go rountine to finished. After receiving the `quit` response. `wg.DOne()` will set the sending part for free, and be able to **escape from the infinite loop** and then shut the whole client process.

Server Logic

- The core part of the implementation is in the `chatserver.go` inside the `chat` folder.
- The server will generate a **go rountine for evey connected client automatically** once you set up the `chat_server.Serve(listener)`
- For different functions like join group, like message, append message, we handle them in a switch statement inside our **service definition function** `ChatRoom()`.
    - For example, if the client wants to join a group, the `Request` field inside the `message` will be set to `j`. We will then call the corresponding function to handle that.
    - We will then broadcast to every client in case needed.

```
chatRoom(){
    message := stream.Recv()
    // check Request field
    switch input{  // send different request content
        case "u":
            UserLogin()
            // send response
        case "j":
            Join(){}
            broadcast()
        ...
        ...
        case "q":
            // do something
        }
}

func UserLogin(){}
func Join(){}
....
....
....
func QuitSession(){}
```

Broadcast Logic

For each group, we store each client's information in the linked list `UserList`. On each `join` function, we store the client's `Chat_ChatRoomServer` interface in the group's linked list. With this interface, we can send to any client we want.

We just loop through the `UserList`, extract the `Chat_CharRoomServer` for different client , and then sending through it.

```
broadcast(){
    client := the_group.UserList.Head
    for client != nil{
        for latest 10 message in the group{
            client.ct.Send()  // sending it
        }
        client = client.Next
    }

}
```