Mandatory 3 – Chit Chat

Alexander Llewelyn Yan Fatt Soo & Aske Thomas Neye

System Architecture:

Our chat service uses a centralized *server-client* architecture. The *server* maintains a list of all connected *clients* and their message streams. The *clients* connect to the server, send messages, and receive broadcast messages.

gRPC communication model

To decide which type of RPC communication to use (unary / server streaming / client streaming / bidirectional streaming) we look at the requirements of each part of the chat service.

When the client publishes a message it sends an individual message using a unary gRPC type (SendMessage).

Looking at the server it broadcasts messages to all client using server streaming but the client must also receive updates while sending messages.

So, we use bidirectional streaming between each client and the server, which makes it possible to achieve this.

Proto design

```
// =====================
// === Service Layer ===
// =====================
service ChatService {
  rpc Chat(stream JoinMsgLeave) returns (stream ServerMessage);
}
```

The Chat RPC method is implemented as a bidirectional streaming call. This means both the *client* and the *server* can send messages independently over an open stream once the connection is established. This is an asynchronous network. Since the two streams are independent, the *client* and *server* can read and write messages in any order.

Clients send messages of type JoinMsgLeave to the server. These messages can represent three event types: a *client* joining (JOIN), sending a chat message (MESSAGE), or leaving (LEAVE) the chat.

The *server* broadcasts messages of type ServerMessage to all participants.

How do we handle the Lamport clock

On the *client* side:

We keep a local variable:

```
var lamport int64 = 0
```

Before sending any message (Join, Message, Leave) we increment the clock:

```
lamport++
msg := &pb.JoinMsgLeave{
    Id:         id,
    ClientName: name,
    Type:       pb.EventType_MESSAGE,
    Timestamp:  lamport,
    Msg:        text,
}
stream.Send(msg)
```

On the *server* side:

The ChatServiceServer struct has the Lamport clock

```
clock   int64 // Lamport clock
```

When the server receives any type of event from a client it updates the clock

```
s.clock = max(s.clock, msg.Timestamp) + 1
currentLamport := s.clock
```

Then the server broadcasts as part of the log the current Lamport timestamp.

```
log.Printf("[Server] Message from %s: %s (L=%d)", msg.ClientName, msg.Msg, currentLamport)
```

Diagram of RPC calls with the Lamport timestamps

Taken from the log: