1. Introduction

Instead of chatting by text messages, nowadays more and more people choose to use some application to communicate to each other, and they can use this application as long as internet is available. For people who are physically far away from each other (e.g in different states/countries), it is a good way to get connected. Since we learned scala and AKKA throughout this semester, we decide to build a chat application using AKKA API.

2. Related work

2.1 Scala.swing package

The scala.swing package provides a collection of thin wrappers around Java Swing classes. It is a library which feels natural to use in Scala. And it is sufficiently close to Java Swing to appeal to Java programmers with existing Swing experience.

2.2 Scala actor package

With the advent of multi-core processors concurrent programming is becoming indispensable. Scala's primary concurrency construct is actors. Actors are basically concurrent processes that communicate by exchanging messages. Actors can also be seen as a form of active objects where invoking a method corresponds to sending a message.

The Scala Actors library provides both asynchronous and synchronous message sends (the latter are implemented by exchanging several asynchronous messages). Moreover, actors may communicate using futures where requests are handled asynchronously, but return a representation (the future) that allows to await the reply.

3. Problem definition

As an example of distributed system application, we decide to build a chatting application as it involves concurrency control and has potential to be used for business purpose. We will make sure messaging is in same order, and there will be additional functions such as group chat, add/delete friends.
Additionally, in our project, we are using KVstore as the supporting database in storage tier. Also, lock and lease are necessary when read or write kvstore due to the need for false tolerance.

4. Algorithm

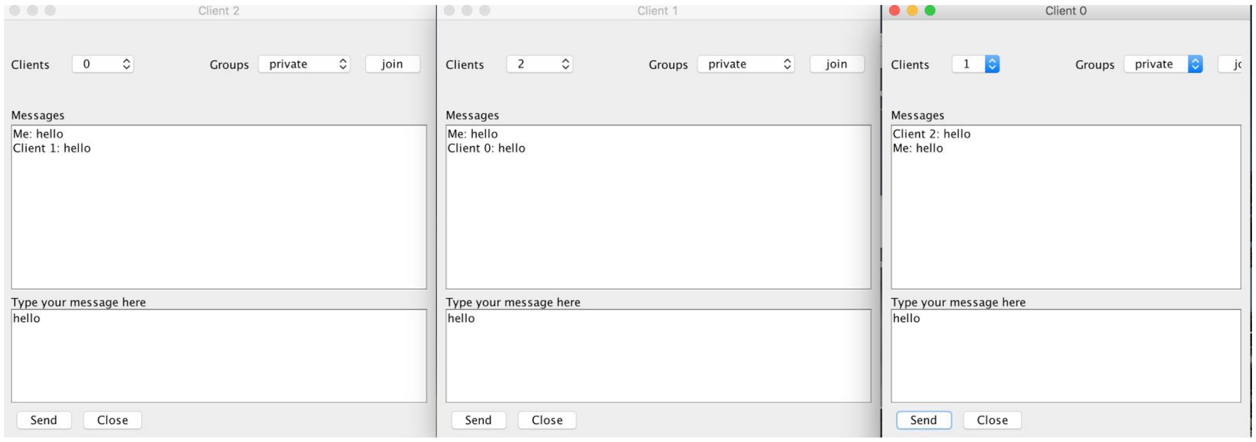
Similar logic as in labs 1 and 2, using KVStore to perform read and write, check if a user belongs to a group,etc. Also we use loadmaster to send bursts of messages to get application started in tests. Besides, when we are using KVStore perform read and write, we want to maintain consistency and fault tolerance, so additional locks and lease management are added.

5. System design

Each client has a UI and a KVClient. Messaging delivery is relied on AKKA actor messaging system. 2 phase locking and 2 phase commit are implemented in KVClient and KVStore to ensure the join & leave operations do not conflict.

5.1 Single Chat - Send

In current version,the client will automatically know the other existing clients. To send a message, the UI capture the input and tell GroupService to display the message. GroupService will decide if the message should be broadcast or send to an individual by checking the string value. Below are some results by running some test cases:

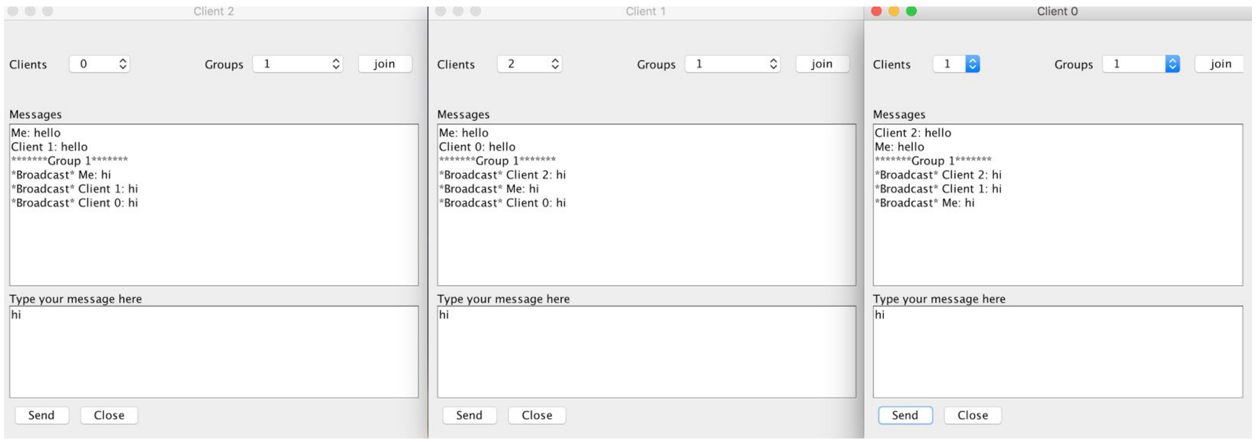


```
Client 2 — Messages:
Me: hello
Client 1: hello
Message: hello

Client 1 — Messages:
Me: hello
Client 0: hello
Message: hello

Client 0 — Messages:
Client 2: hello
Me: hello
Message: hello
```

Here we set number of clients to 3, client 2 send message to client 0, client 1 send message to client 2, and client 0 send message to client 2, we can see that all message are shown in correct receiver window and in correct order.

5.2 Group Chat

In a group chat, clients are capable of joining and leaving groups (of which the GroupServer is a member), and send messages to group members. Each group is assigned a unique id, which will be restored in KVstore. When in group, each client is only able to broadcast the message to whole group, except when the group is private. When in private group, client will be able to send message to other clients individually.Below are some results by running some test cases:



Client 2 — Clients 0, Groups 1

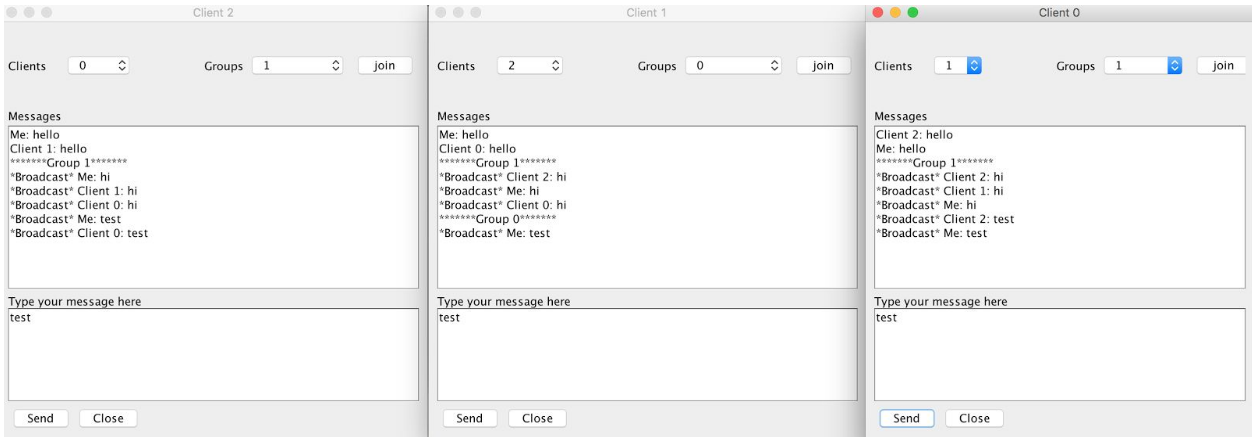
Messages
```
Me: hello
Client 1: hello
*******Group 1*******
*Broadcast* Me: hi
*Broadcast* Client 1: hi
*Broadcast* Client 0: hi
```
Type your message here: hi

Client 1 — Clients 2, Groups 1

Messages
```
Me: hello
Client 0: hello
*******Group 1*******
*Broadcast* Client 2: hi
*Broadcast* Me: hi
*Broadcast* Client 0: hi
```
Type your message here: hi

Client 0 — Clients 1, Groups 1

Messages
```
Client 2: hello
Me: hello
*******Group 1*******
*Broadcast* Client 2: hi
*Broadcast* Client 1: hi
*Broadcast* Me: hi
```
Type your message here: hi



Client 2 — Clients 0, Groups 1

Messages
```
Me: hello
Client 1: hello
*******Group 1*******
*Broadcast* Me: hi
*Broadcast* Client 1: hi
*Broadcast* Client 0: hi
*Broadcast* Me: test
*Broadcast* Client 0: test
```
Type your message here: test

Client 1 — Clients 2, Groups 0

Messages
```
Me: hello
Client 0: hello
*******Group 1*******
*Broadcast* Client 2: hi
*Broadcast* Me: hi
*Broadcast* Client 0: hi
*******Group 0*******
*Broadcast* Me: test
```
Type your message here: test

Client 0 — Clients 1, Groups 1

Messages
```
Client 2: hello
Me: hello
*******Group 1*******
*Broadcast* Client 2: hi
*Broadcast* Client 1: hi
*Broadcast* Me: hi
*Broadcast* Client 2: test
*Broadcast* Me: test
```
Type your message here: test

The first picture shows that client 0, 1, 2 all joins group 1, they all send a greeting message in the order of client 2, 1, 0, we can see that they successfully joined together and multicast freely and the message is in order.
The second picture shows that client 1 leaves group and now the member of group chat are client 0 and 2, which is correctly shown, besides, the message order is in their original sending order.

5.3 Lease/Lock

Every read and write operation regarding to KVstore should require a lease/lock. If anything goes wrong, the operation would abort and rollback. Otherwise, everything will be committed. After either abort or commit, cache, dirtyset and locks will be reset.

2PL is implemented majorly in KVClient and KVStore. In read, KVClient will try to acquire lock before directRead membership lists from KVStore. Until acquire successfully, KVClient will re-attempt after a fixed time interval. Upon acquire successfully, it will check if the value for the given key (groupid) exists. If not, which means no client has ever joined this group, it will create an empty list and directWrite to KVStore. Then it stores the list in cache. In write, KVClient utilizes cache to store the join/leave request received from client. Client will call begin after write, to invoke 2PC implemented in KVStore and KVClient.

6. Experiments

In order to test, we use the LoadMaster calls KVAppService to instantiate the clients, then sends bursts of empty command messages. When receiving a command, a client chooses an action and executes it. Test will terminate after a configured number of commands are sent and acknowledged. Then, we test it by manually input text, and see if the messages are received in order and joining groups works fine.

6.1 Traffic control

In order to affect the amount of message traffic, in the test, we would add execution possibilities on sending, joining and leaving request.

6.2 Racing condition

Due to racing condition, receiving message is not in the same order as message sent, and they are grouped together by receiver’s id. But the order of messages sent by a same member would remain the same.

6.3 Late messages

To avoid the late messages(received after member left the group), we would check the KVStore server when receiving messages. In this way we denied message receiving if the intended receiver has left the group.

Conclusion

We successfully managed to implement the process of distributed messaging and showed it on GUI.