Concord Design Document

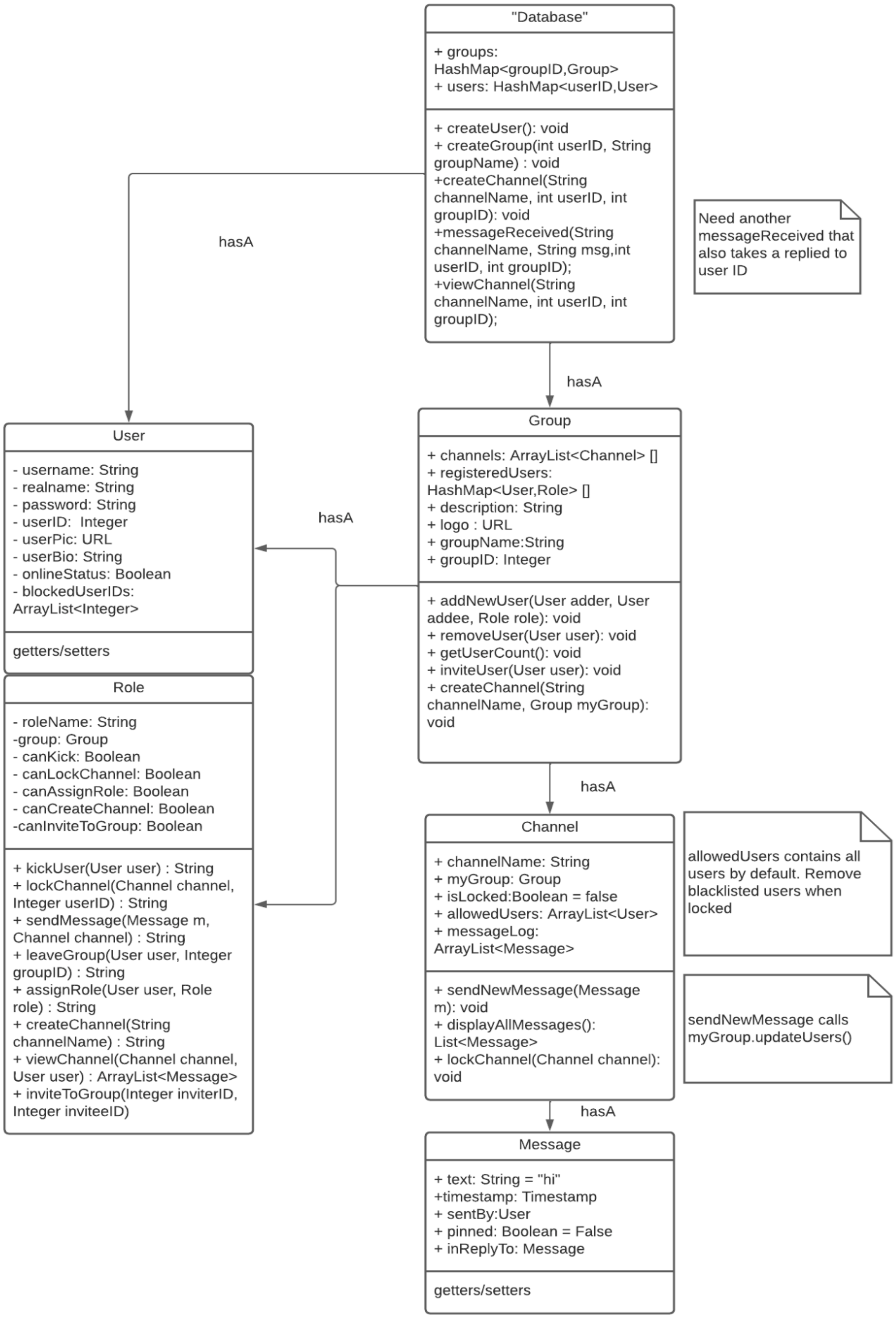
CJS Squad

Our Design:

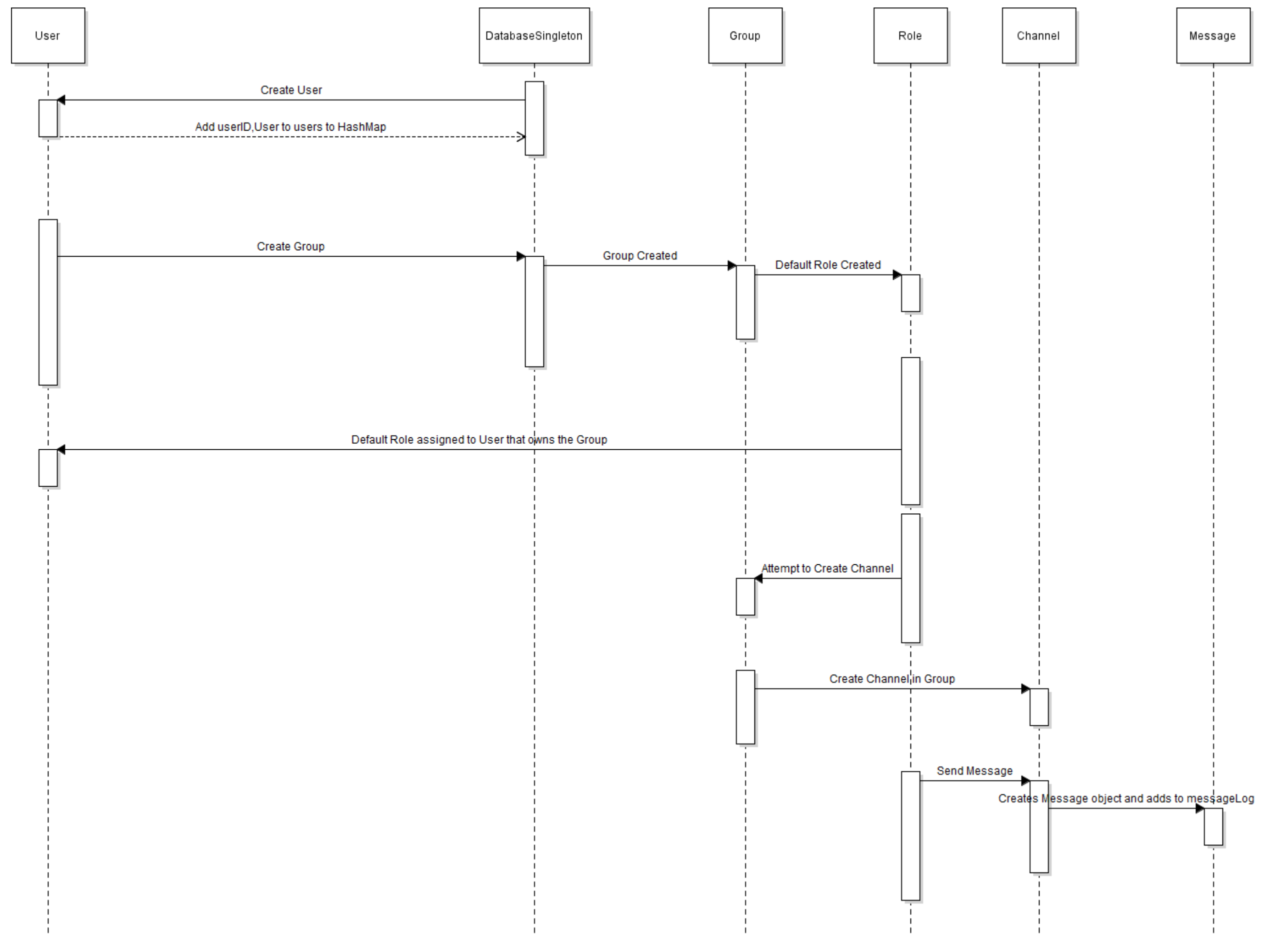
Sprint 1

Problem:
The challenge is to design a communication platform for everyone that provides a place where people can reach a consensus. The product, Concord, should utilize design principles to create a product that is easy for customers to use and easy for developers to maintain and extend. Additionally, the features agreed upon by our peers should be fully implemented.

UML:
https://lucid.app/lucidchart/c0f15fbf-78aa-413a-aabf-e5d6edfbc656/edit?invitationId=inv_0e20c090-0e5c-47f7-bfff-c0b94a91f126


"Database"
+ groups: HashMap<groupID,Group>
+ users: HashMap<userID,User>
+ createUser(): void
+ createGroup(int userID, String groupName) : void
+createChannel(String channelName, int userID, int groupID): void
+messageReceived(String channelName, String msg,int userID, int groupID);
+viewChannel(String channelName, int userID, int groupID);
Need another messageReceived that also takes a replied to user ID
hasA
hasA
User
- username: String
- realname: String
- password: String
- userID: Integer
- userPic: URL
- userBio: String
- onlineStatus: Boolean
- blockedUserIDs: ArrayList<Integer>
getters/setters
hasA
Group
+ channels: ArrayList<Channel> []
+ registeredUsers: HashMap<User,Role> []
+ description: String
+ logo : URL
+ groupName:String
+ groupID: Integer
+ addNewUser(User adder, User addee, Role role): void
+ removeUser(User user): void
+ getUserCount(): void
+ inviteUser(User user): void
+ createChannel(String channelName, Group myGroup): void
Role
- roleName: String
-group: Group
- canKick: Boolean
- canLockChannel: Boolean
- canAssignRole: Boolean
- canCreateChannel: Boolean
-canInviteToGroup: Boolean
+ kickUser(User user) : String
+ lockChannel(Channel channel, Integer userID) : String
+ sendMessage(Message m, Channel channel) : String
+ leaveGroup(User user, Integer groupID) : String
+ assignRole(User user, Role role) : String
+ createChannel(String channelName) : String
+ viewChannel(Channel channel, User user) : ArrayList<Message>
+ inviteToGroup(Integer inviterID, Integer inviteeID)
hasA
Channel
+ channelName: String
+ myGroup: Group
+ isLocked:Boolean = false
+ allowedUsers: ArrayList<User>
+ messageLog: ArrayList<Message>
+ sendNewMessage(Message m): void
+ displayAllMessages(): List<Message>
+ lockChannel(Channel channel): void
allowedUsers contains all users by default. Remove blacklisted users when locked
sendNewMessage calls myGroup.updateUsers()
hasA
Message
+ text: String = "hi"
+timestamp: Timestamp
+ sentBy:User
+ pinned: Boolean = False
+ inReplyTo: Message
getters/setters

Sequence Diagrams:
Creation of User, Group, Role, Channel and Message; Needed to send “Hello World” in a channel


User
DatabaseSingleton
Group
Role
Channel
Message
Create User
Add userID,User to users to HashMap
Create Group
Group Created
Default Role Created
Default Role assigned to User that owns the Group
Attempt to Create Channel
Create Channel in Group
Send Message
Creates Message object and adds to messageLog

Presentation:
https://docs.google.com/presentation/d/1sUB91nG6kgWt6WQqE7OuRxi2RYeTzwDEy5HRks3wVuY/edit?usp=sharing

Design Justification:

- Addresses all requirements agreed upon by the class
- Role class utilizes facade for convenient use of methods
- Role class allows for permissions-based actions
- Design emphasizes composition over inheritance for flexibility
  - Example: Much of group behavior derived from channel(s) it contains
- Focus on modularity, concerns separated logically
  - Example: new roles can be created and assigned to users without any change in User class itself

We find that this design effectively addresses all the design requirements agreed upon by the class. The magic of the design first and foremost starts in the Role class, which utilizes the facade pattern to easily perform all the required methods of the Concord backend.

Each user has a Role associated with them via a HashMap in the Group class, and each role has permissions denoted by boolean attributes. Through this setup, only users with group roles of a certain clearance can perform certain actions, such as removing a user from the group. So, the role not only acts as a convenient facade for calling methods on all the rest of the classes in the system, but also prevents unauthorized users from performing actions they shouldn't, a sort of two-for-one deal.

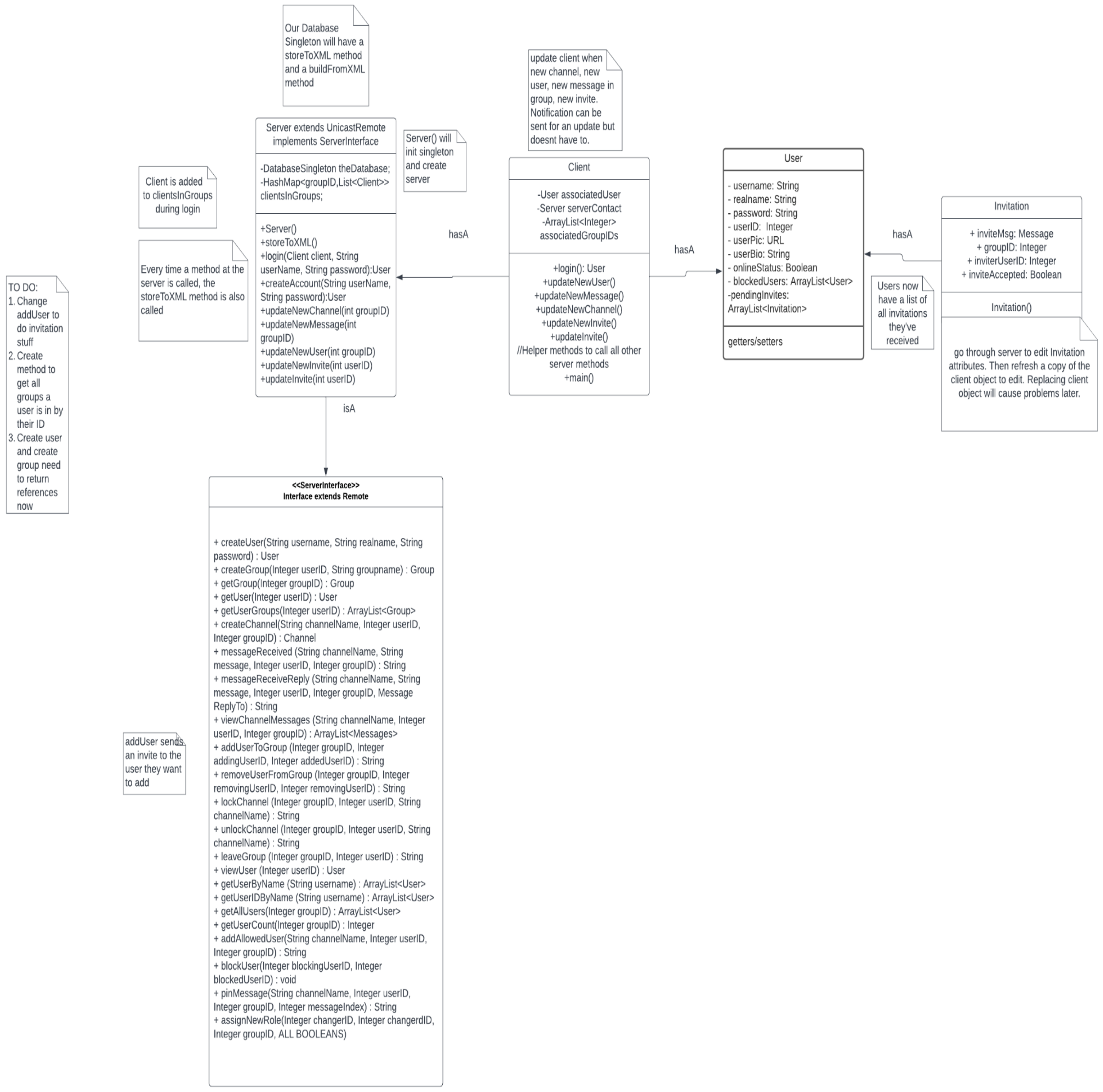
The effectiveness of this design is not just limited to the Role class, however. This design also prioritizes composition over inheritance for behavioral changes without having to modify very much code. Take for example the Group class, which contains an ArrayList of all the channels the Group knows about. If we wanted a new type of Channel that does something differently than a normal Channel, all we would have to do is subclass Channel, add the new behavior we want, and then the Group could use it as normal with little or no modification to the code in Group.

Lastly, along the same lines of the composition over inheritance that this design emphasizes, is a focus on modularity. Each class in this design has methods and objectives within the system that are logically separated based on what the class itself is. For example, the User class is solely concerned with storing attributes about any given user, such as their name, bio, etc. If a User in a Group is promoted to some new Role with more permissions than before, nothing in the instance of the User object representing them needs modified or changed- a new Role is simply created, permissions are set, and this new role is associated with them in the HashMap. This is especially effective since Users will not have a common role across all Groups, so having their Role associated with them within the Group and not the User class itself further helps separate concerns.

Overall, we find this design effectively represents every core part of the Concord application backend, with a convenient means of calling methods, easy extensibility and behavioral change, and a logical separation of concerns for each class.

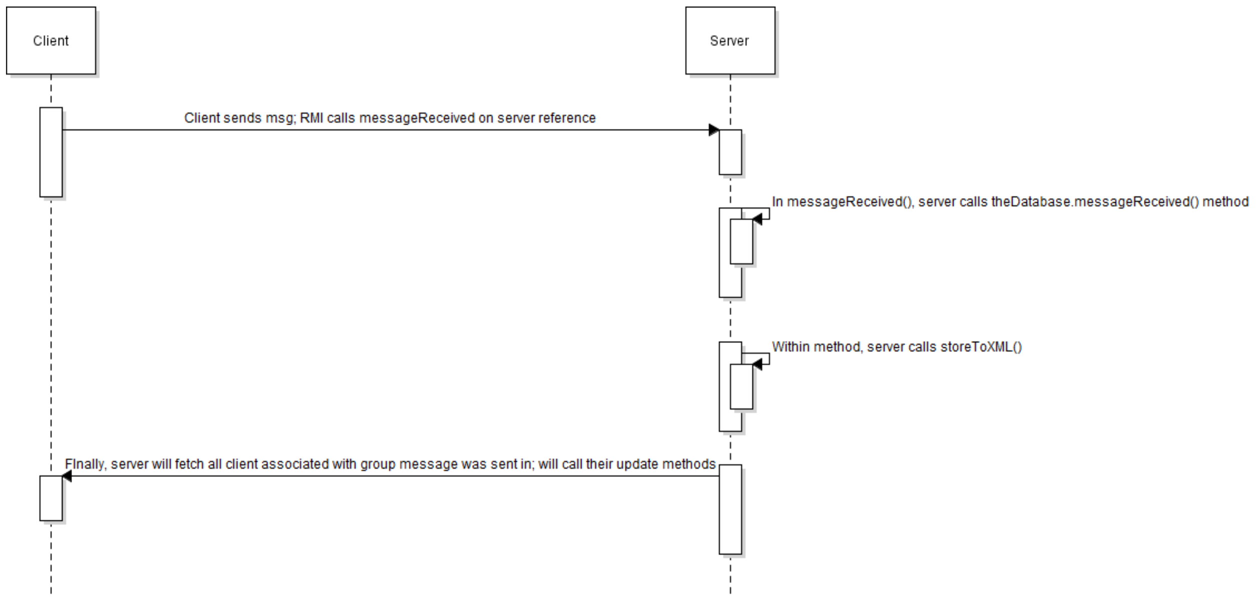
Sprint 2

Problem:
For sprint 2, we had to come up with a design implementation to store and manipulate the data from sprint 1 within a client server infrastructure. All of the data had to be stored on a central server allowing any number of clients to access it. The server has to be able to authenticate user credentials, store all updates to disk and read all the files on disk on startup. The clients act on behalf of the user accounts created and can attempt actions they are and are not authorized to perform. This implementation has to allow users to be created through different client connections and interact with the current database. Essentially, we had to develop a frontend to compliment the “backend” we created in sprint 1, and allow for people to actually interact with groups, channels, etc through a client.

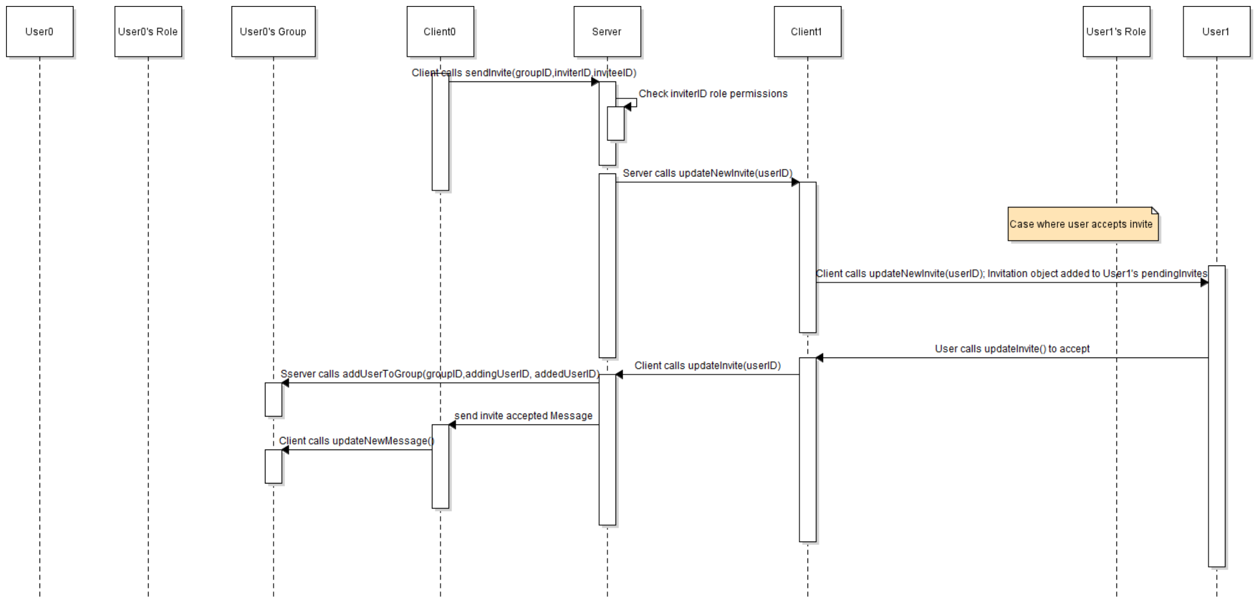

Our Database Singleton will have a storeToXML method and a buildFromXML method
Server extends UnicastRemote implements ServerInterface
-DatabaseSingleton theDatabase;
-HashMap<groupID,List<Client>> clientsInGroups;
+Server()
+storeToXML()
+login(Client client, String userName, String password):User
+createAccount(String userName, String password):User
+updateNewChannel(int groupID)
+updateNewMessage(int groupID)
+updateNewUser(int groupID)
+updateNewInvite(int userID)
+updateInvite(int userID)
Server() will init singleton and create server
update client when new channel, new user, new message in group, new invite. Notification can be sent for an update but doesnt have to.
Client
-User associatedUser
-Server serverContact
-ArrayList<Integer> associatedGroupIDs
+login(): User
+updateNewUser()
+updateNewMessage()
+updateNewChannel()
+updateNewInvite()
+updateInvite()
//Helper methods to call all other server methods
+main()
hasA
hasA
User
- username: String
- realname: String
- password: String
- userID: Integer
- userPic: URL
- userBio: String
- onlineStatus: Boolean
- blockedUsers: ArrayList<User>
-pendingInvites: ArrayList<Invitation>
getters/setters
hasA
Invitation
+ inviteMsg: Message
+ groupID: Integer
+ inviterUserID: Integer
+ inviteAccepted: Boolean
Invitation()
Users now have a list of all invitations they've received
go through server to edit Invitation attributes. Then refresh a copy of the client object to edit. Replacing client object will cause problems later.
Client is added to clientsInGroups during login
Every time a method at the server is called, the storeToXML method is also called
TO DO:
1. Change addUser to do invitation stuff
2. Create method to get all groups a user is in by their ID
3. Create user and create group need to return references now
isA
<<ServerInterface>>
Interface extends Remote
+ createUser(String username, String realname, String password) : User
+ createGroup(Integer userID, String groupname) : Group
+ getGroup(Integer groupID) : Group
+ getUser(Integer userID) : User
+ getUserGroups(Integer userID) : ArrayList<Group>
+ createChannel(String channelName, Integer userID, Integer groupID) : Channel
+ messageReceived (String channelName, String message, Integer userID, Integer groupID) : String
+ messageReceiveReply (String channelName, String message, Integer userID, Integer groupID, Message ReplyTo) : String
+ viewChannelMessages (String channelName, Integer userID, Integer groupID) : ArrayList<Messages>
+ addUserToGroup (Integer groupID, Integer addingUserID, Integer addedUserID) : String
+ removeUserFromGroup (Integer groupID, Integer removingUserID, Integer removingUserID) : String
+ lockChannel (Integer groupID, Integer userID, String channelName) : String
+ unlockChannel (Integer groupID, Integer userID, String channelName) : String
+ leaveGroup (Integer groupID, Integer userID) : String
+ viewUser (Integer userID) : User
+ getUserByName (String username) : ArrayList<User>
+ getUserIDByName (String username) : ArrayList<User>
+ getAllUsers(Integer groupID) : ArrayList<User>
+ getUserCount(Integer groupID) : Integer
+ addAllowedUser(String channelName, Integer userID, Integer groupID) : String
+ blockUser(Integer blockingUserID, Integer blockedUserID) : void
+ pinMessage(String channelName, Integer userID, Integer groupID, Integer messageIndex) : String
+ assignNewRole(Integer changerID, Integer changerdID, Integer groupID, ALL BOOLEANS)
addUser sends an invite to the user they want to add

UML (Above):
https://lucid.app/lucidchart/daf1b815-a743-44bb-8cb8-613ea8b07857/edit?invitationId=inv_5af8b06c-9b9b-435c-bc06-21c3faa049e5

Sequence Diagrams:
Client sends new message and server updates


Client
Server
Client sends msg; RMI calls messageReceived on server reference
In messageReceived(), server calls theDatabase.messageReceived() method
Within method, server calls storeToXML()
FInally, server will fetch all client associated with group message was sent in; will call their update methods

User invites another user to their group


User0
User0's Role
User0's Group
Client0
Server
Client1
User1's Role
User1
Client calls sendInvite(groupID,inviterID,inviteeID)
Check inviterID role permissions
Server calls updateNewInvite(userID)
Case where user accepts invite
Client calls updateNewInvite(userID); Invitation object added to User1's pendingInvites
User calls updateInvite() to accept
Client calls updateInvite(userID)
Sserver calls addUserToGroup(groupID,addingUserID, addedUserID)
send invite accepted Message
Client calls updateNewMessage()

Design Justification:

Our design fully addresses all aspects of the problem through the use of a few design patterns. Firstly, by using the proxy pattern through our Server class, we ensure that clients can easily communicate with the “backend” of our system through RMI. Furthermore, through the login() methods, the Server ensures that only clients that provide a valid username and password can talk to the backend and receive a User object to be associated with. This feature of associating a client with a user also ensures that any client only has the permissions that their associated user does.

The storeToXML() method within the Server provides a convenient way of saving all the important data within the server, such as the known clients and the database attribute it has, and/or reading this data back in on startup. Finally, the combination of the Server and its database attribute allows for a convenient facade for client interaction through the use of many helper methods, also an extension of the proxy pattern.

Beyond these features of the Server class, the Observer pattern is also implemented for the purpose of notifications. The Server class has a HashMap groupIDs which map onto known clients. Upon a message being sent or some other event, the Server will iterate over the client list for a group, and call update methods on them, sending them a notification regarding the event.

We also address the new requirement of user invites through the creation of a new attribute in the User class, and a new Invitation class. Now, every user has an ArrayList of Invitations from other users, which they can then use to join a new group if they so choose. Here again, we prioritize composition over inheritance to get dynamic behavior from a given class.