Muhammad Faisal Muhammad email: mfaisal@aucegypt.edu

Groups Management: Package Implementation Details

github.com/dedis/student_19_groupmgmt

Members in large societies have various interests and beliefs. This variety generates tendency for members to get closer with other members with similar interests and beliefs. Moreover, this resultant tendency to stay with members with similar interests comes from various reasons such as trying to organize activities with common interests or trying to invite others to share their beliefs or interests. As more people with same interests start to get closer, new groups start to emerge. These groups can then take many simple and complex forms such as political parties where its members seek to issue laws that seems best according to their agenda. It can also take the form of an NGO, startup, or similar groups where society members wants to do something collectively. In this document, we discuss some useful groups features and how to implement them.

1 Group formation and management

The need to form groups results from the need of society members to work together on some tasks and therefore some problems would arise such as how to organize these tasks and how to distribute the work with each other.

Group A Group is an organized set of society members working together for some common interests and goals.

1.1 Separation of roles

A first approach where all people would meet together to discuss some important points and then decide how to proceed with their plans may seem impractical specially in large groups. For example, not all members would have the same free time slots and there is no need for a member to go through every single detail themselves. Moreover, in order for a group to execute their task successfully and efficiently, they better distribute roles for each group of members according to that group's expertise and how much they would fit for such role. Moreover, although some roles can be done by many members correctly without special skill set or special expertise such as routine work that just needs some minor human work or also verifying that someone is eligible to become a group member, other tasks can be much more complicated. For example, The task of managing financial assets such as buildings owned by the group or deciding how to put a budget for the group's various activities needs someone with special expertise in finance for example.

Therefore, it may seem more convenient if similar activities can be grouped together to be controlled from some subset of the group members who may seem more suitable for such task.

Role A role is a subset of a group members who authorized to manage some group activities because of their common skill set or expertise.

1.2 Membership management

Each group should be able to manage their members whether by members addition or removal to ensure that all the current members in the group are useful and valuable working members. Such as managing the group financial assets, managing the group members and the membership of members in each role is an important activities that should have its rules and conditions clear, transparent, and managed by a role that have reliable group members.

1.3 Decision making

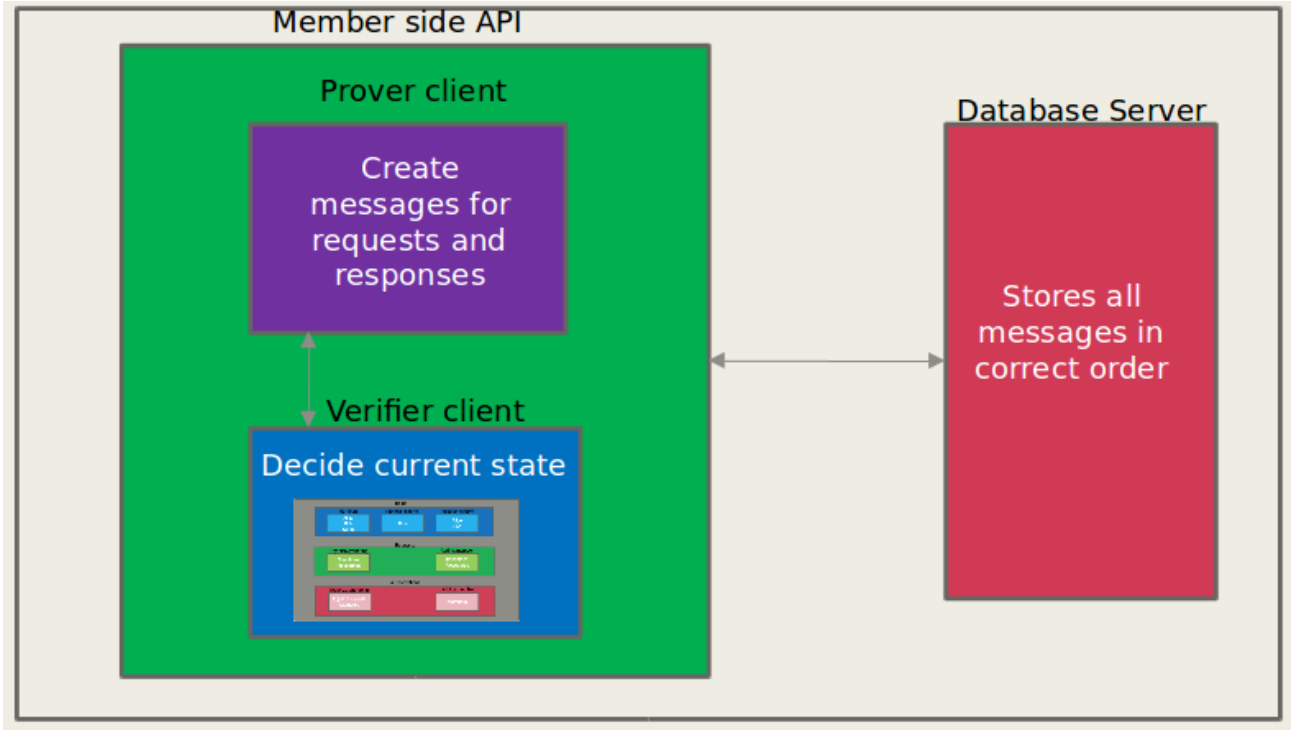
Although as said before some activities would require more expertise than others, every decision taken in a group must be a result from some collective members' action to ensure minimum malicious decisions and also to avoid potential coercion. A simple way to do this is to ensure that every decision is approved from the majority of members who are managing such type of decisions. In other words, if there are two roles who are managing decisions related selling some building, then to proceed with selling such building, it is a must to get approval from more than half the members in these two roles.

1.4 Rules Flexibility

A group management system should be able to count for the changes that may result in the policies or rules controlling the group. In other words, it should be working towards making the target of the system to provide a set of rules to control decisions made through the system, yet provide a way to change such rules.

2 Design and Implementation

A system has been designed to count for the previously mentioned points. This package implements the client side as well as the overall aspects of the communication protocol of this system and will be described in details in the next subsection. The other part of the system, the trusted storage has been implemented as a shared object in the memory. However, for the system to be fully functional and be deployed for public use, that simulated storage should be turned into a passive trusted storage responsible for storing all received messages with the correct timestamp. Moreover, the package for the client side is written in go and will be discussed in the following subsections.



Overall structure of the system.

2.1 Client side API

This can be considered the upper layer of the system and consists from two parts, the prover client and the verifier client. Each user must have a pair of keys and their public key stored in the trusted storage. The pair of keys is stored through the prover class which is called the User class. This class has the main functions to be able to generate messages that can be related to the user to issue some request or reply to such a request according to the need as it will be explained later.

The main structure of the User class is as follows:

```
type User struct {
    UserInfo
    GroupMemberInfos map[string]GroupMemberInfo
}

type UserInfo struct {
    UserID     string
    PublicKey  interface{}
    PrivateKey interface{}
}

type GroupMemberInfo struct {
    PendingRequests []Message
    SignRequest     func(request Request, userInfo UserInfo) interface{}
    SignResponse    func(response Response, userInfo UserInfo, userIDs []string)
}
```

It mainly has two main functions in order to communicate with the trusted storage and send verifiable messages;

```
func (user *User) CreateRequest
    (groupName string, requestName string, args ...interface{})
    RequestMessage

func (user *User) CreateResponseMessage
    (groupName string, requestNumber int, answer bool)
    ResponseMessage
```

Moreover, the client side has another main part which is working as a verifier. It is important that each group member be able to verify all the messages done by the other members to know the current state of the group. The verifier class is called the Group class.

The group class has the following structure and can be considered just a layer to verify the validity and signer of each message in the system.

```
type Group struct {
    BasicGroup

    Messages        map[string]Message
    AllMessages     []interface{}
    PendingRequests map[string]map[string]EmptyStruct
}
```

Each message related to the group must be added in the correct order after retrieval from the trusted storage to the group through the following two functions. If a message passes this layer successfully, it will be echoed to the lower layer called the BasicGroup class to apply the message effect.

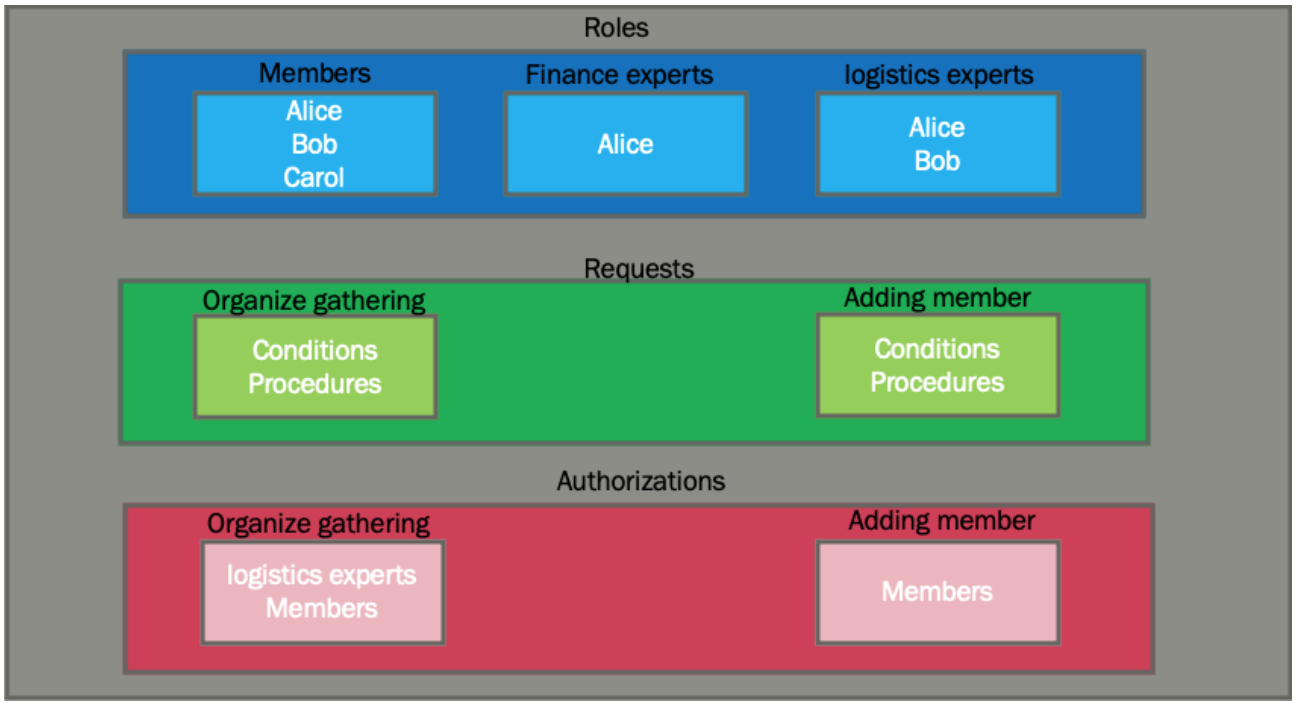
```
func (group *Group) AddRequest(requestMessage RequestMessage) RequestStatus
func (group *Group) AddResponse(responseMessage ResponseMessage) MessageStatus
```

2.2 BasicGroup Structure

The basic group class or structure is the class responsible to apply changes to the group structure. In other words, this is the class that has the functionalities to add members, remove members or such things of group features. However, this functions should not be called directly. However, they will be called once a correct message was received that requests the call of such functions. This means that the verifier part of the client side is first a layer that decides whether a change should happen or not and then another layer that implements the logic of how a change happens.

Please, note the following definitions that will be also explained in detail later.

1. **User:** Any one with access to the system.
2. **Member:** User that has a role in the group.
3. **Role:** A subset of the group members which gets authorization to handle specific types of requests.
4. **Request:** a question accompanied with some data to be passed to the members of some specific roles to get a yes/no answer.
5. **Delegated member:** Member belonging to a role that can handle the mentioned type of requests.



An example of a group structure

The proper way to do a change is through registering a request type and specifying which function to be called in case such request is a success while allowing some parameters to be passed that to be called function. So first a request type is created, then a certain role is given the privilege to reply these requests. This can be done using the following function.

```
func (group *BasicGroup) AddRequestType(requestType string, managingRoles []stri
    successFunction func(args ...interface{}) interface{},
    verifyRequest func(request Request, signature interface{}) RequestStatus,
    verifyResponse func(responseMessage ResponseMessage, previousResponses map[s
    (MessageStatus, map[string]ResponseMessage))
```

There are more functions to ensure more easiness dealing the system. For example adding roles, adding members in roles, merging roles and such possible manipulation functions for roles, request, and authorizations.

3 Package flow

3.1 Creating new group

Groups creation in the package is done in two phases. The first one is to create a BasicGroup Structure. The second one is to build on top of this structure another one to manage the request, responses and afterwards changes.

3.1.1 Creating BasicGroup

The Basic group structure is used to contain main information about the group as described earlier.

A new BasicGroup is created using the function:

```
func CreateNewBasicGroup(...) BasicGroup
```

This function has the following parameters:

1. **groupName:** A string representing a unique name for the group.
2. **description:** A string representing a short description of the group.
3. **creator:** A string containing the username of the group creator.
4. **defaultVerifyRequest:** a function that will be used to verify a received request to the group to check two main things; the request is created by a valid member authorized to create such a request type, and that the request is a valid one.
5. **defaultVerifyResponse:** Similar to the previous function, defaultVerifyResponse is used to verify responses received to collectively construct a response for a previous request. This function should be used to verify that the response is from a delegated member to answer the specified request and also to decide on the final response to a request given the verification of all the so far responses to this request.

The previous function is used to create the very basic group structure. After creating such a BasicGroup with very limited information, the creator can proceed with constructing the group more with the following functionalities.

1. Adding and editing Roles: In this step you can do many operations on roles such as: creating new ones, removing existing ones, or merging two roles in one role using the following functions.

```
func (group *BasicGroup) AddRole(role string) bool
func (group *BasicGroup) RemoveRole(role string) bool
func (group *BasicGroup) MergeRoles(roleOne string, roleTwo string, newRole
```

2. Adding Members to Roles:

```
func (group *BasicGroup) AddMemberInRole(userID string, role string) bool
func (group *BasicGroup) AddInRole(userIDs []string, role string) bool
func (group *BasicGroup) RemoveMemberInRole(userID string, role string) bool
```

3. Adding and editing Requests: Good examples of requests can be found in the basicgroup$_test.go$.

func (group *BasicGroup) AuthorizeForRequestType (roles []string, requestType string) bool

func (group *BasicGroup) UnauthorizeRoleFromRequestType (role string, requestType string) bool

3.1.2 Building on the BasicGroup

Now the initial group structure is ready as all its roles, members, requests and authorizations are specified. The previous group information should now be stored in a layer that would control any changes in the BasicGroup structure.

This can be done through two steps:

1. Embedding the created BasicStructure into a newly created Group Structure as follows:

```
group := Group{}
group.BasicGroup =  //name of the BasicStructure object
group.Messages = make(map[string]Message)
group.AllMessages = make([]interface{}, 0)
group.PendingRequests = make(map[string]map[string]EmptyStruct)
```

2. The other way is through creating an object using the following function, which has same parameters as CreateNewBasicGroup function. After creating the group in such a way, the functions in the 4 steps in the previous subsection should be called on returnedgroup.BasicGroup instead.

```
func CreateNewGroup(...) Group

//example
returnedgroup := CreateNewGroup(...)
returnedgroup.BasicGroup.AddRole("admins")
```

However, in all cases once the initial group structure is created, no modifiers functions should be called the on returnedgroup.BasicGroup object. Instead, a request should be created and then the change will be pending the receiving of the valid responses, check section 3.3 for more details. Moreover, copy of the previous returnedgroup should be copied to all the members' local computers through a trusted storage for example or a peer to peer protocol.

3.2 Creating a new member structure

Since, the group will be managed after creation through requests and responses, A User structure was created to manage such messages creation for requests and response. Each member that is new to the system should register his UserID and PublicKey in the system and keep his private key secret in his local computer storage.

```
type User struct {
    UserInfo
    GroupMemberInfos map[string]GroupMemberInfo
}

type UserInfo struct {
    UserID      string
    PublicKey   interface{}
    PrivateKey  interface{}
}

type GroupMemberInfo struct {
    PendingRequests []Message
    SignRequest     func(request Request, userInfo UserInfo) interface{}
    SignResponse    func(response Response, userInfo UserInfo, userIDs []string)
}
```

Once a user knows he became a member of a group or able to send a request to a group, they must register how to do so in ther User structure by the following function:

```
func (user *User) AddGroupMemberInfo(...)
```

which has three main parameters:

1. groupName: the group name that they became a member in.
2. signRequest: A function that is used to sign requests to the group. It should be consistent with the corresponding verification function inside the group.BasicGroup structure.
3. signResponse: A function similiar to signRequest but to sign Responses instead.

Now a member is able to create and reply to requests as identified in 3.3 section. More inforamtion about using such User structure and creating signature and verification functions can be found in the groupMember_test.go.

3.3 Managing the group using Requests and Responses

Now each member should have both User structure and a copy of the initial group structure created earlier and now also everyone should be responsible to maintian his group copy valid by following the following instructions.

1. Only getters can be called on the group.BasicGroup object.
2. Changes should be done through registering any communication messages through the functions AddRequest and AddResponse. These functions should be able to identify good requests and responses based on the data added to the initial group.BasicGroup when created by the creator.
3. Requests and Responses messages to be accepted into the group can be created through the User strucutre by the functions: CreateRequest, and CreateResponseMessage.
4. It is the member's responsibility to continuously check for new requests and responses messages and to continuously send others his request and response messages. For testing, that is done through a shared object to store all the messages created.
5. To use the CreateResponseMessage function in the User structure, the User strucutre should be updated by the current data in the group Structure by:

```
user.GroupMemberInfos[groupName].PendingRequests =
group.GetPendingRequests(user.UserID)
```

4 Blockchain instead of database

In order to decentralize the system's database, a blockchain can be used then to store both the requests and responses.

However, this requires the following:

1. The order, in which transactions are chosen to be included in the block, should be modified such that responses should be included before including new requests.
2. If a role's members are controlling the addition of a another role's member; Responses or requests from the controlling roles should be added to the block before those of the controlled roles.
3. Not keeping these constraints would for example allow that members pending removal be somehow still able to answer requests.

5 Future plan

1. Finding a way for delegated members to change a malicious response if any happens.
2. Researching on how an election process can happen as a way of choosing delegated members (can we for example use delegated members' previous decisions to decide on the future of their continuity in their role through some reputations system).