Implementation of Group Membership Abstraction using Raft and Go

Stylianos Agapiou
School of Computer and Communication Sciences
EPFL, Lausanne, Switzerland
Email: stylianos.agapiou@epfl.ch

***Abstract*—This is a project of the LPD (Distributed Programming Laboratory) headed by Rachid Guerraoui and it was supervised by Dragos-Adrian Seredinschi. The goal of the project was to design and implement a fault tolerant cluster that provides Group Membership Services using Go programming language and Raft Consensus protocol. The cluster supports large-scale storage services with high availability and consistency. In this scope, the failure detection is managed and coordinated by each node of the GM cluster by sending heartbeats to the client processes it is responsible for. We implemented a service, which extends the basic functionality of GM Abstraction, that accepts join and leave requests from processes explicitly and also provides information about cluster nodes and the processes each one is responsible for.**

***Index Terms*—Group Membership, raft, failure detection, distributed systems, concurrency**

I. Introduction

Distributed systems consist of groups of networked computers, which interact with each other to achieve the same goal. Three significant characteristics of distributed systems are: concurrency of components, lack of a global clock, and independent failure of components. Each node has its own local memory and they communicate with each other by message passing.[1] Concurrency is used with Go programming language, such that each node executes computations while synchronizing with the others. To create a uniform and consistent distributed system we used the consensus algorithm *Raft* to achieve replicated machine state of the log of the cluster nodes. Even though *Paxos* is the consensus algorithm that has been discussed in the Distributed Algorithms class, Raft is used in the project since the implementation of CoreosOS offers flexibility and gives to the programmer the ability to optimize parts of the application separately. As a result, we had to create our own storage layer to persist the Raft log and state. Raft in our days has a major role in many projects such as, Kubernetes, etcd, chain core, cockroachdb and others.

Group membership service relies on the importance and the need to provide accurate information about which processes of a distributed system are functional or crashed. The output of the service is based on the functionality of the consensus and the failure detectors, but it is maintained on a higher level. To create a scalable and consistent solution we deploy a cluster of 3-5 nodes that implement the GM Service. In that way, thousands of other client processes could connect to the cluster and ask information about the groups of the cluster or join a group.

The asynchronous distributed memory system stores clients' membership information depending on the consensus decided values. In our case, each node holds a key value store defined as: **store map[string]map[string]string**. The two main services of the system:

- *stores clients' information*, for each node stores the processes it is responsible for.
  for example:

nodeId	clientId	client IP:Port
store['a']	'clientId1'	'IP1:Port1'
	'clientId2'	'IP2:Port2'
	'clientId3'	'IP2:Port3'
store['b']	'clientId4'	'IP4:Port4'
	'clientId5'	'IP5:Port5'

  This Key-Value store maps the nodes that implement GM Service with the clients each one is responsible for.
- *supports query operations*, a client process shall query about which groups exist or even ask for the members of a specific group by providing the group name.

Our implementation guarantees strong consistency, since all cluster nodes return the same result when queried with achieving high throughput and high availability.

This report is organized as follows: Section II describes specifications of the Group Membership Implementation and analyzes the details of raft's integration in to the project. Section III discusses the architecture and emphasizes on some important concepts and decisions. Section IV presents the properties of the group membership abstraction and some use cases. Section Vdiscusses the challenges we faced and refers to the Go tools and packages used for the project. Finally, this reports concludes in Section VI.

II. Group Membership Specifications and Architecture

We consider an asynchronous distributed system, where processes communicate by exchanging messages. All processes(both cluster nodes and clients) are identified by unique id's. The asynchronous model of execution of concurrent processes follows the one described in [2], with a few extensions. Client processes can join or leave the distributed system at any time. Each member in the group participates in membership

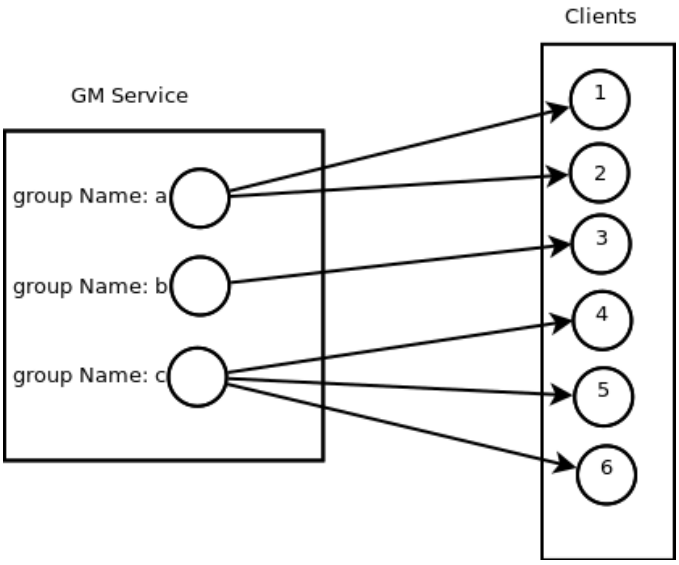

Fig. 1: An instance of our implementation that shows an example of how the cluster nodes of GM Service can be linked to client processes.

decisions, which ensures that either all members see a new member or none. The basic concept of GM Abstraction as described in [2], is that the set of processes is predefined and no new processes can join the groups. Our implementation coordinates join and leave operations and provides a dynamic set of processes in the system, which is essential for online systems. Failure detection is accurate and is provided in a consistent way. Specifically, the GM Service is consisted of nodes, most likely 3-5 that coordinate the abstraction to guarantee high availability, scalability and strong consistency. As a result, our system is organized in groups such that each node is linked to some specific clients, as it is shown in Figure 1. Each node of the GM cluster is assigned a unique node Id, an IP:Port, a hashkey which is derived using MD5 and a group Id, which identifies the group that this node is responsible for. A small representation of the node in Go is the following:

```
type nodeInfo struct {
    nodeId      int
    myIp        string
    myPort      string
    hashKey     string
    groupId     string
}
```

Client processes join the GM Service and each one is randomly assigned to a node of the cluster, which monitors this process. A client process could send a join request to a node of the cluster and at the same time send also it's client Id and it's IP:Port. As a result, the node randomly assigned to monitor this client, informs the GM cluster of the new client process and after everyone proposes this process to consensus, a new View is installed in every node. This is essential to achieve replicated machine state for this GM

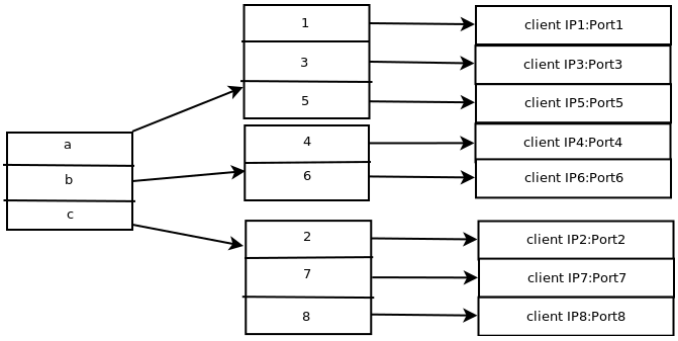

Fig. 2: Key value store

Service. Formally the events and the requests of the GM Service are described as follows:

- **Request 1:** <join, V> | V : Installs a new view to the group membership service **V = nodeId,clientId,client(IP:Port)**
- **Request 2:** <leave, clientInfo> | clientInfo : Removes a client process from the group membership service **clientInfo = clientId,client(IP:Port)**
  A leave event is occurred when a client process is either detected crashed or requested to leave the Service.
- **Request 3:** <snapshot, V> | V : prints a snapshot of the members of a specific group.
- **Event 1:** <ping, V> | V : Pings the members of the group it is responsible for.
- **Event 2:** <remove, clientInfo> | clientInfo : Removes client from the View.

So, having presented the basic idea of how a distributed system implements GM Service, in the next subsection we describe the architecture followed for the project.

Architecture

As it is shown in Figure 3 each node of the GM cluster consists of a key value store and a REST interface, which accepts POST and GET methods(Figure 4). POST method corresponds to the clients' join requests and GET method corresponds to the request to obtain a snapshot of a specific group.

III. Implementation and Integration of Raft

The implementation of our project was based on the Go programming language created by Google in 2007. The nodes of the GM cluster are implemented by the go package cluster, which consists of four .go files (httpapi.go, kvstore.go, listener.go, node.go). Briefly, the *node.go* file contains the go routines and the type struct that defines the nodes of the GM cluster, the *httpapi.go* implements the REST methods

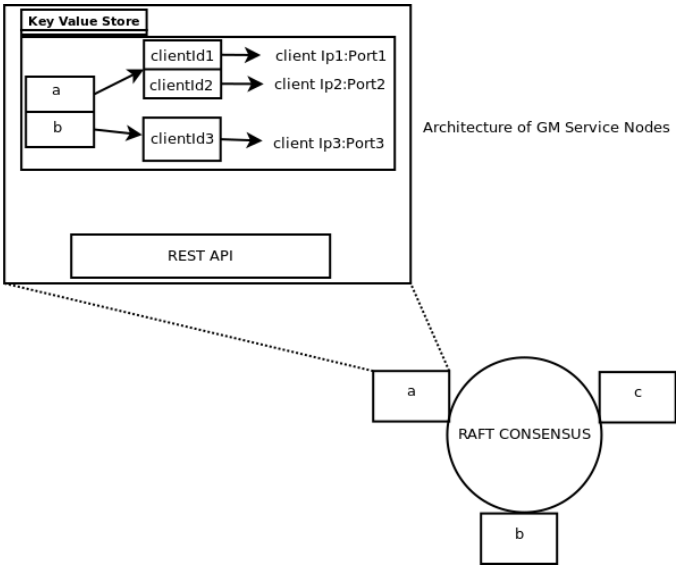

Fig. 3: Overview of the architecture of the Service

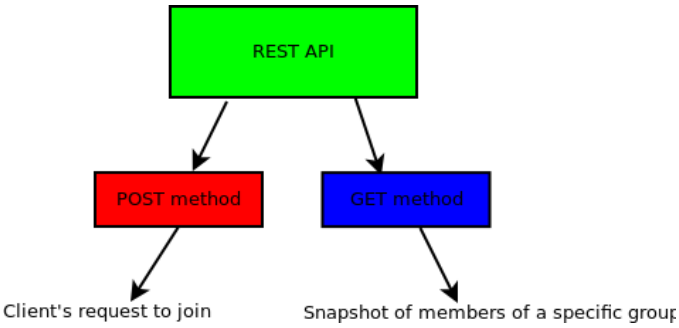

Fig. 4: REST interface

POST and GET, the *listener.go* contains the implementation of Listener that is used by serveRaft() and at last the *kvstore.go* implements the key value store.

```
type KVStore struct {
        mu      sync.RWMutex
        store map[string]map[string]string
        proposeChannel chan<- string
}
```

The mutex mu is used so that operations on the key value store are executed atomically, as a result the store is treated as critical section in our project.

To achieve concurrency we used two channels: the proposeChannel which is used to propose values to raft and the commitChannel which is used for values that have been committed to the log and can be appended to the key value store.

Raft

One of the most challenging aspects of our project was to integrate Raft according to the documentation of etcd. A node in the cluster can have three states: follower, candidate, or leader. Consensus in the cluster is maintained by the leader,

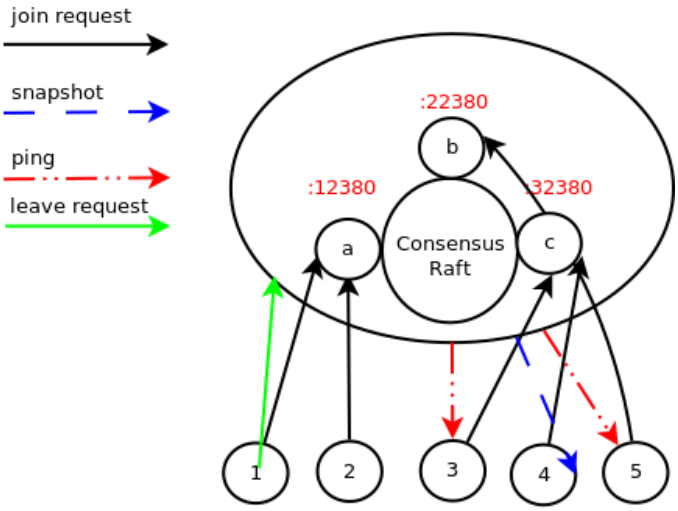

Fig. 5: Sample Execution

which is responsible for log replication. To become the leader a node will change its status from follower to candidate.

- **Leader Election:** the leader in the cluster sends heartbeat messages to the followers. If a follower does not receive a heartbeat within the election timeout it will transition state from follower to candidate. The candidate starts a new election term and increases the term counter, votes for itself and sends a RequestVote message to the cluster members. A member node will vote on the first candidate that sends a RequestVote message, and it will only vote once during a term.

- **Log Replication:** all changes go through the leader, leader gets proposal, creates a log entry (uncommited), then replicates the entry to the followers, when a majority of the followers have written the log entry the leader commits it (the leaders state has now changed), the leader notifies the followers that the entry has been committed. At this point the cluster is in consensus.

- **Cluster Majority:** cluster consensus requires at least two nodes; thus three nodes are needed to create a resilient cluster. In case of a cluster split there will be one part that can achieve consensus, and move forward.

The documentation of raft's implementation by etcd[3] was very instructive. In addition, we had to use the package *rafthttp*, so that the leader can send the committed values to the followers.

IV. Use Cases and Properties of the Abstraction

In Figure 5 we describe a sample execution that shows every use case that could happen in our distributed system. Clients send join requests in case they want to join the GM Service as

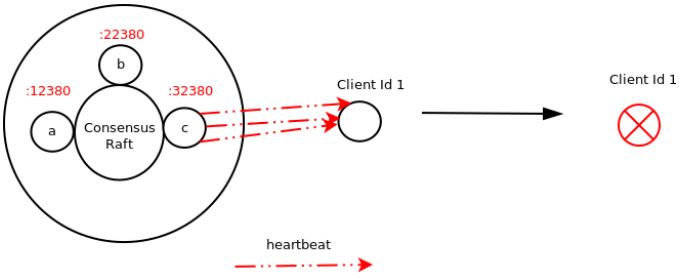

Fig. 6: Failure detection implemented by our system

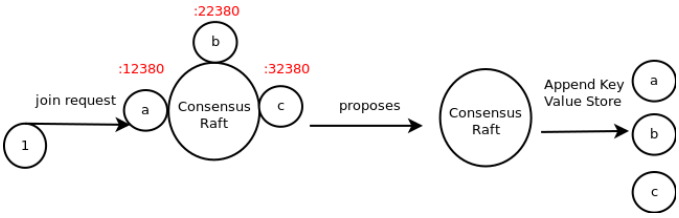

Fig. 7: Sample execution of join request

it is seen in Figure 7, they shall request to leave the service and they shall request a snapshot of the *View* of a specific group, providing the Id of the group. Of course, it is important to refer to the events that can happen in our system. The heartbeat requests, when every 5 seconds a node sends heartbeat requests to the client processes it is responsible to monitor. If a client process does not respond for three consecutive times, then it is considered as crashed and the remove event is triggered so it is removed from the key value stores of each node.

Properties of our system

In this section we are going to analyze and discuss the properties of our system.

- **Uniform Agreement:** when a process installs a view V, then every node installs view V.
- **Completeness:** if a process p crashes then eventually every correct node installs a view V: p not in V and if a process q requests to join then every correct node installs a view V': q in V'.
- **Accuracy:** If some process installs a view V with q not in M, then q has crashed.
- **Validity:** If some process installs a view and q in M, then q previously requested to join.

V. Challenges, Go Tools and Documentation

We faced a lot of challenges until we managed to implement a functional distributed system. We used the rafthttp package of etcd, so as to create a communication link between the nodes of the GM cluster. The command Send() sends the message to the remote peers(followers). In terms of properties it is reliable. The function is non-blocking and has no promise that the message will be received by the remote. When it fails to send message out, it will report the status to underlying raft.

VI. Conclusion

For this project we implemented a scalable distributed system that offers consistent *Group Membership* services providing high availability and large-scale storage services. We achieved *concurrency* with Go programming language and consensus by integrating etcd's raft implementation to our project.

References

[1] Fred B. Schneider, Implementing Fault-tolerant Services using the State Machine Approach, 1987, Cornell University.
[2] Christian Cachin, Luis Rodrigues and Rachid Guerraoui, Introduction to Reliable and Secure Distributed Programming (Chapter 6.7), 2011, Springer.
[3] *Raft Documentation* https://github.com/coreos/etcd/tree/master/raft