Leader Election using RAFT Consensus Protocol

1. Group Members

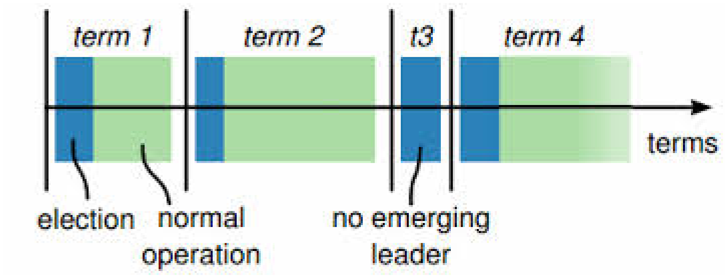
a. Naman Gupta - 133050012
b. Anshul Agarwal - 143050044

2. Introduction

Raft is a consensus algorithm for electing a leader in a distributed environment and managing a replicated log. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes Raft more understandable than existing and well known Paxos and also provides a better foundation for building practical systems. In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety. Here our main focus is on electing a unique leader in a distributed environment, while ensuring safety and liveness.

3. Features

- A Raft cluster contains some number of servers; here we are assuming five is the number, which allows the system to tolerate two failures.
- At any given time each server is in one of three states:
    - LEADER
    - FOLLOWER
    - CANDIDATE
- Under any normal circumstance, there is exactly one leader and all of the other servers are followers.
- Followers are passive: they issue no requests on their own but simply respond to requests from leaders and candidates.
- The leader handles all client requests.
    - Note: if a client contacts a follower, the follower redirects it to the leader.
- The third state, candidate, is used to elect a new leader.
- Raft divides time into terms of arbitrary length, as shown in figure below



- Terms are numbered with consecutive integers.
- Each term begins with an election, in which one or more candidates attempt to become leader
- If a candidate wins the election, then it serves as leader for the rest of the term.
- In some situations an election will result in a split vote.
    - In this case the term will end with no leader.
    - A new term (with a new election) will begin shortly.
- Raft ensures that there is at most one leader in a given term.
- Terms act as a logical clock in Raft, and they allow servers to detect obsolete information such as stale leaders or leaders elected in old term.
- Each server stores a current term number, which increases monotonically over time.

- Current terms are exchanged whenever servers communicate.
- If one server’s current term is smaller than the other’s, then it updates its current term to the larger value.
    - If a candidate or leader discovers that its term is out of date, it immediately reverts to follower state.
    - If a server receives a request with a stale term number, it rejects the request.
- The basic consensus algorithm requires only two types of RPCs
    - ***RequestVote*** : Initiated by candidates during elections
    - ***Append-Entries***: Initiated by leaders to replicate log entries and to provide a form of heartbeat.
- Servers retry RPCs if they do not receive a response in a timely manner, and they issue RPCs in parallel for best performance.

3. **Leader Election Algorithm using RAFT**

   a. Raft uses a heartbeat mechanism to trigger leader election.
   b. Whenever servers start up, they begin as FOLLOWER.
   c. A server remains in Follower state as long as it receives valid RPC from a Leader or Candidate.
   d. Leaders sends periodic heartbeats (Append-Entries RPCs that carry no log entries) to all followers in order to maintain their authority.
   e. If a follower receives no communication over a period of time called the 'election timeout', then it assumes there is no viable leader and begins an election to choose a new leader.
   f. To begin an election, a follower increments its current term and transitions to candidate state.
   g. It then votes for itself and issues Request-Vote RPCs in parallel to each of the other servers in the cluster.
   h. A candidate continues in this state until one of the three things happens:
      i. It wins the election
      ii. Another server establishes itself as leader, or
      iii. A period of time goes by with no winner.
   i. A candidate wins an election if it receives votes from a majority of the servers in the full cluster for the same term.
   j. Each server will vote for at most one candidate in a given term, on a first-come-first-served basis.
   k. The majority rule ensures that at most one candidate can win the election for a particular term.
   l. Once a candidate wins an election, it becomes leader.
   m. It then sends heartbeat messages to all of the other servers to establish its authority and prevent new elections.
   n. While waiting for votes, a candidate may receive an Append-Entries RPC from another server claiming to be a leader.

o. If the leader’s term (included in its RPC) is at least as large as the candidate’s current term, then the candidate recognizes the leader as legitimate and returns to follower state. If the term in the RPC is smaller than the candidate’s current term, then the candidate rejects the RPC and continues in candidate state.

p. The third possible outcome is that a candidate neither wins nor loses the election :

    i. if many followers become candidates at the same time, votes could be split so that no candidate obtains a majority.

q. When this happens, each candidate will time out and start a new election by incrementing its term and initiating another round of Request- Vote RPCs.

r. However, without extra measures split votes could repeat indefinitely.

s. Raft uses randomized election timeouts to ensure that split votes are rare and that they are resolved quickly. To prevent split votes in the first place, election timeouts are chosen randomly from a fixed interval. This spreads out the servers so that in most cases only a single server will time out; it wins the election and sends heartbeats before any other servers time out.

t. The same mechanism is used to handle split votes.

    i. Each candidate restarts its randomized election timeout at the start of an election, and it waits for that timeout to elapse before starting the next election.

    ii. This reduces the likelihood of another split vote in the new election.

4. **Implementation Details :**

   a. **Cluster Creation :** We need group of servers communicating with one another forming a cluster. Cluster information will be provided in a **config.xml** file. Sample :

```
<config>
        <NumServer>2</NumServer>
        <servers>
                <ID>1</ID>
                <HostIP>localhost</HostIP>
                <Port>1234</Port>
        </servers>

        <servers>
                <ID>2</ID>
                <HostIP>localhost</HostIP>
                <Port>1236</Port>
        </servers>
</config>
```

   Id : Unique Id of the server
   HostIP : Ip Address of the Server.
   Port : Port on which Server will listen.

   RafterServerPool.java reads the config file and spawns the server. Servers will listen on the **port** for incoming requests. We will initially planned to use RMI but due to non-availability of computing resources decided to use socket connection where all the servers will be executed on a single machine listening on different ports.

   b. **Leader Election :** Each Server initiates Leader Election when its election timeout goes out without receiving any heartbeat. Following java classes contains implementation Details of component of Leader Election

a. **RaftServerPool.java** : This class is responsible for creating pool/clusters of Servers. It reads config file which contains information about server ip-address and port numbers.

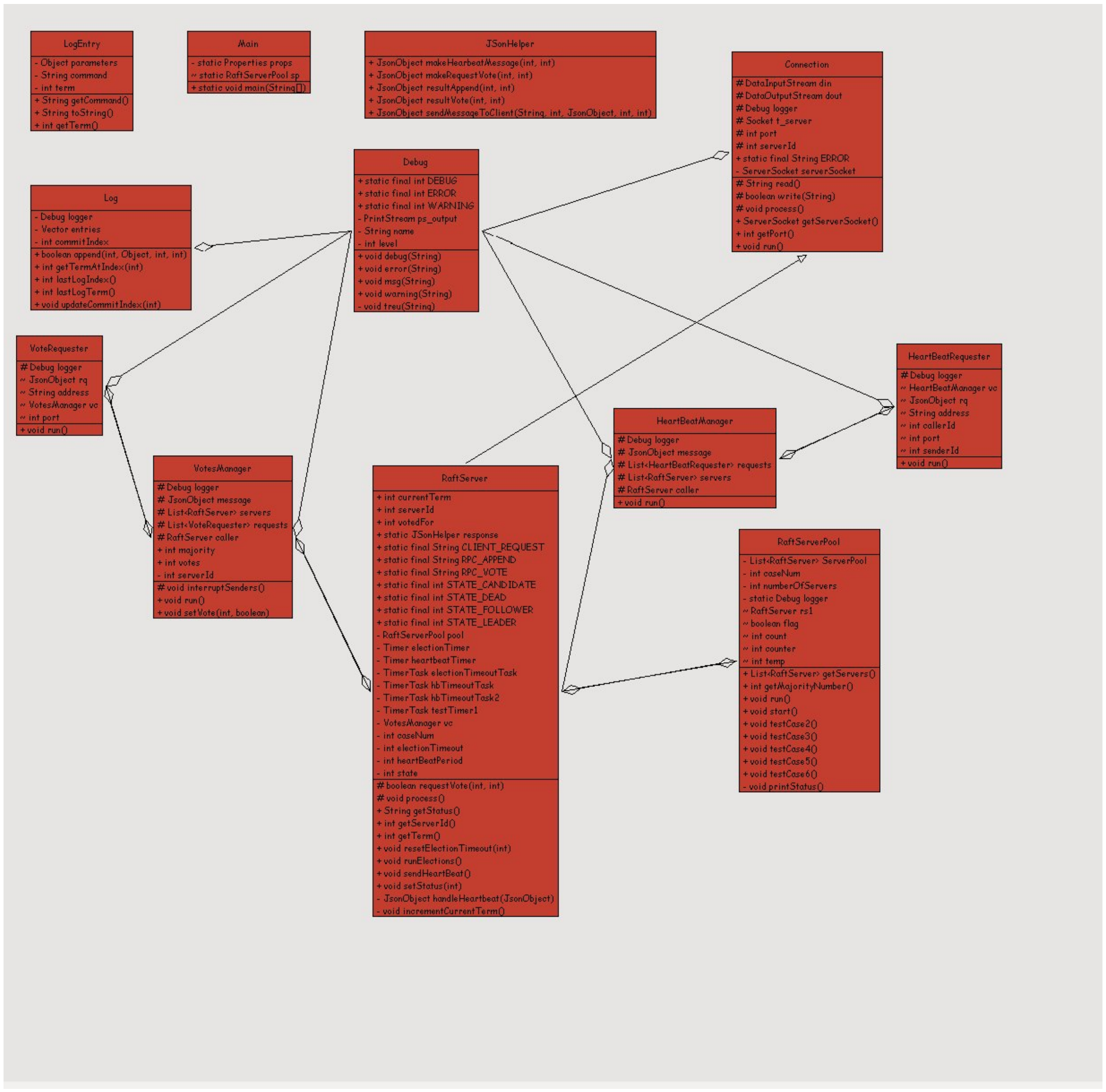
b. **RaftServer.java** : The core functionality of Leader election is written in this class. This class act as a Server. Each server will have its own electionTimeout and heartbeat timeout.

c. **VotesManager.java** : Spawned by RaftServer.java when server's electionTimeout is over.
   i. It creates client socket connection with other servers
   ii. Sends Request Vote Messages
   iii. Based on servers reply, increment the vote count
   iv. Declare the server (which spawned it) as a leader as soon as it gets majority votes.

d. **HeartBeatManager.java :** Responsible for sending heartbeats in parallel to other servers.
   i. Creates client socket connection with other servers
   ii. Sends Heartbeat Messages

e. **JsonHelper.java** : Communication messages for Leader Election are modelled as Json Messages. This class is responsible for creating messages for elections and heartbeat.

5. Class Diagram



6. **Environment: JVM Version :** 1.6+

7. **Simulation Test :** Following Simulations were performed with a cluster having 5 servers.

    a. **Kill one Leader:** As soon as a Leader is elected, the leader is killed and the cluster is observed as to how soon it recognizes that the leader is down, and starts new election process amongst the remaining 4 servers and elects a new leader.

    b. **Cascade Failure of elected Leaders(upto 3 leaders) :** As soon as a leader is elected, it is brought down. The cluster still performs as expected by electing a new leader from possible number of servers if and only if a majority exists. So the leader election process stops at 2 process remaining, because of lack of majority.

    c. **A Leader Goes Down and Comes Again as a follower:** When the leader is down, the cluster soon starts new election process and when the old leader comes back again as a *FOLLOWER*, it participates in the election process by giving votes.

    d. **More than Two Leader go down and come back as a follower:** Now this situation is different from previous case. Here two leaders go down and come again as FOLLOWER. Now when first leader comes up as a follower it should be able to recognize the new leader elected by remaining 3 servers. And when next leader comes up as a follower, the system should still behave in the normal way, as in the fact that the new server should recognize the existing leader and should not cause any instability. It can participate in the election process, but cannot elect itself as the second new leader.

    e. **One Leader Goes Down and Comes back as a leader:** When the leader goes down, the remaining processes elect a new leader. Now when the old leader comes up again as a Leader, then there are 2 leaders in the system sending heartbeats to all servers. The only way to differentiate between them is through use of term numbers. When the new leader and servers receive heartbeats from the old leader, they observe that it has lower term number, indicating that it is an old leader and discard its request. Similarly when the old leader receives heartbeats from new leader, it compares its term number and realizes that it is old and changes itself to Follower. Thus eventually ***SAFETY*** is guaranteed as there is only one leader present in the system.

7. Objectives that were not achieved

Initially, we had planned to develop a key value store which required implementation of two ideas:

a. LEADER ELECTION
b. LOG REPLICATION

Here we could achieve only Robust leader election implementation which elects leader in all possible cases of failure. Also it guarantees **Safety** and **Liveness** in all possible failure situations.

Proper implementation of leader election alone took all the time and to be able to test in all possible conditions was a challenging task. Hence Log replication was not achieved in the stipulated time.

References:

1. Ongaro, D., & Ousterhout, J. (2013). In Search of an Understandable Consensus Algorithm. *Ramcloud.Stanford.Edu*. Retrieved from https://ramcloud.stanford.edu/wiki/download/attachments/11370504/raft.pdf