CS553: Project Proposal
Distributed Key-Value Storage System using Raft Consensus Algorithm

Vennela Chava - vc494
Gayathri Ravipati - gr485

Abstract:

Distributed consensus is one of the fundamental problems in distributed systems. Consensus is a process where multiple servers agree on the same information, it is important for developing fault-tolerant distributed systems. Raft Algorithm is used to achieve consensus among multiple servers to maintain data consistency across the system.

Raft Algorithm:

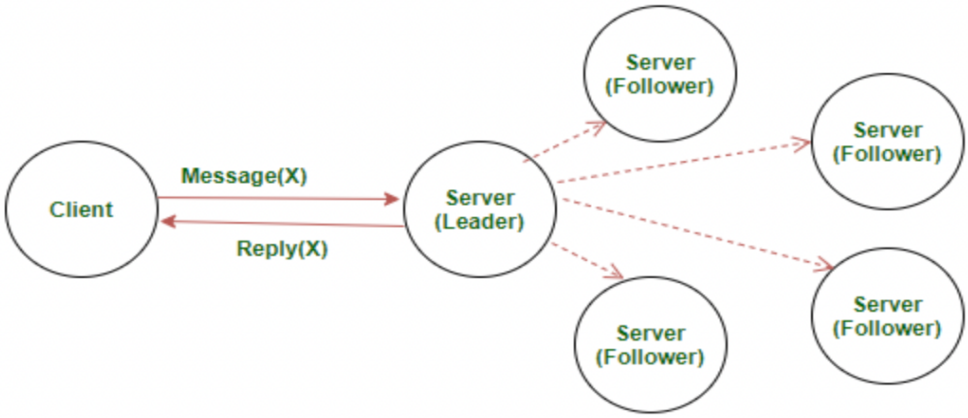
The Raft protocol was created by Diego Ongaro and John Ousterhout at Stanford University, and Diego earned his Ph.D. in 2014. The goal of Raft was to provide a more understandable approach to achieving consensus, which is the process of agreeing on a single value among a group of distributed systems. Earlier Paxos Algorithm was considered very difficult to comprehend and implement. The raft was developed as a response to this challenge, and the title of Diego's paper reflects this: 'In Search of an Understandable Consensus Algorithm'.

Project Description:

Implementation of the Raft Algorithm introduced in the above research paper to insert and retrieve values based on API calls made by the client and analyze the algorithm's performance by varying parameters.

Description of terms used in the implementation:

- Every server will be in one of the three states
    - **Leader -** Any request from the client flows through the Leader. All the other servers interact with the leader to maintain consistency in the log. There is utmost one leader at any point in time.
    - **Candidate -** Any server which doesn't receive an update from the leader for a certain period of time turns out to be a candidate and initiates an election to request votes from the other servers.
    - **Follower -** The follower server interacts with the Leader to sync up the data. If the leader server fails, then the follower turns a candidate.



- Current_term
    - It is incremented every time a server starts an election.

    - It helps followers understand if the leader is sending latest messages with the required information.
    - If the Leader's current_term is less than the follower's current_term then the follower doesn't replicate the information to its log
    - If the candidate's current_term is less than the follower's current_term then the follower rejects its vote for that candidate.
- prev_leader_time
    - Prev_leader_time helps us know what is the latest time when the leader has interacted with any of the servers.
    - It is updated every single time when there is a leader-follower interaction
- election_wait_time
    - Time on how long a candidate waits to receive the votes from other servers.
- leader_tenure
    - Tenure on how long a server can act as a Leader
- time_of_election
    - time_of_election helps us know when is the last time the leader was elected.
    - If the leader's tenure is greater than the current_time - time_of_election then the candidate starts the election
- db - Dictionary to store key-value pairs generated by the client

There are a few variables for each server like port_number(Port_number to listen on), server_id(ID of this server), other_servers(list of all the other servers other than the current one), total_votes, candidate_vote, state, and store(server stores the records when the leader is unknown).

Implementation Details:

Our implementation supports two different API calls from the client focusing on consistency and availability.

- put(key, value) - stores the key-value pair
- get(key) - retrieves the mapped value for the stored key

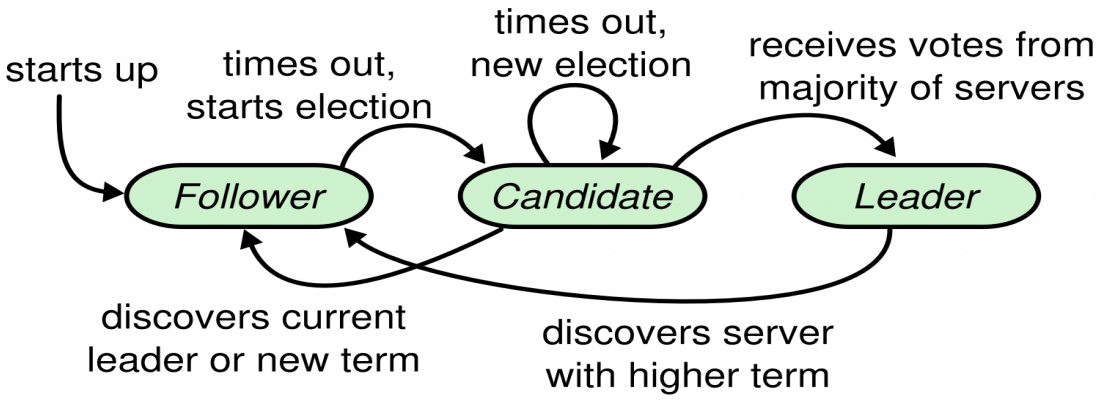
Here is an outline **summary** of our implementation of the Raft Algorithm which supports leader election, log replication, fault tolerance, and safety.

1. The key-value pairs are stored in the Python dictionary.
2. We have used the below message format for communication (between servers)
   The sample message format of a redirect message is:
   redirect_message = {"source": rcvd_message['dst'], "destination": rcvd_message['src'], "leader": self.leader, "req_category": "forward2leader", "Message-ID": rcvd_message[msg_id]}
   We ensured that source, destination, leader, and request category are mandatory and the remaining parameters keep changing based on the category of the message sent.
3. Initially, all the nodes are in a follower state and elect a leader and then the Leader-Follower-Candidate cycle continues based on the conditions.
4. The start method is the one, which is responsible for algorithm execution.
    a. The while loop in the method will continue to run until it is interrupted or terminated, ensuring that the server operates as expected.
    b. A server executes its actions based on its state.
        i. Leader Server - Ensures that the last contacted time to any of the servers (prev_leader_time) is not greater than 0.30 seconds.
        ii. Candidate Server - Starts the election again if the election_wait_time is expired.

iii. Follower Server - If the leader is unknown or leader_tenure is expired and if it has been more than election_wait_time waiting for a leader to be nominated, it turns into a candidate and starts the election.

c. If the server socket has any incoming messages then appropriate methods are called based on the source and the type of the message received.

5. When a server receives a heartbeat message from the leader
    a. If the current state is leader, it begins the election process as this indicates there is another leader existing and there is scope for conflicts
    b. If it is a follower, it ensures that the current_term of the leader is greater than its term and updates the prev_leader_time to the current time
    c. Redirects any unsent messages in its store to the leader and initializes the store
6. When a Client_Request(get/put) is received, it is executed based on the server's state.
    a. Leader: Calls Leader_Client_Request, if **get** then it retrieves the value from the db and replies with the success or failure message. If **put** then it adds the key-value pair to the db, adds the record to the log, and sends the update to all the remaining servers. Once all the other servers or followers are updated, the leader sends a confirmation message to the client stating that the put was successful.
    b. The server is a follower and Leader is known - It redirects the message to the client stating that it is not the leader and updates the client with the leader.
    c. The server is a follower and the Leader is unknown - It adds the message to the store and redirects to the client with the new Leader whenever it receives the heartbeat message.
7. When a server receives Vote_Decision then
    a. The vote is rejected if the candidate's term is less than the follower's term number.
    b. The vote is approved and the approvedvote message is sent to the candidate
8. If the server's state is the candidate and the type of the message is approveVote then
    a. We ensure that the number of votes received is greater than the quorum(half of the servers + 1) and if yes, we declare the candidate as a leader and rebuild the database from the log so that there are no missing records.
9. Whenever there is put request from the client the log is updated(Add_to_log) to all the servers, which is basically the AppendEntries procedure call and whenever an election is started(Start_Election), the Request_Vote procedure call is sent to all the servers from the candidate.



Simulation:

The **overview** of the code structure and functionality of the file simulator.py file are as follows:

1. The client class has the following methods

a. generate_string - Generates a random string of length 8(combination of digits and lowercase letters)
b. des_servers - Tries to find the actual leader so that the client can send the requests
c. generate_get - generates get requests
d. generate_put - generates put requests
e. Calucualte_responses - to calculate the unanswered put and get requests based on the responses received from the server.
f. Form_client_request - Before making a get request, the client picks a random key from its store and makes a get_request to the server with this key. In the case of a put request, a random key_value pair is generated.
g. Send_message - On receiving a response from a server, it retrieves the Message_ID and verifies it with its request Message_ID(safety_check). It also updates wrong_get, repeated_req, latency, and redirects(forward2leader) based on the response message received from the server.

2. The server class has three methods
    a. start_servers - runs the Raft Algorithm(raft.py) and starts the servers
    b. Shutdown_server - shuts down the server when there is a crash or if we intend to do it for measuring the metrics
    c. Server_response - Ensures the target is active and has an established connection for communication, also acknowledges the status based on the message delivery.

3. The Begin class has the following methods:
    a. Start_beg - generates server_id, client_id, starts the clock, servers, and clients, and stores the trigger_points based on the actions given in the metrics and gives the result of the execution.
    b. Shutdown - Shutdowns the required servers
    c. Fail_leader - It shutdowns the leader as requested in the action.
    d. Fail_follower - Shutdowns a random active follower as requested in the action.
    e. Push_get_req - Chooses the random client and generates a client request(get)
    f. Push_put_req - Chooses the random client and generates a client request(put)
    g. Send2server - In case of a failed message delivery, it removes the server id that it has from the active servers.
    h. Ts_q - Based on the values given in the metrics, it registers the timestamp trigger values for all the requests and actions.
    i. Check_address - It validates the server/client ID
    j. Show_results - helps for the output format
    k. Fetch_message - It validates each component of the message received and updates the parameters like total_client_requests based on the destination in the message. In case of any error, it shuts down the server.
    l. Sever_quit - In case of a crash/error, it removes the server ID from the active server list and shuts it down.
    m. Validation - It ensures that our system has met all the minimum performance-related requirements like if the servers answered a minimum number of requests from clients.

Source Code:

Github link - https://github.com/ChavaVennela/DIS-Project
raft.py - server file
simulator.py - simulator file

Commands to execute the code - python simulator.py

To view the result on file: python simulator.py > output.txt

Results:

We have analyzed the algorithm based on its accuracy and performance.

Given Metrics:

```
metrics_1={
      "server_code": 'raft.py',
      "maximum_get_create_fail_fraction": 0.1,
      "appends_batched_fraction": 0.5,
      "maximum_get_reqs_fraction": 0.50,
      "maximum_put_reqs_fraction": 0.50,
      "number_of_clients": 8,
      "total_execution_time": 30,
      "servers": 5,
      "client_requests": 500,
      "gp_ratio" : 0.8,
      "only_hb" : 5,
      "stop_requests" : 2,
      "max_packets" : 20000,
      "actions" : []
   }
```

Result_1:

```
Leaders:
['unkown', '0000']
Servers that crashed during execution:  0
Servers that were failed by simulator:  0
Total messages sent:  2376
Total client get() requests:  397
Total client put() requests:  110
Total duplicate responses:  0
Total unanswered get() requests:  0
Total unanswered put() requests:  0
Total number of messages are redirected to leader by followers:  7
Total get() failure:  0
Total put() failures:  0
Total get() with wrong_get response:  0
Mean request/response latency:  0.0034139447898789504
Median request/response latency:  0.002434968948364258
No errors, everything is working perfectly:)
```

Conclusions_1:

1. Metrics_1 has no actions and can be considered a default metric when we run the simulator file.
2. From Results_1 we can say that there are no server crashes, no failures, no unanswered_requests, and no duplicate_responses and we have also given the requests_latency.

By this, we can conclude that the raft algorithm satisfies the consistency, log_replication, availability, and safety requirements.

metrics_2={

```
    "server_code": 'raft.py',
    "maximum_get_create_fail_fraction": 0.1,
    "appends_batched_fraction": 0.5,
    "maximum_get_reqs_fraction": 0.50,
    "maximum_put_reqs_fraction": 0.50,
    "number_of_clients": 8,
    "total_execution_time": 30,
    "servers": 5,
    "client_requests": 500,
    "gp_ratio": 0.2,
    "only_hb" : 5,
    "stop_requests": 2,
    "max_packets": 20000,
    "actions" : [{"category": "fail_leader", "trigger_point": 8},
                 {"category": "fail_leader", "trigger_point": 16}]
}
```

Result_2:

Leaders:
['unkown', '0001', 'unkown', '0004', 'unkown', '0003']
Servers that crashed during execution: 0
Servers that were failed by the simulator: 2
Total messages sent: 1521
Total client get() requests: 1751
Total client put() requests: 1616
Total duplicate responses: 0
Total unanswered get() requests: 37
Total unanswered put() requests: 153
Total number of messages are redirected to leader by followers: 2867
Total get() failure: 0
Total put() failures: 0
Total get() with wrong_get response: 0
Mean request/response latency: 0.0009187446812769884
Median request/response latency: 0.0006220340728759766
No errors, everything is working perfectly:)

Conclusions_2:

In Metrics_2 we have given action to fail_the _leader twice at two different trigger points, and there was a successful leader election process post the action execution. And the new leader IDs are updated in the Leaders array. By this, the raft algorithm has satisfied leader_election requirements.

How to evaluate different metrics?

We can execute multiple metrics by changing the category(fail_leader / fail_follower) in the actions array. This can be done in simulator.py in the metrics dictionary(Line number -18)

Disadvantages:

The Raft algorithm has a single point of failure in the leader, which means that the system should have a quick response to select a new leader in case of failure. Moreover, as all client communications go through the leader, the system's scalability may be limited, causing performance issues.

Challenges:

1. To get the structure of this code - Even though there are considerably good resources explaining the raft algorithm, it was a little difficult to finalize the structure of the code and get it working.
2. Working on the simulator code structure was a big challenge in this project.
3. Benchmarks - To come up with the idea of what we have to set the metrics upon and the categorization of the same.

References:

1. https://web.stanford.edu/~ouster/cgi-bin/papers/raft-atc14.pdf
2. https://www.geeksforgeeks.org/raft-consensus-algorithm/#
3. http://thesecretlivesofdata.com/raft/
4. https://www.youtube.com/watch?v=IPnesACYRck&ab_channel=MartinKleppmann
5. https://realpython.com/python-sockets/#background
6. https://github.com/saimihirj/raft_consensus_algorithm
7. https://codeburst.io/making-sense-of-the-raft-distributed-consensus-algorithm-part-1-3ecf90b0b361
8. https://codeburst.io/making-sense-of-the-raft-distributed-consensus-algorithm-part-2-4f12057b019a
9. https://codeburst.io/making-sense-of-the-raft-distributed-consensus-algorithm-part-3-9f3a5cdba514