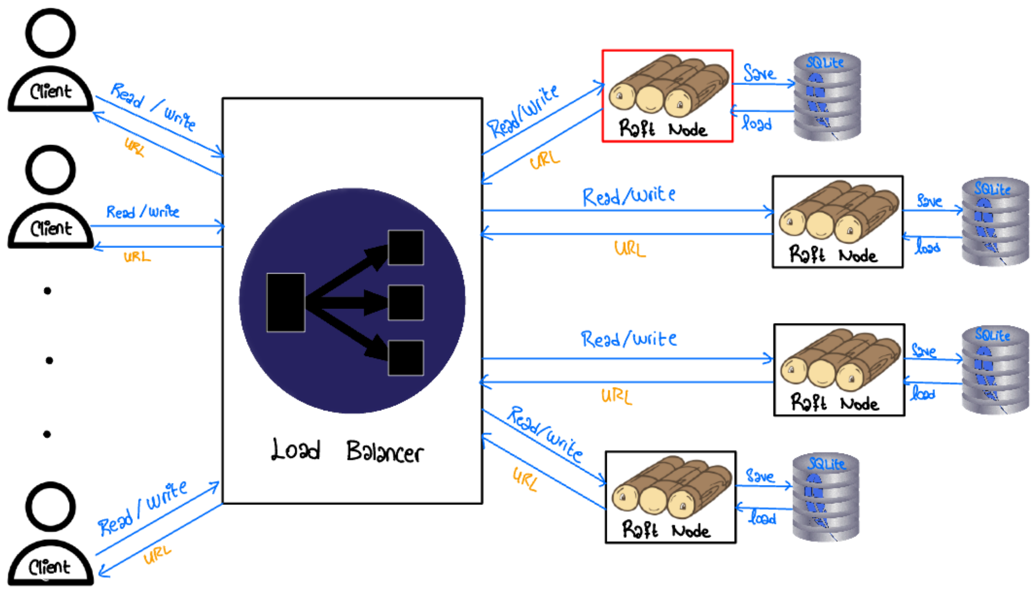
URL Shortener Report Questions

Cloud-Based Data Processing (IN2386)

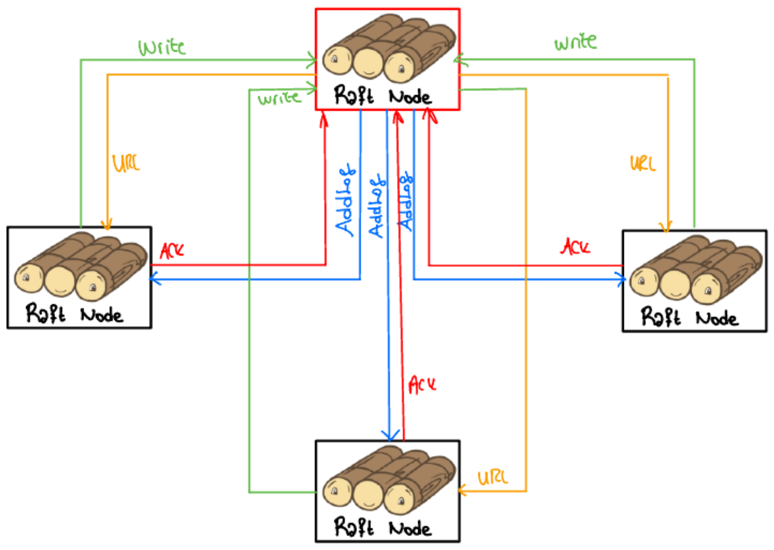
Turker Koc

Ege Kocabas

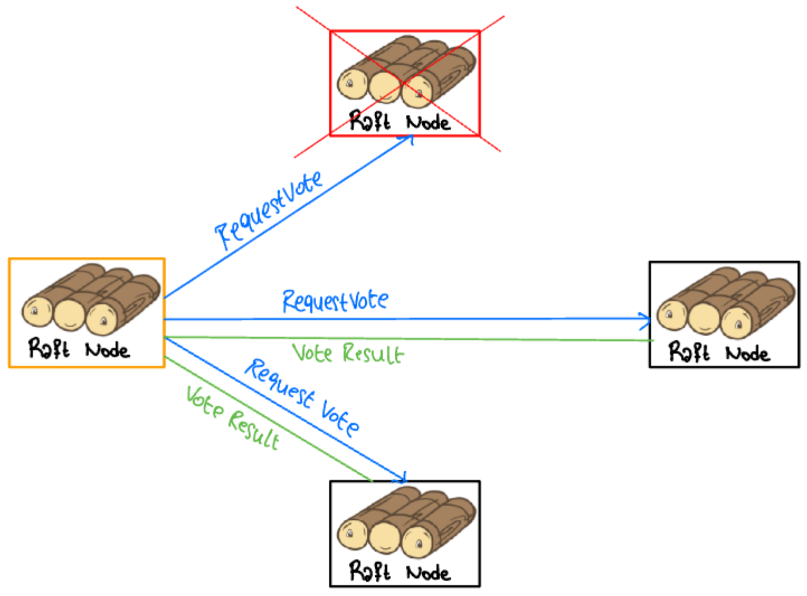
Describe the design of your system



- We have created our *RaftNode*s:
  - Each node commits their log into the SQLite DB.
  - Each node communicates with each other via RPC calls.
  - Each node also communicates with the Load Balancer to get Read/Write requests.
- Our Load Balancer:
  - Load balancer is responsible to receive Read/Write requests from clients.
    - Each request is forwarded to the corresponding node.
    - Our load balancer calls each node sequentially.
    - **Write Requests**: Simply, our RaftNodes are added to a vector and we start to forward requests to the node at index 0, then increment this index when a new request received.
    - **Read Requests**: Similar to the write requests, however we seek a quorum of read requests from our RaftNodes.
- Client:
  - Clients can send Read and Write requests to the load balancer and get the corresponding response.



How does your cluster handle leader crashes?



- For each RaftNode we have 2 timeout variables:
    - **Election Timeout**: At the initialization of each node, we assigned a random double value between 10 and 25. Reason for this is to make the leader election phase faster, since some candidate nodes are less patient than others. Once election times out for the current candidate node it starts a new election with a new term.

    - **Heartbeat Timeout**: This variable is also created at the initialization of each node with fixed double value 10. If the time passed since the leader communicated with the follower is bigger than heartbeat timeout, the follower node starts an election and becomes a candidate.
- As a result, if any node suspects the leader has failed or candidate node's election timed out, a new election is started and once the quorum is obtained a new node becomes the leader, and everything keeps functioning without the failed node.
- On the other hand, if the leader recovers from the crash it can join again. Once it recovers it will initialize it's log from SQLite database and become a follower and update it's log from the new leader.

How long does it take to elect a new leader?

- There are basically two scenarios for an election. First; The leader is unresponsive when the raft node is in the "FOLLOWER" state. The second scenario is when the raft node is in the state of "CANDIDATE" and the election is timeout.
    - *In the first scenario*, the following method is used to detect that the leader is unresponsive.

```
bool RaftNode::isLeaderUnresponsive() {

   return difftime(time(nullptr), last_heartbeat_time) > heartbeat_timout;

}
```

    - The "last_heartbeat_time" variable is updated when a message is received from the leader.
    - The "heartbeat_timout" variable is a fixed value as explained in the previous question.
    - If the last message from the leader exceeds the heartbeat timeout, the raft node starts the election.
    - The time is fixed with 10 seconds. The method “isLeaderUnresponsive” is called in each 5 seconds.
    - *In the second scenario*, the following method is used to realize that the ongoing election timeouts.

```
bool RaftNode::isElectionTimout() {

   return difftime(time(nullptr), last_election_time) > election_timout;

}
```

  - The "last_election_time" variable is updated when we started a new election.
  - The "election_timout" variable is a randomly generated value as explained in the previous questions.
  - If we started an election and we can't collect enough votes until the election timeout, the election process starts over.
  - The method “isElectionTimout” is called in each 5 seconds.
- Also the time for electing a new leader changes due to network speed. If the network is not stable, all the raft cluster may be stuck in an infinite loop trying to elect a leader.

Measure the impact of election timeouts. Investigate what happens when it gets too short / too long.

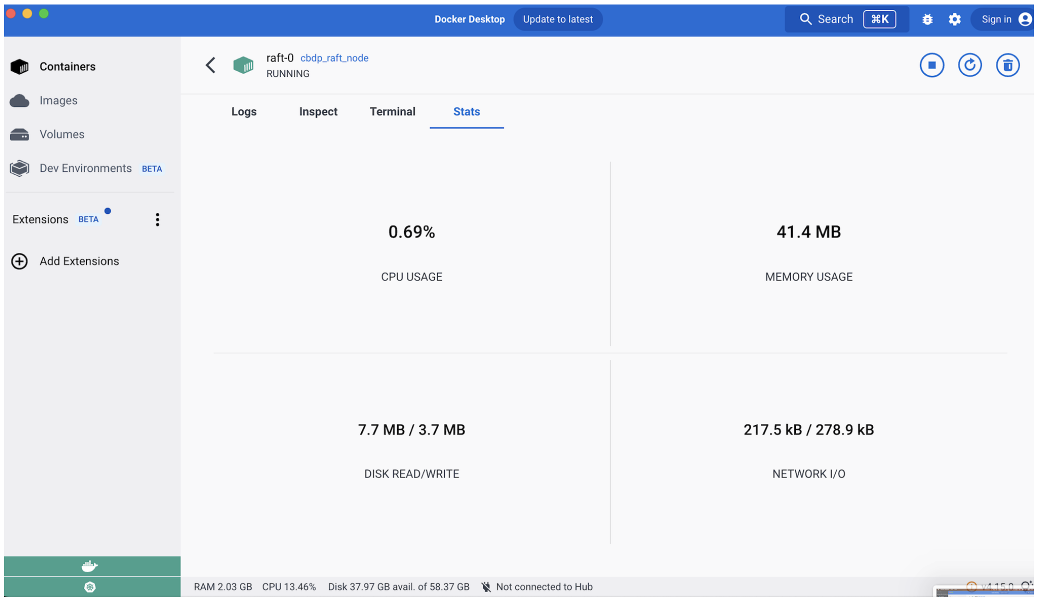
- If the election_timeout value is too low, unnecessary leader elections may occur.
  - This situation arises for the following reason; If the timeout is too low, the followers will constantly switch to candidate status and start an election to become the leader unnecessarily, before the leader replicates log or sends the heartbeath.
- If the election_timeout value is too high, it will take too long to detect the leader crash this time.
  - If the leader becomes unresponsive or crashes and our timeout is too long, it will take too long for us to detect this situation and in the mean time the WRITE operations will stop.

Analyze the load of your nodes

- ***How much resources do your nodes use?***

Observing leader on heavily write request:

  - *Peak CPU:*



raft-0 cbdp_raft_node
RUNNING

0.69%
CPU USAGE

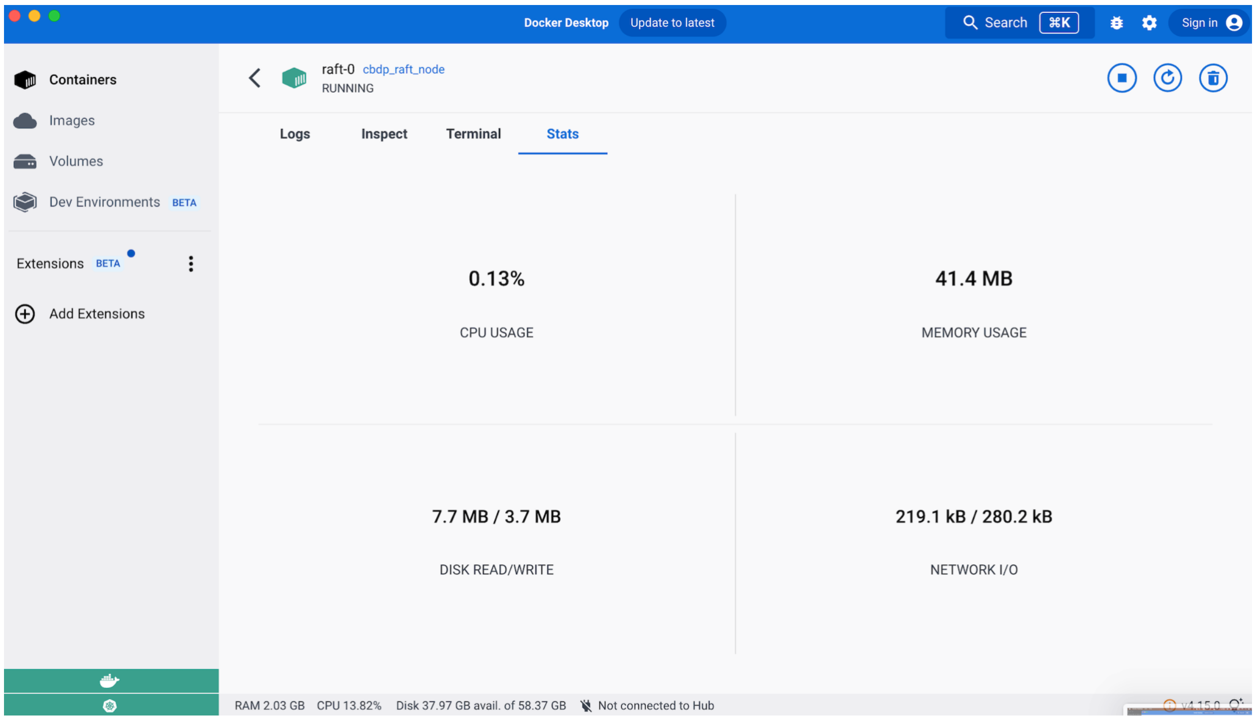
41.4 MB
MEMORY USAGE

7.7 MB / 3.7 MB
DISK READ/WRITE

217.5 kB / 278.9 kB
NETWORK I/O

RAM 2.03 GB CPU 13.46% Disk 37.97 GB avail. of 58.37 GB Not connected to Hub

- *Lowest CPU:*



raft-0 cbdp_raft_node
RUNNING

0.13%
CPU USAGE

41.4 MB
MEMORY USAGE

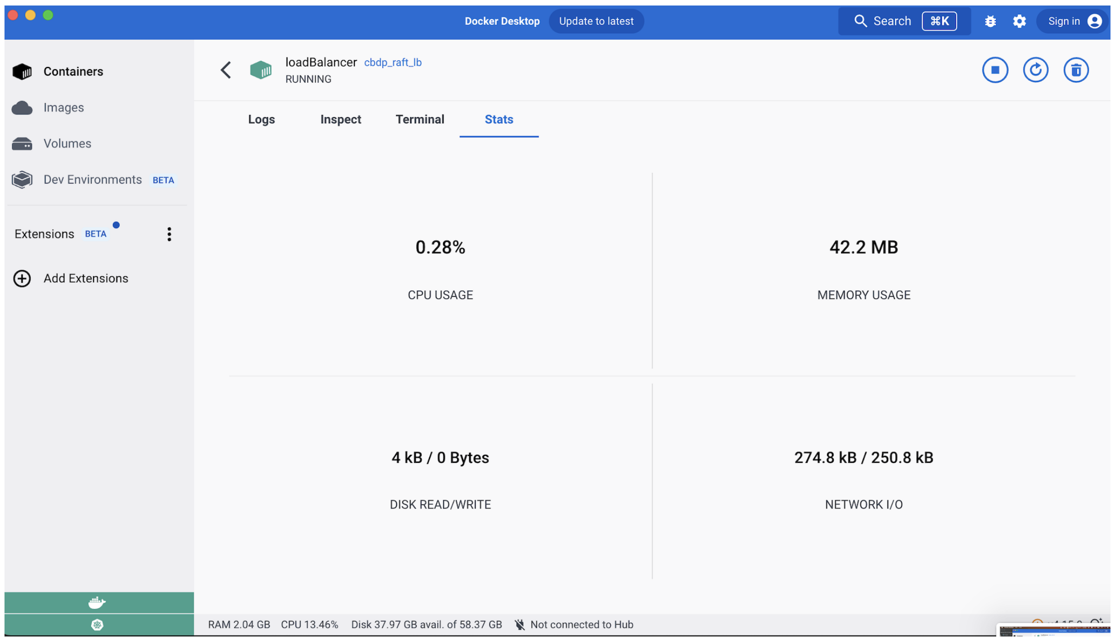
7.7 MB / 3.7 MB
DISK READ/WRITE

219.1 kB / 280.2 kB
NETWORK I/O

RAM 2.03 GB CPU 13.82% Disk 37.97 GB avail. of 58.37 GB Not connected to Hub

Observing load balancer on heavily write & read request:

- *Load balancers CPU usage is the same whether the request is write or read because it only redirects it. We observed that the CPU becomes between 0.25% < X < 0.50%.*



loadBalancer cbdp_raft_lb
RUNNING

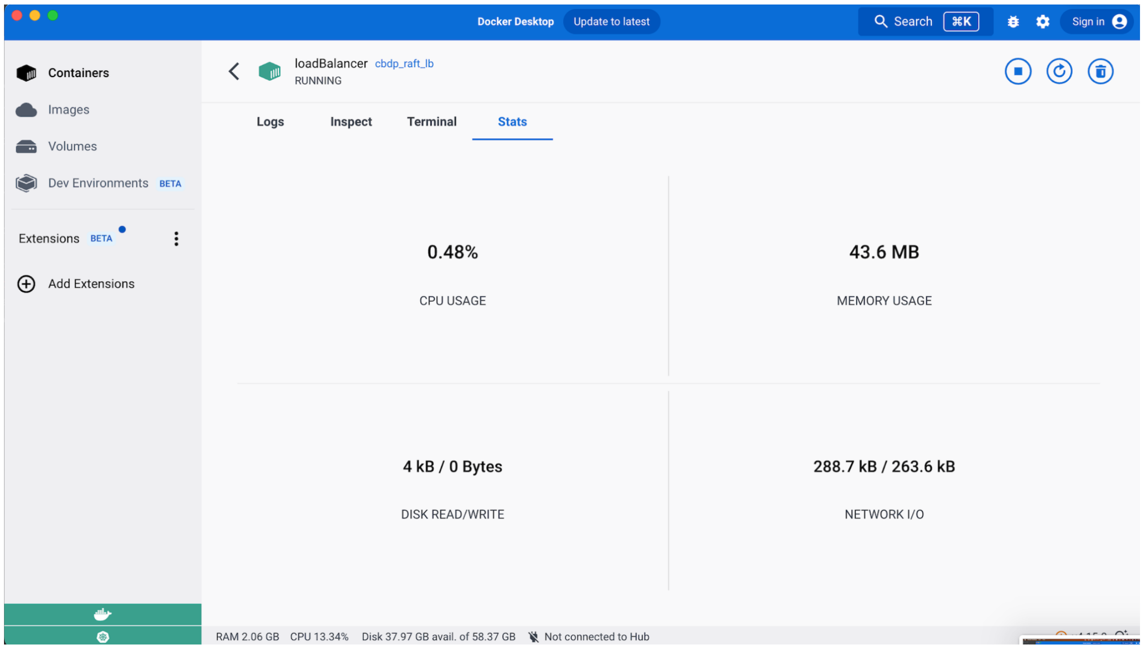
0.28%
CPU USAGE

42.2 MB
MEMORY USAGE

4 kB / 0 Bytes
DISK READ/WRITE

274.8 kB / 250.8 kB
NETWORK I/O

RAM 2.04 GB CPU 13.46% Disk 37.97 GB avail. of 58.37 GB Not connected to Hub



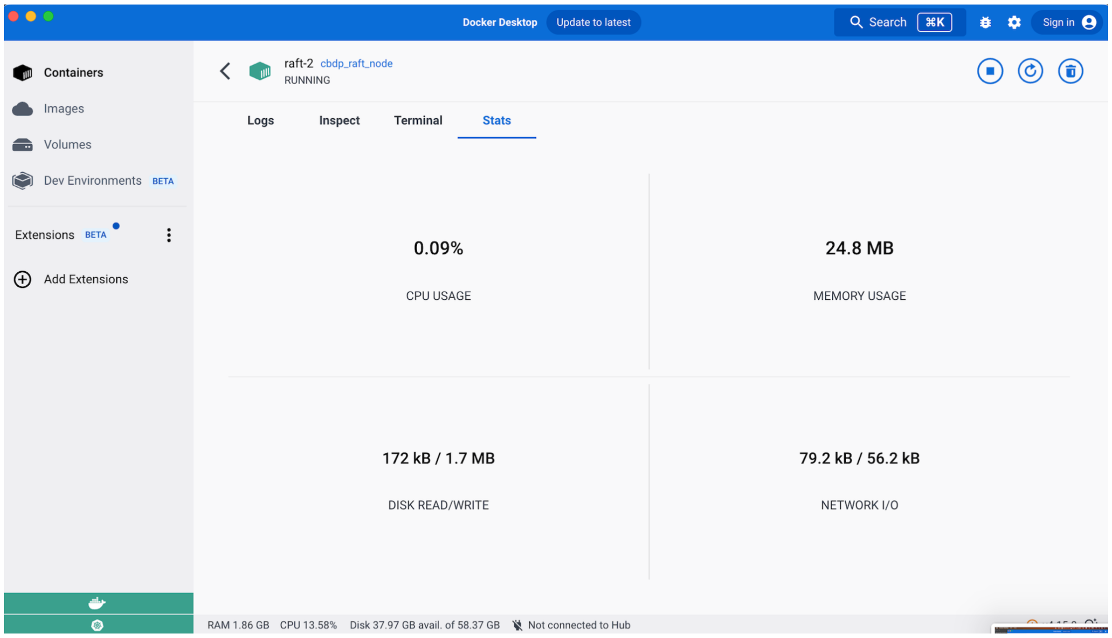
loadBalancer cbdp_raft_lb
RUNNING

0.48% CPU USAGE	43.6 MB MEMORY USAGE
4 kB / 0 Bytes DISK READ/WRITE	288.7 kB / 263.6 kB NETWORK I/O

RAM 2.06 GB CPU 13.34% Disk 37.97 GB avail. of 58.37 GB Not connected to Hub

Observing a follower node on heavily write & read request:

- For writing perspective, most of the time the follower node becomes idle and only periodically cheks timeouts. But whenever leader node sends a data to followers gRPC for adding data, the CPU usage increase.
- For reading perspective, load balancer uses round robin for redirecting read request to the raft nodes. So each node gets read requests in order.
- So for both read & write requests, we observed that CPU usage changes between *0.02% < X < 0.12%.*



raft-2 cbdp_raft_node
RUNNING

0.09% CPU USAGE	24.8 MB MEMORY USAGE
172 kB / 1.7 MB DISK READ/WRITE	79.2 kB / 56.2 kB NETWORK I/O

RAM 1.86 GB CPU 13.58% Disk 37.97 GB avail. of 58.37 GB Not connected to Hub

- ***Where do inserts create most load? -> leader node (writes goes from leader)***

Write request scenario (high CPU usage to lowest):

- Leader node
- Load balancer
- Follower node

For the write requests, the main job is done in the leader node; so it does make sense that the highest load is in the leader. Then if we consider all the requests are handled by the load balancer and follower nodes' gRPCs are called after some periodic updates from the leader, the order is reasonable.

Do lookups distribute the load evenly among replicas?

- Our load balancer handles the load to distribute them evenly. Each client requests Read and Write to load balancer and it handles them.
- Load balancer is aware of the system and knows how many nodes are there.
- We have created an array of RaftNodes in load balancer.
- Load balancer does not know who is the leader, it just forwards requests sequentially to the RaftNodes.
  - If you forward the write request to a follower node, it will forward this request to the leader and forward back the response to the load balancer. And finally the load balancer will forward back the response to the client.
- **Write Requests:**
  - We created an integer variable to keep the indexes of RaftNodes called *writeOrder.*
  - *writeOrder* is initialized as 0, which means that the first Write request will be forwarded to the first RaftNode.
    - If the RaftNode sends a successful response, it is forwarded to the client and writeOrder incremented.
    - If the RaftNode fails or can't be connected, again writeOrder is incremented and this time we try again until all nodes tried, and fail if we can't get response from any node.
    - Incrementation of writeOrder is as following:

```
writeOrder = writeOrder % ((int) (nodesClientService.size()));
```

- **Read Requests:**
  - We created an integer variable to keep the indexes of RaftNodes called *readOrder.*
  - *readOrder* is initialized as 0, which means that the first Read request will be forwarded to the first RaftNode.
  - However, we need a quorum of reads to assume it's a successful read.
    - At the beginning, we calculate what should be the number of successful reads to assume a read is correct as following:

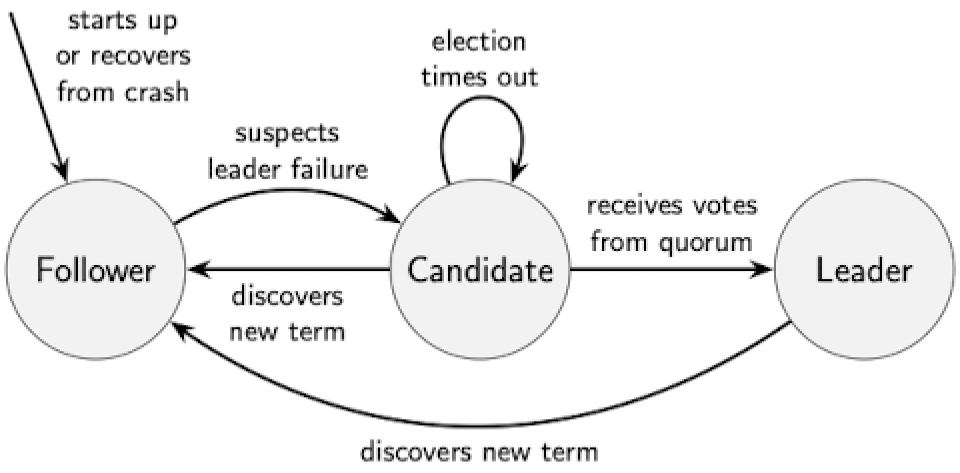
```
int readQuorum = (int)std::ceil(((int)(nodesClientService.size()))/(double)2);
//division round up to get quorum
```

    - Example: If the number of nodes is 5, we need a quorum of: (5/2) + 1 = 3.
  - As a result, we send a read request to *readQuorum* number of nodes. While sending these requests we also increment our *readOrder* integer to keep what index we are left.

        - If the *readQuorum* number of nodes return a successful read, we forward our result back to the client.
        - If the readQuorum can not obtained we return failed response to the client.

How many nodes should you use for this system? What are their roles?

- You can use any number of nodes in this system, however choosing the number of nodes can affect the performance of the system.
    - If you add too many nodes to the system, the leader will be too busy to update logs of the follower nodes. Also, poorly chosen heartbeat timeout can also cause unnecessary elections while the leader is alive. Hence, this can cause slow writes, since the leader will be busy. Also this will cause slow/wrong read results since replicating logs will take too much time.
    - If you choose 2 or 3 nodes in the system, the election will become a problem since every node will vote itself at the beginning and nobody will be chosen.
    - In our implementation we decided to choose 4 RaftNodes. It is the minimum requirement in the system for healthy elections. Also observing the behavior of the system with 4 nodes is better.
- There are 3 types of nodes in our system:
    - **Follower:** Node which communicates with the leader to update its log as the leader says. It can become a candidate if it suspects leader has failed.
    - **Candidate:** Node which suspected the leader has failed and started an election to become a leader. If it sees a node with a bigger term, it steps back and becomes a follower again.
    - **Leader:** This node is responsible for handling Write requests and update the logs of followers in the system.



Measure the latency to generate a new short URL

- ***Analyze where your system spends time during this operation***
- If the write request goes into the follower node, then follower redirects that request to the leader node.
- If the write request goes into the candidate node, then in that case we do not know who is the leader or who will be the leader so we decline this write request.
- The last case is that if the write request goes into the leader node, then below pseudo method runs:

```
Write(WriteRequest, WriteResponse) {
  if we can find longUrl in Map {
      get shortUrl from map
  } else {
     generate shortUrl
     update map
     update log vector
  }

  set response
  return response;
}
```

- Searching/updating map and updating vector operations are fast and seamless.
- Generating short url procedure is based on timestamp and sha256 hashing.
  - We concatanate current timestamp and longUrl and we create sha256 hash of that string.
  - The reason why we use timestamp is that it adds more uniqueness then using the raw hash of longUrl.
  - We can say that creating shortUrl is also runs at constant time.
- We run 4/8/16 nodes of raft clusters. We insert same 500 longUrl's in those clusters. Write times are approximately the same because the code flow is the same in all the leaders and we are not waiting from the followers. The write request times last around approximately 30ms < x < 35ms.
  - We are only waiting the followers for a single longUrl/shortUrl pair in order to commit this log into persistent volume.

Measure the lookup latency to get the URL from a short id

- If the write request goes into the follower node, then follower redirects that request to the leader node.
- If the write request goes into the candidate node, then in that case we do not know who is the leader or who will be the leader so we decline this write request.
- The raft node's state (whether follower, leader or candidate) does not effect the behaviour of our read requests. Since when we get a write request in our node, we add the longUrl/shortUrl pair into our map. So whenever we get a read request we just try to get from the map at a constant time.
- If our map does not exist such longUrl or shortUrl then we assume that this log entry does not exist in our log vector.
- We run 4/8/16 nodes of raft clusters. We insert same 500 longUrl's in those clusters. After writing process, read request times are approximately the same because we are constantly reading the shortId from a map. The read request times last around approximately 30ms < x < 35ms.
- But there is another case where the raft node does not know the shortId. So in that case our load balancer just rotates to the next node and tries to read again. This process continues for half of the nodes. After reading half of the nodes and nobody got a response for us, the load balancer assumes that the cluster does not have this log entry and returns an error to the user. Since when we get ACKs from (n/2)+1 number of nodes then we are committing that log entry so that read quorum needs to be n/2 number of nodes.

- How does your system scale?
    - Measure the latency with increased data inserted, e.g., in 10% increments of inserted short URLs
        - Since we are generating new shortUrls only in leader node, the latency does not change. But when we increase parallel write requests, then leader node becomes a bottleneck in the system.
    - Measure the system performance with more nodes
        - We tried the cluster with 4/8/16 nodes. Write request latency is tied to the leader and leader count does not change in these node setups. Read request performance can be enhanced if our system has a lot of parallel reads.