MAKING RAFT CONSENSUS A LITTLE MORE FAULT-TOLERANT

Kiran Kumar Kondru[1], Saranya R[2]*

Abstract—
Raft algorithm is a strong leader-based distributed consensus mechanism which ensures strong consistency. However it's availability aspect suffers in the presence of partial network partitions resulting in repeated leader elections and reducing the normal operations. We propose adding a new mode called Unavailable to the existing 3 modes to make the algorithm more robust and resistant to this non-trivial edge case.

Keywords—raft, distributed consensus, availability, fault tolerance

[1]Department of Computer Science Central University of Tamil Nadu Thiruvarur, India, kirankondru@ieee.org
[2*]Department of Computer Science Central University of Tamil Nadu Thiruvarur, India,
E-mail:- saranya@cutn.ac.in

***Corresponding Author: -** Saranya R
*Department of Computer Science Central University of Tamil Nadu Thiruvarur, India,
E-mail:- saranya@cutn.ac.in

DOI:- 10.48047/ecb/2023.12.si5a.0273

I. INTRODUCTION

A. Background

Raft algorithm [1] is a leader based strongly-consistent distributed consensus protocol, which has been widely adopted in the industry since its creation. It came as a more easily understood protocol, unlike its predecessor, the Paxos consensus algorithm [2]. The sheet number of citations to Raft shows it's still widely researched on. It's been implemented in many programming languages [3] and thoroughly tested even with formal languages like TLA+ [4]. The primary use of Raft algorithm is in distributed databases or datastores [5]–[9]

B. Motivation

A Raft implementation called *etcd* is widely used by many distributed systems especially by many cloud service providers. Cloudflare[10] has a massive outage[11] of some of its services on 2nd Nov 2020 which impacted many web-based services including social media and streaming services in certain geographical areas. After analysis it's found that a partial network partition caused by raft algorithm resulted in repeated elections and it's normal operations time has drastically come down. A faulty network device caused this which has a cascading effect on second and third order dependent software and services. Any unavailable service results in financial losses, so the cloud service providers always make sure to minimize the downtime.

The thing about this is the problem is pointed out by distributed systems researchers[12] but was ignored partly because it's considered a rare edge case and is trivial. Some alternative called *Prevote*[1] is suggested but not further explored. But the Cloudflare incident brought this issue of raft cluster availability to the forefront and widely discussed by distributed systems researchers. Here [13] the author argues, that with increasing popularity of Raft (due to its ease of understanding) it's adoption is growing fast and wide and it's no longer contained in the well-known and secure data center environments. It's argued that Raft might not be as available in the edge networks where the connections between nodes is intermittent. Here [14] the researchers tried to reproduce as exactly as possible the Cloudflare outage through emulation.

As such, with changing nature of the Internet from well connected duplicated hardware to unpredictable edge networks, we believe, that the core protocols should change to be more resilient.

C. Contribution

In this paper, we discuss the problem, it's existing solution and their limitations and propose our own idea to solve this problem. Our solution involves introducing a new state called *Unavailable* and use it to stop repeated leader elections and hence solve the problem.

II. RELATED WORK

A. Overview of Raft Algorithm

Raft consensus algorithm follows Replicated State Machine to store the clients' instructions (*commands*) in sequence and propagate accordingly. Raft has a strong leader who accepts all the clients' queries and linearizes the *commands* and persists them in the form of sequence of *entries* called a *Log* to the secondary storage. It then asks its *followers* to do the same. With a majority of the *followers* committing the *log*, the leader makes those *entries* permanent and conveys the same to the specific clients.

When a *follower* fails or stops, it doesn't effect the cluster, as long as the failed nodes are not many. Given 'n' is the size of the cluster, and 'f' is the number of faulty nodes that the cluster can tolerate and still operate unhinged, the formula is as follows.

$$n = (2 * f) + 1$$

A raft node starts up and becomes a *follower* first. After some random timer sets off, one of the nodes increments its term number (from initial '1') and starts asking votes from fellow *followers*. This node is said to be in *candidate* mode.

- If this *candidate* receives enough votes, he becomes a *leader*.
- If he receives another *leaders heartbeat* with higher term number, it then becomes a *follower*
- If he doesn't get enough votes and still no *heartbeat* from the leader, then it restarts the election process again by incrementing the term number again.

The following diagram illustrates this state change mechanism that is core to the raft protocol.

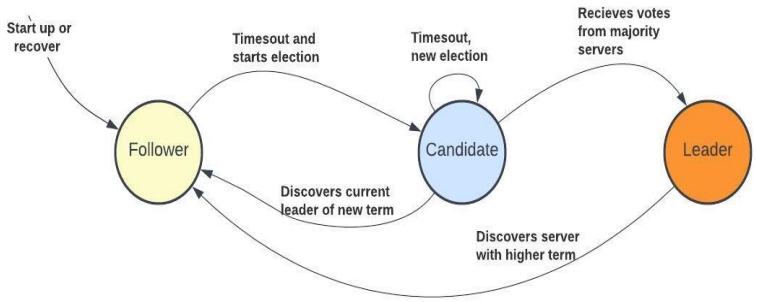

Figure 1: Modes of Raft nodes **[1]**

B. Scenario of a Partial Network Partition

The problem of repeated elections and unavailability of the whole cluster from servicing the clients has been explained here [14]. This article gives us a very good understanding of scenarios that give raise to repeated elections where partial network partitions happen. The following example illustrates this better. In the left part of Figure 2, node A is completely isolated from the rest of the 3 node network. A can be thought of as node failure in this scenario as there is no way for it to send or receive messages from any of the nodes in the cluster. But the right-side figure points to a partial network partition and this produces some interesting results.

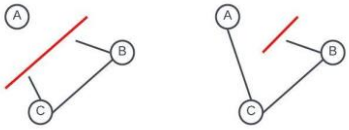

Figure 2: Full and Partial network partition **[14]**

Consider 3 raft nodes A, B and C and let B be the *leader*. The following is a scenario that happens in the case of a partial network partition. We illustrate this example in the following steps with the help of figure 3.

- B sends *heartbeat* messages to both A and C in the form of *AppendEntries* RPC.
- Both A and C respond via *Append Entries Response* to B. All is well here.
- Between A and B a partial network partition is formed (as indicated in the right-side of Figure 2), while A and C are still connected.
- B sends *heartbeat* again to both A and C. A couldn't receive message.
- After a while, since A didn't receive any RPC, it times out, increments the term and starts an election on its own.
- A sends *RequestVote* to both B and C. B couldn't get the message due to partition, but C receives it.
- C updates the term number to the new term sent by the node A but sees that its log index is not up to date. Hence it replies *No* to A's *RequestVote*.

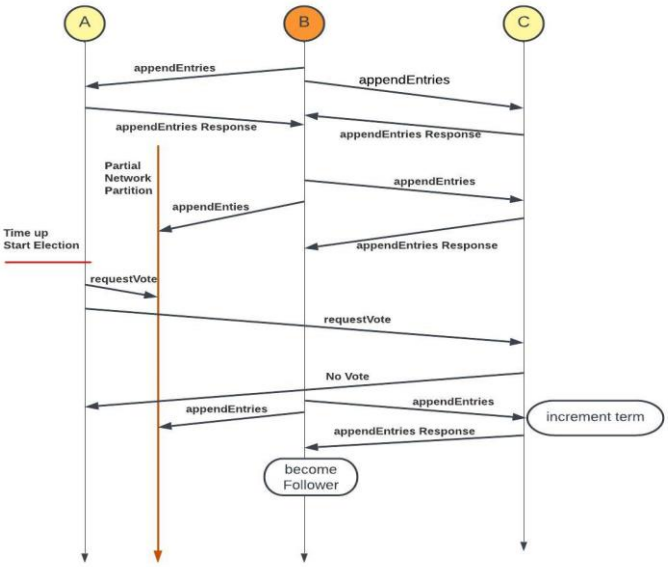

Figure 3: Scenario of Partial network partition [14]

- But since, B is still sending *AppendEntries* RPC as usual, C receives it but in its *Append Entries Response*, it notifies the *leader* B of the higher term.
- With this response, B sees there is a higher term that itself and immediately vacates the leadership position and becomes a *follower*.
- A can try to become leader but only C can receive it's message and always sends *No* as it's log index is not up to date. So, in this scenario, A can never become leader as it can't get majority votes.
- Either C or B can become a candidate and start an election. But if B becomes the *leader* again, the whole cycle will start again, beginning an ever-repeating leader election.

C. Existing Solutions

The original Raft paper proposed the concept of *Prevote*[15] to stop such repeated leader election from happening, but it's not delved into further. *Prevote* suggests to have an extra round of election to verify whether a node is eligible to be elected as a *leader*. If that specific node is isolated like it's been completely cut-offs from the cluster network, then a *Prevote* round would not return any votes and hence that node refrains from contesting an election. However, this is not as simple as it sounds. The blogpost[16] by distributed systems researchers found that *PreVote* itself won't solve this problem and it will further introduce more bugs.

III.PROPOSED SOLUTION

To the problem of partial network failure, where in the there is a scenario in which there is repeated leader election, we propose to increase the number of modes from three to four. We will add a new state called *Unavailable*. The following figure shows the new state transition diagram with the new mode *Unavailable* and the details of when the raft server will enter this mode and come out it.

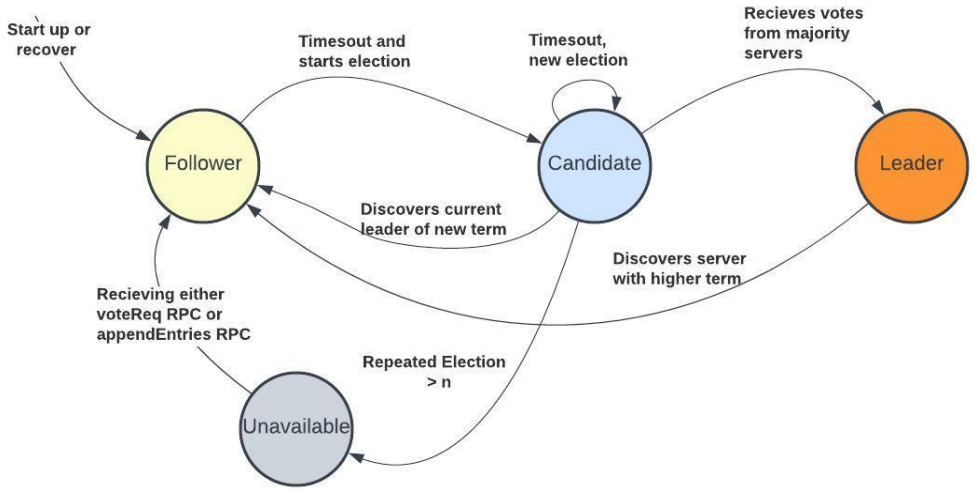

Fig 4: New State Transition Diagram

In the previous section, change of state is described with how and when there will be a change in Raft modes. If Raft algorithm is to have a wider adoption outside the context of data centers where there is a significant duplication of network hardware, preventing almost any kind of link failure, it has to address the so called non-trivial availability issues already discussed above. One way of ensuring this is to make Raft more available by incorporating some minimal changes like introducing a new state and defining when and how the node goes into and comes out this state.

The problem of repeated election is caused when there is a failure in the link or network connection, whether partial or full severance. When a node is unable to receive any RPC from the leader, it becomes a candidate and starts to seek votes from other *followers.* But that specific node is unable to receive any messages especially periodic *heartbeats*. Even after becoming a *candidate*, it neither gets a *heartbeat* from the current leader nor gets the requisite majority votes from available peers. Given this specific knowledge, it can be inferred that when repeated elections are initiated by a specific node, after a certain threshold, we make it stop the election and go into an *Unavailable* mode, preventing repeating elections. This threshold value can be 3 or 5 depending on the requirements.

A. A. Implementation

Algorithm: Proposed changes to the Candidate Routine

All Servers:
prevCandTerm: = 0 and electionCount: = 0.
If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine.
If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower
Candidates:
If (currentTerm – prevCandTerm) => 1 then
electionCount: = electionCount + 1
else
electionCount: = 0
If electionCount >= 'n' repeated attempts, then
Stop resetting election timer.
Set currentRole as 'UNAVAILABLE'.
Set prevCandTerm to currentTerm.
Return
On conversion to **candidate**, start election:
Increment currentTerm
Vote for self
Reset election timer.
Send RequestVote RPCs to all other servers.
If votes are received from the majority of servers: become leader.
If AppendEntries RPC received from new leader: convert to follower.
If election timeout elapses: start new election.

Unavailable:
If RPC received from the Leader, become a CANDIDATE.
And Follow the Candidate routine for RPC.

In the "Rules for Server" section of the original Raft paper, we have made the following changes as illustrated in the above algorithm. We introduced two new variables "***prevCandTerm***" and "***electionCount***" to keep track of how many repeated elections are happening in a single raft server. Also as discussed in the previous section, we introduce a threshold variable 'n' and set the value to our comfort level like 3/5/7 etc.

We keep this threshold value so as not to introduce more bugs in the system. It's normal for an election process to result in a stalemate and have no elected leader for that particular term. The only prevention mechanism in the leader election algorithm is the random timeout of the election timer which forces raft servers to not start the election process at the same time. But random timeout does not guarantee two servers waking up at the same time in the absence of a leader and starting the election process. This in turn results in split votes with two servers receiving minority votes and no definite leader. When this happens, the random timeout occurs again in the candidate phase of the servers, and another election is randomly started with a new term number. So the possibility of repeated election is there and more than one repeated election can happen.

In the above-described algorithm, if we set the threshold 'n' to 5, then this particular server keeps on the process of repeating the election for 5 times, and then it stops. This self- identifying nature of the changed algorithm is what is unique. After this threshold is reached, the election timer is stopped, the current role is set to "UNAVAILABLE" the control is returned. This makes the server inert and since election timer is disabled, there is no chance of starting an election by itself. As there is no chance of starting an election, the problem of repeated election is solved.

IV.RESULTS AND DISCCUSSION

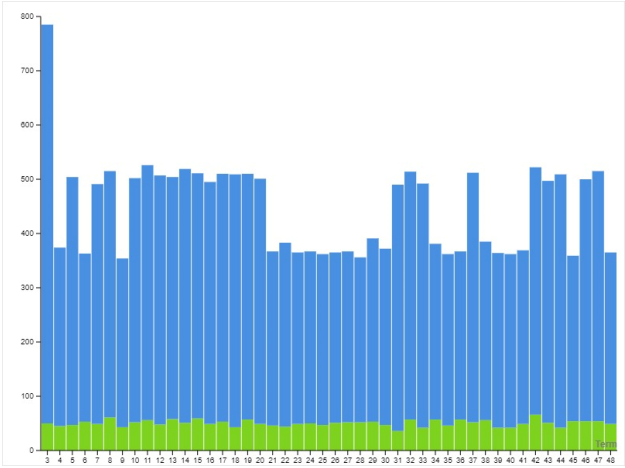

Figure 5: Repeated Leader Election and Normal Operations

We developed a small Discrete Event Simulator in Java, to test out our proposed solution. But to actually show that our solution works, we first designed a raft cluster with partial network partition as shown in the right-side diagram of figure 2. And then ran a simulation. The above graph is the result of the simulation run. The X-axis is the term number, and the Y-axis is the simulated time. We can equate the simulated time to nanoseconds. The blue area of the graph shows normal operations for a particular term whereas the green area represents the election process. During the election process, client requests are not entertained.

As can be inferred from figure 5, though there is normal operations happening for some simulated time, a new election with a new term number is happening. And this is repeating indefinitely (we only ran the simulation for a certain time). With the partial partition intact, this never stops, and this makes the whole cluster unavailable for an extended period of time and normal operations time is restricted. This not a desirable situation.

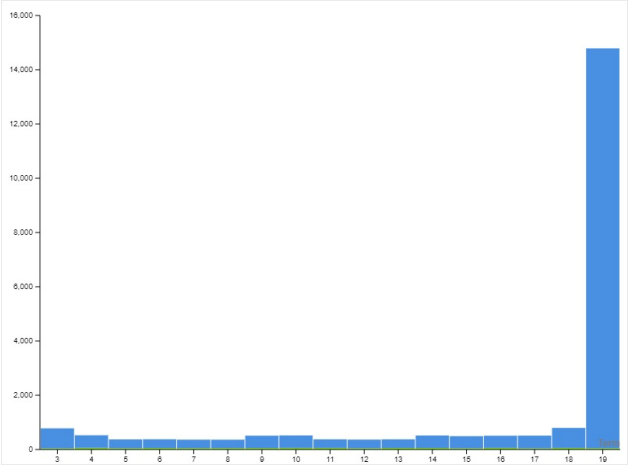

Figure 6: Stopping of repeated election and return of normal operations.

After we introduced the proposed solution, we do have repeated elections, but only up to a limited threshold value we set. The above figure 6, shows the result of a simulation run with the proposed changes to the raft algorithm. The X-axis shows the term number, and the threshold 'n' is set to 9. This configuration tolerates the flip-flop of the leader up to 9*2 = 18 terms. After that the repeated elections stop and we can see the normal operations time has increased significantly in term 19. This indicates the return to a stable normal operations state of the whole raft cluster. Even with other faults like ordinary leader failure, this change won't affect the normal election process. We made sure, there is only limited changes to the algorithm, so as not to introduce any more bugs, as designing any distributed systems normally leads to.

A. Secondary Advantage

Another advantage of having a node in the *unavailable* state is that the client might know about it from interacting with that specific node.

With this knowledge and the clients already having all the IPs of the cluster members, they can simply redirect their query to any of the other members.

V. CONCLUSION

We have proposed adding a new mode to the Raft's existing 3 called *Unavailable* to prevent non-trivial scenario of repeated elections in the presence of partial network partitions. We have explained the scenarios in which repeated elections happen and how we could prevent it with the introduction of this new mode/role. We made sure there are no subsequent bugs introduced because of our change to the algorithm.

We also simulated the original problem with a Discrete Event Simulator and showed the resultant repeated election. We have also shown the result of changes to the raft algorithm and how it stops the whole cluster from going into an indefinite election phase disrupting normal operations.

Since a distributed consensus algorithm like Raft is works in the core part of a distribud system, the effects of even a small improvement, will have second or third order effects in the software that use this. The above mentioned Cloudflare incident is the prime example of this cacading effect.

REFERENCES

1. D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," p. 18.
2. Digital Equipment Corporation and L. Lamport, "The part-time parliament," in Concurrency: the Works of Leslie Lamport, D. Malkhi, Ed., Association for Computing Machinery, 2019. doi: 10.1145/3335772.3335939.
3. "Raft Implementations. "https://raft.github. io/#implementations
4. "TLA+." https://lamport.azurewebsites.net/tla/tla.html
5. "etcd," etcd. https://etcd.io
6. "CoackroachDB." https://www.cockroachlabs.com
7. "Hazelcast." https://github.com/hazelcast/hazelcast
8. "Rethink DB." https://rethinkdb.com
9. "Atomix." https://atomix.io
10. "Cloudflare." https://www.cloudflare.com/
11. "Cloudflare etcd raft outage," Cloudflare etcd raft outage. https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/
12. "Twitter - Raft partial network failure."
13. H. Howard and J. Crowcroft, "Coracle: Evaluating Consensus at the Internet Edge," in Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, London United Kingdom: ACM, Aug. 2015, pp. 85–86. doi: 10.1145/2785956. 27900 10.
14. C. Jensen, H. Howard, and R. Mortier, "Examining Raft's behaviour during partial network failures," in Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems, Online United Kingdom: ACM, Apr. 2021, pp. 11–17. doi: 10.1145/3447851.3458739.
15. "In Search of an Understandable Consensus Algorithm (Extended Version)", [Online]. Available: https://pages.cs.wisc.edu/~remzi/Classes/739/Spring2004/Papers/raft.pdf
16. I. A. Heidi Howard, "Raft Does not guarentee Liveness in the face of Network Faults," Raft does not Guarantee Liveness in the face of Network Faults, Dec. 12, 2020. https://decentralizedthoughts.github.io/2020-12-12-raft-liveness-full-omission/