International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
IJISAE
ISSN:2147-6799 www.ijisae.org Original Research Paper

Exploring the Limits of Raft's Fault Tolerance: Insights from Simulated Network Partitions

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

Submitted: 03/05/2024 **Revised**: 16/06/2024 **Accepted**: 23/06/2024

Abstract: This paper delves into the robustness of the Raft consensus algorithm, particularly focusing on its fault tolerance capabilities and the challenges it faces under network partitions and node failures. This study provides a comprehensive analysis of Raft's mechanisms to ensure data consistency across distributed databases. Through detailed UML diagrams followed by simulations, this work effectively illustrates the leader election algorithmic processes and fault tolerance operations within the Raft. This paper focuses on edge-case failure scenarios and illustrates them with sequence diagrams and complimented by graphs of results from the discrete event simulation of Raft's leader election. Using a custom-built Discrete Event Simulator, we explored the space of Raft's lesser-known failure cases, thus complementing previous studies on consensus mechanisms**.** This study pushes the limits of Raft's liveness and provides a broader picture for better understandability.

Keywords: *Raft Consensus; Fault Tolerance; Aliveness; Discrete Event Simulation;*

I. INTRODUCTION

Modern databases are designed to be distributed in nature from the get-go, as the liveness of the system depends on the availability of the database to service. This distributed nature ensures that even if one of the database servers is out of service, the other backup servers will pick up and process requests for applications. Ensuring that all the backup servers have identical data is the responsibility of the consensus mechanism employed by these distributed databases. These consensus modules are usually chosen from the existing list of consensus protocols, such as Paxos[1] and Raft[2], which are battle-tested and proven mathematically for correctness. They are usually state machine replication protocols that replicate a monotonically increasing log of commands from the clients to all backup servers or replicas. These replicas execute commands in the same sequence from the log, thus achieving the same state as the entire data store. The state machine, after this sequential execution of each of the nodes in the cluster, will have identical information and thus form a single source of truth for the entire application.

These consensus protocols perform some of the core functions of a distributed database. To show the importance of these protocols for databases, we list some databases here and the type of consensus mechanisms they employ (Fig 1: Distributed Databases that use Raft and Paxos consensus mechanisms.).

[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, saranya@acad.cutn.ac.in*
*** ***Corresponding Author****: Kiran Kumar Kondru*
**Department of Computer Science, Central University of Tamil Nadu, Thiruvarur, India*
kirankondru@ieee.org

Databases that use the Raft consensus

- *etcd[3]-a key value* data store that uses Raft to manage its highly available replicated log.
- *CockroachDB*[4]–uses Raft in its replication layer to maintain data consistency, even when machines fail. Each data range has its own raft group.
- *TiDB [5]* - uses Raft with its key value storage engine, TiKV, to ensure data replication and consistency.
- *YugabyteDB* uses Raft for DocDB replication.
- *MongoDB*[6]–uses a variant of the Raft protocol in its replication sets to ensure data consistency and automatic failure.
- *RabbitMQ [7] uses* Raft to implement replicated durable FIFO queues.
- *Neo4j*[8] is a graph database that uses Raft to ensure consistency and safety.
- *Influx DB*[9] - uses Raft for high availability of its metadata nodes.
- *Splunk*[10] used Raft in its Search Head Cluster (SHC) for consensus.
- *Redpanda [11]* uses Raft for data replication.

The core functionality of the raft consensus in these databases is to ensure that all data remain consistent across the cluster. This is crucial to maintaining the integrity of the database, particularly in the event of network partitions or node failures. Therefore, it is imperative that we understand the fault-tolerant features of Raft and how they contribute to the robustness of the system.

One of the key fault-tolerant features of Raft is the concept of log matching, which ensures that if two logs contain an

entry with the same index and term, then the logs are identical in all entries through the index. This property is crucial for ensuring consistency across clusters after a leader changes. Another important aspect is the commitment to log entries. A log entry is committed when it is replicated safely on most nodes.

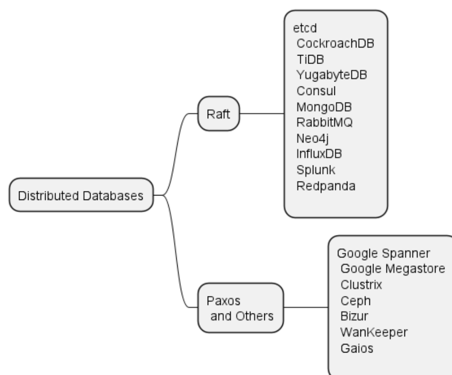

Fig 1*:* Distributed Databases that use Raft and Paxos consensus mechanisms.

Only committed entries apply to the state machine; therefore, even if a leader crashes after committing a log entry, the system can recover and maintain consistency. The new leader will pick up where the old leader left off, ensuring that no committed changes are lost. The heartbeat mechanism in Raft also plays a significant role in fault tolerance. These periodic messages are sent by the leader to all followers to maintain authority and prevent new elections. If followers stop receiving heartbeats, they will initiate a new election, potentially electing a new leader to continue their operations. This helps the system to self-heal in the face of network problems or leader failures, minimising downtime and maintaining availability.

Using Raft, these databases can provide strong consistency guarantees, which are essential for reliable and accurate applications that require transactions.

As such, in this paper, we examine how fault tolerance works in raft consensus and how the cluster is not significantly affected, even when there are failures of the servers or network partitions. We illustrate this through a series of UML sequence diagrams explaining the algorithmic aspects of Raft and various scenarios that lead to failure situations. We also show how Raft cannot handle certain corner cases, and we provide alternative solutions to fix this. We simulated Raft consensus using a discrete event simulator written in Java. Furthermore, we present and discuss the results.

II. BACKGROUND

A. Raft Consensus Algorithm

The Raft distributed consensus algorithm, which serves as the foundation for numerous distributed systems, including several distributed databases, is a single-leader state machine replication algorithm. Instead, consensus was achieved by managing a replicated log across the raft cluster. When the cluster starts with multiple nodes, one of them times out and becomes a candidate to become a leader by seeking votes from other nodes, such as followers. These follower nodes vote for a candidate with a higher term number and vote on a first-come-first-serve basis. A term is a monotonically increasing number that uniquely represents the leader's reign. When a follower times out and becomes a candidate, it first increases its term number. After the candidate receives the majority vote, it becomes the leader and sends a message (RPC) called a heartbeat message. This message informs followers of who the leader is and the term in which it operates. This process is depicted in the election.

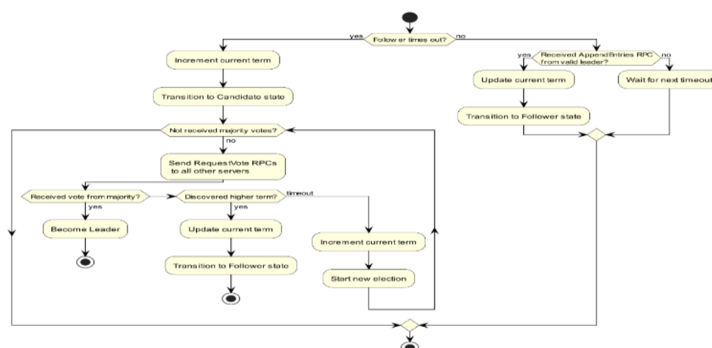

Fig 2: Activity Diagram of Raft Leader Election

A Raft cluster operates as a single entity, although it may have many servers. An example is a simple key-value data store. When a client wants to store or retrieve a value for a key, it communicates through the leader, although the client is aware of all IP addresses of the cluster members. The leader accepts a client's 'set' request of a (key, value) pair. After receiving the command, the leader places this in a log of every increasing size. This process helps linearize the various commands from various clients. Once a client's command is written to the log (disk), this log, with its latest entries, is replicated exactly across the cluster with the followers. These followers acknowledge the latest entries of the replicated log and apply the command in the entries to their own state machine (data store). After receiving the majority of the acknowledgements from the followers, the leader also applies the command in the entry to its own data store and sends a success request to the respective client. This process is illustrated in Figure 3.

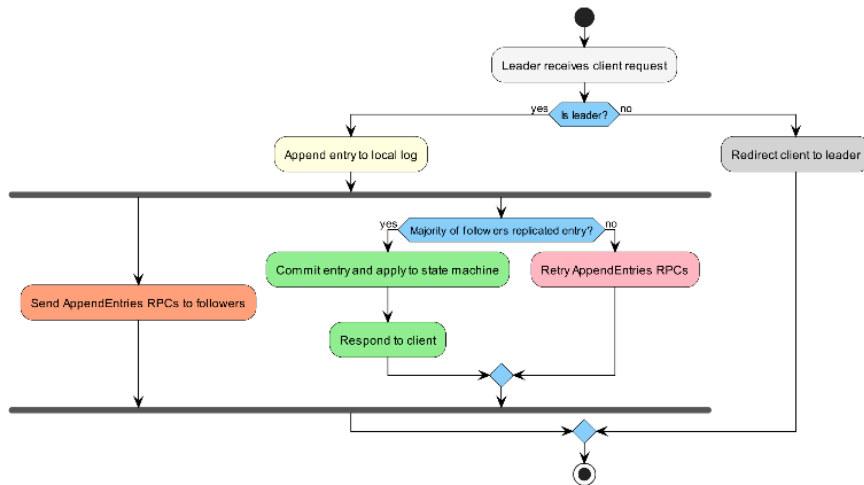

Fig 3: Activity Diagram for Log Replication in Raft Consensus

B. Attempts to improve Raft's fault tolerance

Several enhancements to Raft fault tolerance have been proposed in the literature. For example, introducing a supervisor node and secret sharing can make the Raft Byzantine fault tolerant without requiring invisible trust between the nodes[12]. Combining Raft with a reputation mechanism can help identify and isolate malicious nodes [13]. Hierarchical Byzantine Fault Tolerance (HBFT) has also been integrated with Raft to improve its performance and fault tolerance in large-scale systems [14].

III. RAFT'S FAULT TOLERANCE

The Raft consensus algorithm has many built-in features that make it live even when there are non-functioning servers or network link partitions. In the following cases, we discuss how Raft effectively handles these failure-inducing scenarios and how it recovers from them.

A. Split Vote

Raft is designed specifically to be tolerant of network node failures and network partitions. For a leader election, one of the followers must time out and become a candidate. However, each node randomly assigns a number to wait. This random timeout in milliseconds was set between 150 and 300 ms, according to the original Raft paper. This is set by considering the network delay that naturally occurs. However, there is a chance that two followers might have the same or nearby timeouts, and they might wake up parallel and seek votes, giving rise to a split vote situation.

Figure 4 illustrates this scenario, in which two servers wake up and try to seek votes simultaneously. An equal number of followers might vote for the two competing candidates. This results in two candidates getting equal votes, but no majority. No leader in this term. However, any of the cluster servers may timeout and start a new election with a new term number (term no. 3). Because there is no other candidate with this high term number, all candidates and follower nodes vote for this new candidate, and it subsequently becomes a leader.

B. Fault Tolerance in Log Replication

Raft follows a strict leader-follower architecture, where the leader has all the power, and the followers replicate the leader's commands. Only with the failure of a leader do followers start the election process by giving a vote. However, even the way the log is structured and replicated allows a follower to pick up the commands from the leader and populate its state machine quickly.

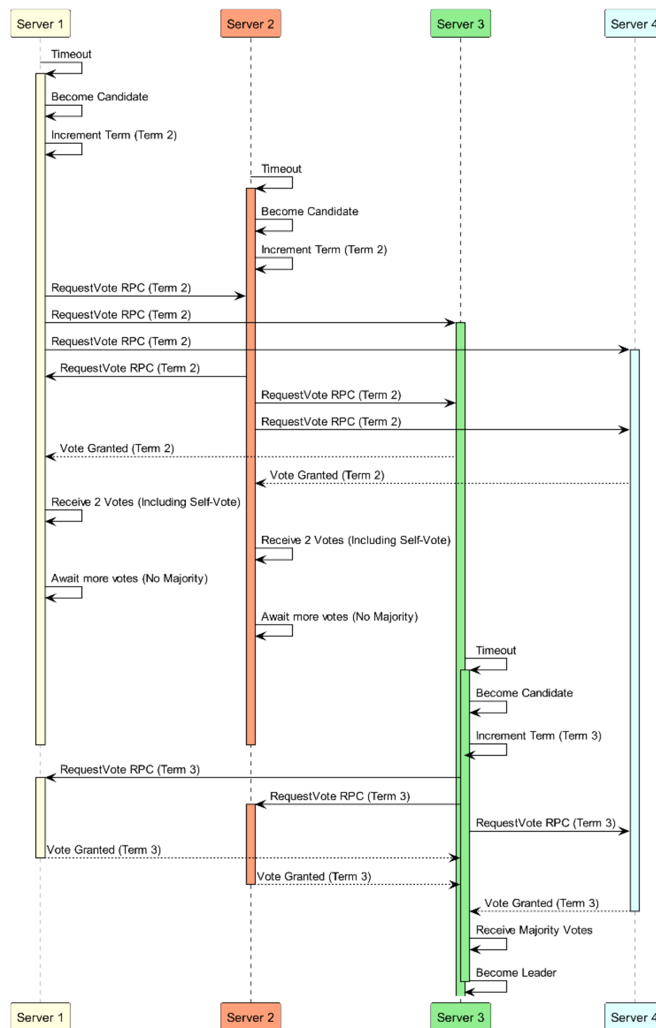

Fig 4: Scenario where during Leader Election, split vote occurs, and Raft recovers from the split vote with another round of voting.

In Fig 5: Even when the leader fails, because the log is up to date, any follower can contest and win to become a if the leader fails, and committed entries are present in the logs of the majority of followers. Even if a follower becomes a candidate and then a leader, the cluster remains operational while maintaining consistency and integrity. This is possible because of the proper replication of the log. Even if a follower goes offline and returns, it seeks the most recent entries in the log. It will inform the leader of the status of its log and, accordingly, the leader will populate the followers' log until it is consistent with the leader's log. Crashes and network faults can be overcome using this fault-tolerant design.

IV. RAFT'S FAILURE SCENARIOS

After the text editing was completed, the paper was ready for the template. Duplicate the template file using the Save As command and use the naming convention prescribed by the conference for the name of the paper. In this newly created file, highlight all the content and import the prepared text file. You are now ready to style your paper; use the scroll-down window on the left of the MS Word Formatting toolbar.

A. Network partitions

There are situations where the Raft cluster is not alive for a significant portion of the time, and this issue is predicted but is considered a corner case and ignored for a long time. However, the Cloudflare[15], [16] incident brought this issue back to the forefront. The article [17] discussed this and reproduced the problem through emulation. These partial network partitions, though covered theoretically, are not given much importance because of the rarity of such real-world scenarios occurring in the real world.

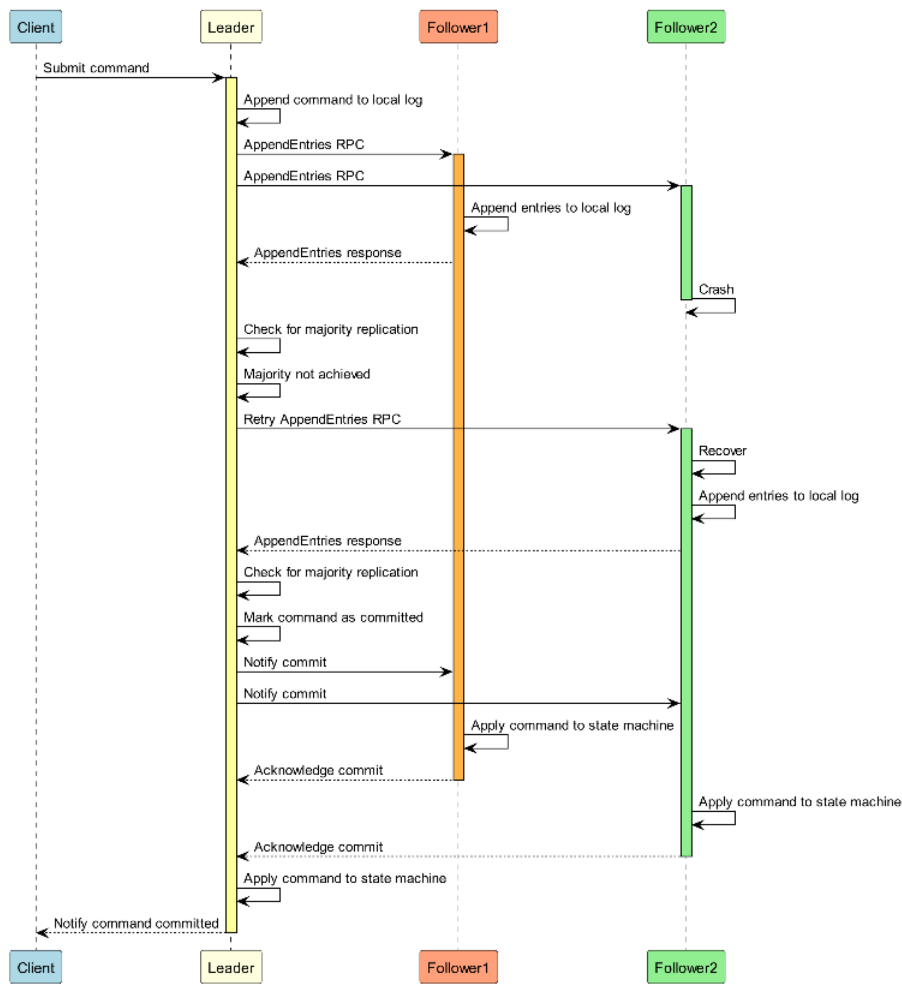

Fig 5: Even when the leader fails, because the log is up to date, any follower can contest and win to become a leader.

Here, we will explain a scenario where there is a partial network failure with a three-node cluster using the sequence diagram in Fig 6: Non-Transitive Rechability - where repeated elections happen because of network partition between two nodes. We assume that the leader election has already occurred and that a leader exists. When the leader receives a request from a client, it adds to its log the latest command as an entity. This log is replicated throughout the cluster using the AppendEntry RPC command. Obtaining a majority of AppendEntriesResponse would suffice, but generally all followers respond positively. Because there are three nodes in the cluster, the leader only needs one more vote from Follower1 apart from his own self-vote to commit the entry to its state machine and respond positively to the client. If there are any remaining followers who are yet to respond, the leader retries the AppendEntries command for all missing log entries of such followers.

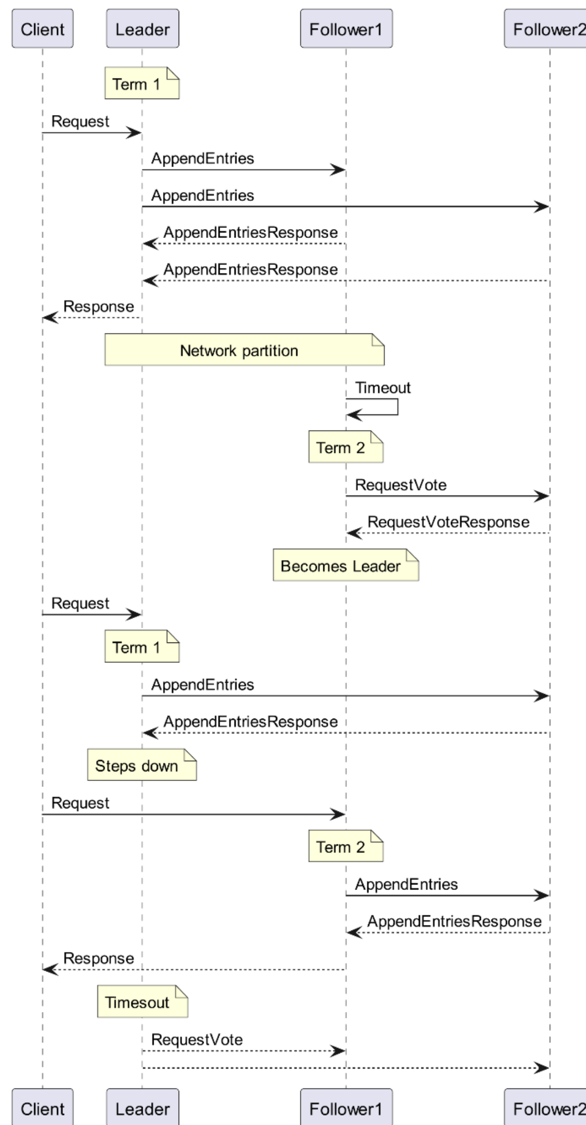

Fig 6: Non-Transitive Rechability - where repeated elections happen because of network partition between two nodes

However, in the scenario described in Fig 6: Non-Transitive Rechability - where repeated elections happen because of network partition between two nodes, we see that a full network partition occurs between the leader and follower1. After a while, follower1 does not receive any heartbeats from the leader and assumes that the leader is not present. Follower1 times out and starts the election process by incrementing the term to two and seeking votes from the other followers. However, since there is no link between Follower1 and the leader, RequestVote does not reach the leader but reaches Follower2. Follower2, seeing the greater term number and an UpToDate log of Follower1 accepts its request and votes for Follower2 to become the new leader of term 2.

When the old leader receives a client request, it attempts *AppendEntries*, which only reaches Follower2. However, Follower2 does not accept the *AppendEnries,* but sends back a 'no' answer with the new term number and the id of the new leader (which is Follower1). This caused the old leader to descend from its leadership position. However, after this, it does not receive any heartbeats from the new leader (aka Follower1), and it breaks and starts a new election with term 3. This process of repeated elections continues to occur until the partition is healed. This removes the operational time of the Raft cluster, as during the leader election, the entire cluster becomes non-responsive to the clients.

B. Partial Network Partition

Another type of network partition is possible, in which only partial communication occurs between a node and the rest of the Raft cluster. This may be due to a faulty or misconfigured network device. We illustrate this problem using a sequence diagram, as shown in the device. In a four-node cluster, when a client requests a set operation, leader node 1 responds with an *AppendEntries* RPC to nodes 2,3, and 4. However, node 4 has a faulty network device between it, which prohibits receiving communication from any of the other nodes, including the leader. This, after a while, makes node 4 time out and starts the election process by incrementing the term number to two and seeking votes

from others. This RequestVote from node 4 goes to the other nodes because simplex communication is still possible. Because this is a higher term and the log is up to date, all nodes oblige and send a positive AppendEntriesResponse back to node 4. The leader (node 1) also steps down, seeing a larger number of terms. However, none of these responses is received by the new candidate node 4.

While node 4 keeps waiting for AppendEntriesResponse, one node (node 3) is reactivated and starts its own election process by increasing its term to 3 and seeking votes. The nodes that can receive the communication respond positively to *VoteRequest* as this is a higher term, but node 4 cannot receive it yet. Node 3 obtains the maximum number of votes from the cluster (nodes 1 and 2, including self-vote) and starts issuing AppendEntries to its followers.

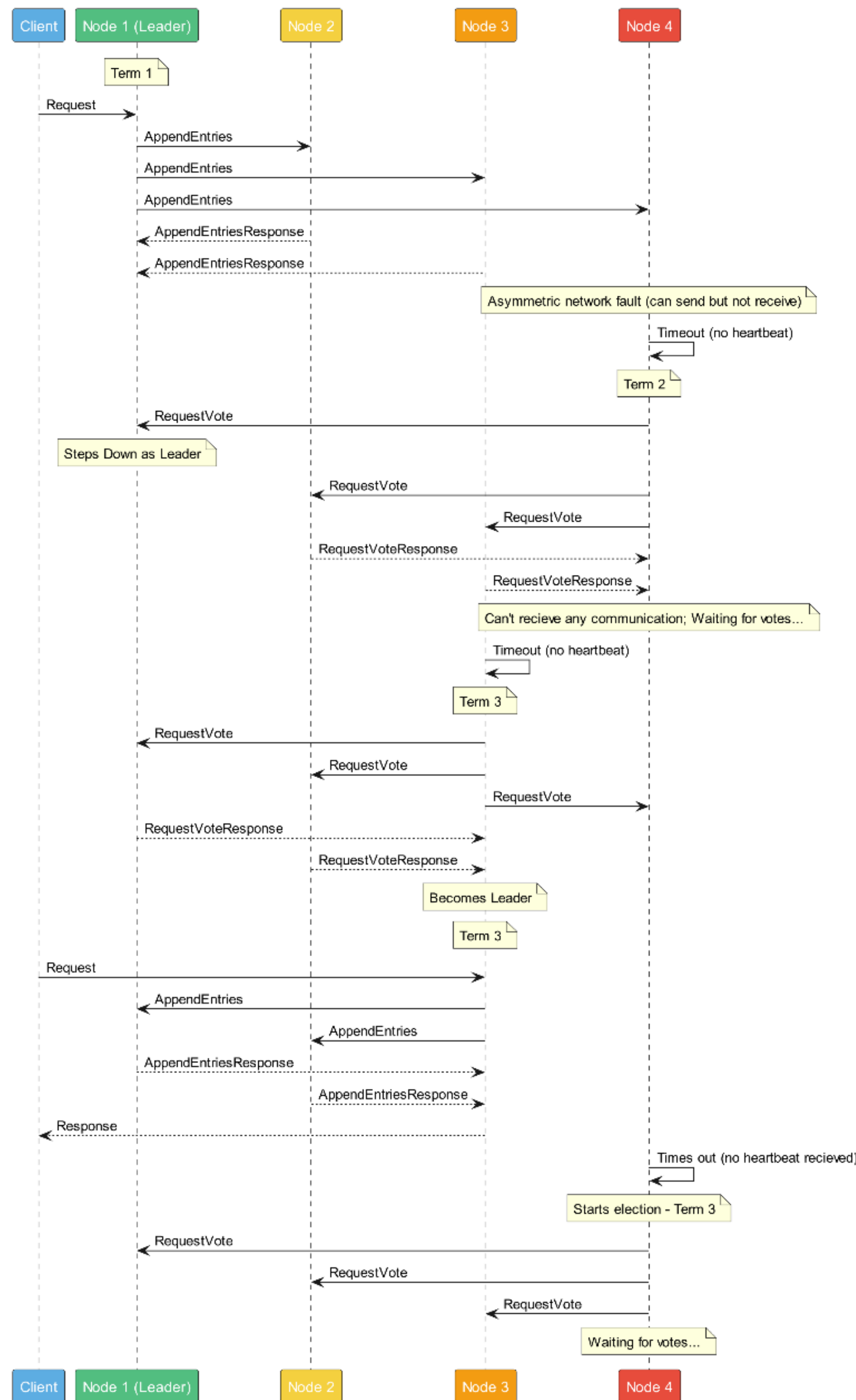

Fig 7: Asymmetric rechability - repeated leader election caused by a node isolated by a partially communicating network device.

Candidate node 4 has received no votes yet and times out, and starts another election with an increased term number, term 3. This is received by the other nodes in cluster (1,2 and 3) but because there is already a leader with term 3 and

with an updated log, all the nodes reject this VoteRequest. However, this RPC also did not reach the candidate (Node 4). After a wait period, it times out and starts with an increased term, Term 4. With this new term number, VoteRequest may be accepted by the other nodes (1,2, and 3) if no client request comes in between, and their logs are identical. If the logs are identical, nodes 1,2 and 3 grant votes to candidate 4. However, this also does not reach node 4, and this cycle is repeated until the faulty network device is fixed.

V. SIMULATING CONSENSUS

To simulate the Raft-distributed consensus to test for a leader election, we created a Discrete Event Simulator coded in Java. We chose to test the leadership algorithm through DES, as it can accurately model the sequence of events in a distributed system. We can simulate node failures, link failures, packet loss, etc. with precision. The simulator allowed us to customize the configuration of the network conditions, track the state of each node, and measure the time to reach a consensus. Through discrete event simulation, we can gain insight into the robustness of the Raft protocol, especially its leadership election process. The simulation also recorded statistics for analysis, highlighting potential areas for optimization and improvement.

We implemented our own Raft algorithm in Java and subjected it to a simulator to verify that the algorithm adheres to the original specification.

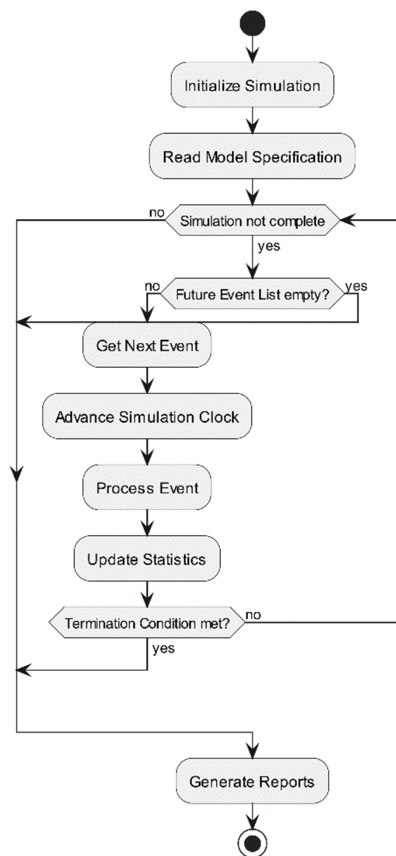

Fig 8: Activity Diagram for a Discrete Event Simulator

However, a discrete event simulator differs from a normal network simulator. DES processes events in chronological order. It has its own simulated time, which is a nondecreasing number representing real-world time. The components of the simulation, such as nodes and links, are abstracted as entities, and the communication messages between these nodes through hardware links are abstracted as events. The internal timers were also events. A single thread (event queue) handles all of these events. This event queue is a priority queue that orders the events based on their scheduled time. Event handlers define how a system should respond to each type of event. Finally, the system state encompasses all the variables that represent the status of the nodes in the network. As with any simulation, we abstract only the necessary parameters relevant to the problem we are solving. Figure 8 shows an activity diagram to illustrate the general flow of control in a DES.

When a node wants to send a message to another node, it sends it as an event with the sender's address, message, and time that it should be scheduled. The event queue registers (accepts) the event and places it in the queue. When the

simulation started, each event was popped and executed. Figure 9 presents a UML state diagram to better understand the internal structure of the DES.

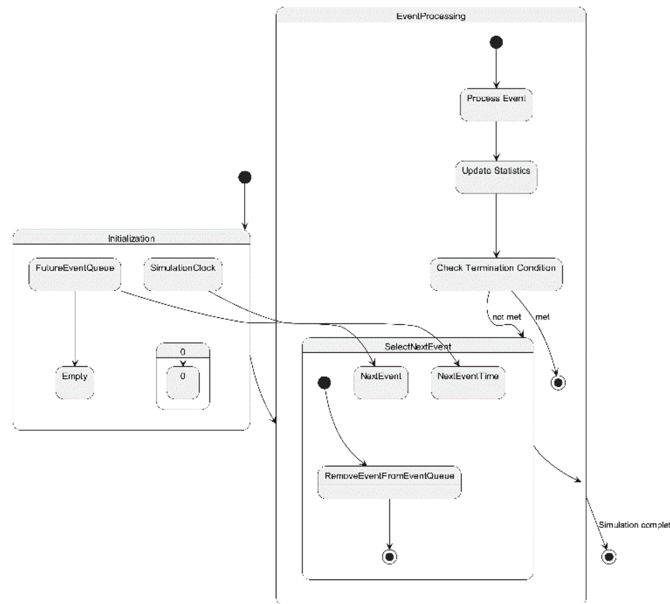

Fig 9: State Diagram for a Discrete Event Simulator

This execution results in the calling of the appropriate event handler, depending on the type of event. In this case, a message is sent from one node to another. The receiving node receives the event through the event handler and starts processing. It updates the system state of the node. The receiving node can respond via another message event by registering the event in the event queue. This process was continued. This generation of new events by responding to previous events makes the simulation run indefinitely, given that events are continuously generated. Termination can be induced programmatically, either through the number of rounds or by stopping after a fixed time. This is a simple scenario of discrete event simulation occurring in the context of distributed systems. This simulation provides the complete behaviour of a system. The following sequence diagram illustrates the complex interactions of the processes in a DES:

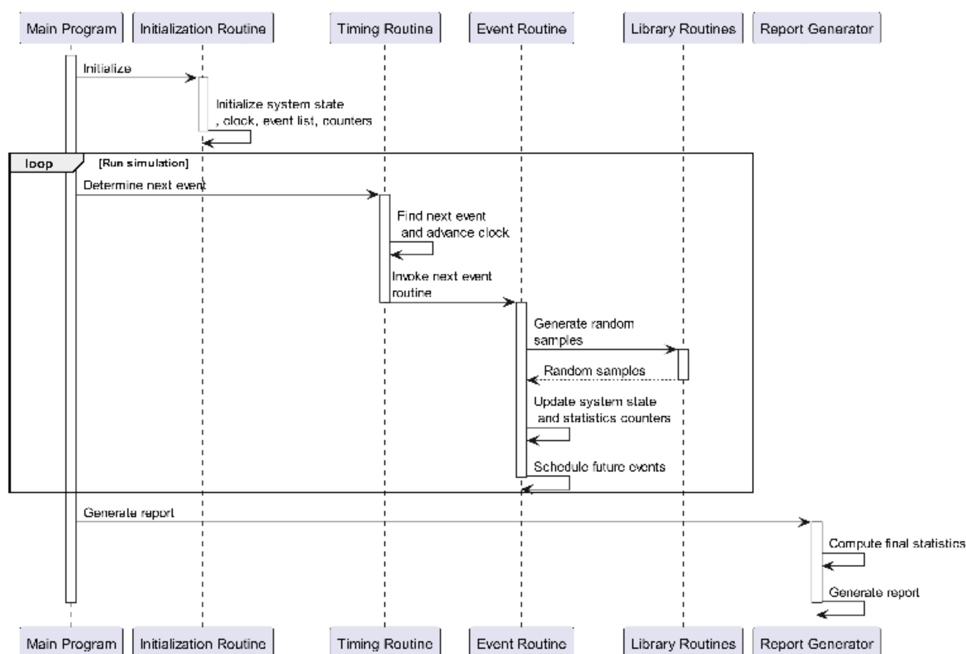

Fig 10: Sequence Diagram of Discrete Event Simulator

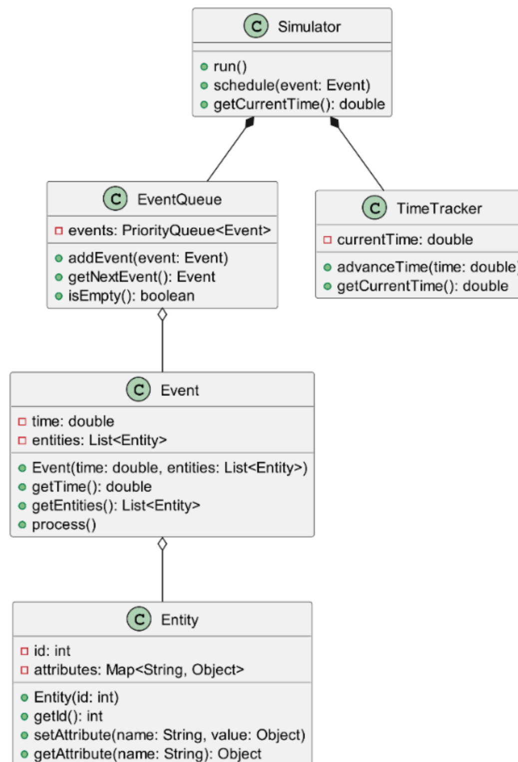

Figure 11: Class diagram of the discrete event simulator.

We also provide a class diagram in Figure 11 for the discrete event simulator that we developed for a better understanding of our architecture.

VI. EXPERIMENT WITH RAFT LEADER ELECTION

We tested our Raft leader election algorithm with the discrete event simulator that we developed, and the results are as follows. The work by Dr. Heidi Howard in her thesis "ARC: Analysis of Raft consensus" tested many parameters regarding leader election. The work includes testing with different parameters for the random timeout for the leader election, from 150ms to 300ms with varying intervals. In this section, we test the changes in different parameters with varying latencies. We tested those that were not covered in the above study, such as increasing the node count from 5 to 25 and changing the latencies.

Initially, we tested our Raft consensus in DES without any leader failure. Subsequently, we induced a link failure link attached to the leader to a follower. Because Raft assumes a one-to-one direct connection with each of the participants of the cluster, failure of one link causes a leader election initiated by the follower on the other side of the link failure. This scenario is illustrated in Figure 6 in Section IV. A (nontransitive repeatability).

The median latency for a normal raft cluster was 30 ms. We tested with varying latencies for each link of the nodes in the cluster from 10-30ms, 20-40ms, and 30-50ms. We found that with increasing latencies, the election time also increased in proportion, as expected. However, anything more than 50 ms causing the cluster to not function properly, resulting in multiple leaders. We conducted this round of simulations with the leader heartbeat at 140ms, and the election timeout for each follower was randomly assigned between 150-300ms. The graph in Figure 12 shows the variation in election times with changing latencies.

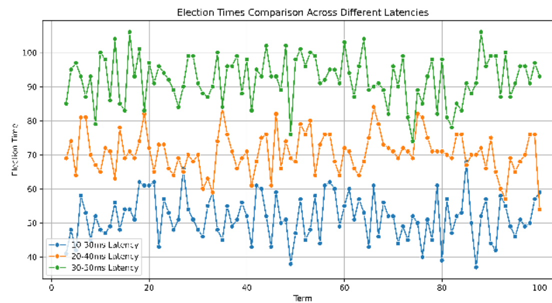

Fig 12: Varying election times with changing latencies.

The median latency for a normal raft cluster was 30 ms. We tested with varying latencies for each link of the nodes in the cluster from 10-30ms, 20-40ms, and 30-50ms. We found that with increasing latencies, the election time also increased in proportion, as expected. However, anything more than 50ms causes the cluster to not function properly, resulting in multiple leaders. The random frequencies generated followed a normal distribution. We continued with our simulations and tested the effect of changing the latencies on the normal operation time of a cluster of five nodes. We found that there was a slight difference, but it did not significantly affect normal operations. Figure 13 shows three types of line graph.

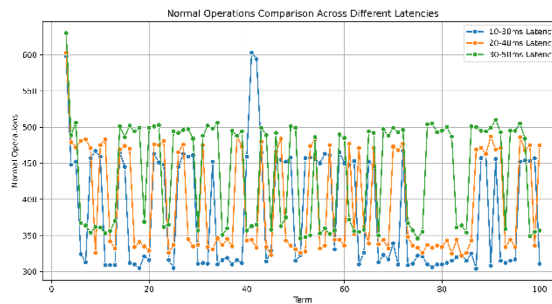

Figure 13: Normal operations with varying latencies.

As previous studies have only tested a Raft cluster until 10 or 12 nodes, there seems to be no data available with an increasing number of nodes and their impact on election time with the leader being repeatedly downed by a faulty link in the network. We tested this with our DES simulation ranging from 5 nodes to 20 node clusters. We attempted to simulate a smaller WAN with five geographically closer nodes to a global network of 20. Because we were not using an original network of nodes spread throughout the world, we included a slightly increased latency for increments of five nodes each. Therefore, for a 5-node cluster, the latency is the lowest, whereas for a 20 node cluster, the latency is the maximum.

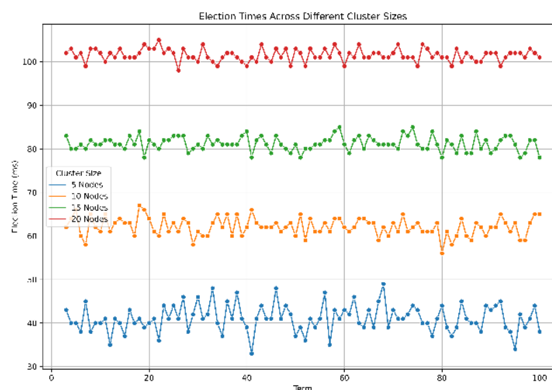

Fig 14: Election times with varying number of nodes from 5 to 20 (with varying latencies)

The graph in Figure 14 depicts the increase in election time as the number of nodes increases (as well as the latency). This difference can be compared to the graph in Figure 12.

We also tested varying election timeouts for followers and repeatedly severed the link between the leader and a random node. The blue line in the graph in Figure 12 shows the normal election time with a latency of 10-30ms. We maintained the same latency and tested both the leader election and normal operation time, but the follower timeout increased from the normal 150-300ms to 300-450ms and 450-600ms. From analyzing the graph in Figure 15, we found that for a 5 node Raft cluster, there is no significant change in election times, but there is a significant change when the follower election timeout is increase to 450-600ms. We found that the cluster remained stable with a leader longer with longer follower timeouts than with shorter timeouts. However, this is not true stability as the leader might not be operational or the network partitioned. With longer timeouts, it takes more time for the followers to realize that there is no leader and start a round of voting.

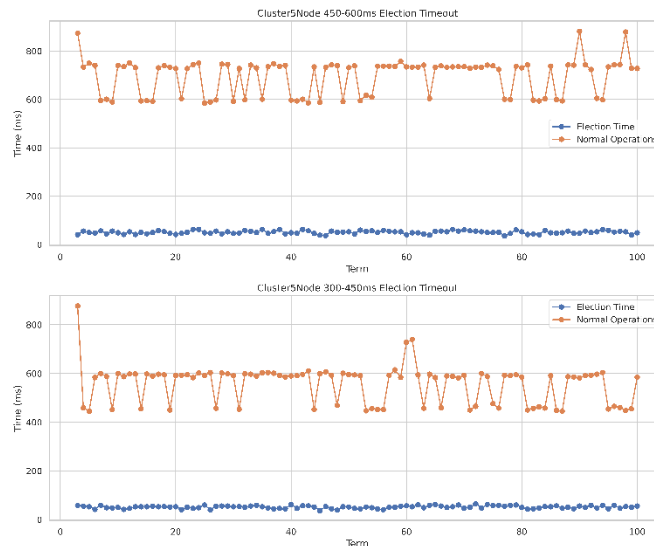

Fig 15: Raft cluster of 5 nodes with varying follower timeouts.

The insights generated from the above results of the discrete event simulation of the raft leader election process can help programmers and system architects choose the parameters for their raft implementation. Although our tests are run on a simulation and not on real testbeds, our simulation results might point to a general birds-eye view of Raft's fault tolerance.

VII. CONCLUSION

In conclusion, this study presents a significant contribution to understanding and improving the fault tolerance capabilities of the Raft consensus algorithm. Through a series of well-illustrated UML sequence diagrams, we effectively dissect the algorithmic processes of Raft, shedding light on its behaviour during various failure scenarios. This paper illustrates edge cases where Raft's liveness is challenged by network partitions. The study illustrated the liveness issues of Raft, especially pertaining to the leader election process, through the use of diagrams and portrayed how the cluster functions even in the case of failures through simulation. We depicted diagrammatically how Raft tolerates failures (and comes back alive) and tested Raft's promises of stability through our discrete event simulation. We tested various corner and edge cases and pushed the simulation to a larger number of nodes. We applied Raft's leader election algorithm by increasing various parameters, including the number of nodes. Through illustrative examples and graphical simulation results, we hope to contribute to the understanding of the Raft consensus algorithm (especially the leadership process), as the original premise of developing Raft is its understandability and not necessarily performance and fault tolerance.

REFERENCES

[1] P. J. Marandi, M. Primi, and F. Pedone, "Multi-ring paxos," *IEEEIFIP Int. Conf. Dependable Syst. Netw. DSN 2012*, p. 1—12.

[2] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," *2014 USENIX Annu. Tech. Conf. USENIX ATC 14*, p. 305—319.

[3] "etcd," etcd. [Online]. Available: https://etcd.io

[4] "CockroachDB." Accessed: Dec. 22, 2023. [Online]. Available: https://www.cockroachlabs.com/

[5] D. Huang *et al.*, "TiDB: a Raft-based HTAP database," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3072–3084, Aug. 2020, doi: 10.14778/3415478.3415535.

[6] C. Gyorodi, R. Gyorodi, G. Pecherle, and A. Olah, "A comparative study: MongoDB vs. MySQL," in *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)*, Oradea, Romania: IEEE, Jun. 2015, pp. 1–6. doi: 10.1109/EMES.2015.7158433.

[7] V. M. Ionescu, "The analysis of the performance of RabbitMQ and ActiveMQ," in *2015 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER)*, Craiova, Romania: IEEE, Sep. 2015, pp. 132–137. doi: 10.1109/RoEduNet.2015.7311982.

[8] D. Fernandes and J. Bernardino, "Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB:," in *Proceedings of the 7th International Conference on Data Science, Technology and Applications*, Porto, Portugal: SCITEPRESS - Science and Technology Publications, 2018, pp. 373–380. doi: 10.5220/0006910203730380.

[9] Department of Computing and Informatics, Mazoon College, Muscat, Sultanate of Oman., M. Nasar, M. A. Kausar, and Department of Information Systems, University of Nizwa, Nizwa, Sultanate of Oman., "Suitability Of Influxdb Database For Iot Applications," *Int. J. Innov. Technol. Explor. Eng.*, vol. 8, no. 10, pp. 1850–1857, Aug. 2019, doi: 10.35940/ijitee.J9225.0881019.

[10] K. Subramanian, "Introducing the Splunk Platform," in *Practical Splunk Search Processing Language*, Berkeley, CA: Apress, 2020, pp. 1–38. doi: 10.1007/978-1-4842-6276-4_1.

[11] "RedPanda," RedPanda. Accessed: Jun. 21, 2024. [Online]. Available: https://redpanda.com/

[12] S. Tian, F. Bai, T. Shen, C. Zhang, and B. Gong, "VSSB-Raft: A Secure and Efficient Zero Trust Consensus Algorithm for Blockchain," *ACM Trans. Sens. Netw.*, vol. 20, no. 2, pp. 1–22, Mar. 2024, doi: 10.1145/3611308.

[13] X. Wu, C. Wang, and Z. Liu, "Raft consensus algorithm based on reputation mechanism," in *International Conference on Computer Network Security and Software Engineering (CNSSE 2022)*, SPIE, Oct. 2022, pp. 272–281. doi: 10.1117/12.2640755.

[14] Z. Zhan and R. Huang, "Improvement of Hierarchical Byzantine Fault Tolerance Algorithm in RAFT Consensus Algorithm Election," *Appl. Sci.*, vol. 13, no. 16, p. 9125, Aug. 2023, doi: 10.3390/app13169125.

[15] "Cloudflare etcd raft outage," Cloudflare etcd raft outage. [Online]. Available: https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/

[16] "A byzantine failure in the real world (Nov 2020).," A byzantine failure in the real world (Nov 2020). [Online]. Available: https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/

[17] 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.