International Journal of Engineering and Information Systems (IJEAIS)
ISSN: 2643-640X
Vol. 7 Issue 5, May - 2023, Pages: 16-21

In Search of a faster Consensus Algorithm (H**SEB**)

Salah M. Ahmed, Emad Abu Alsaid, Bahaa Shtewi, Prof. Hatem Hamad

Salah M. Ahmed
Computer Engineering Dep.
Islamic Univeristy of Gaza
Gaza, Palestine
Salah@ait.ps

Emad Abu Alsaid
Computer Engineering Dep.
Islamic Univeristy of Gaza
Gaza, Palestine
eharbid@gmail.com

Bahaa Shtewi
Computer Engineering Dep.
Islamic Univeristy of Gaza
Gaza, Palestine
bahaaeshteiwy@gmail.com

Prof. Hatem Hamad
Computer Engineering Dep.
Islamic Univeristy of Gaza
Gaza, Palestine
hhamad@iugaza.edu.ps

Abstract*— As our lives increasingly move into the digital space and the internet becomes more ubiquitous, the need for reliable and scalable distributed systems becomes more pressing. In such systems, achieving consensus among multiple servers in the event of a system failure is a critical issue. Two popular algorithms, Paxos and Raft, have been developed to address this problem. While both algorithms have similar performance, Raft is considered simpler than Paxos. However, Raft can encounter difficulties with multiple rounds of leader elections when network delays occur, and the use of a random timeout machine cannot control the participation of candidate nodes in the election process simultaneously. To solve this issue, a novel algorithm is presented in this paper that eliminates the leader election process and selects the leader based on its NodeID number, generated using a timestamp and the MAC address of the node. This approach is expected to improve performance and reduce the time needed for leader appointments.*

Keywords—Consensus, Raft, Paxos, Distributed, state machine

I. INTRODUCTION

Ensuring consistency and reliability in distributed systems, especially in the incidence of faulty processes, is a essential problem [1]. To reach consensus, this requires processes coordination or an agreement between nodes on the needed data value during the computation to act as single entity [2]. Consensus is necessary in various scenarios, such as ensuring ordered updates through reliable multicast, detecting potential failures through failure detection, and enabling exclusive access to a resource (mutual exclusion, leader election process [3]. Consensus applications include various examples such as state atomic broadcasts and machine replication. Consensus is also crucial in practical applications such as cloud computing, smart power grids, and state estimation. Nonetheless, the prevailing consensus algorithms are Paxos [4].

According to Howard and Mortier (2020) research, Although both Paxos and Raft employ a similar approach to achieve distributed consensus, they differ in their leader election mechanism. Paxos is preferred in this regard as unexpected leader elections in Raft may adversely affect performance, especially if the elected leader is located in an unfavorable position. [5]. Nevertheless, Howard and Mortier claims Raft's method is highly effective due to its simplicity, which avoids the need for log entries to be exchanged through leader election, unlike Paxos. However, the Raft algorithm still faces a challenge with multiple rounds of leader elections, especially when network delays occur or when a candidate node is offline [4]. The random timeout mechanism is inadequate for controlling the participation of candidate nodes in the election simultaneously in such cases. To address the issue of extended election times caused by various rounds of elections, this paper proposes a vote modification mechanism, resulting in a consensus algorithm called HSEB. The design of HSEB includes specific techniques such as log replication and safety to enhance leader selection.

II. CONSENSUS IN DISTRIBUTED SYSTEMS

Achieving consensus is a critical concern in distributed systems. It is particularly challenging when the algorithm employed necessitates a shared state or action, in a setting where there is a likelihood of process failures. In such cases, multiple processes (or agents) must agree on a proposed value despite the possibility of partial failures [6]. Nonetheless, a possible solution for achieving consensus is by having all processes (or agents) agree on a majority value, implying that the majority needs at least one more than half of the total available votes (with every process assigned a vote). However, if one or more faulty processes are present, the resulting outcome may be missed or lead to an incorrect consensus. Hence, consensus protocols are developed to handle a limited number of faulty processes. This allows each process (or agent) in the consensus position to provide or decide on a value, while ensuring the following characteristics of consensus [6]:

- Validity: The decision is made from the proposed values.
- Termination: Ultimately, every functional process/server will agree on a value.
- Integrity: If all functional processes/servers have proposed the same value, then every functional process/server must decide on that value.
- Agreement: It is essential for all functional processes/servers to agree unanimously on a single value.

These characteristics ensure that simple algorithms cannot function as consensus algorithms, even without time constraints. Additionally, each node must consider the issue of partial

synchrony time assumptions, which allows an algorithm to meet the requirements for consensus.

III. REPLICATED STATE MACHINES

The concept of a replicated state machine is employed to address various fault tolerance challenges in distributed systems. Consensus algorithms are commonly utilized in the context of replicated state machines [7].

A group of servers can create identical replicas of the same state using state machines. Thus, if some of the servers are non-operational, the cluster can continue functioning.

The use of replicated state machines involves the utilization of a replicated log, as depicted in Figure 1.

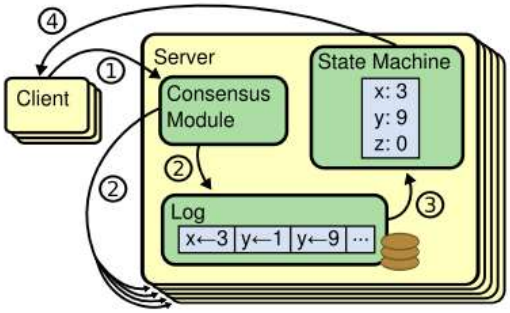

Figure 1:Replicated state machine architecture. Source [5]

To guarantee fault-tolerance in a distributed system, replicated state machines are utilized. Each server within the system maintains a log of executed commands in a specific order. The logs of all servers have the same set of commands in the same order, ensuring deterministic behavior. The consensus algorithm is in charge for maintaining the replicated log and facilitating communication between the consensus modules of each server, guaranteeing that all logs contain the same commands in the similar order, despite if some servers malfunction. Once the commands are appropriately replicated, each server's state machine executes them in log order, and the results are returned to clients. Consequently, the system appears to function with a single, highly consistent state machine. Consensus algorithms employed in practical systems have several critical features.

Firstly, they ensure safety, meaning that they never return an incorrect result below any non-Byzantine conditions, including network delays, partitions, packet loss, duplication, and reordering.

Secondly, they remain fully functional and available as long as a majority of the servers are operational and can communicate with each other and with clients. In a typical cluster of five servers, for example, the system can tolerate the failure of any two servers. Failed servers are assumed to stop, but they may later recover from a stable storage state and rejoin the cluster.

Lastly, these consensus algorithms do not depend on timing to guarantee the consistency of the logs. Faulty clocks and extreme message delays may cause availability problems at worst, but they do not affect the system's consistency. In most cases, a command can be completed as soon as a majority of the cluster has responded to a single round of remote procedure calls. Slow servers in the minority need not influence the general system performance.

IV. APPROACH OF PAXOS & RAFT

According to [8], several consensus algorithms, including Paxos and Raft, employ a leader-based approach to manage distributed consensus. At a high level, the algorithm operates as follows: out of n servers/processes, one is elected as the leader and the rest function as followers. The leader receives all operations from the state machine and adds them to its log/sequence, requesting that the followers do the same. The leader waits for acknowledgement from the followers before applying its operation to its state machine. This process continues until the leader fails, at which point another follower server takes over as the leader. However, electing a leader from the different following servers requires the involvement of most of the other following servers, with the condition that the candidate leader has the latest log/sequence. To sum up, any server/process in the system can have one of the following statuses:

- Follower: A state which it is role only for replying to RPCs.
- Candidate: A state where it role is trying to be a leader by using the RequestVotes RPC.
- Leader: An active state where it is role for adding operations to the replicated log/ sequence using the Append Entries RPC.

V. COMPARISON BETWEEN PAXOS & RAFT

Both Paxos and Raft utilize a leader-based approach to achieve consensus, but they use different terminology. Paxos uses the term "processes" while Raft uses "servers". Similarly, Paxos has "sequences" while Raft has "logs". However, differences in terminology are not the only distinctions between the two algorithms. In fact, there are several other differences between them. To elaborate, Dubinin et al. mentioned that the Raft implementation has better performance efficiency compared to the single-value Paxos, which is essentially incremental. [9]. In support with this finding, HARALD study results revealed that Leader-based Sequence Paxos and Raft have similar performance in geographically distributed placements[5]. Though, the random leader election in Raft may disturb the performance based on leader location. Whereas, Howard and Mortier mentioned in their study that the two algorithms have a lot of similarity and complication level. However, the Raft algorithm was presented clearly in papers than the Paxos [4].

VI. LEADER ELECTION PHASE PROBLEMS

As previously noted, the Raft consensus algorithm faces challenges in leader election. To initiate the process, a follower server will add its own term number and change to a candidate state. The candidate server will then send a request vote RPC to the surrounding follower servers in order to obtain their votes.

The candidate will remain in this state until one of three conditions is met:

1. The candidate becomes the leader if it receives votes from more than half of the follower servers.
2. Another candidate becomes the leader, causing the current candidate to become a follower.

3. No candidate node is able to secure more than half of the votes.

Sometimes, multiple candidate servers may compete to become the leader concurrently, resulting in none of them receiving more than half of the votes from other servers. As a consequence, multiple rounds of leader elections may be necessary. While the Raft algorithm employs a random timeout mechanism to prevent multiple candidate servers from participating in the election simultaneously, it is still vulnerable to election conflicts in situations where the network is delayed or the candidate server is unavailable. To address this issue, we propose a new consensus algorithm called the "HSEB" algorithm in this paper.

VII. THE HSEB CONSENSUS ALGORITHM

The HSEB algorithm is designed to handle a replicated log, as depicted in Figure 2. The algorithm can be summarized in a condensed form, as shown in the figure, for easy reference. The consensus is achieved by first appointing a suitable leader, who is then given complete responsibility for managing and organizing the replicated log. The leader receives client entries and distributes them to other servers, ensuring smooth data flow. In case of leader failure or separation from the rest of the servers, a new leader is appointed. By using a leader-based approach, HSEB breaks down the consensus problem into three relatively independent sub-problems, which are discussed in detail in the following subsections.

Leader election: a different leader must be picked when a current leader fails.

Log replication: the leader must admit log entries from clients and duplicate them through the cluster, imposing the other logs to approve its log.

The Safety Property of State Machine is the primary safety feature of HSEB, as depicted in Figure 3. It ensures that if a node has submitted a particular log entry to its state machine, none of the other nodes will apply different commands for the same log index.

The HSEB consensus algorithm is summarized in a condensed form in the following sections, excluding membership variations and log compaction. The attitude of the node in the upper-left box is defined as a set of independently and frequently triggered rules. A more specific description is necessary to define the algorithm accurately.

VIII. Working MECHANISM of HSEB algorithm

A. Leader's Appointment

At the system's initial state, all nodes are in the Follower role. As the system runs, each server generates a random and unique number. Any method that ensures no overlap and unevenness between numbers can be used. The UUID method, which generates a universally unique identifier using a timestamp and the MAC address of the node, is used in this case. Each server considers this number as its ID number and broadcasts it to the rest of the nodes. When a node receives IDs from other nodes, it creates an array and reorders the ID numbers based on their values. The highest value is considered as the leader. This process occurs during the first run of the system or when the leader fails, as the servers do not receive Heartbeat messages from the leader.

B. Log replication

After receiving instructions from the client, the Leader adds them to its local registry as an Uncommitted state. Simultaneously, the Leader copies the command to other nodes and waits for them to finish writing the command. If any node fails, the Leader resends the command and then returns the result to the Client node. Once the command is successfully executed, the Leader sends messages to the servers containing the highest index of the record, and the contents of the Leader node's log overwrite the log of each Follower node. [10].

C. Safety

The previous section discussed how the algorithm handles leadership appointments and log replication, but it does not guarantee that all state machines execute the same commands in the similar order. For instance, if a Follower node fails after the Leader submits several log entries, a new Leader may be appointed and overwrite the original entries with new ones, resulting in different command sequences being executed on different state machines. To address this issue, our algorithm adds a restriction: after a new Leader is appointed, it sends a log copy request to all nodes in its majority group to obtain the index of their last log record. The Leader then overwrites its log with the highest index received, and updates the log content of Follower nodes accordingly. This ensures that all state machines execute the same commands in the similar order

State

Persistent state on all servers:
(Updated securely on permanent storage prior to responding to RPCs)

NodeID	On first, boot each node will randomly generate a unique identifier by using a timestamp and the MAC address of the of node
Log[]	A log entry includes a command for the state machine and a NodeID indicating when the leader received the entry. The first index is 1.

Volatile state on all servers:

CommitIndex	The index indicating the highest committed log entry is initialized to 0 and then increases monotonically over time.
lastApplied	The highest log entry index that has been applied to the state machine is initially set to 0 and increases monotonically as new entries are applied.
NodesOrder[]	Array used to order the nodes according to the highest NodeID value

Volatile state on leaders:

nextIndex[]	The index of the next log entry to be sent to each server is initialized to the last log index of the leader plus one.
matchIndex[]	The index of the highest log entry that is known to be replicated on each server is initialized to 0 and increases monotonically over time.

AppendEntries RPC

Initiated by leader to duplicate log entries; also utilized as heartbeat

Arguments:

HighIndex	Leader Index
leaderId	Follower can forward clients
prevLogIndex	The index of the log entry immediately preceding the new log entries.
prevLogID	ID of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

HighIndex	Current Leader Index for leader to update itself
Success	leader confirms the consistency and synchronization of its own log with a follower's log when it receives an entry matching the prevLogIndex

Receiver implementation:

1. If there is no log entry at prevLogIndex, respond with false. If there is a conflict between an existing entry and a new entry, remove the existing entry and all the subsequent entries.
2. Attach any new entries that are not already present in the log.
3. If leaderCommit is greater than commitIndex, set commitIndex to the minimum value between leaderCommit and the index of the last new entry.

IDbrodcast RPC

Broadcast by each node to pick the leader

NodeID	NodeID generate by timestamp and the MAC address of the of node
NodesOrder[]	Array that used to order the nodes according to the highest NodeID value

Results:

NodesOrder[]	Array that contains the IDs of nodes according to the highest ID value
HighIDvalue	Means that the node will be picked as leader for the first boot
NodesOrder[]	Array used to order the nodes according to the highest NodeID value

Receiver implementation:

1. Each node will append the NodeID to the NodeOrder array according to its value when received.
2. If the NodeOrder array contains a majority of nodes, the node with the highest index in the array becomes the leader.
3. The leader requests all nodes in the array to send their logs to compare with its own log.

4. If the leader has the latest log, it will replicate it to other nodes. If not, it will update its log and replicate it to other nodes.

Rules for Servers

All Servers:
If the commitIndex is greater than the lastApplied index, the system increments the lastApplied index, and applies the log entry at that index to the state machine.

Followers:
1. Broadcast NodeID
2. Append NodeID to NodesOrder[]
3. Respond to RPCs from the leader.
4. If leader fail, send IDbrodcast RPC to the next highest NodeID until receive a response
5. IF its ID the highest ID, it should receive IDbrodcast RPC from other nodes.
6. If a candidate receives votes from the majority of servers, it becomes the new leader.

Leader:
1. Send initial AppendEntries Remote Procedure Call (RPC) to each node containing no entries (i.e., heartbeat), and repeat periodically during idle times to avoid connection timeouts.
2. Upon receiving a command from a client, add the entry to the local log, respond after applying the entry to the state machine.
3. If the follower's last log index is greater than or equal to nextIndex, send AppendEntries RPC with log entries starting from nextIndex.
4. If successful, update nextIndex and matchIndex for the follower.
5. If the AppendEntries RPC fails due to log inconsistencies, decrease nextIndex and repeat.

Figure 2: A summarization of the HSEB algorithm

IX. Case Study

At the start of the process, there is no leader, and the leader selection process begins with nodes exchanging messages that include their index number (ID). A table is then created with the names of the nodes and their corresponding ID numbers, and the leader is chosen based on the highest ID number. This process only takes place during the initial leader selection. Afterwards, the leader is chosen sequentially based on the highest ID number without nodes exchanging messages with each other. The first leader is chosen using the aforementioned process. Once a leader is chosen, the client sends instructions to the leader. The leader then sends the command to other nodes simultaneously and waits for their responses, which include their ID numbers and an acknowledgement that they are alive. The node with the fastest response becomes the first follower after the leader, and so on.

A. Network Partition

After network disconnection in subnet 2, master node A is unable to communicate with most nodes as depicted in figure 3. As a result, the other nodes fail to receive pulse messages from the leader, which contain the index number and last log number of each node. When the nodes do not receive a response from the leader, the nodes that were unable to contact the leader immediately send messages to the highest indexed node. If the node receiving the messages checks the leader status and does not receive a response, it declares itself as a new leader. In this case, C would become the new leader because it contains the highest index number or is the first among CDE. Therefore, C takes over as the new leader of the network.

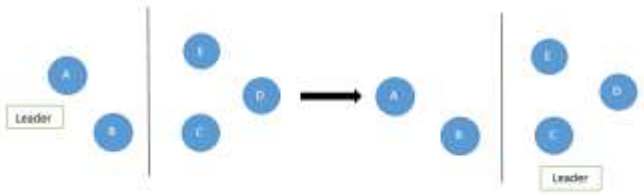

Figure 3. Network Partition

B. Leadership election with network partition.

After the partition isolation is resolved, the new leader C begins transfer heartbeat messages to entire nodes in the network, and waits for their responses. Once the previous leader node becomes available again, it will return as a new node with a new index number assigned to it, located at the end of the index table. However, node C still retains its position as the leader of the network.

X. Conclusion

Consensus protocols are designed to handle a limited number of faulty processes, enabling each process to provide or decide on a value. These algorithms prioritize integrity, validity, and termination characteristics, preventing trivial algorithms from qualifying as consensus algorithms without any time-bound restrictions. Additionally, each node must consider the discussion of partial synchrony time assumptions to satisfy the algorithm. In this paper, we explored the consensus problem in distributed systems and the effectiveness of well-known algorithms such as Paxos and Raft in addressing it. We introduced a new algorithm, HSEB, which offers higher efficiency and speed than Raft, especially during the leader election phase. The improved HSEB algorithm features a leader

appointment mechanism instead of an election phase, and utilizes Node ID values to select a leader for achieving consensus. It also implements the ID Broadcast mechanism during log replication to enhance leader node throughput and consensus efficiency.

REFERENCES

[1] R. Koo and S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems," *IEEE Trans. Softw. Eng.*, vol. SE-13, no. 1, pp. 23–31, 1987, doi: 10.1109/TSE.1987.232562.

[2] M. J. Fischer, "The consensus problem in unreliable distributed systems (a brief survey)," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 158 LNCS, pp. 127–140, 1983, doi: 10.1007/3-540-12689-9_99.

[3] C. C. Fan and J. Bruck, "The raincore distributed session service for networking elements," *Proc. - 15th Int. Parallel Distrib. Process. Symp. IPDPS 2001*, vol. 00, no. Figure 1, pp. 1673–1681, 2001, doi: 10.1109/IPDPS.2001.925154.

[4] H. Howard and R. Mortier, "Paxos vs Raft: Have we reached consensus on distributed consensus?," *Proc. 7th Work. Princ. Pract. Consistency Distrib. Data, PaPoC 2020*, 2020, doi: 10.1145/3380787.3393681.

[5] N. HARALD, "Distributed Consensus: Performance Comparison of Paxos and Raft," 2020.

[6] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th ed. New York, 2012.

[7] L. Lamport, D. Malkhi, and L. Zhou, "Reconfiguring a state machine," *ACM SIGACT News*, vol. 41, no. 1, 2010.

[8] H. Howard and R. Mortier, "Paxos vs Raft: Have we reached consensus on distributed consensus?," *Proc. 7th Work. Princ. Pract. Consistency Distrib. Data, PaPoC 2020*, pp. 8–10, 2020, doi: 10.1145/3380787.3393681.

[9] V. Dubinin, A. Voinov, I. Senokosov, and V. Vyatkin, "Implementation of distributed semaphores in IEC 61499 with consensus protocols," 2018, doi: 10.1109/INDIN.2018.8472078.

[10] L. Lamport and D. Equipment, "The Part-Time Parliament," vol. 16, no. May 1998, pp. 133–169.