Distributed Election Survey

Carol Michail 900194282
Mohamed Abdelmohsen 900201290
Muhammad Azzazy 900202821
Ramy Badras 900194248

Department of Computer Science and Engineering
The American University in Cairo

Fall 2024

1 Introduction

Distributed election is a vital issue for building distributed systems that are fault-tolerant. It is highly critical for distributed systems and difficult since data is distributed among various geographically isolated processes or nodes. A process or node must be chosen to coordinate processes or nodes. The main role of a selected node is to manage the utilization of shared resources efficiently. In distributed systems, nodes or processes communicate via shared memory or message passing. In distributed systems, no central node arbitrates decisions, and each node must talk to the other nodes in the network for decision-making. However, coordination among different nodes is challenging when consistency is required among nodes. Distributed election is a method to break the symmetry of distributed systems. It aims to choose a node that will coordinate the system's activities. Distributed election algorithms rely on the network topology. Some distributed election algorithms are based on the ring (spanning tree) network topology, while others are based on the fully connected network [1]. We need to select which distributed election algorithm to use and why. For load balancing, we will use distributed election so that the nodes on equal footing agree on one of them to carry any coming requests. Not one server would be bombarded with 5 million requests, and the others are idle. We are going to be using distributed election to simulate failures.

2 Distributed Election Algorithms

2.1 A Ring Algorithm

This algorithm is based s based on the use of a logical ring. The assumption is that each process knows its successor. When any process notices that the coordinator is down, it starts the election algorithm. This happens by building an ELECTION message that includes its own process identifier and sends the message to its successor. If the successor is down, it skips over to the next process till it finds the next operating process. At each step along the way, the sender adds its own identifier to the list in the message effectively making itself a candidate to be elected as coordinator. When the messages gets back to the process that intiated it, the process recognizes this event. This happens when it notices that the incoming received message contains its own identifier. The process with the highest identifier becomes the coordinator and a new message is circulated. The message type this time is COORDINATOR and is there to inform all processes who the new coordinator is and who the members of the new ring are. Once the message is circulated, it is removed and all processes go back to work normally. [2]

2.1.1 Example

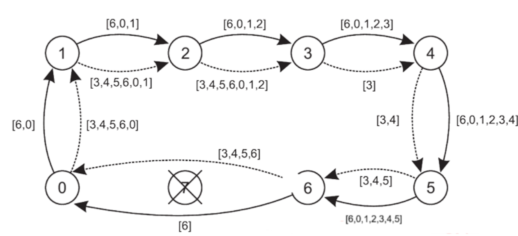

Figure 1: The Ring Algorithm

2.2 Modified Ring-Based Election Algorithm

The same study by EffatParvar et al. proposed an improvement to the traditional ring-based election algorithm. In the standard version, when a leader crashes, every node sends its ID into the ring. Each node forwards the ID it receives, comparing it with its own, until the greatest ID circulates back to the initial sender, who declares the new leader.

The modified algorithm reduces message passing by allowing only the node with the greatest ID to continue forwarding. If a node receives a higher ID than its own, it stops participating. The process continues until the greatest ID reaches the original node, which then declares itself the leader [3].

Message Complexity: $O(n^2)$

2.2.1 Pros

- **Lower Message Overhead**: Only the highest ID circulates, reducing the total number of messages.
- **Faster Election**: Fewer nodes continue to send messages, speeding up the process.

2.2.2 Cons

- **Potential Fragility**: The algorithm relies on only the highest node correctly which may potentially fail after sending its ID to circulate.

2.2.3 Comparison with the Standard Ring-Based Algorithm

- **Efficiency**: The modified version is more efficient, reducing unnecessary messages compared to the standard algorithm, where all nodes participate fully.
- **Scalability**: As the system size increases, the modified version scales better by reducing communication overhead.
- **Resilience**: The standard algorithm is potentially more robust, as it involves all nodes, providing more redundancy in case of message loss or node failures.

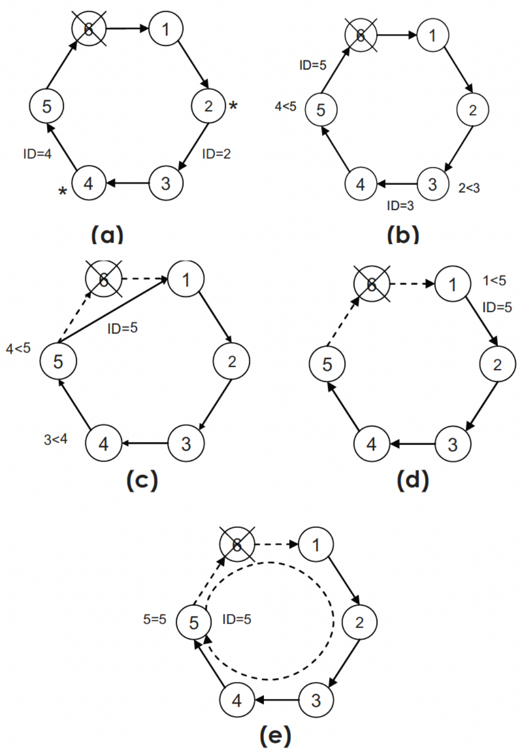

Figure 2: Modified Ring-Based Algorithm

2.3 Heap Tree Election Algorithm

The Heap Tree Election Algorithm, introduced by EffatParvar et al. in 2010, presents a novel approach to leader election in distributed systems [3]. In this algorithm, nodes are organized in a binary heap structure stored in an array, with each node associated with a parent and two children (left and right). The leader, defined as the node with the largest value, resides at the root of the heap tree. When a new node is added, it joins the tree and compares its ID with that of its parent. If the new node's ID exceeds that of its parent, they swap positions, thereby restructuring the tree to maintain the max-heap property [3].

A notable feature of this algorithm is that not all nodes need comprehensive knowledge of the entire tree. When the root node (the leader) crashes, it triggers an election process. Nodes send an election message to their parent until it reaches the children of the crashed root. The left and right children then compare their IDs to determine the new leader. The child with the highest ID is elected as the new root, thus preserving the max-heap property [3].

Figure 1 illustrates the election process in a distributed system utilizing a heap tree. In part (a), the tree structure with all node IDs is depicted, while part (b) shows the corresponding indices. In part (c), the root crashes, and the node at index 6 detects the failure, sending a message to the children of the crashed node. Finally, in part (d), the node with index 3 (ID of 10) becomes the new root after comparing its ID with its sibling [3].

Message Complexity: Each node in the heap tree communicates only with its parent and children, meaning a node sends at most a constant number of messages to its immediate neighbors. Given that messages propagate upward through the tree and the height of the tree is $O(\log(N))$, the overall message complexity is proportional to the number of levels in the tree, resulting in a complexity of $O(\log(N))$.

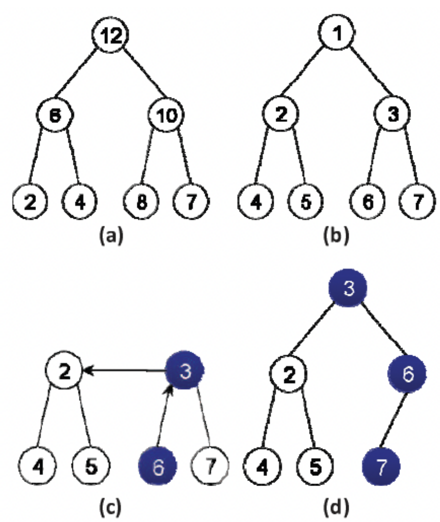

Figure 3: Heap Tree Algorithm

2.4 Timer-Based Election Algorithm

2.4.1 Proposed Algorithm

There are n processes or nodes in a distributed system. We must choose a node among these n nodes or processes to control the utilization of shared resources efficiently. Processes or nodes talk to each other via message passing. At any point of time, one or more processes can realize the necessity of selecting a node. Hence, it is difficult to design a distributed election algorithm that performs well in both scenarios: where one node or process realizes the necessity of distributed election or multiple nodes realize that necessity. A ring topology does not need a central device to control the connectivity between computers. The ring topology allows resources to be distributed equally among processes or nodes. Additionally, the ring topology is simpler to install than other network topologies. The following are the assumptions [1]:

- Each process or node has a unique ID which is an integer that ranges from 0 to n-1 where n is the number of processes in the ring
- Each process has a timer.
- A message takes n hops to make a cycle. Each timer must be set for n hops of time.
- A process is capable of being chosen in the election, and the chosen process may not have the highest ID.
- Message moves in the clockwise direction.
- After receiving a message, a process does not create a message to be the chosen one.
- Each process knows the number of processes in the ring and any process has some info about the processes directly before and after it.
- Channel is 100% reliable.

The variables utilized include [1]:

- *scp_id* is an integer variable which stores the process ID of the chosen process.
- *mcp_id* is an integer variable that contains the ID of the process which created the message.
- *rcp_id* is an integer variable which holds the process ID of the recently crashed elected process.

2.4.2 Message Complexity

The best case occurs when only one process realizes the necessity of distributed election at any time point. A message is created from a process, and it passes through n hops to return to the process which created it, so the best case message complexity is $O(n)$ [1].

The average case occurs when a number of processes realize the necessity of distributed election at any time point. If M is the number of required messages, x is the number of processes that realize the necessity, and n is the total number of processes, then [1]

$$M = xn - x$$

In this case, the message complexity is $O(n)$ [1].

The worst case for message complexity occurs when all processes realize the necessity of a run of the distributed election algorithm and the processes are oriented in decreasing order of process ID. The required number of messages is [1]

$$M = \frac{n^2 + n}{2}$$

Hence, the worst case message complexity is $O(n^2)$ [1].

2.4.3 Time Complexity

If a single process or node realizes the necessity of a run of the distributed election algorithm, then the message created by the process or node which realized the necessity passes through all nodes or processes in the ring. For a message to rotate through a ring, it takes n hops of time. Each process can identify the chosen process once their timer has elapsed. So, the time complexity of the timer-based distributed election algorithm is O(n) [1].

2.4.4 Analysis

The primary contribution of this algorithm is that almost all election algorithms pick the process with the highest ID. However, picking the process with the highest ID is a burden. It might increase the time and message complexities. According to the authors, when a process realizes the need for a run of the election algorithm, then that process is chosen. When more than one process realizes the need for a run of the election algorithm, one of the processes which realized that need and has the highest ID is chosen. This means that the elected process may or may not be the process with the highest ID in the set of n processes. This algorithm is very similar to LCR. In the timer-based election algorithm, announcing the coordinator is unnecessary. The expiration of the timer of each process makes a process know the elected process. Thus, the timer-based algorithm takes n hops of time and n number of message passing less than that of LCR and the ring-based algorithm to choose a process [1].

The performance of the timer-based algorithm relies on the process orientation in the logical ring. The timer-based algorithm performs exceedingly well if the processes are oriented in an increasing fashion in the logical ring. If all the processes realize the need for a run of the distributed election algorithm, the optimal orientation for the processes consumes 2 n-1 messages. The message complexity of the algorithm in the best case is $O(n)$.

2.5 Improved Heap Tree Election Algorithm

2.5.1 Proposed Algorithm

Every node of a heap tree is associated with an element in the array that holds the node's value. The heap tree is filled on all its levels except for maybe the lowest that is filled from the left. The heap is represented using an array, A, which has a couple of attributes [4]:

- the length of the array: length[A]
- the number of elements of the heap stored within the array: heap_size[A]

$A[1]$ represents the root of the heap tree. Provided that we have the index of a node, the indices of the parent, left child, and right child can be calculated with ease. The nodes' values meet the requirements of the modified heap property according to the kind of heap being utilized. There is a subroutine called "", that is important for maintaining the property of the max heap. The subroutine called $BUILD\_MAX\_HEAP()$ has a time complexity of $O(n)$ and generates a max heap using an unsorted input one-dimensional list [4].

Other subroutines that have a time complexity of $O(log\ n)$ and allow the heap to be utilized as a priority queue include [4]:

- HEAP_EXTRACT_MAX()
- HEAP_INCREASE_KEY()
- HEAP_MAXIMUM()
- MAX_HEAP_INSERT()

A new kind of heap sort is modified heap sort. The fundamental concept of the new algorithm resembles the classical heap sort algorithm. The only difference is that the modified algorithm constructs the heap using a different technique. In the worst case, the modfiied algorithm requires $nlogn - 0.788928n$ comparisons. In the average case, the modified algorithm requires $nlogn$ comparisons. The modified algorithm utilizes only a single comparison at every node. Using a single comparison, the child of the node that contains a larger value can be chosen. The node's child is promoted to its parent's position. The modified algorithm traverses the path until a leaf is reached. Since there is no necessity for comparison between children, and it is known that the the left node is greater than the right node at the same level, the left child node will be promoted to the position of its parent [5].

When a root is deleted from a heap tree, the coordinator has crashed. When a node realizes that the coordinator has crashed, it transmits an election message to its parent. The election message moves up to the children of the node which represents the coordinator that just crashed. In the modified heap structure, the identifier of the left child must be greater than or equal to the identifier of the right child. The execution time and the message passing can be significantly diminished using this technique [5].

It is not required that all nodes begin sending their identifiers or election messages in the heap tree. The election message sent by a node reaches its parent in the tree and the node which receives the election message parses it to know whether the election message is a duplicate. The election message is dropped by the node which receives it given that the election message is a duplicate. Hence, the coordinator can be chosen in a time complexity that is less than $O(\log\ n)$. This comes at the cost of a diminished number of messages. Using this approach, every node must store the information of those nodes [5]:

- its parent node
- its right and left children
- its sibling

The memory space required by the modified heap tree method is the same as that of the heap tree method which is $4n$ [5].

The new heap sort algorithm has better performance than the original heap sort algorithm for a larger number of data items [6]. It needs n log n - $0.788928n$ comparisons in the worst case [5] and n log n comparisons in the average case when using the fastest algorithm designed by Gonnet and Munro for constructing heap structures [7]. The heap sort algorithm utilizes merely a single comparison at every node. Conversely, the original heap sorting algorithm requires $2n \log n$ comparisons.

2.5.2 Analysis

Using the modified heap tree approach, merely a couple of messages are needed for electing a node. This is because when a child of a crashed coordinator receives an election message, neither does it compare its identifier with its sibling nor does it transmit the election message. The left child of the crashed coordinator gets chosen as a leader, and the selection message is sent to all nodes. Thus, [4]

$$\text{Total number of messages sent} = 2$$

Once coordinator election begins, it is supposed that each node transmits an election message to its parent except the root. Thus, if no nodes transmit duplicate election messages, then the highest number of election messages sent at the time of coordinator election will be one less than the total number of nodes [4].

2.5.3 Final Remarks

A lower time complexity and fewer messages sent using the modified heap tree approach demonstrates that it will have better performance than the algorithms proposed earlier. It was proven that the modified heap tree algorithm outperforms than the original heap tree algorithm for a larger number of data items. A correct balance between time and message complexity can be acquired through the modified heap tree election algorithm. The modified heap tree algorithm transmits a maximum number of messages that is equal to one less than the total number of nodes. So, in the worst case, the number of messages sent using the proposed algorithm is fewer than the number of messages sent from earlier proposed algorithms [4].

2.6 The Bully Algorithm

The bully algorithm was made by Hector Garcia-Molina in 1982. It starts by considering we have N processes, $\{P_0, \ldots, P_{N-1}\}$, and assuming $id(P_k) = k$. Each node in the system is assigned at system creation time a unique identification number. This identification number is used as a priority by the Bully Algorithm, so that the node with the highest priority (i.e., with the highest identification number) out of the nodes that are participating in the election will become coordinator. When any process recognizes that the coordinator is down and no longer responding, it initiates an election. The election is held by P_k as follows:

1. P_k sends an `election` message to all processes $P_{k+1}, P_{k+2}, \ldots, P_{N-1}$.
2. If no process responds, P_k wins the election and becomes the new coordinator.
3. . If one of the higher-ups answers, it takes over and Pks job is done

Any process can receive an election message from any other processes with a lower identifier. When this happens it sends back an OK message to the sender to indicate that it is alive and will take over. Then, the receiver holds an election. Eventually all processes give up but one, that one becomes the new coordinator. Next, it sends a message to all other processes announcing that it is the new coordinator. If the process that was previously down comes back up, it holds an election. If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinators job. Therefore, the name of the algorithm the bully as the process with the highest identifier value always wins.

2.6.1 Assumptions

This algorithm works under certain assumptions.[8]

1. All nodes cooperate and follow the same election algorithm.
2. The election algorithm relies on certain software facilities, including the local operating system and message handler, which are assumed to be bug-free and reliable.
3. Any message M received by node i from node j was actually sent by j and not spontaneously generated by the system.
4. Nodes have "safe" storage cells where data survives failures, and updates to these cells either complete successfully or not at all, ensuring data consistency.
5. If a node fails, it halts immediately and later resets to a fixed state; failures cannot cause unpredictable behavior, and hardware failures are detected and converted into full crashes.

6. Messages, if received, are received without corruption, with errors being detected and corrected through redundancy.
7. Messages sent from node i to node j are processed by j in the order they were sent, even if some messages are lost.
8. The communication subsystem guarantees the delivery of a message within a fixed time T if the destination node is active, and failure to acknowledge within T indicates node failure.
9. Nodes respond immediately to messages, and delayed responses are treated as node failures, triggering a reset and recovery process.

2.6.2 Example

In this example there are 8 processes. Intially P_7 was the coordinator but then it went down. P_4 was the first one to notice this and it begins the election algorithm.[2]

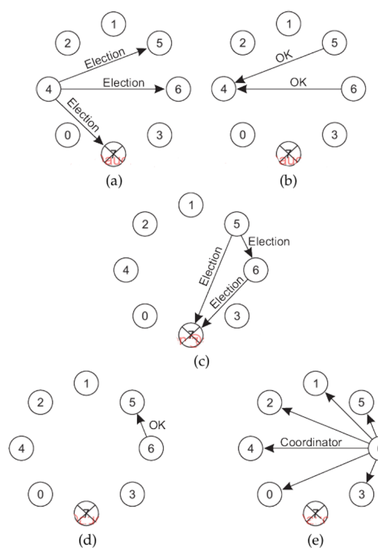

Figure 4: The Bully Algorithm

Figure 3 (a): Process 4 holds an election. Sending the election message to all processes with higher id numbers: P_5, P_6, P_7

Figure 3 (b): Processes 5 and 6 respond, telling 4 to stop.

Figure 3 (c): Now 5 and 6 each hold an election.

Figure 3 (d): Process 6 tells 5 to stop.

Figure 3 (e): Process 6 wins and tells everyone

2.6.3 Complexity

In the worst case, the process with the smallest ID initiates the election, leading to n1 rounds of elections. In each round, it sends election messages to all higher identification processes, leading to a quadratic number of messages in total.

Message Complexity: Best case $O(n)$, Worst case $O(n^2)$.

Time Complexity: Best case $O(1)$, Worst case $O(n)$

2.7 Modified Bully (Coordination Group Approach)

The Coordination Group Election Algorithm modifies the traditional Bully election algorithm by introducing the concept of a coordination group composed of a leader and several backup alternatives. If the leader crashes, the alternatives sequentially take over without the need to re-initiate a global election. Even if, in the worst case, the entire coordination group is unavailable, the system defaults to a modified global election that avoids the ineffiiencies of the traditional Bully algorithm. [9]

2.7.1 Election Process

When a leader crash is detected by process P, the next available alternative in the coordination group automatically assumes leadership after receiving a CRASH-LEADER message from process P, if alternative 1 does not respond in a certain window, alternative 2 takes over and so on. If all the coordination group is unavailable, a full global election is initiated with process P sending an ELECTION message to all higher priority processes. Process P, that initiated the global election, becomes some sort of an overseer for the election, as each process with higher priority replies with the OK message, and finally, P then chooses the highest priority as the new coordinator using a COORDINATOR message and the next K processes as the alternatives using the GRANT message. In the case where no process replies to P, it assumes leadership and chooses its coordination group using the GRANT message the same way in the previous case.

2.7.2 Handling The Worst Case

In the worst case, where multiple processes detect the failure of the leader and the entire coordination group simultaneously, each process might initiate an election, leading to parallel elections. To resolve this, Process P' (another process that detects the failure) receives an ELECTION message from lower-priority processes and waits for a short time before responding. Process P' responds only to the process with the lowest priority and sends an OK message, taking over the election.If Process P' receives an election message from another process (e.g., Process R) with a lower priority than itself but higher than Process P, Process P' sends its priority number to Process R and sends a STOP message to Process P, halting the parallel election. Once the election resolves, the newly elected leader (the highest-priority process) sends a COORDINATOR message to all other processes, announcing its leadership, and chooses its coordination group.

2.7.3 Pros

- **Reduced Message Traffic:** The introduction of a coordination group reduces the need for global elections, lowering the number of messages exchanged. Leadership transitions are handled locally within the coordination group unless all members fail.
- **Efficient Leadership Transitions:** When a leader crashes, the next alternative in the coordination group takes over immediately, reducing the downtime.
- **Parallel Election Prevention:** By using the STOP message and prioritizing responses, the algorithm avoids the risk of multiple processes running parallel elections.

2.7.4 Cons

- **Single Point of Failure During Elections:** Process P, which oversees the election, is crucial for the elections success. If it fails during the election, another process will need to detect the failure and restart the election process, introducing delays.
- **Not Byzantine Fault Tolerant:** Not Byzantine Fault Tolerant:
- **Time Delay in Full Elecitions:** While the coordination group reduces the frequency of global elections, when they do occur, the algorithm might

still experience delays, if multiple higher-priority nodes fail to respond promptly to the CRASH-LEADER message.

2.7.5 Message Complexity

The Coordination Group Election Algorithm reduces the message complexity compared to the original Bully Algorithm. In normal cases, where the coordination group handles leadership transitions, the message complexity is O(1) since communication occurs locally within the group. However, if the entire coordination group fails, a global election is triggered with Process P sending election messages to higher-priority processes. In this scenario, the message complexity is O(n), as each higher-priority process responds with an OK message, and the highest-priority process is selected as the leader. Even in the worst-case scenario, where multiple processes initiate elections simultaneously, the algorithm uses STOP messages to prevent parallel elections, maintaining the message complexity at O(n).

2.8 Modified Bully: Coordinator Variable & Election Flag Approach

The Modified Bully Algorithm works by allowing processes to initiate an election if the current leader crashes using a new mechanism involving an election flag and a coordinator variable to help minimize unnecessary elections and ensures that only one process at a time conducts the election. [10]

2.8.1 Election Process

Initially, all processes have their Election flag set to false, When process P detects that the leader has crashed, it sends an ELECTION message to all processes with a higher ID. All processes now set their election flag to true to prevent any other process from starting a simultaneous election and reset their coordinator variable to 0. The receiver process of the ELECTION message responds with an OK message to indicate that it is functioning an take over the election. Process P then extracts the process ID of the responding process and overwrites its coordinator variable if the new process ID is higher. Finally, after all processes have responded and the highest process ID among them is stored in the coordinator variable. Process P then informs the process that whose ID is stored in the coordinator variable that it has become the new coordinator. The new coordinator finally checks once more if any processes with higher ID exist, if yes the higher takes over, if not, the process announces itself as the new coordinator. Finally, all processes set the coordinator ID in co- ordinator variable and reset election flag to false.

2.8.2 Pros

- **Reduced Message Traffic & Parallel Election Prevention:** By using the election flag, the algorithm ensures that only one election is conducted at a time, which reduces the number of messages exchanged.
- **Faster Leader Selection:** The coordinator variable allows the process to easily determine the highest-priority node, minimizing redundant message exchanges.

2.8.3 Cons

- **Dependency on Timout Duration:** The performance of the algorithm depends heavily on the chosen timeout period. A long timeout can delay the election, while a short timeout may cause the process to miss responses from legitimate processes.
- **Not Byzantine Fault Tolerant:** Not Byzantine Fault Tolerant: Assumes non-Byzantine failures which may again, limit its use.
- **Single Point of Failure During Elections:** If the process that initiates the election fails midway, the election will have to be restarted by another process.

2.8.4 Message Complexity

The modified approach to the Bully algorithm reduces the message complexity from O(n) in the original Bully Algorithm to O(n). In this algorithm, Process P sends an ELECTION message to higher-ID processes, and each alive process responds with an OK message. Process P then selects the highest-ID process and sends a COORDINATOR message to notify the network of the new leader. Since only one election is conducted at a time due to the election flag, and each node responds only once, the message complexity remains O(n), even in the worst case.

2.9 Raft's Consensus Algorithm

Raft is a consensus algorithm designed to manage replicated logs in distributed systems. One of its core components is *leader election*, which ensures that the system maintains a single authoritative leader at any time. This leader is responsible for managing client requests and replicating log entries across servers. Rafts election process is designed to be efficient, easy to understand, and resilient to failures, using a randomized approach to avoid issues like split votes and ensuring a consistent state among servers [11].

2.9.1 Leader Election Process

Raft operates by having servers transition through three possible states: *follower*, *candidate*, and *leader*. In the context of leader election, the process starts when a follower fails to receive a heartbeat from an active leader within a specified time interval, known as the *election timeout.* Upon this timeout, the follower transitions into a *candidate* state, incrementing its term number and initiating a new election by requesting votes from other servers in the cluster [11].

2.9.2 Voting Process

A candidate requests votes by sending `RequestVote` messages to all other servers. Each server can cast one vote per term, and it will vote for the first candidate whose log is at least as up-to-date as its own. If the candidate receives votes from a majority of the servers, it becomes the new leader. Once elected, the leader sends `AppendEntries` messages to maintain its leadership and prevent other servers from initiating new elections [11].

2.9.3 Randomized Election Timeout

To prevent split votes, where multiple candidates receive votes from different subsets of servers without reaching a majority, Raft employs a randomized election timeout. Each server selects a timeout value independently and uniformly from a fixed interval (e.g., 150-300ms). This staggering of timeouts ensures that in most cases, only one server will timeout and begin an election, reducing the chance of multiple simultaneous candidates and split votes [11].

2.9.4 Handling Split Votes

In the rare case that a split vote does occur, where no candidate receives a majority, Raft allows the election process to repeat. Each candidate waits for a new randomized timeout before initiating the next round of `RequestVote` RPCs. Over successive rounds, one candidate typically gains a majority as timeouts are randomized, thus resolving the election quickly and electing a leader efficiently [11].

2.9.5 Pros

- **Clarity and Simplicity**: Raft is designed to be easier to understand by breaking the consensus process into clear stages.
- **Efficient Elections**: The use of randomized timeouts ensures fast leader election, minimizing downtime during leader transitions.

2.9.6 Cons

- **Higher Dependency on the Leader**: If the leader fails, there is a brief period of unavailability during the election process, though mitigated by Raft's quick election mechanism.
- **Not Byzantine Fault Tolerant**: Raft assumes non-Byzantine failures which may limit its use.

2.9.7 Complexity

Rafts algorithm runs in $O(n)$ time complexity, where n is the number of nodes, since it only requires communication with a majority of servers to commit log entries and elect a leader. Its communication cost remains linear, making it scalable for distributed systems [11].

3 Conclusion

References

[1] A. Biswas and A. Dutta, "A timer based leader election algorithm," in *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*, 2016, pp. 432–439.

[2] M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 3rd ed., 2018, (Contains minor corrections in comparison to 3.01).

[3] M. EffatParvar, N. Yazdani, M. EffatParvar, A. Dadlani, and A. Khonsari, "Improved algorithms for leader election in distributed systems," in *2010 2nd International Conference on Computer Engineering and Technology*, vol. 2, 2010, pp. V2–6–V2–10.

[4] D. K. Yadav, C. S. Lamba, and S. Shukla, "A new approach of leader election in distributed system," in *2012 CSI Sixth International Conference on Software Engineering (CONSEG)*, 2012, pp. 1–7.

[5] X. D. Wang and Y. J. Wu, "An improved heapsort algorithm with nlogn 0.788928n comparisons in the worst case," *Journal of Computer Science and Technology*, vol. 22, no. 6, pp. 898–903, 2007. [Online]. Available: https://doi.org/10.1007/s11390-007-9106-7

[6] V. Sharma, P. S. Sandhu, S. Singh, and B. Saini, "Analysis of modified heap sort algorithm on different environment," *World Academy of Science, Engineering and Technology*, vol. 42, 2008.

[7] G. H. Gonnet and J. I. Munro, "Heaps on heaps," *SIAM Journal on Computing*, vol. 15, no. 6, pp. 964–971, 1986.

[8] H. Garcia-Molina, "Elections in a distributed computing system," *IEEE Transactions on Computers*, vol. C-31, no. 1, pp. 48–59, Jan 1982.

[9] M. Gholipour, M. Kordafshari, M. Jahanshahi, and A. Rahmani, "A new approach for election algorithm in distributed systems," in *2009 Second International Conference on Communication Theory, Reliability, and Quality of Service*. IEEE, 2009, pp. 70–74.

[10] P. B. Soundarabai, R. Sahai, K. Venugopal, L. Patnaik *et al.*, "Improved bully election algorithm for distributed systems," *arXiv preprint arXiv:1403.3255*, 2014.

[11] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 305–319.