Chapter 2

Preliminaries and Literature Review

In this chapter, first, we provide some preliminaries regarding the leader election problem in the distributed systems that are important to understand the leader election algorithms. Then we perform an in-detailed literature survey on the leader election problem and identify the several issues and research gaps related to the existing leader election methods.

2.1 Preliminaries

In a distributed system, electing a node as the system leader among the multiple nodes is known as the leader election problem. Here, the leader election algorithms are used to elect the leader for the systems. A leader election algorithm is a distributed algorithm that runs on the different nodes simultaneously and elects the leader. A self-stabilizing leader election algorithm helps to maintain system consistency and improve the overall system performance. In this section, we describe the self-stabilizing conditions, several system models, failure detectors, failure models, and other related matters regarding the design of the leader election algorithms.

2.1.1 Self-stabilizing Conditions

A leader election algorithm is said to be self-stabilizing algorithm if it satisfies the *uniqueness, agreement,* and *termination* conditions. Formally, uniqueness, agreement, and termination guarantee the following properties:

Uniqueness: The algorithm satisfies the uniqueness condition if it always elects only one node as the system leader.

Agreement: The algorithm satisfies the agreement condition if all the nodes agree with the elected leader.

Termination: An algorithm satisfies the termination condition if it is terminated in a finite time.

2.1.2 Failure Detection

In a distributed system, at any time, different system components may fail and hamper the system performance, so failure detection is essential here. A failure detector is a scheme that helps to detect the failure of the components of a system. Every node of a distributed system can have a local failure detection module [27] [17] that works as a failure detector. Each node monitors the other nodes in the system through its local failure detection module. The failure detection module is independent, which means that to invoke this module, a node does not need to depend on the other nodes. There are different approaches to implement a failure detector [73] [4] [5] [4] [72]. Some of these approaches are:

Polling: When a node i monitors another node j through polling, i sends a query message to j and stays waiting for an answer to this message from j [73] [74]. If i does not get an answer after a given time, i will suspect j, i.e., i considers j as a faulty node.

Heartbeat: In the case of the heartbeat method, it is frequently used to monitor the availability of nodes [4] [5]. Every supervised node j sends a heartbeat message periodically to monitoring nodes to notify them that it is still alive. If the monitoring nodes do not receive the expected message from j after a given time, then they will suspect j as a faulty node.

2.1.3 System Models for Leader Election

A system model is a collection of assumptions that allow defining characteristics and constraints that a system might exhibit, such as processing time, communication delay, and failure patterns. One of the critical tasks to design an algorithm is defining the system model in which the algorithm must work. Consequently, to execute an algorithm in a real system, it is important to know if that real system meets the properties of the system model for which it was designed. A distributed system can be modelled by describing its desirable behaviours. Different system models can be obtained based on the time assumptions (clock, processing time, and communication delay) of the distributed systems. These system models are considered to design several distributed algorithms like the leader election algorithm. Some of the distributed system models are as follows.

Synchronous model: A synchronous model [55] always follows the explicit bounds of time. A synchronous distributed system is based on strong boundaries on message delay and computation time. It meets the following properties:

- The execution time of each step in a process is known and confined. There are upper and lower time bounds.
- Every sent message is received within a known limited time.
- The deviation of the clocks of a system is bounded and known.

Asynchronous model: An asynchronous system [44] [34] does not follow any time assumption. An asynchronous distributed system does not depend on the bound of the message transmission delay and computational delay, which means there is no upper bound of message transmission delay and computational delay.

Here coordination is achieved using event-driven architecture triggered by network packet arrival, transitions of signals, handshake protocols, and other methods. Some characteristics of an asynchronous distributed system are:

- There is no concept of global clock and the deviation of the clocks of a system is not bounded.
- A sent message can not be received within a known limited time.
- The execution time of each step in a process is unknown and not confined.

Partially synchronous model: In a partially synchronous system [78], the components have some information about time, although the information might not be exact. A distributed system is partially synchronous if there are bounds on transmission delay, clock drift, and processing time, but these bounds can be unknown to the nodes. Basically, we provide some relaxation on the assumptions of the synchronous system model to build a partially synchronous system model, which makes the system more realistic.

Partially asynchronous model: In a fully asynchronous system, it is impossible to design distributed algorithms with safety and liveness properties. That is why, in a partially asynchronous system model [80] [109], some assumptions are considered over an asynchronous system model that helps build distributed algorithms with safety and liveness properties. The main feature of this system model is the absence of a global timing reference and the use of several distinct local clocks, possibly running at different frequencies.

2.1.4 Failure Models

Distributed systems have the partial failure property. That is, part of the system can fail while the rest continues to work. Partial failures are not at all rare. Properly designed applications and algorithms must take them into account. Several failure models associated with the distributed systems are as follows.

Crash-stop failure model: In the crash-stop failure model [24], also known as the crash failure model, the node that suffers a crash failure stops the execution of its algorithm, and hence it does not send and receive messages ever again. Basically, in this model, a faulty node acts as a correct one before it crashes, and after crashing, it remains inactive forever.

Omission failure model: An omission failure [24] occurs when a node does not send or receive a message that it is supposed to send or receive. In general, the cause of omission failure is network congestion, buffer overflow, transmitter malfunction, collisions at the MAC layer, and receiver out of range. An omission failure can be classified into send-omission and receive-omission. A node i suffers a send-omission failure if it executes a send-message instruction, but the message never reaches the link. On the other hand, a node i suffers a receive-omission when a message is received at its destination, but the message is never delivered inside this node.

Crash-recovery failure model: In a crash-recovery failure model [24], a distributed system component such as node, links, and process may fail and recover from its failure state at any time. When a crashed process recovers, it can affect its ability to remember the previous it had before crashing, i.e., it may lose all pre-crash information, so it should have to start from scratch after recovering. Sometimes, processes are associated with permanent storage, a particular type of memory that holds stored information even if a crash happens. Therefore, a crashed and recovered process could subsequently get essential information to recover its operational state correctly.

Arbitrary failure model: In an Arbitrary failure model [70], a node responds with different responses when different parts of the system communicate with it. It is also known as Byzantine failure. In this particular kind of failure, a node could respond one way when one part of the system talks to it and might respond a completely different way with another part of the system attempts to communicate with it. In other words, a node can respond with arbitrary messages at completely arbitrary times.

2.2 Literature Review

In this section, we study several existing leader election algorithms to perform a literature survey and try to find the associated issues with these existing algorithms. While doing that, to ease out the survey process, we first divide the existing LEAs into two categories based on network topology. One, algorithms for regular network topology, and two, algorithms for arbitrary network topology.

2.2.1 Algorithms for Regular Network Topology

A good number of leader election algorithms are designed considering different regular network topologies. Here we concentrate on studying the LEAs designed for ring, mesh, and torus network topologies because these are very popular regular network topologies for designing distributed systems.

2.2.1.1 Algorithms for Ring Network

The first leader election algorithm was designed for the ring network. In 1977, G. Le Lann proposed this algorithm which is also known as the Ring algorithm [75]. It works on a asynchronous and unidirectional ring network. Every node in the system has a unique Id, and all the system nodes are organized as a logical ring. If a node i realizes that the leader is failed, it generates an election message that contains its node Id and transmits it to the next alive node. Every node attaches its node Id to the message and forwards it to the subsequent node. When the message comes back to i, it chooses the node with the highest node Id in the list as the new system leader. Finally, node i circulates an announcement message and informs all the nodes about the elected leader. This algorithm's message overhead and time overhead are very high, and this algorithm elects the system leader based on the Ids of the nodes. So it may not elect a good quality leader for the system.

In 1979, Chang *et al.* improved the Ring algorithm [75] and proposed the LCR algorithm [28]. This algorithm also works on the unidirectional logical ring network. After realizing the necessity of a leader in this algorithm, a node sends its node Id to other nodes in the ring to initiate the election. While a node gets the election message, it compares the received node Id to its self-Id. If the received node Id is higher than its self-Id, the node keeps on passing the received election message; if the received Id is less than its self-Id, it discards the received message; if the received Id is equal to its self-Id, the node declares itself as the new system leader using a leader declaration message. According to this algorithm, the only node with the highest Id outputs the leader. Though this algorithm reduces the message overhead of the election process, its time overhead is high, and it cannot elect a good quality leader for the system.

Hirschberg and Sinclair proposed the HS algorithm [59] for bidirectional ring network. This algorithm uses a unique Id to identify each node individually. Here the leader election process gets operated in phases i.e., $0, 1, 2, \ldots$. In each phase p, the election initiating node say i sends a message containing its Id in both directions of the ring. These messages are intended to travel a $2p$ distance, then return to their originating node. If both messages come back safely, the node i continues with the next phase. However, the messages might not come back safely. While a message is proceeding in the outbound direction, each other node compares the incoming Id to its self-node Id. If the received Id is less than its self-node Id, then the node discards the received message, whereas if the received Id is greater than its self-node Id, then it relays the message. If a node receives the same Id from both directions of the ring, the node selects the received Id as the new system leader and declares the elected leader using a declaration message. This algorithm reduces the time complexity of the election process from $O(N^2)$ to $O(N \log N)$, but its time complexity is very high. This algorithm also cannot elect a good quality leader for the system.

Biswas and Dutta proposed the Timer-based Leader Election Algorithm [19] for the unidirectional synchronous logical ring network. This algorithm needs a unique Id to identify every node individually. According to this algorithm, the node with the highest Id becomes the system leader among the nodes which realize the leader failure. This algorithm introduces the timer concept with the leader election algorithm that helps to reduce the number of required exchanged messages and time steps to elect a leader. Here the authors also proof the correctness of the proposed algorithm using Hoare logic. Based on the node Ids, this algorithm selects a system leader. It does not consider any quality attribute of the nodes to elect the leader. So this algorithm also cannot elect a good quality leader for the system.

In [20], T. Biswas *et al.* proposed an algorithm to elect a suitable leader for a unidirectional ring network. Every node in the system computes resource strength values by considering available resources like CPU, memory capacity, and residual energy. A node with the highest resource strength over the network is elected as the leader. Though the authors claim that their algorithm improves the message overhead of the election process, it is a minor improvement.

After studying several existing LEAs designed for ring network [1] [120] [20] [19] [59] [75] [28], we find that no leader election algorithm has been designed considering the failure-recovery failure model for the ring network to elect a reliable leader. Most of the existing algorithms considered that node or link failures or recovery do not occur during the election. However, in a practical scenario, node or link failures or recovery can occur during the election.

2.2.1.2 Algorithms for Mesh Network

In 1982, Gracia Molina introduced the Bully algorithm [50] that works on a complete mesh network. According to this algorithm, every node has a unique Id. If a node i determines that the current leader is down, i creates and sends an election message to only the nodes that have a higher Id than i. The node i expects response messages from them if they are alive. If node i does not get any response message from the node with a higher Id, it wins the election and broadcasts a victory message to inform the other nodes that i is the new system leader. If i gets a response from at least one node with a higher Id, i waits a fixed amount of time for any node with a higher Id to declare itself as the new system leader. If i does not get any victory message from a node with a higher Id in time, it rebroadcasts the election message. In this algorithm, when at a time all the nodes realize that the leader is failed, all of them initiate the election simultaneously. Therefore, heavy traffic is created in the network.

Kordafshari *et al.* presented the Modified Bully algorithm [64] to overcome the drawbacks of the Bully algorithm [50]. If a node i notices that the leader is failed, it initiates an election and sends an election message to all nodes with a higher Id. The node with the higher Id sends a response message with its Id number to the node i. When the node i receives all the response messages, it selects the node with the highest Id as the new system leader and transmits the grant message to the highest node. After that, the leader node broadcasts a message and informs all other nodes about the newly elected leader. If no node with a higher Id responses, the node i wins the election and becomes the new system leader. The message overhead of this algorithm is also pretty high.

In [103], G. Singh studied the leader election problem through a new leader election algorithm. This algorithm was designed for the complete mesh networks considering the presence of intermittent link failures. The algorithm can tolerate $(N^2/4 - N/2)$ links failure and its message complexity is $O(N^2)$, where N is the total nodes in the system.

Abu-Amara proposed a fault-tolerant distributed algorithm [3] for an asynchronous complete network to elect a leader. Kutten *et al.* propounded a randomized election algorithm [68] for asynchronous complete networks which are essentially singularly optimal. In [51], Gilbert *et al.* designed a leader election algorithm for a well-connected network. This algorithm can solve the implicit version of leader election in any general network with $O(\sqrt{n}\log^{7/2} n.t_m)$ messages and in $O(t_m \log^2 n)$ time (where n refers to the number of nodes and t_m is the mixing time of a random walk in the network).

2.2.1.3 Algorithms for 2D Torus Network

Only two leader election algorithms have been designed for the 2D torus network. In 2010, M. Refai *et al.* [94] proposed a new algorithm named "Leader Election Algorithm in $2D$ Torus Networks with the Presence of One Link Failure" that works on a $2D$ torus network and can tolerate a single link failure [1]. According to this algorithm, a node can have one of the three states (i.e., Normal, Candidate, or Leader) during the election. This algorithm consists of four phases. The first phase starts when the leader crashes. The node that detects the leader failure informs all the nodes in its row of the failure event. After getting the leader crash information, a node changes its state from normal to candidate state. In the second phase, the candidate state nodes start the election process among the nodes in their column. After the column election, the result (the highest node Id of the column) is sent to the first node in each column. In the third phase, the nodes in the first row of the $2D$ torus start the election among themselves, and one of the nodes herein is elected as a new system leader. In the fourth phase, a declaration message is broadcast (using row broadcasting and column broadcasting) to make all the nodes aware of the system's newly elected leader. Here, the authors claim that in the worst case, the message complexity of this algorithm is $O(N)$. However, we got the worst-case message complexity of this algorithm [94] as $O(N\sqrt{N})$ in our calculation (Our calculation details are given at the end of this section).

In 2014, M. Refai improved the above-discussed algorithm [94] and proposed the "Dynamic Leader Election Algorithm in 2D Torus Network with Multi Links Failure" [8] considering the multiple link failures of the $2D$ torus network [2]. This algorithm [8] has four phases, and the election process is the same as the above-discussed algorithm [94]. In [8], the author improved the algorithm by introducing a detour technique to make the algorithm multiple link failures tolerable. Here, four different

[1]In the next sections, "Leader Election Algorithm in $2D$ Torus Networks with the Presence of One Link Failure" is represented as "LEA with One Link Failure"

[2]In the next sections, "Dynamic Leader Election Algorithm in 2D Torus Network with Multi Links Failure" is represented as "Dynamic LEA with Multi Links Failure"

detour routings are mentioned for four individual link failures of a node. For every individual link failure, the corresponding detour routing is shown in Table 2.1.

Table 2.1: The corresponding detour routing for each link failure of a node

The failed link	The detour routing
+y (Up)	+x (Right), +y (Up), -x (Left)
+x (Right)	+y (Up), +x (Right), -y (Lower)
-y (Lower)	+x (Right), -y (Lower), -x (Left)
-x (Left)	+y (Up), -x (Left), -y (Lower)

From Table 2.1, it is clear that the detour of +y (Up) link failure of a node depends on the +x (Right) link of that node, and the detour of +x (Right) link failure of a node depends on the +y (Up) link of that node. This is why, if the +y and +x links of a node fail simultaneously, this algorithm [8] cannot elect a system leader. Here, the author also did not consider the node failures and did not mention the failure model precisely. In this algorithm [8], the author claims that in the worst case, the message complexity of this algorithm is $O(N + F)$, where F is the number of failed links. Here, the authors also claim that in the worst case, the message complexity of this algorithm is $O(N)$. However, we got the worst-case message complexity of this algorithm [94] as $O(N\sqrt{N})$ in our calculation. We shall now explain how we calculated the worst-case message complexity of these algorithms [94] [8].

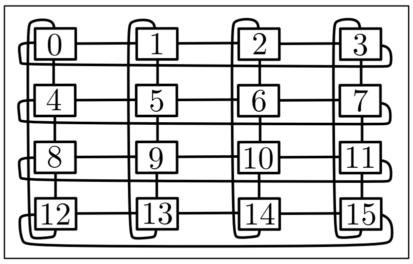

Figure 2.1: A $T_{4,4}$ network, every column of which is arranged in decreasing order according to the node Id.

Each of these two algorithms [94] [8] has four phases. In the first phase, the node that detects the leader's failure informs the other nodes in the same row of the failure by sending messages through its left and right links. In the worst case (when all the nodes detect the failure of the leader simultaneously), $2N$ messages are exchanged in this phase. In the second phase, column-wise election is started, and the node Id of the best node is stored in the node in the first row of that column. If the nodes of every column of the $2D$ torus network are arranged in decreasing order according to their Identification distinguish (Id) (cf. Figure 2.1), then at least $N\sqrt{N} + N$ messages have to be exchanged to complete the second phase of this algorithm. This is because in the second phase, $(N + \sqrt{N})/2$ election messages and $(N + \sqrt{N})/2$ acknowledgement messages need to be exchanged in order to complete the election process within each column. In a square $2D$ torus network, there are $\sqrt{N}$ columns. So, at least $\sqrt{N}(N + \sqrt{N}) = N\sqrt{N} + N$ messages have to be exchanged to complete the second phase of the algorithm. In the third phase, an election is carried out among the best nodes of each column. As the best node Ids are stored in the first row of the network, the leader can be elected by an election among the nodes of the first row by sending messages. In this phase, at least $\sqrt{N}$ election messages and $\sqrt{N}$ acknowledgement messages are required to elect the final leader of the system. In the fourth phase, N messages are exchanged to declare the elected leader. Hence, in the worst case, each of these algorithms has to exchange at least $N\sqrt{N} + 4N + 2\sqrt{N}$ messages to elect a leader. We have also simulate these two algorithms and the simulation results also support that their worst-case message complexity is $O(N\sqrt{N})$.

It is comparatively tough and challenging to design a leader election algorithm considering the crash-recovery model instead of the permanent link and node failure model. Exceptionally, no leader election algorithms have been designed on the 2D torus network considering both the node and link failures and the crash-recovery failure model. The existing election algorithms for 2D torus network can tolerate a few links failure and their message and time overhead are pretty high.

2.2.2 Algorithms for Arbitrary Network

So far, we have studied the leader election algorithms designed for specific network topologies. In this section, we study the election algorithms designed for universal topology or arbitrary network topology.

Mega-Merger [49] [100] is one of the earliest leader election algorithms designed for arbitrary network topology. Robert Gray Gallager developed it at MIT in 1983. It constructs a rooted spanning tree of the network, where the root is the elected leader in the final spanning tree. The rooted spanning trees are merged until a tree that covers the whole network has been constructed. Here merging of two network regions happens in three ways i.e., friendly merge, absorption, and Suspension. In this algorithm it is assumed that no message is lost in transmission. It is deadlock free and it ensures progress and correctly elect a leader. In the worst case, its message complexity is $O(l + N \log N)$ where l is the number of links and N is the number of nodes. The design complexity of Mega-Merger is its main drawback, since it makes any actual implementation difficult to verify.

The Yo-Yo [100] is a leader election algorithm designed for the arbitrary network topology. Unlike the Mega-Merger algorithm, Yo-Yo has simple specifications, and its correctness is simple to establish. The Yo-Yo algorithm consists of two parts i.e., a preprocessing phase and a sequence of iterations. Each iteration is composed of phases "YO-" and "-YO". The preprocessing phase is started with a broadcast. At awake state, each node sends its Id to all neighbors and orients the edge towards the higher-degree node. This process creates three categories of nodes i.e., source nodes, intermediate nodes, and sink nodes. The source nodes and initiate the "Yo-" phase, and sink nodes initiate the "-Yo" phase. In this algorithm, the pruning optimization technique plays an important role. Without pruning, the message complexity of this algorithm is $O(l \log N)$, but with pruning, the complexity analysis is an open research problem.

Recently, Sidik *et al.* proposed a time-bounded practical agile leader election (PALE) algorithm [102] for an arbitrary network topology. Though the authors considered a realistic and practical distributed system with weak assumptions, their assumption regarding clock drift is strong. The authors considered a parameter $maxRatio$ and assumed that the fastest clock in a region is faster than the slowest one by at most a factor of $maxRatio$. The maximum rounds a node in a region needs to execute the algorithm depend on the $maxRatio$. So the performance of the PALE algorithm depends a lot on the $maxRatio$. A node needs to know the $maxRatio$ in advance to execute the PALE algorithm. If $maxRatio$ is set to a small value, the algorithm may not hold uniqueness and agreement properties. On the other hand, if $maxRatio$ is set to a large value, the algorithm will take an unnecessarily long time and exchange extra messages to elect the leader. The authors considered a dynamic system where nodes can leave or join the system frequently while selecting the perfect value of $maxRatio$ is quite tricky. The authors mentioned that a node with the highest rank is elected as a leader, and the node's rank depends on RAM capacity, storage capacity, computing, power and stability counter. However, the authors did not detail how the rank will be calculated.

In [21], Blin and Tixeuil proposed a self-stabilizing leader election algorithm for arbitrary networks with $O(\log d + \log\log n)$ bits per node space complexity, where d is the maximum degree of a node in the network. Dolev *et al.* introduced uniform dynamic self-stabilizing leader election algorithm [38] under read-write atomicity. The time complexity of this is $O(dD \log n)$. Vasudevan *et al.* propounded an election algorithm [114] for mobile ad hoc networks which is partially synchronous. The authors showed that their proposed algorithm is weakly self-stabilizing and terminating. In [79], Malpani *et al.* presented two election algorithms for mobile ad hoc networks. One of them can tolerate a single topological change, and another can tolerate multiple concurrent topological changes. Derhab and Badache presented the time-interval-based computation concept and designed a self-stabilizing leader election algorithm [37] to tolerate multiple concurrent topological changes. In [106], Sudo *et al.* proposed two election protocols for an arbitrary graph in the population protocol model. These two election protocols are loosely stabilizing and guarantee the polynomial convergence time to enter a safe configuration.

After studying several existing leader election algorithms, we found that most algorithms are designed to reduce the time complexity, message complexity, and bit or space complexity of the election process. Some of the algorithms are designed considering the fairness and link and node failures of the system. No leader election algorithm is designed considering the requirement of a distributed real-time system. Several well-known leader election algorithms talked about electing a good quality leader (according to the system requirements) [104] [20] [102]. However, no precise and rigorous leader election method has been proposed to elect a good quality leader based on the system requirements.