Chapter 3

FRLLE: A Failure Rate and Load-based Leader Election Algorithm for a Ring Network

This chapter addresses the leader election problem by proposing a new leader election algorithm (FRLLE) for bidirectional ring networks. The proposed algorithm elects a node with a minimum failure rate and load so that the system gets a more reliable leader that can concentrate on leadership roles and activities comfortably. This algorithm satisfies uniqueness, agreement, and termination conditions, making it a self-stabilizing leader election algorithm that helps build an efficient and consistent distributed system. It reduces the message and time complexity of the election process, which means the algorithm takes fewer time steps and exchanges fewer messages to elect a leader. We simulate the FRLLE algorithm considering the different number of nodes, compare the simulation results of the FRLLE algorithm with the well-known existing leader election algorithms and demonstrate that the FRLLE algorithm exchanges fewer messages and takes fewer time steps to elect the leader. We further carried out a priori complexity analysis and compared the outcome with the simulation results to corroborate our proposal.

Outline: The rest of the chapter is organized as follows. Section 3.1 details the system model, assumptions and definitions. Section 3.2 presents the proposed leader election algorithm. Complexity analysis, proof of self-stabilizing conditions and an illustrative example are also give in this section. The empirical appraisement and comparison with other existing algorithms are presented in Section 3.3. Section 3.4 summaries the chapter.

3.1 System Model

We consider a crash-recovery distributed system [24] composed of N nodes, where N is an finite integer and $N \geq 2$. The nodes are connected through a partially synchronous bidirectional logical ring network and they communicate with one another through message passing [23] and the message transmission delay follows an upper bound. The communication channels of the system could be unreliable. Formally, this system can be represented by a graph $G = (\Pi, L)$; where Π is the set of nodes ($|\Pi| = N$ and $\Pi = \{N\_Id_0, N\_Id_1, N\_Id_2, \ldots, N\_Id_{N-1}\}$, and L is the set of network connections (links) between the nodes ($L = \{l_{ij}\}$, where, l_{ij} is the link between N_Id_i and N_Id_j; $N\_Id_i \neq N\_Id_j$ and $N\_Id_i, N\_Id_j \in \Pi$). According to the crash-recovery model, a node may fail and get recovered anytime. Failure of a node occurs when the node stops functioning correctly, and the recovery of a node is a process that involves restoring the node from an erroneous state to an error-free state. We also consider the omission fault, because it is a kind of failure-recovery fault, with one exception [24]. An omission fault ensues when a node does not send or receive a message that it is supposed to send or receive. In general, omission faults ensue due to buffering overflows or network congestion.

3.1.1 Assumptions

The following assumptions are considered for the work described in this chapter.

- Every node has a unique Id. For the sack of simplicity, we assume that the Id of a node belongs to 0 to $N-1$.
- Every node knows the minimum propagation delay of a message between the node and its adjacent nodes.

3.1.2 Definitions

The following definitions are provided for clarity regarding their usage in the rest of this chapter.

Definition 1 (Fr_i): It represents the failure rate [84] [48] [43] of a node i. The failure rate is the frequency with which a system, a node or a component fails, expressed in failures per unit of time. In general, Weibull distribution [14] [95] is used for failure analysis because of its flexibility in describing failure rate as it can represent all three regions of the bathtub curve. According to two-parameter Weibull distribution, the unreliability $F_{X_i}(t)$ and the failure density function $f_{X_i}(t)$ of a node can be written as

$$F_{X_i}(t) = 1 - e^{-\left(\frac{t}{\eta_i}\right)^{\beta_i}} \tag{3.1}$$

where $\eta_i > 0$ is called the scale parameter, $\beta_i > 0$ is the shape parameter, and t denotes time.

$$f_{X_i}(t) = \frac{d}{dt}F_{X_i}(t) = \frac{\beta_i}{\eta_i}\left(\frac{t}{\eta_i}\right)^{(\beta_i-1)} e^{-\left(\frac{t}{\eta_i}\right)^{\beta_i}} \tag{3.2}$$

So, the failure rate of a node i between times t_0 and t_1 is

$$Fr_i = \int_{t_0}^{t_1} f_{X_i}(t)dt \tag{3.3}$$

Here, $\beta_i < 1$ means the failure rate is a decreasing function of time; $\beta_i = 1$ means the failure rate is constant, and $\beta_i > 1$ means the failure rate is an increasing function of time.

Definition 2 (P_i): It indicates the message processing delay of the i^{th} node. The message processing delay implies the time that a node takes to process the election message.

Definition 3 (P_{ij}): It represents the minimum message propagation delay between i^{th} and j^{th} nodes. The propagation delay refers the time taken by an election message to propagate from the i^{th} node to the j^{th} node.

Definition 4 (Lrt_i): It refers to the time-stamp when the i^{th} node got the last response from the leader. That means every node stores the up-to-date response time-stamp of the leader in its Lrt_i.

Definition 5 (Ttd_i): It represents the sum of the propagation delay (P_{ij}) and the processing delay (P_i) of an election message.

Suppose i, j and k are three subsequent nodes in the ring (cf. Figure 3.1). Node i initiates the election by creating an election message and sends it to the node j. Initially $Ttd_i = P_{ij} + P_i$ and Ttd_i is sent to node j through the election message. Before sending the received election message to node k, node j modifies the total time delay as follows: $Ttd_j = Ttd_i + P_{jk} + P_j$.

Definition 6 ($Emat_i$): It implies the time-stamp of an election message when it arrives at the node i.

Definition 7 (*Recovery condition*): Recovery condition ($Emat_j - Ttd_i < Lrt_j$) is the condition that helps to identify whether the recently crashed leader has recovered from its failure state.

Suppose i and j are two subsequent nodes in the ring (cf. Figure 3.1). Node j gets an election message from node i. If ($Emat_j - Ttd_i < Lrt_j$) is satisfied, that means the node j has got respond from the leader after the election initiation. Thus, the

leader is alive, which means the recently failed leader has recovered from its failure state. If $(Emat_j - Ttd_i < Lrt_j)$ is not satisfied, then the node j has no information about the recovery of the crashed leader.

Definition 8 (Lc_i): It represents the leader coefficient of a node i. We calculate the leader coefficient of a node by considering its average CPU utilization (A_Cu_i), average memory utilization (A_Mu_i), average bandwidth utilization(A_Bu_i) and its failure rate (Fr_i). The following formula is used to compute the value of Lc_i.

$$Lc_i = w(A\_Cu_i) + x(A\_Mu_i) + y(A\_Bu_i) + z(Fr_i) \tag{3.4}$$

where, $w + x + y + z = 1$ and $w, x, y, z \geq 0$

We can directly measure the average CPU, memory, and bandwidth utilization of a node using some commands. According to our requirement, we can take the different values of w, x, y, and z to give the different priorities of the average CPU utilization, memory utilization, bandwidth utilization, and the failure rate. The smaller value of the leader coefficient of a node indicates the higher potentiality of the leadership.

Definition 9 (M_Lc_i): It represents the minimum leader coefficient of a node maintained by the i^{th} node. Initially the i^{th} node stores its self leader coefficient into M_Lc_i. After that, the node i updates M_Lc_i depending upon the received election messages. If the leader coefficient of the received message creator node is less than M_Lc_i, the node i updates M_Lc_i by the leader coefficient of the received message creator node.

Definition 10 (Ein_i): If a node initiates the election by creating an election message, then the node is an election initiating node (election initiator). If the i^{th} node is an election initiator then $Ein_i = True$. If the i^{th} node is not an election initiating node, $Ein_i = False$.

Definition 11 ($Frem_i$): It is used to identify whether the i^{th} node receives an election message for the first time during the election. If the i^{th} node gets an election

message for the first time from any other node, then $Frem_i = True$, otherwise $Frem_i = False$.

3.1.3 Types of Message

The proposed algorithm uses three types of messages to elect the system leader. One of these is the election message, which has four fields, i.e., the election message creator node Id (Emc_Id), the recently failed leader Id (Fl_Id), the leader coefficient (Lcc) of the message creator node and the total time delay (Ttd_i) of the election message. The election message is used to initiate the election. It is represented by $msg[Emc\_Id, Fl\_Id, Ttd_i, Lcc]$. The second type of message is the leader recovery message, which has a single field i.e., the recently failed leader Id (Fl_Id). This message is used to inform the other nodes about the recovery of the recently failed leader during the election. This message is represented by $lrmsg[Fl\_Id]$. The third type of message is the newly elected leader declaration message, which also has a single field named newly elected leader Id (El_Id). This message is used to declare the newly elected leader, and it is represented by $nldmsg[El\_Id]$.

3.2 Proposed Algorithm

The proposed leader election algorithm (FRLLE) elects one of the system's potential nodes. The potential of a node is measured in terms of the failure rate and load of the node. As we consider a crash-recovery distributed system, so a node may fail and get recovered anytime. Hence, the recently failed leader may or may not get recovered during the election. The FRLLE algorithm is designed considering both the cases. Firstly, our algorithm checks the recovery condition to identify whether the leader is recovered. If the leader is recovered, there is no need to elect a new leader. On the other hand, if the leader is not recovered, the algorithm elects the node with the minimum leader coefficient among the existing nodes in the system as the new leader. If more than one node has the minimum leader coefficient, the

algorithm selects the node with the maximum Id among them as the leader. Now, we describe the FRLLE algorithm in detail. In this algorithm, we use two functions i.e., $Create\_ele\_msg()$ and $Modify\_msg()$. Function $Create\_ele\_msg()$ is used to create the election message, and $Modify\_msg()$ is used to modify the received election message. We describe functions $Create\_ele\_msg()$ and $Modify\_msg()$ below by considering that h, i, j and k are subsequent nodes in a ring (cf. Figure 3.1).

***Create_ele_msg*($Emc\_Id, Fl\_Id, Lcc, Ttd_i$)**

`// When a node` i `calls this function.`

1. Create an election message $msg[Emc\_Id, Fl\_Id, Ttd_i, Lcc]$ and send it in both directions of the ring.
2. $N\_Id\_Mlc_i \leftarrow Emc\_Id$.
3. $M\_Lc_i \leftarrow Lcc$

***Modify_msg*($Emc\_Id, Fl\_Id, Lcc, Ttd_i$)**

`// When a node` j `gets an election message from a node` i`, the node` j `calls this function to modify and send the message to the adjacent node` k`.`

1. $N\_Id\_Mlc_j \leftarrow Emc\_Id$, $M\_Lc_j \leftarrow Lcc$
2. $Ttd_j \leftarrow (Ttd_i + P_{jk} + P_j)$
3. Forward the modified election message $msg[Emc\_Id, Fl\_Id, Ttd_j, Lcc]$ to the next node.

This algorithm consists of four phases:

- Election initiation
- Election message processing

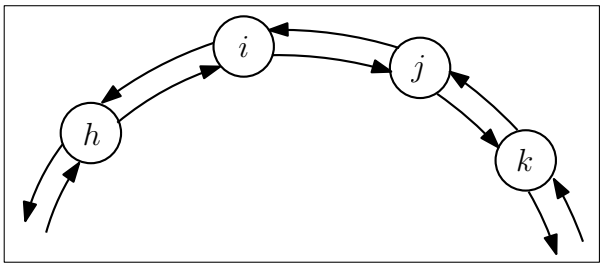


FIGURE 3.1: All the nodes are connected through a bidirectional ring topology. h, i, j and k are four subsequent nodes in this ring.

- Leader recovery message processing
- Newly elected leader declaration

Here, we consider that a node i perceives that the leader is failed, so it initiates an election by creating an election message.

Algorithm 1: Election Initiation

```
// A node i perceives that the leader is failed and creates an election
   message.
1 if (a node i realizes the leader failure) then
2 |   Emc_Id ← N_Id_i
3 |   Lcc ← Lc_i
4 |   Create_ele_msg(Emc_Id, Fl_Id, Lcc, Ttd_i)
5 end
```

Election initiation: This phase (Algorithm 1) is all about how a node initiates an election. Whenever a node realizes that the current system leader is failed, the node creates an election message to initiate the election and sends it in both directions of the ring. Suppose, a node i realizes that the current system leader is failed, so the node i creates the election message $msg[Emc\_Id, Fl\_Id, Ttd_i, Lcc]$, (where $Emc\_Id \leftarrow N\_Id_i$, $Lcc \leftarrow Lc_i$) and sends this message to the nodes j and h. Node i copies self-Id and Lc_i into its $N\_Id\_Mlc_i$ and M_Lc_i respectively.

Election message processing: This phase (Algorithm 2) explains how a node processes a received election message. Suppose, a node j gets an election message from its adjacent node i. The node j first checks the recovery condition ($Emat_j - Ttd_i < Lrt_j$) to know whether it has any information about the recovery of the

Algorithm 2: Election Message Processing

```
// When a node j receives the election message
   msg[Emc_Id, Fl_Id, Ttd_i, Lcc] from a node i.
if (Emat_j − Ttd_i < Lrt_j) then
    Discard the received election message.
    Create a lrmsg[Fl_Id] and send it to node i.
else
    if (M_Lc_j < Lcc) then
        if (Frem_j == True and Ein_j == False) then
            Emc_Id ← N_Id_j, Lcc ← Lc_j
            Create_ele_msg(Emc_Id, Fl_Id, Lcc, Ttd_j)
            Frem_j ← False
        end
        Discard the received message.
    else
        if (M_Lc_j == Lcc) then
            if (Frem_j == True and Ein_j == False) then
                if (N_Id_j > Emc_Id) then
                    Emc_Id ← N_Id_j, Lcc ← Lc_j
                    Create_ele_msg(Emc_Id, Fl_Id, Lcc, Ttd_j)
                else
                    Modify_msg(Emc_Id, Fl_Id, Lcc, Ttd_i).
                end
                Frem_j ← False
            else
                if (N_Id_Mlc_j < Emc_Id) then
                    Modify_msg(Emc_Id, Fl_Id, Lcc, Ttd_i)
                else
                    if (N_Id_Mlc_j == Emc_Id) then
                        Elect the received message creator node as the leader.
                        El_Id ← Emc_Id
                        L_Id_j ← El_Id
                        Create a leader declaration message nldmsg[El_Id] and send it
                         in both directions of the ring.
                    else
                        Discard the received message.
                    end
                end
            end
        else
            Modify_msg(Emc_Id, Fl_Id, Lcc, Ttd_i)
        end
    end
end
```

failed leader. If it is true, that means the leader is alive, so the node discards the received election message, creates a leader recovery message $lrmsg[Fl\_Id]$ and sends it to the adjacent node (i.e., node i) which sent the election message. By sending a leader recovery message, the node j informs that the leader is alive. If the recovery condition is false, the node j compares M_Lc_j with the leader coefficient (Lcc) of the election message creator. If M_Lc_j is greater than Lcc, the node j copies the election message creator Id and Lcc into $N\_Id\_Mlc_j$ and M_Lc_j respectively, calculates the Ttd_j (i.e., $Ttd_j = Ttd_i + P_{jk} + P_j$) and forwards the modified election message ($msg[Emc\_Id, Fl\_Id, Ttd_j, Lcc]$) to the next node (node k). If M_Lc_j is less than Lcc, the node j discards the received election message. Along with that, if the node j itself is not an election initiating node, and the received message is its first received election message, then the node j creates a new election message ($msg[Emc\_Id, Fl\_Id, Ttd_j, Lcc]$, where $Emc\_Id \leftarrow N\_Id_j$ and $Lcc \leftarrow Lc_j$), and sends it in both directions of the ring.

Whenever a node gets the same election message from both directions of the ring, it elects the election message creator node as the new system leader. Then the node creates a newly elected leader declaration message ($nldmsg[El\_Id]$, where $El\_Id \leftarrow Emc\_Id$) and sends it in the both directions of the ring.

Algorithm 3: Leader Recovery Message processing

```
// Suppose a node j receives a leader recovery message lrmsg[Fl_Id] from
   a node k
1  if (the node j itself is a leader recovery message creator node) then
2  |  Discard the received leader recovery message (lrmsg[Fl_Id]).
3  else
4  |  if (received leader recovery message==the first received leader recovery message)
   |   then
5  |  |  L_Id_j ← Fl_Id
6  |  |  Forward the leader recovery message lrmsg[Fl_Id] to the next node.
7  |  else
8  |  |  Discard the received leader recovery message lrmsg[Fl_Id].
9  |  end
10 end
```

Leader recovery message processing: This phase (Algorithm 3) is about the

leader recovery message processing method. Suppose, a node j gets a leader recovery message from a node k. After receiving this message ($lrmsg[Fl\_Id]$), the node j identifies that the recently failed leader gets recovered from its failure state. Therefore, the leader will be the same as earlier and there is no need to elect a new leader. Hence, the L_Id_j should be updated by the previous leader Id (Fl_Id). If the node j itself is not a leader recovery message creator and receives a leader recovery message for the first time, the node forwards the received leader recovery message to the next node. Otherwise, it discards the received leader recovery message.

Algorithm 4: Newly Elected Leader Declaration Message processing

```
// Suppose a node j receives a newly elected leader declaration message
   nldmsg[El_Id] from a node k
if (received leader declaration message==the first received leader declaration
 message) then
    L_Id_j ← El_Id
    Forward the leader declaration message nldmsg[El_Id] to the next node.
else
    Discard the received leader declaration message nldmsg[El_Id].
end
```

Newly elected leader declaration message processing: This phase (Algorithm 4) denotes how a node processes the leader declaration message. Suppose, a node j gets a newly elected leader declaration message ($nldmsg[El\_Id]$) from a node k. After receiving $nldmsg[El\_Id]$ message, the node j copies El_Id into L_Id_j and forwards the message to the next node. In this way, all the nodes know about the newly elected leader. Whenever a node gets the leader declaration message from both directions of the ring, it discards the message.

3.2.1 Proof of Self-stabilization

This section proves that the proposed algorithm satisfies the *uniqueness*, *agreement*, and *termination* conditions that make the algorithm a self-stabilizing leader election algorithm.

Lemma 3.1. *Every proper execution of the FRLLE algorithm elects only one node as the system leader.*

Proof: When the recently failed leader gets recovered from its failure state, the algorithm retains the previous leader instead of electing a new one. So, in this case, there is only one leader in the system. But, when the leader fails, the node with the least leader coefficient among the existing nodes gets elected. Now two cases may occur: (1) all the nodes have different leader coefficients, (2) more than one node has the same leader coefficient. Assume that, at a time, two nodes i and j are elected as the system leader. So, node i has the least leader coefficient (Lc_i), as well as node j also has the least leader coefficient (Lc_j) among the existing nodes in the system. In case (1), all the nodes have different leader coefficients, and at a time nodes i and j are elected as the system leader. So, $Lc_i < Lc_j$ as well as $Lc_j < Lc_i$. This scenario cannot happen because all the nodes have different leader coefficients. Hence, our assumption is wrong. So, the nodes i and j cannot get elected as the system leader concurrently. In case (2), more than one node may have the minimum leader coefficient, and at a time, both nodes i and j are elected as the system leaders. So, nodes i and j have the least leader coefficient, i.e. Lc_i and Lc_j respectively and $Lc_i = Lc_j$. In this case, the propounded algorithm elects the node with maximum Id among the nodes which have the least failure rate. If the node i is elected as the system leader, then $N\_Id_i > N\_Id_j$. If the node j is also elected as the system leader, then $N\_Id_j > N\_Id_i$. According to our system model, every node has a distinct node Id. Hence, at a time $N\_Id_i > N\_Id_j$ and $N\_Id_j > N\_Id_i$ is not possible. So, in this case, our assumption is also wrong. Therefore, at a time, the FRLLE algorithm elects only one node as the system leaders. Hence the uniqueness condition is met here. ■

Lemma 3.2. *In every proper execution of the FRLLE algorithm, all the nodes of the system get to know about the elected leader and agree with it.*

Proof: When a node i gets an election message from both adjacent nodes (both directions of the ring), it gets to know the received election message creator node

is the newly elected leader. The node then stores the newly elected leader Id into its L_Id_i, creates a newly elected leader declaration message ($nldmsg[El\_Id]$), and sends it in both directions of the ring. When a node gets the $nldmsg[El\_Id]$ message, it copies the elected leader Id (El_Id) into its L_Id and forwards the message to the next node. In this way, after $N/2$ hops, all the nodes get the $nldmsg[El\_Id]$ message and know about the newly elected leader. Thus all the nodes copy the elected leader Id into their L_Id and agree with the elected leader. So the FRLLE algorithm satisfies the agreement condition. ■

Lemma 3.3. *The FRLLE algorithm elects a leader for the system and gets terminated in a finite time.*

Proof: An election process starts through creating election message(s) by the node(s). Only the message with the highest leader coefficient retains in the network. When a node gets that message from both directions of the ring, it gets to know who the newly elected leader is, i.e., none other than the node, which created the election message with the highest leader coefficient. Then that receiver node creates a leader declaration message and sends it to both directions of the ring to declare the newly elected leader. Every node will get this leader declaration message. When a node gets this declaration message from both directions, the election message gets terminated. The FRLLE algorithm takes $N/2$ time steps to propagate the election messages and another $N/2$ time steps to propagate the leader declaration message. So, a total of N time steps are required to elect a new leader. In a synchronous system, it takes a finite time to propagate a message from one node to another node. If one time step takes maximum c units of time (where c is constant), then the FRLLE takes maximum cN units of time to complete the whole election process. According to our system model N is finite. Hence, the FRLLE algorithm must terminate in a finite time. ■

Theorem 3.4. *The FRLLE algorithm is a self-stabilizing leader election algorithm.*

Proof: Lemma 3.1, Lemma 3.2 and Lemma 3.3 prove that the FRLLE algorithm satisfies the uniqueness, agreement and termination conditions respectively. That

means the FRLLE algorithm satisfies all the three conditions of a self-stabilizing leader election algorithm. Hence, the FRLLE algorithm is a self-stabilizing leader election algorithm. ■

3.2.2 Complexity Analysis

In this section, we analyze the time complexity and message complexity of the FRLLE algorithm.

3.2.2.1 Message Complexity

In the proposed system model, the nodes communicate with one another through message passing, so message complexity depends on how many messages have been exchanged to elect the leader.

Best Case: Suppose a node realizes that a leader has failed. Meanwhile, its adjacent two nodes realize that the leader is alive. This is the best case scenario of this algorithm. In this case, the former node starts the election process and sends the election messages to its two adjacent nodes. But they already know that the leader is alive. So, after receiving election messages, they create leader recovery messages and send those (two messages) to the former node which initiated the election. So, in this whole process, only four messages are needed. Here, the message complexity becomes $O(1)$.

Worst Case: Suppose all the nodes simultaneously realize that the leader has failed, and it is not recovering. It is the worst-case scenario of the FRLLE algorithm. In this situation, all the nodes start the election process by creating election messages. Among all these messages, the message with the highest leader coefficient retains in the network, and others get discarded. After $(k + 1)^{th}$ steps (where k is an integer), a node gets the election message with the highest leader coefficient from both sides. This is the time when the node gets to know the newly elected leader that is none other than the creator of the message with highest leader coefficient.

Then this node creates a leader declaration message and sends it to the other nodes. If N is odd, then $k = (N-1)/2$ and the total number of required messages is $3 + 5 + 7 + 9 + \ldots\ldots\ldots + k^{th}\ term + 2N + N = (N^2 + 14N - 3)/4$. If N is even then $k = (N-2)/2$ and the total number of required exchanged messages is $4 + 6 + 8 + \ldots\ldots\ldots + k^{th}\ term + 2N + N = (N^2 + 14N - 8)/4$. Therefore, in the worst case, the message complexity of this algorithm is $O(N^2)$.

3.2.2.2 Time Complexity

Time complexity refers to the total time an election algorithm takes to complete the entire election process.

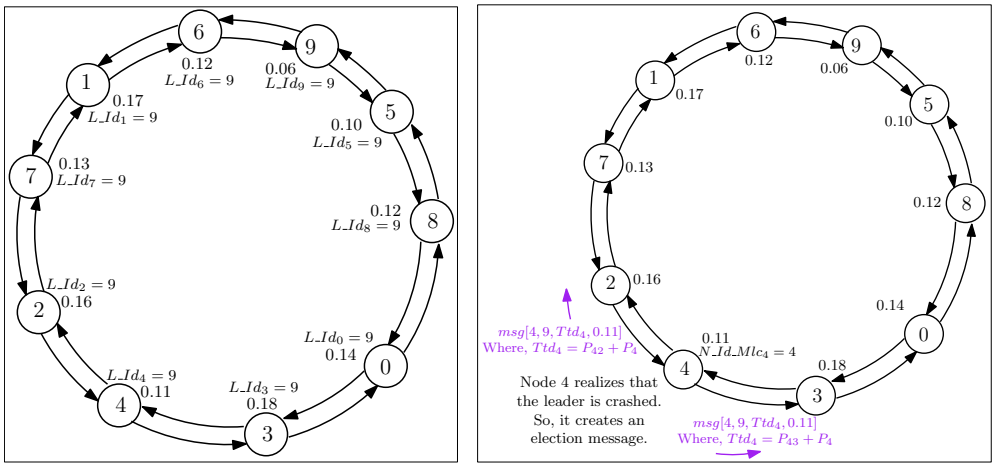
Best Case: In the best case, only four messages are needed to establish that the former leader is alive, and no new leader election is required. This process takes just two time steps. If one time step takes c units of time (where c is constant), then, in the best case, $2c$ units of time are required. Here, the time complexity of the FRLLE algorithm becomes $O(1)$.

Worst Case: In the worst case, the FRLLE algorithm takes $N/2$ time steps to propagate the election messages and $N/2$ time steps to propagate the leader declaration message to elect the new leader. So, a total of N time steps are required to elect a new leader. If one time step takes c units of time (where c is constant), then, in this case, cN units of time are required. Hence, the time complexity is $O(N)$.

3.2.3 An Illustrative Example

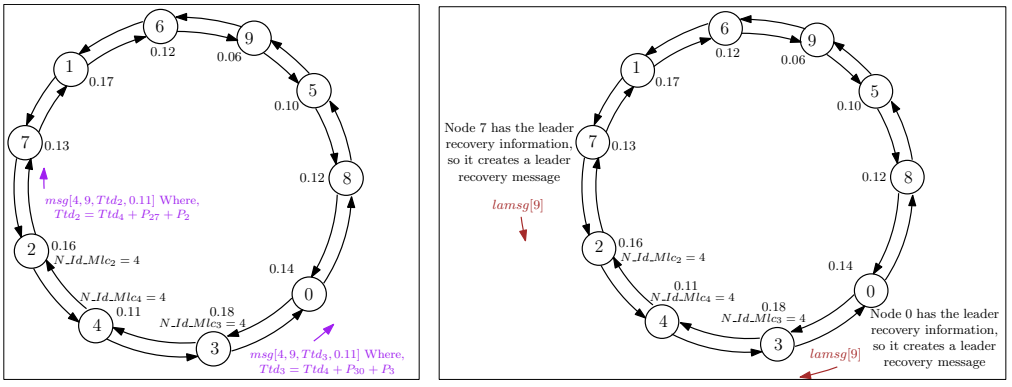
In this section, we illustrate the FRLLE algorithm with the help of an example. First, we detail the case where the recently failed leader gets recovered during the election.

Consider Figure 3.2(a); there are ten nodes in the system with distinct node Ids from 0 to 9. Here, a circle represents a node and integer inside the circle is its Id.



(a) There are 10 nodes in this system with node Ids from 0 to 9. Node 9 is the system leader.

(b) Node 4 perceives that the leader is failed, so it generates an election message to initiate the election.



(c) There are 10 nodes in this system with node Ids from 0 to 9. Node 9 is the system leader.

(d) Node 4 perceives that the leader is failed, so it generates an election message to initiate the election.

FIGURE 3.2: Illustrative example of the proposed election algorithm when failed leader recovers during the election.

The leader coefficient of every node is also shown in this figure. For example, the leader coefficient of node 1 is 0.17. Suppose that the node 9 is the system leader, and the node 4 realizes that the leader is failed, and it creates an election message $msg[4, 9, Ttd_4, 0.11]$ to initiate an election. The node 4 copies its self-Id into its $N\_Id\_Mlc_4$, and sends the election message to the nodes 2 and 3 (cf. Figure 3.2(b)). In between, the node 9 gets recovered from the failure state, and nodes 0 and 7 get the recovery information of node 9.

The nodes 2 and 3 have no recovery information about the recently failed leader

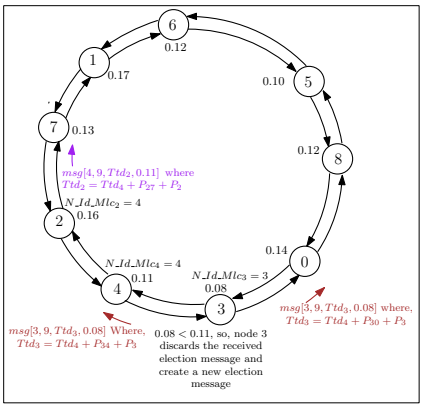
(i.e., node 9), and their leader coefficient is higher than the leader coefficient of node 4. So, after receiving the election message $msg[4, 9, Ttd_4, 0.11]$, nodes 2 and 3 copy the node Id 4 into $N\_Id\_Mlc_2$ and $N\_Id\_Mlc_3$ respectively. They calculate Ttd_2 and Ttd_3, and send the modified election message to nodes 7 and 0 respectively (cf. Figure 3.2(c)).

After receiving the election message created by the node 4, the nodes 7 and 0 check the recovery inequality to identify whether the recently failed leader has been recovered. As the leader is recovered and nodes 7 and 0 have that information, therefore recovery inequality will be satisfied. That's why nodes 7 and 0 discard the received election message, create a leader recovery message $lrmsg[9]$ and send it to the nodes 2 and 3 respectively (cf. Figure 3.2(d)).

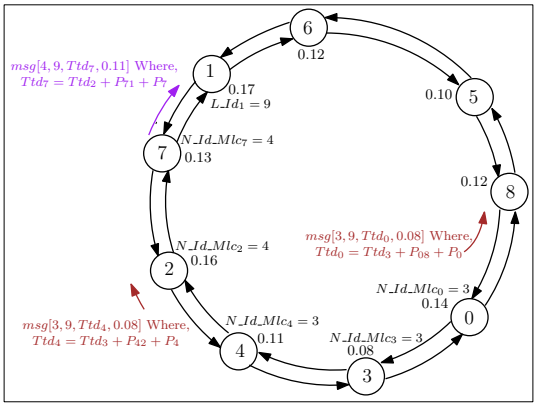
After receiving the leader recovery message, nodes 2 and 3 copy the node Id 9 into LID_2 and LID_3 respectively, and forward the leader recovery message $lrmsg[9]$ to node 4. After receiving the leader recovery message $lrmsg[9]$ the node 4 copies the node Id 9 into its LID_4, and discards both the leader recovery messages received from nodes 2 and 3.

Now we detail the case where the failed leader does not get recovered during the election. In this example, we assume that the node 3 has the minimum leader coefficient, i.e., 0.08. The system leader (node 9) is failed, and the node 4 realizes it. To initiate the election, the node 4 creates an election message $msg[4, 9, Ttd_4, 0.11]$, copies self-Id into its $N\_Id\_Mlc_4$, and sends the election message to the nodes 2 and 3 (cf. Figure 3.2(b)).

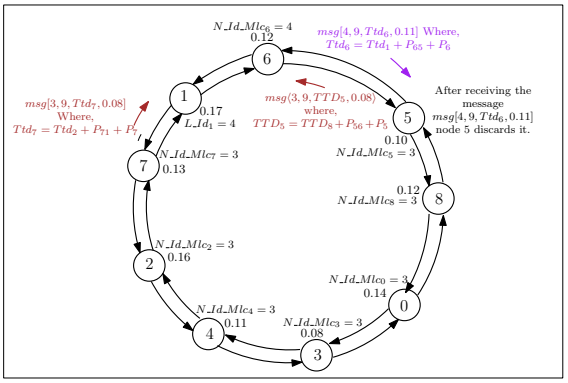
As the recently failed leader does not get recovered during the election, so no node will have the leader recovery information; therefore, recovery inequality will not be satisfied. The leader coefficient of the node 2 is higher than that of the node 4, so after receiving the message $msg[4, 9, Ttd_4, 0.11]$, the node 2 copies the node Id 4 into its $N\_Id\_Mlc_2$, calculates Ttd_2 (where, $Ttd_2 = Ttd_4 + P_{27} + P_2$), forwards the modified message ($msg[4, 9, Ttd_2, 0.11]$) to node 7. When the node 3 receives the message $msg[4, 9, Ttd_4, 0.11]$, it discards this message and creates a new election



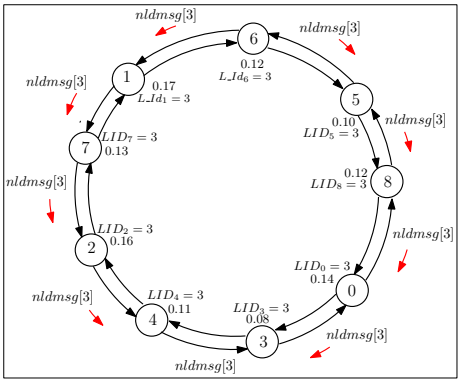
(a) Node 2 copies the node Id 4 and forwards the received message to node 7. Node 3 discards the received election message, creates a new election message, and sends the modified message to node 1.



(b) Nodes 4 and 0 copy the node Id 4, modify the received message and forwards it to the nodes 2 and 8 respectively. Node 7 copies the node Id 4 and forward it to the nodes 4 and 0.



(c) Node 5 discards the received election message (created by node 4) and forwards the election message (created by node 3) to node 6. Node 7 copies node Id 3 and forwards the election message (created by node 3) to node 1.



(d) Node 6 gets the election message (created by node 3) from both directions of the ring, so it creates a leader declaration message and sends the message to all the nodes to declare the node 3 as the newly elected leader of the system.

FIGURE 3.3: Illustrative example of the proposed election algorithm when failed leader does not recover during the election.

message $msg[3, 9, Ttd_3, 0.08]$ because its leader coefficient is less than that of node 4. It sends the newly created message to nodes 0 and 4, copies self-Id into its $N\_Id\_Mlc_3$ (cf. Figure 3.3(a)).

After receiving the message $msg[3, 9, Ttd_3, 0.08]$, the node 4 copies the node Id 3 into its $N\_Id\_Mlc_4$ and forwards the message to the node 2 because the node 3 has the minimum leader coefficient. At the same time, node 7 gets the message

$msg[4, 9, Ttd_2, 0.11]$. It copies node Id 4 into its $N\_Id\_Mlc_7$, calculates Ttd_7, forwards the modified message, i.e. $msg[4, 9, Ttd_7, 0.11]$ to node 1. When node 0 gets the message $msg[3, 9, Ttd_3, 0.08]$, it copies node Id 3 into its $N\_Id\_Mlc_0$, calculates Ttd_0, forwards $msg[3, 9, Ttd_0, 0.08]$ to the next node i.e., node 8 (cf. Figure 3.3(b)).

After getting the election message created by the node 4, nodes 1 and 6 perform the same procedure as node 7. Likewise, after receiving the message created by the node 3, the nodes 8 and 5 follow the same procedure as node 0. When the message $msg[4, 9, Ttd_6, 0.11]$ comes to the node 5, it discards this message because the node 5 has already got the election message that is created by the node 3, which has the minimum failure rate. On the other hand, after receiving the election message which is created by the node 3, the nodes 2 and 7 copy the node Id 3 and forward the message to the next node (cf. Figure 3.3(c)).

Finally, when the node 6 or/and 1 gets the same election message (created by the node 3) from both directions of the ring, they create a leader declaration message ($nldmsg[3]$) to declare the node 3 as the new leader and sends that message to every node of the ring. After getting the leader declaration message ($nldmsg[3]$), each node gets to know about the newly elected leader (cf. Figure 3.3(d)).

3.3 Empirical Appraisement

This section describes the simulation details of the propounded algorithm. By considering some practical scenarios, we simulate and assess the efficiency of our proposed algorithm.

3.3.1 Experiment Setup

All experiments were performed on a single machine. The machine has an Intel Core(TM) i5-2410M processor (2.3GHz, 4MB cache, 2.9GHz Turbo Boost), 8GB RAM 1TB HDD, NVIDIA GeForce graphics, running Ubuntu Linux Release 16.04

(xenial kernel 4.4). For the C programming environment, we use GCC version 5.4.0, and for the message passing interface (MPI), we use MPICH version 3.2.

We can use our algorithm for solving the different problems, including managing the data inconsistency of the replicated servers, clock synchronization, job scheduling, atomic commitment, and load balancing. We simulate our algorithm for managing the data inconsistency of the replicated servers. Nowadays, replicated servers are used to increase the system fault tolerance, performance, reliability, and availability. The replicated servers could be situated in different places, and every replicated server of the system contains identical copies of the same data. This kind of system may face some data inconsistency at the time of data modification in the servers. The data inconsistency arises only as a consequence of the execution of multiple writing commands in a different order in different servers. To overcome this problem, we elect a server as a leader that helps to sequence the data modification commands that come from the clients. Whenever a server gets the data modification commands, it sends the command to the leader server. The leader server performs ordering of the commands in proper sequence and sends them to all the servers to process the commands in the same order.

By considering the above scenario, we have simulated the proposed algorithm (FRLLE) and some existing algorithms to observe the number of exchanged messages and time steps to elect a new system leader with a different number of nodes like $10, 20, 30, ..., 100$. The obtained results are shown using the Figures 3.4 and 3.5. Figures 3.4 (a) and (b) show the number of exchanged messages of different algorithms to elect a leader in the best case and worst case respectively. Figures 3.5 (a) and (b) show the election time of the different algorithms to elect a leader in the best case and worst case respectively.

3.3.2 Performance Comparison and Discussion

We compare the performance and efficiency of the FRLLE algorithm with that of five familiar ring based leader election algorithms such as the Ring [75], the LCR

[28], the HS [59], the Timer-based [19] and Abraham *et al.* [2] algorithms using two comparison parameters i.e., the required amount of time steps and exchanged messages. The ring can be embedded into complete mesh topology [85]. So, we also compare the performance of the FRLLE algorithm with that of well-known complete mesh topology based algorithms like Bully algorithm [50] and Modified Bully algorithm [64]. Tables 3.1 and 3.2 show a priori analysis [33] [101] of the time and message complexity of the different election algorithms. The required amount of time steps and exchanged messages are calculated in terms of N (where N is the number of nodes in the system), and the time and message complexity are represented by O-notation [33] [101]. On the other hand, Figures 3.4 and 3.5 show the simulation results (i.e., election time and the number of exchanged messages) of different algorithms.

After the analysis of Tables 3.1 and 3.2 and Figures 3.4 and 3.5, we get that in the best case scenario, the message complexity and time complexity of Ring, LCR, HS, and Timer-based algorithm is $O(N)$. In contrast, the message and time complexity of the FRLLE algorithm is $O(1)$. That means the FRLLE algorithm reduces the message complexity as well as time complexity from $O(N)$ to $O(1)$. Even if, in the best case, the performance of the FRLLE algorithm is better than that of Bully and Modified Bully algorithms when the number of message passing is concerned. Because the message complexity of the Bully and Modified Bully is $O(N)$, but that of the FRLLE is $O(1)$. In the worst-case scenario the FRLLE algorithm exchanges $(7N^2-14N+8)/4$, $(N^2-8N+8)/4$, $(3N^2-10N+4)/4$, $(N^2-4N+8)/4$, $(N^2-12N+8)/4$, and $(5N^2-8N+8)/4$ numbers of fewer messages than that of Ring, LCR, Bully, Modified Bully, Timer-based, and Abraham *et al.* algorithms respectively. In the worst case, when the number of nodes in the system is less than 230, the FRLLE algorithm exchanges fewer messages than that of the HS algorithm, but when the number of system nodes is greater than 230, HS algorithm exchanges fewer messages than that of FRLLE algorithm. It is because, the message required by the HS and FRLLE algorithms are expressed by $4\big[\sum_{k=1}^{(\lceil \log_2 N \rceil)} 2^k * \lceil N/(2^{k-1}+1)\rceil\big] + 5N$ and $(N^2+14N-8)/4$ respectively, where N is the number of nodes. When $N \leq 230$,

then $(4\big[\sum_{k=1}^{(\lceil \log_2 N \rceil)} 2^k * \lceil N/(2^{k-1}+1)\rceil\big] + 5N)$ is greater than $(N^2+14N-8)/4$. And when $N > 230$, then $(4\big[\sum_{k=1}^{(\lceil \log_2 N \rceil)} 2^k * \lceil N/(2^{k-1}+1)\rceil\big] + 5N)$ is less than $(N^2+14N-8)/4$. When we analyze the required time steps, we find that in the worst case, the FRLLE algorithm takes N, $2N-1$, $5N-2$, and $2N$ fewer time steps than that of Ring, LCR, HS, and Abraham *et al.* algorithms respectively. That means our algorithm is faster than the Ring, LCR, and HS algorithm. On the other hand, the time complexity of the Bully and Modified Bully algorithms is better than that of the FRLLE algorithm. To execute the algorithms like Bully, Modified Bully and Timer-based, a node needs to know the global information of the system (i.e., the total number of nodes in the system and their Ids). Whereas, to execute the FRLLE algorithm, instead of knowing the total nodes and their Ids, a node only needs to know the information of its neighbor nodes. Additionally, in FRLLE a node needs its self-information only to calculate the average CPU, memory, and bandwidth utilization. When the number of nodes in the system is greater than 230, the HS algorithm exchanges fewer messages than the FRLLE algorithm, but it is near about 6 times slower than the FRLLE algorithm. A distributed system gets another vital advantage from the proposed algorithm (FRLLE) than the existing algorithms. The FRLLE algorithm elects the node with the minimum failure rate and the minimum load among the existing nodes in the system as the new leader. So, the elected leader's failure probability and load are also minimum, which means the FRLLE algorithm produces a more reliable leader for the system. As the load of the elected leader is minimum, so the leader is able to manage the system more efficiently.

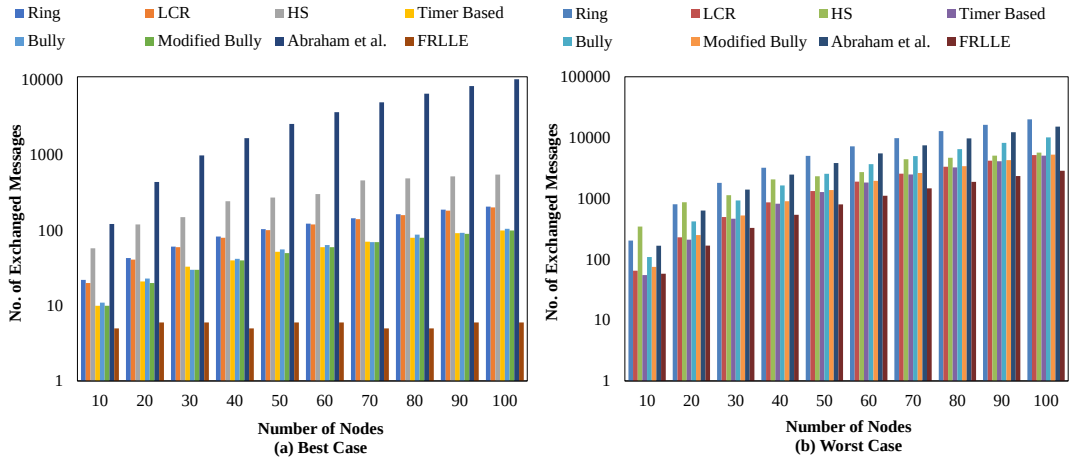


FIGURE 3.4: In the best and worst cases, the total number of exchanged messages of different leader election algorithms.

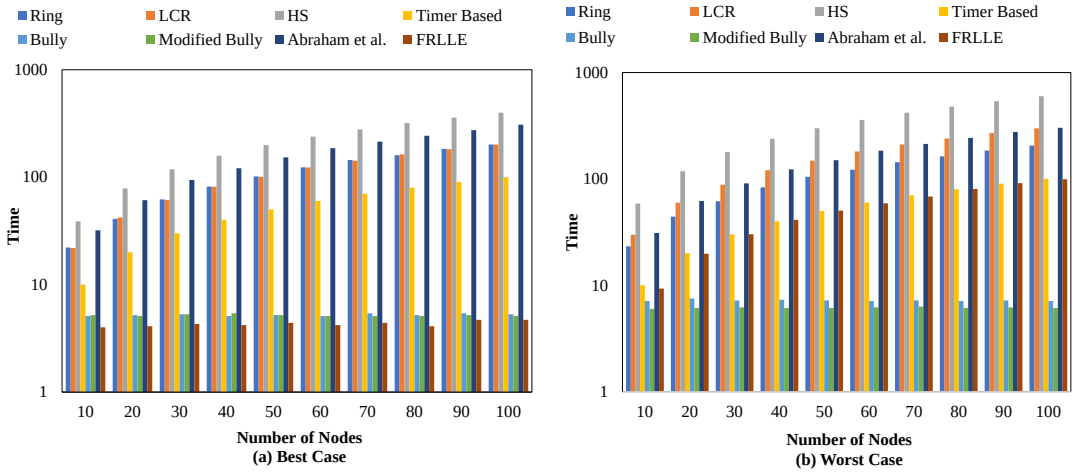


FIGURE 3.5: In the best and worst cases, the total election time of different leader election algorithms.

TABLE 3.1: Message complexity in terms of O-notation of N and number of exchanged messages in terms of N, where N is the total number of nodes.

Algorithm	Message complexity		Number of total exchanged messages	
	Best case	Worst case	Best case	Worst case
Ring [75]	$O(N)$	$O(N^2)$	$2N$	$2N^2$
LCR [28]	$O(N)$	$O(N^2)$	$2N$	$(N^2+3N)/2$
HS [59]	$O(N)$	$O(N \log N)$	$4\left[\sum_{k=0}^{(\lceil \log_2(N/2) \rceil -1)} 2^k\right] + 3N$	$4\left[\sum_{k=1}^{(\lceil \log_2 N \rceil)} 2^k * \lceil n/(2^{k-1}+1) \rceil\right] + 5N$
Timer-based [19]	$O(N)$	$O(N^2)$	N	$(N^2+N)/2$
Bully [50]	$O(N)$	$O(N^2)$	N	N^2+N-1
Modified Bully [64]	$O(N)$	$O(N^2)$	N	$(N^2+5N)/2$
Abraham et al. [2]	$O(N^2)$	$O(N^2)$	N^2+2N	$(3N^2+3N)/2$
FRLLE	$O(1)$	$O(N^2)$	4	$(N^2+14N-8)/4$

TABLE 3.2: Time complexity in terms of O-notation of N and number of required time steps in terms of N, where N is the total number of nodes.

Algorithm	Time complexity		Required time steps	
	Best case	Worst case	Best case	Worst case
Ring [75]	$O(N)$	$O(N)$	$2N$	$2N$
LCR [28]	$O(N)$	$O(N)$	$2N$	$3N-1$
HS [59]	$O(N)$	$O(N)$	$4N-2$	$6N-2$
Timer-based [19]	$O(N)$	$O(N)$	N	N
Bully [50]	$O(1)$	$O(1)$	3	5
Modified Bully [64]	$O(1)$	$O(1)$	3	4
Abraham et al. Algorithm [2]	$O(N)$	$O(N)$	$3N$	$3N$
FRLLE Algorithm.	$O(1)$	$O(N)$	2	N

3.4 Summary

In this chapter, we have devised and illustrated a leader election algorithm for a crash-recovery partially synchronous distributed system by considering the failure rate and average load of the nodes. Here, we introduce the concept of the leader coefficient to elect a reliable and competent leader. Such a leader can utilize the resources appropriately and improve overall system performance. Till now, the existing algorithms that work on ring networks have $O(N)$ message and time complexity in the best case scenario. On the other hand, this proposed algorithm has $O(1)$ message and time complexity in the best case. In the worst case scenario, the FRLLE algorithm exchange fewer messages than the Ring, LCR, Timer-based, Abraham *et al.* algorithms that work on ring network and it is $2, 3, 3$ and 6 times faster than Ring, LCR, Abraham *et al.* and HS algorithms respectively. We have also proved that the FRLLE algorithm satisfies the uniqueness, agreement and termination conditions that help to build a consistent distributed system.