Computers **2012**, *1*, 3-23; doi:10.3390/computers1010003

OPEN ACCESS

computers

ISSN 2073-431X

www.mdpi.com/journal/computers

Article

Enhanced Bully Algorithm for Leader Node Election in Synchronous Distributed Systems

Md. Golam Murshed and Alastair R. Allen *

School of Engineering, University of Aberdeen, Aberdeen, AB24 3UE, Scotland, UK;
E-Mail: mg.murshed@abdn.ac.uk

* Author to whom correspondence should be addressed; E-Mail: a.allen@abdn.ac.uk;
Tel.: +44-1224-272501; Fax: +44-1224-272497.

Received: 6 April 2012; in revised form: 19 May 2012 / Accepted: 5 June 2012 /
Published: 25 June 2012

Abstract: In distributed computing systems, if an elected leader node fails, the other nodes of the system need to elect another leader. The bully algorithm is a classical approach for electing a leader in a synchronous distributed computing system. This paper presents an enhancement of the bully algorithm, requiring less time complexity and minimum message passing. This significant gain has been achieved by introducing node sets and tie breaker time. The latter provides a possible solution to simultaneous elections initiated by different nodes. In comparison with the classical algorithm and its existing modifications, this proposal generates minimum messages, stops redundant elections, and maintains fault-tolerant behaviour of the system.

Keywords: leader node election; distributed systems; bully algorithm

1. Introduction

Centralised control in distributed systems helps to achieve some specific goals such as mutual exclusion, synchronization, load balancing, and time scheduling. This type of distributed system often requires a unique node to play the role of leader or coordinator of the other nodes. As node crash failure is not uncommon in distributed systems, failure of a leader node requires special attention and needs extra tasks to elect another one to act as leader. A leader election algorithm is one of the fundamental activities of distributed systems, as it acts as a basis for more complex and high level algorithms and applications. Leader election is important in various systems, for example in certain implementations of distributed web services. Distributed systems require some special capabilities of a good and efficient leader election algorithm, such as leader longevity, low communication overhead, low complexity in terms of time and messages, and providing uniqueness to the elected leader. Several algorithms had been proposed to deal with the leader node failure problem, and the Bully Algorithm is the classical one amongst them for electing a leader node in synchronous systems, although this algorithm demands a large number of messages between the nodes. A distributed system can be defined as synchronous if some bounds are predefined: (a) lower and upper bound times to execute processes of a node are known in advance; (b) within a known bounded time the underlying communication network is able to transmit a message from one node to another node; (c) each node contains a local clock whose drift rate has a known bound [1]. So it can be inferred from these properties that a synchronous distributed system is deterministic by nature. This implies that a leader election algorithm is also a deterministic procedure that can be achieved by a number of communications among the nodes.

Generally, in fault-tolerant distributed systems the leader node has to perform some specific controlling tasks and this node is well known to the other nodes. This node does not necessarily possess any extra processing feature to become elected, but election algorithms need a special mechanism to elect the leader. After crash failure of the leader node, it is urgently needed to reorganize the existing active nodes to call for an election and to elect a leader in order to continue the operation of the entire system.

The main focal point of an efficient and robust leader election algorithm is to minimise the number of messages generated for the election procedure and hence to reduce the complexity of the overall execution time. Based on simulation and analysis, this paper shows that Garcia-Molina's bully algorithm [2] can be modified to enhance its performance in message passing and subsequently demand less time complexity that results in electing a new leader node faster. The paper is organised as follows: Section 2 describes the features of a leader election algorithm and Section 3 reviews related work. The enhancements of the bully algorithm are described in Section 4, and the operation of the algorithm is clarified with the aid of an example in Section 5. Section 6 presents a performance analysis of the proposed and former algorithms. Section 7 draws the conclusion of this paper.

2. Features of a Leader Election Algorithm

Leader election is a procedure that is embedded in every node of the distributed system. Any node which detects the failure of the leader node can initiate a leadership election. The election concludes its operation when a leader is elected and all the nodes are aware of the new leader and agree on that.

In a leader election like the bully algorithm the following assumptions are made: (a) each node must have a unique id determined by the operating system; (b) each node knows the id of the others; (c) nodes are not aware of the current state of other nodes. Along with these, the following requirements must be met by the algorithm [3]:

1. Safety. All the live nodes must agree on the elected leader [4]. Each node contains a local variable *ldr* which indicates the current leader of the system. Moreover, each node also contains another local variable *state*, which represents the current status of the node. It may be in *normal*, *elect* or *wait* state. When the node is in normal operation and there is an active leader in the system then the state variable contains the value *normal*. By contrast, *elect* and *wait* are exceptional cases. State

elect occurs when the system is in the process of electing a new leader. State *wait* is a special case during the election: this state is used when a node actively takes part in the election and is waiting for the result.

2. Liveness. After participating in an election all the live nodes and the elected leader node must come into a position where all of them are in operational state *normal*. Each node must set its local variable $ldr = m$, where *m* is the node id of the elected leader [4–7].

A node calling an election means that it initiates an election algorithm. According to [4,8], a node can not initiate more than one election at a time. But in a distributed system *n* nodes can initiate *n* elections concurrently, where $n > 1$ is an integer.

3. Literature Review

Failures with particular features in a distributed system determine the solution to be proposed. A leader election algorithm requires some system features to be satisfied in order to work properly. The bully algorithm proposed by Garcia-Molina [2] assumes that the system has the following properties:

1. The system is synchronous, and consists of a fixed set of nodes that are connected by a reliable communication network. Nodes communicate with each other by message passing.
2. Nodes in the system never halt temporarily and reply to incoming messages immediately.
3. Integers are used to identify nodes. Every node knows the ids of the others.
4. All nodes use the same leader election algorithm.
5. A node has no prior knowledge that the current leader node has crashed: a timeout policy is used to detect node failure.
6. Crashed nodes may recover and may rejoin the system provided that they agree upon the current election algorithm.

After completion of a successful election according to the bully algorithm, (a) the node with the highest id of all live nodes is elected as the leader or coordinator: there should be only one leader; (b) all nodes in the system agree on the newly elected leader. This algorithm uses three types of message:

election: This message is generated by the node which detects failure, to announce an election.
answer: Recipient nodes send this acknowledgment in response to the *election* message.
coordinator: The newly elected leader node announces itself as leader by sending this message to all other nodes.

An election is initiated when one node detects, through time-out, that the leader node has crashed. Since the system is synchronous and the upper limit to pass a message is known a priori, the local failure detector of a node reports possible leader failure if it does not receive an expected message within a predefined time period. More than one node may discover the crash and announce more than one election concurrently [4,8,9]. At the outset of an election, the detector node *i* sends the *election* message to all the nodes of its group whose id number is greater than *i*. It then waits for *answer* message from the recipient nodes. If it does not receive any within a predefined time bound, it declares itself as the leader by sending *coordinator* message to all nodes with lower id numbers than itself. If it receives *answer* message from any of the nodes, it waits a further period to receive *coordinator* message that informs about the newly elected leader. If the first election initiator node does not receive *coordinator* message within a predefined time, it starts another election. This procedure is carried on until a new leader node is elected and all the nodes agree on the new leader node.

A node *k* that recognizes the failure of the leader node can immediately announce itself as the leader if it finds that no other live node contains a higher node id than itself (*i.e.*, $k > i$ where $i \in I$ and $I = \{\text{IdsOfAllLivingNodes}\}$). Otherwise, it starts an election by sending *election* message to the higher numbered nodes. If an ordinary (not a leader) node recovers and joins the group, it first sends *election* message to nodes having higher id than itself and some elections take place in the system sequentially. If a former leader recovers, it initiates another election and bullies the other nodes into submission.

The bully algorithm has some drawbacks. (a) Every time a node (former leader or ordinary node) recovers from a crash failure, it initiates an election, which consumes significant system resources. (b) Although this algorithm ensures liveness, it sometimes fails to meet the safety condition. This can happen when a former leader node is replaced by a node with the same id number while the election procedure is in progress [4]. The newly elected node and the former leader node (which was down for a while) will both announce themselves as leaders simultaneously. (c) This algorithm elects a new leader node with the help of a number of redundant elections. In the worst case, it can require a large number of messages to elect a leader node. The worst case occurs when a node with the lowest id initiates an election: the role for initiating election is handed over to a node with next higher id and this continues till the node with the highest id takes over the role. As a result, at least $n - 2$ redundant elections take place in the entire system where n is the number of nodes. (d) This algorithm does not provide an efficient solution for the simultaneous detection of leader node failure by more than one node. More than one election may take place at the same time, which imposes a heavy load on the network. (e) This algorithm does not provide a robust solution in an exceptional case like the potential electioneer node crash.

A significant modification to the conventional bully algorithm has been proposed by Mamun *et al.* in [9]. In this paper, two more new message-types have been introduced: *ok* message and *query* message. Redundant elections have been stopped using these two messages. The authors suggest that whenever a node detects failure of the leader node it announces an election. In response to this election all recipient nodes will send back *ok* messages to the announcer. Upon receipt of all *ok* messages, the electioneer-node knows the identity of the highest id holder of all the live nodes. It then sends *coordinator* message to all the nodes, announcing the new leader. If the former leader wakes up, it just announces itself as leader. If an ordinary node wakes up, it sends *query* message to the higher nodes and upon receipt of *answer* messages it comes to know who the present leader is.

There are several drawbacks to this modified algorithm as well. (a) This algorithm does not guarantee to stop more than one simultaneous election. This usually happens when a lower numbered node detects the failure of the leader while an election is in progress which was initiated by another node of higher id. Because of its ignorance about the ongoing election it will start another election, which causes more message generation in the system. Moreover, the failure detector modules of more than one node may detect the crash of the current leader node at exactly the same time and may initiate more than one election in parallel. The worst case occurs when all the live nodes except the next highest id node simultaneously recognise the failure of the leader and begin more than one new election concurrently. In that case, all the live nodes have to reply with *ok* messages to their respective sender nodes, which incurs $O(n^2)$ messages over the network. (b) This algorithm does not propose any action to be taken if a new node with a higher node id is added to the system while the former leader is down. (c) This algorithm does not provide any solution, or does not suggest any urgent action to be taken, when an exceptional case happens, such as the electioneer node failing during the election.

Another modification to the conventional bully algorithm has been proposed in [10] as a solution to the large number of messages generated. The authors proposed a coordination group consisting of several preselected ordered nodes. The aim of forming this group is to prevent global elections between all nodes. After detecting the crash of the current leader node, the detector node sends the *crash-leader* message to the next candidate/alternative node and waits for the reply. If the alternative node is alive, it sends back an *ok* message to the detector node and also sends a message to the crashed node. Being sure about the crash of the current leader node, the alternative node declares itself as the leader, broadcasting *coordinator* message to all the nodes of the system. In the case of failure of all the candidate nodes in the coordinator group, a new election takes place. In this procedure, the detector node forms a new coordinator group, sending and receiving *election* and *ok* messages respectively. The proposed modification certainly minimizes the number of messages compared to the original algorithm, but electing a new leader might take a long time. Moreover the paper did not mention about the possible cardinality of the coordinator group and did not explain some extraordinary cases like the detector node failure case, former leader revival case, *etc*.

To make the traditional algorithm fault tolerant, the authors in [11] proposed a minor modification to the election process. In this algorithm, when a node detects a possible failure of the current leader node, it sends *election* message to all nodes with higher id numbers. Upon receiving the *ok* messages, the detector node selects the node with highest id of all the living nodes and declares that by broadcasting *coordinator* messages. The authors assumed that the messages might get corrupted or lost and that is why an extra round in the election process has been proposed. The newly declared node again initiates another election by sending *election* messages to the higher id nodes and finishes the election by declaring the existing highest id node as the leader node based on the reply sent by the higher id nodes. Garcia-Molina mentions some assumptions about synchronous distributed systems: assumption 6 is “there are no transmission errors”, and assumption 8 is “the communication subsystem does not fail”. So it is quite clear that a node might fail but not the messaging system. The proposed modification in [11] takes a long time to elect a new leader node and does not consider some more exceptional cases that could arise in the system.

Although the traditional bully algorithm had been proposed in 1982, it is still a very useful and preferred algorithm for some important applications. In [12], the author proposed to use a variation called Fast Bully Algorithm (FBA) to implement a Web Service Community. Some web services that are functionally similar are grouped as a community to make the whole service faster, convenient and available. In this application, one of the web services plays the role of the Master Web Service (Coordinator) and FBA is used to elect a new Master Web Service in case of coordinator failure. The author also developed Weather Community as a prototype of the web service community, and illustrated the performance of FBA. A similar application has been proposed in [13], where web services have been implemented using the Whisper architecture. One of the components of this architecture is Peer

Group where the members provide similar functionality and implement the bully algorithm to elect a coordinator.

4. Proposed Enhanced Bully Algorithm

This paper proposes some modifications to Garcia-Molina's bully algorithm [2] and the modified bully algorithm [9]. The basic system assumptions are as in [2], and the types of message used are very similar to the modification proposed in [9]. Our algorithm is detailed in the Appendix.

4.1. Proposed Modifications

Analyzing the shortcomings of these algorithms, here we are proposing a *set division*. According to this concept, all the nodes of a synchronous distributed system are divided into two sets: *Candidate* and *Ordinary*. *Candidate* is comprised of $\lceil N/2 \rceil$ nodes, where *N* is the number of nodes in the system. The other nodes will be in *Ordinary*, such that any node in *Candidate* has a higher node id than any node in *Ordinary*. Every node of the system should be aware of the other nodes' sets and ids. The number of nodes in each set is fixed for now, and no reshuffle is needed for the time being as it would add extra overhead to the system. But if new nodes are added regularly to the system, after a certain number of additions the set cardinality will be rearranged. Cardinality arrangement is not discussed in this paper.

Along with the *set* concept we also propose several important corrections to the algorithm defined in [9].

1. We assume that *election* message contains the id of the failure detector node and the id of the failed node, and that *answer* message contains the node ids of the leader and *Candidate* set.
2. We introduce the notion of *tiebreaker time* δ which is greater than the node to node transmission time and is unique and different for every node. Higher id nodes should have lower δ and lower id nodes should have higher δ.
   The operating system will determine the value of δ for every node individually using the following Equation:
   $$\delta_i = \frac{\alpha}{i} + (N - (i-1))t_{TX} \tag{1}$$
   where *i* is node id, *N* is number of nodes in the system, t_{TX} is the average node to node message transmission time in the system, and α is a real number constant set by the operating system dynamically. Here we follow the assumptions 8 and 9 in [2] made by Garcia-Molina about node to node message transmission time t_{TX}.
   So δ_i is the tiebreaker time of a particular node with id *i* which is unique and varies from other nodes' tiebreaker times. For instance, if the value of α is 2.0 and the number of nodes is 10, the value of tiebreaker time of node 5 is $\delta_5 = 6t_{TX} + 0.4$, and for node 6 is $\delta_6 = 5t_{TX} + 0.33$. Notice that the difference in tiebreaker time between two nodes is at least t_{TX}.
3. We also propose a correction to the waiting time (time-out). After sending *election* message, node *i* waits for time
   $$T_{el,i} = 3t_{TX} + 2t_P + \delta_i$$

where t_P is the processing time at the sender node. If we assume $t_{TX} \gg t_P$, then

$$T_{el,i} \cong 3t_{TX} + \delta_i.$$

Additionally, if a node *i* sends *ok* message, it will wait for time

$$T_{ok,i} = 2t_{TX} + t_P + \delta_i$$
$$T_{ok,i} \cong 2t_{TX} + \delta_i.$$

4.2. The Election Procedure

It is assumed that each node knows its respective id and which set it belongs to, and the ids and sets of all other nodes. Our proposed election algorithm is capable of handling possible exceptional situations that could arise in a synchronous distributed system and these are described below and detailed in the Appendix.

4.2.1. Ideal Case (IC)

When a node $i \in$*Ordinary* detects the failure of the leader node it sends *election* message to the nodes of *Candidate* and waits for $T_{el,i}$ time to receive *ok* message. On receiving *election* message, all the live nodes of *Candidate* will reply with *ok* messages to the sender node. Upon receipt of *ok* messages, detector node learns the highest id node of all living nodes at present. It then generates *coordinator* messages and informs all the nodes of both sets about the new leader node. If a node from *Candidate* detects the failure, it will send *election* message to higher id nodes of its own set. The recipient nodes confirm about the election by sending *ok* messages to the electioneer node and the electioneer node now knows the live highest id node and broadcasts *coordinator* messages to declare the new leader. In this way, eventually a leader will be elected. If a node notices that it is the next highest id node after the crashed leader node, it declares itself as the leader directly. The Ideal Case is detailed in Algorithms 1–4 in the Appendix.

4.2.2. Candidates Failure Case (CFC)

If a detector node from *Ordinary* does not get a reply from *Candidate* within $T_{el,i}$ time, it assumes all the nodes in *Candidate* have crashed and it then sends *election* message to the nodes of its own set with higher id than itself. After that, the election takes place as normal. In this situation the election procedure takes more time and messages than that of the ideal case. However, this case is very rare.

4.2.3. Electioneer Failure Case (EFC)

This case occurs when the detector (electioneer) node crashes just after sending *election* messages. After receiving the *election* message, all recipient nodes reply by sending *ok* message and wait for $T_{ok,i}$ time. If a potential candidate node *i* does not receive any *coordinator* message within $T_{ok,i} \cong 2t_{TX} + \delta_i$ time, it will declare itself as the new leader node by sending *coordinator* messages to all the nodes of the system. Here we assume that the multicast *election* messages are received by the potential candidates at nearly the same time.

4.2.4. Simultaneous Election Case (SEC)

More than one node may detect that the leader node has crashed. In that case, detector nodes will initiate separate elections simultaneously [4,9]. Every election will follow the rules of the ideal case described in Section 4.2.1. A potential node in Candidate Set to become leader may receive more than one *election* message simultaneously. A detector node must send *election* message to the potential nodes and start its clock for $T_{el,i}$ time-out period. Potential nodes that received *election* message will reply with *ok* message only to the highest id node of all the sender nodes. Here after sending *ok* messages, nodes will wait for $T_{ok,i}$ time. Upon receiving *ok* messages the detector node declares the new leader by broadcasting *coordinator* message to all nodes of the system. If the time-out period, $T_{el,i}$ (and $T_{ok,i}$ as well), is over, the following steps have to be taken.

1. If the detector node belongs to *Ordinary*, it assumes that no node in *Candidate* is alive and it initiates another election inside its own set. But it is very rare that all the nodes in *Candidate* set (which is around 50% of all nodes in the system) have crashed.
2. If the detector node belongs to *Candidate* it will declare itself as the new leader. The highest id node has to wait less time compared with the others, because the tiebreaker time δ varies significantly from one node to another. Hence, one node will declare itself as leader before the others. Here, as δ is unique to every node, this algorithm will meet both the liveness and safety requirements of the leader election algorithm.

4.2.5. Node Revival Case (NRC)

When a node of *Ordinary* recovers from failure, it will first generate a *query* message and send it to the nodes of *Candidate*. If the revived node is of *Candidate* it will send *query* message to the higher id nodes of its own set. The nodes which received *query* messages will reply with *answer* messages to the newly revived node. The purpose of *answer* message in Garcia-Molina's bully algorithm and in our proposal is not the same. In our proposed modification, *answer* message acts not only as an acknowledgment but also as an information message: it contains the id of the current leader node and the ids of the members of *Candidate*. After receiving *answer* messages, it comes to know the identity of the leader at this moment. If the leader is a lower id node than itself, it will initiate an election. It will do nothing otherwise. A new node can be added to the system to meet the requirements of computational purposes. If this node gets the highest node id and is elected as leader while a former leader is down, the former one may not know about the new leader. But our proposed modification in *answer* message informs the revived former leader node or an ordinary node about the current live nodes in *Candidate*.

5. Example Election

The whole election process proposed in this paper can be described with the help of an example. Let there be 10 nodes with id $1 \dots 10$. Nodes $1 \dots 5$ belong to *Ordinary* and nodes $6 \dots 10$ belong to *Candidate*. Each node has its own δ assigned by the operating system. Each node knows: which set it belongs to, the other nodes' ids, which sets the other nodes belong to, and which node is the current leader. We assume that node 10 is the current leader. At present it is down and node 3 has detected that.

Node 3 sends *election* messages to *Candidate* (Figure 1). The nodes of *Candidate* send *ok* messages to node 3, the detector, which is depicted in Figure 2. As shown in Figure 3, node 3 now knows which is the highest id node and declares node 9 as the new leader using *coordinator* message.

Figure 1. Election initialisation.

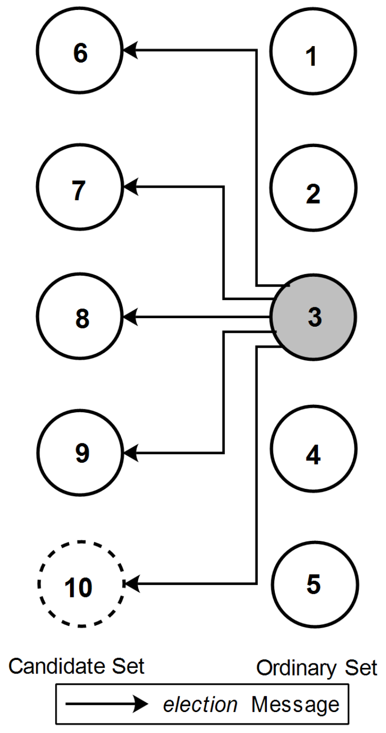


Figure 2. *ok* message.

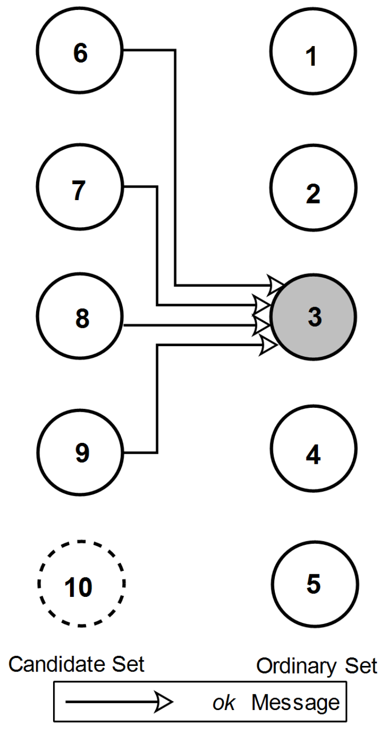
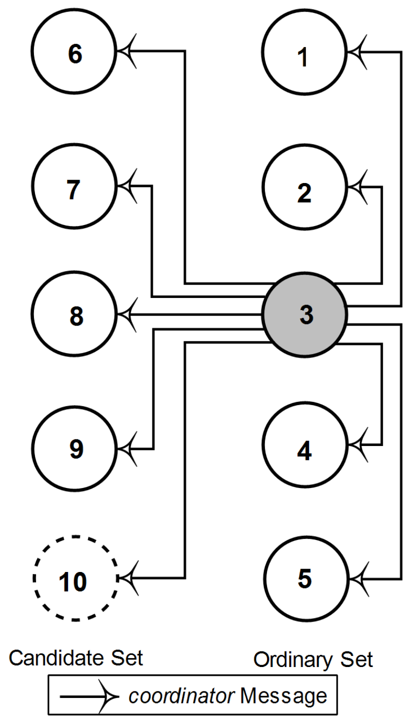
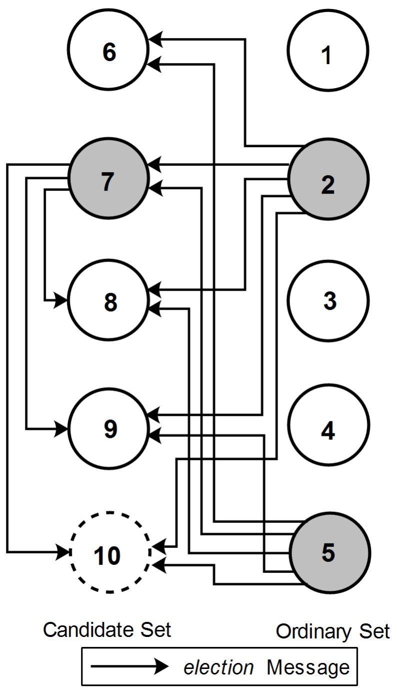


Figure 3. Leader declaration.



Now suppose node 10 is the leader and it crashes, and nodes 2, 5 and 7 detect that simultaneously. Now nodes 2, 5 and 7 will initiate three separate elections by sending *election* messages to *Candidate*, as shown in Figure 4.

Figure 4. Election initialisation in parallel.



Upon receipt of *election* messages nodes 9 and 8 will send *ok* message to node 7. At the same time nodes 7 and 6 will send *ok* message to node 5 as shown in Figure 5, because to each of these nodes in *Candidate* it is the highest id node among the detector nodes. After sending *ok* messages, nodes will start their own timer to wait $T_{ok,i}$ time to receive *coordinator* message. Detector nodes 7, 5 and 2 wait up to $T_{el,7}$, $T_{el,5}$, $T_{el,2}$ times respectively to receive *ok* message. Here we should note that $T_{el,i} > T_{ok,i}$ and $\delta_i \neq \delta_j$ where $i \neq j$. However, node 7 knows about the highest live node id, and declares the new leader, *i.e.*, node 9, and Figure 6 shows that.

Figure 5. *ok* message to the highest id electioneer nodes.

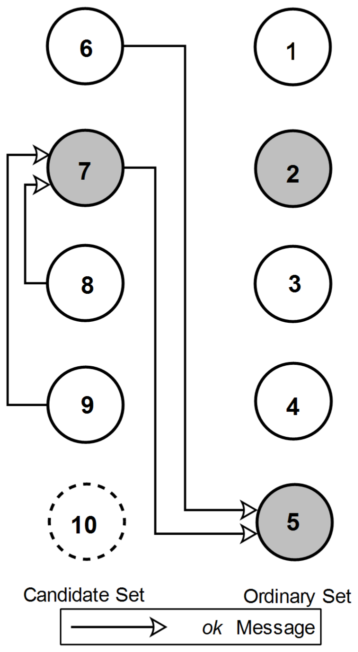
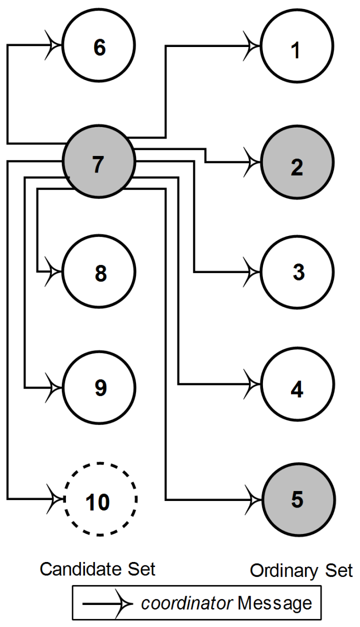


Figure 6. Declaration of the new leader node.



6. Performance Analysis

Based on message generation in the system, a comparative analysis of [2,9] and our proposed algorithm would be appropriate to determine which algorithm performs better than the others.

Garcia-Molina's bully algorithm requires $N - 1$ messages to elect a leader node in the best case, where N is the number of nodes. The best case happens when the node having the next highest id number detects the failure of the leader node and hence announces an election. In the worst case it requires $O(N^2)$ messages to elect a leader node. The worst case happens when the lowest id node of the system detects the failure of the leader node. It will send *election* messages to $N-1$ nodes having higher id than itself. Each of the nodes eventually initiates a separate election one by one. In this algorithm, a previously failed node which was not a leader node initiates an election after recovery. But if it was a former leader, it just broadcasts *coordinator* messages to other nodes to announce itself as the new leader. Hence, it requires $O(N^2)$ messages to elect a leader node in the worst case and $N - 1$ messages in the best case. The Simultaneous Election Case (SEC) happens when more than one node detects the leader node's failure at the same time or within a very short time interval. In our example analysis, we assumed nodes 2, 5 and 7 detected the failure in parallel and started elections individually.

The modified Bully Algorithm proposed in [9] also requires $N - 1$ messages in the best case. But it gains a significant improvement in the worst case. It requires only $O(n)$ messages to elect a new leader node in the worst case. On the recovery of a node, $N - 1$ messages are required if the recovered node is a former leader and $2(N - 1)$ messages are required if the recovered node has the lowest id.

The algorithm proposed in this paper also requires $N - 1$ messages in the best case. But it requires at most $2N - 1$ messages in the worst case if at least one node in *Candidate* is live. Therefore there is almost a 30% gain with respect to the algorithm proposed in [9]. It is very unlikely that all the nodes in *Candidate*, which is around 50% of the total nodes, are down at the same time. On the recovery of a former leader, the algorithm necessitates $N - 1$ messages. If the recovered node holds the lowest id, N messages are required.

Neither the algorithm in [2] nor in [9] explains the situation when more than one node detects the crash of the leader simultaneously and initiates separate elections concurrently. Basically in concurrent election, the worst case occurs when all the living nodes (except the next highest id node) detect the failure of the leader node. In the case of the bully algorithm and its modification proposed in [9], almost $O(N^2)$ messages are required, although ignored, to elect a leader in the worst case. But in our proposed modification we tried to minimize the number of messages by using set division and tie breaker time. The tie breaker time stops redundant and unnecessary elections and helps to reduce the number of messages needed to elect a new leader node. To compare our proposed modification with [2,9], we assumed that all the detector nodes will initiate separate elections and complete the elections until an identical, new leader node is elected. Tables 1–3 show that at least a 37% reduction in messages has been achieved compared with [9] in the Simultaneous Election Case (SEC).

Table 1. Performance analysis: $N = 5$, average $T_{el} = 1201\mu$s.

Case	Key nodes	Number of messages			Latency (μs)		
		[2]	[9]	Our alg.	[2]	[9]	Our alg.
Best case	n_4	4	4	4	800	800	800
Worst case	n_1	20	11	9	8804	3401	3403
SEC	n_1, n_3	20	18	11	8804	3401	3001
NRC	n_2	13	5	5	6201	1000	1000

Table 2. Performance analysis: $N = 10$, average $T_{el} = 1701\mu$s.

Case	Key nodes	Number of messages			Latency (μs)		
		[2]	[9]	Our alg.	[2]	[9]	Our alg.
Best case	n_9	9	9	9	1800	1800	1800
Worst case	n_1	90	26	18	33309	6901	6201
SEC	n_2, n_5, n_7	73	56	26	28208	6501	4600
NRC	n_3	58	13	9	23507	2600	1800

Table 3. Performance analysis: $N = 20$, average $T_{el} = 2703\mu$s.

Case	Key nodes	Number of messages			Latency (μs)		
		[2]	[9]	Our alg.	[2]	[9]	Our alg.
Best case	n_{19}	19	19	19	3800	3800	3800
Worst case	n_1	380	56	38	127357	13903	12204
SEC	n_4, n_5, n_{16}	275	124	52	98248	12703	7200
NRC	n_{20}	19	19	13	3800	3800	3800

Moreover [2,9] do not indicate what would happen if the electioneer node crashes before it announces the new leader. It can be guessed that both algorithms allow another node to detect the leader failure again through time-outs, and initiate a new election. But we are proposing (in Section 4.2.3) a robust method that must elect a leader after initiating an election, whether or not the electioneer node crashes in the middle of the election. So, our algorithm completely ensures both liveness and safety requirements of the leader election algorithm.

In order to compare the performance of the algorithms, we execute them in three test cases where the systems comprised 5, 10, and 20 nodes respectively.

We consider a fast Ethernet environment with data transfer speed 100 Mb/s and latency 200 μs. For our proposed algorithm here we assume $t_{TX} = 200\mu$s, $\alpha = 3.0$. We takes as an example a system with 10 nodes, having node id $1 \ldots 10$ and leader node 10 has recently crashed. The *tiebreaker times* may be calculated by Equation 1: $\delta_i = \{2003, 1802, 1601, 1401, 1201, 1000, 800, 600, 400, 200\}$. Since Garcia-Molina's Bully Algorithm [2] and its modified version [9] utilize the *time-out* period to recognize any failure of nodes, we assume the time-out period is the average of all the $T_{el,i}$, that is, the waiting times for election messages. Therefore we estimate a waiting time of 1701 μs for the Garcia-Molina [2] and Mamun [9] algorithms. We analyzed the algorithms for several different cases: Best Case, Worst Case, Simultaneous Election Case (SEC), and Node Revival Case (NRC). Table 2 presents the results.

Best case: Here in the example of Table 2, node 9 first detects that the current leader node, node 10, has crashed, and declares itself as the new leader, sending 9 *coordinator* messages. The number of messages for this case is the same for all three algorithms. Assuming a latency of 200 μs, the overall time needed to elect a new leader node in this case is 1800 μs.

Worst case: The traditional Bully algorithm elects a new leader node, in this case, by performing a series of redundant elections and ends up producing 90 messages in total. In each election, which is 9 in number in this case, the electioneer node waits the average T_{el} time which is 1701 μs. So the total time needed is $90 \times 200 + 9 \times 1701$ μs. Algorithm [9] needs 26 messages to elect a new leader and the electioneer node needs to wait the average T_{el} time only once. Our proposed algorithm needs less messages but the waiting time for node 1 is higher than the average T_{el} time since each node has been assigned a unique waiting time after sending *election* message.

Simultaneous election case: The traditional bully algorithm performs 8 elections in this case and requires 73 messages. Each election requires 1701 μs for waiting time. Since three elections are taking place simultaneously, algorithm [9] requires 24 messages where each takes 200 μs. During the election, node 2 has to wait the average T_{el} time to complete the election. So the total time needed is $24 \times 200 + 1 \times 1701$ μs. Our algorithm needs 16 effective messages and T_{el} time = 1400 for node 7. So the total time is $16 \times 200 + 1 \times 1400$ μs.

We tested these three algorithms for systems comprised of 5 and 20 nodes as well. The tie breaker times have been calculated as in the previous example. In the case of 5 nodes, the waiting time has been calculated for [2,9] to be 1201μs and for 20 nodes, 2703μs. The results are given in Tables 1 and 3.

7. Conclusions

This paper presents some modifications to the classical bully algorithm which overcome the limitations of this algorithm, and make it efficient and fast to elect a leader in synchronous distributed systems. The performance of the proposed algorithm has been compared with the original bully algorithm [2] and its modification [9] and our proposal produces a better outcome. The algorithm is fast and guarantees correctness and robustness, and the simulation results show that it requires fewer messages to elect a new leader. There is also scope to propose an algorithm for an asynchronous system and this will be future work. We also aim to investigate in detail the probability of occurrence of the various cases described in this paper.

References

1. Sinha, P.K. *Distributed Operating Systems Concepts and Design*; Prentice-Hall: Upper Saddle River, NJ, USA, 2002; pp. 332–334.
2. Garcia-Molina, H. Elections in a distributed computing system. *IEEE Trans. Comput.* **1982**, *C-13*, 48–59.
3. Wang, X.; Teo, Y.M.; Cao, J. Message and time efficient consensus protocol for synchronous distributed systems. *J. Parallel Distrib. Comput.* **2008**, *68*, 641–654.
4. Coulouris, G.; Dollimore, J.; Kidberg, T. *Distributed Systems Concept and Design*; Pearson Education: Essex, UK, 2003; pp. 431–436.
5. Kim, J.L.; Belford, G.G. A distributed election protocol for unreliable networks. *J. Parallel Distrib. Comput.* **1996**, *35*, 35–42.
6. Peleg, D. Time-optimal leader election in general networks. *J. Parallel Distrib. Comput.* **1990**, *8*, 96–99.
7. Stoller, S.D. Leader Election in Distributed Systems with Crash Failures; Technical Report 481; Computer Science Department, Indiana University: Indiana, IN, USA, 1997.
8. Tanenbaum, A.S.; Steen, M.V. *Distributed Systems Principles and Paradigms*; Prentice-Hall: Upper Saddle River, NJ, USA, 2003; pp. 262–263.
9. Mamun, Q.E.K.; Masum, S.M.; Mustafa, M.A.R. Modified Bully Algorithm for Electing Coordinator in Distributed Systems. In *Proceedings of the 3rd WSEAS International Conference on Software Engineering, Parallel and Distributed Systems*, Salzburg, Austria, 13–15 February 2004.
10. Gholipour, M.; Kordafshari, M.; Jahanshahi, M.; Rahmani, A. A New Approach for Election Algorithm in Distributed Systems. In *Proceedings of the Second International Conference on Communication Theory, Reliability and Quality of Service*, Colmar, France, 20–25 July 2009; pp. 70–74.
11. Effatparvar, M.; Effatparvar, M.; Bemana, A.; Dehghan, M. Determining a Central Controlling Processor with Fault Tolerant Method in Distributed System. In *Proceedings of the International Conference on Information Technology: New Generations*, Las Vegas, NV, USA, 2–4 April 2007; pp. 658–663.
12. Subramanian, S. Highly-Available Web Service Community. In *Proceedings of Sixth International Conference on Information Technology: New Generations*, Las Vegas, NV, USA, 2007; pp. 296–301.
13. Cardoso, J. Semantic Integration of Web Services and Peer-to-Peer Networks To Achieve Fault-Tolerance. In *Proceedings of IEEE International Conference on Granular Computing*, Atlanta, GA, USA, 10–12 May 2006; pp. 796–799.

Appendix

$$\begin{aligned}
N &= \text{number of nodes}\\
i &= \text{node number}\\
\mathit{Candidate} &= \text{set of } \lceil N/2 \rceil \text{ higher id nodes}\\
\mathit{Ordinary} &= \text{set of } \lfloor N/2 \rfloor \text{ lower id nodes}\\
\mathit{state} &= \text{state of node}\\
\mathit{ldr} &= \text{local variable containing the node id of the current leader node}\\
\delta &= \text{tie breaker time set for individual nodes}\\
T_{\mathit{el},\, i} &= 3t_{\mathit{tx}} + \delta_i\\
T_{\mathit{ok},\, i} &= 2t_{\mathit{tx}} + \delta_i\\
t_{\mathit{tx}} &= \text{transmission time}
\end{aligned}$$

Algorithm 1 Electn

```
procedure ELECTN
    Pre: Node $I$ recognises that leader node has crashed
    $state_I \leftarrow elect$
    Post: New leader node elected
    $state_I \leftarrow normal$

    if $I \in Candidate$ then
        Procedure ElectnCand
    else
        Procedure ElectnOrd
    end if
end procedure
```

Algorithm 2 Update

```
procedure UPDATE(var, val)
    $var_i \leftarrow val \mid \forall i$
    $state_i \leftarrow normal \mid \forall i$
end procedure
```

Algorithm 3 ElectnCand

```
procedure ELECTNCAND          // $I \in$ Candidate
    if ($I$ is next highest node id after previous leader node) then
        Broadcast(coordinator, $I$)          // Broadcast coordinator, $I$ is new leader
        Procedure Update(ldr, $I$)
    else
        Multicast election message to nodes $i \in$ Candidate $\mid \forall i > I$
        Wait($T_{el,\,i}$)          // Electioneer waits time $T_{el,\,i}$ for all ok messages
        if (any coordinator message received within $T_{el,\,i}$ stating $J$ is new leader) then
            Procedure Update(ldr, $J$)
        else
            if (any ok message received within time $T_{el,\,i}$ ) then
                find highest id node $J$ from the responders
                Broadcast(coordinator, $J$)
                Procedure Update(ldr, $J$)
            else
                Broadcast(coordinator, $I$)
                Procedure Update(ldr, $I$)
            end if
        end if
    end if
end procedure
```

Algorithm 4 ElectnOrd

```
procedure ELECTNORD        // $I \in Ordinary$
    Multicast election message to all nodes $i \in Candidate$
    Wait($T_{el,\,i}$)        // Electioneer node waits time $T_{el,\,i}$ for all ok messages
    if (any coordinator message received within time $T_{el,\,i}$ stating $J$ is new leader) then
        Procedure Update(ldr, I)
    else
        if (any ok message received within time $T_{el,\,i}$) then
            find highest id node $J$ from the responders
            Broadcast(coordinator, J)
            Procedure Update(ldr, J)
        else
            if ((no ok message received within time $T_{el,\,i}$) $\wedge$ ($I \in Ordinary$ is next highest id)) then
                Broadcast(coordinator, I)
                Procedure Update(ldr, I)
            else
                Multicast election message to all nodes $i \in Ordinary \mid \forall i > I$
                Wait($T_{el,\,i}$)
                if (any coordinator message received within time $T_{el,\,i}$ stating $J$ is new leader) then
                    Procedure Update(ldr, J)
                else
                    if (any ok message received within time $T_{el,\,i}$) then
                        find highest id node $J$ from the responders
                        Broadcast(coordinator, J)
                        Procedure Update(ldr, J)
                    else
                        Broadcast(coordinator, I)
                        Procedure Update(ldr, I)
                    end if
                end if
            end if
        end if
    end if
end procedure
```

Algorithm 5 ElectnEFC

```
procedure ElectnEFC
        // A node $I$ has received election messages from other nodes
    Get node id of failed leader node from election message
    if ((no ok message was sent during the last $T_{ok,\,i}$ time) $\wedge$ ($ldr_i$ is the same as that of the election message)) then
        $state_i \leftarrow wait$
        Send ok message to detector node with highest id
        Wait($T_{ok,\,i}$)        // Node $I$ will wait for coordinator message for time $T_{ok,\,i}$
        if (node $I$ receives coordinator message within time $T_{ok,\,i}$) then
        // $J$ is the new leader node, where $J > I$
            Procedure Update($ldr$, $J$)
        else
            Broadcast($coordinator$, $I$)
            Procedure Update($ldr$, $I$)
        end if
    end if
end procedure
```

Algorithm 6 ElectnNRC

```
procedure ElectnNRC          // A node $I$ recovers from failure
    if $I \in$ Candidate then
        Multicast query message to each node $i \in$ Candidate $\mid i > I$
        Wait($T_{ok,\,i}$)          // Node $I$ will wait for answer messages
        if ($I$ receives $\geq 1$ answer messages within time $T_{ok,\,i}$) then
            Get the id $J$ of the current leader node
            if $J > i$ then
                Procedure Update(ldr, J)
            else
                Broadcast(coordinator, I)
                Procedure Update(ldr, I)
            end if
        else
            Broadcast(coordinator, I)
            Procedure Update(ldr, I)
        // As time $T_{ok,\,i}$ expired and all higher id nodes are down
        end if
    else          // $I \in$ Ordinary
        Multicast query message to each node $i \in$ Candidate
        Wait($T_{ok,\,i}$)          // Node $I$ will wait for answer message for time $T_{ok,\,i}$
        if ($I$ receives $\geq 1$ answer messages within time $T_{ok,\,i}$) then
            Get the id $J$ of the current leader node
            Procedure Update(ldr, J)
        else
            Multicast query message to each node $i \in$ Ordinary
        // $T_{ok,\,i}$ expired and all nodes in Candidate are down
            Wait($T_{ok,\,i}$)
            if ($I$ receives $\geq 1$ answer messages within time $T_{ok,\,i}$) then
                Get the id $J$ of the current leader node
                if $J > i$ then
                    Procedure Update(ldr, J)
                else
                    Broadcast(coordinator, I)
                    Procedure Update(ldr, I)
                end if
            else
                Broadcast(coordinator, I)
                Procedure Update(ldr, I)
            end if
        end if
    end if
end procedure
```