An Efficient Approach of Election Algorithm in Distributed Systems

SANDIPAN BASU

Post graduate Department of Computer Science,
St. Xavier's College, 30 Park Street (30 Mother Teresa Sarani), Kolkata – 700016, West Bengal, INDIA.
mail.sandipan@gmail.com

Abstract

In this paper, I have proposed an election algorithm that can be used in distributed systems to elect a coordinator. This is a betterment of an already existing Election algorithm (also known as **Bully Algorithm**) proposed by Hector Garcia-Monila (1982).The proposed algorithm is an efficient approach than Bully Algorithm.

Keywords: COORDINATOR; REQUEST; OK; ELECTION; PROCESS STATUS TABLE; UPDATE.

1. Introduction

In distributed systems, many of the algorithms that have been used are typically not completely symmetrical, and some process has to take the lead in initiating the algorithm. Consequently, it is sometimes necessary that, from a set of processes, a process must be selected as a leader or coordinator. Leader election requires, that all processes agree on a common distinguished process, also termed as a leader. So, to achieve this, several Election algorithms have been proposed so far.
In this algorithm we assumed, that, in the system, every process is assigned a unique priority number and every process knows the priority number of every other process in the system. Every process maintains the process status table. A process status table contains the priority number of each process and its corresponding status in the current system. When the system is initialized then the process with the highest priority number is selected as the coordinator.

2. Existing Algorithms

(1) Hector Garcia-Molina,(1982; also known as **Bully Algorithm**).
(2) Fredrickson and Lynch,(1987).
(3) Singh and Kurose,(1994).

3. Preconditions

The effectiveness of an algorithm depends on the validity of the assumptions that are made. In distributed mutual exclusion environment certain assumptions must be considered to make the work successful. The following assumptions are made for this algorithm:-

(1) All nodes in the system are assigned a unique identification numbers from 1 to N.
(2) All the nodes in the system are fully connected.
(3) On recovery, a failed process can take appropriate actions to rejoin with the set of active processes.
(4) When a process wants some service from the coordinator, the coordinator is bound to response within the fixed time out period; besides its other tasks.
(5) We assume that a failure cannot cause a node to deviate from its algorithm and behave in an unpredictable manner.
(6) Lamport's concept of logical clock is used in distributed system that we are considering.

As we are considering distributed systems, hence, some assumptions also need to make about the communications network. This is very important because nodes communicate only by exchanging messages with each other. The following aspects about the reliability of the distributed communications network should be considered.

(1) Messages are not lost or altered and are correctly delivered to their destination in a finite amount of time; i.e., no communication failure occurs.
(2) Messages reach their destination in a finite amount of time, but the time of arrival is variable.
(3) Nodes know the physical layout of all nodes in the system and know the path to reach each other.
(4) A node never pauses and always responds to incoming messages with no delay.

4. Algorithm

The proposed algorithm works as follows:-

When a process (say) P_i sends a message (any request) to the coordinator and does not receive a response within a fixed timeout period, it assumes that the coordinator has somehow failed. Process P_i refers to its process status table, to see who is process having the second highest priority number. It then initiates an election, by sending an *ELECTION* message to the process (say) P_j, having priority just below the failed coordinator; i.e. process with the second highest priority number.

Here two cases may appear:-

4.1 *Case 1*

When P_j receives an election message (from P_i), in reply, P_j sends a response message *OK* to the sender, informing that it is alive and ready to be the new coordinator. Therefore, P_j will send a message *COORDINATOR* to all other live processes (having priority less than P_j) in the system. Hence, P_i starts its execution from the point where it was stopped.

4.2 *Case2*

If P_i does not receive any response to its election message, within a fixed timeout period; it assumes that process P_j also has somehow failed. Therefore, process P_i sends the election message to the process (say, P_k) having the priority just below the process P_j. This process continues, until P_i receives any confirmation message *OK* from any of the process having higher priority than P_i. It may be the case that, eventually P_i has to take the charge of the coordinator. In that case, P_i will send the *COORDINATOR* message to all other processes having lower priority than P_i.

A different scenario occurs, when a process recovers from its failed state. Consider process P_m recovers from its failed state. Immediately, it sends a *REQUEST* message to any of its live neighbors. The purpose of the *REQUEST* message is to get the process status table from its neighbor. So, as soon as any of P_m's live neighbors receives a *REQUEST* message, it sends a copy of the current process status table to P_m. After receiving the process status table, P_m checks whether its own priority number is less than the process having the highest priority (i.e. current coordinator's priority) or not. Here two cases may appear:-

4.3 *Case1*

If the current coordinator's priority is higher than P_m's priority, in that case, P_m will send its priority number and an *UPDATE* messages to all other processes in the system, to tell them to update P_m's status (from *CRASHED* to *NORMAL*) in their own process status table.

4.4 *Case 2*

If P_m's priority is higher than the current coordinator's priority; then P_m will be the new coordinator and update the process status table and sends the *COORDIANTOR* message to all other processes in the system, and takes over the coordinator's job form the currently active coordinator.

5. Performance Analysis

Now we will look the performance of the proposed algorithm against the Bully algorithm. Consider a system having n processes. In our discussion the following cases may appear.

5.1 Case1: *Failure of the coordinator*

5.1.1 *Worst case*

The process having the lowest priority (in the system) noticed that the coordinator has just crashed.

Bully algorithm

In the above mentioned case, according to the bully algorithm, the lowest priority process initiates an election; altogether *(n-2)* elections are performed one after the other. That is all the processes except the active process with the highest priority and the coordinator process that has just failed, perform elections by sending messages to all other processes with higher priority number. Hence, in the worst case, the bully algorithm requires $O(n^2)$ messages.

Proposed algorithm

In the above mentioned case, according to the proposed algorithm, the number of messages required is ***2***. The 1st message is required to send the *ELECTION* message to the process having priority just below the failed coordinator. And the 2nd message is the *OK* message sends by the process, willing to take over the coordinator to the one who initiates the election.
In addition, in both the above cases *(n-2) COORDINATOR* messages are required to announce, who the current coordinator is.
So, in the worst case the number of messages required in both the algorithm can be tabulated as follows- (eliminating the fact of sending the *COORDINATOR* message to all live processes (lower priority), which is common in both the cases)

Table 1: Performance Analysis-Worst Case I

Algorithm	Worst case
Bully algorithm	**$O(n^2)$**
Proposed algorithm	**2 (constant)**

5.1.2 *Best case*

The process having the priority just below the failed coordinator, detects that the coordinator has failed.

Bully Algorithm
Then according to bully algorithm, it immediately elects itself as the new coordinator and sends *(n-2) COORDINATOR* messages. Hence in the best case, Bully algorithm requires *(n-2)* messages.

Proposed Algorithm
In the same scenario the proposed algorithm also sends only *(n-2) COORDINATO*R messages.

So, in the best case, there is no need to initiate an election algorithm, hence no need exchange any message between nodes (except the *COORDINATOR* messages)

5.2 *Case2*: *A failed process (either a former coordinator or a normal process) recovers from its failed state.*

5.2.1 *Worst case*

Bully Algorithm

In the worst case, the process with the lowest priority has to initiate recovery action, and hence requires ***O*** (n^2) messages.

Proposed Algorithm

In case of proposed algorithm, ***2*** messages are required to acquire the information about the current process status table. And *(n-1)* messages are required to send its priority number to all other processes in the system to update its status. So the total number of messages required is -

$$2 + (n-1) = (n+1).$$

So, if the lowest priority process has recovered from its failure, then the total number of messages required in this case is *(n+1).*
In tabular form it can be expressed as-

Table 2: Performance Analysis- Worst Case II

Algorithms	Worst Case
Bully Algorithm	**O** (n^2)
Proposed Algorithm	**O(n)**

5.2.2 *Best case*

In the best case the bully algorithm requires *(n-1)* processes.
The proposed algorithm requires only *[2+(n-1)]=(n+1)* messages.

Table 3: Performance Analysis- Best Case

Algorithm	Best case
Bully Algorithm	**O (n)**
Proposed Algorithm	**O (n)**

6. Illustrations

The above algorithm can best be explained by an example -

Consider a system, with 10 processes, *P1* to *P10*.*P10* is considered as highest priority process and *P1* is the process with lowest priority process. So, according to the convention, currently *P10* is acting as the coordinator in the entire system.

Table 4: Process Status Table

Process Priority	Status
P1	NORMAL
P2	NORMAL
P3	NORMAL
P4	NORMAL
P5	NORMAL
P6	NORMAL
P7	NORMAL
P8	NORMAL
P9	NORMAL
P10	**COORDINATOR**

Now, suppose, process *P4* wants some service from the coordinator. So, *P4* sends a request to the coordinator *(P10)*.Now, if process *P4* does not receive a response within a fixed period of time, then process *P4* assumes that the coordinator is somehow crashed. Having a look at the current process status table, process *P4* will send an *ELECTION* message to the process having priority just below the failed coordinator's priority (*P9*, in this case).

6.1 Case1

When *P9* receives an election message (from *P4*), it sends a response message *OK* to the sender, informing that it is alive and will take over the charge. Therefore, *P9* will send a message *COORDINATOR* to all other live processes in the system. Consequently, *P4* starts its execution from the point where it stopped. In this case the number of messages require is [1(*ELCETION*) + 1 (*OK*) + 8 (*COORDINAOTR*)] =**10**. Whereas, in the same scenario the bully algorithm would take **43** messages to complete the election procedure.

Table 5: Process Status Table

Process Priority	Status
P1	NORMAL
P2	NORMAL
P3	NORMAL
P4	NORMAL
P5	NORMAL
P6	NORMAL
P7	NORMAL
P8	NORMAL
P9	**COORDINATOR**
P10	*CRASHED*

6.2 Case2

If *P4* does not get any response to its election message within a fixed timeout period; it assumes that process *P9* also has failed. Therefore, it sends the election the election message to the process having the priority just below the process *P9* (*P8*, in this case). This process continues, until *P4* receives any confirmation message *OK* from any of the process having higher priority than *P4*. It may be the case that, eventually *P4* has to take the charge of

coordinator. In that case, *P4* will send the *COORDINATOR* message to all other processes having lower priority than *P4*.

7. Critical comment on Bully Algorithm

In the Bully Algorithm, we know that, a process initiates an election algorithm when it notices that the coordinator has crashed. So, the purpose of initiating the election algorithm is to elect a new coordinator, when an existing one has failed. **So, the election algorithm comes into the act, whenever a new coordinator needs to be elected**. But in another scenario, we observe that, when a failed process (**except a former coordinator**) recovers, it must initiate the election algorithm. So, in the Bully algorithm, the purpose or reason of initiating the election algorithm violates. Because, in the latter case, in spite of having a coordinator in the current system, a failed process (**which was not a coordinator, just before**) initiates an election algorithm. But, in these kinds of cases, there is no need of initiating the election algorithm, rather it is meaningless.

That's why in the proposed algorithm, we suggest that, when a failed process recovers then it must not initiate any election algorithm. It obtains the information of current process status table from its neighbor by sending only *REQUEST* message. Hence, it conserves the purpose of initiating the election algorithm (only when, a coordinator has crashed and the system needs a new coordinator).

8. Conclusion

From the above discussions, we can see that the proposed algorithm overcomes the overhead of sending too many messages between nodes. Hence, it is very clear that the proposed algorithm is a better approach than Bully Algorithm, in terms of the number of messages needed to perform an election and recovery of a failed process.

9. Acknowledgment

The proposed algorithm is based upon the Bully Algorithm. So, some ideas and notions were taken from Bully Algorithm.

References

[1] Tanenbaum A.S, Distributed Operating System, Pearson Education, 2007.
[2] Sinha P.K, Distributed Operating Systems Concepts and Design, Prentice-Hall of India private Limited, 2008.
[3] Attiya H. and Welch J., Distributed Computing Fundamentals, Simulation and Advanced Topics.
[4] Kshemkalyani A.D. and Singhal M., Distributed Computing principles, Algorithms, and Systems, Cambridge University Press.