International Journal on Emerging Technologies **4**(1): 17-25(2013)

ISSN No. (Print) : 0975-8364
ISSN No. (Online) : 2249-3255

Analysis of Mutual Exclusion Algorithms with the significance and need of Election Algorithm to solve the coordinator problem for Distributed System

Dr. S.K. Gandhi and Pawan Kumar Thakur
Department of Computer Science & Engineering,
AISECT University, Bhopal (M.P.) India

(Received 05 November, 2012, Accepted 02 December, 2013)

ABSTRACT: The fundamental concept of distributed systems is the concurrency and collaboration among multiple processes. In many cases, this also means that processes will need to simultaneously access the same resources. To prevent that such concurrent accesses corrupt the resource, or make it inconsistent, solutions are needed to grant mutual exclusive access by processes. In systems that involve multiple processes often utilize critical regions. When a process has to read or update certain shared structures it enters a critical section, performs its operation and leaves the critical section. We use special constructs to serialize access to critical sections (semaphores, monitors). These techniques do not support distributed systems because there is no single shared memory image. We need new techniques to achieve mutual exclusion. There are many algorithm used for distributed mutual exclusion. The main points to consider when evaluating of these algorithms are:

- **What happens when messages are lost?**
- **What happens when a process or node crashes?**

None of the algorithms have described to tolerate the loss of messages, process or node crashes. An algorithm for choosing a unique process to play a particular role is called an election algorithm. The purpose of this paper is to study various mutual exclusion algorithms and find out the drawback of these algorithms with respect to fault tolerance or node failure. An election algorithm is needed for this choice.

Keywords: Mutual Exclusion, critical section, Token, Safety, Liveness, Fairness, Message, centralized, synchronization, messages, Delay, Election.

I. INTRODUCTION

Mutual exclusion makes sure that concurrent process access shared resources or data in a serialized way. The is well known critical section problem. If a process, say Pi, is executing in its critical section, then no other processes can be executing in their critical sections. For example updating a DB or sending control signals to an IO device. In other words, n processes accessing a shared resource such as file on a hard drive, database entry, printer, .etc. It is necessary to protect the shared resource using mutual exclusion. Systems involving multiple processes are often most easily programmed using critical regions. Mutual exclusion algorithms have a big drawback with respect to fault tolerance or node failure. An election algorithm is needed for this choice [1]. In this work section 2 represents the system model, requirements and performance metrics for mutual exclusion, section 3, 4 & 5 provide the various mutual algorithms with their performance and correctness, section 6 represents the comparative study of these algorithm, section 7 represent the need of election.

II. SYSTEM MODEL

The system consists of N sites, S1, S2... SN. Without loss of generality, we assume that a single process is running on each site. The process at site Si is denoted by pi. All these processes communicate asynchronously over an underlying communication network. A process wishing to enter the critical section, requests all other or a subset of processes by sending REQUEST messages, and waits for appropriate replies before entering the critical section while waiting the process is not allowed to make further requests to enter the critical section [1, 2].

A site can be in one of the following three states: requesting the critical section, neither executing the critical section or neither requesting nor executing the critical section (i.e., idle). In the 'requesting the critical section' state, the site is blocked and cannot make further requests for the critical section. In the 'idle' state, the site is executing outside the critical section. In the token-based algorithms, a site can also be in a state where a site holding the token is executing outside the critical section. Such state is referred to as the idle token state.

At any instant, a site may have several pending requests for critical section. A site queues up these requests and serves them one at a time. We do not make any assumption regarding communication channels if they are FIFO or not. This is algorithm specific. We assume that channels reliably deliver all messages, sites do not crash, and the network does not get partitioned [10].

A. Requirements of Mutual Exclusion Algorithms

A mutual exclusion algorithm should satisfy the following properties:

1. Safety Property. The safety property states that at any instant, only one process can execute the critical section

2. Liveness Property. This property states the absence of deadlock and starvation. Two or more sites should not endlessly wait for messages which will never arrive.

3. Fairness. Fairness in the context of mutual exclusion means that each process gets a fair chance to execute the critical section.

B. Performance Metrics

The performance of mutual exclusion algorithms is generally measured by the following four metrics:

1. Message complexity. It is number of messages that are required per critical section execution by a site.

2. Synchronization delay. After a site leaves the critical section, it is the time required and before the next site enters the critical section. Note that normally one or more sequential message exchanges may be required after a site exits the critical section and before next site can enter the critical section.

3. Response time. It is the time interval a request waits for its critical section execution to be over after its request messages have been sent out. Thus, response time does not include the time a request waits at a site before its request messages have been sent out.

4. System throughput. It is the rate at which the system executes requests for the critical section. If SD is the synchronization delay and E is the average critical section execution time, then the throughput is given by the following equation: system throughput=1/(SD+E)[2].

5. Low and high load performance. The load is determined by the arrival rate of critical section execution requests. Performance of a mutual exclusion algorithm depends upon the load and we often study the performance of mutual exclusion algorithms under two special loading conditions, viz., "low load" and "high load". Under low load conditions, there is seldom more than one request for the critical section present in the system simultaneously. Under heavy load conditions, there is always a pending request for critical section at a site.

6. Best and worst case performance. Often for mutual exclusion algorithms, the best and worst cases coincide with low and high loads, respectively. For examples, the best and worst values of the response time are achieved when load is, respectively, low and high; in some mutual exclusion algorithms the best and the worse message traffic is generated at low and heavy load conditions, respectively [4].

III. CENTRALIZED ALGORITHM

The easiest way to achieve mutual exclusion in a distributed system is to simulate how it is done in a one-processor system. One process is elected as the coordinator (e.g., the one running on the machine with the highest network address).

Whenever a process wants to enter a critical region, it sends a request message to the coordinator stating which critical region it wants to enter and asking for permission. If no other process is currently in that critical region, the coordinator sends back a reply granting permission, as shown in Fig. 1.1 (a). When the reply arrives, the requesting process enters the critical region.

(a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted. (b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply. (c) When process 1 exits the critical region, it tells the coordinator, which then replies to 2.

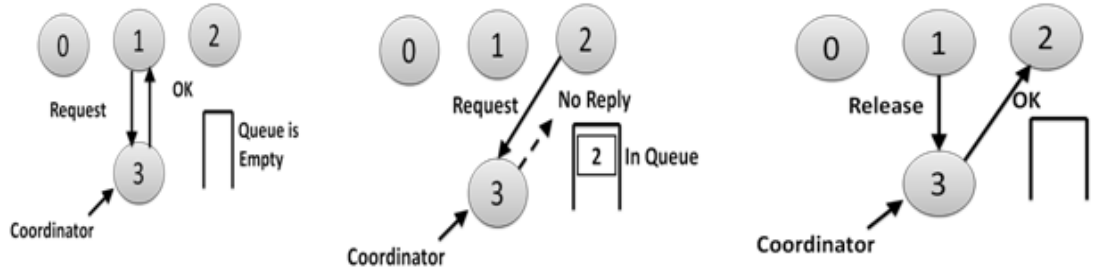

Fig. 1.1(a) **Fig.1.1(b)** **Fig. 1.1(c)**

Fig. 1 .1 Centralized algorithm

Now suppose that another process, 2 in Fig. 1.1 (b), asks for permission to enter the same critical region. The coordinator knows that a different process is already in the critical region, so it cannot grant permission. The exact method used to deny permission is system dependent. In Fig. 1.1 (b), the coordinator just refrains from replying, thus blocking process 2, which is waiting for a reply. Alternatively, it could send a reply saying ‘‘permission denied.’’ Either way, it queues the request from 2 for the time being and waits for more messages [1].

When process 1 exits the critical region, it sends a message to the coordinator releasing its exclusive access, as shown in Fig. 1.1 (c). The coordinator takes the first item off the queue of deferred requests and sends that process a grant message. If the process was still blocked (*i.e.,* this is the first message to it), it unblocks and enters the critical region. If an explicit message has already been sent denying permission, the process will have to poll for incoming traffic or block later. Either way, when it sees the grant, it can enter the critical region.

Correctness. Centralized algorithm achieves mutual exclusion. It is easy to see that the algorithm guarantees mutual exclusion:

(i). The coordinator only lets one process at a time into each critical region. It is also fair, since requests are granted in the order in which they are received. No process ever waits forever (no starvation).

(ii). The scheme is easy to implement, too, and requires only three messages per use of a critical region (request, grant, release).

(iii). It can also be used for more general resource allocation rather than just managing critical regions.

Problems. The centralized approach also has shortcomings.

The coordinator is a single point of failure, so if it crashes, the entire system may go down. If processes normally block after making a request, they cannot distinguish a dead coordinator from ‘‘permission denied’’ since in both cases no message comes back. In addition, in a large system, a single coordinator can become a performance bottleneck.

1. The coordinator can become a performance bottleneck.
2. The coordinator is a critical point of failure:
    - If the coordinator crashes, a new coordinator must be created.
    - The coordinator can be one of the processes competing for access an election algorithm (see later) has to be run in order to choose one and only one new coordinator.

IV. DISTRIBUTED ALGORITHM

Having a single point of failure is frequently unacceptable, so researchers have looked for distributed mutual exclusion algorithms. Lamport’s 1978 [4] paper on clock synchronization presented the first one. Ricart and Agrawala (1981) [5] made it more efficient.

A. Lamport's algorithm

The algorithm works as follows. When a process wants to enter a critical region, it builds a message containing the name of the critical region it wants to enter, its process number, and the current time. It then sends the message to all other processes, conceptually including it. The sending of messages is assumed to be reliable that is, every message is acknowledged. Reliable group communication if available can be used instead of individual messages. When a process receives a request message from another process, the action it takes depends on its state with respect to the critical region named in the message. Three cases have to be distinguished:

Case 1: If the receiver is not in the critical region and does not want to enter it, it sends back an OK message to the sender.

Case 2: If the receiver is already in the critical region, it does not reply. Instead, it queues the request.

Case 3: If the receiver wants to enter the critical region but has not yet done so, it compares the timestamp in the incoming message with the one contained in the message that it has sent everyone. The lowest one wins. If the incoming message is lower, the receiver sends back an OK message. If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing. After sending out requests asking permission to enter a critical region, a process sits back and waits until everyone else has given permission. As soon as all the permissions are in, it may enter the critical region. When it exits the critical region, it sends OK messages to all processes on its queue and deletes them all from the queue. Let us try to understand how the algorithm works. If there is no conflict, it clearly works. However, suppose that two processes try to enter the same critical region simultaneously, as shown in Fig. 1.2(a).

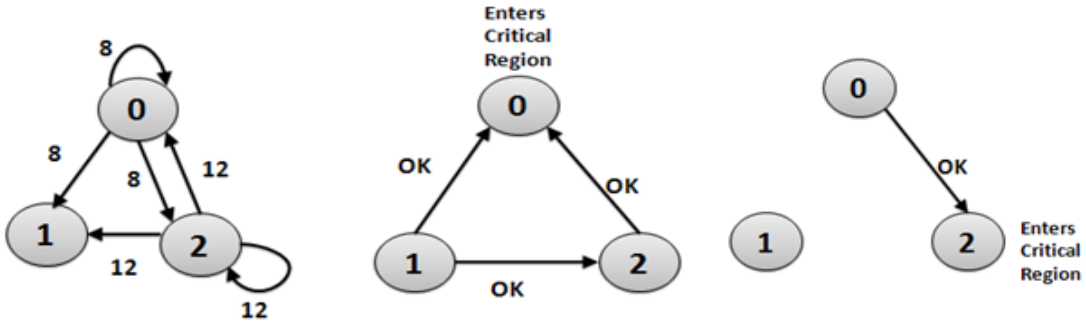

Fig. 1.2(a) **Fig.1.2(b)** **Fig.1.2(c)**

Figure 1.2. Lamport's algorithm

(a) Two processes want to enter the same critical region at the same moment.

(b) Process 0 has the lowest timestamp, so it wins.

(c) When process 0 is done, it sends an OK also, so 2 can now enter the critical region.

Process 0 sends everyone a request with timestamp 8, while at the same time, process 2 sends everyone a request with timestamp 12. Process 1 is not interested in entering the critical region, so it sends OK to both senders. Processes 0 and 2 both see the conflict and compare timestamps. Process 2 sees that it has lost, so it grants permission to 0 by sending OK. Process 0 now queues the request from 2 for later processing and enters the critical region, as shown in Fig. 1.2(b). When it is finished, it removes the request from 2 from its queue and sends an OK message to process 2, allowing the latter to enter its critical region, as shown in Fig. 1.2(c). The algorithm works because in the case of a conflict, the lowest timestamp wins and everyone agrees on the ordering of the timestamps [4].

Note that the situation in Fig.1.2 would have been essentially different if process 2 had sent its message earlier in time so that process 0 had gotten it and granted permission before making its own request. In this case, 2 would have noticed that it itself was in a critical region at the time of the request, and queued it instead of sending a reply.

Correctness. Lamport's algorithm achieves mutual exclusion.

Performance:
(a). For each critical section execution, Lamport's algorithm requires (N − 1) REQUEST messages, (N − 1) REPLY messages, and (N−1) RELEASE messages.
(b). Lamport's algorithm requires 3(N−1) messages per critical section invocation. Synchronization delay in the algorithm is T.
Problems. As with the centralized algorithm discussed above, mutual exclusion is guaranteed without deadlock or starvation. The number of messages required per entry is now 2(n − 1), where the total number of processes in the system is n. (i). Best of all no single point of failure exists. Unfortunately, the single point of failure has been replaced by n points of failure. If any process crashes, it will fail to respond to requests. This silence will be interpreted (incorrectly) as denial of permission, thus blocking all subsequent attempts by all processes to enter all critical regions. Since the probability of one of the n processes failing is at least n times as large as a single coordinator failing, we have managed to replace a poor algorithm with one that is more than n times worse and requires much more network traffic to boot.
(ii). The algorithm can be patched up by the same trick that we proposed earlier. When a request comes in, the receiver always sends a reply, either granting or denying permission. Whenever either a request or a reply is lost, the sender times out and keeps trying until either a reply comes back or the sender concludes that the destination is dead. After a request is denied, the sender should block waiting for a subsequent OK message.
(iii). Another problem with this algorithm is that either a group communication primitive must be used, or each process must maintain the group membership list itself, including processes entering the group, leaving the group, and crashing. The method works best with small groups of processes that never change their group memberships.
(iv). Finally, recall that one of the problems with the centralized algorithm is that making it handle all requests can lead to a bottleneck. In the distributed algorithm, all processes are involved in all decisions concerning entry into critical regions. If one process is unable to handle the load, it is unlikely that forcing everyone to do exactly the same thing in parallel is going to help much. Various minor improvements are possible to this algorithm. For example, getting permission from everyone to enter a critical region is really overkill. All that is needed is a method to prevent two processes from entering the critical region at the same time. The algorithm can be modified to allow a process to enter a critical region when it has collected permission from a simple majority of the other processes, rather than from all of them. Of course, in this variation, after a process has granted permission to one process to enter a critical region, it cannot grant the same permission to another process until the first one has released that permission. Other improvements are also possible, such as proposed by Maekawa (1985), but these easily become more complicated.
(v). This algorithm is slower, more complicated, more expensive, and less robust that the original centralized one. Why bother studying it under these conditions? For one thing, it shows that a distributed algorithm is at least possible, something that was not obvious when we started. Also, by pointing out the shortcomings, we may stimulate future theoreticians to try to produce algorithms that are actually useful.

B. Ricart-Agrawala Algorithm

Ricart and Agrawala algorithm (1981) assumes there is a mechanism for "totally ordering of all events" in the system (e.g. Lamport's algorithm) [4] and a reliable message system. A process wanting to enter critical sections critical section sends a message with (critical section name, process id, current time) to all processes (including itself). When a process receives a critical section request from another process, it reacts based on its current state with respect to the critical section requested. There are three possible cases:
Case 1: If the receiver is not in the critical section and it does not want to enter the critical section, it sends an OK message to the sender.
Case 2: If the receiver is in the critical section, it does not reply and queues the request.
Case 3: If the receiver wants to enter the critical section but has not yet, it compares the timestamp of the incoming message with the timestamp of its message sent to everyone. {The lowest timestamp wins.}

- If the incoming timestamp is lower, the receiver sends an OK message to the sender.
- If its own timestamp is lower, the receiver queues the request and sends nothing.

The process that requires entry to critical section multicasts the request message to all other processes competing for the same resource; it is allowed to enter the critical section when all processes have replied to this message. The request message consists of the requesting process' timestamp (logical clock) and its identifier. Each process keeps its state with respect to the critical section: released, requested, or held [6].

Rule for process initialization
/* performed by each process Pi at initialization */
[RI1]: statePi := RELEASED.
Rule for access request to critical section
/* performed whenever process Pi requests an access to the critical section */
[RA1]: statePi := REQUESTED.
TPi := the value of the local logical clock corresponding to this request.
[RA2]: Pi sends a request message to all processes; the message is of the form (TPi, i), where i is an identifier of Pi.
[RA3]: Pi waits until it has received replies from all other n-1 processes.
Rule for executing the critical section
/* performed by Pi after it received the n-1 replies */
[RE1]: statePi := HELD.
Pi enters the critical section.
Rule for handling incoming requests
/* performed by Pi whenever it received a request
(TPj, j) from Pj */
[RH1]: if statePi = HELD or ((statePi = REQUESTED)
and ((TPi, i) < (TPj, j))) then
Queue the request from Pj without replying.
else
Reply immediately to Pj.
end if.
Rule for releasing a critical section
/* performed by Pi after it finished work in a critical section */
[RR1]: statePi := RELEASED.
Pi replies to all queued requests

Algorithm 1.1
A request issued by a process Pj is blocked by another process Pi only if Pi is holding the resource or if it is requesting the resource with a higher priority[6] (this means a smaller timestamp) then Pj as shown in Fig. 1.3.
Performance:
-For each critical section execution, Ricart-Agrawala algorithm requires (N − 1) REQUEST messages and (N−1) REPLY messages. Thus, it requires 2(N−1) messages per critical section execution.
-Synchronization delay in the algorithm is T.
Problems:
- The algorithm is expensive in terms of message traffic; it requires 2(n-1) messages for entering a critical section: (n-1) requests and (n-1) replies.
-The failure of any process involved makes progress impossible if no special recovery measures are taken.

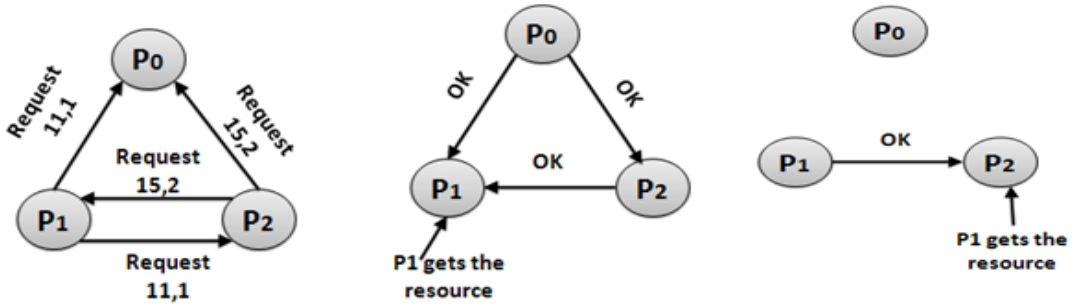

Fig. 1.3 Ricart- Agrawala Algorithm

V. TOKEN RING ALGORITHM

A completely different approach to achieving mutual exclusion in a distributed system is illustrated in Fig. 1.4. Here we have a bus network, as shown in Fig. 1.4(a), (e.g., Ethernet), with no inherent ordering of the processes. In software, a logical ring is constructed in which each process is assigned a position in the ring, as shown in Fig.1.4 (b). The ring positions may be allocated in numerical order of network addresses or some other means. It does not matter what the ordering is. All that matters is that each process knows who is next in line after itself [8].

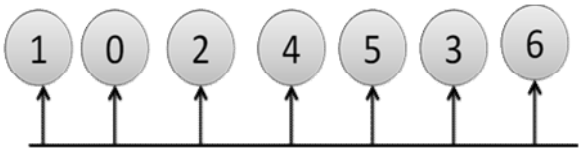

Fig. 1.4(a). Bus Network - Token ring algorithm.

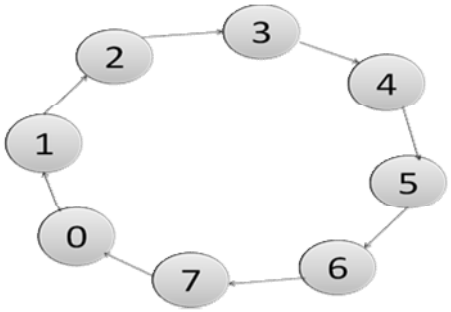

Fig. 1.4(b). Ring Network-Token ring algorithm.

(a) An unordered group of processes on a network. (b) A logical ring constructed in software.

Correctness. The correctness of this algorithm is easy to see.

(i). Only one process has the token at any instant, so only one process can actually be in a critical region. Since the token circulates among the processes in a well-defined order, starvation cannot occur.

(ii). Once a process decides it wants to enter a critical region, at worst it will have to wait for every other process to enter and leave one critical region.

Problem:

(i). If the token is ever lost, it must be regenerated. In fact, detecting that it is lost is difficult, since the amount of time between successive appearances of the token on the network is unbounded.

(ii). The fact that the token has not been spotted for an hour does not mean that it has been lost somebody may still be using it.

(iii). The algorithm also runs into trouble if a process crashes, but recovery is easier than in the other cases.

(iv). If we require a process receiving the token to acknowledge receipt, a dead process will be detected when its neighbor tries to give it the token and fails. At that point the dead process can be removed from the group, and the token holder can throw the token over the head of the dead process to the next member down the line, or the one after that, if necessary. Of course, doing so requires that everyone maintains the current ring configuration.

VI. COMPARISON OF THE MUTUAL EXCLUSION ALGORITHMS

A brief comparison of the mutual exclusion algorithms we have looked at is informative. In Table 1.1 we have listed the algorithms and three key properties:

1. Number of messages. The number of messages required for a process to enter and exit a critical region.

2. Delay before entry. The delay before entry can occur (assuming messages are passed sequentially over a network), and some problems associated with each algorithm. Extreme, the token may sometimes circulate for hours without anyone being interested in it. In this case, the number of messages per entry into a critical region is unbounded.

The delay from the moment a process needs to enter a critical region until its actual entry also varies for the three algorithms. When critical regions are short and rarely used, the dominant factor in the delay is the actual mechanism for entering a critical region. When they are long and frequently used, the dominant factor is waiting for everyone else to take their turn. It takes only two message times to enter a critical region in the centralized case, but 2(n - 1) message times in the distributed case, assuming that messages are sent one after the other. For the token ring, the time varies from 0 (token just arrived) to n - 1 (token just departed) [9].

3. Event of crashes. All these algorithms suffer badly in the event of crashes. Special measures and additional complexity must be introduced to avoid having a crash bring down the entire system. It is ironic that the distributed algorithms are even more sensitive to crashes than the centralized one. In a fault-tolerant system, none of these would be suitable, but if crashes are very infrequent, they might do.

Table 1. Comparison of mutual exclusion algorithms.

Algorithm	Message Per Entry /Exit	Delay before Entry (In Message Times)	Problems
Centralized	2	2	Coordinator Crash
Distributed	2(n-1)	2(n-1)	Crash of any Process
Token Ring	1 to ∞	0 to n-1	Lost token, Process Crash

The centralized algorithm is simplest and also most efficient. It requires only three messages to enter and leave a critical region: a request, a grant to enter, and a release to exit. The distributed algorithm requires n -1 request messages, one to each of the other processes, and an additional n -1 grant messages, for a total of 2(n- 1). (We assume that -to-point communication channels are used.) With the token ring algorithm, the number is variable. If every process constantly wants to enter a critical region, then each token pass will result in one entry and exit, for an average of one message per critical region entered.

Finally, The *Centralized algorithm* is simple and efficient, but suffers from a single point-of failure. The *Distributed algorithm* has nothing going for it – it is slow, complicated, and inefficient of network bandwidth, and not very robust. The *Token-Ring algorithm* suffers from the fact that it can sometimes take a long time to reenter a critical region having just exited it. All perform poorly when a process crashes, and they are all generally poorer technologies than their non-distributed counterparts. Only in situations where crashes are very infrequent should any of these techniques be considered. The main points to consider when evaluating the above algorithms with respect to fault tolerance are:

- **What happens when messages are lost?**
- **What happens when a process crashes?**

VII. NEED AND SIGNIFICANCE OF ELECTION ALGORITHM

Mutual exclusion distributed algorithms require one process to act as a coordinator or in general perform some special role. None of the algorithms that we have described would tolerate the loss of messages, if the channels were unreliable.

(i). Central algorithm. At initialization or whenever the coordinator crashes, a new coordinator has to be elected. The coordinator is a single point of failure, so if it crashes, the entire system may go down. If processes normally block after making a request, they cannot distinguish a dead coordinator from ‘‘permission denied’’ since in both cases no message comes back. In addition, in a large system, a single coordinator can become a performance bottleneck.

The central server algorithm can tolerate the crash failure of a client process that neither holds nor has requested the token. For example, in the central-server algorithm, if the server fails it must be established whether it or one of the client processes held the token.

(ii). Distributed algorithm. The Ricart and Agrawala algorithm as we have described it can be adapted to tolerate the crash failure of such a process, by taking it to grant all requests implicitly.

Even with a reliable failure detector, care is required to allow for failures at any point (including during a recovery procedure), and to reconstruct the state of the processes after a failure has been detected. The failure of any process involved makes progress impossible if no special recovery measures are taken.

Best of all no single point of failure exists. Unfortunately, the single point of failure has been replaced by n points of failure. If any process crashes, it will fail to respond to requests. This silence will be interpreted (incorrectly) as denial of permission, thus blocking all subsequent attempts by all processes to enter all critical regions. Since the probability of one of the n processes failing is at least n times as large as a single coordinator failing, we have managed to replace a poor algorithm with one that is more than n times worse and requires much more network traffic to boot.

(iii). Token ring algorithm. When the process holding the token fails, a *new process has to be elected which generates the new token*.

If the token is ever lost, it must be regenerated. In fact, detecting that it is lost is difficult, since the amount of time between successive appearances of the token on the network is unbounded. The fact that the token has not been spotted for an hour does not mean that it has been lost somebody may still be using it. The algorithm also runs into trouble if a process crashes.

VIII. CONCLUSION

After studying and comparing all the mutual exclusion algorithms we conclude that it does not matter which process is elected. *What important is that one and only one process is chosen we call this process the coordinator and all processes agree on this decision.* Election is typically started after a failure occurs. The detection of a failure (e.g. the crash of the current coordinator) is normally based on time-out if a process that gets no response for a period of time suspects a failure and initiates an election process. An election process is typically performed in two phases: (a). Select a leader with the highest priority. (b). Inform all processes about the winner. An election algorithm is an algorithm for solving the coordinator election problem.

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]. Lamport, L. (1978). *"Time, clocks, and the ordering of events in a distributed system"*. Communications of the ACM 21 (7): 558–565.doi:10.1145/359545.359563.

[4]. LIN, S.D., LlAN, Q., CHEN, M., , and ZHANG, Z.: *"A Practical Distributed Mutual Exclusion Protocol in Dynamic Peer-to-Peer Systems."* Proc. Third Int'l Workshop on Peer-to- Peer Systems, vol. **3279** of Lect. Notes Compo Sc., (La Jolla, CA). Berlin: Springer-Verlag, 2004. pp. 1] -21. Cited on pages 254, 255.

[5]. Maekawa (1985). Ricart and Agrawala. fully symmetric algorithm: all processes run exactly the same algorithm. improvements by fiddling with messages.

[6]. The Ricart-Agrawala algorithm (RA) [RA81] for achieving mutual exclusion ..... in computer networks. *Comm. ACM*, **24**(1): 9–17, 1981. Corr. ibid. 1981, p.581

[7]. Savio S.H. Tse and Francis C.M. Lau. An approximation solution for the 2-median problem on two-dimensional meshes. Advanced Information Networking and Applications, *International Conference on*, **2**: 457–460, 2005.

[8]. G. Cao and M. Singhal, *"A Delay-Optimal Quorum-Based Mutual Exclusion Algorithm for Distributed Systems"*, *IEEE Transactions on Parallel and Distributed Systems*, Vol. **12**, No. 12, pp. 1256-1268, Dec. 2001.

[9]. Y. Chang, M. Singhal, and Mike Liu, *"A Fault-Tolerant Mutual Exclusion Algorithm for Distributed Systems"*, in the *Proc. of the 9th Symposium on Reliable Distributed Software and Systems*, October 1990, pp. 146-154.

[10]. A. Goscinski, *"Two Algorithms for Mutual Exclusion in Real-Time Distributed Computing Systems"*, *Journal of Parallel and Distributed Computing*, Vol. **9**, 1990.