Journal of Computer Science 5 (5): 398-404, 2009
ISSN 1549-3636

Distributed Mutual Exclusion Based on Causal Ordering

Mohamed Naimi and Ousmane Thiare
Department of Computer Science, University of Cergy-Pontoise,
2, Avenue Adolphe Chauvin 95000 Cergy-Pontoise France

Abstract: Problem statement: Causality among events, more formally the causal ordering relation, is a powerful tool for analyzing and drawing inferences about distributed systems. The knowledge of the causal ordering relation between processes helps designers and the system itself solve a variety of problems in distributed systems. In distributed algorithms design, such knowledge helped ensure fairness and liveness in distributed algorithms, maintained consistent in distributed databases and helped design deadlock-detection algorithm. It also helped to build a checkpoint in failure recovery and detect data inconsistencies in replicated distributed databases. **Approach:** In this study, we implemented the causal ordering in Suzuki-Kasami's token based algorithm in distributed systems. Suzuki-Kasami's token based algorithm in distributed algorithm that realized mutual exclusion among n processes. Two files sequence numbers were used one to compute the number of requests sent and the other to compute the number of entering in critical section. **Results:** The causal ordering was guaranteed between requests. If a process P_i requested the critical section before a process P_j, then the process P_i will enter its critical section before the process P_j. **Conclusion:** The algorithm presented here, assumes that if a request req was sent before a request req's, then the request req will be satisfied before req's.

Key words: Causal ordering, distributed mutual exclusion, consistent distributed database

INTRODUCTION

The mutual exclusion problem states that only a single process can be allowed access in its Critical Section (CS). Hence, the mutual exclusion problem plays an important role in the design of computer systems. Several distributed algorithms are proposed to solve this problem in distributed systems and based on asynchronous messages passing and without global clock. Distributed mutual exclusion can be divided into two groups: Permission-based algorithms and token-based algorithms.

In the first class Permission-Based Algorithms[2,8,10,16,19,20], where all involved processes vote to select one which receives the permission to access the CS. Lamport[8] was the first to design a fully distributed permission based mutual exclusion algorithm using logical timestamps. In his algorithm, each request se is the entire distributed system. Then, if n is the number of processes in the distributed system, the algorithm requires (n-1) request, (n-1) reply and (n-1) releases. The algorithm requires 3(n-1) messages per critical section execution. Ricart and Agrawala[18] have reduced the number of messages in Lamport's algorithm to 2(n-1). Carvalho and Roucairol's algorithm[2] has further improved the number of messages in Ricart and Agrawala's algorithm by avoiding some unnecessary request and reply messages. They have shown that the number of messages exchanged in their algorithm is between 0 and 2(n-1). In[10], Maekawa uses the quorum principle to solve the distributed mutual exclusion and reduces the number of messages from O(n) to O(√n).

In the second class, token-based algorithms[1,3,4,13-16,21-24], in which only one process holding a special message called the token, may enter the critical section. The dynamical spanning tree is presented in[22,23] to ensure the mutual exclusion. The reversal path permits to reduce the number of messages to log(n)[6,7,9,12], where n is the number of processes in the network. The performance metrics of the mutual exclusion algorithms are: The average number of messages necessary per critical section invocation, the response time, the fault tolerance. The mutual exclusion algorithm should be starvation-free and fairness.

MATERIALS AND METHODS

Definition of Causality: Causal ordering of events in a distributed system is based on the well-known "happened before" relation noted →[8]. The "happened

Corresponding Author: Ousmane Thiare, Department of Computer Science Gaston Berger University, UFR/SAT BP. 234 Saint-Louis Senegal

before" relation → defined by Lamport is defined by the following three rules:

- If a and b are events in the same process and a comes before b, then a → b
- If a is the sending of a message by one process and b is the receipt of the same message by another process, then a → b
- If a → b and b → c, then a → c

Note that → is ir-reflexive, asymmetric and transitive, i.e., it is a strict partial order. The → relation is also referred to as the causality relation in[8].

Lamport describes a mechanism for total ordering of events in a distributed system. It is based on logical clocks and requires each site to have at least one message from every other site in the system. Causal ordering is a weaker ordering than total ordering. Causal ordering of the events a and b means that every recipient of both a and b receive message a before message b. Since there is no global clock in distributed systems, information is added to the messages to indicate the knowledge of other messages in the system that were sent before it. A message is said to depend upon other messages in the system that were sent before it and a message cannot be delivered until all messages that it depends upon have been delivered. The transitive closure of this relation denotes the "transitive dependencies" or "dependency chain". A convenient way to visualize distributed computations is with time diagram. Figure 1 shows an example for a system comprising three processes. A directed line symbolizes the progress of each process.

On Fig. 1, the causal ordering is not guaranteed. Message m_1 is sent before message m_2, but the process P_3 receives the message m_2 before m_1:

$e_{11} \rightarrow e_{21} \rightarrow e_{22} \rightarrow e_{32} \rightarrow e_{33}$

From $e_{11} \rightarrow e_{21} \rightarrow e_{22}$, we deduce that $e_{33} \rightarrow e_{22}$. The events $e_{22} \rightarrow e_{31} \rightarrow e_{32} \rightarrow e_{23}$ and $e_{32} \rightarrow e_{12} \rightarrow e_{13}$. From $e_{22} \rightarrow e_{31} \rightarrow e_{32,}$ we deduce that $e_{13} \rightarrow e_{12}$.

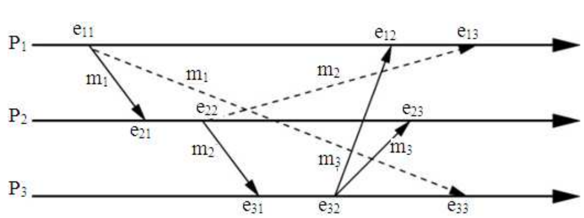

Fig. 1: The causal ordering is not guaranteed

On Fig. 2, a request is sent to P_1 to all other at e_{11}. This request is received and stored by P_2 (e_{21}) and received by P_3 (e_{33}). When process P_2 requests the critical section, it sends all waiting requests stored in its fifo queue (the request of P_1 is placed before the request of P_2). Process P_3 holds the token and receives a request from P_2 (e_{31}). The process P_3 sends the token to process P_1 and not to process P_2.

Logical time approaches: In the literature, two types of causal ordering protocols were found: Logical clock based and physical clock based. By far, the majority of work on causal ordering protocols has been done in the logical clock domain. In fact, only one protocol based on physical clocks was uncovered. Therefore, this study surveys the logical clock mechanisms. In order to describe the protocols, a definition for logical clock must be given.

As defined by Lamport in[8], a clock is away of assigning a number to an event where the number is the time at which the event occurred. Since the clock has no relation to physical time, it is called a logical clock H_i. Counters can implement logical clocks with no actual timing mechanism. A logical clock is correct if it observes the following clock condition: if an event a occurs before another event b, then a should happen at an earlier time than b. In other words for any event a and b: If a → b then H(a) < H(b).

To guarantee that the system of clocks satisfies the clock condition, the following implementation rules are followed:

- Each process P_i increments H_i between any two successive events
  - if event a is the sending of a message m by process P_i, then the message m contains a timestamp $T_m = H_i(a)$
  - Upon receiving a message m, process P_j sets H_j as max(T_m, H_j)

Vectors timestamps: The causal history approach can be improved by observing that for each processor, the causal history is sufficiently characterized by the largest index among its members, i.e., its cardinality.

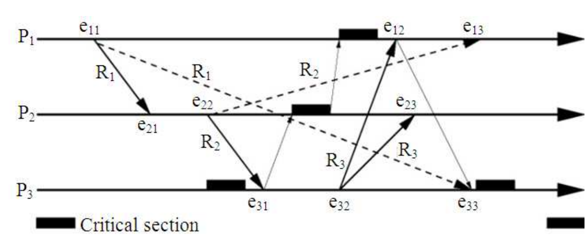

Fig. 2: Mutual exclusion without causal ordering

Thus, the causal history can be uniquely represented by an n-dimensional vector V of integers. A definition for vector time is given in[11]. The vector time V_i of a process P_i is maintained according to the following rules:

- $V_i[k] \leftarrow 0$, for k = 1,…, n processes
- On each internal event e, process P_i increments V_i as follows: $V_i[i] \leftarrow V_i[i]+1$
- On sending message m, P_i updates V_i as in the second point and attaches the new vector to m
- On receiving a message m with attached vector time V, P_i increments V_i as in the second point. Next P_i updates its current V_i as follows: $V_i[k] \leftarrow \max(V_i, V)$

Since there is a correspondence between vector time and causal history, we can determine causal relationships between events by analyzing the vector timestamps of the event in question.

Fidge-Mattern protocol: The protocol refers two protocols by Fidge[5] and Mattern[11] that are similar. This protocol uses a vector of logical clocks to implement causal ordering[17]. In this algorithm, every process maintains a natural number to represent their local clocks. Each process initializes its local clock to 0 and increments it at least once before performing each event. When processes send and receive messages, they pass on whatever local clock information they have to each other. Hence, each process maintains its own local clock information and also whatever local clock information of the other processes it can obtain from received messages. The logical time is defined by a vector of length n, where n is the number of sites in the system. The logical time vector is noted V_i, which represents the logical time on site process P_i and V for the timestamp of message m. The logical time of a site evolves in the following way:

- When a local event occurs at process P_i, the ith entry to the vector V_i is incremented by one: $V_i[i] \leftarrow V_i[i]+1$
- When a site S_i receives a message m, timestamp V, the rules states:
  - For j=i, $V_i[j] \leftarrow V_i[i]+1$
  - For j≠i, $V_i[j] \leftarrow \max(V_i[j], V[j])$

As stated in the discussion on vector clocks, the major drawback of this protocol is the size of the time vectors. If the number of processors is large, the amount of timestamp data that has to be attached to each message is unacceptable.

Suzuki-Kasami's algorithm: The algorithm is presented in[21]. A process holding the token is allowed to enter into the critical section. A single process has the privilege and a node requesting critical section broadcasts a request message to all the other nodes. A process sends the privilege if the toke is idle with the site. The site having token can continuously enter critical section until it sends the token to some other site. The request message has the format request (j, h_j), which means site j is requesting its critical section. Each node maintains an array RN of size N for recording latest sequence number receives from each of the other nodes. The TOKEN message has the format TOKEN (LN), where LN is an array of size N where LN[j] is the latest critical section executed by a node j. if RN[j] = LN[j]+1, it means that a node j has sent a request for its new sequence of critical section and the node having the privilege adds this to the queue and if token is idle, the node sends the TOKEN (LN) to the node requesting critical section. The number of messages per critical section entry is (N-1) REQUEST messages plus one TOKEN message so N messages in all or 0 if the node having the token wants to enter critical section.

- When done with the critical section, process P_i sets $LN_i[i] = RN_i[i]$
- For every process P_j it appends P_j in waiting queue if $RN_i[j] = LN_i[j]+1$
- If the waiting queue is not empty, it extracts the process at the head of the waiting queue and sends the token to that process

Suzuki-Kasami's algorithm based on causal ordering:

Concurrent requests: Let R_i and R_j are two vectors of two processes P_i and P_j respectively.

Definition: For any two time vectors R_i and R_j:

$R_i \le R_j$ iff $R_i \le R_j$ and it exists k such as $R_i[k] < R_j[k]$
$R_i < R_j$ iff $R_i \le R_j$ and it exists k such as $R_i[k] < R_j[k]$
$R_i \parallel R_j$ iff $\neg (R_i < R_j)$ and $\neg (R_j < R_i)$

Principle: To implement the causal ordering, we use, for every process P_i the vector timestamp R_i where $R_i[k]$ is the last request time sent by process P_k and received by P_i. The new requests received by process P_i are stored in a waiting local queue Q_i.

When a process P_i holding the token, requests the critical section, it enters its critical section without sending the message. In another way, it increases $R_i[i]$ by one, appends (i, $R_i[i]$) to Q_i, sends the request "REQ (Q_i)" to all other processes, sets Q_i to empty and waits for the token.

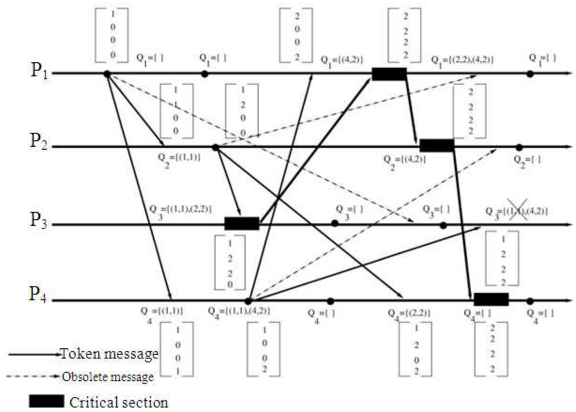

Fig. 3: Mutual exclusion with causal ordering

When a process P_j receives a request "REQ (Q)" from another process, P_i removes from all queues Q_i and Q the obsolete request and appends Q to Q_i to obtain by merging a queue Q_i. A process P_i holding the idle token, sends it to the head of its waiting local queue Q_i and sets Q_i to empty.

Approach:

Example: In Fig. 3 we consider a distributed system $\{P_1, P_2, P_3, P_4\}$, the process P_3 holds the token. We consider the following scenario:

T_0: The process P_3 requests the critical section and enters its critical section, without sending the request message.

T_1: Process P_1 requests the critical section, it increases its logical time $R_i[i]$ by one, appends (1, $R_1[1]$) to its waiting queue Q_1, sends "REQ (Q_1)" to others processes, sets Q_1 to empty and waits for the token.

T_2: Process P_2 receives the request "REQ (Q)" from P_1. The process P_2 deletes from Q_2 and Q the obsolete request, afterwards, it appends Q to Q_2.

T_3: Process P_4 receives the request "REQ (Q)" from P_1. The process P_4 deletes from Q_4 and Q the obsolete request, afterwards, it appends Q to Q_4.
$R_1=(1,0,0,0)$, $R_2=(1,1,0,0)$, $R_4=(1,0,0,1)$, $R_1< R_2$ and $R_1< R_4$ but we have $R_2 \parallel R_4$.

T_4: Process P_4 requests the critical section, it increases its logical time $R_4[4]$ by one, appends (4, $V_4[4]$) to its waiting queue Q_4, sends "REQ (Q_4)" to others processes, sets Q_4 to empty and waits for the token.

T_5: Process P_2 requests the critical section, it increases its logical time $R_2[2]$ by one, appends (2, $V_2[2]$) to its waiting queue Q_2, sends "REQ (Q_2)" to others processes, sets Q_2 to empty and waits for the token.

T_6: Process P_3 receives the request from P_2. Process P_3 holds the token, but it uses it. The process P_3 deletes from Q the obsolete requests; afterwards, it appends Q to Q_4. $Q_4= \{(1, 1), (2, 2)\}$.

T_7: Process P_1 receives the request from P_4. The process P_1 deletes from Q the obsolete requests; afterwards, it appends Q to Q_1. $Q_1= \{(4, 2)\}$.

T_8: The process P_3 releases the critical section, sends the token message "TOKEN (Q_4)" to the head of Q_4 and sets Q_4 to empty.

T_9: Process P_1 receives the request from P_2. The process P_1 deletes from Q the obsolete requests; afterwards, it appends Q to Q_1. $Q_1= \{(4, 2), (2, 2)\}$.

T_{10}: Process P_1 receives the token message "TOKEN (Q)" from P_3. The process P_1 deletes from Q_1 the obsolete requests, afterwards, it append P_1 to Q.
$Q_1= \{(4, 2), (2, 2)\}$. When the process P_1 releases its critical section, it sends the token to the process P_4.

Definition: A request with timestamp (i, h) is said obsolete if for all k, we have ($h\le R_k[i]$) or ($h\le T[i]$), where $R_k[i]$ and T[i] are the vector timestamps of requesting and entering the critical section by process P_i.

Local variable at process P:

R_i: Vector of timestamps where $R_i[i]$ denotes the last timestamp of requesting critical section by process P_i.

T: Vector of timestamps where T[i] denotes the last timestamp critical section execution by process P_i.

Q_i: Waiting Fifo queue of (j, h_j) where j is the process P_j and h_j is the timestamp request.

HT_i: Boolean true if process P_i holds the token, false otherwise. Initially one process holds the token.

$InCS_i$: Boolean true if process P_i is in the critical section and false otherwise.

$Next_i$: Pointer denotes the next process to which, the token will be sent.

Messages of the algorithm: We consider two kinds of messages exchanged between processes:

REQ (Q): This message is sent to all others process to obtain the token.

TOKEN (Q, T): This message to denote the permission to enter the critical section.

Algorithm: We define the concatenation operator "*" as follows: the operator "*" merges the waiting received Q and local Q_i and we denote it by "Q*Q_i". We consider the two following cases:

- When a process P_i receives waiting queue Q attached to token message, it deletes from Q_i all obsolete messages. For all (k, h) ϵ Q such than (k, h') ϵ Q_i, remove (k, h) from Q_i
- When a process P_i receives waiting queue Q attached to request message, it deletes from Q and Q_i all obsolete messages

Rule$_1$: P_i requests the critical section

If (HT_i=False) Then
    $R_i[i] \leftarrow R_i[i] +1$
    $Q_i \leftarrow Q_i$*(i, $R_i[i]$)
    For all k Send REQ (Q_i) To P_k
    $Q_i \leftarrow [\ ]$
EndIf

Rule$_2$: P_i receives REQ (Q)

$Q_i \leftarrow Q_i$*Q
For all k ϵ Q_i $R_i[k] \leftarrow \max (R_i[k], R[k])$
$R_i[i] \leftarrow \max (R_i[k])$

Rule$_3$: P_i receives TOKEN (Q, T)

$HT_i \leftarrow$ True
For all k $R_i[k] \leftarrow \max (R_i[k], T[k])$
$Q_i \leftarrow Q_i$*Q
$InCS_i \leftarrow$ True

Rule$_4$: P_i releases the critical section

$InCS_i \leftarrow$ False
$T[i] \leftarrow R_i[i]$
$Next_i \leftarrow$ Head (Q_i)
If ($Next_i \neq$ Nil) **Then**
    $HT_i \leftarrow$ False
    $Q_i \leftarrow$ **Remove** (Head (Q_i))
    Send TOKEN (Q_i, T) **To** $Next_i$
    $Next_i \leftarrow$ Nil
    $Q_i \leftarrow [\ ]$
EndIf

RESULTS

Correctness and proof of the algorithm:
Theorem: The algorithm based on causal ordering ensures the mutual exclusion.

Proof: To show that the algorithm achieves mutual exclusion, we have to show two or more processes can never be executing critical section simultaneously. Initially, only the process holding the token can enter in its critical section. When a process P_i releases its critical section, it sends the token to only one requesting process at the head in the waiting queue Q_i.

Lemma: For all i, j ϵ [1... n], $R_i[i] \leq T[i] +1$ is an invariant.

Proof: Initially the property is true. We suppose the contrary, $R_i[i] > T[i]+1 \rightarrow R_i[i] - T[i] > 1$, that implies than the process P_i has sent several requests before the token. This is impossible because every process cannot send a new request until it receives the token.

Lemma: For all i ϵ $0 \leq |Q_i| \leq n$ is invariant.

Proof: Initially the property is true. We suppose the contrary, $|Q_i| > n$. That is the file Q_i contains two couples at least (k, h) ϵ Q_i and (k, h') ϵ Q_i. Therefore, they must have $h \leq h'$ or $h' \leq h$, by examining algorithm, this is impossible.

Let Q be a waiting queue of process holding the token.

Lemma: All requests in waiting queue Q respect the causal ordering.

Proof: When a process P_j receives a request REQ (Q) message from another process P_i, it deletes from Q all obsolete requests and appends Q to Q_j. When the process requests the critical section, it increases its vector timestamp by one, appends its request at the end of waiting queue Q_i, sends the request REQ (Q_i) to all other processes.

The processes holding the token will receive either the request REQ (Q) from P_j or a request "REQ (Q_i)" from P_i. In both cases, the process P_j will receive the token before process P_i.

Theorem: If process P_i requests the critical section before process P_j, then process P_i enters its critical section before P_j.

Proof: The causal ordering between two requests is not guaranteed, if for any two requests req (i, h_i) → req (j, h_j), the process P_j receives the token before process P_i. We examine two cases: in the first case, the process P_j receives the request req (Q) from process P_i, this request is put in the waiting queue Q_j. After P_j requests the critical section, puts its request at the end of Q_j after the request req (i, h_i) and we have $h_i < h_j$. In the second case, we assume that there is a process P_k such as it receives the requests req (Q_i, h_i) and req (Q_j, h_j) from P_i and P_j respectively. The process P_k concatenates the two files into its local waiting queue Q_k which contains the request of P_i before that of P_j.

DISCUSSION

The new algorithm for distributed mutual exclusion can be used in several applications which require the causal ordering. Other algorithms can be transformed, according to the same principle.

CONCLUSION

In this study, we have presented a Distributed Mutual Exclusion algorithm based on causal ordering. The causal ordering is guaranteed between requests. If a process P_i requests the critical section before a process P_j, then the process P_i will enter its critical section before the process P_j. The number of messages necessary to satisfy each request is 0 when a process holds the token and n in the other case.

REFERENCES

1. Bernabeu, Auban, J. and M. Ahamad, 1989. Applying a path-compression technique to obtain an efficient distributed mutual exclusion algorithm. Lecture Notes Comput. Sci., 392: 33-44. DOI: 10.1007/3-540-51687-5
2. Carvalho, O. and G. Roucairol, 1983. On mutual exclusion in computer networks. CACM., 26: 146-147.
3. Chang, Y.I., 1996. A Dynamic request based algorithm for mutual exclusion in distributed systems. Operat. Syst. Rev., 30: 52-62. http://cat.inist.fr/?aModele=afficheN&cpsidt=3062377
4. Chang, Y.I., M. Singhal and T. Liu, 1991. A dynamic token-based distributed mutual exclusion algorithm. Proceeding of the 10th International Conference on Computers and Communications, Mar. 27-30, IEEE Xplore Press, Scottsdale, Arizona, USA., pp: 240-246. DOI: 10.1109/PCCC.1991.113817
5. Fidge, C., 1991. Logical time in distributed computing systems. Computer, 24: 28-33. DOI: 10.1109/2.84874
6. Ginat, D, Sleatord, D and R.E. Tarjan, 2003. A tight amortized bound for path reversal. Inform. Process. Lett., 31: 3-5. http://portal.acm.org/citation.cfm?id=63829
7. Giorgetti, A., 2003. An asymptotic study for path reversal Theor. Comput. Sci., 299: 585-602. http://portal.acm.org/citation.cfm?id=782770
8. Lamport, L., 1978. Time, clock and the ordering of events in distributed system. Commun. ACM., 21: 558-565. http://portal.acm.org/citation.cfm?id=359563
9. Lavault, C., 1992. Analysis of an efficient distributed algorithm for mutual exclusion: Average-case analysis of path reversal. Proceedings of the 2nd Joint International Conference on Vector and Parallel Processing, Sept. 1-4, Springer-Verlag,London, UK., pp: 133-144. http://portal.acm.org/citation.cfm?id=703065
10. Maekawa, M., 1985. A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. ACM. Trans. Comput. Syst., 3: 145-159. http://portal.acm.org/citation.cfm?id=214445
11. Mattern, F., 1989. Virtual time and global states on distributed systems. Proceeding of the International Conference on Parallel and Distributed Computing, (ICPDC'89), North-Holland, pp: 215-226. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.7435
12. Naimi, M., M. Trehel and A. Arnold, 1996. A log(n) distributed mutual exclusion algorithm based on path reversal. J. Parall. Distribut. Comput., 34: 1-13. http://cat.inist.fr/?aModele=afficheN&cpsidt=3076194
13. Naimi, M. and M. Trehel, 1987. How to detect a failure and regenerate the token in the log(n) distributed mutual exclusion? Lecture Notes Comput. Sci., 312: 155-166. http://portal.acm.org/citation.cfm?id=674994
14. Neilson, M.L. and M. Mizuno, 1991. A dag based algorithm for distributed mutual exclusion. Proceeding of the 11th IEEE International Conference on Distributed Computer Systems, May 20-24, IEEE Xplore Press, Dallas, pp: 354-360. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=148689
15. Perez, J., 2004. Extending distributed mutual exclusion algorithms to support multithreading. PhD Thesis, Universidad Catolica de Chili. http://ing.utalca.cl/~jperez/papers/perez_issads05.pdf
16. Raynal, M. and M. Singhal, 1996. Logical time: Capturing causality in distributed systems. Computer, 29: 49-56. DOI: 10.1109/2.485846
17. Raynal, M., 1991. A simple taxonomy of distributed mutual exclusion algorithms. ACM Operat. Syst. Rev., 25: 47-50. http://portal.acm.org/citation.cfm?id=122123
18. Ricart, G. and A.K. Agrawala, 1981. An optimal algorithm for mutual exclusion in computer networks. Commun. ACM., 24: 9-17. http://portal.acm.org/citation.cfm?id=358537

19. Saxena, P.C and J. Rai, 2005. A survey of permission-based distributed mutual exclusion algorithms. Comput. Stand. Interfaces, 24: 159-181. http://portal.acm.org/citation.cfm?id=780794
20. Singhal, M., 1993. A taxonomy of distributed mutual exclusion. Journal of Parallel and Distributed Computing, 18: 94-101. http://portal.acm.org/citation.cfm?id=167558
21. Suzuki, I. and Kasami, T. 1982. An optimality theory for mutual exclusion algorithms in computer networks. Proceedings of the 3rd Conference on distributed Computing Systems, Oct. 1982, Miami, pp: 365-370.
22. Trehel, M. and Naimi, M. 1987. Un algorithme distribue d'exclusion mutuelle en log(n). TSI., 6: 141-150.
23. Trehel, M and Naimi, M. 1987. A distributed algorithm for mutual exclusion based on data structures and fault tolerance. Proceeding of the 6th Annual International Phoenix Conference on Computer Communications, Scottsdale, Arizona, USA., pp: 35-39.
24. Van De Snepsheut, J.L.A., 1987. Fair mutual exclusion on a graph of processes. Distribut. Comput., 2: 113-115. http://www.springerlink.com/content/r73257l1r2546213/