American Journal of Applied Sciences 5 (7): 829-834, 2008
ISSN 1546-9239
© 2008 Science Publications

A Dynamic Resource Synchronizer Mutual Exclusion Algorithm for Wired/Wireless Distributed Systems

Ahmed Sharieh, Mariam Itriq and Wafa Dbabat
Department of Computer Science, The University of Jordan, Amman 11942, Jordan

Abstract: A mobile host has small memory, relatively slow processor, low power batteries, and communicate over low bandwidth wireless communication links. Existing mutual exclusion algorithms for distributed systems are not enough for mobile systems because of several limitations. In this study, a mutual exclusion algorithm that is more suitable for mobile computer systems is developed. The algorithm tends to minimize the number of messages needed to be transmitted in the system, by reducing the number of sites involved in the mutual exclusion decision, and reducing the amount of storage needed at different sites of the system.

Key words: Distributed systems, synchronization, mutual exclusion, mobile computing

INTRODUCTION

A mobile computing system is a distributed system consisting of a number of mobiles and fixed processing units. A distributed application consists of a collection of processes executing on a set of computers in the mobile system. The sites do not share any memory and communicate completely by message passing. The wireless communication channels used by the mobile system have a lower bandwidth than the wired communication links. Any distributed mutual exclusion algorithm should take this constraint into consideration[1,7,9].

Distributed mutual exclusion is an important activity that is required to coordinate access to shared resources (usually called critical sections CS) in a distributed system. A set of n processors synchronize their access to a shared resource by requesting an exclusive privilege to access the resource.

The privilege is sometimes represented as a token, where access to the token can represent the ownership of the shared resource. Another method for accessing the shared resource is by requesting to and granting by a central coordinator. Access to the CS can also be based upon the idea of broadcasting and timestamps in networks that supports broadcasting[15].

In a distributed system, the design of a mutual exclusion algorithm consists of defining the protocols used to coordinate access to a shared object. A distributed algorithm for mutual exclusion is characterized by: all processes having an equal amount of information, and all processes making a decision based on local information[15].

Many distributed algorithms for mutual exclusion have been proposed. In Lamport's algorithm[9], each process has a queue. A process that wants to execute a critical section broadcasts a request message with a time-stamp. The return of time-stamped acknowledgements allows it to check whether a process has invoked the critical section earlier than itself.

Several algorithms reducing the number of messages were presented in Ricart and Agrawala[12], Carvalho and Roucairol[3]. The number of messages was proportional with the number of processors-which, and is denoted by N. The algorithm presented by Chandy and Misra[4] (in which the permission is in the form of a fork) is the most efficient of the three in terms of message complexity per resource. Maekawa[10] introduced the notion of arbitrating processes. A process wishing resource access must obtain the permission from a fixed set of $\sqrt{N}$ arbitrating processes. Each arbitrating process gives permission on behalf of itself and ($\sqrt{N}$-1) other processes. Kumar[8] presents a quorum consensus algorithm that requires O($\sqrt{N}$) messages per request.

A solution for the above limitations needs to consider the following assumptions and conditions for the distributed environment:

- All nodes in the system are assigned unique identification numbers from 1 to N
- There is only one requesting process executing at each node. Mutual exclusion is implemented at the node level
- Processes are competing for a single resource

Corresponding Author: Ahmed Sharieh, Department of Computer Science, The University of Jordan, Amman 11942, Jordan

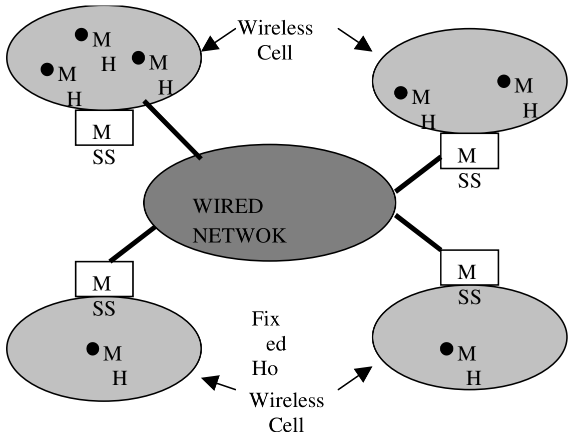

Fig. 1: System model[1]

Table 1: Underling communications network properties

Property	Meaning
Message delivery guaranteed.	Messages are not lost or altered and are correctly delivered to their destination in a finite amount of time.
Message-order Preservation.	Messages are delivered in the order they are sent. There is no message overtaking.
Message transfer delays are finite, but unpredictable.	Messages reach their destination in a finite amount of time, but the time of arrival is variable.
The topology of the network is known.	Nodes know the physical layout of all nodes in the system and know how to communicate with each other under wireless environments.

- At any time, each process initiates at most one outstanding request for mutual exclusion

Also, the aspects as shown in Table 1 about the reliability of the underlying communications network should be considered.

As we mentioned previously, the mobile systems have special constraints that cannot be captured by traditional distributed systems. These constraints are memory limitations, limited battery life, and working under low bandwidth.

In this paper, the proposed algorithm takes in consideration these constraints based on the same system model used in[1] as shown in Fig. 1. A host that can move while retaining its network connections is a MobileHhost (MH). The infrastructure machines that communicate directly with the mobile hosts are called Mobile Support Stations (MSS). A cell is a logical or geographical coverage area under an MSS. All MHs that have identified themselves with a particular MSS, are considered to be local to the MSS.

An MH can directly communicate with an MSS (and vice versa) only if the MH is physically located within the cell serviced by the MSS, and each MH belongs to only one cell at a time.

In method section, we present two versions of the proposed algorithm. In the discussion, we show how the new algorithm guarantees mutual exclusion, and derive the algorithm message complexity. Finally, in conclusion section, we present conclusions and remarks.

MATERIALS AND METHODS

Based on the above system model, we propose a new algorithm for distributed mutual exclusion- which can be used in mobile computing environments. We refer to it as Dynamic Resource Synchronizer (DRS) algorithm, because the node that manages the critical section "synchronizer" is dynamically changed according to certain criteria that reduce message traffic among the nodes.

Assume that the system consists of n independent mobile nodes labeled (N_0, N_1, …, N_n). These nodes communicate by a message passing over a wireless network. Assumptions on the mobile nodes and the network are:

- The nodes have unique node identifiers, (i.e. node i have identifier N_i)
- A node failure does not occur
- Communication links are bi-directional and First In First Out priority
- Communication links failures are predictable- providing a reliable communication
- A partition in network does not occur

Each node in the system is assumed to be running an application whose states are partitioned into four states: WAITING, CRITICAL, SYNCHRONIZER and REMINDER. In the WAITING state, the node has requested access to the CS. In the CRITICAL state, the node is executing the CS. In the SYNCHRONIZER state, the node is currently responsible for handling mutual exclusion access to the CS. There is one and only one node in this state in the system at any moment. Initially one node is set to this state. A node exits the SYNCHRONIZER state if any other node exits CS. In the REMINDER state, the node is neither requesting nor executing the CS. All nodes are initialized to this state. Nodes are in the system cycle through REMINDER to WAITING to CRITICAL to REMINDER to SYNCHRONIZER state. Other nodes do not need to stop executing while one is in a CRITICAL state.

Table 2: Major data structures used in DRS

Data structure	Description
Status	Indicates the state of a node.
Next	Pointer to the process next in the logical ring. Processes are connected to each other forming a logical ring.
Queue	Pointer to the process next in the waiting queue. This pointer is set to nil if the process is the one at the end of the queue or if it is not involved in the queue.
Busy	A Boolean flag used only by the SYNCHRONIZER. It is set to TRUE if and only if any node in the system is currently in CRITICAL state. Initially this is FALSE in all nodes.
Synch	Address of the current SYNCHRONIZER node.
Critical	Address of the node currently in the CRITICAL state. The node uses this variable when it is in the SYNCHRONIZER state.

Table 3: Types of messages and events used in DRS

Message	Description
REQUEST(r,c)	A message sent by a REMINDER node r, that is wishing to be in CRITICAL state, to c.
GRANT(s,q)	A message sent by the SYNCHRONIZER s to the next process q in queue.
RELEASE(s,c)	A message sent by the CRITICAL c to the SYNCHRONIZER s when exiting the CS.
YAS(w,a,s) (You Are synchronizer)	A message sent by the current SYNCHRONIZER s to transfer the synchronization state to new node w.
ADD(q,s)	A message sent by the SYNCHRONIZER s to add a new node q to the queue.
GHANGE(c,s)	A message sent by the SYNCHRONIZER s to the CRITICAL c to inform it of the SYNCHRONIZER state transfer.

The major data structures used by the DRS algorithm are shown in Table 2. There are six types of messages for communication in the system, as shown in Table 3.

The DSR algorithm for mutual exclusion is event-driven. An event at a node consists of receiving a message from another node-or input-from the application on the node to request or release the CS. Modules are assumed to be executed automatically.

Rule 1 (Requesting the CS): When a node wishes to enter the CS, it checks status .If it is at the REMINDER state, it prepares a REQUEST message containing it's address and sends it to next and sets the status to WAITING. The requesting node has no idea of who is the current synchronizer. If status is SYNCHRONIZER and busy is FALSE, it sends a YAS message to next, changes it's status to CRITICAL and enters the CS. If status is SYNCHRONIZER and busy is TRUE, then another node is currently at the CS. Thus, the SYNCHRONIZER adds itself to the end of the queue, sends a YAS message to next and finally changes its status to WAITING.

Rule 2 (handling a REQUEST message): When a REQUEST message arrives at a node, it checks its status. If it is not SYNCHRONIZER it simply forwards the message to next. If the state is SYNCHRONIZER, the node checks busy; if another node is currently using the CS, the SYNCHRONIZER adds the requesting node to the end of the queue. To add a node to the end of the queue, the SYNCHRONIZER checks its queue pointer. If it is not nil, a message is prepared with the address of the requesting node and sent to the node in queue. Then, queue is set to the address of the requesting process, else, the address of the requesting process is stored in queue. If there isn't any node that is currently CRITICAL (busy is FLASE and status is SYNCHRONIZER), a GRANT message is sent to the requesting node, the address of the requesting node is stored in critical and busy is set to TRUE.

Rule 3 (handling a GRANT message): when a node receives a GRANT message from the SYNCHRONIZER, it sets the status to CRITICAL; enters the CS, and saves the address of SYNCHRONIZER in synch.

Rule 4 (exiting the CS): when a node exits the CS, it sends a RELEASE message to the SYNCHRONIZER (using the address stored in synch). Then it changes its status to REMINDER.

Rule 5 (handling a RELEASE message): when the SYNCHRONIZER receives a RELEASE message, it changes its status to REMINDER, sets busy to FALSE, and sends a YAS message to the node that completed the CS (i.e. node from which it received the RELEASE message). Together with the message, the SYNCHRONIZER sends the address of the node currently at the end of the queue (current contents of queue), sending a nil if there is not any nodes currently in the queue.

Rule 6 (handling a YAS message): when a node receives a YAS message-if it is not in the REMINDER state-it forwards the message to next, and sends a CHANGE message containing next to critical(note that it knows critical from the YAS message). However, if the node is currently in the REMINDER state, it should handle the message. First, it changes its status to SYNCHRONIZER. Then, the node checks the contents of queue; if it is not nil a GRANT message is sent to the node in queue and busy is set to TRUE. After that, the node stores the address attached with the message in queue.

Rule 7 (Handling an ADD message): When a node receives an ADD message, it stores the address in the message in a queue. The ADD message is used in Rule 1and in Rule 2 when there is a node at the CS.

Rule 8 (Handling a CHANGE message): when a node receives a CHANGE message, it overwrites its synch to the address in the message.

The rules of the Dynamic Synchronizer Algorithm are shown in Fig. 2.

An illustration of the algorithm is depicted in Fig. 3. Snapshots of the state of the system during algorithm execution are shown, with time increasing from 3(A) to 3(F). The logical ring connections (next) are shown as dashed lines connecting circular nodes.

In Fig. 3A, node N_1 is initially SYNCHRONIZER and all other nodes are in REMINDER state. Node N_2 will send a REQUEST message that will follow the logical ring from N_2 to N_3 to N_4 and then to the SYNCHRONIZER (N_1). When the SYNCHRONIZER receives the REQUEST message, it will check busy. Since there isn't any node currently CRITICAL, it sends a GRANT message to node N_2 , this in turn changes it's state to CRITICAL and enters the CS as in Fig. 3B.

Figure 3C shows the system state after node N_3 has sent a REQUEST message. Since one node (node N_2) is currently CRITICAL, the SYNCHRONIZER will add node N_3 to the queue by sending N_3's address to N_2 in an ADD message The SYNCHRONIZER will store N_3's address in it's queue pointer.

In Fig. 3D the SYNCHRONIZER has received a REQUEST message from node N_5. Since queue is not nil, the SYNCHRONIZER will add node N_5 to the queue and sends an ADD message containing it's address to N_3 (node at the end of the queue). This in turn stores that address in its queue pointer. The SYNCHRONIZER will also change its end of queue pointer by storing N_5's address in its queue pointer.

Figure 3E depicts the system after node N_2 has finished the CS; sent a RELEASE message to SYNCHRONIZER, and changed its status to REMINDER. In Fig. 3F, after SYNCHRONIZER received the RELEASE message, it will send a YAS message to N_2. The address of N_5 (current content of queue pointer in SYNCHRONIZER which points to the node at the end of queue) will be attached with the message. Then the SYNCHRONIZER will change its status to REMINDER. When node N_2 receives the YAS message, it will change its status to SYNCHRONIZER, send a GRANT message to current content of it's queue pointer (node N_3), and store the address attached with the YAS message in queue. Now N_3's status changed to

```
Rule 1: When a node $N_i$ requests access to the CS:
    if status = REMINDER.
            REQUEST(next, $N_i$).
            status = WAITING.
    if status = SYNCHRONIZER and busy = FALSE
            YAS(next, $N_i$, $N_i$).
            status = CRITICAL.
    if status = SYNCHRONIZER and busy = TRUE
            ADD(queue, $N_i$).
            YAS(next, critical, Ni).
            status = WAITING.
Rule 2: When a REQUEST($N_j$, $N_i$) is received by a node $N_j$:
    if status $\neq$ SYNCHRONIZER, REQUEST(next, $N_i$).
    if status = SYNCHRONIZER and busy = FALSE.
            GRANT($N_i$, $N_j$).
            critical = $N_i$.
            queue = $N_i$.
            busy = TRUE.
    if status = SYNCHRONIZER and busy = TRUE.
          ADD(queue, $N_i$).
          queue = $N_i$.
Rule 3: When a GRANT($N_i$, $N_j$) is received, from SYNCHRONIZER, by node $N_i$:
    synch = $N_j$.
    status = CRITICAL.
Rule 4: When a node $N_i$ exits the CS:
    RELEASE(synch, $N_i$).
    status = REMINDER.
Rule 5: When a node $N_j$ receives RELEASE($N_j$, $N_i$):
            YAS($N_i$, nil , queue).
            status = REMINDER.
            busy = FALSE.
Rule 6: When a node $N_j$ receives YAS($N_j$, $N_i$, $N_k$):
    if status $\neq$ REMINDER.
          forward YAS(next, $N_i$, $N_k$)to next.
          CHANGE($N_i$, next).
    if status = REMINDER.
            status = SYNCHRONIZER.
            if queue $\neq$ nil
                    GRANT(queue, $N_j$).
                    busy = TRUE.
                    queue = $N_k$.
Rule 7: When a node $N_i$ receives an ADD($N_i$, $N_j$):
     queue = $N_j$.
Rule 8: When a node $N_i$ receives CHANGE($N_i$, $N_j$).
    synch = $N_j$.
```

Fig. 2: Rules used by the dynamic synchronizer algorithm

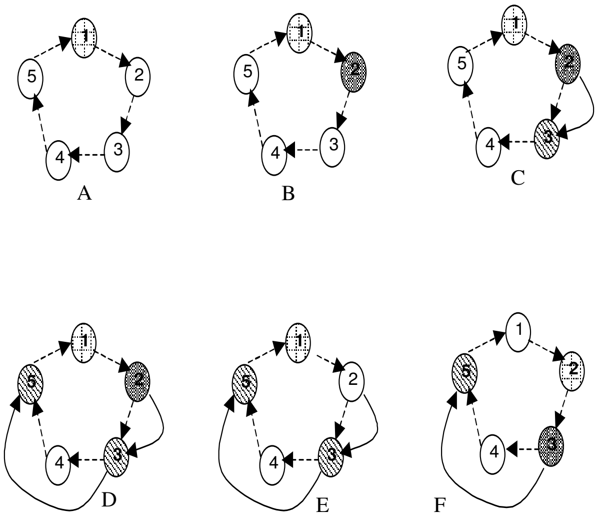

Fig. 3: Snapshot of some operations of the dynamic synchronizer algorithm

CRITICAL and the system have a new SYNCHRONIZER (node N_2).

A slightly different version of the DSR algorithm is presented. This version of the algorithm takes into consideration the fact that it is not desirable in a dynamic system for the same node to be in the SYNCHRONIZER state for a long time, in order to insure fairness in the system. Using this version, the SYNCHRONIZER state will circulate among all the nodes in the system and none of the nodes will remain in SYNCHRONIZER state forever. In this algorithm, a node exits the SYNCHRONIZER state either if any other node exits the CS, or its quantum time is finished. A node that frequently enters the CS will have higher probability to be in the SYNCHRONIZER state.

A new data structure called timer needs to be maintained by the SYNCHRONIZER. Assuming T is the maximum period of time a node may stays in SYNCHRONIZER state, some handling events will be altered, and new events will occur. This version complies with the rules: Rule 1 to Rule 8 for the first version. Rule 6 is modified as follows.

Rule 6 (handling a YAS message): as stated before, plus the timer is set to T

Rule 9 (expiration of timer): if the node is in the SYNCHRONIZER state and the timer value becomes zero, the node prepares a YAS message and sends it to next, attached with the message the address of the node at the end of the queue. A nil address is attached if there isn't any node currently in the queue. A critical is also attached. Then, it stores next in a CHANGE message and sends it to the critical. After that, it changes its status to REMINDER. Rule 9 is added to check the expiration of the quantum time.
According to previous changes, the rules of the algorithm (DRS) will be modified.

RESULTS AND DISCUSSION

We are concern to prove three conditions: the mutual exclusion is satisfied; using DRS will not lead to deadlock or starvation.

To show that the algorithm achieves mutual exclusion, we have to show that two or more nodes can never execute the CS simultaneously. That is, one node exits the CS before any other node can enter the CS. This will be shown by contradiction.

Assume that two nodes N_i and N_j are executing the CS simultaneously. This means that both nodes have received a GRANT message from the SYNCHRONIZER node. But, according to our algorithm, a GRANT massage is sent in two cases: either by the SYNCHRONIZER when it receives a REQUEST message and busy is FALSE (Rule 2), or by a node that receives a YAS message from the SYNCHRONIZER after it exits the CS (Rule 4, Rule 5 and Rule 6). It is clear that in both cases only one GRANT message is sent and that message is sent when busy = FALSE (no other process is currently in the CS). As a result, mutual exclusion is reserved.

The system of nodes is said to be deadlocked when no requesting node can ever proceed to critical section. This can occur as consequence for any of the following situations: either, no node is SYNCHRONIZER or SYNCHRONIZER node is not aware that other nodes have requested the critical section.

As we assumed in the algorithm, one node must be initiated as SYNCHRONIZER. During the time the algorithm is working, Rule 1, 5 and 6 manage the YAS message that is used to transfer the synchronization state from one node to another.

Rule 2 shows how the SYNCHRONIZER becomes aware when other nodes require the grant to enter the critical section. It either stores the address of requesting node in its queue or sends it to the node in its queue when busy is true; which is in turn serves the waiting nodes based on Rule 6, 7 and 8. In Summary, the DRS algorithm is deadlock free.

Starvation occurs when few sites repeatedly execute their CS while other sites wait indefinitely for their turns to do so. It means that there exists a node (call it N_i) that can enter the CS two or more times while another node in the WAITING state (call it N_j) and cannot enter the CS at all. According to Rule 5, when a node N_i exits the CS, it changes it's state to REMINDER, to enter the CS again it must send a REQUEST message. Using Rule 2, if there is any other node in the WAITING state (node N_j), node N_i will be added to the end of the queue after node N_j. So, node N_j will enter the CS before node N_i. Accordingly, there is no starvation.

The number of messages generated-per critical section invocation-has traditionally evaluated the performance of most distributed mutual exclusion algorithms. Also, a useful mutual exclusion algorithm is characterized as fair to all nodes in the distributed system; being starvation-free, and deadlock-free[2,13].

The DRS algorithm results in a substantial reduction in message traffic generated due to executing the CS. The number of messages incurred is much lower than in some other algorithms according to system assumptions that were previously illustrated.

The best-case performance happens when the synchronizer is the immediate neighbor to the sender from the direction of sending, and no one is waiting for the resource. In this case, the number of messages to enter and exit CS is 1(REQUEST) + 1(GRANT) + 1 (RELEASE). This is a constant value of 3 messages. The waiting time in queue in this case = 0.

The worst- case performance happens when the synchronizer is in the far middle of the ring (longest path node), and all other processors want the resource (waiting in queue) and each one of them will use the resource for the longest possible time (max resource use). Thus, the number of messages to enter and exit CS is 1(REQUEST)*(n/2) + (n-1) (GRANT)*(n-1)/2 + (n-1)(RELEASE)*(n-1)/2, which is $O(n^2)$.

The waiting time in queue is (n-1)*(max time for allocating the resource), where n is the number of processors. This is O(n).

The average case performance depends upon the synchronizer location according to the request node. If you assume all nodes have the same chance to be synchronizer at any time, then each node has the probability (1/n) to be synchronizer and its location can be: the immediate neighbor to the sender or the next one, or the far middle in the ring. Then, the number of messages are 3,6,…,3*(n/2-1) respectively. So, the average number of messages equals to (1/n)*□(3*i),I = 1,2,… (n/2)-1 and i is node location. Which ≅ 3(n/2-1). The waiting time equals to the average use time of the resource.

CONCLUSION

The design of algorithms for distributed systems and their communication costs have been based on the assumptions that do not take into consideration the special characteristics of mobile systems such as low bandwidth, limited storage, and constrained energy consumption. This makes existing algorithms no longer valid for mobile systems.

This work focuses on the mutual exclusion problem for mobile systems. A system model for the mobile computing environment is first presented combined with the general principle for structuring algorithms for mobile systems. This differs from token-based algorithms since a token is not used at all, which means there is no token lost problem.

A new algorithm is developed to achieve mutual exclusion in distributed systems is explained (first version). The first version of the algorithm is then updated to a (second version) that takes into consideration the energy savings of mobile hosts. Finally, this work shows how the algorithms are adapted to work in mobile system environments.

Starting from the fact that the distributed algorithms are more sensitive to crashes than centralized ones, we are working in a new version for this algorithm that can work in fault-tolerant systems-even if there are frequent crashes.

REFERENCES

1. Badrinath, B.R., 1994. Structuring Distributed Algorithms for Mobile Hosts, Department of Computer Science, Rutgers University, New Brunswick, NJ, USA.
2. Bernstein, A. and P. Lewis, 1983. Concurrency in Programming and Database Systems. Jones and Bartlett.
3. Carvalho, O. and G. Roucairol, 1983. On Mutual Exclusion in Computer Networks; Comm. ACM, 26: 146-147.
4. Chandy, M. anD J. Misra, 1984. The Drinking Philosopher Problem,; ACM TOPLAS, 6: 632-646.
5. Chang, Y.I., M. Signhal and M.T. Liu, 1990. An Improved O (log(n)) Mutual Exclusion Algorithm for Distributed Systems; Int'l Conference on Parallel Processing, pp: 295-302.
6. Imielinski, T. and B.R. Badrinath, 1992.Querying in Highly Mobile Distributed Environments; 18th Intl. Conference on Very Large Databases, pp: 41-52.
7. Ioannidis, J., D. Duchamp and G.Q. Maguire, 1991. IP-Based Protocols for Mobile Internetworking; Proc. of ACM SIGCOMM Symposium on Communication, Architectures and Protocols, pp: 235-245.
8. Kumar, A., 1991. Hierarchical Quorum Consensus: A New Algorithm for Managing Replicated Data. IEEE Trans. On Computers, 40: 994-1004.
9. Lamport, L., 1978. Time, Clocks and the Ordering of Events in a Distributed system. Comm. ACM, 21: 558-565.
10. Maekawa, M., 1985. An Algorithm for Mutual Exclusion in Decentralized Systems. ACM. TOCS, 3: 145-159.
11. Raymond, K., 1989. A Tree-based Algorithm for Distributed Mutual Exclusion. ACM Trans. On Computer Systems, 7: 61-77.
12. Ricart, G. and A. Agrawala, 1981. An Optimal Algorithm for Mutual Exclusion in Computer Networks. Comm. ACM, 24: 9-17.
13. Singhal, M., 1993. A Taxonomy of Distributed Mutual Exclusion. Journal of Parallel and Distributed Computing, 18: 94-101.
14. Teraoka, F., Y. Yokote and M. Tokoro, 1991. A Network Architecture Providing Host Migration Transparency. Proc. of ACM SIGCOMM'91.
15. Thanebaum, A.S., 1995. Distributed Operating Systems, Printic-Hall, pp: 134-158.