A Permission Based Hierarchical Algorithm for Mutual Exclusion

Mohammad Ashiqur Rahman
Dept. of SIS, University of North Carolina at Charlotte, North Carolina, USA
Email: mrahman4@uncc.edu

Md. Mostofa Akbar
Dept. of CSE, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh
Email: mostofa@cse.buet.ac.bd

Abstract— **Due to the growing application of peer-to-peer computing, the distributed applications are continuously spreading over extensive number of nodes. To cope with this large number of participants, various cluster based hierarchical solutions have been proposed. Cluster based algorithms are scalable by nature. Several of them are quorum based solutions. All of these solutions exploit the idea of coordinator/leader of cluster. Thus, fault tolerance of these algorithms is low. If any coordinator fails, election of new one is required. Here we propose a cluster based network architecture of two layers of hierarchy and present a hierarchical permission based algorithm, which is free of coordinator use. We simulate our proposed algorithm and show that it outperforms related ME algorithms.**

Index Terms— **cluster, distributed algorithm, hierarchical, mutual exclusion, permission, quorum**

I. Introduction

In distributed systems, different processes run on different nodes of a network and they often need to access shared data and resources, or need to execute some common events. The portion of an event or application, where any shared component or common events are accessed, is the Critical Section (CS). Mutual Exclusion (ME) algorithms ensure the consistent execution of the CS by the processes. As the shared memory is absent in distributed systems, the solutions of the ME problem is not straightforward. Due to the enormous importance of ME and the difficulty of its solution, it is an active research area since last three decades. The classic algorithms for Mutual exclusion that have been proposed for fixed networks can be classified in two types: centralized and distributed approaches. In the centralized solutions, a node is designated as coordinator to deliver permission to the other nodes to access the CS, while in the distributed solutions the permission is obtained from consensus among all network nodes.

Distributed mutual exclusion algorithms are generally classified in two categories: token based and permission based [18]. Permission based mutual algorithms [4, 5, 6, 7] impose that a requesting node must receive permissions from other nodes (a set of nodes or all other nodes) to access the CS. In the token-based mutual exclusion algorithms [1, 2, 3], a unique token is shared among the set of nodes. The node holding the token can enter the CS. In the best case, no communication is necessary since the token may be available locally. Otherwise, a mechanism is needed to locate the token. If the node holding the token fails, expensive token regeneration protocols [17] must be executed. Nisho presented a highly resilient, though still complex, token based mutual exclusion algorithm [2]. Naimi proposed an algorithm [3] that takes $O(\log n)$ cost too.

Ricart and Agrawala proposed the first permission based algorithm [4] in 1981, which takes $2(n-1)$ messages for a node to get consensus. To access the CS a node needs consensus from all other $n-1$ nodes in the network.

Concept of quorum improves the performance of permission-based algorithms to a great extent, where to access the CS a node needs to have permissions only from all of the nodes of a quorum. This kind of algorithms takes lower message cost and is resilient to node and communication failures. Message cost of these algorithms is proportional to the quorum size. Therefore these algorithms try to achieve the two goals: small quorum size with high degree of fault tolerance.

The majority quorum algorithm [15] is the first algorithm of this kind. According to this algorithm, a node must obtain permission from a majority of nodes in the network. Maekawa presented an ME algorithm [5] by forming a *logical structure* on the network. In this scheme, a set of nodes is associated with each node, and this set has a nonempty intersection with all sets corresponding to the other nodes. As the size of each set is $\sqrt{n}$, the algorithm incurs $\sqrt{n}$ order of cost.

Garcia-Molina and Barbara [16] have properly defined the concept of quorums with the notion of *coterie*. A coterie is a set of sets with the property that any two members of a coterie have a nonempty intersection. Combining the idea of logical structures and the notion of

Manuscript received January 30, 2010; revised April 05, 2010; accepted April 18, 2010.
Corresponding author: Mohammad Ashiqur Rahman, Email: mrahman4@uncc.edu

coteries, an efficient and fault tolerant quorum generation algorithm for ME is proposed by Agarwal and Abbadi [6]. Here the nodes form a logical binary tree to generate quorums. The quorum can be regarded as attempting to obtain permissions from nodes along a root–to–leaf path. If the root fails, then the obtaining permissions should follow two paths: one root–to–leaf path on the left subtree and one root-to-leaf path on the right subtree. This algorithm tolerates both node failures and network partitions. In the best case, this algorithm incurs logarithmic cost considering the size of the network. However, the cost increases with the increase of node failures.

At present, the number of distributed nodes has become very large. With the increasing importance of peer-to-peer computing [23] as well as grid computing [24], the distributed applications have been extended over a large number of nodes. The performance of these distributed applications depends on the number of participating nodes. Classical ME algorithms illustrated above do not consider these matters. Hence, we need scalable algorithms that reduce the number of participating nodes.

Sometimes the nodes in a network are divided into several groups where each group is often called a cluster. According to this concepts some hybrid ME solutions [8, 9] are proposed. Actual goal of these proposed algorithms was to combine two different approaches in different layers, intra-group (lower layer) and inter-group (upper layer). Two distributed ME solutions are presented by Erciyes [10] using a logical structure, where clusters are arranged on a ring. Bertier et al. proposed two token-based algorithms in [11] taking into account the hierarchical network topology, which reduce both latency cost and number of message. These three solutions are modification of Naimi's token-based algorithm [3] for proxy-based cluster. As these algorithms are basically token based, they suffer due to token failure. A two-layer permission based algorithm is presented by Rahman et al. in [12], where the coordinators (named as message routers) of the clusters form the upper layer. For quorum formation, tree-quorum algorithm [6] is used in each layer of the algorithm. Rahman and Akbar extended this algorithm in [13] for multilayer clustered network. However, the main problem of these algorithms [12, 13] is the failures of coordinators. Any failed coordinator has to be replaced using a leader election algorithm, which incurs extra cost.

In this paper, we propose a cluster-based two-layer distributed ME algorithm that uses no specific coordinator for a particular cluster, but improves the performance by reducing participating nodes. Fault tolerance of our proposed solution is much higher than the algorithm of Rahman et al. [12] and incurs less communication cost especially when failures occur. We also choose *tree-quorum* algorithm for selecting quorums in both of the layers, so that we can compare the performance of our algorithm with that of Rahman et al.'s algorithm. In Section II we describe the proposed network. Our proposed solution along, its proofs of correctness and liveness, and analysis for optimality are presented in Section III. At the end of this section, we also propose a maintenance algorithm in case of node failures. Next section includes the simulated comparison of the algorithm especially with Rahman et al. Finally, we present a critical evaluation of our algorithm followed by a conclusion.

II. Proposed Network

A. Description of the Environment

We consider an asynchronous distributed system, which follows the model proposed in [20]. Each pair of nodes is connected through a communication channel. Links may fail causing intermittent message drop. The message delays for communication and processing are finite. No assumption is made for the relative speeds of the nodes. A node may fail by stopping or crashing, in accordance with fail-stop model [19]. However, a failed node may restart afterwards (after a reasonably long time), which will be referred as recovery. When a process fails, it loses all its states. However, it may use local stable storage to save some information (that never changes throughout the execution of the algorithm), so that it can retrieve them later for proper execution of ME algorithm.

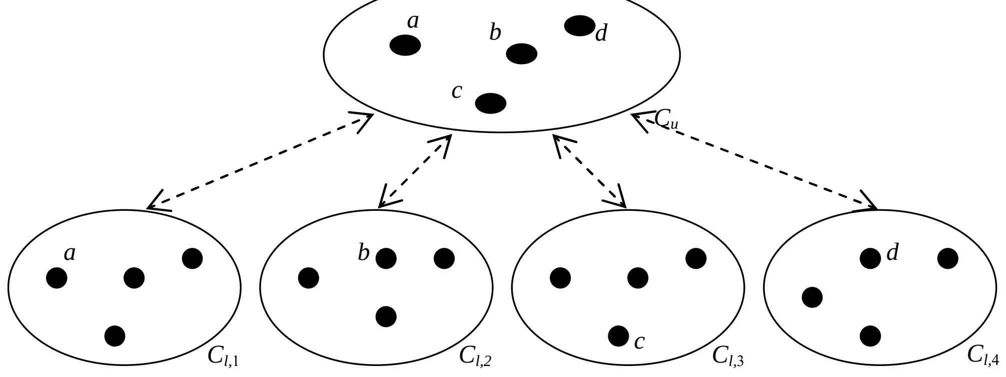


Figure 1. Network architecture of the proposed algorithm

B. Network Architecture

The nodes in a network are partitioned into several nonintersecting groups. Each group is called a cluster. The nodes inside the clusters form the lower layer of nodes. Though any number of arbitrary nodes forms the next (upper) layer, as a matter of guideline, from each of the clusters an arbitrary node is taken for the upper layer. Thus, the number of nodes of this layer is equal to the number of clusters. The upper layer can be considered as a single cluster. In Fig.1, $C_{l,1}$, $C_{l,2}$, $C_{l,3}$ and $C_{l,4}$ are four nonintersecting lower layer clusters. Four nodes a, b, c and d, taken from each of the lower layer clusters respectively, form the upper layer cluster C_u, which is only cluster of the upper layer. Note that, each node is a member of a lower layer cluster; so each member of the upper layer cluster is also a member of one of the lower layer clusters.

Each node knows the members of its lower layer cluster as well as the members of the upper layer cluster. As the memberships to the clusters remain unchanged throughout the program, a node keeps this information in stable storage, so that it can retrieve this data after recovery.

III. Proposed Algorithm

A. Brief Outline

Two layers of nodes participate in our proposed algorithm: the members of a lower layer cluster and the members of the upper layer cluster. A coordination algorithm, as in [12, 13], communicates between these two layers of executions. In both layers, we use the tree-quorum algorithm [6] for quorum creation. When a node *x* wants to access the CS, first it requires to have the consensus from the nodes inside its lower layer cluster. When *x* gets permission at this layer, it needs to have permission from the upper layer. To get this permission, *x* selects an arbitrary node *y* among the members of the upper layer cluster as its *representing node*. To *y*, *x* is identified as the *represented node*. At the upper layer *y* executes the ME algorithm on behalf of *x*. If *y* gets consensus from the members of the upper layer cluster, it informs *x* about the consent. At this point, the node *x* has the total consensus, i.e., permissions from both layers, and can use the CS safely. We apply classical quorum based ME algorithm [5, 6] inside a cluster for collecting consensus.

A. Message and States

As we apply quorum based ME algorithm inside the clusters, our ME algorithm uses *Request*, *Reply*, *Release*, *Inquire* and *Yield* messages. Since some nodes are members of two clusters of both layers, Layer No is added to the contents of these messages to identify the layer (i.e., the cluster) of executing ME algorithm. Possible values of Layer No are only 0 and 1, which identify the upper layer and the lower layer respectively.

The communication between the CS requesting node, a member of a lower layer cluster, and its representing node, a member of the upper layer cluster, is done through *Layer_Request*, *Layer_Reply* and *Layer_Release* messages, similar to [12, 13]. Functions of these messages are briefly described below:

- When a node *x* gets the consensus from a quorum of its lower layer cluster, it sends a *Layer_Request* message to an arbitrary node *y*, its representing node, to process the request in the upper layer cluster.
- If *y* receives consensus inside the upper layer cluster, it sends a *Layer_Reply* message to *x*.
- In order to release the consensus in the upper layer *x* sends a *Layer_Release* message to *y*.

Timestamp, a global logical time [14], consists in each message and denotes the time of sending the message. This timestamp is crucial for *Request* message. As multiple requests can come from different nodes to a single node, a node often needs to keep them in a queue to process afterward. A node keeps the incoming *Request* messages in a minimum priority queue, be identified as QUEUE, according to their timestamps. A node denotes its queues at the upper layer and the lower layer using QUEUE[0] and QUEUE[1] respectively. If timestamps of two *Request* messages are equal, then node identification numbers are used for this determination. This ordering is crucial for avoiding deadlock and starvation. A member node of the upper layer cluster also maintains a normal first-in first-out queue, be identified as FIFO, for *Layer_Request* messages.

Different nodes participate in the algorithm. At an instant, a node, according to its role, owns a state. As some nodes play at both layers, they possess same or different states at these two layers. To represent the proposed system, we define following arrays of states, where each element of an array represents the state of the corresponding node at a particular layer. Size of each array is only two with 0 and 1 indices: 0 for the upper layer and 1 for the lower layer.

- *REQUESTING*[0…1]: A node set this state to true if it sends *Request* messages to its fellow nodes (members of its cluster).
- *LOCKED*[0…1]: This state is set to true when a node sends *Reply* to a fellow *REQUESTING* node. Latter node is the *locking node* of the former.
- *BUSY*[0…1]: When a requesting node of the upper layer (Layer 0) cluster gets consensus from its cluster, its *BUSY*[0] state becomes true. In case of the lower layer (Layer 1), when a requesting node of a lower layer cluster gets consensus from its cluster as well as from the upper layer cluster, its *BUSY*[1] state is set to true.
- *INQUIRING*[0…1]: When a *LOCKED* node wants its consensus back, it sends an *Inquire* message to its locking node. At this point, the node sets the *INQUIRING* state to true.

State *LAYER_REQUESTING* is also defined for the nodes of the lower layer cluster. When a requesting node of a lower layer cluster gets necessary consensus from its cluster, it sends a *Layer_Request* message to its representing node. Its *LAYER_REQUESTING* state, at this time, is set to true. Initially, all the states are false.

We will use ‘a state is set’ or similar languages to mean that the state is set to true. Similarly, we will use ‘a state is unset’ to mean that the state is false and ‘a state is reset’ to mean that the state is set to false. We will also use “a node gets into a state from another state” or similar writings in order to mean that the latter state resets while the former state sets.

C. Algorithm

In order to obtain consensus, a requesting node requires consensus from two separate quorums. Firstly, the requesting node forms a quorum in its lower layer cluster and seeks consensus from the quorum. Next, its representing node seeks consensus by forming a quorum in the upper layer cluster.

Fig. 2 shows the basic state transition diagram of a node at any layer according to the proposed algorithm. In the diagram, two dummy sates IDLE and IDLE2 are used. IDLE denotes the state when both of *LOCKED* and *INQUIRING* states are false, while IDLE2 denotes the state when all of *REQUESTING*, *BUSY* and *LAYER_REQUESTING* states are false. State transitions are labeled with numbers so that we can specify them later.

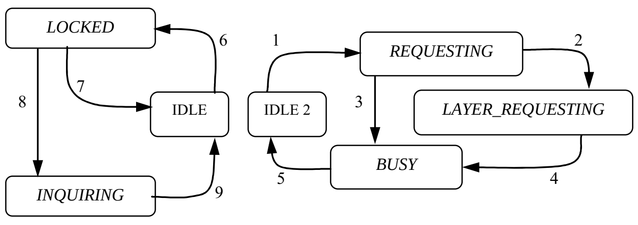


Figure 2. State transition diagram

A node, if its *LOCKED* or *INQUIRING* state is unset, chooses the *Request* message at the head of its QUEUE and sends a *Reply* message, as its consensus, to the requesting node and sets its *LOCKED* state (State transition 6). Once a node exits from its critical section, it sends *Release* messages to all nodes of the lower layer quorum and, through the representing node, to all nodes of the upper layer quorum. After getting the release message, a node resets its *LOCKED* state (State transition 7) and chooses another request residing at top of its QUEUE, if it is not empty, for next processing.

The following points briefly describe the algorithm:

- Requests for the CS are generated at lower layer clusters. These requests are processed sequentially in both layers. The clusters $C_{l,1}$ and C_u in Fig. 1, for example, represent the participating clusters, if a node of $C_{l,1}$ places a request for the CS.
- The CS requesting node x first executes ME algorithm in its lower layer cluster C_l. It selects a quorum q_l and sends a *Request* message to each of the quorum members. Now, its *REQUESTING*[1] state is set (State transition 1). When x receives *Reply* messages from all members of q_l, it has the consensus in this layer. Now it picks an arbitrary node y (as representing node) from the members of the upper layer cluster C_u and sends a *Layer_Request* message to y. At this point, x’s *REQUESTING*[1] state is reset and *LAYER_REQUESTING* state is set (State transition 2).
- When y gets *Layer_Request* from x (as represented node), it queues the message into its FIFO, if it is already processing another *Layer_Request* (i.e., any of its states at this layer is true). Otherwise, it selects a quorum q_u inside C_u and executes ME algorithm on behalf of x. It’s *REQUESTING*[0] state is set at this stage (State transition 1). When y gets *Reply* messages from all q_u members, it has the consensus in C_u. Now it resets *REQUESTING*[0] state and sets *BUSY*[0] state (State transition 3) and sends a *Layer_Reply* message to x.
- When x receives the *Layer_Reply* message from y, it has the total consensus, i.e., permissions from both layers, and so it resets its *LAYER_REQUESTING* state and sets its *BUSY*[1] state (State transition 4) and executes the CS. In this way, a node mutual exclusively enters the CS. After execution of the CS, x resets its *BUSY*[1] state (State transition 5) and sends *Release* messages to the members of q_l. At the same time, it also sends a *Layer_Release* message to y.
- When y gets *Layer_Release* from x, it normally resets its *BUSY*[0] state (State transition 5) and sends release messages to the members of q_u. But if there is one or more *Layer_Request* message in y’s FIFO, y will not leave the consensus. Rather it will choose one from the FIFO and will give consensus directly to that node through *Layer_Reply* message. This reduces the message cost per CS entry. It may seem that this step also reduces the fairness of ME algorithm. As the maximum number of *Layer_Request* messages in C_u is equal to the number of lower layer clusters and as the representing nodes are chosen randomly, fair randomness distributes *Layer_Request* messages among the nodes of C_u equally.

Inquire and *Yield* messages works to avoid deadlock, similarly as in [6, 12, 13]. Let a node u receives a request from node v that possesses an earlier timestamp than its currently processing request of a node w, to which it has already sent *Reply* message. Then u puts the request of v into QUEUE and sends an *Inquire* message to w and waits for either a *Yield* or *Release* message from w. At this time, its *LOCKED* state is reset and *INQUIRING* state is set (State transition 8). When w receives the *Inquire* message from u, it relinquishes the consensus of u as well as sends a *Yield* message to u if and only if it has not received all replies from its requesting quorum members. If w has already acquired all necessary replies to access the CS and may be already executing the CS, then it simply ignores the *Inquire* message and proceeds normally, that is, it continues to execute the CS. After finishing the execution, it sends a *Release* message to the inquiring node u. When u receives the *Yield* message, it resets it *INQUIRING* state (State transition 9) and puts back the request of *w* into its QUEUE. Now it pop out the request from the top of QUEUE and accordingly sends a *Reply* message to corresponding node and gets into *LOCKED* state again (State transition 6). In the mean time if no *Request* with earlier timestamp has come, the *Request* of *y* is the selected request message. If *u* receives *Release* message instead of *Yield* message from *w*, it does the same (State transition 9) except reinserting the request of *w* into QUEUE, since the request has been served already.

D. An Extension to the Algorithm

According to the above description of the algorithm, the progression of the algorithm is sequential since consensus must be ensured in a lower layer cluster before processing starts in the upper layer cluster. Now we extend our proposed algorithm by incorporating parallelism in order to reduce this sequential behavior:

- When a node *x* begins to execute ME algorithm in its associated lower layer cluster, it also sends an in-advance special request, called *Advance_Layer_Request*, to its arbitrary selected representing node *y* at the upper layer. When *y* receives this *Advance_Layer_Request*, it starts to execute ME algorithm in the upper layer cluster without setting its represented node. If *x* gets consensus within its lower layer cluster, it sends *Layer_Request* to *y*. When *y* gets the request, it sets *x* as its represented node. Within the time, *y* may have already received consensus in the upper layer cluster. Otherwise, *y* has reached a point towards getting consensus.
    - If *y* gets consensus before receiving *Layer_Request* from a node of one of the lower layer clusters, it will wait for a threshold period for the request and gets into a new state, named as *WAITING*, from *REQUESTING*[0] state. Within this threshold time, if any *Layer_Request* come, it sets the request sending node as its represented node and sends *Layer_Reply* to that node. It now gets into *BUSY*[0] state from *WAITING* state. If no *Layer_Request* comes (has no represented node), *y* will release this consensus (and resets its *WAITING* state) so that other competing nodes need not to wait long.
- It is common and usual to have other nodes besides *x* in the lower layer clusters competing in order to get consensus. Obviously, each of those nodes will send an *Advance_Layer_Request* to their corresponding representing nodes. It is easily possible for *y* to be selected as representing node by multiple nodes of same or different lower layer clusters. When *y* is in processing for a *Layer_Request* or an *Advance_Layer_Request* message, it may receive more *Layer_Request* or *Advance_Layer_Request* messages. How such a *Layer_Request* handles is described already in the earlier subsection. In case of *Advance_Layer_Request*, the node only counts the number of pending *Advance_Layer_Request* messages in order to process later.
    - Let *x* and *z* both are requesting for consensus in same or two different lower layer clusters. Both of the nodes have selected *y* as their representing nodes. Presently *y* is running ME algorithm in the upper layer cluster in response to *x*'s *Advance_Layer_Request*, since the request of *x* has reached *y* earlier than that of *z*. Thus, *y*'s request is treated as a pending request. However, *z* gets consensus before *x*. So, *z* sends *Layer_Request* to *y*. Now *y* sets *z* as its represented node and the ongoing ME processing continues. Before that, *y* was executing the ME algorithm for an anonymous node of a lower layer cluster (i.e., without having represented node), although the initiator of the processing was the *Advance_Layer_Request* of *x*.

After getting *Layer_Release* from *z*, *y* will begin advance processing again, if its counter shows that there are some pending *Advance_Layer_Request* messages to serve. However, at this time, *x* may get consensus in its lower layer cluster and then it will send a *Layer_Request* to *y*.

E. Proofs

1. Correctness:

A mutual exclusion solution is said to be correct if no more than one node gets simultaneous access to the CS. For quorum based algorithms, this condition holds, if there is at least one common node between any two quorums. In our proposed two-layer ME solution, we must get consensus at each layer: in a lower layer cluster and in the upper layer cluster. So, a quorum of our solution can be defined as $q = q_l \cup q_u$, where q_l and q_u are the quorums formed respectively with the members of a lower layer cluster and with the members of the upper layer cluster.

Similarly, another quorum could be $p = p_l \cup p_u$, where p_l and p_u are the quorums respectively of a lower layer cluster and of the upper layer cluster. Now,

$$p \cap q = (p_l \cap q_l) \cup (p_u \cap q_u)$$

Here, $p_u \cap q_u \neq \phi$, since these are the quorums selected from the members of the same cluster, the only cluster of the upper layer, and these are formed according to the tree-quorum algorithm, which ensures the intersection between any two quorums. So, $p \cap q \neq \phi$. Thus, the solution must maintain mutual exclusion for accessing the CS.

2. Liveness:

To prove the liveness, we need to show that each request is served within a finite period. Let the *C* nodes of a cluster be N_1, N_2, … …, N_C. Consider the worst case scenario where the requests from each of the fellow nodes are queued at N_p. The timestamp of the queued request of

N_i at N_p are T_i, where $1 \le i \le C$. The request sent by N_m contains the maximum timestamp. Thus, $T_{current} > T_m > T_i$, where $T_{current}$ is the current time and $i \neq m$. Let, the timestamp of the next request coming from N_j (after completion of its earlier request with timestamp T_j) is denoted by T_j^{next}. Definitely, $T_j^{next} \ge T_{current}$. Hence, $T_j^{next} > T_m$. Therefore, the request from N_m will be served in a finite duration (after C–1 outstanding requests are served) as it has the earlier timestamp than the next group of requests. So, any request of a node in a cluster of the lower layer or the upper layer must be served in a finite period.

In our two-layer architecture, a requesting node must have consensus at both layers. After getting consensus from its lower layer cluster, the node asks (by sending a *Layer_Request* message) its representing node to collect consensus in the upper layer cluster. Since the representing node processes the *Layer_Request* messages in FIFO manner, every *Layer_Request* will be processed in a finite time. Again, as the request of the representing node in the upper layer cluster must be served, ultimately the request is served in both layers within a finite period. So, liveness is proved.

F. Optimal Cluster Size

To find the optimal cluster size of the proposed solution, we do analysis in this section considering the tree-quorum algorithm [6] that is applied for quorum formation. According to the tree-quorum algorithm, the expected quorum size for a network with size n is expressed as $c_h = f(c_{h-1}+1)+(1-f)(2c_{h-1})$, where f denotes the fraction of the quorums that include the root of the binary tree with height h, while h is approximately $\log n - 1$. So, f is the probability of the root to be included in the quorum, when all the quorums are equally probable. As we assume that the root is included in the quorum if it is available, f is equivalent to the probability of the root being available. In the recursive equation of expected quorum size, each node becomes the root of a subtree in a particular level of the tree. Therefore, f denotes the probability that a node is available at a particular instant, simply, the availability of a node. The average number of messages needed against a single request of a node to enter the CS is proportional to quorum size. Thus c_h, expected quorum size, represents message cost function. Solving this recurrence, following equation is found:

$$c_h = \begin{cases} \dfrac{(2-f)^h - f}{1-f}, & \text{when } f \neq 1 \\ h+1, & \text{when } f = 1 \end{cases} \tag{1}$$

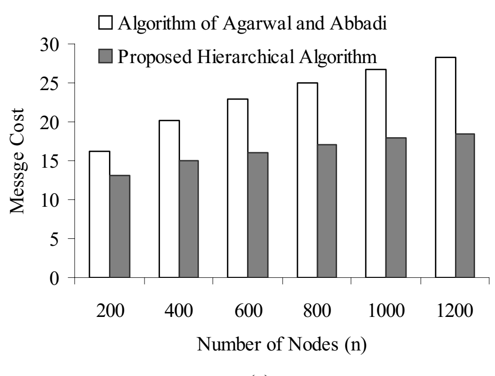
In our algorithm, there are participating clusters in two layers: a lower layer cluster and the upper layer cluster. We consider two parameters: n, the number of nodes in the network and C, cluster size of each lower layer cluster. Therefore, the number of lower layer clusters is n/C and this is cluster size of the upper layer cluster. The height of the tree formed by the nodes inside a lower layer cluster is $h_l = \log C - 1$. Thus, the height of the tree formed by the nodes of the upper layer cluster is $h_u = h - h_l - 1$. Now, the total message cost of the proposed solution is:

$c = c_{h_l} + c_{h_u}$ + number of representing node for coordination

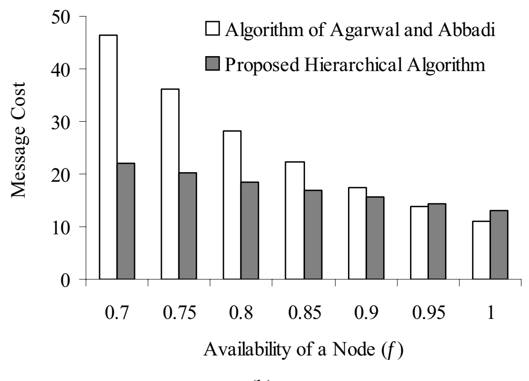
Here, c_{h_l} and c_{h_u} are the message costs at the lower layer and the upper layer respectively. Hence,

$$c = \frac{(2-f)^{h-h_l-1} - f}{1-f} + \frac{(2-f)^{h_l} - f}{1-f} + 1 \tag{2}$$

Taking the derivative of c with respect to h_l and equating it to zero, we get $C = \sqrt{n}$ as the condition of optimality. Then, cluster size of the upper layer cluster is $n/\sqrt{n}$, i.e., $\sqrt{n}$. Thus, cluster size of each cluster is $\sqrt{n}$ for optimality. Fig. 3(a) and Fig. 3(b) show theoretical message cost of our proposed hierarchical algorithm along with that of the ME algorithm of Agarwal and Abbadi [6]. Message cost of proposed algorithm is calculated by (2) taking optimal cluster size into account. Both figures show that the new algorithm analytically outperforms the other. Our simulation result in Section IV also justifies our analytical result. Note that, the message costs shown in the figures are actually

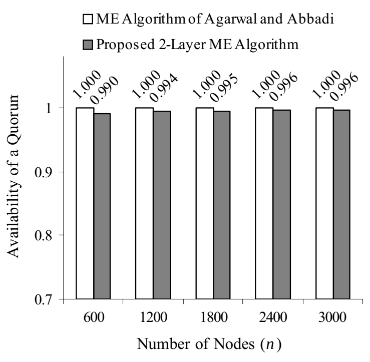


(a)

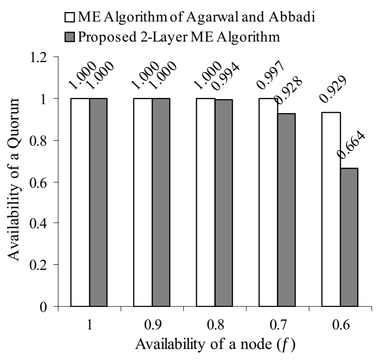


(b)

Figure 3. Message cost of algorithms by (a) varying n with f=0.8 and (b) varying f with n=1200



(a)



(b)

Figure 4. Availability of a quorum in algorithms by (a) varying n with f=0.8 and (b) varying f with n=1200

expected quorum sizes.

B. Availability of a Quorum

The probability that a node is available at any time is f. According to the tree quorum algorithm [6], the availability of a quorum in a cluster is computed by formulating a recurrence relation. The recurrence relation is in terms of the availabilities of forming quorums in the subtrees of a binary tree. Let A_i be the availability of forming a quorum in a tree of height i. Thus, A_{i+1}, the availability of forming a quorum in a tree of height $i + 1$ is given as

A_{i+1} = Probability (root is up) × Availability (Left subtree) × Availability (Right subtree)

+ Probability (root is up) × Unavailability (Left subtree) × Availability (Right subtree)

+ Probability (root is up) × Availability (Left subtree) × Unavailability (Right subtree)

+ Probability (root is down) × Availability (Left subtree) × Availability (Right subtree).

Using f as the probability of the root being up and A_i as the availability of a subtree of height i, we can write the above expression as

$$A_{i+1} = fA^2_i + f(1 - A_i)\, A_i + fA_i\, (1 - A_i) + (1 - f)A^2_i,$$

$$\text{i.e., } A_{i+1} = 2fA_i + (1 - 2f)\, A^2_i.$$

Here, $1 - A_i$ is the unavailability of a subtree of height i. Note that the availability of a quorum in a tree with a single node (height 0) is f, i.e., $A_0 = f$.

In our proposed solution, the number of nodes in the system is n and the number of nodes in a cluster (i.e., cluster size) is C. According to Subsection III.F, all clusters have the same size C (i.e., $\sqrt{n}$) for optimal solution. Letting the height of the binary tree formed by C nodes is h', the availability of a quorum in a cluster is

$$A_{h'} = 2fA_{h'-1} + (1 - 2f)\, A^2_{h'-1}.$$

A quorum q of our solution is defined as $q_l \cup q_u$, where q_l and q_u are quorums formed from the nodes of a lower layer cluster and the upper layer cluster respectively. So the availability of a quorum of our solution is aggregation of the availabilities of 2 quorums. As each cluster has size C (i.e., a binary tree of height h') the availability is

$$A_h = A_{h'} \times A_{h'} = (A_{h'})^2$$

Fig. 4(a) and Fig. 4(b) show the availability of a quorum in the ME algorithm of Agarwal and Abbadi [6] and that in our proposed algorithm varying the number of nodes (n) and availability of a node (f) respectively. For higher values of n and f, especially when f>0.7, quorum availabilities in both algorithms are almost equal. High quorum availability is crucial in order to ensure the access to the CS for each node.

F. Node Failures and Maintenance Algorithm

Maintenance algorithm maintains the correctness of our mutual exclusion algorithm in case of failure of a node that is executing or participating in the ME algorithm. It is assumed that when a node u is interrelated with another node v because of the execution of ME algorithm, then if v fails, u can detect the failure [25][26]. When u detects such failure, it executes the maintenance algorithm. This algorithm along with its correctness proof is illustrated below taking the help of Fig. 5.

1. In case of node failures:

Following two cases are required to handle in case of a node failure:

a. Let x be a member of the upper layer cluster C_u. As a representing node, it is processing the *Layer_Request* of y, a member of a lower layer cluster $C_{l,1}$, in C_u. Some nodes in C_u are in *LOCKED* state due to this request processing. Some other nodes in C_u have x's *Request* waiting in their request queues. Now, x failed. How the maintenance algorithm cancels the engagement

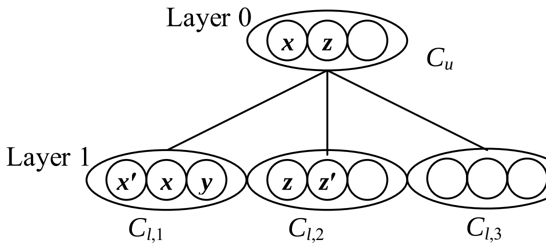


Figure 5. A clustered network showing the clusters

of the nodes with *x* and so keeps the correctness of the algorithm is described as follows:

i. *z* in C_u is in *LOCKED* state by *x*: When *z* detects that its locking node *x* is failed, it withdraws its consensus after waiting for a long enough time to ensure that there is no node of any lower layer cluster in *BUSY* state due to having consensus (*Layer_Reply*) from *x*. After the threshold period, *z* is free to give consent to any pending request.

ii. In C_u, *z* has *Request* from *x* waiting in its QUEUE: After detecting *x*'s failure, *z* discards the request from the QUEUE.

iii. Represented node *y* sent *Layer_Request* to its representing node *x*: If *y* is in *LAYER_REQUESTING* state, i.e., waiting for *Layer_Reply* from *x*, it selects another representing node in C_u and sends *Layer_Request* to its new representing node.

b. Let *x* be a member of a lower layer cluster $C_{l,1}$. It is processing its request in order to access the CS. Some nodes in $C_{l,1}$ are in *LOCKED* state due to this request processing while some other nodes in $C_{l,1}$ have *Request* message of *x* waiting in their request queues. Node *x* may send *Layer_Request* to its representing node *z* at upper layer. Now, *x* is failed. How the maintenance algorithm cancels the engagement of the nodes with *x* and so keeps the correctness of the algorithm is described as follows:

i. *y* in $C_{l,1}$ is in *LOCKED* state for *x*: When *y* detects that its locking node *x* is failed, it withdraws its consensus. Node *y* is now free to give consent to any pending request.

ii. In $C_{l,1}$, *y* has *Request* from *x* waiting in its QUEUE: After detecting *x*'s failure, *y* discards the request from the QUEUE.

iii. *x* sent *Layer_Request* to its representing node *z* in C_u: Action of *z* due to failure of *x* depends on its FIFO. If FIFO is empty, *z* stops processing and sends *Release* messages to all nodes of the quorum in C_u to which it sent *Request* messages. But if FIFO is not empty, the action differs. It does not stop processing; rather it takes out a *Layer_Request* message from its FIFO and makes the message sender its represented node at lower layer. If *z* is in *BUSY* state, it also sends a *Layer_Reply* message to the new represented node.

c. If a node *x* gets consensus from a node *y* through *Reply* (both nodes are at the same layer) or *Layer_Reply* (*x* is at the lower layer and *z* is at the upper layer), it does not relinquish the consensus if *y* fails until it has finished the use of this consensus. (This action keeps correctness along with the delay action stated earlier in a.i and the actions taken after *x*'s recovery stated in next subsection)

2. In case of node recovery:

Let, *x* recovers after its failure. Now it can participate in ME solution starting as an *IDLE* state. But at the time of failure it could be in *LOCKED* state giving consensus to a fellow node *z*. If *z* is still using the consensus of *x*, no way *x* can give consensus to any other fellow node. Again, as a representing node, *x* might be in *BUSY* state at the time of its failure by sending *Layer_Reply* to *y*. So, after recovery, *x* should take care whether *y* is still using its consensus. Former scenario is possible when *x* is a member of a cluster at any layer while the latter scenario is possible only when *x* is a member of the upper layer cluster. How the maintenance algorithm handles the correctness of the algorithm in these scenarios is described below:

- After recovery, if *x* finds that it was in *LOCKED* state at the time of its failure and its locking node was *z*, it sends a message called *Node_Recovery* to *z*. If *z* is not using the consensus of *x*, it sends back a *Release* message to *x* immediately. Otherwise, *z* does not respond to the *Node_Recovery* message. At that time, *x* will receive *Release* message from *z* after a period, when *z* finishes the use of *x*'s consensus. Until getting *Release* message from *z*, *x* will not start to give consensus against any *Request* message. So, the *Request* messages received in this period will be queued in the QUEUE.
- When *x* recovers, it sends a *Node_Recovery* message to its represented node *y* if *x* was in *BUSY* state at the time of its failure. If *y* is not using the consensus of *x*, it sends back a *Layer_Release* message to *x* immediately. Otherwise, *y* does not respond to this *Node_Recovery* message. At that time, *x* will receive *Layer_Release* message from *y* after a period, when *y*'s *BUSY* state becomes false. Similar to previous case, until getting *Layer_Release* message from *z*, *x* will not start to process *Layer_Request*. In the mean time, if any *Layer_Request* comes to *x*, the message is queued in its FIFO.

Remind that when a node recovers it retrieves the information about its clusters and its parent cluster from its stable storage (Subsection II.*B*). We are now taking another assumption to maintain the correctness of the algorithm in case of failure/recovery: Each node keeps the status of its *BUSY* and *LOCKED* states along with the values of its locking node and its represented node saved in stable storage, so that these data can be retrieved after recovery. After recovery all other states and variables are reset, i.e., initialized with their default values.

G. Critical Evaluation

The availability of a quorum in our solution is lower than the original classic algorithm [6] at low *f* along with small *n*, which is depicted in Fig. 4. Though the algorithm is especially suitable for large *n*, it is possible that, in some clusters quorum formation may become impossible due to lack of necessary live nodes. Then the requesting nodes of those clusters will be starved, though there might be enough live nodes in the system. In the simulation (presented in the next section), we assume that no cluster will be in such a situation. For the justification of the assumption, we keep either *n* or *f* or both such high, so that each cluster can have feasible number of live nodes for quorum formation.

In the lower layer, it is possible to associate a node with one or more clusters other than its primary cluster. Then, if the node is unable to form quorum in its cluster, it can utilize other clusters to form quorum. At that time, some issues will be come up to keep consistency. However, we consider this issue as a topic for further research.

IV. Simulation

We simulate our proposed ME algorithm using PARSEC [21], which is a parallel C–based discrete–event simulation language. The aim of the simulation is to compare the performance of our algorithm with that of Rahman et al. [12]. For comparison we take two performance metrics: *message cost* and *waiting time* per CS entry. Message cost is the average message complexity per request to enter into the CS, i.e., the average number of messages required for a node from request placing to getting consensus in order to execute the CS. Waiting time is defined as the average time that a node spends in waiting to enter the CS after its request.

A. Simulation Environment

Our simulation is executed on a peer-to-peer network having an arbitrary number (up to 1200) of nodes randomly spread over the network. The network is assumed to ensure the ordered delivery of messages between a source and a destination. So, the communication latency between two specific nodes is taken constant. The latencies for message communication and message preparation or processing follow normal distribution with a mean of 12 and 8 time units respectively and a common variance of 50% of the mean. The time of CS execution is also taken as a normal random number with a mean of 20. These values are arbitrary. If we change, for example, the mean time of communication latency, the behavior of simulation result for waiting time will remain the same, except, with different scales of magnitudes.

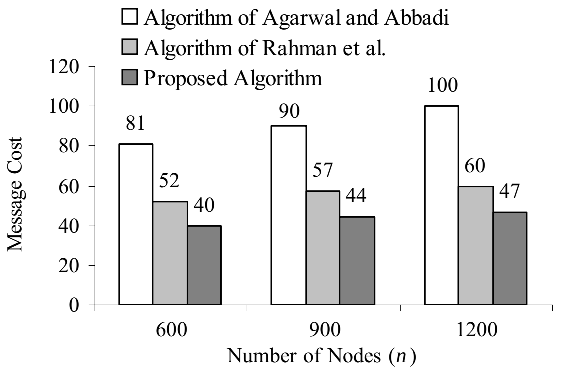
A node requests for the CS following a Poisson process with 0.0000002 request (arrival) rate. Node failures are modeled as a Poisson process with a failure rate calculated from the value of *f* and the recovery time. Recovery time is an exponential distribution with a mean of 1000000 time units.

In the simulation, initially we form the clusters arbitrarily according to the optimal cluster size and select the nodes of the upper layer cluster by taking a random node from each cluster. Elections of the message routers are not simulated for [12]. Rather, a cost (both time and message) is assumed according to a leader election algorithm [22].

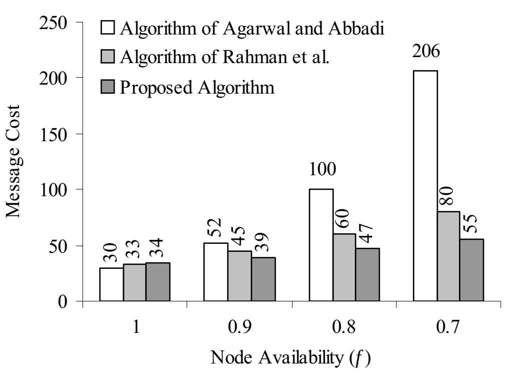
A. Performance Comparison

1. Message cost:

Message costs per CS entry for the classic tree-quorum based ME algorithm of Agarwal and Abbadi [6], the two-layer ME algorithm of Rahman et al. [12] and our proposed two-layer ME algorithm are plotted in Fig. 6(a) and Fig. 6(b) against different network sizes (*n*) and different availability of nodes (*f*) respectively. Both of the figures show that our algorithm simply outperforms the others especially when *f* decreases. According to the tree-quorum algorithm, the quorum size is inversely proportional to the value of *f* and varies from $\log V$ (for high values of *f*) to $V/2$ (for low values of *f*), where *V* is the number of participating nodes. Both of the two-layer algorithms reduce this size of participating nodes to *C*, i.e., $\sqrt{n}$, while the participating nodes in the classical algorithm is *n*. So, both algorithms outperforms the latter when $f<1$. Since Rahman et al. uses coordinators of the clusters, it needs to reelect new coordinator when any coordinator fails. For lower *f*, the rate of reelection increases. As each election of coordinator takes extra message cost, for lower *f* the message cost of Rahman et al. is significantly more than that of our proposed algorithm.

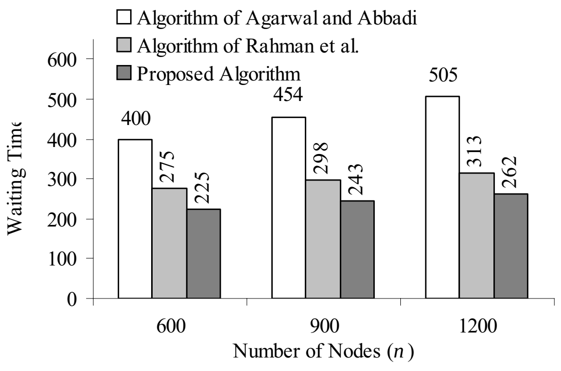


(a)

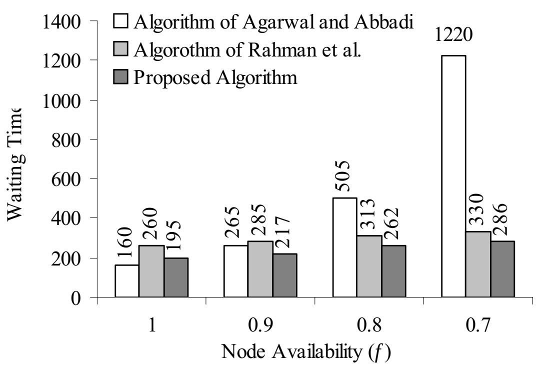


(b)

Figure 6. Message cost of ME algorithms for (a) different *n* with *f*=0.9 and (b) different *f* with *n*=1200



(a)



(b)

Figure 7. Waiting times of ME algorithms for (a) different n with f=0.9 and (b) different f with n=1200

If we decrease the recovery time for a particular f, the failure rate will increase and as a result the performance of Rahman et al. will be worse. The same will occur in case of reduced request rate. For opposite cases, the performance of Rahman et al. will be better.

2. Waiting time:

Average waiting time for a CS entry is plotted in Fig. 7(a) and Fig. 7(b) against different network sizes (n) and different node availabilities (f) respectively. Both of the figures show that our proposed algorithm outperforms ME algorithms of Rahman et al. [12] and Agarwal and Abbadi [6], especially when f<1. Though both algorithms of Rahman et al. and ours use network hierarchy of two layers, the latter takes less waiting time because of its parallel processing.

V. Discussion

We have proposed a two-layer quorum based solution for distributed mutual exclusion. Though the proposed algorithm is cluster based, there is no use of specific coordinator (message router) for a cluster. Thus, no reelection of coordinator is required. We have devised the optimal cluster size taking message cost into consideration. At the end, we have presented a simulation result that demonstrates noticeable improved performance of our algorithm comparing to other related algorithms.

We are currently extending this solution along with some enhancements (described in the earlier section) for multilevel clustered network like [13], which is left for future research publication. Group mutual exclusion (GME) is a recent variant of the classical mutual exclusion problem, which proposed first in [27]. We are also going to extend our hierarchical approach for the solution of GME problems.

References

[1] K. Raymond, "A Tree based Algorithm for Distributed Mutual Exclusion", *ACM Transactions on Computer Systems*, vol. 7, pp. 61–77, February 1989.

[2] S. Nisho, K.F. Li and E.G. Manning, "A resilient mutual exclusion algorithm for computer networks", *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 344–355, 1990.

[3] M. Naimi, M. Trehel, and A. Arnold, "A log (N) distributed mutual exclusion algorithm based on path reversal," *Journal of Parallel and Distributed Computing*, vol. 34, pp. 1–13, April 1996.

[4] G. Ricart and A. K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks", *Communications of the ACM*, vol. 24, pp. 9–17, January 1981.

[5] M. Maekawa, "A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems", *ACM Transaction on Computer Systems*, vol. 3, No. 2, pp. 145–159, 1985.

[6] D. Agarwal and A. El Abbadi, "An Efficient and Fault–Tolerant Solution for Distributed Mutual Exclusion", *ACM Transactions on Computer Systems*, vol. 9, pp. 1–20, February 1991.

[7] P. C. Saxena and J. Rai, "A survey of permission-based distributed mutual exclusion algorithms", *Elsvier Science Publishers B. V.*, vol. 25, pp. 159-181, May 2003.

[8] Q. E. K. Mamun, M. Ali, S. M. Masum and M. A. R. Mustafa, "A Two–Layer Hybrid Algorithm for Achieving Mutual Exclusion In Distributed Systems", *WSEAS Transactions on Systems*, vol. 3, pp. 1193–1198, May 2004.

[9] Ahmed Housni and Michel Trelhel, "Distributed Mutual Exclusion Token-Permission based by Prioritized Groups", *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*, pp. 253–259, June 2001.

[10] K. Erciyes, "Distributed Mutual Exclusion Algorithms on a Ring of Clusters", *ICCSA, SV-Lecture Notes in Computer Science,* 2004.

[11] M. Bertier, L. Arantes and P. Sens, "Distributed Mutual Exclusion Algorithms for Grid Applications: a Hierarchical Approach", *Journal of Parallel and Distributed Computing (JPDC)*, Elsevier, vol. 66, pp. 128-144, 2006.

[12] Mohammad Ashiqur Rahman, Mohammad Shafiul Alam, M. M. Akbar, "A Two Layer Quorum Based Distributed Mutual Exclusion Algorithm", *The 9th International Conference on Computer and Information Technology (ICCIT)*, December, 2006.

[13] Mohammad Ashiqur Rahman, M. M. Akbar, "A Quorum Based Distributed Mutual Exclusion Algorithm for Multi-Level Clustered Network Architecture", *Workshop on Algorithms and Computation (WALCOM), Dhaka*, February, 2007.

[14] L. Lamport, "Time, clocks, and the ordering of events in a distributed system.", *Communications of the ACM*, vol. 21, pp. 558-564, July 1978.

[15] R. H. Thomas, “A majority consensus approach to concurrency control”, *ACM Transaction on Database System*, vol. 4, pp.180-209, June 1979.
[16] H. Garcia-Molina and D. Barbara, “How to Assign votes in a Distributed System”, *Journal of the Association for Computer Machinery*, vol. 32, pp. 841-860, 1985.
[17] J. Misra, “Detecting termination of distributed computations using markers”, *Proceedings of the 2nd ACM Annual Symposium on Principles of Distributed Computing*, pp. 237–249, 1985.
[18] M. Raynal, “A simple taxonomy for distributed mutual exclusion algorithms”, *ACM SIGOPS Operating Systems Review*, vol. 25, pp. 47-50, 1991.
[19] R. D. Schlichting and F. B. Schneider, “Fail-stop processors: an approach to designing fault-tolerant computing systems”, *ACM Trans. on Computing Systems*, vol. 1, pp. 222–238, 1983.
[20] F. Cristian and C. Fetzer, “The timed asynchronous distributed system model”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, pp. 642–627, 1999.
[21] R. Bagrodia, R. Meyerr, and et al., “PARSEC: a parallel simulation environment for complex system”, *UCLA Technical Report*, 1997.
[22] Scott D. Stoller, “Leader election in asynchronous distributed systems”, *IEEE Transactions on Computers*, vol. 49, pp. 283–284, 2000.
[23] R. Steinmetz and K. Wehrle, “Peer-to-Peer Systems and Applications”, *Lecture Notes in Computer Science*, vol. 3485, 2005.
[24] Mark Baker, Rajkumar Buyya and Domenico Laforenza, “Grids and Grid Technologies for Wide-Area Distributed Computing”, *Software: Practice and Experience (SPE), Wiley Press*, vol. 32, pp. 1437-1466, December 2002.
[25] Raimundo Jose and Araujo Macedo, “Failure Detection in Asynchronous Distributed Systems”, *Proceedings of II Workshop on Tests and Fault-Tolerance*, pp. 76-81, 2000.
[26] C. Fetzer: “Perfect Failure Detection in Timed Asynchronous Systems”, *IEEE Transactions on Computers*, vol. 52, pp. 99-112, February 2003.
[27] Y. J. Joung, “Asynchronous Group Mutual Exclusion”, *Proceedings of the 17th A CM Symposium on Principles of Distributed Computing*, pp. 51–60, June 1998.

Mohammad Ashiqur Rahman He received his BSc and MSc in Computer Science and Engineering from BUET, Dhaka, Bangladesh in 2004 and 2007 respectively. His primary research area focused on distributed systems and computing, computer networks, information and network security.

Currently he is a PhD student of the Department of Software and Information Systems, University of North Carolina at Charlotte, USA. Before that, he was working as an Assistant Professor, Dept. of CSE, Ahsanullah University of Science and Technology, Dhaka, Bangladesh.

Md. Mostofa Akbar He received his BSc and MSc in Computer Science and Engineering from BUET, Dhaka, Bangladesh in 1996 and 1998 respectively. He received his PhD from University of Victoria, Canada in 2002. His research interests focus on operations research, distributed systems, computer networks, wireless sensor and mobile ad hoc networks.

Currently he is working as an Associate Professor, Dept. of CSE, BUET.