# A Permission based Multilevel Parallel Solution for Distributed Mutual Exclusion

Mohammad Ashiqur Rahman† and M. Mostofa Akbar‡
†Dept of Software and Information Systems, University of North Carolina Charlotte, USA
‡Dept of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Bangladesh
Emails: mrahman4@uncc.edu, mostofa@cse.buet.ac.bd

***Abstract*—Due to the growing application of peer-to-peer computing, the distributed applications are continuously spreading over an extensive number of nodes. To cope with this large number of participants, various cluster based hierarchical solutions have been proposed. Cluster or group based solutions are scalable for a large number of participants. All of these solutions exploit the idea of coordinators, leaders or proxies of the clusters. If any such node fails, the election of a new one is required. Thus, fault tolerance of these algorithms is low. Again, as the number of participating nodes increases every day, it is necessary to devise highly scalable distributed mutual exclusion algorithms. This research presents a permission based parallel solution of distributed mutual exclusion by modeling a multilevel clustered network, where clusters are formed at different hierarchies. This technique enhances the scalability by reducing the cluster size, as it requires consensus from only one cluster at each level. As the algorithm has no use of coordinators, it possesses high fault tolerance. The paper also addresses the problem of achieving optimal level of clustering in a network for distributed mutual exclusion.**

***Index Terms*—Distributed algorithm, cluster, consensus, multilevel, mutual exclusion, parallel**

## 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. These processes should be consistent and the access to these shared entities should be mutually exclusive. 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. 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 has been an active research area for the 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 the coordinator to deliver permission to the other nodes to access their CS, while in the distributed solutions the permission is obtained from consensus among all network nodes.

Distributed mutual exclusion algorithms are mainly classified into two categories: token based and permission based [1]. In permission based ME algorithms [7][8][9][10], a requesting node is required to receive permissions from other nodes (a set of nodes or all other nodes). In the token based ME algorithms [2][3][4][5][6] [12], a unique token is shared among the set of nodes. A node must own the unique token (sometimes cited as privilege message) before entering the CS. Though token based algorithms incur low message cost, they suffer from poor failure resiliency, because complex token regeneration protocols [11] must be executed if the node holding the token fails.

In permission-based algorithms, a node needs to have permissions usually only from a set of nodes, known as a quorum. Quorum formation algorithms must satisfy that any two possible quorums have a nonempty intersection. Quorum based algorithms [10] are resilient to node and communication failures and network partitioning. Communication cost of these algorithms is proportional to the quorum size. Therefore, these algorithms try to achieve two goals: small quorum size and a high degree of fault tolerance. Efficient quorum based ME algorithms take *logarithmic* cost [9] in best cases, where $n$ is the number of nodes. However, the cost increases rapidly with the increase of node failures.

At present, the number of distributed nodes has become very large. With the emergence of peer-to-peer computing [26] and grid computing [27], the distributed applications have been spread over a large number of nodes. The performance of these distributed applications depends on the number of participating nodes and the latency gaps among nodes. But, the classical ME algorithms do not consider these matters. There are some two-layer hierarchical algorithms proposed during the last decade, which consider these issues to some extent. In these algorithms [13][14] [15], the nodes in the network are usually divided into several groups where each group is often called a cluster. Since these algorithms are mainly token based (used in one or both of upper and lower layers), they suffer due to token failures. We also presented a two-layer but fully permission based algorithm in [34], where the coordinators (named as message routers) of the clusters form the upper layer. These above mentioned cluster based algorithms use two levels of network hierarchy. They run ME algorithm inside the cluster and among the clusters so that mutually exclusive access prevails.


doi:10.4304/jcp.7.8.1831-1846

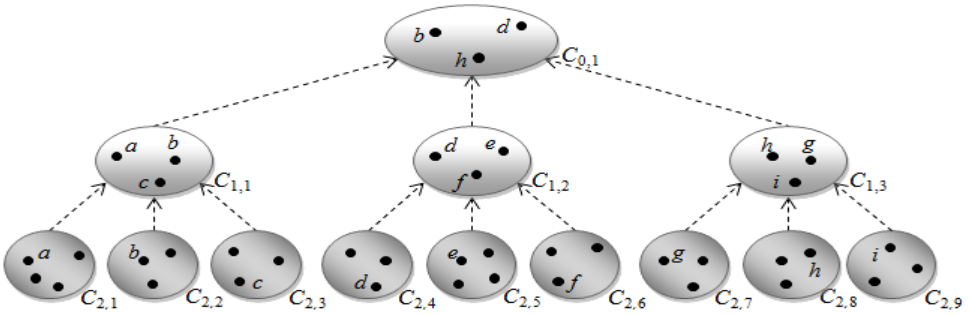


Fig. 1. A network with 2 level of clustering.

Chubby [37] and Zookeeper [38] are very well known distributed locking services, especially in case of web service data centers. These services are intended mainly for *coarse-grained* synchronization – for example, to elect a master among a set of candidates, which would then handle all access to a data for a considerable time, perhaps hours or days. Among the masters a quorum-based algorithm, typically majority quorum [17], is executed to have write access for a node, while read access is controlled by corresponding master alone. They provide a mechanism similar to hierarchical namespace.

Though the number of participating nodes is reduced to an extent in the above mentioned approaches, further reduction is achievable for extra large networks with higher levels of hierarchy. So, we extended our algorithm of [34] for multilevel clustered network in [25] where subclusters are formed inside the clusters. Chubby [37] and Zookeeper [38] are also extensible to multilevel hierarchy. However, the main problem of these algorithms is the failures of coordinators/masters. Any failed coordinator has to be replaced using a leader election algorithm, which adds significant overhead.

This paper proposes a multilevel clustered network architecture and presents a generalized parallel distributed ME solution based on this multilevel network. Our proposed solution is fully permission based. It improves the performance by reducing participating nodes in each cluster. Running existing ME algorithms in the clusters is not sufficient to achieve the mutual exclusion in a network of hierarchical clusters. Customization of existing classical algorithms is necessary to execute at different hierarchical levels. The consensuses inside the clusters must be collected and combined in a hierarchical order to achieve the global mutual exclusion. A distributed algorithm is required to execute this process for coordination among the clusters of different levels. Our proposed distributed ME solution incorporates this coordination where ME algorithm is executed parallelly in clusters of different levels. But for coordination, no fixed coordinator is used for a particular cluster. The absence of a fixed coordinator increases the fault tolerance. The parallel running of the algorithm in different hierarchical levels improves the performance. In this research, we also present necessary mechanisms to maintain the correctness of our mutual exclusion algorithm in case of node failures. The preliminary part of the research work is published in [35], which considers only single level of clustering. This is the extended version for multilevel clustered network.

The following sections of this paper are organized as follows. In Section II we describe the multilevel-clustered network architecture for ME algorithm. In Section III, the ME solution is illustrated followed by its theoretical analyses for optimal clustering. Detailed observation along performance comparison with simulation results are described in Section IV. Section V presents the critical evaluation of the proposed algorithm and the last section concludes the paper by summarizing the major contributions and future research directions.

## II. Network Model

In this section, we describe the network model that we assume in our proposed mutual exclusion solution. First we describe the system with the necessary assumptions. Then we propose a logical hierarchical network topology by introducing $\ell$ *level of clustering*. Our proposed ME solution works upon this hierarchical topology.

### A. Description of the System

We consider an asynchronous distributed system, which follows the model proposed in [29]. Each pair of nodes is connected through a communication channel. The message delays for communication and processing are finite. No assumption is made for the relative processing 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 is referred to as recovery. When a process fails, it loses the values of its states and variables used in the algorithm except for a few values, which are necessary for maintaining the correctness of the algorithm after recovery. So, it uses local stable storage to keep such crucial information.

### B. Proposed Network Architecture

The nodes in a network are logically partitioned into several nonintersecting groups. Each group is called a cluster. In multilevel clustering, some smaller clusters can form a larger cluster again and so forth.

We propose $\ell$ level of clustering where $\ell$ can be any positive integer number. A collection of nodes forms a Level $\ell$ cluster. In this way, a number of nonintersecting Level $\ell$ clusters are formed. Next Level $\ell$–1 clusters are

formed such that a group of Level $\ell$ clusters are associated with a Level $\ell$–1 cluster. The number of Level $\ell$–1 clusters is equal to the number of groups of Level $\ell$ clusters, where the groups are nonintersecting. So, the number of Level $\ell$–1 clusters is smaller than that of Level $\ell$ clusters. In this way, the clusters are formed from Level $\ell$ to Level 0. There is only one cluster at Level 0 and it is associated with all Level 1 clusters. Note that, each node is a member of a Level $\ell$ cluster; so each member of an upper layer cluster is also a member of one or more clusters of the lower levels. Also note that, no node is member of more than one cluster at any particular level.

To have a guideline of cluster formation, we follow the following rules for $\ell$ level of clustering: All nodes are divided into a number of nonintersecting Level $\ell$ clusters. Then taking an arbitrary node from each of the clusters of a collection of Level $\ell$ clusters (be identified as child clusters) a Level $\ell$–1 cluster (be identified as parent cluster) is formed. Thus, the number of nodes of a Level $\ell$–1 parent cluster is equal to the number of its Level $\ell$ child clusters. Similarly a Level $\ell$–2 cluster is formed from a collection of Level $\ell$–1 clusters. In this way, all clusters are formed at each Level. According to this guideline, a member of a Level $k$ ($0 \le k < \ell$) cluster, it is also a member of a cluster at each level from Level $\ell$ to Level $k$+1.

There is only one cluster at Level 0 and this is the topmost cluster. So it can be called as the root cluster. Level $\ell$ clusters are the bottommost clusters and can be called as leaf clusters. If $\ell$ is 0, then there is only one cluster with all nodes of the network. In this case, no clustering is done in the network.

Fig. 1 shows the topology for 2 level of clustering ($\ell$ = 2). To specify a cluster in the figure, we use $C_{i,j}$ which denotes the $j$th cluster at Level $i$. All $C_{2,j}$ ($1 \le j \le 9$) are the bottommost level (Level 2) clusters. Taking three arbitrary member a, b and c of $C_{2,1}$, $C_{2,2}$ and $C_{2,3}$ respectively a Level 1 cluster $C_{1,1}$ is formed. Similarly other two Level 1 clusters $C_{1,2}$ and $C_{1,3}$ are formed from another two different collections of Level 2 clusters. $C_{0,1}$, the only Level 0 cluster, is formed by taking an arbitrary member from each of the Level 1 clusters. It is to be noted that b, a member of $C_{0,1}$, is also members of $C_{1,1}$, a Level 1 cluster, and $C_{2,1}$, a Level 2 cluster.

Each member of a Level $k$ ($0 \le k \le \ell$) cluster knows the identifications of the members of its Level $k$ cluster and the upper level (Level $k$–1, $0 < k \le \ell$) parent cluster. As the memberships to the clusters remain unchanged throughout the program, a node keeps these data in stable storage, so that it can retrieve them after recovery if fails. For cluster formation, any suitable clustering algorithm [23][24] can be used.

## III. Proposed Solution

In this section, we propose an algorithm for distributed mutual exclusion. At first we write the brief outline of the algorithm. Next we describe the messages and states, which are required for the algorithm. Then we present our algorithm in detail, which is followed by the safety and liveness proofs. We present a detailed analysis of the proposed solution, in which we find the optimal $\ell$ level of clustering. At the end, we propose a maintenance algorithm in order to keep our solution continuing correctly in case of a node failure.

### A. Brief Outline

Our proposed network model creates a logical connection among the clusters of different levels. It represents a tree of clusters with parent-child relationship among the clusters. We propose a distributed multilevel ME solution that hierarchically executes the ME algorithm in clusters at different levels (Fig. 2). To collect consensus inside the clusters as well as to coordinate among the hierarchical executions, we extend quorum based ME algorithm [8][9].

Multiple layers of nodes participate in our proposed algorithm: the members of Level $\ell$ to Level 0 clusters. In all layers, we use the tree-quorum algorithm [9] for quorum creation. When a node $x$ wants to access the CS, first it requires the consensus from the nodes inside its Level $\ell$ cluster. Then it needs to have permission from the upper layer (Level $\ell$–1) cluster. 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 similarly to $x$. In this way consensus is sought at each layer up to Level 0. If $y$ gets consensus from its (Level $\ell$–1) cluster and its upper layer (Level $\ell$–2) cluster, it informs $x$ about the consent. At this point, the node $x$ has the total consensus, i.e., permissions from all layers, and can use the CS safely. In Section III.*C*, the proposed solution will be discussed in detail.

### B. Messages and States

The messages required for our proposed ME algorithm depend on the classic ME algorithm applied to the clusters. As we apply quorum based ME algorithm [8][9] inside the clusters, our ME algorithm uses *Request*, *Reply*, *Release*, *Inquire* and *Yield* messages. Since a node can be members of clusters at several levels, *Level number* is added to the contents of these messages to identify the level of the cluster executing ME algorithm.

The communication between a represented node, a

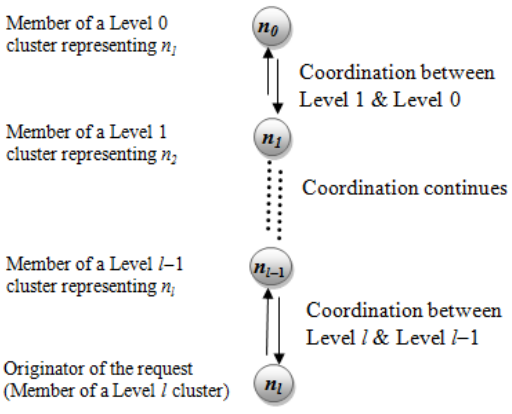


Fig. 2. Coordination between different levels in $l$ level of clustering

member of a Level $k$ cluster, and its corresponding representing node, a member of a Level $k-1$ cluster, is done through *PreCRequest*, *CRequest*, *CReply* and *CRelease* messages. Functions of these messages are briefly described below:

- When a node $x$ starts to execute ME algorithm in its Level $k$ ($0 < k \leq \ell$) cluster, it selects an arbitrary node $y$ among the members of its Level $k-1$ parent cluster as its representing node $y$ at Level $k-1$ and sends a *PreCRequest* message to $y$.
- When $x$ gets the consensus from the quorum of its Level $k$ cluster, it sends *CRequest* message to $y$ to process the request at Level $k-1$.
- $y$ sends a *CReply* message to $x$, if it has the total permission, i.e., permission inside its Level $k-1$ cluster as well as the permission from the upper level (Level $k-2$) parent cluster (if $k-2 \geq 0$).
- $x$ sends a *CRelease* message to $y$ to propagate the release message to the upper levels (Level $k-1$ to Level 0).

All the messages except *Request* have the same structure: *Message* {*source*, *level*}. First element *source* is the identification number of the node, which sends the message to a destination node, while the second element *level* is the number indicating the level of the cluster in which the sender node resides. *Request* message has another element called *timestamp*. It is the global logical time [16] when the message has been sent. It is used for ordering the requests to avoid deadlock and starvation. As multiple requests can come from different fellow nodes to a single node at a level, a node often needs to keep them in a queue to process afterward. If a node is a member of a Level $k$ ($0 \leq k \leq \ell$) cluster, it maintains a minimum priority queue, be identified as *queue*, at this level to keep the incoming *Request* messages in order of their timestamps. If timestamps of two *Request* messages are equal, then node identification numbers are used for determination of order. This ordering is crucial for avoiding deadlock and starvation. A member node of Level $k$ ($0 \leq k < \ell$) cluster also maintains a first-in first-out queue, be identified as *fifo*, for *CRequest* messages.

Each node might have different roles in different levels. As a result, each node might have different states for different levels. To represent the proposed system we define following array of states. Each element of the array represents the state of the node at a particular level. Throughout the paper we use the word 'set' for a state to denote that the state is true. On the other hand, the word 'reset' is used to denote false.

- *REQUESTING* [0…$\ell$]: This state at a level is set when a node sends *Request* messages to the fellow nodes of its cluster at Level $k$.
- *LOCKED* [0…$\ell$]: This state at Level $k$ is set when a node sends *Reply* to a *REQUESTING* fellow node, from which it has received a *Request*.
- *BUSY* [0…$\ell$]: When a requesting node at Level $k$ receives consensus inside its Level $k$ ($0 \leq k \leq \ell$) cluster and from its Level $k-1$ (if $k > 0$) parent cluster, it enters into *BUSY* state at this level. If $k < \ell$, it is obvious that it has already received *CRequest* from a node of its Level $k+1$ child clusters.
- *BUSYWAITING* [0…$\ell$]: When a node at Level $k$ gets a *PreCRequest* from a node of its Level $k+1$ child clusters, it starts processing of collecting consensus inside its Level $k$ cluster as well as from its Level $k-1$ (if $k>0$) parent cluster. If, afterward, it receives consensus from both of its cluster and the parent cluster, before receiving *CRequest* from any node of the child clusters, this state at Level $k$ is set.
- *CREQUESTING* [0…$\ell$]: When a requesting node at Level $k$ (if $k>0$) gets necessary consensus in its cluster of this level, it sends *CRequest* to its representing node at Level $k-1$ for the consensus of that level, and so sets its *CREQUESTING* state at this level.
- *INQUIRING* [0…$\ell$]: If a node $x$ at Level $k$ gets a request from a fellow node $y$ that has a timestamp earlier than that of a previously received request from a node $z$, which has been replied already, then $x$ sends an *Inquire* message to $z$, as well as, sets the *INQUIRING* state at this level.

### C. Algorithm

Following variables are used in our ME algorithm:

- *queue* [0…$\ell$]: Each entry is a min-priority queue to keep the *Request* messages at Level $k$ ($0 \leq k \leq \ell-1$). Priority is measured using timestamps of the messages.
- *fifo* [0…($\ell-1$)]: A first-in first-out queue for each level to keep the *CRequest* messages at Level $k$ ($0 \leq k \leq \ell-1$).
- *replies* [0…$\ell$]: Storage to keep the received *Reply* messages at Level $k$ ($0 \leq k \leq \ell$).
- *representing-node* [1…$\ell$]: if *representing-node*[$k$]>0, it represents the representing node at Level $k-1$.
- *represented-node* [0…($\ell-1$)]: if *represented-node*[$k$]>0, it represents the represented node at Level $k+1$.
- *num-of-PreCRequest-pending* [0…($\ell-1$)]: It is the number of pending *PreCRequest* messages at Level $k$ ($0 \leq k \leq \ell-1$).
- *locking-node* [0…$\ell$]: It represents the locking node at Level $k$ ($0 \leq k \leq \ell$).

We use '**Send** *Message* **to** *destination*' command in order to send a message (*Message*) to a node (*destination*). The proposed algorithm consists of different key procedures, pseudocodes of which are shown below. Each node in the system executes the algorithm according to its role. Except for the first four procedures, each function is invoked by a node when an associated message is received.

---

**Procedure Do-Request** ($k$)
1 *REQUESTING*[$k$] ← TRUE
2 **if** $k > 0$
3 *representing-node*[$k$]←Select a representing node at $k$–1
4 **Send** *PreCRequest*{*id*, $k$} **to** *representing-node*[$k$]
5 Select a quorum within its cluster

```
    6  Send Request{id, k, current time} to all members of
   the quorum
7.  Initialize replies[k]

Procedure Do-CS-Request
1   Do-Request(ℓ)

Procedure Process-Use-CS
1   BUSY[ℓ] ← TRUE
2   /* Use of the CS */
3   BUSY[ℓ] ← FALSE
4   if ℓ > 0
5      Send CRelease{id, ℓ} to representing-node[ℓ]
6      representing-node[ℓ] ← 0
7   Send Release{id, ℓ} message to all members of quorum

Procedure Time-Out-BUSYWAITING (k)
1   BUSYWAITING[k] ← FALSE
2   num-of-PreCRequest-pending[k] ← 0
3   if k > 0
4      Send CRelease{id, k} to representing-node[k]
5   Send Release{id, ℓ} message to all members of quorum

Procedure Process-Request (request)
1   k ← request.level
2   if LOCKED[k] = FALSE and INQUIRING[k] = FALSE
3      LOCKED[k] ← TRUE
4      locking-node[k] ← request.source
5      Send Reply{id, k} to locking-node[k]
6   else
7      Insert request into queue[k]
       8  if LOCKED[k] = TRUE and request has higher
   priority than the request of locking-node[k]
9          LOCKED[k] ← FALSE
10         INQUIRING[k] ← TRUE
11         Send Inquire{id, k} to locking-node[k]

Procedure Process-Reply (reply)
1   k ← reply.level
2   Insert reply into replies[k]
    3  if replies[k] possesses Reply messages from all
   members of a requesting quorum
4      REQUESTING[k] ← FALSE
5      if k = 0
6         if k = ℓ
7            Process-Use-CS()
8         else
9            if represented-node[k] ≠ 0
10              BUSY[k] ← TRUE
11              Send CReply{id, k} to represented-node[k]

12           else
13              BUSYWAITING[k] ← TRUE
                14Timer for Time-Out-BUSYWAITING(k)
              starts
15     else
16        CREQUESTING[k] ← TRUE
17        Send CRequest{id, k} to representing-node[k]

Procedure Process-Release (release)
1   k ← release.level
2   LOCKED[k] ← FALSE
3   INQUIRING[k] ← FALSE
4   if queue[k] is not empty
5      request ← Extract from queue[k]
6      LOCKED[k] ← TRUE
7      locking-node[k] ← request.source
8      Send Reply{id, k} to locking-node[k]
Procedure Process-CRequest (cRequest)
1   k ← cRequest.level – 1
2   if represented-node[k] ≠ 0
3      Insert cRequest into fifo[k]
4   else
5      represented-node[k] ← cRequest.source
6      if BUSYWAITING[k] ← TRUE
7         BUSY[k] ← TRUE
8         BUSYWAITING[k] ← FALSE
9         Send CReply{id, k} to represented-node[k]
       10   else  if REQUESTING[k] = FALSE and
     CREQUESTING[k] = FALSE
11        Do-Request (k)

Procedure Process-PreCRequest (preCRequest)
1   k ← preCRequest.level – 1
    2  num-of-PreCRequest-pending[k] ← num-of-
   PreCRequest-pending[k] + 1
    3  if represented-node[k] = 0 and REQUESTING[k] =
   FALSE and CREQUESTING[k] = FALSE
       4     num-of-PreCRequest-pending[k] ← num-of-
     PreCRequest-pending[k] – 1
5      Do-Request (k)

Procedure Process-CReply (cReply)
1   k ← cReply.level + 1
2   CREQUESTING[k] ← FALSE
3   if k = ℓ
4      Process-Use-CS ()
5   else
6      if represented-node[k] ≠ 0
7         BUSY[k] ← TRUE
8         Send CReply{id, k} to represented-node[k]
9      else
10     BUSYWAITING[k] ← TRUE
11        Timer for Time-Out-BUSYWAITING(k) starts

Procedure Process-CRelease (cRelease)
1   k ← cRelease.level – 1
2   BUSY[k] ← FALSE
3   represented-node[k] ← 0
4   if fifo[k] is not empty
5      cRequest ← Extract from fifo[k]
6      represented-node[k] ← cRequest.source
7      BUSY[k] ← TRUE
8      Send CReply{id, k} to represented-node[k]
9      if num-of-PreCRequest-pending[k]>0
          10 num-of-PreCRequest-pending[k] ← num-of-
       PreCRequest-pending[k] – 1
11  else
12     if k > 0
13        Send CRelease{id, k} to representing-node[k]
14        representing-node[k] ← 0
15     Send Release{id, k} message to all members of
quorum
16  if num-of-PreCRequest-pending[k]>0
       17    num-of-PreCRequest-pending[k] ← num-of-
     PreCRequest-pending[k] – 1
18     Do-Request (k)
```

The points below describe our algorithm. We also utilize Fig. 1 along to explain the algorithm.

**Basic Multilevel ME Algorithm:**

◊Requests for the CS are generated at Level $\ell$ cluster (*Do-CS-Request*). These requests are processed in different levels- Level $\ell$ to Level 0- sequentially. For example, the clusters $C_{2,4}$, $C_{1,2}$ and $C_{0,1}$ in Fig. 1 represent the participating

clusters if a member of $C_{2,4}$ places a request for the CS.

◊A CS requesting node $x$ first starts ME algorithm (*Do-Request*) in its Level $\ell$ cluster. It selects a quorum $q_l$ and sends a *Request* message to each of the quorum members. Now, its *REQUESTING*[$l$] state is set (State transition 1). If its request is granted within the cluster, it selects an arbitrary node $y$ among the members of its Level $\ell-1$ parent cluster, as its *representing-node* at Level $\ell-1$, and sends *CRequest* to it (*Process-Reply* at lines 3-11). At this point, $x$'s *REQUESTING*[$l$] state is reset and *CREQUESTING*[$l$] state is set. To $y$, $x$ is identified as the *represented-node* at Level $\ell$. When $y$ gets *CRequest* from $x$, it executes ME algorithm in its Level $\ell-1$ cluster (*Process-CRequest* at lines 10-11). If $y$ gets consensus here, then it sends *CRequest* to its Level $\ell-2$ parent cluster. This process continues until Level 0 is reached.

◊When a node of a Level $k$ ($0 \le k \le \ell$) cluster makes a request, it needs the permission or consensus from two sides in two sequential steps: firstly from the nodes of its cluster and secondly from the upper level (Level $k-1$) parent cluster through the representing node. If there is no upper level ($k = 0$), then only the consensus from its cluster is required. After getting the total consensus, a node of a Level $k$ cluster sends *CReply* downward to its represented node, a member of a Level $k+1$ cluster (*Process-Reply* at line 5, and *Process-CReply* at lines 6-8). At this point, it resets its *CREQUESTING*[$k$] state (if $k>0$) and sets its *BUSY*[$k$] state. In this way, when the requesting node at the bottommost level ($k = \ell$), the originator of the request, gets the total consensus (*Process-Reply* at lines 3-7, and *Process-CReply* at lines 3-4), it executes the CS exclusively (*Process-Use-CS*).

◊After execution of the CS, the requesting node resets *BUSY*[$l$] state and sends *Release* messages to the nodes inside its cluster, a Level $\ell$ cluster (*Process-Use-CS* at line 7). If $\ell>0$, it also sends *CRelease* message to its representing node at Level $\ell-1$ (*Process-Use-CS* at lines 4-6). When the representing node at Level $k$ ($0 \le k < \ell$) cluster gets *CRelease* from its represented node at Level $k+1$, it resets its *BUSY*[$k$] state and sends *Release* messages to the member nodes of its Level $k$ cluster (*Process-CRelease* at line 15). If $k>0$, it also sends *CRelease* message to its representing node at Level $k-1$ (*Process-CRelease* at lines 12-14). In this way, consensus is released up to Level 0.

◊A node at Level $k$, if it is not in *LOCKED*[$k$] or *INQUIRING*[$k$] state, sends a *Reply* message, as its permission, to the requesting node and sets its *LOCKED*[$k$] state (*Process-Request* at lines 1-5). Otherwise it inserts the *Request* message into *queue*[$k$]. When it receives the release message, a node resets its *LOCKED*[$k$] state and chooses the request residing at top of its *queue*[$k$], if it is not empty, for next processing (*Process-Release*).

◊It is possible for a node of a Level $k$ ($0 \le k < \ell$) cluster to be selected as representing node by multiple nodes of different Level $k+1$ child clusters. For example, $h$, a node of $C_{0,1}$, is selected as representing node at Level 0 simultaneously by $e$ and $g$, two nodes of two different Level 1 child clusters $C_{1,1}$ and $C_{1,2}$ respectively. So, $h$ will receive *CRequest* messages from both of $e$ and $g$. If $h$ receives these messages at the same time, then it serves the *CRequest* message that comes first. So, it keeps pending *CRequest* messages into *fifo* (*Process-CRequest* at lines 2-3). When a node get consensus for the currently serving *CRequest*, it also serves all other *CRequest* messages with this consensus one after one until its queue becomes empty (*Process-CRelease* at lines 4-10). Let, $h$ has received *CRequest* of $e$ before that of $g$. So, $h$ is serving the request of $e$, which is now the represented node of $h$, while the request of $g$ is waiting in the queue. When $h$ receives consensus from $C_{0,1}$, it sends *CReply* to $e$. After a while, when $h$ gets *CRelease* from $e$, it does not release the consensus; rather it sends *CReply* to $g$, which is now the represented node of $h$. This process continues until $h$ gets *CRelease* from its represented node and finds out that its *CRequest* queue is empty. Then it releases the consensus in its cluster and sends *CRelease* upward to its representing node if any upper level exists (*Process-CRelease* at lines 12-15).

According to the above description of the algorithm, the progression of the algorithm is sequential since consensus must be ensured in a Level $k+1$ cluster before processing starts in a Level $k$ cluster. Now we discuss how this sequential behavior is significantly reduced by incorporating parallelism.

**Parallel Execution of ME Algorithm:**

◊When a node $x$ begins to execute ME algorithm in its associated Level $k$ ($0 < k \le \ell$) cluster, it also sends *PreCRequest* to its arbitrarily selected representing node $y$ at Level $k-1$ (*Do-Request* at lines 2-4). After getting *PreCRequest* from $x$, $y$ starts to execute ME algorithm in its Level $k-1$ cluster without setting its represented node (*Process-PreCRequest* at lines 3-5). If $x$ gets consensus within its Level $k$ cluster, it places *CRequest* to $y$. When $y$ gets the *CRequest*, it sets its represented node to $x$. Within the time $y$ may have already received consensus in its Level $k-1$ cluster. Otherwise, $y$ has reached a point towards getting consensus.

- If $y$ gets consensus before receiving *CRequest* from a node of its Level $k$ child clusters, it sets its *BUSYWAITING* state and waits for a threshold period for the request (*Process-Reply* at lines 12-14, and *Process-CReply* at lines 9-11). Within this threshold

time, if any *CRequest* come, *y* accepts the requesting node as its represented node and gets into *BUSY* state. It time expires, *y* will release this consensus so that other competing nodes need not to wait any longer and *BUSYWAITING* is reset (*Time-Out-BUSYWAITING*).

◊It is common and usual to have other nodes besides *x* in the corresponding Level *k* child clusters to be competing in order to get consensus. Obviously, each of those nodes will send *PreCRequest* to their representing nodes. It is quite possible for *y* to be selected as the representing node at Level *k*−1 by more than one nodes of Level *k*. Since *y* can process only one in-advance request, at a time, if *y* is already processing of such a request, whether it is *CRequest* or *PreCRequest*, it will not process any additional *PreCRequest* until the current processing finishes. It only counts the pending in-advance requests for later processing (*Process-PreCRequest* at line 2).

- Let *x* and *z* (in the same or different Level *k* clusters) both are requesting for consensus, which are child clusters of a Level *k*−1 cluster. Both of the nodes have selected *y*, a member of that Level *k*−1 cluster, as their representing nodes. *y* is running ME algorithm in the cluster in response to *x*'s *PreCRequest*, since the request of *x* has reached *y* earlier than that of *z*. However, *z* gets consensus (in its cluster) before *x*. So, *z* sends *CRequest* to *y*. Now *y* sets *z* as its represented node at Level *k* and ongoing ME algorithm continues (*Process-CRequest*).
- After receiving *CRelease* from *z*, *y* begins advance processing again, since its counter shows that there is still one *PreCRequest* to serve (*Process-CRelease* at lines 16-18). At this time, *x* may get consensus in its Level *k* cluster and then it will send *CRequest* to *y*.

*Inquire* and *Yield* messages works to avoid deadlock, similarly as in [7][8][35]. The pseudocodes of the processes, which are invoked when a node receives these messages, are not mentioned here due to space limitation. Deadlock avoidance procedure is described below.

**Deadlock Avoidance Algorithm:**

◊Let a node *u*, at some Level *k*, receives a request from a node *v* that possesses a timestamp earlier than the request of a node *w* currently being processed; to which it has already sent a *Reply* message. Then *u* puts the request of *v* into *queue*[*k*] and sends an *Inquire* message to *w* and waits for either a *Yield* or *Release* message from *w* (*Process-Request* at lines 8-11). At this time, its *LOCKED*[*k*] state is reset and *INQUIRING*[*k*] state is set.

◊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 requested 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*[*k*] state and puts back the request of *w* into its *queue*[*k*]. Now it pop out the request from the top of *queue*[*k*] and accordingly sends a *Reply* message to the corresponding node and gets into *LOCKED*[*k*] state again. In the mean time if no *Request* with earlier timestamp has come, the *Request* of *v* is the selected request message. If *u* receives *Release* message instead of *Yield* message from *w*, it does the same except for reinserting the request of *w* into *queue*[*k*] (*Process-Release* at line 3), since the request has been served already.

Necessary proofs for the new ME solution are given in following Subsection. The solution is also theoretically analyzed in Subsection II.*F* for computing optimal level of clustering. Since failures of nodes disrupt any ongoing process of our ME algorithm, an algorithm is presented in Subsection II.*G* for maintaining the proper execution of the ME algorithm.

### *D. Proof*

Correctness of our ME algorithm is guaranteed by proving two properties: *safety* and *liveness*.

1) *Safety*

A mutual exclusion solution is said to be *safe* if no more than one node gets access to the CS at a time. For quorum based algorithms, this condition holds, if there is at least one common node between any two quorums for accessing the CS [18]. For $\ell$ level of clustering, we must get consensus at each level. A quorum of our solution can be defined as

$$q = q_l \cup \cdots \cup q_k \cup \cdots \cup q_0 = \bigcup_{1 \le k \le l} q_k ,$$

Here $q_k$ is a quorum formed from the nodes of a Level *k* cluster.

Similarly, another quorum could be $p = p_l \cup \cdots \cup p_k \cup \cdots \cup p_0 = \bigcup_{1 \le k \le l} p_k$, where $p_k$ is a quorum formed from the nodes of a Level *k* cluster. Now,

$$p \cap q = (p_l \cap q_l) \cup \ldots\ldots\ldots \cup (p_1 \cap q_1) \cup (p_0 \cap q_0)$$

Here, $p_0 \cap q_0 \neq \phi$, as these are the quorums of the only cluster in Level 0 and inside the cluster classical quorum formation algorithms are used, which ensure the intersection between any two quorums [9]. So, $p \cap q \neq \phi$. Thus, the solution must maintain mutual exclusion in entering the CS.

2) *Liveness*

To prove the liveness, we need to show that any request is served after a finite period of time. Let the *C* nodes of a Level *k* cluster be $N_1$, $N_2$, … …, $N_C$. Consider the worst-case scenario, where the requests are queued at $N_i$ from each of the fellow nodes. The timestamps of the

queued requests at $N_i$ are $T_1$, $T_2$, … …, $T_C$. 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 any node $N_j$ (after completion of its earlier request with timestamp $T_j$) is denoted by $T_j^{next}$. Definitely, $T_j^{next} \geq T_{current}$. Hence, $T_j^{next} > T_m$. Therefore, the request from $N_m$ must be served within a finite duration (after $C$–1 outstanding requests are served) as it has the timestamp earlier than that of the next group of requests.

For $\ell$ level of clustering, we must have consensus at all levels. After getting consensus from a Level $k$ ($\ell \leq k < 0$) cluster, we send request (*CRequest*) to the representing node at Level $k$–1 as we need further consensus from Level $k$–1 to Level 0. At this moment, the representing node's execution of ME algorithm at its Level $k$–1 cluster may be already in a position close to getting consensus due to *PreCRequest* message. After receiving *CRequest*, the representing node continues the ME execution, if the process is ongoing, otherwise it starts the processing. As, *CRequest* message will reach all levels up to Level 0, ME algorithm is executed in a cluster at each level. Since any request must be served within a finite period in a cluster, ultimately we must get consensus at all levels ($\ell \leq k \leq 0$) after a finite time. So, liveness is proved. Note that, in the worst case a request needs to wait for the completion of $(\ell+1) \times (C-1)$ other requests.

### *E. Analysis*

The analysis of the proposed solution is presented in this section considering tree-quorum algorithm [9] running in the clusters of each level. According to 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 $f$ is the availability of a node in a quorum (probability of a node being available or alive) and $h = \log_2 n$ (height of the binary tree formed by $n$ nodes). Note that, according to [9], $f$ denotes the fraction of the quorums that include the root of the tree of level $\ell$+1. It is actually the probability that the root is included in the quorum when all the quorums are equally probable. In this research, we assume that the root is included in the quorum if it is available. Thus, $f$ is equivalent to the probability that the root is available. In the recursive equation of expected quorum size, each node becomes a root in a particular level of the cluster. That is why $f$ is denoted as the probability that a node is available at a particular instant. The message cost is the average number of messages needed for (a single request from) a node to enter the CS and this is proportional to the quorum size, $c_h$. Hence, this $c_h$ represents the cost function. Solving this recurrence, following equation is found:

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

As the time requirement in a distributed algorithm is almost proportional to the number of messages, the following analysis will also be applicable for average time requirement.

1) *Optimal Cluster Size*

Let level of clustering be 1. So, there are two levels of clusters: a number of Level 1 clusters and a Level 0 cluster. We consider two parameters: $n$, the number of nodes in the network, and $C$, cluster size of each Level 1 cluster assuming all clusters are of the same size. Therefore, the number of Level 1 clusters is $n/C$. This is the cluster size of Level 0 cluster since taking one arbitrary node from each of the Level 1 clusters forms the Level 0 cluster.

The height of the tree formed by the nodes of a Level 1 cluster is $h_1 = \log_2 C$. Similarly, the height of the tree formed in Level 0 cluster is

$$h_0 = \log_2 (n/C)$$
$$\Rightarrow h_0 = \log_2 n - \log_2 C$$
$$\Rightarrow h_0 = h - h_1$$

Therefore, total cost of the proposed solution:

$c = c_0 + c_1 +$ Coordination cost between Level 0 and Level 1

Here, $c_0$ and $c_1$ are the costs of executing tree-quorum algorithm at Level 0 and Level 1 respectively. Coordination cost is proportional to the number of representing nodes doing coordination. Here, only one representing node does the coordination between Level 0 and Level 1.

Hence, $c = \frac{(2-f)^{h-h_1} - f}{1-f} + \frac{(2-f)^{h_1} - f}{1-f} + 1$

Taking the derivative of $c$ with respect to $h_1$ and equating it to zero, we get the following condition of optimal clustering:

$$2h_1 = h$$
$$\Rightarrow 2\log_2 C = \log_2 n$$
$$\therefore C = \sqrt{n}$$

That is, the optimal cluster size for a network with single level of clustering is the square root of the network size. The number of Level 1 clusters (i.e., the Level 0 cluster size) is $n/C = n/\sqrt{n} = \sqrt{n}$. Therefore, the cluster sizes of all of the clusters of both the levels are the same. This is proved for 1 level of clustering where tree-quorum algorithm is used within the clusters. We can easily derive the following results by extending the result of 1 Level of clustering:

"*For optimality, the cluster sizes of different levels of clusters are equal for $\ell$ level ($\ell \geq 1$) of clustering when tree-quorum algorithm is used inside the clusters*".

It is worth mentioning that, we have not considered the heterogeneity of the network, i.e., the heterogeneity of the communication and the heterogeneity of the clusters. Rather, for convenience of analysis we determine optimality based on the message costs.

2) *Optimal Level of Clustering*

In order to obtain the optimal message cost for $\ell$ level of clustering of $n$ nodes we find $n = C^{l+1}$. This can be explained as: at first $n$ nodes are partitioned into several clusters where each of them has size $C$. So, total number of clusters is $n/C$. Now new clusters of another level are

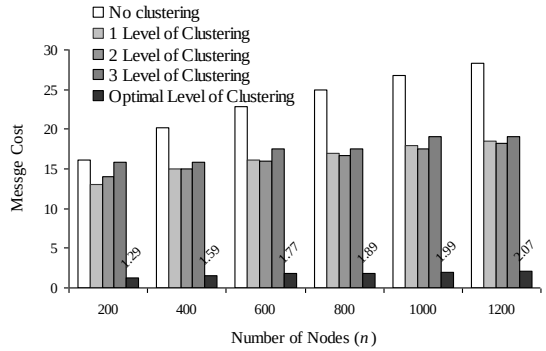


Fig. 3. Comparison among different levels of clustering hierarchy for the networks with different sizes and fixed $f = 0.8$.

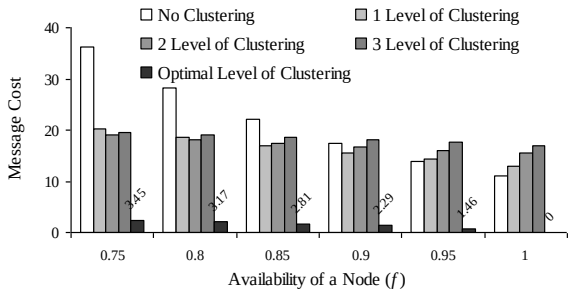


Fig. 4. Comparison among different levels of clustering for different $f$ in a network with 1200 nodes.

generated by taking $C$ number of clusters of the previous level. Thus, this level contains $n/C^2$ clusters. This clustering process will continue up to $\ell$+1 times. At last, topmost level will contain only a single cluster. Now we will determine the optimal value of $\ell$ if the tree-quorum algorithm is applied in each level. Remember that, inside any cluster of any level, only $C$ number of nodes participate in the ME algorithm.

Now, let $h$ is the height of the tree formed by the whole network of $n$ nodes. Then $h_1$, the height of the tree formed in each cluster by $C$ participating nodes, is $h_1 = \frac{h}{(l+1)}$. Thus, the total cost $c$ can be expressed as follows:

$$c = \sum_{1}^{l+1} \frac{(2-f)^{\frac{h}{l+1}} - f}{1-f} + \text{Level to Level Coordinati on Cost}$$

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

Taking $dc/dl = 0$, the optimal value for $\ell$ can be expressed as:

$$l = \frac{h \log_e (2-f)}{1+k} - 1 \quad \text{..... (2)}$$

Here $k$ is a dummy parameter taken to solve the equation. An equation-solving tool named 'DeadLine' [28] is used to find the value of $k$ from an intermediate equation. A rough estimate of the value of $k$ can be represented by $(0.27846 \times (1-f) - f)$ for $f \neq 1$. The optimal value of $\ell$ is 0 for $f$ =1, i.e., theoretically no clustering is needed where all the nodes are available in the system.

Fig. 3 and Fig. 4 show theoretical message cost for different levels of clustering. Optimal level of clustering for a particular $n$ and $f$ is determined by Equation (2). Optimal level of clustering is shown in both of the figures at the rightmost of the message cost bars. Note that, the message costs shown in these graphs are actually (accumulation of) expected quorum sizes at different levels. Following are the observations considering the message cost from the presented figures:

- Clustering is very much effective for large $n$ and low $f$. If $f$=1, no clustering is required irrespective of the network size.
- Clustering reduces the message cost up to a certain level of clustering, which is defined as the optimal level of clustering. Further clustering increases the message cost.
- For a particular $f$ if $n$ increases, higher level of clustering performs better than lower level of clustering. Similarly for a particular $n$ if $f$ decreases, higher level of clustering performs better than lower level of clustering.

These graphs are plotted using the equations derived from the theoretical analysis. In Section IV, we will verify this theoretical result with the simulation result.

3) *Minimum Delay for a Request to Satisfy*

Let $q$ is the average quorum size of a cluster of any level, T is the average transmission time and P is the average processing time. We make two assumptions: (i) at least one quorum is available in a cluster; (ii) there is only a single request for CS in the system to process and no other request will be issued until its processing completes.

The processing time for $q$ number of *Request* messages is $q$P, that is, the $q$'th message is sent after $q$P delay. Thus, the time for the last message to reach the destination node is $q$P + T. In the mean time, earlier messages must have reached their destinations. Now, in Level $\ell-1$ cluster the request is reached through a *PreCRequest*. Time for the *PreCRequest* to reach the representing node at the next upper level is P + T. As a result, all nodes of a quorum at Level $\ell-1$ will receive a *Request* message within (P + T) + $q$P + T. In this way, the time needed for *PreCRequest* to reach the representing node of Level 0 is $\ell$ (P + T) and consequently the last *Request* message in this level reaches its destination at $\ell$ (P + T) + $q$P + T.

When a node receives a *Request*, it sends *Reply* immediately if it is free (i.e., it is not *LOCKED* and no other requests is in its queue). Since we have assumed that, only a single CS request exists in the system, after receiving a *Request*, a *Reply* will be sent just incurring the processing time. Thus at Level $\ell$, total time until the last *Reply* reached to the requesting node is $q$P + T + (P + T). Similarly at Level 0, the requesting node receives the last *Reply* on $\ell$(P + T) + $q$P + T + (P + T), i.e., ($\ell$ + 1)(P + T) + $q$P + T.

After getting the consensus, *CReply* is sent to downward. It takes (P + T) time to reach from Level $k$ to Level $k$+1. In this way, *CReply* will be reached from

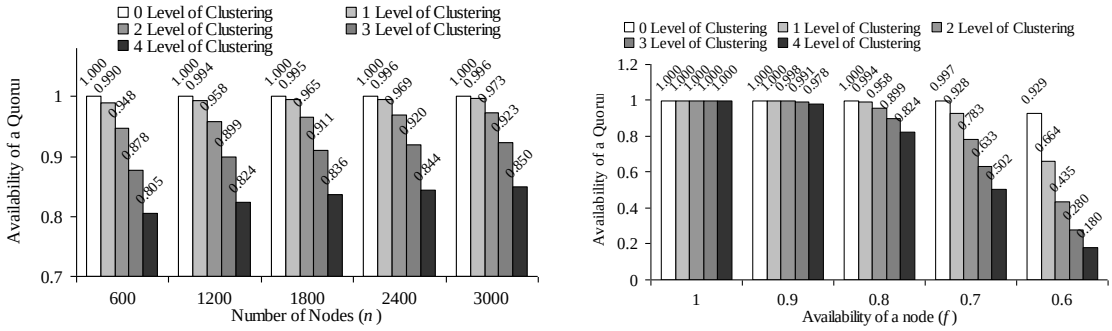


Fig. 5. Availability of a quorum for different levels of clustering by (a) varying $n$ with $f$=0.8 and (b) varying $f$ with $n$=1200.

Level 0 to the CS requesting node in $\ell$(P + T) time. So, the total time for the CS requesting node needs from requesting (i.e., start processing the first *Request* to be sent) to getting consensus (i.e., receiving *CReply*) is ($\ell$ + 1) (P + T) + $q$P + T + $\ell$(P + T), that is, (2$l$ + 1) (P + T) + $q$P + T.

We know that $q$ is a function of the number of participating nodes, which is the cluster size $C$, and it varies from $\log_2 C$ to $C$/2 depending on the $f$, the availability of a node. It is worth mentioning that, the delay for request satisfaction in case of no clustering is $q'$P + T + (P + T), where $q'$ varies from $\log_2 n$ to $n$/2. Here, $n$ is the number of nodes in the system.

4) *Availability of a Quorum*

The probability that a node is available at any time is $f$. According to the tree quorum algorithm [9], 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, $A_i$ as the availability of a subtree of height $i$, and 1 – $A_i$ as the unavailability of a subtree of height $i$ we can write the above expression as follows:

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

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 $C$. For optimal solution, all clusters have the same size $C$. Letting the height of the binary tree formed by $C$ nodes as $h_1$, the availability of a quorum in a cluster is $A_{h_1} = 2fA_{h_1-1} + (1 - 2f) A^2_{h_1-1}$.

A quorum $q$ of our solution is defined as $q_l \cup \cdots \cup q_k \cup \cdots \cup q_0$ or $\bigcup_{1 \le k \le l} q_k$, where $q_k$ is a quorum formed from the nodes of a Level $k$ cluster. So the availability of a quorum in our solution is an aggregation of the availabilities of $\ell$ + 1 quorums. As each cluster has size $C$, i.e., height $h_1$, the availability is as follows:

$A_h = A_{h_1} \times A_{h_1} \times \ldots\ldots$ up to ($\ell$ + 1) times = $(A_{h_1})^{\ell+1}$

Fig. 5(a) and 5(b) show the availability of a quorum for different level of clustering varying number of nodes ($n$) and availability of a node ($f$) respectively. For higher level of clustering and lower values of $f$, availability of a quorum reduces significantly. It is to be noted that availability should be considered with message cost in order to choose optimal level of clustering, so that no node starves when it requires using the CS. However, to calculate optimal level of clustering, we do not consider availability of a quorum assuming that at any instance at least a quorum is available.

### *F. An Extension to our Proposed Algorithm*

A node of a Level $k$ ($0 \le k < \ell$) cluster gets *PreCRequest* messages from different requesting nodes of its Level $k$+1 child clusters. At a time, it does in-advance processing for a single *PreCRequest* message; so, it maintains a counter to keep the number of pending *PreCRequest* messages, for which it will do processing after the completion of the current one. So, when it gets a *CRelease* message from a member node of its Level $k$+1 cluster and its *fifo* is empty, it may get into *BUSYWAITING* state for a threshold time, be identified as THRESHOLD2, with the hope that, within THRESHOLD2 a *CRequest* will come from a requesting node of its Level $k$+1 child clusters.

Usefulness of this technique depends on the counter of pending *PreCRequest* messages and the value of $k$. If a requesting node at Level $k$ gets consensus without executing ME algorithm at upper levels starting from Level $k$–1 to Level 0, message cost is reduced and time is shortened. So, for larger values of the counter and $k$, this technique is more beneficial. We apply this extended technique by following the guidelines below:

1. When the counter value of pending *PreCRequest* messages is zero, then this technique

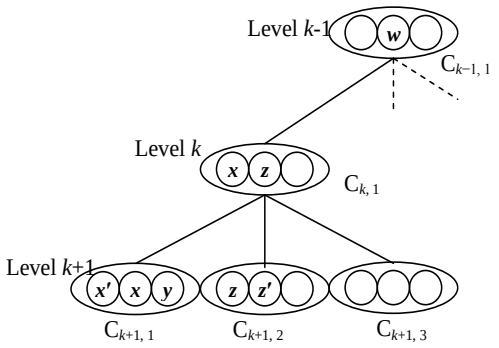


Fig. 6. An *l* level of clustered network showing only three levels (Level *k*+1 to *k*-1).

is not applied.

2. If the counter value is not zero, then according to a probability, be identified as PROBABILITY, on the counter value, this technique is applied. We can use higher PROBABILITY in case of Level *k*+1 ($0 \le k < \ell-1$) clusters than in case of Level *k* clusters, because cost saving is higher in case of the levels having higher depth from the top. PROBABILITY can have value 1 to 0. If it is 0, then extended technique will not be applied at all. In case of high PROBABILITY, the fairness among the competing requesting nodes with respect to time-requirement of getting consensus (delay for a request to satisfy) becomes low. If PROBABILITY is reduced in case of consecutive application of this extended technique by a representing node, the fairness will increase.

### *G. Maintenance Algorithm*

Maintenance algorithm maintains the correctness of our mutual exclusion algorithm in case of failures of the nodes that are executing or participating in the ME algorithm. It is assumed that when a node *x* is interrelated with another node *y* because of the execution of ME algorithm, then if *y* fails, *x* can detect the failure [30][31][32]. When *x* detects such a failure it executes the maintenance algorithm. This algorithm has two parts. One part works in case of node failure while another part handles the case of node recovery. The algorithm along with its correctness proof is illustrated below with the help of Fig. 6.

1) *In case of Node Failure*

Let *x* be the failed node. It is a member of a Level *k* cluster $C_{k,1}$. As a representing node, it might be processing the *CRequest* of *y*, a member of a Level *k*+1 cluster $C_{k+1,1}$, in $C_{k,1}$. Some nodes of Cluster $C_{k,1}$ can be in the locked state due to this request processing. Some nodes of $C_{k,1}$ can have *Request* message of *x* waiting in their QUEUEs at Level *k*. Nodes of Cluster $C_{k-1,1}$ can also be in similar situations due to the processing of the *CRequest* issued by *x* to *w*, representing node of *x* at Level *k*–1. More nodes of different clusters in the hierarchy might be engaged with this request. How the maintenance algorithm cancels the engagement of the nodes with *x* is described as follows:

- A node *z* in Level *k* cluster $C_{k,1}$ is in *LOCKED* state by the failed node *x*: When the locked node *z* detects that its locking node *x* has failed, it withdraws its consensus after waiting for a long enough time to ensure that there is no node downward in the hierarchy in *BUSY* state. Node *z* is now free to give consent to any pending request.
- In Level *k* cluster $C_{k,1}$, *z* has *Request* from *x* waiting in its QUEUE: After detecting *x*'s failure, *z* discards the request from the QUEUE.
- Representing node *w* at Level *k*–1 is processing the *CRequest* of its represented node *x*: If *w* is in *BUSY* state, it waits for a long enough time to ensure that there is no node downward in the hierarchy in busy state. Otherwise, no waiting is necessary. Now it sends *Release* messages to all the nodes of Cluster $C_{k-1,1}$ to which it has sent requests. *w* also sends a *CRelease* message to its representing node at Level *k*−2 (if $k-2 \ge 0$), if it has already sent a *CRequest* message to it. Following two points are raised in this case:
  - A node *u* at Level *k* is not in *LOCKED* state for a node *v* but receives *Release* message from *v*: This occurs when *v*'s *Request* is pending in *u*'s QUEUE at Level *k*. *u* just discards the request from its QUEUE.
  - A node *u* at Level *k* is not in *BUSY* state but receives *CRelease* message from a node *v*, member of a Level *k*+1 child cluster: If *v* is the represented node of *u*, actions differ depending on *u*'s FIFO at Level *k*. If the FIFO is not empty, *u* takes the next *CRequest* in the FIFO to process. As processing is ongoing, it just replaces *v* as its represented node with the sender of *CRequest*. Otherwise, *u* sends *Release* messages to all the nodes of Cluster $C_{k-1,1}$ to which it has sent requests. It also sends *CRelease* to it representing node at Level *k*−1, if it has already sent a *CRequest* to the representing node. If *v* is not the represented node of *u*, *v*'s *CRequest* is awaiting in its FIFO. This time, *u* discards the *CRequest* from its FIFO.
- Represented node *y* sent *CRequest* to its representing node *x* at Level *k*: If *y* is in *CREQUESTING* state, i.e., waiting for *CReply* from *x*, it selects another representing node at Level *k* (an arbitrary member of $C_{k,1}$) and sends *CRequest* to its new representing node.

2) *In case of Node Recovery*

Let, *x* recovers after its failure. Now it can participate in ME solution starting with 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 *CReply* to *y*. So, after recovery, *x* should take care whether *y* is still using its consensus. 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 *NodeRecovery* 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 *NodeRecovery* message. In this case, *x* will receive *Release* message from *z* after a period of time, when *z* finishes the use of *x*'s consensus. Until getting *Release* message from *z*, *x* will not start to participate in ME algorithm.
- When *x* recovers, it sends a *NodeRecovery* 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 (*CReply*) of *x* at present, it sends back a *CRelease* message to *x* immediately. Otherwise, *y* does not respond to *NodeRecovery* message. In this case, *x* will receive *CRelease* message from *y* after a period of time, when *y*'s *BUSY* state becomes false. Similar to the previous case, until getting *CRelease* message from *z*, *x* will not start participating in ME algorithm.

Remember 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 making 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 locking node and represented node saved in stable storage so that these data can be retrieved after recovery." After recovery all other states and variables are initialized with their default values.

## IV. SIMULATION

We simulate our solution using PARSEC [22], which is a parallel C–based discrete–event simulation language. We observe different performance metrics when the number of nodes in the network increases or the availability of nodes changes.

Two performance metrics are considered in our simulation: *Message Cost* and *Waiting Time* per CS Entry. *Message Cost* is the average message complexity per request to enter CS, i.e. the average number of messages required for a node from the request placing to getting consensus for entering CS. *Waiting Time* is defined as the average time that a node spends in waiting for the CS after its request is placed.

### A. Simulation Environment and Parameters

We simulate a peer to peer to network consisting an arbitrary number (up to 1200) of nodes (entities in PARSEC) randomly spread over the network. Though the performance of the proposed solution is analytically significant for a large number of nodes, more than 1200 nodes are not possible because of the limitation of this simulation tool. Each run of the simulation has been triggered for 4000000000 STU (simulation time unit) on the average.

We assume that the network ensures the ordered delivery of messages between source and destination. So, Communication latency between two specific nodes is taken as constant. Latency time follows a normal distribution with mean 12 STU and variance 50% of the mean. The time required for preparing and delivering a message (processing time) also follows a normal distribution with mean 8 and variance 50% of the mean. 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 but with different scales of magnitudes.

A node requests for the CS following a Poisson process with 0.0000002 (arrival) rate of requests. Thus, the delay between two requests for CS of a node is an exponential random variable with mean 5000000 STU. Node failures are modeled as a Poisson process with a failure rate. The recovery/restart time from failure/stoppage of a node follows an exponential distribution with mean 100000. The failure rate is calculated from the availability of a node and the recovery time.

In our simulation we take the assumption that at any instance at least a quorum is available in a cluster. This assumption is important to satisfy each CS request. In Subsection III.*E*.4, the availability of a quorum in our system is discussed. Graphs are also plotted there to show the availability of a quorum in different level of clustering varying number of nodes (*n*) and availability of a node *f*. From the graph we find that in case of a higher (>3) level of clustering, the availability of a quorum becomes very low for $n \leq 1200$ and $f \leq 0.8$, which is not feasible for our assumption. Since *n* cannot be taken as more than 1200 and we vary *n* from 600 to 1200 and *f* from 1 to 0.75 to show the impact on performance metrics, we limit level of clustering from 0 to 3.

### B. Performance Variation of Multilevel Clustering

In the simulation experiment, we have varied the number of nodes (*n*), the availability of a node (*f*) to observe the behavior of the performance metrics (i.e., *Message Cost* and *Waiting Time*) as a function of *n* or *f*. We compare the results of different levels of clustering to find out the effect of multilevel clustering for different values of *n* and *f*. Remember that, for quorum formation in the clusters of different levels tree-quorum algorithm [9] is used.

1) *Message Cost*

Fig. 7(a) and 7(b) demonstrate the existence of optimal level of clustering. Here we have taken a single quorum at a time for requesting in any level of clusters. We find

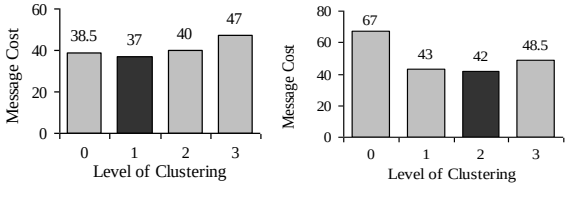


Fig. 7. Message cost of ME algorithm for different level of clustering in two networks with *n*=1200, (a) *f*=0.95 and (b) *f*=0.85.

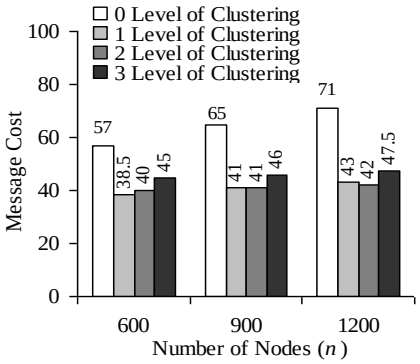


Fig. 8. Message cost of ME algorithm for different values of $n$ using different level of clustering in a network with $f$=0.85.

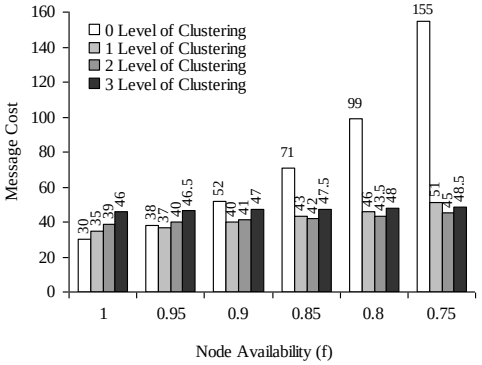


Fig. 9. Message cost of ME algorithm for different values of $f$ using different level of clustering in a network with $n$=1200.

in Fig. 7(a) that, for a network of 1200 nodes with node availability of 0.95 optimal level of clustering is 1. Apparently, it seems that with the increase of level of clustering, the number of nodes in a cluster decreases resulting in reduced message cost. But this is not correct for larger level of clustering. There is optimal level of clustering, which takes minimum message cost. The reason of it is straightforward. Firstly, if level of clustering increases the number of participating clusters along the path of request processing increases. At the same time, the layer to layer communication increases. Secondly, message cost for ME algorithm within a cluster is linearly proportional to the quorum size. The quorum size in tree-quorum algorithm is inversely proportional to the value of $f$ and varies from $\log_2 C$ (for high values of $f$) to $C/2$ (for low values of $f$), where $C$ is the cluster size (see Equation (1)). Thus larger and smaller sizes of clusters are desired for high and low values of $f$ respectively to achieve optimality. These indicate the situations of high and low level of clustering respectively. Ultimately, network size and availability affects message cost for ME algorithm and optimal level of clustering.

Remember Equation (2) derived in Section III for optimal level of clustering. It also justifies our explanation that optimal level of clustering ($\ell$) depends on network size and availability. The optimal point that we see in Fig. 7(b) is different from that we see in Fig. 7(a). This is due to different $f$ (0.85), though $n$ is the same. Hence, optimal level of clustering is not fixed. It changes with the change of network parameters $n$ and $f$. The impact of $n$ and $f$ on optimal level of clustering is demonstrated in Fig. 8 and Fig. 9 respectively.

2) *Waiting Time per CS Entry*

Waiting time for a CS entry is plotted in Fig. 10 and Fig. 11 against different network sizes ($n$) and different values of $f$ respectively. Both of the figures show that a clustered network outperforms a network without clustering at $\ell$>0. From the figures it is found that though message cost at some $k$ level of clustering is lower than that of $k$–1 level of clustering, the waiting time at former $k$ level of clustering is still higher than that of the latter. The reason behind it is that the higher depth for sequential transmission of *CReply* and *PreCRequest*.

### C. Performance Comparison

The aim of this subsection is to compare the performance of our proposed algorithm with that of [25]. For comparison we take two performance metrics: *message cost* and *waiting time* per CS entry. For this comparison, we consider the result for different $f$ at the optimal level of clustering only for each of the algorithms. We also take two other algorithms to compare with our algorithm, so that the results appear more comprehensive. These are the classic distributed ME algorithm of Agarwal and Abbadi [9] and the naïve primary/backup

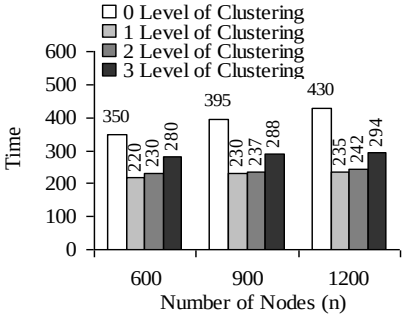


Fig. 10. Waiting time per CS entry of ME algorithm for different $n$ using different level of clustering with $f$=0.85.

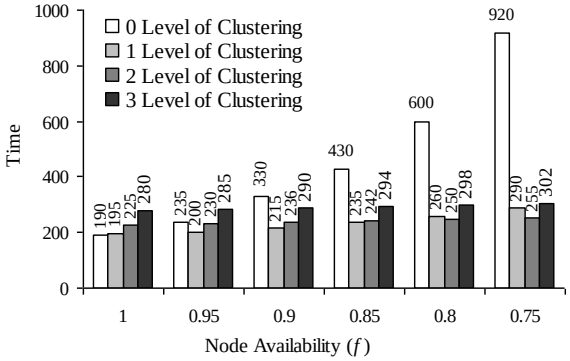


Fig. 11. Waiting time per CS Entry of ME algorithm for different $f$ using different level of clustering with $n$=1200.

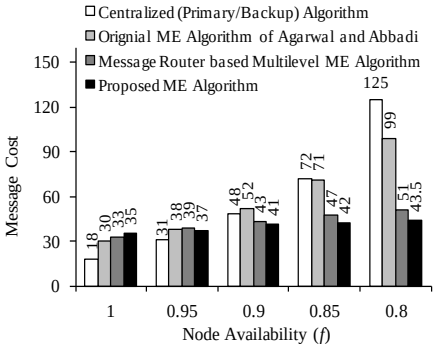


Fig. 12. Message cost per CS entry of different ME algorithms for different *f* with *n*=1200.

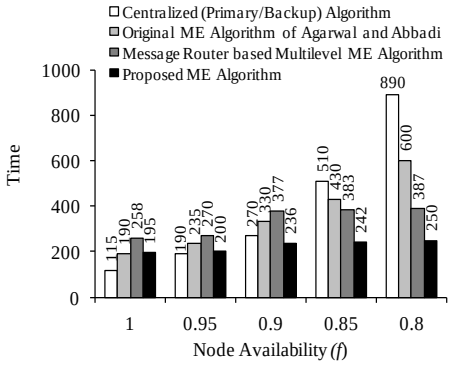


Fig. 13. Waiting time per CS entry of different ME algorithms for different *f* with *n*=1200.

centralized ME algorithm of Alsberg and Day [39]. In the latter case, we take one primary server and one backup server. Note that our algorithm, when $\ell$ = 0 (no clustering), it is actually the mutual exclusion algorithm of Agarwal and Abbadi. No coordination or maintenance takes place at $\ell$=0.

1) *Message Cost*

Message costs per CS entry for the centralized algorithm of [39], classic tree-quorum based ME algorithm of [9], the multi-level ME algorithm of [25] and our proposed multilevel ME algorithm are plotted in Fig. 12 against different availability of nodes (*f*). For the last two algorithms, we plot only the optimal message cost at different *f*. Figure 12 shows that our algorithm simply outperforms the others especially when *f* decreases. It is worth to mention that, if we decrease the recovery time for a particular *f*, the failure rate will increase and as a result the performance of [25] will be worse. The same will occur in case of reduced request rate. In case of primary/backup centralized algorithm, though it does not have single-point failure, it still suffers from two-points of failure and failover time during which requests can be lost. Hence, with the decrease of *f*, performance of the centralized algorithm deteriorates significantly.

*2. Waiting time:*

Waiting times per CS entry for the three ME algorithms are plotted in Fig. 13 against *f*. For multilevel algorithms, we plot the waiting only for those levels of clustering where we get optimal message costs. Figure 13 shows that our proposed algorithm outperforms ME algorithms of [39], [9], and [25], especially when *f*<0.95. Though both of the ME algorithm of [25] and our proposed algorithm use multilevel network hierarchy, the latter takes less waiting time because of its parallel processing.

## V. CRITICAL EVALUATION

The availability of a quorum in our solution is lower than the original classic algorithm [9] at low *f* along with small *n*, which is depicted in Fig. 5. 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, 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 come up to keep consistency. However, we consider this issue as a topic for further research.

Multilevel organization of a network according to the optimal level of clustering is important for best performance. If the members of a network change significantly, it is better to reorganize the network according to the new optimal level of clustering. If this kind of change is frequent, every time it will incur cost for reorganization.

## VI. RELATED WORK

Token based ME algorithm proposed in [5] takes $O(\log n)$ message cost, where *n* is the number of nodes, which suffers significantly if the node holding the token fails or token is lost. Nisho, in [12], presented a highly resilient, although still complex, token based ME algorithm based on Suzuki-Kasami's algorithm [4].

Ricart and Agrawala [7] proposed the first permission based ME algorithm where a node needs to collect permissions from all other nodes for the CS access. Though the algorithm is deadlock and starvation free, it is vulnerable to node and communication failures and it is expensive in terms of communication cost too. 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. Quorum based algorithms [10] are resilient to node and communication failures and network partitioning. Communication cost of these algorithms is proportional to the quorum size. Therefore, these algorithms try to achieve two goals: small quorum size and a high degree of fault tolerance. The majority quorum

algorithm [17] can be considered as the first algorithm of this kind where to attain mutual exclusion a node must obtain permissions from a majority of nodes in the network. Maekawa [8] proposed an ME algorithm by imposing a *logical structure* on the network where a node needs consensus from a specific set of $O(\sqrt{n})$ nodes to achieve ME.

Garcia-Molina and Barbara [18] 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 coteries an efficient and fault tolerant quorum generation algorithm for ME is proposed by Agarwal and Abbadi [9]. Here the nodes form a logical binary tree to generate quorums. The quorum can be regarded as an attempt 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. The 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.

Sometimes the nodes in a network are divided into several groups where each group is often called a cluster. Ahmed and Trelhel [13] proposed a prioritized group based hybrid algorithm combining token based approach with permission based approach. Two distributed ME solutions are presented by Erciyes [14] using a logical structure where clusters are arranged on a ring. Bertier proposed two token based algorithms [15] using the hierarchical network topology, which reduce both latency cost and number of messages. These two solutions are modifications of Naimi's token based algorithm [6] for proxy based cluster. As these algorithms are mainly token based, they suffer due to token failures. Moreover, though the number of participating nodes is reduced to an extent in these approaches, further reduction is required for extra large networks. ME solutions, like our proposed algorithm, that apply higher level of hierarchy are suitable in such case.

## VII. Conclusion

We have proposed a multilevel 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 compared to other related algorithms.

Group mutual exclusion (GME) is a recent variant of the classical mutual exclusion problem, which was proposed first in [35]. We are going to extend our hierarchical approach for the solution of GME problems. Our multilevel technique can be extended for ad-hoc network to achieve better performance as the rate of node failures is usually high in ad-hoc networks.

## Acknowledgement

We would like to thank Reazul Karim Mazumdar, MphasiS Corp., Houston, Texas, USA, for his precious comments on the paper that helps us to improve the work.

## References

[1] M. Raynal, "A simple taxonomy for distributed mutual exclusion algorithms", *ACM SIGOPS Operating Systems Review*, Vol. 25, No. 2, pp. 47-50, 1991.

[2] G. Le Lann, "Distributed systems: towards of a formal approach", *IFIP Congress, North-Holland*, pp. 155-160, 1977.

[3] A.J. Martin, "Distributed mutual exclusion on a ring of processes", *Science of Computer Programming*, Vol. 5, No. 3, pp. 265-276, 1985.

[4] I. Suzuki and T. Kasami, "A distributed mutual exclusion algorithm", *ACM Transaction on Computer Systems*, Vol. 3, No. 4, pp. 344-349, 1985.

[5] K. Raymond, "A Tree based Algorithm for Distributed Mutual Exclusion", *ACM Transactions on Computer Systems*, Vol. 7, No. 1, pp. 61–77, February 1989.

[6] 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, no. 1, pp. 1–13, 10 April 1996.

[7] G. Ricart and A. K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks", *Communications of the ACM*, Vol. 24, No. 1, pp. 9–17, January 1981.

[8] 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.

[9] D. Agarwal and A. El Abbadi, "An Efficient and Fault–Tolerant Solution for Distributed Mutual Exclusion", *ACM Transactions on Computer Systems*, Vol. 9, No. 1, pp. 1–20, February 1991.

[10] P. C. Saxena and J. Rai, "A survey of permission-based distributed mutual exclusion algorithms", *Elsvier Science Publishers B. V.*, Vol. 25, Issue 2, pp. 159-181, May 2003.

[11] J. Misra, "Detecting termination of distributed computations using markers", *Proceedings of the 2nd ACM Annual Symposium on Principles of Distributed Computing*, pp. 237–249, August 1985.

[12] 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, No. 3, pp. 344-355, 1990.

[13] 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.

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

[15] 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.

[16] L. Lamport: "Time, clocks, and the ordering of events in a distributed system.", *Communications of the ACM*, Vol. 21, No. 7, pp. 558-564, July 1978.

[17] R. H. Thomas, "A majority consensus approach to concurrency control", *ACM Transaction on Database System*, Vol. 4, No. 2, pp. 180-209, June 1979.
[18] H. Garcia-Molina and D. Barbara, "How to Assign votes in a Distributed System", *Journal of the Association for Computer Machinery*, Vol. 32, No. 4, pp. 841-860, 1985.
[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, No. 3, pp. 222-238, 1983.
[20] Scott D. Stoller, "Leader Election in Asynchronous Distributed Systems," *IEEE Transactions on Computers*, Vol. 49, No. 3, pp. 283-284, March 2000.
[21] C. Fetzer and M. Subkraut: "Leader Election in the Timed Finite Average Response Time Model", *The 12th Pacific Rim International Symposium on Dependable Computing*, pp. 375–376, December 2006.
[22] R. Bagrodia, R. Meyerr, and et al.: "PARSEC: A Parallel Simulation Environment for Complex System", UCLA Technical Report, 1997.
[23] C. Olson: "Parallel algorithms for hierachical clustering", *Parallel Computing*, Vol. 21, No. 8, pp. 1331-1325, 1995.
[24] S. Rajasekaran: "Efficient Parallel Hierarchical Clustering Algorithms", *IEEE Transactions on Parallel and Distributed Algorithms*, Vol 16, No. 6, pp. 497-502, June 2005.
[25] M. A. Rahman and M. M. Akbar: "A Quorum Based Distributed Mutual Exclusion Algorithm for Multi-Level Clustered Network Architecture", *Proceedings of Workshop on Algorithm and Computation (WALCOM), Dhaka*, February 2007.
[26] Ralf Steinmetz and Klaus Wehrle (Eds). "Peer-to-Peer Systems and Applications", *Lecture Notes in Computer Science*, Vol 3485, September 2005.
[27] 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, No. 15, pp. 1437-1466, December 2002..
[28] DeadLine- a free equation solver, http://deadline.3x.ro/, http://www.soft32.com/Download/Free/DeadLine/4-7467-1.html.
[29] F. Cristian and C. Fetzer; "The Timed Asynchronous Distributed System Model", *IEEE Transactions on Parallel and Distributed Systems***,** Vol. 10, No. 6, pp. 642–627, June 1999.
[30] T. D. Chandra and S. Toueg: "Unreliable failure detectors for reliable distributed systems", *Journal of the ACM*, Vol. 43, No. 2, pp. 225-267, March 1996.
[31] Raimundo Jose and Araujo Macedo: "Failure Detection in Asynchronous Distributed Systems", *Proceedings of II Workshop on Tests and Fault-Tolerance*, pp. 76-81, July 2000.
[32] C. Fetzer: "Perfect Failure Detection In Timed Asynchronous Systems", *IEEE Transactions on Computers*, Vol. 52, No. 2, pp. 99-112, February 2003.
[33] Pragyansmita Paul and S. V. Raghavan , "Survey of multicast routing algorithms and protocols", *Proceedings of the 15th international conference on Computer communication*, pp. 902-926, August 2002.
[34] M. Ashiqur Rahman, M. S. Alam and M. M. Akbar, "A Two Layer Quorum Based Distributed Mutual Exclusion Algorithm", *Proceedings of the 9th International Conference on Computer and Information Technology (ICCIT)*, December 2006.
[35] M. Ashiqur Rahman, M. M. Akbar and M. S. Alam, "A Permission Based Hierarchical Algorithm for Mutual Exclusion", *Special Issue, Journal of Computers*, Vol. 5, No. 12, December 2010.
[36] Y. J. Joung, "Asynchronous Group Mutual Exclusion", *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing*, pp. 51–60, June 1998.
[37] Mike Burrows, "The Chubby lock service for loosely coupled distributed systems". In *Proceeding of OSDI*, 2006.
[38] ZooKeeper, http://sourceforge.net/projects/zookeeper.
[39] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, The primary-backup approach", *Distributed Systems (2nd Edition), ACM Press, NY, USA*, pp. 199–216, 1993.

**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.

**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 a Professor in Department of CSE, BUET.