Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2013, Article ID 703414, 15 pages
http://dx.doi.org/10.1155/2013/703414

Research Article

A Hybrid Distributed Mutual Exclusion Algorithm for Cluster-Based Systems

Moharram Challenger,[1,2] Elif Haytaoglu,[1] Gorkem Tokatli,[1] Orhan Dagdeviren,[1] and Kayhan Erciyes[3]

[1] *International Computer Institute, Ege University, 35100 Izmir, Turkey*
[2] *Department of Computer Engineering, Shabestar Branch, Islamic Azad University, 53815 Shabestar, Iran*
[3] *Department of Computer Engineering, Izmir University, 35140 Izmir, Turkey*

Correspondence should be addressed to Moharram Challenger; m.challenger@gmail.com

Received 26 April 2013; Accepted 10 June 2013

Academic Editor: Guanghui Wen

Distributed mutual exclusion is a fundamental problem which arises in various systems such as grid computing, mobile ad hoc networks (MANETs), and distributed databases. Reducing key metrics like message count per any critical section (CS) and delay between two CS entrances, which is known as synchronization delay, is a great challenge for this problem. Various algorithms use either permission-based or token-based protocols. Token-based algorithms offer better communication costs and synchronization delay. Raymond's and Suzuki-Kasami's algorithms are well-known token-based ones. Raymond's algorithm needs only $O(\log_2(N))$ messages per CS and Suzuki-Kasami's algorithm needs just one message delivery time between two CS entrances. Nevertheless, both algorithms are weak in the other metric, synchronization delay and message complexity correspondingly. In this work, a new hybrid algorithm is proposed which gains from powerful aspects of both algorithms. Raysuz's algorithm (the proposed algorithm) uses a clustered graph and executes Suzuki-Kasami's algorithm intraclusters and Raymond's algorithm interclusters. This leads to have better message complexity than that of pure Suzuki-Kasami's algorithm and better synchronization delay than that of pure Raymond's algorithm, resulting in an overall efficient DMX algorithm pure algorithm.

1. Introduction

Using shared resources among different processes is a primary need in distributed systems. For this reason, distributed mutual exclusion (DMX) has drawn great attention over the years and a good number of algorithms have been proposed in this area. These algorithms are used in distributed systems such as mobile ad hoc networks (MANETs), sensor networks [1, 2], grids, and distributed databases. Messages sent for acquiring and releasing CS are an important measure for DMX algorithms and have a great effect on system's overall performance. Safety, liveness, and fairness are the main requirements for any mutual exclusion algorithm. Lamport's algorithm [3] and Ricart-Agrawala's (RA) [4] algorithm are considered as two of the most important fair distributed mutual exclusion algorithms in the literature. Generally, DMX algorithms can be classified into two major groups, token-based algorithms and permission-based ones. In the first case, a node enters a CS after receiving permission from all of the nodes in its set for the critical section. For token-based algorithms, however, processes are on a logical ring and possession of a system-wide unique token provides the right to enter a critical section. Suzuki-Kasami's algorithm [5] and Raymond's tree-based algorithm [6] are milestone token-based algorithms.

Suzuki-Kasami's algorithm has a low synchronization delay of one message between each two consecutive CS entrances, meanwhile its message communication number per any CS is N and that is fairly high for vast distributed systems which can restrict system scalability. On the other hand, Raymond's algorithm requires low message communication

for each CS entrance, but its delay is order of $O(\log_2(N))$ messages between two consecutive CS entrances.

In this study, a new hybrid DMX algorithm is proposed that is called Raysuz. Raysuz's algorithm uses a clustered graph infrastructure. Suzuki-Kasami's algorithm is run inside the clusters and Raymond's algorithm is run among cluster leaders. Therefore it has better synchronization delay than pure Raymond's algorithm and better message complexity than pure Suzuki-Kasami's algorithm.

In addition, cluster leaders collect internal CS requests and serve them once they receive token from other clusters. This prevents ping-pong style communication of token which is the matter of issue in Raymond's algorithm. Also using shortest paths inside clusters, Raysuz's algorithm gains even better performance than Suzuki-Kasami's algorithm too.

In rest of the paper, we have presented related work in Section 2. In Section 3, the new hybrid algorithm is described both formally and informally. The description of the algorithm is supported by the sample scenario in this section. Then in Section 4, the proposed algorithm is evaluated. Finally, in Section 5, we discuss the proposed algorithm and conclude the paper.

2. The Literature Study

In this section the related studies are elaborated and some performance metrics are discussed which are used in comparison of related algorithms.

2.1. Related Work. DMX algorithms can be classified as permission-based and token-based. In the permission based approaches, if a node needs to enter the CS, it should take permission from all other nodes. There are many algorithms which use this approach, for example, the studies of Singhal [7], Maekawa [8], Agrawal and El Abbadi [9], and Lodha and Kshemkalyani [10]. In the Lamport's algorithm, a node which needs to enter CS should broadcast its CS request, wait for acknowledge from all nodes, and finally enter the CS. After exiting CS, the node should broadcast a release message indicating *I have exited the CS*. This algorithm sends $3(N-1)$ messages for each CS [3].

In order to reduce message complexity, Ricart Agrawala has made improvements to Lamport's algorithm which sends only $2(N-1)$ messages per each CS [4]. Ricart Agrawala achieved this by removing *release message* step of Lamport's algorithm. Instead, the node exiting the CS only sends release message to the nodes which have sent request messages and wait for permission. A queue for holding requests that come from other nodes is also added to the algorithm.

Agrawal and El Abbadi [9] and Maekawa [8] have proposed quorrum-based algorithms which dramatically reduce the message complexity and belong to permission-based approach [8, 9]. Agrawal and El Abbadi use tree-structured quorums which require permission from only $O(\log_2(N))$ nodes in best case, and $O(N)$ in the worst case. Maekawa proposed a new DMX algorithm which only uses $c\ \sqrt[2]{N}$ messages to create a mutual exclusion in a computer network. The network consists of number of subsets whose intersection set is not empty. DMX algorithms in [11–15] are also permission-based algorithms.

Additionally, there are also a number of algorithms which use token-based DMX approach [5, 6, 16–21]. Main idea of token-based algorithms is that the node having the token will have opportunity to enter CS. One of the most popular approaches is Suzuki-Kasami's algorithm [5] which uses N messages per each CS. Another one is Raymond's tree based algorithm [6] which reduces the message complexity using its dynamic tree structure. Raymond's and Suzuki-Kasami's algorithms are basis for our new approach which is described in Sections 2.2 and 2.3. In [22], a new DMX algorithm which is based on path reversal is proposed.

Although many DMX algorithms exist, the state-of-the-art technologies still require adapted DMX algorithms for their circumstances. For example, Edmondson et al. [23] propose a QoS-enabled DMX algorithm for public clouds [24].

2.2. Performance Metrics. Performance bounds of DMX algorithms can vary due to the network load. When a few number of nodes want to enter a CS, the network is assumed as lightly loaded; otherwise it can be called highly loaded. There exist some metrics for evaluating performance and efficiency of DMX algorithms. "*number of messages per request*" is one of them which denotes the total number of messages used for entering CS. It is a critical metric for determining limits of required network bandwidth. Another key metric is "*response time*" which implies the time interval between one node's requesting of a CS permission and entering the CS. The last metric is "*synchronization delay.*" It denotes the latency between one node exiting and the next permitted node entering the CS.

2.3. Suzuki-Kasami's and Raymond's Algorithms. One of the most important token-based DMX algorithms is Suzuki-Kasami's algorithm [5]. It works on fully connected network and its main idea is to reduce the synchronization delay. The algorithm has three data structures. The first one is N-sized array named *request* which is used for holding requests of all nodes in the network. The number of requests which is made by some node i is stored in i_{th} place of request array. When a new request comes from node j, j_{th} element of the request array is incremented by one. The other two structures are stored only in token. One of them is an N-sized array named *last* which holds the number of CS entrances for each node. *SuzQ* is a queue for holding node identities that are waiting for the token.

When node i wants to enter the CS, it increments its request number on the *request* array and broadcasts it. Upon receiving request from node i, any node $j \neq i$ updates its ith element of *request* array with new value of variable in arriving message. After exiting the CS, a node compares the *last* and *request* arrays and enqueues node v if v_{th} element of request array is one more than the v_{th} element of *last* array and v does not already exist in queue.

Suzuki-Kasami's synchronization delay is lower than Raymond's algorithm (0 or T) but its "*number of messages*

per request" is higher than many of the other token-based algorithms (N).

Considering message complexity, Raymond's algorithm is one of the most powerful algorithms amongst token-based solutions for DMX [6]. The algorithm is based on unrooted minimum spanning tree. The tree can be physically or logically built. Each node does not need to know about all the network topology; they only have to know about their neighbors.

Entering a CS for a node requires having the *PRIVILEGE*. Every node in the tree has a variable named *HOLDER* indicating the direction to its neighbor which is on the shortest path to the node that has the *PRIVILEGE*. If the node owns the *PRIVILEGE*, its *HOLDER* variable shows itself. If a nonprivileged node wants to enter the CS, it sends *REQUEST* message to *HOLDER*. When a nonprivileged node receives the *REQUEST* message from its neighbor, it sends the *REQUEST* message to its *HOLDER* node. This process continues until the *REQUEST* message reaches the privileged node. When the privileged node receives request message and finishes executing its CS, it becomes a nonprivileged node and sends *PRIVILEGE* message to the sender. If the privileged node receives a *REQUEST* message while executing its CS, it sends *PRIVILEGE* message to the sender after it finishes CS.

As can be inferred from the description of the algorithm, the "*number of messages per request*" is decreased dramatically in comparison with the other DMX algoritms. However, this improvement comes with a trade-off. While achieving a reasonable decrease on the "*number of messages per request,*" "*synchronization delay*" increases. In Table 1, performance metrics of Raymond's tree-based algorithm are shown as well as information for some other algorithms.

2.4. Comparison. In this section, some of the DMX algorithms are evaluated with respect to performance metrics and their results, as shown in Table 1. In centralized algorithm, three messages are sent per any CS both in high and low network loads [16]. Its synchronization delay is measured as $2T$. Its primary drawback is known as single point of failure. However, Lamport's algorithm requires $3(N-1)$ messages per CS, and its synchronization delay is only T [3]. Ricart and Agrawala (RA) have made an improvement and their algorithm uses only $2(N-1)$ messages per CS both in high and low network loads. Also, it has synchronization delay of only T [4].

Maekawa's algorithm is quorum based and sends $3\sqrt[2]{N}$ messages and $5\sqrt[2]{N}$ messages per any CS for low load and high loads, respectively [8]. Its synchronization delay is evaluated as $2T$. Another quorum-based algorithm, Agrawal and El Abbadi's algorithm, uses $O(\log_2(N))$ messages per CS in low load and $(N/2)$ messages per CS in high load [9]. Suzuki-Kasami's algorithm, which is one of the token-based algorithms, uses N messages per CS in low load, and its synchronization delay is to be measured as T [5]. While Suzuki-Kasami's algorithm uses N messages per CS, Raymond's tree-based algorithm uses only $O(\log_2(N))$ messages per CS. Although synchronization delay of Suzuki-Kasami's algorithm is only T, Raymond's algorithm spends $4T$ as synchronization delay [6].

Table 1: Comparing different DMX algorithms.

Algorithm	Criteria		
	Number of messages per CS (low load)	Number of messages per CS (high load)	Synch (CS) delay
Permission based			
Centralized	3	3	$2T$
Message passing			
Lamport	$3(N-1)$	$3(N-1)$	T
Ricart-Agrawala	$2(N-1)$	$2(N-1)$	T
Singhal	$N-1$	$3/2(N-1)$	T
Lodha	$N-1$	$2(N-1)$	T
Quorum based			
Maekawa	$3\sqrt{N}$	$5\sqrt{N}$	$2T$
Agrawal-El Abbadi	$\log_2(N)$	$N/2$	—
Token passing			
Token ring	$N/2$	—	$N/2$
Suzuki-Kasami	N	N	T
Raymond	$\log_2(N)$	$\log_2(N)$	$4T$

As a result, Suzuki-Kasami's algorithm and Raymond's algorithm have less synchronization delay and less message count per CS, respectively. Their message count per CS and synchronization delay is not efficient correspondingly and we have dealt with these weaknesses.

3. Raysuz Algorithm

Suzuki-Kasami's algorithm is known to have lower synchronization delay than Raymond's with the expense of higher message complexity. Raymond's algorithm has much less message complexity, but synchronization delay is higher because of using only the edges of the minimum spanning tree (MST).

In this section, we propose an algorithm which combines the better sides of these algorithms. Initially, the nodes are grouped in clusters and each cluster has a leader node. Suzuki-Kasami's algorithm is used inside clusters and Raymond's algorithm is used to pass token among the cluster leaders. Among the leaders, token can travel only on the edges belonging to the MST paths, but inside clusters shortest paths can be used. The high message traffic of broadcasting CS requests of Suzuki-Kasami's algorithm is decreased by limiting broadcast only inside the clusters. Each cluster leader will be responsible for requesting token from outer clusters and dealing with their requests.

3.1. System Model and Data Structures. Before explaining the algorithm, we describe the system model. Each node has a unique ID and it can only communicate with its neighbors via edges. We also assume that communication channels have FIFO structure. Asynchronous communication model is used

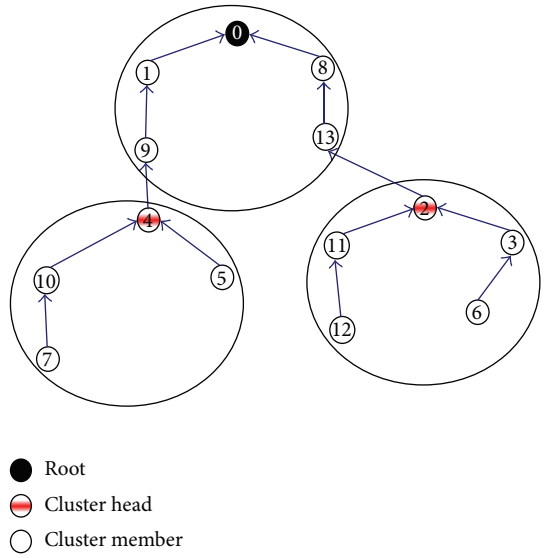

Figure 1: An example network clustered with DSTA.

in this algorithm. It is supposed that the communication channel and nodes are failure free and there are no malicious nodes. We suppose that all of the edges in the network have the same weight. However, other assumptions will not violate our results. Interested readers for weighted graphs and analysis of clustering complex networks with distributed preference mechanism can refer to [25].

Distributed spanning tree-based clustering algorithm (DSTA) [26] can be implemented to cluster the network. DSTA constructs clusters with controllable diameter and the spanning tree of cluster heads (leaders) directing to the root (sink) node as the backbone. The depth parameter of the DSTA is used to adjust the diameter of the clusters. The root node sends *PARENT*(*nhops*) message to its immediate neighbors to start the execution of the DSTA algorithm. When a node first receives *PARENT*(*nhops*) message, it sends *PARENT*(*nhops* + 1 mod *depth*) message to its neighbors. The recipients of the message with *nhops* = 0 are the *CLUSTERHEADS*, and those with *nhops* ≤ *depth* are the *MEMBER* nodes. Figure 1 shows an example network clustered with DSTA, where *depth* = 2. The borders of the clusters are circled and the communication edges are shown with lines.

Inside clusters, Suzuki-Kasami's algorithm is used to pass the token. Broadcasting requests for token in this algorithm will be achieved by unicasting them on the shortest paths for each node in the cluster. The leader only spectate the internal token passing operations. Token passing between clusters is made by cluster leaders using Raymond's algorithm. When internal requests take place, the leader node in the cluster will be responsible for requesting and taking token from other cluster leaders. When the token is in the cluster and other clusters need it, the leader of the cluster steals token from its cluster and sends it to the leader of related cluster.

The ordinary and leader nodes must hold some variables in order to run the algorithm. These variables are as follows.

(i) *Dir*: each node holds the direction which shows the leader of the cluster having the token. This variable helps cluster leaders requests to reach the cluster which has the token.

(ii) *TokInside*: this variable is held by the leader nodes. It states whether the token is inside the cluster or not.

(iii) *Asked*: this variable states whether a Raymond's algorithm token request has been made or not. Using this variable prevents sending token requests for each incoming request. It is used by cluster leaders and other nodes who are making token transfer on the transfer path.

(iv) *RequestArray*: it is an array which is used in the same way as Suzuki-Kasami's algorithm. Related array element is incremented for each internal Suzuki-Kasami's algorithm request.

(v) *LastArray*: this array is used in the same way as *last* array in Suzuki-Kasami's algorithm. It holds the number of times each node in cluster has entered the CS. Nodes increment their array element after finishing critical section. There is only one *LastArray* for each cluster and it travels with internal tokens.

(vi) *SuzQ*: it is the queue which is used by nodes that are waiting to enter CS in the same cluster, the same as Suzuki-Kasami's algorithm. It also travels with internal token, which is always sent to head of this queue. New requests from the same cluster are detected by comparing *request* and *last* arrays and there requester nodes are added to this queue by nodes exiting CS.

(vii) *RaymQ*: this queue is the same as Raymond's algorithm. During the token travel among the cluster leaders, the nodes in the path use this queue. Cluster leaders forward their token to the head of this queue after their clusters have finished their jobs. Ordinary nodes in the path also help the transfer of token by using this queue.

There are also several message types, used in the proposed algorithm, which are listed below.

(i) *InTok*: a token message which lives in the clusters.

(ii) *ExTok*: a token message which travels between cluster leaders.

(iii) *InReq*: it denotes internal Suzuki-Kasami's algorithm requests made within the clusters.

(iv) *ExReq*: it denotes external token requests (handled using Raymond's algorithm) which are sent by the cluster leaders.

(v) *Pulse*: this input denotes the need of entering CS. Only ordinary nodes can have pulse and enter CS.

3.2. Informal Description of Raysuz's Algorithm. In Raysuz's algorithm, we have two types of nodes, namely, ordinary and leader. Ordinary nodes only have information about their own neighbors and use Suzuki-Kasami's algorithm to enter the CS. These nodes broadcast a CS request to their cluster

using the shortest path routing with all possible edges. When the nodes acquire the *InTok*, these nodes will do the same procedure as that of Suzuki-Kasami's algorithm. The only difference between ordinary nodes in pure Suzuki-Kasami's algorithm and Raysuz's algorithm is that the ordinary nodes in the proposed algorithm use Raymond's algorithm procedures to transfer external CS requests and external token between cluster leaders. When an external CS request comes to a node, this node adds the requester to its *RaymQ* and if another external CS request has not been sent earlier, it sends an external CS request towards the direction in *Dir*. Whenever an external token arrives at the node, it sends this token to the head of *RaymQ*, probably itself.

Leader nodes are dedicated to request token from other clusters. The leader nodes can process two threads in order to act as leader and ordinary nodes simultaneously. Their job is to send external token requests when their cluster needs token and give the token back when other clusters need it. They send external requests and external token according to Raymond's algorithm routines. When there is an internal Suzuki-Kasami's algorithm request, *InReq*, inside the cluster, the cluster leader sends an external request to direction, *Dir*, if the token is not in the cluster. Each node in the path will forward the message up to the leader of the cluster having the token. When the leader of the token owner cluster receives the external request, *ExReq*, it sends an ordinary Suzuki-Kasami's request to its cluster, takes the token in its turn, and sends it to the requester. Cluster leaders add a level of indirection and obtain the illusion that there are only cluster leaders which are implementing Raymond's algorithm. They also hide the outer system from their cluster nodes. When the token is outside the cluster, the ordinary nodes will believe that the token is in their cluster leader.

As mentioned above, this clustered algorithm combines the low *synchronization delay* feature of Suzuki-Kasami's algorithm and low message complexity feature of Raymond's algorithms. When an external requests arrives, the cluster leader does not grab and does not send the token immediately. Instead, it makes a Suzuki-Kasami's algorithm request to take and send the token. This behaviour ensures that significant amount of internal CS demands which have arrived before external requests are fulfilled. This prevents token to travel along the graph for each request like in a ping-pong style communication, thus reducing token travelling distance for each request and reducing *synchronization delay* in comparison with pure Raymond's algorithm. Moreover, using the shortest paths for Suzuki-Kasami's operations fastens the token travelling inside the cluster, therefore reduces the *synchronization delay* and *number of messages per request*.

3.3. Formal Description of Raysuz's Algorithm. In this section, Raysuz's algorithm is illustrated formally as two finite state machines (FSM) for both leader and ordinary nodes. The related FSMs for leader and ordinary nodes can be seen in Figures 2 and 3, respectively. The pseudocode of these algorithms and used legends are provided in Table 3, Algorithm 1, and Algorithm 2.

3.3.1. The Leader Node. As it can be seen in the FSM of Figure 2, the leader node has three states, *IDLE*, *WAITEXTOKEN,* and *INTOKENSTEAL*.

IDLE. The leader nodes are in *IDLE* state when the token is in the cluster and there are no external requests, or the token is outside and there are no internal nodes which need to enter CS currently. When the token is inside and external request arrives, leader node makes a Suzuki-Kasami's algorithm request, adds the requester's ID to *RaymQ,* and passes to *INTOKENSTEAL* state. If the token is not inside the cluster and an internal Suzuki-Kasami's algorithm request arrives, the leader adds itself to *RaymQ* and sends an external request to the direction of *Dir* (if not previously sent) and transits to *WAITEXTOKEN* state.

INTOKENSTEAL. The leader nodes are in *INTOKENSTEAL* state when they are waiting for the internal token to send to external requester. When the internal token arrives, the leader looks at its *RaymQ* and *SuzQ* sizes. If their sizes both are one (which is the same the leader itself), it means that there is no more external or internal requester waiting for the token. Therefore, it sends the token to the requester and updates the direction to it and finally goes to *IDLE* state. If one of the queues is greater than one, it means that the token is needed back, either for internal CS demands or external requests. In this case, the leader puts itself into its *RaymQ* and sends external request just after sending the token and finally it transits to *WAITEXTOKEN* state.

WAITEXTOKEN. The leader nodes are in this state when they are waiting for the external token either for internal CS demands or external requests. When the external token for another cluster arrives to the leader, it just forwards the token to head of *RaymQ*, changes direction, and transits its state to *IDLE*. If the external token arrives to the leader, it looks at the size of its *RaymQ*. In case the size of its *RaymQ* is one, this means there are no other external requests waiting, so it creates and sends the internal token to its cluster and goes to *IDLE* state. If it is greater than one, this means that there are other clusters waiting for the token, thus it makes a Suzuki-Kasami's algorithm request just after sending the token into its cluster and goes to *INTOKENSTEAL* state.

The routine which takes place in a leader of any cluster is clearly defined in the Algorithm 1. This algorithm consists of six steps. Step one is triggered when the cluster has no token inside and the leader receives a request, either from outside or inside cluster. In both cases, it transits to *WAITEXTOKEN* state but in earlier case the leader puts external request in the *RaymQ* while in latter case it puts its own request in this queue.

Step two describes the state which occurs when the token owner cluster's leader receives a token request. The actions to take place are determined with regard to whether the request comes from intracluster or not. The former one forces leader to operate ordinary Suzuki-Kasami's algorithm's related part. In the other case, the leader makes Suzuki-Kasami's request

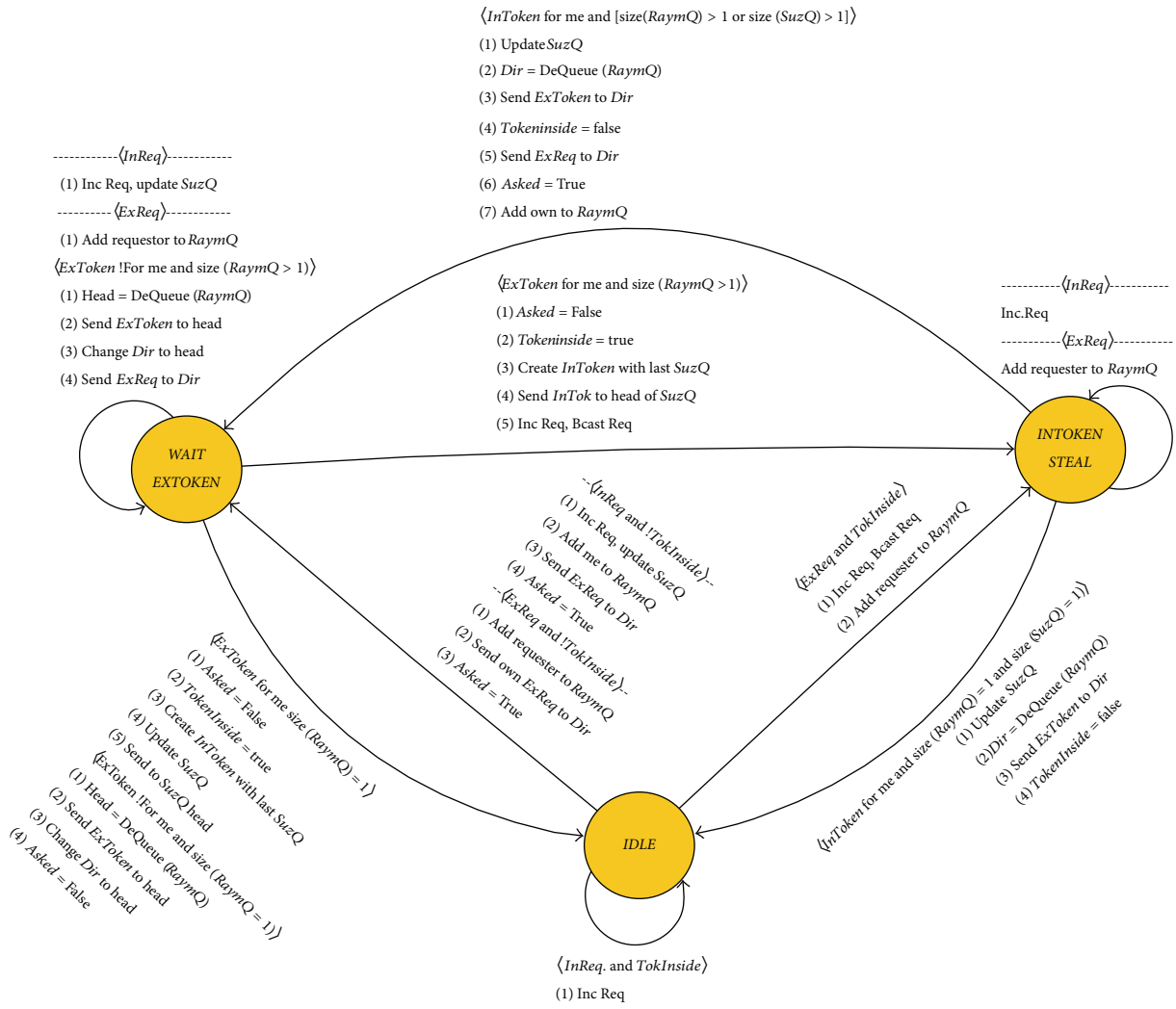

Figure 2: Leader nodes' finite state machine.

on behalf of the original requester to steal the token from its own cluster to send it outside of the cluster (to the origin of the request). At this point, the leader's state is changed to *INTOKENSTEAL* state.

In step three the leader which is in token steal state receives it and sends the token to the requester's cluster. Until receiving the token, this leader reacts with a new request from other clusters, if any, in step four. Step five deals with the leaders that are waiting for the external token and receive a request from other clusters. Nevertheless, the actions listed in step six will be taken if this leader gets the token for its cluster.

3.3.2. The Ordinary Node. The ordinary node has also three states, *IDLE*, *HAVEREQUEST,* and *HAVETOKEN*.

IDLE. This state means that the ordinary node does not have a CS demand. It follows Raymond's algorithm routine when external requests or the external token arrives. When a pulse indicating CS demand arrives, it makes Suzuki-Kasami's algorithm request and goes to *HAVEREQUEST* state.

HAVEREQUEST. The node waits for the token in this state. When internal token arrives, it enters the CS. Upon exiting the CS, it checks the *SuzQ*. If it is empty, this means there are no token waiting nodes in the cluster. Thus, it goes to *HAVETOKEN* state. If *SuzQ* is not empty, it sends the token to the head of *SuzQ* and goes to *IDLE* state.

HAVETOKEN. This state means that there are no more token waiting nodes in the cluster and the token remains at the node after the CS; then it can freely enter the CS whenever it needs. When an internal request arrives, it sends the token to the requester and transits to *IDLE* state.

The routine which takes place in an ordinary node is defined in the algorithm of Algorithm 2. This algorithm contains seven steps. The first step deals with any idle or token requested ordinary node which receives a request from any other node. In second step, the actions required upon receiving an external token for idle nodes or nodes which have just requested a token are listed. The event of receiving

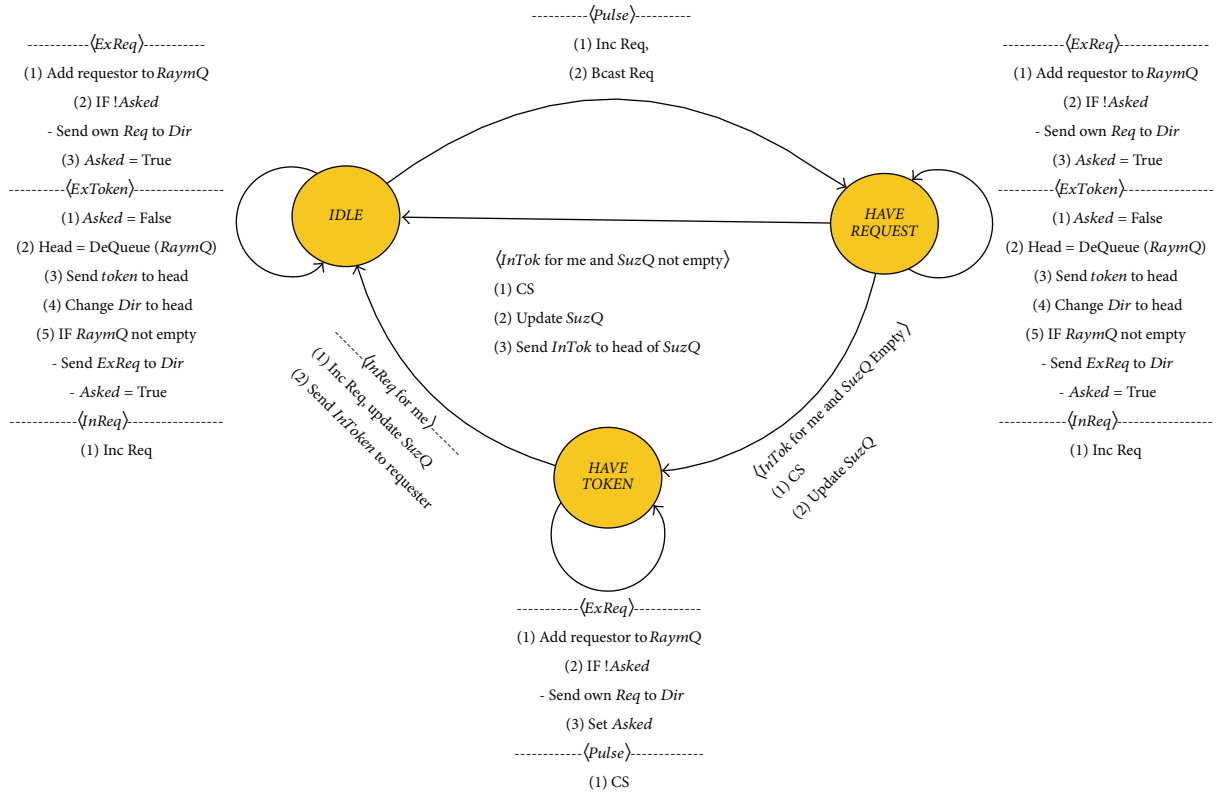

Figure 3: Ordinary Nodes' Finite State Machine.

a pulse enforces the idle node to have a request in the third step.

The fourth and fifth steps demonstrate the original related part of Suzuki-Kasami's algorithm in which any requested node receives the token. Finally, the sixth and seventh steps handle receiving request and pulse, respectively, for an ordinary node which is in the *HAVETOKEN* state.

3.4. A Sample Scenario Using Raysuz's Algorithm. For clarity of Raysuz's algorithm, we present a sample scenario for the algorithm execution. According to the algorithm, the network is clustered by an algorithm such as [26, 27]. In this scenario, initially, the token belongs to a node T, in cluster C_0. There are some requests from other nodes in this scenario: firstly, N_0 in C_0 requests the token. Then the other node, N_1 in C_1, asks for the token shortly after N_0's request. Meanwhile, N_2 in C_2 also wants to grab the token. This scenario shows how the algorithm deals with these requests.

The first event of the scenario can be seen in Figure 4(a). Firstly, node N_0, which needs the token, makes a Suzuki-Kasami request and broadcasts it to its cluster. Since it is the first node which has requested the token, it receives the token immediately as shown in event 1 in Figure 4(b). After that, node N_1 in cluster C_1 requests the token from its own cluster with broadcasting a Suzuki-Kasami's request as shown in event 2 in Figure 4(b). Due to lack of the tokens in this cluster C_1's leader L_1 sends the request to the leader of the cluster which is the owner of the token, L_0, as in Figure 5(a). After L_0 receives the token request, it makes a Suzuki-Kasami request as it does for itself, as shown in event 1 in Figure 5(b). Shortly after that, node N_2 from another cluster, C_2, wants the token and makes a request to its own cluster as shown in event 2 in Figure 5(b). Since the token is not in C_2, its leader, L_2, sends the request to N_0's cluster leader as shown in event 1 in Figure 5(c).

At this point, N_0 has the token and N_1 and N_2 are waiting for it. When N_0 quits the critical section, it sends the token to the leader of its cluster as shown with event 2 in Figure 5(c) and the leader forwards it to the leader of C_1 accompanied with a request (for request of N_2) as shown in event 1 in Figure 5(d). Upon receiving the token by L_1, it delivers the token to N_1, which has asked for it, and then L_1 broadcasts an internal request in its cluster to steal the token and send it back to L_0 as shown in event 2 in Figure 5(d).

As soon as N_1 exits the critical section, due to prior internal request, it sends the token to L_1 as shown in event 1 in Figure 5(e). Afterwards, the leader returns the token to L_0 as shown in event 2 in Figure 5(e). Then, L_0, according to the requests saved in its queue, sends the token to L_2, which delivers it to N_2 as shown in events 3 and 4, respectively, in Figure 5(e).

Now, we examine this scenario deeply according to algorithm's detail to understand what is going on in this algorithm. When node N_0 makes a Suzuki-Kasami request as is shown in Figure 4(a), L_0 only increases the request array in

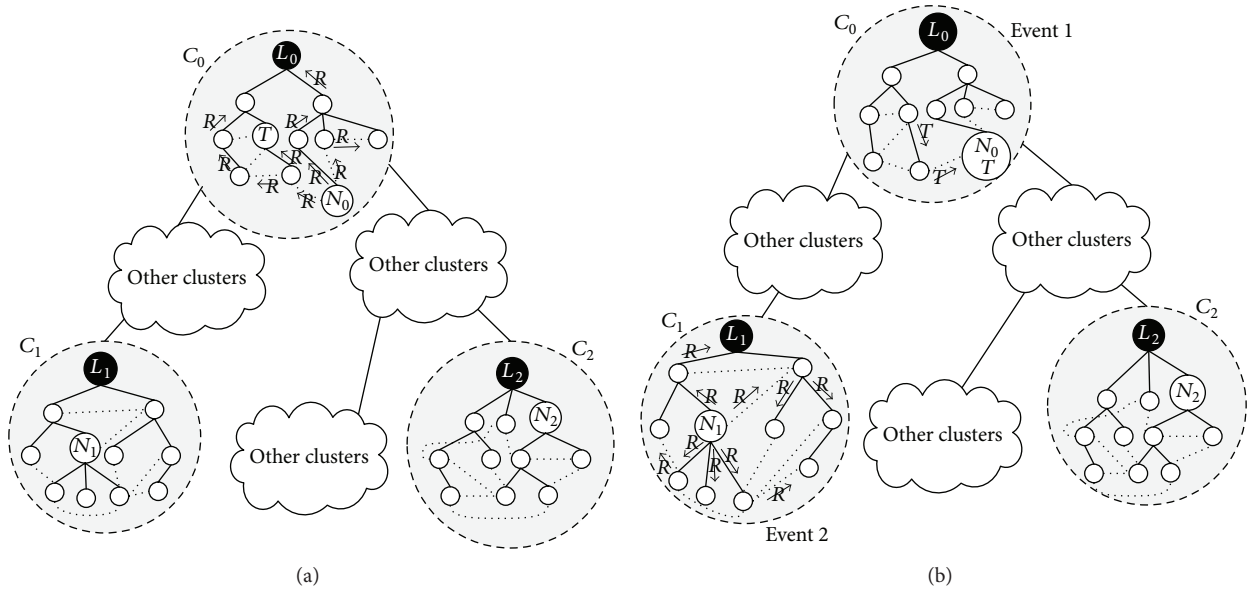

Figure 4: Scenario for Suzuki-Kasami's part of the Raysuz algorithm.

the same way as an ordinary node does in the same cluster. So far everything is straightforward. Then node N_1 in C_1 makes a request in its cluster to enter the CS; see Figure 4(b). Before N_1 requests for the token, N_0 receives the token from the token holder node, as shown in Figure 4(b).

At this time, in C_1, L_1 which is aware of the token absence prepares a request to get token and sends it to node whose identity is the same as the *Dir*'s content, as shown in Figure 5(a). Upon receiving the external request from N_1, L_0 adds N_1's request to its *RaymQ* and prepares a request to get the token from its own cluster. This is achieved by making a Suzuki-Kasami request as shown in event 1 of Figure 5(b), as if L_0 wants the token for itself to give it to N_1. While L_0 is dealing with acquiring the token, another node, N_2 in C_2, needs to enter the CS, so it makes a Suzuki-Kasami request as shown with event 2 in Figure 5(b). After some while, L_2 sends an external request to L_0 to get the token as shown in event 1 in Figure 5(c). Then L_0 adds this node (L_2) to its own *RaymQ*. In this way, L_0 knows that there is another node, N_2, which wants to grab the token after N_1. Later on, N_0 sends back the token to L_0 as shown in event 2 in Figure 5(c).

Now, L_0 has the token and dequeues *RaymQ* and sets *Dir* variable with dequeued node. Afterwards, it sends the token to dequeued node which is the first external node which wants token from L_0. Since the *RaymQ* is not empty, L_0 also sends an external request along with the token as shown in event 1 in Figure 5(d).

As L_1 receives the request and the token, it queues the external request to the *RaymQ* and forwards the token to the N_1 using *SuzQ*. Taking the request into consideration, L_1 also broadcasts a Suzuki-Kasami request in the cluster, as can be seen in event 2 of Figure 5(d). Since there is no other internal request in C_1, after exiting the CS, N_1 forwards the token to the L_1 as shown in event 1 of Figure 5(e).

At this point, L_1's *RaymQ* is not empty, so it sends the token to the *RaymQ*'s head (in this case L_0 with dequeuing it as shown in event 2 of Figure 5(e)). Analogous to the previous event, L_0 sends the token to L_2 using its *RaymQ* as shown in event 3 of Figure 5(e).

Finally, L_2, receiving an external token, sends the token to the head of *SuzQ* which is N_2 as shown in event 4 in Figure 5(e). In this way, three nodes received the token in order and the scenario is accomplished.

4. Analysis of the Proposed Algorithm

In this section, we present the theoretical analysis of Raysuz's algorithm along with relevant proofs. This theoretical analysis contains correctness, message complexity, energy consumption, synchronization delay, and response time for the proposed algorithm.

4.1. Correctness of Raysuz's Algorithm. The correctness of Raysuz's algorithm is examined according to safety and liveness attributes of the algorithm.

4.1.1. Safety. Safety of Raysuz's algorithm is analyzed from the single token existence and mutual exclusion points of view, which will be discussed in Theorem 3.

Lemma 1. *In Raysuz's algorithm at most one token exists in the network concurrently.*

Proof. Assume the contrary. In this case, there exists more than one token in the network concurrently. Therefore, according to finite-state machine given in Figure 3, more than one ordinary node should be in *HAVETOKEN* state. Assume that these nodes are in the same cluster. Suzuki-Kasami's algorithm is used in intracluster communication; thus multiple token existence is impossible [5].

On the other hand, assume that more than one node belonging to different clusters is in the CS at the same time.

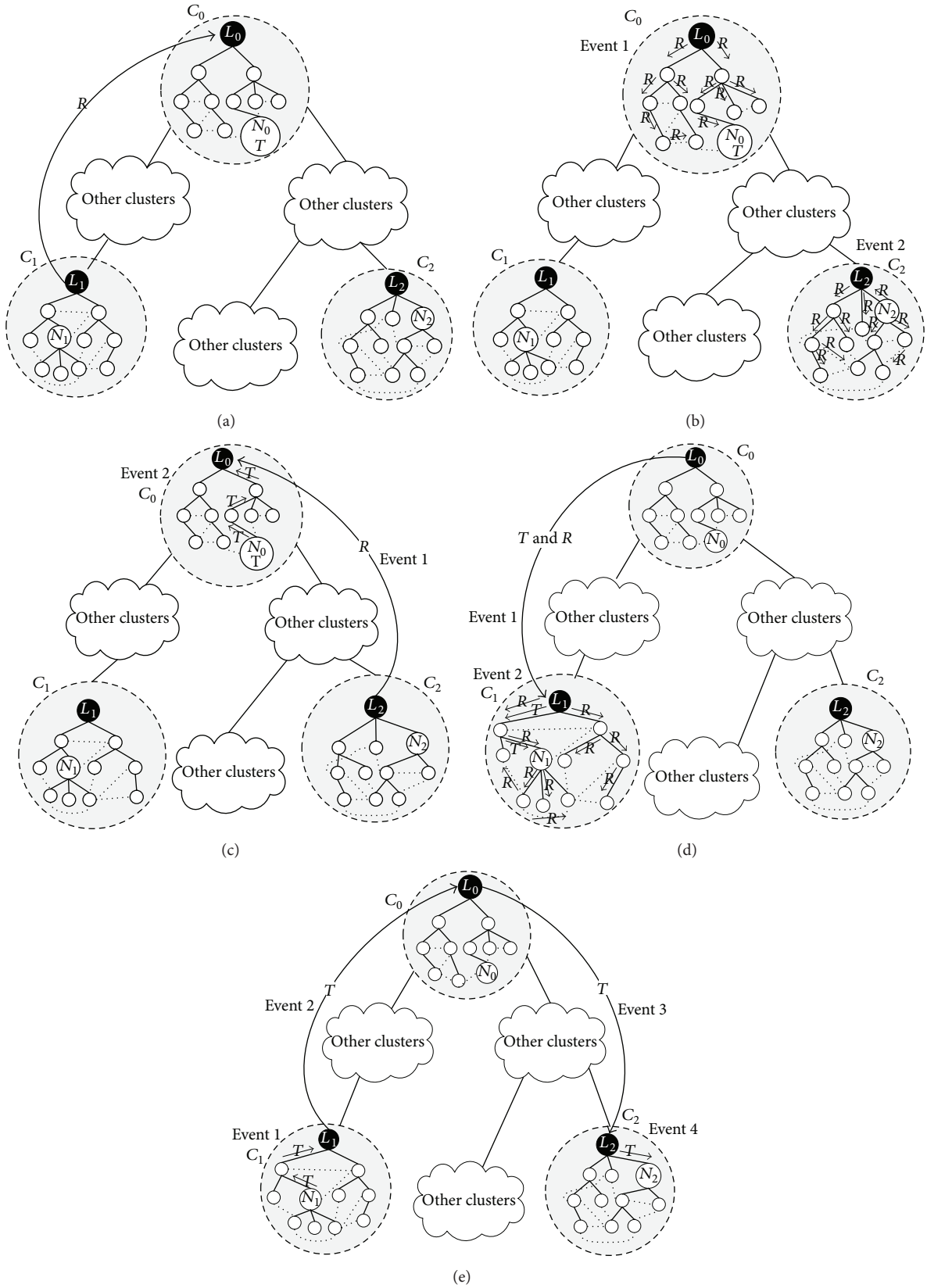

Figure 5: Scenario for Raymond's part of the Raysuz algorithm.

In this case, according to pseudocode of leader node in Algorithm 1, multiple leaders should be in *INTOKENSTEAL* state. Since Raymond's algorithm is used for intracluster communications, this is also not possible [6].

There is no other possibility; therefore we contradict our assumption. □

Lemma 2. *At most one node can be in the CS at any time, ensuring mutual exclusion.*

Proof. Assume the contrary, that more than one node can execute CS at any time concurrently. In Raysuz's algorithm, when a node receives token, it enters CS. If there is more than one token in the network, then more than one node can execute CS at the same time. However, it is proven in Lemma 1 that this case is not possible. Therefore, we contradict our assumption. □

Theorem 3. *There always exists exactly one token in Raysuz algorithm, which also provides at most one node entering CS at any time, thus satisfying Safety attribute.*

Proof. The theorem holds since Lemmas 1 and 2 are true. □

4.1.2. Liveness

Theorem 4. *Raysuz's algorithm is deadlock- and starvation-free.*

Proof. We use Suzuki-Kasami's algorithm for intracluster communication and Raymond's algorithm for intercluster, communication. Both algorithms are deadlock- and starvation free which is proven in [5, 6]. Thus, Raysuz's algorithm is deadlock- and starvation-free. □

4.2. Message Complexity. The message complexities of Raysuz's, Raymond's and Suzuki-Kasami's algorithms are comparatively explained in this section. We assume the topology of the network as multihop cluster tree, where there are C clusters and N nodes.

In the graph consisting of N nodes, the mean distance between any two nodes (also mean unicast distance between two nodes), $E(L_u)$, is almost surely of order $O(\log_2(N)/\log_2(d))$, where d is the weighted average of the sum of squares of the expected degrees [28].

We consider the power law graphs with lots of nodes, so d is a very small value in comparison with N. Thus we use the complexity $O(\log_2(N))$ instead of $O(\log_2(N)/\log_2(d)$ as a mean distance between two arbitrary nodes.

4.2.1. The Worst Case. In Raymond's algorithm, token requester and token holder nodes will be on the farthest sides in the worst situation. In this case, request message will travel up to the root and down to the token holder. Similarly, token holder will send token back in the same path. Thus, the message complexity for a request is $N-1+N-1=2N-2=O(N)$.

In Suzuki-Kasami's algorithm on a tree, the requester will broadcast request on tree, which uses a total of $N-1$ messages. Then the token holder node will receive request and send token to requester using the shortest path routing information in messages. Therefore, the total message count for a request is: $N-1+N-1=2N-2=O(N)$. Raysuz's algorithm has six phases for requesting and entering CS. The phases and number of messages sent in each phase are described as follows.

(i) Requester node sends request to its leader: intra cluster requests are broadcasted along the tree, and one message will be sent for each node in cluster except the requester. This requires $N/C-1=O(N/C)$ messages.

(ii) Leader forwards the request to the token holder node's cluster: in this phase, the two token leaders will communicate. In the worst situation, these leaders will be in the farthest sides of the tree. Then, the distance between them will be $N-1=O(N)$, which is also the number of messages needed in this phase.

(iii) Token holder leader makes Suzuki-Kasami's request: the leader of the token holding cluster will broadcast request to its cluster to take the token. This broadcast needs $N/C-1=O(N/C)$ messages.

(iv) Token holder node sends token to its leader: the token holder node will send token to leader at the cluster root using $O(N/C)$ messages.

(v) Token holder cluster's leader sends token to requester's leader: similar to phase 2, the two token leaders will communicate. Thus, the number of messages needed is $N-1=O(N)$.

(vi) Token requester node's leader sends token to the requester node using $O(N/C)$ messages.

The total of these phases yields $O(N)$.

4.2.2. The Best Case. The best case situation occurs when only the token holder node wants to enter CS. In this case, no message transmission is needed. Therefore, the number of messages needed for Raymond's, Suzuki-Kasami's and Raysuz's algorithm is 0 in the best case.

4.2.3. The General Case. The general case for message complexity is elaborated in this section. The required notation is shown in Table 2. We assume that underlying topology is a balanced tree with height $O(\log_2(N))$.

LHL denotes the level of the token holder leader in the tree. *LHN* denotes the level of token holder node. *LRN* is level of requesting node and *LRL* is the level of requesting leader.

In Raymond's algorithm, request message will travel $LRN+LHN$ hops. Similarly, the token will travel back $LHN+LRN$ hops, resulting in a total of $2(LRN+LHN)$ messages travelling through the path. This yields $O(\log_2(N))$ message complexity for Raymond's algorithm. According to Suzuki-Kasami's algorithm, $N-1$ messages are needed to broadcast the request. The token will be sent to requester in *LHN* number of messages, resulting in $N-1+O(\log_2(N))=O(N)$ messages.

Table 2: Legends of the analysis.

Legend	Description
LHL	Level of the token holder leader
LRL	Level of requesting leader
LHN	Level of the token holder node
LRN	Level of requesting node

Table 3: Legends of the algorithms.

Legend	Description
L_j	Any leader j
L_{hold}	Leader's cluster holding token
$L_{waitExToken}$	Leader who sent request
$L_{tokenSteal}$	Leader who sends Suzuki-Kasami's request to inner cluster
C_j	Cluster j
P_j	Process j
O_j	Any node
O_{idle}	The node which is not interested in token
$O_{haveToken}$	The node which has the token
$O_{haveRequest}$	The node which has requested the token

In Raysuz's algorithm, we have the same 6 phases as in Section 4.2.1.

(i) The requester node sends request to its leader (using Suzuki-Kasami's algorithm) with $N/C - 1$ messages.

(ii) The leader forwards the request to token holder cluster's leader in $LRL + LHL$ number of messages.

(iii) Token holder leader broadcasts Suzuki-Kasami's algorithm request to take the token. The number of messages needed is $N/C - 1$.

(iv) Token holder node sends token to its cluster leader using $LHN - LHL$ messages.

(v) Token holder cluster leader sends token to leader of token requester node in $LRL + LHL$ number of messages.

(vi) Leader of token requesting node sends message to requesting node in $LRN - LRL$ number of messages.

The total of these phases results is $N/C - 1 + LRL + LHL + N/C - 1 + LHN - LHL + LRL + LHL + LRN - LRL = 2N/C - 2 + LRL + LHL + LHN + LRN = O(N/C + \log_2(N))$.

For example, if the number of clusters, C, is big enough, then the average message complexity becomes $\Omega(\log_2(N))$.

4.3. Energy Consumption. Energy consumption is important for wireless systems and especially for MANETs where nodes are battery-powered. Generally transceiver is the dominant energy consumer, so that transmitted byte counts should be minimized to maximize the network lifetime. With regard to this, our energy consumption analysis depends on message size and message complexity. We first analyze message size of Raysuz's algorithm and then we analyze the worst case, the best case, and the general case for the transmitted bit count.

The complexities of size of *ExTok*, *InReq*, *ExReq*, and *Pulse* messages are $O(\log_2(N))$ bits since these messages include constant number of fields (such as *source*, *destination*, and *requestor*) and each of these fields can be represented with numbers in $[0, N]$. On the other hand, the *InTok* message's size is $O(N/C\ \log_2(N))$ bits since Suzuki-Kasami token should include information about all nodes in the cluster.

4.3.1. The Worst Case. In Raymond's algorithm, $O(N)$ messages are sent at the worst case. Since the message size in Raymond's algorithm is $O(\log_2(N))$ bits, the transmitted bit count of Raymond's algorithm is $O(N\log_2(N))$ bits.

At the worst case, the message complexity of Suzuki-Kasami's algorithm is $O(N)$. Token includes information of all nodes, so that the size of the token is $O(N\log_2(N))$ bits. In this case, $O(N^2\log_2(N))$ bits are transmitted in total.

Raysuz's algorithm has six phases in the worst case as given in Section 4.2.1. The transmitted bit count of each phase is added to find the total value as follows: $O(N/C)\ O(N/C\log_2(N)) + O(N)\ O(\log_2(N)) + O(N/C)\ O(N/C\log_2(N)) + O(N/C)\ O(N/C\log_2(N)) + O(N)\ O(\log_2(N)) + O(N/C)\ O(N/C\log_2(N)) = O((N/C)^2\log_2(N))$. At the worst case for $C = 1$, the complexity of transmitted bit count is $O(N^2\log_2(N))$ bits.

4.3.2. The Best Case. At the best case, token holder wants to enter CS. No transmissions are needed; thus, the transmitted bit counts of Raymond's, Suzuki-Kasami's and Raysuz's algorithm are 0 bits.

4.3.3. The General Case. In the general case, Suzuki-Kasami's and Raymond's transmitted byte counts are $O(N^2\log_2(N))$ bits and $O(\log_2(N)^2)$ bits, respectively. The general case of Raysuz's algorithm can be found by adding the complexities of intracluster and intercluster operations as follows: $O(N/C)\ O(N/C\log_2(N)) + O(N/C{+}\log_2(N))\ O(\log_2(N)) = O((N/C)^2\log_2(N) + (N/C{+}\log_2(N))\log_2(N))$. For $C = 1$, the worst case equals $O(N^2\log_2(N))$ bits, the same as the value given in Section 4.3.1. For $C = N$, $\Omega(\log_2(N)^2)$ bits are the lower bound of the general case.

4.4. Synchronization Delay. In synchronization delay formulations, we have used notation T to indicate the unit time for sending a message.

4.4.1. The Worst Case. The worst case synchronization delay of Suzuki-Kasami's algorithm is NT for a network consisting of N nodes, since distance of any two nodes is at most N. Thus synchronization delay of Suzuki-Kasami's algorithm is $O(N)$ at the worst case. The worst synchronization delay of Raymond's algorithm is NT, resulting in $O(N)$ time complexity which is equal to diameter of the tree at the worst case.

The worst case of Raysuz's algorithm has 3 phases. In the first phase token holder finishes executing CS and sends token to its leader. This phase requires $O(N/C)$ time. In the second phase, the token holding leader sends token to leader

Step 1. Upon a leader $L_j \neq L_{hold}$ receives request from process P_i and $L_j = L_{idle}$
1.1 if $P_i \cap C_j = \emptyset$
1.1.1 add P_i to *RaymQ*
1.1.2 send L_j's request to *Dir* (as an external request)
1.2 else
1.2.1 increase corresponding request array element by 1
1.2.2 enqueue *RaymQ* with L_j
1.2.3 send *ExReq* to *Dir*
1.3 end if
1.4 $L_j = L_{waitExToken}$
Step 2. Upon a $L_j = L_{hold}$ receive request from process P_i
2.1 if $P_i \cap C_j = \emptyset$ **then**
2.1.1 increase *j*th element of request array and broadcast request
2.1.2 enqueue P_i to *RaymQ*
2.1.3 $L_j = L_{inTokenSteal}$
2.2 else
2.2.1 increase *i*th element of request array by 1
2.3 end if
Step 3. Upon a $L_j = L_{inTokenSteal}$ receive *InToken* for L_j
3.1 if *RaymQ*.length > 1 or *SuzQ*.length > 1 **then**
3.1.1 update *SuzQ*
3.1.2 dequeue *RaymQ* and set *Dir* with dequeued node
3.1.3 send *ExToken* and *ExReq* to *Dir*
3.1.4 enqueue *RaymQ* with L_j
3.1.5 $L_j = L_{waitExToken}$
3.2 else
3.2.1 update *SuzQ*
3.2.2 dequeue *RaymQ* and set *Dir* with dequeued node
3.2.3 send *ExToken* to *Dir*
3.2.4 $L_j = L_{idle}$
3.3 end if
Step 4. Upon a $L_j = L_{inTokenSteal}$ receive request from P_j
4.1 if $P_i \cap C_j = \emptyset$ **then**
4.1.1 add P_j to *RaymQ*
4.2 else
4.2.1 increase *j*th element of request array
4.3 end if
Step 5. Upon a $L_j = L_{waitExToken}$ and receives request from P_i
5.1 if $P_i \cap C_j = \emptyset$
5.1.1 enqueue P_i to *RaymQ*
5.2 else
5.2.1 increase corresponding request array element by 1
5.2.2 Update *SuzQ*
5.3 end if
Step 6. Upon a $L_j = L_{waitExToken}$ and receives *ExToken* for L_j
6.1 update *SuzQ* and last and add them in *InToken*

Algorithm 1: Raysuz's algorithm for leader node.

of the requester. This phase needs $O(N)$ time. In the last phase, token requester's leader sends token to the requester. This requires $O(N/C)$ time. Consequently, the total messages needed in these three phases are $O(N)$ time which is needed at the worst case.

4.4.2. The Best Case. The best case situation occurs when requester node is the token holder node. In this case, Raymond's, Suzuki-Kasami's, and Raysuz's algorithm will have 0 synchronization delay.

4.4.3. General Case. In Suzuki-Kasami's algorithm, the best case and general case have the same delay. The synchronization delay of Suzuki-Kasami's algorithm is $\log_2(N)T$ in general case for power law graphs. In Raymond's algorithm, synchronization delay is $(LHN + LRN)T = 2\log_2(N)T$ in general case.

In Raysuz's algorithm, the first phase takes $(LHN - LHL)T$ time. The second and the third phases require $(LRL + LHL)T$ and $(LRN - LRL)T$, respectively. The total time is: $(LHN + LRL + LRN)T = 2\log_2(N)T$.

Step 1. Upon $O_j = O_{idle}$ or $O_j = O_{haveRequest}$ and receives request from P_i
1.1 if $P_i \cap C_j = \emptyset$ **then**
1.1.1 enqueue P_i to *RaymQ*
1.1.2 if !*Asked*
1.1.2.1 send own request to *Dir*
1.1.2.2 set *Asked* True
1.1.3 end if
1.2 else
1.2.1 increase corresponding *request* array element by 1
1.3 end if
Step 2. Upon $O_j = O_{idle}$ or $O_j = O_{haveRequest}$ and receives *ExToken*
2.1 set *Asked* False
2.2 dequeue *RaymQ* and set *Dir* with dequeued node
2.3 send *ExToken* to *Dir*
2.4 if *RaymQ*.length ≥ 1 **then**
2.4.1 send *ExReq* to *Dir*
2.4.2 set *Asked* True
2.5 end if
Step 3. Upon $O_j = O_{idle}$ and receives *Pulse*
3.1 increase corresponding *request* array element by 1 and broadcast request
3.2 $O_j = O_{haveRequest}$
Step 4. Upon $O_j = O_{haveRequest}$ receives *InToken* for O_j and *SuzQ*.length = 0
4.1 enter CS
4.2 update *SuzQ*
4.3 send *InToken* to head of *SuzQ*
4.4 set *Dir* as head
Step 5. Upon $O_j = O_{haveRequest}$ receives *InToken* for O_j and *SuzQ*.length > 0
5.1 enter CS
5.2 update *SuzQ*
5.3 $O_j = O_{haveToken}$
Step 6. Upon $O_j = O_{haveToken}$ receives request from P_i
6.1 if $P_i \cap C_j = \emptyset$ **then**
6.1.1 enqueue P_i to *RaymQ*
6.1.2 if !*Asked*
6.1.2.1 send own *request* to *Dir*
6.1.2.2 set *Asked* True
6.1.3 end if
6.2 else
6.2.1 increase corresponding *request* array element by 1
6.2.2 update *SuzQ*
6.2.3 send *InToken* to P_i
6.2.4 $O_j = O_{idle}$
6.3 end if
Step 7. Upon $O_j = O_{haveToken}$ and receives *Pulse*
7.1 enter CS

Algorithm 2: Raysuz's algorithm for ordinary node.

This case is for intercluster token transmission between two leaf clusters. The fact is that the token transmissions are not always from end to end nodes. So, the synchronization delay is less than $O(\log_2(N))$. Furthermore, due to the locality of reference, in high load critical section request, a good number of requests will arise in the clusters, there exist C clusters in the system each having N/C nodes. In these cases, the synchronization delay is as much as the best case which is much less than $O(\log_2(N))$.

4.4.4. Clustering Effect on Raysuz's Algorithm. Raysuz's algorithm takes the advantage of using clusters and applying Suzuki-Kasami's algorithm inside the clusters. In some cases, nodes that are close to the token owner node are more likely to make CS request than further nodes. This locality of reference property leads to have lower synchronization delay using Suzuki-Kasami's algorithm inside clusters.

Assume that the requester and the token holder node are in the same cluster. Then, in the worst case, the token

requester will wait for the next node of its cluster to enter CS. This takes $\log_2(N/C)T$ time.

However, in Raymond's algorithm, the worst case time is $\log_2(N)T$. It can be seen that Raysuz's algorithm has lower synchronization delay when requester is in the same cluster.

4.5. Response Time. At the worst case, the response of time of Raysuz's algorithm is $2\log_2(N/C)T$ for intracluster communications and $O(\log_2(N)T)$ for intercluster communications. The response times of Raymond's and Suzuki's algorithms are $2\log_2(N)T$ at the worst case. As explained in previous section, Raysuz is favorable when requests are mostly made by nodes that are in the same cluster as the token requestor.

4.6. Fairness. A distributed mutex algorithm can be stated as fair if CS requests are always satisfied in the increasing order of their request timestamps. Raymond's algorithm [6] is known to be not fair in certain situations. One of these situations can be stated as follows.

Assume that nodes a and b are in the same cluster and node a is the token holder and cluster leader of node b. Also, nodes c and d are in the same cluster and the leader of these nodes is denoted as node e. This topology can be seen in Figure 6. Considering this topology, suppose that the CS requests are made in d, b, and c chronological order. The leaders of the requesters hear the requests and send requests to the token holding cluster, node a. Node a sends the token to the direction of d and immediately sends a request to grab and send token to node b. Node e which is the leader of node d receives the token and forwards it to d. After node d exits the CS, it immediately sends the token to node c, since they are in the same cluster and Suzuki-Kasami's part of Raysuz is processed. Meanwhile, node e makes Suzuki-Kasami's request, since it has received a request from node a (which belongs to node b). After c exits CS, e takes the token back and sends to a, which will finally send token to b. In this scenario, c enters CS before b while b has sent request earlier.

Our proposed algorithm uses Raymond's algorithm for intercluster communications; therefore, intercluster operations are unfair as in Raymond's algorithm. Intracluster requests are satisfied fairly as in Suzuki-Kasami's algorithm. On the whole, unfairness of Raysuz's algorithm is generally lower than, and in the worst cases the same as the unfairness of Raymond's algorithm.

Kanrar and Chaki [29] had solved the unfairness issue by adding extensions to original Raymond's algorithm. The same approach can be applied to Raysuz's algorithm easily; therefore, the unfairness issue can be solved.

5. Discussion and Conclusion

In this work, we have proposed a new token-based hybrid DMX algorithm. The algorithm works on a clustered graph and executes Suzuki-Kasami's algorithm in the clusters, meanwhile running Raymond's algorithm between cluster's leaders. It stores the CS requests in clusters and serves them whenever token arrives into the cluster.

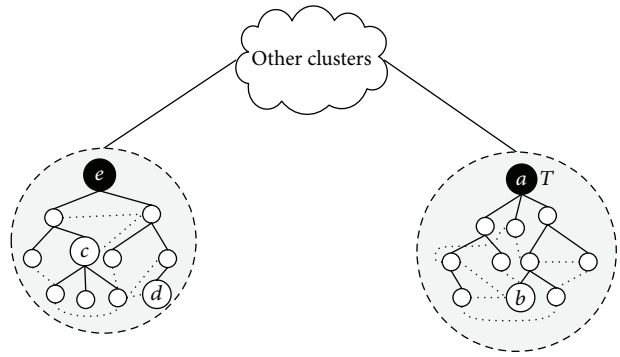

Figure 6: The topology of the scenario.

In general, Raysuz's algorithm has a stronger fairness property than Raymond's algorithm. In the worst case scenario, the algorithm is as fair as Raymond's algorithm. However, there are algorithms to make Raymond's algorithm fair which can also be added into Raysuz's algorithm [29]. Instead, Raysuz algorithm gives better message complexity than pure Suzuki-Kasami's algorithm and better CS delay than pure Raymond's algorithm. In addition, storing CS request and serving them as soon as external token arrives leads to preventing ping-pong style token communication in intraclusters which can be matter of issue in Raymond's algorithm.

Using positive points of both Raymond's and Suzuki-Kasami's algorithms makes it possible to balance the amount of message communication (increasing with bigger clusters) and the synchronization delay (increasing with smaller clusters). Therefore, having a parameter like cluster size as a tuning variable can make desired balance between CS delay and communication messages. Thus, the algorithm will be useful for both interactive systems with multiusers, in which traffic reduction is important, and real-time systems, in which lower CS delay is critical. In the real distributed systems, usually networks are not fully connected. The proposed algorithm uses Suzuki-Kasami's algorithm and customizes it by finding an MST in the clusters and using the shortest paths in them.

The reduction of message count used in the communication leads to have lower energy consumption. This is an important issue for wireless systems in general and makes our algorithm more suitable to be used in MANETs. Also, using clusters and executing CS in the clusters can lead to have less energy consumptions. Group mutual exclusion [30] is another field that can be adapted to proposed algorithm [3, 31]. In any cluster, CS executions can be considered as groups of nodes which tend to enter a CS.

Acknowledgment

The authors would like to thank the anonymous reviewers for their accurate comments on the previous version of the paper. The authors have been able to improve both their work and the paper significantly by taking these anonymous reviewers' critical comments into account.

References

[1] J. Mou, W. Zhou, T. Wang, C. Ji, and D. Tong, "Consensus of the distributed varying scale wireless sensor networks," *Mathematical Problems in Engineering*, vol. 2013, Article ID 862518, 9 pages, 2013.

[2] D. Y. Kim and M. Jeon, "Robust distributed Kalman filter for wireless sensor networks with uncertain communication channels," *Mathematical Problems in Engineering*, vol. 2012, Article ID 238597, 12 pages, 2012.

[3] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[4] 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, 1981.

[5] I. Suzuki and T. Kasami, "A distributed mutual exclusion algorithm," *ACM Transactions on Computer Systems*, vol. 3, no. 4, pp. 344–349, 1985.

[6] K. Raymond, "A tree-based algorithm for distributed mutual exclusion," *ACM Transactions on Computer Systems*, vol. 7, no. 1, pp. 61–77, 1989.

[7] M. Singhal, "A heuristically-aided algorithm for mutual exclusion for distributed systems," *IEEE Transactions on Computers*, vol. 38, no. 5, pp. 70–78, 1989.

[8] M. Maekawa, "A root N algorithm for mutual exclusion in decentralized systems," *ACM Transactions on Computer Systems*, vol. 3, no. 2, pp. 145–159, 1985.

[9] D. Agrawal and A. El Abbadi, "An efficient solution to the distributed mutual exclusion problem," *ACM Transactions on Computer Systems*, vol. 9, no. 1, pp. 1–20, 1991.

[10] S. Lodha and A. Kshemkalyani, "A fair distributed mutual exclusion algorithm," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 6, pp. 537–549, 2000.

[11] O. S. F. Carvalho and G. Roucairol, "On mutual exclusion in computer networks," *Communications of the ACM*, vol. 26, no. 2, pp. 146–147, 1983.

[12] A. Goscinski, "Two algorithms for mutual exclusion in real-time distributed computer systems," *Journal of Parallel and Distributed Computing*, vol. 9, no. 1, pp. 77–82, 1990.

[13] Y. Chang, "Design of mutual exclusion algorithms for real-time distributed systems," *Journal of Inforamtion Science and Engineering*, vol. 11, no. 4, pp. 527–548, 1994.

[14] M. Challenger, P. Bayat, and M. R. Meybodi, "A reliable optimization on distributed mutual exclusion algorithm," in *Proceedings of the 2nd International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TRI-DENTCOM '06)*, pp. 566–574, Barcelona, Spain, 2006.

[15] M. Challenger, V. Khalilpour, P. Bayat, and M. R. Meibodi, "A new robust centralized DMX algorithm," in *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN '07)*, pp. 367–374, Innsbruck, Austria, February 2007.

[16] A. S. Tanenbaum, *Distributed Operating System, Pearson Education*, 2007.

[17] G. Hosseinabadi and N. H. Vaidya, "Exploiting opportunistic overhearing to improve performance of mutual exclusion in wireless Ad Hoc networks," in *Wired/Wireless Internet Communication*, vol. 7277 of *Lecture Notes in Computer Science*, pp. 162–173, 2012.

[18] P. Chaudhuri and T. Edward, "An $O(\sqrt{n})$ distributed mutual exclusion algorithm using queue migration," *Journal of Universal Computer Science*, vol. 12, no. 2, pp. 140–159, 2006.

[19] F. Mueller, "Prioritized token-based mutual exclusion for distributed systems," in *Proceedings of the 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*, pp. 791–795, April 1998.

[20] M. L. Neilsen and M. Mizuno, "A dag-based algorithm for distributed mutual exclusion," in *Proceedings of the 11th International Conference on Distributed Computing Systems*, pp. 354–360, May 1991.

[21] M. Naimi and M. Trehel, "A distributed algorithm for mutual exclusion based on data structures and fault tolerance," in *Proceedings of the 6th International Phoenix IEEE International Conference on Computer Communications*, pp. 35–39, Scottsdale, Ariz, USA, 1987.

[22] 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, 1996.

[23] J. Edmondson, D. Schmidt, and A. Gokhale, "QoS-enabled distributed mutual exclusion in public clouds," in *On the Move to Meaningful Internet Systems: OTM*, vol. 7045 of *Lecture Notes in Computer Science*, pp. 542–559, 2011.

[24] M. Chen, W. Cai, and L. Ma, "Cloud computing platform for an online model library system," *Mathematical Problems in Engineering*, vol. 2013, Article ID 369056, 7 pages, 2013.

[25] G. Wen and Z. Duan, "Dynamics behaviors of weighted local-world evolving networks with extended links," *International Journal of Modern Physics C*, vol. 20, no. 11, pp. 1719–1735, 2009.

[26] K. Erciyes, D. Ozsoyeller, and O. Dagdeviren, "Distributed algorithms to form cluster based spanning trees in wireless sensor networks," in *Proceedings of the 8th International Conference of Computational Science (ICCS '08)*, vol. 5101 of *Lecture Notes in Computer Science*, pp. 519–528, Springer, 2008.

[27] Q. Min and R. Zimmermann, "VCA: an energy-efficient voting-based clustering algorithm for sensor networks," *Journal of Universal Computer Science*, vol. 13, no. 1, pp. 87–109, 2007.

[28] F. Chung and L. Lu, "The average distance in a random graph with given expected degrees," *Internet Mathematics*, vol. 1, no. 1, pp. 91–113, 2003.

[29] S. Kanrar and N. Chaki, "Modified Raymond's algorithm for priority (MRA-P) based mutual exclusion in distributed systems," in *Proceedings of the 3rd International Conference on Distributed Computing and Internet Technology (ICDCIT '06)*, pp. 325–332, 2006.

[30] A. Swaroop and A. K. Singh, "A token-based group mutual exclusion algorithm for cellular wireless networks," in *Proceedings of the Annual IEEE India Conference (IN-DICON '09)*, pp. 1–4, 2009.

[31] A. Housni and M. Trehel, "Distributed mutual exclusion token-permission based by prioritized groups," in *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*, pp. 253–259, 2001.