GROUP *k*-MUTUAL EXCLUSION FOR DISTRIBUTED SYSTEMS

Jehn-Ruey Jiang
Department of Information Management
Hsuan Chuang University
Hsin-Chu, 300, Taiwan
E-mail:jrjiang@hcu.edu.tw

Abstract

In this paper, we propose an algorithm to solve the group *k*-mutual exclusion (G*k*-ME) problem for distributed systems. The G*k*-ME problem is concerned with controlling the concurrent accesses of some resource by at most *k* nodes with the constraint that no two distinct resources can be accessed simultaneously. The proposed algorithm utilizes a token circulation mechanism and does not require the nodes to have identifications. The delay and session switches of the proposed algorithm are $\Omega(n^2)$ and $\Omega(n)$, respectively.

Key Words

mutual exclusion, group mutual exclusion, k-mutual exclusion, distributed systems, tokens, concurrency

1. Introduction

In this paper, we propose an algorithm to solve the group *k*-mutual exclusion (G*k*-ME) problem for distributed systems. A distributed system consists of interconnected, autonomous nodes which communicate with each other by passing messages. The group *k*-mutual exclusion problem is an extension of the group mutual exclusion (GME) problem introduced in [16]. The GME problem deals with both mutual exclusion and concurrency. Consider a distributed system consisting of *n* nodes and *m* shared resources. Nodes requesting to access the same resource may do so concurrently. However, if two nodes request to access different resources, only one node can proceed. In GME problem, the number of nodes that can access the shared resource is not restricted. However, if we restrict that no more than *k* nodes can be allowed to access the shared resource concurrently, we get a brand-new problem, which we name the group *k*-mutual exclusion (G*k*-ME) problem.

Paper [16] gives the following CD-Jukebox scenario as an example of the GME problem: Suppose that data are stored in a CD-Jukebox of a distributed system. When a disc is loaded for accessing, nodes requiring data on the same disc can concurrently access the disc. However, nodes requiring data on a different disc must wait until the current disc is unloaded. If the CD-Jukebox can be accessed by any number of nodes, then how to control the access of the CD-Jukebox becomes the GME problem. However, if the CD-Jukebox is restricted to be accessed by at most *k* nodes at a time (the concurrent accessing capacity of the CD-Jukebox is *k*), then how to control the access of the CD-Jukebox becomes the G*k*-ME problem.

The group *k*-mutual exclusion (G*k*-ME) problem is different from the group mutual *l*-exclusion (GM*l*-E) problem [27, 28]. In the latter, *l* types of resources out of totally *t* types of resources are allowed to be accessed concurrently and each type of resource has an unbounded concurrent accessing capacity. While in the former, only one type of resource out of totally *t* types of resources is allowed to be accessed and each resource has a concurrent accessing capacity of *k*. When the total number of types of shared resource is 1, i.e., *t*=1, the G*k*-ME problem becomes the *k*-mutual exclusion (*k*-ME) problem, which allows at most *k* nodes to access the unique shared resource simultaneously [8]. However, *k*-ME algorithms [2, 3, 4, 5, 9, 11, 13, 18, 21, 23, 25, 26] cannot be easily adapted to be G*k*-ME ones since the former deal with the conflict of more than *k* nodes accessing the unique resource simultaneously while the latter deals with mainly the conflict of the accesses of different-type resources (with the limitation of at most *k* simultaneous accesses for each type of resource). Thus, it is better to search for solutions to the G*k*-ME problem by exploring existent GME algorithms. There are several GME algorithms proposed for different system models. The algorithms in [1, 6, 15, 17, 29] are designed for the distributed message passing model; the ones in [10, 16, 19], for the shared memory model; the one in [7], for the self-stabilizing model; the one in [14], for the ad hoc mobile networks. Since we concentrate on the distributed message passing model only, below we only discuss the algorithms in [6, 15, 17, 29].

In [17], two algorithms based on the famous Ricart and Agrawala's algorithm [24] are proposed. A node should send request messages to and get replies from all of the systems nodes to access a shared resource, say *X*. Each message is attached with a totally-ordered priority which is a pair of sender's id and a sequence number observing Lamport's causality rules [20]. The group mutual exclusion is guaranteed because a node may defer the reply if it is accessing a resource different from *X* or if it is waiting for accessing a resource different from *X* and the priority of its message is higher than the message received. Note that the two algorithms in [17] are similar

except that the second one applies the "*capturing*" concept to allow a high-priority node currently accessing *X* to solicit a low-priority node to access *X* immediately to increase the degree of concurrency of accessing *X*. It is shown that the algorithm with the capturing concept apparently outperforms the one not using the capturing concept in the sense that the former has higher degree of concurrency and less switches of accesses of different resources.

In [29], two algorithms under the unidirectional ring are proposed. Before accessing a resource, say *X*, a node should send a request message along the unidirectional ring and wait for the message to return after circulating around all nodes. Similar to the algorithms in [17], each message is attached with a priority. Since a node may defer forwarding the request message if it is accessing a resource different from *X* or if it is waiting for accessing a resource different from *X* and the priority of its message is higher than the message received, the group mutual exclusion is then guaranteed. The first algorithm in [29] does not use the capturing concept, while the second algorithm does use the capturing concept. Again, the second algorithm is shown to outperform the first one much.

In [15], three Maekawa-type quorum-based algorithms [22] are proposed for solving the GME problem. These algorithms utilize a quorum structure called *surficial quorum system* to reduce the message complexity. The basic concept of the three algorithms is similar to that of the algorithms in [16]. However, a node only sends request messages to a subset (quorum) of the nodes instead of to all of the nodes (note that this is why the message complexity is reduced). Similarly, the capturing concept is again applied to enhance the performance.

Note that all the algorithms just mentioned utilize message priorities which have unbounded components. Consequently, the message sizes of the above-mentioned algorithms are also unbounded. The GME algorithm in [6] uses no message priority and thus its message size is bounded. It does not use node id and assumes an underlying token circulation environment, where the token carries two fields — one field indicating the resource allowed to be accessed currently and the other field indicating the resource allowed to be accessed later. Every node receiving the token may act according to the values of the two fields to solve the GME problem. The algorithm in [6] also applies the capturing concept to enhance the performance.

Like the algorithm in paper [6], the G*k*-ME algorithm proposed in this paper also assumes an underlying token circulation environment and uses no message priority to eliminate the disadvantage of unbounded message size. The proposed algorithm also applies the capturing concept to increase the degree of concurrency and to decrease the switches of accesses of different resources. At this moment, we want to point out that none of the above-mentioned algorithms defines precisely the condition of applying the capturing concept. Since the capturing concept does improve the performance significantly, it is worthwhile to define the condition of applying the capturing concept precisely in this paper. The contribution of this paper is thus threefold: First, a new class of the group *k*-mutual exclusion (G*k*-ME) problem is defined. Second, a property called *capturing-preemption* is proposed to define precisely the condition of applying the capturing concept. Third, an algorithm solving the G*k*-ME problem with the capturing-preemption property is proposed and analyzed.

This paper is organized as follows. In section 2, we elaborate some preliminaries of the GME and the G*k*-ME problems. In section 3, we introduce the proposed algorithm and also prove its correctness. In section 4, we analyze the algorithm in terms of delay and switches of accesses of different resources. And at last, we give a concluding remark in section 5.

2. The Problem

In this section we give formal definitions of the group mutual exclusion (GME) problem and the group *k*-mutual exclusion (G*k*-ME) problem. Consider a distributed system consisting of *n* nodes and *m* shared resources. Nodes are assumed to cycle through a non-critical section (*NCS*), an entry section (*ES*), a wait section (*WS*), a critical section (*CS*), and an exit section (*XS*). A node *i* can access the shared resource only within the critical section. Every time when node *i* wishes to access a shared resource R_i, node *i* moves from the *NCS* to the *ES*, waiting for entering the *CS*. Note that node *i* may or may not enter the *WS*; it depends on whether or not *i* has once observed *k* nodes in the *CS* concurrently. When node *i* has completed the access of the shared resource, it moves from the *CS* to the *XS* and from the *XS* to the *NCS* finally. Please refer to Figure 1 to see the cycle of a node.

```
loop forever
  NCS (non-critical section)
  ES (entry section)
  WS (wait section)
  CS (critical section)
  XS (exit section)
endloop
```

Figure 1. The cycle of a node.

The Group Mutual Exclusion (GME) problem is concerned with how to design an algorithm satisfying the following property:

Mutual Exclusion:
If two distinct nodes, say *i* and *j*, are in the *CS* simultaneously, then $R_i = R_j$.
Bounded Delay:
If a node enters the *ES*, then it eventually enters the *CS*.

Concurrent Entering:
If there are some nodes requesting to access the same resource while no node is requesting for a different resource, then all the requesting nodes can enter the *CS* concurrently.

The Group *k*-Mutual Exclusion (G*k*-ME) problem is similar to the GME problem except that the "Concurrent Entering" property is replaced by the following "*k*-Concurrent Entering" property:

***k*-Concurrent Entering**:
If there are some nodes requesting to access the same resource while no node is requesting for a different resource, then **at most *k*** requesting nodes can enter the *CS* concurrently.

To increase the degree of concurrency and to decrease the switches of accesses of different resources, the algorithm in [17] allow a high-priority node to "capture" a low-priority one to access the resource (in [17], the priority is a pair of the node *id* and a sequence number observing Lamport's causality rule [20]). For example, suppose there are two nodes *i* and *j* that intend to access resource *X*, but a third node *h* intending to access another resource *Y* has obtained a priority in between *i*'s and *j*'s. Then, *i* and *j* cannot access the resource *X* concurrently because the low-priority node *i* must wait for high-priority node *h* to complete the access of *Y*. If node *i* is allowed to capture node *j* to preempt *h* to access the resource *X*, then the degree of concurrency will increase. Surely, a captured node cannot in turn capture other nodes to avoid the possibility of starvation, *i*.e., only the nodes whose requests are issued earlier *j*'s request is issued can capture other nodes to access resource *X*. We formally define the capturing behavior of the algorithm in [17] as the "Capturing Preemption" property below. Note that we say that a node *h* is *preempted* by a node *j* if and only if *h* and *j* request for different resources, and *h*'s request has higher priority than *j*'s, and *j* enters the *CS* earlier than *h*.

Capturing Preemption:
A node *h* requesting for resource *Y* can be preempted by node *j* requesting for accessing resource *X*, $X \neq Y$, if and only if there exists at least one node *i* in the CS, and *i* requests for resource *X*, and *i* (*i*'s request) has higher priority than *h* (*h*'s request).

3. Proposed Algorithm and Correctness

In this section, we propose a distributed algorithm to solve the group *k*-mutual exclusion (G*k*-ME) problem. The algorithm is assumed to execute in a distributed system consisting of *n* nodes and *m* shared resources. Nodes are labeled as 0, 1, …, *n*-1, and resources are labeled as 0, 1, …, *m*-1. Note that the nodes are assumed to have no identification, and the labels 0, 1, …, *n*-1 are used for representation only. An underlying token circulation environment is assumed where node *i* can only send message to node $i \oplus 1$ ($\oplus$ means the modulo *n* addition operation).

The algorithm will use the following variables and messages:

- TARGET: a local variable storing the label of the resource that the node wishes to access. The initial value of TARGET is *Null*.
- STATE: a local variable indicating the state of the node, which may be *NCS*, *ES*, *WS*, *CS* or *XS*. The initial value of STATE is *NCS*.
- PREEMPT: a local variable indicating whether or not the node is preempting another node. The initial value of PREEMPT is *false*.
- TOKEN(*N*, *P*, *WN*, *WP*, *R*, *Q*): a unique message with six fields *N*, *P*, *WN*, *WP*, *R*, and *Q*, where *N*, *P*, *WN*, *WP*, and *N* are non-negative integers, *R* is a variable indicating the resource that is currently being accessed (*R* is *Null* if no resource is being accessed), and *Q* is a queue for storing the labels of requested resources. Note that *Q.head* returns the first element inserted in *Q*, and *Q.dequeue*( ) removes *Q.head* from *Q*, and *Q.enqueue*(*q*) inserts element *q* in the rear of *Q*, and *Q* is denoted as *Null* if it is empty.

The main idea of the algorithm is that the circulating TOKEN(*N*, *P*, *WN*, *WP*, *R*, *Q*) carries six fields of information for synchronizing the entrances of critical sections. The *R* field indicates the resource that can be accessed at present, and the *Q* field is a queue storing all the requested resources that cannot be accessed at present. Note that when a node receives TOKEN with nonempty *Q*, it is assumed to access resource *R* (if possible) while preempting the nodes waiting for accessing resources in *Q*. Thus, a node sets or resets its local variable PREEMPT accordingly. In order to fit the capturing–preemption property for our algorithm, we assume that a node *i* has higher priority than a node *j* if and only if *i* and *j* are in *ES* or *WS* and *i* receives TOKEN earlier than *j*. The two fields *N* and *P* together store the number of nodes that are accessing resource *R*. The field *N* counts the number of nodes that enter the *CS* with no preemption (*N* stands for "no" preemption), while the field *P* counts the number of nodes that enter the *CS* with preemption (*P* stands for "preemption"). The two fields *WN* and *WP* together store the number of nodes that is waiting to access resource *R* when there are already *k* nodes accessing resource *R* (*W* stands for "waiting"). The field *WN* (resp. *WP*) counts the number of waiting nodes that will later access resource *R* without (resp. with) preemption.

Initially, there is a unique TOKEN(0, 0, 0, 0, *Null*, *Null*) message circulating in the system. Once in a while, a node *i* may wish to access a shared resource *X*. Then, node *i* changes its state from *NCS* to *ES*, waiting for entering the *CS* to access the shared resource. When *i* receives TOKEN(0, 0, 0, 0, *Null*, *Null*) (*R*=*Null* means that no node is known to wish to access any resource), node *i* can now enter the *CS* to access the shared resource.

To allow another node to enter the *CS* concurrently, *i* sends out TOKEN(1, 0, 0, 0, *X*, *Null*), *i*.e., a token message with *N*=1, *P*=0, *WN*=0, *WP*=0, *R*=*X* and *Q*=*Null*.

Any node *j* receiving TOKEN(*N*, *P*, *WN*, *WP*, *R*, *Q*) with (*N*+*P*)<*k* and TARGET=*R*=*X*, increments *N* (*j* increments *P* instead of *N* if *Q* is not empty) by 1, enters the critical section and sends out TOKEN message. However, if *j* receives TOKEN message with (*N*+*P*)≥*k* and TARGET=*R*=*X*, it cannot access the resource immediately due to the limitation of the *k*-concurrent entering property. Instead, it enters the *WS* (wait section) and increments *WN* (*j* increments *WP* instead of *WN* if *Q* is not empty) by 1 to reserve the entrance of the *CS*.

Any node *j* receiving TOKEN(*N*, *P*, *WN*, *WP*, *R*, *Q*) with TARGET≠*R* must insert the resource label TARGET into *Q* if TARGET is not in *Q* yet. However, even when TARGET=*R*, *j* must insert TARGET into *Q* if TARGET is not in *Q* and *Q*≠*Null* and (*N*+*WN*)=0. Note that *Q*≠*Null* means that there are nodes requesting resources different from *R* and (*N*+*WN*)=0 means that *j* cannot enter the *CS* by preemption (we will explain this later). In such a case, *j* must wait for the nodes requesting to access the resources in *Q* to complete their accesses.

When a node leaves the critical section, it enter the *XS* (exit section) and waits for receiving TOKEN. When TOKEN(*N*, *P*, *WN*, *WP*, *R*, *Q*) is received, a node decrements *N* (or *P* if PREEMPT is *true*) by 1 and check if *N*+*P*+*WN*+*WP*=0. If so, it means *N*=0 and *P*=0 and *WN*=0 and *WP*=0, which in turn means that all nodes accessing *R* and waiting for accessing *R* have completed their tasks. Thus, the node performs *R*=*Q*.head and *Q*.dequeue() operations and sends out the TOKEN to start the access of the resource *Q*.head.

Below, we present the algorithm in CSP-like commands [12] of the form:

[$G_1 \rightarrow C_1 \,\square\, G_2 \rightarrow C_2 \,\square \ldots \square\, G_t \rightarrow C_t$],

where $C_1,\ldots,C_t$ are guarded commands. The guarded command C_x , $1 \le x \le t$, is executed only if its guard G_x is enabled. A guard is enabled if the Boolean expression (if any) specified in it evaluates to be *true* and the message reception (if any) specified in it has been done. If more than one guarded commands have enabled guards, then one of them is chosen for execution in a non-deterministic way. Note that a pair of brackets with a preceding asterisk, *i*.e., *[…], represents a infinite loop of guarded commands. Also note that each line of the algorithm is labeled by a number, and the start and the end of a loop of commands are labeled with (L#) and (/L#), respectively. The labels are not part of the commands, they are only used for explaining the algorithm.

Algorithm for node *i*

```
(L0) *[
1.0  i wishes to access resource X →
1.1    TARGET:=X;
1.2    STATE:=ES;
2.0  □i wishes to release resource X →
2.1    TARGET:=Null;
2.2    STATE:=XS;
3.0  □receives TOKEN(N, P, WN, WP, R, Q) →
(L1)   [
3.10   (STATE=ES) ∧ (R=Null) →
3.11     R=TARGET;
3.12     N:=1;
3.13     PREEMPT:=false;
3.14     STATE:=CS;
3.15     send TOKEN(N, P, WN, WP, R, Q) to i⊕1;
3.20   □(STATE=ES ) ∧ (R=TARGET) ∧ (Q=Null) ∧ ( (N+P) < k ) →
3.21     N:=N+1;
3.22     PREEMPT:=false;
3.23     STATE:=CS;
3.24     send TOKEN(N, P, WN, WP, R, Q) to i⊕1;
3.30   □(STATE=ES ) ∧ (R=TARGET) ∧ (Q≠Null) ∧ ( (N+WN) > 0 ) ∧
( (N+P) < k ) →
3.31     P:=P+1;
3.32     PREEMPT:=true;
3.33     STATE:=CS;
3.34     send TOKEN(N, P, WN, WP, R, Q) to i⊕1;
3.40   □(STATE=ES ) ∧ (R=TARGET) ∧ (Q=Null) ∧ ( (N+P) ≥ k ) →
3.41     WN:=WN+1;
3.42     PREEMPT:=false;
3.43     STATE:=WS;
3.44     send TOKEN(N, P, WN, WP, R, Q) to i⊕1;
3.50   □(STATE=ES ) ∧ (R=TARGET) ∧ (Q≠Null) ∧ ( (N+WN) > 0 ) ∧
         ( (N+P) ≥ k ) →
3.51     WP:=WP+1;
3.52     PREEMPT:=true;
3.53     STATE:=WS;
3.54     send TOKEN(N, P, WN, WP, R, Q) to i⊕1;
3.60   □(STATE=WS) ∧ ((N+P) < k ) →
3.61     If (PREEMPT=false) Then {WN=WN–1; N=N+1;} Else
{WP=WP–1; P=P+1;}
3.62     STATE:=CS;
3.63     send TOKEN(N, P, WN, WP, R, Q) to i⊕1;
3.70   □(STATE=ES) ∧ (R ≠ TARGET) ∧ (TARGET ∉ Q) →
3.71     Q.enqueue(TARGET);
3.72     send TOKEN(N, P, WN, WP, R, Q) to i⊕1;
3.80   □(STATE=ES ) ∧ (R=TARGET) ∧ (TARGET∉ Q) ∧ (Q≠Null) ∧
         ( (N+WN)=0 ) ) →
3.81     Q.enqueue(TARGET);
3.82     send TOKEN(N, P, WN, WP, R, Q) to i⊕1;
3.90   □(STATE=XS) →
3.91     If (PREEMPT=false) Then N=N–1; Else P=P–1;
3.92     If ((N+P+WN+WP)=0 ) Then {R=Q.head; Q.dequeue();}
3.93     STATE:=NCS;
3.94     send TOKEN(N, P, WN, WP, R, Q) to i⊕1;
3.a0   □otherwise →
3.a1     send TOKEN(N, P, WN, WP, R, Q) to i⊕1;
(/L1)  ]
(/L0) ]
```

Below, we prove the correctness of the proposed algorithm by the following theorems. Note that in the theorem proofs, we use a bracketed number, for example (2.1), to stand for the algorithm code line labeled by the number.

Theorem 1 (Mutual Exclusion). If two distinct nodes, say *i* and *j*, are in the *CS* simultaneously, then $R_i = R_j$, where R_i and R_j are the resources accessed by *i* and *j* respectively.

Proof:

Without loss of generality, we assume that node *i* enters the *CS* prior to node *j*. After node *i* enters the *CS* (or enters the *WS* to reserve to enter the *CS*), we have $R=R_i$ for the unique TOKEN(*N*, *P*, *WN*, *WP*, *R*, *Q*) message by (3.11), (3.20), (3.30), (3.40) and (3.50). When node *j* receives the TOKEN(*N*, *P*, *WN*, *WP*, *R*, *Q*) message and enters the *CS* (or enters the *WS* to reserve to enter the *CS*), we have $R=R_j$ by (3.20), (3.30), (3.40) and (3.50). Thus, we have $R=R_i=R_j$. □

Theorem 2 (Bounded Delay). If a node enters the *ES*, then it eventually enters the *CS*.

Proof:

Suppose that node *i* enters the *ES* and waits to access resource *X*. When *i* receives the unique TOKEN(*N*, *P*, *WN*, *WP*, *R*, *Q*), there are three cases to consider:

Case 1. Node *i* can enter the *CS* immediately if (*R*=*Null*) (3.10) or (*R*=*X*) ∧ (*Q*=*Null*) ∧ ((*N*+*P*) < *k*) (3.20) or (*R*=*X*) ∧ (*Q*≠*Null*) ∧ ((*N*+*WN*) > 0) ∧ ((*N*+*P*) < *k*) (3.30).

Case 2. Node *i* enters the *WS* if (*R*=*X*) ∧ (*Q*=*Null*) ∧ ((*N*+*P*) ≥ *k*) (3.40) or (*R*=*X*) ∧ (*Q*≠*Null*) ∧ ((*N*+*WN*) > 0) ∧ ((*N*+*P*) ≥ *k*) (3.50). In this case, node *i* enters the *CS* when later receiving TOKEN with (*N*+*P*)<*k* (3.60). Note that when a node leaves the *CS*, it is in the *XS*. It then decrements *N* or *P* by 1 when receiving TOKEN, so (*N*+*P*)<*k* is eventually satisfied and node *i* can enter the *CS*.

Case 3. Node *i* inserts *X* into *Q* if (*R*≠X) ∧ (X∉*Q*) (3.70) or (*R*=X) ∧ (X∉*Q*) ∧ (*Q*≠*Null*) ∧ ((*N*+*WN*)=0) (3.80). Without loss of generality, let *Q* be (R_1, R_2, …, *X*, …). When all the nodes complete the access of resource *R* (with or with no preemption), the last node leaving the *XS* will perform *R*=*Q.head* and *Q.dequeue*() operations. Thus, we have $R=R_1$ and $Q=(R_2, \ldots, X, \ldots)$. Eventually *R* will be *X* and either (Case 1) or (Case 2) can be applied. That is, node *i* enters the *CS* eventually. □

Theorem 3 (*k*-Concurrent Entering). If there are some nodes requesting to access the same resource *X* while no node is requesting for a different resource *Y*, then up to *k* requesting nodes can enter the *CS* concurrently.

Proof:

When a node enters the *CS* by receiving TOKEN(*N*, *P*, *WN*, *WP*, *R*, *Q*), we have

Case 1. (*R*=*Null*) (3.10) or

Case 2. ((*N*+*P*)<*k*) (3.20), (3.30) and (3.60).

Since *N*+*P* records the number of nodes in the *CS* and *R*=*Null* implies the algorithm is just starting (with *N*=*P*=0) or (*N*+*P*+*WN*+*WP*)=0 (3.92), the node entering the *CS* obeys (*N*+*P*)≤*k*, which in term suggests the *k*-concurrent entering property. □

Theorem 4 (Capturing Preemption). A node *j* requesting for resource *Y* can be preempted by node *h* requesting for resource *X* if there exists at least one node *i* requesting for resource *X* and *i* has higher priority than *j* and *i* is in the *CS*.

Proof:

A node *h* preempts node *j* when *h* receives TOKEN(*N*, *P*, *WN*, *WP*, *R*, *Q*) with (*h*'s STATE=*ES*) ∧ (*R*=*X*) ∧ (*Q*≠*Null*) ∧ ((*N*+*WN*) > 0) ∧ ((*N*+*P*) < *k*) (3.30) and with (*h*'s STATE=*ES*) ∧ (*R*=*X*) ∧ (*Q*≠*Null*) ∧ ((*N*+*WN*) > 0) ∧ ((*N*+*P*) ≥ *k*) (3.50). In these two cases, *Q*≠*Null* means there are some nodes, say *j* and others, requesting for resources different from *X*, and (*N*+*WN*) > 0 means that there are nodes, say *i* and others, which are now in the *CS* (or will be in the *CS*) to access *R*=*X* and which have higher priority than *j*. Thus, the capturing preemption property is satisfied. □

4. Analysis

In this section, we analyze the algorithm in terms of the delay and session switches. Below is the definitions of the two measures:

Delay: the maximum number of *CS* entries that can occur while a node is waiting to enter the *CS*. *Session switches*: the maximum number of times of changes of accessed resources while a node is waiting to enter the *CS*.

If we assume that a node accesses different resources in two consecutive entries of the *CS*, then the delay and session switches are $\Omega(n^2)$ and $\Omega(n)$, respectively. The worst case occurs when a node *i* waits for other *n*-1 nodes, say *i*Θ1, *i*Θ2, …, *i*Θ(*n*-1), to access *n*-1 different resources (note that Θ stands for modulo *n* subtraction operation). Thus, there are *n*-1 session switches. Moreover, while node *i*Θ2 accesses its requested resource, node *i*Θ1 is captured to access the same resource, and while node *i*Θ3 accesses its requested resource, nodes *i*Θ1 and *i*Θ2 are captured to access the same resource, …., and while node *i*Θ(*n*-1) accesses its requested resource, nodes *i*Θ1, *i*Θ2, …, *i*Θn-2 are captured to access the same resource. Thus, there may be $1+2+\ldots+(n-1)=\Omega(n^2)$ entries of *CS*.

5. Concluding Remarks

In this paper, we have proposed an algorithm to solve the group *k*-mutual exclusion (G*k*-ME) problem for distributed systems. The proposed algorithm utilizes a token circulation mechanism and does not require the nodes to have identifications. The delay of the proposed algorithm is $\Omega(n^2)$ and the session switches complexity is $\Omega(n)$.

Like other token based algorithms, the proposed algorithm lacks the ability of fault-tolerance. Think about that if the token is lost, the algorithm will stop working. So, we are seeking for the possibility of solving the G*k*-ME problems with other mechanisms, such as the self-stabilizing mechanism or the quorum mechanism [15], which will endow the algorithm fault-tolerant ability.

References

[1] J. Beauquier, S. Cantarell, A. Datta and F. Petit, "Group Mutual Exclusion in Tree Networks," *Journal of Information Science And Engineering*, vol 19, 415-432, 2003.

[2] S. Bulgannawar and N. H. Vaidya, "A distributed *k*-mutual exclusion algorithm," in *Proc. of the 15th IEEE International Conference on Distributed Computing Systems*, pp.153-160, 1995.

[3] Ye-In Chang and Bor-Hsu Chen, "A generalized grid quorum strategy for k-mutual exclusion in distributed systems," *Information Processing Letters*, 80:205-212, 2001.

[4] Y.-I. Chang and B.-H. Chen, "An extended binary tree quorum strategy for *k*-mutual exclusion in distributed systems," in *Proc. of the 1997 Pacific Rim International Symposium on Fault-Tolerant Systems(PRFTS '97)*, pp. 110-15, 1997.

[5] Y.-I. Chang and B.-H. Chen, "An extended degree-*k*-tree quorum strategy for *k*-mutual exclusion in distributed systems," in *Proc. 12th International Conference on Information Networking (ICOIN-12)* , pp. 484-7, 1998.

[6] S. Cantarell, A. K. Datta, F. Petit, and V. Villain, "Token based group mutual exclusion for asynchronous rings," in *Proc. of 21th International Conference on Distributed Computing Systems (ICDCS 2001)*, pages 691-694, 2001.

[7] S. Cantarell and F. Petit, "Self-stabilizing group mutual exclusion for asynchronous rings," in *Proc. of 4th International Conference On Principles Of DIstributed Systems (OPODIS 2000)*, pages. 71-90, 2000.

[8] M. Fisher, N. Lynch, J. Burns, and A. Borondin, "Resource allocation with immunity to limited process failure," in *Proc. of 20th IEEE annual symposium on foundations of Computer Science*, pages 234–254, 1979.

[9] S. Fujita, "A quorum based *k*-mutual exclusion by weighted *k*-quorum systems," *Information Processing Letters*, 67(4):191-197, 1998.

[10] V. Hadzilacos, "A note on group mutual exclusion," in *Proc. of the 20th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 2001.

[11] S.-T. Huang, J.-R. Jiang and Y.-C. Kuo, "*k*-Coteries for fault-tolerant *k* entries to a critical section," in *Proc. of the 13th IEEE International Conference on Distributed Computing Systems*, pp.74-81, 1993.

[12] C. A. R. Hoare, "Communicating sequential processes," *CACM*, 21(8):666-677, 1978.

[13] J.-R. Jiang, S.-T. Huang, and Y.-C. Kuo, "Cohorts structures for fault-tolerant *k* entries to a critical section," *IEEE Trans. on Computers*, 48(2):222-228, 1997.

[14] J.-R. Jiang, "A group mutual exclusion algorithm for ad hoc mobile networks," in *Proceedings of the 6th International Conference on Computer Science and Informatics*, pp. 266-270, 2002.

[15] Y.-J. Joung, "Quorum-based algorithms for group mutual exclusion," in *Proc. 15th International Symposium on Distributed Computing (DISC'01)*, Springer Lecture Notes in Computer Science 2180, 2001.

[16] Y.-J. Joung, "Asynchronous group mutual exclusion (extended abstract)," in *Proc. 17th Annual ACM Symposium on Principles of Distributed Computing (PDOC)*, pages 51-60, 1998.

[17] Y.-J. Joung, "The congenial talking philosophers problem in computer networks (extended abstract)," in *Proc. 13th International Symposium on Distributed Computing (DISC'99)*, 1999.

[18] H. Kakugawa, S. Fujita, M. Yamashita and T. Ae, "A distributed *k*-mutual exclusion algorithm using *k*-coterie," *Information Processing Letters*, 49:213-238, 1994.

[19] P. Keane and M. Moir, "A simple local-spin group mutual exclusion algorithm," in *Proc. 18th Annual ACM Symposium on Principles of Distributed Computing (PODC'99)*, pages 23-32. ACM Press, 1999.

[20] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *CACM.*, 21(7):145-159, 1978.

[21] Y. Manabe, and S. Aoyagi, "A distributed *k*-mutual exclusion algorithm using *k*-coterie," *IEICE Technical Report* , pp.93-43, 1993.

[22] M. Maekawa, "A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems," *ACM Trans. Comput. Syst*., 3(2):145-159, 1985.

[23] K. Makki, P. Banta, K. Been, N. Pissinou, and E. Park, "A token based distributed *k* mutual exclusion algorithm," in *Proc. of the IEEE Symposium on Parallel and Distributed Processing*, pp. 408-411, 1992.

[24] G. Ricart and A. K. Agrawala, "An optimal algorithm for mutual exclusion in computer networks," *CACM*., 24(1):9-17, 1981.

[25] K. Raymond, "A distributed algorithm for multiple entries to a critical section," *Inf. Process. Lett*., vol. 30, no. 4, pp. 189-193, 1989.

[26] P. K. Srimani and R. L. N. Reddy, "Another distributed algorithm for multiple entries to a critical section," *Inf. Process. Lett*., vol. 41, no. 1, pp. 51-57, 1992.

[27] K. Vidyasankar, "A highly concurrent group mutual *l*-exclusion algorithm," in *Proc. 21st ACM Symposium on Principles of Distributed Computing (PODC 2002)*, 2002

[28] K. Vidyasankar, "A simple group mutual l-exclusion algorithm," *Inf. Process. Lett*., vol. 85, no. 2, pp. 79-85, 2003.

[29] K.-P. Wu and Y.-J. Joung, "Asynchronous group mutual exclusion in ring networks," in *Proc. 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP '99)*, pp. 539-543, 1999.