A Distributed Group *k*-Exclusion Algorithm using *k*-Write-Read Coteries

Jehn-Ruey Jiang
Department of Computer Science and Information Engineering
National Central University, Chung-Li, 320, Taiwan
E-mail: jrjiang@csie.ncu.edu.tw

Abstract

In this paper, we introduce a novel quorum system, called *k*-write-read coterie to aid the design of the first group *k*-exclusion algorithm for the distributed system. A *k*-write-read coterie is an extension of both the write-read coterie and the *k*-coterie. It is a pair (W, R) of collections of sets (quorums), where W is a *k*-coterie and every quorum Q in W intersects every quorum P in R. We show how to construct *k*-write-read coteries with the help of *torus* structures and propose a distributed group *k*-exclusion algorithm using *k*-write-read coteries. The proposed algorithm has the merits of unbounded degree of concurrency and can accommodate unlimited number of resources. We prove the correctness of the proposed algorithm and analyze it in terms of message complexity and synchronization delay. The proposed algorithm has message complexity of $2|Q|+4$ to $6n+2|Q \cup P|+1$ and synchronization delay of 3 to $|Q \cup P|+3$, where n is the number of system nodes, $Q \in$ W *and* $P \in$ R.

Keywords: *mutual exclusion, group mutual exclusion, k-exclusion, distributed algorithms, concurrency*

1. Introduction

In this paper, we propose a novel quorum system, called *k*-write-read coterie to aid the design of group *k*-exclusion algorithm for distributed systems. A distributed system consists of shared resources and autonomous nodes which are interconnected and can communicate with each other by passing messages. A node may occasionally request to enter the *critical section (CS)* to access one of the resources. According to different accessing criteria, there arise different synchronization problems, such as the mutual exclusion, the *k*-exclusion, the group mutual exclusion and the group *k*-exclusion problems, etc.

The mutual exclusion problem deals with how to control nodes so that only one node is allowed to access a shared resource at a time. For the group mutual exclusion problem, a group of 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. The *k*-exclusion problem is an extension of the mutual exclusion problem. It restricts that at most *k* nodes can access shared resources at a time. The group *k*-exclusion problem is an extension of both the *k*-exclusion and the group mutual exclusion problems. It restricts that there are at most *k* different groups of nodes accessing *k* different resources concurrently. We would like to point out here that we adopt the definition of the group *k*-exclusion proposed in [14] rather than the one proposed in [6].

A quorum system is a collection of sets, called *quorums*, satisfying the *intersection property*. There are various types of quorum systems with different intersection properties, such as the coterie [2], the *k*-coterie [1, 3], the write-read coterie [4] and the *m*-group quorum system [8], etc. They are useful for designing distributed algorithms to solve mutual exclusion related problems. Algorithms using quorum systems are resilient to node failures and/or network partitioning and usually have low message cost. Thus, this paper will concentrate on quorum-based algorithms.

To the best of our knowledge, there exists no quorum-based group *k*-exclusion algorithm for distributed systems. The existent group *k*-exclusion algorithms [14, 15] are designed for the shared memory model. However, there do exist quorum-based *k*-exclusion algorithms and group mutual exclusion algorithms for distributed systems. The quorum-based *k*-exclusion algorithms [5, 9] cannot be easily adapted to be group *k*-exclusion ones. This is because the former deals with the conflict of more than *k* nodes requesting to access shared resources simultaneously while the latter deals with mainly the conflict of requests from different groups of nodes for accessing different types of resources (with the limitation that at most *k* groups of nodes are allowed to proceed). Thus, it is better to search for quorum-based solutions to the group *k*-exclusion problem by exploring existent quorum-based group mutual exclusion algorithms.

There are four quorum-based distributed group mutual exclusion algorithms proposed in the literature [8, 13]. In [8], Joung proposes three quorum-based algorithms using *m*-group quorum systems for solving the group mutual exclusion problem, where an *m*-group quorum system is a list $(C_1,\ldots,C_m)$ of m collections of sets (quorums). The basic idea of the first algorithm is simple: Each resource R_i is associated with a collection C_i of quorums, $1 \le i \le m$. A node u requesting R_i should send request messages to collect permissions (grants) from all members of an arbitrary quorum Q in C_i. The group mutual exclusion is guaranteed because a node is allowed to grant its permission to only one node at a time, and any quorum in C_i intersects any quorum in C_j, $i \ne j$. To avoid deadlock, each message is attached with a totally-ordered priority of Lamport's time stamp [10]. On receiving a request sent by node u for resource R_i, a node v defers to grant the request if v has granted a request sent by some node w for resource R_j, and w's request has higher priority than u's. On the other hand, if the granted request of w has lower priority, node v should send PREEMPT message to node w to get back its permission. On receiving the PREEMPT message from v, node w should immediately send back the granted permission to v if it is not in CS. As shown in [8], the message complexity of the first algorithm is $3|Q|$ to $6|Q|$, where Q is the arbitrarily selected quorum.

Joung's first algorithm in [8] has the drawback that the degree of concurrency is limited. For example, the number of nodes that can access R_i concurrently is limited by the *degree* of C_i (i.e., the number of pairwise disjoint quorums in C_i). To eliminate the drawback, Joung proposes the second algorithm, which is similar to the first algorithm except that it allows a node to grant its permission to more than one node at a time. A node u's request for resource R_i is granted if there is no request conflict, i.e., there is no pending/granted requests for some resource other than R_i. However, if conflicts occur, they should be resolved as follows: A node requesting resource R_j should yield to another node u requesting resource R_i if u's request has priority higher than any pending/granted request for R_j. As shown in [8], the message complexity is $3|Q|$ to $3|Q|+3r|Q|$, where Q is the arbitrarily selected

quorum and r is the number of permissions which can be granted by a node. As suggested in [8], r should be (the number of nodes that may request R_i)/(degree of C_i) to keep message complexity low. However, the limitation of r leads to bounded degree of concurrency.

To achieve unbounded degree of concurrency, Joung proposes the third algorithm. It is similar to the second algorithm except that it requires a node to send requests to members of a selected quorum by a small-ID to large-ID order. To be more precise, a node u requesting resource R_i randomly selects a quorum Q in C_i and only sends a request to the member (say v) of Q with the smallest ID. When node v decides to grants its permission to u, it does not send permission to u directly but instead sends a request on behalf of u to the member of Q with the second small ID. Sending requests to members of Q according small-ID to large-ID order then goes on, and only the member with the largest ID has to send its permission to u. In this manner, the message complexity is reduced to be $2|Q|+1$ and no deadlock will occur because there is no circular waiting. However, the synchronization delay is prolonged, which is the delay from the time a node requests to enter CS until the time it enters CS. This is because a node sends requests to all members of Q in the one-by-one manner. Joung's third algorithm further utilizes the concept of *reference node* to achieve unbounded degree of concurrency. On receiving node u's request for resource R_i, node v grants the request and sets u as the reference node if v has not granted its permission yet. Subsequent requests for R_i are also granted so long as the reference node is still in CS. In addition to unbounded degree of concurrency, the concept of the reference node is also used to minimize the *context switch* complexity. (We said that a context switch occurs when the current entry to CS and the next entry to CS are for different resources.)

Except their individual drawbacks, Joung's three algorithms have a common drawback that the number of shared resources is limited to be m for an m-group quorum system. To eliminate the drawback of limited number of shared resouces, Toyomura et al. propose a group mutual exclusion algorithm using coteries in [13]. Toyomura el al.'s algorithm is similar to Joung's second algorithm except that it uses coteries instead of m-group quorum systems. To access a resource R_i, a node u should select an arbitrary quorum Q in the coterie and send requests to all members of Q. A node v in Q grants the request and sets R_i as the *current resource* if it has not granted its permission to any node yet. A subsequent request for the current resource R_i can be granted immediately if there is no request conflict. The conflict resolution and the deadlock prevention mechanisms are also similar to those of Joung's second algorithm. To render unbounded degree of concurrency, Toyomura et al.'s algorithm does not lay a limitation on the number of permissions that a node can grant. However, this makes the message complexity to be $3|Q|$ to $3|Q|+3n|Q|$, where n is the number of nodes in the system. Furthermore, it does not take the advantage of the reference node concept. Thus, Toyomura et al.'s algorithm might have a high context switch complexity.

In this paper, we propose a distributed group k-exclusion algorithm using *k-write-read coteries* to achieve unbounded degree of concurrency and unlimited number of resources. A k-write-read coterie is an extension of both the write-read coterie and the k-coterie. It is a pair (W, R) of collections of sets (quorums), where W is a k-coterie and every quorum Q in W intersects every quorum P in R. As we will show, we can construct k-write-read coteries with the help of *torus* structures [11]. We will prove the correctness of the proposed algorithm and analyze it in terms of message complexity and synchronization delay.

This rest of the paper is organized as follows. In section 2, we introduce the k-write-read coterie and show how to construct it with the help of torus structures. In section 3, we propose a distributed group k-exclusion algorithm using k-write-read coteries. The correctness proof of the proposed algorithm is given in section 4. In section 5, we analyze the proposed algorithm and compare it with related ones. And at last, we give concluding remarks in section 6.

2. *k*-Write-Read Coteries

A quorum system is a collection of sets (quorums), satisfying the intersection property, which states that any pair of quorums should have a non-empty intersection. There are various types of quorums systems with different types of intersection properties, such as the coterie [2], the k-coterie [1, 3], the write-read coterie [4] and the m-group quorum system [8], etc. They are useful for designing distributed algorithms to solve different synchronization problems. In this section, we propose a novel quorum system, called *k-write-read coterie*, to aid the design of the group k-exclusion algorithm for distributed systems. Below, we first give the formal definition of the k-write-read coterie.

A *k-write-read coterie* (*k-wr coterie*) is a pair (W, R), where W and R are collections of subsets (quorums) of U. A *k-wr coterie* should satisfy the following properties:

Write k-Intersection Property: There are at most k pairwise disjoint quorums in W.

Write Non-intersection Property: For any h ($< k$) pairwise disjoint quorums $Q_1,...,Q_h$ in W, there exists a quorum Q_{h+1} in W such that $Q_1,...,Q_{h+1}$ are pairwise disjoint.

Write-Read Intersection Property: Every quorum Q in W intersects every quorum P in R.

Write Quorum Minimality Property: Any quorum in W is not a super set of another quorum in W.

Read Quorum Minimality Property: Any quorum in R is not a super set of another quorum in R.

By the definition, the reader can check that W is a k-coterie. For example, let W={{1, 2, 3}, {1, 2, 4}, {5, 6, 7}, {5, 6, 8}, {1, 7, 8}, {2, 7, 8}, {3, 4, 5}, {3, 4, 6} } and R={{1, 3, 5, 7}, {1, 3, 5, 8}, {1, 3, 6, 7}, {1, 3, 6, 8}, {1, 4, 5, 7}, {1, 4, 5, 8}, {1, 4, 6, 7}, {1, 4, 6, 8}, {2, 3, 5, 7}, {2, 3, 5, 8}, {2, 3, 6, 7}, {2, 3, 6, 8}, {2, 4, 5, 7}, {2, 4, 5, 8}, {2, 4, 6, 7}, {2, 4, 6, 8}}. We can check that (W, R) is a 2-*wr* coterie because every quorum is minimal, W is a 2-coterie, and any quorum Q in W intersects any quorum P in R.

Below, we show how to construct k-*wr* coteries with the help of torus structures. A *torus* structure is an array of nodes of r rows and c columns [11]. The rows are arranged in a wraparound manner; i.e., a row i is followed by row i+1 for $1 \le i < r$ and row r is followed by row 1. In [11], Land and Mao utilize the torus structure to construct k-torus quorums to constitute k-coteries. A k-torus quorum is defined to be a set containing all the nodes of some row j, plus one node from each of the $\lfloor r/(k+1) \rfloor$ rows following row j. In [11], the collection of k-torus quorums is shown to be a k-coterie. (Actually, the non-intersection property is not proved in [11]. The non-intersection property, however, can easily be proved if we restrict $\lfloor r/(k+1) \rfloor < \lfloor r/k \rfloor$ according to the comments in [16].) The minimal quorum size of a k-torus quorum is approximately $2(\sqrt{n/(k+1)})$, which is obtained by choosing $c=\sqrt{n/(k+1)}$ and $r=\sqrt{n(k+1)}$, where n the number of nodes in the system.

We can easily construct k-wr coteries on the basis of k-torus quorums. If we let W be the collection of k-torus quorums, and let R be the collection of row covers, each of which is a set

containing a node from every row, then the pair (W, R) is a *k-wr* coterie. The write quorum of the *k-wr* coterie (W, R) has the minimal size $2(\sqrt{n/(k+1)})$, and the read quorum of the *k-wr* coterie (W, R) has the size of *r*, where *r* is the number of rows of the torus structure.

3. The Proposed Algorithm

Consider a distributed system consisting of *n* nodes and several 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 *u* can access the shared resource only within the critical section. Every time when node *u* wishes to access a shared resource, node *u* moves from *NCS* to *ES*, waiting for entering *CS*. Node *u* may or may not enter *WS*; it depends on whether or not *u* has observed any existent reference node that is or will be in *CS*. When node *u* has completed the access of the shared resource, it moves from *CS* to *XS* and from *XS* to *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.

Nodes are said to be of the same *group* if they request to access the same resource. The group *k*-exclusion problem is concerned with how to control distributed nodes to satisfy the following properties:

***k*-Exclusion**: At most *k* groups of nodes are allowed to enter CS concurrently.

***k*-Progressing**: If there are less than *k* groups of nodes in CS at a time, then one more group of nodes are allowed to enter CS concurrently.

Concurrent Entering: Nodes of a group are allowed to enter CS concurrently if at least one node of the group is in CS.

Bounded Delay: If a node enters *ES*, then it eventually enters *CS*.

We can check that when *k* is taken as 1, the group *k*-exclusion problem becomes the group mutual exclusion problem (however, there is no *k*-progressing property for the case of *k*=1). Furthermore, if we modify the concurrent entering property to be that only one node of a group is allowed to enter CS at a time, then we got the *k*-exclusion problem, which in turn becomes the mutual exclusion problem when *k* is taken as 1.

Below, we introduce the proposed algorithm, which uses *k*-wr coteries for solving the group k-exclusion problem. The basic idea of the proposed algorithm is as follows: A node should send request messages to collect enough grants of a write quorum to enter CS to access a shared resource. To increase the concurrency, a node is allowed to enter CS immediately if a *reference node* in CS grants it to do so. A node is said to be a reference node if it has got grants from all members of some write quorum. A node is a *follower* if it enters CS with a grant from a reference node. Once a node becomes a follower, it sends release message to all the nodes that it has sent request messages. A reference node *r* registers its ID for every node of a read quorum so that other nodes of the same group can check *r*'s existence when collecting grants from members of a write quorum (recall that a read quorum intersects any write quorum). There may be several reference nodes; one of the reference nodes is chosen as the *leader*. The leader is responsible of initiating and closing the entry of CS for a group of nodes. To close the entry of CS, the leader should clear its registration of being the leader. And after all followers have left CS, the leader should send release messages to all the nodes that it has sent request messages to leave CS; other nodes just send release message to its reference node to leave CS. There are at most *k* leaders at a time since there are at most *k* pairwise disjoint write quorums. Thus, no more than *k* groups of nodes can be in CS concurrently.

To avoid deadlock, the request message send by node *u* is attached with a totally-ordered priority (*t*, *u*), which is a time stamp observing Lamport's causality rules [10]. The priority (*t*, *u*) is said to be higher than priority (*t'*, *u'*) if $t<t'$ or if $t=t'$ and $u<u'$. Each subsequent message related to the request message is also attached with the same priority parameter (*t*, *u*) for the sake of distinguishability. Furthermore, we assume that each node has a unique ID and that each message is not lost and is delivered in FIFO order.

Our algorithm uses the following messages and variables:

- REQUEST(*t*, *u*, *x*): the request message sent by node *u* for entering CS to access resource *x* with priority (*t*, *u*).
- GRANT(*t*, *u*, *x*): the message to reply to REQUEST(*t*, *u*, *x*) message.
- R-GRANT(*t*, *u*, *x*): the message sent by a reference node to allow node *u* to enter CS.
- COMPETE(*t*, *u*, *x*): the message for *u* to compete as the leader for accessing resource *x*.
- SUCCESS(*t*, *u*, *x*): the message to notify that node *u* succeeds to be the leader for accessing resource *x*.
- FAIL(*t*, *u*, *x*): the message to notify that node *u* fails to be the leader for accessing resource *x*.
- NOTICE(*t*, *u*, *x*): the message to notify that node *u* will be a follower of some node.
- FOLLOW(*t*, *u*, *x*): the message to send to *u* to notify that *u* will be a follower of some node.
- RELEASE(*t*, *u*, *x*): the message for *u* to release the grant related to message REQUEST(*t*, *u*, *x*).
- R-RELEASE(*t*, *u*, *x*) : the message for *u* to release the grant sent from the reference node.
- CLEAR(*t*, *u*, *x*): the message to clear the registration of *u* being the leader for accessing resource *x* related to REQUEST(*t*, *u*, *x*).
- PREEMPT(*t*, *u*, *x*): the message for a node to get back its permission related to REQUEST(*t*, *u*, *x*).
- YIELD(*t*, *u*, *x*): the message for *u* to send back the permission related to REQUEST(*t*, *u*, *x*).
- (W, R): a *k*-wr coterie.
- QUEUE: a local variable, which is a priority queue used by a node to store REQUEST messages received.
- GRANTOR: a local variable, which is a set used by a node to store the IDs of the nodes that have replied GRANT messages to the node.
- GRANTEE: a local variable, which is a set used by a node to store the REQUEST messages to which the node has replied GRANT message. Since a node just replies GRANT message to one REQUEST message at a time, GRANTEE is thus either a singleton or an empty set.
- REFERENCE: a local variabale, which is a set used by a node to store the ID of the nodes that have replied R-GRANT message, i.e., a set to store the reference node.

- FOLLWER: a local variable, which is a set used by a node to store the IDs of its followers.
- REGISTRAR: a local variable, which is a set used by a node to store the IDs of the nodes that have registered u to be the leader.
- LEADER.x: a local variable, which is used to register the leader ID for accessing resource x. A node may register leaders for different resources. Thus, there may be several LEADER variables, each is attached with a resource ID for the sake of distinguishability.

Below, we describe in detail the operations of the proposed algorithm.

- When node u requests to access resource x:

Node u first enters ES (the entry section). It then selects an arbitrary write quorum Q from W, concurrently sends REQUEST(t, u, x) message to each member in Q, and waits for the reply. Node u puts the IDs of the nodes that have replied GRANT(t, u. x) messages into GRANTOR. If GRANTOR equals to Q (i.e., all members in Q reply GRANT(t, u, x) messages), then u enters WS (the wait section). If not so, node u enters the *pending period* to wait for receiving enough GRANT messages to enter WS. In the pending period, u puts into GRANTOR the ID of every node sending GRANT(t, u, x), and checks if GRANTOR contains a write quorum in W. If so, then u can enter WS. Before entering WS, node u adjusts GRANTOR to be a quorum in W. To maximize the concurrency, node u sends RELEASE(t, u, x) message to each node v before entering WS, where v is the node which is not in GRANTOR and to which u has sent REQUEST(t, u, x).

If u still cannot enter WS after a specified period of time of pending, then u randomly reselects a quorum Q', $Q' \neq Q$. Node u then sends REQUEST(t, u, x) message to each node v and enters the pending period as mentioned above, where node v is the node that is in Q' but not in GRANTOR.

However, if a node u in ES receives a R-GRANT(t, u, x) message from some node r, then u sends RELEASE(t, u, x) to every node which u has sent REQUEST(t, u, x), sets GRANTOR to be empty, sets REFERENCE=$\{r\}$ and enters CS immediately. Node r is said to be the *reference node* of node u. Afterward, if node u receives any GRANT(t, u, x) message, node u just ignores the message. However, if node u receives R-GRANT(t, u, x) message from some node w, $w \neq r$, then it sends R-RELEASE(t, u, x) to w immediately.

After a node u enters WS, it selects an arbitrary read quorum P from R and sends COMPETE(t, u, x, $Q \cup P$) message to nodes in $Q \cup P$ to compete as the leader of the group of nodes requesting resource x. The sending of COMPETE message is with the small-ID to large-ID order. To be more precise, node u first sends COMPETE(t, u, x, $Q \cup P$, Z) to node f, where Z is a set that is empty initially and f=first($Q \cup P$), the node of the smallest ID in $Q \cup P$. On receiving COMPETE(t, u, x, $Q \cup P$), a node v forwards the message to next(v, $Q \cup P$) and sets LEADER.x (LEADER for resource x) to be $\{u\}$ if LEADER.x is empty. For the case of LEADER.x being empty, node v also sends NOTICE message to every node w, and sends FOLLOW messages to node u for every node w, where w is in QUEUE that requests for resource x and w is not in Z. The NOTICE and the FOLLOW messages are to notify that node w is a follower of node u. After sending the NOTICE and the FOLLOW messages, node v adds every node w into the set Z. Note that if v is the node with the largest ID in $Q \cup P$, v does not forward COMPETE(t, u, x, $Q \cup P$, Z) but instead sends SUCCESS(t, u, x) to notify that u is the leader to open the entry of CS for accessing resource x. On the other hand, if LEADER.x is set to be w when node v receives COMPETE(t, u, x, $Q \cup P$, Z), node v decides that w is preferable to u to be the leader. For this case, node v sends to u message FAIL(t, u, x) to notify u that it fails to be the leader and it will be a follower of node w. Node v then sends FOLLOW(t, u, x) message to w on behalf of node u. On receiving FOLLOW(t, u, x) message, w sends R-GRANT(t, u, x) to node u once it can enter CS or if it has been in CS.

- When u wishes to release resource x:

Node u first leaves CS and enters XS. If node u is the leader, then node u sends CLEAR(t, u, x) message to every node in REGISTRAR and set REGISTRAR to be empty. Afterwards, node u checks if FOLLOWER is empty, which means that node u has received R-RELEASE messages from all members in FOLLOWER. If so, node u sends RELEASE message to every node in GRANTOR, sets GRANTOR to be empty and enters NCS. If node u is not the leader, then it just checks if FOLLOWER is empty. If so, it sends R-RELEASE message to the node in REFERENCE, set REFERENCE to be empty and enters NCS.

- When v receives REQUEST(t, u, x) message from u:

If LEADER.x is set to be w, then v sends message NOTICE(t, u, x) to u to notify that u will be a follower of some node. Node v also sends message FOLLOW(t, u, x) to node w on behalf of u to notify that u is a follower waiting for w's grant to enter CS.

If LEADER.x is not set and GRANTEE is empty, then v replies GRANT(t, u, x) and adds REQUEST(t, u, x) into GRANTEE. If GRANTEE is not empty, then v inserts REQUEST(t, u, x) into QUEUE. Let the sole element of GRANTEE is REQUEST(t', u', x'). If REQUEST(t, u, x) is not at the front of QUEUE, or REQUEST(t, u, x) has lower priority than REQUEST(t', u', x'), then v adds REQUEST(t, u, x) into QUEUE. On the contrary, if REQUEST(t, u, x) is at the front of QUEUE and REQUEST(t, u, x) has higher priority than REQUEST(t', u', x'), then v sends PREEMPT(t', u', x') to u' to get back the permission granted and waits for u' to reply. If u' is still in the entry section, u' will reply YIELD(t', u', x') message when receiving PREEMPT(t', u', x') from v. If u' has already been in the critical section, then u' sends R-GRANT(t', u', x') to u' and put u' in GRANTEE. Note that if v has already sent PREEMPT(t', u', x') message to u' due to some REQUEST message received earlier than REQUEST(t, u, x), then u does not have to send any PREEMPT(t', u', x') message to v.

When node v receives YIELD(t', u', x') message from u', then v exchanges the storing locations of REQUEST(t', u', x') and REQUEST(t, u, x), i.e., moves REQUEST(t, u, x) to GRANTEE, and moves REQUEST(t', u', x') to QUEUE. Afterwards, node v sends GRANT(t, u, x) message to node u.

- When node u receives NOTICE(t, u, x) message from v:

Node u sets GRANTOR to be empty and sends RELEASE(t, u, x) to every node w, where w is a node to which u has ever sent REQUEST(t, u, x) message but has not sent RELEASE(t, u, x) message yet. Then, node u just keep waiting R-GRANT(t, u, x) message to enter CS.

- When node w receives FOLLOW(t, u, x) message from v:

On receiving FOLLOW(t, u, x), node w will add u into FOLLOWER. Node w will sends R-GRANT(t, u, x) to u to enable u to enter CS after w enters CS.

- When node u receives SUCCESS(t, u, x) message from v:

Node u becomes the leader for accessing resource x and it can enter CS immediately. It sets REGISTRAR to be $Q \cup P$, where $Q \cup P$ is the set chosen by node u when sending out COMPETE(t, u, x, $Q \cup P$) to compete as the leader.

● When node *u* receives FAIL(*t*, *u*, *x*) message from *v*:

Node *u* sends RELEASE(*t*, *u*, *x*) to every node *w*, where *w* is a node to which *u* has ever sent REQUEST(*t*, *u*, *x*) message but has not sent RELEASE(*t*, *u*, *x*) message yet. Node *u* also sets GRANTOR to be empty and ignore any subsequent GRANT(*t*, *u*, *x*) message. Node *u* also sends CLEAR(*t*, *u*, *x*) message for every node in $P \cup Q$ whose ID is smaller than *v* to clear the registration of *u* being the leader.

● When node *u* receives R-GRANT(*t*, *u*, *x*) message from *w*:

Node *u* sets REFERENCE to be {*w*} and enters CS.

● When *v* receives RELEASE(*t*, *u*, *x*) message from *u*:

Node *v* removes REQUEST(*t*, *u*, *x*) from GRANTEE or from QUEUE. If QUEUE is not empty (let REQUEST(t', u', x') be at the front of QUEUE), then *v* removes REQUEST(t', u', x') from QUEUE, adds REQUEST(t', u', x') into GRANTEE, and sends GRANT(t', u', x') to u'.

● When *w* receives R-RELEASE(*t*, *u*, *x*) message from *u*:

Node *w* removes *u* from FOLLOWER. If *w* is the leader and *w* is in XS and FOLLOWER is empty, then node *w* sends RELEASE message to every node in GRANTOR, sets GRANTOR to be empty and enters NCS. If node *w* is not the leader, then it just checks if FOLLOWER is empty. If so, it sends R-RELEASE message to the node in REFERENCE, set REFERENCE to be empty and enters NCS.

● When *v* receives CLEAR(*t*, *u*, *x*) message from *u*:

Node *v* just set LEADER.*x* to be empty.

● When *u* receives PREEMPT(*t*, *u*, *x*) message from *v*:

If *u* is not in CS and *v* is in GRANTOR, then *u* removes *v* from GRANTOR, and sends YIELD(*t*, *u*, *x*) message to *v*. On the other hand, if *u* is in CS or in WS, then *u* just does nothing.

4. Correctness

In this section, we show the correctness of the proposed algorithm.

Lemma 1. There is no circular following relationship.

Proof:

Suppose there is a circular following relationship: $u \xrightarrow{a} v \xrightarrow{b} \dots \xrightarrow{c} w \xrightarrow{d} u$, where $u \xrightarrow{a} v$ stands for that *u* is a follower of the node *v*, which is detected when *u* tries to register itself as the leader in node *a* but finds that *v* has registered as the leader at node *a* already. Since the proposed algorithm demands a node to register itself as the leader by the small-ID to large-ID order, we have $a<b<c<d$. By $w \xrightarrow{d} u$, it follows that *u* has registered at node *d*, which means *u* should have registered at node *a*. Contradicts occurs; the lemma thus holds. □

Lemma 2. There is at most one leader for each resource.

Proof:

Suppose that both node *u* and node *v* are elected as the leader for accessing resource *x*. We have that *u* has registered at every node of $Q \cup P$ and *v* has registered at every node of $Q' \cup P'$ for *Q*, $Q' \in \mathsf{W}$ and *P*, $P' \in \mathsf{R}$. Since only a unique node can register as the leader at one node and $(P \cup Q) \cap (P' \cup Q') \neq \varnothing$, we can conclude that contradiction occurs. The lemma thus holds. □

Lemma 3. There is exactly one leader of a group of nodes for each resource.

Proof: This is a direct consequence of Lemma 1 and Lemma 2. □

Theorem 1. (*k*-Exclusion) No more than *k* groups of nodes can enter CS concurrently.

Proof:

In the proposed algorithm, the leader of a group of nodes is responsible of initiating and closing the entry of CS. Below, we show that there are at most *k* leaders to prove the theorem. A node should collect grants from all nodes of a write quorum to be the leader of some group. Since there are at most *k* pairwise disjoint write quorums in a *k*-wr coterie and every node only gives its grant to only one node at a time, there are at most *k* leaders. □

Theorem 2. (*k*-Progressing) If there are less than *k* groups of nodes in CS, then one more group of nodes are allowed to enter CS concurrently.

Proof:

The proposed algorithm demands a non-leader node to send back the grants it has taken from the members of write quorums as soon as it knows that itself cannot be the leader. The theorem holds due to the write quorum non-intersection property of the *k*-wr coterie. □

Theorem 3. The proposed algorithm incurs no deadlock.

Proof:

Suppose deadlock occurs due to the following waiting cycle: $u \xrightarrow{a} v \xrightarrow{b} \dots \xrightarrow{c} w \xrightarrow{d} u$, where $u \xrightarrow{a} v$ stands for that *v* preempts *u*, whose priority is *a*. We can conclude that $a \propto b \propto c \propto d \propto a$, where $a \propto b$ stands for that priority *a* is lower than priority *b*. We have $a \propto a$, which is a contradiction. The theorem thus holds. □

Theorem 4. The proposed algorithm incurs no starvation.

Proof:

Suppose a group *G* of nodes requesting resource *x* starve. Let node *u* be the node of *G* whose request has the smallest time stamp (the highest priority). Without loss of generality, we may assume the following scenario of starving for node *u*: No nodes of *G* can collect enough grants to enter WS after *u* sending REQUEST(*t*, *u*) to write quorum members, say *q* and others, while some node *v* repeatedly gets grants from *q* and others to enter WS and CS for accessing resources other than *x* again and again. The repeated entrances and exits of CS of node *v* demand the following repeated message sending actions: *v* sends REQUEST to *q*, *q* sends GRANT to *v* (or *q* sends FOLLOW to some reference node *w*, and *w* sends R-GRANT to *v*), and *v* sends RELEASE to *q*. Because when each node receives a message, it adjusts its logical clock, the first component of the time stamp, to be one more than the maximum logical clock ever seen, we can conclude that eventually *v*'*s* REQUEST has a time stamp larger than (*t*, *u*). Thus, *q* will not send its grant to *v* because *v*'*s* request has lower priority than *u*'*s*. Contradiction occurs; the theorem thus holds. □

5. Analysis

In this section, we analyze the proposed algorithm in terms of message complexity and synchronization dealy. Below, we first analyze the message complexity. We first consider scenario that there is only one node *u* requesting to access resource *x*. We analyze the best case for such a scenario as follows. When node *u* requests to access resource *x*, it selects an arbitrary write quorum *Q* from W, concurrently sends REQUEST(*t*, *u*, *x*) message to each member in *Q*, and waits for the reply. Node *u* can enter WS after it has received a GRANT message from every member of *Q*. Node *u* then sends a COMPETE message for each member in $Q \cup P$, where *P* is a quorum in R. The member of $Q \cup P$ with the largest ID then sends a SUCCESS message to node *u* to make node *u* the leader to enter CS. Afterwards, node *u* sends CLEAR message for each member of $Q \cup P$ and sends RELEASE message for each member of *Q*. Thus, the total message complexity for node *u* to enter and leave CS is $2|Q|+2|Q \cup P|+1$, where $Q \in \mathsf{W}$ and $P \in \mathsf{R}$.

Below, we analyze the worst case for the scenario where there is only one node *u* requesting for accessing resource *x*. When node

u requests to access resource *x*, it selects an arbitrary write quorum *Q* from W, concurrently sends REQUEST(*t*, *u*, *x*) message to each member in *Q*, and waits for the reply. If a node *u* cannot get GRANT messages from all members of *Q* immediately and if *u* does not receive any NOTICE message, node *u* will continue to select further quorums to resend REQUEST messages to enter WS. This makes the proposed algorithm fault-tolerant because *u* can still gather enough permissions to enter WS (and enter CS later) when some members of *Q* fail. However, resending request messages incurs more messages. In the worst case, the message complexity for a node to enter WS is 6*n*, where *n* is the number of system nodes. The worst case occurs when *u* sends REQUEST messages to each node *v*, *v* sends PREEMPT message to some node *w*, *w* sends YIELD message to *v*, *v* sends GRANT message to *u*, *u* sends RELEASE message to *v*, and at last *v* sends GRANT message to *r*. Note that in the worst case just mentioned, node *u* will send one message to every node (including *u* itself), and each REQUEST message will incur 5 other messages, namely PREEMPT, YIELD, GRANT, RELEASE and GRANT. Thus, the worst case message complexity for a node to enter WS will be 6*n*. When node *u* enters WS, it still needs to send a COMPETE message for every node in $Q \cup P$ to be the leader of accessing resource *x*. The node in $Q \cup P$ of the largest ID will send a SUCCESS message to *u* to announce it to be the leader. Afterwards, the leader *u* can enter CS. Finally, the leader *u* should send a CLEAR message to each member of $Q \cup P$ to leave CS. Thus, a node needs $2|Q \cup P|+1$ message to be the leader after it enters the WS. Therefore, the total message complexity for a node to enter and leave CS if there is only one node accessing resource *x* will be $6n+2|Q \cup P|+1$, where $Q \in$ W, and $P \in$ R.

Below, we analyze for the scenario where there are already some nodes in CS accessing resource *x*. When node *u* requests to access resource *x*, it selects an arbitrary write quorum *Q* from W, concurrently sends REQUEST(*t*, *u*, *x*) message to each member in *Q*, and waits for the reply. If some node *v* in *Q* finds that a reference node *r* is now in CS accessing resource *x*, it will send NOTICE message to node *u*, and send FOLLOW message to node *r*. On receiving the NOTICE message from *v*, node *u* will send RELEASE message to all members of *Q*. On receiving the FOLLOW message, node *r* will send R-GRANT message to allow *u* to enter CS. At last, node *u* sends R-RELEASE message to *r* when it leaves CS. Thus, the message complexity for the scenario just mentioned is thus $2|Q|+4$.

Below, we analyze the synchronization delay, which is the delay from the time a node invokes a request to enter CS until the time it enters CS. The synchronization delay is usually measured by the number of message transmission time. For the scenario where there is only one node requesting resource *x*, the synchronization delay will be at least $|Q \cup P|+3$. This is because a node *u* should send REQUEST message to every node of a write quorum *Q* in W, and waits to receive GRANT message from every node of *Q* to enter WS. After node *u* enters WS, it should send COMPETE message according to the small-ID to large-ID order for every node in $Q \cup P$ to be the leader, where *P* is a read quorum in R. And the node in $Q \cup P$ with the largest ID sends SUCCESS message to node *u* to claim that *u* is the leader. Afterwards, *u* can enter CS finally.

For the scenario where there are already some nodes in CS, the synchronization delay will be at least 3. This is because when a node *u* requests to access resource *x*, it selects an arbitrary write quorum *Q* from W, concurrently sends REQUEST(*t*, *u*, *x*) message to each member in *Q*, and waits for the reply. If some node *v* in *Q* finds that a reference node *r* is now in CS accessing resource *x*, it will send NOTICE message to node *u* and FOLLOW message to node *r*. On receiving the FOLLOW message, node *r* will send R-GRANT message to allow *u* to enter CS. Thus, the synchronization delay is at least 3.

6. Conclusion

In this paper, we have proposed a novel quorum system, called *k*-write-read coterie, to solve the group *k*-exclusion problem for distributed systems. We have also shown how to construct *k*-write-read coteries with the help of *torus* structures and have proposed a distributed group *k*-exclusion algorithm using *k*-write-read coteries. The proposed algorithm has the merits of unbounded degree of concurrency and unlimited number of resources. The proposed algorithm also utilizes the concept of reference node to reduce the context switch complexity. We have proved the correctness of the proposed algorithm and analyze it in terms of message complexity and synchronization delay. For a distributed system of *n* nodes, the proposed algorithm has message complexity of $2|Q|+4$ to $6n+2|Q \cup P|+1$ and synchronization delay of 3 to $|Q \cup P|+3$, where *Q* is a write quorum, and *P*, a read quorum of a *k*-write-read coterie.

References

[1] S. Fujita, M. Yamashita and T. Ae, "Distributed *k*-mutual exclusion problem and *k*-coteries," in *Proc. 2nd Internl. Symp. on Algorithms, Lecture Notes in Computer Science 557*, Springer, Berlin, pp. 22-31, 1991.

[2] H. Garcia-Molina and D. Barbara, "How to assign votes in a distributed system," *JACM.*, vol. 32, no. 4, pp. 841-860, Oct. 1985.

[3] 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.

[4] T. Ibaraki and T. Kameda, "A theory of coteries: mutual exclusion in distributed systems," *IEEE Trans. Parall. and Distrib. Syst.*, vol. 4, no. 7, pp. 779-794, July 1993.

[5] J.-R. Jiang, S.-T. Huang, and Y.-C. Kuo, "Cohorts structures for fault-tolerant *k* entries to a critical section," *IEEE Trans. on Comp.,* vol. 48, no. 2, pp. 222-228, 1997.

[6] J.-R. Jiang, "A group *k*-mutual exclusion algorithm for distributed systems," in *Proc. of the International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, pp. 140-145, 2003.

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

[8] 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.

[9] H. Kakugawa, *S.* Fujita, M. Yamashita, and T. Ae, "A distributed *k*-mutual exclusion algorithm using *k*-coterie," *Inf. Process. Lett.*, vol. 49, pp. 213-238, 1994.

[10] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *CACM.*, vol. 21, no. 7, pp. 145-159, 1978.

[11] S.-D. Lang and L. J. Mao, "A comparison of two torus-based *k*-coteries," in *Proc. of IEEE Int'l Conf. on Parallel and Distributed Systems*, 1998.

[12] M. Maekawa, "A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 2, pp. 145-159, 1985.

[13] M. Toyomura, S. Kamei and H. Kakugawa, "A quorum-based distributed algorithm for group mutual exclusion," in *Proc. of 4th Int'l Conf. on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, 2003.

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

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

[16] S.-M. Yuan and H.-K. Chang, "Comments on 'Availability of *k*-Coterie' ", *IEEE Trans. on Comp.*, vol. 42, no. 12, page 1457, 1994.