International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 9, Number 2, pages 147–160, July 2019

Asynchronous message-passing distributed algorithm for the global critical section problem

Sayaka Kamei
Dept. of Information Engineering, Graduate School of Engineering, Hiroshima University,
1-4-1 Kagamiyama, Higashi Hiroshima Hiroshima, 739-8527 JAPAN

Hirotsugu Kakugawa
Department of Applied Mathematics and Informatics, Ryukoku University,
Seta, Otsu 520-2194, Japan

Received: January 22, 2019
Revised: April 7, 2019
Accepted: June 5, 2019
Communicated by Akihiro Fujiwara

Abstract

This paper considers the global (l, k)-critical section problem which is the problem of controlling a system in such a way that at least l and at most k processes must be in the critical section at any time in the network, while each process alternates between in the critical section and out of the critical section. In this paper, first, a distributed solution for l-mutual inclusion is proposed in the asynchronous message-passing model. The proposed algorithm uses an ordinary quorum system and all processes play the same role, unlike existing algorithms for k-mutual exclusion. After that, using the proposed algorithm for l-mutual inclusion, we propose a distributed solution for the global (l, k)-critical section problem. The proposed approach is a versatile composition of algorithms for l-mutual inclusion and k-mutual exclusion. Its message complexity is typically $O(\sqrt{n})$, where n is the size of the network.

Keywords: distributed algorithm, mutual exclusion, mutual inclusion, process synchronization

1 Introduction

The mutual exclusion problem is a fundamental process synchronization problem in concurrent systems [6],[22],[24]. It is the problem of controlling a system in such a way that no two processes execute their critical sections (CSs) at any time. Various generalized versions of mutual exclusion have been studied extensively, *e.g.*, k-mutual exclusion [12][3][4][1][5][19], mutual inclusion [9], and l-mutual inclusion [10]. The k-mutual exclusion problem refers to the problem of controlling a system in such a way that at most k processes execute their CSs at any time. The mutual inclusion problem is the complement of the mutual exclusion problem. For the latter, at most one process is in the CS, whereas for the former, at least one process is in the CS. Similarly, the l-mutual inclusion problem is the complement of the k-mutual exclusion problem, where at least l processes are in the CSs. These problems were unified into a framework called *the CS problems* in [11].

This paper considers the l-mutual inclusion problem and the global (l, k)-CS problem. Informally, the global (l, k)-CS problem can be defined as follows. In the entire network, the global (l, k)-CS problem has at least l and at most k processes in the CSs where $0 \le l < k \le n$ and n is the network size. The "global" means the problem specification consider the entire network. In l-mutual inclusion (resp. k-mutual exclusion) problem, no execution to enter (resp. exist) the CS breaks safety. However, in the global (l,k)-CS problem, no execution to change process states may be able to guarantee its safety. Thus, it is not trivial to design the algorithms to guarantee the safety and liveness properties for the problem. Additionally, this problem is interesting not only theoretically but also practically. It is a formulation of the dynamic invocation of servers for load balancing. The minimum number of servers which are invoked for quickly responding to requests or for fault tolerance is l. The number of servers is dynamically changed based on system load. The total number of servers is limited to k to control costs. Instead of preparing k servers in advance, it is useful to prepare a large number of servers so that they can be replaced for maintenance.

In this paper, we first propose a distributed algorithm for the l-mutual inclusion problem. To reduce message complexity, the proposed algorithm uses an ordinary quorum system [8] for information exchange between processes. That is, messages for a request are sent to at most $|Q|$ processes, where $|Q|$ is the maximum size of a quorum (subset of processes) in a coterie. Based on the complementary theorem shown in [11] (explained in Section 2), we can derive an algorithm for l-mutual inclusion from existing algorithms for k-mutual exclusion. Existing algorithms require a specialized quorum system for k-mutual exclusion or some processes play special roles. In contrast, the proposed algorithm uses an ordinary quorum system and all processes play the same role. Additionally, using the algorithms for l-mutual inclusion and k-mutual exclusion as gadgets, we propose a distributed algorithm for the global (l,k)-CS problem. Its message complexity is $O(|Q|)$, typically $O(\sqrt{n})$.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 presents several definitions and problem statements. Section 4 describes the proposed global l-mutual inclusion algorithm. Section 5 describes the proposed algorithm for the global (l,k)-CS problem. Section 6 gives the conclusions and suggestions for future work.

2 Related Work

k-mutual exclusion has been extensively studied. Existing approaches can be classified into two categories, namely token-based algorithms and permission-based algorithms.

Token-based algorithms maintain k tokens. Only processes that hold a token can be in the CS. The algorithms proposed in [15], [16], [3], and [5] belong to this category. The message complexities of [15], [16], and [3] are $O(n)$, and that of [5] is $O(\sqrt{n})$. However, the algorithm in [5] needs $\sqrt{n}$ processes in the global group to play special roles.

Permission-based algorithms allow a process to enter the CS if it obtains sufficient permission from other processes. The algorithms proposed in [17], [23], [7], [12], [4], [2], [20] and [21] belong to this category. In [17], [23], [2], [20] and [21], a process contacts all other processes to obtain their permission. Their message complexity is $O(n)$. In [7], [12] and [4], a process contacts a quorum of a k-coterie to obtain permission to enter the CS. The k-coterie is a specialized quorum system for k-mutual exclusion. Their message complexity is $O(|Q|)$, where $|Q|$ is the number of processes in a quorum of a k-coterie.

To the best of our knowledge, there is no algorithm for the l-mutual inclusion problem. However, using the following complementary theorem in [11], an algorithm can be derived from an existing algorithm for the k-mutual exclusion problem.

Theorem 1 (Complementary Theorem) *Let $\mathcal{A}^{\mathcal{G}}{}_{(l,k)}$ be an algorithm for the global (l,k)-CS problem and Co-$\mathcal{A}^{\mathcal{G}}{}_{(l,k)}$ be the complement algorithm of $\mathcal{A}^{\mathcal{G}}{}_{(l,k)}$, which is obtained by swapping the process states, namely* in the CS *and* out of the CS. *Then, Co-$\mathcal{A}^{\mathcal{G}}{}_{(l,k)}$ is an algorithm for the global $(n-k,n-l)$-CS problem.*

By this theorem, an algorithm for $(n-l)$-mutual exclusion can be transformed into an algorithm for l-mutual inclusion. Then, the Exit() (resp. Entry()) method of l-mutual inclusion can be derived from the Entry() (resp. Exit()) method of $(n-l)$-mutual exclusion by swapping the process states. The Exit() (resp. Entry()) method is a method to exit (resp. enter) the CS for each process.

In [13], an algorithm was proposed for the local version of the (l, k)-CS problem. The global CS problem is a special case of the local CS problem when the network topology is complete. Thus, we can apply the algorithm in [13] to the global CS problem. The message complexity of this algorithm for the local CS problem is $O(\Delta)$, where Δ is the maximum degree. Because the maximum degree is $n-1$ for the global CS problem, the message complexity of this algorithm for the global CS problem is $O(n)$. To the best of our knowledge, our algorithm for the global (l, k)-CS problem is the first distributed algorithm for the problem.

3 Preliminary Information

3.1 Computational Model

Let $G = (V, E)$ be a complete graph, where $V = \{P_1, P_2, ..., P_n\}$ is a set of processes and $E \subseteq V \times V$ is a set of bidirectional communication links between a pair of processes. We assume that $(P_i, P_j) \in E$ if and only if $(P_j, P_i) \in E$. Each communication link is first-in first-out. We consider G to be a distributed system. The number of processes in $G = (V, E)$ is denoted by $n(= |V|)$. We assume that the distributed system is asynchronous, *i.e.*, there is no global clock. A message is delivered eventually but there is no upper bound on the delay time and the execution speed of a process may vary.

3.2 Global Critical Section Problem

Below we present the *CS class* which defines a common interface for algorithms that solve a CS problem, including the (l, k)-CS problem, mutual exclusion, mutual inclusion, k-mutual exclusion and l-mutual inclusion.

Definition 1 *A* CS object*, say o, is a distributed object (algorithm) shared by processes for coordination of access to the CS. Each process P_i has a local variable which is a reference to the object. A class of CS objects is called the* CS class*. The CS class has the following member variables and methods.*

- $o.state_i \in \{\mathsf{InCS}, \mathsf{OutCS}\}$ *: the state variable of P_i.*
- o.Exit() *: a method for changing the object state from* InCS *to* OutCS*.*
- o.Entry() *: a method for changing the object state from* OutCS *to* InCS*.*

For any given process P_i, we say that the CS method invocation convention (CSMIC) *for object o at P_i is confirmed if and only if the following two conditions are satisfied at P_i.*

- o.Exit() *is invoked only when $o.state_i = \mathsf{InCS}$ holds.*
- o.Entry() *is invoked only when $o.state_i = \mathsf{OutCS}$ holds.*

We say that CSMIC for object o is confirmed globally if and only if CSMIC for object o at P_i is confirmed for each $P_i \in V$.

For each CS object o, the vector of local states $(o.state_1, o.state_2, \ldots, o.state_n)$ of all processes forms a *configuration* (global state) of a distributed system. For each configuration C for object o, let $\mathcal{CS}_o(C)$ be the set of processes P_i with $o.state_i = \mathsf{InCS}$ in C. Under each object o, the behaviour of each process P_i is as follows, where we assume that when $o.state_i$ is OutCS (resp. InCS), P_i eventually invokes o.Entry() (resp. o.Exit()) and changes its state to InCS (resp. OutCS).

```
/* $o.state_i$ = (Initial state of $P_i$ in the initial configuration) */
while true {
  if ($o.state_i$ = OutCS) {
    $o$.Entry();    /* $o.state_i$ = InCS */
    }
    if ($o.state_i$ = InCS) {
      $o$.Exit();    /* $o.state_i$ = OutCS */
    }
  }
```

Definition 2 (Global CS problem) *Assume that a pair of numbers l and k $(0 \le l < k \le n)$ is given on complete network $G = (V, E)$. For each configuration C for a CS object (l,k)-GCS, let $\mathcal{CS}_{(l,k)\text{-GCS}}(C)$ be the set of processes P_i with the* InCS *state in C. Then, the object (l,k)-GCS solves* the global CS problem *on G if and only if the following two conditions hold in each configuration C.*

- *Safety: $l \le |\mathcal{CS}_{(l,k)\text{-GCS}}(C)| \le k$ at any time.*
- *Liveness: Each process $P_i \in V$ alternates between* OutCS *and* InCS *states infinitely often if it continues to change its state.*

For given l and k, the global CS problem is referred to as *the global (l,k)-CS problem.* Note that $k \neq l$ holds because the safety property is broken if some process exits when k $(= l)$ processes are in the CS.

We assume that for object (l,k)-*GCS* for the global (l,k)-CS problem, the initial configuration C_0 is safe; that is, C_0 satisfies $l \le |\mathcal{CS}_{(l,k)\text{-GCS}}(C_0)| \le k$. Note that existing algorithms for CS problems assume that their initial configurations are safe. For example, for the mutual exclusion problem, most algorithms assume that initially each process is in the OutCS state, and some algorithms (*e.g.*, token-based algorithms) assume that initially exactly one process is in the InCS state and that the other processes are in the OutCS state. Hence, our assumption for the initial configuration is consistent with those of existing algorithms.

3.3 Performance Measures

The typical performance measures applied to algorithms for the CS problem are as follows.

- *Message complexity*: the number of message exchanges triggered by a pair of invocations of Exit() and Entry().
- *Waiting time*[1] *for exit (resp. entry)*: the time between a process making a request (*i.e.*, the invocation of Exit() (resp. Entry())) and it actually exiting (resp. entering) the CS, assuming that each message transmission consumes one time unit and that the local computation time is zero.
- *Maximum waiting time*: the maximum of the waiting times for exit or entry.

3.4 Coterie

To reduce message complexity, the proposed algorithm uses a coterie [8] for information exchange between processes.

Definition 3 (Coterie [8]) *A* coterie *C under a set V is a set of subsets of V, i.e., $C = \{Q_1, Q_2, \ldots, Q_n\}$, where $Q_i \subseteq V$ and satisfies the following conditions.*

1. *Nonemptiness: For each $Q_i \in C$, $Q_i \neq \emptyset$.*
2. *Intersection property: For any distinct $Q_i, Q_j \in C$, $Q_i \cap Q_j \neq \emptyset$ holds.*
3. *Minimality: For any distinct $Q_i, Q_j \in C$, $Q_i \not\subseteq Q_j$ holds.*

Each member $Q_i \in C$ is called a quorum.

[1]The name of this performance measure differs among previous studies, with some (*e.g.*, [22]) referring to this performance measure as the *synchronization delay.*

We assume that for each P_i, Q_i is defined as a constant and is a quorum used by P_i.

Some examples of a coterie are given below.

Example 1 (Majority Coterie [8]) *A majority coterie C_{maj} under a set V is defined as follows:*

- *When n is odd, $|Q_i| = (n+1)/2$.*
- *When n is even, $C_{maj} = C_1 \cup C_2$ satisfying the following:*
  - *In C_1, each $Q_i \in C_1$ holds $|Q_i| = n/2$, and*
  - *In C_2, each $Q_j \in C_2$ holds $|Q_j| = (n/2)+1$ and $Q_i \nsubseteq Q_j$ for any $Q_i \in C_1$.*

Example 2 (Grid Coterie [14]) *A grid coterie C_{grid} is $\{Q_{x,y} : 0 \leq x, y < \sqrt{n}\}$, where $Q_{x,y} = \bigcup_{0\leq i<\sqrt{n}}\{P_{i+y\sqrt{n}}\} \cup \bigcup_{0\leq j<\sqrt{n}}\{P_{x+j\sqrt{n}}\}$. For each $Q_{x,y}$, $|Q_{x,y}| = 2\sqrt{n}-1$ holds.*

By using a coterie, the algorithm proposed in [14] for mutual exclusion achieves a message complexity of $O(|Q|)$, where $|Q|$ is the maximum size of a quorum in a coterie. If a typical coterie with size $\sqrt{n}$ is used in this algorithm, the message complexity will be $O(\sqrt{n})$.

4 Proposed Algorithm for l-Mutual Inclusion

Now, we propose the class *MUTIN*(l) for l-mutual inclusion. It can be used as a gadget in the algorithm (l,k)-*GCS* proposed in Section 5. A formal description of the class *MUTIN*(l) for each process $P_i \in V$ is provided in Algorithms 1 and 2.

First, we present an outline of how each process finds the set of processes in the InCS state in a distributed manner with quorums. When P_i changes its state, it notifies each process in a quorum Q_i of its state change. When P_i wants to find the set of processes in the InCS state, P_i contacts each process in Q_i. For each process $P_k \in V$, because of the intersection property of quorums, there exists at least one process $P_j \in Q_k \cap Q_i \neq \emptyset$. Hence, P_k sends its state to P_j, which then sends the state of P_k to P_i. For this reason, when P_i contacts each process in Q_i, it obtains information about all the processes.

In the proposed algorithm, each P_i maintains a local variable $procsInCS_i$ that keeps track of a set of processes in the InCS state in R_i, where $R_i = \{P_k \mid P_k \in V \wedge P_i \in Q_k\}$ is the set of processes which inform P_i about the process states. The value of $procsInCS_i$ is maintained in the following way.

- When P_i is in the InCS state and wishes to change its state to OutCS in Exit(), it sends an Acquire message to each $P_j \in Q_i$.
- When P_i changes its state to InCS in Entry(), it sends a Release message to each $P_j \in Q_i$.
- When P_i receives a Release message from P_j, it adds P_j to $procsInCS_i$.
- When P_i receives an Acquire message from P_j, it deletes P_j from $procsInCS_i$.

We assume that the initial value of $procsInCS_i$ is the set of processes $P_j \in R_i$ in the InCS state in the initial configuration.

Next, we describe the procedure used to guarantee safety. When P_i changes its state to InCS in Entry(), it immediately sends a Release message to each $P_j \in Q_i$. In Entry(), the number of processes in InCS increases by 1. Thus, safety is trivially maintained.

When P_i wishes to change its state to OutCS in Exit(), safety is maintained in the following way.

- First, P_i sends a Query message to each process $P_j \in Q_i$. Then, each $P_j \in Q_i$ sends a Response1 message with $procsInCS_j$ back to P_i.
- P_i stores $procsInCS_j$ which P_i received from each $P_j \in Q_i$ in the variable $currentInCS_i$. That is, $currentInCS_i = \bigcup_{P_j \in Q_i} procsInCS_j$ holds.

Algorithm 1 Description of class $MUTIN(l)$ for l-mutual inclusion

```
Constants:
  $Q_i$ : set of processIDs;
  $R_i$ : $\{P_k \mid P_k \in V \wedge P_i \in Q_k\}$, set of processIDs;

Local Variables:
  $mx$ : CS object for mutual exclusion;
  $reqCnt_i$ : integer, initially 0;
  $procsInCS_i$ : set of processIDs, initially $\{P_j \in R_i \mid state_j = \mathsf{InCS}\}$ in a safe initial configuration;
  $currentInCS_i$ : set of processIDs, initially $\emptyset$;
  $ackFrom_i$ : set of processIDs, initially $\emptyset$;
  $responseAgainTo_i$ : processID, initially nil;
  $respAgainReqCnt_i$ : integer, initially 0;

Exit():
      /* $state_i = \mathsf{InCS}$ */
  $mx$.Entry();
  $reqCnt_i := reqCnt_i + 1$;
  $currentInCS_i := \emptyset$;
  for-each $P_j \in Q_i$
    send $\langle \mathsf{Query}, reqCnt_i, P_i \rangle$ to $P_j$;
  wait until $(|currentInCS_i| \geq l + 1)$;
  $ackFrom_i := \emptyset$;
  for-each $P_j \in Q_i$
    send $\langle \mathsf{Acquire}, P_i \rangle$ to $P_j$;
  wait until $(ackFrom_i = Q_i)$;
  $mx$.Exit();
      /* $state_i = \mathsf{OutCS}$ */

Entry():
      /* $state_i = \mathsf{InCS}$ */
  for-each $P_j \in Q_i$
    send $\langle \mathsf{Release}, P_i \rangle$ to $P_j$;
```

Algorithm 2 Description of class *MUTIN*(l) for l-mutual inclusion (continued)

```
On receipt of a $\langle \mathsf{Query}, reqCnt, P_j \rangle$ message:
  send $\langle \mathsf{Response1}, procsInCS_i, reqCnt, P_i \rangle$ to $P_j$;
  $responseAgainTo_i := P_j$;
  $respAgainReqCnt_i := reqCnt$;

On receipt of a $\langle \mathsf{Response1}, procsInCS, reqCnt, P_j \rangle$ message:
  if $(reqCnt_i = reqCnt)$ $currentInCS_i := currentInCS_i \cup procsInCS$;

On receipt of a $\langle \mathsf{Acquire}, P_j \rangle$ message:
  $procsInCS_i := procsInCS_i \backslash \{P_j\}$;
  send $\langle \mathsf{Ack}, P_i \rangle$ to $P_j$;
  $responseAgainTo_i :=$ nil;
  $respAgainReqCnt_i := 0$;

On receipt of a $\langle \mathsf{Ack}, P_j \rangle$ message:
  $ackFrom_i := ackFrom_i \cup \{P_j\}$;

On receipt of a $\langle \mathsf{Release}, P_j \rangle$ message:
  $procsInCS_i := procsInCS_i \cup \{P_j\}$;
  if $(responseAgainTo_i \neq$ nil$)$ {
    send $\langle \mathsf{Response2}, procsInCS_i, respAgainReqCnt_i, P_i \rangle$ to $responseAgainTo_i$;
    $responseAgainTo_i :=$ nil;
    $respAgainReqCnt_i := 0$;
  }

On receipt of a $\langle \mathsf{Response2}, procsInCS, reqCnt, P_j \rangle$ message:
  if $(reqCnt_i = reqCnt)$
    $currentInCS_i := currentInCS_i \cup procsInCS$;
```

- If $|currentInCS_i| \geq l + 1$ holds, then at least $l + 1$ processes are in the InCS state. Thus, even if P_i changes its state from InCS to OutCS, at least l processes remain in the InCS state. Thus, safety is maintained. Therefore, only if the condition $|currentInCS_i| \geq l + 1$ is satisfied, P_i sends an Acquire message to each $P_j \in Q_i$, and changes its state to OutCS.

The above procedure guarantees safety if only one process wishes to change its state to OutCS, but does not if more than one process wishes to change its state to OutCS. To avoid the latter situation, we serialize requests that occur concurrently. A serialization technique employed in many distributed mutual exclusion algorithms is to use priority based on the timestamp and preemption mechanism of permissions [18]. We use this technique for serialization; however, to simplify the description of the proposed algorithm, we use an ordinary mutual exclusion algorithm [14] in the proposed algorithm instead of explicitly using the timestamp and preemption mechanism. This is because typical ordinary mutual exclusion algorithms use the same mechanism for serialization, and hence the underlying mechanism is essentially the same. We denote the object for ordinary mutual exclusion by *mx*. When a process wishes to change its state to OutCS, it invokes the *mx*.Entry() method, which allows it to enter the CS with mutual exclusion. After the process changes its state to OutCS, it invokes the *mx*.Exit() method, which allows it to exit the CS with mutual exclusion. Thus, by incorporating the distributed mutual exclusion algorithm *mx*, the state change from InCS to OutCS is serialized between processes. Additionally, before the execution of P_i's *mx*.Exit(), P_i waits to receive Ack messages which are responses from each $P_j \in Q_i$ to an Acquire message sent by P_i. Thus, the update of the variable $procsInCS_j$ is atomic. In this way, it is ensured that each process $P_k \in currentInCS_i$ is in InCS. Thus, $\#L \geq |currentInCS_i|$ is guaranteed, where $\#L$ is the number of processes with *state* = InCS.

Finally, we explain the procedure used to guarantee liveness. We assume that when $state_i$ is OutCS (resp. InCS), P_i eventually invokes Entry() (resp. Exit()) and changes its state to InCS (resp. OutCS). When exactly l processes are in the InCS state, P_i observes this from the Query/Response1 message exchange and is blocked. When process P_k enters the CS, its Release message is sent to each process in Q_k, and some $P_j \in Q_k \cap Q_i$ sends a Response2 message to P_i. Hence, P_i is eventually unblocked. Note that there exists at least one such P_j because of the intersection property of quorums.

Even if there are more than l processes in the InCS state, P_i may observe that the number of processes in the InCS state is l from the Query/Response1 message exchange. When this occurs, P_i is blocked not to violate safety. This case occurs if the Release message from some P_k is in transit to $P_j \in Q_k \cap Q_i$ due to message delay because of asynchrony when P_j handles the Query message from P_i. Even if this case occurs, the Release message from P_k eventually arrives at some $P_j \in Q_k \cap Q_i$. Then, P_j sends a Response2 message to P_i. Hence P_i is eventually unblocked. Because P_i is unblocked by a single Response2 message, it is sufficient for each process to send a Response2 message at most once.

Class $MUTIN(l)$ uses the following local variables for each process $P_i \in V$.

- $reqCnt_i$: **integer**, **initially** 0
  - The request counter of P_i. This value is used by a Response1/Response2 message to distinguish it from the corresponding Query message.
- $procsInCS_i$: **set of processIDs**
  - A set of processes in the InCS state found by P_i.
- $currentInCS_i$: **set of processIDs**
  - A set of processes in the InCS state found by P_i. That is, each process in this set is known to be in the InCS state by some process in quorum Q_i.
- $ackFrom_i$: **set of processIDs**, **initially** $\emptyset$
  - A set of processes from which P_i received an Ack message. An Ack message is an acknowledgment of an Acquire message sent to each $P_j \in Q_i$, where P_i waits until $ackFrom_i = Q_i$ holds. This handshake guarantees $P_i \notin procsInCS_j$ for each $P_j \in Q_i$ before P_i invokes *mx*.Exit().
- $responseAgainTo_i$: **processID**, **initially nil**
  - A process id P_j to which P_i should send a Response2 message when P_j is waiting for $|currentInCS_j|$ to exceed l. This value is set when P_i receives a Query message.
- $respAgainReqCnt_i$: **integer**, **initially** 0
  - The request count value for the Query of the process $responseAgainTo_i$.

4.1 Proof of Correctness of $MUTIN(l)$

In this subsection, we again denote the number of processes with *state* = InCS by $\#L$.

Lemma 1 (Safety) *The number of processes in the InCS state is at least l at any time.*

Proof. First, at each point of the execution, for each P_i, we show that $P_j \in procsInCS_i \Rightarrow state_j =$ InCS.

In the initial configuration, $procsInCS_i$ is the set of processes in R_i in InCS. Thus, $P_j \in procsInCS_i \Rightarrow state_j =$ InCS holds.

Consider the case in which, in a configuration where $P_j \in procsInCS_i \Rightarrow state_j =$ InCS holds, $state_j$ changes from InCS to OutCS. This case occurs only when P_j invokes Exit(). In the Exit()

execution of P_j, because of mx.Enter()/mx.Exit() and waiting to update $procsInCS_i$ by Ack message, P_j is not included in any $procsInCS_i$ when P_j finishes the execution of Exit(). Thus, $P_j \in procsInCS_i \Rightarrow state_j =$ InCS holds.

Now, we show that safety is guaranteed. In the initial configuration, it is clear that safety is guaranteed because $\#L \geq l$. We thus discuss the subsequent execution. In the algorithm, only when $|currentInCS_i| \geq l + 1$ is satisfied does P_i exit from the CS. The value of $currentInCS_i$ is computed based on Response1 and Response2 messages. That is, $currentInCS_i = \bigcup_{P_j \in Q_i} procsInCS_j$ holds. Because of mx in Exit() and because processes other than P_i do not invoke Exit(), $P_j \in currentInCS_i \Rightarrow state_j =$ InCS holds. Thus, $\#L \geq |currentInCS_i|$ holds. Therefore, because $\#L \geq l + 1$, even if P_i changes its state to OutCS, $\#L \geq l$ holds. That is, the lemma holds. □

Lemma 2 (Liveness) *Each process $P_i \in V$ alternates between OutCS and InCS states infinitely often.*

Proof. By contradiction, suppose that some processes do not alternate between the OutCS and InCS states infinitely often. Let P_i be any of these processes. Because Entry() has no blocking operation, we assume that P_i is blocked from executing the Exit() method. There are three possible reasons that P_i is blocked in the Exit() method: (1) P_i is blocked by mx.Entry(), (2) P_i is blocked by the first **wait** statement in the Exit() method, or (3) P_i is blocked by the second **wait** statement in the Exit() method.

No process is blocked forever by case (3) because each $P_j \in Q_i$ immediately sends back an Ack message in response to an Acquire message. Below, we consider cases (1) and (2).

First, we consider the case in which all of the blocked processes are blocked by mx.Entry(), that is, all of the blocked processes are in case (1). However, this situation never occurs because liveness is incorporated into the mutual exclusion algorithm. Thus, at least one process is blocked in case (2).

The number of processes blocked in case (2) is exactly one because no two processes reach the corresponding statement at the same time by mx.Entry().

Additionally, we claim that all of the processes are eventually blocked in case (1), except P_k blocked in case (2). Each non-blocked process in the InCS state eventually calls the Exit() method and is then blocked by mx.Entry() because P_k obtains the mutual exclusion lock. Now the system reaches a configuration in which P_k is blocked in case (2), the remaining $n - 1$ processes are blocked in case (1), and all the processes are in the InCS state.

Finally, we show that P_k is eventually unblocked. Recall that P_k is blocked in case (2), *i.e.*, it is waiting until the condition $|currentInCS_k| \geq l + 1$ becomes true.

The size of a collection $\bigcup_{P_j \in Q_k} procsInCS_j$, each of which is attached to the Response1 message sent from P_j to P_k, is at least l, *i.e.*, $|currentInCS_k| \geq l$ holds, because the atomic update of each $procsInCS_j$, $\#L \geq l$ holds by the safety property, and, for any $P_x \in V$, there exists $P_j \in Q_i$ such that $P_j \in Q_x$ by the intersection property of quorums.

Although it is assumed that $|currentInCS_k| = l$ holds and P_k is blocked, a Release message from some P_y which is not in $currentInCS_k$ eventually arrives at some P_j in Q_k, and P_j sends a Response2 message which includes P_y to P_k. Note that such process P_y exists because $n > l$ is assumed and $Q_y \cap Q_k \neq \emptyset$ holds by the intersection property of quorums. Hence, P_k observes $|currentInCS_k| = l+1$ when it receives the Response2 message and is unblocked. □

Lemma 3 *The message complexity of MUTIN(l) is $O(|Q|)$, where $|Q|$ is the maximum size of the quorums of a coterie used by MUTIN(l).*

Proof. As noted above, we incorporate a distributed mutual exclusion algorithm with a message complexity of $O(|Q|)$, such as that proposed in [14]. Thus, mx requires $O(|Q|)$ messages. In the Exit() method, P_i sends $|Q_i|$ Query messages. For each $P_j \in Q_i$, P_j sends exactly one Response1 message for each Query message: $|Q_i|$ Response1 messages. P_i sends $|Q_i|$ Acquire messages. Then, each $P_j \in Q_i$ sends an Ack message: $|Q_i|$ Ack messages. Hence, $O(|Q|)$ messages are exchanged. In the Entry() method, P_i sends $|Q_i|$ Release messages. For each $P_j \in Q_i$, P_j sends at most one Response2 message

Algorithm 3 (l,k)-*GCS*

```
Local Variables:
  lmin : CS object for l-mutual inclusion;
  kmex : CS object for k-mutual exclusion;

Exit():
      /* $state_i$ = InCS */
  lmin.Exit();  /* Request */
      /* $state_i$ = OutCS */
  kmex.Exit();  /* Release */

Entry():
      /* $state_i$ = OutCS */
  kmex.Entry();  /* Request */
      /* $state_i$ = InCS */
  lmin.Entry();  /* Release */
```

for Query messages: $|Q_i|$ Response2 messages. Therefore, $O(|Q|)$ messages are exchanged. In total, $O(|Q|)$ messages are exchanged. □

Lemma 4 *The waiting time of $MUTIN(l)$ is 7 time units.*

Proof. The waiting time is 3 time units for the mutual exclusion algorithm employed by $MUTIN(l)$, as described by Maekawa [14] (2 for Entry() and 1 for Exit(); see [22].) In Exit(), a chain of messages, i.e., Query, Response1, Acquire, and Ack, is exchanged between P_i and the processes in Q_i. Hence, 4 additional time units are required. In total, the waiting time for exit is 7 time units. In Entry(), a Release message and a Response2 message are exchanged between P_i and the processes in Q_i. The waiting time for entry is 2 time units. Thus, the waiting time is 7 time units. □

By Lemmas 1-4, we derived the following theorem.

Theorem 2 *$MUTIN(l)$ solves the l-mutual inclusion problem with a message complexity of $O(|Q|)$, where $|Q|$ is the maximum size of a quorum of a coterie used by $MUTIN(l)$. The maximum waiting time of $MUTIN(l)$ is 7 time units.* □

5 Proposed Algorithm for the (l,k)-CS Problem

In this section, we propose a distributed algorithm for (l,k)-CS problem based on algorithms for l-mutual inclusion and k-mutual exclusion. Our algorithm (l,k)-*GCS* is a composition of two objects, *lmin* and *kmex*, which are $MUTIN(l)$ and k-mutual exclusion derived from $MUTIN(n-k)$, respectively. The algorithm (l,k)-*GCS* for each process $P_i \in V$ is presented in Algorithm 3. In (l,k)-*GCS*, we note that each process state changes to OutCS (resp. InCS) immediately in (l,k)-*GCS*.Exit() (resp. (l,k)-*GCS*.Entry()), just after the execution of *lmin*.Exit() (resp. *kmex*.Entry()), before the execution of *kmex*.Exit() (resp. *lmin*.Entry()). We assume that for each P_i, $(l,k)\text{-}GCS.state_i = lmin.state_i = kmex.state_i$ holds in the initial configuration.

In (l,k)-*GCS*, safety is maintained by *lmin*.Exit() and *kmex*.Entry() because objects *lmin* and *kmex* guarantee their respective safety properties with these methods. That is, *lmin*.Exit() blocks if l processes are in InCS, and *kmex*.Entry() blocks if k processes are in InCS.

5.1 Proof of Correctness of Algorithm (l,k)-*GCS*

In this subsection, for each P_i, let $\#G_i$ (resp. $\#L_i, \#K_i$) be 1 if $(l,k)\text{-}GCS.state_i =$ InCS (resp. $lmin.state_i =$ InCS, $kmex.state_i =$ InCS) holds; otherwise, let it be 0. Additionally, let $\#G$ (resp.

$\#L, \#K$) be $\sum_{P_i} \#G_i$ (resp. $\sum_{P_i} \#L_i, \sum_{P_i} \#K_i$). That is, $\#G = |\mathcal{CS}_{(l,k)\text{-GCS}}(C)|$ (resp. $\#L = |\mathcal{CS}_{lmin}(C)|, \#K = |\mathcal{CS}_{kmex}(C)|$) in a configuration C. Then, $l \le \#L \le n$ holds by the safety of *lmin*, and $0 \le \#K \le k$ holds by the safety of *kmex*. Because we assume that the initial configuration C_0 is safe, *i.e.*, $l \le \#G \le k$ holds in C_0.

Lemma 5 *In the initial configuration C_0, lmin and kmex satisfy their respective safety properties.*

Proof. In C_0, because (l,k)-*GCS.state*$_i$ = *lmin.state*$_i$ = *kmex.state*$_i$ holds for each P_i, $\#G_i = \#L_i = \#K_i$ holds. Hence, $\sum_{P_i} \#G_i = \sum_{P_i} \#L_i = \sum_{P_i} \#K_i$ holds. Thus, $\#G = \#L = \#K$ holds. Because $l \le \#G \le k$ holds in C_0, $l \le \#L \le k$ and $l \le \#K \le k$ hold in C_0. Thus, *lmin* and *kmex* satisfy their safety properties in C_0. □

Lemma 6 *In any execution of (l,k)-GCS, CSMIC for lmin and kmex is confirmed globally.*

Proof. Let P_i be any process. Because (l,k)-*GCS.state*$_i$ alternates by invocations of (l,k)-*GCS*.Exit() and (l,k)-*GCS*.Entry(), CSMIC for (l,k)-*GCS* is confirmed at P_i. We show that CSMIC for *lmin* and *kmex* is also confirmed at P_i. Below, we show only the case for *lmin*; we omit the case for *kmex* because it can be shown similarly.

First, we show that an invariant (l,k)-*GCS.state*$_i$ = *lmin.state*$_i$ holds whenever (l,k)-*GCS*.Exit() and (l,k)-*GCS*.Entry() have been just invoked. In C_0, it is assumed that (l,k)-*GCS.state*$_i$ = *lmin.state*$_i$ holds. Hence the invariant holds. We assume that (l,k)-*GCS.state*$_i$ = *lmin.state*$_i$ holds when (l,k)-*GCS*.Exit() and (l,k)-*GCS*.Entry() are invoked.

- When (l,k)-*GCS*.Exit() is invoked, we have (l,k)-*GCS.state*$_i$ = *lmin.state*$_i$ = InCS at the beginning of invocation. Then, P_i invokes *lmin*.Exit() with *lmin.state*$_i$ = InCS. When this invocation finishes, we have (l,k)-*GCS.state*$_i$ = *lmin.state*$_i$ = OutCS.

- When (l,k)-*GCS*.Entry() is invoked, we have (l,k)-*GCS.state*$_i$ = *lmin.state*$_i$ = OutCS at the beginning of invocation. Then, P_i invokes *lmin*.Entry() with *lmin.state*$_i$ = OutCS. When this invocation finishes, we have (l,k)-*GCS.state*$_i$ = *lmin.state*$_i$ = InCS.

Hence, any invocation of (l,k)-*GCS*.Exit() and (l,k)-*GCS*.Entry() maintains the invariant.

Now, we show that CSMIC for *lmin* is confirmed at P_i. Because CSMIC for (l,k)-*GCS* is confirmed at P_i, (l,k)-*GCS*.Exit() is invoked only when (l,k)-*GCS.state*$_i$ = InCS holds, and (l,k)-*GCS*.Entry() is invoked only when (l,k)-*GCS.state*$_i$ = OutCS holds. Because of the invariant, *lmin*.Exit() is invoked only when *lmin.state*$_i$ = InCS holds, and *lmin*.Entry() is invoked only when *lmin.state*$_i$ = OutCS holds. Hence, CSMIC for *lmin* is confirmed at P_i.

Because CSMIC for *lmin* is confirmed at P_i for each P_i, CSMIC for *lmin* is confirmed globally.□

Lemma 7 *In any execution of (l,k)-GCS, lmin and kmex satisfy their safety and liveness properties.*

Proof. By Lemma 5, in C_0, *lmin* and *kmex* satisfy their respective safety properties because (l,k)-*GCS.state*$_i$ = *lmin.state*$_i$ = *kmex.state*$_i$ holds for each P_i. By Lemma 6, CSMIC for *lmin* and *kmex* is confirmed globally. Because CSMIC is the precondition for the safety and liveness of *lmin* and *kmex*, the lemma holds. □

Lemma 8 (Safety) *The number of processes in the* InCS *state is at least l and at most k at any time.*

Proof. By the definition of (l,k)-*GCS*, CSMIC for (l,k)-*GCS* is confirmed globally and (l,k)-GCS.*state*$_i$ = *lmin.state*$_i$ = *kmex.state*$_i$ in C_0. We have $\#G_i = \#L_i = \#K_i$ in C_0. Thus, the values of $\#G_i$, $\#L_i$ and $\#K_i$ at each point of the execution of P_i are as follows.

```
(l,k)-GCS.Exit():
            // $(\#G_i, \#L_i, \#K_i) = (1, 1, 1)$
    lmin.Exit();
            // $(\#G_i, \#L_i, \#K_i) = (0, 0, 1)$
    kmex.Exit();
            // $(\#G_i, \#L_i, \#K_i) = (0, 0, 0)$

$(l, k)$-GCS.Entry():
            // $(\#G_i, \#L_i, \#K_i) = (0, 0, 0)$
    kmex.Entry();
            // $(\#G_i, \#L_i, \#K_i) = (1, 0, 1)$
    lmin.Entry();
            // $(\#G_i, \#L_i, \#K_i) = (1, 1, 1)$
```

Therefore, the invariant $\#G_i \geq \#L_i \wedge \#G_i \leq \#K_i$ is satisfied.

Because $\#G = \sum_{P_i} \#G_i \geq \sum_{P_i} \#L_i = \#L$ and $\#G = \sum_{P_i} \#G_i \leq \sum_{P_i} \#K_i = \#K$ hold, we have invariants $\#G \geq \#L$ and $\#G \leq \#K$. Because $\#G \geq \#L \geq l$ and $\#G \leq \#K \leq k$ hold by the safety of *lmin* and *kmex*, $l \leq \#G \leq k$ holds. □

Lemma 9 (Liveness) *Each process $P_i \in V$ alternates between states infinitely often.*

Proof. By contradiction, suppose that some processes do not alternate between OutCS and InCS states infinitely often. Let X be the set of such processes. In the *kmex*.Exit() (resp. *lmin*.Entry()) method, because P_i just releases the right to be in InCS (resp. OutCS), the method does not block any process P_i forever. Thus, in (l, k)-*GCS*, P_i is blocked only in *lmin*.Exit() of (l, k)-*GCS*.Exit() and *kmex*.Entry() of (l, k)-*GCS*.Entry().

Consider the case in which a process $P_i \in X$ is blocked in (l, k)-*GCS*.Exit() forever. Note that we omit the proof of the case in which P_i is blocked in (l, k)-*GCS*.Entry() forever because it is symmetric to the following proof.

If other processes invoke (l, k)-*GCS*.Exit() and (l, k)-*GCS*.Entry() alternately and complete their execution of these methods infinitely often, they complete the execution of *lmin*.Exit() and *lmin*.Entry() infinitely often. However, because *lmin* satisfies its liveness property, P_i is not blocked forever. Therefore, for the assumption, not only P_i but also all other processes must be blocked in (l, k)-*GCS*.Exit() or (l, k)-*GCS*.Entry() forever. That is, $X = V$ and all processes are blocked in *lmin*.Exit() or *kmex*.Entry() forever.

Recall that it is assumed that $l \leq \#L \leq n$ holds by the safety of *lmin*, and $0 \leq \#K \leq k$ holds by the safety of *kmex*. By Lemma 8, $l \leq \#G \leq k$ holds. If a process P_j is blocked in *lmin*.Exit(), (l, k)-$GCS.state_j = lmin.state_j = kmex.state_j =$ InCS holds, and if P_j is blocked in *kmex*.Entry(), (l, k)-$GCS.state_j = lmin.state_j = kmex.state_j =$ OutCS holds. Therefore, $\#G = \#L = \#K$ holds.

- Consider the case in which all processes are blocked in *lmin*.Exit(). Then, $\#L = n$ holds. However, by the assumption that *lmin* satisfies its safety property, $l = n$ holds. This is a contradiction because $l < k \leq n$ must hold by assumption.

- Consider the case in which there exists a process which is blocked in *kmex*.Entry(). By the assumption that *lmin* satisfies its safety property, $\#L \geq l$ holds.

  – Consider the case in which $\#L = l$ holds. Because it is assumed that $l < k$ holds, $\#L < k$ holds, that is, $\#L = \#K < k$ holds. Because *kmex* satisfies its liveness property, a process which is blocked in *kmex*.Entry() is eventually unblocked. This is a contradiction by the assumption that all processes are blocked forever.

  – Consider the case in which $\#L > l$ holds. Because *lmin* satisfies its liveness property, a process which is blocked in *lmin*.Exit() is eventually unblocked. This contradicts the assumption that all processes are blocked forever. □

By Lemmas 8 and 9, we derived the following theorem.

Theorem 3 *(l, k)-GCS solves the global (l, k)-CS problem.* □

Finally, we discuss the case in which, by the complementary theorem (Theorem 1), in (l,k)-*GCS*, we use the proposed class as $MUTIN(l)$ to obtain object *lmin* and as $MUTIN(n-k)$ to obtain object *kmex*.

Then, by the proof of Lemma 3, the message complexity is $O(|Q|)$. Additionally, by the proof of Lemma 4, the waiting times for both the exit and entry of (l,k)-*GCS* are 9 time units. Thus, by Theorem 3, we derive the following theorem.

Theorem 4 *(l,k)-GCS solves the global (l,k)-CS problem with a message complexity of $O(|Q|)$, where $|Q|$ is the maximum size of a quorum of a coterie used by (l,k)-GCS. The maximum waiting time of (l,k)-GCS is 9 time units.* □

6 Conclusion

In this paper, we considered the global CS problem in asynchronous message passing distributed systems. The proposed algorithm uses an ordinary quorum system. Its message complexity is $O(|Q|)$, and typically $O(\sqrt{n})$. Because this problem is relevant for the fault tolerance and load balancing of distributed systems, we can consider various future applications.

In the future, we plan to perform extensive simulations and confirm the performance of our algorithms under various application scenarios. Additionally, we plan to design a fault tolerant algorithm for the problem.

Acknowledgement

This research was supported by the Japan Science and Technology Agency (JST) SICORP, KAKENHI No. 19K11828 and No. 19K11826.

References

[1] Uri Abraham, Shlomi Dolev, Ted Herman, and Irit Koll. Self-stabilizing l-exclusion. *Theoretical Computer Science*, 266(1-2):653–692, 2001.

[2] Mathieu Bouillageut, Luciana Arantes, and Pierre Sens. Fault tolerant k-mutual exclusion algorithm using failure detector. In *Proceedings of the International Symposium on Parallel and Distributed Computing*, pages 343–350, 2008.

[3] Shailaja Bulgannawar and Nitin H. Vaidya. A distributed k-mutual exclusion algorithm. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 153–160, 1995.

[4] Ye-In Chang and Bor-Hsu Chen. A generalized grid quorum strategy for k-mutual exclusion in distributed systems. *Information Processing Letters*, 80(4):205–212, 2001.

[5] Pranay Chaudhuri and Thomas Edward. An algorithm for k-mutual exclusion in decentralized systems. *Computer Communications*, 31(14):3223–3235, 2008.

[6] Edgar W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[7] Satoshi Fujita, Masafumi Yamashita, and Tadashi Ae. Distributed k-mutual exclusion problem and k-coteries. In *Proceedings of the 2nd International Symposium on Algorithms*, pages 22–31, 1991.

[8] Hector Garcia-Molina and Daniel Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, October 1985.

[9] Rob R. Hoogerwoord. An implementation of mutual inclusion. *Information Processing Letters*, 23(2):77–80, 1986.

[10] Hirotsugu Kakugawa. Mutual inclusion in asynchronous message-passing distributed systems. *Journal of Parallel Distributed Computing*, 77:95–104, 2015.

[11] Hirotsugu Kakugawa. On the family of critical section problems. *Information Processing Letters*, 115:28–32, 2015.

[12] Hirotsugu Kakugawa, Satoshi Fujita, Masafumi Yamashita, and Tadashi Ae. Availability of k-coterie. *IEEE Transaction on Computers*, 42(5):553–558, 1993.

[13] Sayaka Kamei and Hirotsugu Kakugawa. An asynchronous message-passing distributed algorithm for the generalized local critical section problem. *Algorithms*, 10(38), 2017.

[14] Mamoru Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transaction on Computer Systems*, 3(2):145–159, 1985.

[15] Kia Makki, P. Banta, K. Been, N. Pissinou, and E. K. Park. A token based distributed k mutual exclusion algorithm. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, pages 408–411, 1992.

[16] Kia Makki, Niki Pissinou, and E. K. Park. An efficient solution to the critical section problem. In *Proceedings of the International Conference on Parallel Processing*, volume 2, pages 77–80, 1994.

[17] Kerry Raymond. A distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, 30:189–193, February 1989.

[18] Michel Raynal. *Algorithms for Mutual Exclusion.* North Oxford Academic, 1986. (Translated by D. Beeson).

[19] Vijay Anand Reddy, Prateek Mittal, and Indranil Gupta. Fair k mutual exclusion algorithm for peer to peer systems. In *Proceedings of the 28th International Conference on Distributed Computing Systems*, 2008.

[20] Luiz A. Rodrigues, Jaime Cohen, Luciana Arantes, and Elias P. Duarte. A robust permission-based hierarchical distributed k-mutual exclusion algorithm. In *Proceedings of the12th International Symposium on Parallel and Distributed Computing*, pages 151–158, 2013.

[21] Luiz A. Rodrigues, Elias P. Duarte Jr., and Luciana Arantes. A distributed k-mutual exclusion algorithm based on autonomic spanning trees. *Journal of Parallel and Distributed Computing*, 115:41–55, 2018.

[22] Papa C. Saxena and Jagmohan Rai. A survey of permission-based distributed mutual exclusion algorithms. *Computer Standards & Interfaces*, 25(2):159–181, 2003.

[23] Pradip K. Srimani and Rachamallu L.N. Reddy. Another distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, 41(1):51–57, 1992.

[24] Nisha Yadav, Sudha Yadav, and Sonam Mandiratta. A review of various mutual exclusion algorithms in distributed environment. *International Journal of Computer Applications*, 129(14), 2015.