An asynchronous message-passing distributed algorithm for the global critical section problem*

Sayaka Kamei† Hirotsugu Kakugawa‡

Abstract

This paper considers the global (l, k)-CS problem which is the problem of controlling the system in such a way that, at least l and at most k processes must be in the CS at a time in the network. In this paper, a distributed solution is proposed in the asynchronous message-passing model. Our solution is a versatile composition method of algorithms for l-mutual inclusion and k-mutual exclusion. Its message complexity is $O(|Q|)$, where $|Q|$ is the maximum size for the quorum of a coterie used by the algorithm, which is typically $|Q| = \sqrt{n}$.

1 Introduction

The mutual exclusion problem is a fundamental process synchronization problem in concurrent systems [1],[2],[3]. It is the problem of controlling the system in such a way that no two processes execute their critical sections (abbreviated to CSs) at a time. Various generalized versions of mutual exclusion have been studied extensively, *e.g.*, k-mutual exclusion, mutual inclusion, l-mutual inclusion. They are unified to a framework *the critical section problem* in [4].

This paper discusses the global (l, k)-CS problem 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 $0 \leq l < k \leq n$. 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 always invoked for quick response to requests or for fault-tolerance is l. The number of servers is dynamically changed by system load. However, the total number of servers is limited by k to control costs.

This paper is organized as follows. Section 2 reviews related works. Section 3 provides several definitions and problem statements. Section 4 provides the first solution to the global (l, k)-CS problem. This solution uses a solution for

*This is a modified version of the conference paper in PDAA2017.

†Dept. of Information Engineering, Graduate School of Engineering, Hiroshima University, s-kamei@se.hiroshima-u.ac.jp

‡Dept. of Computer Science, Graduate School of Information Science and Technology, Osaka University, kakugawa@ist.osaka-u.ac.jp

the global l-mutual inclusion provided in Section 5 as a gadget. In section 6, we discuss our concrete algorithm for the global (l, k)-CS problem. In section 7, we give a conclusion and discuss future works.

2 Related Work

The k-mutual exclusion problem is controlling the system in such a way that at most k processes can execute their CSs at a time. The k-mutual exclusion has been studied actively, and algorithms for this problem are proposed in, for example, [5],[6],[7],[8],[9],[10]. However, most of them use a specialized quorum system for k-mutual exclusion, like k-coterie.

The mutual inclusion problem is the complement of the mutual exclusion problem; unlike mutual exclusion, where at most one process is in the CS, mutual inclusion places at least one process in the CS. Algorithms for this problem are proposed in [11] and [12].

The l-mutual inclusion problem is the complement of the k-mutual exclusion problem; l-mutual inclusion places at least l processes in the CSs. For this problem, to the best of our knowledge, there is no algorithm. However, the following complementary theorem is shown in [4].

Theorem 1 (Complementary Theorem) *Let ${\mathcal{A}^{\mathcal{G}}}_{(l,k)}$ be an algorithm for the global (l, k)-CS problem, Co-${\mathcal{A}^{\mathcal{G}}}_{(l,k)}$ be a complement algorithm of ${\mathcal{A}^{\mathcal{G}}}_{(l,k)}$, which is obtained by swapping the process states,* 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, if we have an algorithm for $(n - l)$-mutual exclusion, then we can transform it to an algorithm for l-mutual inclusion. Then, Exit() (*resp.* Entry()) method of l-mutual inclusion can make from Entry() (*resp.* Exit()) method of $(n - l)$-mutual exclusion by swapping the process states.

In [13], an algorithm is proposed for the local version of (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 use the algorithm in [13] as the algorithm for the global CS problem. However, the message complexity of [13] is $O(\Delta)$, where Δ is the maximum degree, as the algorithm for the local CS problem. That is, because the maximum degree is n for the global CS problem, the message complexity of [13] is $O(n)$ as the algorithm for the global CS problem.

3 Preliminary

Let $G = (V, E)$ be a 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 FIFO. We consider that G is 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 running speed of a process may vary.

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

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

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

Each critical section object guarantees safety and liveness for accessing the critical section if critical section method invocation convention (CSMIC), which is defined below, for object o is confirmed globally.

Definition 2 *For any given process* P_i*, we say that* critical section 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.*

Definition 3 *We say that* critical section method invocation convention (CSMIC) for object o is confirmed globally *if and only if critical section method invocation convention (CSMIC) for object o at* P_i *is confirmed for each* $P_i \in V$*.*

For each critical section 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 into 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 4 *(The global critical section problem). Assume that a pair of numbers l and k $(0 \leq l < k \leq n)$ is given on network $G = (V, E)$. Then, an object (l, k)-GCS solves* the global critical section problem *on G if and only if the following two conditions hold in each configuration C.*

- *Safety: $l \leq |\mathcal{CS}_{(l,k)\text{-GCS}}(C)| \leq k$ at any time.*
- *Liveness: Each process $P_i \in V$ changes* OutCS *and* InCS *states alternately infinitely often.*

For given l and k, we call the global CS problem as *the global (l, k)-CS problem.*

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

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 period between the invocation of the Exit() (*resp.* Entry()) and completion of the exit from (*resp.* entry to) the CS.
- *Waiting time*: the maximum one of the waiting times for exit or entry.

Our proposed algorithm uses a coterie [14] for information exchange between processes.

Definition 5 (Coterie [14]) *A* coterie $\mathcal{C}$ *under a set V is a set of subsets of V, i.e., $\mathcal{C} = \{Q_1, Q_2, ...\}$, where $Q_i \subseteq V$ and it satisfies the following two conditions.*

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

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

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

The algorithm proposed by [15] is a distributed mutual exclusion algorithms that uses a coterie and it achieves a message complexity of $O(|Q|)$, where $|Q|$ is the maximum size of the quorums in a coterie. For example, the finite projective plane coterie and the grid coterie achieve $|Q| = O(\sqrt{n})$, where n is the total number of processes [15].

[1] The name of this performance measure differs among previous studies and some (*e.g.*, [2]) refer to this performance measure as the *synchronization delay*.

Algorithm 1 (l,k)-*GCS*

Local Variables:
  lmin : **critical section object for l-mutual inclusion**;
  kmex : **critical section 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 */

4 Proposed Algorithm

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*. *lmin* is an algorithm for l-mutual inclusion, and *kmex* is an algorithm for k-mutual exclusion. The algorithm (l,k)-*GCS* for each process $P_i \in V$ is presented in Algorithm 1. In (l,k)-*GCS*, we regards that each process state changes into OutCS (*resp.* InCS) immediately in (l,k)-*GCS*.Exit() (*resp.* (l,k)-*GCS*.Entry()), just after execution of *lmin*.Exit() (*resp.* *kmex*.Entry()), before execution of *kmex*.Exit() (*resp.* *lmin*.Entry()). We assume that, for each P_i, (l,k)-*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 each of their safety properties by these methods. That is, *lmin*.Exit() blocks if l processes are InCS, and *kmex*.Entry() blocks if k processes are InCS.

4.1 Proof of correctness of algorithm (l,k)-*GCS*

For each P_i, let $\#G_i$ (*resp.* $\#L_i, \#K_i$) be 1 if (l,k)-*GCS*.$state_i$ = InCS (*resp.* *lmin*.$state_i$ = InCS, *kmex*.$state_i$ = InCS) holds, otherwise 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, $l \leq \#L \leq n$ holds by the safety of *lmin*, and $0 \leq \#K \leq k$ holds by the safety of *kmex*. Because we assume that the initial configuration C_0 is safe, $l \leq \#G \leq k$ holds in C_0.

Lemma 2 *In the initial configuration C_0, lmin and kmex satisfy their 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 $l \leq \#G \leq k$ holds in C_0, $l \leq \#L \leq k$ and $l \leq \#K \leq k$ holds in C_0. Thus, *lmin* and *kmex* satisfy their safety in C_0. □

Lemma 3 *In any execution of (l,k)-GCS, CSMIC for lmin and kmex are 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* are also confirmed at P_i. Below we show only the case of *lmin*; we omit the case for *kmex* because it is 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() are 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 4 *In any execution of (l,k)-GCS, lmin and kmex satisfy safety and liveness properties.*

Proof. By lemma 2, in C_0, *lmin* and *kmex* satisfy their safety properties because (l,k)-*GCS.state*$_i$ = *lmin.state*$_i$ = *kmex.state*$_i$ holds for each P_i. By

lemma 3, CSMIC for *lmin* and *kmex* are confirmed globally. Because they are preconditions for the safety and liveness of *lmin* and *kmex*, the lemma holds. □

Lemma 5 (Safety) *The number of processes in* 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, each value of $\#G_i$, $\#L_i$ and $\#K_i$ in each point of the execution is 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 following 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 $\#G \leq \#K \leq k$ holds by the safety of *lmin* and *kmex*, $l \leq \#G \leq k$ holds. □

Lemma 6 (Liveness) *Each process $P_i \in V$ alternates its state infinitely often.*

Proof. By contrast, suppose that some processes do not change OutCS and InCS states alternately infinitely often. Let X be the set of such processes. In *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 that 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 symmetry 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, P_i is not blocked forever. Therefore, for the assumption, not only P_i but also all 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 5, $l \leq \#G \leq k$ holds. If a process P_j is blocked in *lmin*.Exit(), (l,k)-$GCS.state_j$ = $lmin.state_j$ = $kmex.state_j = \mathsf{InCS}$ holds, and if P_j is blocked in *kmex*.Entry(), (l,k)-$GCS.state_j$ = $lmin.state_j = kmex.state_j = \mathsf{OutCS}$ holds. Therefore, $\#G = \#L = \#K$ holds.

- Consider the case that all processes are blocked in *lmin*.Exit(). Then, $\#L = n$ holds. However, by the assumption that *lmin* satisfies its safety, $l = n$ holds. This is a contradiction because $l < k \leq n$ must hold by assumption.
- Consider the case that there exists a process which is blocked in *kmex*.Entry(). By the assumption that *lmin* satisfies its safety, $\#L \geq l$ holds.
  - Consider the case that $\#L = l$ holds. Because it is assumed that $l < k$ holds, $\#L < k$ holds, that is, $\#L = \#K < k$ holds. Because *kmex* satisfies its liveness, 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 that $\#L > l$ holds. Because *lmin* satisfies its liveness, a process which is blocked in *lmin*.Exit() is eventually unblocked. This contradicts the assumption that all processes are blocked forever. □

By lemmas 5 and 6, we derived the following theorem.

Theorem 7 *(l,k)-GCS solves the global (l,k)-CS problem.* □

5 An Example of the l-Mutual Inclusion

Now, to show a concrete algorithm (l,k)-*GCS* based on the discussion in section 4, we propose a class $MUTIN(l)$ for l-mutual inclusion. A formal description of the class $MUTIN(l)$ for each process $P_i \in V$ is provided in Algorithm 2.

First, we present an outline how each process know the set of processes in InCS state in a distributed manner with quorums. When P_i changes its state, P_i notifies each process in a quorum Q_i its state. When P_i wants to know the set of processes in InCS, P_i contacts with 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 notifies its state to P_j, and P_j sends the state of P_k to P_i. For this reason, when P_i contacts each process in Q_i, P_i 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 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 about the states of processes to

Algorithm 2 A class description $MUTIN(l)$ 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 : **critical section 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 A class description $MUTIN(l)$ 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 $(reqCnt_i = reqCnt)$
    $currentInCS_i := currentInCS_i \cup procsInCS$;

P_i. Note that, $P_k \in R_i \Leftrightarrow P_i \in Q_k$ holds. The value of $procsInCS_i$ is maintained by the following way.

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

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

When P_i wishes to change its state into OutCS by Exit(), the safety is maintained by 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 $procsInCS$ which P_i received from each $P_j \in Q_i$ in variable $currentInCS_i$. That is, $currentInCS_i = \bigcup_{P_j \in Q_i} procsInCS_j$ holds.
- If $|currentInCS_i| \geq l + 1$ holds, then at least $l + 1$ processes are in InCS state. Thus, even if P_i changes its state from InCS to OutCS, at least l processes remain in InCS state. Then, 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.

Above idea guarantees safety if only one process wishes to change its state into OutCS, however, it does not if more than one processes wish to change their state into OutCS. To avoid this situation, we serialize requests which occur concurrently. One of the typical techniques to serialize is using the priority based on the timestamp and the preemption mechanism of permissions. This technique is employed in a lot of distributed mutual exclusion algorithms. We use this technique for serialization, however, for simplicity of the description of the proposed algorithm, we use an ordinary mutual exclusion algorithm [15] in the proposed algorithm instead of explicitly use timestamp and preemption mechanism. This is because typical ordinary mutual exclusion algorithms use the same mechanism for serialization, and hence underlaying mechanism is essentially the same. We denote the object for the ordinary mutual exclusion with *mx*. When a process wishes to change its state into OutCS, it invokes the *mx*.Entry() method and this allows it to enter the CS of mutual exclusion. After changing its state into OutCS successfully, it invokes the *mx*.Exit() method and this allows it to exit the CS of mutual exclusion. Thus, by incorporating a distributed mutual exclusion algorithm *mx*, the state change from InCS to OutCS is serialized between processes. Additionally, before 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. By this way, it is ensured that each process $P_k \in currentInCS_i$ is in InCS. Thus, $\#L \geq |currentInCS_i|$ is guaranteed.

Finally, we explain the idea to guarantee liveness. When exactly l processes are in InCS state, P_i observes this by the Query/Response1 message exchange, and P_i is blocked. When process P_k enters 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 such P_j because of the intersection property of quorums.

Even if there are more than l processes in InCS state, there is a case that P_i observes that the number of processes in InCS state is l by the Query/Response1 message exchange. When this occurs, P_i is blocked not to violate the safety. This case occurs if the Release message from some P_k is in transit towards $P_j \in Q_k \cap Q_i$ by asynchrony of message passing when P_j handles the Query message from P_i. Even if this case occurs, the Release message of P_k eventually arrives to 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 single Response2 message, it is enough for each process to send 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 Response1/Response2 message to distinguish it from the corresponding Query message.
- $procsInCS_i$: **set of processIDs**
  - A set of processes in InCS state to the best knowledge of P_i.
- $currentInCS_i$: **set of processIDs**
  - A set of processes in InCS state, which are gathered by P_i. That is, each process in this set is known to be in InCS state by some process in quorum Q_i.
- $ackFrom_i$: **set of processIDs**, **initially** $\emptyset$
  - A set of processes from which P_i receives 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 while $ackFrom_i = Q_i$ holds. Due to this handshake, $P_i \notin procsInCS_j$ is guaranteed 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 sets when P_i receives a Query message.
- $respAgainReqCnt_i$: **integer**, **initially** 0
  - Request count value for the Query of the process $responseAgainTo_i$.

5.1 Proof of correctness of $MUTIN(l)$

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

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

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

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

Consider the case that, in the configuration such that $P_j \in procsInCS_i \Rightarrow state_j = \mathsf{InCS}$ holds, $state_j$ changes from InCS to OutCS. Such 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, $P_j \in procsInCS_i \Rightarrow state_j = \mathsf{InCS}$ holds.

Now, we show that the safety is guaranteed. In the initial configuration, it is clear that the safety is guaranteed because $\#L \geq l$. We observe the execution after that. In the algorithm, only when $|currentInCS_i| \geq l + 1$ is satisfied, P_i exits 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. By mx in Exit(), because other processes than P_i do not invoke Exit(), $P_j \in currentInCS_i \Rightarrow state_j = \mathsf{InCS}$ 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, lemma holds. □

Lemma 9 (Liveness) *Each process* $P_i \in V$ *changes* OutCS *and* InCS *states alternately infinitely often.*

Proof. By contrast, suppose that some processes do not change OutCS and InCS states alternately 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 Exit() method, or (3) P_i is blocked by the second **wait** statement in Exit() method.

Any process is not 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 that 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 we have incorporated a mutual exclusion algorithm with liveness. Thus, at least one process is blocked in case (2).

The number of processes that is blocked in case (2) is exactly one because no two process 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 in case (2). Each non-blocked process in InCS state eventually calls the Exit() method and it is then blocked by mx.Entry() because P_k obtains the lock of mutual exclusion. Now the system reaches a configuration in which

P_k is blocked in case (2), remaining $n-1$ processes are blocked in case (1), and all the processes are in InCS state.

Finally, we show that P_k is unblocked eventually. Recall that P_k is blocked in case (2), *i.e.*, it is waiting for a 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 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 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 it is unblocked. □

Lemma 10 *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 by [15]. 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 for Query messages: $|Q_i|$ Response2 messages. Therefore, $O(|Q|)$ messages are exchanged.

In total, $O(|Q|)$ messages are exchanged. □

Lemma 11 *The waiting time of MUTIN(l) is 7.*

Proof. The waiting time is 3 for the mutual exclusion algorithm employed by *MUTIN*(l), which was described by Maekawa [15] (2 for Entry() and 1 for Exit(); see [2].)

In Exit(), a chain of messages, i.e., Query, Response1, Acquire, 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 8-11, we derived the following theorem.

Theorem 12 *MUTIN*(l) *solves the* l*-mutual inclusion problem with a message complexity of* $O(|Q|)$ *where* $|Q|$ *is the maximum size of the quorums of a coterie used by MUTIN*(l)*. The waiting time of MUTIN*(l) *is 7.* □

6 Discussion

In this section, finally, we discuss the case that, 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 10, the message complexity is $O(|Q|)$. Additionally, by the proof of lemma 11, both of waiting times for exit and entry of (l, k)-*GCS* are 9. Thus, by theorem 7, we derive the following theorem.

Theorem 13 (l, k)*-GCS solves the global* (l, k)*-CS problem with a message complexity of* $O(|Q|)$ *where* $|Q|$ *is the maximum size of the quorums of a coterie used by* (l, k)*-GCS. The waiting time of* (l, k)*-GCS is 9.* □

7 Conclusion

In this paper, we discuss the global critical section problem in asynchronous message passing distributed systems. Because this problem is useful for 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 work is supported in part by KAKENHI No. 16K00018 and 26330015.

References

[1] E. W. Dijkstra, Solution of a problem in concurrent programming control, Communications of the ACM 8 (9) (1965) 569.

[2] P. C. Saxena, J. Rai, A survey of permission-based distributed mutual exclusion algorithms, Computer Standards & Interfaces 25 (2) (2003) 159–181.

[3] N. Yadav, S. Yadav, S. Mandiratta, A review of various mutual exclusion algorithms in distributed environment, International Journal of Computer Applications 129 (14).

[4] H. Kakugawa, On the family of critical section problems, Information Processing Letters 115 (2015) 28–32.

[5] H. Kakugawa, S. Fujita, M. Yamashita, T. Ae, Availability of k-coterie, IEEE Transaction on Computers 42 (5) (1993) 553–558.

[6] S. Bulgannawar, N. H. Vaidya, A distributed k-mutual exclusion algorithm, in: Proceedings of the 15th International Conference on Distributed Computing Systems, 1995, pp. 153–160.

[7] Y.-I. Chang, B.-H. Chen, A generalized grid quorum strategy for k-mutual exclusion in distributed systems, Information Processing Letters 80 (4) (2001) 205–212.

[8] U. Abraham, S. Dolev, T. Herman, I. Koll, Self-stabilizing l-exclusion, Theoretical Computer Science 266 (1-2) (2001) 653–692.

[9] P. Chaudhuri, T. Edward, An algorithm for k-mutual exclusion in decentralized systems, Computer Communications 31 (14) (2008) 3223–3235.

[10] V. A. Reddy, P. Mittal, I. Gupta, Fair k mutual exclusion algorithm for peer to peer systems, in: Proceedings of the 28th International Conference on Distributed Computing Systems, 2008.

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

[12] H. Kakugawa, Mutual inclusion in asynchronous message-passing distributed systems, Journal of Parallel Distributed Computing 77 (2015) 95–104.

[13] S. Kamei, H. Kakugawa, An asynchronous message-passing distributed algorithm for the generalized local critical section problem, Algorithms 10 (38).

[14] H. Garcia-Molina, D. Barbara, How to assign votes in a distributed system, Journal of the ACM 32 (4) (1985) 841–860.

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