algorithms

MDPI

Article

An Asynchronous Message-Passing Distributed Algorithm for the Generalized Local Critical Section Problem †

Sayaka Kamei [1,*] and Hirotsugu Kakugawa [2]

1 Graduate School of Engineering, Hiroshima University, 1-4-1 Kagamiyama, Higashi-Hiroshima, Hiroshima 739-8527, Japan
2 Graduate School of Information Science and Technology, Osaka University, 1-4, Yamadaoka, Suita, Osaka 565-0871, Japan; kakugawa@ist.osaka-u.ac.jp
* Correspondence: s-kamei@se.hiroshima-u.ac.jp; Tel.: +81-82-424-7685

† This paper is an extended version of our paper published in International Conference on Network, Communication and Computing 2016, Kyoto, Japan, 17–21 December 2016.

Academic Editor: Andras Farago
Received: 27 January 2017; Accepted: 22 March 2017; Published: 24 March 2017

Abstract: This paper discusses the generalized local version of critical section problems including mutual exclusion, mutual inclusion, k-mutual exclusion and l-mutual inclusion. When a pair of numbers (l_i, k_i) is given for each process P_i, it is the problem of controlling the system in such a way that the number of processes that can execute their critical sections at a time is at least l_i and at most k_i among its neighboring processes and P_i itself. We propose the first solution for the generalized local $(l_i, |N_i| + 1)$-critical section problem (i.e., the generalized local l_i-mutual inclusion problem). Additionally, we show the relationship between the generalized local (l_i, k_i)-critical section problem and the generalized local $(|N_i| + 1 - k_i, |N_i| + 1 - l_i)$-critical section problem. Finally, we propose the first solution for the generalized local (l_i, k_i)-critical section problem for arbitrary (l_i, k_i), where $0 \le l_i < k_i \le |N_i| + 1$ for each process P_i.

Keywords: critical section; mutual exclusion; mutual inclusion

1. Introduction

The mutual exclusion problem is a fundamental process synchronization problem in concurrent systems [1–3]. It is the problem of controlling the system in such a way that no two processes execute their critical sections (CSs) at a time. Generalizations of mutual exclusion have been studied extensively, e.g., k-mutual exclusion [4–9], mutual inclusion [10] and l-mutual inclusion [11]. 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 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. In a similar way, the l-mutual inclusion problem is the complement of the k-mutual exclusion problem; unlike k-mutual exclusion, where at most k processes are in the CSs, l-mutual inclusion places at least l processes in the CSs. These generalizations are unified to a framework "the critical section problem" in [12]. Informally, the global (l, k)-CS problem is defined as follows. For each $0 \le l < k \le n$, the global (l, k)-CS problem has at least l and at most k processes in the CSs in the entire network.

This paper discusses the generalized local CS problem, which is a new version of CS problems. When the numbers l_i and k_i are given for each process P_i, it is the problem of controlling the system in such a way that the number of processes that can execute their CSs at a time is at least l_i and at most k_i among its neighbors and itself. In this case, we call this problem "the generalized local

(l_i, k_i)-critical section problem". Note that the local (l, k)-CS problem assumes that the values of l and k are shared among all processes in the network, whereas the generalized local CS problem assumes that the values of l_i and k_i are set for each process P_i. These are the generalizations of local mutual exclusion [13–17], local k-mutual exclusion [18] and local mutual inclusion [11]. If every process has $(0, 1)$, then the problem is the local mutual exclusion problem. If every process has $(0, k)$, then the problem is the local k-mutual exclusion. If every process has $(1, |N_i| + 1)$, then the problem is the local mutual inclusion problem, where N_i is the set of P_i's neighboring processes. The global CS problem is a special case of the local CS problem when the network topology is complete. However, to the best of our knowledge, our algorithm in this paper is the first solution for the generalized local (l_i, k_i)-CS problem.

The generalized local (l_i, k_i)-CS problem is interesting not only theoretically, but also practically, because it is useful for fault-tolerance and load balancing of distributed systems. For example, we can consider the following future applications.

- One application is a formulation of the dynamic invocation of servers for the load balancing. The minimum number of servers that are always invoked for quick responses to requests for P_i is l_i. The number of servers is dynamically changed by the system load. However, the total number of servers is limited by available resources like bandwidth for P_i, and the number is k_i.
- Another is fault-tolerance services if each process in the CS provides a service for the network. Because every process has direct access to at least l_i servers, it guarantees fault-tolerant services. However, because providing services involve a significant cost, the number of servers should be limited at most k_i for each process.
- The other is that each process in the CS provides service A, and other processes provide service B for the network. Then, every process in the network has direct access to at least l_i servers of A and has direct access to at least $|N_i| + 1 - k_i$ servers of B.

In each case, the numbers l_i and k_i can be set for each process.

In this paper, we propose a distributed algorithm for the generalized local (l_i, k_i)-CS problem for arbitrary (l_i, k_i), where $0 \leq l_i < k_i \leq |N_i| + 1$ for each process P_i. To this end, we first propose a distributed algorithm for the generalized local $(l_i, |N_i| + 1)$-CS problem (we call it generalized local l_i-mutual inclusion problem). It is the first algorithm for the problem. Next, we show that the generalized local (l_i, k_i)-CS algorithms and the generalized local $(|N_i| + 1 - k_i, |N_i| + 1 - l_i)$-CS algorithms are interchangeable by swapping process state, in the CS and out of the CS. By using this relationship between these two problems, we propose a distributed algorithm for the generalized local (l_i, k_i)-CS problem for arbitrary (l_i, k_i), where $0 \leq l_i < k_i \leq |N_i| + 1$ for each process P_i. We assume that there is a process P_{LDR}, such that $|N_{LDR}| \geq 4$, $l_{LDR} \leq k_{LDR} - 3$ and $l_q \leq k_q - 3$ for each $P_q \in N_{LDR}$.

This paper is organized as follows. Section 2 provides several definitions and problem statements. Section 3 provides a solution to the generalized local $(l_i, |N_i| + 1)$-CS (i.e., generalized local l_i-mutual inclusion) problem. Section 4 presents an observation on the relationships between the generalized local (l_i, k_i)-CS problem and the generalized local $(|N_i| + 1 - k_i, |N_i| + 1 - l_i)$-CS problem. Section 5 provides a solution to the generalized local (l_i, k_i)-CS problem. In Section 6, we give a conclusion and discuss future works.

2. Preliminaries

2.1. System Model

Let $V = \{P_1, P_2, ..., P_n\}$ be a set of n processes and $E \subseteq V \times V$ be a set of bidirectional communication links in a distributed system. Each communication link is FIFO. Then, the topology of the distributed system is represented as an undirected graph $G = (V, E)$. By N_i, we denote the set of neighboring processes of P_i. That is, $N_i = \{P_j \mid (P_i, P_j) \in E\}$. By $dist(P_i, P_j)$, we denote the distance between processes P_i and P_j. 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.

A set of local variables defines the local state of a process. By Q_i, we denote the local state of each process $P_i \in V$. A tuple of the local state of each process $(Q_1, Q_2, ..., Q_n)$ forms a configuration of a distributed system.

2.2. Problem

We assume that each process $P_i \in V$ maintains a variable $state_i \in \{\mathsf{InCS}, \mathsf{OutCS}\}$. For each configuration C, let $\mathcal{CS}(C)$ (resp., $\overline{\mathcal{CS}(C)}$) be the set of processes P_i with $state_i = \mathsf{InCS}$ (resp., $state_i = \mathsf{OutCS}$) in C. For each configuration C and each process P_i, let $\mathcal{CS}_i(C)$ (resp., $\overline{\mathcal{CS}_i(C)}$) be the set $\mathcal{CS}(C) \cap (N_i \cup \{P_i\})$ (resp., $\overline{\mathcal{CS}(C)} \cap (N_i \cup \{P_i\})$). The behavior of each process P_i is as follows, where we assume that P_i eventually invokes entry-sequence when it is in the OutCS state, and P_i eventually invokes exit-sequence when it is in the InCS state.

```
state_i := (Initial state of P_i in the initial configuration C_0);
while true{
  if(state_i = OutCS){
    Entry-Sequence;
    state_i := InCS;
    /* Critical Section */
  }
  if(state_i = InCS){
    Exit-Sequence;
    state_i := OutCS;
    /* Remainder Section */
  }
}
```

Definition 1. *(The generalized local critical section problem). Assume that a pair of numbers l_i and k_i ($0 \le l_i < k_i \le |N_i| + 1$) is given for each process $P_i \in V$ on network $G = (V, E)$. Then, a protocol solves the generalized local critical section problem on G if and only if the following two conditions hold in each configuration C.*

- *Safety: For each process $P_i \in V$, $l_i \le |\mathcal{CS}_i(C)| \le k_i$ at any time.*
- *Liveness: Each process $P_i \in V$ changes* OutCS *and* InCS *states alternately infinitely often.*

We call the generalized local CS problem when l_i and k_i are given for each process P_i "the generalized local (l_i, k_i)-CS problem".

We assume that the initial configuration C_0 is safe, that is each process P_i satisfies $l_i \le |\mathcal{CS}_i(C_0)| \le k_i$. In the case of $(l_i, k_i) = (0, 1)$ (resp., $(1, |N_i| + 1)$), the initial state of each process can be OutCS (resp., InCS) because it satisfies the condition for the initial configuration. In the case of $(l_i, k_i) = (1, |N_i|)$, the initial state of each process is obtained from a maximal independent set I as follows; a process is in the OutCS state if and only if it is in I. 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 the OutCS state initially, and some algorithms (e.g., token-based algorithms) assume that exactly one process is in the InCS state and other processes are in the OutCS state initially. Hence our assumption for the initial configuration is common for existing algorithms.

2.3. Performance Measure

We apply the following performance measure as message complexity to the generalized local CS algorithm: the number of message exchanges triggered by a pair of invocations of exit-sequence and entry-sequence.

3. Proposed Algorithm for the Generalized Local l_i-Mutual Inclusion

In this section, we propose an algorithm l_i-LMUTIN for the case that $k_i = |N_i| + 1$.

First, we explain how the safety is guaranteed. Initially, the configuration is safe, that is each process P_i satisfies $l_i \leq |\mathcal{CS}_i(C_0)| \leq |N_i| + 1$. When P_i wishes to be in the OutCS state, P_i requests permission by sending a $\langle \mathsf{Request}, ts_i, P_i \rangle$ message for each process in $N_i \cup \{P_i\}$. When P_i obtains a permission by receiving a $\langle \mathsf{Grant} \rangle$ message from each process in $N_i \cup \{P_i\}$, P_i changes to the OutCS state. Each process P_j grants permissions to $|N_j| - l_j + 1$ processes at each time. Hence, at least l_j processes in $N_j \cup \{P_j\}$ cannot be in the OutCS state at the same time. When P_i wishes to be in the InCS state, P_i changes to the InCS state and sends a message $\langle \mathsf{Release}, P_i \rangle$ for each process in $N_i \cup \{P_i\}$ to manage the next request for exiting the CS.

Next, we explain how the liveness is guaranteed. We incorporate the timestamp mechanism proposed by [19] in our algorithm. Based on the priority of the timestamp for each request to change the state, a process preempts a permission when necessary, as proposed in [11,20,21]. The proposed algorithm uses $\langle \mathsf{Preempt}, P_i \rangle$ and $\langle \mathsf{Relinquish}, P_i \rangle$ messages for this purpose.

In the proposed algorithm, each process P_i maintains the following local variables.

- $state_i$: The current state of P_i: InCS or OutCS.
- ts_i: The current value of the logical clock [19].
- $nGrants_i$: The number of grants that P_i obtains for exiting the CS.
- $grantedTo_i$: A set of timestamps (ts_j, P_j) for the requests to P_j's exiting the CS that P_i has been granted, but that P_j has not yet released.
- $pendingReq_i$: A set of timestamps (ts_j, P_j) for the requests to P_j's exiting the CS that are pending.
- $preemptingNow_i$: A process id P_j such that P_i preempts a permission for P_j's exiting the CS if the preemption is in progress.

For each request, a pair (ts_i, P_i) is used as its timestamp. We implicitly assume that the value of the logical clock [19] is attached to each message exchanged. Thus, in the proposed algorithm, we omit a detailed description of the maintenance protocol for ts_i. The timestamps are compared as follows: $(ts_i, P_i) < (ts_j, P_j)$ iff $ts_i < ts_j$ or $(ts_i = ts_j) \wedge (P_i < P_j)$.

Formal description of the proposed algorithm for each process P_i is presented in Algorithms 1 and 2. When each process P_i receives a message, it invokes a corresponding message handler. Each message handler is executed atomically. That is, if a message handler is being executed, the arrival of a new message does not interrupt the message handler. In this algorithm description, we use the statement wait until (conditional expression). By this statement, a process is blocked until the conditional expression is true. While a process is blocked by the wait until statement and it receives a message, it invokes a corresponding message handler.

Algorithm 1 Local variables for process P_i in algorithm l_i-LMUTIN

$state_i \in \{\mathsf{InCS}, \mathsf{OutCS}\}$, **initially** $state_i = \begin{cases} \mathsf{InCS} & (P_i \in \mathcal{CS}(C_0)) \\ \mathsf{OutCS} & (P_i \notin \mathcal{CS}(C_0)) \end{cases}$

ts_i : **integer**, **initially** 0;
$nGrants_i$: **integer**, **initially** 0;
$grantedTo_i$: **set of (integer, processID)**, **initially** $\{(0, P_j \in \mathcal{CS}_i(C_0))\}$;
$pendingReq_i$: **set of (integer, processID)**, **initially** $\emptyset$;
$preemptingNow_i$: **processID**, **initially nil**;

Algorithm 2 Algorithm l_i-LMUTIN: exit-sequence, entry-sequence and message handlers.

Exit-Sequence:
$ts_i := ts_i + 1;$
$nGrants_i := 0;$
for-each $P_j \in (N_i \cup \{P_i\})$ **send**$\langle \mathsf{Request}, ts_i, P_i \rangle$ **to** P_j;
wait until $(nGrants_i = |N_i| + 1)$;
$state_i :=$ OutCS;

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

On receipt of a $\langle \mathsf{Request}, ts_j, P_j \rangle$ **message:**
$pendingReq_i := pendingReq_i \cup \{(ts_j, P_j)\}$;
if $(|grantedTo_i| < |N_i| - l_i + 1)\{$
$(ts_h, P_h) := deleteMin(pendingReq_i)$;
$grantedTo_i := grantedTo_i \cup \{(ts_h, P_h)\}$;
send $\langle \mathsf{Grant} \rangle$ **to** P_h;
$\}$ **else if** $(preemptingNow_i = \textbf{nil})\{$
$(ts_h, P_h) := max(grantedTo_i)$;
if $((ts_j, P_j) < (ts_h, P_h))\{$
$preemptingNow_i := P_h$;
send $\langle \mathsf{Preempt}, P_i \rangle$ **to** P_h;
$\}$
$\}$

On receipt of a $\langle \mathsf{Grant} \rangle$ **message:**
$nGrants_i := nGrants_i + 1$;

On receipt of a $\langle \mathsf{Release}, P_j \rangle$ **message:**
if $(P_j = preemptingNow_i)$ $preemptingNow_i := \textbf{nil}$;
Delete $(*, P_j)$ from $grantedTo_i$;
if $(pendingReq_i \neq \emptyset)\{$
$(ts_h, P_h) := deleteMin(pendingReq_i)$;
$grantedTo_i := grantedTo_i \cup \{(ts_h, P_h)\}$;
send $\langle \mathsf{Grant} \rangle$ **to** P_h;
$\}$

On receipt of a $\langle \mathsf{Preempt}, P_j \rangle$ **message:**
if $(state_i = \mathsf{InCS})\{$
$nGrants_i := nGrants_i - 1$;
send $\langle \mathsf{Relinquish}, P_i \rangle$ **to** P_j;
$\}$

On receipt of a $\langle \mathsf{Relinquish}, P_j \rangle$ **message:**
$preemptingNow_i := \textbf{nil}$;
Delete $(*, P_j)$ from $grantedTo_i$, and let (ts_j, P_j) be the deleted item;
$pendingReq_i := pendingReq_i \cup \{(ts_j, P_j)\}$;
$(ts_h, P_h) := deleteMin(pendingReq_i)$;
$grantedTo_i := grantedTo_i \cup \{(ts_h, P_h)\}$;
send $\langle \mathsf{Grant} \rangle$ **to** P_h;

3.1. Proof of Correctness

In this subsection, we show the correctness of l_i-LMUTIN.

Lemma 1. *(Safety) For each process* $P_i \in V$, $l_i \leq |\mathcal{CS}_i(C)| \leq |N_i| + 1$ *holds at any configuration* C.

Proof. We assume that the initial configuration C_0 is safe, i.e., $l_i \leq |\mathcal{CS}_i(C_0)|$. Therefore, we consider the process P_i, which becomes unsafe first for the contrary. Suppose that $|\mathcal{CS}_i(C)| < l_i$, that is $|\overline{\mathcal{CS}_i(C)}| > |N_i| + 1 - l_i$. Because $|\mathcal{CS}_i(C_0)| \geq l_i$, consider a process $P_j \in N_i \cup \{P_i\}$, which became the OutCS state by the $(|N_i| + 2 - l_i)$-th lowest timestamp among processes in $\overline{\mathcal{CS}_i(C)}$. Then, P_j obtains permission to be the OutCS state from each process in $N_j \cup \{P_j\}$. This implies that P_i receives a request $\langle \mathsf{Request}, ts_j, P_j \rangle$ from P_j and that P_i sends a permission $\langle \mathsf{Grant} \rangle$ to P_j. Because P_i grants at most $|N_i| + 1 - l_i$ permissions to exit the CS at each time, P_j cannot obtain a permission from P_i; this is a contradiction. □

Lemma 2. *(Liveness) Each process P_i changes into the* InCS *and* OutCS *states alternately infinitely often.*

Proof. By contrast, suppose that some processes do not change into the InCS and OutCS states alternately infinitely often. Let P_i be such a process where the lowest timestamp value for its request to be the OutCS state is (ts_i, P_i). Without loss of generality, we assume that P_i is blocked in the InCS state. That is, P_i is blocked by the wait until statement in the exit-sequence (recall that each process changes into the InCS state eventually when it is in the OutCS state). Let P_j be any process in N_i.

- Suppose that P_j changes into the InCS and OutCS states alternately infinitely often. After P_j receives the $\langle \mathsf{Request}, ts_i, P_i \rangle$ message from P_i, the value of (ts_j, P_j) exceeds the timestamp (ts_i, P_i) for P_i's request. Because, by this algorithm, the request with the lowest timestamp is granted preferentially, it is impossible for P_j to change into the InCS and OutCS states alternately infinitely often. Then, P_j eventually sends a $\langle \mathsf{Grant} \rangle$ message to P_i, and P_i eventually sends a $\langle \mathsf{Grant} \rangle$ message to itself.
- Suppose that P_j does not change into the InCS and OutCS states alternately infinitely often. Because the timestamp of P_i is smaller than that of P_j, by assumption, P_j's permission is preempted, and a $\langle \mathsf{Grant} \rangle$ message is sent from P_j to P_i. In addition, P_i sends a $\langle \mathsf{Grant} \rangle$ message to itself.

Therefore, P_i eventually receives a $\langle \mathsf{Grant} \rangle$ message from each process in $N_i \cup \{P_i\}$, and the wait until statement in the exit-sequence does not block P_i forever. □

3.2. Performance Analysis

Lemma 3. *The message complexity of l_i-LMUTINfor $P_i \in V$ is $3(|N_i| + 1)$ in the best case and $6(|N_i| + 1)$ in the worst case.*

Proof. First, let us consider the best case. In exit-sequence, for P_i's exiting the CS, P_i sends a $\langle \mathsf{Request}, ts_i, P_i \rangle$ message to each process in $N_i \cup \{P_i\}$; each process in $N_i \cup \{P_i\}$ sends a $\langle \mathsf{Grant} \rangle$ message to P_i. In entry-sequence, after P_i's entering the CS, P_i sends a $\langle \mathsf{Release}, P_i \rangle$ message to each process in $N_i \cup \{P_i\}$. Thus, $3(|N_i| + 1)$ messages are exchanged.

Next, let us consider the worst case. For P_i's exiting the CS, P_i sends a $\langle \mathsf{Request}, ts_i, P_i \rangle$ message to each process P_j in $N_i \cup \{P_i\}$. Then, P_j sends a $\langle \mathsf{Preempt}, P_j \rangle$ message to the process P_m to which P_j sends a $\langle \mathsf{Grant} \rangle$ message, P_m sends a $\langle \mathsf{Relinquish}, P_m \rangle$ message back to P_j and P_j sends a $\langle \mathsf{Grant} \rangle$ message to P_i. After P_i's entering the CS, P_i sends a $\langle \mathsf{Release}, P_i \rangle$ message to each process P_j in $N_i \cup \{P_i\}$. Then, P_j sends a $\langle \mathsf{Grant} \rangle$ message to return a grant to P_m or grant to some process with the highest priority in $pendingReq_j$. Thus, $6(|N_i| + 1)$ messages are exchanged. □

Theorem 1. *l_i-LMUTIN solves the generalized local $(l_i, |N_i| + 1)$-critical section problem with a message complexity of $O(\Delta)$, where Δ is the maximum degree of a network.*

4. The Generalized Local Complementary Theorem

In this section, we discuss the relationship between the generalized local CS problems.

Let $\mathcal{A}^{\mathcal{G}}{}_{(l,k)}$ be an algorithm for the global (l,k)-CS problem, and $\mathcal{A}^{\mathcal{L}}{}_{(l_i,k_i)}$ be an algorithm for the generalized local (l_i,k_i)-CS problem. By Co-$\mathcal{A}^{\mathcal{G}}{}_{(l,k)}$(resp., Co-$\mathcal{A}^{\mathcal{L}}{}_{(l_i,k_i)}$), we denote a complement algorithm of $\mathcal{A}^{\mathcal{G}}{}_{(l,k)}$(resp., $\mathcal{A}^{\mathcal{L}}{}_{(l_i,k_i)}$), which is obtained by swapping the process states, InCS and OutCS.

In [12], it is shown that the complement of $\mathcal{A}^{\mathcal{G}}{}_{(l,k)}$ is a solution to the global $(n-k, n-l)$-CS problem. We call this relation the complementary theorem. Now, we show the generalization of the complementary theorem for the settings of local CS problems.

Theorem 2. *For each process P_i, a pair of numbers l_i and k_i $(0 \le l_i < k_i \le |N_i|+1)$ is given. Then,* Co-$\mathcal{A}^{\mathcal{L}}{}_{(l_i,k_i)}$ *is an algorithm for the generalized local $(|N_i|+1-k_i, |N_i|+1-l_i)$-CS problem.*

Proof. By $\mathcal{A}^{\mathcal{L}}{}_{(l_i,k_i)}$, at least l_i and at most k_i processes among each process and its neighbors are in the CS. Hence, by Co-$\mathcal{A}^{\mathcal{L}}{}_{(l_i,k_i)}$, at least l_i and at most k_i processes among each process and its neighbors are out of the CS. That is, at least $|N_i|+1-k_i$ and at most $|N_i|+1-l_i$ processes among each process and its neighbors are in the CS. □

By Theorem 2, Co-(l_i-LMUTIN) is an algorithm for the generalized local $(0,k_i)$-CS problem where $k_i = |N_i|+1-l_i$. We call it k_i-LMUTEX.

5. Proposed Algorithm for the Generalized Local CS Problem

In this section, we propose an algorithm LKCSfor the generalized local (l_i,k_i)-CS problem for arbitrary (l_i,k_i), where $0 \le l_i < k_i \le |N_i|+1$ for each process P_i. We assume that, the initial configuration C_0 is safe. Before we explain the technical details of LKCS, we explain the basic idea behind it.

5.1. Idea

The main strategy in LKCS is the composition of two algorithms, l_i-LMUTIN and k_i-LMUTEX. In the following description, we simply call these algorithms LMUTIN and LMUTEX, respectively. The idea of the composition in LKCS is as follows.

Exit-Sequence:
Exit-Sequence for LMUTIN;
Exit-Sequence for LMUTEX;

Entry-Sequence:
Entry-Sequence for LMUTEX;
Entry-Sequence for LMUTIN;

This idea does not violate the safety by the following observation.

- Exit-sequence keeps the safety because invocation of exit-sequence for LMUTIN keeps the safety, and invocation of exit-sequence for LMUTEX trivially keeps the safety.
- Similarly, entry-sequence keeps the safety because invocation of entry-sequence for LMUTEX keeps the safety, and invocation of entry-sequence for LMUTIN trivially keeps the safety.

Because invocations of exit-sequence for LMUTIN in exit-sequence and entry-sequence for LMUTEX in entry-sequence may block a process forever, i.e., deadlocks and starvations, we need some mechanism to such situation which makes the proposed algorithm non-trivial.

A problem in the above idea is the possibility of deadlocks in the following situation. There is a process P_u with $state_u = \mathsf{InCS}$ such that $|\mathcal{CS}_u(C)| = l_u$ or P_u has a neighbor $P_v \in N_u$ with $|\mathcal{CS}_v(C)| = l_v$. Then, P_u cannot change its state by exit-sequence until at least one of its neighbors $P_w \in N_u$ with $state_w = \mathsf{OutCS}$ changes P_w's state by entry-sequence. If $|\mathcal{CS}_w(C)| = k_w$ or P_w has a neighbor $P_x \in N_w$ with $|\mathcal{CS}_x(C)| = k_x$, P_w cannot change its state by entry-sequence until at least one of its neighbors $P_y \in N_w$ with $state_y = \mathsf{InCS}$ changes P_y's state by exit-sequence. In the network, if every process is in such situation, a deadlock occurs.

To avoid such a deadlock, we introduce a mechanism "sidetrack", meaning that some processes reserve some grants, which are used only when the system is suspected to be a deadlock. Hence, in a normal situation, i.e., not suspected to be a deadlock, the number of processes in the CS is limited. In this sense, LKCS is a partial solution to the (l_i, k_i)-CS problem unfortunately. Currently, a full solution to the problem is not known and left as a future task.

The idea of the "sidetrack" in LKCS is explained as follows. We select a process, say P_{LDR}, with $|N_{LDR}| \geq 4$ as a "leader", and each process P_q within two hops from the leader may allow at least $l_q + 1$ and at most $k_q - 1$ processes to be in the CSs locally in a normal situation. We assume that $k_q - l_q \geq 3$, because $k_q - 1 - (l_q + 1) \geq 1$. Other processes P_i may allow at least l_i and at most k_i processes to be in the CSs locally in any situation and $k_i - l_i \geq 1$. The leader observes the number of neighbor processes that may be blocked, and when the leader itself and all of the neighbors can be blocked, the leader suspects that the system is in a deadlock situation. Then, the leader designates a process within one hop (including the leader itself) to use the "sidetrack" to break the chain of cyclic blocking. Because the designated process P_q uses one extra CS exit/entry, the number of processes in the CSs is at least $(l_q + 1) - 1 = l_q$ and at most $(k_q - 1) + 1 = k_q$, and hence, LKCS does not deviate from the restriction of the (l_i, k_i)-CS problem. The suspicion by the leader process P_{LDR} is not always correct, i.e., P_{LDR} may suspect that the system is in a deadlock when this is not true. However, incorrect suspicion does not violate the safety of the problem specification.

5.2. Details of LKCS

We explain the technical details of LKCS below. Formal description of LKCS for each process P_i is presented in Algorithms 3–7. The execution model of this algorithm is the same as the previous section, except that the while statement is used in LKCS. By while (conditional expression) $\{statement\}$, a process is blocked until the conditional expression is true. While a process is blocked by this statement, it executes only the statement between braces and message handlers. While a process is blocked by this statement and it receives a message, it invokes a corresponding message handler. That is, if the statement between braces is empty, this while statement is same as wait until statement.

Algorithm 3 Local variables and macros for process P_i in algorithm LKCS

Local Variables:

enum *at* $\{\mathsf{MUTEX}, \mathsf{MUTIN}\}$;

$state_i \in \{\mathsf{InCS}, \mathsf{OutCS}\}$, **initially** $state_i = \begin{cases} \mathsf{InCS} & (P_i \in \mathcal{CS}(C_0)) \\ \mathsf{OutCS} & (P_i \notin \mathcal{CS}(C_0)) \end{cases}$

ts_i : **integer**, **initially** 1;

$nGrants_i[at]$: **set of processID**, **initially** $\emptyset$;

$grantedTo_i[at]$: **set of (integer, processID)**, **initially** $\{(1, P_j \in \mathcal{CS}_i(C_0))\}$;

$pendingReq_i[at]$: **set of (integer, processID)**, **initially** $\emptyset$;

$preemptingNow_i[at]$: **(integer, processID)**, **initially nil**;

Local Variable only for a leader P_{LDR}:

$candidate_{LDR}$: **set of (integer, processID)**, **initially** $\emptyset$;

Macros:

$L_i \equiv \begin{cases} l_i & dist(P_{LDR}, P_i) > 2 \\ l_i + 1 & dist(P_{LDR}, P_i) \leq 2 \end{cases}$

$K_i \equiv \begin{cases} k_i & dist(P_{LDR}, P_i) > 2 \\ k_i - 1 & dist(P_{LDR}, P_i) \leq 2 \end{cases}$

$Grant_{LDR} \equiv \{P_j \mid (*, P_j) \in grantedTo_{LDR}[\mathsf{MUTIN}] \wedge (*, P_j) \in grantedTo_{LDR}[\mathsf{MUTEX}]\}$

$Waiting_{LDR} \equiv |pendingReq_{LDR}[\mathsf{MUTIN}]| + |pendingReq_{LDR}[\mathsf{MUTEX}]| + |Grant_{LDR}|$

$Cond_i \equiv (at = \mathsf{MUTEX} \wedge |grantedTo_i[\mathsf{MUTEX}]| < K_i) \vee (at = \mathsf{MUTIN} \wedge |grantedTo_i[\mathsf{MUTIN}]| < |N_i| - L_i + 1)$

Algorithm 4 Algorithm LKCS: exit-sequence and entry-sequence.

Exit-Sequence:
$ts_i := ts_i + 1$;
$nGrants_i[\mathsf{MUTIN}] := \emptyset$;
for-each $P_j \in (N_i \cup \{P_i\})$ **send**$\langle \mathsf{Request}, \mathsf{MUTIN}, ts_i, P_i \rangle$ **to** P_j;
while $(|nGrants_i[\mathsf{MUTIN}]| < |N_i| + 1)\{$
  if $(P_i = P_{LDR} \wedge Waiting_{LDR} = |N_{LDR}| + 1)\{$
    /* The configuration may be in a deadlock. */
    TriggerNomination();
    wait until$(Waiting_{LDR} < |N_{LDR}| + 1)$;
  }
}
$state_i :=$ OutCS;
for-each $P_j \in (N_i \cup \{P_i\})$ **send**$\langle \mathsf{Release}, \mathsf{MUTEX}, P_i \rangle$ **to** P_j;

Entry-Sequence:
$nGrants_i[\mathsf{MUTEX}] := \emptyset$;
for-each $P_j \in (N_i \cup \{P_i\})$ **send**$\langle \mathsf{Request}, \mathsf{MUTEX}, ts_i, P_i \rangle$ **to** P_j;
while $(|nGrants_i[\mathsf{MUTEX}]| < |N_i| + 1)\{$
  if $(P_i = P_{LDR} \wedge Waiting_{LDR} = |N_{LDR}| + 1)\{$
    /* The configuration may be in a deadlock. */
    TriggerNomination();
    wait until$(Waiting_{LDR} < |N_{LDR}| + 1)$;
  }
}
$state_i :=$ InCS;
for-each $P_j \in (N_i \cup \{P_i\})$ **send**$\langle \mathsf{Release}, \mathsf{MUTIN}, P_i \rangle$ **to** P_j;

When the leader P_{LDR} suspects that the system is in a deadlock, it invokes the *TriggerNomination*() function and selects a process P_q within one hop (P_{LDR} itself or a neighbor of P_{LDR}) as a "trigger", and P_{LDR} sends a message (Trigger message type) to P_q so that P_q issues a special request. Then, P_q sends a special request message (RequestByTrigger message type) to each neighbor $P_r \in N_q$. This message also cancels the current request of P_q. After each P_r receives such special request from P_q, then P_r cancels the request of P_q by deleting $(*, P_q)$ from its pending list ($pendingReq_r$) and its granted list ($grantedTo_r$), inserts the special request $(0, P_q)$ to its granted list and immediately grants by using the "sidetrack". The deleted element $(*, P_q)$ is a request that P_r or other neighbors of P_q kept it waiting if the system is in a deadlock, and the inserted element $(0, P_q)$ cannot be preempted because it has the maximum priority.

We explain the technical details how the leader P_{LDR} suspects that the system is in a deadlock. When $Waiting_{LDR} = |N_{LDR}| + 1$, then $|pendingReq_{LDR}[\mathsf{MUTIN}]| + |pendingReq_{LDR}[\mathsf{MUTEX}]| + |Grant_{LDR}| = |N_{LDR}| + 1$. Because a request is not sent if a previous one is kept waiting, two pending lists $pendingReq_{LDR}[\mathsf{MUTIN}]$ and $pendingReq_{LDR}[\mathsf{MUTEX}]$ are disjoint. Thus, if there is a neighbor $P_q \in N_{LDR}$, which is not in these pending lists, then P_q's request is granted by P_{LDR}, but is kept waiting by other neighbor than P_{LDR} in the deadlock configuration. That is, P_q's request is in both of $grantedTo_{LDR}$. To the suspicion possible, we assume that, at the leader process, $(k_{LDR} - 1) - (l_{LDR} + 1) \geq 1$ holds, i.e., $k_{LDR} - l_{LDR} \geq 3$. The underlying LMUTIN (resp., LMUTEX) algorithm sends at most $|N_{LDR}| + 1 - (l_{LDR} + 1)$ (resp., $k_{LDR} - 1$) grants; the total number of grants of the two underlying algorithms is at most $|N_{LDR}| - l_{LDR} + k_{LDR} - 1$. Because $k_{LDR} - l_{LDR} \geq 3$, $|N_{LDR}| - l_{LDR} + k_{LDR} - 1 \geq |N_{LDR}| + 2 > |N_{LDR}| + 1 = |N_{LDR} \cup \{P_{LDR}\}|$ holds. This implies that there exists at least a process $P_q \in (N_{LDR} \cup \{P_{LDR}\})$ that receives both grants of LMUTIN and LMUTEX from P_{LDR}.

Algorithm 5 Algorithm LKCS: message handlers (1).

On receipt of a $\langle \mathsf{Request}, at, ts_j, P_j \rangle$ **message:**
$pendingReq_i[at] := pendingReq_i[at] \cup \{(ts_j, P_j)\};$
if $(Cond_i)\{$
$(ts_h, P_h) := deleteMin(pendingReq_i[at]);$
$grantedTo_i[at] := grantedTo_i[at] \cup \{(ts_h, P_h)\};$
send $\langle \mathsf{Grant}, at, P_i \rangle$ **to** P_h;
$\}$ **else if** $(preemptingNow_i[at] = \textbf{nil})\{$
$(ts_h, P_h) := max(grantedTo_i[at]);$
if $((ts_j, P_j) < (ts_h, P_h))\{$
$preemptingNow_i[at] := (ts_h, P_h);$
send $\langle \mathsf{Preempt}, at, P_i \rangle$ **to** P_h;
$\}$
$\}$

On receipt of a $\langle \mathsf{Grant}, at, P_j \rangle$ **message:**
if $(P_j \notin nGrants_i[at])\{$
$nGrants_i[at] := nGrants_i[at] \cup \{P_j\};$
$\}$

On receipt of a $\langle \mathsf{Release}, at, P_j \rangle$ **message:**
if $((*, P_j) = preemptingNow_i[at])preemptingNow_i[at] := \textbf{nil};$
Delete $(*, P_j)$ from $grantedTo_i[at]$;
if $(pendingReq_i[at] \neq \emptyset)\{$
$(ts_h, P_h) := deleteMin(pendingReq_i[at]);$
$grantedTo_i[at] := grantedTo_i[at] \cup \{(ts_h, P_h)\};$
send $\langle \mathsf{Grant}, at, P_i \rangle$ **to** P_h;
$\}$

On receipt of a $\langle \mathsf{Preempt}, at, P_j \rangle$ **message:**
if $((at = \mathsf{MUTEX} \wedge state_i = \mathsf{OutCS}) \vee (at = \mathsf{MUTIN} \wedge state_i = \mathsf{InCS}))\{$
Delete P_j from $nGrants_i[at]$;
send $\langle \mathsf{Relinquish}, at, P_i \rangle$ **to** P_j;
$\}$

On receipt of a $\langle \mathsf{Relinquish}, at, P_j \rangle$ **message:**
$preemptingNow_i[at] := \textbf{nil};$
Delete $(*, P_j)$ from $grantedTo_i[at]$, and let (ts_j, P_j) be the deleted item;
$pendingReq_i[at] := pendingReq_i[at] \cup \{(ts_j, P_j)\};$
$(ts_h, P_h) := deleteMin(pendingReq_i[at]);$
$grantedTo_i[at] := grantedTo_i[at] \cup \{(ts_h, P_h)\};$
send $\langle \mathsf{Grant}, at, P_i \rangle$ **to** P_h;

- If the system is in a deadlock, P_q is definitely involved in the deadlock. Giving special grants by the sidetrack resolves the deadlock.
- If the system is not in a deadlock, P_q is not be involved in the deadlock. Furthermore, in this case, LKCS gives special grants by the sidetrack. This is because exact deadlock avoidance mechanisms require global information collection, and they incur large message complexity.

With this local observation at the leader P_{LDR}, the deadlock is avoided with less message complexity.

Algorithm 6 Algorithm LKCS: function *TriggerNomination*() for the leader P_{LDR}.

$TriggerNomination()\{$
  if $(|grantedTo_{LDR}[\mathsf{MUTIN}]| = |N_{LDR}| - L_{LDR})\{$
    if $(pendingReq_{LDR}[\mathsf{MUTEX}] \neq \emptyset)\{$
      $(ts_h, P_h) := deleteMin(pendingReq_{LDR}[\mathsf{MUTEX}]);$
    $\}$ **else**$\{$
      for-each $P_j \in Grant_{LDR}\{$
        if $((ts_j, P_j) \in grantedTo_{LDR}[\mathsf{MUTIN}] \wedge (ts_j, P_j) \in grantedTo_{LDR}[\mathsf{MUTEX}])\{$
          /* P_j may be waiting for grant messages to enter. */
          $candidate_{LDR} := candidate_{LDR} \cup \{(ts_j, P_j)\};$
        $\}$
      $\}$
      $(ts_h, P_h) := min(candidate_{LDR});$
      $candidate_{LDR} := \emptyset;$
    $\}$
    send $\langle \mathsf{Trigger}, \mathsf{MUTEX}, ts_h \rangle$ to P_h;
  $\}$ **else if** $(|grantedTo_{LDR}[\mathsf{MUTEX}]| = K_{LDR} - 1)\{$
    if $(pendingReq_{LDR}[\mathsf{MUTIN}] \neq \emptyset)\{$
      $(ts_h, P_h) := deleteMin(pendingReq_{LDR}[\mathsf{MUTIN}]);$
    $\}$ **else**$\{$
      for-each $P_j \in Grant_{LDR}\{$
        if $((ts_j + 1, P_j) \in grantedTo_{LDR}[\mathsf{MUTIN}] \wedge (ts_j, P_j) \in grantedTo_{LDR}[\mathsf{MUTEX}])\{$
          /* P_j may be waiting for grant messages to exit. */
          $candidate_{LDR} := candidate_{LDR} \cup \{(ts_j, P_j)\};$
        $\}$
      $\}$
      $(ts_h, P_h) := min(candidate_{LDR});$
      $candidate_{LDR} := \emptyset;$
    $\}$
    send $\langle \mathsf{Trigger}, \mathsf{MUTIN}, ts_h \rangle$ to P_h;
  $\}$
$\}$

Algorithm 7 Algorithm LKCS: message handlers (2).

On receipt of a $\langle \mathsf{Trigger}, at, ts \rangle$ **message:**
  if $(state_i = \mathsf{InCS} \wedge at = \mathsf{MUTIN} \wedge ts = ts_i \wedge |nGrants_i[\mathsf{MUTIN}]| < |N_i| + 1)\{$
    /* P_i is waiting for grant messages to exit. */
    $nGrants_i[\mathsf{MUTIN}] := \emptyset;$
    for-each $P_j \in (N_i \cup \{P_i\})$ **send**$\langle \mathsf{RequestByTrigger}, \mathsf{MUTIN}, P_i \rangle$ **to** P_j;
    /* Request message as a trigger. */
  $\}$ **else if** $(state_i = \mathsf{OutCS} \wedge at = \mathsf{MUTEX} \wedge ts = ts_i \wedge |nGrants_i[\mathsf{MUTEX}]| < |N_i| + 1)\{$
    /* P_i is waiting for grant messages to enter. */
    $nGrants_i[\mathsf{MUTEX}] := \emptyset;$
    for-each $P_j \in (N_i \cup \{P_i\})$ **send**$\langle \mathsf{RequestByTrigger}, \mathsf{MUTEX}, P_i \rangle$ **to** P_j;
    /* Request message as a trigger. */
  $\}$

On receipt of a $\langle \mathsf{RequestByTrigger}, at, P_j \rangle$ **message:**
  Delete $(*, P_j)$ from $pendingReq_i[at]$;
  Delete $(*, P_j)$ from $grantedTo_i[at]$;
  $grantedTo_i[at] := grantedTo_i[at] \cup \{(0, P_j)\};$
  send $\langle \mathsf{Grant}, at, P_i \rangle$ **to** P_j;

In the proposed algorithm, each process P_i maintains the following local variables, where *at* is the algorithm type, MUTEX or MUTIN. These variables work as the same as those of l_i-LMUTIN, and we omit the detailed description here.

- $state_i$: The current state of P_i: InCS or OutCS.
- ts_i: The current value of the logical clock [19].
- $nGrants_i[at]$: A set of process ids from which P_i obtains grants for exiting/entering to the CS.
- $grantedTo_i[at]$: A set of timestamps (ts_j, P_j) for the requests to P_j's exiting/entering to the CS that P_i has been granted but that P_j has not yet released.
- $pendingReq_i[at]$: A set of timestamps (ts_j, P_j) for the requests to P_j's exiting/entering to the CS that are pending.
- $preemptingNow_i[at]$: A timestamp (ts_j, P_j) of a request such that P_i preempts a permission for P_j's exiting/entering to the CS if the preemption is in progress.

5.3. Proof of Correctness

In this subsection, we show the correctness of LKCS. We assume the initial configuration is safe. First, we show that the process P_i with $dist(P_{LDR}, P_i) > 2$ cannot become unsafe by the proof by contradiction. Next, we show that other processes P_j cannot become unsafe because they normally execute the algorithm as their instance $(l_j + 1, k_j - 1)$. Thus, we can derive the following lemma.

Lemma 4. *(Safety) For each process $P_i \in V$, $l_i \leq |\mathcal{CS}_i(C)| \leq k_i$ holds at any configuration C.*

Proof. We assume that the initial configuration C_0 is safe. First, we consider the process P_i for which $dist(P_{LDR}, P_i) > 2$ holds becomes unsafe first in a configuration C for the contrary.

- Suppose that $|\mathcal{CS}_i(C)| < l_i$, that is $|\overline{\mathcal{CS}_i(C)}| > |N_i| + 1 - l_i$. Because $|\mathcal{CS}_i(C_0)| \geq l_i$, consider a process $P_j \in N_i \cup \{P_i\}$, which became the OutCS state by the $(|N_i| + 2 - l_i)$-th lowest timestamp among processes in $\overline{\mathcal{CS}_i(C)}$. Then, P_j obtains permission to be the OutCS state from each process in $N_j \cup \{P_j\}$. This implies that P_i receives a permission request $\langle \mathsf{Request}, \mathsf{MUTIN}, ts_j, P_j \rangle$ from P_j and that P_i sends a permission $\langle \mathsf{Grant}, \mathsf{MUTIN}, P_i \rangle$ to P_j. Because P_i grants at most $|N_i| + 1 - l_i$ permissions to exit the CS at each time by the condition $Cond_i$, P_j cannot obtain a permission from P_i; this is a contradiction.
- Suppose that $|\mathcal{CS}_i(C)| > k_i$. Because $|\mathcal{CS}_i(C_0)| \leq k_i$, consider a process $P_j \in N_i \cup \{P_i\}$ that became the InCS state by the $(k_i + 1)$-th lowest timestamp among processes in $\mathcal{CS}_i(C)$. Then, P_j obtains permission to be the InCS state from each process in $N_j \cup \{P_j\}$. This implies that P_i receives a permission request $\langle \mathsf{Request}, \mathsf{MUTEX}, ts_j, P_j \rangle$ from P_j and that P_i sends a permission $\langle \mathsf{Grant}, \mathsf{MUTEX}, P_i \rangle$ to P_j. Because P_i grants at most k_i permissions to enter the CS at each time by the condition $Cond_i$, P_j cannot obtain a permission from P_i; this is a contradiction.

Next, we consider P_i that has $dist(P_{LDR}, P_i) \leq 2$. Note that the leader P_{LDR} sends a trigger request $\langle \mathsf{Trigger}, at, ts \rangle$ to exactly one of its neighbors or itself at a time. Let P_q be the receiver. If P_q does not request to invert its state as a trigger, we can discuss by the same way as above, and $l_i + 1 \leq |\mathcal{CS}_i(C)| \leq k_i - 1$ because of condition $Cond_i$ (of course, if $P_i = P_{LDR}$, $l_{LDR} + 1 \leq |\mathcal{CS}_{LDR}(C)| \leq k_{LDR} - 1$). When P_q requests to invert its state as a trigger by sending a message $\langle \mathsf{RequestByTrigger}, at, P_q \rangle$, all of its neighbors P_j grant it without attention to $|\mathcal{CS}_j(C)|$, and P_q inverts its state without attention to $|\mathcal{CS}_q(C)|$. Thus, $|\mathcal{CS}_i(C)|$ becomes $l_i + 1 - 1$ or $k_i - 1 + 1$ (if $P_i = P_{LDR}$, $|\mathcal{CS}_{LDR}(C)|$ becomes $l_{LDR} + 1 - 1$ or $k_{LDR} - 1 + 1$). Therefore, P_i does not become unsafe. □

Next, we consider that a deadlock occurs in a configuration. Then, processes waiting for grant messages constitute chains of deadlocks unless at least one process on chains changes its state. However, then, the leader process designates one of its neighbors or itself as a trigger, and the trigger changes its state by its preferential rights. Therefore, we can derive the following lemma.

Lemma 5. *(Liveness) Each process P_i changes into the* InCS *and* OutCS *states alternately infinitely often.*

Proof. For the contrary, we assume that a deadlock occurs in the configuration C. Let D be a set of processes that cannot change its state, that is they are in the deadlock. First, we assume that all of process P_u in D has $state_u = \mathsf{OutCS}$. Then, all of their neighbors P_v have k_v neighbors P_w with $state_w = \mathsf{InCS}$. However, such neighbors P_w are not in D and eventually change their state to OutCS. Thus, P_u eventually can change its state; this is a contradiction. Therefore, in D, there is a process P_u with $state_u = \mathsf{InCS}$ such that it cannot change its state by the exit-sequence. Then, it holds $|\mathcal{CS}_u(C)| = l_u$ or has a neighbor $P_v \in N_u$ with $|\mathcal{CS}_v(C)| = l_v$. It is waiting for grant messages and cannot change its state until at least one of its neighbors $P_w \in N_u$ with $state_w = \mathsf{OutCS}$ changes P_w's state by the entry-sequence. If $|\mathcal{CS}_w(C)| = k_w$ holds or P_w has a neighbor $P_x \in N_w$ with $|\mathcal{CS}_x(C)| = k_x$, P_w cannot change its state by the entry-sequence until at least one of its neighbors $P_y \in N_w$ with $state_y = \mathsf{InCS}$ changes P_y's state by the exit-sequence. By such chain relationship, it is clear that the waiting chain can be broken if at least one process on this chain changes its state. Thus, for the assumption, all processes in V are in such a chain in C, that is $V = D$.

However, in such a C, all of P_{LDR} and its neighbors are waiting for grant messages from their neighbors. That is, their requests are in $grantedTo_{LDR}$ or $pendingReq_{LDR}$, and $Waiting_{LDR}$ is equal to $|N_{LDR}| + 1$. Additionally, we assume that $k_{LDR} - l_{LDR} \geq 3$ and the number of grants P_{LDR} can send with MUTIN (resp., MUTEX) is $|N_{LDR}| + 1 - (l_{LDR} + 1)$ (resp., $k_{LDR} - 1 \geq l_{LDR} + 2$). Because of safety, $|grantedTo_{LDR}[\mathsf{MUTIN}]| = |N_{LDR}| - l_{LDR}$ or $|grantedTo_{LDR}[\mathsf{MUTEX}]| = k_{LDR} - 1$ holds. Then, P_{LDR} sends a $\langle \mathsf{Trigger}, at, ts \rangle$ message to a neighbor P_q and P_q becomes a trigger if P_q is also waiting for grant messages. Processes P_r that receive $\langle \mathsf{RequestByTrigger}, at, P_q \rangle$ grant the request without attention to $|\mathcal{CS}_r(C)|$. Then, P_q can change its state, and after that, P_{LDR} can change its state. Therefore, the waiting chain in C can be broken. This is a contradiction. □

5.4. Performance Analysis

Lemma 6. *The message complexity of LKCS for $P_i \in V$ is $6(|N_i| + 1)$ in the best case, and $12(|N_i| + 1)$ in the worst case.*

Proof. First, let us consider the best case.

- For P_i's exiting the CS, P_i sends a $\langle \mathsf{Request}, \mathsf{MUTIN}, ts_i, P_i \rangle$ message to each process in $N_i \cup \{P_i\}$; each process P_j in $N_i \cup \{P_i\}$ sends a $\langle \mathsf{Grant}, \mathsf{MUTIN}, P_j \rangle$ message to P_i; then P_i sends a $\langle \mathsf{Release}, \mathsf{MUTEX}, P_i \rangle$ message to each process in $N_i \cup \{P_i\}$. Thus, $3(|N_i| + 1)$ messages are exchanged for P_i's exiting the CS.
- For P_i's entering the CS, P_i sends a $\langle \mathsf{Request}, \mathsf{MUTEX}, ts_i, P_i \rangle$ message to each process in $N_i \cup \{P_i\}$; each process P_j in $N_i \cup \{P_i\}$ sends a $\langle \mathsf{Grant}, \mathsf{MUTEX}, P_j \rangle$ message to P_i; then P_i sends a $\langle \mathsf{Release}, \mathsf{MUTIN}, P_i \rangle$ message to each process in $N_i \cup \{P_i\}$. Thus, $3(|N_i| + 1)$ messages are exchanged for P_i's entering the CS.

Thus, the message complexity is $6(|N_i| + 1)$ in the best case.
Next, let us consider the worst case.

- For P_i's exiting the CS, P_i sends a $\langle \mathsf{Request}, \mathsf{MUTIN}, ts_i, P_i \rangle$ message to each process P_j in $N_i \cup \{P_i\}$. Then, P_j sends a $\langle \mathsf{Preempt}, \mathsf{MUTIN}, P_j \rangle$ message to the process P_m to which P_j sends a $\langle \mathsf{Grant}, \mathsf{MUTIN}, P_j \rangle$ message; P_m sends a $\langle \mathsf{Relinquish}, \mathsf{MUTIN}, P_m \rangle$ message back to P_j; and P_j sends a $\langle \mathsf{Grant}, \mathsf{MUTIN}, P_j \rangle$ message to P_i. After P_i exits the CS, P_i sends a $\langle \mathsf{Release}, \mathsf{MUTEX}, P_i \rangle$ message to each process P_j in $N_i \cup \{P_i\}$. Then, P_j sends a $\langle \mathsf{Grant}, \mathsf{MUTEX}, P_j \rangle$ message to some process with the highest priority in $pendingReq_j[\mathsf{MUTEX}]$. Thus, $6(|N_i| + 1)$ messages are exchanged.
- For P_i's entering the CS, P_i sends a $\langle \mathsf{Request}, \mathsf{MUTEX}, ts_i, P_i \rangle$ message to each process P_j in $N_i \cup \{P_i\}$. Then, P_j sends a $\langle \mathsf{Preempt}, \mathsf{MUTEX}, P_j \rangle$ message to the process P_m to which P_j sends a $\langle \mathsf{Grant}, \mathsf{MUTEX}, P_j \rangle$ message; P_m sends a $\langle \mathsf{Relinquish}, \mathsf{MUTEX}, P_m \rangle$ message back to P_j; and P_j

sends a $\langle \mathsf{Grant}, \mathsf{MUTEX}, P_j \rangle$ message to P_i. After P_i enters the CS, P_i sends a $\langle \mathsf{Release}, \mathsf{MUTIN}, P_i \rangle$ message to each process P_j in $N_i \cup \{P_i\}$. Then, P_j sends a $\langle \mathsf{Grant}, \mathsf{MUTIN}, P_j \rangle$ message to some process with the highest priority in $pendingReq_j[\mathsf{MUTIN}]$. Thus, $6(|N_i| + 1)$ messages are exchanged.

Thus, the message complexity is $12(|N_i| + 1)$ in the worst case. □

Theorem 3. *LKCS solves the generalized local (l_i, k_i)-critical section problem with a message complexity of $O(\Delta)$, where Δ is the maximum degree of a network.*

6. Conclusions

In this paper, we consider the generalized local (l_i, k_i)-critical section problem, which is a new version of critical section problems. Because this problem is useful for fault-tolerance and load balancing of distributed systems, we can consider various future applications. We first proposed an algorithm for the generalized local l_i-mutual inclusion. Next, we showed the generalized local complementary theorem. By using this theorem, we proposed an algorithm for the generalized local (l_i, k_i)-critical section problem.

In the future, we plan to perform extensive simulations and confirm the performance of our algorithms under various application scenarios. Additionally, we plan to improve the proposed algorithm in message complexity and time complexity and to design an algorithm that guarantees exactly $l_i \leq |\mathcal{CS}_i(C)| \leq k_i$ in every process.

Acknowledgments: This work was supported by JSPS KAKENHI Grant Numbers 16K00018 and 26330015.

Author Contributions: Sayaka Kamei and Hirotsugu Kakugawa designed the algorithms. Sayaka Kamei analyzed the algorithms and wrote the paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Dijkstra, E.W. Solution of a problem in concurrent programming control. *Commun. ACM* **1965**, *8*, 569.
2. Saxena, P.C.; Rai, J. A survey of permission-based distributed mutual exclusion algorithms. *Comput. Stand. Interfaces* **2003**, *25*, 159–181.
3. Yadav, N.; Yadav, S.; Mandiratta, S. A review of various mutual exclusion algorithms in distributed environment. *Int. J. Comput. Appl.* **2015**, *129*, 11–16.
4. Kakugawa, H.; Fujita, S.; Yamashita, M.; Ae, T. Availability of k-coterie. *IEEE Trans. Comput.* **1993**, *42*, 553–558.
5. Bulgannawar, S.; Vaidya, N.H. A distributed *k*-mutual exclusion algorithm. In Proceedings of the 15th International Conference on Distributed Computing Systems, Vancouver, BC, Canada, 30 May–2 June 1995; pp. 153–160.
6. Chang, Y.I.; Chen, B.H. A generalized grid quorum strategy for *k*-mutual exclusion in distributed systems. *Inf. Process. Lett.* **2001**, *80*, 205–212.
7. Abraham, U.; Dolev, S.; Herman, T.; Koll, I. Self-stabilizing l-exclusion. *Theor. Comput. Sci.* **2001**, *266*, 653–692.
8. Chaudhuri, P.; Edward, T. An algorithm for *k*-mutual exclusion in decentralized systems. *Comput. Commun.* **2008**, *31*, 3223–3235.
9. Reddy, V.A.; Mittal, P.; Gupta, I. Fair k mutual exclusion algorithm for peer to peer systems. In Proceedings of the 28th International Conference on Distributed Computing Systems, Beijing, China, 17–20 June 2008.
10. Hoogerwoord, R.R. An implementation of mutual inclusion. *Inf. Process. Lett.* **1986**, *23*, 77–80.
11. Kakugawa, H. Mutual inclusion in asynchronous message-passing distributed systems. *J. Parallel Distrib. Comput.* **2015**, *77*, 95–104.
12. Kakugawa, H. On the family of critical section problems. *Inf. Process. Lett.* **2015**, *115*, 28–32.
13. Lynch, N.A. Fast allocation of nearby resources in a distributed system. In Proceedings of the 12th Annual ACM Symposium on Theory of Computing, Los Angeles, CA, USA, 28–30 April 1980; pp. 70–81.

14. Attiya, H.; Kogan, A.; Welch, J.L. Efficient and robust local mutual exclusion in mobile ad hoc networks. *IEEE Trans. Mobile Comput.* **2010**, *9*, 361–375.
15. Awerbuch, B.; Saks, M. A dining philosophers algorithm with polynomial response time. In Proceedings of the 31th Annual Symposium on Foundations of Computer Science, St. Louis, MO, USA, 22–24 October 1990; Volume 1, pp. 65–74.
16. Chandy, M.; Misra, J. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.* **1984**, *6*, 632–646.
17. Beauquier, J.; Datta, A.K.; Gradinariu, M.; Magniette, F. Self-stabilizing local mutual exclusion and daemon refinement. In Proceedings of the International Symposium on Distributed Computing, Toledo, Spain, 4–6 October 2000; pp. 223–237.
18. Khanna, A.; Singh, A.K.; Swaroop, A. A leader-based k-local mutual exclusion algorithm using token for MANETs. *J. Inf. Sci. Eng.* **2014**, *30*, 1303–1319.
19. Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **1978**, *21*, 558–565.
20. Maekawa, M. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.* **1985**, *3*, 145–159.
21. Sanders, B.A. The information structure of mutual exclusion algorithm. *ACM Trans. Comput. Syst.* **1987**, *5*, 284–299.