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.

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 \( 0 \leq l_i < k_i \leq |N_i| + 1 \) 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 \( 0 \leq l < k \leq n \), 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 ≤ l_i < k_i ≤ |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 ≤ l_i < k_i ≤ |N_i| + 1 for each process P_i. We assume that there is a process P_{LDR}, such that |N_{LDR}| ≥ 4, l_{LDR} ≤ k_{LDR} − 3 and l_q ≤ k_q − 3 for each P_q ∈ 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 ⊆ V × 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 | (P_i, P_j) ∈ 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 $\text{state}_i \in \{\text{lnCS}, \text{OutCS}\}$. For each configuration $C$, let $\mathcal{CS}(C)$ (resp., $\overline{\mathcal{CS}(C)}$) be the set of processes $P_i$ with $\text{state}_i = \text{lnCS}$ (resp., $\text{state}_i = \text{OutCS}$) in $C$. For each configuration $C$ and each process $P_i$, let $\overline{\mathcal{CS}_i(C)}$ (resp., $\mathcal{CS}_i(C)$) 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 $\text{OutCS}$ state, and $P_i$ eventually invokes exit-sequence when it is in the $\text{lnCS}$ state.

\[
\text{state}_i := (\text{Initial state of } P_i \text{ in the initial configuration } C_0);
\]
\[
\text{while true\{}
  \text{if}(\text{state}_i = \text{OutCS})\{ \\
    \text{Entry-Sequence}; \\
    \text{state}_i := \text{lnCS}; \\
    /* \text{Critical Section} */ \\
  \}
  \text{if}(\text{state}_i = \text{lnCS})\{ \\
    \text{Exit-Sequence}; \\
    \text{state}_i := \text{OutCS}; \\
    /* \text{Remainder Section} */ \\
  \}
\}
\]

**Definition 1.** (The generalized local critical section problem) Assume that a pair of numbers $l_i$ and $k_i$ $(0 \leq l_i < k_i \leq |N_i| + 1)$ 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 $P_i \in V$, $l_i \leq |\mathcal{CS}_i(C)| \leq k_i$ at any time.
- **Liveness:** Each process $P_i \in V$ changes $\text{OutCS}$ and $\text{lnCS}$ 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 $l_i \leq |\mathcal{CS}_i(C_0)| \leq k_i$. In the case of $(l_i, k_i) = (0, 1)$ (resp., $(1, |N_i| + 1)$), the initial state of each process can be $\text{OutCS}$ (resp., $\text{lnCS}$) 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 $\text{OutCS}$ 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 $\text{lnCS}$ state initially, and some algorithms (e.g., token-based algorithms) assume that exactly one process is in the $\text{lnCS}$ state and other processes are in the $\text{OutCS}$ 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 $I_{l_i}$-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 $l_i \leq |CS_i(C_0)| \leq |N_i| + 1$. When $P_i$ wishes to be in the OutCS state, $P_i$ requests permission by sending a $\langle$Request, $ts_i$, $P_j$$\rangle$ message for each process in $N_i \cup \{P_i\}$. When $P_i$ obtains a permission by receiving a $\langle$Grant$\rangle$ message from each process in $N_i \cup \{P_i\}$, $P_i$ changes to the OutCS state. Each process $P_j$ grants permissions to $|N_j| - l_j + 1$ processes at each time. Hence, at least $l_j$ processes in $N_j \cup \{P_j\}$ cannot be in the OutCS state at the same time. When $P_i$ wishes to be in the InCS state, $P_i$ changes to the InCS state and sends a message $\langle$Release, $P_i$$\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 (Preempt, $P_i$) and (Relinquish, $P_i$) 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 $P_i$'s exiting the CS that $P_i$ has been granted, but that $P_i$ has not yet released.
- $pendingReq_i$: A set of timestamps ($ts_j$, $P_j$) for the requests to $P_i$'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: ($ts_j$, $P_j$) < ($ts_k$, $P_k$) iff $ts_j < ts_k$ or ($ts_j = ts_k$) and ($P_i < P_j$).

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.

<table>
<thead>
<tr>
<th>Algorithm 1</th>
<th>Local variables for process $P_i$ in algorithm $I_{l_i}$-LMUTIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>$state_i \in {\text{InCS, OutCS}}$, initially $state_i = \left{ \begin{array}{ll} \text{InCS} &amp; (P_i \in CS(C_0)) \ \text{OutCS} &amp; (P_i \notin CS(C_0)) \end{array} \right.$</td>
<td></td>
</tr>
<tr>
<td>$ts_i$: integer, initially 0;</td>
<td></td>
</tr>
<tr>
<td>$nGrants_i$: integer, initially 0;</td>
<td></td>
</tr>
<tr>
<td>$grantedTo_i$: set of (integer, processID), initially ${(0, P_i \in CS(C_0))}$;</td>
<td></td>
</tr>
<tr>
<td>$pendingReq_i$: set of (integer, processID), initially $\emptyset$;</td>
<td></td>
</tr>
<tr>
<td>$preemptingNow_i$: processID, initially nil;</td>
<td></td>
</tr>
</tbody>
</table>
Algorithm 2 Algorithm $l_l$-LMUTIN: exit-sequence, entry-sequence and message handlers.

Exit-Sequence:

\begin{itemize}
\item $ts_j := ts_j + 1;
\item $nGrants_j := 0;
\item \textbf{for-each} $P_j \in (N_j \cup \{P_j\})$ \textbf{send} (Request, $ts_j$, $P_j$) to $P_j$;
\item \textbf{wait until} ($nGrants_j = |N_j| + 1$);
\item $state_j := \text{OutCS};$
\end{itemize}

Entry-Sequence:

\begin{itemize}
\item $state_j := \text{InCS}$;
\item \textbf{for-each} $P_j \in (N_j \cup \{P_j\})$ \textbf{send} (Release, $P_j$) to $P_j$;
\end{itemize}

On receipt of a (Request, $ts_j$, $P_j$) message:

\begin{itemize}
\item $\text{pendingReq}_j := \text{pendingReq}_j \cup \{(ts_j, P_j)\};$
\item \textbf{if} ($|\text{grantedTo}_j| < |N_j| - l_j + 1$) { 
\begin{itemize}
\item $(ts_h, P_h) := \text{deleteMin}(\text{pendingReq}_j);$
\item $\text{grantedTo}_j := \text{grantedTo}_j \cup \{(ts_h, P_h)\};$
\item \textbf{send} (Grant) to $P_h$;
\end{itemize}
\item \textbf{else if} ($\text{preemptingNow}_j = \text{nil}$) { 
\begin{itemize}
\item $(ts_h, P_h) := \max(\text{grantedTo}_j);$
\item \textbf{if} ($(ts_j, P_j) < (ts_h, P_h))$ { 
\begin{itemize}
\item $\text{preemptingNow}_j := P_h$;
\item \textbf{send} (Preempt, $P_j$) to $P_h$;
\end{itemize}
\end{itemize}
\}
\end{itemize}

On receipt of a (Grant) message:

$nGrants_j := nGrants_j + 1;$

On receipt of a (Release, $P_j$) message:

\begin{itemize}
\item Delete $(*, P_j)$ from $\text{grantedTo}_j$;
\item \textbf{if} ($\text{pendingReq}_j \neq \emptyset$) { 
\begin{itemize}
\item $(ts_h, P_h) := \text{deleteMin}(\text{pendingReq}_j);$
\item $\text{grantedTo}_j := \text{grantedTo}_j \cup \{(ts_h, P_h)\};$
\item \textbf{send} (Grant) to $P_h$;
\end{itemize}
\}
\end{itemize}

On receipt of a (Preempt, $P_j$) message:

\begin{itemize}
\item \textbf{if} ($state_j = \text{InCS}$) { 
\begin{itemize}
\item $nGrants_j := nGrants_j - 1;$
\item \textbf{send} (Relinquish, $P_j$) to $P_j$;
\end{itemize}
\}
\end{itemize}

On receipt of a (Relinquish, $P_j$) message:

\begin{itemize}
\item $\text{preemptingNow}_j := \text{nil};$
\item Delete $(*, P_j)$ from $\text{grantedTo}_j$, and let $(ts_j, P_j)$ be the deleted item;
\item $\text{pendingReq}_j := \text{pendingReq}_j \cup \{(ts_j, P_j)\};$
\item $(ts_h, P_h) := \text{deleteMin}(\text{pendingReq}_j);$
\item $\text{grantedTo}_j := \text{grantedTo}_j \cup \{(ts_h, P_h)\};$
\item \textbf{send} (Grant) to $P_h$;
\end{itemize}

3.1. Proof of Correctness

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

Lemma 1. (Safety) For each process $P_i \in V, l_i \leq |CS_i(C)| \leq |N_i| + 1$ holds at any configuration $C$. 
The Generalized Local Complementary Theorem

Proof. We assume that the initial configuration $C_0$ is safe, i.e., $l_i \leq |CS_i(C_0)|$. Therefore, we consider the process $P_i$, which becomes unsafe first for the contrary. Suppose that $|CS_i(C)| < l_i$, that is $|CS_i(C)| > |N_i| + 1 - l_i$. Because $|CS_i(C_0)| \geq l_i$, 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 $CS_i(C)$. Then, $P_j$ obtains permission to be the OutCS state from each process in $N_i \cup \{P_i\}$. This implies that $P_i$ receives a request $\langle \text{Request}, ts_i, P_i \rangle$ from $P_j$ and that $P_i$ sends a permission $\langle \text{Grant} \rangle$ 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 $\langle \text{Request}, ts_i, P_i \rangle$ message from $P_i$, the value of $(ts_i, P_i)$ exceeds the timestamp $(ts_i, P_i)$ for $P_j$'s request. Because, by this algorithm, the request with the lowest timestamp is granted preferentially, it is impossible for $P_i$ to change into the InCS and OutCS states alternately infinitely often. Then, $P_i$ eventually sends a $\langle \text{Grant} \rangle$ message to $P_j$, and $P_j$ eventually sends a $\langle \text{Grant} \rangle$ message to itself.

- Suppose that $P_j$ does not change into the InCS and OutCS states alternately infinitely often. Because the timestamp of $P_j$ is smaller than that of $P_i$, by assumption, $P_j$'s permission is preempted, and a $\langle \text{Grant} \rangle$ message is sent from $P_j$ to $P_i$. In addition, $P_i$ sends a $\langle \text{Grant} \rangle$ message to itself.

Therefore, $P_i$ eventually receives a $\langle \text{Grant} \rangle$ message from each process in $N_i \cup \{P_j\}$, 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 $l_i$-LMUTIN for $P_i \in V$ 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 $\langle \text{Request}, ts_i, P_i \rangle$ message to each process in $N_i \cup \{P_i\}$; each process in $N_i \cup \{P_i\}$ sends a $\langle \text{Grant} \rangle$ message to $P_i$. In entry-sequence, after $P_i$'s entering the CS, $P_i$ sends a $\langle \text{Release}, P_i \rangle$ 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 $\langle \text{Request}, ts_i, P_i \rangle$ message to each process $P_j$ in $N_i \cup \{P_i\}$. Then, $P_j$ sends a $\langle \text{Preempt}, P_j \rangle$ message to the process $P_m$ to which $P_j$ sends a $\langle \text{Grant} \rangle$ message, $P_m$ sends a $\langle \text{Relinquish}, P_m \rangle$ message back to $P_j$ and $P_j$ sends a $\langle \text{Grant} \rangle$ message to $P_j$. After $P_j$'s entering the CS, $P_j$ sends a $\langle \text{Release}, P_j \rangle$ message to each process $P_j$ in $N_i \cup \{P_i\}$. Then, $P_j$ sends a $\langle \text{Grant} \rangle$ message to return a grant to $P_m$ or grant to some process with the highest priority in $\text{pendingReq}_j$. 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 $\Delta$ 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 $A^G_{(l,k)}$ be an algorithm for the global $(l,k)$-CS problem, and $A^L_{(l,k)}$ be an algorithm for the generalized local $(l,k)$-CS problem. By Co-$A^G_{(l,k)}$ (resp., Co-$A^L_{(l,k)}$), we denote a complement algorithm of $A^G_{(l,k)}$ (resp., $A^L_{(l,k)}$), which is obtained by swapping the process states, InCS and OutCS.

In [12], it is shown that the complement of $A^G_{(l,k)}$ 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 $k_i$ ($0 \leq l_i < k_i \leq |N_i| + 1$) is given. Then, Co-$A^L_{(l_i,k_i)}$ is an algorithm for the generalized local $(|N_i| + 1 - k_i, |N_i| + 1 - l_i)$-CS problem.

**Proof.** By $A^L_{(l_i,k_i)}$, at least $l_i$ and at most $k_i$ processes among each process and its neighbors are in the CS. Hence, by Co-$A^L_{(l_i,k_i)}$, 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 LKCS 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, 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 $state_u = \text{InCS}$ such that $|CS_u(C)| = l_u$ or $P_u$ has a neighbor $P_v \in N_u$ with $|CS_v(C)| = l_v$. Then, $P_u$ cannot change its state by exit-sequence until at least one of its neighbors $P_w \in N_u$ with $state_w = \text{OutCS}$ changes $P_w$’s state by entry-sequence. If $|CS_w(C)| = k_w$ or $P_w$ has a neighbor $P_x \in N_w$ with $|CS_x(C)| = k_x$, $P_w$ cannot change its state by entry-sequence until at least one
of its neighbors $P_q \in N_w$ with $state_q = \text{InCS}$ changes $P_q$’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 $P_{\text{LDR}}$, with $|N_{\text{LDR}}| \geq 4$ 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 $P_{\text{LDR}}$ is not always correct, i.e., $P_{\text{LDR}}$ 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 \{MUTEX, MUTIN\};
- state_i \in \{\text{InCS, OutCS}\}, initially state_i = \{ \text{InCS} \ (P_i \in CS(C_0)) \text{OutCS} \ (P_i \notin CS(C_0)) \}.
- ts_i : \text{integer}, initially 1;
- nGrants_i[\text{at}]: \text{set of processID, initially } \varnothing ;
- grantedTo_i[\text{at}]: \text{set of (integer, processID), initially } \{ (1, P_j \in CS_i(C_0)) \} ;
- pendingReq_i[\text{at}]: \text{set of (integer, processID), initially } \varnothing ;
- preemptingNow_i[\text{at}]: (\text{integer, processID), initially nil} ;

**Local Variable only for a leader $P_{\text{LDR}}$:**

- candidateLDR: \text{set of (integer, processID), initially } \varnothing ;

**Macros:**

$L_i \equiv \{ l_i \ \text{dist}(P_{\text{LDR}}, P_i) > 2 \}$

$L_i + 1 \text{dist}(P_{\text{LDR}}, P_i) \leq 2 \}

$K_i \equiv \{ k_i \ \text{dist}(P_{\text{LDR}}, P_i) > 2 \}$

$K_i - 1 \text{dist}(P_{\text{LDR}}, P_i) \leq 2 \}

Grant_{LDR} \equiv \{ P_j | (*, P_j) \in grantedTo_{LDR}[\text{MUTIN}] \land (*, P_j) \in grantedTo_{LDR}[\text{MUTEX}] \}$

Waiting_{LDR} \equiv |L_i| + \text{pendingReq}_{LDR}[\text{MUTIN}] + \text{pendingReq}_{LDR}[\text{MUTEX}] + |Grant_{LDR}|

Cond_i \equiv \{ at = MUTEX \land |grantedTo_i[\text{MUTEX}]| < K_i \lor \ (at = MUTIN \land |grantedTo_i[\text{MUTIN}]| < |N_i| - L_i + 1) \}$
Algorithm 4: Algorithm LKCS: exit-sequence and entry-sequence.

Exit-Sequence:

\( ts_i := ts_i + 1; \)
\n\( nGrants_i[MUTIN] := 0; \)
\nfor-each \( P_j \in (N_i \cup \{P_i\}) \) send \{Request, MUTIN, \( ts_i, P_j \)\} to \( P_j \);
\nwhile \((\{nGrants_i[MUTIN]\} < |N_i| + 1)\) {
  if \((P_i = P_{LDR} \land Waiting_{LDR} = |N_{LDR}| + 1)\) {
    \( \text{// The configuration may be in a deadlock.} /\)
    TriggerNomination();
    wait until \((Waiting_{LDR} < |N_{LDR}| + 1)\);
  }
  \( state_i := \text{OutCS}; \)
  for-each \( P_j \in (N_i \cup \{P_i\}) \) send \{Release, MUTEX, \( P_i \)\} to \( P_j \);
}
\nEntry-Sequence:

\( nGrants_i[MUTEX] := 0; \)
\nfor-each \( P_j \in (N_i \cup \{P_i\}) \) send \{Request, MUTEX, \( ts_i, P_j \)\} to \( P_j \);
\nwhile \((\{nGrants_i[MUTEX]\} < |N_i| + 1)\) {
  if \((P_i = P_{LDR} \land Waiting_{LDR} = |N_{LDR}| + 1)\) {
    \( \text{// The configuration may be in a deadlock.} /\)
    TriggerNomination();
    wait until \((Waiting_{LDR} < |N_{LDR}| + 1)\);
  }
  \( state_i := \text{InCS}; \)
  for-each \( P_j \in (N_i \cup \{P_i\}) \) send \{Release, MUTIN, \( P_i \)\} 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 (\( pending Req \)) and its granted list (\( granted Req \)), 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 \( P_q \) or other neighbors of \( P_q \) keep 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 \( Waiting_{LDR} = \mid N_{LDR} \mid + 1 \), then \( \mid pending Req_{LDR}[MUTIN] \mid + \mid pending Req_{LDR}[MUTEX] \mid + \mid Grant_{LDR} \mid = \mid N_{LDR} \mid + 1 \). Because a request is not sent if a previous one is kept waiting, two pending lists \( pending Req_{LDR}[MUTIN] \) and \( pending Req_{LDR}[MUTEX] \) 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 \( granted Req_{LDR} \). 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 \( P_{LDR} \).
On receipt of a \langle Request, at, ts_j, P_j \rangle message:

\( \text{pendingReq}_i[ at ] := \text{pendingReq}_i[ at ] \cup \{ (ts_j, P_j) \};\)

if \( \text{Cond}_i \):

\[ (ts_h, P_h) := \text{deleteMin}(\text{pendingReq}_i[ at ]); \]

\( \text{grantedTo}_i[ at ] := \text{grantedTo}_i[ at ] \cup \{ (ts_h, P_h) \}; \)

send \langle Grant, at, P_i \rangle to P_h;

} else if \( \langle preemptingNow_i[ at ] = \text{nil} \rangle \}{

\( (ts_h, P_h) := \text{max}(\text{grantedTo}_i[ at ]); \)

if \( ((ts_j, P_j) < (ts_h, P_h)) \}{

\( \text{preemptingNow}_i[ at ] := (ts_h, P_h); \)

send \langle Preempt, at, P_i \rangle to P_h;

}

On receipt of a \langle Grant, at, P_i \rangle message:

if \( (P_i \notin nGrants_i[ at ]); \)

\[ nGrants_i[ at ] := nGrants_i[ at ] \cup \{ P_i \}; \]

On receipt of a \langle Release, at, P_i \rangle message:

if \( (\bullet, P_i) = \text{preemptingNow}_i[ at ]; \)

\( \text{preemptingNow}_i[ at ] := \text{nil}; \)

Delete \( (\bullet, P_i) \) from \( \text{grantedTo}_i[ at ]; \)

if \( (\text{pendingReq}_i[ at ] \neq \emptyset) \}{

\( (ts_h, P_h) := \text{deleteMin}(\text{pendingReq}_i[ at ]); \)

\( \text{grantedTo}_i[ at ] := \text{grantedTo}_i[ at ] \cup \{ (ts_h, P_h) \}; \)

send \langle Grant, at, P_i \rangle to P_h;

}

On receipt of a \langle Preempt, at, P_i \rangle message:

if \( (((\text{at} = \text{MUTEX} \land \text{state}_i = \text{OutCS}) \lor (\text{at} = \text{MUTIN} \land \text{state}_i = \text{InCS})) \}{

Delete \( P_i \) from \( nGrants_i[ at ]; \)

send \langle Relinquish, at, P_i \rangle to P_j;

}

On receipt of a \langle Relinquish, at, P_i \rangle message:

\( \text{preemptingNow}_i[ at ] := \text{nil}; \)

Delete \( (\bullet, P_i) \) from \( \text{grantedTo}_i[ at ]; \), and let \( (ts_j, P_j) \) be the deleted item;

\( \text{pendingReq}_i[ at ] := \text{pendingReq}_i[ at ] \cup \{ (ts_j, P_j) \}; \)

\( (ts_h, P_h) := \text{deleteMin}(\text{pendingReq}_i[ at ]); \)

\( \text{grantedTo}_i[ at ] := \text{grantedTo}_i[ at ] \cup \{ (ts_h, P_h) \}; \)

send \langle 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$_{LDR}$[MUTIN]| = |$N_{LDR}$| - $L_{LDR}$) {
    if (pendingReq$_{LDR}$[MUTEX] $\neq \emptyset$) {
      (ts$_{h}, P_{h}$) := deleteMin(pendingReq$_{LDR}$[MUTEX]);
    } else {
      for-each $P_{j} \in$ Grant$_{LDR}$ {
        if ((ts$_{j}, P_{j}$) \in grantedTo$_{LDR}$[MUTIN] \land (ts$_{j}, P_{j}$) \in grantedTo$_{LDR}$[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 (Trigger, MUTEX, ts$_{h}$) to $P_{h}$;
    }
  } else if (|grantedTo$_{LDR}$[MUTEX]| = $K_{LDR}$ - 1) {
    if (pendingReq$_{LDR}$[MUTIN] $\neq \emptyset$) {
      (ts$_{h}, P_{h}$) := deleteMin(pendingReq$_{LDR}$[MUTIN]);
    } else {
      for-each $P_{j} \in$ Grant$_{LDR}$ {
        if ((ts$_{j} + 1, P_{j}$) \in grantedTo$_{LDR}$[MUTIN] \land (ts$_{j}, P_{j}$) \in grantedTo$_{LDR}$[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 (Trigger, MUTIN, ts$_{h}$) to $P_{h}$;
    }
  }
}

Algorithm 7 Algorithm LKCS: message handlers (2).

On receipt of a (Trigger, at, ts) message:
  if (state$_{i}$ = InCS \land at = MUTEX \land ts = ts$_{i}$ \land |nGrants$_{i}$[MUTIN]| < |$N_{i}$| + 1) {
    /* $P_{i}$ is waiting for grant messages to exit. */
    nGrants$_{i}$[MUTIN] := \emptyset;
    for-each $P_{j} \in (N_{i} \cup \{P_{i}\})$ send (RequestByTrigger, MUTIN, $P_{j}$) to $P_{j}$;
    /* Request message as a trigger. */
  } else if (state$_{i}$ = OutCS \land at = MUTEX \land ts = ts$_{i}$ \land |nGrants$_{i}$[MUTEX]| < |$N_{i}$| + 1) {
    /* $P_{i}$ is waiting for grant messages to enter. */
    nGrants$_{i}$[MUTEX] := \emptyset;
    for-each $P_{j} \in (N_{i} \cup \{P_{i}\})$ send (RequestByTrigger, MUTEX, $P_{j}$) to $P_{j}$;
    /* Request message as a trigger. */
  }

On receipt of a (RequestByTrigger, at, $P_{j}$) message:
  Delete (at, $P_{j}$) from pendingReq$_{[at]}$;
  Delete (at, $P_{j}$) from grantedTo$_{[at]}$;
  grantedTo$_{[at]}$ := grantedTo$_{[at]}$ \cup \{(0, $P_{j}$)\};
  send (Grant, at, $P_{j}$) 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 $I_r$-L MUTIN, and we omit the detailed description here.

- **state**: The current state of $P_i$: InCS or OutCS.
- **ts**: The current value of the logical clock [19].
- **nGrants[at]**: A set of process ids from which $P_i$ obtains grants for exiting/entering to the CS.
- **grantedTos[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[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[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 $P_0$ with $\text{dist}(P_0, P_i) > 2$ cannot become unsafe by the proof by contradiction. Next, we show that other processes $P_i$ cannot become unsafe because they normally execute the algorithm as their instance $(I_i + 1, k_i - 1)$. Thus, we can derive the following lemma.

**Lemma 4.** (Safety) For each process $P_i \in V$, $l_i \leq |CS_i(C)| \leq k_i$ 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 $\text{dist}(P_0, P_i) > 2$ holds becomes unsafe first in a configuration $C$ for the contrary.

- Suppose that $|CS_i(C)| < l_i$, that is $|CS_i(C)| > |N_i| + 1 - l_i$. Because $|CS_i(C_0)| \geq l_i$, 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 $CS_i(C)$. Then, $P_j$ obtains permission to be the OutCS state from each process in $N_j \cup \{P_i\}$. This implies that $P_i$ receives a permission request $\langle \text{Request, MUTIN, } ts_j, P_j \rangle$ from $P_j$ and that $P_i$ sends a permission $\langle \text{Grant, MUTIN, } P_j \rangle$ 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 $\text{Cond}_i$, $P_i$ cannot obtain a permission from $P_j$; this is a contradiction.

- Suppose that $|CS_i(C)| > k_i$. Because $|CS_i(C_0)| \leq k_i$, 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 $CS_i(C)$. Then, $P_j$ obtains permission to be the InCS state from each process in $N_j \cup \{P_i\}$. This implies that $P_i$ receives a permission request $\langle \text{Request, MUTEX, } ts_j, P_j \rangle$ from $P_j$ and that $P_i$ sends a permission $\langle \text{Grant, MUTEX, } P_j \rangle$ to $P_j$. Because $P_i$ grants at most $k_i$ permissions to enter the CS at each time by the condition $\text{Cond}_i$, $P_i$ cannot obtain a permission from $P_j$; this is a contradiction.

Next, we consider $P_i$ that has $\text{dist}(P_0, P_i) \leq 2$. Note that the leader $P_{LDR}$ sends a trigger request $\langle \text{Trigger, } at, ts \rangle$ 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 $l_i + 1 \leq |CS_i(C)| \leq k_i - 1$ because of condition $\text{Cond}_i$ (of course, if $P_i = P_{LDR}$, $l_{LDR} + 1 \leq |CS_{LDR}(C)| \leq k_{LDR} - 1$). When $P_q$ requests to invert its state as a trigger by sending a message $\langle \text{RequestByTrigger, } at, P_j \rangle$, all of its neighbors $P_i$ grant it without attention to $|CS_i(C)|$, and $P_j$ inverts its state without attention to $|CS_q(C)|$. Thus, $|CS_i(C)|$ becomes $l_i + 1 - 1$ or $k_i - 1 + 1$ (if $P_i = P_{LDR}$, $|CS_{LDR}(C)|$ 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 $u = \text{OutCS}$. Then, all of their neighbors $P_v$ have $k_v$ neighbors $P_w$ with state $w = \text{InCS}$. 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 $P_a$ with state $u = \text{InCS}$ such that it cannot change its state by the exit-sequence. Then, it holds $|\mathcal{CS}_a(C)| = l_u$ or has a neighbor $P_b \in N_u$ with $|\mathcal{CS}_b(C)| = l_u$. 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 $w = \text{OutCS}$ changes $P_a$’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 exit-sequence until at least one of its neighbors $P_y \in N_w$ with state $y = \text{InCS}$ 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 $\text{grantedTo}_{LDR}$ or $\text{pendingReq}_{LDR}$, and $\text{Waiting}_{LDR}$ 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, $|\text{grantedTo}_{LDR}[\text{MUTIN}]| = |N_{LDR}| - l_{LDR}$ or $|\text{grantedTo}_{LDR}[\text{MUTEX}]| = k_{LDR} - 1$ holds. Then, $P_{LDR}$ sends a $\langle \text{Trigger, at}, ts \rangle$ message to a neighbor $P_q$ and $P_q$ becomes a trigger if $P_q$ is also waiting for grant messages. Processes $P_i$ that receive $\langle \text{RequestByTrigger, at}, P_q \rangle$ grant the request without attention to $|\mathcal{CS}_q(C)|$. 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 $\langle \text{Request, MUTIN, ts}, P_i \rangle$ message to each process in $N_i \cup \{P_i\}$; each process $P_j$ in $N_i \cup \{P_i\}$ sends a $\langle \text{Grant, MUTIN, P}_j \rangle$ message to $P_i$; then $P_i$ sends a $\langle \text{Release, MUTEX, P}_i \rangle$ 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 $\langle \text{Request, MUTEX, ts}, P_i \rangle$ message to each process in $N_i \cup \{P_i\}$; each process $P_j$ in $N_i \cup \{P_i\}$ sends a $\langle \text{Grant, MUTEX, P}_j \rangle$ message to $P_i$; then $P_i$ sends a $\langle \text{Release, MUTEX, P}_i \rangle$ message to each process in $N_i \cup \{P_i\}$. Thus, $3(|N_i| + 1)$ messages are exchanged for $P_i$’s 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 $\langle \text{Request, MUTIN, ts}, P_i \rangle$ message to each process $P_j$ in $N_i \cup \{P_i\}$. Then, $P_i$ sends a $\langle \text{Preempt, MUTIN, P}_j \rangle$ message to the process $P_m$ to which $P_j$ sends a $\langle \text{Grant, MUTIN, P}_j \rangle$ message; $P_m$ sends a $\langle \text{Relinquish, MUTIN, P}_m \rangle$ message back to $P_j$; and $P_j$ sends a $\langle \text{Grant, MUTIN, P}_j \rangle$ message to $P_i$. After $P_i$ exits the CS, $P_i$ sends a $\langle \text{Release, MUTEX, P}_i \rangle$ message to each process $P_j$ in $N_i \cup \{P_i\}$. Then, $P_i$ sends a $\langle \text{Grant, MUTEX, P}_i \rangle$ message to some process with the highest priority in $\text{pendingReq}[\text{MUTEX}]$. Thus, $6(|N_i| + 1)$ messages are exchanged.
- For $P_i$’s entering the CS, $P_i$ sends a $\langle \text{Request, MUTEX, ts}, P_i \rangle$ message to each process $P_j$ in $N_i \cup \{P_i\}$. Then, $P_i$ sends a $\langle \text{Preempt, MUTEX, P}_j \rangle$ message to the process $P_m$ to which $P_j$ sends a $\langle \text{Grant, MUTEX, P}_j \rangle$ message; $P_m$ sends a $\langle \text{Relinquish, MUTEX, P}_m \rangle$ message back to $P_j$; and $P_j$
sends a \(\langle \text{Grant}, \text{MUTEX}, P_i \rangle\) message to \(P_i\). After \(P_i\) enters the CS, \(P_i\) sends a \(\langle \text{Release}, \text{MUTIN}, P_j \rangle\) message to each process \(P_j \in N_i \cup \{P_i\}\). Then, \(P_j\) sends a \(\langle \text{Grant}, \text{MUTIN}, P_j \rangle\) message to some process with the highest priority in \(\text{pendingReq}_i[\text{MUTIN}]\). 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 \(\Delta\) 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 \(l_i \leq |CS_i(C)| \leq k_i\) 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**


