International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 12, Number 1, pages 103–130, January 2022

A self-stabilizing token circulation with graceful handover on bidirectional ring networks

Hirotsugu Kakugawa
Ryukoku University
Seta Oe, Otsu, Shiga, 520-2194 Japan

Sayaka Kamei
Hiroshima University
Kagamiyama, Higashi-Hiroshima City, Hiroshima, 739-8527 Japan

and

Yoshiaki Katayama
Nagoya Institute of Technology
Gokiso, Showa-ku, Nagoya, Aichi, 466-8555 Japan

Received: July 15, 2021
Revised: October 23, 2021
Accepted: November 29, 2021
Communicated by Masahiro Shibata

Abstract

In self-organizing distributed systems in which there is no centralized controller, cooperation of processes and fault-tolerance are crucial. The former can be formalized by process synchronization, which is one of the fundamental problems in concurrent, parallel and distributed computing. The latter can be formalized by self-stabilization. A self-stabilizing distributed algorithm is a class of fault-tolerant distributed algorithms that tolerates a finite number of any kind of transient faults. It can be considered as a self-organizing system because it does not need a globally synchronized initialization nor reset, and the system automatically converges to some legitimate configuration.

In this paper, we propose a self-stabilizing distributed algorithm for a token ring with the graceful handover on bidirectional ring networks with the message-passing communication model. The motivation of this work is to design a protocol, by a formal approach, which is useful for the self-organizing multi-node security camera system that guarantees continuous observation. More specifically, a system consists of several nodes each of which is equipped with a video camera, some of the nodes are active in monitoring, and others are inactive to save energy. The problem is to design an algorithm with the graceful handover of active nodes. That is, at least one node is active at any time, in other words, there is no time instant at which no node is active. This problem is formalized as the mutual inclusion problem, which is a process synchronization problem such that at least one process is in the critical section. To this end, we propose an algorithm for circulating two tokens on bidirectional ring networks under the state-reading model by extending Dijkstra's self-stabilizing token ring. We also propose the concept of the model gap tolerance property for the graceful handover. The proposed algorithm is self-stabilizing, and it guarantees the graceful handover in message-passing distributed systems.

Keywords: token ring, mutual inclusion, mutual exclusion, self-stabilization

1 Introduction

1.1 Background

Process synchronization is one of the fundamental problems in concurrent, parallel and distributed computing. A section of the algorithm (or program code) that is critical for coordination and competition of processes is called the *critical section*. The mutual *exclusion* problem is a typical process synchronization problem : *at most* one process is in the critical section at any time, that is, no two processes are in the critical section simultaneously. Mutual exclusion is used, for example, to avoid concurrent updates of a shared object. The mutual *inclusion* problem is another process synchronization problem in which *at least* one process is in the critical section at anytime [6]. Unification of mutual exclusion and mutual inclusion is proposed in [9] as the (ℓ, k) *critical section problem*.

A self-stabilizing distributed algorithm (system) is a class of distributed algorithms proposed by Dijkstra [2]. It tolerates any kind and any finite number of transient faults, for example, memory corruption by soft error, message loss and/or corruption. Self-stabilization is considered as one of the theoretical foundations of self-organizing systems because self-stabilizing systems need no globally synchronized initialization when it starts, nor global reset when some fault occurs. In addition, it is adaptive to changes of network topology. An important point in self-stabilizing distributed algorithms is that it is controlled in a distributed manner, that is, there is no centralized controller.

Self-stabilizing mutual inclusion has an interesting application, for example, to environmental monitoring IoT system which consists of a set of small nodes (physical entities of processes) with rechargeable batteries. Suppose that a network of nodes equipped with a sensor (e.g., camera), and each node executes a mutual inclusion algorithm. A node in the critical section actively monitors the environment, and other nodes are inactive to reduce energy consumption and can charge energy with solar cells or any energy harvesting device. Then, a (set of) node(s) which actively monitors the environment changes from moment to moment, however, there is at least one node that monitors the environment at any time. In other words, there is no time instant at which the environment is not monitored. The system is self-organizing because it is self-stabilizing.

1.2 Related works

The concept of self-stabilization is proposed by Dijkstra [2] as a framework of fault-tolerant distributed algorithms, and he proposed self-stabilizing token rings. A token ring can be considered as a mutual exclusion algorithm because (1) the token can be considered as a privilege to enter the critical section, (2) the number of tokens is exactly one at any time, and (3) each process eventually holds the token. Inspired by this pioneering work, self-stabilizing distributed algorithms are proposed extensively. A self-stabilizing multi-token ring is proposed in [3]. Here, a multi-token ring means that there are some constant numbers of tokens and they circulate a ring. In a token system (with single or multiple tokens), there is at least one process that holds a token and mutual inclusion is also achieved. Superstabilizing mutual exclusion algorithm on rings are proposed in [4, 15], where superstabilizing is an extension of self-stabilization in such a way that, in addition to the self-stabilizing property, a system recovers from an almost-legitimate configuration (such as a resultant configuration by a single transient fault at the legitimate configuration, for example), and the system keeps some safety predicate during convergence from almost-legitimate configuration.

The mutual inclusion problem is proposed by Hoogerwoord in [6] as a complement to the mutual exclusion problem, and he proposed a solution for only two processes with semaphores. A distributed algorithm in the fully asynchronous message-passing model with an arbitrary number of processes is proposed in [8], and a self-stabilizing distributed algorithm is proposed in [10]. In [9], a unification of the mutual inclusion problem and the mutual exclusion problem is proposed as the (ℓ, k) critical section problem in which at least ℓ and at most k processes are in the critical section, where $0 \leq \ell \leq k \leq n$ and n is the number of processes. Distributed algorithms for the (ℓ, k) critical section problem are found in [12–14].

Transformation schemes are often used such as [5, 7, 16] to execute self-stabilizing distributed algorithms in real sensor networks with message-passing communication. This is why many self-stabilizing distributed algorithms, e.g., [2, 3, 10, 14], assume the state-reading model as a communication model which is a kind of distributed shared-memory such that update of a local variable (memory) is immediately observed by other processes. Specifically, they transform an algorithm designed assuming the state-reading model to a program code executable in the message-passing model. These transformation schemes target a class of self-stabilizing distributed algorithms such that no process makes a move after the network is stabilized. The development of sensor networks for self-stabilizing algorithms that fall in this class is found in [17]. Unfortunately, algorithms for the mutual exclusion and inclusion are not in this class, and some consideration is necessary when we use these transformation schemes.

Preliminary version of this paper appeared in [11]. The time complexity of the convergence time is improved to $O(n^2)$ in this paper, whereas the conference version is $O(n^3)$.

1.3 Our contribution

We tackle a self-stabilizing mutual inclusion in the message-passing model (such as wireless sensor networks) in this paper. An application of transformation scheme proposed in [5] to token rings proposed in [2, 3] does not guarantee mutual inclusion in the message-passing model despite what they do in the state-reading model. So, we need some technique to achieve mutual inclusion in the message-passing model with a transformation scheme, and it is our motivation for this work.

First, we propose a self-stabilizing mutual inclusion algorithm on bidirectional rings in the state-reading model for communication and the composite atomicity model for execution. We present a formal description and proof of correctness of the proposed algorithm. Specifically, it guarantees that the number of processes in the critical section is at least one and at most two at any time. That is, it is also a solution to the $(1, 2)$ critical section problem. Then, we discuss how we can guarantee mutual inclusion in the message-passing model when we use a transformation scheme proposed in [5] which requires a small overhead at runtime. Unfortunately, the targets of the transformation are only silent self-stabilizing algorithms, where silence means that no process changes its local state after convergence. Because a token circulation algorithm is not silent, it is not known whether an application of the transformation scheme works correctly or not, and we need careful consideration. To this end, we propose a concept of the *model gap tolerance* such that mutual inclusion is guaranteed also in the message-passing model with the transformation scheme. We show that the proposed algorithm has the model gap tolerant property, and it can be executed in the message-passing model.

In summary, the algorithm design in this paper is the following three steps. (1) Design an algorithm with the model gap tolerance in the state-reading model, (2) prove the correctness of the algorithm in the state-reading model, and (3) apply the transformation scheme to run in the message-passing model. The benefit of our approach makes design and verification of the algorithm simple.

1.4 Organization of this paper

In section 2, we present definitions of computational model, self-stabilization, and the mutual inclusion problem. In section 3, we present a formal description of the proposed algorithm. In section 4, we show the proof of correctness and time complexity of the proposed algorithm. In section 5, we discuss the execution of the proposed algorithm by a transformation scheme proposed in [5], and introduce the concept of the model gap tolerance property. We show that the proposed algorithm has this property. In section 6, we give concluding remarks.

2 Preliminary

In this section, we present formal definitions of the network model and the concept of self-stabilization. Then, we briefly explain Dijkstra's self-stabilizing token ring algorithm.

2.1 The network model

A distributed system considered in this paper is a set of processes, which is an abstraction of nodes, and a set of communication links. Let $V = \{P_0, P_1, ..., P_{n-1}\}$ be the set of processes, where n is the number of processes. We assume that a network is a *bidirectional ring* in which each process P_i has two communication links $(P_{i-1 \bmod n}, P_i)$ and $(P_i, P_{i+1 \bmod n})$, that is, the set of communication links is $E = \{(P_{i-1 \bmod n}, P_i), (P_i, P_{i+1 \bmod n}) \mid 0 \leq i < n\}$. For simplicity of presentation, by P_{i+1} (resp., P_{i-1}), we denote $P_{i+1 \bmod n}$ (resp., $P_{i-1 \bmod n}$). For each P_i $(0 \leq i < n)$, we call P_{i+1} (resp., P_{i-1}) the *successor* (resp., *predecessor*) of P_i.

Let Q_i be the finite set of *local states* of P_i $(0 \leq i < n)$. It is assumed that $Q = Q_i$ for each $0 \leq i < n$, that is, all processes have the same set Q of local states, however, we adopt notation Q_i for simplicity of explanation. A *configuration* is an n-tuple of process states which represents the whole state of the network. When $q_i \in Q_i$ is the current local state of P_i for each $0 \leq i < n$, a configuration of the network is an n-tuple $(q_0, q_1, ..., q_{n-1})$. By Γ, we denote a set of all configurations, that is, $\Gamma = Q_0 \times Q_1 \times \cdots \times Q_{n-1} = Q^n$.

An *algorithm* for each P_i is a finite set of *guarded commands* in the following form.

if $\langle$guard 1$\rangle$ then $\langle$command 1$\rangle$
if $\langle$guard 2$\rangle$ then $\langle$command 2$\rangle$
if $\langle$guard 3$\rangle$ then $\langle$command 3$\rangle$
$\vdots$

A *guard* of P_i is a predicate on local states of P_i and its neighbors, that is, the j-th guard of P_i is a boolean function $G_{i,j}(q_i, q_{i-1}, q_{i+1})$. We say that a process is *enabled* if and only if it has a guard which evaluates to true. A *command* of P_i is a statement that updates q_i according to the values of q_i, q_{i-1} and q_{i+1}, that is, the j-th command of P_i is a form of $q_i \leftarrow C_{i,j}(q_i, q_{i-1}, q_{i+1})$.

Each process P_i can read and write its local variable q_i, and it can read its neighbors' local variables but cannot write. Reading neighbor's local variables completes without delay. Such a communication model is called the *state-reading* model. It is assumed that each process performs Read, Compute and Write in an atomic step. Such an execution model is called the *composite atomicity* model.

An *execution* X, starting from $\gamma_0 \in \Gamma$, is a maximal (possibly infinite) sequence of configurations $X = \gamma_0, \gamma_1, \gamma_2, \cdots$ such that $\gamma_t \to \gamma_{t+1}$ for each $t \geq 0$, where the binary relation $\to \subseteq \Gamma \times \Gamma$ represents configuration transition and it will be explained shortly. The first configuration γ_0 is called an *initial configuration* of the execution. Intuitively, an execution is a sequence of configurations by moves of processes. When there are two or more processes that are enabled in γ_t, selection scheme of a set of processes that make a move is called *scheduler* or *daemon*. A scheduler which selects an arbitrary nonempty set of enabled processes at each step is called the *distributed daemon*. A scheduler that selects exactly one enabled process at each step is called the *central daemon*. The distributed (resp., central) daemon is called *unfair* if it never yields an execution in which a process is continuously enabled forever. In other words, an unfair daemon may not select a process even if it is continuously enabled forever. Hence an algorithm must be correct for every possible selection by the unfair daemon. So, the design of self-stabilizing algorithms under the unfair daemon is not trivial. In this paper, we assume the unfair distributed daemon.

Let us define the binary relation $\to$. Intuitively, we have $\gamma_t \to \gamma_{t+1}$ if, in configuration γ_t, some processes selected by daemon make move, and γ_{t+1} is the next configuration. For any two configurations $\gamma^t, \gamma^{t+1} \in \Gamma$, we have $\gamma^t \to \gamma^{t+1}$ if and only if the following three conditions hold.

- Let $\gamma_t = (q_0^t, q_1^t, ..., q_{n-1}^t)$ and $\gamma_{t+1} = (q_0^{t+1}, q_1^{t+1}, ..., q_{n-1}^{t+1})$.
- Let $V' \subseteq V$ be a set of processes selected by daemon to move in γ_t.
- $q_i^{t+1} = C_{i,j}(q_i^t, q_{i-1}^t, q_{i+1}^t)$ for each $P_i \in V'$ and $q_k^{t+1} = q_k^t$ for other processes P_k.

So far we defined the case of bidirectional ring. Similarly, we can define *unidirectional ring* in such a way that link is one way from P_{i-1} to P_i for each $0 \leq i < n$. That is, a command to update the local state of P_i is a form of $q_i \leftarrow C_{i,j}(q_i, q_{i-1})$.

Algorithm 1 Dijkstra's K-state token ring SSToken

Constant integer K $(> n)$
Variable integer $x_i \in \{0, 1, 2, ..., K-1\}$
For the bottom process P_0
  Rule D1 : if $x_i = x_{i-1}$ then
    $x_i \leftarrow x_{i-1} + 1 \bmod K$
  Token condition : $x_i = x_{i-1}$
For the other process P_i $(0 < i < n)$
  Rule D2 : if $x_i \neq x_{i-1}$ then
    $x_i \leftarrow x_{i-1}$
  Token condition : $x_i \neq x_{i-1}$

2.2 Self-stabilization

Let us introduce the concept of self-stabilization proposed by Dijkstra [2]. Let Γ be a set of all configurations and $\Lambda \subseteq \Gamma$ be a set of configurations, and each $\gamma \in \Lambda$ is called a *legitimate* configuration. A configuration $\gamma \in (\Gamma \backslash \Lambda)$ is called an *illegitimate* configuration. A distributed system is *self-stabilizing* with respect to Λ if and only if the following two properties hold.

- *Closure* : Starting from any legitimate configuration $\gamma \in \Lambda$, the next configuration γ' such that $\gamma \rightarrow \gamma'$, we have $\gamma' \in \Lambda$.

- *Convergence* : Starting from any (possibly illegitimate) configuration $\gamma \in \Gamma$, the system eventually reaches a legitimate configuration $\gamma' \in \Lambda$.

Intuitively, the closure property means that once the system becomes legitimate, it remains so forever (as long as no fault occurs). Note that a legitimate configuration and the next configuration may or may not be the same. Specifically, in cases of mutual exclusion and inclusion, configurations change forever among legitimate ones after convergence is achieved.

A self-stabilization is preferable, especially for distributed systems. For example, (1) globally synchronized reset is not necessary to initialize a distributed system, (2) it tolerates any (finite) number of transient faults, such as soft error in processor and memory devices, message corruption, loss and duplication. We regard the configuration just after these undesirable events as an initial configuration of a system, and by the convergence property of self-stabilization, it is guaranteed that the system is automatically brought to a legitimate configuration again.

Because we assume the unfair distributed daemon as a process scheduler, the convergence property implies that, for *any* scheduling of processes, the system reaches a legitimate configuration. This property is strong in a sense that there is no execution in which the system is illegitimate forever.

2.3 Dijkstra's self-stabilizing K-state token ring

The proposed *mutual inclusion* algorithm is based on the algorithm of Dijkstra's token ring [2], and we briefly review it here. Dijkstra proposed three algorithms for self-stabilizing *mutual exclusion* as a token ring, and we adopt so-called the K-state token ring among them shown in Algorithm 1. It assumes the state-reading model, the composite atomicity model and distributed daemon, as same as our algorithm assumes.

The K-state token ring assumes a unidirectional ring network with n processes $P_0, P_1, P_2, ..., P_{n-1}$. Process P_0 is a distinguished process called the *bottom process*, and each process $P_1, P_2, ..., P_{n-1}$ is called the *other process*. The bottom process runs its algorithm which is different from the other processes, and other processes are identical and they run the same algorithm. Each process P_i $(0 \leq i < n)$ has a local variable x_i and its domain is $\{0, 1, 2, ..., K-1\}$, where K is any constant such that $K > n$ when the token ring is executed under the distributed daemon.

Process P_i holds a *token*, which is a virtual object, if and only if it is enabled. In legitimate configurations, exactly one token is circulated in the ring, and when a process holds the token, it may enter the critical section to take a privileged action.

A configuration $(x_0, x_1, ..., x_{n-1})$ is legitimate if and only if, for some $x \in \{0, 1, ..., K-1\}$ and $1 \le \ell \le n-1$, it is a form of $(\overbrace{x, x, ..., x}^{n})$ or $(\overbrace{x+1, ..., x+1}^{\ell}, \overbrace{x, ..., x}^{n-\ell})$, where arithmetic is modulo K.

3 The proposed mutual inclusion algorithm

In this section, we first explain the overview of the proposed algorithm SSRmin, then, the technical detail is presented.

One may think that we can use Dijkstra's token ring for the application of monitoring systems because the number of tokens is exactly one at any time. It is true in the state-reading model, however, it is not true in the message-passing model because token passing is not instant because of message transmission delay. That is, there is no token in the system between the time a process releases a token and the time the next process receives it. Furthermore, one may think that we can use general mutual inclusion algorithm because the number of tokens is at least one at any time. It is true, however, the number of tokens can be too many and it can be resource-consuming. So, the requirements to our algorithm is that (1) there is at least one token in the message-passing model, and (2) the number of tokens is as small as possible. To this end, we first develop an algorithm under the state-reading model with model gap tolerance (explained shortly) in subsection 3.2, and verify its correctness in section 4. Then, we transform the algorithm into the message-passing model in section 5. The model gap tolerance guarantees that there is at least one token by the transformed algorithm in the message-passing model. Because the transfer of tokens is controlled in such a way that there is no time instant without a token, the *graceful handover* is achieved.

3.1 Overview of the algorithm

The proposed algorithm assumes bidirectional ring with two tokens. Each process P_i $(0 \le i < n)$ has access to local variables at P_{i-1} and P_{i+1}, in addition to P_i's local variables. In legitimate configurations, exactly two tokens are circulated in the ring, however, a process may hold two tokens at the same time and, in such a situation, the number of processes that hold a token is one. Otherwise, there are two processes that hold a token. In our algorithm, different from other multi-token circulation algorithms, two processes that hold tokens are neighbors (or the same).

It is important to notice that a token is *not* implemented by a virtual data object in this paper. It is defined by a *predicate* on local variables, *i.e.*, a process decides whether it holds a token or not by evaluating some predicate, what we call a *token condition*, on the values of local variables of itself and its neighbors. For simplicity of description, we often use phrases 'a process sends a token to a neighbor' or 'a token is moved to a neighbor'. Their precise meanings are that a process changes the values of its local variables so that its token condition becomes false and, at the same time, a neighbor's token condition becomes true.

The tokens are named as the *primary token* and the *secondary token*. Intuitively, the two tokens move in a ring like an inchworm. In legitimate configurations, when the primary token is located at P_i, the secondary token is located at P_i or P_{i+1}. Two variables rts_i (which means 'ready to send' the secondary token) and tra_i (which means 'token receipt acknowledged' for the secondary token) for each process P_i are introduced to control the movement of two tokens. Specifically, these two variables are used for handshaking between two processes P_i (the holder of the primary token) and P_{i+1}.

- The primary token is maintained by the Dijkstra's token ring with minor modification for movement control. This plays the role of the tail of an inchworm. When the primary token is located at P_i, it moves to P_{i+1} if the secondary token is located at P_{i+1}.

Algorithm 2 Abstraction of Dijkstra's K-state token ring

Macro
  For the bottom process P_i $(i = 0)$:
    $G_i \ \equiv \ x_i = x_{i-1}$
    $C_i \ \equiv \ x_i \leftarrow x_{i-1} + 1 \bmod K$
  For the other process P_i $(0 < i < n)$:
    $G_i \ \equiv \ x_i \neq x_{i-1}$
    $C_i \ \equiv \ x_i \leftarrow x_{i-1}$
Rule δ :
  if G_i then C_i

- The secondary token is our extension. This plays the role of the head of an inchworm. When the secondary token is located at P_i, it moves to P_{i+1} if the primary token is also located at P_i.

The condition such that process P_i holds a token is as follows.

- P_i holds the primary token if and only if G_i is true. Here, G_i is the condition for P_i to have a token by Dijkstra's token ring. It is formally defined in Algorithm 2 and will be explained shortly.
- P_i holds the secondary token if and only if $(tra_i = 1) \vee (rts_i = 1 \wedge rts_{i+1} = 0 \wedge tra_{i+1} = 0)$ is true.

The idea of the token movement is controlled by the following abstract actions, explained as follows. The set of abstract actions are presented here for intuitive understanding only, and the concrete actions are presented shortly based on the abstract actions.

1. Initially, we assume that P_i holds the primary and the secondary tokens.
2. *Abstract action* α_1 *by* P_i (Ready to send the secondary token) :
   P_i sets $rts_i = 1$. This means that P_i is ready to send the secondary token to P_{i+1}.
3. *Abstract action* β *by* P_{i+1} (Receive the secondary token) :
   When P_{i+1} observes $rts_i = 1$, it sets $tra_{i+1} = 1$. This means that the secondary token is moved from P_i to P_{i+1}, and P_{i+1} receives the secondary token. Note that primary token remains at P_i.
4. *Abstract action* α_2 *by* P_i (Send the primary token) :
   When P_i observes $tra_{i+1} = 1$, it executes a rule of Dijkstra's token ring algorithm, and sends the primary token from P_i to P_{i+1}. At the same time, it sets $rts_i = 0$.

   Now the primary and the secondary tokens are located at P_{i+1}, and a cycle of token movement is achieved. By repeating the above (abstract) actions, two tokens move the ring network.

Because, in the Dijkstra's token ring, the bottom process P_0 has a different guarded command from the other processes P_i $(1 \leq i < n)$. For simplicity of understanding of the proposed algorithm, here, we collapse the rules for the bottom process and for the other processes into a single rule "if G_i then C_i", where G_i is a guard and C_i is a command as shown in Algorithm 2. Then, with G_i and C_i, the idea presented above (abstract actions) is described as follows. Rules A_1 and B control the movement of the secondary token, and Rules A_2 controls the movement of the primary token. Note that abstract action β is explained above as an action of P_{i+1}, however, it is described as an action of P_i here. We also introduce Rule F to fix inconsistent local state for the convergence property. For simplicity of description, we implicitly assume the priority of rules $A_1 > A_2 > B > F$ at each process (A_1 is the highest and F is the lowest). That is, if the guard of a rule is true, rules with lower priority are ignored. So, a process is enabled by at most one rule.

Step	P_0	P_1	P_2	P_3	P_4
1	PS	$-$	$-$	$-$	$-$
2	P	S	$-$	$-$	$-$
3	$-$	PS	$-$	$-$	$-$
4	$-$	P	S	$-$	$-$
5	$-$	$-$	PS	$-$	$-$
6	$-$	$-$	P	S	$-$
$\vdots$					

Figure 1: Movement of the two tokens; 'P' (resp., 'S') represents the primary (resp., secondary) token

1. Rule A_1 (For abstract action α_1; Ready to send the secondary token) :
   if $rts_i = 0 \wedge G_i$ then
   $rts_i \leftarrow 1;\ \ tra_i \leftarrow 0;$
2. Rule A_2 (For abstract action α_2; Send the primary token) :
   if $rts_i = 1 \wedge tra_{i+1} = 1 \wedge G_i$ then
   $rts_i \leftarrow 0;\ \ tra_i \leftarrow 0;\ \ C_i;$
3. Rule B (For abstract action β; Receive the secondary token) :
   if $rts_{i-1} = 1 \wedge tra_i = 0 \wedge \neg G_i$ then
   $rts_i \leftarrow 0;\ \ tra_i \leftarrow 1;$
4. Rule F (Fix inconsistent local state) :
   if (locally incorrect) then
   $rts_i \leftarrow 0;\ \ tra_i \leftarrow 0;$

An example of token movement with five processes in legitimate configuration is illustrated in Figure 1. Handshaking mechanism between processes P_i and P_{i+1} with two variables *rts* and *tra* is illustrated in Figure 2. In this figure, Rules A_1, A_2 and B are shown, however, Rule F is not because it is a rule to converge to a legitimate configuration from illegitimate ones.

Let us consider the condition for the secondary token. One may think that a condition $tra_i = 1$ will suffice for the secondary token. If we take such a condition, the secondary token disappears when the primary token moves to the process where the secondary token resides, and it appears when it moves to the next process. So it extincts when two tokens are virtually located at the same process. This is not a problem in the state-reading model because there exists at least one token at any time, however, the proposed condition guarantees what we call the model gap tolerance property when the proposed algorithm is executed in the message-passing model. We will discuss this issue in section 5.

3.2 Detail of the algorithm

The concrete and formal description of the proposed algorithm SSRmin is presented in Algorithm 3. It has five rules and we assume that a rule with a smaller number has priority over rules with a larger rule number. So, each process is enabled by at most one rule. Rules 1, 2 and 4 are executed when G_i, the guard of Dijkstra's token ring, is true, while Rules 3 and 5 are executed when G_i is false. To make the algorithm self-stabilizing, we relax the conditions of the guards so that the rules can be applied for illegitimate configurations to converge. For example, when $rts_i = 1$ and $tra_i = 1$, we continue the token circulation as much as possible or reset these variables. Figure 3 shows possible rules to execute for each value of $\langle rts_i.tra_i \rangle$, and it may help the readers to follows the proof.

Definition 1 *The set of all configuration is* $\Gamma = \{0, 1, ..., K-1\} \times \{0, 1\} \times \{0, 1\}$. *We use a notation* $x_i.rts_i.tra_i$ *to write local state of process* P_i *for each* $0 \le i < n$. *A configuration*

$$(x_0.rts_0.tra_0, x_1.rts_1.tra_1, \ldots, x_{n-1}.rts_{n-1}.tra_{n-1})$$

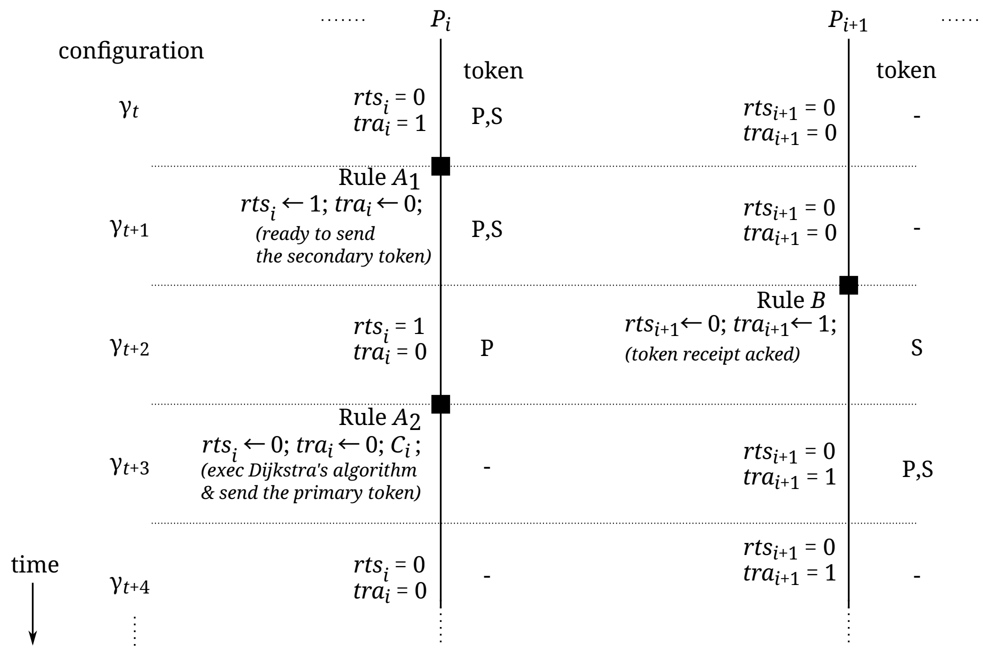

Figure 2: Handshaking between P_i and P_{i+1} to pass tokens

is legitimate if and only if it is one of the following forms for some $x \in \{0, 1, ..., K-1\}$ (arithmetic on x is modulo K). For readability, a process with a token is under-waved.

- P_0 *holds the primary and the secondary tokens :*

  $(\utilde{x.0.1}, x.0.0, x.0.0, \ldots, x.0.0)$,

  $(\utilde{x.1.0}, x.0.0, x.0.0, \ldots, x.0.0)$

- P_0 *holds the primary token and* P_1 *holds the secondary token :*

  $(\utilde{x.1.0}, \utilde{x.0.1}, x.0.0, \ldots, x.0.0)$

- P_i $(1 \leq i \leq n-1)$ *holds the primary and the secondary tokens :*

  $(x+1.0.0, \ldots, x+1.0.0, \utilde{x.0.1}, x.0.0, \ldots, x.0.0)$,

  $(x+1.0.0, \ldots, x+1.0.0, \utilde{x.1.0}, x.0.0, x.0.0, \ldots, x.0.0)$

- P_i $(1 \leq i \leq n-1)$ *holds the primary token and* $P_{(i+1) \bmod n}$ *holds the secondary token :*
  $(x+1.0.0, \ldots, x+1.0.0, \utilde{x.1.0}, \utilde{x.0.1}, x.0.0, \ldots, x.0.0)$

We denote, by Λ, the set of all legitimate configurations. □

An execution example which starts in a legitimate configuration with five processes is shown in Figure 4. At each step, status of each process P_i is shown, and the values of local variables are written in the form of $x_i.rts_i.tra_i$. In addition to local variables, 'P' (resp., 'S') indicates that P_i holds the primary (resp., secondary) token, and '$/g$' $(1 \leq g \leq 5)$ indicates that the guard of Rule g evaluates to true at P_i. For example, '$1.0.1PS/1$' for P_i means that $x_i = 1$, $rts_i = 0$, $tra_i = 1$, P_i holds the primary and the secondary tokens, and the guard of Rule 1 evaluates to true at P_i.

Algorithm 3 Mutual inclusion algorithm SSRmin for each process P_i

Constant

$n \geq 3$; *// the number of processes*

$K > n$; *// state space for x*

Variable

integer $x_i \in \{0, 1, ..., K-1\}$; *// state of the Dijkstra's K-state token ring*

boolean rts_i; *// ready to send a token*

boolean tra_i; *// token receive acknowledged*

Macro

For the bottom process P_i $(i = 0)$

$G_i \equiv x_i = x_{i-1}$

$C_i \equiv x_i \leftarrow x_{i-1} + 1 \bmod K$

For the other process P_i $(1 \leq i < n)$

$G_i \equiv x_i \neq x_{i-1}$

$C_i \equiv x_i \leftarrow x_{i-1}$

Rule set :

Rule 1 : if $G_i \wedge (\langle rts_{i-1}.tra_{i-1},\ rts_i.tra_i,\ rts_{i+1}.tra_{i+1} \rangle$
$= \langle ?.?,\ 0.0,\ ?.? \rangle$ or
$= \langle ?.?,\ 0.1,\ ?.? \rangle$ or
$= \langle ?.?,\ 1.1,\ ?.? \rangle)$ then *// ready to send the secondary token*
$\langle rts_i.tra_i \rangle = \langle 1.0 \rangle$;

Rule 2 : if $G_i \wedge (\langle rts_{i-1}.tra_{i-1},\ rts_i.tra_i,\ rts_{i+1}.tra_{i+1} \rangle$
$= \langle ?.?,\ 1.0,\ 0.1 \rangle)$ then *// send the primary token*
$\langle rts_i.tra_i \rangle = \langle 0.0 \rangle$; C_i;

Rule 3 : if $\neg G_i \wedge (\langle rts_{i-1}.tra_{i-1},\ rts_i.tra_i,\ rts_{i+1}.tra_{i+1} \rangle$
$= \langle 1.0,\ 0.0,\ ?.? \rangle$ or
$= \langle 1.0,\ 1.0,\ ?.? \rangle$ or
$= \langle 1.0,\ 1.1,\ ?.? \rangle)$ then *// receive the the secondary token*
$\langle rts_i.tra_i \rangle = \langle 0.1 \rangle$;

Rule 4 : if $G_i \wedge (\langle rts_{i-1}.tra_{i-1},\ rts_i.tra_i,\ rts_{i+1}.tra_{i+1} \rangle$
$\neq \langle 0.0,\ 1.0,\ 0.0 \rangle)$ then *// fix inconsistent local state when G_i is true*
$\langle rts_i.tra_i \rangle = \langle 0.0 \rangle$; C_i;

Rule 5 : if $\neg G_i \wedge (\langle rts_{i-1}.tra_{i-1},\ rts_i.tra_i,\ rts_{i+1}.tra_{i+1} \rangle$
$\neq \langle 1.0,\ 0.1,\ ?.? \rangle$ and
$\neq \langle ?.?,\ 0.0,\ ?.? \rangle)$ then *// fix inconsistent local state when G_i is false*
$\langle rts_i.tra_i \rangle = \langle 0.0 \rangle$;

Token condition

the primary token : G_i

the secondary token : $\langle rts_{i-1}.tra_{i-1},\ rts_i.tra_i,\ rts_{i+1}.tra_{i+1} \rangle$
$= \langle ?.?,\ ?.1,\ ?.? \rangle$ or
$= \langle ?.?,\ 1.?,\ 0.0 \rangle$

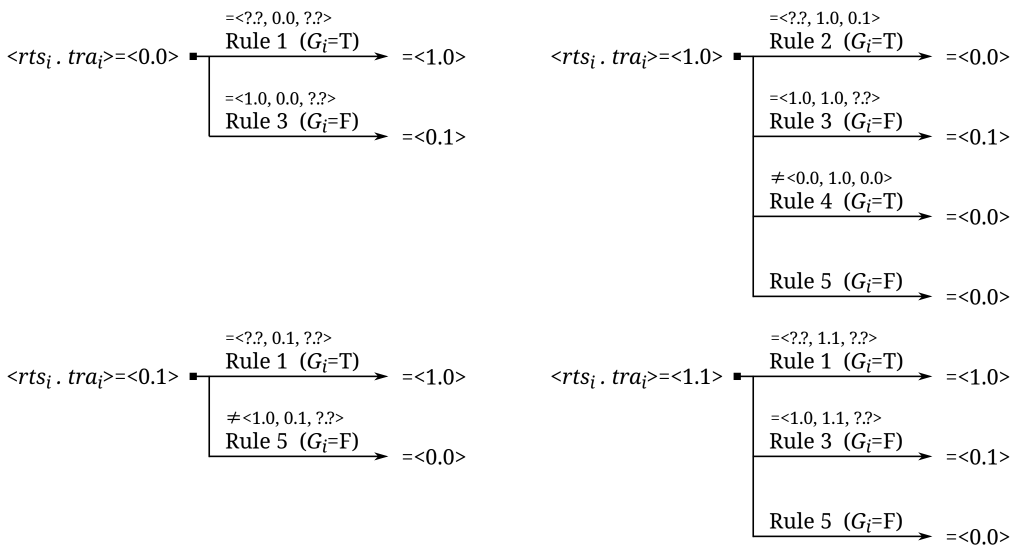

Figure 3: Possible rules for each $\langle rts_i.tra_i \rangle$ values

Step	P_0	P_1	P_2	P_3	P_4
1	$\underline{3.0.1}PS/1$	3.0.0	3.0.0	3.0.0	3.0.0
2	$3.1.0PS$	$\underline{3.0.0}/3$	3.0.0	3.0.0	3.0.0
3	$\underline{3.1.0}P/2$	$3.0.1S$	3.0.0	3.0.0	3.0.0
4	4.0.0	$\underline{3.0.1}PS/1$	3.0.0	3.0.0	3.0.0
5	4.0.0	$3.1.0PS$	$\underline{3.0.0}/3$	3.0.0	3.0.0
6	4.0.0	$\underline{3.1.0}P/2$	$3.0.1S$	3.0.0	3.0.0
7	4.0.0	4.0.0	$\underline{3.0.1}PS/1$	3.0.0	3.0.0
8	4.0.0	4.0.0	$3.1.0PS$	$\underline{3.0.0}/3$	3.0.0
9	4.0.0	4.0.0	$\underline{3.1.0}P/2$	$3.0.1S$	3.0.0
10	4.0.0	4.0.0	4.0.0	$\underline{3.0.1}PS/1$	3.0.0
11	4.0.0	4.0.0	4.0.0	$3.1.0PS$	$\underline{3.0.0}/3$
12	4.0.0	4.0.0	4.0.0	$\underline{3.1.0}P/2$	$3.0.1S$
13	4.0.0	4.0.0	4.0.0	4.0.0	$\underline{3.0.1}PS/1$
14	$\underline{4.0.0/3}$	4.0.0	4.0.0	4.0.0	$3.1.0PS$
15	$4.0.1S$	4.0.0	4.0.0	4.0.0	$\underline{3.1.0}P/2$
16	$\underline{4.0.1}PS/1$	4.0.0	4.0.0	4.0.0	4.0.0
⋮					

Figure 4: An execution example of SSRmin with five processes (local state $x_i.rts_i.tra_i$ is underlined if enabled)

4 Proof of correctness and performance analysis

In this section, we show proof of correctness of the proposed algorithm SSRmin and the analysis of time complexity. The distributed daemon is assumed as a process scheduler.

Lemma 1 *(Closure) For any legitimate configuration $\gamma \in \Lambda$, the configuration γ' which follows γ is also legitimate.*

Proof. Let γ_0 be a configuration $\gamma_0 = (x.0.1, x.0.0, x.0.0, \cdots, x.0.0)$ for some $x \in \{0, 1, ..., K-1\}$. This configuration is defined as legitimate, and we select it as an initial configuration. To prove the lemma, it is enough to show that (a) every configuration that is reachable from γ_0 is legitimate, and (b) γ_0 is reachable from every legitimate configuration. In the proof of part (a) below, observe that every legitimate configuration (including γ_0) defined by Definition 1 is reachable from γ_0. So part (b) is easily verified from the proof of part (a).

In each configuration γ in any execution that starts from γ_0, we show below that, by simply following an execution sequence, (1) there exists exactly one enabled process, and (2) the next configuration of γ is also legitimate. So, the distributed daemon has no free choice : it must select the only enabled process. For readability, the local state of an enabled process is underlined in the configuration shown below. Figure 4 may help the readers.

- In $\gamma_0 = (\underline{x.0.1}, x.0.0, x.0.0, \cdots, x.0.0)$, according to the proposed algorithm, P_0 is the only enabled process (by Rule 1) in this configuration. Hence the possible choice for the distributed daemon is only P_0 (Rule 1). Then, we have the next configuration γ_1 which is legitimate shown below.

- In $\gamma_1 = (x.1.0, \underline{x.0.0}, x.0.0, \cdots, x.0.0)$, P_1 is the only enabled process (by Rule 3), and it executes the rule. The next configuration is γ_2 which is legitimate shown below.

- In $\gamma_2 = (\underline{x.1.0}, x.0.1, x.0.0, \cdots, x.0.0)$, P_0 is the only enabled process (by Rule 2), and it executes the rule.

  The next configuration is γ_3 which is legitimate, and we show below general form of configurations $\gamma_3, \gamma_4, \gamma_5, ..., \gamma_{3n-4}$.

- In $\gamma_{3i} = (x+1.0.0, \cdots, x+1.0.0, \underline{x.0.1}, x.0.0, \cdots, x.0.0)$, where $1 \le i \le n-2$, P_i is the only enabled process (by Rule 1), and it executes the rule. The next configuration is γ_{3i+1} which is legitimate shown below.

- In $\gamma_{3i+1} = (x+1.0.0, \cdots, x+1.0.0, x.1.0, \underline{x.0.0}, x.0.0 \cdots, x.0.0)$, P_{i+1} is the only enabled process (by Rule 3), and it executes the rule. The next configuration is γ_{3i+2} which is legitimate shown below.

- In $\gamma_{3i+2} = (x+1.0.0, \cdots, x+1.0.0, \underline{x.1.0}, x.0.1, x.0.0 \cdots, x.0.0)$, P_i is the only enabled process (by Rule 2), and it executes the rule. The next configuration is γ_{3i+3}, and it is easy to see that it is legitimate.

  Above three steps are repeated from $i = 1$ to $n-2$, and we have a configuration γ_{3n-3} shown below and it is legitimate. Note that, when $i = n-1$, $P_{i+1 \bmod n}$ is the bottom process P_0 and we cannot handle it as a general case.

- In $\gamma_{3n-3} = (x+1.0.0, \cdots, x+1.0.0, \underline{x.0.1})$, P_{n-1} is the only enabled process (by Rule 1), and it executes the rule. The next configuration is γ_{3n-2} which is legitimate shown below.

- In $\gamma_{3n-2} = (\underline{x+1.0.0}, \cdots, x+1.0.0, x.1.0)$, P_0 is the only enabled process (by Rule 3), and it executes the rule. The next configuration is γ_{3n-1} which is legitimate shown below.

- In $\gamma_{3n-1} = (x+1.0.1, \cdots, x+1.0.0, \underline{x.1.0})$, P_{n-1} is the only enabled process (by Rule 2), and it executes the rule. The next configuration is γ_{3n} which is legitimate shown below.

- Now we have $\gamma_{3n} = (\underline{x+1.0.1}, x+1.0.0, \cdots, x+1.0.0)$, and γ_{3n} and γ_0 differ only the value of x, which is incremented by one in every process. It is easy to see that the same observation applies for initial configuration γ_{3n} with $x+1 \bmod K$. Hence it is verified that the next configuration of a legitimate configuration is also legitimate. By repeating this observation K times, γ_0 is reached, *i.e.*, γ_0 is reachable from γ_0. □

Lemma 2 *For each legitimate configuration $\gamma \in \Lambda$, the number of the primary token is exactly one, and the number of the secondary token is also exactly one in γ.*

Proof. It is easily verified by the definition of legitimate configurations and the tokens. □

Lemma 3 *For any configuration $\gamma \in \Gamma$, there exists P_i such that G_i evaluates to true, that is, $x_0 = x_{n-1}$ holds or $x_i \neq x_{i-1}$ holds for some i $(1 \leq i < n)$.*

Proof. By definition of G_i, it is the same as the guard of the Dijkstra's token ring presented in Algorithms 1 and 2. That is, G_0 for P_0 is the guard of Rule D1, and for each $1 \leq i < n$, G_i for P_i is the guard of Rule D2. Hence the proof in [2] applies here and there exists at least one process P_i such that G_i is true. □

Lemma 4 *(No Deadlock) For any configuration $\gamma \in \Gamma$, there exists at least one process P_k such that P_k has a rule whose guard evaluates to true.*

Proof. Let P_j be any process such that $\langle rts_j.tra_j \rangle = \langle 1.1 \rangle$. If G_j evaluates to true, P_j is enabled by Rule 1. Otherwise, it is enabled by Rule 3 or Rule 5. In the following, we consider configurations in which no such process exists. Let P_i be any process such that G_i evaluates to true. Recall that there exists such P_i by Lemma 3.

In case $\langle rts_{i-1}.tra_{i-1},\ rts_i.tra_i,\ rts_{i+1}.tra_{i+1} \rangle = \langle ?.?,\ 0.0,\ ?.? \rangle$ or $\langle ?.?,\ 0.1,\ ?.? \rangle$, P_i is enabled by Rule 1. In case $\langle rts_{i-1}.tra_{i-1},\ rts_i.tra_i,\ rts_{i+1}.tra_{i+1} \rangle = \langle ?.?,\ 1.0,\ ?.? \rangle$, we consider the following cases:

- Case $\langle rts_{i-1}.tra_{i-1},\ rts_i.tra_i,\ rts_{i+1}.tra_{i+1} \rangle = \langle ?.?,\ 1.0,\ 0.0 \rangle$: If G_{i+1} evaluates to true, P_{i+1} is enabled by Rule 1. Otherwise, P_{i+1} is enabled by Rule 3.
- Case $\langle rts_{i-1}.tra_{i-1},\ rts_i.tra_i,\ rts_{i+1}.tra_{i+1} \rangle = \langle ?.?,\ 1.0,\ 0.1 \rangle$: P_i is enabled by Rule 2.
- Case $\langle rts_{i-1}.tra_{i-1},\ rts_i.tra_i,\ rts_{i+1}.tra_{i+1} \rangle = \langle ?.?,\ 1.0,\ 1.0 \rangle$: P_i is enabled by Rule 4.

Because there exists an enabled process in any configuration, deadlock never occurs. □

By this lemma, any maximal execution is infinite.

Lemma 5 *For any configuration $\gamma_0 \in \Gamma$, $3n$ is the maximum length of execution that does not include any execution of Rule 2 and Rule 4.*

Proof. Let us observe an execution $\gamma_0, \gamma_1, \gamma_2, ...$ in which Rules 2 and 4 are never executed. In such an execution, each process P_j never executes C_j (the command part of Dijkstra's token ring). As a result, the value of G_j never changes for each P_j throughout the execution.

Let P_i be any process which executes a rule in γ_0. Note that the following discussion holds any execution under the unfair distributed daemon, *i.e.*, P_i and other process(es) may execute simultaneously in γ_0.

Case P_i executes Rule 1 in γ_0. In γ_1, we have $\langle rts_i.tra_i \rangle = \langle 1.0 \rangle$ and G_i remains true. Then, Rules 2 and 4 are the rules that make P_i enabled in the next time. However, it is assumed that P_i never executes these rules, P_i never executes any rule forever. Hence the maximum number of executions of rules by P_i is one (Rule 1 only) in this case.

Case P_i executes Rule 3 in γ_0. In γ_1, we have $\langle rts_i.tra_i\rangle = \langle 0.1\rangle$ and G_i remains false. Then, Rule 5 is the only rule that makes P_i enabled in the next time. Since P_i executes Rule 3 in γ_0, $\langle rts_{i-1}.tra_{i-1}\rangle = \langle 1.0\rangle$ holds in γ_0. For P_i to execute Rule 5 in γ_t for some $t > 0$, $\langle rts_{i-1}.tra_{i-1}\rangle \neq \langle 1.0\rangle$ must hold in γ_t. This is possible only if P_{i-1} executes Rule 3 or 5 in some configuration $\gamma_{t'}$ $(0 \leq t' < t)$, and, if it occurs, we have $\langle rts_{i-1}.tra_{i-1}\rangle = \langle 0.1\rangle$ or $\langle 0.0\rangle$ in $\gamma_{t'+1}$. Since P_{i-2} never executes Rules 2 and 4 and G_{i-1} is false forever, the rules that P_{i-1} may execute is Rules 3 and 5 in $\gamma_{t'+1}$ and later configurations. So, we have $\langle rts_{i-1}.tra_{i-1}\rangle = \langle 0.1\rangle$ or $\langle 0.0\rangle$ in $\gamma_{t'+1}$ and later configurations. If P_i executes Rule 5 in γ_t, we have $\langle rts_i.tra_i\rangle = \langle 0.0\rangle$ and then, P_i is never enabled thereafter. Hence the maximum number of executions of rules by P_i is at most two (Rules 3 and 5) in this case.

Case P_i executes Rule 5 in γ_0. In γ_1, we have $\langle rts_i.tra_i\rangle = \langle 0.0\rangle$ and G_i remains false. Then, Rule 3 is the only rule that makes P_i enabled in the next time. For P_i to execute Rule 3 in some configuration γ_t $(t > 0)$, we must have $\langle rts_{i-1}.tra_{i-1},\ rts_i.tra_i,\ rts_{i+1}.tra_{i+1}\rangle = \langle 1.0,\ 0.0,\ ?.?\rangle$ in γ_t. We consider two cases : G_{i-1} is true or false.
Case G_{i-1} is true. Because it is assumed that P_{i-1} never executes Rules 2 and 4, we have $\langle rts_{i-1}.tra_{i-1}\rangle = \langle 1.0\rangle$ in γ_t and later configurations. Once P_i executes Rule 3 in γ_t, we have $\langle rts_i.tra_i\rangle = \langle 0.1\rangle$ and P_i is never enabled in γ_{t+1} and later configurations. Hence the maximum number of executions of rules by P_i is at most two (Rules 5 and 3) in this case.
Case G_{i-1} is false. Rule 1 is the only rule for P_{i-1} to yield $\langle rts_{i-1}.tra_{i-1}\rangle = \langle 1.0\rangle$, however, P_{i-1} never executes it since G_{i-1} is false. Hence, for P_i to execute Rule 3 in γ_t, we must have $\langle rts_{i-1}.tra_{i-1}\rangle = \langle 1.0\rangle$ in γ_0 and remains so until γ_t. This means that, after P_i executes Rule 5 in γ_0, we have $\langle rts_{i-1}.tra_{i-1},\ rts_i.tra_i,\ rts_{i+1}.tra_{i+1}\rangle = \langle 1.0,\ 0.0,\ ?.?\rangle$ in γ_1. Then, P_i is enabled by Rule 3 in γ_1, and remains so until γ_t. After P_i executes Rule 3 in γ_t, we have $\langle rts_i.tra_i\rangle = \langle 0.1\rangle$ in γ_{t+1}. Let us consider the following subcases for execution of P_{i-1} in γ_t.

- If P_{i-1} simultaneously executes Rule 3 in γ_t, we have $\langle rts_{i-1}.tra_{i-1},\ rts_i.tra_i,\ rts_{i+1}.tra_{i+1}\rangle = \langle 0.1,\ 0.1,\ ?.?\rangle$ in γ_{t+1}. Then, in γ_{t+1} and later configurations, we have $\langle rts_{i-1}.tra_{i-1}\rangle = \langle 0.1\rangle$ or $\langle 0.0\rangle$ because G_{i-1} is false. This means that Rule 5 is the only possible rule for P_i to execute in later configurations, and, if P_i executes Rule 5, it is not enabled forever.

- If P_{i-1} simultaneously executes Rule 5 in γ_t, we have $\langle rts_{i-1}.tra_{i-1},\ rts_i.tra_i,\ rts_{i+1}.tra_{i+1}\rangle = \langle 0.0,\ 0.1,\ ?.?\rangle$ in γ_{t+1}. Then, in γ_{t+1} and later configurations, we have $\langle rts_{i-1}.tra_{i-1}\rangle = \langle 0.1\rangle$ or $\langle 0.0\rangle$ because G_{i-1} is false. Hence Rule 5 is the only possible rule for P_i to execute in later configurations, and, if P_i executes Rule 5, it is not enabled forever.

- If P_{i-1} does not execute any rule in γ_t, we have $\langle rts_{i-1}.tra_{i-1},\ rts_i.tra_i,\ rts_{i+1}.tra_{i+1}\rangle = \langle 1.0,\ 0.1,\ ?.?\rangle$ in γ_{t+1}. Hence, in γ_{t+1}, P_{i-1} is enabled by Rule 3 or 5 but P_i is not enabled.

  If P_{i-1} executes Rule 3, we have $\langle rts_{i-1}.tra_{i-1},\ rts_i.tra_i,\ rts_{i+1}.tra_{i+1}\rangle = \langle 0.1,\ 0.1,\ ?.?\rangle$, and the same observation applies as above.

  If P_{i-1} executes Rule 5, we have $\langle rts_{i-1}.tra_{i-1},\ rts_i.tra_i,\ rts_{i+1}.tra_{i+1}\rangle = \langle 0.0,\ 0.1,\ ?.?\rangle$, and the same observation applies as above.

The maximum number of executions of rules by P_i is at most three (Rules 5, 3, and then 5) in this case.

According to the observation above, each process executes rules at most three times. Hence $3n$ is the upper bound on the maximum length of execution which does not include Rules 2 and 4. □

Lemma 6 *(Convergence) For each configuration $\gamma_0 \in \Gamma$ and any execution that starts from γ_0, eventually legitimate configuration in Λ is reached.*

Proof. Let γ_0 be any configuration. According to the proof of Lemma 5, some process P_i such that G_i (the guard part of Dijkstra's token ring) evaluates to true executes C_i (the command part of Dijkstra's token ring) at least once in every $3n$ steps, and eventually the part of Dijkstra's token ring in SSRmin converges.

The followings are the general properties of rules after Dijkstra's token ring converges.

- We have $\langle 0.0 \rangle$ by executing Rules 2 and 4 at P_i, and then G_{i+1} becomes true at P_{i+1}.
- There is no rule to yield $\langle 1.1 \rangle$.
- The rule to yield $\langle rts_i.tra_i \rangle = \langle 1.0 \rangle$ is only Rule 1 and it is executed by P_i only if G_i evaluates to true, and we have $\langle rts_i.tra_i \rangle = \langle 0.0 \rangle$ when G_i becomes false which occurs by execution of Rules 2 or 4.

Let γ_1 be the configuration such that the part of Dijkstra's token ring is converged and P_0 holds the primary token, *i.e.*, G_0 evaluates to true. Because γ_1 may not be legitimate, we show below that configuration eventually becomes legitimate.

Let us observe the configuration, say γ_2, just after the primary token circulates the ring once and G_0 becomes true again. From the general properties of rules, the following conditions hold in γ_2.

- For each P_i $(0 \le i < n)$, we have $\langle rts_i.tra_i \rangle = \langle 0.0 \rangle$, $\langle 0.1 \rangle$ or $\langle 1.0 \rangle$ regardless the value of G_i.
- For P_0, we have $\langle rts_0.tra_0 \rangle = \langle 0.0 \rangle$ or $\langle 0.1 \rangle$. This is because $\langle rts_0.tra_0 \rangle = \langle 1.0 \rangle$ never occurs because P_0 executes Rule 2 in some configuration from γ_1 to γ_2. The former case occurs if P_0 never executes Rule 3 in any configuration from γ_1 to γ_2. The latter case occurs if P_0 executes Rule 3 in some configuration from γ_1 to γ_2.
- For each P_i $(1 \le i < n)$, we have $\langle rts_i.tra_i \rangle = \langle 0.0 \rangle$ because $\langle rts_{i-1}.tra_{i-1} \rangle = \langle 1.0 \rangle$ never occurs after P_{i-1} executes Rule 2 or 4 in configurations from γ_1 to γ_2.

In summary, we have $\gamma_2 = (x.0.1, x.0.0, x.0.0, \cdots, x.0.0)$ or $(x.0.0, x.0.0, x.0.0, \cdots, x.0.0)$ for some x. For the former case, the configuration is legitimate and convergence is completed. For the latter case, the configuration is not legitimate and we need more observation for convergence. By simply following the execution from the configuration, we have the next sequence of configurations (the enabled process is underlined) :

1. $(\underline{x.0.0}, x.0.0, x.0.0, \cdots, x.0.0) = \gamma_2$ and P_0 executes Rule 1,
2. $(x.1.0, \underline{x.0.0}, x.0.0, \cdots, x.0.0)$ and P_1 executes Rule 3,
3. $(\underline{x.1.0}, x.0.1, x.0.0, \cdots, x.0.0)$ and P_0 executes Rule 2, then
4. $(x+1.0.0, \underline{x.0.1}, x.0.0, \cdots, x.0.0)$ which is legitimate. □

Theorem 1 *The proposed algorithm SSRmin is a self-stabilizing mutual inclusion algorithm on bidirectional ring such that (1) the number of privileged processes is at least one and at most two, (2) the number of states per process is $4K$, where K is a constant such that $K > n$.*

Proof. It is self-stabilizing because the closure property is shown by Lemma 1 and the convergence property is shown by Lemma 6. By Lemma 2, at least one and at most two processes are privileged in legitimate configuration. The number of states per process is $4K$ because x_i takes K values, and rts_i and tra_i are binary. □

Lemma 7 *Let $\gamma = (x_0.rts_0.tra_0, x_1.rts_1.tra_1, \cdots, x_{n-1}.rts_{n-1}.tra_{n-1})$ be any configuration such that the part of Dijkstra's token ring of SSRmin is converged, i.e., $(x_0, x_1, ..., x_{n-1})$ is a legitimate configuration of Dijkstra's token ring. Then, for any execution starting from γ, SSRmin converges within $O(n^2)$ steps.*

Proof. By Lemma 5, at most $3n$ steps are executed before a single step (execution of C_i at some P_i) of Dijkstra's token ring is performed. As we observed in the proof of Lemma 6, SSRmin converges after the token of Dijkstra's token ring circulates the ring plus four steps. Hence a legitimate configuration of SSRmin is reached in $3n \cdot n + 4 = O(n^2)$ steps. □

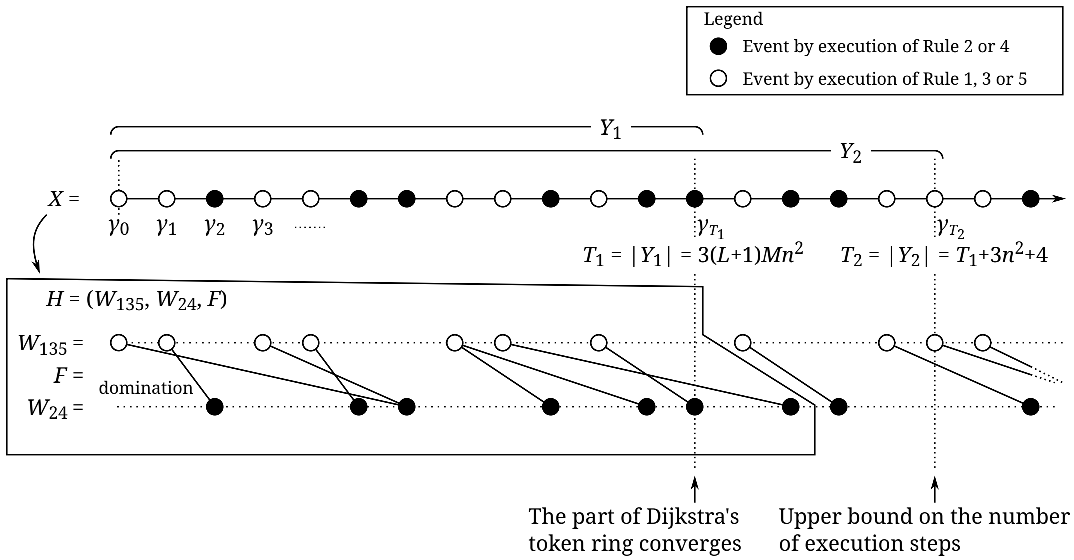

Figure 5: Construction of a bipartite graph $H = (W_{135}, W_{24}, F)$ from execution X

Lemma 8 *For any initial configuration γ_0 and for any execution starting from γ_0 of SSRmin, the part of Dijkstra's token ring of SSRmin converges, i.e., $(x_0, x_1, ..., x_{n-1})$ becomes a legitimate configuration of Dijkstra's token ring, in $O(n^2)$ steps.*

Proof. It is shown in [1] that $3n(n-1)/2$ is the upper bound on the convergence time of Dijkstra's token ring under the unfair distributed daemon. So, it is sufficient to show that $3n(n-1)/2$ steps of Dijkstra's token ring (Rules 2 and 4) are executed in $O(n^2)$ steps of SSRmin.

First, we explain the outline of the proof. Let $X = \gamma_0, \gamma_1, \gamma_2, \cdots$ be any infinite execution of SSRmin. Below we use a notation $|Y|$ for any prefix Y of X to denote the length of Y. Let Y_1 be a prefix of X such that Y_1 includes at least $3n(n-1)/2$ executions of the steps of Dijkstra's token ring, *i.e.*, executions of Rules 2 and 4 of SSRmin. We denote the length of Y_1 by T_1, *i.e.*, $T_1 = |Y_1|$. Then, the part of Dijkstra's token ring is converged in γ_{T_1}. Additional $3n^2+4$ steps are sufficient to reach a legitimate configuration of SSRmin by Lemma 7. We show below that $O(n^2)$ is sufficient for T_1, and hence the time complexity of SSRmin is $T_1 + 3n^2 + 4 = O(n^2)$, which completes the proof of this lemma.

Let us start by defining some symbols used in this proof. For each $0 \le i < n$ and $z \ge 1$, let e^i_z be the event such that it is the z-th execution of a rule at P_i in Y_2, where Y_2 is the prefix of X such that $|Y_2| = T_1 + 3n^2 + 4$. Below we denote the length of Y_2 by T_2, *i.e.*, $T_2 = |Y_2|$. Let W_{135} (resp., W_{24}) be a set of events such that $e \in W_{135}$ (resp., $e \in W_{24}$) if and only if e is an event of execution of Rule 1, 3 or 5 in Y_1 (resp., Rule 2 or 4 in X). Note that W_{135} is a finite set and W_{24} is an infinite set, however, vertices in W_{24} that are not related to the convergence analysis are removed from W_{24} at the final stage of the proof, and finally W_{24} becomes a finite set.

We construct a bipartite graph $H = (W_{135}, W_{24}, F)$ such that $(e, f) \in F$ if and only if $e \in W_{135}$ is dominated by $f \in W_{24}$. We use the concept of domination to show that the number of executions of Rules 1, 3 and 5 ($= |W_{135}|$) is within a constant factor of the number of executions of Rules 2 and 4 ($= |W_{24}|$). Figure 5 is an intuitive illustration of H constructed from X, and its technical detail is explained shortly. For simplicity, the figure shows a special case of X in which exactly one event occurs in each configuration, however, one or more events occur in general. We say that $e \in W_{135}$ is *dominated* by $f \in W_{24}$, if e occurs at P_i, f must occur at some P_j for further execution of P_i in X. Intuitively, if e occurs, P_i is unblocked by f for further executions. Formally speaking, if we modify the execution X in such a way that the occurrence of f is inhibited, the number of events that occur at P_i after e is at most some constant in the modified execution. Note that an event in W_{135} is dominated by one or more events in W_{24}, and an event in W_{24} dominates one or

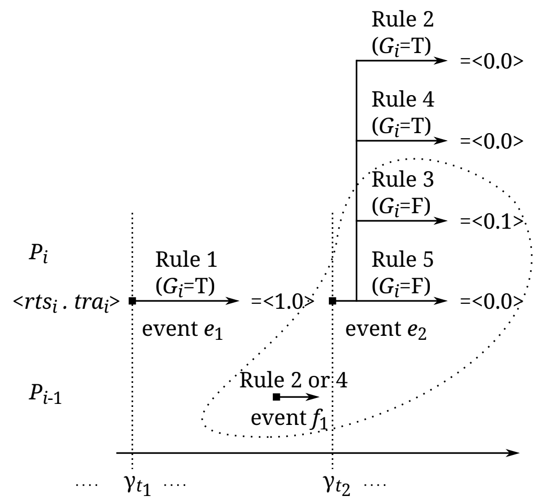

Figure 6: Possible executions of rules at P_i and related executions at P_{i-1} after P_i executes Rule 1

more events in W_{135}. As we will see later that P_j at which a dominating event occurs is limited to $j \in \{i, i-1, i-2\}$ which limits the number of events in domination relation.

Shortly, we present the domination relation of two events, and we also show the followings as fundamental observations to prove the upper bound on the time complexity:

1. For each event in W_{135}, it is dominated by some event in W_{24}.

2. There exists a constant L such that the degree of each $f \in W_{24}$ in graph H is at most L.

3. There exists a constant M such that for any occurrence of event in W_{135} at any P_i, its dominating event in W_{24} occurs before the next M events at P_i occur.

Intuitive interpretations of these three are as follows. Items 1 and 2 claim the *bound on domination size*. The bound on the number of events in W_{135} is a constant factor of the number of occurrences of events in W_{24}. Item 3 claims the *bound on time delay* at each P_i. For any occurrence of event in W_{135}, a dominating event in W_{24} occurs within a constant number steps at P_i. Note that the bound on time delay is local to P_i and executions of corresponding events of P_i and dominating events are interleaved in X with other processes.

By these observations, the prefix Y_1 of X includes $3n(n-1)/2$ occurrence of events in W_{24} if $T_1(=|Y_1|)$ is some constant factor of $3n(n-1)/2$. Specifically, $T_1 = 3(L+1)Mn^2$ is sufficient by the following reason. First, let us count the upper bound on the number of events. We have at most $3(L+1)n(n-1)/2$ events in Y_1 so that $3n(n-1)/2$ events in W_{24} occur in Y_1 because at most L events in W_{135} occurs for each event in W_{24}. Next, let us find the upper bound on the number of steps in X for the $3(L+1)n(n-1)/2$ events. For any series of (consecutive) s_i events at P_i in X, at least $\lfloor s_i/2M \rfloor$ events in W_{24} occur because any series of $2M$ events contains an interval of series of M events that starts by an event in W_{135} and the interval includes an event in W_{24}. For any length $S = \sum_i s_i$, the prefix of X of length S includes at least $\sum_i \lfloor s_i/2M \rfloor > (\sum_i s_i/2M) - n = S/2M - n$ events in W_{24}. For the value of S so that $3n(n-1)/2$ events in W_{24} to occur, $S = 3(L+1)Mn^2$ is sufficient, and we choose $T_1 = 3(L+1)Mn^2$.

Now we present the domination relation. Let e_1 be any event in W_{135}, P_i be the process at which e_1 occurs, and γ_{t_1} is the configuration in which e_1 occurs. Let γ_{t_2} be the configuration in which P_i executes a rule in the next time ($t_1 < t_2$) and P_i does not execute any rule in $\gamma_{t_1+1}, ..., \gamma_{t_2-1}$. Let e_2 be the event that P_i executes a rule in γ_{t_2}. We observe executions of processes to make the occurrence of e_2 possible and to find a dominating event of e_1.

Case event e_1 is an execution of Rule 1.

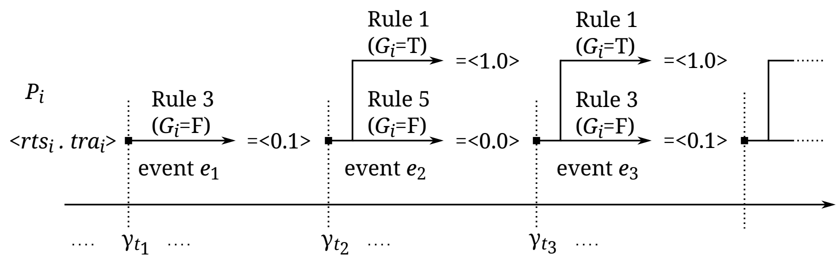

Figure 7: Possible executions of rules at P_i after P_i executes Rule 3

Figure 6 illustrates executions of processes to help to understand this case. In γ_{t_1}, G_i evaluates to true to execute Rule 1, and we have $\langle rts_i.tra_i \rangle = \langle 1.0 \rangle$ in γ_{t_1+1} and remain so until γ_{t_2}. Possible rules for P_i as event e_2 in γ_{t_2} are Rule 2, 3, 4 and 5. (See also Figure 3 for possible rules.)

- Case event e_2 is an execution of Rule 2 or 4:

  Because e_2 is in W_{24} and, by the occurrence of e_2, P_i proceeds to execute the next event in W_{135}, e_1 is dominated by e_2. An edge (e_1, e_2) is added to F.

- Case event e_2 is an execution of Rule 3 or 5:

  The value of G_i changes from true (in γ_{t_1}) to false (in γ_{t_2}). This change occurs only if P_{i-1} executes Rule 2 or 4 in some configuration $\gamma_{t_1}, \gamma_{t_1+1}, ..., \gamma_{t_2-1}$, and let f_1 is the corresponding event by P_{i-1}. Because G_i must be false to execute Rules 3 and 5 for P_i, e_1 is dominated by f_1. An edge (e_1, f_1) is added to F.

The bound on domination size (each $f \in W_{24}$ dominates at most constant number of events by Rule 1) and the bound on time delay are shown as follows.

- Each $f \in W_{24}$ dominates at most one event by Rule 1 for each P_i: For any process P_i, let e be any event by Rule 1 at P_i. Let γ_{t_1} (resp., γ_{t_2}) be the configuration in which e occurs (resp., the next execution of a rule by P_i occurs). Then f occurs in some configuration in $\gamma_{t_1}, \gamma_{t_1+1}, \cdots, \gamma_{t_2-1}$. Hence, the event by Rule 1 at P_i that f dominates is the event e that occurs in γ_{t_1} and f does not dominate other events by Rule 1 at P_i.
- Each $f \in W_{24}$ dominates events by Rule 1 that occur at a constant number of processes: For each process P_i, let $f \in W_{24}$ be any event which occurs at P_i. Then, for each event e by Rule 1 which is dominated by f, e occurs at P_{i-1} or P_i. Equivalently, each $f \in W_{24}$ at P_i never dominates any event by Rule 1 which occurs at P_j ($j \notin \{i-1, i\}$).
- For each occurrence of $e \in W_{135}$ at P_i, its dominating event occurs before the next event not in W_{24} occurs at P_i.

Case event e_1 is an execution of Rule 3.

Figure 7 illustrates executions of processes to help to understand this case. In γ_{t_1}, G_i evaluates to false to execute Rule 3, and we have $\langle rts_i.tra_i \rangle = \langle 0.1 \rangle$ in γ_{t_1+1} and remain so so until γ_{t_2}. Possible rules for the next execution of P_i as e_2 in γ_{t_2} are Rules 1 and 5. (See also Figure 3 for possible rules.)

- Case (1a): Event e_2 is an execution of Rule 1.

  The value of G_i changes from false (in γ_{t_1}) to true (in γ_{t_2}), and this change occurs only if P_{i-1} executes Rule 2 or 4 in some configuration $\gamma_{t_1}, \gamma_{t_1+1}, ..., \gamma_{t_2-1}$, and let f be the corresponding event of the execution by P_{i-1}. Because G_i must be true to execute Rule 1 for P_i, e_1 is dominated by f. An edge (e_1, f) is added to F.

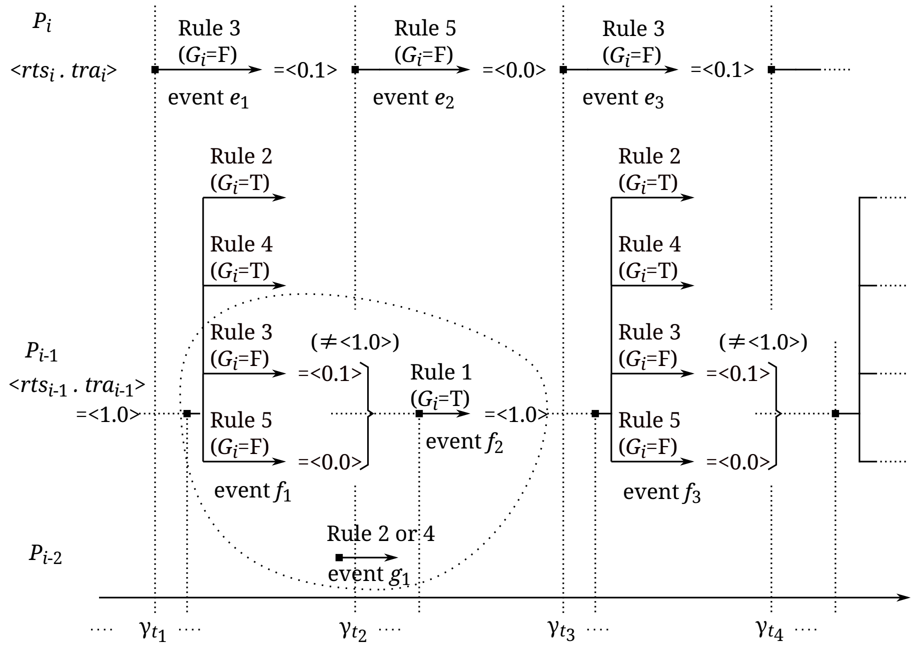

Figure 8: Cycles of executions of Rules 3 and 5 at P_i and related executions at P_{i-1} and P_{i-2}

- Case (1b): Event e_2 is an execution of Rule 5.

  We have $\langle rts_i.tra_i \rangle = \langle 0.0 \rangle$ in γ_{t_2+1} as a result of execution of Rule 5. Then, possible rules for the next execution of P_i are Rules 1 and 3 as shown in Figure 7. Let e_3 be the event for the next execution, and let γ_{t_3} be the configuration in which e_3 occurs.

We continue to find a dominating event of e_1 by observing events that enable e_3 to occur.

- Case (2a): Event e_3 is an execution of Rule 1.

  A similar observation applies as case (1a). The event e_3 occurs only if P_{i-1} executes Rule 2 or 4 in some configuration $\gamma_{t_2}, \gamma_{t_2+1}, ..., \gamma_{t_3-1}$, and let f be the corresponding event of the execution by P_{i-1}. Because G_i must be true to execute Rule 1 for P_i, e_1 is dominated by f. An edge (e_1, f) is added to F.

- Case (2b): Event e_3 is an execution of Rule 3.

  We have $\langle rts_i.tra_i \rangle = \langle 0.1 \rangle$ in γ_{t_3+1} as a result of execution of Rule 3. Then, possible rules for the next execution of P_i are Rules 1 and 5.

Then, the same observation repeats as cases (1a), (1b), (2a) and (2b) as long as Rule 1 is not executed by P_i. Because we have not found a dominating event yet in the repeated executions of Rules 3 and 5 by P_i, let us observe the repeated executions in detail. Figure 8 illustrates such executions to help to understand this case.

When P_i executes Rule 3 in γ_{t_1}, we have $\langle rts_{i-1}.tra_{i-1} \rangle = \langle 1.0 \rangle$ in γ_{t_1}. When P_i executes Rule 5 in γ_{t_2}, we have $\langle rts_{i-1}.tra_{i-1} \rangle \neq \langle 1.0 \rangle$ in γ_{t_2}, which implies that P_{i-1} executes a rule in some configuration in $\gamma_{t_1}, \gamma_{t_1+1}, \cdots, \gamma_{t_2-1}$. Possible rules for P_{i-1} are Rules 2, 3, 4 and 5. Let f_1 be the corresponding event by execution of a rule by P_{i-1}.

- If P_{i-1} executes Rule 2 or 4, f_1 is in W_{24}. Because f_1 makes e_2 to occur (f_1 enables execution of Rule 5 at P_i in γ_{t_2}), f_1 is a dominating event of e_1. An edge (e_1, f_1) is added to F.

- If P_{i-1} executes Rule 3 or 5, P_i executes Rule 5 in γ_{t_2}. Then, P_i is enabled by Rule 3 in the next time, and executes it in γ_{t_3} as we assumed in this case. When P_i executes Rule 3 in

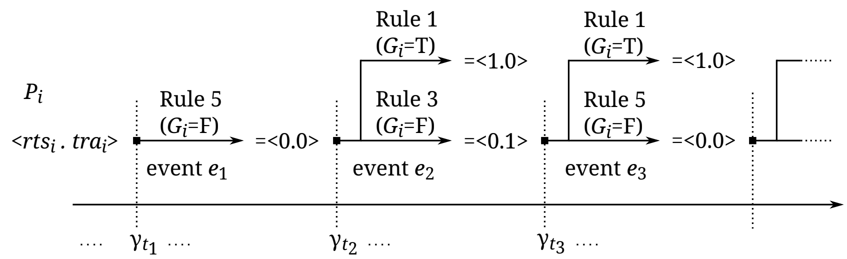

Figure 9: Possible executions of rules at P_i after P_i executes Rule 5

γ_{t_3}, we have $\langle rts_{i-1}.tra_{i-1}\rangle = \langle 1.0\rangle$. Hence the value of $\langle rts_{i-1}.tra_{i-1}\rangle$ need to change from non-$\langle 1.0\rangle$ (by f_1) to $\langle 1.0\rangle$. This change occurs only by executing Rule 1 by P_{i-1}, and let f_2 be the corresponding event.

Because the value of G_{i-1} changes from false (when f_1 occurs at P_{i-1}) to true (when f_2 occurs at P_{i-1}), P_{i-2} executes Rule 2 or 4 in between. Let g_1 be the corresponding event by P_{i-2}. In summary, event g_1 enables the execution of Rule 1 by P_{i-1} (event f_2), which enables the execution of Rule 3 by P_i (event e_3). Hence g_1 is the dominating event of e_1, and an edge (e_1, g_1) is added to F.

The bound on domination size (each $f \in W_{24}$ dominates at most constant number of events by Rule 3) and the bound on time delay are shown as follows.

- Each $f \in W_{24}$ dominates at most one event by Rule 3 for each P_i: For any process P_i, let e be any event by Rule 3 at P_i. Let γ_{t_1} (resp., γ_{t_3}) be the configuration in which e occurs (resp., the next execution of Rule 3 by P_i occurs). Then f occurs in some configuration in $\gamma_{t_1}, \gamma_{t_1+1}, \cdots, \gamma_{t_3-1}$, where f is equal to the event by Rule 2 or 4 at P_{i-1} in case (2a) that dominates e_2 (in Figure 7), f_1 by Rule 2 or 4 (in Figure 8), or g_1 (in Figure 8). Hence, the event by Rule 3 at P_i that f dominates is the event e that occurs in γ_{t_1} and f does not dominate other events by Rule 3 at P_i.
- Each $f \in W_{24}$ dominates events by Rule 3 that occur at a constant number of processes: For each event e by Rule 3 which is dominated by f, e occurs at P_{i-2}, P_{i-1} or P_i. Equivalently, each $f \in W_{24}$ at P_i never dominates any event by Rule 3 which occurs at P_j ($j \notin \{i-2, i-1, i\}$).
- For each occurrence of $e \in W_{135}$ at P_i, its dominating event occurs before the next two events occur at P_i.

Case event e_1 is an execution of Rule 5.

This case is similar to the case of Rule 3 as we observed above. Figure 9 illustrates executions of processes to help to understand this case. In γ_{t_1}, G_i evaluates to false to execute Rule 5, and we have $\langle rts_i.tra_i\rangle = \langle 0.0\rangle$ in γ_{t_1+1} and remain so so until γ_{t_2}. Possible rules for the next execution of P_i as e_2 in γ_{t_2} are Rules 1 and 3. (See also Figure 3 for possible rules.)

- Case (1a): Event e_2 is an execution of Rule 1.

  The value of G_i changes from false (in γ_{t_1}) to true (in γ_{t_2}), and this change occurs only if P_{i-1} executes Rule 2 or 4 in some configuration $\gamma_{t_1}, \gamma_{t_1+1}, ..., \gamma_{t_2-1}$, and let f be the corresponding event of the execution by P_{i-1}. Because G_i must be true to execute Rule 1 for P_i, e_1 is dominated by f. An edge (e_1, f) is added to F.

- Case (1b): Event e_2 is an execution of Rule 3.

  We have $\langle rts_i.tra_i\rangle = \langle 0.1\rangle$ in γ_{t_2+1} as a result of execution of Rule 3. Then, possible rules for the next execution of P_i are Rules 1 and 5 as shown in Figure 9. Let e_3 be the event for the next execution, and let γ_{t_3} be the configuration in which e_3 occurs.

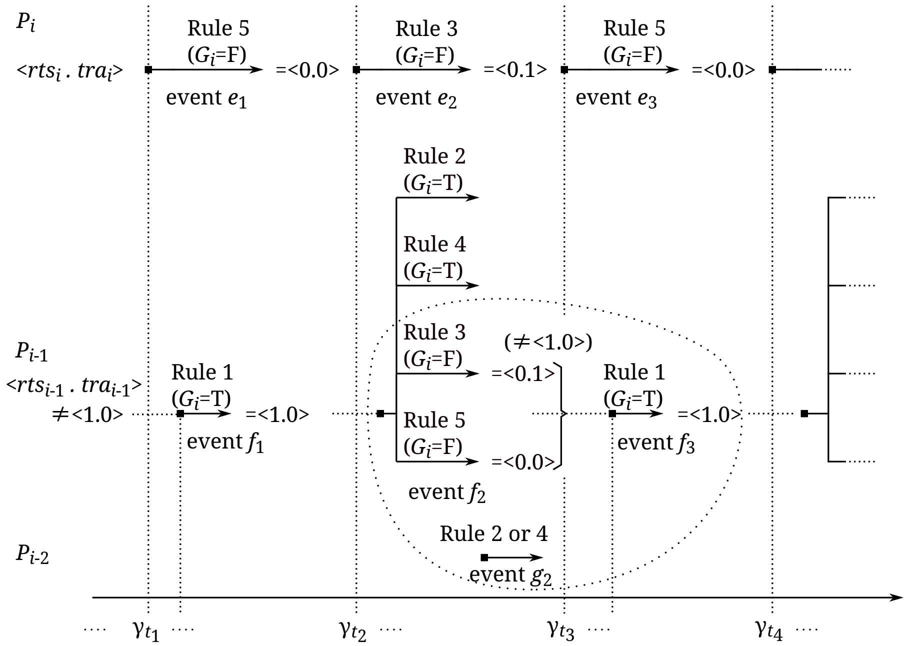

Figure 10: Cycles of executions of Rules 3 and 5 at P_i and related executions at P_{i-1} and P_{i-2}

We continue to find a dominating event of e_1 by observing events that enable e_3 to occur.

- Case (2a): Event e_3 is an execution of Rule 1.

  The event e_3 occurs only if P_{i-1} executes Rule 2 or 4 in some configuration $\gamma_{t_2}, \gamma_{t_2+1}, ..., \gamma_{t_3-1}$, and let f be the corresponding event of the execution by P_{i-1}. Because G_i must be true to execute Rule 1 for P_i, e_1 is dominated by f. An edge (e_1, f) is added to F.

- Case (2b): Event e_3 is an execution of Rule 5.

  We have $\langle rts_i.tra_i\rangle = \langle 0.0\rangle$ in γ_{t_3+1} as a result of execution of Rule 5. Then, possible rules for the next execution of P_i are Rules 1 and 3.

Then, the same observation repeats as cases (1a), (1b), (2a) and (2b) as long as Rule 1 is not executed by P_i. Because we have not found a dominating event yet in the repeated executions of Rules 5 and 3 by P_i, let us observe the repeated executions in detail. Figure 10 illustrates such executions to help to understand this case.

When P_i executes Rule 5 in γ_{t_1}, we have $\langle rts_{i-1}.tra_{i-1}\rangle \neq \langle 1.0\rangle$ in γ_{t_1}. When P_i executes Rule 3 in γ_{t_2}, we have $\langle rts_{i-1}.tra_{i-1}\rangle = \langle 1.0\rangle$ in γ_{t_2}, which implies that P_{i-1} executes a rule in some configuration in $\gamma_{t_1}, \gamma_{t_1+1}, \cdots, \gamma_{t_2-1}$. A possible rule for P_{i-1} is Rule 1 only because it is the only rule to yield $\langle rts_{i-1}.tra_{i-1}\rangle = \langle 1.0\rangle$. Let f_1 be the corresponding event by execution of Rule 1 by P_{i-1}. After execution of Rule 1, we have $\langle rts_{i-1}.tra_{i-1}\rangle = \langle 1.0\rangle$, and then, P_i executes Rule 3 as event e_2. Then, by a similar observation as the case of an execution of Rule 3 by P_i in γ_{t_1}, we have an event f that dominates e_1, where f is equal to f_2 by Rule 2 or 4, or g_2 (both in Figure 10). An edge (e_1, f) is added to F.

The bound on domination size (each $f \in W_{24}$ dominates at most constant number of events by Rule 5) and the bound on time delay are shown as follows.

- Each $f \in W_{24}$ dominates at most one event by Rule 5 for each P_i: For any process P_i, let e be any event by Rule 5 at P_i. Let γ_{t_1} (resp., γ_{t_3}) be the configuration in which e occurs (resp., the next execution of Rule 5 by P_i occurs). Then f occurs in some configuration in $\gamma_{t_1}, \gamma_{t_1+1}, \cdots, \gamma_{t_3-1}$, where f is equal to the event by Rule 2 or 4 at P_{i-1} in case (2a) that dominates e_2 (in Figure 7), f_2 by Rule 2 or 4 (in Figure 10), or g_2 (in Figure 10). Hence, the event by Rule 5 at P_i that f dominates is the event e that occurs in γ_{t_1} and f does not dominate other events by Rule 5 at P_i.

- Each $f \in W_{24}$ dominates events by Rule 5 that occur at a constant number of processes: For each event e by Rule 5 which is dominated by f, e occurs at P_{i-2}, P_{i-1} or P_i. Equivalently, each $f \in W_{24}$ at P_i never dominates any event by Rule 5 which occurs at P_j ($j \notin \{i-2, i-1, i\}$).
- For each occurrence of $e \in W_{135}$ at P_i, its dominating event occurs before the next two events occur at P_i.

For completeness of the construction of graph H, we remove every vertex f in W_{24} if f is not incident to any edges. Now we have a finite bipartite graph H in which each vertex in W_{135} and W_{24} has an adjacent vertex.

The bounds on domination and time delay:
It is easy to see that there exists a constant L (resp., M) for the bound on domination size (resp., the bound on time delay) from the observations above. Specifically, it is sufficient that $L = |\{P_{i-2}, P_{i-1}, P_i\}| \cdot |\{\text{Rule 1, Rule 3, Rule 5}\}| = 9$ because each dominating event dominates at most one event for every three rules and every three processes, and $M = 2$ because a dominating event occurs before the next two events in W_{135} occur at the same process.

By construction of H, we can see that the part of Dijkstra's token ring of SSRmin converges in $O(n^2)$ steps because (1) $T_1 = O(n^2)$, and (2) $3n(n-1)/2$ events by Rules 2 and 4 occur in configurations $\gamma_0, \gamma_1, \cdots, \gamma_{T_1}$. □

Theorem 2 *For any initial configuration γ_0 and for any execution starting from γ_0, SSRmin converges within $O(n^2)$ steps under the unfair distributed daemon.*

Proof. For any initial configuration γ_0, the part of Dijkstra's token ring converges in $O(n^2)$ steps by Lemma 8, and remains so for any execution thereafter. Then, SSRmin converges in $O(n^2)$ steps by Lemma 7. In total, $O(n^2)$ is the worst case time complexity of SSRmin for convergence. □

5 Execution issue in the message-passing model

We investigate the behavior of the proposed algorithm when it is executed in the message-passing model by a transformation method of existing work. Below, we use a term *node* to say a physical device that emulates a *process* of the proposed algorithm, and we use a symbol v_i for a node which corresponds to process P_i for each $0 \leq i < n$, however, we use v_i and P_i interchangeably. We assume that each communication link can transmit only one message in each direction at a time. In other words, a node v_i can send a message to its neighbor node v_j only if there is no message transiting on the communication link from v_i to v_j.

The computational model that the proposed algorithm assumes is the state-reading model for communication, the composite atomicity model for granularity of execution unit, and the distributed daemon for execution scheduling. It seems that the computational model assumed is far from the real environment such as a network of IoT devices with wireless message-passing communication. Several works exist to fill the gap of computational models, such as [5, 7, 16, 17]. Algorithm 4 shows the outline of the transformation scheme, called cached sensornet transform (CST), proposed in [5], and this method is used in this paper. The main idea of the transformation is that (1) each process has a cache of local variables of neighbors, and (2) the value of a local variable is transmitted to neighbors when it is updated and periodically. Note that it is important for self-stabilization of real network that the value of local variables is periodically transmitted to neighbors to fix incorrect cache contents.

Algorithm 4 Cached sensornet transform (CST) to execute in the message-passing model for each v_i [5]

Constant
  N_i: a set of neighbor node of v_i
Variable;
  q_i — the (set of) local variable(s) of the original algorithm
  $Z_i[v_k]$ — a cache of q_k for each $v_k \in N_i$
Action
  on receipt of message $\langle \mathsf{state}, q \rangle$ **from** $v_k \in N_i$
    $Z_i[v_k] \leftarrow q$;
    Execute a rule and update q_i (access the cache $Z_i[v_k]$ instead of q_k);
    send $\langle \mathsf{state}, q_i \rangle$ **to each** $v_k \in N_i$;
  on interval timer
    send $\langle \mathsf{state}, q_i \rangle$ **to each** $v_k \in N_i$;

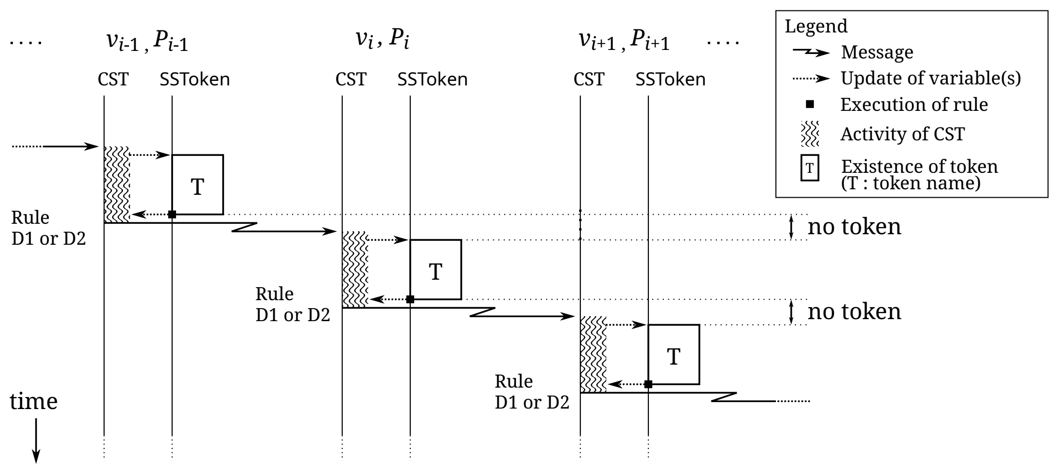

Figure 11: Token extinction of SSToken in the message-passing model ('T' : the token)

The important issue that we must be aware of for the transformed version is that the cache at each node may not be the latest. That is why an update of the local variable of P_i is not instantly reflected to cache at every neighbor, the granularity of execution of a node is different from the composite atomicity model, and the nodes that are executed are different from the distributed daemon model.

Let us present the notion of cache-coherence.

Definition 2 *(Cache-coherence [5]) We say that cache is* coherent *if and only if each node v_i holds the latest value of local variable q_k of each node v_k, i.e., $\forall v_i, v_k \in N_i : Z_i[v_k] = q_k$.* □

An algorithm that reaches a fixed point of configuration (called *silent*), which means that no process is enabled in legitimate configuration and the configuration does not change, execution of CST eventually becomes cache-coherent and remains so thereafter. It is shown that the transformed version of silent algorithms by CST converges with some randomization factor in execution timing [5, 17], and once cache becomes coherent it remains so thereafter.

On the other hand, algorithms for token circulation and mutual inclusion and exclusion algorithms, such as the proposed algorithm SSRmin, never reach a fixed point of configuration (called *non-silent*). In executions of such algorithms, coherence and non-coherence of a cache may be repeated infinitely many times. So, we need careful verification for the proposed algorithm. As we mentioned above, a token is defined by a predicate on local variables, and it is true even in the

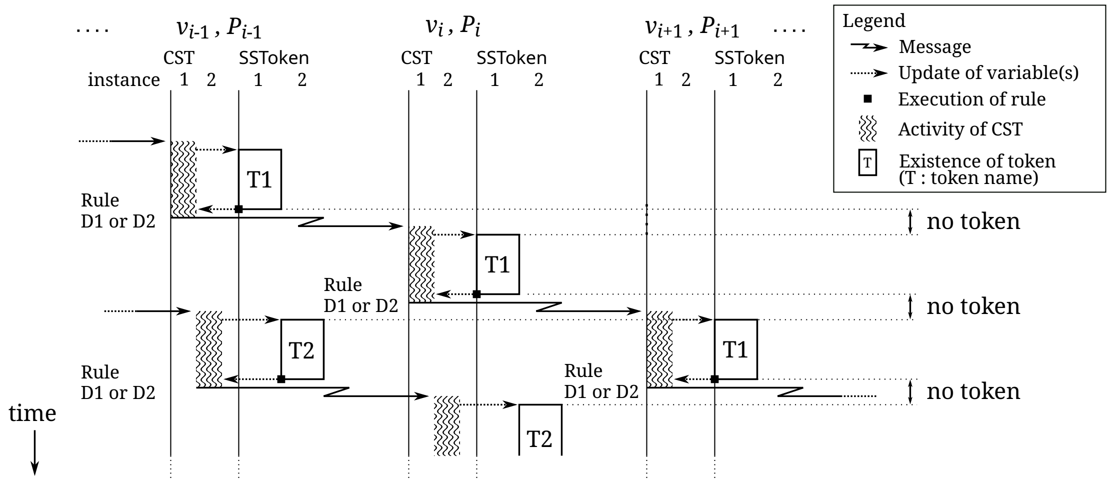

Figure 12: Concurrent executions of two instances of SSToken in the message-passing model ('T1' : the token by instance 1, 'T2' : the token by instance 2)

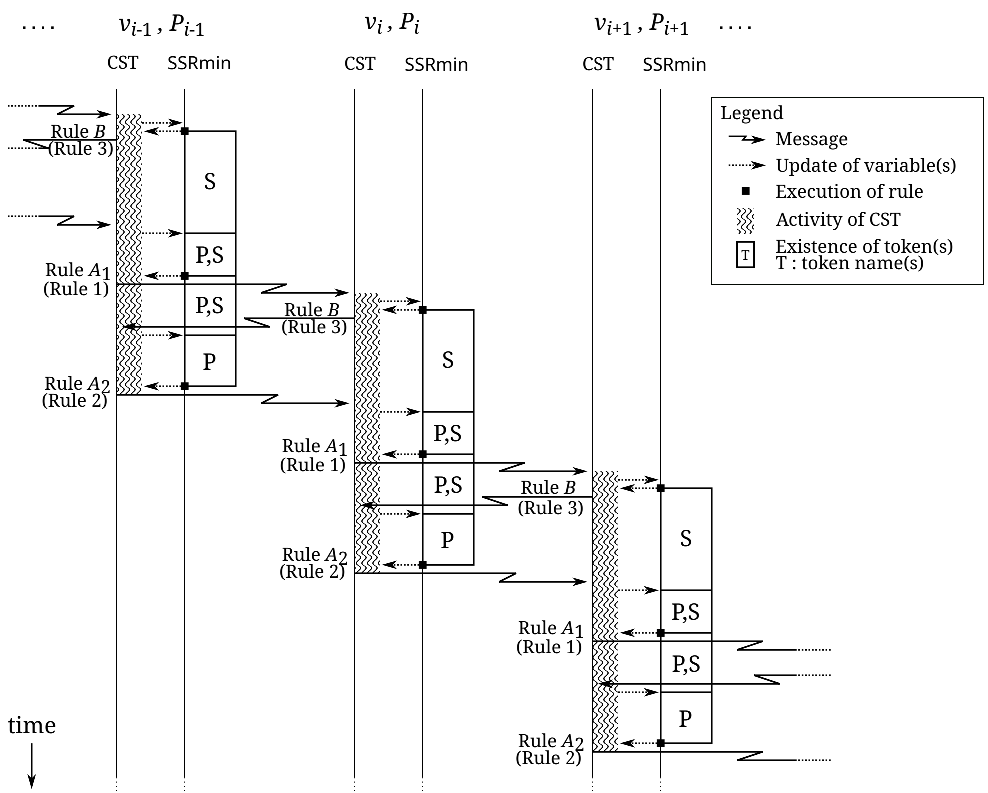

Figure 13: Mutual inclusion by SSRmin with the model gap tolerant property in the message-passing model ('P' : the primary token, 'S' : the secondary token)

message-passing version by CST. So, each process decides whether it holds a token or not by a predicate on values of local variables of itself and *cache* of neighbors. A phrase 'a process sends a token' does *not* mean sending a virtual token object but just updating local variables, and sending a message is used only to update the cache of neighbors.

Consider Dijkstra's token ring which is executed in the state-reading model as presented in Algorithm 1. Suppose that process P_i holds a token, and it executes a rule. Then, P_i releases the token and its neighbor P_{i+1} immediately receives it without any time instant that no process holds the token. Let us observe a transformed version of Dijkstra's token ring which is executed in the message-passing model. Suppose similarly that P_i holds a token, and it executes a rule. Then, according to Algorithm 4 and as shown in Figure 11, P_i releases the token, however, P_{i+1} does not hold it immediately; there is some time period between the release by P_i and the receipt by P_{i+1}. That is, the cache is not coherent during such a time period. Although there is at least one (specifically, exactly one) token at any time instant in the state reading model, we cannot use the transformed version of Dijkstra's token ring as a mutual inclusion in the message-passing model by CST shown in Algorithm 4. Furthermore, as illustrated in Figure 12, we cannot use two instances of the transformed version of Dijkstra's token ring executed independently and concurrently as a mutual inclusion algorithm because we have a time instant such that there is no token if two nodes execute the rule at the same time. For the same reason, the algorithm proposed [3] for a ring with multiple token is not sufficient for our purpose. So we need some kind of control to guarantee mutual inclusion, and it is the main motivation of current work.

Such phenomena observed in the transformed version is originated in that the transformation scheme does not exactly simulate the original computational model for the purpose of execution efficiency or low overhead at run-time. Let us call such difference of behavior as *model gap*. In our case of mutual inclusion, a mutual inclusion algorithm which is correct in the state-reading model is not a correct one in the message-passing model by the transformation scheme CST. Below, let us show that the proposed algorithm SSRmin is *model gap tolerant* in a sense that it is also correct in the message-passing model by the transformation scheme CST. We formally show definitions of the model gap below. Although it can be defined in a general form for arbitrary networks, we present a definition of a bidirectional ring for simplicity of description.

Definition 3 *(Model gap) For each node $v_i \in V$, let h_i be a function such that $h_i : Q_i \times Q_{i-1} \times Q_{i+1} \to D_i$ for some set D_i, and let h be a function $h : D_0 \times D_1 \times \cdots \times D_{n-1} \to H$ for some set H. Let $Z_i[v_k]$ be the cache of local variables of v_k at node v_i. We say that an algorithm is* model gap tolerant *with respect to h_i $(0 \leq i < n)$ and h if and only if, for each configuration $\gamma_t = (q_0, q_1, ...q_{n-1})$ and cache contents $Z.[\cdot]$ that appear in the execution starting from legitimate configuration with cache-coherence, the following equation holds.*

$$\begin{aligned} & h(h_0(q_0, q_{n-1}, q_1), h_1(q_1, q_0, q_2), ..., h_{n-1}(q_{n-1}, q_{n-2}, q_0)) \\ & \quad = \quad h(h_0(q_0, Z_0[v_{n-1}], Z_0[v_1]), h_1(q_1, Z_1[v_0], Z_1[v_2]), ..., h_{n-1}(q_{n-1}, Z_{n-1}[v_{n-2}], Z_{n-1}[v_0])) \end{aligned}$$

Otherwise, we say that the algorithm has a model gap. □

Intuitively, in the case of the proposed algorithm SSRmin, h_i is a boolean function such that "v_i holds a token", and h is a boolean function such that "at least one node holds a token". The concept of model gap tolerance formalizes that, despite cache contents may not be coherent temporarily, a correctness measure is not violated in the message-passing version, *e.g.,* an existence of a token at any time in case of SSRmin.

Let $\gamma_0 \in \Lambda$ be any legitimate configuration and observe an execution starting from γ_0. (Recall that Figure 4 shows an example of execution starting from a legitimate configuration.) So, we observe, in the message-passing model, whether at least one node holds a token or not even if an updated local state is transmitted with a delay as shown in Figure 13.

Theorem 3 *Staring from any legitimate configuration with cache-coherence. Then, the transformed version of the proposed algorithm SSRmin in the message-passing model guarantees that the number* *of nodes that hold a token is at least one and at most two. That is, the proposed algorithm is model gap tolerant.*

Proof. In legitimate configurations, as we observed in the proof of Lemma 1, exactly one process is enabled. The rules that make a process enabled are Rules 1, 2 and 3. We define a term *transient period* as a time duration between (1) an event that a process updates its local state and (2) an event that its neighbors receive a new local state by receiving a message. We verify the number of tokens in the transient period for each execution of a rule. We observe, one by one, each execution of a rule as follows.

- Rule 1 (or A_1; P_i sends the secondary token) : When P_i is enabled by the rule, it holds the primary and the secondary tokens. The local state of P_i is a form of $x.0.1$ for some $0 \leq x < K$, and it is changed to $x.1.0$ by execution of Rule 1. Just after P_i executes a rule, in the transient period, P_i holds the two tokens.

  The local state of P_i is transmitted to P_{i+1} by CST, and it is eventually received and cached at P_{i+1}. Then, P_{i+1} is enabled by Rule 3. Even if a message that contains the local state of P_i is lost by some fault, CST periodically transmits the local state and eventually, the local state is successfully received and cached at P_{i+1}. At the same time, the system becomes cache-coherent again.

- Rules 3 (or B; P_{i+1} receives the secondary token) : When P_{i+1} is enabled by the rule, its local state is a form of $x.0.0$. Its neighbor P_i has local state $x.1.0$, and it holds the primary and the secondary tokens. By execution of the rule, local state of P_{i+1} is changed from $x.0.0$ to $x.0.1$. Just after P_{i+1} executes the rule, in the transient period, P_{i+1} holds the secondary token, and at the same time, P_i holds the primary and the secondary token because of its local cache.

  The local state of P_{i+1} transmitted to P_i is eventually received and cached, and then, P_i is enabled by Rule 2. At the same time, the system becomes cache-coherent again.

- Rules 2 (or A_2; P_i sends the primary token): When P_i is enabled by the rule, P_i (resp., P_{i+1}) holds the primary (resp., secondary) token. Local state of P_i is a form of $x.1.0$, and it is changed to $x + 1.0.0$ by execution of the rule. Just after P_i executes the rule, in the transient period, P_i holds no token, and P_{i+1} holds the secondary token. When P_{i+1} receives the local state of P_i, P_{i+1} holds the primary token.

  The local state of P_i transmitted to P_{i+1} is eventually received and cached, and then, P_{i+1} holds the primary and the secondary tokens. At the same time, the system becomes cache-coherent again.

So far, we observed that, in a transient period, the number of processes that hold a token is at least one and at most two. After the transient period is over, the system becomes cache-coherent and the hypothesis of the theorem holds again. Hence the proposed algorithm is model gap tolerant. □

As we observed in the proof of Theorem 3, cache status alternates coherence and incoherence in an execution that starts from a legitimate configuration with cache-coherence. We classify the incoherence of cache into two types: good and bad. We say that cache-incoherence is *good* if it appears in an execution starting from a legitimate configuration with cache-coherence. Otherwise, we say that cache-incoherence is *bad*. The next lemma proves, using the proof technique resented in [5, 17], that any execution that starts from arbitrary, possibly illegitimate, configuration with bad cache-incoherence eventually reaches a legitimate configuration with cache-coherence, which means that the hypothesis of Theorem 3 is satisfied. Then, bad cache-incoherence never appears thereafter. In the following lemma and theorem, we assume that message loss events occur uniformly at random. This assumption is a sufficient condition for ease of probabilistic analysis and not a necessary condition.

Lemma 9 *Assume that events of message loss occur uniformly at random. Starting from an arbitrary configuration and arbitrary cache values, the proposed algorithm* SSRmin *eventually reaches legitimate a configuration with cache-coherence.*

Proof. There exists a time period in which no message loss occurs enough long time for convergence of SSRmin. If the local state is transmitted to a neighbor without message loss, the cache value of the neighbor becomes correct. Hence cache eventually becomes coherent. Because SSRmin assumes the distributed daemon, execution of two (or more) nodes at the same time does not prevent the convergence. So, eventually, the configuration becomes legitimate. □

From Lemma 9 and Theorem 3, we have the following theorem.

Theorem 4 *Assume that events of message loss occur uniformly at random. Starting from an arbitrary configuration and arbitrary cache values, the proposed algorithm* SSRmin *eventually reaches a configuration in which the number of nodes that hold a token is at least one and at most two, and remains so forever.* □

6 Conclusion

We proposed a self-stabilizing token ring algorithm for bidirectional message-passing ring networks based on the K-state token ring proposed by Dijkstra [2]. It assumes the unfair distributed daemon which is a general process scheduler but under which algorithm design is difficult. It has interesting applications, for example, self-organizing IoT monitoring systems with continuous observation : by mutual inclusion, at least one node is guaranteed to be active to monitor the environment, and by self-stabilization, it tolerates transient faults and nodes can start in arbitrary initial states without global reset. To achieve such a distributed algorithm, we designed an algorithm in a higher level of a computational model, the state-reading model and we applied the transformation scheme. To guarantee the correctness in the transformed version, we introduced the concept of the model gap tolerance and proven that the proposed algorithm is model gap tolerant. Future tasks are design of a self-stabilizing mutual inclusion algorithm with model gap tolerance for general network topology, and application of the concept of model gap tolerance for other non-reactive algorithms. Specifically, instead of Dijkstra's token ring SSToken used as a base algorithm in SSRmin, using the superstabilizing mutual exclusion proposed in [15] as a base algorithm is an interesting task.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Numbers 19K11826, 19H04085, 19K11828, 19K11823 and Israel & Japan Science and Technology Agency (JST) SICORP (Grant #JPMJSC1806). The authors would like to thank anonymous reviewers for their careful reading of our manuscript and constructive comments.

References

[1] K. Altisen, S. Devismes, S. Dubois, and F. Petit. *Introduction to Distributed Self-Stabilizing Algorithms*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2019.

[2] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[3] M. Flatebo, A. K. Datta, and A. A. Schoone. Self-stabilizing multi-token rings. *Distributed Computing*, 8(3):133–142, 1995.

[4] T. Herman. Superstabilizing mutual exclusion. *Distributed Computing*, 13(1):1–17, 2000.

[5] T. Herman. Models of self-stabilization and sensor networks. In *Proceedings of the 5th International Workshop of Distributed Computing (IWDC)*, pages 205–214, 2003.

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

[7] S. T. Huang, L. C. Wuu, and M. S. Tsai. Distributed execution model for self-stabilizing systems. In *Proceedings of the 14th International Conference on Distributed Computing Systems (ICDCS)*, pages 432–439, 1994.

[8] H. Kakugawa. Mutual inclusion in asynchronous message passing distributed systems. *Journal of Parallel and Distributed Computing*, 77:95–104, March 2015.

[9] H. Kakugawa. On the family of critical section problems. *Information Processing Letters*, 115(1):28–32, January 2015.

[10] H. Kakugawa. A self-stabilizing distributed algorithm for local mutual inclusion. *Information Processing Letters*, 115:6–8, 2015.

[11] H. Kakugawa and S. Kamei. A self-stabilizing token circulation with graceful handover on bidirectional ring networks. In *Proceedings of the International Parallel and Distributed Processing Symposium Workshops (IPDPSW), The 23rd Workshop on Advances in Parallel and Distributed Computational Models (APDCM)*, pages 596–604, May 2021.

[12] S. Kamei and H. Kakugawa. An asynchronous message-passing distributed algorithm for the generalized local critical section problem. *Algorithms*, 10(2), March 2017.

[13] S. Kamei and H. Kakugawa. Asynchronous message-passing distributed algorithm for the global critical section problem. *International Journal of Networking and Computing*, 9(2):147–160, January 2019.

[14] S. Kamei and H. Kakugawa. A self-stabilizing distributed algorithm for the local $(1, |N_i|)$-critical section problem. *Concurrency and Computation: Practice and Experience,*, 33(12):e5628, December 2019.

[15] Y. Katayama, E. Ueda, H. Fujiwara, and T. Masuzawa. A latency optimal superstabilizing mutual exclusion protocol in unidirectional rings. *Journal of Parallel and Distributed Computing*, 62(5):865–884, 2002.

[16] M. Mizuno and H. Kakugawa. A transformation of self-stabilizing programs for distributed computing environments. In *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG), LNCS 1151*, pages 304 – 321, 1996.

[17] V. Turau and C. Weyer. Randomized self-stabilizing algorithms for wireless sensor networks. In *Proceedings of the First International Workshop on Self-Organizing Systems (IWSOS)*, pages 74–89, 2006.