A Universal Self-Stabilizing Mutual Exclusion Algorithm*

広島大学 角川 裕次 (Hirotsgu Kakugawa)
Hiroshima Univ.
九州大学 山下 雅史 (Masafumi Yamashita)
Kyushu Univ.

Abstract

A distributed system consists of a set of processes and a set of communication links. A distributed system is said to be self-stabilizing if it converges to a correct system state from arbitrary initial system states. A self-stabilizing system is considered to be a fault tolerant system, since it tolerates any kind and any finite number of transient failures.

In this paper, we investigate a class of networks on which the leader election and mutual exclusion problems can be solved. We show graph theoretical characterization of networks on which these problems assuming central and distributed daemons for execution scheduling model and registers for communication model.

1 Introduction

A distributed system consists of a set of processes and a set of communication links, each connecting a pair of processes. The leader election and mutual exclusion problems are fundamental problems in distributed systems, and they have been investigated extensively.

The concept of self-stabilization is introduced by Dijkstra in [3]. A distributed system is said to be self-stabilizing if it converges to a correct system state from arbitrary initial system states. A self-stabilizing system is considered to be a fault tolerant system, since it tolerates any kind and any finite number of transient failures. The self-stabilizing leader election and mutual exclusion problems are also fundamental problems in self-stabilizing distributed systems, and have been studied extensively.

Breaking symmetry is essential to solve these problems, since exactly one process must be elected as a leader or granted to enter a critical section. Thus, algorithms for these problems assume unique process identifiers, randomization, or special network topology. Study on self-stabilizing leader election and mutual exclusion problems have been focused on mainly the space complexity of processes and convergence time.

In [7], Yamashita and Kameda introduced a concept of view and discussed a possibility of leader election on anonymous networks. View is a tree-structures data that a process can obtain at best by communicating with other processes. It is shown that the leader election problems can be solved on a network if and only if each process has a unique view[7]. We use view to investigate possibility of self-stabilizing leader election and mutual exclusion. In [2], Boldi et al. discusses symmetry breaking with graph fibrations. In [1], Boldi proposes universal self-stabilizing algorithm based on graph fibrations.

In this paper, we investigate a class of networks on which the leader election and mutual exclusion problems can be solved assuming register communication model under central and distributed daemons. We show graph theoretical characterization of networks on which these problems assuming central and distributed daemons for execution scheduling model and registers for communication model.

This paper is organized as follows. In section 2, network model, computational model and view are defined. In section 3, we propose a self-stabilizing view computation algorithm is proposed. This algorithm will be used as a building block to form leader election and mutual exclusion algorithms. In section 4, we investigate possibility of leader election and mutual exclusion of networks under distributed daemon. We give a characterization of a class of network on which the problems can be solved in terms of views, and show an algorithms for these problems. In section 5, we investigate possibility of leader election and mutual exclusion of networks under central daemon. In section 6, we summarize the results in this paper and refer to the future task.

*This work is supported in part by the Ministry of Education, Science and Culture under grant No. 11780229 and 10205221.
2 Preliminary

2.1 Network Model

A system \( N = (G, id) \) is a pair of graph \( G = (V, E) \), where \( V = \{v_1, ..., v_n\} \) is a set of processes and \( E \subseteq V \times V \) is a set of bidirectional links, and \( id \) is a process labeling function.

Let \( n \) be the number of processes in \( N \), and by \( deg_G(v_i) \), we denote the degree of \( v_i \). Each process \( v_i \) has an identifier \( id_i \), which may or may not be unique. Processes are distinguished by number such as \( v_1, v_2, ... \) but it is used in discussions of this paper. Each process cannot use such number in algorithms they execute, but they can use \( id_i \).

Registers

A link is formed by a pair of incoming and outgoing registers. Processes use registers to communicate with each other. When a process \( v_i \) sends data to process \( v_j \) (when there is a link between them), process \( v_i \) write data into a register, and then process \( v_j \) read from the register. In this paper, we use two terms "link" and "register" interchangeably. A register can be thought as a shared variable of single writer and single reader.

For each link \((v_i, v_j) \in E\), there is a pair of incoming and outgoing registers, denote by \( Reg_{i}^{in}, Reg_{i}^{out} \), respectively. Each pair of registers is labeled by a positive integer. arbitrarily from one to \( deg_G(v_i) \). Register labeling is a set of labeling functions \( f_v \) for each process \( v, f = \{f_v \mid v \in V\} \), where \( f_v \) is a bijective function from the set of incident link \((v_i, v_j) \in E \) to a set of integers \( \{1, 2, ..., deg_G(v_i)\} \). Note that incoming and outgoing registers connecting the same neighbor process are labeled by the same number.

Information given to processes

Each process \( v_i \) is given identifier \( id_i \), which may or may not be unique. As a local information, a process knows the degree, i.e., the number of neighbor processes. In this paper, we consider following two cases:

- Each process knows the total number of processes in a network, denoted by \( n \). This information can be used in algorithms to be executed.
- Each process does not know \( n \).

Description of algorithms

An algorithm executed by processes is described by a set of guarded commands. A guard is a boolean function on local state of process and contents of incoming registers. A command is an assignment statement to change local state of a process and contents of outgoing registers, based on local state and contents of incoming registers. In this paper, we assume algorithms are deterministic, i.e., next local state is determined uniquely. When there is a guard evaluated to true at process \( v_i \), then we say that process \( v_i \) is privileged.

Scheduler

We assume distributed daemon and central daemon. A scheduler called distributed daemon selects arbitrary nonempty set of privileged processes to execute. A scheduler called central daemon selects only one privileged process to execute. We also assume execution of processes is fair, i.e., any privileged process is executed eventually.

Execution

State of a distributed system \( N \) is called configuration, which is defined by a tuple of local state of processes and contents of registers. Formally, a configuration \( c \) is a tuple \((q_1, q_2, ..., q_n, Reg_{1}^{out}, Reg_{2}^{out}, ..., Reg_{1}^{in}, Reg_{2}^{in}, ...),\)

where \( q_i \) is a local state of process \( v_i \), \( Reg_{i}^{out}, Reg_{i}^{in} \) is contents of each outgoing register, and \( Reg_{1}^{in}, Reg_{2}^{out}, ... \) is contents of each incoming register.

Execution of \( N \) is a sequence of atomic steps. An atomic step is a sequence of following three actions:

1. read contents of incoming registers,
2. compute next value local state and outgoing registers based on current value of local state and contents of input registers, and
3. write values to outgoing registers.

Let \( c = (q_1, ..., q_i, ..., q_n, Reg_{1}^{out}, ..., Reg_{n}^{in}, ...) \) be a configuration of \( N \). Then, configuration \( c' \) followed by \( c \) is \((q_1, ..., q_i, ..., q_n, Reg_{1}^{out'}, ..., Reg_{1}^{in'}, ...),\)

where \( q_i, Reg_{i}^{out}, Reg_{i}^{in} \) are local state, outgoing register, and incoming register, respectively, whose contents are changed by an atomic step. Note that the number of processes that are executed at a time depends of daemon. When we assume central daemon, exactly one process is selected to be execute, and when we assume distributed daemon, more than one processes may be executed.

An execution of \( N \) is described by a (possibly infinite) sequence of configurations \( c_1, c_2, c_3, ... \), where \( c_1 \) is an initial configuration and \( c_{i+1} \) follows \( c_i \) by execution of some processes.

2.2 Self-stabilization

Let \( Z \) be a predicate on configuration of a distributed system \( N \). A system \( N \) is called self-stabilizing (SS for short) with respect to \( Z \) if and only if the following conditions hold: There exists a integer \( k \) such that \( Z(c_k) \) is true for any fair execution \( c_1, c_2, c_3, ... \).
starting from initial configuration $c_1$ (convergence). In addition, $Z(C_i)$ is true for each $i \geq k$ (closure).

Intuitively, a system configuration of $N$ is guaranteed to transit to "correct" configuration (defined by property $Z$), and system configuration remains correct forever once it becomes correct.

A self-stabilizing system is considered to tolerate any kind of and any finite number of transient faults. Suppose a configuration $c$ just after transient faults finished. Condition of self-stabilization guarantees that any execution starting from $c$ eventually reaches to a correct configuration and configuration remains correct forever. Therefore, self-stabilization is one of theoretical frame works for fault-tolerant distributed systems.

### 2.3 The SS leader election problem

The leader election problem (or LE, for short) is a problem such that exactly one process is elected as a leader. The leader process must know that it is the leader, and non-leader processes must know that they are not.

Formally, let $Z^i_{LE}$ be a boolean function of process $v_i \in V$ on local state of $v_i$. Let $q_i$ be a local state of process $v_i$. A process $v_i$ is a leader if and only if $Z^i_{LE}(q_i)$ holds.

An algorithm is said to solve the SS leader election problem if and only if following conditions hold: For any initial configuration $c_1$ and any execution $c_1, c_2, c_3, \ldots$, there exists a process $v_i$ and an integer $k$ such that $Z^i_{LE}(q_i)$ and $\neg Z^i_{LE}(q_i)$ for each $i \neq k$ holds for each $c_k, c_{k+1}, c_{k+2}, \ldots$.

### 2.4 The SS mutual exclusion problem

The mutual exclusion problem (or MX, for short) is a problem such that each process enters its critical section one after another, and the number of process which is in its critical section is at most one at a time.

Formally, let $Z^i_{MX}$ be a boolean function of process $v_i \in V$ on local state of $v_i$. Let $q_i$ be a local state of process $v_i$. A process $v_i$ is in its critical section if and only if $Z^i_{MX}(q_i)$ holds.

An algorithm is said to solve the SS mutual exclusion problem if and only if following conditions hold: For any initial configuration $c_1$ and any execution $c_1, c_2, c_3, \ldots$, there exists an integer $k$ such that the number of process $v_i$ such that $Z^i_{MX}$ is at most one for each $c_k, c_{k+1}, c_{k+2}, \ldots$. In addition, for each process $v_i \in V$, $F^i_{MX}$ become true infinitely many times during an infinite execution $c_k, c_{k+1}, c_{k+2}, \ldots$.

### 2.5 Views

A process in a distributed system communicate with other process and gather information to achieve a task. Existence of a solution of a distributed problem depends on information each process can obtain.

View\cite{7} is a tree-structured information on distributed system that a process can obtain at best by communicating with other processes. Each process has its own view; view may be different on processes.

Let $T_f(v_i)$ be a view of process $v_i \in V$ with register labeling $f$. Each node of a view corresponds to a process in a distributed system. For each view node $x$, we denote a process corresponding to $x$ by $\bar{x}$. The root node $x$ of a view $T_f(v_i)$ corresponds to process $v_i(=\bar{x})$. For each view node $y$, $j$-th child node $z_j$ of a node $y$ corresponds to the $j$-th neighbor process\footnote{Process $\bar{x}$ is the end of a link labeled $j$ at process $\bar{y}$, i.e., $f_{\bar{y}}(\bar{x}, \bar{y}) = j$.} of $\bar{x}$ for each $1 \leq j \leq \deg(\bar{y})$. Thus, the number of children of a view node $x$ is $\deg_x(\bar{x})$. In general, there are more than one view node which correspond to a process $v$ for each $v \in V$.

Let $x$ be a node of view, and $y$ be a child of $x$. View node $x$ is labeled by $id(\bar{x})$. An edge in a view from $x$ to $y$ is labeled by two labels $\ell$ (x's end) and $\ell'$ (y's end) as follows: $\ell$ is the register label of a link $(x, y) \in E$ at $\bar{x}$, and $\ell'$ is the register label of a link $(x, y) \in E$ at $\bar{y}$. In other words, $\ell = f_x(\bar{x}, \bar{y})$, and $\ell' = f_y(\bar{x}, \bar{y})$ for a register labeling $f$.

By $T_f(N)$, we denote a set of views of processes, i.e., $T_f(N) = \{ T_f(v) \mid v \in V \}$. By $T^d_f(v)$, we denote a view of process $v$ truncated to depth $d \geq 0$, and by $T_f(N)$, we denote a set of truncated views, i.e., $T^d_f(N) = \{ T^d_f(v) \mid v \in V \}$. When $N$ are obvious from context, we may omit $N$ and denote a set of views of processes by $T_f$.

In this paper, we use a term process as a abstraction of computer in a distributed system, and a term node as a graph theoretical node in a view. In addition, we use a term link for a abstraction of communication device in a distributed system, and a term edge for a graph theoretical edge in a view tree.

### 2.6 Universal algorithms

Let $Nets(A, P)$ be a class of networks $N$ such that algorithm $A$ solves a problem $P \in \{ MX, LE \}$ on $N$ for any register labeling $f$. (Note that there may be a network $N \notin Nets(A, P)$ that an algorithm $A$ solves a problem $P$ for some register labeling $f$.) Formally, $Nets(A, P)$ is defined as follows:

$Nets(A, P) = \{ N \mid A \text{ solves } P \text{ on } N \text{ for any register labeling } f \}$
Let \( N_{\mathrm{LE}} (N_{\mathrm{MX}}) \) be a class of networks \( N \) such that there exists an algorithm \( A \) which solves the leader election problem (the mutual exclusion problem) on \( N \) for any register labeling \( f \). Similarly, let \( N_{\mathrm{LE}}^{\mathrm{SS}} (N_{\mathrm{MX}}^{\mathrm{SS}}) \) be a class of networks \( N \) such that there exists an algorithm \( A \) which solves the self-stabilizing leader election problem (the self-stabilizing mutual exclusion problem) on \( N \) for any register labeling \( f \).

An algorithm \( A \) for problem \( P \) is universal if and only if \( \text{Nets}(A, P) = N_{\mathrm{P}} \) holds. For example, an algorithm \( A_{\mathrm{LE}} \) is universal if and only if following condition holds: \( \text{Nets}(A_{\mathrm{LE}}, \mathrm{LE}) = N_{\mathrm{LE}} \).

## 3 A Self-Stabilizing View Construction Algorithm

In this section, we propose a self-stabilizing view construction algorithm under distributed daemon. Lemma 4 in [8] states that each process \( v \in V \) can compute \( \mathcal{T}_{\mathrm{f}}^{2(n-1)}(N) \) from \( \mathcal{T}_{\mathrm{f}}^{2(n-1)}(v) \). Based on this property, we show a self-stabilizing algorithm to compute \( \mathcal{T}_{\mathrm{f}}^{2(n-1)}(v) \) at each process \( v \in V \). Since truncation depth of view and register labeling is clear from context, we denote a truncated view of \( v \) by \( T_{i} \).

This algorithm constructs a tree at each process; in a view tree \( T_{i} \) (of height \( 2(n - 1) \)) at process \( v_{i} \), the root node is labeled \( \text{id}(v_{i}) \) and \( j \)-th child of the root node is neighbor process of \( v_{i} \) which is other end of \( j \)-th outgoing/incoming registers.

The outline of the algorithm is as follows. A process compares each subtree of view tree and view of corresponding neighbor process. If they are different, copy a view of neighbor process and reconstruct a subtree of view.

### 3.1 The algorithm

#### Local variables of each process \( v_{i} \):
- \( T_{i} \) : \( v_{i} \)'s view.
- \( \text{id}_{i} \) : \( v_{i} \)'s local information. This value may or may not unique.

#### Network information of each process \( v_{i} \):
- \( N_{i} \) : The number of neighbor processes of \( v_{i} \).

Note that labeling for incoming and outgoing links for a neighbor process \( v_{j} \) of \( v_{i} \) may different, in general.

#### Functions:
- \( \text{GetRoot}(T_{i}) \) – Return a root node of a tree \( T_{i} \).
- \( \text{Height}(T_{i}) \) – The height of a tree \( T_{i} \).

- \( \text{Child}(T_{i}, j) \) – Return a subtree rooted by the \( j \)-th child of the root node of \( T_{i} \).
- \( \text{Cut}(T_{i}, d) \) – Return a tree with cutting subtrees of \( T_{i} \) whose depth is greater than \( d \). (The height of obtained tree is at most \( d \).)
- \( \text{SetChild}(T_{i}, j, T_{j}, j') \) – Reconstruct a tree by substituting \( T_{j} \) for the \( j \)-th child of the root node of \( T_{i} \). Mark that \( v_{i} \) is the \( j' \)-th neighbor at \( v_{j} \), where \( v_{j} \) is the \( j \)-th neighbor of \( v_{i} \).
- \( \text{NChildren}(T_{i}) \) – Return a number of children of the root node of \( T_{i} \).

The algorithm SS-View is shown in Figure 1. In this figure, each process \( v_{i} \) writes a pair of its local view \( T_{i} \) (which may under construction) and local label of output link to a neighbor \( j \) into \( \text{Reg}_{j}^{\text{Out}} \).

## 4 Distributed Daemon

In this section, we investigate a condition that the self-stabilizing leader election and mutual exclusion problems can be solved on networks when we assume distributed daemon as a scheduler.

When a network is symmetric, it is easy to see that there is no algorithm that solve the self-stabilizing LE and MX problems. As a metric of symmetry of a network, symmetricity is introduced in [7].

**Definition 1** [7] For register labeling \( f \) of network \( N = (G(V, E), \text{id}) \) of size \( n \), we define \( s_{f} \) by \( s_{f} = n/|T| \) The symmetricity of a network \( N \) under distributed daemon, denoted by \( \sigma_{d}(N) \), is defined as \( \sigma_{d}(N) = \max\{s_{f} \mid f \text{ is a register labeling for } G\} \)

In this paper, we assume that each process knows \( n \) (the number of processes in a system).

**Theorem 1** The leader election problem can be solved if and only if \( \sigma_{d}(N) = 1 \) under distributed daemon, and there exists a universal algorithm.

**Theorem 2** The mutual exclusion problem can be solved for any register labeling \( f \) if and only if \( \sigma_{d}(N) = 1 \) under distributed daemon when each process knows \( n \).

**Corollary 1** The mutual exclusion problem can be solved if and only if the leader election problem can be solved for any register labeling \( f \) under distributed daemon.

**Corollary 2** The leader election problem can be solved if and only if \( s_{f} = 1 \) on a network \( N \) with register labeling \( f \).

**Corollary 3** The mutual exclusion problem can be solved if and only if \( s_{f} = 1 \) on a network \( N \) with register labeling \( f \).
5 Central Daemon

When we assume distributed daemon, possibility of solving the leader and the mutual exclusion problems is determined by symmetricity \( \sigma_d(N) \). Since distributed daemon may execute all the processes, the number processes with the same view defines symmetricity of a network. When we assume central daemon, on the other hand, there may be a chance to break symmetry. Since central daemon selects only one process to execute at a time, two neighboring processes \( v_i, v_j \) with the same view can change their local state to be different each other.

In this section, we introduce a notion of symmetricity under central daemon denoted by \( \sigma_c(N) \), and discuss possibility of solving the leader election and the mutual exclusion problems. Note that it is obvious that both problems are solved under central daemon if \( \sigma_d(N) = 1 \).

Definition 2 Let \( s^f_i \) for a network \( N = (G(V,E), id) \) with register labeling \( f \) is defined as a minimum node coloring of a graph \( G = (V,E) \) satisfying the following conditions.

1. \( Q^f_1, Q^f_2, \ldots, Q^f_k \) be partitions of \( V \),
2. Each nodes in the same partition has the same view,
3. There is no link between nodes in the same partition, i.e., \( E \cap (Q^f_i \times Q^f_i) = \emptyset \) for each \( i \),
4. A subgraph induced by a node set \( Q^f_{i,j} = Q^f_i \cup Q^f_j \ (i \neq j) \) forms a regular bipartite graph, i.e., \( G_{i,j} = (Q^f_{i,j}, E \cap (Q^f_{i,j} \times Q^f_{i,j})) \) is a regular bipartite graph.

The symmetricity of a network \( N \) under central daemon, denoted by \( \sigma_c(N) \), is defined as \( \sigma_c(N) = \max_f \{ s^f \} \).

We have the following property.

Property 1 For any network \( N = (G(V,E), id) \), \( \sigma_c(N) \leq \sigma_d(N) \) holds.

Lemma 1 The leader election problem cannot be solved for any register labeling \( f \) if \( \sigma_c(N) > 1 \) under central daemon, even if each process knows \( n \).

Because a correct self-stabilizing algorithm under distributed daemon is also correct under central daemon, we can use algorithms in the previous section: If \( \sigma_d(N) = 1 \), which imply \( \sigma_c(N) = 1 \), we can use a leader election algorithm designed for distributed daemon to elect a unique leader under central daemon. Similarly, if \( \sigma_d(N) = 1 \), we can use a mutual exclusion algorithm designed for distributed daemon.
Macros:

\[ X_i = \{ x_j \mid j \in N_i \} : \]

a set of \( x_j \) of neighbor processes of process \( i \)

\[ d_i : \]

the number of incoming links at process \( i \)

Variables:

- integer \( x_i \):
- range of \( x_i \) is \( 0..d_i \).

\[ e[\]

// Rule 1. Obtain a locally unique \( x_i \)

\[ x_i \in X_i \rightarrow \]

\[ x_i = \min(\{0..d_i\} - X_i); \]

// Use SS-View.

// Become a leader
\[
\square T_i \text{ is the smallest view} \rightarrow \\
\text{process } i \text{ is a unique leader}
\]

\[
\square T_i \text{ is not the smallest view} \rightarrow \\
\text{process } i \text{ is not a unique leader}
\]

Figure 2: A universal self-stabilizing leader election algorithm

... can be used for mutual exclusion under central daemon.

Now we consider a case when \( \sigma_c(N) = 1 \) and \( \sigma_d(N) > 1 \). We show that there exists a universal algorithm for the leader election problem.

**Lemma 2** For a network \( N \) with register labeling \( f \), \( |Q^f_i| \) is greater than 1 for each \( i \), if \( |Q^f_j| > 1 \) for some \( j \). □

**Lemma 3** There exists a universal leader election algorithm for any register labeling \( f \) if \( \sigma_c(N) = 1 \) under central daemon when each process knows \( n \).

**Proof:** See Figure 2. □

We have the following theorem.

**Theorem 3** There exists a universal leader election algorithm if and only if \( \sigma_c(N) = 1 \) under central daemon when each process knows \( n \). □

We can obtain following theorem by a similar discussion.

**Theorem 4** The mutual exclusion problem can be solved if and only if \( \sigma_c(N) = 1 \) under central daemon when each process knows \( n \). □

**Corollary 4** Let \( R_n \) be a bidirectional ring network of size \( n \) without sense of direction, where \( n \) is prime number. Then, there exists a leader election algorithm and a mutual exclusion algorithm for \( R_n \) under central daemon. □

### 6 Conclusion

In this paper, we discussed conditions that self-stabilizing leader election and mutual exclusion problems can be solved. We proposed a self-stabilizing view construction algorithm which can be used to construct a self-stabilizing leader election algorithm. Based on this algorithm, we showed a self-stabilizing mutual exclusion algorithm. Symmetry proposed in [7] is defined for an execution model such that all processes are executed at each step. We proposed a symmetry for central daemon which can be used to discuss possibility of leader election by central daemon scheduler.

### References


