数理解析研究所講究録 1205 巻 2001 年 107-112

A Universal Self-Stabilizing Mutual Exclusion Algorithm*

広島大学　角川 裕次　(Hirotsugu 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 extensibly.

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 th 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 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 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_j^{In}, Reg_j^{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: $\mathbf{f} = \{f_v \mid v \in V\}$, where f_v is a bijective function from the set of incident link $(v, v_j) \in E$ to a set of integers $\{1, 2, ..., deg_G(v)\}$. 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 unique. As a local information, a process known 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 $\langle q_1, q_2, ..., q_n, Reg_1^{\mathrm{Out}}, Reg_2^{\mathrm{Out}}, ...Reg_1^{\mathrm{In}}, Reg_2^{\mathrm{In}}, ...\rangle$, where q_i is a local state of process v_i, $Reg_1^{\mathrm{Out}}, Reg_2^{\mathrm{Out}}, ...$ is contents of each outgoing register, and $Reg_1^{\mathrm{In}}, Reg_2^{\mathrm{Out}}, ...$ 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 $c = \langle q_1, ..., q_i, ..., q_n, ...Reg_j^{\mathrm{Out}}, ...Reg_j^{\mathrm{In}}, ...\rangle$ be a configuration of N Then, configuration c' followed by c is $\langle q_1, ..., q_i', ...q_n, ...Reg_j^{\mathrm{Out}\prime}, ...Reg_j^{\mathrm{In}\prime}, ...\rangle$, where $q_i', Reg_j^{\mathrm{Out}}$ and Reg_j^{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 $Z^i_{\mathrm{LE}}(q_i)$ 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_ℓ and an integer k such that $Z^\ell_{\mathrm{LE}}(q_\ell)$ and $\neg Z^i_{\mathrm{LE}}(q_i)$ for each $i \neq \ell$ 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 $Z^i_{\mathrm{MX}}(q_i)$ 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 te 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[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 $T_{\mathbf{f}}(v_i)$ be a view of process $v_i \in V$ with register labeling $\mathbf{f}$. 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 $\overline{x}$. The root node x of a view $T_{\mathbf{f}}(v_i)$ corresponds to process $v_i (= \overline{x})$. For each view node y, j-th child node z_j of a node y corresponds to the j-th neighbor process[1] of $\overline{x_j}$ for each $1 \leq j \leq deg(\overline{y})$. Thus, the number of children of a view node x is $deg_G(\overline{x})$. 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 $id(\overline{x})$. An edge in a view from x to y is labeled by two labels ℓ (x's end) and ℓ' (y's end) as follows: ℓ is the register label of a link $(\overline{x}, \overline{y}) \in E$ at $\overline{x}$, and ℓ' is the register label of a link $(\overline{x}, \overline{y}) \in E$ at $\overline{y}$. In other words, $\ell = f_{\overline{x}}(\overline{x}, \overline{y})$, and $\ell' = f_{\overline{y}}(\overline{x}, \overline{y})$ for a register labeling $\mathbf{f}$.

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

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

$$Nets(A, P) = \{N \mid A \text{ solves } P \text{ on } N \text{ for any register lebeling } \mathbf{f}\}$$

[1]Process $\overline{z_j}$ is the end of a link labeled j at process $\overline{y}$, i.e., $f_{\overline{y}}(\overline{y}, \overline{x_j}) = j$.

Let N_{LE} (N_{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 $\mathbf{f}$. 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 $\mathbf{f}$.

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

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 $\mathcal{T}_{\mathbf{f}}^{2(n-1)}(N)$ from $\mathcal{T}_{\mathbf{f}}^{2(n-1)}(v)$. Based on this property, we show a self-stabilizing algorithm to compute $\mathcal{T}_{\mathbf{f}}^{2(n-1)}(v)$ 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_i 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 $id(v_i)$ 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.
- $\boldsymbol{id_i}$: 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:

- *GetRoot*(T_i) – Return a root node of a tree T_i.
- *Height*(T_i) – The height of a tree T_i.
- *Child*(T_i, j) – Return a subtree rooted by the j-th child of the root node of T_i.
- *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.)
- *SetChild*(T_i, j, T_j, j') – Reconstruct a tree by substituting T' 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.
- *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 $\mathit{Reg}_j^{\mathrm{Out}}$.

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 schedular.

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* $\mathbf{f}$ *of network* $N = (G(V, E), id)$ *of size* n*, we define* $s_{\mathbf{f}}$ *by* $s_{\mathbf{f}} = n/|\mathcal{T}_{\mathbf{f}}|$ *The* symmetricity *of a network* N *under distributed daemon, denoted by* $\sigma_d(N)$*, is defined as* $\sigma_d(N) = \max\{s_{\mathbf{f}} \mid \mathbf{f} \text{ is a register labeling for } G\}$ □

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* $\sigma_d(N)$ *is* 1 *for any register labeling* $\mathbf{f}$ *under distributed daemon, and there exists a universal algorithm.* □

Theorem 2 *The mutual exclusion problem can be solved for any register labeling* $\mathbf{f}$ *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* $\mathbf{f}$ *under distributed daemon.* □

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

Corollary 3 *The mutual exclusion problem can be solved if and only if* $s_{\mathbf{f}} = 1$ *on a network* N *with register labeling* $\mathbf{f}$. □

```
macro UpdateRegisters:
  for each 1 ≤ j ≤ N_i
    Reg_j^Out := ⟨T_i, j⟩;

macro RegistersAreCorrect:
  ∀j(1 ≤ j ≤ N_i)[Reg_j^Out = ⟨T_i, j⟩]

*[
  // Rule 1. Obtain a view of neighbor
  // and construct a view.
  RegistersAreCorrect ∧ (GetRoot(T_i) = id_i)
   ∧ (NChildren(T_i) = N_i)
   ∧ (Height(T_i) ≤ 2(n − 1))
   ∧ ∃j ∈ N_i[(Child(T_i, j) ≠ Cut(T_j, 2(n − 1) − 1))
                  ∧ (Height(T_j) ≤ 2(n − 1))] →
    SetChild(T_i, j, Cut(T_j, 2(n − 1) − 1));
    UpdateRegisters;

  // Rule 2. Fix an incorrect view.
□ RegistersAreCorrect ∧ (GetRoot(T_i) = id_i)
   ∧ (NChildren(T_i) = N_i)
   ∧ (Height(T_i) > 2(n − 1)) →
    T_i = Cut(T_i, 2(n − 1));
    UpdateRegisters;

  // Rule 3. Fix an incorrect view.
□ RegistersAreCorrect ∧ (GetRoot(T_i) = id_i)
   ∧ (NChildren(T_i) ≠ N_i) →
    T_i := "a tree whose root is a node labeled id_i
              with N_i children labeled nothing";
    UpdateRegisters;

  // Rule 4. Fix an incorrect view.
□ RegistersAreCorrect ∧ GetRoot(T_i) ≠ id_i →
    T_i := "a tree whose root is a node labeled id_i
              with N_i children labeled nothing";
    UpdateRegisters;

  // Rule 5. Fix an incorrect view.
□ ¬RegistersAreCorrect →
    T_i := "a tree whose root is a node labeled id_i
              with N_i children labeled nothing";
    UpdateRegisters;
]
```

Figure 1: SS-View

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 $s^c_{\mathbf{f}}$ for a network $N = (G(V,E), id)$ with register labeling $\mathbf{f}$ is defined as a minimum node coloring of a graph $G = (V,E)$ satisfying the following conditions.*

1. *$Q^{\mathbf{f}}_1, Q^{\mathbf{f}}_2 ... Q^{\mathbf{f}}_k$ 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., $E \cap (Q^{\mathbf{f}}_i \times Q^{\mathbf{f}}_i) = \emptyset$ for each i.*
4. *A subgraph induced by a node set $Q^{\mathbf{f}}_{i,j} = Q^{\mathbf{f}}_i \cup Q^{\mathbf{f}}_j$ $(i \neq j)$ forms a regular bipartite graph, i.e., $G_{i,j} = (Q^{\mathbf{f}}_{i,j}, E \cap (Q^{\mathbf{f}}_{i,j} \times Q^{\mathbf{f}}_{i,j})$ is a regular bipartite graph.*

The symmetricity *of a network N under central daemon, denoted by $\sigma_c(N)$, is defined as $\sigma_c(N) = \max_{\mathbf{f}}\{s^c_{\mathbf{f}}\}$.* □

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 $\mathbf{f}$ 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 $0..d_i$.

```
*[
// Rule 1. Obtain a locally unique x_i
  x_i ∈ X_i →
    x_i = min({0..d_i} − X_i);

// Use SS-View.

// Become a leader
□ T_i is the smallest view →
    process i is a unique leader
□ T_i is not the smallest view →
    process i 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 $\mathbf{f}$, $|Q_i^{\mathbf{f}}|$ is greater than 1 for each i, if $|Q_j^{\mathbf{f}}| > 1$ for some j.* □

Lemma 3 *There exists a universal leader election algorithm for any register labeling $\mathbf{f}$ 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. Symmetricity proposed in [7] is defined for an execution model such that all processes are executed at each step. We proposed a symmetricity for central daemon which can be used to discuss possibility of leader election by central daemon schedular.

References

[1] Paolo Boldi. Self-stabilizing universal algorithms. In *Proceedings of the Second Workshop on Self-Stabilizing Systems (WSS97)*, pages –, 1997.

[2] Paolo Boldi, Bruno Codenotti, Peter Gemmell, and Janos Simon. Symmetry breaking in anonymous networks: Characterization. In *Proceedings of the 4th Islael Symposium on Theory of Computing and Systems (ISTCS96)*, pages 16–26, 1996.

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

[4] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 9(1):3–16, 1993.

[5] Shing-Tsaan Huang. Leader election in uniform rings. *ACM Transactions on Programming Languages and Systems*, 15(3):563–573, July 1993.

[6] N. Norris. Universal covers of graphs: isomorphism to depth $n-1$ implies isomorphism to all depth. *Discrete Applied Mathematics*, 56:61–74, 1995.

[7] Masafumi Yamashita and Tsunehiko Kameda. Computing on anonymous networks: Part I: characterizing the solvable cases. *IEEE Transactions on Parallel and Distributed Systems*, 7(1):69–89, January 1996.

[8] Masafumi Yamashita and Tsunehiko Kameda. Leader election problem on networks in which processor identity numbers are not distinct. *IEEE Transactions on Parallel and Distributed Systems*, 10, 1999.