RAPPORT DE RECHERCHE

L R I

A SELF-STABILIZING LINK-COLORING PROTOCOL RESILIENT TO UNBOUNDED BYZANTINE FAULTS IN ARBITRARY NETWORKS

MASUZAWA T / TIXEUIL S

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud – LRI

01/2005

Rapport de Recherche N° 1396

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 490
91405 ORSAY Cedex (France)

A Self-Stabilizing Link-Coloring Protocol Resilient to Unbounded Byzantine Faults in Arbitrary Networks[1]

Toshimitsu Masuzawa* Sébastien Tixeuil*

* Osaka University, Japan, email: masuzawa@ist.osaka-u.ac.jp
* LRI-CNRS UMR 8623 & INRIA Grand Large, France, email: tixeuil@lri.fr

[1]This work is supported in part by a JSPS, Grant-in-Aid for Scientific Research((B)(2)15300017), and "The 21st Century Center of Excellence Program" of the Ministry of Education, Culture, Sports, Science and Technology, Japan. This work is also supported in part by the French Ministry of Research ACI projects FRAGILE and SR2I.

Abstract

Self-stabilizing protocols can tolerate any type and any number of transient faults. However, in general, self-stabilizing protocols provide no guarantee about their behavior against permanent faults. This paper proposes a self-stabilizing link-coloring protocol resilient to (permanent) Byzantine faults in arbitrary networks. The protocol assumes the central daemon, and uses $2\Delta - 1$ colors where Δ is the maximum degree in the network. This protocol guarantees that any link (u, v) between nonfaulty processes u and v is assigned a color within $2\Delta + 2$ rounds and its color remains unchanged thereafter. Thus, our protocol achieves Byzantine-fault tolerance with containment radius of one, which is trivially optimal.

Keywords

distributed protocol, self-stabilization, link-coloring, byzantine fault, fault tolerance, fault containment

Résumé

Les protocoles auto-stabilisants sont capables de tolérer tout type et tout nombre de fautes transitoires. Cependant, en général, les protocoles auto-stabilisants ne proposent aucune garantie vis à vis des défaillances permanentes. Dans cet article, nous proposons un algorithme de coloriage des liens qui est auto-stabilisant et tolère un nombre arbitraire de défaillances Byzantines dans des réseaux de topologie quelconque. Le protocole suppose un démon central, et utilise $2\Delta - 1$ couleurs, où Δ est le degré maximal du réseau. Ce protocole garantit que tout lien (u, v) entre deux processus non défaillants u et v est colorié en moins de $2\Delta+2$ tours et que sa couleur ne change plus par la suite. Donc, notre protocole tolère des défaillances Byzantines et cloisonne celles-ci à distance 1, ce qui est triviallement optimal.

Mots clef

algorithme réparti, auto-stabilisation, coloriage des liens, fautes Byzantines, tolérance aux fautes, cloisonnement des fautes.

Chapter 1

Introduction

Self-stabilization [4] is one of the most effective and promising paradigms for fault-tolerant distributed computing [5]. A self-stabilizing protocol is guaranteed to achieve its desired behavior eventually regardless of the initial network configuration (*i.e.*, global state). This implies that a self-stabilizing protocol is resilient to any number and any type of transient faults since it converges to its desired behavior from any configuration resulting from transient faults. However the convergence to the desired behavior is guaranteed only under the assumption that no further fault occurs during convergence.

There exists several researches on self-stabilizing protocols that are also resilient to permanent faults [1, 3, 6, 8, 2, 7, 9, 11]. Most of those consider only crash faults, and guarantee that each nonfaulty process achieves its intented behavior regardless of the initial network configuration. Nesterenko *et al.* [9] provide solutions that are self-stabilizing and tolerate unbounded Byzantine faults. The main difficulty in this setting is caused by arbitrary and unbounded state changes of the Byzantine process: processes around the Byzantine processes may change their states in response to the state changes of the Byzantine processes, and processes next to the processes changing their states may also change their states. This implies that the influence of the Byzantine processes could expand to the whole system, preventing every process from conforming to its specification forever. In [9], the protocols manage to contain the influence of Byzantine processes to only processes near them, the other processes being able to eventually achieve correct behavior. The complexity measure they introduce is the *containment radius*, which is the maximum distance between a Byzantine process and a processor affected by the Byzantine process. They also propose self-stabilizing protocols resilient to Byzantine faults for the vertex coloring problem and the dining philosophers problem. The containment radius is one for the vertex coloring problem and two for the dining philosophers problem. In [10], the authors consider a self-stabilizing link-coloring protocol resilient to Byzantine faults in oriented tree networks, achieving a containment radius of two. Link-coloring of the distributed system is an assignment of colors to the communication links such that no two communication links with the same color share a process in common. Link-coloring has many applications in distributed systems, *e.g.*, scheduling data transfer and assigning frequency band in wireless networks. From a Byzantine containment point of view, link

coloring is harder than vertex coloring and dining philosophers for the following reason: while the two latter problems require only one process to take an action to correct a single fault, link colors result from an agreement of two neighboring nodes, and thus can result in the update of two nodes to correct a single failure.

In this paper, we present a self-stabilizing link-coloring protocol resilient to unbounded Byzantine faults. Unlike the protocol of [10], we consider abritrary anonymous networks, where no pre-existing hierarchy is available. As it was proved necessary in [10] to achieve constant containment radius, we assume the central deamon, *i.e.* exactly one process can execute an action at a given time. We use $2\Delta - 1$ colors, where Δ is the maximum degree in the network. Our protocol guarantees that any link (u, v) between nonfaulty processes u and v is assigned a color within $2\Delta + 2$ rounds and its color remains unchanged thereafter. As far as fault containment is considered, our protocol is optimal, since the influence of Byzantine processors is limited to themselves. Thus, our protocol also trivially achieves Byzantine-fault containment with containment radius of one.

Chapter 2

Preliminaries

2.1 Distributed System

A *distributed system* $S = (P, L)$ consists of a set $P = \{v_1, v_2, \ldots, v_n\}$ of processes and a set L of communication links (simply called links). A link is an unordered pair of distinct processes. A distributed system S can be regarded as a graph whose vertex set is P and whose link set is L, so we use some graph terminology to describe a distributed system S. A *subsystem* $S' = (P', L')$ of a distributed system $S = (P, L)$ is such that $P' \subseteq P$ and $L' = \{(u, v) \in L | u \in P', v \in P'\}$.

Processes u and v are called *neighbors* if $(u, v) \in L$. The set of neighbors of a process v is denoted by N_v, and its cardinality (the *degree* of v) is denoted by $\Delta_v (= |N_v|)$. The degree Δ of a distributed system $S = (P, L)$ is defined as $\Delta = \max\{\Delta_v \mid v \in P\}$. We do not assume existence of a unique identifier of each process. Instead we assume each process can distinguish its neighbors from each other by locally arranging them in some arbitrary order: the k-th neighbor of a process v is denoted by $N_v(k)$ $(1 \leq k \leq \Delta_v)$.

Each process is modeled by a state machine that can communicate with its neighbors through link registers. For each pair of neighboring processes u and v, there are two link registers $r_{u,v}$ and $r_{v,u}$. Message transmission from u to v is realized as follows: u writes a message to link register $r_{u,v}$ and then v reads it from $r_{u,v}$. The link register $r_{u,v}$ is called an *output register* of u and is called an *input register* of v. The set of all output (resp. input) registers of u is denoted by Out_u (resp. In_u), i.e., $Out_u = \{r_{u,v} \mid v \in N_u\}$ and $In_u = \{r_{v,u} \,| v \in N_u\}$.

For convenience, we use variables to denote process states and link register states, and we assume that processor code is given as one function (or action) that is regularly executed in an atomic manner. The execution of an action of u is called a *step* of u.

A global state of a distributed system is called a *configuration* and is specified by a product of states of all processes and all link registers. We define C to be the set of all possible configurations of a distributed system S. For each configuration $\rho \in C$, $\rho|u$ and $\rho|r$ denote the process state of u and the state of link register r in configuration ρ respectively. For a process u and two configurations ρ and ρ', we denote $\rho \overset{u}{\mapsto} \rho'$ when ρ changes to ρ'

by executing an action of u. Notice that ρ and ρ' can be different only in the states of u and the states of output registers of u.

A *schedule* of a distributed system is an infinite sequence of processes. Let $Q = u^1, u^2, \ldots$ be a schedule. An infinite sequence of configurations $e = \rho_0, \rho_1, \ldots$ is called an *execution* from an initial configuration ρ_0 by a schedule Q, if e satisfies $\rho_i \overset{u^{i+1}}{\mapsto} \rho_{i+1}$ for each i ($i \geq 0$). The set of possible schedules in a distributed system is sometimes modeled by a scheduler called a *daemon*. In this paper, we consider the *central daemon* where no two processes can execute their actions at the same time.

The set of all possible executions from an initial configuration $\rho_0 \in C$ is denoted by E_{ρ_0}. The set of all possible executions is denoted by E, that is, $E = \bigcup_{\rho \in C} E_\rho$. We consider *asynchronous* distributed systems where we can make no assumption on schedules except that any schedule is *weakly fair*: every process appears in the schedule infinitely often.

In this paper, we consider (permanent) *Byzantine faults*: a Byzantine process (i.e., a Byzantine-faulty process) can arbitrarily behave independently from its actions. If v is a Byzantine process, v can repeatedly change its variables and its output registers arbitrarily.

Let $BF = \{f_1, f_2, \ldots, f_c\}$ be the set of Byzantine processes. We call a proceess v ($\notin BF$) a *correct process*. In distributed systems with Byzantine processes, execution by a schedule $Q = u^1, u^2, \ldots$ is an infinite sequence of configurations $e = \rho_0, \rho_1, \ldots$ satisfying the following conditions.

- When u^{i+1} is a correct process, $\rho_i \overset{u^{i+1}}{\mapsto} \rho_{i+1}$ holds (possibly $\rho_i = \rho_{i+1}$).
- When u^{i+1} is a Byzantine process, $\rho_{i+1}|u^{i+1}$ and $\rho_{i+1}|r$ ($r \in Out_{u^{i+1}}$) can be arbitrary states. For any process v other than u^{i+1}, $\rho_i|v = \rho_{i+1}|v$ and $\rho_i|r = \rho_{i+1}|r$ ($r \in Out_v$) hold.

In asynchronous distributed systems, time is usually measured by *asynchronous rounds* (simply called *rounds*). Let $e = \rho_0, \rho_1, \ldots$ be an execution from configuration ρ_0 by a schedule $Q = u^1, u^2, \ldots$. The first round of e is defined to be the minimum prefix of e, $e' = \rho_0, \rho_1, \ldots, \rho_k$, such that $\{u^i \mid 1 \leq i \leq k\} = P$. Round t ($t \geq 2$) is defined recursively, by applying the above definition of the first round to $e'' = \rho_k, \rho_{k+1}, \ldots$. Intuitively, every process has a chance to update its state in every round.

2.2 Self-Stabilizing Protocol Resilient to Byzantine Faults

In this paper, we treat only *static problems*, *i.e.*, once the system reaches a desired configuration, the configuration remains unchanged forever. For example, the spanning-tree construction problem is a static problem, while the mutual exclusion problem is not [5]. Some static problems can be defined by a *specification predicate*, $spec(v)$, for each process v, which specifies the condition that v should satisfy at the desired configuration. A specification predicate $spec(v)$ is a boolean expression consisting of the variables of $P_v \subseteq P$ and link registers $R_v \subseteq R$, where R is the set of all link registers.

A self-stabilizing protocol is a protocol that guarantees each process v satisfies $spec(v)$ eventually regardless of the initial configuration. By this property, a self-stabilizing protocol can tolerate any number and any type of transient faults. However, since we consider permanent Byzantine faults, faulty processes may not satisfy $spec(v)$. In addition, non-faulty processes near the faulty processes can be influenced by the faulty processes and may be unable to satisfy $spec(v)$. Nesterenko and Arora [9] define a *strictly stabilizing protocol* as a self-stabilizing protocol resilient to Byzantine faults. Informally, the protocol requires each process v more than ℓ away from any Byzantine process to satisfy $spec(v)$ eventually, where ℓ is a constant called *stabilization radius*. A *strictly stabilizing protocol* is defined as follows.

Definition 1 *A configuration ρ_0 is a BF-*stable configuration with stabilizing radius ℓ *if and only if, for any execution $e = \rho_0, \rho_1, \ldots$ and any process v, the following condition holds:*

> *If the distance from v to any Byzantine process is more than ℓ, then for any i ($i \geq 0$) (i) v satisfies $spec(v)$ in ρ_i,* (ii) *$\rho_i|v = \rho_{i+1}|v$ holds, and* (iii) *$\rho_i|r = \rho_{i+1}|r$ ($r \in Out_v$) holds.*

Definition 1 states that, once the system reaches a stable configuration, a process v more than ℓ away from any Byzantine process satisfies $spec(v)$ and never changes the states of v and r ($r \in Out_v$) forever.

Definition 2 ([9]) *A protocol A is a* strictly stabilizing protocol with stabilizing radius ℓ *if and only if, for any execution $e = \rho_0, \rho_1, \ldots$ of A starting from any configuration ρ_0, there exists ρ_i that is a BF-stable configuration with radius ℓ. We say that the* stabilizing time *of A is k for the minimum k such that the last configuration of the k-th round is a BF-stable configuration in any execution of A.*

Definition 3 *A protocol A is* Byzantine insensitive *if and only if every process eventually satisfies its specification in $S' = (P', L')$, the subsystem of all correct processes.*

Notice that if a protocol is Byzantine insensitive, it is also strictly stabilizing with stabilizing radius of 1, but the converse is not necessarily true. So, the former property is strictly stronger than the latter.

2.3 Link-Coloring Problem

The *link-coloring problem* consists in assigning a color to every link so that no two links with the same color are adjacent to the same processor. In the following, let $CSET$ be a given set of colors, and let $Color(u, v) \in CSET$ be the color of link (u, v).

Definition 4 *In the link-coloring problem, the specification predicate $spec(v)$ for a process v is given as follows:*

$$\forall x, y \in N_v : x \neq y \Longrightarrow Color(v, x) \neq Color(v, y)$$

In the following, we denote a link-coloring protocol with b colors as a *b-link-coloring protocol*.

Chapter 3

Link-Coloring Protocol

3.1 Link-Coloring Protocol on arbitrary networks

Our protocol is presented as Algorithm 3.1.1. It is informally described as follows: each process maintains a list of colors assigned to its incident links and periodically exchanges the list with each neighboring process. From the list received from its neighbor v, a processor u can propose a color for the link (u, v). Since the set of colors is of size $2\Delta - 1$, u can choose a color that is not used at u or v. If both u and v are correct, once they settle on a color c for link (u, v), this color is never changed. In case of a Byzantine process, it may happen however, that a Byzantine process keeps proposing colors conflicting with other neighbors proposals. To ensure that this behavior may not occur infinitely often, we use a priority list so that neighbors of a particular node u get round robin priority when proposing conflicting colors.

3.2 Correctness Proof

Let u and v be neighboring processes, and let v be the k-th neighbor of u. We say that register $r_{u,v}$ is consistent if $PC_{u,v} = outCol_u(k)$ and $USET_{u,v} = \{outCol_u(m) \mid 1 \leq m \leq \Delta_u, m \neq k\}$ hold.

Lemma 1 *Once a correct process executes an action, its output registers becomes consistent and remain so thereafter.*

Proof By the code of the algorithm. □

Corollary 1 *In the second round and later, all output registers of correct processes are consistent.*

The following lemma also holds clearly.

Algorithm 3.1.1 The SS link-coloring protocol

```
constants
    Δ = the maximum degree of the network
    Δ_v = the degree of v
    N_v(k) (1 ≤ k ≤ Δ_v) = the k-th neighbor of v
    CSET = {1, 2, ..., 2Δ − 1} // set of all colors

local variables of node v
    outCol_v(x) (1 ≤ x ≤ Δ_v); // color proposed by v for the x-th incident link
        // We assume outCol_v(x) takes a value from CSET ∪ {⊥}
        // The value ⊥ is used temporarily only during execution of an atomic step
    Decided_v : subset of {1, 2, ..., Δ_v}; // the set of neighbor u such that the color of (u, v) is accepted (or finally decided)
    UnDecided_v : ordered subset of {1, 2, ..., Δ_v}; // the ordered set of neighbor u such that the color of (u, v) is not accepted
        // We assume Decide_v ∪ UnDecided_v = {1, 2, ..., Δ_v} holds in the initial configuration

variables in shared register r_{v,u}
    PC_{v,u}; // color proposed by v for the link (v, u)
    USET_{v,u}; // colors of links incident to v other than (v, u)
    // in-register r_{u,v} has PC_{u,v} and USET_{u,v}

function LINKCOLORING {
    // check the conflict on the accepted color
    // This is against that a Byzantine process changes the accepted color.
    // Also, this is against the initial illegitimate configuration (meaningful only in the first two round)
    for each k ∈ Decided_v{
        if (PC_{N_v(k),v} ≠ outCol_v(k)) or (outCol_v(k) = outCol_v(k') for some k'(≠ k))
            then { // something strange happens
                outCol_v(k) := ⊥;
                remove k from Decided_v;
                append k to UnDecided_v as the last element;
            // if this occurs in the third round or later, N_v(k) is a Byzantine process
            }
    }
    // check whether v's previous proposals were accepted by neighbors
    for each k ∈ UnDecided_v{
        if PC_{N_v(k),v} = outCol_v(k)
            then { // v's previous proposed was accepted by N_v(k)
                remove k from UnDecided_v;
                append k to Decided_v;
            }
            else // v's previous proposed was rejected by N_v(k)
                outCol_v(k) := ⊥;
    }
    // check whether v can accept the proposal made by neighbors
    for each k ∈ UnDecided_v in the order in UnDecided_v {
        // the order in UnDecided_v is important to avoid infinite obstruction of Byzantine processes
        if PC_{N_v(k),v} ∉ {outCol_v(m) | 1 ≤ m ≤ Δ_v}
            then { // accept the color proposed by N_v(k)
                outCol_v(k) := PC_{N_v(k),v};
                remove k from UnDecided_v;
                append k to Decided_v;
            }
            else // make proposal of a color for undecided links
                outCol_v(k) := min(CSET − (({outCol_v(m) | 1 ≤ m ≤ Δ_v} − {⊥}) ∪ USET_{N_v(k),v}))
            // at least one color is available (remark that outCol_v(k) = ⊥ holds)
    }
    for k := 1 to Δ_v { // write to its own link registers
        PC_{v,N_v(k)} := outCol_v(k);
        USET_{v,N_v(k)} := {outCol_v(m) | 1 ≤ m ≤ Δ_v, m ≠ k};
    }
}
```

Lemma 2 *Once a correct process v executes an action, $outCol_v(k) \neq outCol_v(k')$ holds for any k and k' $(1 \leq k < k' \leq \Delta_v)$ at any time (except that $outCol_v(k) = outCol_v(k') = \perp$ holds temporarily during execution of an action).*

Proof The lemma clearly holds from the following facts:

- When $outCol_v(k) = outCol_v(k')$ and $\{k, k'\} \subseteq Decided_v$ hold, then either $outCol_v(k)$ or $outCol_v(k')$ is reset to $\perp$. ($outCol_v(k) = outCol_v(k')$ and $\{k, k'\} \subseteq Decided_v$ may hold in the initial configuration.)
- v assigns a color c to $outCol_v(k)$ only when $outCol_v(k') \neq c$ holds for any k' $(k' \neq k)$.

□

Let u and v be any neighboring processes, and let v be the k-th neighbor of u. In the followings, we say that process u *accepts* a color c for a link (u, v) if $k \in Decided_u$ and $outCol_u(k) = c$ holds.

Lemma 3 *Let u and v be any correct neighboring processes, and let v be the k-th neighbor of u and u be the k'-th neighbor of v.*

Once v accepts a color of (u, v) in the second round or later, $outCol_u(k)$ and $outCol_v(k')$ never change afterwards. Moreover, u accepts the color of (u, v) in the next round or earlier.

Proof When process v completes its action at which v accepts a color c of (u, v),

$$\begin{array}{ll} & outCol_u(k) = PC_{u,v} = outCol_v(k') = PC_{v,u} = c \\ \wedge & outCol_u(k) \notin \{outCol_u(m) \mid 1 \leq m \leq \Delta_u, m \neq k\} \\ \wedge & outCol_v(k') \notin \{outCol_v(m) \mid 1 \leq m \leq \Delta_v, m \neq k'\} \end{array}$$

holds.

Process u or v never accepts a proposal c for any other incident link, and never makes a proposal c for any other incident link, as long as $outCol_u(k) = outCol_v(k') = c$ holds. This implies that $outCol_u(m) \neq c$ $(m \neq k) \wedge outCol_v(m) \neq c$ $(m \neq k')$ holds as long as $outCol_u(k) = outCol_v(k') = c$ holds.

Now we show that $outCol_u(k) = outCol_v(k') = c$ remains holding. We assume for contradiction that either $outCol_u(k)$ or $outCol_v(k')$ changes. Without loss of generality, we can assume that $outCol_u(k)$ changes first. This change of the color occurs only when $outCol_u(m) = c$ holds for some m such that $m \neq k$. This contradicts the fact that $outCol_u(m) \neq c$ $(m \neq k)$ remains holding as long as $outCol_u(k) = c$ holds.

It is clear that u accepts the color c for the link (u, v) when u is activated and $outCol_u(k) = PC_{v,u} = c$ holds. Thus, the lemma holds. □

Lemma 4 *Let u and v be any correct neighboring processes. Process u accepts a color for the link (u, v) within $2\Delta_u + 2$ rounds.*

Proof Let v be the k^{th} neighbor of u. Let t_1, t_2 and t_3 ($t_1 < t_2 < t_3$) be the steps (i.e., global discrete times) when u, v and u are activated respectively, and u is never activated between t_1 and t_3. We consider the following three cases of the configuration immediately before u executes an action at t_3. In what follows, let c be the color such that $outCol_u(k) = c$ holds immediately before u executes an action at t_3.

1. If $PC_{v,u} = c(= outCol_u(k))$ holds: Process u accepts the color c for (u, v) in the action at t_3.

2. If $PC_{v,u}(= c') \neq c(= outCol_u(k))$ holds and v is the first process among processes w such that $PC_{w,u} = c'$ in $UnDecided_u$: Process u accepts the color c' of $PC_{v,u}$ for (u, v) in the action at t_3.

3. If $PC_{v,u}(= c') \neq c(= outCol_u(k))$ holds and v is not the first process among processes w such that $PC_{w,u} = c$ in $UnDecided_u$: Process u cannot accept color c' for (u, v) in the action at t_3. Process u accepts the color c' for the link (u, w) such that w is the first process among processes x such that $PC_{x,u} = c'$ in $UnDecided_u$.

In the third case, Process w is removed from $UnDecided_u$. From Lemma 3, w is never appended to $UnDecided_u$ again when w is a correct process. When w is a Byzantine process, w may be appended to $UnDecided_u$ again but its position is after the position of u. This observation implies that the third case occurs at most $\Delta - 1$ times for the pair of u and v before u accepts a color for (u, v).

Now we analyze the number of rounds sufficient for u to accept a color of the link (u, v). Consider three consecutive rounds. Let t be the time when u is activated last in the first round of the three consecutive rounds, and let t' be the time when u is activated first in the last round of the three consecutive rounds. It is clear that v is activated between t and t'. This implies that we have at least one occurrence of the t_1, t_2 and t_3 described above between t and t'. We repeat this argument by regarding the last round of the three consecutive rounds as the first round of the three consecutive rounds we consider next. Thus, u accepts a color of (u, v) within $2\Delta_v + 2$ rounds. □

From Lemma 4, we can obtain the following theorem.

Theorem 1 *The protocol is a Byzantine insensitive link-coloring protocol for arbitrary networks. The stabilization time of the protocol is* $2\Delta + 2$ *rounds.*

Bibliography

[1] E. Anagnostou and V. Hadzilacos. Tolerating transient and permanent failures. *Lectures Notes in Computer Science, Vol 725 (Springer-Verlag)*, pages 174–188, 1993.

[2] J. Beauquier and S. Kekkonen-Moneta. Fault-tolerance and self-stabilization: impossibility results and solutions using self-stabiling failure detectors. *International Journal of Systems Science*, 28(11):1177–1187, 1997.

[3] J. Beauquier and S. Kekkonen-Moneta. On ftss-solvable distributed problems. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, page 290, 1997.

[4] E. W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.

[5] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[6] A. S. Gopal and K. J. Perry. Unifying self-stabilization and fault-tolerance. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, pages 195–206, 1993.

[7] T. Masuzawa. A fault-tolerant and self-stabilizing protocol for the topology problem. In *Proceedings of the 2nd Workshop on Self-Stabilizing Systems*, pages 1.1–1.15, 1995.

[8] H. Matsui, M. Inoue, T. Masuzawa, and H. Fujiwara. Fault-tolerant and self-stabilizing protocols using an unreliable failure detector. *IEICE Transactions on Information and Systems*, E83-D(10):1831–1840, 2000.

[9] M. Nesterenko and A. Arora. Tolerance to unbounded byzantine faults. In *Proceedings of 21st IEEE Symposium on Reliable Distributed Systems*, pages 22–29, 2002.

[10] Yusuke Sakurai, Fukuhito Ooshita, and Toshimitsu Masuzawa. A self-stabilizing link-coloring protocol resilient to byzantine faults in tree networks. In *Proceedings of the 2004 International Conference on Principles of Distributed Systems (OPODIS'2004)*, Lecture Notes in Computer Science. Springer-Verlag, December 2004.

[11] S. Ukena, Y. Katayama, T. Masuzawa, and H. Fujiwara. A self-stabilizing spanning tree protocol that tolerates non-quiescent permanent faults. *IEICE Transaction*, J85-D-I(11):1007–1014, 2002.