International Journal of Principles and Applications of Information Science and Technology
December 2007, Vol.1, No.1

Stabilizing Link-Coloration of Arbitrary Networks with Unbounded Byzantine Faults

Toshimitsu Masuzawa[1] and Sébastien Tixeuil[2]

[1] Osaka University, Japan
[2] Univ. Pierre & Marie Curie - Paris 6 & INRIA Futurs, France
masuzawa@ist.osaka-u.ac.jp, Sebastien.Tixeuil@lip6.fr

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 considers self-stabilizing link-coloring resilient to (permanent) Byzantine faults in arbitrary anonymous networks. First, we show that stabilizing link-coloring is impossible in anonymous cycles when there is no constraint on the spatial scheduling of processes. Then, given the assumption that no correct neighbors execute their actions simultaneously, we present a self-stabilizing link-coloring protocol that is also resilient to Byzantine faults in arbitrary anonymous networks. The protocol uses $2\Delta - 1$ colors where Δ is the maximum degree in the network. This protocol guarantees that any link (u, v) between non faulty processes u and v is assigned a color within $2\Delta + 2$ rounds and its color remains unchanged thereafter. Our protocol is Byzantine insensitive in the sense that the subsystem of correct processes remains operating properly in spite of unbounded Byzantine faults.

Key words: distributed protocol, self-stabilization, link-coloring, Byzantine fault, fault tolerance, fault containment

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.

The problem of vertex or link coloring consists in ensuring that two neighboring vertices (resp. links) are assigned different colors. Many variants of the problem have important applications related to resource allocation in distributed systems and networks (*e.g.* frequency or time slot allocation in wireless networks), and have been largely studied in the area of self-stabilization. Self-stabilizing protocols for distance one vertex coloring have been studied in [7, 10, 11, 14, 21–23], and for distance two vertex coloring in [9, 13]. Self-stabilizing protocols for link coloring have been studied in only a few papers [12, 15, 20, 24] and is further discussed thereafter.

There exist several researches on self-stabilizing protocols that are also resilient to permanent faults [1–3, 8, 16–20, 25]. Most of those consider only crash faults, and guarantee that every non faulty process achieves its intended behavior regardless of the initial network configuration. Only a few papers [17, 19, 20] provide self-stabilizing solutions that tolerate unbounded Byzantine faults. The main difficulty in this setting is caused by arbitrary and unbounded state changes of the Byzantine process: to adapt to any (initial) configuration, some process has to execute its action when it detects inconsistency between a neighbor and itself. Thus, 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 [19], the proposed protocols manage to contain the influence of Byzantine processes to only those in their vicinity, so that the remaining processes are eventually able to exhibit correct behavior. The complexity measure introduced in [19] is the *containment radius*, which is the maximum distance between a Byzantine process and a process 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 [20], the authors consider a self-stabilizing link-coloring protocol resilient to Byzantine faults in oriented tree networks, achieving a containment radius of two.

When the network is uniform (all nodes execute the same code) and anonymous (nodes have no possibility to distinguish from one another), a self-stabilizing link coloring algorithm cannot make the assumption that the color of a link is completely determined by a single node. Indeed, since nodes are uniform, it could be that two nodes have decided (differently) on the color of the link. As a result, the color of a link must come from some kind of coordination between at least two nodes. In this paper, we make the realistic assumption that a link color is decided only by its adjacent nodes. In this context, it follows that, 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 (and the aforementioned papers make that assumption), 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 [20], we consider arbitrary anonymous networks, where no pre-existing hierarchy is available. We show that link-coloring is impossible in anonymous rings when the spatial scheduling of the nodes is unconstrained (*i.e.* two neighboring nodes may be scheduled for execution at the same time). Thus, in the remaining of the paper, we assume that multiple processes may execute their actions simultaneously only if they are not neighbors. With this assumption, we present a link-coloring algorithm that is both self-stabilizing and tolerates an unbounded number of Byzantine nodes. Our protocol uses $2\Delta - 1$ colors, where Δ is the maximum degree in the network. It guarantees that any link (u, v) between non faulty 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 processes is limited to themselves. Thus, our protocol also trivially achieves Byzantine-fault containment with containment radius of one.

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 bidirectional 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 edge set is L, so we use some graph terminology to describe a distributed system S. A *subsystem* $S' = (P', L')$ of S 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 locally 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 neighbors u and v, there are two link registers $r_{u,v}$ and $r_{v,u}$ (Fig. 1). 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 $In_u = \{r_{v,u} | v \in N_u\}$.

A process may take *actions* during the execution of the system. In an action, the process executes the followings in an atomic manner: it reads from all of its input registers, changes its state and writes into all of its output registers. The literatures of self-stabilization commonly adopt a shared memory model [5], where a process is able to read the whole state of its neighbors. We adopt the shared registers instead, in order to narrow the communication capabilities to what is actually needed to solve the problem.

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

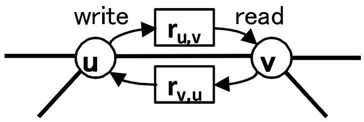

Fig. 1. Link registers $r_{u,v}$ and $r_{v,u}$ between processes u and v

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.

The configuration of a distributed system changes only by processes' actions, and behavior of the distributed system is represented by an (infinite) sequence of configurations called an *execution*. Even from the same configuration, different executions result from different orders of processes to execute their actions. The order of processes to execute their actions is called a *schedule*, say Q, which is represented by an infinite sequence of subsets of processes, e.g., $Q = U^1, U^2, \ldots$ where $U^i \subseteq P$ for each i ($i \geq 1$). The schedule implies that processes in U^1 execute their actions first at the same time, then processes in U^2 execute their actions at the same time and so on. 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 state and its output registers arbitrarily.

Let $BF = \{f_1, f_2, \ldots, f_c\}$ be the set of Byzantine processes. We call a process v ($\notin BF$) a *correct process*. In distributed systems with Byzantine processes, execution by a schedule $Q = U^1, U^2, \ldots$ cannot be uniquely determined because of actions of Byzantine processes. An infinite sequence of configurations $e = \rho_0, \rho_1, \ldots$ can be an execution by a schedule $Q = U^1, U^2, \ldots$ if the following conditions hold for each i ($i \geq 0$).

- For each correct process u in U^{i+1}, $\rho_{i+1}|u$ and $\rho_{i+1}|r$ ($r \in Out_u$) should result from an action of u at ρ_i.
- For each Byzantine process u in U^{i+1}, $\rho_{i+1}|u$ and $\rho_{i+1}|r$ ($r \in Out_u$) can be arbitrary.
- For any process v not in 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, $e' = \rho_0, \rho_1, \ldots, \rho_k$, satisfying $\bigcup_{1 \leq i \leq k} U^i = P$. 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

The *link coloring problem* considered in this paper is a so-called *static problem*, *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 [19] 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 *containment radius*. A *strictly stabilizing protocol* is defined as follows.

Definition 1. *A configuration ρ_0 is a BF*-stable configuration with containment radius ℓ *if and only if, for any execution $e = \rho_0, \rho_1, \ldots$ and any process v, the following condition holds (Fig. 2):*

> *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 BF-stable configuration, a process v more than ℓ away from any Byzantine process keeps satisfying $spec(v)$ and never changes the states of v and r ($r \in Out_v$) afterwards.

Definition 2 ([19]). *A protocol A is a* strictly stabilizing protocol with containment 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 containment 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 (i.e., P' is the set of all correct processes).*

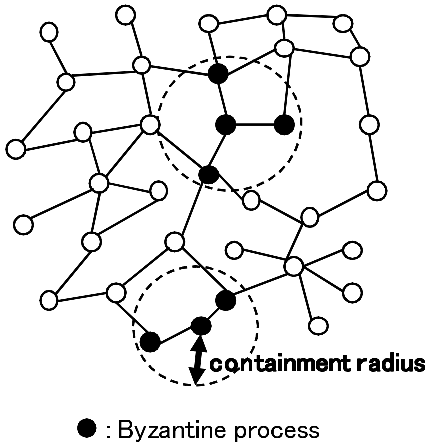

Fig. 2. BF-stable configuration: white processes satisfy *spec*(v)

Notice that if a protocol is Byzantine insensitive, it is also strictly stabilizing with containment 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 process. 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:*

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

We make the realistic assumption that a link color is decided only by the states of its adjacent nodes. In the following, we denote a link-coloring protocol with b colors as a *b-link-coloring protocol.*

3 Impossibility of Link-Coloring in Cycles

When we make no restriction on admissible schedules, the following impossibility result holds.

Theorem 1. *When two neighboring processes can execute their actions at the same time, self-stabilizing link-coloring of anonymous ring networks is impossible. This impossibility holds even when no Byzantine process exists and any number of colors are available.*

Proof. Consider a ring network $S = (P, L)$ of n processes ($n \geq 3$) where $P = \{v_1, v_2, \ldots, v_n\}$ and $L = \{(v_i, v_{i+1}) \mid 1 \leq i \leq n-1\} \cup \{(v_n, v_1)\}$. Assume that all processes have the same state in the initial configuration. Since the color of a link is determined (or encoded) by the states of its incident processes, all links have the same color in the initial configuration.

Assume all processes are correct. When all the processes execute their actions simultaneously, all the processes have the same state in the resulting configuration, and thus, all the links have the same color. Repeating the argument shows existence of an infinite execution where link-coloring cannot be attained.

4 Link-Coloring Protocol

4.1 Link-Coloring Protocol on Arbitrary Networks

From Theorem 1, some additional hypothesis on process scheduling is required to enable self-stabilizing link-coloration of arbitrary networks. In this section, we allow multiple processes to execute their actions at the same time provided that no correct neighbors execute their actions at the same time. Note that Byzantine processes may still be arbitrarily scheduled. Our protocol is presented as Figures 3 and 4. To simplify description of the protocol, we use program pseudo-codes, instead of state transition functions, to represent behavior of processes.

The protocol is informally described as follows: each process maintains a list of colors assigned to its incident links and periodically exchanges the list with each neighbor. From the list received from its neighbor u, a process v can propose a color for the link (u, v). This proposed color must not appear in the set of incident colors of u or v. Since the set of colors is of size $2\Delta - 1$, v can choose a color that is not used at u or v. Note that neighbors u and v cannot propose colors at the same time. 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. If the color proposed by the Byzantine process neighboring to v conflicts with a color on which two neighbors u and v have settled on, the proposition is ignored. The remaining case is when a node v has two neighbors u and w (where u and v are correct processes and w is Byzantine), and has not settled on any color with either u or w. The Byzantine process w may continuously proposed colors that conflict with u to v, and v could always chose the color proposed by w. To ensure that this behavior may not occur infinitely often, we use a priority list ($UnDecided_v$ in Fig. 3) so that neighbors of a particular node v get round robin priority when proposing conflicting colors (Fig. 5). Then, once u and v (the two correct processes) settle on a color for the link (u, v), the following proposals from w (the Byzantine process) are ignored by u.

```
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(k) (1 ≤ k ≤ Δ_v);
        // color proposed by v for the k-th incident link
        // We assume outCol_v(k) takes a value from CSET ∪ {⊥}
        // The value ⊥ is used temporarily only during execution of an atomic action
    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 Decided_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}
```

Fig. 3. SS link-coloring protocol (Part 1: constants and variables)

4.2 Correctness Proof

Let u and v be neighbors, and let v be the k-th neighbor of u. We say that register $r_{u,v}$ is consistent (with the state of u) 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 become consistent and remain so thereafter.*

Proof. By the code of the algorithm (see the last three lines).

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

The following lemma also holds clearly.

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 $outCol_v(k) = outCol_v(k') = \bot$ holds temporarily during execution of an action).*

Proof. The lemma clearly holds from the following facts:

```
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 rounds)
    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 proposal was accepted by N_v(k)
                remove k from UnDecided_v;
                append k to Decided_v;
            }
            else  // v's previous proposal 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 output registers
        PC_{v,N_v(k)} := outCol_v(k);
        USET_{v,N_v(k)} := {outCol_v(m) | 1 ≤ m ≤ Δ_v, m ≠ k};
    }
}
```

Fig. 4. SS link-coloring protocol (Part 2: LINKCOLORING function)

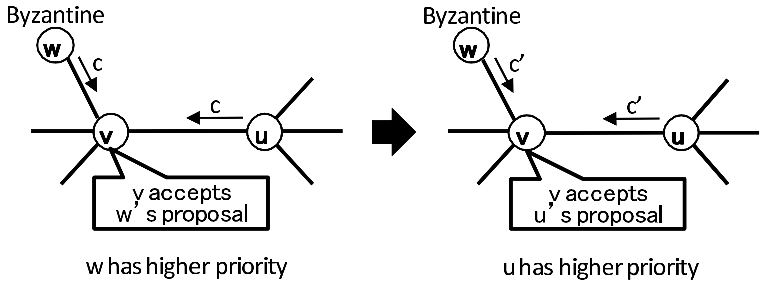

Fig. 5. Round robin priority for avoiding infinite disturbance of a Byzantine process

- 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 neighbors, 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 neighbors, 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),

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

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 $outCol_u(m) \neq c$ (for each $m \neq k$) and $outCol_v(m) \neq c$ (for each $m \neq k'$) hold as long as $outCol_u(k) = outCol_v(k') = c$ holds.

Now we show that $outCol_u(k) = outCol_v(k') = c$ remains holding once $outCol_u(k) = outCol_v(k') = c$ holds. 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 neighbors. 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 $PC_{v,u} = c$ holds: Process u accepts the color c for (u, v) in the action at t_3.
2. If $PC_{v,u}(= c') \neq c$ 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 $PC_{v,u}(= c') \neq c$ 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 2. *The protocol is a Byzantine insensitive link-coloring protocol for arbitrary networks. The stabilization time of the protocol is $2\Delta + 2$ rounds.*

5 Conclusion

In this paper, we presented the first self-stabilizing link-coloring algorithm that can be used on uniform anonymous and general topology networks. In addition to being self-stabilizing, it is also Byzantine insensitive, in the sense that the subsystem of correct processes resumes correct behavior in finite time regardless of the number and placement of potentially malicious (so called Byzantine) processes.

The system hypothesis that we assumed (*i.e.*, restriction on schedules) is necessary to ensure self-stabilizing bounded fault-containment of Byzantine processes. However, we assumed that the number of link colors that is available is $2\Delta - 1$, where Δ is the maximum degree of the graph. It is well known that $\Delta + 1$ colors are sufficient for link coloring general graphs. Recently, a distributed (non-stabilizing and non fault tolerant) solution [6] that uses only $\Delta + 1$ colors was provided. There remains the open question of a possible tradeoff between the number of colors used for link coloring and the fault-tolerance properties of distributed solutions.

Acknowledgements

This work is supported in part by Global COE (Centers of Excellence) Program of MEXT, Grant-in-Aid for Scientific Research ((B)19300017) of JSPS, Grant-in-Aid for Scientific Research on Priority Areas (16092215) of MEXT, and SCOPE (Strategic Information and Communications R&D Promotion Programme) of MIC. This work is also supported in part by the FRAGILE project of the ACI "Sécurité et Informatique", the SOGEA project of the ANR "SSIA", and the FRACAS project of INRIA.

References

1. E. Anagnostou and V. Hadzilacos. Tolerating transient and permanent failures. In *Proc. 7th International Workshop on Distributed Algorithms*, 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. S. Gandham, M. Dawande, and R. Prakash. Link scheduling in sensor networks: Distributed edge coloring revisited. In *Proc. Infocom 2005*, 2005.
7. S. Ghosh and M. H. Karaata. A self-stabilizing algorithm for coloring planar graphs. *Distributed Computing*, 7(1):55–59, 1993.
8. A. S. Gopal and K. J. Perry. Unifying self-stabilization and fault-tolerance. In *Proc. 12th Annual ACM Symposium on Principles of Distributed Computing*, pages 195–206, 1993.
9. M. Gradinariu and C. Johnen. Self-stabilizing neighborhood unique naming under unfair scheduler. In *Proc. 7th International Euro-Par Conference*, pages 458–465, 2001.

10. M. Gradinariu and S. Tixeuil. Self-stabilizing vertex coloration and arbitrary graphs. In *Proc. 4th International Conference on Principles of Distributed Systems*, pages 55–70, 2000.
11. S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. Linear time self-stabilizing colorings. *Information Processing Letters*, 87(5):251–255, 2003.
12. T. Herman, I. Pirwani, and S. Pemmaraju. Oriented edge coloring and link scheduling in sensor networks. In *Proc. International Conference on Communication Software and Middleware*, pages 1–6, 2006.
13. T. Herman and S. Tixeuil. A distributed TDMA slot assignment algorithm for wireless sensor networks. In *Proc. 1st International Workshop on Algorithmic Aspects of Wireless Sensor Networks*, pages 45–58, 2004.
14. S.-T. Huang, S.-S. Hung, and C.-H. Tzeng. Self-stabilizing coloration in anonymous planar networks. *Information processing letters*, 95(1):307–312, 2005.
15. S.-T. Huang and C.-H. Tzeng. Distributed edge coloration for bipartite networks. In *Proc. 8th International Symposium on Stabilization, Safety and Security of Distributed Systems*, pages 363–377, 2006.
16. T. Masuzawa. A fault-tolerant and self-stabilizing protocol for the topology problem. In *Proc. 2nd Workshop on Self-Stabilizing Systems*, pages 1.1–1.15, 1995.
17. T. Masuzawa and T. Sébastien. Bounding the impact of unbounded attacks in stabilization. In *Proc. 8th International Symposium on Stabilization, Safety and Security of Distributed Systems*, pages 440–453, 2006.
18. 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.
19. M. Nesterenko and A. Arora. Tolerance to unbounded byzantine faults. In *Proc. 21st IEEE Symposium on Reliable Distributed Systems*, pages 22–29, 2002.
20. Y. Sakurai, F. Ooshita, and T. Masuzawa. A self-stabilizing link-coloring protocol in tree networks with permanent byzantine faults. *Journal of Aerospace Computing, Information, and Communication*, 3(8):420–436, 2006.
21. S. Shukla, D. Rosenkrantz, and S. Ravi. Developing self-stabilizing coloring algorithms via systematic randomization. In *Proc. International Workshop on Parallel Processing*, pages 668–673, 1994.
22. S. Shukla, D. Rosenkrantz, and S. Ravi. Observations on self-stabilizing graph algorithms for anonymous networks. In *Proc. the 2nd Workshop on Self-stabilizing Systems*, pages 7.1–7.15, 1995.
23. S. Sur and P. K. Srimani. A self-stabilizing algorithm for coloring bipartite graphs. *Information Sciences: an International Journal*, 69(3):219–227, 1993.
24. C.-H. Tzeng, J.-R. Jiang, and S.-T. Huang. A self-stabilizing $(\delta + 4)$-edge-coloring algorithm for planar graphs in anonymous uniform systems. *Information processing letters*, 101(4):168–173, 2007.
25. 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.