情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2011-AL-134 No.21
2011/3/7

時間限定ビザンチン故障に対する 故障封じ込め自己安定リーダー選挙プロトコル

山内　由紀子[†1]　増　澤　利　光[†2]　Doina Bein[†3]

時間限定ビザンチン故障は有限回の任意の（悪意のある）振る舞いを行う．本論文では，時間限定ビザンチン故障に対して故障封じ込めと自己安定の 2 つの性質を保証するリーダー選挙プロトコルを提案する．はじめに，時間限定ビザンチン故障が挿入する偽の情報が拡散する範囲を抑制する pumping という通信手法を提案する．提案するリーダー選挙プロトコルは pumping を用いてリーダー選挙を行うことにより，故障の影響を受けるプロセスの数を時間限定ビザンチン故障の任意の振る舞いの回数にのみ依存した数に抑える．

Self-Stabilizing and Fault-Containing Leader Election Resilient to Time-Bounded Byzantine Faults

Yukiko Yamauchi,[†1] Toshimitsu Masuzawa[†2] and Doina Bein[†3]

In this paper, we propose a novel adaptive fault-containment method against time-bounded Byzantine faults. A *time-bounded Byzantine fault* behaves correctly after it consumes a finite number of malicious actions. We propose a self-stabilizing and fault-containing leader election protocol that is resilient to time-bounded Byzantine faults. The proposed protocol is based on a novel information diffusion method, called *pumping*, that prevents fictitious information injected by time-bounded Byzantine fault from spreading. By using pumping, the proposed leader election protocol promises adaptive fault containment property that guarantees that the number of perturbed processes depends on the number of malicious actions at time-bounded Byzantine processes.

†1 奈良先端科学技術大学院大学 Nara Institute of Science and Technology y-yamauchi@is.naist.jp
†2 大阪大学 Osaka University masuzawa@ist.osaka-u.ac.jp
†3 The Pennsylvania State University, USA siona@psu.edu

1. Introduction

A distributed system consists of a collection of processes that communicate with each other so that the entire system satisfies a given specification. Fault tolerance is one of the main challenges in the design of distributed systems because a distributed system is more prone to faults, such as memory crashes at processes and malicious users, as the number of processes in the entire system increases.

A *transient fault* changes the memory contents at processes arbitrarily. A self-stabilizing protocol[3] promises that starting from any arbitrary initial configuration, the system eventually satisfies its specification (*i.e.,* convergence) and after that, it never violates its specification. Though self-stabilization was originally designed for autonomous adaptability against transient faults, many papers try to extend the target fault model to the *Byzantine fault* that allows arbitrary (malicious) actions at faulty processes[4],[5],[7]. The Byzantine fault model is generally classified as permanent faults. In the context of self-stabilization, researchers have tried to contain the effect of permanent Byzantine faults to a constant distance from Byzantine faults, and make other processes outside the distance achieve self-stabilization. However, Nesterenko et al. showed that for global problems such as the leader election problem and the spanning tree construction problem, it is impossible to contain the effect of permanent Byzantine faults to a constant distance[7].

We propose a weaker type of Byzantine fault model, called *time-bounded Byzantine fault* (TB-Byzantine fault, for short) that consumes only a finite number of malicious actions. It is obvious that a self-stabilizing protocol eventually satisfies its specification after the finite number of malicious actions. We focus on the fault-containment against TB-Byzantine faults and propose an adaptive containment method that guarantees that the scale of perturbation caused by Byzantine faulty processes depends on the number of malicious actions at Byzantine processes. We propose a self-stabilizing and TB-Byzantine fault resilient fault-containing protocol for the leader election problem which is one of the most important problems in design of distributed systems.

Related work. The difficulty in combining self-stabilization and Byzantine faults is how to obtain and keep consensus among correct processes in the presence of Byzantine

Vol.2011-AL-134 No.21
2011/3/7

faults because Byzantine faulty processes take malicious actions during and after convergence. For local problems, Nesterenko et al. proposed the notion of *strict stabilization* that contains the effect of a Byzantine faulty process to a constant distance, called *containment radius*, and the remaining processes realize self-stabilization[7]. They proposed a strict stabilizing vertex coloring protocol and a strict stabilizing dining philosophers protocol. For global problems, Masuzawa et al. proposed the notion of *strong stabilization* by relaxing the requirements of strict stabilization: permanent influence of Byzantine processes is contained within a contamination radius and temporary influence is allowed to spread over the entire network[5]. They proposed a strong stabilizing tree orientation protocol[5]. However, these papers focus on permanent Byzantine faults.

It has been pointed out that traditional permanent Byzantine fault model is too extreme and several weaker Byzantine fault models are proposed. One is Byzantine with crash (for the dining philosopher problem[6] and for the consensus problem[8]), and the other is Byzantine with recovery (for the consensus problem[2] and the clock synchronization problem[1]). However, they do not focus on self-stabilization and fault-containment against malicious actions.

Our contribution. In this paper, we first propose a novel bounded Byzantine fault model, called *time-bounded Byzantine fault*. Then, we propose a global information diffusion method *pumping* that contains the effect of TB-Byzantine processes in the sense that the number of processes that receives fictitious information depends on the number of malicious actions at TB-Byzantine processes. Based on the pumping method, we propose a self-stabilizing and fault-containing leader election protocol against TB-Byzantine faults.

2. Preliminary

2.1 System model

A system is represented by a graph $G = (V, E)$ where the vertex set $V = \{P_0, P_1, \cdots, P_{n-1}\}$ is the set of processes and the edge set $E \subseteq V \times V$ is the set of bidirectional communication links. Two processes P_i and P_j are *neighboring* if $(P_i, P_j) \in E$ $(0 \leq i, j \leq n-1)$. The distance between two processes is the length of the shortest path between the two.

Each process P_i has a unique ID denoted by ID_i and maintains a set of local variables. A subset of the local variables at each process is called the *output variables*. The state of a process is an assignment of values to all its local variables. We adopt the *state reading model* for communication among processes. Process P_i can read the values of the local variables at its immediate neighbors, while P_i can change only the values of its own local variables.

Each process P_i changes its state according to a protocol that consists of a finite set of guarded actions of the form $\langle label \rangle : \langle guard \rangle \rightarrow \langle action \rangle$. A *guard* is a Boolean expression involving the local variables of P_i and of its neighboring processes. An *action* is a statement that changes the values of the local variables at P_i. A guard is *enabled* if it is evaluated to *true*. A process with an enabled guard is called *enabled*.

A *configuration* of a system is an assignment of values to all local variables of all processes. A *schedule* of a distributed system is an infinite sequence of sets of processes. Let $S = R^1, R^2, \cdots$ be a schedule where $R^i \subseteq V$ holds for each i $(i \geq 1)$. For a process set R and two configurations σ and σ', we denote $\sigma \xrightarrow{R} \sigma'$ when σ changes to σ' by executing an action of each process in R simultaneously. An infinite sequence of configurations $E = \sigma_0, \sigma_1, \cdots$ is called an *execution* from an initial configuration σ_0 by schedule S if $\sigma_i \xrightarrow{R^{i+1}} \sigma_{i+1}$ holds for each $i \geq 0$. We say a process in R^{i+1} is *activated* in configuration σ_i.

We adopt the *distributed daemon* as a scheduler that allows any subset of processes to execute actions simultaneously. If a selected process has no enabled guard then it does not change its state. If a selected process has multiple enabled guards, then the process executes the action corresponding to only one of the enabled guards.

A distributed daemon allows *asynchronous* executions. In an asynchronous execution, the time is measured by *rounds*. Let $E = \sigma_0, \sigma_1, \cdots$ be an asynchronous execution by schedule $S = R^1, R^2, \cdots$. The first round $\sigma_0, \sigma_1, \cdots, \sigma_j$ is the minimum prefix of E such that $\bigcup_{i=1}^{j} R^i = V$. The second round and the latter rounds are defined recursively by applying the definition of the first round to the remaining suffix $E' = \sigma_j, \sigma_{j+1}, \cdots$ and $S' = R^{j+1}, R^{j+2}, \cdots$.

A *Byzantine faulty process* behaves arbitrarily and independently from the protocol. A state change at a process is a *malicious action* if the state change does not conform to the protocol, otherwise a *normal action*. When a Byzantine process P_i is activated, P_i consumes one normal action or one malicious action. If a Byzantine process does not change its state by ignoring the behavior of the protocol when it is activated, P_i consumes one malicious action.

A *time-bounded Byzantine fault* is a subclass of the Byzantine fault model such that the number of malicious actions at each Byzantine faulty process is finite. An execution $E = \sigma_0, \sigma_1, \cdots$ is (f,k)-*faulty* if the number of Byzantine processes in E is f and the number of malicious actions at each Byzantine process is at most k. For an (f,k)-faulty execution, a Byzantine process is called k-TB-Byzantine process. An (f,k)-faulty execution contains at most $f \cdot k$ malicious actions. An execution is *correct* if it contains no malicious action.

2.2 Self-stabilization and fault-containment

Self-stabilization promises autonomous adaptability against any finite number of any type of transient faults by considering the configuration obtained by the last fault as the initial configuration. Hence, the stabilization in the presence of a bounded number of malicious actions (*i.e.*, TB-Byzantine processes) is clear. However, self-stabilization guarantees nothing in the presence of malicious actions during and after convergence, and even for a single malicious action in a legitimate configuration, the effect may spread over the entire network.

A *problem* (*task*) $\mathcal{T}$ is defined by a *validity predicate* on output variables at all processes. Intuitively, a configuration of a protocol is valid if it satisfies the validity predicate of $\mathcal{T}$. However, the protocol may have local variables other than the output variables, and the values of these variables may introduce updates of output variables. Hence, we define a *legitimate configuration* σ of a protocol $\mathcal{P}_\mathcal{T}$ as the one such that any configuration (including σ itself) appearing in any correct execution starting from σ satisfies the validity predicate of $\mathcal{T}$. The set of legitimate configurations are denoted by $\mathcal{C}_L(\mathcal{P}_\mathcal{T})$. (We omit $\mathcal{P}$ and $\mathcal{T}$ when they are clear.)

Definition1 (**Self-stabilization**) Protocol $\mathcal{P}_\mathcal{T}$ is self-stabilizing if the system eventually reaches a legitimate configuration of $\mathcal{P}_\mathcal{T}$ in any correct execution starting from any configuration.

The *convergence time* is the maximum (worst) number of rounds that is necessary for the system to reach a legitimate configuration in any correct execution starting from any configuration.

We consider self-stabilization and fault-containment in the presence of malicious actions. As stated above, any self-stabilizing protocol eventually reaches a legitimate configuration in any (f,k)-faulty execution if f and k are finite. However, malicious actions during convergence may delay the convergence to a legitimate configuration. Also, malicious actions after convergence (*i.e.*, malicious actions in a legitimate configuration) may perturb the system. To measure the disturbance and perturbation by malicious actions, we introduce the disruption and the perturbation as follows. For an (f,k)-faulty execution $E = \sigma_0, \sigma_1, \cdots$, the *disruption* is the minimal prefix of E, denoted by $E' = \sigma_0, \sigma_1, \cdots \sigma_j$ such that σ_j is a legitimate configuration. The disruption of E represents the convergence in spite of or after malicious actions. If E starts from a legitimate configuration σ_0 immediately followed by at least one malicious action, we define perturbation as follows. The *perturbation* of E is the minimal prefix of E, denoted by $E' = \sigma_0, \sigma_1, \cdots \sigma_j$ $(j \geq 1)$ such that σ_j is a legitimate configuration. A disruption (perturbation) is called (f',k')-disruption ((f',k')-perturbation, respectively) if it contains malicious actions of f' processes and at most k' malicious actions for each of the TB-Byzantine processes. Hence, in an (f',k')-disruption and an (f',k')-perturbation, there are at most $f' \cdot k'$ malicious actions.

An (f,k)-faulty execution can contain an (f',k')-disruption for $f' \leq f$ and $k' \leq k'$. The (f',k')-*disruption time* is the maximum (worst) number of rounds of any (f',k')-disruption. We note that when $f' \cdot k' = 0$, the execution is correct and the disruption time is equal to the convergence time.

For an (f',k')-perturbation E', we say that a correct process is *perturbed* in E' if the process changes its output in E'. The (f',k')-*perturbation number* is the maximum (worst) number of perturbed processes in any (f',k')-perturbation. The (f',k')-*perturbation time* is the maximum (worst) number of rounds of any (f',k')-perturbation. Note that we define the perturbation number only with the output variables.

Definition2 (**TB-Byzantine resilient fault-containment**) A self-stabilizing protocol $\mathcal{P}$ is TB-Byzantine resilient fault-containing if (f',k')-perturbation number depends on $\min\{f' \cdot k', n\}$ and/or $(f' \cdot k')$-perturbation time depends on $\min\{f' \cdot k', n\}$.

3. Proposed method

The *leader election problem* is to make all processes in the system recognize a single process, called *leader*. The proposed leader election protocol is based on information diffusion technique that is resilient to TB-Byzantine faults. Each process diffuses its ID and stores received IDs, then chooses the minimum ID among the locally stored IDs.

For simplicity, we consider an oriented ring $G = (V, E)$ of n processes. Each process P_i in V has two neighbors $P_{i-1 \bmod n}$ and $P_{i+1 \bmod n}$ $(i \in [0..n-1])$[⋆1]. P_{i-1} (P_{i+1}) is called the *predecessor* (the *successor*, respectively) of P_i. The output variable at each process is the leader's ID variable. Additionally, each process maintains a local table to store the received IDs. To select the minimum ID among existing processes, it is necessary that each process stores IDs of all existing processes and does not store any fictitious IDs (*i.e.*, IDs of non-existent processes).

An arbitrary initial configuration allows processes store initial fictitious IDs. To achieve self-stabilization, each process should remove fictitious IDs and diffuse its ID to all other processes. Each process P_i diffuses its ID and distance value (0 at P_i) and each time the ID is forwarded by some process, the distance value is incremented. If a fictitious ID ℓ' is stored at all processes, there exists at least one process P_j that has the locally minimum distance value but $ID_j \neq \ell'$. Otherwise, it is detected at least at one process by each process comparing the list of locally stored IDs with the predecessor. Then, by making this process invoke a removal action, this fictitious ID is removed from the entire network.

A malicious action at each TB-Byzantine process can perturb the entire network in two ways. First, a TB-Byzantine process can stop forwarding the smallest ID that should be chosen as the leader's ID. This may delay the convergence, however, because the number of malicious actions at each TB-Byzantine process is finite, eventually all IDs are forwarded and each process chooses the leader's ID from a correct set of IDs.

Secondly, a TB-Byzantine process can diffuse fictitious IDs. If one of the fictitious IDs is smaller than any IDs of existing processes, all processes choose the ID. Our basic strategy to tolerate TB-Byzantine faults is to exhaust TB-Byzantines by making them consume malicious actions to perturb other processes. We introduce a mechanism called *pumping* that makes each process keep on changing its state to push an ID further and further. More specifically, each process should change its state h times in order to diffuse an ID to all process at distance at most h.

To elect a correct leader, each process should generate at least n waves to diffuse its ID to all other processes. We also use this pumping method to remove fictitious IDs because removing locally stored IDs can introduce global perturbation. Consider a protocol that allows each process to discard a locally stored ID if its predecessor does not store the same ID. Let an ID ℓ be locally stored at all processes. If one process P_i removes ID ℓ, then its successor P_{i+1} removes ℓ. After that, P_{i+2} removes ℓ. This behavior becomes global and starts a removal wave that spreads fast over the entire system.

Finally, after all ID diffusions are finished, the locally stored minimum ID is selected at each process.

In the following, we fist show the pumping protocol $\mathcal{PUMP}$ in Section 3.1 and the proposed leader election protocol $\mathcal{PLE}$ in Section 3.2.

3.1 Pumping protocol $\mathcal{PUMP}$

Process P_i is called *source* for the diffusion of ID_i and process P_{i-1} is called *tail*. Processes $P_{i+1}, P_{i+2}, \cdots, P_{i-2}$ are called *forwarders*. The pumping mechanism for each P_i is implemented with a sequence of waves that consists of a tuple of ID_i and a TTL value. The source process repeatedly generates waves with incrementing the TTL by one. (The initial wave has TTL of one.) Hence, P_i has to generate a sequence of h waves with incrementing the TTL values to diffuse ID_i to process P_{i+h}. Forwarders forward a wave with decrementing the TTL value until the TTL value reaches zero. The diffusion is finished when a wave reaches the tail P_{i-1} and the acknowledgment wave is forwarded from P_{i-1} to P_i through all the forwarders.

The pumping protocol called $\mathcal{PUMP}$ is shown as Protocol 3.1. For the diffusion of ID ℓ, the source, the forwarders, and the tail is defined by the three predicates *IsSource*(ℓ), *IsForwarder*(ℓ), and *IsTail*(ℓ). We assume that during an execution of $\mathcal{PUMP}$, *IsSource*(ℓ) (*IsTail*(ℓ)) continuously holds at the source (tail, respectively)

⋆1 For the rest of the paper, we omit mod$n$ and P_i means $P_{i \bmod n}$.

Protocol 3.1 $\mathcal{PUMP}(T_i, \mathit{IsSource}_i(\ell), \mathit{IsForwarder}_i(\ell), \mathit{IsTail}_i(\ell))$ at P_i for ID ℓ

```
Parameters at $P_i$
    Local variable
        $C_i(\ell)$: the counter value of entry $(\ell, C_i(\ell))$ in $T_i$
    Output at $P_i$
        $C_i(\ell)$ in $T_i$.
    Predicates at $P_i$
        $\mathit{IsSource}_i(\ell)$: Boolean predicate that takes true if $P_i$ is the source for ID $\ell$, otherwise false.
        $\mathit{IsForwarder}_i(\ell)$: Boolean predicate that takes true if $P_i$ is a forwarder for ID $\ell$,
                otherwise false.
        $\mathit{IsTail}_i(\ell)$: Boolean predicate that takes true if $P_i$ is the tail for $\ell$, otherwise false.
        $\mathit{CntIncons}_i(\ell) = (C_i(\ell) \geq 0) \wedge (C_{i+1}(\ell) \geq 0) \wedge \neg(1 \leq C_i(\ell) - C_{i+1}(\ell) \leq 2)$
        $\mathit{AckIncons}_i(\ell) = (C_{i-1}(\ell) = \phi) \wedge \{(C_i(\ell) \neq \phi) \vee (C_i(\ell) = \mathit{undef})\}$

Actions at process $P_i$
        $S_1$  $\mathit{IsSource}_i(\ell) \wedge [\mathit{CntIncons}_i(\ell) \vee (C_i(\ell) = \perp) \vee (C_i(\ell) = \mathit{undef}) \vee$
             $\vee (C_{i+1}(\ell) = \perp) \vee (C_{i+1}(\ell) = \mathit{undef})] \vee \{(C_i(\ell) = \phi) \wedge (C_{i+1}(\ell) \neq \phi)\}]$
                 $\longrightarrow \mathit{create}((\ell, 1), T_i)$  //reset pumping
        $S_2$  $\mathit{IsSource}_i(\ell) \wedge (C_i(\ell) = 0) \longrightarrow \mathit{create}((\ell, 1), T_i)$  //restart waves
        $S_3$  $\mathit{IsSource}_i(\ell) \wedge (C_i(\ell) \geq 0) \wedge (C_{i+1}(\ell) \geq 0) \wedge (C_i(\ell) - C_{i+1}(\ell) = 1)$
                 $\longrightarrow \mathit{increment}(C_i(\ell), T_i)$  //generate a new wave by incrementing TTL
        $S_4$  $\mathit{IsSource}_i(\ell) \wedge (C_i(\ell) > 0) \wedge (C_{i+1}(\ell) = \phi)$
                 $\longrightarrow \mathit{ack}(C_i(\ell), T_i)$  // receipt of acknowledgment

        $S_5$  $\mathit{IsForwarder}_i(\ell) \wedge (C_{i-1}(\ell) = 1) \wedge (C_i(\ell) \neq 0)$
                 $\longrightarrow \mathit{create}((\ell, 0), T_i)$  // become a head of a wave
        $S_6$  $\mathit{IsForwarder}_i(\ell) \wedge [\{(C_{i-1}(\ell) \geq 0) \wedge (C_i(\ell) \geq 0) \wedge (C_{i+1}(\ell) \geq 0) \wedge$
             $(C_{i-1}(\ell) - C_i(\ell) = 2) \wedge (C_i(\ell) - C_{i+1}(\ell) = 1)\} \vee \{(C_{i-1}(\ell) = 2) \wedge (C_i(\ell) = 0)\}]$
                 $\longrightarrow \mathit{increment}(C_i(\ell), T_i)$  //forward a wave
        $S_7$  $\mathit{IsForwarder}_i(\ell) \wedge [\mathit{CntIncons}_i(\ell) \vee \mathit{AckIncons}_i(\ell) \vee$
             $\{(C_i(\ell) = 0) \wedge (C_{i+1}(\ell) = \phi)\} \vee \{(C_i(\ell) > 1) \wedge (C_{i+1}(\ell) = \mathit{undef})\}]$
                 $\longrightarrow \mathit{reset}(C_i(\ell), T_i)$  // generate reset signal
        $S_8$  $\mathit{IsForwarder}_i(\ell) \wedge (C_i(\ell) \neq 0) \wedge (C_i(\ell) \neq 1) \wedge (C_{i+1}(\ell) = \perp)$
                 $\longrightarrow \mathit{create}((\ell, \perp), T_i)$  // forward reset signal
        $S_9$  $\mathit{IsForwarder}_i(\ell) \wedge (C_{i-1}(\ell) > 0) \wedge (C_i(\ell) > 0) \wedge (C_{i+1}(\ell) = \phi)$
             $(1 \leq C_{i-1}(\ell) - C_i(\ell) \leq 2)$
                 $\longrightarrow \mathit{ack}(C_i(\ell), T_i)$  //forward acknowledgment

        $S_{10}$ $\mathit{IsTail}_i(\ell) \wedge C_{i-1}(\ell) = 1 \wedge C_i(\ell) \neq \phi \longrightarrow \mathit{create}((\ell, \phi), T_i)$  // become a head
        $S_{11}$ $\mathit{IsTail}_i(\ell) \wedge [\mathit{CntIncons}_i(\ell) \vee \mathit{AckIncons}_i(\ell) \vee \{(C_{i-1}(\ell) = \perp) \wedge (C_i(\ell) \neq \perp)\}$
                 $\longrightarrow \mathit{ack}(C_i(\ell), T_i)$ //generate reset signal
```

process.

$\mathcal{PUMP}$ has four parameters: T_i is a local table at P_i to store the received wave, and there are three predicates $\mathit{IsSource}_i(\ell)$, $\mathit{IsForwarder}_i(\ell)$, and $\mathit{IsTail}_i(\ell)$ to define the source, the forwarders, and the tail for ID ℓ. We later show how we define these predicates when we use $\mathcal{PUMP}$ in $\mathcal{PLE}$.

Each entry of T_i is in the form of (ℓ, c) where ℓ is an ID and c is the counter (*i.e.*, TTL) of the latest wave[⋆1]. For any entry $(\ell, c) \in T_i$, the second element is denoted by $C_i(\ell)$, *i.e.*, for $(\ell, c) \in T_i$, $C_i(\ell)$ returns c. The counter value $C_i(\ell)$ returns either a natural number, $\perp$, ϕ or *undef*. The value $\perp$ is a *reset signal*, and ϕ is an *acknowledgment signal*. If T_i does not have an entry with ID ℓ, $C_i(\ell)$ returns *undef*. The output variable of $\mathcal{PUMP}$ for ID ℓ at process P_i is $C_i(\ell)$.

There are five operations for each entry in T_i: *create*, *increment*, *reset*, *ack* and *comparison*. Operation *create*$((\ell, c), T_i)$ creates an entry (ℓ, c) if T_i does not have an entry with ID ℓ, otherwise it sets the value of $C_i(\ell)$ to c. Operation *increment*$(C_i(\ell), T_i)$ increments the value of $C_i(\ell)$ if $C_i(\ell)$ is in T_i. Operation *reset*$(C_i(\ell), T_i)$ assigns $\perp$ to $C_i(\ell)$ if $C_i(\ell)$ is in T_i. Operation *ack*$(C_i(\ell), T_i)$ assigns ϕ to $C_i(\ell)$ if $C_i(\ell)$ is in T_i. The comparison operation on the value of $C_i(\ell)$ is also possible as long as $C_i(\ell)$ takes an integer. When $C_i(\ell)$ takes an integer and compared with an integer, it returns the result, otherwise, *false*. We say the counter value $C_i(\ell)$ at P_i is *consistent* if and only if $C_i(\ell) - C_{i+1}(\ell) = 1$ or 2 holds, otherwise *inconsistent*. A sequence of consistent processes starting from a source process is carrying waves and the head of the wave is the farthest process whose TTL value is 0.

A source process P_s starts pumping with a wave $(\ell, 1)$ (S_1 or S_2). A wave (ℓ, j) generated at P_s is forwarded through $P_{s+1}, P_{s+2}, \cdots, P_{s+j}$ $(s + j \leq t)$ by each forwarder decrementing the TTL of the wave (S_5 and S_6). P_s continues to generate waves with incrementing TTL values (S_3). When a wave reaches the tail process P_t, P_t generates an acknowledgement signal (S_{11}), and the acknowledgment wave is returned to P_s through forwarders (S_9), and received by P_s (S_4). When a forwarder or the tail finds inconsistency among the counter values at its predecessor or successor, it generates a reset signal (S_7 or S_{11}), and the reset signal is returned to P_s through forwarders (S_8). When P_s receives a reset wave, it restarts pumping with a wave $(\ell, 1)$.

$\mathcal{PUMP}$ guarantees *fictitious diffusion containment* property, *i.e.*, in any (f, k)-perturbation, any fictitious ID injected by a TB-Byzantine process P_j is forwarded

⋆1 For simplicity, we assume each entry in T_i has a unique ID value. Hence, for any ℓ if (ℓ, c) in T_i, no entry (ℓ, c') with $c \neq c'$ exists in T_i. This is a data structure consistency problem and a solution for it can be easily implemented and applied to T_i. We do not address this problem in the paper.

to at most k processes, $P_j, P_{j+1}, \cdots, P_{j+k}$. The pumping mechanism makes each forwarder forward a wave with TTL of c only after it forwards a wave with TTL of $(c-1)$. Hence, a source process has to start with a wave with TTL 0 when it starts to diffuse its ID. This property forces a TB-Byzantine process to consume malicious actions during diffusion of fictitious IDs and in any k-TB-Byzantine process can diffuse fictitious IDs to at most k correct processes.

A configuration is legitimate for $\mathcal{PUMP}$ for ID ℓ, if the following predicate $LP_{\mathcal{PUMP}}(\ell)$ holds at any process P_i in V:

$$LP_{\mathcal{PUMP}}(\ell) \equiv$$
$$\forall i : s \le i \le t \text{ s.t. } \mathit{IsSource}_s(\ell) = \mathit{true} \text{ and } \mathit{IsTail}_t(\ell) = \mathit{true} : C_i(\ell) = \phi$$

3.2 Leader election protocol $\mathcal{PLE}$

The proposed leader election protocol $\mathcal{PLE}$ uses $\mathcal{PUMP}$ to diffuse IDs and to remove fictitious IDs. The leader election protocol $\mathcal{PLE}$ is shown in Protocol 3.2.

Each process P_i maintains the output variable LID_i and two tables of received waves: the diffusion table DfT_i and the removal table RmT_i. The diffusion table DfT_i is used to diffuse IDs of processes and the removal table RmT_i is used to remove IDs. $\mathcal{PLE}$ executes $\mathcal{PUMP}$ for these two tables concurrently. A wave diffused with diffusion tables is called a *diffusion wave* and a wave diffused with removal tables is called a *removal wave*. A source for diffusing an ID (removal of an ID) is called diffusion source (removal source, respectively). We add an *remove* operation $remove(\ell, DfT_i)$ ($remove(\ell, RmT_i)$) on each entry of DfT (RmT, respectively) that removes the entry with ID ℓ.

To remove the initial fictitious IDs, an ID is diffused with the distance from the source. Each entry of DfT_i is a triple $(\ell, DD_i(\ell), DC_i(\ell))$ where $DD_i(\ell)$ is the distance value and $DC_i(\ell)$ is the counter value for ID ℓ. The value of $DD_i(\ell)$ takes $(i-j)$ for P_j where $ID_j = \ell$. We replace the create operation and increment operation defined in Section 3.1 as follows: Operation $create((\ell, c), T_i)$ is replaced with $create((\ell, c), DfT_i)$ that

- When c is 1 or $\perp$, creates an entry $(\ell, 0, c)$ in DfT_i.
- When c is 0 or ϕ, creates an entry $(\ell, DD_{i-1}(\ell)+1, c)$ in DfT_i.

Operation $increment(C_i(\ell))$ is replaced with $increment(DC_i(\ell))$ that increments $DC_i(\ell)$ by one and changes the value $DD_i(\ell)$ to $DD_{i-1}(\ell)+1$.

Protocol 3.2 $\mathcal{PLE}$ at P_i

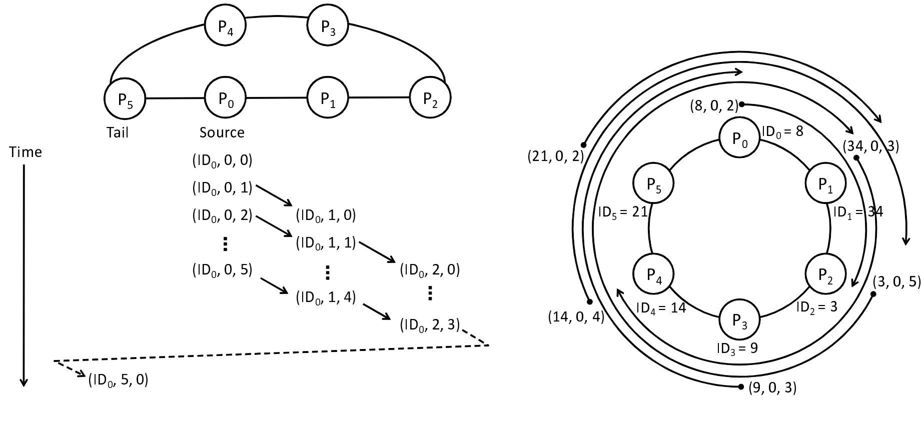
```
Local variables at $P_i$
    $LID_i$: leader's ID
    $DD_i(\ell)$: the distance value of entry of ID $\ell$ in $DfT_i$
    $DC_i(\ell)$: the counter value of entry of ID $\ell$ in $DfT_i$
    $RC_i(\ell)$: the counter value of entry of ID $\ell$ in $RmT_i$
Output variable at $P_i$
    $LID_i$
Predicates at $P_i$
    $IsDS_i(\ell) \equiv \ell = ID_i$
    $IsDF_i(\ell) \equiv \ell \ne ID_i \wedge ID_{i+1} \ne \ell \wedge (\ell, DD_{i-1}(\ell), DC_{i-1}(\ell)) \in DfT_{i-1}$
    $IsDT(\ell) \equiv ID_{i+1} = \ell$
    $IsRS_i(\ell) \equiv \{DC_{i-1}(\ell) = undef \wedge (\ell, DD_{i-1}(\ell), DC_i(\ell)) \in DfT_i \wedge \ell \ne ID_i\} \vee$
                  $\{\ell \ne ID_i(\ell, DD_{i-1}(\ell), DC_{i-1}(\ell)) \in DfT_{i-1} \wedge (\ell, DD_i(\ell), DC_i(\ell)) \in DfT_i \wedge$
                  $(\ell, DD_{i+1}(\ell), DC_{i+1}(\ell)) \in DfT_{i+1} \wedge DD_i(\ell) < DD_{i-1}(\ell) \wedge$
                  $DD_i(\ell) < DD_{i+1}(\ell)\}$
    $IsRF_i(\ell) \equiv (\ell, DD_{i-1}(\ell), DC_{i-1}(\ell)) \in DfT_{i-1} \wedge (\ell, DD_i(\ell), DC_i(\ell)) \in DfT_i \wedge$
                  $(\ell, DD_{i+1}(\ell), DC_{i+1}(\ell)) \in DfT_{i+1}$
    $IsRT(\ell) \equiv \{(\ell, DD_i(\ell), DC_i(\ell)) \in DfT_i \wedge DC_{i+1}(\ell) = undef\} \vee$
                  $\{ (\ell, DD_{i-1}(\ell), DC_{i-1}(\ell)) \in DfT_{i-1} \wedge (\ell, DD_i(\ell), DC_i(\ell)) \in DfT_i \wedge$
                  $(\ell, DD_{i+1}(\ell), DC_{i+1}(\ell)) \in DfT_{i+1} \wedge (RC_{i-1}(\ell) = 0)\}$

Actions at process $P_i$
    $S_1$  $true \longrightarrow$
           $\forall \ell$ execute $\mathcal{PUMP}(DfT_i, IsDS_i(\ell), IsDF_i(\ell), IsDT_i(\ell))$ // Diffusion wave
    $S_2$  $\exists \ell \ne ID_i : DC_{i-1}(\ell) = undef \wedge (\ell, DD_i(\ell), DC_i(\ell)) \in DfT_i \wedge DC_{i+1}(\ell) = undef$
           $\longrightarrow remove(\ell, DfT_i); remove(\ell, RmT_i)$ // Discarding locally
    $S_3$  $\neg\{\exists \ell \ne ID_i : DC_{i-1}(\ell) = undef \wedge (\ell, DD_i(\ell), DC_i(\ell)) \in DfT_i \wedge DC_{i+1}(\ell) = undef\}$
           $\longrightarrow \forall \ell \ne ID_i$ execute $\mathcal{PUMP}(RmT_i, IsRS_i(\ell), IsRF_i(\ell), IsRT_i(\ell))$ // Removal wave
    $S_4$  $\exists \ell \ne ID_i : RC_{i-1}(\ell) = \phi \wedge RC_i(\ell) = \phi \wedge RC_{i+1}(\ell) = undef \longrightarrow$
           $remove(\ell, DfT_i); remove(\ell, RmT_i)$ // Discarding fictitious IDs
    $S_5$  $\forall (\ell, DD_i(\ell), DC_i(\ell)) \in DfT_i : DC_i(\ell) = \phi \longrightarrow$
           $LID_i = \min\{\ell | (\ell, DD_i(\ell), DC_i(\ell)) \in DfT_i\}$ // Selecting leader's ID
```

An entry in RmT_i is a tuple denoted by $(\ell, RC_i(\ell))$.

In $\mathcal{PLE}$, each process P_i executes $\mathcal{PUMP}$ in S_1 for diffusing ID_i and also in S_3 for removal waves. The source, forwarder, and the tail for diffusion waves are defined by the three predicates $IsDS_i(\ell)$, $IsDF_i(\ell)$, and $IsDT_i(\ell)$. $IsDS_i(\ell)$ is evaluated to *true* if $ID_i = \ell$ holds, *i.e.*, P_i is the source of ID_i. $IsDT_i(\ell)$ is evaluated to *true* at P_{i-1}. Figure 1 shows an example of ID diffusion on a ring of six processes $P_0, P_1, \cdots, P_5$. Processes concurrently diffuse their IDs downstream.

The source, forwarder, and the tail for removing waves are defined by the three predicates $IsRS_i(\ell)$, $IsRF_i(\ell)$, and $IsRT_i(\ell)$. Process P_i is the source for removal of ID $\ell \ne ID_i$ when it finds it stores $(\ell, DD_(i)(\ell), DC_i(\ell))$ while its predecessor does not store



(*a*) Diffusion waves generated at process P_0 (*b*) Concurrent diffusion waves

図 1 Diffusion waves on a ring of six processes in $\mathcal{PLE}$

$(\ell, DD_{i-1}(\ell), DC_{i-1}(\ell))$ or when it finds its distant value is locally minimal. Process P_i becomes the tail for removal of ID ℓ when it stores $(\ell, DD_i(\ell), DC_i(\ell))$ while $DC_{i+1}(\ell)$ is *undef* at its successor P_{i+1} or its successor is a source of a removal wave. After the diffusion of removal waves for ID ℓ is finished, ID ℓ is removed by the execution of S_4 at the tail, each forwarder, and the source. When the fictitious ID is stored locally, it is removed by the execution of S_2.

The leader's ID variable is changed by the execution of S_5 only when the diffusions of IDs are finished.

A configuration is legitimate for $\mathcal{PLE}$, if the following predicate $LP_{\mathcal{PLE}}$ holds at any process P_i in V:

$$\begin{aligned} LP_{\mathcal{PLE}} \equiv \{LID_i = \min\{ID_\ell | P_\ell \in V)\}\} \wedge \{\forall P_j \in V : DC_i(ID_j) = \phi\} \wedge \\ \{\forall(\ell, DD_i(\ell), DC_i(\ell)) \in DfT_i : \exists P_j \in V : ID_j = \ell\} a \end{aligned}$$

4. Correctness proof

We present the sketch of the correctness proofs and performance evaluation of $\mathcal{PUMP}$ and $\mathcal{PLE}$. $\mathcal{PUMP}$ guarantees self-stabilization and fictitious diffusion containment for a given ID ℓ. Based on these properties of $\mathcal{PUMP}$, $\mathcal{PLE}$ guarantees self-stabilization and TB-Byzantine resilient fault-containment.

We first show self-stabilization of $\mathcal{PUMP}$. In the following, we focus on executions of $\mathcal{PUMP}$ for an ID ℓ, the source process P_s, and the tail process P_t for ℓ.

During the convergence, there are three types of waves, diffusion waves, reset waves, and acknowledgment waves. Until the system reaches a legitimate configuration, at least one of the three waves is forwarded in every round. Hence, after the first $2(t-s)-1$ rounds, P_s receives no more reset signal. Then, the source process keeps on generating waves and eventually the diffusion is completed. Because the forwarding of waves is pipelined, it takes all the diffusion waves to be forwarded at most $2(t-s)-1$ rounds in a correct execution. After that, the acknowledgment signal is forwarded from the tail to the source through forwarders.

Theorem1 $\mathcal{PUMP}$ is self-stabilizing and for diffusion from P_s to P_t, the convergence time is $5(t-s)-2$ rounds.

The propagation of reset signal and acknowledgment signal is implemented with *fast* waves that are forwarded immediately to the neighbors. On the other hand, the diffusion of IDs is implemented with *slow* waves that the source should keep on generating waves with incrementing the TTL value and the forwarders allow these waves to propagate in the FIFO order. To diffuse a fictitious ID to k' processes, a TB-Byzantine process should generate a sequence of waves with TTL values of $1, 2, \cdots, k'$. It requires k' malicious action at the TB-Byzantine process. Hence, a k'-TB-Byzantine process diffuses fictitious IDs to at most k' correct processes.

Theorem2 $\mathcal{PUMP}$ has the property of fictitious diffusion containment.

Then, we show the self-stabilization and TB-Byzantine resilient fault-containment of $\mathcal{PLE}$. $\mathcal{PUMP}$ does not remove fictitious IDs after it is propagated to a small number of perturbed processes. By using $\mathcal{PUMP}$ for diffusion waves and removal waves, $\mathcal{PLE}$ achieves self-stabilization and fault-containment for the leader election problem.

For an existing process P_i, the diffusion source does not change during any execution. Hence, until the diffusion of ID_i is completed, $\mathcal{PUMP}$ continuously generates waves for ID_i and IDs of existing processes are eventually stored at diffusion tables at all processes by the execution of $\mathcal{PUMP}$.

For each fictitious ID, the source process for removal are fixed in the first n rounds. We consider a maximum sequence of processes $P_{s'}, P_{s'+1}, \cdots, P_{t'}$ where each process stores an entry for ℓ in its diffusion table. Maximality means, $P_{s'-1}$ and $P_{t'+1}$ do not store an entry for ℓ in their diffusion tables or $P_{s'-1} = P_{t'}$. After the first n rounds, we have at least one removal source for ℓ. If all processes stores ℓ in its diffusion table, there exists at least one process P_j that has the locally minimum distance value for ℓ but $ID_j \neq \ell$, otherwise, there exists at least one process whose predecessor does not store the entry for ℓ in its diffusion table. In both cases, the removal sources and tails are fixed in the first n rounds. After that, $\mathcal{PUMP}$ guarantees that eventually removal waves are completed and all processes remove the entry for ℓ by the execution of S_3. Consequently, all fictitious IDs are eventually removed. And the leader ID is selected from a correct list of locally stored IDs at each process.

Theorem3 $\mathcal{PLE}$ is self-stabilizing.

Process P_i is perturbed when the process changes LID_i since the output variable of $\mathcal{PLE}$ is LID_i. $\mathcal{PLE}$ allows each process to change its LID only when all diffusion waves are locally completed. Hence, to make a correct process change its LID, it is necessary that at least two TB-Byzantine processes cooperate to finish the diffusion or removal using $\mathcal{PUMP}$. Consequently, from Theorem 2, at most $(f'-1)k'$ processes change their LID values during (f', k')-perturbation.

After the final malicious action in the perturbation, $\mathcal{PLE}$ diffuses IDs of existing processes and removes fictitious IDs. All these correcting actions progress concurrently, however, S_5 allows each process to change its LID value only after all the diffusion is completed locally. Hence, for any (f', k')-perturbation, the number of perturbed processes is at most $\min\{(f'-1)k', n\}$. After the last malicious action, the diffusion source of a removed ID starts diffusion of its ID with a wave of TTL 1 and the perturbation time depends on the convergence time of $\mathcal{PLE}$.

Theorem4 $\mathcal{PLE}$ is TB-Byzantine resilient fault-containing and the (f', k')-perturbation number is $\min\{f' \cdot k', n\}$. The (f', k')-perturbation time is $(\min\{f' \cdot k', n\} + 1)(5n-2)$ rounds.

5. Conclusion

In this paper, we introduced a novel fault model called TB-Byzantine fault and proposed an adaptive fault-containment method against TB-Byzantine faults for the leader election problem. Though the perturbation number of the proposed protocol is bounded by the number of malicious actions during a perturbation, the time complexity depends on the number of processes in the entire system. Our future work is to develop a more efficient self-stabilizing and fault-containing technique against TB-Byzantine faults.

謝辞 This work is supported in part by MEXT Global COE Program, JSPS Grant-in-Aid for Scientific Research ((B)22300009), and JSPS Grant-in-Aid for Young Scientists (Start-up) (21800031).

参 考 文 献

1) Barak, B., Halevi, S., Herzberg, A. and Naor, D.: Clock synchronization with faults and recoveries (extended abstract), *Proc. of 9th Annual ACM Symposium on Principles of Distributed Computing*, pp.133–142 (2000).
2) Biely, M. and Huttle, M.: Consensus when all processes may be Byzantine for some time, *Proc. of 11th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems*, pp.120–132 (2009).
3) Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control, *Communications of ACM*, Vol.17, No.11, pp.643–644 (1974).
4) Dubois, S., Masuzawa, T. and Tixeuil, S.: The impact of topology on Byzantine containment in stabilization, *Proc. of 24th International Symposium on Distributed Computing*, p.July (2010).
5) Masuzawa, T. and Tixeuil, S.: Bounding the impact of unbounded attacks in stabilization, *Proc. of 8th Int'l Symposium on Stabilization, Safety and Security of Distributed Systems*, pp.440–453 (2006).
6) Nesterenko, M. and Arora, A.: Dining philosophers that tolerate malicious crashes, *Proc. of 22nd Int'l Conf. on Distributed Computing Systems*, p.191 (2002).
7) Nesterenko, M. and Arora, A.: Tolerance to unbounded byzantine faults, *Proc. of 21st Symposium on Reliable Distributed Systems*, pp.22–29 (2002).
8) Widder, J., Gridling, G., Weiss, B. and Blanquart, J.: Synchronous consensus with mortal Byzantines, *Proc. of 37th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks*, pp.102–112 (2007).