2006－AL－104（6）
2006／1／20

木ネットワーク上での トークン巡回故障封じ込め自己安定プロトコル

中村友貴，片山喜章，高橋直久
名古屋工業大学大学院工学研究科情報工学専攻

概要: 自己安定プロトコルとは，任意のネットワーク状況から開始しても，有限時間内に解を求めて安定する分散プロトコルである．このためネットワーク中でプロセスの一時故障が生じても，その状況を初期状況と考えれば，やがて解状況に到達することから故障耐性を持つプロトコルである．一般に分散システム上で，同時に多数のプロセスが故障することはまれである．よって小規模な故障による故障状況からの実行において，効率よく解状況に到達することが望ましい．そこで本稿では，1 故障状況からの再安定時間が $O(1)$，変動プロセス数が $2\triangle+2$ である木ネットワーク上でのトークン巡回のための故障封じ込め自己安定プロトコルを提案する．

A Fault-Containment Self-Stabilizing Protocol for Token Passing on Tree Networks.

Tomoki NAKAMURA, Yoshiaki KATAYAMA, Naohisa TAKAHASHI
Department of Computer Science and Engineering,
Graduate School of Engineering, Nagoya Institute of Technology

abstract: Self-stabilizing protocols can tolerate any type and any number of transient faults. Generally, it is rare case that large number of faults occurs simultaneously. We propose a fault-containment self-stabilizing protocol for token passing on trees. From 1-faulty configuration, proposed protocol can stabilize $O(1)$ time, and its contamination number is $2\triangle+2$.

1 はじめに

自己安定プロトコルとは，分散システム (通信リンクによって接続された複数のプロセスで構成されるシステム) 上でプロセスのプログラムカウンタの破壊，通信メッセージの破壊や改変といった一時的な故障 (一時故障) が生じても，やがて解状況 (正当な状況) に到達する分散プロトコルである [10]．従って，自己安定プロトコルは通常の分散プロトコルとは異なり，プロトコル開始時のネットワーク状況 (初期状況) を仮定する必要はなく，任意の状況からプロトコルを開始しても解状況に到達する．これらのことから，自己安定プロトコルは一時故障に対し耐性を持ち，分散システムの初期化が不要である特徴を持つ．

分散システム上で分散プロトコルを運用する場合，同時に多数のプロセスが故障を起こすことはあまり考えられない．一般には少数のプロセスが一時故障を起こし，解状況から少し変動した状況 (小変動状況) になる．自己安定プロトコルはこの状況から有限時間内に解状況に到達することを保障するが，システム全体が少数のプロセスの故障による影響うけたり，修復が明らかに簡単な場合でも長い時間かけて再安定するときもある．このため，少数のプロセスが一時故障を起こした状況から解状況に到達するまでの過程を考慮したプロトコルの設計は非常に重要である．

ネットワーク上でトークンを巡回させる自己安定プロトコルは数多く研究されている ([1]〜[6])．しかし，小変動状況から再安定の実行を考慮した研究は少ない ([3])．既存のプロトコルでは故障によって生じたトークン Tf は再安定するまでの間，ネットワーク内を移動し，故障の影響がシステム全体に及ぶ可能性がある．

そこで本稿では 1 故障状況から再安定するまでの実行において，一時故障によって生じた冗長なトークンを，他のプロセスに移動させないことで，故障の影響を局所的に抑えるプロトコルを提案する．トークン巡回は相互排除問題の解決や各端末から/へ情報を収集・配信するのに有効な手段の一つである．これらのことを効率的に行うために生成木を用いる．生成木上を深さ優先でトークンが巡回すれば，2n-2 回 (n はプロセスの数) の移動でトークンは全てのプロセスを巡回可能である．また [7] や [8] のように生成木構成問題を解く自己安定プロトコルも研究されている．そこで，木ネットワーク上を深さ優先でトークンを巡回させる故障封じ込め自己安定プロトコルを提案する．

2 モデル

本章では扱うネットワーク及びプロトコルなどのモデルに関して述べる．

2.1 分散システム

分散システムは，無向連結グラフ $D=(P,L)$ で定義される．ここで，P は頂点集合 ($P=\{p_r,p_1,p_2,\cdots,p_{n-1}\}$)，$L$ は辺の集合とする．それぞれの頂点はプロセスを表し，辺は双方向通信リンクを表す．本論文では木構造のネットワークを扱う．根プロセスを p_r とする．各プロセスは相異なる識別子をもっており，簡単の為にプロセス p_i と識別子を区別せず，単に p_i と書く．また $(p_i,p_j)\in L$ ならプロセス p_i とプロセス p_j は互いに隣接していると呼び，p_i の隣接プロセス集合は N_i で表される．各プロセス p_i は隣接するプロセスが自分の親であるのか，子であるのかを区別ができるものとする．p_i の親プロセスは $p_i.P$(但し，$p_r.P=Null$)，また子プロセス集合は $Children_i$ で表される．

各プロセス p_i は状態変数 $p_i.S$ と CurrentChild ポインタ (CC ポインタ)$p_i.CC$ をもつ．状態変数 は根プロセス p_r は 0,2 を，葉プロセスは 1,3 を，内部接点は 0,1,2,3 のいずれかの値をとる．状態変数は次のようなローテーション関数となっている．$0\overrightarrow{+1}1\overrightarrow{+1}2\overrightarrow{+1}3\overrightarrow{+1}0\overrightarrow{+1}1$・・・の

ように 3+1=0 になる．つまり $|3-0| = |0-3| = 1$ である．CC ポインタ $p_i.CC$ は p_i の子プロセス集合 $Children_i$ のうち 1 つのプロセスを指すポインタであり，p_i によって指されているプロセスを p_i の CurrentChild(以後 CC) と呼ぶ (但し，葉プロセス p_l では $p_l.CC = Null$)．$Children_i$ の各要素は順序付けされており，その順序に従って p_i は CC ポインタが指すプロセスを変えていく．CC ポインタが x 番目に指すプロセスは $ChildrenList_i(x)$ で表され，$|CurrenChild_i| = m(0 \leq m \leq n-1)$ とすると，$ChildrenList_i(m)$ を $p_i.LastC$ と表す．また，CC ポインタは $p_i.LastC$ を指した後は，再び $ChildrenList_i(1)(p_i.FirstC$ と表す) を指す．

通信方式は，隣接する全てのプロセスの内部状態 (識別子やローカル変数) を直接参照することができる状態通信モデルを仮定する．

2.2 デーモン

ネットワーク上でのプロトコルの計算状況 (ネットワーク状況) を各プロセスの状態を列挙することにより表す．各プロセス p_i の状態 (ローカル変数などの内部状態) を s_i とし，ある時点でのネットワーク状況を $c = (s_r, s_2, \cdots, s_{n-1}, (P, L))$ と表す．またネットワークのとり得る全てのネットワーク状況の集合を C と表す．つまり，p_i がとり得る全ての状態を Q_i とすると，$C = S_r \times S_2 \times \cdots S_{n-1}$ である．

プロセスの部分集合を $S \subseteq P$ とする．あるネットワーク状況 $c_i \subseteq C$ で，S に属するプロセスが同時にアルゴリズム A の 1 原子動作を実行することによって c_{i+1} になったとき，$c_{i+1} = c_i(S, A)$ と表す．

定義 1 (スケジュール)．空でないプロセッサ集合の無限系列をスケジュールと呼ぶ．プロトコル A，スケジュル $T = Q(0), Q(1), Q(2), \cdots$．について，ネットワークの無限系列 $E = c_0, c_1, c_2, \cdots$ が $c_{i+1} = c_i(Q(i), A)$ を満たすとき，E を「初期状況 c_0, スケジュール T に対するプロトコル A の実行」と呼び，$E(A, T, c_0)$ と表す． □

定義 2 (公平なスケジュール)．無限系列であるスケジュール T に全てのプロセス $p_i \in P$ が無限回現れる時，T は公平なスケジュールである． □

本稿では，公平なスケジュールのみを対象とし，単にスケジュールと呼ぶ．スケジュール T に含まれる各プロセスは 1 原子動作のみを行うことが出来る．スケジュール T に含まれるプロセス集合 Q の大きさと原子動作の違いによって，いくつかのモデルが考えられているが，本研究では以下のモデルを扱う (C デーモンと呼ぶ)．

・プロセス数：任意の $t(t \geq 0)$ について $|Q(t)| \geq 1$.

・原子動作：全隣接プロセスから内部状態を読み込み，自分の内部状態を変更.

2.3 自己安定プロトコル

$LE \subseteq C$ を，ネットワーク状況の任意の集合とする．次の (1), (2) の条件を満たすとき，「プロトコル A は LE に関して自己安定である」といい，$SS(A, LE)$ と書く．また，$SS(A, LE)$ が成立するとき，LS を「プロトコル A に関して正当な状況」という．ただし，A が明らかな場合，単に正当な状況という．

(1) 到達可能性

任意のネットワーク状況 $c \in C$ と任意のスケジュール T に対し，c から始まるスケジュール T によるプロトコル A の実行 $E(A, T, c)$ に，LS に含まれるネットワーク状況が現れる．つまり，$E(A, T, c_0) = c_0, c_1, \cdots$ とするとき，$c_i \in LS$ となる $i \geq 0$ が存在する．

(2) 閉包性

LE 中の任意のネットワーク状況を c，プロセスの任意の集合を S とする．このとき，$c' = c(Q, A)$ が LE に属する．

2.4 故障封じ込め

本稿では正当な状況に安定後，単一プロセスの一時故障からの復旧に対して優れた特性を持つ自己安定プロトコルについて考察する．

定義 3 (1 故障状況)．プロトコル A を，正当な状況 LE に関して自己安定なプロトコルとする．あるネットワーク状況 $c \in LE$ において一つのプロセスの状況を任意に変えることによって得られる状況 c' とする．この時 $c' \notin LE$ ならば，状況 c' を 1 故障状況とよび，状態を変えたプロセスを故障プロセスと呼ぶ． □

実システムでは，1 故障状況から始まる任意の実行における再安定までに動作するプロセス数，再安定までの時間は重要である．つまり，故障の影響を局所的に抑え，できる限り迅速に再安定することが望ましい．これらの特徴を持つ自己安定プロトコルを故障封じ込め自己安定プロトコルという．

定義 4 (同期スケジュール)．スケジュール $T = Q(0), Q(1), \cdots$ において，任意の $i(i \geq 1)$ について $Q(i) = P$ を満たすとき，スケジュール T を同期スケジュールと呼ぶ． □

定義 5 (変動プロセス数と再安定時間)．プロトコル A を正当な状況 LE に関する自己安定プロトコルとする．任意の *1* 故障状況 c から始まる任意の実行 E が再び LE を満たすまで *(*再安定するまで*)* に状態遷移する最大プロセス数を「変動プロセス数 *(contamination number)*」，それにかかる時間 *(*同期スケジュールにおいて再安定するまでの時間*)* を再安定時間 *(convergence time)*」という． □

一般に故障封じ込めの性能は，この変動プロセス数と再安定時間で評価される．しかし，トークン巡回問題では解状況でも動作可能なプロセスが存在する問題である．この性質から，故障封じ込めプロトコルを施しても，故障プロセス，及びその故障プロセスから定数距離あるプロセスは動作しないが，故障前から存在するトークン (正トークンと呼ぶ) の移動に伴って，プロセスが動作し続ける場合がある．このため，単純に，故障が生じた状況から再安定までに変動するプロセスの数や，再安定時間を評価対象にするのは難しい．そこで，本稿では変動プロセス数を次のように定義する．

定義 6 (変動プロセス数)．プロトコル A を正当な状況 LE に関する自己安定プロトコルとする．任意の *1* 故障状況 c から始まる任意の実行 E が再び LE を満たすまで，正トークンの移動に関係なく状態遷移する最大プロセス数を変動プロセス数とする． □

3 故障封じ込めトークン巡回自己安定プロトコル $FCTP$

本章では，トークン巡回故障封じ込め自己安定プロトコル $FCTP$ を示す．これはネットワーク中にトークン

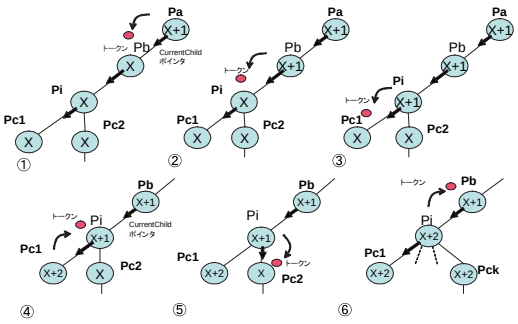

図 1: プロトコル TP の実行

が複数存在する状況からプロトコルを開始してもやがてトークンは唯一な状況 (正当な状況と呼ぶ) となり，そのトークンは木ネットワーク上を深さ優先で巡回するプロトコルである．また $FCTP$ は単一プロセスよる故障の影響を封じ込める性質を持つ．これはプロセスの一時故障によって冗長なトークン (故障トークンと呼ぶ) が生じても，その 1 故障状況から再安定するまでの実行において，その故障トークンが他のプロセスに移動しないことを保障し，かつすばやく再安定させることで故障の影響を局所的に抑えるものである．まず，トークンを唯一とし深さ優先で巡回させる (自己安定) アイデアとその詳細を説明したあと，故障封じ込めについて詳細を説明する．

3.1 $FCTP$ の自己安定のアイデア

$FCTP$ は，ふたつの部分 TP と RP からなる．TP はトークンを生成木上を巡回させるための部分であり，RP は各プロセスの局所情報から，正当な状況でないと判断された時に正当な状況になるよう状態遷移させるための部分である．各プロセスは，局所情報から TP,RP いずれかの部分を実行し，トークン巡回自己安定プロトコルを実現している．

FCTP においては，ネットワーク中に常に少なくともひとつのトークンが存在するようにトークンを定義する．さらにネットワーク中でトークンが唯一存在する状況において，任意のプロセス p_i で次の二つのうちいずれかの条件を満たす（述語 $cons$ と呼ぶ) よう設計する（詳細は後述）．(条件 1)$p_i.S$ が偶数，かつ $(p_i.P).S$ および $((p_i.P).CC).S$ が偶数 (条件 2)$p_i.S$ が奇数，かつ $\forall p_j \in Children_i, p_j.S$ が奇数つまり，上記の条件のいずれも満たさないプロセスが存在すれば，ネットワーク中にトークンが二つ以上存在することになる．

FCTP の RP 部分は，$cons$ が満たされないプロセスが $cons$ が満たされるよう状態を変化させる．これを繰り返すことで正当な状況に到達する．一方，TP 部分は，トークンを深さ優先で巡回させる．

以降では，まず TP について述べ，続いて RP の詳細について説明する．

3.1.1 プロトコル TP

TP は文献 [5] で紹介されており，木ネットワーク上でトークンの深さ優先の巡回を実現しているプロトコルである．TP の動作を説明する前にトークンの定義を行う．

定義 7 (トークン)．プロセス p_i が次の条件 $To1$，$To2$ のいずれかを満たす場合，p_i はトークンを持つ．(**p_p は p_i の親プロセス，p_c は p_i の子プロセスとする**)

To1　$p_i.S =$ 奇数 $\wedge\ p_p.S = p_i.S + 1 \wedge p_p.CC = p_i$

To2　$p_i.S =$ 偶数 $\wedge\ p_c.S = p_i.S + 1 \wedge p_c = p_i.CC$　□

TP の動作について図 1 に示して説明する. 図 1 ではプロセス内の値は状態変数 S の値を示しており，また矢印は各プロセスの CC ポインタを示す. 図 1.　では p_b はトークンの条件 To1 を満たすことから，トークンを持つ. To1 を満たすプロセス p_i は自分の状態変数 $p_b.S$ の値を 1 増加させると，図 1.　のように p_b の子プロセス p_i がトークンの条件 To1 を満たす．つまり，p_b が p_i にトークンを送信する．このように親から子へ CurrentChild ポインタに沿って葉プロセスまでトークンが送信される (図 2.　～　)．葉プロセス p_{c1} がトークンを受信すると，状態変数 $p_{c1}.S$ の値を 2 増加させることで，p_{c1} の親プロセス p_i にトークンを返す (図 1.　～　).

次に子から親プロセスにトークンが送信される動作を示す．図 1.　のように p_i がトークンを子プロセスから受信すると，自分の CC ポインタの指すプロセスを p_{c2} に変更する．p_{c2} が To1 の条件を満たし，トークンを持つようになる（図 1.　）．このようにプロセスはトークンを子プロセスから受信すると，他の子プロセスにトークンを送信する．また，全ての子プロセスに 1 度トークンを送信した後，再びトークンを子プロセスから受信すると，図 1.　のように自分の状態変数の値を 1 増加させることで親プロセスにトークンを送信する．これらを繰り返すことでトークンは木ネットワーク上を深さ優先で巡回する．

次に TP の詳細を説明する．まずプロトコル中で使われる関数を定義する．(但し，$p_p = p_i.P$，$m = |Children_i|$，$p_i.CC$=$ChildrenList_i(k)$ とする)

・$NEXTC(p_i)$：$ChildrenList_i(k+1)$ を返す関数. 但し，$k = m$ ならば $p_i.FirstC$ を返す.

・$LASTC(p_i)$：$p_i.CC = p_i.LastC$ ならば真を返す関数.

・$CURRENTC(p_i)$：$p_p.CC = p_i$ ならば真を返す関数.

・$s\_even(p_i)$：$p_i.S$ の値が偶数ならば真を返す関数

TP のプロトコルを図 2 に示す．プロトコルはいくつかの文 (statement) からなり各プロセスに対して与えられる．文は条件部 (gurad) とそれに対する動作部 (action) から構成される．各プロセスのいずれかの文が成り立つ時，そのプロセスは動作可能と呼ぶ．動作可能なプロセスがデーモンにスケジュールされた時，プロセスは条件部が成立する文の動作部を不可分に実行する．プロセス p_i がトークンを持つならば，$T0$～$T5$ のいずれかの条件部を満たし, 動作部を実行することで, p_i はトークンを隣接プロセスに送信する．

3.1.2 プロトコル RP

RP はトークンを唯一にするために用いられるプロトコルである．詳細を示すための準備としてトークンが唯一である状況において，各プロセスが満たすべき条件 (述語 $cons$) を定義する．まず，その説明で用いる変数及び関数を定義する．

・$P\_even\_Children_i$：p_i の子プロセスの中で状態変数 S の値が偶数であるプロセスの集合.

・$p\_even\_Child(p_i)$：$|P\_even\_Children_i| = 1$ ならば，p_i の子プロセスの中で状態変数 S の値が偶数であるプロ

{ For the root process p_r }
[T0 :] $(p_r.CC.S = p_r.S + 1) \wedge \neg LASTC(p_r)$
$\longrightarrow p_r.CC := NEXTC(p_r);$
[T1 :] $(p_r.CC.S = p_r.S + 1) \wedge LASTC(p_r)$
$\longrightarrow p_r.S = p_r.S + 2;$
$p_r.CC := NEXTC(p_r);$
{ For the leaf process p_l }
[T2 :] $(p_l.P.S = p_l.S + 1) \wedge CURRENTC(p_l)$
$\longrightarrow p_l.S := p_l.S + 2;$
{ For the inner process p_i }
[T3 :] $\neg s\_even(p_i) \wedge (p_i.P.S = p_i.S + 1) \wedge$
$CURRENTC(p_i)$
$\longrightarrow p_i.S := p_i.S + 1;$
[T4 ::] $s\_even(p_i) \wedge (p_i.CC.S = p_i.S + 1) \wedge \neg LASTC(p_i)$
$\longrightarrow p_i.CC := NEXTC(p_i);$
[T5 :] $s\_even(p_i) \wedge (p_i.CC.S = p_i.S + 1) \wedge LASTC(p_i)$
$\longrightarrow p_i.S = p_i.S + 1;$
$p_i.CC := NEXTC(p_i);$

図 2: プロトコル TP

セスの ID を返す関数．$|P\_even\_Children_i| \neq 1$ ならば $null$ を返す．

トークンの定義から以下のことが言える．

観測 1．プロセス p_k の CC を p_j とする．$p_k.S$=偶数かつ $p_j.S$=奇数ならば，p_k，p_j のいずれかは必ずトークンを持つ． □

観測 2．任意の状況において，プロセス p_i の状態変数 $p_i.S$ の値が偶数ならば，p_i から CC ポインタに沿って葉プロセスまでの経路上に，トークンを持つプロセスが少なくとも一つ存在する．また，ネットワーク中に少なくともトークンを持つプロセスが一つ存在する．

証明. 全ての葉プロセス p_l は必ず $p_l.S$=奇数であることと，観測 1 から明らか． □

p_i から CC ポインタに沿った葉プロセスまでの経路を「p_i の C 経路」と呼ぶ．観測 2 より各プロセス p_i は隣接プロセスの状態から，ネットワーク上にトークンが二つ以上存在することを推測できる．例えば，$p_i.S$=奇数であり，$p_c.S$=偶数 (p_c は p_i の子プロセス) の時を考える．観測 2 より根プロセス p_r の C 経路上にトークンを持つプロセスが少なくとも一つ存在し，$p_c.S$=偶数より p_c の C 経路上にもトークンが存在するからである．このことからトークンが唯一である状況で各プロセス p_i が隣接プロセス p_j に対して満たすべき条件 (述語 $cons_i(p_j)$) を次に示す．

定義 8 ($cons_i(p_j)$)．プロセス p_i の隣接プロセスを p_j とする．以下に定義される述語 $cons_i(p_j)$ を満たす時，p_i は「p_j に対して無矛盾」という．

{ p_jが p_iの子プロセスの時 $(p_i = p_j.P)$ }
$cons_i(p_j)=$
$(s\_even(p_i) \wedge \neg s\_even(p_j) \wedge \neg CURRENTC(p_i)) \vee$
$(s\_even(p_i) \wedge CURRENTC(p_i)) \vee$
$(\neg s\_even(p_i) \wedge \neg s\_even(p_j))$
{ p_jが p_iの親プロセスの時 $(p_j = p_i.P)$ }
$cons_i(p_j)=$
$(s\_even(p_i) \wedge s\_even(p_j) \wedge CURRENTC(p_i)) \vee$
$(\neg s\_even(p_i))$

但し，$cons_r(p_r.P)$ は常に真とする． □

従って，p_i の任意の隣接プロセスを p_x とすると $cons_i(p_x)$=$cons_x(p_i)$ である．

また，述語 $cons_i(p_i)$ を定義する．

定義 9 (無矛盾状態)．以下の条件を満たすとき，かつそのときのみ $cons_i(pi)$ が成立し，そのとき p_i は無矛盾状態であるという．

$[\forall p_j \in N_i, cons_i(p_j) = TRUE]$

また，無矛盾状態でない状態を矛盾状態であるという． □

簡単のためにプロセス p_i で $cons_i(p_i)$ が成立することを，「プロセス p_i で cons が成立する」と呼ぶ．

定義 10 (正常化代入)．矛盾状態のプロセス p_j を無矛盾にするような $p_j.S$ 及び $p_j.CC$ が存在する時，その代入を「正常化代入」と呼ぶ．p_j の正常化代入の有無を表す述語 $can\_Cons(p_j)$ は次のように定義される．

{ ***For the root process*** p_r }
$can\_Cons(p_r)$=$\neg cons_r(p_r) \wedge |P\_even\_Children_r| \leq 1$
{ ***For the leaf process*** p_l }
$can\_Cons(p_l)$=$FALSE$
{ ***For the inner process*** p_i }
$can\_Cons(p_i)$=$\neg cons_i(p_i) \wedge$
$((|P\_even\_Children_i| = 0)$
$(|P\_even\_Children_i| = 1 \wedge s\_even(p_i.P) \wedge$
$CURRENTC(p_i))$ □

観測 2 及び $cons$ の定義を用いて以下のことが導かれる．

補題 1．$cons$ が成立しないプロセスが存在すれば，ネットワーク中に少なくとも二つ以上のトークンが存在する． □

補題 2．全てのプロセス p_i で $cons_i(p_i)$ が成立すれば，ネットワーク中にトークンは唯一存在する．

証明. ネットワーク中にトークンは二つ以上存在すると仮定し，背理法を用いて証明する．トークンを持つプロセスを p_a, p_b とし，それぞれの親プロセスを p_x, p_y とする．トークン及び $cons$ の定義より，$p_x.S$ = 偶数, $p_y.S$= 偶数を満たす．また，$cons$ の定義より，状態変数 S の値が偶数ならば，その親プロセスの状態変数 S の値も偶数である．つまり，p_x から p_r までの経路上，及び p_y から p_r までの経路上に存在するプロセスの状態変数 S の値は偶数である．よって，状態変数 S の値が偶数である子プロセスを二つ持つプロセス p_z が存在することになる．$cons$ の定義より $cons_z(p_z)$=$FALSE$ となり，仮定と矛盾する． □

補題 1 と補題 2 から次のことが言える．

定義 11 (正当な状況)．全てのプロセス p_i で $cons_i(p_i)$ が成立する状況を正当な状況という． □

つまり，正当な状況とは「ネットワーク中にトークンが唯一存在する」状況である．

これらのことを用いて，RP のアイデアとその詳細 (図 3) を示す．補題 1 より正当な状況，つまりトークンが唯一になるには，全てのプロセスで $cons$ が成立しなければならない．よって，$cons$ が成立しないプロセス p_i は正常化代入を行う ($R0$,$R3$)．しかし，必ずしも正常化代入があるとは限らない．そこで正常化代入が存在しない場合，自分の親プロセス $p_i.P$ に対して無矛盾となる，つまり $cons_i(p_i.P) = TRUE$ となるように p_i は状態を変

{ For the root process p_r }
[R0 :] $|P\_even\_Children_r| = 1;$
$\longrightarrow p_r.CC := p\_even\_Child(p_r);$
{ For the leef process p_l }
no move.
{ For the inner process p_i }
[R3 :] $s\_even(p_i.P) \wedge CURRENTC(p_i) \wedge$
$|P\_even\_Children_i| = 1;$
$\longrightarrow p_i.S := p_i.P.S;$
$p_i.CC : p\_even\_Child(p_i);$
[R4 :] $s\_even(p_i.P) \wedge CURRENTC(p_i) \wedge$
$|P\_even\_Children_i| \neq 1;$
$\longrightarrow p_i.S := p_i.P.S;$
[R5 :] $s\_even(p_i.P) \wedge \neg(CURRENTC(p_i);$
$\longrightarrow p_i.S := p_i.P.S + 1;$
[R6 :] $\neg s\_even(p_i.P);$
$\longrightarrow p_i.S = p_i.P.S;$

図 3: プロトコル RP

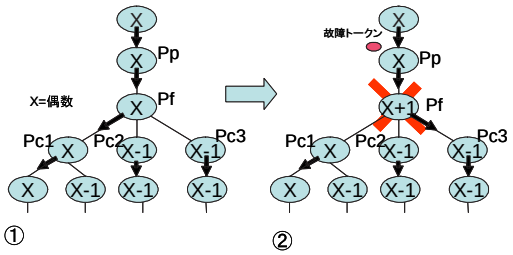

図 4: 1 故障状況

化させる ($R3, R4, R5$)．$cons$ の定義より全てのプロセス p_j で親プロセスと $cons_j(p_j.P)$ が成立すれば，全てのプロセスで $cons$ が成立するので，これを繰り返すことでやがてすべてのプロセスが $cons$ が成立し，正当な状況に到達する．

3.2 故障封じ込め

正当な状況からあるプロセスに一時故障が生じた場合のシステムの動作を考える．本稿では一時故障とはプロトコルの意図に反して，プロセスの状態が変化することをいう．図 4.　において p_f で一時故障が図 4.　になったとする．p_f の一時故障により，p_p はトークンを持つことになり，そのトークンがシステム内を巡回し始め，システム全体に影響を与えることになる．これを防ぐには 1 故障状況からの再安定において，一時故障によって生じたトークン (故障トークンと呼ぶ) を持つプロセスはトークンを送信しないことを保障すればよい．また，すばやく再安定するには故障プロセスが自分の状態を故障直前の状態に変更するのが望ましい．以下，FCTP における故障封じ込めの手法について述べる．

3.2.1 詳細

正当な状況 c において p_f の一時故障によって，1 故障状況 c_f になったとする．c_f 前から存在するトークンを正トークンと呼ぶ．c_f 以降，正トークンを持つプロセス以外で動作可能 (つまり TP 及び RP を実行し，状態遷移が可能) なのは，矛盾状態であるプロセス，または故障トークンを持つプロセスである．

まず矛盾状態のプロセス p_i を考える．$cons$ の定義より c_f において矛盾状態であるプロセスは高々 p_f とその隣接プロセスのみである．また，can_Cons の定義より

S1 :　$Q1$　→プロトコル TP を実行
S2 :　$Q2 \vee Q3$ →プロトコル RP を実行

図 5: 提案プロトコルの概略

c_f において p_f で can_Cons が成立するのは明らか．このことから，次のことが言える．

(条件 1)$p_i = p_f$ ならば: $cons_i(p_i)$=$FALSE$ かつ $can\_Cons(p_i)$=$TRUE$ が成立．

(条件 2)$p_i \in N_f$ ならば: p_i の隣接プロセスの中に $cons$ が成り立たないプロセス p_x が唯一存在し，その p_x で $can\_Cons(p_x)$ が成立．

このように (条件 1) または (条件 2) を満たすプロセスは，1 故障状況だと判断する．(条件 1) を満たすプロセス p_j はプロトコル RP に従って動作すれば，$cons_j(p_j)$ となり，全てのプロセスで $cons$ が成立するので，正当な状況に再安定する．また (条件 2) を満たすプロセス p_k が自分の状態を変化させないことで，故障の影響を抑える．

次に故障トークンを持つプロセス p_t を考える．p_t が故障トークンを持つのは，p_t 自身が故障プロセスの場合，もしくはトークンの定義より $p_f.S$=奇数ならば p_t の親が故障プロセス，$p_t.S$=偶数ならば p_t の CC が故障プロセスの場合のみである．よって，故障トークンを巡回させないために，各プロセスは自分が無矛盾状態かつ，親プロセスが無矛盾状態もしくは自分の CC が無矛盾状態を満たさない限り，トークンを隣接プロセスに送信しないという制限を加える．

以上より，$FCTP$ の詳細を示す．まずはプロトコル中で用いられる関数を定義する．

- $\#fa\_neigh(p_i)$: p_i の全隣接プロセスの中で cons が成立しないプロセスの数．つまり，
  $\#fa\_neigh(p_i)$=$|\{p_j \in N_i | \neg cons_j(p_j)\}|$
- $\#ca\_neigh(p_i)$: p_i の全隣接プロセスの中で can_Cons が成立するプロセスの数．つまり，
  $\#ca\_neigh(p_i)$=$|\{p_j \in N_i | can\_Cons_j(p_j)\}|$
- $T(p_i)$: p_i がトークンを持つならば真を返す関数．
  $T(p_i)$=$(s\_even(p_i) \wedge p_i.CC.S = p_i.S + 1) \vee$
  $(\neg s\_even(p_i) \wedge CURRENTC(p_i) \wedge$
  $p_i.P.S = p_i.S + 1)$

次に述語 Q1～Q3 を次のように定義する．(但し，$p_p = p_i.P$，$p_c = p_i.CC$ とする)

- Q1: $cons(p_i) \wedge T(p_i) \wedge ((s\_even(p_i) \wedge cons_c(p_c)) \vee$
  $(\neg s\_even(p_i) \wedge cons_p(p_p)))$
- Q2: $\neg cons(p_i) \wedge can\_Cons(p_i)$
- Q3: $\neg cons(p_i) \wedge \neg can\_Cons(p_i) \wedge \neg cons_i(p_i.P) \wedge$
  $(\#fa\_neigh(p_i) \neq 1 \vee \#ca\_neigh(p_i) \neq 1)$

これらの述語を使い，故障封じ込めを実現した提案プロトコルの概要は図 5 のようになる．

図 4.　において，$cons$ が成立しないプロセスは p_f，p_{c1} である．また can_Cons が成立するプロセスは p_f のみ．$S2$ を満たすのは p_f のみであり，p_f の隣接プロセスは $\neg S1$ かつ $\neg S2$ なので状態を変化させることはない．よって，それ以後の実行において，p_f で can_Cons が成立し RP を実行すれば，p_f と p_{c1} で $cons$ が成立し再安定する．

ここで Q1，Q3 で用いている $cons_p(p_p)$，$cons_c(p_c)$，$\#fa\_neigh(p_i)$，$\#ca\_neigh(p_i)$ を考える．これらは p_i とその隣接プロセスの状態からだけでは評価できない．そ

(※ $p_p = p_i.P,\ p_c = p_i.CC,\ \forall p_x \in N_i - \{p_p\},\ \forall p_y \in N_i - \{p_c\}$ $\forall p_k \in N_i$ とする．)

S10: $cons_i(p_i) \wedge T(p_i) \wedge \neg s\_even(p_i) \wedge q_{ip} = \bot \wedge a_{pi} = \bot$
$\rightarrow q_{ip} := ask$
S11: $cons_i(p_i) \wedge T(p_i) \wedge s\_even(p_i) \wedge q_{ic} = \bot \wedge a_{ci} = \bot$
$\rightarrow q_{ic} := ask$
S12: $\neg cons_i(p_i) \wedge \neg can\_Cons(p_i) \wedge q_{ik} = \bot \wedge a_{ki} \bot$
$\rightarrow q_{ik} := ask$
S13: $\exists j \in N_i : a_{ij} \neq f_i(q_{ji}, all(p_i))$
$\rightarrow a_{ij} := f_i(q_{ji}, all(p_i))$
S14: $((cons_i(p_i) \wedge \neg T(p_i)) \vee$
$(\neg cons_i(p_i) \wedge can\_Cons(p_i))) \wedge q_{ik} \neq \bot$
$\rightarrow q_{ik} := \bot$
S15: $cons_i(p_i) \wedge T(p_i) \wedge \neg s\_even(p_i) \wedge q_{ix} = ask$
$\rightarrow q_{ix} := \bot$
S16: $cons_i(p_i) \wedge T(p_i) \wedge \neg s\_even(p_i) \wedge q_{iy} = ask$
$\rightarrow q_{iy} := \bot$

図 6: 同期機構を実現するプロトコル

こで，[9] で述べられているのと同様の手法を用いて，これらの条件を評価するのに必要な情報を p_i の隣接プロセスから教えてもらう．以下では同期機構について述べる．

・ q_{ij} : $\bot$，ask のいずれかをとる．p_iから p_j への質問が発行中かどうかを表す変数

・ a_{ij} : p_j が，p_i の発行した質問に対する返答を表す変数．返答は q_{ji} および p_i と p_i の全隣接プロセスの状態から決定される．つまり，p_i 及び p_i の全隣接プロセスの状態を $all(p_i)$ とすると，a_{ij} は q_{ji} と $all(p_i)$ を引数とする二引き数関数 f_i で表され，$a_{ij} = f_i(q_{ji}, all(p_i))$ となる．また，質問が発行されてない場合は $\bot$ となる．

$$f_i = \begin{cases} 0 & \text{if } cons_i(p_i) \wedge \neg can\_Cons(p_i) \wedge q_{ji} = ask \\ 1 & \text{if } \neg cons_i(p_i) \wedge \neg can\_Cons(p_i) \wedge q_{ji} = ask \\ 2 & \text{if } \neg cons_i(p_i) \wedge can\_Cons(p_i) \wedge q_{ji} = ask \\ \bot & \text{otherwise.} \end{cases}$$

同期機構の動作は次の通りである．p_i が質問 q_{ij} に対する正しい返答 a_{ji} を得るには，a_{ji} が $\bot$ になるのを待って q_{ij} を ask にする．次に a_{ji} が $\bot$ 以外の値を持った場合，それは q_{ij} を ask にした，つまり質問を発行した時点以降の $all(p_j)$ から得られる値である．p_i が q_{ij} が ask に変えるのは，隣接プロセス p_j の $cons_j(p_j)$，$can\_Cons(p_j)$ の評価が必要なときである．つまり $cons_i(p_i) = FALSE$ または $cons_i(p_i) = TRUE$ かつ p_i がトークンを持つ場合である．p_i は，a_{ji} が $\bot$ 以外の値を持った場合，その値をもとに提案プロトコルに従って動作をする．この同期機構プロトコルを図 6 に示す．

各プロセス p_i では a_{ij} の値は常に評価されており (S11)，隣接プロセスが情報を必要としている場合，a_{ij} は $\bot$ 以外の値が代入される．つまり，q_{ji} が ask の間，常に a_{ij} は更新されている (同じ値に更新することもある) ことに注意する．

同期機構について，以下の補題が証明される．

補題 3. 任意の隣接プロセス p_i,p_j に対して，$q_{ij} = \bot$ または $a_{ji} = \bot$ を満たすならば，これ以降の任意の状況において，$q_{ij} = ask \wedge a_{ji} \neq \bot$ であれば，a_{ji} は，直前に p_i が q_{ij} を ask にした時点以降に計算された値である． □

また，隣接プロセス p_j の $cons_j(p_j)$，$can\_Cons(p_j)$ を評価する時点で同期機構の動作が終了している，つまり正しい値が返答されていることを保障するために述語 Q0:$\exists p_j \in N_i$ $[(q_{ij} = \bot \wedge a_{ji} \neq \bot) \vee (q_{ij} = ask \wedge a_{ji} = \bot)]$ を追加する．Q0 が真の場合，同期機構による質問，返答の途中である．従って，各プロセスが動作するのは $\neg Q0$ の時のみである．以上のことから提案プロトコルを図 7 に示す．

S1 : $\neg Q0 \wedge Q1$ →プロトコル TP を実行
S2 : $\neg Q0 \wedge Q2 \vee Q3$ →プロトコル RP を実行

図 7: 提案プロトコル

4 正当性

まず正当な状況について再定義する．

定義 12 (正当な状況)．全てのプロセス p_i で $cons_i(p_i) = TRUE$ を満たし，かつ次の条件を満たすシステム状況を正当な状況という．

○トークンを持つプロセス p_t :

- $p_t.S =$ 奇数 **ならば** $p_p (= p_t.P)$ **に対し，**
  $a_{tp} = \bot \vee a_{tp} = 0$
  各 $p_x \in N_t - \{p_p\}$ **に対し，**
  $q_{tx} = \bot \wedge (a_{tx} = \bot \vee a_{tx} = 0)$
- $p_t.S =$ 偶数 **ならば** $p_c (= p_t.CC)$ **に対し，**
  $a_{tc} = \bot \vee a_{tc} = 0$
  各 $p_y \in N_t - \{p_c\}$ **に対し，**
  $q_{ty} = \bot \wedge (a_{ty} = \bot \vee a_{ty} = 0)$

○ p_t 以外の全プロセス p_o :

- **各** $p_y \in N_o$ **に対し，**
  $q_{oy} = \bot \wedge (a_{oy} = \bot \vee a_{oy} = 0)$ □

また $cons_i(p_i) = TRUE$ であるとき，p_i の p_j に対する質問及び返答が正しいかどうかを示す述語 $ast_i(p_j)$ を定義する．

定義 13 ($ast_i(p_j)$)．p_i の隣接プロセスを p_j とする．$ast_i(p_j)$ 以下のように定義する．

- $ast_i(p_j) = cons_i(p_i) \wedge ast\_q_i(p_j) \wedge ast\_a_i(p_j)$
  $ast\_q_i(p_j) =$
  $q_{ij} = \bot \vee$
  $(q_{ij} = ask \wedge T(p_i) \wedge ((\neg s\_even(p_i) \wedge p_j = p_i.P) \vee$
  $(s\_even(p_i) \wedge p_j = p_i.CC)))$
  $ast\_a_i(p_j) =$
  $(a_{ij} = \bot \vee (a_{ij} = 0 \wedge cons_i(p_i))) \vee$

また，p_i が全ての隣接プロセス p_k に対して $ast_i(p_k)$ が成立するなら $ast_i(p_i) = TRUE$ と定義する． □

簡単のために p_i が $ast\_i(p_i) = TRUE$ を満たす時，「p_i は ast が成立する」と呼ぶ．正当な状況の定義より正当な状況では必ず全てのプロセス p_a で $ast_a(p_a)$=TRUE が成立する．

まず，提案プロトコルが自己安定であることを証明する．そのために任意の状況からプロトコルを開始しても，やがて正当な状況に到達することを証明する．なお，紙面の都合上，略証としている．

補題 4. $\neg cons_i(p_i)$ もしくは $cons_i(p_i) \wedge T(p_i)$ のいずれかをみたす場合いつか必ず $\neg Q0$ となり，$cons_i(p_i) \wedge \neg T(p_i)$ が成立するまで $\neg Q0$ である． □

補題 5. 任意の状況から開始しても，やがて全てのプロセス p_i で $cons_i(p_i)$ が成立する．

証明. 任意のプロセス p_i について，$p_i.P = p_p$ とする．プロトコルの動作より，$cons_p(p_p)$ が成立する p_i において，$cons_p(p_p.P)$ が成立する限り $cons_i(p_p)$ が成立し続けることがいえる．

一方，$cons_r(p_r.P) = TRUE$ である．このとき，プロトコルより任意の $p_i \in N_r$ においてある時点で $cons_i(p_r)$ が成立するとき，それ以降 $cons_i(p_r)$ は成立したままであることがいえる．同様のことが $p_j \in N_i$ にもいえる．

以上より，各プロセスにおいてプロトコルが繰り返し適用されることにより，根から葉にむかって順に無矛盾状態になっていくことが，帰納法によって証明される． □

補題 6. 全てのプロセス p_i において，$cons_i(p_i)$ が成立すれば，いずれ正当な状況となる．

証明. プロトコルより，$cons_i(p_i)$ を満たすプロセス p_i がプロトコルを実行しても $cons_i(p_i)$ が満たされることがいえる。また同期機構よりある時点以降，全てのプロセス p_i で $ast_i(p_i)$ が成立することが言える． □

補題 5 と補題 6 より，次のことがいえる．

補題 7. $FCTP$ は到達性を満たす． □

補題 8. $FCTP$ は閉包性を満たす．

証明. プロトコルより，$cons_i(p_i)$ を満たすプロセス p_i がプロトコルを実行しても $cons_i(p_i)$ 及び $ast_i(p_i)$ が満たされることがいえる。 □

補題 7 と補題 8 より次の定理が示せる．

定理 1. 提案プロトコルはトークン巡回自己安定プロトコルである． □

最後に，提案プロトコルの故障封じ込め性を証明する．まず一時故障が生じても，トークンの数が増加しない，つまりネットワーク中でトークンが唯一であり続ける場合を考える．プロセス p_f の一時故障によって p_f の補助変数 q，a のみが変化した場合や，q，a と共に $p_f.S$ 及び $p_f.CC$ の値が変化しても，$cons_f(p_f)$ が成立するならば，故障後も全てのプロセスで $cons$ が成立する．つまり補題 2 よりネットワーク中でトークンは唯一である．このように一時故障が生じてもトークンの数が増加しない場合，定数時間で再安定することを証明する．

補題 9. 正当な状況 c からプロセス p_f で一時故障が生じ，1 故障状況 c_f になったとする．c_f においてネットワーク中にトークンが唯一であれば，変動プロセス数は高々 $\triangle + 1$，再安定時間は $O(1)$ である．

証明. 補題 2 より c_f において全てのプロセスで $cons$ が成立する．また，プロトコルより $cons_i(p_i)$ を満たす任意のプロセス p_i は，$a_{ji}(p_j$ は p_i の任意の隣接プロセス) の値に関わらず，プロトコルを実行しても $cons_i(p_i)$ が満たされることがいえるので，c_f 以降の任意の状況において全てのプロセスで $cons$ が成立し，トークンは常に唯一である．

トークンを持つプロセス以外で同期機構の動作 (q，a の変更) を行うのは明らかに p_f とその隣接プロセスのみであり，高々 1 度ずつ同期動作を行うのみ．よって変動プロセス数は $\triangle + 1$，再安定時間は $O(1)$ である． □

次に，一時故障によってネットワーク中にトークンが二つ以上になる場合を考える．$cons_f(p_f) = FASLE$ となる一時故障がプロセス p_f で起きた場合，その故障によってトークンが生じ，ネットワーク中にトークンが二つ以上になる．トークンの定義より p_f の一時故障によってトークンが生じるのは，p_f かその隣接プロセスのみであるのは明らか．そこで故障トークンを次のように定義する．

定義 14 (故障トークン). 1 故障状況かつネットワーク中にトークンが二つ以上存在する状況において，故障プロセスもしくは，故障プロセスの隣接プロセスに存在するトークンを故障トークンと呼ぶ． □

また故障トークン以外のトークンを正トークンと呼ぶ．

以降，一時故障によって，故障トークンが生じた場合について，故障封じ込め性を証明する．以降，正当な状況 c において，p_f で一時故障が生じ，1 故障状況 c_f になったとし，c_f において故障トークンが存在する場合を考える．この時，c_f において $cons_f(p_f) = FALSE$ であることに注意する．また p_f の親プロセスを p_p，p_f の任意の子プロセスを p_{fc} とする．

まず同期機構について次の証明を行う．

補題 10. p_f の任意の隣接プロセスを p_j とする．c_f において $cons_f(p_f) = FLASE$ ならば，c_f 以降の任意の状況において，$q_{jf} = ask \wedge a_{fj} \neq \perp$ であれば，q_{if} は c_f 以降に $\perp$ から ask に変化しており，かつ a_{fj} は q_{if} が ask に変化した後で計算された値である．

証明. c_f において $q_{jf} = \perp$ ならば補題 3 より，$q_{jf} = ask \wedge a_{fj} \neq \perp$ の時点において，a_{fj} は c_f 以降に q_{if} が ask に変化した後で計算された値である．

よって，c_f において $q_{jf} = ask$ の場合を考える．c_f において $q_{jf} = ask$ となるのは，同期機構より c において p_j がトークンを持ち，かつ $p_j.S$=奇数ならば $p_f.S$=偶数$\wedge\ p_f.CC = p_j$ の場合，もしくは $p_j.S$=偶数ならば $p_f.S$ = 奇数 $\wedge\ p_j.CC = p_f$ の場合のみである．

このことと c において $cons_f(p_f)$ が成立することから，$p_j.S$ の値が偶数，奇数に関わらず，c において $p_p.S$=偶数$\wedge\ p_p.CC = p_f \wedge p_{fc}.S$=奇数を満たすといえる．このとき，$cons$ の定義より $p_f.S$，$p_f.CC$ の値に関わらず，必ず $cons_f(p_f)$ が成立するので，p_f で一時故障が生じても，$cons_f(p_f)$ が成立し続ける．つまり，c_f において $cons_f(p_f) = FALSE$ なら $q_{jf} = ask$ であるプロセス p_j は存在しない．よって補題は正しい． □

$cons$ の定義より次の補題は明らか．

補題 11. c_f において $cons$ が成立しないのは p_f とその隣接プロセスのうち一つのプロセスのみである． □

また補題 11 と can_Cons の定義より，次の補題は明らか．

補題 12. c_f において can_Cons が成立するのは p_f のみ，または p_f とその隣接プロセスのうち一つのプロセスのみである． □

ここで 1 故障状況 c_f について考える．c において，$cons_f(p_f)$ が成立することと，補題 11 より c_f において次のことがいえる．

観測 3. c_f において次のいずれかを満たす．
(1)$cons$ **が成立しないプロセスは p_f と p_p のみであり，p_f の任意の子プロセスの状態変数の値が奇数である．**
(2)$cons$ **が成立しないプロセスは p_f とその子プロセスのうち一つのプロセス p_k のみであり，「$p_p.S$=偶数$\wedge\ p_p.CC = p_f \wedge p_k$ を除く p_f の任意の子プロセスの状態変数の値が奇数」が成り立つ．** □

補題 13. c_f において動作可能 (状態変数 S 及び CC を変更可能) なプロセスは正トークンを持つプロセスと can_Cons が成立するプロセスのみ．

証明. 正トークンを持つプロセス p_t は $cons_t(p_t) \wedge T(p_t)$ が成立し，p_f に隣接しないことから，$S1$ が成立する．また，can_Cons が成立するプロセスは，プロトコルより必ず $Q2$ が成立し，$S2$ が成立する．

よって，次にこれら以外のプロセスが動作しないことを示す．p_f に隣接しないプロセスは，(正トークンを持つプロセスを除いて)$cons$ が成立し，かつトークンを持たないので $\neg S1$，$\neg S2$ であり，動作しないのは明らか．最後に p_f の任意の隣接プロセスは can_Cons が成立しない限り動作しないことを，$cons$ が成立するプロセスと，$cons$ が成立しないプロセスに分けて示す．

p_f の隣接プロセスの中で $cons_x(p_x)$ が成立する任意のプロセスを p_x とする．p_x は $cons_x(p_x) = TRUE$ より $\neg S2$ である．また観測 3 の (1)(2) より，p_f の隣接プロセスのうち $cons$ が成立するプロセスは状態変数 S の値が奇数ならば p_f の子プロセスであり，状態変数 S の値が偶数ならば，CC が p_f である．つまり，p_x は $p_x.S$=奇数ならば $p_x.P = p_f \wedge \neg cons_f(p_f)$ より $\neg S1$ であり，$p_x.S$=偶数ならば $p_x.CC = p_f \wedge \neg cons_f(p_f)$ より $\neg S1$ となる．よって，p_x は $\neg S1$，$\neg S2$ より動作しない．

p_f の隣接プロセスの中で $\neg cons_y(p_y) \wedge \neg can\_Cons(p_y)$ である任意のプロセスを p_y とする．p_y は $\neg cons_y(p_y)$ より $\neg S1$ であり，$\neg can\_Cons(p_y)$ より $\neg Q2$ である．また，補題 11 と補題 12 より，$\#fa\_neigh(p_y)$=$\#ca\_neigh(p_y) = 1$ が成立するので，$\neg Q3$ である．よって，p_y は $\neg S1$，$\neg S2$ より動作しない． □

補題 14．c_f 以降の任意の実行において，can_Cons が成立するプロセスが一度動作すれば，全てのプロセスで $cons$ が成立する．また，変動プロセス数は変動プロセス数 $2\triangle+2$，再安定時間 $O(1)$ となる．

証明. 補題 11 より 1 故障状況において $cons$ が成立しないプロセスは p_f とその隣接プロセスのうち一つのプロセス (p_h とする) のみであり，p_f は p_h に対してのみ矛盾し，p_h も p_f に対してのみ矛盾する ($cons_f(p_h)$=$cons_h(p_f)$=$FALSE$)．また補題 12 より can_Cons が成立するのは p_f のみ，もしくは p_f と p_h のみである．p_f がプロトコルを実行すれば，$can\_Cons_f(p_f)$ が成立するので，$Q2$ が成立し，プロトコルに従って状態を変化させることで，$cons_f(p_f) = TRUE$ となる．また，$cons_f(p_f) = TRUE$ ならば，$cons$ の定義より $cons_f(p_h) = cons_h(p_f) = TRUE$ となるので，p_h でも $cons_h(p_h)$ が成立する．つまり，全てのプロセスで $cons$ が成立する．同様に，$can\_Cons(p_h)$ が成立する時，p_h でプロトコルを実行すれば，p_h 及び p_f で $cons$ が成立し，全てのプロセスで $cons$ が成立する．

よって，全てのプロセスで $cons$ が成立するまでに，正トークンを持つプロセス以外で状態変数 S 及び CC ポインタの値を変更するのは，p_f または p_h のいずれか一方のみであり，同期動作を行うのは p_f と p_h，およびそれぞれの隣接プロセスである．また，全てのプロセスで $cons$ が成立すると，それ以後，正トークンを持つプロセス以外で同期動作は行われず，また同期動作は明らかに定数回の動作で停止する．よって，変動プロセス数 $2\triangle+2$，再安定時間 $O(1)$ である．となる． □

補題 9 と補題 14 より次の定義が言える．

定理 2．提案プロトコルは変動プロセス数 $2\triangle+2$，再安定時間 $O(1)$ であるトークン巡回障封じ込め自己安定プロトコルである．また，*1* 故障状況から再安定するまでの間，故障トークンは他のプロセスに移動しない． □

5 むすび

本稿では，1 故障状況からの実行において変動プロセス数 $2\triangle+2$，再安定時間 $O(1)$，故障状況から再安定するまでの間，故障によって生じたトークンは他のプロセスに移動しない，トークン巡回故障封じ込め自己安定プロトコルを提案した．

今後の課題としては D デーモンへの対応や，木ネットワーク以外のネットワークに拡張することが挙げられる．

謝辞 本研究の一部は，平成 16 年度日本学術振興会科学研究補助金（基盤研究 (C)16500028，若手 (B)16700010）の研究助成によるものである．

参考文献

[1] S. Dolev, E. Schiller, J. Welch, "Random walk for self-stabilizing group communication in ad-hoc networks." Proc. 21st Ann. ACM Symp. on Principles of Distributed Computing, (2002).

[2] 中村友貴, 片山喜章, 高橋直久,"ノード及びリンク故障を考慮したエージェント巡回自己安定プロトコルについて", 信学技法 (COMP2003-93), Vol.103, No.723, pp.57-64, (2003).

[3] 千星裕, 桝田秀夫, 辻野嘉宏, 都倉信樹,"リングネットワーク上での排他制御問題に対する故障封じ込め自己安定アルゴリズム"情報処理学会 研究会報告, 1997-PRO-018, Vol.1998, No.030, (1998)

[4] 角川裕次, 山下雅史, "アドホックネットワーク向けトークン巡回自己安定分散アルゴリズム", 情報処理学会 アルゴリズム研究会, 2003-AL-92, pp. 9-16, (2003).

[5] Gianluigi Alari, Joffroy Beauquier, Ajoy K. Datta, Colette Johnen, Visalakshi Thiagarajan,"A Fault Tolerant Token Passing Algorithm on Tree Networks",IEEE International Performance Computing, and Communications Conference , (1998).

[6] E.W.Dijkstra. "Self-stabilizing Systems in spite of distributed constrol".Communications of the ACM17(11),pp643-634,(1974)

[7] 片山喜章, 長谷川敏之, 高橋直久, "任意の単一リンク故障を考慮した生成木構成強安定プロトコル", 電子情報通信学会論文誌,VOL.J88-D1 No.11, (2005)

[8] N.Chen, H.Yu, and S.Huang, "A self-stabilizing algorithm for constructing spanning trees," Information Processing Letters vol.39 pp.147-151,(1991)

[9] S.Ghosh,A.Gupta,and S.V.Pemmaraju,"A fault-containing self-stabilizing algorithm for spanning trees,"J.Computing and Information,vol2,pp.322-338,(1996)

[10] S.Dolev,"Self-Stabilization",MIT Press,(2000)