通信効率のよい全域木構成自己安定アルゴリズムのトポロジ変化に対する出力安定性の実現

Output-stabilization and Communication-efficient for Self-stabilizing Protocols

北口峻行 *　角川裕次 †　増澤利光 ‡
Toshiyuki Kitaguchi　Hirotsugu Kakugawa　Toshimitsu Masuzawa

1　はじめに

複数の計算機がネットワークを介しながら，協動するシステムが**分散システム**である．近年，分散システムの大規模化が著しいが，大規模分散システムでは一部の計算機の故障やネットワークの形状変化は避けがたく，これらが生じてもシステム全体としてのサービスを継続できる，あるいは，外部から干渉なしに自律的にサービスを復旧できる**適応性**がより重要になってきている [?].

計算機の一時故障やネットワークの形状変化に対する高度な適応性を有する分散アルゴリズムとして，**自己安定アルゴリズム** [?, ?] が注目されている．自己安定アルゴリズムとは，任意の状況から実行を開始しても，いずれ所望の状況（**正当な状況**とよぶ）に到達することを保証する．このため，分散システムに一時故障や形状変化が生じても，外部からの干渉なしに自律的にサービスを復旧できる．

自己安定アルゴリズムはその高度な適応性を実現するため，システムが正当な状況であるかどうかを常に監視する必要がある．このため正当な状況においても，システム状況の監視のために一定のメッセージ交換と処理が必要であり，分散システムの運用コスト増大の一因となっている．[?] はこの問題点に着目し，自己安定アルゴリズムの正当な状況での通信量を削減を目的とした**通信効率**という概念を導入した．これまでに，いくつかの問題について，通信効率のよい自己安定アルゴリズムが提案されている [?, ?, ?]．特に，[?] は，本稿で扱う全域木構成問題に対して，通信効率のよい自己安定アルゴリズムを提案している．このアルゴリズムでは，全域木が構成された正当な状況では，ネットワーク状況の監視のために，各プロセスは全域木の親プロセスとのみ通信を行っている．また，この自己安定アルゴリズムが通信効率に関して最適であることも示している．

自己安定アルゴリズムは，分散システムに大規模な一時故障や形状変化が生じても，正当な状況に自律的に復帰できることを保証している．しかし，大規模な一時故障や形状変化が生じることはごく稀であり，頻繁に生じるのは小規模な一時故障や形状変化と考えられる．そのため，小規模な一時故障や形状変化に対するすぐれた特性を実現することが重要である．例えば，小規模な一時故障や形状変化に対する，自己安定アルゴリズムの効率的な適応性の実現を目的として，**故障封じ込め**という概念が導入され [?, ?]，故障封じ込め自己安定アルゴリズムについて多くの研究が行われている [?, ?, ?, ?, ?]．本稿では，ネットワーク形状の小規模な変化として，複数のリンク消滅を取り上げ，全域木リンク消滅からの復旧において，リンク消滅の影響を軽減することを目指す．具体的には，全域木リンク消滅からの復旧時の**出力安定**を実現することを目指す．

出力安定は，自己安定アルゴリズムの任意の状況からの復旧過程における外部への影響を抑制するために，外部からアクセスできる出力変数に着目し，その変動を抑制することを目的として導入された概念である [?]．自己安定アルゴリズムは任意の状況からの正当な状況への自律的な復旧を保証するが，この復旧過程においては何も保証しない．そのため，復旧過程においてシステム状況が頻繁に変化し，この自己安定アルゴリズムを利用する他の分散システムや外界に大きな影響を与える可能性がある．そのため，自己安定アルゴリズムの出力安定を実現することとは，より安定な分散システムを実現するために重要であると考えられている．

本稿では，全域木構成問題を解く，通信効率のよい自己安定アルゴリズムに対し，リンク消滅に対する出力安定を実現する手法を提案する．提案する自己安定アルゴリズムは，増澤らのアルゴリズム [?] に加え，全域木が構成された正当な状況時には，各プ

* 大阪大学 Osaka University of Information Science and Technology
† 大阪大学 The same as the first author
‡ 大阪大学 The same as the first author

ロセスは全域木の親プロセスと通信を行うことに加え、リンクの消滅時にのみ親にリンクの消滅を報告できるものとしている。また，正当な状況で全域木のひとつのリンクが消滅したときには，各プロセスは親プロセスを高々 1 回変更するだけで全域木を再構築することで出力安定を実現している．このときの正当な状況への復帰に要する時間は，増澤らのアルゴリズム [?] と同様に $O(n)$ である．ここで，n は分散システムのプロセス数を表す．

本稿の構成は以下のとおりである．まず，**??**節において，分散システムのモデルと自己安定アルゴリズムを定義する．**??**節において提案する自己アルゴリズムを示し，**??**節において正当性の証明と性能評価を示す．最後に**??**章において本研究をまとめる．

2 諸定義

2.1 システムモデル

分散システム $S = (P, L)$ は，プロセスの集合 P = $\{v_0, v_1, \ldots, v_{n-1}\}$ と双方向通信リンクの集合 L から構成される．各プロセスは，固有の ID を持つと仮定する．プロセス間にリンクが存在する場合，隣接しているといい，プロセス v と隣接するプロセス集合を N_v で表す．

各プロセス v は，1 原子動作として，(1) すべての隣接プロセスの状態を読み出し，(2) それらの状態と v の状態から v の次状態を決定し，(3)v の状態をその次状態に変更する．分散システムの状況は，n 項組 $\sigma = (s_0, s_1, \ldots, s_{n-1})$ で表される．ここで，s_i はプロセス v_i の状態を表す．

プロセス集合 Q に対し，プロセス集合 Q の動作により，状況が σ から σ' に遷移するとき，$\sigma \xrightarrow{Q} \sigma'$ と表す．

スケジュールは，空ではないプロセス集合の無限列 $\zeta = Q_1, Q_2, \ldots$ である．本稿では，スケジュールは弱公平と仮定する．つまり，すべてのプロセスが無限回現れるスケジュールのみを考える．スケジュールはプロセスが動作する順序を指名するものである．本稿では弱公平なスケジュールを考えるが，これはネットワークが非同期式であるが，すべてのプロセスが無限に動作し続ける（状態が変化し続けるとは限らない）ことを意味している．状況 σ_0 とスケジュール $\zeta = Q_1, Q_2, \ldots$ に対し，各 $i (i \geq 1)$ について，$\sigma_{i-1} \xrightarrow{Q_i} \sigma_i$ を満たす状況の無限列 $E = \sigma_0, \sigma_1, \sigma_2, \ldots$ を状況 σ_0 から始まるスケジューラらによる実行とよぶ．また，状況 σ_0 を実行 E の初期状況とよぶ．

時間複雑度を評価するために，実行 E に対し，非同期式ラウンドを導入する．スケジュール $\zeta = Q_1, Q_2, \ldots$ による実行 $E = \sigma_0, \sigma_1, \sigma_2, \ldots$ の最初のラウンドは，$P = \cup_{1 \leq j \leq k} Q_j$ を満たす，最小の接頭部実行 $\sigma_0, \sigma_1, \ldots, \sigma_k$ と定義する．実行 E の二番目以降のラウンドは，実行 $\sigma_k, \sigma_{k+1}, \sigma_{k+2}, \ldots$ に対して再帰的に定義する．

2.2 自己安定アルゴリズム

自己安定アルゴリズムは，任意の状況から実行を開始しても，いずれ所望の状況（正当な状況）に到達して安定する分散アルゴリズムである．本稿で扱う全域木構成問題の正当な状況は，次のように定義する．分散システムの 1 つのプロセスが根として指定される．各プロセス v は隣接プロセスの一つを親として保持するための出力変数 $prnt_v$ を持つ．ただし，根プロセスは親を持たないので $prnt_r = \perp$ とする．また，各プロセスは収束するまでの隣接の状況に応じて，全部で 5 状態の内いずれかの状態を保持している．全プロセスの $prnt_v$ が分散システムのある全域木の親を保持し，かつ全プロセスが状態 $legal$ を保持していれば，その状況は正当な状況である．

2.3 通信効率

各プロセス $v \in Q_i$ のスケジューラ $\zeta = Q_1, Q_2, \ldots$ による実行 $E = \sigma_0, \sigma_1, \sigma_2, \ldots$ に対し，σ_{i-1} においてプロセス v の状態遷移に影響を与える隣接プロセスの集合を $R_v^i(E)$ とし，$R_v(E) = R_v^1(E) \cup R_v^2(E) \cup \ldots$ とする．つまり、v は $R_v^i(E)$ に属さないプロセスの状態を読み取る必要はない。また、$v \notin Q_i$ のとき、$R_v^i(E) = \emptyset$ とする．

Definition 2.1 (◇-k-通信効率化) すべての実行 E に対し，$\sum_{i=0}^{n-1} |R_{v_i}(E')| \leq k$ となる E の接尾部分実行 E' が存在するとき，プロトコルが ◇-k-通信効率であるといえる．

3 提案アルゴリズム

本節では，提案アルゴリズムを示す．

3.1 プロセスの変数

各プロセスは，以下の変数を持つ．

root:
根プロセスでのみ $true$ となる論理型変数.
(各プロセスは自身が根プロセスかどうかを認識しており，値は常に正しい)

prnt:
親プロセスの ID. 根プロセスあるいは，親プロセスが未定のプロセスは値 $\perp$ を持つ.

prnt_set:
先祖プロセスの ID の集合.

state:
状態 $legal$，状態 $floating$，状態 $correctT$，状態 pre_legal，状態 $any_floating$ を持ち、正当な状況では状態 $legal$ を持つ.

consistent:
自身の状態が無矛盾なとき $true$ となる論理型変数.

論理型変数 $root$ は根プロセスは常に $true$，それ以外は常に $false$ を持つ．根以外の各プロセスは隣接プロセスの中から一つを親プロセスとして選び，その ID を変数 $prnt_set$ に保持する．根プロセスは親プロセスを選ばないので，変数 $prnt$ に値 $\perp$ を保持する．各プロセスは，構成している木において，根からそのプロセスへの経路に現れるプロセスの ID の集合 (ただし，自身は除く) を変数 $prnt$ に保持する．全域木が構成されているときに，全域木のリンクが消滅すると，消滅リンクから全域木に沿って全てのプロセスが状態変化を行い $state$ を状態 $legal, , correctT$ までで管理する，また各プロセスは，自身の変数が無矛盾なら論理型変数 $consistent$ を $true$ にする.

3.2 正当な状況

提案アルゴリズムの正当な状況を定義する.

Definition 3.1 (正当な状況) 各プロセスが以下の条件を満たす状況は正当な状況である.

1. 根プロセス r

- $root_r = true$
- $prnt_r = \perp$
- $prnt_set_r = \{\phi\}$
- $state_r = 1$
- $consistent_r := true$

1. 根以外のプロセス v

- $root_v = false$
- $prnt_v \in N_v$
- $prnt_set_v = prnt_set_{prnt_v} \cup \{prnt_v\}$
  $\wedge v \notin prnt_set_v$
- $state_v = 1$
- $consistent_v := true$

正当な状況では，全域木が構築され，各プロセスの変数 $prnt$ は，構成された全域木における親プロセスの ID が保持されている．定義 4 の正当な状況では，全域木が構成されていることが，親を順にたどっても閉路を構築しないことから証明できる.

3.3 通信効率のよい自己安定全域木構成のアイデア

提案アルゴリズムは，正当な状況に収束できる．任意の状況において，根プロセスに正しく接続された木を $correct\,tree$ とよぶ．$correct\,tree$ に含まれないプロセスが閉路に属していても，変数 $prnt_set$ から，矛盾を検出できるため，$prnt$ を $\perp$ にする．その後も $correct\,tree$ に含まれないプロセス同士で，繰り返し親を更新することが発生しても，最終的に $correct\,tree$ に接続される.

正当な状況に復帰した後は，根プロセス以外の各プロセスは，親プロセスの変数 $prnt_set$ を確認することで，構築した全域木を保持できる．よって通信効率もよいとゆえる.

3.4 プロセスの状態説明

提案アルゴリズムでは，任意の状況から正当な状況まで収束する際，また全域木のリンクが消滅した際に、全プロセスがそれぞれの状況に応じ状態変化を行う．それぞれの状態変化について，以下で説明する。

1. 正当な状況
   - 状態 $legal$:
     各プロセスが正当な状況下に置かれた状態
2. 全域木のリンク消滅時
   - 状態 $floating$:
     消滅リンクの子孫で再接続が必要なプロセスの状態，状態 $correctT$ のプロセスに対して再接続を行う.

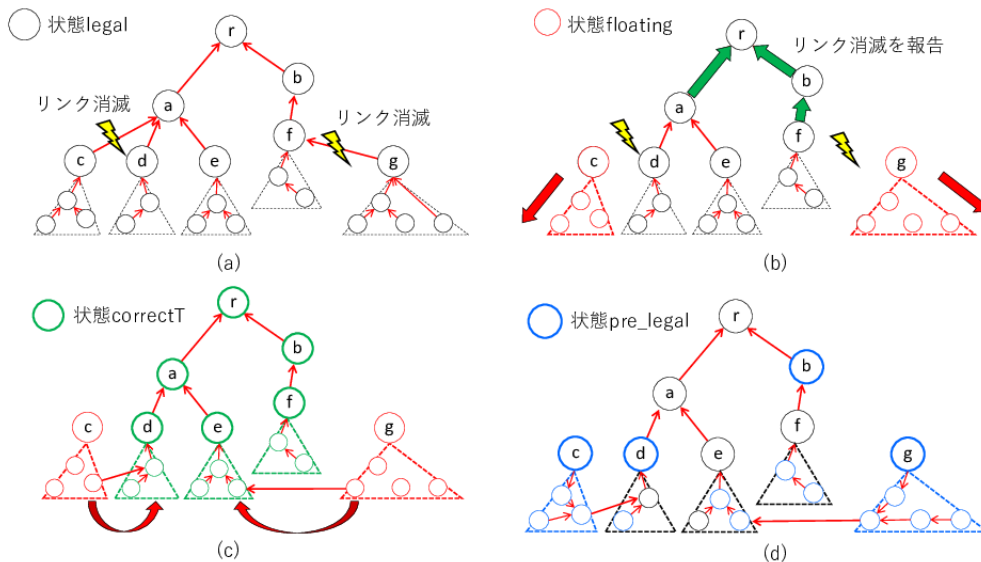

Figure 1: リンク消滅時

- 状態 *correctT* :
  消滅リンクの発生を検知した、*correct tree* に属するノードの状態．隣接に状態 *floating* がいないことが確認されると状態 *correctT* に移行する．
- 状態 *pre_legal* :
  リンク消滅後、状態 *legal* に戻るための準備．隣接に状態 *correctT* がいないことが確認されると状態 *legal* に移行する．

3. 任意の状況からの開始時

- 状態 *any_floating* :
  correct_tree に属していない状態．状態変化を行い再接続を行う．

3.5 リンク消滅に対する処理

全域木が構築された正当な状況で，全域木のリンクの消滅が起こった状況を考える (Figure **??**(a))．消滅リンクの子孫プロセスは根 r を含む *correct tree* から分断されるため，全域木を再構築するために親の変更が必要になプロセスが存在する．このとき，親を適切に選ばないと，繰り返し親を変更しなければならない場合がある．そこで本アルゴリズムでは，そのような無駄な親の変更が発生しないように，*correct tree* に属することが保証された隣接プロセスのみを親として選択する．

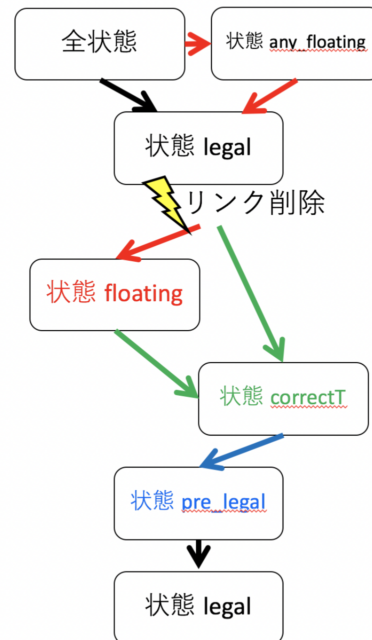

Figure 2: 状態変化

これは以下のように実現する．リンク (a,c)，(f,g) の消滅を子側のプロセス c, g が検知すると，変数 $state_c, state_g$ に *floating* を格納し，状態変化させ

る．この状態を、c, g の子孫が確認することで状態 $floating$ を伝搬させる．また、リンクの消滅を親側の a, f が検知すると、親に報告を送り、順々に根までリンク消滅を伝搬させる (Figure **??**(b)).
根にリンク消滅が伝搬すると，変数 $state_r$ に $correctT$ を格納し，状態変化させる．根の子が状態変化を確認することで状態変化を起こし、伝搬させる．このようにすることで再接続が必要な部分と，根に正しく接続されている部分を状態分けし，判定できるようにしている (Figure **??**(c)). c および g の子孫は，$state$ が $correctT$ の隣接プロセスを親として選択することで，無駄な親の変更を回避する．親の候補が，複数存在する場合，根により近いプロセスを親とする．根への距離が等しいプロセスが存在する場合は ID の小さいプロセスを親とする．隣接プロセスに親の候補となるプロセスが存在しない場合は隣接プロセスが状態 $correctT$ に移行するまで，親を $\perp$ に設定し待機する．
状態 $floating$ の全プロセスが，状態 $correctT$ に状態変化すると，隣接に状態 $floating$ がいなくなったプロセスから順に，状態 pre_legal に状態変化を行う．状態 pre_legal を導入することで，リンク消滅前で状態変化がまだ始まっていない状態 $legal$ と状態変化を経た状態 $legal$ かどうかを判定することができるようにしている．また状態 pre_legal も，状態 $correctT$ のプロセスがいなくなったプロセスから状態 $legal$ に状態変化する．

3.6 擬似コード

提案アルゴリズムの疑似コードを Algorithm1(各プロセスが持つ ID，変数の説明，根プロセスの動作), Algorithm2,3(根以外のプロセスの動作) に示す．

4 アルゴリズムの正当性と性能

本節では，アルゴリズムの正当性を証明し，次にいくつかの評価尺度に対して，既存研究と比較する．

4.1 アルゴリズムの正当性

本節ではアルゴリズムの正当性を証明する．アルゴリズムから，次の補題は明らかに成立する．

Lemma 4.1 (閉包性) 正当な状況に到達すると，提案アルゴリズムはその状況から変化しない．

正当な状況にいずれ到達すること（収束性）を示すのに，状況 σ における $correct\,tree\,T_\sigma$ を導入する．

Algorithm 1 変数と根プロセス r の原始動作

```
1:  /*プロセス v の変数（常に正しい値を保持）*/
2:  ID_v : identifier ;
3:  root_v : bool ;
4:  state_v : legal  or floating or correctT or pre_legal
    or any_floating ;
5:  prnt_v : {⊥} ∪ identifier ;
6:  prnt_set_v : ID の集合 ;
7:  consistent_v : bool ;
8:
9:  /* 根プロセス v(= r) の動作*/
10: prnt_r = ⊥ ;
11: ID_v = φ ;
12: consistent_v : true ;
13: if (bottomup from child) then
14:     state_r = correctT ;
15: if (∀i ∈ N_r | state_i = correctT or pre_legal)
    then
16:     state_r = pre_legal ;
17: if (∀i ∈ N_r | state_i = pre_legal or legal) then
18:     state_r = legal ;
```

Algorithm 2 根以外のプロセスの動作

```
1:  /*根以外のプロセス v の動作*/
2:  switch (consistent_v)
3:  case true:
4:      if (prnt_v ∉ N_v) ∨ (v ∈ prnt_set_{prnt_v}) ∨
    (prnt_set_v ≠ prnt_set_{prnt_v} ∪ prnt_v) then
5:          consistent_v =false ;
6:      if (bottomup from child) then
7:          send bottomup to prnt_v ;
8:      if (state_{prnt_v} = correctT) then
9:          state_v = correctT ;
10:     if (∀i ∈ N_v | state_i = correctT or pre_legal)
    then
11:         state_v = pre_legal ;
12:     if (∀i ∈ N_v | state_i = pre_legal or legal)
    then
13:         state_v = legal ;
14: end case
```

Definition 4.1 ($correct\,tree$) 提案アルゴリズムでの任意の状況を σ としたとき，σ において $correct\,tree\,T_\sigma$ を次のように定義する．

1. T_σ は根プロセス r を含み，$prnt_r = \perp$, $prnt_set_r = \{\phi\}$, $consistent_r := true$, $state =$

Algorithm 3 根以外のプロセスの動作

```
15: case false:
16:     if (prnt_v ∉ N_v ∪ {⊥}) then
17:         prnt_v = ⊥ ; prnt_set_v = φ ; state_v = floating ;
18:     else if (prnt_v ∈ N_v) then
19:         if (v ∈ prnt_set_v) then
20:             prnt_v = ⊥;prnt_set_v = φ;state_v = any_floating;
21:         else
22:             if (prnt_set_{prnt_v} ≠ φ) ∨ (root_{prnt_v} = true) then
23:                 prnt_set_v ≠ prnt_set_{prnt_v} ∪ prnt_v; state_v = state_{prnt_v}; consistent_v := true;
24:             else
25:                 if state_{prnt_v} = floating then
26:                     prnt_v = ⊥; prnt_set_v = φ; state_v = floating;
27:                 else
28:                     prnt_v = ⊥; prnt_set_v = φ; state_v = any_floating;
29:     else
30:         prnt_set_v = φ; state_v = any_floating;
31:     if (state = floating) then
32:         N_cand = {∃u ∈ N_v | state_v = correctT};
33:         if (N_cand ≠ φ) then
34:             u = arg min_{u∈N_cand}(| prnt_set_u | ∪ ID_u);
35:             prnt_v = ID_u; prnt_set_v = prnt_set_u ∪ {ID_u}; consistent_v := true; state_v = correctT;
36:     if (state = any_floating) then
37:         N_cand = {∃u ∈ N_v | state_v = correctT};
38:         if (N_cand ≠ φ) then
39:             u = arg min_{u∈N_cand}(| prnt_set_u | ∪ ID_u);
40:             prnt_v = ID_u; prnt_set_v = prnt_set_u ∪ {ID_u}; consistent_v := true; state_v = state_{prnt_u};
41: end case
42: end switch
```

$legal$ を満たす．r の変数がこれらの条件を満たさないときは，T_σ を空グラフとする．

2. T_σ に含まれる根プロセス以外のプロセス v は，T_σ に含まれるプロセスの中から変数 $prnt$ を選び，$prnt_set_v = prnt_set_{prnt_v} \cup \{prnt_v\}$, $state = regal$
$prnt \in N_v$, $v \notin prnt_set_v, consistent_v := true$ を満たす．

Lemma 4.2 (*correct tree* の安定性) 任意の状況 σ において，プロセス v が，*correct tree* T_σ に含まれるならば，σ 以降 v は状態を変更せず，σ 以降の任意の状況 σ' で v は T_σ に含まれる．

Proof 1 任意のプロセス v の論理変数 $consistent_v$ は，親に隣接しない場合，親の変数 $prnt_set_{prnt_v}$ の値に矛盾がある場合に，*true* から *false* になる．*correct tree* に含まれるプロセスは，根から変数 $prnt$ と変数 $prnt_set$ に無矛盾な値を格納し，*correct tree* を構成する．また構成中はリンクの消滅は行われないため $state$ も 1 のままである．よって *correct tree* に含まれるプロセス v の $consistent_v$ が *false* になることはない．

Lemma 4.3 (収束性) 任意の状況から実行を開始しても，提案アルゴリズムはいずれ正当な状況に収束する．

Proof 2 *correct tree* がいずれ全域木になることを証明すればいい．
背理法を使用し，*correct tree* が決してネットワークの全プロセスを含むことがないと仮定する．これは補題 2 より，ある状況以降，全域木でない *correct tree* が変化しないことを意味する．この *correct tree* を T とし，T に含まれるプロセスの集合を P_T，T に含まれないプロセスの集合を $P_{\overline{T}}$ とする．

Table 1: 解析結果

対象アルゴリズム / 評価尺度	既存研究 [?]	提案アルゴリズム
収束ラウンド数	$O(n)$	$O(n)$
リンク消滅からの復帰ラウンド数	$O(n)$	$O(n)$
リンク消滅からの親の変更回数	$O(n^2)$	$O(n)$

これ以上変化のない *correct tree* T が構築された状況を σ とする．σ の後の任意の次状況 σ' を考え，σ' での $P_{\overline{T}}$ に含まれる全プロセスのうち空集合でない $prnt_set$ の要素数 $|prnt_set|$ が最も少ないものを $|prnt_set|^{min}_{\sigma'}$ とする．

各プロセス $v \in P_{\overline{T}}$ は，根プロセスではないため，$prnt_v$ に隣接プロセスを選択するか，$\perp$ を設定する．また，$prnt_set_v = \phi$ が成立する．$P_{\overline{T}}$ のプロセスは，*correct tree* に含まれないので，$prnt_v$ に隣接プロセス u を設定するときは，$u \in P_{\overline{T}}$ が成り立つ．このとき，$prnt_set_v$ も更新し，$|prnt_set_v| = |prnt_set_u| + 1$ が成立する．このことから，$prnt_set$ が空でないプロセスが存在する限り，各ラウンドで $|prnt_set|^{min}_{\sigma}$ は少なくとも 1 増加する．従って，各プロセス $v \in P_{\overline{T}}$ で $prnt_set = \phi$ とならなければ，n ラウンド以内に $v \in prnt_set_v$ となり，$prnt_set_v = \phi$ となる．根以外のプロセスは，$prnt$ が空なら親として選択されないので，n ラウンド以内にすべての $v \in P_{\overline{T}}$ で $prnt_set_v = \phi$ となる．このとき，P_T に隣接プロセスを持つ $P_{\overline{T}}$ のプロセス v が存在し，v は P_T のプロセスを親として選択し，*correct tree* に含まれる．これは，*correct tree* が T から変化しないことに矛盾する．

Theorem 4.1 提案アルゴリズムは，全域木構成問題に対する，$\diamond$-$(n-1)$ 通信効率でかつ，$\diamond$-*1* 安定の自己安定アルゴリズムである．

Proof 3 補題 *1* および *3* までより，提案アルゴリズムは，自己安定アルゴリズムである．また，根プロセスは *read action* を実行せず，正当な状況では根以外のプロセスは，親のみと通信を行う．従って，$\diamond$-$(n-1)$ 通信効率でかつ，$\diamond$-*1* 安定である．

4.2 アルゴリズムの性能評価

以下の評価尺度を用いて，提案アルゴリズムの性能評価を行う．

- 収束ラウンド数
  任意の初期状況から実行を開始したときに，正当な状況に到達するまでに要する最悪時のラウンド数
- リンク消滅からの復帰ラウンド数
  正当な状況で，全域木のリンクが消滅したときに，再び正当な状況に到達するまでに要する最悪時のラウンド数
- リンク消滅からの親の変更回数
  正当な状況で，全域木のリンクが消滅したときに，再び正当な状況に到達するまでに変数 $prnt$ の変更の総数．

[?] のアルゴリズムと提案アルゴリズムそれぞれの各評価尺度での評価結果を，表 1 にまとめる．

4.2.1 収束ラウンド数

任意の状況から正当な状況に復帰するまでの収束ラウンド数を解析する．
状況 σ で，*correct tree* に含まれないプロセスの中で一番小さい先祖集合の要素数を $|prnt_set|^{min}_{\sigma}$ とする．*correct tree* に含まれないプロセスは，根プロセスと違い，必ず親を選択を選択する必要があるため，閉路を構成する．そのため，$|prnt_set|^{min}_{\sigma}$ の値はインクリメントされ続け，開始から $2n$ ラウンド以内で $|prnt_set|^{min}_{\sigma}$ が *correct tree* に含まれない全プロセス数を超えるため，各プロセスが *correct tree* に接続され始める．そこから構成に n ラウンドかかるので，収束するまでに $O(n)$ ラウンドで正当な状況までかかる．

4.2.2 リンク消滅からの復帰ラウンド数と親の変更回数

正当な状況において，全域木のリンクの消滅が起こった際に，再び正当な状況に復帰するのに必要なラウンド数と親の変更回数について解析を行う．

- 提案アルゴリズム
  全域木のリンク消滅時，そのリンクの子孫は状態遷移を行い，変数 $state = floating$ にする．また，リンク消滅を根プロセスに伝搬させ，$correct_tree$ に属するプロセスの状態を $state = correctT$ にする．状態 $floating$ のプロセスは，隣接の状態 $correctT$ に対して接続し，なければ親を変更することなく隣接プロセスが $correct\,tree$ に含まれるのを待つ．

  部分木の全プロセスに状態遷移が伝搬するのに最大 $n-1$ ラウンドかかる．1 ラウンドで少なくとも 1 つのプロセスが新たに $correct\,tree$ に含まれる．従って，リンク消滅からの復帰ラウンド数は $O(n)$ である．次に親の変更回数について考える．リンク消滅のため $correct\,tree$ に含まれないプロセスは，$prnt$ を一度 $\perp$ に変更することはあるが，隣接プロセスに設定するときは，必ず $correct\,tree$ に含まれる．補題 2 より，$correct\,tree$ のプロセスは親を変更することはないため，リンク消滅からの親の変更回数は $O(n)$ である．

  Theorem 4.2 提案アルゴリズムのリンクからの復帰ラウンド数，親の変更回数はともに $O(n)$ である．

5 まとめ

本稿では，全域木構成が行われた $\diamond$-$(n-1)$ 通信効率化の通信効率のよい自己安定アルゴリズムを紹介し，そのアルゴリズムが所望される状況の際，全域木のリンクが消滅した場合に無駄な親の交換が起こらないような新しい改良を加えたアルゴリズムを提案した．また提案したアルゴリズムが所望する状況まで正しく収束する正当性と，親の交換回数の削減を証明した．

References

[1] S. Devismes, T. Masuzawa, and S. Tixeuil: "Communication efficiency in self-stabilizing silent protocols," Proceedings of the 29th IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 474–481, 2009.

[2] S. Dolev: "Self-stabilization," MIT Press, 2000.

[3] S. Ghosh, A. Gupta, T. Herman, and S. V. Pemmaraju; "Fault-containing self-stabilizing algorithms," Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 45–54, 1996.

[4] S. Ghosh, A. Gupta, T. Herman, and S. V. Pemmaraju; "Fault-containing self-stabilizing distributed protocols," Distributed Computing, Vol. 20, Issue 1, pp 53–73, 2007.

[5] S. Ghosh and x. He, "Fault-containing self-stabilization using priority scheduling,"

[6] S. Kutten and T. Masuzawa: "Output stability versus time till output," Proceedings of the 21st International Symposium on Distributed Computing (DISC), pp.343–357, 2007.

[7] S. Kutten and B. Patt-Shamir: "Stabilizing time adaptive protocols,"

[8] S. Kutten and D. Zinenko: "Low communication self-stabilization through randomization," Proceedings of the 24th International Symposium on Distributed Computing (DISC), pp.465–479 (2010).

[9] T. Masuzawa, T. Izumi, Y. Katayama and K. Wada.: "Brief announcement: Communication efficient self-stabilizing protocols for spanning tree construction," Proceedings of the 13th International COnference on Principles of Distributed Systems (OPODIS), pp. 219–224, 2009.

[10] Y. Yamauchi, S. Kamei, F. Ooshita, Y. Katayama, H. Kakugawa, and T. Masuzawa: "Timer-based composition of fault-containing self-stabilizing protocols," Information Sciences, Vol. 180, No. 10, pp. 1802–1816, 2010.

[11] Y. Yamauchi, S. Kamei, F. Ooshita, Y. Katayama, H. Kakugawa, and T. Masuzawa: "Hierarchical composition of self-stabilizing protocols preserving the fault-containment property," IEICE Transactions, Vol. 92-D, No. 3, pp.451–459, 2009.

[12] Y. Yamauchi, T. Masuzawa, and D. Bein: "Preserving the fault-containment of ring protocols executed on trees," The Compututer Journal, Vol. 52, No. 4, pp.483–498, 2009.

[13] 増澤利光, 山下雅史: T. Masuzawa, T. Izumi, Y. Katayama and K. Wada.: "適応的分散アルゴリズム," 共立出版, 2010.