アルゴリズム 41−11
（1994. 9. 21）

メッセージ通信モデルでの均一な自己安定アルゴリズムへの拡張について

青柳滋己　真鍋義文
NTT 基礎研究所

通常の分散アルゴリズムは、アルゴリズム開始時にネットワークやプロセッサが特定の状態(初期状態)にあると仮定することが多いが、自己安定アルゴリズムは初期状態に何も仮定せず、有限時間内に問題を解く事ができる分散アルゴリズムである。Katz らはプロセッサが識別子を持ち、準均一な任意のメッセージ通信ネットワーク上で、任意の非自己安定プログラムを自己安定化する方法を考案した。本稿ではその方法を拡張し、プロセッサに識別子のない均一な任意のメッセージ通信ネットワーク上で動作する自己安定化プログラムを示す。

Uniform Self-Stabilizing Extension Algorithm under Message Passing Model

Shigemi Aoyagi　Yoshifumi Manabe
NTT Basic Research Laboratories

Many distributed algorithms assume an initial state (processor state and network state) and must start from the initial state, while self-stabilizing algorithms assume no initial state. Katz,et al. proposed an algorithm for automatically creating a self-stabilizing extension of any distributed algorithm under the model which there exists one special processor. In this paper, we present a new enhanced version of their self-stabilizing extension algorithm which works under the uniform message passing model. Uniform means all processors have no identifier in the system.

1 はじめに

通常の分散アルゴリズム [13] は、アルゴリズム開始時にシステムが特定の初期状態にあると仮定することが多い。例えば、各プロセッサがある定められた初期状態にあって、ネットワークにはリンクを伝播中のメッセージはないと仮定する。これに対し、自己安定アルゴリズム (self-stabilizing algorithm)[3] [4][5][7][8] [11][12] では、初期状態に何も仮定をおかない。つまり、各プロセッサの状態、リンクを伝播中のメッセージの有無やその内容にかかわらず有限時間内に問題を解くことが可能である。したがって、以下のような特徴を持つ。

- 各プロセッサのプログラムさえ無事ならば、プロセッサのデータの一部の破壊、リンクを伝播しているメッセージの紛失や内容の変質などが生じても、その後十分長い間故障が生じなければ問題を解くことができる。
- アルゴリズムの実行中にネットワークが動的に変化しても、十分に長い間変化がなければ問題を解くことができる。プロセッサやリンクの故障や復旧が生じても、十分に長い間故障や復旧が生じなければ、無故障な部分で問題を解くことができる。

最初は、相互排除や生成木構成問題といった個別の分散問題に対して自己安定アルゴリズムが考えられてきた。しかし、最近では任意の非自己安定プログラムを自己安定化する方法も考案され始めた。

Awerbuch ら [1] は、静的なネットワーク上で動く同期通信のアルゴリズムを、動的な非同期ネットワーク上で自己安定化する方法について述べている。また、Katz ら [9] は各プロセッサが識別子を持ち、システム内に 1 つ特別なプロセッサを持つ準均一なメッセージ通信モデルにおいて、任意の非自己安定アルゴリズムを自己安定拡張するための方法が述べられている。

本稿ではこの Katz らのアルゴリズムを拡張し、識別子を持たず、システム内に特別なプロセッサがない均一なモデルにおいて、任意の非自己安定アルゴリズムを自己安定拡張するための方法を述べる。

2 モデル

システムは n 個のプロセッサとプロセッサを結ぶ通信チャネルから成る。各プロセッサは状態機械 (state machine) であり、自分の状態と通信チャネルからの情報から次の状態を決定する。

自己安定アルゴリズムで用いられるネットワークモデルは通信方式の違いから以下の 3 つに分類できる。

1. 状態通信モデル ……隣接プロセッサのすべての内部変数の状態を知ることができる。
2. レジスタ通信モデル ……各チャネル方向に対して 1 つのレジスタが存在し、そこに隣接プロセッサへのメッセージを書き込むことにより通信を行なう。
3. メッセージ通信モデル ……各プロセッサはメッセージの送受信により通信を行なう。

ここではネットワークモデルにメッセージ通信モデルを用いる。メッセージ通信モデルにおいては、各チャネルは双方向の FIFO キューである。チャネルのバッファは無限だが、メッセージは有限時間内に相手に到着するものとする。プロセッサは受信命令を実行すると自分につながったチャネルからメッセージを一つ読み込むが、もしメッセージがない場合は、メッセージが到着するまで他の処理をせず待ち続けるものとする。

自己安定アルゴリズムはシステムのすべてのプロセッサを同一の状態機械とするかどうかで次のように分類できる。

- 均一 ……すべてのプロセッサは同一の状態機械であり、同一のプログラムを実行する。分散アルゴリズムの議論では、同一のプログラムを実行するとしながら、実際には各プロセッサが異なる識別子を持ち、その識別子をパラメータとするプログラムを実行することもあるが、ここでの均一はそのような識別子も存在しないことを表す。
- 準均一 ……定数個の特別なプロセッサがあり、それ以外のプロセッサは同一の状態機械である。定数個の特別なプロセッサはそれぞれ異なる状態機械であってもよい。
- 識別子付き ……各プロセッサは相異なる識別子を持ち、識別子をパラメータとする状態機械である。(識別子をパラメータとするプログラムを実行する)

また、許されるスケジュールの違いから次のように分類できる。

- C デーモン (Central daemon) ……同時に 1 つのプロセッサしか動かないという制限を加えたモデル。
- D デーモン (Distributed daemon) ……同時に複数のプロセッサが動作するモデル。

ここでは均一なモデルおよび D デーモンを仮定する。

プロセッサの局所状態 (local state) は、変数の値とプログラムカウンタの値で表される。システムの全域状態

(global state) は、システムを構成するプロセッサの局所状態とFIFOチャネルに残っているメッセージのリストで、システム内のプロセッサ数をn、通信チャネルの数をmとし、$S_i(1 \leq i \leq n)$ をプロセッサ P_i の状態の集合としたとき、全域状況は $c=(s_1, s_2, \ldots, s_n, L_{e_1}, \ldots, L_{e_m})$ で表される。ただし、$s_i \in S_i$ かつ L_{e_j} はチャネル e_j の状態 (e_j にあるメッセージのリスト) とする。

プロセッサ P_i は原子動作 (atomic step) を実行すると、状態遷移関数に δ にしたがって状態をかえ、プロセッサの状態は s_{i_1} から s_{i_2} に、P_i に接続している l 本のチャネル e_{k_j} の状態は $L_{e_{k_j}}$ から $L'_{e_{k_j}}(1 \leq j \leq l)$ になる。

$$\delta(i, s_{i_1}, L_{e_{k_1}}, \ldots, Le_{k_l}) = (s_{i_2}, L'_{e_{k_1}}, \ldots, L'_{e_{k_l}})$$

A を任意のアルゴリズム、Q をシステム内のプロセッサ集合の任意の部分集合、任意の全域状況を c とする。Q に属するすべてのプロセッサが A で決まる原子動作を行ない全域状況 c_{i+1} になったとき、これを $c_i \rightarrow (Q, A)c_{i+1}$ あるいは簡単に $c_i \rightarrow c_{i+1}$ で表す。

プロセッサ集合の無限系列をスケジュールと呼ぶ。$T = Q_0, Q_1, \ldots$ を任意のスケジュールとする。このとき、全域状況の無限系列 $E = c_0, c_1, \ldots$ が各 i について $c_i \rightarrow (Q_i, A)c_{i+1}$ が成り立つとき、E を初期状況 c_0、スケジュール T に対するアルゴリズム A の実行系列と呼ぶ。スケジュール T にすべてのプロセッサが無限回表れるとき、T は公平 (fair) であるという。また、スケジュール T が公平な実行系列 E は公平であるという。

実行系列の集合 LE を定義された正しい実行 (legal execution) とする。任意の全域状況 c_0 で始まるアルゴリズム A の公平な実行 $E = c_0, c_1, \ldots$ に対して、ある j が存在し、$E_j = c_j, c_{j+1}, \ldots$ が LE に属するとき、A は LE に関して自己安定であるという。また、このとき、E_j 中の各全域状況 $c_j, c_{j+1}, \ldots$ を正当な状況 (legitimate state) とよぶ。

3 均一なモデル上での分散プログラムの自己安定拡張

Katzら [9] はメッセージ通信モデルにおいて非自己安定のプログラムを自己安定拡張にするための方法について述べている。具体的には、分散アルゴリズムPに自己安定コントロールプログラムを加えるという方法をとる。コントロールプログラムはPのステップの間に入って、

1. 全域状態のスナップショットをとる。
2. スナップショットが正当な状況であるかどうかをテストする。
3. 正当な状況でないときには、各プロセッサの局所状態をリセットしてデフォルトの正当な状況に移す。

ということを繰り返し行なう。

しかし、このコントロールプログラムでは特定のプロセッサがスナップショットを行い、さらに各プロセッサの識別子を用いて、プロセッサが同じメッセージを何度も処理しないようにしている。したがって、準均一なモデル上でのみしか動作しない。

そこで本稿では、プロセッサが識別子を持たない均一なモデルにおいて動作する自己安定コントロールプログラムを述べる。基本的なアイデアは、まずリーダを選挙するアルゴリズムを走らせて選ばれたリーダプロセッサがスナップショットを始めるようにする。しかし、自己安定アルゴリズムは通常の分散アルゴリズムと異なり、一般にアルゴリズムが問題を問くことを終了したかどうか判定できないため、2つのアルゴリズムが並行に動作するようにアルゴリズムの合成を行なう。

レジスタ通信モデルにおける自己安定アルゴリズムの合成は次のように定義される [4]。

Definition レジスタ通信モデルでの公平な合成

自己安定アルゴリズムA1とA2の公平な合成とは、各プロセッサがA1とA2の原子動作を交互に実行することである。ただし、通信用レジスタに書き込まれる値は構造体になっており、レジスタへの書き込みは互いに他のアルゴリズムで用いる変数を壊さないものとする。

命題 [4] A_1 を任意のネットワークで問題 P_1 を解く自己安定アルゴリズム、A_2 を P_1 の解を前提として問題 P_2 を解くアルゴリズムとする。A_2 が書き換える変数を A_1 が使用しなければ、A_1 と A_2 の公平な合成によって得られるアルゴリズムは、任意のネットワークで P_2 を解く自己安定アルゴリズムである。

自己安定アルゴリズム A_1 は A_2 と独立に動作するため、A_1 は問題 P_1 の解状態で安定する。この状況で A_2 が動作すれば、いずれ A_2 は問題 P_2 の解状態で安定する。任意のネットワークでリーダ選挙を行なう自己安定アルゴリズムを A_1、リーダがいるネットワークにおいてスナップショットを行なうアルゴリズムを A_2 とすると、任意のネットワークでスナップショットを行なう自己安定アルゴリズムが得られる。

メッセージ通信モデルにおいては、レジスタの値がいつでも読めるレジスタ通信モデルと異なりメッセージの

到着を待つこともある。2つの自己安定アルゴリズム A_1,A_2 を動作させたときには、

- 1つのメッセージは A_1,A_2 の両方のメッセージを含む。
- メッセージ受信後、その受信メッセージをもとに A_1, A_2 の実行を行なう。また、A_1,A_2 を実行後、双方の結果を1つのメッセージとして送信する。

として合成する。

以下では

1. 識別子のない均一なネットワークにおいてリーダを選挙する自己安定アルゴリズム
2. リーダが与えられたときに snapshot を行なう自己安定アルゴリズム

の2つに分けて説明する。

3.1 リーダ選挙アルゴリズム

リーダ選挙アルゴリズムとして、レジスタ通信モデルにおいてリーダ選挙を行なう均一な自己安定アルゴリズム [5] をメッセージ通信モデルにおいて動くように変更する。

メッセージ通信モデルでは、空のチャネルからメッセージを受信しようとしたときには、相手プロセッサがメッセージを送信してくるまで何も仕事をしない。したがって、以下のことが言える。

1. メッセージ通信モデルにおける自己安定アルゴリズムにおいては、メッセージ送信と受信は atomic に行われ、かつ受信より前に送信が存在しなければならない。(∵すべてのチャネルが空の状態で、各プロセッサが受信を行えばデッドロックになるのは明らか。Dolev ら [5] は初期状態では必ず1つはメッセージがチャネルに存在するという仮定を置いて、原子動作を受信・送信の順としている)
2. 各プロセッサにおいて、送信プロセッサ集合と受信プロセッサ集合が全隣接プロセッサでなく、かつあるチャネルについて受信メッセージ数が送信数より多いときには受信が待たされることがあり、木でないネットワークでデッドロックを起こす可能性がある。(∵閉路が存在したとき、送信してこない相手を受信状態で待つ、というループが生じる可能性がある)

したがって、本稿では以下のような仮定を置く。

> プロセッサは自分につながるすべてのチャネルにメッセージ送信をした直後に、すべてのチャネルからメッセージ受信を行ない、メッセージ送信から受信までは atomic に行なわれるものとする。

Katz ら [9] の場合、システム内に一つ特殊なプロセッサを置き、そのプロセッサが定期的にメッセージを送り続けることでデッドロックを回避している。

3.1.1 アルゴリズムの概略

このリーダ選挙アルゴリズムは、システム内に1つの根付き木を構成することでリーダ (根) を選ぶものである。

各プロセッサはクロックを持つ。ここでのクロックは Lamport[10] と同一のものであり、メッセージにこのクロックをのせることで、メッセージの情報の新旧の判別を行なう。

各プロセッサは tid(tree id) を持つ。tid は木に共通の値であり、bit 列で表される。根プロセッサのみが tid を変更できる。tid の変更はランダムに選んだ bit を tid に追加することで行なわれる (つまり tid は単調増加)。すべてのプロセッサは自分の tid の値をメッセージと一緒に送信する。各プロセッサはすべてのチャネルからメッセージを受信後、メッセージと一緒に送られた tid の最大値と自分の tid を比較し、もし送られてきた tid の方が大きいときには自分の tid を送られてきた tid に変更し、その tid を送ったプロセッサを親とする。各プロセッサは根からの距離を計算する。その距離が増加するならば木にサイクルがあると判断して、自分が根になる。

システム内の全プロセッサが同じ tid を持つようになったとき、システム内部に同じ tid を持つ木が複数存在する可能性がある。以下の手続きにより、それを検出する。根プロセッサが8色中から前回の色と今の色を除いた6色からランダムに1色選び、子プロセッサに色を伝えていく。子プロセッサは自分のすべての子プロセッサについて色が変わったというしるしである ack が true になったら親に ack=true を返す。根プロセッサはすべての子プロセッサの ack が true になったらまた色を変更する。このようにして、あるプロセッサが色が異なるプロセッサ (つまり、別の木のプロセッサ) を発見した場合、*other_trees* というフラグを true にして親プロセッサに伝える。*other_trees* = *true* となった根プロセッサは、自分の tid の bit を増やす。色の選択や tid の bit はランダムに選ばれるので、このアルゴリズムは非決定的アルゴリズムである。

```
repeat
  atomicbegin
```

```
    for j := 1 to d do (* メッセージ送信 *)
       send_message {tid := tid; (*tree id*)
          dis := dis; (* 根との距離 *)
          order := j; (* チャネルの順番 *)
          parent := f; (* 親 *)
          sender_clock := clock; (* メッセージのクロック *)
          rclock := rclock; (* 根のクロック *)
          color := color; (* 色 *)
          ack := ack; (* 色チェックが終われば true*)
          other_trees := other_trees; (* 他の木が存在 *)
       } to j
    endfor;
    for j := 1 to d do
       mes[j] := receive_message(j) endfor;
    (*- メッセージ受信 *)
  atomicend;

  mtid := max(mes[j].tid);
  mdis := min {mes[j].dis|mes[j].tid = mtid} + 1;
  mf := {first j|mes[j].tid = mtid
     and mes[j].dis = (mdis − 1) }

  if (tid, dis) > (mtid, mdis) then
     dis := 0; (* 根になる *)
  else if not (dis = 0 and tid ≥ mtid) then
        (*tid変更不要の根でない場合 *)
        (* 最大の tid を持つ隣接プロセッサを親とする *)
        f := mf; tid := mtid; dis := mdis; endif;
  endif;
  if (∀j|mes[j].tid = tid) then
     one_tree_election() endif;
  (*one tree election の実行 *)

  clock := max(clock − 1, mes[j].sender_clock) + 1;
  if dis = 0 then rclock := clock; (* 根の場合 *)
  else rclock := mes[f].rclock endif; (* 根でない場合 *)
  if color_modify = true then
     color_clock := rclock;
     color_modify := false
  endif
until forever;

function son(j : integer) : boolean;
begin
  if mes[j].order = mes[j].parent then son := true
  else son := false endif
end;

procedure one_tree_election();
begin
  if (dis = 0 and {∀j|son(j) → mes[j].ack = true
      and mes[j].color = do_color})
   then
     if {∃j|mes[j].other_trees = true or
        (mes[j].color ≠ do_color
         and mes[j].rclock ≥ color_clock) } then
        tid := extend_tid(tid); (*tree id の変更 *)
     endif;
     do_color := choose_color(previous_color, do_color);
     color_modify := true; (* 色の変更 *)
  else (*non root *)
     if (dis ≠ 0) and (do_color ≠ mes[f].color) then
        other_trees := false; (* 親の色が変更された *)
        do_color := mes[f].color;
        color_modify := true;
        ack := false;
     else if ((not ack) and
        {∀j|son(j) → mes[j].do_color = do_color
           and mes[j].ack}
        and {∀j|mes[j].rclock ≥ color_clock}) then
          if ( {∃j|son(j) and mes[j].other_trees}
           or {∃j|mes[j].color ≠ do_color})
           then other_trees := true endif;
          ack := true
        endif
     endif
  endif
end
```

図1: [リーダ選挙アルゴリズム]

以下、[5][7][8] と異なる部分の証明の概略を述べる。

Lemma 1 clock は単調増加である。 □

Lemma 2 任意のプロセッサ P_i における任意の実行において、(tid_i, dis_i) は単調増加である。ただし、(tid,dis)>(tid',di は、tid>tid' のとき、あるいは tid=tid' かつ dis<dis' のときに成り立つものとする。 □

Definition 1 良い全域状況
全域状況 c が良い状況であるとは、任意のチャネルにおいて、存在するメッセージを受信される順番に $M_1, ..., M_n$、送信プロセッサを P_j としたとき、

(1) $clock_{P_j} \geq M_n.sender\_clock \geq M_{n-1}.sender\_clock \geq \ldots \geq M_1.sender\_clock$

(2) $(tid_{P_j}, dis_{P_j}) \geq (M_n.tid.M_n.dis) \geq (M_{n-1}.tid, M_{n-1}.dis) \geq \ldots \geq (M_1.tid, M_1.dis)$

が成り立つことである。 □

Definition 2 FSG(father-son relation graph)
任意の全域状況 c において、P_j は P_i を親と思っているとき、FSG(c) には P_i から P_j への有効枝がある。 □

Lemma 3 システムはいつか良い全域状況になり、良い全域状況は続く。 □

以下、c は良い全域状況になってから各プロセッサがメッセージ送受信とその後の状態変更の繰り返しを 1 度以上実行した全域状況とする。

Lemma 4 c において P_s が P_f を親と考えているような任意の P_f, P_s について、
$(tid_f, dis_f) > (tid_s, dis_s)$
Proof ある時点において、P_s が P_f を親と思ったとき、最後の iteration における P_f からのメッセージの tid,dis をそれぞれ tid'_f, dis'_f とすると、$tid'_f = tid_s, dis'_f = dis_s + 1$ である。したがって、$(tid'_f, dis'_f) > (tid_s, dis_s)$。また、良い全域状況の定義より $(tid_f, dis_f) \geq (tid'_f, dis'_f)$。以上より、$(tid_f, dis_f) > (tid_s, dis_s)$ □

Lemma 5 c においては、FSG(c) は林になる。
Proof c において FSG(c) に閉路が存在したと仮定し、

閉路のプロセッサを $P_{k1}, P_{k2}, \ldots, P_{kn}$ とする。Lemma 4より $(tid_{k1}, dis_{k1}) > (tid_{k2}, dis_{k2}) > \ldots > (tid_{kn}, dis_{kn}) > (tid_{k1}, dis_{k1})$ となり矛盾。また、各プロセッサは根であるか、親を持つかのいずれかであるので、FSG(c) は林である。 □

Lemma 6 c において R(c) は c における根プロセッサの集合とする。任意の連続する全域状況 $c_i \rightarrow c_{i+1}$ ならば、$R(c_i) \supseteq R(c_{i+1})$ である。

Proof 新たに根が作られるのは根以外のプロセッサの繰り返しにおいて (tid,dis)>(mtid,mdis) が成り立つときのみである。状況 c よりあとの実行のある繰り返しにおける、あるプロセッサの実行で、この条件が成り立ったと仮定する。このとき、このプロセッサでは良い状況になってから少なくとも 2 回メッセージの送受を行なっている。

(1) tid>mtid が成り立つとき。
このプロセッサは根でないので、今の繰り返しの 1 つ前の繰り返しの実行の際に親を決定しているはずである。その時の繰り返しにおいて、親となったプロセッサから受信したメッセージを M'_f、今回そのプロセッサから受信したメッセージを M_f とすると、$M'_f.tid = tid > mtid \geq M_f.tid$ となり、良い状況であることに反する。

(2) tid=mtid かつ dis<mdis のとき。
(1) 同様、1 つ前の繰り返しで親から受信したメッセージを M'_f、今回受信したメッセージを M_f とする。tid は単調増加だから $M'_f.tid = M_f.tid$ である。このとき、$M'_f.dis+1 = dis < mdis = M_f.dis+1$ となり、良い状況であることに反する。 □

Lemma 7 葉のプロセッサの ack はいつか true になる。

Proof ack が false から true になるのは回りのプロセッサと自分の tid が同じで、かつ任意の隣接プロセッサの rclock が color_clock 以上を満たすときである。tid については大小関係で比較するので、いつか自分と隣接プロセッサは同じ tid を持つようになる。color_clock については変更するのは親の色が変わったときだけであり、rclock はそのプロセッサが知り得るもっとも最近の根プロセッサの clock なので、単調増加していく。したがって、いつか color_clock の値を越える。よって、いつか ack は true になる。 □

Lemma 8 任意のプロセッサの ack はいつか true になる。

Proof あるプロセッサの子プロセッサの色がすべて自分と同じで、かつ ack がすべて true になったと仮定する。このとき、子でないプロセッサについては Lemma 7 と同様の証明で、いつかメッセージの rclock の値が color_clock を越える。したがって、いつか自分も ack が true になる。よって、葉のプロセッサから順番に親に行くにしたがって ack が true になるので、任意のプロセッサはいつか ack が true になる。 □

3.1.2 自己安定スナップショットアルゴリズム

[9] は識別子つきのプロセッサ上でのアルゴリズムであるため、そのままでは識別子のないモデル上では動かない。しかし、3.1.1で述べたリーダ選挙アルゴリズムを用いれば各プロセッサは独自の識別子を持つことができる。例えば、根プロセッサの識別子を 0 とし、親プロセッサの識別子に、親から見たチャネルの番号の値を付加したものをそのチャネルにつながる子プロセッサの識別子にすれば unique な識別子を得ることができる。この識別子は [9] では同じマーカを何度も処理するのを避けるのに用いる。また、リーダ選挙のアルゴリズムを実行する際、各プロセッサが子孫のプロセッサの数を保持し、その数を親に知らせることで、根のプロセッサはシステム内の全プロセッサ数を知ることができる。

Katz[9] のアルゴリズムでは Chandy ら [2] のスナップショットと同様に自分に接続されたチャネルのうちの 1 つからメッセージを読み込み、それに対して処理を行ない、必要に応じて broadcast を行なう。ここでは、3.1.1のリーダ選挙アルゴリズムと合成するために、以下のように変更する。

1. すべてのチャネルからメッセージを 1 つ受信。
2. 各チャネルからのメッセージに応じて処理を行なう。broadcast が必要なメッセージについては、各チャネルごとに用意された queue に送るメッセージを保存しておく。
3. 各チャネル毎に、queue に保存されたメッセージをすべて送信。

この変更では、すべてのチャネルからメッセージを 1 つ受信してしまうものの、1 つずつ処理していくので、受信処理の順番が変わる可能性はあるがスナップショットの動作に影響はない。1 つのメッセージには 3.1.1のリーダ選挙の情報も含まれるているが、一度に複数のメッセージを送ってもリーダ選挙の情報部分はすべて同じなので、リーダ選挙アルゴリズムの動作にも影響はない。これ以外の部分は Katz[9] のアルゴリズムと同一である。

3.2 合成したアルゴリズム

以下、合成したアルゴリズムを示す。簡単のため、3.1.1で説明した部分は省略する。

```
const d : (*-チャネルの数 *);

type
  ID:; (*bit sequence*)
  message = record
   tid : ID; (*tree id*)
   dis : integer; (* 根からの距離 *)
   order : 1..d; (*channel order of sender processor*)
   parent : 1..d; (* 親プロセッサ *)
   sender_clock : integer; (* メッセージのクロック *)
   rclock : integer; (* 根のクロック *)
   color : 1..8; (* 木の色 *)
   ack:boolean; (*true if all children have checked color*)
   other_trees : boolean; (* 他の tree が存在 *)


   snap_flag : boolean; (*true if snapshot message*)
   r_flag : boolean; (*true if message contains a report*)
   marker : boolean; (*true if message is a marker*)
   report_content; (*content of report message*)
   val : integer;
   id : ID; (*sender process id*)
   path : (*list of id*)
   end;

var
  j : integer;
  mes : array[1..d] of message; (*message buffer*)
  clock : integer; (*clock of the processor*)
  rclock : integer : (*possible known clock of the root*)
  color_clock:integer; (*last color-changed time of the root proc.*)
  color_modify : boolean; (*true if changed*)
  tid, mtid : ID; (*tree id*)
  dis, mdis : integer; (* 根からの距離 *)
  f, mf : integer; (* 親プロセッサ *)
  do_color, previous_color : 1..6; (* 色 *)
  ack : boolean; (* 子孫の色チェックが終了 *)
  other_trees : boolean;

  current : integer; (*current iteration number of snapshot*)
  node_id:; (*identifier*)
  basic_state : state; (*basic program P の state*)
  initiated : boolean; (*true if initiated*)
  eoc:array[1..d] of boolean; (*j から marker を受け取った *)
  rec : array[1..d] of array[1..M] of message;
  (*eoc[k] が false の間に k から受けたメッセージ *)
  (*M: large enough to save all the messages*)
  global_state : array of state;
  report_flag : array[1..d] of boolean;
  report_content : array[1..d] of queue;
  report_queue : queue;
  messge_queue : arrary[1..d] of queue;

repeat
  atomicbegin
    for j := 1 to d do
      repeat
        send_message(j, deque(message_queue[j]))
      until empty(message_queue[j]);
      (*message queue[j] に入っているメッセージを送信 *)
    endfor;
    for j := 1 to d do
      mes[j] := receive_message(j)
      (* チャネル j からメッセージを一つ受信 *)
    endfor;
  atomicend;

  (* リーダ選挙のアルゴリズムがここに入る *)

  (*snapshot のアルゴリズム *)
  for j := 1 to d do
    if snap_flag then
      if dis = 0 then (* 根の場合 *)
        if ismarker(j) then
          snapshot(j);
          if finished(j) then report() endif
        endif
        if (isctrl(j) and (not seen(j))
         and isreport(j)) then
          save_report_info(j);
          if finshed(j) then report() endif
        endif
        if (isctrl(j) and (not seen(j))
         and (not isreport(j))) then
          if finshed(j) then report() endif
        endif
        initialize();
      else (* 根でない場合 *)
        if rep_message(j) then
          push_report_info(j);
          report_flag[j] := true;
        endif
        if ismarker(j) then
          snapshot(j);
          if finished(j) = true then report() endif
        endif
        if (isctrl(j) and (not seen(j))
         and isnext(j)) then
          current = mes[j].val;
          propagate(j);
        endif
        if (isctrl(j) and (not seen(j))
         and (not isnext(j))) then
          propagate(j);
          if finshed(j) then report() endif
        endif
      endif
    endif
  endfor
until forever;

function rep_message(j : integer) : boolean;
begin
  rep_message := mes[j].r_flag
end;

procedure save_report_info(j : integer);
begin
  report_queue :=
  push(report_queue, mes[j].report_content);
end;

function ismarker(j) : boolean;
begin
  if eoc[j] = false and mes[j].val = current
   then ismarker := true
  else ismarker := false endif
end;

function isctrl(j : integer) : boolean;
begin
  isctrl := (not ismarker(j));
end;
```

```
function seen(j : integer)  : boolean;
begin
   if ((exist(node_id, mes[j].path))
    and ((node_id ≠ null)  or (path ≠ null)))
    then seen := true
   else seen := false endif
end;

function exist(id, path)  : boolean;
begin
   (*id が path に存在するなら true*)
end;

function isnext(j : integer)  : boolean;
begin
   if (mes[j].val > current)  then isnext := true
   else isnext := false endif
end;

function finished(j)  : boolean;
begin
   if ((mes[j].val ≠ current)
    or (∀k, eoc[k]))  then
      finished := true
   else finished := false endif
end;

procedure report();
begin
   report_queue :=
   push(report_queue, id, basic_state, rec)
end;

procedure snapshot(j : integer); (*j からのメッセージ*)
begin
   if (not initiated)  then
      initiated := true;
      for i := 1 to d do
         eoc[i] := false;  record [j] := nil
      endfor
      eoc[j] := true;
      basic_state := state of P(*-save checkpoint *)
      (*隣接プロセッサにマーカを送る*)
   else if (initiated and (not eoc[j]))  then
         eoc[j] := true;  endif
   endif
end;

(*メッセージをキューに保存*)
procedure propagate(j : integer);
begin
   for j := 1 to d do
      message_queue
       := enqueue(message_queue, j, mes[j].path, val)
   endfor
end;

procedure initialize(); (*根で自発的に実行*)
begin
   if (∀j|report_flag[j])  then
      (*reported from all proc.*)
      for j := 1 to d do report_flag[j] = false endfor;
      current := current + 1;
      initiated := false;
   end;
   (*path=(),val=current として broadcast*)
end
```

図 2: [合成したアルゴリズム]

4 まとめ

本論文では、均一なメッセージ通信モデルにおいてリーダ選挙および分散スナップショットを行う自己安定アルゴリズムとそれを用いた非自己安定プログラムの自己安定化について述べた。メッセージ通信モデルにおいては、チャネルに強い仮定を置くことにより、レジスタ通信モデルで動くアルゴリズムの一部は、メッセージ遅延だけを考慮すれば、ほとんど変更なしで動かすことができる可能性がある。

参考文献

[1] B. Awerbuch and G. Varghese: "Distributed Program Checking: a Paradigm for Building Self-stabilizing Distributed Protocols," *Proc. 32nd FOCS*, pp.258–267(1991).

[2] K. M. Chandy and L. Lamport: "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM TOCS*, Vol.3, No. 1, pp. 63–75(1985).

[3] E.W. Dijkstra: "Self-stabilizing systems in spite of distributed control," *Comm. ACM*, Vol.17, No.11, pp.643–644(1974).

[4] S. Dolev, A. Israeli, and S. Moran: "Self Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity, " *ACM Proc. of 9th PODC*, pp.103–117(1990).

[5] S. Dolev, A. Israeli, and S. Moran: "Resource Bounds ofr Self Stabilizing Message Driven Protocols," *ACM Proc. of 10th PODC*, pp.281–293(1991).

[6] S. Dolev and A. Israeli: "Uniform Self-Stabilizing Leader Election (Extended Abstract)," WDAG,(1991).

[7] S. Dolev, A. Israeli, and S. Moran: "Uniform Self-Stabilizing Leader Election Part 1: Complete Graph Protocol," *TR 807*, Computer Science Dept., Technion.(1994).

[8] S. Dolev, A. Israeli, and S. Moran: "Uniform Self-Stabilizing Leader Election Part 2:General Graph Protocol," *TR 94-039*, Dept. Computer Science, Texas A&M Univ.(1994).

[9] S. Katz and K.J. Perry: "Self-stabilizing Extensions for Message-passing Systems," *ACM Proc. of 9th PODC*, pp.91–103(1990).

[10] L. Lamport: "Time, Clocks, and the Ordering of Events in a Distributed Systems," *Comm. ACM*, Vol. 21, No.7, pp. 558-563, July (1988).

[11] 増澤、片山: "自己安定アルゴリズムについて," 情報処理, Vol.34, No.11 (1993).

[12] M. Schneider: "Self-Stablization,"*ACM Computing Surveys*,Vol.25, No.1, pp.45–67(1993).

[13] 山下、亀田: "分散アルゴリズム," 近代科学社 (1994).