

Cデーモンによるリングの方向付け自己安定アルゴリズム

正員 片山 喜章[†] 正員 増澤 利光[†] 正員 都倉 信樹^{††}

Self-Stabilizing Ring Orientation Algorithm under the C-Daemon

Yoshiaki KATAYAMA[†], Toshimitsu MASUZAWA[†] and Nobuki TOKURA^{††}, *Members*

あらまし 自己安定アルゴリズムとは、任意の初期状況からアルゴリズムを開始しても、有限時間内に解を求めて安定する分散アルゴリズムである。本論文では、リングネットワークの方向付け問題(ROP)を解く自己安定アルゴリズムについて考察する。ROPとは、リングネットワーク上のすべてのプロセッサを、時計回りか反時計回りのいずれか一方に方向づける問題である。これまでに、偶数個のプロセッサからなるリングネットワークでは、Dデーモン、R/WデーモンのもとでROPを解く決定性自己安定アルゴリズムが存在しないことが知られている。また、Dデーモンのもとで任意サイズのリングネットワークのROPを解く確率的な自己安定アルゴリズムが知られている。本論文では、Cデーモンのもとで任意サイズのリングネットワークのROPを解く決定性自己安定アルゴリズムを提案する。この結果は、CデーモンとDデーモンで解ける問題のクラスに真に差があることを意味している。

キーワード 分散アルゴリズム, リングネットワーク, 自己安定, リングの方向付け

1. ま え が き

通信用リンクで接続されたプロセッサが、メッセージを交換しながら協調して問題を解くアルゴリズムを、分散アルゴリズム(distributed algorithm)と言う。通常の分散アルゴリズムでは、アルゴリズム実行開始時のネットワーク状況に、次の仮定を置く。

- ・各プロセッサは、初期化されている。
- ・ネットワーク(通信リンク)中に伝搬中のメッセージは存在しない。

これに対し、任意のネットワーク状況から始めても解を求めて安定する分散アルゴリズムが自己安定アルゴリズム(self-stabilizing algorithm)である。自己安定アルゴリズムの長所には次の三つが挙げられる。

- (1) ネットワーク全体のいかなる初期化も必要としない。
- (2) アルゴリズム実行中にプロセッサのもつデータの破壊(プログラムカウンタの値の破壊も含む)など「一

時故障(transient failure)」「ローカルには検出できない」が起こっても、その後十分に長い間故障が起こらなければ問題を解くことができる。

(3) アルゴリズム実行中にネットワーク形状が変化しても、その後十分に長い間形状変化が起こらなければ問題を解くことができる。

特に、長所の(2)、(3)は、自己安定アルゴリズムが任意の初期状況から解を求めることができることによるものである。すなわち、一時故障やネットワーク形状の変化が起きても、その状況を初期状況と考えれば、やがて解を求めることができる。

最初に提案された自己安定アルゴリズムは、方向づけられたリングネットワーク上で排他制御を行うものであった⁽³⁾。最近では、リングの方向付け問題を解くアルゴリズムなど多様なアルゴリズムが提案されている。これらの自己安定アルゴリズムを特徴づける一つの要因として、以下のデーモン(プロセッサの原子動作、および複数のプロセッサの同時動作が可能かどうかを規定する)が考えられている。但し、プロセッサ間の通信は、各リンクの各通信方向に用意された通信用レジスタを介して行うものとする。

- ・Cデーモン(Central daemon)

「一時に一つのプロセッサしか動作しない」かつ「一原

[†] 奈良先端科学技術大学院大学情報科学研究科, 生駒市
Graduate School of Information Science, Nara Institute of Science and Technology, Ikoma-shi, 630-01 Japan

^{††} 大阪大学基礎工学部情報工学科, 豊中市
Faculty of Engineering Science, Osaka University, Toyonaka-shi, 560 Japan

子動作で全隣接レジスタから情報を読み込み、自分の状態を変化させ、必要があれば全隣接レジスタへの書き込みができる」。

・D デーモン (Distributed daemon)

「同時に複数のプロセッサが動作できる」かつ「一原子動作で全隣接レジスタから情報を読み込み、自分の状態を変化させ、必要があれば全隣接レジスタへの書き込みができる」。

・R/W デーモン (Read/Write daemon)

「一時に一つのプロセッサしか動作できない」かつ「一原子動作で一つのレジスタからの読み込みと内部状態の変化、あるいは一つのレジスタへの書き込みと内部状態の変化ができる」。

R/W デーモンで動作する自己安定アルゴリズムが変更なしに D デーモンでも動作するという意味で、R/W デーモンは D デーモンよりも弱いモデルである。また D デーモンで動作するものが変更なしに C デーモンで動作するという意味で、D デーモンは C デーモンよりも弱いモデルである。なお、複数のプロセッサが同時に動作することを許すように R/W デーモンの制限を弱めたモデルも考えられるが、このように制限を緩めても R/W デーモンと本質的な差は生じないことが知られている⁽¹⁾。

文献(3)は、特別なプロセッサが存在する (non-uniform) 任意サイズの方向づけられた (oriented) リングネットワークで排他制御問題 (ME) を解くための自己安定アルゴリズムを示した。ここで、方向づけられたリングネットワークとは、リングネットワーク (以下、単にリングと呼ぶ) 上のすべてのプロセッサが同じ方向 (例えば時計回り) を共通して認識しているリングである。このようなネットワークは、方向感覚付きリングと呼ばれることもある。また文献(3)は、特別なプロセッサが存在しない均一 (uniform) (すべてのプロセッサが対等で、識別子も存在しない) リングでは、サイズが合成数の場合、リングが方向づけられていても、ME を解く自己安定アルゴリズムは存在しないことも示している。すなわち、リングに対し「方向づけられていること」と「特定なプロセッサが存在すること」の二つを仮定した。一方、文献(4)では、方向づけられたリング上でサイズが素数の場合には、特別なプロセッサが存在しなくても (すなわち均一なリングであっても) ME を解く自己安定アルゴリズムが存在することが示された。また文献(5)では、特別なプロセッサが存在する任意のネットワーク上で、R/W デーモンのもと

Deterministic	C daemon	D daemon	R/W daemon
any size	$A_c, 6$	×	×
even size	$A_c, 6$	×	×
odd size	$A_c, 6$	6	6

× : unsolvable (1)

6 : results of (6)

A_c : our results

図 1 既知の結果と今回の結果

Fig. 1 Known negative results and our positive results.

で ME を解く自己安定アルゴリズムが提案され、その結果特別なプロセッサが存在すれば、方向づけられていないリング上でも ME 問題が解けることがわかった。また文献(2)で、均一な方向づけられたリング上で、ME を解く確率的な自己安定アルゴリズムを示した。

方向づけられていないリングを方向づける問題を、リングの方向付け問題 (ROP) と言う。つまり、リング上のすべてのプロセッサに同一方向 (例えば時計回り方向) を共通認識させる問題である。文献(1)は、偶数サイズの均一なリングでは、D デーモンのもとで (すなわち R/W デーモンのもとでも) ROP を解く決定性自己安定アルゴリズムが存在しないことを示している。また、文献(1)では、任意サイズの均一なリング上で、D デーモンのもとで ROP を解く確率的な自己安定アルゴリズムが提案されている。このアルゴリズムと文献(2)のアルゴリズムに対して、公平な合成⁽⁶⁾という、複数の自己安定アルゴリズムを合成する手法を用いることによって、均一なリング上で ME を解く確率的な自己安定アルゴリズムが導出される。

本論文では、C デーモンのもとで、任意サイズの均一なリング上で ROP を解く決定性アルゴリズムを示す。D デーモンのもとでは、偶数サイズのリングで ROP が解けないことが知られており、この結果は C デーモンと D デーモンで解ける問題のクラスに真に差があることを初めて示したという意味でも興味深い (図 1)。また、自己安定アルゴリズムの正しさを、単調減少関数を利用して証明している。

2. モデル

ここでは、本論文で扱うネットワークやアルゴリズムなどのモデルとその定義について述べる。

2.1 ネットワークとプロセッサ

本論文では、 n 個のプロセッサがリング状に (通信) リンクで接続された均一なネットワーク N_n を扱う。均一

なネットワークの意味は、 N_n 中のすべてのプロセッサは同等で、かつ識別子をもたず、アルゴリズム中で自分や隣接プロセッサの識別子を利用しないということである。この意味で、匿名 (anonymous) ネットワークと呼ばれることもある。但し、ここでは説明の便宜上、各プロセッサを識別子を用いて表す。

N_n 中のプロセッサを P_0, P_1, \dots, P_{n-1} とし、その集合を \mathcal{D} とする。すなわち $\mathcal{D} = \{P_0, \dots, P_{n-1}\}$ 。 N_n において、プロセッサ P_i は P_{i-1} 、 P_{i+1} ($i-1, i+1$ は $\text{mod } n$ で考える) との間に通信リンク $L(i, i-1)$ 、 $L(i, i+1)$ をもつ。リンクでつながれた二つのプロセッサは隣接していると言う。すなわち、 $\text{mod } n$ で1 違いの添字をもつプロセッサ同士は隣接している。リンク $L(i, j)$ は二つの (通信用) レジスタ $R(i, j)$ 、 $R(j, i)$ をもち、 P_i が $R(i, j)$ に書き込んだ値を P_j が読み出すことによって、 P_i から P_j への通信が行え、同様に $R(j, i)$ を用いて P_j から P_i への通信が行える (図 2)。しかし、プログラム中で $R(i, j)$ を使うとプロセッサの識別子を利用していることになり、ネットワークの均一性の仮定に反する。そこで、各プロセッサは一種の代名詞として、二つの隣接プロセッサのどちらか一方を第1プロセッサ、他方を第2プロセッサとして区別する。すなわち、 P_i が第1、第2プロセッサと呼ぶプロセッサをそれぞれ $P_i(1)$ 、 $P_i(2)$ と表す。但し、 $P_i(1)$ が実際には P_{i-1} なのか P_{i+1} なのかは、ここで考えるアルゴリズムでは知り得ない。そして、 $P_i(1)$ 側のリンクの読出し/書込みレジスタを R_1/W_1 と表し、同様に $P_i(2)$ 側のレジスタを R_2/W_2 と表す。こうして、アルゴリズム中では $\text{read}(R_1)$ 、 $\text{write}(W_1)$ 、 $\text{read}(R_2)$ 、 $\text{write}(W_2)$ などとして、読出し/書込み動作を表す。読出し/書込みレジスタを指定したとき、実際に対応するレジスタはアルゴリズムのものと階層のプロトコルで決定される (例えば、物理層のポート番号の対応で決まるなど) と考えればよい。また、アルゴリズム実行中、この対応は不変であることも併せて仮定する。

リングが方向づけられた状況というのは、各プロセッサが第1プロセッサか第2プロセッサのどちらかを選

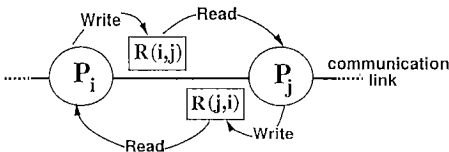


図2 プロセッサと通信レジスタ
Fig. 2 Processors and communication registers.

んだ状況で、かつすべてのプロセッサが選んだ方向がリング上の時計回りか、あるいは反時計回りのどちらか1方向に一致している状況である。

[定義1 (方向づけられたリングネットワーク)] 各プロセッサが以下の条件を満たすとき、リングネットワークが方向づけられていると言う。

各プロセッサ P_i は、アルゴリズムを実行して第1プロセッサ側か第2プロセッサ側のいずれか一方を *head*、他方を *tail* と決める。 P_i によって *head*、*tail* と指定されたプロセッサをそれぞれ $\text{head}(P_i)$ 、 $\text{tail}(P_i)$ と表す。このときすべてのプロセッサ $P_i \in \mathcal{D}$ について、

$$\text{tail}(\text{head}(P_i)) = P_i$$

が成立するとき、 N_n は方向づけられたと言う。□

2.2 デモン

N_n のアルゴリズムにかかわる (計算) 状況を次のように定義する。プロセッサ P_i のとり得るプログラムカウンタの値と、内部変数の値からなる状態の集合を Q_i とする。実際には、均一なネットワーク上のプロセッサで同一のアルゴリズムを走らせるので、 Q_i は単に Q と表してもよい。また、(通信用) レジスタのとり得る値の集合を R とする。今、ある時点で N_n を見たとき、プロセッサ P_i の状態が $q_i \in Q$ 、レジスタ $R(i, j)$ の状態が $r_{ij} \in R$ となっているとして、

$$C = (q_0, q_1, \dots, q_{n-1}, r_{01}, r_{0(n-1)}, r_{12}, r_{10}, \dots, r_{(n-1)0}, r_{(n-1)(n-2)})$$

をそのときの N_n のネットワーク状況と呼ぶ。 N_n のとり得るすべてのネットワーク状況の集合を C と表す。すなわち、 $C = Q^n \times R^{2n}$ 。

アルゴリズム \mathcal{A} は、個々のプロセッサのプログラムとして具体化され共有されているが、分散アルゴリズムの性質から、すべてのプロセッサが同時に動作するとは限らない。 N_n 全体でのアルゴリズムの実行は、連続的でなく段階的 (step-by-step) に進行すると考えることができるので、第 i ステップに実際に動作するプロセッサの集合を $S(i)$ と表す。実時間的に見れば、 $S(i) = \emptyset$ となるステップもあり得るが、ここでは簡単のため、 $S(i) \neq \emptyset$ となる時点ごとにステップを考える。各ステップでの $S(i)$ およびプロセッサの原子動作の違いにより、次のようなモデルが考えられている^{(1)~(5)}。

・C デモン：最も強いモデル

(1) 一時に一つのプロセッサのみ動作する。すなわち、各 t に対して $|S(t)| = 1$ 。

(2) 一原子動作： R_1 、 R_2 から情報を読み出し、内部計算を行い、必要なら W_1 、 W_2 に情報を書き込む。

・D デーモン：

(1) 同時に複数のプロセッサの動作を許す。すなわち、各 t に対して $|S(t)| \geq 1$ 。

(2) 一原子動作：C デーモンと同じ。

・R/W デーモン：最も弱いモデル

(1) 一時に一つのプロセッサのみ動作する。すなわち、各 t に対して $|S(t)| = 1$ 。

(2) 一原子動作：次の二つのいずれか一つ。

(a) R_1, R_2 のいずれか一方のレジスタからの読出しと内部計算。

(b) W_1, W_2 のいずれか一方のレジスタへの書込みと内部計算。

[定義2 (スケジュールと実行)] 集合 X の要素の無限列を X^∞ と表す。ステップ i で動作するプロセッサの集合を $S(i)$ と表すとき、

$$T = S(0), S(1), \dots \in (2^{\mathcal{P}})^\infty$$

をスケジュールと呼ぶ。

アルゴリズム \mathcal{A} に対し、ネットワーク状況 $c \in C$ で、 $S \subset \mathcal{D}$ のプロセッサがそれぞれ一原子動作を行って得られる新しいネットワーク状況を $c(\mathcal{A}, S)$ と表す。

あるネットワーク状況 c_0 から、デーモンの定めるあるスケジュール T でアルゴリズム \mathcal{A} に従って動作するときの N_n のネットワーク状況の無限系列

$$c_0, c_1, \dots \quad (\text{但し, } c_{i+1} = c_i(\mathcal{A}, S(i)), i \geq 0)$$

を、ネットワーク状況 c_0 から始まる \mathcal{A} によるスケジュール T の実行と呼び、 $E(\mathcal{A}, T, c_0)$ と表す。□

[定義3 (公平なスケジュール)] スケジュール T にすべてのプロセッサ $P_i \in \mathcal{D}$ が無限回現れるとき、スケジュール T は公平であると言う。□

C, D, R/W デーモンの定めるスケジュールの集合をそれぞれ $T(C)$, $T(D)$, $T(R/W)$ と表す。これらのスケジュールはすべて公平であると仮定する。スケジュールが公平でない場合、ある時点以後全く動作しないプロセッサが存在し、アルゴリズムが解を求められないことがあるという意味で、この条件は最低限満たさなければならない。

なお、本論文ではC デーモンを扱う。

2.3 自己安定

$\mathcal{L} \subset C$ を N_n の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「アルゴリズム \mathcal{A} は、デーモン X のもとで \mathcal{L} に関して自己安定である」と言い、 $SS(\mathcal{A}, \mathcal{L}, X)$ と書く。また、 $SS(\mathcal{A}, \mathcal{L}, X)$ が成立するとき、 \mathcal{L} を「 (\mathcal{A}, X) に関して正当な状況」と言う。但し、 \mathcal{A}, X が明らかな場合、単に正当な状況と

言う。

(1) 到達可能性

任意のネットワーク状況 $c \in C$ とデーモン X の定める任意のスケジュール $T \in T(X)$ に対し、 c から始まる、アルゴリズム \mathcal{A} によるスケジュール T の実行 $E(\mathcal{A}, T, C)$ の列中に、いつかはネットワーク状況 \mathcal{L} が現れる。すなわち、 $E(\mathcal{A}, T, C) = c_0, c_1, \dots, c_i, \dots$ かつ、 $c_i \in \mathcal{L}$ なる $i \geq 0$ が存在する。

(2) 閉包性

\mathcal{L} 中の任意のネットワーク状況 $c (\in \mathcal{L})$ から始まる任意のスケジュール $T \in T(X)$ についても、それ以降のネットワーク状況は \mathcal{L} にとどまり続ける。すなわち、 $E(\mathcal{A}, T, c) \in \mathcal{L}^\infty$ 。

つまり任意の初期状況から開始しても、アルゴリズム \mathcal{A} による任意の公平なスケジュールの実行は、有限時間内に正当な状況に到達し、かつ1度正当な状況に達すると、それ以降正当な状況のままということである。

そして、正当な状況に属する任意のネットワーク状況、すなわち $c \in \mathcal{L}$ なる任意のネットワーク状況 c が定義1の条件を満たすとき、 \mathcal{L}, X に関して「 X デーモンのもとでリングの方向付け問題 (ROP) を解く自己安定アルゴリズム」と言う。

3. C デーモンでのリングの方向付け問題

ここでは、C デーモンのもとで任意の N_n について ROP を解く決定性自己安定アルゴリズム A_c を示す。

3.1 基本的なアイデア

トークンリングによる LAN では、トークンを一つ回している。ここでもトークンを用いるが、自己安定アルゴリズムでは初期状況でトークンの数が一つということとは保証できない。そこで、各プロセッサでは次のようにトークンを扱う。プロセッサは第1プロセッサから受け取ったトークンを第2プロセッサへ渡し、第2プロセッサから受け取ったトークンは第1プロセッサへ渡すように決めておく。また、既にトークンをもつプロセッサが隣接プロセッサからトークンを受け取るとき、トークンの衝突が発生したと言い、一つを残して他のトークンを消滅させる。

C デーモンのもとでは、同時に1個のプロセッサしか動作しないのでトークンのすれ違いは生じない。よって、互いに逆方向に中継されるトークンはいつか必ず衝突し、時計回りあるいは反時計回りのいずれか1方向に中継されるトークンだけが残る。従って、各プロ

セッサが最も最近トークンを送信したリンクに *head*, 他方に *tail* と印を付けることにより, ROP が解ける.
 [注] この方法は, D デーモンのもとではトークンのすれ違いが生じるためうまく動作しない. 実際に D デーモンのもとでは, 偶数サイズの均一なリングで ROP が解けないことが知られている⁽¹⁾.

3.2 トークンの実現

前述のアイデアは, 初期状況でどのプロセッサもトークンをもたない場合, 正しく動作しない. 自己安定アルゴリズムでは初期状況に何も仮定できないので, この問題点を解決するためにはトークンをうまく実現する必要がある. そこで, まず各プロセッサに次の二つの変数を用意する.

- *direction*

プロセッサの現在の向きを表す. 次の条件に従い, 1, 2 いずれかの値をとる. 任意のプロセッサ P_i で,
 $direction = 1 \Leftrightarrow head(P_i) = P_i(1)$
 $direction = 2 \Leftrightarrow head(P_i) = P_i(2)$
 すなわち, $direction = 1$ とは第 1 プロセッサを *head* と指定し, $direction = 2$ とは第 2 プロセッサを *head* と指定している場合である.

- *weight*

最も最近に送信したトークンがもっていた重み (自然数. ここでは上記の問題点を解決するために, トークンに自然数の重みを付けている).

以後あるプロセッサ P の内部変数 *weight*, *direction* の値を, $W(P)$, $D(P)$ と表す.

ここで, これらの変数の値によってトークンを定義する.

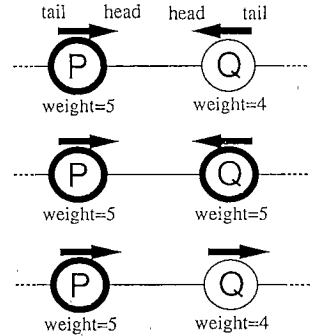
[定義 4 (トークン)] 任意のプロセッサを P , また $Q = head(P)$ とする. プロセッサ P は, 次の (1) か (2) の条件が成立するとき, (Q への) トークンをもつと言う (図 3).

(1) $head(Q) = P$ かつ $W(P) \geq W(Q)$

(2) $tail(Q) = P$ かつ $W(P) > W(Q)$

図 3 ではプロセッサの向きを矢印で示しているが, P と Q が互いに相手と向き合っている状態では, 重み変数 (*weight*) の値の大きい方 (等しい場合は両方) がトークンをもつと考え, 同一方向を向いている場合は, (2) の条件下で P がトークンをもつとする. \square

リングが方向づけられていなければ, 隣接プロセッサと向き合うプロセッサ P と Q が少なくとも 1 組存在し, 定義 4 の (1) より, トークンが少なくとも一つ存在することになる. 一方, リングが方向づけられている



processor with a token

図 3 トークン (1)

Fig. 3 Tokens (1).

ときは (2) の場合のみで, リング中に一つでも変数 *weight* の値の違うプロセッサがあるとトークンが存在する. これらより, トークンがないネットワーク状況に達すれば, リングは方向づけられていることになる.

3.3 アルゴリズム A_c

プロセッサ P_i ($0 \leq i \leq n-1$) の動作を以下に示す. 但し, レジスタ R は, (*Weight* (重み), *Direction* (方向)) をもつ構造体として表され, $r = read(R)$ はレジスタ R から値を読み出して構造体である変数 r に代入し, $write(R, r)$ はレジスタ R に構造体の変数 r の値を書き込む手続きである.

/* Algorithm A_c for ring orientation under C-daemon */

```

struct  $r_1, r_2$  {
    int  $w$ ; /* weight (integer) */
    char  $d$ ; /* direction (head or tail) */
}

do forever
begin /* 原子動作の開始 */
     $r_1 := read(R_1); r_2 := read(R_2);$ 

    if  $r_1.w > weight$  and  $r_1.d = head$  then begin
         $direction := 2; weight := r_1.w$  end
    elseif  $r_1.w = weight$  and  $r_1.d = head$  and  $direction = 1$  then begin
         $direction := 2; weight := r_1.w + 1$  end;
    if  $r_2.w > weight$  and  $r_2.d = head$  then begin
         $direction := 1; weight := r_2.w$  end
    elseif  $r_2.w = weight$  and  $r_2.d = head$  and  $direction = 2$  then begin
         $direction := 1; weight := r_2.w + 1$  end;
    if  $direction = 2$  then begin
         $write(W_1, (weight, tail)); write(W_2, (weight, head))$  end
    else begin
         $write(W_2, (weight, tail)); write(W_1, (weight, head))$  end
end /* 原子動作の終了 */

```

3.4 アルゴリズムの正当性

任意のプロセッサ P_i がアルゴリズム A_c の原子動作を 1 回以上実行すると、次の条件が成り立つ。

・ $direction=1$ ならば $W_1=(W(P_i), head)$, $W_2=(W(P_i), tail)$

・ $direction=2$ ならば $W_1=(W(P_i), tail)$, $W_2=(W(P_i), head)$

(但し, $W(P_i) > 0$)

従って、以下ではこの条件が成立しているものとする。

まず、アルゴリズム A_c に関する正当な状況を定義する。

[定義 5 (アルゴリズム A_c に関する正当な状況)] 正当な状況の集合 \mathcal{L}_A を、以下の 2 条件を満たすすべてのネットワーク状況からなる集合とする。

(1) 各プロセッサ P_i ($0 \leq i \leq n-1$) で $tail(head(P_i))=P_i$ (定義 1 の条件) が成立する。

(2) すべてのプロセッサの内部変数 $weight$ の値が等しい。 □

\mathcal{L}_A に属する任意のネットワーク状況は、定義 1 の条件を満たすので ROP の解である。従って以下では、 $SS(A_c, \mathcal{L}_A, C)$ が成立すること、すなわちアルゴリズム A_c が \mathcal{L}_A に関して自己安定である (到達可能性と閉包性を満たす) ことを示す。

トークンの定義、およびアルゴリズム A_c より、次の補題 1 が成立する。

[補題 1] P_i を任意のプロセッサとする。 P_{i-1} も P_{i+1} も P_i へのトークンをもたないときに P_i が動作しても、ネットワーク状況は変化しない。 □

定義 5 より、正当な状況ではトークンをもつプロセッサが存在しない。従って補題 1 より、次の補題 2 が成立する。

[補題 2] アルゴリズム A_c は、 \mathcal{L}_A に関する閉包性を満たす。 □

次にアルゴリズムが到達可能性を満たすことを証明するために、いくつか関数を定義する。

[定義 6 (関数 t)] 任意の二つのプロセッサ P_i, P_j に対し、真偽関数 $t(P_i, P_j)$ を定義する。 P_i の向き (tail から head の方向) に P_i, P_{i+1}, \dots, P_j ($j \geq i+1$, 添字は mod n) と仮定したとき、次の条件を満たす場合に $t(P_i, P_j)$ が成立する。

$$W(P_i) > W(P_j)$$

つまり、 P_j の向きにかかわらず、 P_i が P_j に対してトークンをもつ場合と考えることができる。また $t(P_i, P_j)$ が成立しないことを $\overline{t(P_i, P_j)}$ と表す。 □

[定義 7 (関数 f)] 定義 6 と同じ仮定のもとで、プロセッサ P_i に対して関数 $f(P_i)$ を次のように定義する。

$$f(P_i) = k : ([\forall l(i+1 \leq l \leq i+k) t(P_i, P_l)] \wedge \overline{t(P_i, P_{i+k+1})}) \text{ を満たす } k)$$

つまり関数 $f(P_i)$ は直観的に、 P_i がトークンをもっていると仮定したときに、そのトークンの重みが変わったり、消滅したりせず中継されるプロセッサ数と考えることができる。次に、あるネットワーク状況における各プロセッサの関数 f の値の総和 T_f を定義する。

[定義 8 (関数 T_f)] あるネットワーク状況 c でのすべてのプロセッサの $f(P_i)$ ($0 \leq i \leq n-1$) の値の総和を $T_f(c)$ とする。すなわち、

$$T_f(c) = \sum_{i=0}^{n-1} f(P_i) \quad \square$$

[定義 9 (関数 T_k)] あるネットワーク状況 c において、ネットワーク中に存在するトークンの数を $T_k(c)$ と表す。 □

更に定義 8, 9 から、次の関数 $F(c)$ を定義する。

[定義 10 (関数 F)] あるネットワーク状況 c における関数 $F(c)$ を次のように定義する。

$$F(c) = (T_k(c), T_f(c)) \quad \square$$

ここで、関数 F を用いて正当な状況を表す。

[補題 3 (関数 F による正当な状況)] 正当なネットワーク状況 \mathcal{L}_A において、かつそのときのみ

$$F(\mathcal{L}_A) = (0, 0)$$

が成立する。 □

(証明) 定義 5 より、正当な状況ではすべてのプロセッサの内部変数が等しく、かつトークンが存在しない。すなわち、 $T_k=0$ かつ $T_f=0$ が成立する。

$T_k=0$ の場合を考える。任意の二つの隣接プロセッサ P, Q ($=head(P)$) を考えたとき、 $head(Q)=P$ が成立しているとトークンの定義 (定義 4) より、必ずいずれか一方がトークンをもつ。従って、

$$head(P)=Q \text{ かつ } tail(Q)=P$$

すなわち、

$$\forall P, tail(head(P))=P \quad (1)$$

が成立しているはずである。また、トークンをもたないこと、および定義 4 より、すべての P, Q に対して、

$$tail(Q)=P \text{ ならば } W(P) \leq W(Q)$$

が成立しなければならない。よって、

$$\forall P, Q, W(P)=W(Q) \quad (2)$$

が成り立つ。すなわち、すべてのプロセッサの内部変数 $weight$ の値が等しい。式 (1)、式 (2) は定義 5 を満たすので、正当な状況である。 □

表1 アルゴリズム A_c の状態遷移表

	動作前状態	重みの関係	動作後状態	$T_k(c)$	$T_f(c)$
1.	$\vec{Q} < [\vec{P}] < [\vec{R}]$		$\vec{Q} < [\vec{P}] = \vec{R}$	-1	減少
2.	$\vec{Q} \geq \vec{P} < [\vec{R}]$	$\{W(Q) < W(R)\}$	$\vec{Q} < [\vec{P}] = \vec{R}$	± 0	減少
3.	$\vec{Q} \geq \vec{P} < [\vec{R}]$	$\{W(Q) \geq W(R)\}$	$\vec{Q} \geq \vec{P} = \vec{R}$	-1	減少
4.	$\vec{Q} < [\vec{P}] < [\vec{R}]$		$\vec{Q} < [\vec{P}] = \vec{R}$	-1	減少
5.	$[\vec{Q}] > \vec{P} < [\vec{R}]$	$\{W(Q) < W(R)\}$	$\vec{Q} < [\vec{P}] = \vec{R}$	-1	減少
6.	$[\vec{Q}] = [\vec{P}] < [\vec{R}]$	上記以外の場合	$\vec{Q} < [\vec{P}] = \vec{R}$	-2	減少
7.	$[\vec{Q}] = [\vec{P}] < [\vec{R}]$	$W(P) + 1 = W(R)$ かつ, $P(1) = Q$	$\vec{Q} < [\vec{P}] > \vec{R}$	-2	減少
8.	$[\vec{Q}] > \vec{P} < [\vec{R}]$	$\{W(Q) = W(R)\}$	$\vec{Q} < [\vec{P}] > \vec{R}$	-1	$O(n)$ 増加
9.	$[\vec{Q}] > \vec{P} < [\vec{R}]$	$\{W(Q) > W(R)\}$	$\vec{Q} = [\vec{P}] > \vec{R}$	-1	減少
10.	$\vec{Q} < [\vec{P}] \geq \vec{R}$		$\vec{Q} < [\vec{P}] \geq \vec{R}$	非動作	
11.	$\vec{Q} \geq \vec{P} \geq \vec{R}$		$\vec{Q} \geq \vec{P} \geq \vec{R}$	非動作	
12.	$\vec{Q} < [\vec{P}] \geq \vec{R}$		$\vec{Q} < [\vec{P}] \geq \vec{R}$	非動作	
13.	$[\vec{Q}] = [\vec{P}] \geq \vec{R}$		$\vec{Q} < [\vec{P}] > \vec{R}$	-1	$O(n)$ 増加
14.	$[\vec{Q}] > \vec{P} \geq \vec{R}$		$\vec{Q} = [\vec{P}] > \vec{R}$	± 0	減少
15.	$\vec{Q} < [\vec{P}] < \vec{R}$		$\vec{Q} < [\vec{P}] < \vec{R}$	非動作	
16.	$\vec{Q} \geq \vec{P} < \vec{R}$		$\vec{Q} \geq \vec{P} < \vec{R}$	非動作	
17.	$\vec{Q} < [\vec{P}] < \vec{R}$		$\vec{Q} < [\vec{P}] < \vec{R}$	非動作	
18.	$[\vec{Q}] = [\vec{P}] < \vec{R}$		$\vec{Q} < \vec{P} \leq \vec{R}$	-2	減少
19.	$[\vec{Q}] > \vec{P} < \vec{R}$	$\{W(Q) \leq W(R)\}$	$\vec{Q} = \vec{P} \leq \vec{R}$	-1	減少
20.	$[\vec{Q}] > \vec{P} < \vec{R}$	$\{W(Q) > W(R)\}$	$\vec{Q} = [\vec{P}] > \vec{R}$	± 0	減少
21.	$\vec{Q} < [\vec{P}] \geq \vec{R}$		$\vec{Q} < [\vec{P}] \geq \vec{R}$	非動作	
22.	$\vec{Q} \geq \vec{P} \geq \vec{R}$		$\vec{Q} \geq \vec{P} \geq \vec{R}$	非動作	
23.	$\vec{Q} < [\vec{P}] \geq \vec{R}$	$\{W(Q) \leq W(R)\}$	$\vec{Q} < [\vec{P}] \geq \vec{R}$	非動作	
24.	$\vec{Q} < [\vec{P}] \geq \vec{R}$		$\vec{Q} < [\vec{P}] \geq \vec{R}$	非動作	
25.	$[\vec{Q}] = [\vec{P}] \geq \vec{R}$	$\{W(Q) \geq W(R)\}$	$\vec{Q} < [\vec{P}] > \vec{R}$	-1	$O(n)$ 増加
26.	$[\vec{Q}] > \vec{P} \geq \vec{R}$		$\vec{Q} = [\vec{P}] > \vec{R}$	± 0	減少

この補題より、アルゴリズムの実行に従って関数 F が辞書式順序の上で単調減少することを示し、いつかは $F=(0,0)$ になることを言えば、到達可能性が証明できる。

次の補題の証明のために、アルゴリズム A_c の状態遷移表を表1に示す。表中に挙げられた状態が、三つの隣接プロセッサに関する方向、重みのすべての関係である。表中の動作前(後)状態は、隣接する三つのプロセッサ P, Q, R に対して、矢印でそのプロセッサの向きを表し、等号・不等号で内部変数 *weight* の大小関係を表している。また[]で囲まれたものはトークンをもつプロセッサである。

[補題4] $F \neq 0$ であるときネットワーク状況が変化すると F は必ず減少する。すなわち、 $c' = c(\mathcal{A}, S)$ とす

ると、 $F(c) > F(c')$ (不等号は辞書式順序) が必ず成立する。

(証明) 補題3より、 $F \neq 0$ である場合正当な状況ではない。すなわちネットワーク中にトークンが少なくとも一つ以上存在する。このような場合、表1よりトークンが存在するプロセッサの隣接プロセッサのうちいずれか一方が必ず動作する。このとき、いかなる動作(非動作はあり得ない)を行っても、必ず T_f が減少するか、もし増加してもそのときは必ず T_k が減少する。すなわち、辞書式順序の上では、必ず F は減少する。□

補題4より次の補題が成立する。

[補題5] いつかは $F(c) = (0,0)$ になる。□

補題3, 5より次の補題が成立する。

[補題6] アルゴリズム A_c は, \mathcal{L}_A に関して到達可能性を満たす.

\mathcal{L}_A に属する任意のネットワーク状況が ROP の解であること, および補題2, 6より, 次の定理が成立する.
[定理1] アルゴリズム A_c は, C デーモンのもとで任意サイズの均一なリングの方向付け問題を解く自己安定アルゴリズムである. \square

4. む す び

本論文では, 均一なリングネットワーク上で, C デーモンのもとで, 任意サイズのリングネットワークの方向付け問題を解く決定性自己安定アルゴリズム A_c を提案した.

この結果, C デーモンと D デーモンで解ける問題のクラスに真に差があることを初めて示したという意味で興味深い. 更に本論文では文献(6)の証明方法に対し, アルゴリズムの実行に伴って単調減少する関数を定義し, それを用いた自己安定アルゴリズムの正当性の証明を行った.

自己安定アルゴリズムの効率の評価尺度として, 安定するまでのステップ数が考えられるが, 今後の課題として, 本アルゴリズムの効率の評価および, より効率の良いアルゴリズムの開発が挙げられる.

また, 筆者らは文献(6)で, R/W デーモンのもとで, 奇数サイズの均一なリング上で ROP を解く自己安定アルゴリズムも提案している.

謝辞 本研究について適切な御助言, 御協力を頂きました松下電器産業(株)三浦康史氏に深く感謝致します.

文 献

- (1) Israeli A. and Jalfon M. : "Self-stabilizing ring orientation", Proc. of 4th International Workshop on Distributed Algorithms (LNCS 486) pp. 1-13 (1990).
- (2) Israeli A. and Jalfon M. : "Token management schemes and random walks yield self stabilizing mutual exclusion", Proc. of 9th PODC, pp. 119-131 (Aug. 1990).
- (3) Dijkstra E. W. : "Self-stabilizing systems in spite of distributed control", Communications of the ACM, **17**, 11, pp. 643-644 (1974).
- (4) Burns J. E. and Pachl J. : "Uniform self-stabilizing rings", ACM TOPLAS, **11**, 2, pp. 330-344 (1989).
- (5) Israeli A., Dolev S. and Moran S. : "Self stabilization of dynamic systems assuming only read/write atomicity", Proc. of 9th PODC, pp. 103-118 (Aug. 1990).
- (6) 片山喜章, 増澤利光, 都倉信樹 : "リングの方向付け問題を解く自己安定アルゴリズム", 情報学アルゴリズム研報, 92-AL-25-8 (1992-01).

(平成6年3月16日受付)

片山 喜章



平2 阪大・基礎工・情報卒, 平6 同大大学院博士課程中退, 同年奈良先端科学技術大学院大学情報科学研究科助手, 分散アルゴリズムに関する研究などに従事.

増澤 利光



昭57 阪大・基礎工・情報卒, 昭62 同大大学院博士課程了, 工博, 同年同大情報処理教育センター助手, 同大基礎工学部助教授を経て, 平6 奈良先端科学技術大学院大学情報科学研究科助教授, 現在に至る, 平5 コーネル大学客員準教授(文部省在外研究員), 分散アルゴリズム, 並列アルゴリズムに関する研究に従事, ACM, IEEE, 情報処理学会各会員.

都倉 信樹



昭38 阪大・工・電子卒, 昭43 同大大学院博士課程了, 工博, 同年阪大・基礎工・講師, 現在, 同教授(情報工学科), プログラムの技法と理論, 計算機言語, VLSI の計算複雑さの理論などの研究に従事, 本会業績賞(平2年度)受賞, ACM, 情報処理学会, 日本ソフトウェア科学会, 人工知能学会各会員.