木ネットワーク上での自己安定負荷分散アルゴリズムについて

片山喜章 †，増澤利光 ‡，和田幸一 †

Self-Stabilizing Load Balancing Algorighm on Rooted Tree Networks

Yoshiaki KATAYAMA†, Toshimitsu MASUZAWA‡, Kouichi WADA†

† 名古屋工業大学大学院工学研究科情報工学専攻
Department of Computer Science and Engineering,
Graduate School of Engineering, Nagoya Institute of Technology
‡ 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University

あらまし　自己安定アルゴリズムとは，任意のネットワーク状況からアルゴリズムの実行を開始しても，やがてあらかじめ与えられたネットワーク状況に到達する (解を求める) アルゴリズムである．本稿では，木ネットワーク中の各プロセスに任意の負荷 (整数) が与えられると，やがて任意のプロセス間の負荷の差が高々 1 になる自己安定アルゴリズムを提案する．

1　はじめに

複数のプロセスが相互に通信リンクで接続されたネットワークを分散システムという．分散システム上で，プロセスが互いに情報を交換することで与えられた問題を解くアルゴリズムを，分散アルゴリズムという．通常の分散アルゴリズムでは，実行開始時のネットワーク状況 (大域状況) を仮定する．一方，Dijkstra が [1] で提案した自己安定アルゴリズム (self-stabilizing algorithm) は，任意の初期大域状況から実行を開始しても，やがて問題を解く (解状況に到達する) 分散アルゴリズムである．この性質により自己安定アルゴリズムは，プロセスでの変数やプログラムカウンタの破壊，通信メッセージの改変などの一時的な故障 (一時故障:transient fault) が生じても，やがて解状況に到達可能 (再安定) であり，任意の一時故障に対して故障耐性を持つ．そのため，長期間に渡りネットワーク状況を安定に保ち，一時故障に対して対応する必要がある分散システムの運用に適したアルゴリズムである．自己安定アルゴリズムについては文献 [2] に詳しい．

負荷分散 (load balancing) 問題とは，あるタスクが複数のプロセス上で並行に行なわれる際，その負荷 (タスクの量) がすべてのプロセスに対して均等になるよう分散させる問題である．例えば，WWW や電子メールサービスなど，多くの要求が集中するサービスに対して複数のサーバを用意し，要求を各サーバに適切に分散させる場合などに負荷分散技術が用いられている．この例の場合，一旦特定のプロセスがすべての要求を受け付け，それを適切に配分する「集中制御型」で実現することがある．一方，ネットワーク中に存在するプロセスで個別にサービスに対する要求が生じる場合もある．例えば P2P サービスにおける資源の検索要求などがこれにあたる．この例では，各プロセスはサービス要求を出すクライアントであると同時に，サービスを提供するサーバでもあるため，前述の例のような集中制御型は難しく，分散的解決が望まれる．負荷分散問題の分散的解決手法はいくつか知られているが，そのうち自己安定アルゴリズムによる解決は，文献 [3]，[4] などがあげられる．文献 [3] では，任意のネットワーク上で各プロセスの持つ負荷の最大差がネットワークの直径となる負荷分散アルゴリズムが提案されている．また文献 [4] では，根付き木ネットワークにおいて任意のプロセス間の負荷の差が 1 になる負荷分散アルゴリズムが提案されている．

本稿では，根付き木ネットワーク上の各プロセスに与えられる負荷が整数で表現されるとき，任意の初期大域状況からやがて各プロセスの持つ負荷の差が高々 1 になる自己安定アルゴリズムを提案する．提案アルゴリズムは分散的手法で負荷分散を実現しており，すべての要求を一旦集める集中制御プロセスは必要ない．また文献 [4] と比べた場合，[4] の手法では，あるプロセスの先祖 (根に近いプロセス) に負荷の差が 1 の親子組が存在するかどうかという，ある種の大域情報の伝搬を必要としている．一方，提案プロトコルは大域情報を利用することなく，隣接プロセス間の負荷の値だけから動作を決定しており，より局所的な性質を持つといえる．

本稿の以降の構成は以下の通りである．2 章で提案アルゴリズムのモデルについて述べる．続く 3 章で提案アルゴリズムについて説明し，4 章でアルゴリズムの正当性の証

明を行なう．最後に 5 章でまとめと今後の課題について述べる．

2 モデル

本章では，本稿で扱うネットワークやプロトコルなどのモデルについて述べる．

2.1 ネットワークとプロセス

本稿では，n 個のプロセスが通信リンクで互いに接続された，根付き木ネットワーク N を扱う．一般にプロセスと通信リンクからなるネットワークは，プロセスを頂点，通信リンクを辺と見ることで自然にグラフで表現できるので，グラフに対する用語や記法をネットワークに対しても用いる．

ネットワーク N は $N = (P, L)$ で表され，P をプロセス集合，L をプロセス間の通信リンクの集合とする．各プロセスは相異なる識別子を持つものとし，それぞれ $P = \{r, 1, 2, \cdots, n-1\}$ と表し，r のプロセスを根プロセスとする．以下ではプロセスとプロセスの持つ識別子を区別せず，単に識別子で表現する．プロセス i, j 間の通信リンクは $(i, j)(\in L)$ と表し，i, j は互いに隣接しているという．

各プロセスは自分の識別子と隣接プロセスの識別子の集合 N_i，および N_i 中のどれが親プロセス，子プロセス (集合) かを知っているものとし，それぞれ $\mathrm{parent_i}$, $\mathrm{Child_i}(= \{\mathrm{Child_i}(0), \mathrm{Child_i}(1), \cdots\})$ で表す．また，N_i の要素は固定された全順序を持つとする．各プロセス i は (整数) 変数 $\mathrm{load_i}$ を持ち，これを負荷 (load) とよぶ．さらに，各プロセスは根からの距離が偶数 (位置) であるか奇数 (位置) であるかを知っているものとし，奇数位置のとき $\mathrm{oe_i} = \mathrm{e}$，偶数位置のとき $\mathrm{oe_i} = \mathrm{o}$ で表す．

隣接プロセス間の通信は，互いの内部状態 (識別子や内部変数) 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 スケジュールと実行

各プロセス i の状態 (変数などの内部状態) を q_i とするとき，$c = (q_r, q_1, \cdots, q_{n-1})$ を N のネットワーク状況とよぶ．また，N の取り得るすべてのネットワーク状況の集合を C と表す．つまり，プロセス i の取り得るすべての状態を Q_i とするとき，$C = Q_r \times Q_1 \times \cdots \times Q_{n-1}$ となる．

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 $c_i \in C$ で，S に属するプロセスが同時にアルゴリズム $\mathcal{A}$ 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，$c_{i+1} = c_i(S, \mathcal{A})$ と表す．

定義 1 （スケジュールと実行）空でないプロセス集合の無限系列をスケジュールと呼ぶ．アルゴリズム $\mathcal{A}$，スケジュール $T = S(0), S(1), \cdots$ について，ネットワーク状況の無限系列 $E = c_0, c_1, \cdots$ が $c_{i+1} = c_i(S(i), \mathcal{A})(0 \leq i)$ を満たすとき，E を「初期状況 c_0，スケジュール T に対するアルゴリズム $\mathcal{A}$ の実行」とよび，$E(\mathcal{A}, T, c_o)$ と表す． □

定義 2 （公平なスケジュール）スケジュール T にすべてのプロセス $i \in P$ が無限回現れるとき，T は公平であるという． □

本稿では公平なスケジュールのみを対象とし，以降，単にスケジュールと呼ぶ．

スケジュールによって選ばれたプロセスは，1 原子動作のみを行うことができる．スケジュール T が選び出すプロセス数と 1 原子動作の違いによりいくつかのモデルが考えられる．本稿では，以下のモデルを扱う（D デーモンと呼ぶ）．

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

・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

$\mathcal{LE} \subseteq C$ を，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 $\mathcal{LE}$ に関して自己安定である」といい，$SS(\mathcal{A}, \mathcal{LE})$ と書く．また，$SS(\mathcal{A}, \mathcal{LE})$ が成立するとき，$\mathcal{LE}$ を「アルゴリズム $\mathcal{A}$ に関して正当な状況」という．ただし，$\mathcal{A}$ 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 によるアルゴリズム $\mathcal{A}$ の実行 $E(\mathcal{A}, T, c)$ に，$\mathcal{LE}$ に含まれるネットワーク状況が現れる．つまり，$E(\mathcal{A}, T, c) = c_0, c_1, \cdots$ とするとき，$c_i \in \mathcal{LE}$ なる $i \geq 0$ が存在する．

（2）**閉包性**

$\mathcal{LE}$ 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，$c' = c(S, \mathcal{A})$ が $\mathcal{LE}$ に属する．

すなわち，任意の初期状況から開始しても，任意の公平なスケジュールによるアルゴリズムの実行は，有限時間内に正当な状況に到達し，一度正当な状況に達すると，それ以降の状況は正当な状況である（正当な状況で安定する）．

本稿では，負荷が整数の場合に負荷分散問題を解く自己安定アルゴリズムについて述べる．

3 アルゴリズム $\mathcal{LB}$

本章では，負荷が整数の場合に負荷分散問題を解く自己安定アルゴリズム $\mathcal{LB}$ を示す．提案アルゴリズム $\mathcal{LB}$ は，文献 [5] のエージェント巡回自己安定アルゴリズム $\mathcal{HM}$ を利用している．以下では，まず $\mathcal{HM}$ の概略を説明し，続いて提案アルゴリズムのアイデアと詳細を述べる．

3.1 アルゴリズム $\mathcal{HM}$ の概略

文献 [5] のエージェント巡回アルゴリズム $\mathcal{HM}$ は，木ネットワーク上のすべてのプロセスにエージェントが存在する状況で，隣接プロセスとのエージェント交換をラウンドロ

$\mathrm{N_v}[\mathrm{next_v}] = w \ \wedge \ \mathrm{N_w}[\mathrm{next_w}] = v \longrightarrow$
　$\{v$ と w に存在するエージェントを交換
　$\mathrm{next_v} := (\mathrm{next_v} + 1) \bmod \delta_v\}$

図 1: アルゴリズム $\mathcal{HM}$ (v 上)

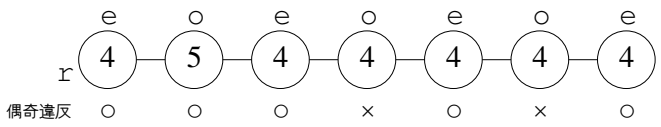

図 2: リニアネットワークでの負荷分散の様子

ビン的に繰り返すことによって，すべてのエージェントに木を巡回させるアルゴリズムである．

アルゴリズムを図 1 に示す．図中の変数の意味は以下の通りである．

- $\mathrm{next_v}$: プロセス v が何番目隣接プロセスとエージェントを交換するかを示すカウンタ値を格納する変数．
- $\mathrm{N_v}[\,]$: プロセス v の持つ隣接プロセス集合 (配列)．隣接プロセス集合の要素は固定された全順序でソートされており $\mathrm{N_v}[1], \mathrm{N_v}[2], \cdots$ で参照可能．
- δ_v: プロセス v の隣接プロセス数 (次数)

アルゴリズム $\mathcal{HM}$ では，ある 1 組のプロセス間でエージェント交換が行なわれる際，それらふたつのプロセスが同時に 1 原子動作を行なうことを仮定していることに注意する[1]．

本稿で提案するアルゴリズム $\mathcal{LB}$ では，アルゴリズム $\mathcal{HM}$ を利用し，互いに隣接したふたつのプロセスでエージェントが交換される部分で，負荷分散のための計算 (負荷計算) を行なうのが基本的なアイデアである．

3.2 アルゴリズム $\mathcal{LB}$ のアイデア

提案アルゴリズム $\mathcal{LB}$ における負荷分散計算のアイデアについて説明する．アルゴリズム $\mathcal{LB}$ においては，集中制御を行なう特別なプロセスの存在を仮定しておらず，隣接プロセス間の局所的な負荷の計算によってネットワーク全体への負荷の均等分散を実現している．

アルゴリズム $\mathcal{LB}$ 実現の主なアイデアは以下の通りである．各プロセス (i とする) は，隣接プロセス中に自プロセスとの負荷の差が 2 以上のものが存在すればそのプロセスとの間で互いに負荷を足して 2 で割ったものを新たな負荷とする．このとき，負荷の和が偶数の場合は両者が同じ負荷を持つ．一方，奇数だと 2 で割り切れない．そこで，商の偶奇および自分の根からの距離の偶奇 ($\mathrm{oe_i}$ の値) から，新しい負荷を決定する．まず,「商」と「商+1」を新たな負荷の値の候補とする (偶数と奇数であることに注意)．もし自分が偶数位置 ($\mathrm{oe_i} = e$) ならば偶数を，奇数位置 ($\mathrm{oe_i} = o$) なら奇数を新たな負荷の値として採用する ($\mathrm{oe_i} = e/o$ なら相手プロセス j では $\mathrm{oe_j} = o/e$ であることに注意)．

しかし，ネットワーク全体の負荷の総和や負荷の初期値によっては，必ずしも偶数／奇数位置のプロセスが偶数／奇数の値を持てるとは限らない．位置の偶奇と負荷の偶奇が一致しない場合を「偶奇違反」と呼ぶことにする．アルゴリズム $\mathcal{LB}$ では，偶奇違反が生じたら，生じたプロセスの子孫すべての負荷の値が等しくなるようにする．こうすることで，根から葉への経路上に出現する負荷の値は，例えば図 2 のようになる．このように負荷を配置することで，負荷の差が 2 以上のプロセスの存在をそれらのプロセスに隣接するプロセスが発見でき，したがって負荷の差を高々1 にすることが可能となる．

[1]本稿での提案アルゴリズムは，この仮定をおかなくても動作する．

i が動作した場合の状態遷移図
(ただし $(x, y) = (flag_i(j), flag_j(i))$ とする)

遷移前	遷移後	説明
(0,0)	(2,0)	計算準備状態へ (HM が実行される)
(0,0)	(0,0)	他のプロセスと計算中の場合 (NOP)
(0,1)	(0,1)	j の終了待ち状態 (NOP)
(0,2)	(2,2)	計算準備完了状態へ (HM が実行される)
(0,2)	(0,2)	他のプロセスと計算中の場合 (NOP)
(1,0)	(0,0)	計算終了へ
(1,1)	(0,1)	j の終了待ち状態へ
(1,2)	(1,2)	j の計算待ち状態 (NOP)
(2,0)	(2,0)	j の計算準備待ち状態 (NOP)
(2,1)	(1,1)	負荷計算後は計算終了状態へ
(2,2)	(1,2)	負荷計算後は計算終了待ち状態へ

表 1: フラグの状態遷移表

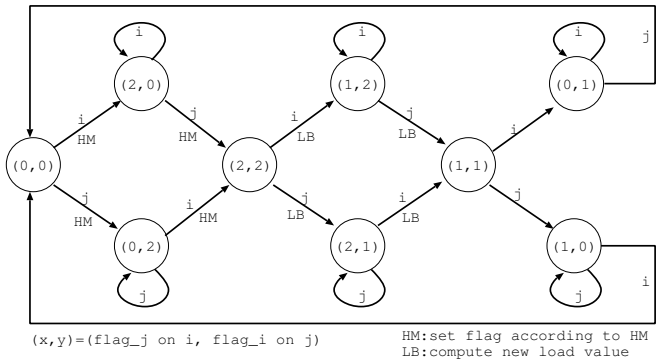

図 3: フラグの状態遷移図

3.3 アルゴリズム $\mathcal{LB}$ の詳細

アルゴリズム $\mathcal{LB}$ における負荷計算は単純である．しかし，負荷計算をを常に一対一のプロセスで行なうためには，複数の隣接プロセスのうちいずれかひとつを選びだし，かつ互いに合意の元 (選びあった状態) で計算しなければならない．そこで，文献 [5] のアルゴリズム $\mathcal{HM}$ を利用する．$\mathcal{HM}$ では，常に一対一でのエージェント交換を可能にしており，この機構を利用することで確実な一対一の負荷計算

を実現する．

具体的に説明する．プロセス i は，$\mathcal{HM}$ のガード (矢印の前部分) が成立した時点で対応する隣接プロセス (j とする) に対して「計算可能フラグ」を立てる．もし，j でも i に対応するフラグ (「計算可能」あるいは「計算終了待ち」) が立っている場合，j との負荷計算を行ない，i は「計算終了待ちフラグ」を立てる．双方が「計算終了待ち」となったところで，フラグを下ろす．これを繰り返すだけである．

i におけるフラグを表す変数を $flag_i(j)(j \in N_i)$ とする．このとき，$flag_i(j)$ の持つ値の意味は以下のいずれかとなる．

- $flag_i(j) = 0$: 他のプロセスで計算中
- $flag_i(j) = 1$: j との計算終了 (計算終了待ちフラグ)
- $flag_i(j) = 2$: j との計算可能 (計算可能フラグ)

i のフラグに注目した状態遷移の様子を表 1，および図 3 にあげておく．

次に，3.2 で述べた負荷の配置を実現する (負荷計算) 方法を述べる．まず負荷計算の種類を定義する．

定義 3 *(*負荷調整と負荷交換*)* 負荷計算は，以下のいずれかである．

- 負荷調整 *:* $lb(i,j)$
  互いに隣接するプロセス i と j の間で負荷の分散を行なう．具体的な手続きは以下の通り．
  - 負荷の和が偶数の場合
    $\mathrm{load_i}$, $\mathrm{load_j}$ に $(\mathrm{load_i} + \mathrm{load_j})/2$ を代入
  - 負荷の和が奇数の場合
    $\lfloor(\mathrm{load_i} + \mathrm{load_j})/2\rfloor$ および $\lfloor(\mathrm{load_i} + \mathrm{load_j})/2\rfloor + 1$ のうち，偶数位置のプロセスに偶数を，奇数位置のプロセスに奇数を代入する．
- 負荷交換 *:* $swap(i,j)$
  対外に隣接するプロセス i と j の間で負荷の交換を行なう．具体的な手続きは以下の通り．
  $\mathrm{load_i} := \mathrm{load_j}$, $\mathrm{load_j} := \mathrm{load_i}$ □

アイデアで述べた負荷の配置を実現するために，以下の 5 つのルールを用いる．ただし，ルールはプロセス i で適用されるもので，i が j と負荷計算を行なうとする．

R1 j が子，かつ j との負荷の差が 2 以上の場合，i と j で負荷調整 ($lb(i,j)$) を行なう．

R2 j が子，かつ自分と負荷の差が 2 以上の隣接プロセスが存在せず，かつ j との負荷の差が 2 以上の隣接プロセスが存在する場合，i と j で負荷交換 ($swap(i,j)$) を行なう．

R3 j が子，かつ自分と負荷の差が 2 以上の隣接プロセスが存在せず，かつ j との負荷の差が 2 以上の隣接プロセスが存在せず，かつ親 p が偶奇違反しておらず，かつ自分 i,j がともに偶奇違反している場合，i と j の間で負荷交換 ($swap(i,j)$) を行なう．

R4 j が子，かつ自分と負荷の差が 2 以上の隣接プロセスが存在せず，かつ j との負荷の差が 2 以上の隣接プロセスが存在せず，かつ親 p が偶奇違反，かつ自分 i,j がともに偶奇違反していない場合，i と j の間で負荷交換 ($swap(i,j)$) を行なう．

R5 j が親の場合，親の指示 ($exec_j(i)$) にしたがって負荷調整か負荷交換を行なう．

厳密に書くと以下の通り．なお $exec_i(j)$ とは，子 j への負荷計算の指示を表す変数とする．lb の場合は負荷調整，$swap$ の場合は負荷交換を行なう指示をしている状態である．また，プロセス i の偶奇違反状態を述語 oe_v_i で表し，偶奇違反していない場合に true，偶奇違反の場合に false であるとする．

R1 $j \in \mathrm{Child_i} \wedge |\mathrm{load_i} - \mathrm{load_j}| \geq 2 \Longrightarrow lb(i,j), exec_i(j) := lb$

R2 $j \in \mathrm{Child_i} \wedge (\forall s \in N_i : |\mathrm{load_i} - \mathrm{load_s}| \leq 1) \wedge (\exists t \in N_i : |\mathrm{load_j} - \mathrm{load_t}| \geq 2) \Longrightarrow swap(i,j), exec_i(j) := swap$

R3 $j \in \mathrm{Child_i} \wedge (\forall s \in N_i : |\mathrm{load_i} - \mathrm{load_s}| \leq 1) \wedge (\forall s,t \in N_i : |\mathrm{load_s} - \mathrm{load_t}| \leq 1) \wedge oe\_v_{\mathrm{parent_i}} \wedge \neg oe\_v_i \wedge \neg oe\_v_j \Longrightarrow swap(i,j), exec_i(j) := swap$

R4 $j \in \mathrm{Child_i} \wedge (\forall s \in N_i : |\mathrm{load_i} - \mathrm{load_s} \leq 1) \wedge (\forall s,t \in N_i : |\mathrm{load_s} - \mathrm{load_t}| \leq 1) \wedge \neg oe\_v_{\mathrm{parent_i}} \wedge oe\_v_i \wedge oe\_v_j \Longrightarrow swap(i,j), exec_i(j) := swap$

R5 $j = \mathrm{parent_i} \wedge exec_j(i) = swap \Longrightarrow swap(i,j)$
または
$j = \mathrm{parent_i} \wedge exec_j(i) = lb \Longrightarrow lb(i,j)$

直観的に R2 は，直接負荷調整が行なえない「負荷の差が 2 以上」のプロセス同士で負荷調整を行なう (R1) ための準備を行なうルールである．R3，R4 は，図 2 のような状態 (○，×の並び) になるよう調整する．これらのルールをすべてのプロセスに適用し続けることにより，やがてアイデアで述べた状況に到達する．

アルゴリズム $\mathcal{LB}$ の詳細を，図 4 にあげる．なお，アルゴリズムの記述において「$A \Longrightarrow B$」は，述語 A が真のときに B を行なうことを意味する．また「状態遷移表にしたがって状態遷移」とは，プロセス i の現在の $\mathrm{flag_i(j)}$ と j の $\mathrm{flag_j(i)}$ の状態を表の「遷移前」と比べ，合致した場合にそのときの状況に応じて「遷移後」と同じ状態になるよう $\mathrm{flag_i(j)}$ の値を変化させることを意味する．

[注]:本アルゴリズムでは，説明を簡単にするために各プロセスが根からの距離の偶奇 (というある種の大域情報) を識別可能としている．しかし，アルゴリズムをこの仮定がなくても正しく動作するように容易に変更できる．これは，偶

奇情報を利用しているルール R3，R4 において，条件が偶奇違反に対して対称になっていること，およびそれらの条件成立時の処理 ($swap(i,j)$) が同じだからである．

4 正当性の証明

本節では，アルゴリズム $\mathcal{LB}$ が負荷分散問題を解く自己安定アルゴリズムであることを証明する．

負荷分散問題を解く自己安定アルゴリズムの正当な状況 $\mathcal{L}_{LB}$ を定義する．

定義 4 （正当な状況）以下のふたつの状況を満たすネットワーク状況を，負荷分散問題を解く自己安定アルゴリズムに関して正当な状況といい，$\mathcal{L}_{LB}$ と表す．

LE1 $\forall i,j : |\mathrm{load_i} - \mathrm{load_j}| \leq 1$
LE2 $\neg oe\_v_i \Longrightarrow \forall j \in i$ の子孫 : $\mathrm{load_i} = \mathrm{load_j}$ □

以後の証明では，アルゴリズム実行中に一時故障が生じないと仮定する．

次の補題は，文献 [5] および状態遷移表より明らかにいえる．

補題 1 すべてのプロセスは，$\mathcal{HM}$ によって順にすべての隣接プロセスとの間でルールの評価が可能となる．またルールの評価が可能となったプロセス対 i,j は，適用できるルールが存在した場合には必ず負荷計算が行なわれ，かつ負荷計算が双方で終了するまで他のプロセスとの間でのルールの評価は行なわない． □

また，各ルールの条件の述語から，以下の補題がいえる．

補題 2 *ルール R1〜R5* のすべてのプロセスでいずれも適用されないネットワーク状況は，正当な状況である．

[証明] 証明略 □

次の補題により，一度正当な状況に到達したら (ネットワーク状況が $\mathcal{L}_{LB}$ を満たす状況になったら)，その後も正当な状況であり続けることがいえる (アルゴリズムの閉包性)．

補題 3 *(閉包性)* 正当な状況では，どのルールも適用されない．

[証明] 証明略 □

さらに以下の補題で，公平なスケジュールによりいつかはすべてのルールが適用できない状況 (正当な状況) に到達することをいう．

補題 4 *ルール R1* は有限回しか適用されない．

[証明] 証明略 □

補題 5 *ルール R2* が適用されると，その後または同時に必ず *R1* が適用される． □

// algorithm $\mathcal{LB}$ for process i
//START //$\mathcal{HM}$
HM:
$\mathrm{N_i[next_i]} = j \wedge \mathrm{N_j[next_j]} = i \ \rightarrow$
  {$\mathrm{flag_i(j)} := 2$
  goto **STRANS**
  HM_CONT:
  $\mathrm{next_i} := (\mathrm{next_i} + 1) \bmod \delta_i$(次数)}
goto **HM**
//STRANS
STRANS:
"状態遷移表にしたがって $flag$ を状態遷移"
$\mathrm{work_i} := \mathrm{load_i}$
$(\mathrm{flag_i(j)}, \mathrm{flag_j(i)}) = (2,2) \vee (2,1) \rightarrow$
  {$exec_i :=$ NULL
  goto **LB**}
$(\mathrm{flag_i(j)}, \mathrm{flag_j(i)}) = (0,0) \vee (0,2) \rightarrow$
  goto **HM_CONT**
goto **STRANS**
//$\mathcal{LB}$
LB:
R1; $exec_i := lb$
R2; $exec_i := swap$
R3; $exec_i := swap$
R4; $exec_i := swap$
R5
goto **STRANS**
//END

//サブルーチン
//$swap(i,q)$
$swap(i,q)$: $\mathrm{load_i} := \mathrm{work_q}$

//$lb(i,q)$
$lb(i,q)$:
  $sum := \mathrm{load_i} + work_j$
  $lnew := \lfloor \mathrm{load_i}/2 \rfloor$
  sum が偶数 $\rightarrow load_i := lnew$
  $lnew$ が奇数 $\wedge\ oe_i = e \rightarrow \mathrm{load_i} := lnew + 1$
  $lnew$ が奇数 $\wedge\ oe_i = o \rightarrow \mathrm{load_i} := lnew$
  $lnew$ が偶数 $\wedge\ oe_i = e \rightarrow \mathrm{load_i} := lnew$
  $lnew$ が偶数 $\wedge\ oe_i = o \rightarrow \mathrm{load_i} := lnew + 1$

図 4: アルゴリズム $\mathcal{LB}$

[証明] 証明略 □

補題 4，5 より，以下の補題がいえる．

補題 6 アルゴリズムの実行において，ある時点以降，ルール *R1*，*R2* が適用されなくなる． □

以降，ルール R1,R2 が適用されなくなったと仮定する．

補題 7 アルゴリズムの実行において，ある時点以降，ルール *R3*，*R4* が適用されなくなる．

[証明] 証明略 □

補題 6，7 より，以下の補題 (到達可能性) がいえる．

補題 8 *(*到達可能性*)* アルゴリズムの実行において，いずれのルールも適用されない状況に到達する．

[証明] ルール R1〜R4 が適用されないと，R5 も適用されないのは明らか． □

以上より以下の定理を得る．

定理 1 アルゴリズム $\mathcal{LB}$ は，任意のプロセスの持つ負荷の差が高々*1* になる，負荷分散問題を解く自己安定アルゴリズムである． □

5 まとめと今後の課題

本稿では，負荷の値が整数で表される場合に，任意のふたつのプロセスが持つ負荷の値の差が高々1 となるような，負荷分散問題を解く自己安定アルゴリズムを提案した．

今後の課題としては，アルゴリズムの解析（安定時間）があげられる．

謝辞

本研究の一部は，日本学術振興会科学研究費補助金・若手 (B)(課題番号 16700010) および基盤研究 (B)(2)15300017 の研究助成によるものである．

参考文献

[1] E.W.Dijkstra. “Self-stabilizing systems in spite of distributed control” *CACM*, *17(11): 643–644*, 1974.

[2] S.Dolev. “Self-Stabilization” “MIT Press”, 2000.

[3] W.Aiello, B.Awerbuch, B.Maggs and S.Rao. “Approximate load balancing on dynamic and asynchronous networks” *Proc. of the 25th Annual ACM Sympo. on Theory of Computing: 632–641*, 1993.

[4] A.Arora. “A foundation of fault-tolerant computing(Chapter 7)” *Ph.D. Dissertation, University of Texas at Austin*, 1992.

[5] T.Herman, T.Masuzawa. “Self-Stabilizing Agent Traversal” *Proc. of 5th Int. Workshop on Self-Stabilizing Sys. (WSS2001) (LNCS2194): 152–166*, 2001.