自己安定分散コンピューターシステムに関する研究

代表研究者　角川裕次　広島大学 情報教育研究センター 講師
共同研究者　水野 匡章　カンサス州立大学 計算機情報科学科 助教授

1　研究目的

従来では大型計算機による一括処理を行なっていたものが、高性能な計算機と高速計算機ネットワークを使用した分散処理へと、システム形態の変化が急激に進行している。ハードウエア自体は高性能、低価格化が進んではいるが、ネットワーク向けソフトウエアの開発は困難である。なぜなら、分散システム (複数の計算機とそれらをつなぐネットワークより成るシステム) を構成している計算機は、他の計算機と通信をしながら処理を進めて行く形態のプログラム (分散プログラム) を開発する必要がある。さらに、複数の計算機が互いに通信をしながら処理を進める際に、通信のエラーや一部の計算機の一時的な故障は比較的良く起きるので、故障の対策も必要である。そのため、従来の逐次処理プログラムと比べると、分散アルゴリズムの設計や分散プログラムの開発は困難である。大型計算機で処理を行なっていた場合は、計算機一台のみが動いていればよく、そのような問題はない。

本研究では、分散システムの耐故障性に関する研究で、とくに自己安定 (self-stabilization) についての研究である。自己安定を抽象的に説明すれば、システムがいかなる非安定状態になっても、システム自ら安定状態に遷移する仕組みを持つものである。これを計算機ネットワークの立場から言い替えると、次のようになる: 自己安定システムとは、故障が生じてもその原因に関係なく、システム自体が自動的に元の正常な状態に修復するアルゴリズム、あるいは通信プロトコルである。システムを構成しているコンピューターそれぞれが互いに協調し、正常なシステム状態へ自律的に復帰をする。

故障の型にはいろいろなものがあるが、任意の一時故障 (一時的に機能が停止するが、しばらくしたら再び元の機能を回復する型の故障) を想定する。この型の故障には、停電による計算機の再起動 (それに伴ってメモリ内容が失われる)、ノイズなどによる伝送メッセージの消失などが挙げられる。自己安定システムは、任意の種類の一時故障が任意の数生じても、(再び一時故障が生じない限り) 正常なシステム状態に自動復帰する。このような性質があるため、分散システムが自己安定性を持つことは重要である。

ところが、故障を想定しない分散アルゴリズムの設計自体が難しいのに、さらに自己安定性を持たせるのはさらに難しくなる。このことを解決するため、分散アルゴリズムの設計が容易となるよう、アルゴリズム変換を導入する。以下が、アルゴリズム変換を使っての、自己安定分散アルゴリズムの設計手順である。我々が対象とする計算モデルに、逐次実行モデルに似たものを選び、分散性によるアルゴリズム設計の難しさを減らす。そしてそのモデルで、自己安定分散アルゴリズムを設計し、検証をする。最後に、アルゴリズム変換を用いてアルゴリズムの変換を行い、現実の分散システムのモデルで動作するものを得る。アルゴリズム変換は、逐次モデルでのアルゴリズムの動作や正当性を保証するよう、分散モデルで動作するアルゴリズムに変換する必要がある。

そのようなアルゴリズム変換に関する研究は [6] や [5] に見られる。[6] で提案されている手法は完全に自動化されておらず、元となるアルゴリズムを熟知した人による手作業が変換の際に必要となる。また [5] での手法は変換は完全に自動化されているものの、変換によって得られたアルゴリズムは、必ずしも正しさが保証さ

れていない。いずれの手法も、変換後のアルゴリズムの正当性の証明を人間が行なわなくてはならないが、これは困難な仕事である。というのも、アルゴリズム設計段階は抽象度の高いモデルであっても、正当性の証明は抽象度の低い、実際の分散システムのモデルで行なう必要があるからである。本稿で提案する手法は、元のアルゴリズムが正しければ変換後も正しさを保証する。従って変換後の検証は必要ない。

本報告では、アルゴリズム変換と、その評価について報告する。第 2節では、分散アルゴリズム設計で用いる計算モデルを説明する。第 4節では、アルゴリズム変換を提案する。第 5節では、計算機シミュレーションによる評価結果を示す。最後に第 6節では、本稿のまとめをおこなう。

2 計算モデル

この節では、計算モデルを説明する。逐次モデルは、アルゴリズム記述の段階で使用するモデルであり、一方分散モデルは現実の分散システムをモデル化したものである。以降では、分散システムを構成する計算機をプロセスと呼ぶ。分散モデルは、逐次モデルで記述された分散アルゴリズム (または分散システム) の動作を模倣するものである。

2.1 逐次モデル

逐次モデルは、以下の性質をもつ。

- プロセス間の通信は、メッセージによる通信ではなく、他のプロセスの状態を直接参照することで行なう。状態の参照は、時間遅れなく瞬時に行なわれる。(この通信のモデルは、状態通信モデルと呼ばれる。)
- 並行実行はなく、プロセスの内のどれかひとつのみが選ばれ、実行される。プロセスの実行は、他プロセスの状態の参照と状態の遷移とを一単位とし、不可分に実行される。(この実行スケジューリングのモデルは、C デーモンと呼ばれる。)

逐次モデルでのプログラムは、5 つ組 $S_s = (n, \mathcal{P}_s, \mathcal{R}_s, \mathcal{Q}_s, \mathcal{A}_s)$ で表現される。ここで、

1. n は分散システム S_s でのプロセス数である。
2. $\mathcal{P}_s = \{\rho_1, \rho_2, ..., \rho_n\}$ は、プロセス集合である。
3. $\mathcal{R}_s \subseteq \mathcal{P}_s \times \mathcal{P}_s$ は、依存関係を表す。もし ρ_j が ρ_i の状態を参照するのであれば、$(\rho_i, \rho_j) \in \mathcal{R}_s$ である。

   もし $(\rho_i, \rho_j) \in \mathcal{R}_s$ であるとき、ρ_i は ρ_j の書き込みプロセスであるといい、そして ρ_j は ρ_i の読み出しプロセスであるという。$Writers(\rho_i)$ は ρ_i の書き込みプロセスすべての集合とし、$Readers(\rho_i)$ は ρ_i の読み出しプロセスすべての集合とする。$(\rho_i, \rho_j) \in \mathcal{R}_s$ または $(\rho_j, \rho_i) \in \mathcal{R}_s$ であるとき、ρ_i と ρ_j は隣接しているという。
4. $\mathcal{Q}_s = \{Q_{s_1}, Q_{s_2}, ..., Q_{s_n}\}$ はプロセス状態の組であり、Q_{s_i} は ρ_i が取り得る状態のすべての集合である。

   プロセス ρ_i は、変数 q_i に状態を保持するものとする。S_s での大局状態 (global state) は、各プロセスの状態の組 $(q_1, q_2, ..., q_n)$ で表される。
5. $\mathcal{A}_s = \{A_{s_1}, A_{s_2}, ..., A_{s_n}\}$ は、アルゴリズムの n 個組で、A_{s_i} は ρ_i が実行するアルゴリズムである。各 A_{s_i} は、以下の形式をしている。

$$
\begin{array}{l}
*[\\
\quad g_{i,1}(q_i, q_{i,1}, q_{i,2}, ..., q_{i,k}) \\
\qquad \rightarrow q_i := f_{i,1}(q_i, q_{i,1}, q_{i,2}, ..., q_{i,k}) \\
\quad [] \\
\qquad \vdots \\
\quad [] \; g_{i,m}(qi, q_{i,1}, q_{i,2}, ..., q_{i,k}) \\
\qquad \rightarrow q_i := f_{i,m}(qi, q_{i,1}, q_{i,2}, ..., q_{i,k}) \\
]
\end{array}
$$

各記号は以下の通りである。

- $Writers(\rho_i) = \{\rho_{i,1}, ..., \rho_{i,k}\}$ である。なお、$\rho_{i,j}$ は ρ_i の j 番目の書き込みプロセスである。
- 各 $g_{i,j}(\cdots) \rightarrow q_i := f_{i,j}(\cdots)$ は、ガード付コマンド (guarded command) という。ただし、$g_{i,j} \rightarrow \cdots$ は、A_{s_i} の j 番目のカード付コマンドである。
- $g_{i,j}$ は、$Q_i \times Q_{i,1} \times \cdots \times Q_{i,k} \rightarrow boolean$ である述語である。
- $f_{i,j}$ は、$Q_i \times Q_{i,1} \times \cdots \times Q_{i,k} \rightarrow Q_i$ である関数である。$q_i := f_{i,j}(\cdots)$ はコマンドと呼び、$f_{i,j}$ はコマンド関数である。

もしプロセス ρ_i が真であるガード $g_{i,j}$ を持っていれば、ρ_i は特権を持っているという。(複数のプロセスが同時に特権を持つこともある。) 上記の形で記述されたプログラムの実行は、以下の無限の繰り返しである。

1. 各プロセスは、すべてのガードを評価する。
2. C デーモンと呼ばれる集中的実行スケジューラーが、特権を持つプロセス (ρ_iとする) を非決定的にひとつ選択する。
3. プロセス ρ_i は、真であるガードから一つを非決定的に選び、実行する。

2.2 分散モデル

分散モデルは、以下の性質をもつ。

- メッセージ通信によって、プロセスは互いに情報のやりとりをする。
- 送られたメッセージが届くまでには時間遅れがあり、しかも一定とは限らない。
- すべてのプロセスは並行して動作し、動作速度もまばらである。

以下に、逐次モデルで記述されたプログラムを模倣する、分散モデルでのプログラムを示す。ここではモデルだけを説明し、具体的な変換については後の節で示す。

分散モデルでのプログラムは、6 つ組 $S_d = (n, \mathcal{P}_d, \mathcal{R}_d, \mathcal{Q}_d, \mathcal{C}_d, \mathcal{A}_d)$ で表現される。ここで、

1. n は分散システム S_d でのプロセス数である。
2. $\mathcal{P}_d = \{P_1, P_2, ..., P_n\}$ は、プロセス集合である。P_i は S_s でのプロセス ρ_i を動作を模倣する。
3. $\mathcal{R}_d \subseteq \mathcal{P}_d \times \mathcal{P}_d$ は、依存関係を表す。もし P_j が P_i の状態を参照するのであれば、$(P_i, P_j) \in \mathcal{R}_d$ である。

   書き込みプロセス、読み出しプロセス、そして記法 $Writers(P_i)$ と $Readers(P_i)$ は、S_s での場合と同様に定義する。
4. $\mathcal{Q}_d = \{Q_{d_1}, Q_{d_2}, ..., Q_{d_n}\}$ はプロセス状態の組であり、Q_{d_i} は P_i がとり得る状態のすべての集合であり、P_i の局所変数すべてのデカルト積である。P_i は ρ_i の動作を模倣するので、局所変数のひとつは q_i である。その他の局所変数は S_s を模倣するために内部で使われる作業用変数であり、外部からの観測者にとっては q_i の変化のみが重要である。

   S_d の模倣大局状態 (simulated global state) を、$(q_1, q_2, ..., q_n)$ とする。
5. $\mathcal{C}_d = \{..., C_{d_{i,j}}, ...\}$ は、P_i から P_j への通信リンクが取り得る状態の集合である。$C_{d_{i,j}}$ は、P_i から P_j へのメッセージの列として表される。

   S_d での実行の大局状態 (global state) を、各プロセスの状態と各通信リンクの状態の組で表す。
6. $\mathcal{A}_d = \{A_{d_1}, A_{d_2}, ..., A_{d_n}\}$ は、アルゴリズムの n 個組で、A_{d_i} は P_i が実行するアルゴリズムである。各 A_{d_i} は、S_s での場合と同様に、ガード付コマンドの形式をしている。

   真であるガードがあれば、それに対応したコマンドが実行される。実行スケジューラーは集中的なものでなく、各プロセス独立に実行が行なわれる点が、逐次モデルと違う点である。さらに、S_d の各プロセス P_i は他のプロセスの状態を直接見ることがで

きず、P_i の局所変数しか参照できない。アルゴリズム $A_{d,i}$ の記述は、他プロセスからのメッセージの受信、他プロセスへのメッセージ送信のための動作も含む。

3 自己安定アルゴリズム

アルゴリズム変換を説明する前に、自己安定アルゴリズムについて述べる。逐次モデルでのプログラム S_s が大局状態についての述語 $\mathcal{T}$ に関して自己安定 (self-stabilizing) であるとは、以下の条件を満たすときである。

1. 閉包性: 述語 $\mathcal{T}$ が S_s の実行に関して閉じている。すなわち、ひとたび $\mathcal{T}$ が真になれば、(一時故障が生じない限り) 以降の以下なる実行においても $\mathcal{T}$ は偽にならない。
2. 収束性: 任意の初期状態から実行を開始しても、S_s は有限回の実行ステップで $\mathcal{T}$ が真となる大局状態になることが保証されている。

述語 $\mathcal{T}$ を満たす大局状態を正当な大局状態 (legitimate states) といい、正当な大局状態のすべての集合を $\mathcal{L}_s$ と表す。

分散モデルでのプログラム S_d の自己安定性についても、同様に定義される。

図 1は、前川によって提案された分散相互排除アルゴリズム [10] に自己安定性の拡張を行ない、逐次モデルで記述したものである。元のアルゴリズムは分散モデルで設計されていて、故障は全く生じない仮定の上にアルゴリズムが設計されていた。無故障を仮定していても、アルゴリズムは複雑で正当性の証明が複雑であった。しかし逐次モデルで記述すると正当性の証明は容易であり、なおかつ容易に自己安定性を持つように改良することができた。このようなことから、分散モデルでは複雑となる分散アルゴリズムでも、逐次モデルでは設計と正当性の証明が容易となることが分かる。

4 アルゴリズム変換

逐次モデルで記述されたアルゴリズムを、分散モデルで動作させるためのアルゴリズム変換法を図 2に示す。図 2は、ある逐次モデルで記述されたアルゴリズム S_s を模倣する分散モデルのプログラム S_d となっている。S_s を模倣するに当たり、以下が重要なポイントである。

- S_d での各プロセス P_i は、 S_s での各プロセス ρ_i の動作を模倣する。(図 2の行 A02)
- S_s での状態通信モデルの実現

  — S_d の各プロセス P_i は、隣接するプロセス P_j が模倣する S_s でのプロセス ρ_j の状態 q_j を、局所変数 $nq_i[j]$ に保持する。これにより 各 P_i は、隣接プロセス P_j が模倣するプロセス ρ_j の状態 q_j を、メッセージ通信を使うことなく局所変数 $nq_i[j]$ で参照できる。

  各 P_i は、それが模倣している ρ_i の状態 q_i が変われば、隣接プロセス P_j に新しい状態を送る。(図 2の行 B06) これにより $nq_j[i]$ が更新される。(図 2の行 E01 – E05)
- 自己安定な状態通信モデルの実現の必要性

  — 状態通信モデルを実現するにあたり、伝達途中のメッセージが消失するなど、一時故障が生じる場合がある。このため、状態通信モデルの実現そのものも、自己安定でなければならない。一定期間 τ ごとに、各プロセス P_i は状態 q_i を隣接プロセスに送る。(図 2の行 H01 – H03)
- C デーモンと等価な実行スケジュールの実現

  — 分散モデルでは、各プロセスの実行を制御する機構がなく、各プロセスは完全に並列に動作をする。このようなモデルの上で、C デーモンと等価な実行を実現する必要がある。データベースにおけるトランザクション実行の直列化 (serialization) の概念を使って解決する。つまり、各 ρ_i の状態である q_i をデータ項目とみなし、q_i の更新をトランザクションとみなす。分散環境で直列化を行なうため、各トランザクションに対して時刻印 (timestamp) を付ける。(図 2の行 A03) あるトランザクションの実行が直列化可能であればコミットされる

```
1   Process ρ_i
2   const
3     R_i       /* ρ_i's request set */
4     A_i       /* ρ_i's acquired set A_i ≡ {ρ_j | ρ_i ∈ R_j} */

5   var
6     A_request : bool
7     mode : {idle, waiting, locked}
8     ts : timestamp
9     who : process_id /* a process that obtained ρ_i's lock */

10  *[
11    A_request ∧ (mode = idle) ∧ (∀ρ_j ∈ R_i  ρ_j.who ≠ ρ_i) →
12          ts:= new_timestamp()
13          mode := waiting

14    [] mode = waiting ∧ (∀ρ_j ∈ R_i[ρ_j.who = ρ_i]) →
15          mode := locked

16    [] ¬A_request ∧ (mode ≠ idle) →
17          mode := idle

18    [](who = ⊥ ∨ (who = ρ_j ∧ ρ_j.mode ≠ locked))∧
      (∃ρ_k ∈ A_i such that who ≠ ρ_k and
      ρ_k.ts = min{ρ_l.ts | ∀ρ_l ∈ A_i : ρ_l.mode ≠ idle}) →
19          who := ρ_k

20    [] who = ρ_j ∧ ρ_j.mode = idle →
21          who := ⊥
22  ]
```

図 1: 自己安定な拡張を行なった前川の分散相互排除アルゴリズム (逐次モデル)

```
    *[
A01     (mode = NO_PRIV) ∧ (∃j : g_ij(...) = true) →
A02       q_tmp_i := f_ij(...);  mode_i := PRIV;  granted_i := ∅;  req_ts_i := max_ts_i;
A03       multicast req_commit(req_ts_i) to all P_j ∈ Writers(P_i)
A04       commit_timer.start_timer();

B01     [] receive grant(ts_j, ts'_i) from P_j →
B02       if (mode_i = PRIV) ∧ (req_ts_i = ts'_i)
B03         granted_i := granted_i ∪ {j};
B04         if (granted_i = Writers(P_i))
B05           q_i := q_tmp_i;  mode_i := NO_PRIV;
B06           multicast commit(req_ts_i, q_i) to all q_j ∈ Readers(P_i);
B07           state_timer.start_timer();
B08           update_timestamps ();    /* max_ts_i := req_ts_i := max_ts_i + delta */
B09         fi
B10       fi

C01     [] receive abort(ts_j, ts'_i) from P_j →
C02       if (ts'_i = req_ts_i)  mode_i := NO_PRIV;  fi

D01     [] receive commit(ts_j, q_j) from P_j →
D02       nq_i[j] := q_j;  mode_i := NO_PRIV;
D03       update_timestamps ();    /* max_ts_i := req_ts_i := max_ts_i + delta */

E01     [] receive state(ts_j, q_j) from P_j →
E02       if (q_j ≠ nq_i[j])
E03         nq_i[j] := q_j;  mode_i := NO_PRIV;
E04         update_timestamps ();    /* max_ts_i := req_ts_i := max_ts_i + delta */
E05       fi

F01     [] receive req_commit(ts_j) from P_j →
F02       if (mode_i = NO_PRIV) ∨ (ts_j < req_ts_i)    /* use PIDs to break a tie */
F03         send grant(req_ts_i, ts_j) to P_j;
F04       else
F05         send abort(req_ts_i, ts_j) to P_j;
F06       fi

G01     [] commit_timer goes off →
G02       mode_i := NO_PRIV;  commit_timer.start_timer();

H01     [] state_timer goes off →
H02       multicast state(req_ts_i, q_i) to all P_j ∈ Readers(P_i);
H03       state_timer.start_timer();
    ]
```

図 2: アルゴリズム変換

が、そうでなければアボートされ、トランザクションをやりなおす。(図 2の行 C01 – C02, D01 – D03, F01 – F06)

トランザクションの実行は完全に分散しており、直列化の制御自体も自己安定化して、メッセージの消失などの一時故障からの復帰ができなければならない。プロセスがコミット要求をする時に、一定時間経っても返事がなければアボートされる。この実現には、タイマー commit_timer を使い、t_c 時間経てばタイマー切れが生じてアボート動作を行なうようにしてある。(図 2の行 A04, G01 – G02)

S_d での大局状態に関する述語 $\mathcal{T}$ を、以下のように定義する。

定義 **1** *1.* 計算開始時に存在した不要メッセージが通信リンクに存在せず、かつ

2. すべてのプロセス P_i に関して、以下のいずれかひとつが成立する。

(a) $mode$ が NO_PRIV であり、かつ全ての $P_j \in Writers(P_i)$ に対して

i. $q_j = nq_i[j]$ であり $commit$ メッセージが P_j から P_i へ伝達途中でない、または、

ii. q_j を運んでいる $commit$ メッセージが P_j から P_i へ伝達途中である、あるいは、

(b) $mode$ が $PRIV$ であり、P_i は今の状態になる前には NO_PRIV であり、そしてそのとき上記の *i.* または *ii.* が成立していた。

$\mathcal{T}$ に関して以下の補題が成立する。この補題は、不要なメッセージは取り除かれ、あらたに不要なメッセージが生じないことを示す。

補題 **1** S_d は $\mathcal{T}$ に関して自己安定である。

S_d での実行開始直後は、通信リンクに不要なメッセージが残っていたり、プロセスの局所変数がどのようような値を保持しているか保証されていない。そのため、それらが正常な状態に移行するのにいくらかの時間がかかる。S_d での計算開始からの有限ステップの間では S_s の模倣はできないことがあるが、それ以降は S_s の模倣が行なわれる。これに関して、以下の定理が成立する。

定理 **1** S_d での任意の実行は、最初の有限ステップを除いて、直列化可能な実行である。

この定理は、S_d での任意の実行 (最初の有限ステップを除く) は、 S_s でのある実行に対応することを示している。どのような実行に対しても S_s での大局状態が正当な大局状態になることが保証されていれば、S_d での大局状態も正当となる。従って、図 2の変換は、逐次モデルでのプログラムの正当性を保証したまま、分散モデルでのプログラムに変換をする。

5 評価

本節では、計算機シュミレーションによる変換法の評価を行なう。ここでは、ダイクストラによって提案された分散相互排除を行なう K状態アルゴリズムを用いる [3]。これは n 個のプロセスより成る単方向リングネットワークのためのアルゴリズムで、図 3の通りである。このアルゴリズムは、 n 個の内のいずれかひとつのプロセスが特権を持ち、特権を持つプロセスがリングに沿って巡回する形の動作をする。プロセスが特権を持っているときに、そのプロセスが臨界領域 (critical section) に入るものとする。

S_d の計算開始直後の状態では、各プロセスの局所変数の値や通信リンクの内容は不定である。そのため、それらの状態が正常な状態に安定させる (述語 $\mathcal{T}$ が真になる) ためには、各プロセスの状態を隣接プロセスに知らせることが必要なため、タイムアウト間隔 τ は短いほうが良い。しかし、ひとたび $\mathcal{T}$ が真になると、メッセージのやりとりは各プロセスの状態の変化があった時だけで十分であり、タイムアウト間隔 τ は短いと、不必要に多くのメッセージが送信されることになる。このため、τ の選び方は重要である。

一般的なメッセージの複雑度について考える。プロセス P_i の書き込みおよび呼び出しプロセ

- 各 $1 \leq i \leq n$ に対して、$(\rho_i, \rho_{((i \bmod n)+1)}) \in \mathcal{R}_s$ である。
- 各 $1 \leq i \leq n$ に対して $Qs_i = \{1, 2, \cdots, K\}$ である。(ただし $K > n$ である。)
- $As_1 : *[q_1 = q_n \rightarrow q_1 := (q_1 \bmod K)+1]$ であり、$2 \leq i \leq n$ に対して $As_i : *[q_i \neq q_{i-1} \rightarrow q_i := q_{i-1}]$ である。
- $\mathcal{T}$ は、ちょうどひとつのプロセスが特権を持つとき、およびその時に限って真である。

図 3: Dijkstra による自己安定分散相互排除アルゴリズム (逐次モデル)

スの数を、それぞれ w_i と r_i とする。P_i でのコミットされるトランザクションの実行では、一回当たり $2w_i + r_i$ 個のメッセージが必要となる。(内訳は、req_commit メッセージが w_i 個、grant メッセージが w_i個、そして最後に commit メッセージが r_i個である。) アボートされるトランザクションの実行では、一回当たり最高 $2w_i$ のメッセージが必要である。(内訳は、req_commit メッセージが w_i個、grant メッセージが $w_i - 1$ 個、abort メッセージが 1 個である。)

ダイクストラの K状態アルゴリズムでは、各プロセス P_i に対して $w_i = r_i = 1$ である。ひとたびシステムが安定すれば、特権を持つプロセスは一つだけであり、トランザクションは必ずコミットされ、アボートは起きない。従って、タイムアウトによる状態の通知が起きない限り、プロセスが臨界領域に進むために 3 個のメッセージが送信される。

計算機シミュレーションは、以下の条件で行なった。(最後に*が付いているものは、その値が期待値であるような指数分布に従う乱数である。)

- プロセス数 n: 6
- ネットワークの伝送遅延: 2 単位時間*
- プロセスが臨界領域に入っている時間 MXTIME: 10 単位時間*
- 各ガードの実行時間: 0.2 単位時間*
- commit_timer の初期値 t_c: 10.0

シミュレーションは、以下の τ の値に対して行なった。(1) MXTIME (= 10) (2) MXTIME$\cdot(n-1)$ (= 50) (3) MXTIME$\cdot(n-1)\cdot 5$ (= 250) 表 1は、5000 単位時間にわたるシミュレーションでの、各プロセスが特権を得るのに要したメッセージ数の期待値を示す。

τ の値を大きく (MXTIME$\cdot(n-1)\cdot 5$) しても、各プロセスが特権を持つのに要するメッセージ数は 3 よりもわずかに大きいだけである。このことは、以下の理由による。

1. 述語 $\mathcal{T}$ が成立するまでの、実行の最初の段階で多くのメッセージが送られる。
2. タイマー commit_timer が切れ、アボート動作が数度生じる。

6 まとめ

本稿では、逐次モデルで記述された自己安定アルゴリズムを、実際のネットワークコンピューターシステムのモデルである分散モデルで動作させるための変換方法について議論した。ここで示した変換法は時刻印に基づいたものであり、トランザクションの分散実行の形をしている。そしてトランザクションの分散実行自体も自己安定であり、変換されたアルゴリズムもまた自己安定であることが保証される。

参考文献

[1] Bernstein, P.A, Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Co., 1987.

[2] Burns J.E. and Pachl J. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11(2):330–344, April 1989.

表 1: メッセージ数の期待値

τ value	10	50	250
messages/entry	12.84	6.01	3.27

[3] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[4] Gouda M.G. and Multari M.J. Stabilizing communication protocols. *IEEE Tran. on Computers*, 40(4):448–458, 1991.

[5] S.-T. Huang, L.-C. Wuu, and M.-S. Tsai. Distributed execution model for self-stabilizing systems. In *Proceedings of the International Conference of Distributed Computing Systems (ICDCS '96)*, pages 432–439, 1994.

[6] Katz S. and Perry K.J. Self-stabilizing extensions for message passing systems. *Distributed Computing*, 7:17–26, 1993.

[7] T. Kameda and M. Yamashita. *Distributed Algorithms*. Kindai Kagakusya, 1995. (in Japanese).

[8] J. L. W. Kessels. An exercise in proving self-stabilization with a variant function. *Information Processing Letters*, 29(1):39–42, September 1988.

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[10] M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, March 1985.

[11] M. Mizuno. A distributed/parallel algorithm development system. Kansas State University, 1994. (software).

[12] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer network. *Communications of the ACM*, 24(1):9–17, January 1981.

[13] Schneider M. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.

発表資料

1. Masaaki Mizuno and Hirotsugu Kakugawa, "A Timestamp Based Transformation of Self-Stabilizing Programs for Distributed Computing Environments", 10th International Workshop on Distributed Algorithms (WDAG '96), in *Lecture Notes in Computer Science 1151*, p.304–321, Bologna, Italy, 1996 October.

2. Masaaki Mizuno, Mikhail Nesterenko, and Hirotsugu Kakugawa, "Lock Based Self-Stabilizing Distributed Mutual Exclusion Algorithms," in the Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS), p.708–716, Hong Kong, 1996 May.

3. Hirotsugu Kakugawa and Masafumi Yamashita, "Uniform and Selfstabilizing Token Rings Allowing Unfair Daemon," IEEE Transactions on Parallel & Distributed Systems, Vol. 8 No. 2, p.154–163, Feb 1997.