論　文

重み最小生成木を構成する故障封じ込め自己安定プロトコル

片山　喜章†　　増澤　利光††

A Fault-Containing Self-Stabilizing Protocol for Constructing a Minimum Spanning Tree

Yoshiaki KATAYAMA† and Toshimitsu MASUZAWA††

あらまし　自己安定プロトコルとは，任意のネットワーク状況から実行を開始しても，やがて解を求めて安定するプロトコルである．また，故障封じ込めとは，解状況から少数のプロセスが故障した場合に，再び解状況に到達するまでに状態遷移するプロセス数及び時間を限定したものである．これは，一般に起こりやすいと考えられる小規模な故障による故障状況からの実行において，故障の及ぶ範囲を制限しかつ素早く再安定することを目的としている．本論文では，重み最小生成木（MST）を構成する故障封じ込め自己安定プロトコルを提案する．

キーワード　自己安定プロトコル，故障封じ込め，重み最小生成木

1. まえがき

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（分散システム）において，通常の分散プロトコル（アルゴリズム）は，プロトコルの実行開始時のネットワーク（大域）状況を仮定する．一方，Dijkstra [1] が初めて提案した自己安定プロトコル（self-stabilizing protocol）とは，任意の初期ネットワーク状況から実行を始めても問題を解くことができる（解状況に到達する）分散プロトコルである．この性質から自己安定プロトコルは，プロセスでのプログラムカウンタや変数の破壊，通信メッセージの改変など，一時的な故障（一時故障）によって解状況から任意の状況に陥っても，再び解状況に戻り，再安定する．つまり，一時故障に対して故障耐性をもつ分散プロトコルであり，長期間ネットワーク状況を安定に保ち，一時故障に柔軟に対応する必要がある分散システムを動作させるのに適しており，現在盛んに研究されている分野の一つである．

分散システム上で分散プロトコルを長期間運用する場合，同時に多数のプロセスが故障することはまれで，一般には少数のプロセスが一時故障を起こし，解状況からわずかに変動した（小変動）ネットワーク状況になることがほとんどである．従来の自己安定プロトコルは，小変動状況を考慮しておらず，例えば単一のプロセスの一時故障による影響が，ネットワーク全体に及ぶことがあり，実際のシステムには適さない．したがって，小変動状況から再安定の実行を考慮した自己安定プロトコルは非常に重要である．最近，小変動からの再安定実行において優れた性質をもつ自己安定プロトコルに関する研究が，活発に行われている（[2]〜[6], [9], [11], [12]）．

文献 [4]〜[6] では，1 故障状況（解状況から，1 プロセスが故障で状態変化して得られるネットワーク状況）から再安定するまでの過程において，状態遷移するプロセス数と，再安定するまでの時間を小さくする故障封じ込め（fault-containing）自己安定プロトコルを提案している．つまり故障封じ込めとは，プロセスの故障の影響を局所的なものに抑え，また復旧するまでの時間を短くすることによって，分散システムの利用者に対して故障をほぼ隠ぺいすることを目的としている．例えば，文献 [5] では，生成木構成問題に対し，1 故障状況から再安定するまでに状態遷移するプロセス数 $\Delta+1$，時間 $\Theta(\Delta)$（Δ：プロセスとリンクによって構成されるグラフの次数の最大値．以降プロセスの次数の上界と呼ぶ）である故障封じ込め自己安定プロ

† 奈良先端科学技術大学院大学情報科学センター，生駒市
Information Technology Center, Nara Institute of Science and Technology, 8916-5 Takayama-cho, Ikoma-shi, 630–0101 Japan
†† 大阪大学大学院基礎工学研究科，豊中市
Graduate School of Engineering Science, Osaka University, 1-3 Machikaneyama-cho, Toyonaka-shi, 560–8531 Japan

トコルを提案している．

本論文では，重み最小生成木（minimum weight spanning tree）を構成する故障封じ込め自己安定プロトコルを提案する．重み最小生成木構成問題は，放送や経路制御プロトコルにとって有用であり，この問題を解く故障封じ込め自己安定プロトコルは重要であると考える．提案するプロトコルは，1 故障状況から再安定するまでの実行において，以下が成り立つ．

- 故障プロセス以外は木構造（親子関係）を変更しない．
- 状態遷移するプロセス数：Δ^2+1
- 再安定するまでの時間：$O(1)$

（Δ はプロセスの次数の上界）

文献 [5] の生成木構成自己安定プロトコルは，1 故障状況から再安定するまでの状態遷移プロセス数 $\Delta+1$，時間 $\Theta(\Delta)$ と評価しているが，本論文で用いる評価尺度ではそれぞれ Δ^2+1，$O(1)$ となる．つまり，本論文の重み最小生成木構成自己安定プロトコルの 1 故障状況から再安定するまでの状態遷移プロセス数と時間は，文献 [5] の生成木構成自己安定プロトコルと同等である．

以降では，**2.** で本論文で扱うモデルを説明し，**3.** で提案するプロトコルを紹介する．最後に **4.** で本論文の結果のまとめを述べる．

2. モ デ ル

本章では，本論文で扱うネットワークやプロトコルなどのモデルに関して述べる．

2.1 グ ラ フ

グラフ G は，2 項組 $G=(V,E)$ で定義される．V は空でない頂点集合，E は無向辺の集合である．$u,v\in V$ に関して，$(u,v)\in E$ のとき，u と v は隣接するという．

G の相異なる頂点からなる系列 $\langle v_1,v_2,\cdots,v_m\rangle$ が，各 $i(1\leqq i<m)$ に対して $(v_i,v_{i+1})\in E$ を満たすとき，この系列を v_1-v_m 道といい，$m-1$ をこの道の長さという．1 頂点 v のみからなる系列も，長さ 0 の $v-v$ 道と考える．

2 頂点 u,v の距離を $d(u,v)$ と表し，最短の $u-v$ 道の長さと定義する．ただし，$u-v$ 道が存在しない 2 頂点に対しては $d(u,v)=\infty$ とする．

長さ 2 以上の v_1-v_m 道と辺 (v_1,v_m) が存在するとき，系列 $\langle v_1,\cdots,v_m,v_1\rangle$ を閉路という．閉路がない連結無向グラフを木という．特に根である頂点（r とする）が指定されている木を，r を根とする根付き木という．本論文では根付き木を単に木と呼ぶ．r を根とする木上の隣接する 2 頂点 u,v について，$d(r,u)=d(r,v)-1$ が成立するとき u を v の親と呼び，v を u の子と呼ぶ．

$G=(V,E)$ と $G'=(V',E')$ に対して，$V'\subseteqq V$ かつ $E'\subseteqq E$ が成り立つとき，G' を G の部分グラフといい，$G'\subseteqq G$ と表す．

［定義 1］（重み最小生成木）$G'=(V',E')$ を $G=(V,E)$ の部分グラフとする．$V'=V$ かつ G' が木であるとき，G' を G の生成木という．また，G の辺に重みが与えられているとき，生成木の中で辺の重みの和が最小の生成木を重み最小生成木（minimum weight spanning tree. 以下 MST）と呼ぶ． □

2.2 ネットワークとプロセス

本論文では，n 個のプロセスが通信リンクで接続された任意の形状のネットワーク N を扱う．

N の n 個のプロセス集合を $\mathcal{P}=\{P_0,P_1,\cdots,P_{n-1}\}$ とする．N 中の通信リンクの集合を $\mathcal{L}$ とする．よって，ネットワーク N は 2 項組 $N=(\mathcal{P},\mathcal{L})$ で定義される．

ネットワークは，プロセスを頂点，リンクを辺とするグラフとみなせるので，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．

各プロセスは相異なる識別子をもつ．簡単のためプロセス P_iとその識別子を区別せず，単に P_iと書く．$(P_i,P_j)\in\mathcal{L}(0\leqq i,j\leqq n-1)$ のとき，P_i，P_j間に全 2 重リンクが存在する．各リンクは，それぞれ相異なる正の重みをもち(注 1)，リンク (P_i,P_j) の重みを $w(i,j)$ と表す．なお，各プロセスは自分の識別子と自分に接続するリンクの重みを知っているものとする．

プロセス間の通信は，状態通信モデルを仮定する．つまり，隣接するプロセスは互いの内部状態（識別子，変数の値）を直接知ることができる．

2.3 デ ー モ ン

N 上でのプロトコルの計算状況（ネットワーク状況）を各プロセスの状態を列挙することにより表す．つまり，各プロセス P_iの状態（プログラムカウンタ及び内部変数の値からなる）が q_i であるネットワー

(注 1)：リンクの重みが相異なると仮定しているのは，議論を簡単にするためである．複数のリンクが同じ重みをもつ場合でも，リンク (P_i,P_j) の重みを 3 項組 $(w(i,j),P_i,P_j)$（ただし，$P_i<P_j$）とすることにより，すべての重みが相異なるとみなせるので，本論文のプロトコルはそのまま適用できる．

ク状況を $c=(q_0,q_1,\cdots,q_{n-1})$ と表す．また 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．つまり，P_iのとり得るプロセス状態の集合を Q_i とすると，$C=Q_0\times Q_1\times\cdots\times Q_{n-1}$ である．

プロセスの部分集合を $S\subseteqq\mathcal{P}$ 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プロトコル $\mathcal{A}$ 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 2］（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．プロトコル $\mathcal{A}$，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 $c_{i+1}=c_i(S(i),\mathcal{A})(0\leqq i)$ を満たすとき，E を「初期状況 c_0，スケジュール T に対するプロトコル $\mathcal{A}$ の実行」と呼び，$E(\mathcal{A},T,c_o)$ と表す．　□

［定義 3］（公平なスケジュール）スケジュール T にすべてのプロセス $P_i\in\mathcal{P}$ が無限回現れるとき，T は公平であるという．　□

本論文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．本論文では，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 $t(t\geqq 0)$ について $|S(t)|\geqq 1$
- 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

2.4 自己安定

$\mathcal{LE}\subseteqq C$ 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「プロトコル $\mathcal{A}$ 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「プロトコル $\mathcal{A}$ 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 によるプロトコル $\mathcal{A}$ 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 $i\geqq 0$ 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プロトコルの実行は，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）．

任意の $c\in\mathcal{LE}$ が「各プロセスが MST 上での親プロセスを知っている」という条件を満たすとき，$\mathcal{LE}$ に関して自己安定なプロトコルを「MST 構成問題を解く自己安定プロトコル」という．

2.5 故障封じ込め

本論文では，正当な状況に安定後，単一プロセスの一時故障からの復旧に関して優れた特性をもつ自己安定プロトコルについて考察する．

［定義 4］（1 故障状況）プロトコル $\mathcal{A}$ を，正当な状況 $\mathcal{LE}$ に関して自己安定なプロトコルとする．あるネットワーク状況 $c\in\mathcal{LE}$ において，一つのプロセスの状態を任意に変えることによって得られる状況 c' を 1 故障状況と呼び，状態を変えたプロセスを故障プロセスと呼ぶ．　□

実システムでは，1 故障状況から始まる任意の実行における再安定までに動作するプロセス数，再安定までの時間は重要である．つまり，故障の影響を局所的なものに抑え，かつできる限り迅速に再安定することが望まれる．これらの特長を有する自己安定プロトコルを故障封じ込め自己安定プロトコルという．故障封じ込めの性能は，次に定義する変動プロセス数と再安定時間で評価する．

［定義 5］（同期スケジュール）スケジュール $T=S(0),S(1),\ldots$ において，任意の $i(i\geqq 0)$ について $S(i)=\mathcal{P}$ を満たすとき，スケジュール T を同期スケジュールと呼ぶ．　□

［定義 6］（変動プロセス数と再安定時間）プロトコル $\mathcal{A}$ を正当な状況 $\mathcal{LE}$ に関する自己安定プロトコルとする．任意の 1 故障状況 c から始まる任意の実行 E が，再び $\mathcal{LE}$ を満たすまで（再安定するまで）に状態遷移する最大プロセス数を「変動プロセス数（contamination number）」，それにかかる時間（同期スケジュールにおいて再安定するまでの時間）を「再安定時間（convergence time）」という．　□

本論文では，1 故障状況から再安定するまでの実行に関して優れた性質をもつ，MST 構成問題を解く故障封じ込め自己安定プロトコルについて述べる．

3. MST 構成問題を解く故障封じ込め自己安定プロトコル $\mathcal{MST}$

本章では，MST を構成する故障封じ込め自己安定プロトコル $\mathcal{MST}$を示す．$\mathcal{MST}$は，文献 [10] の MST を構成する自己安定プロトコルを拡張したものである．そこでまず，文献 [10] のプロトコルの概略を紹介する．なお以降では，ネットワーク中に MST の根となる特別なプロセス（P_r とする）が一つだけ存在し，全プロセスが P_rを知っているものとする．

3.1 MST 構成自己安定プロトコル（$\mathcal{SMST}$）

文献 [10] の MST を構成する自己安定プロトコル $\mathcal{SMST}$は，任意の生成木に次の操作を繰り返し適用すれば MST が構成できることに基づいている．

- 生成木に含まれないリンク l を生成木に追加し，生じた閉路中で最大重みをもつリンクを削除することにより，生成木を更新する．

プロトコル $\mathcal{SMST}$（プロセス P_iの一原子動作）を図 1 に示す．各プロセスは図 1 のプロトコルを永久に繰り返し実行する．プロトコル中の定数 P_i（識別子），N_i（P_iの隣接プロセス集合）は，任意の状況において正しい値を保持しているものとする．プロトコル中の変数及び関数は，以下のように定義される．

```
root process（P_r）:
  mypath_r := ⟨P_r⟩

non-root process（P_i ≠ P_r）:
if parent(mypath_i)∈ N_i then
  /*case of parent process*/
  Let P_j be parent(mypath_i)
  if /*parent's path is clearly wrong that is*/
    "P_i is in mypath_j" or "mypath_j = ∅" then
      wpath := ∅ ; mypath_i := ∅
  else wpath :=append(mypath_j, w(i, j), P_i)
else wpath := ∅
for each P_j ∈ N_i−parent(mypath_i) do
  /*case of non-parent process*/
  if /*neighbor's path is clearly wrong that is*/
    "P_i is in mypath_j" or "mypath_j = ∅" then
      wpath := ∅
  else if mypath_i = ∅ then
        wpath :=append(mypath_j, w(i, j), P_i)
    else if sellink(mypath_i, mypath_j, w(i, j)) then
        wpath :=append(mypath_j, w(i, j), P_i)
od
mypath_i := wpath
```

図 1　MST 構成自己安定プロトコル $\mathcal{SMST}$
Fig. 1　Self-stabilizing protocol $\mathcal{SMST}$ for constructing MST.

- $mypath_i$：根 P_rから P_iまでの経路を表す変数．プロセスの識別子とリンクの重みの交代列

$$\langle P_r, w_0, P_{j_1}, \cdots, P_{j_k}, w_k, P_i \rangle$$

で表される．空系列（$\emptyset$ と表す）を値としてとり得るものとする．ただし，簡単のため，任意の状況において，$mypath_i$の長さが 3 以上なら $P_{j_k} \in N_i$，$w_k = (j_k, i)$ とする．また，値が空でない場合，先頭の要素は常に P_rであるとする．

- parent($mypath_i$)：$mypath_i$における P_iの直前のプロセス（$mypath_i$の最後から 3 番目の要素）を返す関数．ただし，$mypath_i$の長さが 2 以下の場合は，空を返す．

- append($mypath, w, ID$)：$mypath$ の最後に w, ID を追加して得られる系列を返す関数．

- ischild($mypath_i, mypath_j$)：$mypath_i$ = append ($mypath_j, w, ID$) なる w, ID が存在するとき真を返す関数．

- sellink($mypath_i$, $mypath_j$, w)： $mypath_i$, $mypath_j$ 及びそれらの終点の間の重み w のリンク l から構成されるグラフにおいて，リンク l を含む単純閉路の最大重みリンクが $mypath_i$ に存在するとき真を返す関数．ただし，リンク l を含む単純閉路が生じないとき（例えば，ischild($mypath_i, mypath_j$) が真のとき）は，偽を返す．

文献 [10] では，正当な状況を次のように定義し，$\mathcal{SMST}$が MST 構成問題を解く自己安定プロトコルであることを示している．

［定義 7］（正当な状況）各プロセス P_iが次の条件を満たすネットワーク状況を，正当な状況という．

MST 上での根から P_iまでの経路が $mypath_i$に保持されている．　□

プロトコル $\mathcal{SMST}$は，1 故障状況から再安定までに，最悪の場合ほとんどすべてのプロセスが変数 $mypath$ を変更する．例えば根に近いプロセスが故障した場合，故障プロセスが動作する前にその隣接プロセスが動作すると，誤った $mypath$ 情報をもとに自分の $mypath$ を変更し，それが次々に伝搬することによって，ほとんどすべてのプロセスが $mypath$ を変更する可能性がある．これを防ぐには，1 故障状況から

の再安定において，各プロセスが 1 故障状況と思われる状況では動作せず，かつ故障プロセスが動作すれば正当な状況になることを保証すればよい．以下，プロトコル $\mathcal{SMST}$に対して故障封じ込めの性質をもたせる拡張について述べる．

3.2 プロトコル $\mathcal{MST}$のアイデア

MST を構成する故障封じ込め自己安定プロトコル $\mathcal{MST}$では，故障封じ込め性として，1 故障状況から再安定するまでの実行で MST 構成情報（変数 $mypath$）を変更するプロセスを故障プロセスのみに限定することを考える．

$\mathcal{MST}$の基本的な構成は，$\mathcal{SMST}$に Ghosh ら [5] と同様の手法を適用したものである．Ghosh らの手法では，各プロセスが周辺のプロセス状態から 1 故障状況であることを判断し，自分以外のプロセスの故障による 1 故障状況と判断すると，自分はすぐに動作せず，故障プロセスの動作を待つ．

$\mathcal{SMST}$において，プロセス P_iが動作可能（変数 $mypath_i$を変更可能）であるとき，P_iの状態を「矛盾状態（定義 8）」という．1 故障状況において，プロセス P_iが矛盾状態であるのは以下のいずれかの場合である．

（1） P_iが故障プロセス

（2） P_iの隣接プロセス P_jが故障プロセス

（1）の場合は，P_iが $\mathcal{SMST}$に従って動作すれば，直ちに正当な状況に再安定する．一方,（2）の場合は，故障封じ込めを実現するためには，P_jが $\mathcal{SMST}$に従って動作し，再安定するのを待たねばならない．つまり，故障を封じ込めを実現するためには，$\mathcal{SMST}$に従って動作してよい場合と，してはいけない場合を，各プロセスが判断する必要がある．この判断条件を $\mathcal{SMST}$に追加することによって $\mathcal{MST}$を得る．

3.3 プロトコル $\mathcal{MST}$

まず，以下の関数を定義する．

- $\mathrm{maxl}(mypath_i, mypath_j, w)$: $mypath_i, mypath_j$ 及びそれらの終点の間の重み w のリンク l から構成されるグラフにおいて，リンク l を含む単純閉路中で，最大重みを返す関数．ただし，リンク l を含む単純閉路が生じないとき（例えば，$\mathrm{ischild}(mypath_i, mypath_j)$ が真のとき），0 を返す．

次に，述語 $cons(P_i)$ を定義する．

［定義 8］（無矛盾状態）以下に定義される述語 $cons(P_i)$ を満たすプロセス P_i を「無矛盾状態」という．また満たさないものを「矛盾状態」という．

$$
\begin{aligned}
&cons(P_i) = \\
&\quad \forall P_j \in N_i[\mathrm{ischild}(mypath_j, mypath_i) \vee \\
&\quad \mathrm{ischild}(mypath_i, mypath_j) \vee \\
&\quad (\mathrm{maxl}(mypath_i, mypath_j, w(i,j)) = w(i,j))].
\end{aligned}
$$

なお，$cons(P_r)$ は常に真であるとする． □

つまり，$cons(P_i)$ を満たす（無矛盾状態の）プロセスとは，各隣接プロセスに対して，その隣接プロセスが「自分の親プロセス」か「自分の子プロセス」，あるいは「その隣接プロセスとの間のリンクを追加した結果生じた単純閉路中で，最大重みをもつリンクが追加したリンクである（つまり，自分と隣接プロセスの経路情報から判断する限り，現在の自分の経路が MST 上の経路と思われる）」のいずれかが成り立つ場合である．

なお，各プロセス P_iの $mypath_i$が MST 上での P_r から P_iまでの経路を保持している場合，明らかにすべてのプロセス P_iで $cons(P_i)$ が成立する．

矛盾状態のプロセス P_iを無矛盾状態にするような $mypath_i$の値が存在するとき，その代入を「安定代入」という．矛盾状態のプロセス P_iにおける安定代入の有無を表す述語 $can\_stab(P_i)$ を定義する．

$$
\begin{aligned}
&can\_stab(P_i) \\
&\quad = \exists mypath_i'(\forall P_j \in N_i \\
&\quad [ischild(mypath_i', mypath_j) \vee \\
&\quad ischild(mypath_j, mypath_i') \vee \\
&\quad (maxl(mypath_i', mypath_j, w(i,j) = w(i,j)))] \wedge \\
&\quad (mypath_i' \neq mypath_i)).
\end{aligned}
$$

つまり，現在の $mypath_i$と異なる $mypath_i'$ が存在し，その $mypath_i'$ が $cons(P_i)$ を満たす，すなわち $mypath_i$ を変更することによって無矛盾状態になり得るとき，$can\_stab(P_i)$ が成立し安定代入が存在する．

故障封じ込めを実現する，つまり 1 故障状況において故障プロセス以外は変数 $mypath$ を変更しないようにするための具体的な方法について述べる．

故障プロセス P_fによる 1 故障状況において，$mypath_f$が変更された場合，必ず $can\_stab(P_f)$ が成立する．また，$mypath_f$の変更によって，P_fの隣接プロセスも can_stab が成立することがある．実際，1 故障状況において can_stab が成立するプロセスは，故障プロセス，故障プロセスを MST 上の親プロセスとする葉プロセス，あるいは故障プロセスを MST 上

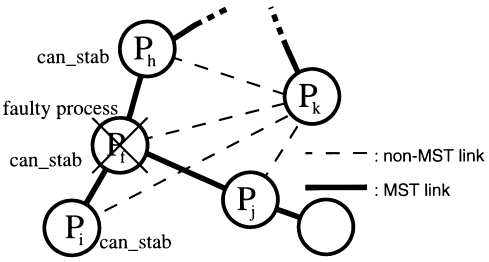


図 2　1 故障状況で can_stab を満たすプロセス
Fig. 2　can_stab processes in 1-faulty state.

の唯一の子プロセスとするプロセスのいずれかのみとなる（補題 7）.

図 2 において，故障プロセス P_fでは $can\_stab(P_f)$ が成立し，P_fの隣接プロセス P_i，P_hでは can_stabが成立する可能性がある．一方，P_j，P_kでは can_stab が成立することはない．更に，1 故障状況において故障プロセス以外のプロセスで，述語 $cons(P_l)$ が成立しないプロセス P_l は，故障プロセスが動作することによって $cons(P_l)$ が成立することに注意する.

以上より，故障封じ込めを実現するために，1 故障状況においてプロセスが変数 $mypath$ を変更してはいけない条件を導く.

具体的な条件を示す前に，条件の記述に用いる関数を定義する.

- $\#cs\_neigh(P_i)$：P_iの全隣接プロセス中で can_stab を満たすプロセス数．つまり $\#cs\_neigh(P_i) = |\{P_j \in N_i | can\_stab(P_j) = true\}|$.
- $mkcons_i(P_j)$：隣接プロセス P_j が動作し，$mypath_j$を変更することによって，P_i（$mkcons_i$ を実行したプロセス）が無矛盾になる場合に真を返す関数.

述語 $Q1$〜$Q4$ を次のように定義する.

$Q1$:$can\_stab(P_i)$

$Q2$:$\#cs\_neigh(P_i) = 1 \wedge$
$\forall P_j \in N_i[can\_stab(P_j) \Rightarrow mkcons_i(P_j)]$

$Q3$:$\#cs\_neigh(P_i) \geqq 2 \wedge$
$\exists P_p \in N_i[can\_stab(P_p) \wedge mkcons_i(P_p) \wedge$
$\forall P_j \in N_i - \{P_p\}[can\_stab(P_j) \Rightarrow$
$P_p = parent(mypath_j) \vee$
$P_j = parent(mypath'_p)]]$
（ただし $mypath'_p$ は，$can\_stab(P_p)$ が成立する場合に，現在の $mypath_p$ とは異なる $mypath'_p$ で，$cons(P_p)$ を満たす）

$Q4$:$\forall P_j \in N_i[\neg ischild(mypath_i, mypath_j)] \wedge$
$can\_stab(parent(mypath_i)) \wedge$
$mkcons_i(parent(mypath_i))$

$Q2, Q3, Q4$ は，以下のように説明できる.

- $Q2$：P_iの隣接プロセス中に can_stab を満たすプロセスがただ一つ存在（P_j）し，かつ P_jが動作すると P_iが無矛盾状態になる.
- $Q3$:P_iの隣接プロセス中に can_stabを満たすプロセスが複数存在（集合 P）し,「あるプロセス $P_p \in P$ が存在し，P の他のプロセスは MST 上での P_p の子プロセスあるいは親プロセス」が満たされる．かつ，P_p が動作すると P_iが無矛盾状態になる.
- $Q4$：P_iに子プロセスが存在せず（葉プロセス），かつ P_iの親プロセスが can_stab を満たし，かつ親プロセスが動作すると P_iが無矛盾状態になる.

これらの述語を使い，1 故障状況で自分が故障プロセスではないと判断される条件，つまりプロセスが動作してはいけない条件は以下のように記述できる.

$$(Q1 \wedge Q4) \vee Q2 \vee Q3$$

$(Q1 \wedge Q4)$ を満たすプロセスは，図 2 の P_iである．また，$Q2$ を満たすプロセスは同図の P_hと P_j，$Q3$ を満たすプロセスは同じく P_k である.

この条件が成立しない場合，すなわち 1 故障状況ではない，あるいは自分の故障による 1 故障状況と判断したプロセスは，$\mathcal{SMST}$を実行すればよい．つまり故障封じ込めを実現した MST 構成プロトコル $\mathcal{MST}$ の概要は図 3 となる.

しかし，$Q2$〜$Q4$ で用いている $can\_stab(P_j)$，$mkcons_i(P_j)$，$can\_stab(P_p)$，$mkcons_i(P_p)$，$\mathrm{parent}(mypath'_p)$，$can\_stab(parent(mypath_i))$，及び $mkcons_i(parent(mypath_i))$ は，P_iとその隣接プロセスの状態だけからは評価できない．つまり，状態通信モデルにおいてそのままでは局所的な評価は不能である．そこで，文献 [5] で述べられている同期機構と同様の手法(注 2)を用い，これらの値を評価するのに

S1:	$((\neg Q1 \vee \neg Q4) \wedge \neg Q2 \wedge \neg Q3)$	$\Longrightarrow$	exec. $\mathcal{SMST}$
S2:	$((Q1 \wedge Q4) \vee Q2 \vee Q3)$	$\Longrightarrow$	no move

図 3　プロトコル $\mathcal{MST}$ の概要
Fig. 3　Overview of protocol $\mathcal{MST}$.

(注 2)：ここで扱う手法は一般的な同期手法ではなく，あるプロセスからある隣接プロセスへの一方的な情報伝達を制御するためのものである．したがって，一般の同期機構で議論されるデッドロック問題などは生じない.

S10: $\neg cons(P_i) \wedge q_{ij} = \perp \wedge a_{ji} = \perp \implies q_{ij} := ask$
S11: $\exists j \in N_i : a_{ij} \neq f_i(q_{ji}, all(P_i)) \implies a_{ij} := f_i(q_{ji}, all(P_i))$
S12: $cons(P_i) \wedge q_{ij} \neq \perp \implies q_{ij} := \perp$

図 4　同期機構を実現するプロトコル
Fig. 4　The synchronization scheme.

必要な情報を隣接プロセス P_j（$P_p, parent(mypath_i)$ も隣接プロセスである）から知らせてもらうことにより評価する．この同期機構は，図 5 で示すプロトコル $\mathcal{MST}$ と並行に実行される．以下では同期機構について述べる．

すべてのプロセス P_iは，各隣接プロセス $P_j \in N_i$ に対して以下の変数 q_{ij}, a_{ij} をもつ．

- q_{ij} :$\perp, ask$ のいずれかをとる．P_iから P_jへの質問が発行中かどうかを表す変数．
- a_{ij}: P_jが P_iに発行した質問に対する返答を表す変数．返答は q_{ji} 及び P_iと P_iの全隣接プロセスの状態から決定される．つまり，P_i及び P_iの全隣接プロセスの状態を $all(P_i)$ とすると，a_{ij} は q_{ji} と $all(P_i)$ を引き数とする 2 引き数関数 f_i で表され，$a_{ij} = f_i(q_{ji}, all(P_i))$ となる．質問が発行されていない場合，$\perp$ となる．

同期機構の動作は，次のとおりである．P_iが質問 q_{ij} に対する正しい返答 a_{ji} を得るには，a_{ji} が $\perp$ になるのを待って q_{ij} を ask にする．次に a_{ji} が $\perp$ 以外の値をもった場合，それは q_{ij} を ask にした，つまり質問を発行した時点以降の $all(P_j)$ から得られる値である．P_iが q_{ij} を ask に変える（隣接プロセスの情報が必要である場合）のは，隣接プロセス P_jの $can\_stab(P_j)$，$mkcons_i(P_j)$ 及び $parent(mypath'_j)$ の評価が必要な場合，つまり $cons(P_i)$ が成立しない（矛盾状態の）場合である．P_iは，a_{ji} が $\perp$ 以外の値をもった場合，その値をもとにプロトコル $\mathcal{MST}$に従って動作する．その結果，$cons(P_i)$ が成立すると q_{ij} を $\perp$ にする．この同期機構プロトコルを図 4 に示す．なお P_iは，図 4 の S11〜S12 以外の場合は動作しない．

次に f_i（2 引き数関数）を定義する．P_jで局所評価不能かつ評価する必要がある述語のうち，$mkcons_j(P_i)$ は，プロトコル $\mathcal{MST}$ において P_iで $can\_stab(P_i)$ が成立し，かつ P_iが動作することによって $cons(P_j)$ が成立するかどうかを知るために利用している．つまりこれを P_jで評価するためには，P_iで $can\_stab(P_i)$ が成立する場合に，現在とは異なる変数 $mypath_i$の値（$mypath'_i$ とする）で，$cons(P_i)$ を満たすものが必要である．この値 $mypath'_i$ は，同様に局所評価不能である $\mathrm{parent}(mypath'_i)$ を評価する際にも利用する．

よって f_i は，$can\_stab(P_i)$ 及び $can\_stab(P_i)$ が真の場合は $cons(P_i)$ を満たす $mypath'_i$ を返せばよい．

$$f_i = \begin{cases} \langle can\_stab(P_i), mypath'_i \rangle & \\ \qquad\qquad \text{if } q_{ji} = ask & \\ \perp \qquad\qquad \text{otherwise.} & \end{cases}$$

（$mypath'_i$ は，$can\_stab(P_i)$ が真の場合は $cons(P_i)$ を満たす変数 $mypath_i$の値，それ以外は $\perp$ を返す.）

なお，各プロセス P_iでは，a_{ij} の値は常に評価されており (S11)，f_i の定義に従って，隣接プロセスが情報を必要としている場合，すなわち隣接プロセス P_j が矛盾状態となり，q_{ji} を ask にしている場合に，a_{ij} として $\perp$ 以外の値を代入する．つまり，q_{ji} が ask の間，常に a_{ij} は更新されている（同じ値に更新することもある）ことに注意する．

プロトコル $\mathcal{MST}$の概要（図 3）の $Q2, Q3, Q4$ を評価するために，隣接プロセス P_jの $can\_stab(P_j)$ 及び $mkcons_i(P_j)$ を評価する部分を，同期機構を利用して実現する．具体的には，図 3 の $Q2$〜$Q4$ の $can\_stab(P_j)$ を以下の述語に置換する．

$$(q_{ij} = ask \wedge a_{ji} = \langle can\_stab(P_j), mypath'_j \rangle \wedge \\ can\_stab(P_j))$$

ただし，上式の最後の条件 $can\_stab(P_j)$ は，a_{ji} 中の $can\_stab(P_j)$（P_jから得た $can\_stab(P_j)$ の値）を表す．更に，$mkcons_i(P_j)$ と $\mathrm{parent}(mypath'_j)$ の評価は，同期機構によって P_jから得られた $mypath'_j$ を使用し評価する．また，$can\_stab(parent(mypath_i))$ も同様に変更する（P_p に関しても同様）．

更に，$can\_stab(P_j)$ を評価する時点で同期機構の動作が終了している，つまり正しい値が返答されていることを保証するために，述語 $Q0 : \exists P_j \in N_i[(q_{ij} = \perp \wedge a_{ji} \neq \perp) \vee (q_{ij} = ask \wedge a_{ji} = \perp)]$ を追加する．$Q0$ が真の場合，同期機構による質問，返答の途中である．したがって，各プロセスが動作するのは $\neg Q0$ のときのみでなければならない．

S1: $\neg cons(P_i) \wedge \neg Q0 \wedge ((\neg Q1 \vee \neg Q4) \wedge \neg Q2 \wedge \neg Q3) \implies$ exec. $\mathcal{SMST}$
S2: $cons(P_i) \vee Q0 \vee ((Q1 \wedge Q4) \vee Q2 \vee Q3)) \implies$ no move

図5 プロトコル $\mathcal{MST}$
Fig. 5 The protocol $\mathcal{MST}$.

これらの拡張を行ったプロトコル $\mathcal{MST}$を図 5 に示す.

3.4 正 当 性

まず，正当な状況を定義する.

［定義 9］（$\mathcal{MST}$の正当な状況）各プロセス P_iが次の条件を満たすネットワーク状況を正当な状況という.

- MST 上での根から P_iまでの経路が $mypath_i$ に保持されている
- 各 $P_j \in N_i$ に対し，$q_{ij} = a_{ij} = \perp$ □

$\mathcal{MST}$の正当性を証明する.

プロトコル $\mathcal{SMST}$は自己安定プロトコルなので，自己安定の条件（**2.4**）の閉包性を満たしている．プロトコル $\mathcal{MST}$は，$\mathcal{SMST}$の動作を抑制しているだけなので，$\mathcal{MST}$も閉包性を満たすことは明らかである.

同期機構プロトコルより，次の補題 1 は明らか.

［補題 1］ 各プロセスが S10〜S12 を少なくとも一度ずつ評価すると，それ以降 a_{ji} の値は，直前に P_iで q_{ij} が ask になった時点以降に計算された値である．また，$\neg cons(P_i)$ の場合，いつか必ず $\neg Q0$ となり，$cons(P_i)$ が成立するまで $\neg Q0$ のままである． □

自己安定プロトコルでは，初期状況に仮定をおかない．よって初期状況で各プロセスの変数 $mypath$ には，ネットワークに存在しない経路を含む場合がある．このような経路を「偽経路」と呼ぶ．また，実存する経路を「実経路」と呼ぶ.

［補題 2］ $\mathcal{MST}$の任意の実行において，ある時点以降，すべてのプロセスの $mypath$ は実経路になる.

（証明） 任意の状況において，最短の偽経路長を f とし，その偽経路をもつプロセスを P_kとする.

P_kは $\neg cons(P_k)$ であり，かつ隣接プロセスの状態変化によって $cons(P_k)$ になることはないので，$\neg Q2, \neg Q3, \neg Q4$ である．また，補題 1 より $\neg Q0$ となる．したがって，いずれ必ず S1 が実行され，$mypath_k$ を変更する．このとき，変更後の $mypath_k$は，ある隣接プロセス P_jの $mypath_j$に，$w(j,k)$ 及び P_kを追加したものである．よって，変更後の $mypath_k$が偽経路の場合（つまり $mypath_j$が偽経路の場合），その長さは $f+2$ 以上となる．このことから，いずれ長さ f 以下の偽経路が存在しない状況になることが示せる．この議論を繰り返すことにより，最短の偽経路長が単調増加であることが示せる．一方，付加される辺及び識別子は実存するものであり，プロトコルより変数 $mypath$ に付加するプロセスは重複を許さないので，偽経路の長さはたかだか $l+2n$（l は初期状況における最長偽経路の長さ）にしかならない．以上，偽経路長の単調増加性と上限の存在より補題が成立する. □

以降，すべてのプロセスの $mypath$ は実経路であると仮定する.

［補題 3］ すべてのプロセス P_iで $cons(P_i)$ が成立すれば，いずれ正当な状況になる.

（証明）プロトコル $\mathcal{MST}$より，$cons(P_i)$ が成立するプロセス P_iは $mypath_i$を変更しないので，すべてのプロセス P_iで $cons(P_i)$ が成立すれば，それ以降，すべてのプロセス P_iで $cons(P_i)$ が成立し続ける．このことと，同期機構（図 4）より，ある時点以降，すべての隣接プロセス P_i，P_jに対して $q_{ij} = a_{ij} = \perp$ が成立するのは明らか．以下ではすべてのプロセスで $cons(P_i)$ が成立するとき，MST が完成していることを示す.

すべてのプロセス P_iで $cons(P_i)$ が成立し，かつ MST が完成していないと仮定する．つまり，$cons(P_j)$ が成立しているにもかかわらず $mypath_j$が MST 上の経路とは異なるプロセス P_jが存在する.

$cons(P_j)$ が成立しているので，$mypath_j$は，MST 上での親プロセス P_kの $mypath_k$に $w(k,j), P_j$ を追加したものである．このとき，もし $mypath_k$が MST 上の経路であれば，$cons(P_j)$ の定義より $mypath_j$も MST 上の経路である．したがって，$mypath_k$は MST 上の経路ではない.

この議論を繰り返し適用すると，根プロセス P_rの変数 $mypath$ が MST の経路と異なることになるが，これは $mypath_r = \langle P_r \rangle$ に矛盾する． □

続いて，ある時点以降，すべてのプロセス P_iで $cons(P_i)$ が成立することを証明する.

［補題 4］ $(P_r, P_{i_1}, \cdots, P_i, P_j)$ を MST 上の経路とす

る．経路中のプロセス $P_r, P_{i_1}, \cdots, P_i$ は，MST 上の経路を変数 $mypath$ に保持しており，P_jは MST 上の経路を $mypath_j$に保持していない（つまり，$mypath_j \neq (P_r, w(r,i_1), P_{i_1}, \cdots, P_i, w(i,j), P_j)$）とする．このとき以下が成り立つ．

（1） プロセス $P_r, P_{i_1}, \cdots, P_i$ は，$mypath$ を変更することはない．

（2） P_jは，必ず S1 を実行し，S1 を実行すると $mypath_j$は MST 上の経路になる．

（証明） すべてのプロセスの変数 $mypath$ が実経路であることから，(1) は明らか．(2) についてのみ証明する．

明らかに $\neg cons(P_j)$ である．また，P_jにおいて $mypath_j$=append$(mypath_i, w(i,j), P_j)$ にならない限り $cons(P_j)$ は成立し得ない．つまり，P_jのどの隣接プロセス P_kが $mypath$ を変化させても $cons(P_j)$ は成立せず，$mkcons_j(P_k)$ が満たされない．よって $\neg Q2, \neg Q3, \neg Q4$ となり，また補題 1 より $\neg Q0$ となるので，必ず S1が実行される．また，S1が実行され，P_j が $\mathcal{SMST}$に従って $mypath_j$を変更すると，$mypath_j$ は MST 上の経路をもつ． □

補題 3，4 より次の定理を示せる．

［定理 1］ プロトコル $\mathcal{MST}$は MST を構成する自己安定プロトコルである． □

最後に，プロトコル $\mathcal{MST}$の故障封じ込め性を証明する．はじめに同期機構について，以下の補題を証明する．

［補題 5］ 1 故障状況から始まる実行の任意の状況において，任意の隣接プロセス P_i，P_jに対して，$q_{ij} = ask \wedge a_{ji} \neq \perp$ であれば，a_{ji} は，直前に P_i が q_{ij} を ask にした時点以降に計算された値である．

（証明） 1 故障状況で，故障によって変数 a, q が変化しなければ補題が成立することは明らか．したがって P_i，P_jのいずれかで故障が起きたと仮定する．

（1） P_jが故障（$a_{ji} \neq \perp, q_{ij} = \perp$）の場合：S10 で q_{ij} が ask になる前に，a_{ji} は S11 で $\perp$ になる．よって a_{ji} は，直前に q_{ij} が ask になった後に計算された値．

（2） P_iが故障（$a_{ji} = \perp, q_{ij} = ask$）の場合：明らかに a_{ji} は，直前に q_{ij} が ask になった後に計算された値． □

以下，1 故障状況からの実行において，$\mathcal{MST}$が故障封じ込め性をもつことを証明するために，いくつかの補題を証明する．

can_stab の定義から，次の補題は明らかである．

［補題 6］ $mypath$ が変更されない故障による 1 故障状況では，can_stab が成立するプロセスは存在しない． □

補題 8 のために，次の補題を証明する．

［補題 7］ 1 故障状況において can_stabが成立するプロセスは，故障プロセス，故障プロセスを MST 上の親プロセスとする葉プロセス，あるいは故障プロセスを MST 上での唯一の子プロセスとするプロセスのいずれかのみである．

（証明） 以下，MST 上の親プロセス，子プロセスを，単に親プロセス，子プロセスという．故障プロセス（P_f）が can_stab になるのは明らか．P_fの隣接プロセスのうち，P_f以外を子プロセスとしてもつプロセス（P_i）は，それら子プロセスがもつ $mypath$ が正しい値なので，無矛盾状態になる，現在の $mypath_i$と異なる $mypath'_i$ が存在しない．つまり $can\_stab(P_i)$ は成立しない．

P_fの隣接プロセスのうち，MST 上のリンクでないリンクで接続された葉プロセスを P_j，P_jの（MST 上での）親プロセスを P_p とする．このとき，MST の性質より $w(f,j) > w(p,j)$ なので，無矛盾状態になる，現在の $mypath_j$とは異なる $mypath'_j$ は存在しない．つまり，$can\_stab(P_j)$ は成立しない． □

［補題 8］ 1 故障状況からの実行で，変数 $mypath$ を変更するのは故障プロセスのみである．

（証明） 1 故障状況での故障プロセスを P_fとする．P_fでは $can\_stab(P_f)$ が成立し，かつ P_f以外のどのプロセスが $mypath$ を変更しても $cons(P_f)$ は成立しない．つまり，$\neg Q2, \neg Q3, \neg Q4$ となり，また補題 1 より $\neg Q0$ になるので，必ず S1 を実行する．

P_fの任意の隣接プロセスを P_iとすると，P_iは以下のように場合分けできる（根プロセスは除く）．

（1） P_fの親の場合

（a） $can\_stab(P_i)$ が不成立（$\neg Q1$）の場合

補題 7 より，P_iの隣接プロセスのうち can_stabが成立するのは P_fのみ，あるいは P_iの隣接プロセスのうち P_fを親プロセスとする葉プロセスである．これらは P_fが動作すると無矛盾状態になる（$cons$ が成立）．したがって，$Q2, Q3$ のいずれかが成立し，よって S1 を実行しない．

（b） $can\_stab(P_i)$ が成立（$Q1$）の場合

補題 7 より，$can\_stab(P_i)$ が成立するのは P_fが P_i の唯一の子プロセスである場合のみ．また，補題 7

及び P_fが動作すると $cons(P_i)$ が成立することより，$Q2, Q3$ のいずれかが成立し，よって S1 を実行しない．

（2） P_fの子プロセスで，葉プロセスでない場合

補題 7 より，$can\_stab(P_i)$ は不成立（$\neg Q1$）．また，補題 7，及び P_fが動作すると $cons(P_i)$ が成立することより，$Q2, Q3$ のいずれかが成立し，よって S1 を実行しない．

（3） P_fの子プロセスで，葉プロセスの場合

（a） $can\_stab(P_i)$ が不成立（$\neg Q1$）の場合

補題 7，及び P_fが動作すると $cons(P_i)$ が成立することより，$Q2, Q3$ のいずれかが成立し，よって S1 を実行しない．

（b） $can\_stab(P_i)$ が成立する（$Q1$）の場合

明らかに $Q4$ が成立するため S1 を実行しない．

（4） P_fと MST 以外のリンクで隣接する場合

補題 7，及び P_fが動作すると $cons(P_i)$ が成立することより，$Q2, Q3$ のいずれかが成立し，よって S1 を実行しない． □

［補題 9］ 変動プロセス数はたかだか Δ^2+1 である．

（証明） 同期機構の動作（同期動作）を行うのは，$\neg cons(P_i)$ であるプロセス P_iのみ．$mypath$ が変化しない故障の場合，同期動作を行うプロセスは明らかにたかだか故障プロセス及びその隣接プロセスのみ．$mypath$ が変化した場合，$\neg cons$ であるプロセスは，故障プロセス及びその隣接プロセスの一部である（補題 7）．したがって，同期動作を行う（a, q を変更する）のは故障プロセスから距離 2 以内のプロセスである．また，$mypath$ を変更するのは故障プロセスのみ（補題 8）．よって変動プロセス数は Δ^2+1 である（Δ はプロセスの次数の上界）． □

［補題 10］ 1 故障状況からの再安定時間は $O(1)$ である．

（証明） 補題 8 より，1 故障状況からの実行において $mypath$ を変更するのは故障プロセスのみ．また，故障プロセスがプロトコルに従って一度 $mypath$ を変更すると $mypath$ の値は正しくなる（$cons$ が成り立つ）．一方，$cons$ が成り立つと新たに同期動作は行われない．よって，補題 9 より，故障プロセスが動作し $cons$ が成立するまでの間にたかだか Δ^2+1 個のプロセスが同期動作を行い，かつ，故障プロセスで $cons$ が成り立つとそれ以降同期動作は行われない．また同期動作は明らかに定数回の動作で停止する．したがって，同期スケジュールによる実行において，各プロセスが定数回動作すれば再安定する．よって再安定時間は $O(1)$ である． □

定理 1 及び補題 9，10 より，以下の定理がいえる．

［定理 2］ プロトコル $\mathcal{MST}$は，変動プロセス数 Δ^2+1，再安定時間 $O(1)$ の，重み最小生成木を構成する故障封じ込め自己安定プロトコルである．更に，1 故障状況から始まる実行において，MST の情報を格納する変数（$mypath$）を変更するのは故障プロセスのみである． □

4. むすび

本論文では，1 故障状況からの実行において，再安定時間 $O(1)$，変動プロセス数 Δ^2+1（Δ はプロセスの次数の上界），故障状況からの実行で MST の経路情報を変更するのは故障プロセスのみである，故障封じ込め自己安定 MST 構成プロトコル $\mathcal{MST}$を提案した．

本プロトコルの更なる改善としては，現在は根プロセスから自分までの MST 上の経路情報をすべてもっているが，これを局所情報のみに限定し，変数領域を小さくすることなどが挙げられる．

謝辞 本研究を進めるにあたり，MST 構成プロトコルについて議論してくれた小谷晃一氏（現在，阪急電鉄株式会社）に深謝します．また日ごろから活発な議論を行って下さる奈良先端科学技術大学院大学情報科学研究科の井上美智子助手と浮穴学慈氏に深く感謝します．本研究の一部は，文部省科学研究費補助金（特定領域研究 (B)(2)10205218），日本学術振興会科学研究費（基盤研究 (c)(2)12680349）による．

文　献

[1] E.W. Dijkstra, “Self-stabilizing systems in spite of distributed control,” Commun. ACM, vol.17, no.11, pp.643–644, 1974.

[2] S. Dolev and T. Herman, “Superstabilizing protocols for dynamic distributed systems,” Proc. 2nd Workshop on Self-Stabilizing Systems, pp.3.1–3.15, 1995.

[3] T. Herman, “Superstabilizing mutual exclusion,” Proc. International Conf. on Parallel and Distributed Systems, pp.31–40, 1995.

[4] S. Ghosh, A. Gupta, T. Herman, and S.V. Pemmaraju, “Fault-containing self-stabilizing algorithms,” Proc. 40th PODC, pp.45–54, 1996.

[5] S. Ghosh, A. Gupta, and S.V. Pemmaraju, “A fault-containing self-stabilizing algorithm for spanning trees,” J. Computing and Information, vol.2, pp.322–338, 1996.

[6] S. Ghosh and A. Gupta, “An exercise in fault-containment: Self-stabilizing leader election,” Inf. Process. Lett., vol.59, pp.281–288, 1996.

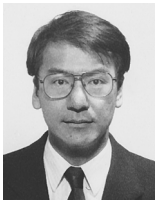
[7] S. Dolev, A. Israeli, and S. Moran, “Self stabilization

of dynamic systems assuming only read/write atomicity," Proc. 9th PODC, pp.103–117, 1990.
[8] G. Antoniu and P.K. Srimani, "Distributed self-stabilizing algorithm for minimum spanning tree construction," LNCS1300, pp.480–487, 1997.
[9] S. Kutten and B. Patt-Shamir, "Time-adaptive self stabilization," Proc. 16th PODC, pp.149–158, 1997.
[10] 小谷晃一，片山喜章，増澤利光，都倉信樹，"ネットワークの重み最小生成木を構成する自己安定分散アルゴリズムについて，" 信学技報，COMP92–5, 1992.
[11] 桝田秀夫，片山喜章，増澤利光，都倉信樹，"C-daemonでのリングの方向付けのための自己安定アルゴリズムについて，" 信学技報，COMP93–96, 1994.
[12] E. Ueda, Y. Katayama, T. Masuzawa, and H. Fujiwara, "A latency-optimal superstabilizing mutual exclusion protocol," Proc. 3rd Workshop on Self-Stabilizing Systems, pp.110–124, 1997.

（平成 12 年 7 月 27 日受付，13 年 1 月 29 日再受付）

片山　喜章　（正員）

平 2 阪大・基礎工・情報卒．平 6 同大大学院博士後期課程中退．同年奈良先端科学技術大学院大学情報科学研究科助手．平 7 同大情報科学センター助手．分散プロトコルなどに関する研究に従事．博士（工学）．情報処理学会会員．



増澤　利光　（正員）

昭 57 阪大・基礎工・情報卒．昭 62 同大大学院博士後期課程了．同年同大情報処理教育センター助手．同大基礎工助教授を経て，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，平 12 大阪大学大学院基礎工学研究科教授，現在に至る．平 5 コーネル大客員準教授（文部省在外研究員）．分散アルゴリズム，並列アルゴリズム，テスト容易化設計，テスト容易化高位合成に関する研究に従事．工博．ACM，IEEE，EATCS，情報処理学会各会員．