

述語列のリアルタイム収束性の リスポンシブプロトコル設計への適用

川島健一 角田良明 菊野 亨

大阪大学 基礎工学部 情報工学科

〒560 大阪府豊中市待兼山町 1-1

あらまし 情報通信システムの大規模化に伴って通信プロトコルの高信頼性やリアルタイム性が強く要求されるようになってきている。プロセス間の同期のずれにより異常状態に陥ってもリアルタイムに正常状態に回復するための機能を備えているプロトコルをリスポンシブプロトコルと呼ぶ。本稿では、リスポンシブプロトコルの設計法を提案する。リスポンシブプロトコルの設計は、まず、(1) 回復機能を持たないプロトコルの各プロセスに、正常状態に回復するための遷移を付加することにより自己安定化し、次に(2) プロトコルを効率化することによりリアルタイム回復化を行う。リスポンシブブロードキャストプロトコルの設計例では、タイマーを用いることにより自己安定化を実現し、ブロードキャストする木の高さを最小に設定することによりリアルタイム回復化を実現している。

Application of Real-Time Convergence of Predicates Sequences to Responsive Protocols Design

Kenichi Kawashima, Yoshiaki Kakuda and Tohru Kikuno

Department of Information and Computer Sciences

Faculty of Engineering Science, Osaka University

1-1, Machikaneyama-cho, Toyonaka-shi, Osaka 560, Japan

{kawasima, kakuda, kikuno}@ics.osaka-u.ac.jp

Abstract As communication systems become large and complicated, high reliability and high performance are required for communication protocols in presence of faults within protocols. Communication protocols that can make real-time recovery from any abnormal state are called responsive protocols. This paper proposes a new design method of responsive protocols. In this method, first, (1) in order to satisfy self-stabilizing property, some transmissions are added to processes in an original protocol which does not have a recovery function, and then, (2) in order to satisfy real-time convergence property, the protocol is reformed to work more efficiently. In an example of design for a broadcasting protocol, a timer is used in each process for recovery, and then a tree for broadcasting is formed so that the height of tree is minimum.

1 まえがき

情報通信システムの大規模化に伴い、システムに障害が起きて処理をリアルタイムに続行する必要性が高まっている。このためには、リアルタイム性を考慮した通信プロトコルの高信頼化が不可欠である。このようなリアルタイム高信頼化を実現したプロトコルをリスポンシブプロトコル (responsive protocol) と呼ぶ [3, 4, 5]。

異常状態から正常状態への回復を例外処理のルーチンを用いずに行うプロトコルは自己安定プロトコル (self-stabilizing protocol) [2, 9] と呼ばれ、既にいくつかのプロトコルが提案されている。自己安定プロトコルの利点としては、プロセス間の同期がずれた場合に例外処理ルーチンを用いずに解決できる点や、プロセスの初期化をネットワークのグローバルな状態を把握せずにこなせる点がある。リスポンシブプロトコルには、異常状態に陥っても正常状態へ回復する機能の他に、その回復処理をある決められた時間内に行なうリアルタイム回復機能が必要である。

文献 [6, 7] では、述語列のリアルタイム収束性に基づいたリスポンシブプロトコルの検証法が提案されている。検証では、プロトコル内のプロセスやチャネルの各状態を述語として表現し、各異常状態から正常状態へのすべての遷移系列を求めることにより自己安定性を検証している。又、各遷移系列の実行に要する時間が要求される回復時間より小さいかどうかの比較判定を行なうことによりリアルタイム回復性を検証している。本稿では、このような遷移系列のリアルタイム収束性の判定を適用して、リスポンシブプロトコルを設計する方法を提案する。

以下では、2. で本稿で扱う拡張有限状態機械に基づく通信プロトコルとそれに関連する正常状態や異常状態などの諸定義を与える。3. ではリスポンシブプロトコルの形式的な定義を与える。4. では、ブロードキャストプロトコルを例に用いてリスポンシブプロトコルの設計法について論じる。最後に、5. で本研究の結果と今後の課題について述べる。

2 通信プロトコル

2.1 通信プロトコルのモデル化

通信プロトコルは、プロセスとチャネルで構成される通信システムでのメッセージのやりとりを表す通信規約である。本節では、各プロセスを以下に定義する拡張有限状態機械でモデル化し、プロセス間のチャネルを FIFO キューでモデル化することにより通信プロトコルを定義する。

定義 1 プロセス P をモデル化した拡張有限状態機械 PM を次のような 7 項組 $(S, s_I, V, V_I, B, T, \delta)$ で定義する。

$S = \{s_1, \dots, s_n\}$: プロセス P のとり得る状態の有限集合。

s_I : プロセス P の初期状態で S の要素の 1 つ。

$V = \{v_1, \dots, v_m\}$: プロセス P の保持する変数の有限集合。

$V_I = \{v_{I1}, \dots, v_{Im}\}$: 各プロセス変数 v_i の初期値の集合。

$B = \{b_1, \dots, b_o\}$: 真偽値をとる述語の集合 ($TIMEOUT$ も含む)。

$T = \{t_1, \dots, t_p\}$: 命令の集合。各命令 t_k は受信命令 (+message) と送信命令 (-message) のいずれかとプロセス変数の計算を表す。

$\delta = S \times B \times T \rightarrow S$: プロセス P の状態遷移関数。状態 $s_i (\in S)$ で $b_j (\in B)$ が真で、かつ、命令 $t_k (\in T)$ が実行できるとき、遷移 δ は s_i で実行可能であるという。

定義 2 プロセス間のチャネル C は容量に制限のある FIFO キューでモデル化される。以下では、チャネル C の容量を $|C|$ と書く。また、チャネル上の各メッセージは有限個のメッセージ変数の系列 $(mv_1, \dots, mv_m)$ で表わされる。

定義 3 定義 1, 2 に基づいて通信プロトコルを次のような 2 項組 (P, C) で定義する。

$P = \{PM_i | 1 \leq i \leq n\}$: 各 PM_i はプロセス P_i をモデル化した拡張有限状態機械であり、 n はプロセス数を表す。

$C = \{C_{ij} | i \neq j, 1 \leq i, j \leq n\}$: 各 C_{ij} はプロセス P_i と P_j の間のチャネルを表す。ただし、 $|C_{ij}| = 0$ のときは、 P_i から P_j へのチャネルが存在しないことを表す。

2.2 通信プロトコルのグローバル状態

本節では、通信プロトコルの状態を表すためにグローバル状態を定義する。また、グローバル状態を初期状態から正常な遷移系列で到達可能な正常状態と到達不可能な異常状態に分類する。

定義 4 通信プロトコルのグローバル状態は、次の (1), (2) に定義するプロセスのローカル状態 ps とチャネルのローカル状態 cs で表される。

(1) プロセスのローカル状態 $ps = \langle s_i : pred \rangle$

プロセス変数に関する述語 $pred$ が成り立つような状態 s_i にプロセスがあることを ps は表す。ただし、 $\langle s_i : true \rangle$ はプロセス変数には無関係にプロセスの状態が s_i であることを表すものとする。

(2) チャネルのローカル状態 $cs = \langle message_1 : pred_1, \dots, message_m : pred_m \rangle$

チャネル内の各メッセージ $message_i$ のメッセージ変数が $pred_i$ を満たすチャネル状態であることを cs は表す。ここで、 m はチャネル内のメッセージ数 ($1 \leq m \leq |C|$) を表す。また、 $message : true$ はメッセージ変数に制約のない任意のメッセージを表し、 $cs = ()$ はチャネルが空であることを表すものとする。

各プロセス P_i のローカル状態を ps_i 、各チャネル C_{ij} のローカル状態を cs_{ij} とすると、通信プロトコルのグローバル状態は $gs = (ps_1, \dots, ps_n, cs_{12}, \dots, cs_{nn-1})$ と表せる。

文献 [1] では、グローバル変数に基づいた述語が成立する状態としてグローバル状態を定義している。グローバル変数がプロセス変数とメッセージ変数のみで構成される通信プロトコルでは、グローバル変数に基づいた述語はプロセス変数に関する述語の集合とメッセージ変数に関する述語の集合の積と考えることができる。従って、以降では、状態を、プロセス変数に関する述語の集合とメッセージ変数に関する述語の集合の積で表すこととする。

定義 5 グローバル状態の中で次の 2 つの条件 $C1, C2$ を満たすものを初期グローバル状態 (gs_0) もしくは、単に初期状態と呼ぶ。ここで、 s_I はプロセスの初期状態、 $v_{i1} (1 \leq i \leq m)$ は初期値を表す。

条件 C1: $\forall i \ [ps_i = \langle s_I: v_1 = v_{1I} \wedge \dots \wedge v_m = v_{mI} \rangle]$

条件 C2: $\forall i, j (i \neq j) \ [cs_{ij} = \langle \rangle]$

グローバル状態 gs_i で、あるプロセスが $t_j (\in T)$ により遷移 δ を実行可能で、遷移後のグローバル状態が gs_j である時、グローバル状態 gs_i からグローバル状態 gs_j へ正常に遷移可能であると言う。また、グローバル状態 gs_i から正常に遷移可能なグローバル状態の集合を $succ(gs_i)$ と表す。 $succ$ で定義されない遷移は、フォールトによる異常遷移と考える。

定義 6 正常グローバル状態を以下の (1), (2), (3) のように再帰的に定義する。

- (1) 初期状態 gs_0 は正常グローバル状態である。
- (2) グローバル状態 gs_i が正常グローバル状態であるとき、任意の $gs_j (\in succ(gs_i))$ は正常グローバル状態である。
- (3) (1), (2) で得られるもののみが正常グローバル状態である。

正常グローバル状態に含まれないグローバル状態を異常グローバル状態と定義する。フォールトによる異常遷移で正常グローバル状態から異常グローバル状態に入ると考える。以下では、簡単化のために正常グローバル状態と異常グローバル状態を、それぞれ、正常状態と異常状態と呼ぶ。

3 リスポンシブプロトコル

リスポンシブプロトコルは文献 [3, 4] で定義されている。本章では、リスポンシブプロトコルを述語が成立するグローバル状態の遷移系列を用いて形式的に定義する。

定義 7 グローバル状態 gs_i から正常状態に回復するまでに要する最大遷移数 (ms_i) を以下のように再帰的に定義する。ただし、 GS_{normal} はすべての正常状態の集合、 $GS_{abnormal}$ はすべての異常状態の集合を表す。

- $\forall gs_i \in GS_{normal} \ [ms_i := 0]$
- $\forall gs_i \in GS_{abnormal} \left[\begin{array}{l} succ(gs_i) = \phi \rightarrow ms_i := \infty \\ succ(gs_i) \neq \phi \rightarrow ms_i := \max\{ms_j\} + 1 \end{array} \right] \ (gs_j \in succ(gs_i))$

まず、プロトコルの自己安定性を定義する。

定義 8 プロトコルが次の 2 つの条件を満たすとき、そのプロトコルは自己安定性を満たすと定義する。ここでは、 K は正の整数とする。

閉包条件: $\forall gs_i \in GS_{normal} [succ(gs_i) \cap GS_{abnormal} = \phi]$

収束条件: $\exists K \forall gs_i \in GS_{abnormal} [ms_i < K]$

定義 8 の条件 1 は、プロトコルのすべての正常状態に対して、その状態から正常に遷移可能な状態が正常状態であることを示している。つまり、プロトコルが正常状態から遷移を開始した場合、異常状態に陥ることなく、いつまでも正常状態の中を遷移することを表している。条件 2 は、プロトコルが異常状態に陥っても有限時間内に正常状態に回復することを表している。

次に、プロトコルのリアルタイム収束性を定義 9 で定義する。

定義 9 プロトコルが次の条件を満たすとき、そのプロトコルはリアルタイム収束性を満たすと定義する。ただし、 T は正の整数で、プロトコル設計者から要求される遷移数を表す。

リアルタイム収束条件: $\forall gs_i \in GS_{abnormal} \ [ms_i < T]$

定義 9 は、述語が成立するグローバル状態の遷移系列、つまり述語列のリアルタイム収束性を表している。詳しくいえば、プロトコルの各グローバル状態を述語で表して、プロトコルの動作に対応する述語列の系列を考えるとき、異常状態を表す述語から正常状態を表す述語へ、ある系列長 T の範囲内で収束することを表している。

定義 8, 9 に基づいて、リスポンシブプロトコルを次のように定義する。

定義 10 プロトコルが自己安定性 (定義 8) とリアルタイム収束性 (定義 9) を満たすとき、そのプロトコルはリスポンシブプロトコルである。

4 リスポンシブプロトコルの設計

4.1 設計法

本稿で提案するリスポンシブプロトコルの設計は以下の手順で行なわれる (図 1 参照)。

[ステップ 1] プロトコルの実行中にフォールトが発生して異常状態に陥ることはないと仮定して、閉包条件 (定義 8) を満たすように、つまり正常状態内で正しく動作するようにプロトコルを設計する。

[ステップ 2] プロトコルが異常状態に陥った時に、収束条件 (定義 8) を満たすように、つまり正常状態に回復するための遷移を各プロセスに付け加えることにより、自己安定プロトコルに作り変える (自己安定化と呼ぶ)。

[ステップ 3] 自己安定プロトコルがリアルタイム収束性 (定義 9) を満たすように、つまり回復の効率化を図ることにより、リスポンシブプロトコルに作り変える (リアルタイム収束化と呼ぶ)。

ステップ 2 とステップ 3 は、説明の都合上 2 つに分けているが、設計の効率化を図るためにはステップ 3 のリアルタイム収束化も考慮しながらステップ 2 の処理を行なうことが望ましい。

本章では、ネットワーク上に木を張ってその木に沿って同一データをすべてのノードに伝播するブロードキャストプロトコルをリスポンシブ化するための設計例を示す。以下では、4.2 で異常状態からの回復機能を持たない既存のブロードキャストプロトコルを説明する。4.3 では 4.2 で説明したブロードキャストプロトコルの自己安定化の方法を示す。4.4 では自己安定化されたブロードキャストプロトコルのリアルタイム収束化の方法を示す。

4.2 ブロードキャストプロトコル

この節では、ネットワーク上に張られた木に対して根ノードから同一データを伝播するためのブロードキャストプロトコルを説明する。

例 1 図2のような形状のネットワークでノード1からネットワーク上のすべてのノードに対して *data* を伝播する場合、ブロードキャストプロトコルでは図3に示すようにノード1を根とするような木を張り、その木(図4)に沿ってデータを送る。

ブロードキャストプロトコルの動作を以下に示す。

根ノードはブロードキャスト要求があるとすべての子ノードに対して *data* を送り、*ack* 待ち状態に入る。根および葉以外のノードは *data* を受けとると、すべての子ノードに対して親ノードから受けとった *data* を送り、*ack* 待ち状態に入る。葉ノードは、親ノードから *data* を受けとると、*ack* を親ノードに返し、*data* 待ち状態に戻る。根および葉以外のノードは、*ack* 待ち状態ですべての子ノードから *ack* を受けとると親ノードに対して *ack* を返し、*data* 待ち状態に戻る。根ノードはすべての子ノードから *ack* を受けとると *data* 待ち状態に戻りその *data* に関するブロードキャストを終了し、新しくブロードキャスト要求が来るのを待つ。新しくブロードキャスト要求が来ると同様の動作を繰り返す。

プロトコル内の各ノードで実行されるプロセス P_i は以下のように記述できる。以降では、ノードはプロセスと等価な意味で用いる。

• ノード P_i

$S_i = \{wait_data, wait_ack\}$, $s_{iI} = wait_data$ とする。根ノード、根および葉以外の中間ノード、葉ノードの動作を図5 (a)~(c) の状態遷移図で示す。

状態 S_i が *wait_data* のとき、ノード P_i が親ノードからの *data* の到着を待っていることを表す。 S_i が *wait_ack* のとき、ノード P_i が子ノードからの *ack* の到着を待っていることを表す。

このプロトコルはフォールトによる異常遷移が起こらないかぎり正しく動作する。根ノードを除く各ノード P_i と親ノード P_p に関して次の述語 R_i が成り立つ。ただし、 S_p は P_i の親ノード P_p の状態である。

$$R_i : S_i = wait_ack \Rightarrow S_p = wait_ack$$

例 2 図6に正常時の木の各ノードの状態の一例を示す。状態が *wait_ack* であるすべてのノード P_3, P_4, P_5, P_7 (P_1 は除く) の親ノード P_1, P_3, P_5 の状態は *wait_ack* である。よって、根ノードを除くすべてのノードに対して R_i が成立する。

このプロトコルでは、異常状態に陥ったときに正常状態に回復することができない。例えば、*ack* の消失が起こった場合、その *ack* を受けとるべきノードはいつまでもその *ack* を待ち続けデッドロック状態に陥ってしまう。よって、このプロトコルはリスポンシブ性を満たさないプロトコルである。

4.3 自己安定化

本節では、4.2 のブロードキャストプロトコルを自己安定化する方法について述べる。自己安定化を行なうためには、プロトコルが任意の異常状態に陥っても有限時間内に各プロセス P_i について述語 R_i が成り立つようにする必要がある。

文献 [1] では、ノードとその親ノードがそれぞれの状態を相互に知ることができるという仮定を置いて自己安定化

を行なっている。この仮定は、グローバル変数を持つことを意味している。実際の通信プロトコルではこの仮定は許されない。ノードとその親ノードはそれぞれの状態をメッセージの送受信で知らなければならない。

プロトコルの自己安定化は、以下の3つの方針に従って、プロトコル内の各ノードに遷移を加えることにより実現する。

[方針1] ノード P_i が親ノードより送られてきた *data* により述語 R_i が成り立たないと判断したとき、親ノードの状態や変数の値にあわせてノード内の状態や変数の値を修正する。

[方針2] ノード P_i が子ノード P_j より送られてきた *ack* により述語 R_j が成り立たないと判断したとき、その *ack* を無視する。また、決められた時間待っても予期した *ack* が返ってこない場合、*data* の再送を行なうことにより P_j に述語 R_j が成り立っていないことを知らせる。再送のタイミングを決定するために各ノードはタイマーを使用する。*data* を送信するときにタイマーを初期化し、タイムアウトが起こると *data* の再送を行なう。

[方針3] 各ノードが送られてきた *data* が新しく送られてきたものか、再送されたものかを判断できるように、各 *data* にセッション番号をつけて送る。

プロトコル内の各ノードで実行されるプロセス P_i は以下のように記述できる。

• ノード P_i

$S_i = \{wait_data, wait_ack\}$, $s_{iI} = wait_data$, $V_i = \{sn_i\}$, $V_{iI} = \{0\}$,

根ノード、根および葉以外の中間ノード、葉ノードの動作を図7 (a)~(c) の状態遷移図で示す。太線で示した遷移が新しく追加された遷移である。

sn_i はノード P_i が最も新しく受理した *data* のセッション番号を保持する変数であって、親ノードから送られてきた *data* が新しくブロードキャストされたものか、再送されたものかを2値 (0 または 1) で表す。

TIMEOUT は真偽値 (真 または 偽) をとる述語で、タイマーが初期化されて一定時間が経過したときだけ真になる。その時間 T_i はノード P_i から送られた *data* が葉に到達し、*ack* が返ってくるまでの時間よりも十分に長いものと仮定する。

改良したプロトコルはフォールトによる異常遷移が起こらないとき正しく動作し、そのとき根ノードを除く各ノード P_i と親ノード P_p に関して以下の述語 R'_i が成り立つ。ただし、 S_p は P_i の親ノード P_p の状態である。

$$\begin{aligned} R'_i : S_i = wait_ack &\Rightarrow (S_p = wait_ack \wedge sn_i = sn_p) \\ &\wedge (S_i = wait_data \wedge S_p = wait_ack) \Rightarrow sn_i \neq sn_p \\ &\wedge (S_i = wait_data \wedge S_p = wait_data) \Rightarrow sn_i = sn_p \end{aligned}$$

R'_i は、(1) P_i が *ack* 待ち状態ならば P_p も *ack* 待ち状態であつ、 P_i と P_p は同一のセッション番号を持ち、(2) P_i が *data* 待ち状態、かつ、 P_p が *ack* 待ち状態ならば、 P_i と P_p は異なるセッション番号を持ち、(3) P_i が *data* 待ち状態、かつ、 P_p が *data* 待ち状態ならば、 P_i と P_p は同一のセッション番号を持つことを表す。(1) は R_i を含んでいる。

このプロトコルが自己安定性を満たすことは、根を除くすべてのノード P_i において、 R'_i が有限時間内に成立することを示せばよい。証明の詳細は紙面の都合上省略する。

4.4 リアルタイム収束化

ブロードキャストプロトコルの自己安定化は、あるプロセスにフォールトが起こってプロトコルが異常状態に陥ったとき、フォールトが起こったと思われるプロセスが親プロセスに合わせて自分の状態を修正することにより実現されている。このため、もしあるプロセス P_i がフォールトにより状態が変わってしまった場合、そのプロセス P_i がそれに気付いて修正する前にそのプロセスの子プロセス P_j が P_j にフォールトが起こったと判断して誤って自分の状態を修正してしまうことがある。 P_j の子孫も同じように誤った修正を行なう可能性がある。そのため、最悪の場合その影響が葉ノードまで達してしまい、最終的に正常な状態に回復するには、 P_i から葉ノードに向けて順に回復していくのを待たなければならない。従って、最悪時にはあるノード P_i にフォールトが起こった場合に正常状態に回復するのに要する遷移数は P_i を根とする木の高さの3倍となる。

例3 図8(a)に図4の木の高さが4の部分木の一例を示す。この部分木に対して自己安定ブロードキャストプロトコルを適用することによって、正常状態に回復するまでに最多の遷移数を要する系列を図8(b)に示す。状態1はノード P_7 まで data が到着したときに、ノード P_5 にフォールトが起こり P_5 のプロセス状態が ack 待ち状態から data 待ち状態に変化してしまった状況を表している。状態1~9は異常状態で、状態10は正常状態である。正常状態に回復するのに要する遷移数は9である。図8(b)では、述語 R_i^j を満たさないノード P_i とその親ノードの間のリンクを点線で示してある。

プロトコルが正常状態に回復するのに要する時間は、フォールトの起こるプロセスの位置により決まり、最悪の場合、木の高さの3倍の遷移数を要する。このことに着目して、ブロードキャストプロトコルのリアルタイム収束化は以下の方針で行なう。

[方針] リアルタイム収束化を図るためにプロトコルのプロセス自身は変更せず、ブロードキャストする木の形状を変更する。具体的には、ネットワーク上に高さが最小になるように木を張り、その木に対して、4.2で自己安定化したプロトコルを適用する。

例4 図2のネットワークに対して木の高さが最小になるように木を張った一例を図9(a)に示す。このときの木の高さは2となる(図9(b))。例3と同様に、この木の高さが最大の部分木(この場合の木の高さは2)に対してブロードキャストプロトコルを適用したときに正常状態に回復するまでに要する遷移数の最大値は3となる(図9(c))。

4.5 リスponシブプロトコルの検証法

述語列のリアルタイム収束性に基づいたリスponシブプロトコルの検証法は文献[6]で既に提案されている。検証では、プロトコル内のプロセスやチャネルの各状態を述語として表現し、各異常状態から正常状態へのすべての遷移系列を求めることにより、自己安定性を検証している。また、各遷移系列の実行に要する時間が要求される回復時間より小さいかどうかの比較判定を行なうことにより、リアルタイム収束性を検証している。プロトコルと任意の木の形状を与えることにより、文献[6]の方法で自動検証することが可能である。従って、ステップ2およびステップ3

のあとの自己安定性、リアルタイム収束性の検証には、文献[6]の検証法を適用すればよい。

本章では、ブロードキャストプロトコルを例に用いてリスponシブプロトコルの設計法の一例を説明したが、この例に用いた設計法はネットワーク上に木を張って運用されるその他のプロトコル[10]に対しても有効な方法である。

5 あとがき

本稿では、リスponシブプロトコルを形式的に定義し、それに基づいたリスponシブプロトコルの設計法を提案した。リスponシブプロトコルの定義は、プロトコルの自己安定性とリアルタイム収束性を定義することにより行なっている。また、リスponシブプロトコルの設計法では、ブロードキャストプロトコルを例に用いて自己安定化およびリアルタイム収束化の方法について述べた。

今後の課題としては、大規模で複雑な実用プロトコルの設計に適用できる設計法を開発することがある。

文献

- [1] A. Arora and M. Gouda: "Closure and convergence: A formulation of fault-tolerant computing", Proc. of FTCS-22, pp.396-403 (1992).
- [2] M. G. Gouda and N. J. Multari: "Stabilizing communication protocols", IEEE Trans. on Computers, Vol.40, No.9, pp.448-458 (April 1991).
- [3] Y. Kakuda and T. Kikuno: "Issues in responsive protocols design", Proc. of 2nd Int'l. Workshop on Responsive Computer Systems, pp.8-12 (Oct. 1992).
- [4] Y. Kakuda, T. Kikuno, M. Malek and H. Saito: "A unified approach to design of responsive protocols", Proc. the 1992 IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, pp.8-15 (July 1992).
- [5] 角田, 菊野: "リスponシブシステムと通信プロトコル", 信学技報 CPSY 91-73, 実時間処理に関するワークショップ (March 1992).
- [6] 川島, 角田, 菊野: "述語列のリアルタイム収束性に基づくリスponシブプロトコルの検証法", 信学技報 CPSY 91-82, 実時間処理に関するワークショップ (March 1992).
- [7] 川島, 行友, 角田, 菊野: "リスponシブプロトコルのリアルタイム性検証法", 信学会秋全大 D-117 (Sep. 1992).
- [8] M. Malek: "Responsive systems(A challenge for the nineties)", Proc. 16th Symp. on Microprocessing and Microprogramming 30, pp.9-16 (1990).
- [9] N. J. Multari: "Toward a theory for self-stabilizing protocols", Ph.D. Dissertation, University of Texas, Austin (1989).
- [10] A. Segall: "Distributed network protocols", IEEE Trans. on Information Theory, IT-29, 1, pp.23-35 (Jan. 1983).

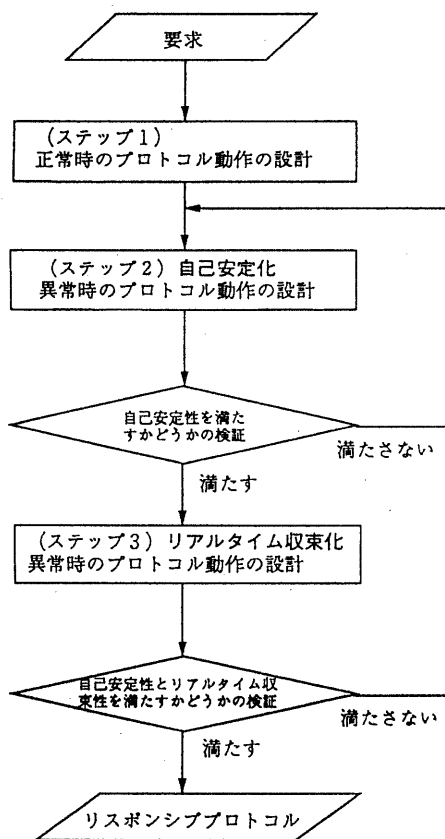


図1 リスポンシブプロトコルの設計手順

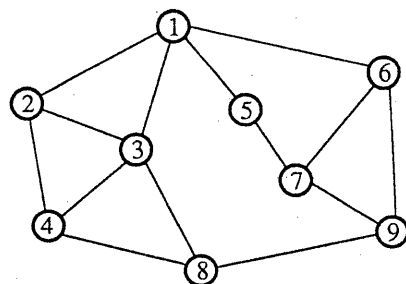


図2 ブロードキャストするネットワーク

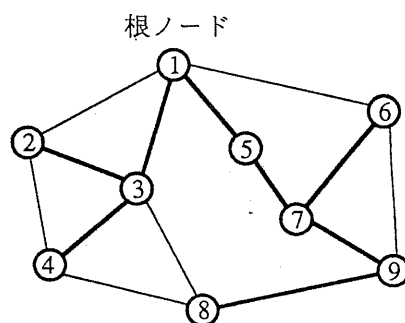


図3 ネットワーク上に張られた
ノード1を根とする木

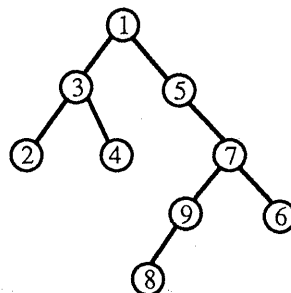
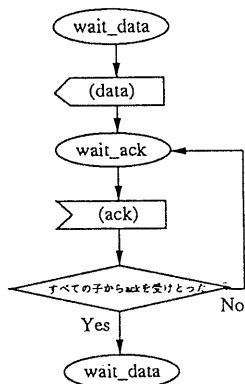
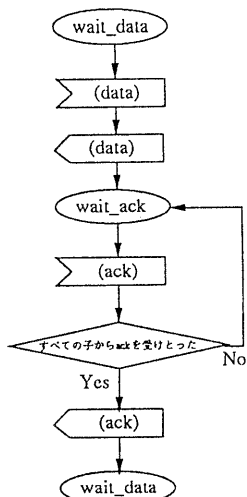


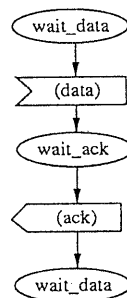
図4 ブロードキャストする木



(a) 根ノードの状態遷移図



(b) 根と葉以外のノードの状態遷移図



(c) 葉ノードの状態遷移図

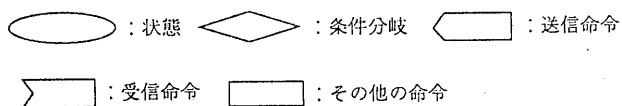


図5 各ノードの状態遷移図

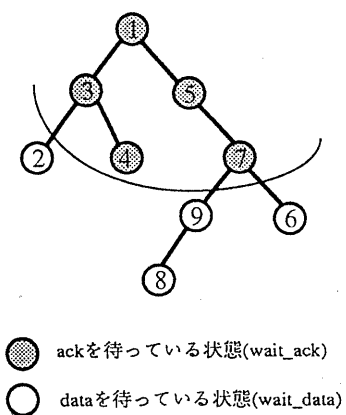


図6 正常時の木の状態

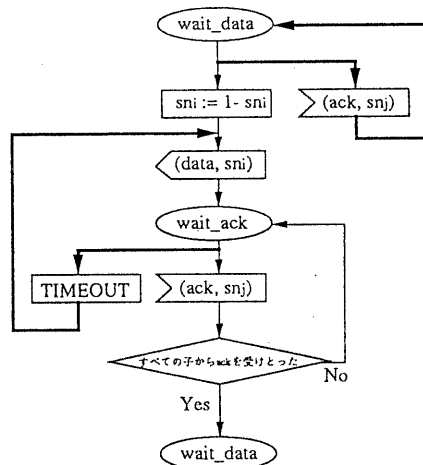


図7 (a) 根ノードの状態遷移図

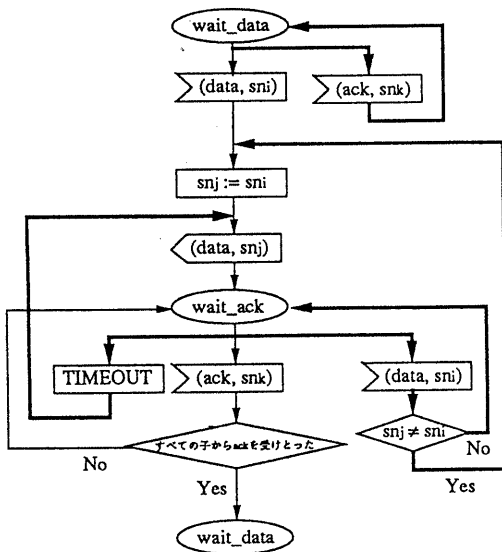


図 7 (b) 根と葉以外のノードの状態遷移図

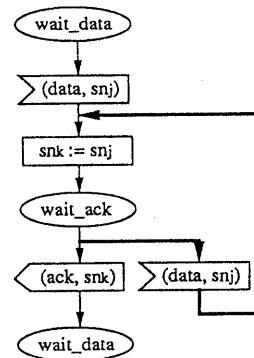


図 7 (c) 葉ノードの状態遷移図

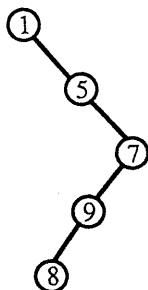


図 8 (a) 高さ 4 の部分木

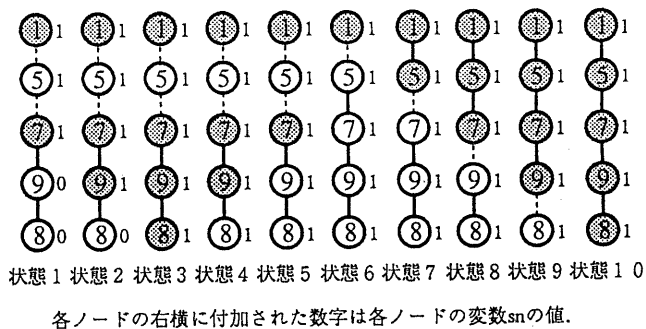


図 8 (b) 状態系列

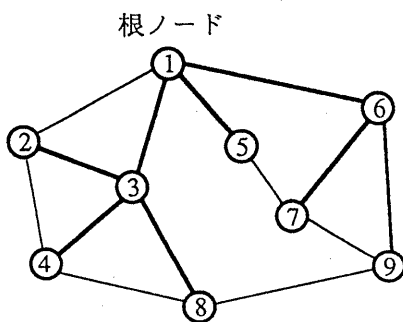


図 9 (a) 高さが最小の木

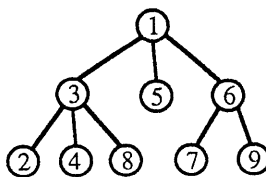


図 9 (b) ブロードキャストする木

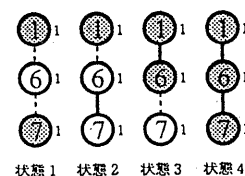


図 9 (c) 状態系列