非同期分散環境における大域状態決定方式

六 沢 一 昭†

終了検出の一方式である WTC（Weighted Throw Counting）方式を応用した大域状態決定方式を述べる．WTC 方式は，プロセスとメッセージに**重み**を持たせ，その合計を制御することによって終了を検出するものである．重みに**世代**を導入することによって，この方式を大域状態決定に応用することができた．同じ世代の重みを持ったプロセスの状態とメッセージの集合が，その世代の大域状態を構成する．プロセスの状態やメッセージを記録したならば，付いていた重みを返却し世代を進める．ある世代の重みがすべて返却された時点における記録内容の集合がその世代の大域状態となる．

Global States Detection in an Asynchronous Distributed Environment

Kazuaki Rokusawa†

This paper proposes a scheme for global states detection in a distributed environment. The scheme is based on the *weighted throw counting* method which is a distributed termination detection scheme. We associate generation with each weight to apply the scheme to global states detection. States of processes and messages with the same generation forms the global states. When processes record their states or received messages, they return back their own weights or the weights of the messages. When all weights with the same generation are returned, the collection of records in all processes forms global states of the generation.

1. はじめに

非同期分散環境では大域的な状態を把握することが難しい．大域時刻が存在しないため，すべてのプロセスがそれぞれの状態などを同じ時刻に記録することができないからである．またメッセージ通信が非同期であるため，送信はなされたが受信されていないメッセージが存在し，これが処理を難しくしている．たとえば，すべてのプロセスの状態を集めただけでは大域的な状態にはならない．

大域状態とは，すべてのプロセスの状態とメッセージの辻褄のあった集合である☆.「辻褄があう」というのは，プロセスの状態とメッセージの「因果関係が正しい」ということである．たとえば，メッセージを受信してプロセスの状態が変化した場合,「変化前のプロセスの状態と受信メッセージ」あるいは「変化後のプロセスの状態のみ」を大域状態の要素とするのは辻褄があっている．これに対して「受信メッセージと，変化後のプロセスの状態」を大域状態とすると辻褄があわなくなる．

大域状態決定とは上述した大域状態を求める処理である．この処理は通常処理の上に重ねる形で実装するべきものである．通常処理を妨害してはいけない．たとえば通常処理のメッセージ送信を中断させるなどしてはいけない．

大域状態決定の主な応用はフォールトトレラントの実現である．システムに異常が起こったとき，それ以前に求まっている大域状態にシステムを戻す（rollback）ことによりフォールトトレラントを実現できる．また，並列プログラムのデバッグにも応用することができる．大域状態決定により，各プロセスの変数やメッセージなどの辻褄のあった集合を様々な時点で得ることができる．変数の値をたどることによって逐次プログラムのデバッグを行うことがあるが，これと同様に，すべてのプロセスの変数の値とメッセージをたどることによって並列プログラムのデバッグを行うことができる．

大域状態を決定する方式はこれまでに数多く提案されている．文献 1) の方式はその代表的なものであり，これをベースに非常に多くの方式が生まれている．文

† 千葉工業大学情報工学科
Department of Computer Science, Chiba Institute of Technology

☆ 大域状態の厳密な定義は文献 1) でなされている．

献 2) は最も新しい文献の 1 つであり，移動コンピューティングにおける方式を述べている．

本稿では，WTC（Weighted Throw Counting)[3] 方式を応用した大域状態決定方式を提案する☆．WTC 方式は，終了検出方式の 1 つである．プロセスとメッセージに重みを持たせ，その合計を制御することによって終了を検出する．**重みに世代を導入**することによってこの方式を大域状態決定に応用することができた．

以下，まず計算モデルを述べる．次に，大域状態を説明し，その決定の難しさを考察する．そして，従来方式を示した後，WTC 方式を説明し，提案する方式を述べる．

2. 計算モデル

仮定する計算モデルを以下に示す．

- 有限個のプロセスが論理的な通信チャネルで互いに結合している☆☆．
- **非同期メッセージ通信**によって各プロセスは通信し合う．メッセージは任意の有限時間内に到着する．通信チャネル上で**メッセージの追い越し**の起こることがある．送信された順序に到着する保証はない．
- 各プロセスは他のすべてのプロセスへメッセージを送ることができる．また，自分の状態と，受信したメッセージを（大域状態の構成要素として）記録することができる．
- **検出プロセス**という特別なプロセスが存在し☆☆☆，大域状態が求まったことを検出する．検出プロセスと各プロセスは互いにメッセージ通信を行うことができる．
- 大域状態を求める処理は，何度でも繰り返されるが，オーバラップしない．ある時点の処理が終了しないうちに，次の処理を開始することはない．

なお,「送信されたがまだ受信されていないメッセージ」を以下では**通信中のメッセージ**と呼ぶ．

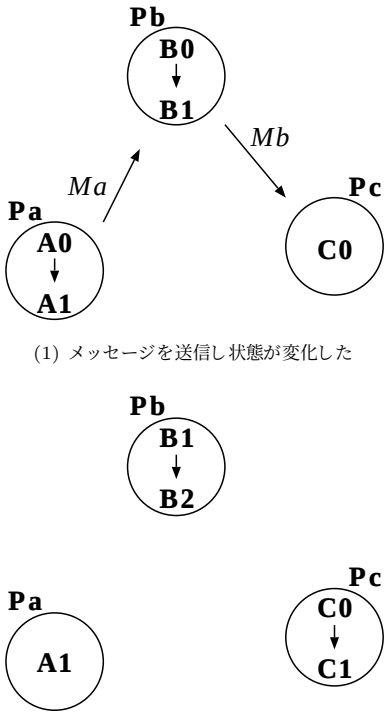
3. 大域状態とは

プロセスと通信チャネルの状態の**辻褄のあった**集合が大域状態である．

ある通信チャネルの状態とは，

- そのチャネルの送信側プロセスが自分の状態を**記録する前に送信**し，
- 受信側プロセスが自分の状態を**記録した後に受信**した

☆ 本稿は，文献 4) の改良改定版である．
☆☆ 物理的な通信路については何も仮定しない．
☆☆☆（通常の）プロセスが検出プロセスを兼ねてもよい．



(1) メッセージを送信し状態が変化した

(2) メッセージを受信して状態が変化した

図 1 プロセス群とメッセージの例
Fig. 1 Examples of processes and messages.

メッセージの集合である.「状態を記録した後に送信したメッセージ」や「状態を記録する前に受信したメッセージ」ではない．これらを含むと辻褄があわなくなる．以下に大域状態の具体例を示す．

図 1，**図 2** では以下の現象が起きている．

(1) プロセス **Pa** はメッセージ *Ma* を送信して状態が **A0** から **A1** に変化した．**Pb** は *Mb* を送信して **B0** から **B1** に変化した．

(2) **Pb** は *Ma* を受信して **B1** から **B2** に変化した．**Pc** は *Mb* を受信して **C0** から **C1** に変化した．

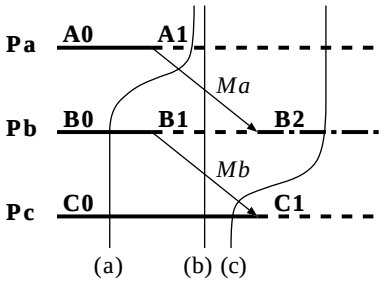
これらの状況変化において以下はいずれも**正しい**大域状態である（図 2 (1) 参照）．

(a) **Pa:A1**, **Pb:B0**, **Pc:C0**, **Cab:***Ma*
(b) **Pa:A1**, **Pb:B1**, **Pc:C0**, **Cab:***Ma*, **Cbc:***Mb*
(c) **Pa:A1**, **Pb:B2**, **Pc:C0**, **Cbc:***Mb*

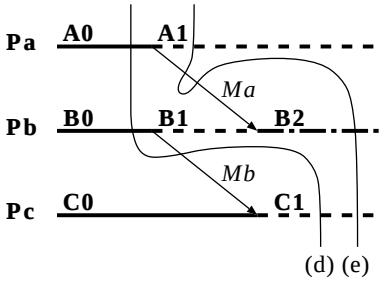
（**Pa:**，**Pb:**，**Pc:** はプロセスの，**Cab:**，**Cbc:** は通信チャネルの状態を示す.）

一方，以下は大域状態としていずれも**正しくない**（図 2 (2) 参照）．

(d) **Pa:A0**, **Pb:B0**, **Pc:C1**, **Cbc:***Mb*
(e) **Pa:A1**, **Pb:B2**, **Pc:C1**, **Cab:***Ma*



(1) 正しい大域状態の例



(2) 正しくない大域状態の例

図 2 大域状態の例
Fig. 2 Examples of global states.

(d) は，メッセージ *Mb* を通信チャネルの状態として含んでいるが，これは，プロセス **Pb** が自分の状態を（**B0** であると）**記録した後に送信**したものである．(e) は，*Ma* を含んでいるが，これは，**Pb** が自分の状態を（**B2** であると）**記録する前に受信**したメッセージである．

4. 大域状態決定の難しさ

大域状態は,「自分の状態」と「（自分への）通信チャネルの状態」を各プロセスが記録することにより求めることができる．

「自分の状態」の記録は容易である．検出プロセスが記録指示メッセージ（*record*）をブロードキャストし，*record*を受信したプロセスは受信時の自分の状態を記録すればよい．難しいのは,「通信チャネルの状態」である.「通信中のメッセージの存在」と「メッセージ追い越しの可能性」が処理を難しくしているのである．

以下に，各プロセスが自分にかかわる状態を記録する上での課題を述べる．

送信が状態記録前/後の判別

record を受信し，大域状態の構成要素として自分の状態を記録したプロセスを考える．このプロセスが他のプロセスからメッセージを受信したとき，そのメッセージが「送信元プロセスが自分の状態を記録する**前**に送信した」ものであるか,「記録した**後**に送信した」

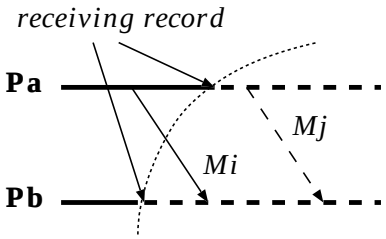


図 3 送信が状態記録前/後の判別
Fig. 3 Before/after message sending.

ものであるかが判別できなくてはならない．前者は，大域状態の構成要素なので，**記録しなければならない**が，後者は，大域状態の構成要素ではないので，**記録してはいけない**からである．

たとえば，**図 3** でプロセス **Pb** は *record* 受信後にメッセージを 2 つ（*Mi* と *Mj*）**Pa** から受信しているが,「*Mi* は **Pa** が状態を記録する**前**に，*Mj* は状態を記録した**後**に送信した」ことが分からなくてはならない．

決定処理の終了検出

大域状態決定処理の終了（大域状態が求まったこと）を検出プロセスが検出できなければならない．この検出ができないと，求まった（はずの）大域状態を利用することができない．また，処理に使った制御用データなどを解放することもできない．

すべてのプロセスで「自分の状態」が求まったことは容易に検出できる．各プロセスは,「自分の状態」の求まった（求めた）ことが自分で分かる．このため，検出プロセスが，すべてのプロセスからその旨の連絡を受ければよい．

これに対して「通信チャネルの状態」は単純でない．この状態を確定するには,「状態記録前に送信されたメッセージのすべてが宛先プロセスに到着したこと」を検出する必要がある．たとえば図 3 では，*Mi* のようなメッセージがすべて **Pb** に到着したことを検出できなければならない．メッセージの追い越しがあるかもしれない，たとえば図 3 で *Mj* が *Mi* より先に到着するかもしれないので，順序関係を利用することができない．状態記録後に送信されたメッセージを受信しても，その送信元が状態記録前に送信したメッセージをすべて受信したとは限らないのである．このため，（自分への）通信チャネルの状態が確定したことを各プロセスは自分で分かることができない．

5. 従来方式

まず，従来の代表的な方式である「追い出しメッセー

ジ（を用いて通信チャネルからメッセージを追い出す）方式[1)]」を示す．次に，多くの従来方式が用いているタイムスタンプについて述べる．

5.1 追い出しメッセージ方式

この方式は，追い出しメッセージ（*marker*）をすべての通信チャネルに流すことにより，状態記録前/後の判別と終了検出を行うものである．メッセージの追い越しのない環境を仮定し，到着順序を本質的に利用している．

各プロセスは，自分の状態を記録したらすべての通信チャネルに *marker* を流す．あるチャネルからメッセージを受信した場合，そのチャネルから *marker* をまだ受信していないのであれば，そのメッセージは状態記録前送信なので記録する．一方，*marker* 受信済みならば，記録後送信なので記録しない．

すべての通信チャネルから *marker* を受信したプロセスは，状態記録前に送信されたメッセージをもはや受信することがない．そのプロセスによる状態記録処理は終了したといえる．すべてのプロセスがこの状況になると大域状態決定処理は終了したことになる．

前述したように，この方式はメッセージの到着順序を本質的に利用している．このため，そのままでは，メッセージの追い越しの起こりうる環境には使えない．また，すべての通信チャネルに，たとえそこにメッセージが流れていなくとも，*marker* を流すため，通信チャネルの数に比例した数の *marker* 通信が発生してしまう．

5.2 タイムスタンプ

プロセスとメッセージに仮想的な時刻を付けると，事象の順序付けが可能になり，状態記録前/後を判別することができる．この「仮想的な時刻」が**タイムスタンプ**である．

タイムスタンプとしては，通常，**ベクトル時刻**[6)]あるいはこれをベースにした時刻が用いられる．ベクトル時刻は,「自プロセスから見た他プロセスの時刻」と「自プロセスのローカルな時刻」の集合をベクトルで表したものである．n 個のプロセスが存在した場合，i 番目のプロセスが保持する時刻は

$$\langle v_i[0], \ldots, v_i[i], \ldots, v_i[n-1] \rangle$$

となる．ここで $v_i[j]\,(i \neq j)$ は他プロセスの，$v_i[i]$ は自プロセスの時刻である．

タイムスタンプの問題点は，時刻（を構成するベクトル）の大きさがプロセス数に依存することである．プロセスが増えると，メッセージの運ぶデータも，それに従って増えてしまうのである．

6. WTC 方式

ここでは，終了検出方式の 1 つである WTC（Weighted Throw Counting）方式[3)]を簡単に説明する．

WTC 方式は，分散 GC 方式の 1 つである WRC（Weighted Reference Counting）方式[7),8)]をプロセス管理に応用したものであり☆，**重み**を用いて終了検出を行う☆☆．到着確認メッセージを用いたり[10)]，観測を繰り返す[11),12)]ことなくプロセス群の終了を効率良く検出することができる．

この方式は並列記号処理言語の分散メモリ処理系実装に用いられており[13),14)]，簡単な定量的評価も行われている[15)]．また，大域 GC のマーキング終了検出にも使用されており[16),17)]，評価もなされている．

WTC 方式では，すべてのプロセスと通信中のメッセージ，それに検出プロセスが**重み**を持つ．プロセスとメッセージの重みは正の整数，検出プロセスの重みは負の整数である．そして

「すべての重みの合計がゼロ」

を保つように以下の制御を行う．

- プロセスや検出プロセスがメッセージを送信するときは，メッセージに（正の整数の）重みを付け，自分の重みはその分だけ減らす．メッセージを受信したら，メッセージに付いていた重みを自分の重みに加える．
- プロセスは終了すると，自分の重みをすべて検出プロセスへ返却する．

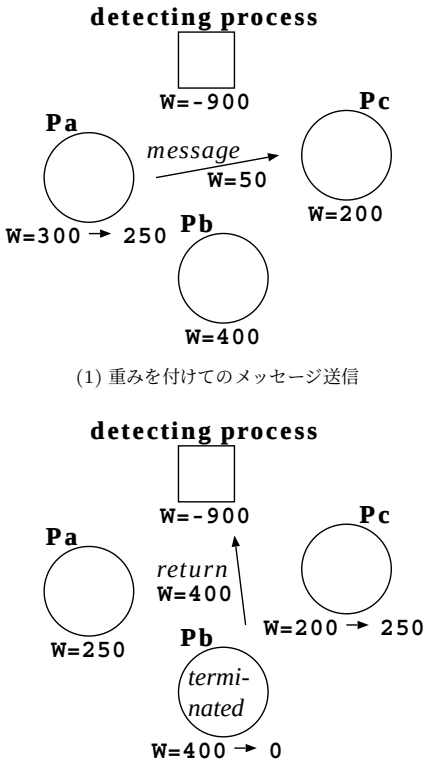
上記制御を行うと，検出プロセスの重みがゼロならば，「すべてのプロセスが終了し通信中のメッセージも存在しない」ことが保証される．

WTC 方式における場面の例を**図 4** に示す．(1) で，プロセス **Pa** はメッセージ（*message*）に重み 50 を付けて送信した．その結果，自分の重みは 300 から 250 に変化した．(2) で，**Pc** はメッセージを受信し，付いていた重み 50 を自分の重みに加えた．一方，**Pb** は終了し，自分の重みをすべて検出プロセス（**detecting process**）に返却した．*return* は，重みを返却するメッセージである．

7. 提案する方式

まず，前述した課題を解決する方法を述べ，そして方式全体の処理を示す．

☆ 分散 GC 方式から終了検出方式の導出は文献 9) が述べている．
☆☆ 表現形式は異なるが本質的には同じ方式を文献 5) が述べている．



(1) 重みを付けてのメッセージ送信

(2) メッセージの受信とプロセスの終了

図 4 WTC 方式
Fig. 4 The WTC scheme.

7.1 課題の解決

重みに世代を導入することによって，状態記録前/後の判別と，決定処理の終了検出を可能にした．

送信が状態記録前/後の判別

各プロセスは，*record* を受信して自分の状態を記録したら自分の重みの世代を進める．この結果，状態記録後に送信されるメッセージに付く重みの世代は新しいものとなる．一方，状態記録前に送信されたメッセージの重みの世代は前のままである．図 3 では，*Mi* の重みには前の世代（実線）が，*Mj* には新しい世代（破線）が付く．

他のプロセスからメッセージを受信したとき，そのメッセージの重みの世代を自分の重みの世代と比較することにより，状態記録前/後を判別することができる．メッセージが前の世代ならば状態記録前，同じ世代ならば状態記録後である．これはメッセージの追い越しに影響されることはない．

さて，*record* を受信して重みの世代を進めた後に送信したメッセージが，宛先プロセスに *record* **よりも先に**到着することが起こりうる．たとえば図 3 で，*Mj* が **Pb** に *record* よりも先に到着する可能性がある．したがって，プロセスは新しい世代の重みを持った（*record*以外の）メッセージを受信する可能性がある．このため，プロセスは 3 世代（自分より前/同じ/新しい）のメッセージ（**図 5** の *Mi*, *Mj*, *Mk*）を受信しうる．ただし処理のオーバラップがないため，前世代と新世代のメッセージが同居することはない．前世代についての大域状態が求まってから，次の大域状態を求めるべく新しい *record* が送信され，それを受けて新世代のメッセージが送信されるからである．図 5 では *Mk* の後に *Mi* を受信することはない．

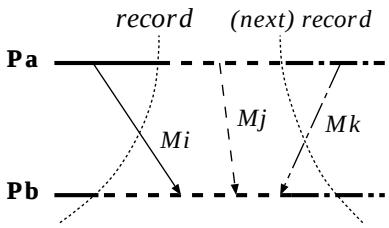


図 5 プロセスが受信する 3 世代のメッセージ
Fig. 5 Three generations of received messages.

この新世代メッセージの受信は，検出プロセスから *record* が送信されていること（大域状態決定処理の始まっていること）を意味する．*record* が送信されないと世代は進まないからである．したがって，新しい世代のメッセージを受信した場合は，*record* を受信したときと同じ処理を行えばよい．図 5 で，**Pb** は *Mk* 受信によって世代を進めている．

決定処理の終了検出

同じ世代の重みを持ったプロセスの状態とメッセージが（その世代の）大域状態を構成する．

プロセスは自分の状態を記録したら自分の重みを検出プロセスへ返却する．メッセージを記録した場合もやはりその重みを検出プロセスへ返却する．記録済みプロセスおよび記録済みメッセージの重みを返却するのである．図 3 において **Pb** は，*record* 受信時に自分の重みを，*Mi* 受信時には *Mi* の重みを検出プロセスへ返却する．

一方，検出プロセスは，同一世代のすべての重みの返却を WTC 方式により確認できる．ある世代の重みがすべて返却されたということは，その世代のプロセスの状態とメッセージがすべて記録されたこと，すなわち大域状態が決定できたことを意味する．

7.2 全体の処理

検出プロセスと各プロセスそれぞれの処理を述べる．

重みに付く世代は初期状態においてすべて等しいものとする．プロセスどうしが通常処理を行うために送受信するメッセージを以下では**通常メッセージ**と呼ぶ

ことにする．

検出プロセスの処理

- **決定処理の開始：** 以下の処理を行う．
  (1) **旧世代重みカウンタ**を用意し，自分の重みで初期化する．このカウンタの値は「大域状態としてまだ記録されていないプロセスと通常メッセージの重みの合計」を示す．
  (2) 自分の重みをゼロに設定し，世代を 1 つ進める．
  (3) すべてのプロセスへ *record* を送信する．各 *record* には重みを付け，自分の重みはその分だけ減らす．

  すべてのプロセスへの *record* の送信が終わった段階では，検出プロセスの重みと *record* に付けた重みの合計がゼロになっている．
- ***return* を受信したら，** *return* の重みを旧世代重みカウンタに加える．
- **決定処理の終了検出：**（*return* の重みを加えた結果）旧世代重みカウンタの値がゼロになったら大域状態が決定できたこと，すなわち，大域状態を構成するすべてのプロセスの状態と通常メッセージの内容の記録されたことが確認される．

各プロセスの処理

- *record* **を受信したら**，*record* の重みの世代と自分の重みの世代を比較する．
  - *record* **の方が新しい**ならば，自分の状態を記録し，自分の重みを *return* で検出プロセスへ返却する．そして自分の重みを *record* の重みで再設定する．これによって自分の重みの世代が進む．
  - **2 つの世代が等しい**ならば☆，*record* の重みを自分の重みに加える．
  - **自分の方が新しい**ならば☆☆，*record* の重みを *return* で検出プロセスへ返す．
- **通常メッセージを受信したら**，そのメッセージの重みの世代と自分の重みの世代を比較する．
  - **自分の方が新しい**ならば，メッセージの内容を記録し，メッセージの重みを *return* で検出プロセスへ返却する．
  - **2 つの世代が等しい**ならば，メッセージの重みを自分の重みに加える．
  - **メッセージの方が新しい**ならば，自分の状態を記録し，自分の重みを *return* で検出プロセスへ返す．そして自分の重みをこのメッセージの重みで再設定する．この結果，自分の重みの世代が進む．なお，これは前述した「新しい世代の *record* を受信したとき」と同じ処理である．

図 6，**図 7** は大域状態決定の例である（図 7 は，図 6 (1)〜(4) を 1 つにまとめたものである）．

図 6 において，`Old` は**旧世代重みカウンタ**である．また，`W=300o` は前の世代の重み `300` を，`W=500n` は新しい世代の重み `500` を示す．図 6，図 7 において，*record(B1)* および *record(M2)* は，自プロセスの状態 **B1** を記録したことおよび受信メッセージ *M2* を記録したことを示す．

図 6 (1) は処理開始場面である．(2) で *record* を受信したプロセス **Pb**，**Pc** は，自分の状態を記録し，自分の重みを返却した．(3) で，前の世代の重みを持った通常メッセージ *M2* が **Pc** に到着し，**Pc** はこれを記録して重みを返却した．(4) で，新しい世代の通常メッセージ *M3* が，まだ *record* を受信していない **Pa** に到着し，**Pa** は自分の状態を記録し，前の世代の自分の重みを返却した．

7.3 検　　討

ここでは，処理が終了することの保証と，必要な世代の数について検討する．

7.3.1 処理が終了することの保証

旧世代重みカウンタがゼロになったら大域状態決定処理終了である．したがって，このカウンタが有限時間内にゼロになることを示せば，処理の終了を保証できる．

旧世代重みカウンタは，返却されていない前の世代の重みを示している．前の世代の重みには以下の 2 つがある．

- *record* 受信前のプロセスの重み．
- *record* 受信前のプロセスが送信した（通信中の）通常メッセージの重み．

前者は，このプロセスに *record*（あるいは新世代メッセージ）が到着すると検出プロセスに返却される．後者は，この通常メッセージが *record* 受信前のプロセスに到着すると前者と同じものになり，*record* 受信後のプロセスに到着するとただちに返却される．

メッセージは有限時間内に宛先に到着するので，上述した重みはすべて有限時間内に検出プロセスに返却される．このため，旧世代重みカウンタは有限時間内にゼロになる．

☆ 新しい世代の通常メッセージ受信によって大域状態決定処理を開始した後に *record* を受信した場合である．

☆☆ 新しい世代の通常メッセージ受信によって大域状態決定処理を開始し，大域状態が求まり，さらに次の大域状態決定処理が始まってから *record* を受信した場合である．

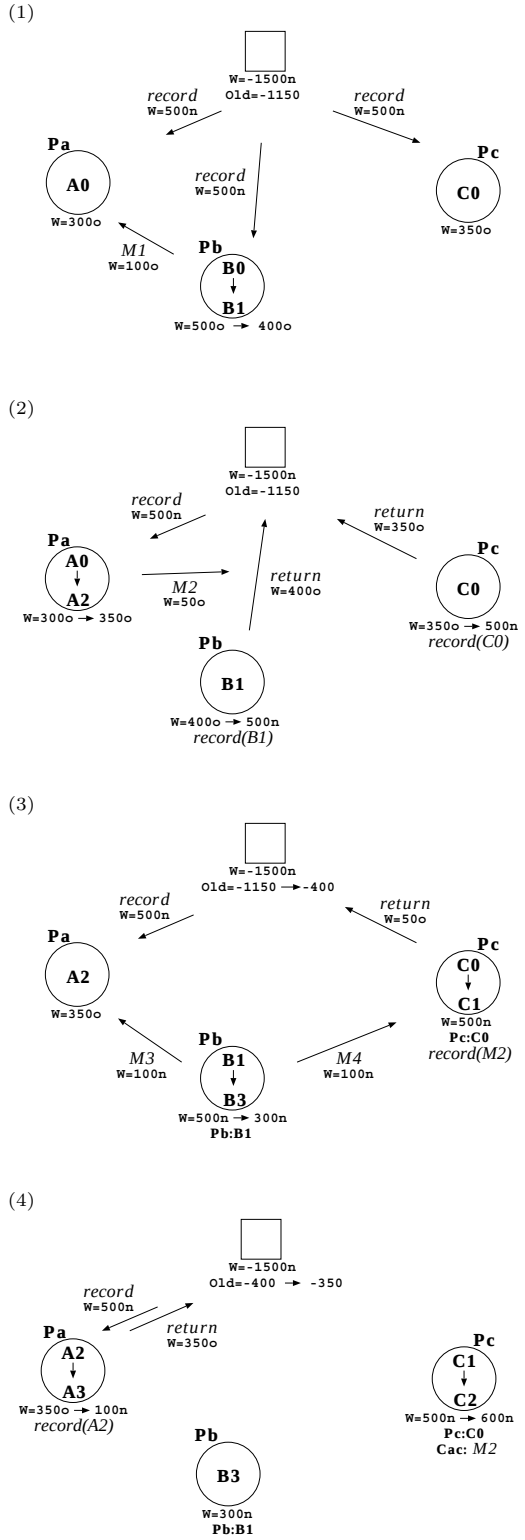


図 6 大域状態決定の例
Fig. 6 An example of global state detection.

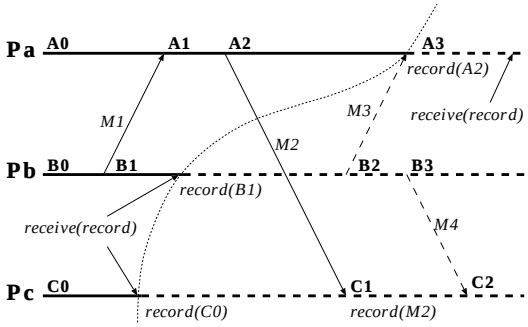


図 7 大域状態決定の例（1 つにまとめたもの）
Fig. 7 An example of global state detection (summary).

7.3.2 必要な世代の数

世代は単調増加していくものである．しかし，大域状態決定処理中（検出プロセスが決定処理を開始してから終了が検出されるまで）に存在する世代は 2 つしかない.「検出プロセスと同じ世代」と「前の世代」だけである．

プロセス側は，処理中（*record* あるいは新世代メッセージを受信してから終了が検出されるまで）は,「自分より前の世代」と「同じ世代」のメッセージを受信し，処理の終了後は,「同じ世代」と「新しい世代」のメッセージを受信する．

このため，世代は 2 つで足りそうに思える．プロセス側では，自分と異なった世代が，処理中は「前の世代」，終了後は「新しい世代」と考えられるからである．

ところが，プロセスは，処理中と終了後を区別することができない．決定処理の終了したことが分からないからである．*record* あるいは新世代メッセージの受信により，次の決定処理の始まったことを知り，そして，前の決定処理が終了していたことを認識するのである．このため 3 つの世代を設ける必要がある☆．

これは，世代は,「論理的には単調増加であるが，実装上は 3 つで足りる」ということである．たとえば，世代として a, b, c を設け，新旧を $a < b, b < c, c < a$ と定義し，$a \to b \to c \to a \to b \ldots$ と変化させればよい．

7.4 従来方式との比較

本方式は，5 章で述べた従来方式と比べて以下の利点を持つ．

- メッセージの到着順序を利用していないため，メッ

☆ 大域状態が求まったら，その旨を検出プロセスがすべてのプロセスへ伝えるのであれば，世代は 2 つで済む．しかしこの方式では，プロセスの数の 2 倍のメッセージ（伝えるメッセージと到着確認メッセージ）が新たに必要になる．

セージの追い越しが起こりうる環境にもそのまま適用することができる．

- 大域状態を決定するがために使用する情報は重みと世代だけである．これらの大きさはプロセス数に依存しない．このため，プロセスが増えても，各プロセスが記憶しメッセージが運ぶデータの大きさは増加しない．

8. 考　　察

世代の導入について

WTC 方式への世代の導入は，文献 18) が述べる分散プロセスの状態変更方式において著者らが行ったものである．プロセスの状態に世代を付加して状態変更前/後を判別した．また，文献 17) が述べる大域 GC 方式では，1 ビットの色（color）をメッセージに付けて,「送信が GC の開始前/後の判別」を実現している．

これらに対し，本方式は重みへ世代を付加したものであり，WTC 方式への世代の導入としてより一般的だといえよう．重みに世代を付加することによって，終了という一度しか起こらない状態変化の検出ではなく，状態制御という何度でも繰り返す処理へ WTC 方式を応用できたのである．

必要なメッセージ数

本方式では，*record* と *return* が大域状態決定を行うがために送受信されるメッセージである．*record* は処理の開始時にすべてのプロセスへ送るだけである．したがって，*record* の送信数はプロセスの数に等しい．一方，*return* は，自分の重みを返却するときと，受信した通常メッセージの重みを返却するときに送信する．前者の数はプロセスの数に等しく，後者は，一般に，プロセスの数に比例する．

以上のことから，本方式ではプロセス数に比例した数のメッセージで大域状態を決定することができるといえよう．

9. お わ り に

非同期分散環境における終了検出方式の 1 つである WTC（Weighted Throw Counting）方式を応用した大域状態決定方式を提案した．

重みに世代を導入したことによって，メッセージの「記録すべき/記録してはいけない」の判別が可能になり，また，終了検出と同じ原理で，大域状態決定処理の終了を検出することができた．この結果，メッセージの到着順序を利用したり，タイムスタンプを用いたりすることなく，大域状態が決定できた．

本方式はメッセージの追い越しに対処でき，プロセス数に比例した数のメッセージで大域状態を決定することができる．

謝辞　ICOT において，大域状態についてともに検討した（株）三菱総合研究所市吉伸行氏，東京大学近山隆教授に深謝いたします．豊橋技術科学大学中島浩教授からは，世代の数にかかわる鋭いコメントをいただきました．感謝いたします．

参 考 文 献

1) Chandy, K.M. and Lamport, L.: Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Trans. Computer Systems*, Vol.4, No.1, pp.63–75 (1985).
2) Higaki, H. and Takizawa, M.: Checkpoint-Recovery Protocol for Reliable Mobile Systems, 情報処理学会論文誌，Vol.40, No.1, pp.236–244 (1999).
3) Rokusawa, K., Ichiyoshi, N., Chikayama, T. and Nakashima, H.: An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems, *Proc. Intl. Conf. on Parallel Processing*, Vol.I, pp.18–22 (1988).
4) 六沢一昭：非同期分散環境における大域状態決定方式，第 9 回並列処理シンポジウム JSPP '97, pp.61–68 (1997).
5) Mattern, F.: Global Quiescence Detection Based on Credit Distribution and Recovery, *Inf. Proc. Lett.,* Vol.30, No.4, pp.195–200 (1989).
6) Mattern, F.: Virtual Time and Global States of Distributed Systems, Cosnard, M. and Quinton, P. (Eds.), *Parallel and Distributed Algorithms*, North-Holland, pp.215–226 (1989).
7) Watson, P. and Watson, I.: An Efficient Garbage Collection Scheme for Parallel Computer Architectures, *Proc. Parallel Architectures and Languages Europe*, LNCS, Vol.259, Vol.II, pp.432–443 (1987).
8) Bevan, D.I.: Distributed Garbage Collection Using Reference Counting, *Parallel Computing*, Vol.9, No.2, pp.179–192 (1989).
9) Tel, G. and Mattern, F.: The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes, *Proc. Parallel Architectures and Languages Europe*, LNCS, Vol.505, Vol.I, pp.137–149 (1991).
10) Shavit, N. and Francez, N.: A New Approach to Detection of Locally Indicative Stability, *Proc. Intl. Colloquium on Automata, Languages and Programming*, LNCS, Vol.226, pp.344–358 (1986).
11) Dijkstra, E.W., Feijen, W.H.J. and van

Gasteren, A.J.M.: Derivation of a Termination Detection Algorithm for Distributed Computations, *Inf. Proc. Lett.,* Vol.16, No.5, pp.217–219 (1983).
12) Mattern, F.: Algorithms for Distributed Termination Detection, *Distributed Computing*, Vol.2, pp.161–175 (1987).
13) Nakajima, K., Inamura, Y., Ichiyoshi, N., Rokusawa, K. and Chikayama, T.: Distributed Implementation of KL1 on the Multi-PSI/V2, *Proc. Intl. Conf. on Logic Programming*, pp.436–451 (1989).
14) Rokusawa, K., Nakase, A. and Chikayama, T.: Distributed Memory Implementation of KLIC, *Proc. 11th Intl. Conf. on Applications of Prolog*, pp.10–17 (1998).
15) Rokusawa, K., Nakase, A. and Chikayama, T.: Quiescence Detection in a Distributed KLIC Implementation, *Proc. EURO-PAR '95 Parallel Processing*, LNCS, Vol.966, pp.539–551 (1995).
16) 六沢一昭，中島克人，市吉伸行，稲村　雄：大域 GC における WTC 方式を用いたマーキング終了検出，第 43 回情報処理学会全国大会論文集，5L-11 (1991).
17) Kamata, T., Matsuoka, S. and Yonezawa, A.: Efficient Parallel Global Garbage Collection on Massively Parallel Computers, *Proc. Supercomputing '94*, pp.79–88 (1994).
18) 六沢一昭，市吉伸行：ブロードキャストと WTC 方式を用いた分散プロセス制御方式，情報処理学会論文誌，Vol.33, No.3, pp.320–329 (1992).

(平成 11 年 9 月 3 日受付)
(平成 12 年 3 月 2 日採録)

六沢　一昭（正会員）

1981 年早稲田大学理工学部電子通信学科卒業．1983 年早稲田大学大学院理工学研究科電気工学専攻修士課程修了．同年沖電気工業（株）入社．1986〜1990 年および 1993〜1995 年（財）新世代コンピュータ技術開発機構（ICOT）に出向．1998 年（財）日本情報処理開発協会先端情報技術研究所客員研究員．1999 年千葉工業大学情報工学科助教授．博士（工学）．主として，論理プログラミング，並列記号処理言語の処理系実装とプログラミング環境の研究に従事．電子情報通信学会，日本ソフトウェア科学会，人工知能学会各会員．