計算機アーキテクチャ 89−17
（1991. 7. 19）

# Paradise: 分散環境向けデバッグ環境
## — 分散プロセスの遅延停止について —

川邉 恵久　　山下 一郎　　守屋 康正
富士ゼロックス システム技術研究所

プロセス間通信をRPCに限定した分散システムにおいて, ブロードキャスト通信を必要としない分散プロセスの遅延中断アルゴリズムを提案した. さらに分散システムとデバッガについてモデル化, および形式化を行なった上で, このアルゴリズムがブレークポイントによる中断をしてもデバッグしている分散プログラムの振舞いを保存することを示すことで正当性の検証を行なった. またRPC検出機能を組み込んだ分散デバッガにこの分散遅延アルゴリズムと等価な実装向けアルゴリズムを示した.

# Paradise: Distributed Debugging Environment
## — Distributed and Delayed Halting for Distributed Processes —

Sigehisa Kawabe　　Ichiro Yamashita　　Yasumasa Moriya
Fuji Xerox Co., Ltd. System Technology Research Lab.
2274 Hongo Ebina-Shi Kanagawa-Ken 243-04 Japan

Distributed and delayed halting algorithm without broadcast communication is presented for distributed systems, in which interprocess communication is restricted to the Remote Procedure Calls(RPCs). We give the distributed system models, the debbugger models and their formalization and then show the correctness of this algorithm in that it preserves the behavior of the debugged distributed programs against the distributed breakpointing. Further we show the equivalent algorithms for implementation to the distributed debugger with RPC detecting mechanisms.

# 1 まえがき

ネットワークに分散されたノード上で協調して並行・並列動作するプログラムを記述する方法の一つとして, 処理を提供するサーバと, 結果を利用するクライアントとして機能分離をおこない, モジュール化プログラミングを行う方法がある. 特に遠隔ノードでの計算を手続きとして抽象化した遠隔手続き呼出し (Remote Procedure Call, 以後 **RPC**) を用いれば, プロセス間通信による通信手順を意識せずに, 限られた型ではあるが比較的容易に分散計算を扱うことができる [2][9].

しかし, プロセス間通信が **RPC** に限定された分散システムでは, プログラミング時には単一と考えることができる制御が, 実行時には複数のプロセスに至る. そのため, 遠隔でない通常の手続きを用いたプログラムでは発生しなかった分散システム固有の問題により, デバッグが複雑で困難になると考えられる. 例えば, 分散された実行の制御を, 単一のプロセスにより実現されたデバッガで追跡するのは困難であろう [11].

また, プログラムを中断したり, 1 ステップずつ, あるいはゆっくり実行させ, 制御を追跡するデバッグ手法は, 分散プロセスでは一概に有効とはいえない. 例えば, サーバがブレークポイントで中断されたり, プログラムエラーやネットワーク障害などにより, **RPC** の応答の遅れ/消失が, caller プロセスがタイムアウトを引き起こすかもしれない. タイムアウトが発生すると, 制御が枝別れしたり, 実行の振舞いが非決定的となるかもしれない. 従来のタイムアウトを考慮しないデバッガでは, デバッグ時の動作の再現性が失われるだろうし, 分散システムの非決定的な動作により, エラーを発生するイベントの系列の再現が正しく行なわれない場合もあり, よりデバッグを困難にすると考えられる. 実行の再現は, エラーの発生を特定するための全ての情報を蓄積し, それを元に振舞いを再現することで可能となるが, 一般にはこの情報は莫大なものとなる.

Instant Replay [4] では割り込みや時刻を参照するイベントが存在しない場合, モニタリングフェーズで各共有変数のアクセスの履歴に限って保存し, リプレイフェーズでは履歴を元に共有変数のアクセス順序を保存するように実行順序を制御することで, 非決定的な動作を含む分散システムの再実行を実現する. また共有変数のアクセスに換えて, メッセージの交換を記録することで, メッセージパッシングに基づいた分散プロセスの動作の再現を提供することが可能な場合もある.

Mayflower OS では, ノード間の通信は **RPC** が担っており, Pilgrim と呼ばれる分散型のデバッガが開発されている [5]. Pilgrim ではメッセージ交換のログやモニタリング機能を提供するほか, 論理時計 (logical clock) を導入しブレークポイント含む分散プロセス間の時間の整合性を維持することで, タイムアウトも含みデバッグ時の振舞いを正しく扱おうとしている.

Pilgrim の環境でのブレークポイントによる分散プロセスの中断は, できる限り短時間内にデバッグ中の全プロセスの停止をするため, ブロードキャスト通信を用いている. しかし厳密な意味でのブロードキャスト通信が提供されていないため, 次の RPC が発生するまでに最悪で数個のプロセスしか中断できない場合がある.

ここで注意すべき事は, 分散プロセスのブレークポイントによる中断は, Instant Replay と同様に非同期割り込みや時刻を参照するイベントを含まなければ, 共有変数のアクセスやメッセージイベントの (半) 順序関係が保存される限り, 必ずしも全てのプロセスを同時に中断する必要はなく, 中断を 共有変数のアクセスやメッセージイベント発生まで遅らせられる点である [5].

本報告では, **RPC** の追跡を基本とした分散環境向けデバッガ **Paradise**[11](実装中) での分散中断手法として, プロセス間通信を **RPC** に限定した分散プログラムのノン・リプレイデバッグ向け分散遅延中断アルゴリズムを提案する. アルゴリズムはブレークポイントによる分散プログラムの中断を,**RPC** イベントの発生時に分散的に行なうことでブロードキャスト通信を提供していない環境においても, 中断を正しく扱うことができる.

以下, 2節では本報告で用いる記法や概念の定義を行ない, 3節で遅延中断アルゴリズムを示す. さらに 4節ではこのアルゴリズムで分散プロセスを中断しても, プログラムの振舞いが保存されることを証明する. 最後に, 遅延中断アルゴリズムの実装について簡単に述べる.

# 2 準備

この節では, 本報告中で用いる記法, 定義等を与える.

## 2.1 分散システムのモデル

ここでは, **RPC** に基づく分散プログラムにおいて, call のあと **RPC** からの return によって caller に制御が戻ってくるまで, caller が待ち続ける型の遠隔手続きを考える. はじめに, インストラクションの実行やイベ

ントの系列を形式化する.

**定義 2.1** 実行素片 $e$ とは連続した $l$ 個のインストラクションの実行またはイベント (以下合わせてイベントと呼ぶ) の系列 $\{s_1, \cdots, s_l\}$ とし, 一つの実行素片には複数のプロセスのイベントは含まれない. すなわち単一のプロセス中のイベント系列とする. また, 議論を単純にするためイベントにはプロセスの分岐および合流 (*ex. fork, join*) は含まれないと仮定する. 本報告では, 分岐および合流をするプロセスは扱わない.

**定義 2.2** ここで, すべてのイベントの集合を $L$, 分散プログラムでのユニークなプロセス $id$ の集合を $I$ とし, イベント $l \in L$ を与えると $l$ のプロセス $id$ を返す関数 $\overline{pid} : L \to I$ を定める. 一つの実行素片に含まれるイベントは同一プロセスに含まれるとしたので,

$s_i, s_j \in e = \{s_1, \cdots, s_l\}$ について,

$\overline{pid}(s_i) = \overline{pid}(s_j)$

**定義 2.3** 関数 $\overline{pid}$ のドメイン $L$ をすべての実行素片の集合 $S$ に自然に拡張して, $e \in S$ を与えると $e$ のプロセス $id$ を返す関数として $pid : S \to I$ を定める. すなわち,

$e = \{s_1, \cdots, s_l\} \in S$ について,

$pid(e) = \overline{pid}(s_1) = \cdots = \overline{pid}(s_l)$

となるよう関数 $pid$ を定める.

**定義 2.4** 実行素片の実行時間 $T$ を与える関数を $time : S \to T$ とする. すなわち,

$e \in S$ について $time(e) =$ 実行素片 $e$ の実行時間

次に, 逐次実行, あるいは手続きや遠隔手続きの call および return をはさんで, 2 つの実行素片の連続した実行を表す実行素片の集合 $S$ 上の関係 $\cdot \subseteq S \times S$ を定める.

**定義 2.5** 実行素片の集合 $S$ について, 2 つの実行素片 $e_i = \{s_1, \cdots, s_m\}$, $e_j = \{t_1, \cdots, t_n\} \in S$ で, $t_1$ が $s_m$ の直後に実行されるイベントである[1] とき, $e_i$ は $e_j$ に隣接するとし, 次のように記す.

$e_i \cdot e_j$

関係 $\cdot \subseteq S \times S$ は全ての $e_i \cdot e_j (e_i, e_j \in S)$ を含む $S$ 上の最小の関係とする.

ここで単一の制御による逐次的な実行を, 隣接した実行素片の系列として形式化する.

**定義 2.6** 逐次実行 $E$ はプログラムの実行開始時に起動される仮想的な実行素片 $e_0$ ($e_0$ の実行時間すなわち $time(e_0) = 0$ とし $pid(e_0) = pid(e_1)$ とする) に続く $m$ 個の隣接した実行素片の集合とする. すなわち

$E = \{e_i | i = 0, \cdots, m\}$, かつ,

$i = 1, \cdots, m$ について $e_{i-1} \cdot e_i (e_{i-1}, e_i \in E)$

**定義 2.7** 実行素片 $e_i \in E$ の開始時刻 (プログラムの実行開始時刻からの経過時間) $T$ を返す関数 $start : E \to T$ を次のように定める. すなわち

- $start(e_0) = 0$
- $start(e_i) = \sum_{j=0}^{i-1} time(e_j)$ ($i \neq 0$ の場合)

この定義から明らかなように $e_i$ の開始時刻は $time(e_i)$ には依存しない.

**定義 2.8** 実行素片の集合 $S$ について, 関係 $\preceq \subseteq S \times S$ は次の条件を満足する $S$ 上の最小の関係である. すなわち,

1. $e \in S$ について $e \preceq e$
2. $e_i, e_j \in S$ について $e_i \cdot e_j \Rightarrow e_i \preceq e_j$
3. $pid(e_i) = pid(e_j)$ かつ
   $start(e_i) \leq start(e_j) \Rightarrow e_i \preceq e_j$
4. $e_i, e_j, e_k \in S$ について
   $e_i \preceq e_j$ かつ $e_j \preceq e_k \Rightarrow e_i \preceq e_k$

**定理 2.1** 実行素片の集合 $S$ について, $S$ 上の半順序関係 $\preceq$ が与えられて, $e_i, e_j \in S$ について,

$pid(e_i) = pid(e_j)$ かつ

$e_i \preceq e_j \Rightarrow start(e_i) \leq start(e_j)$

**証明:**

背理法で証明する. $pid(e_i) = pid(e_j)$ かつ $e_i \preceq e_j$ のとき, $start(e_j) < start(e_i)$ を仮定する. $\preceq$ の定義より,

$start(e_j) < start(e_i) \Rightarrow e_j \preceq e_i$

かつ, プロセス $ID$ が異なるので, $e_i \neq e_j$ が導かれる. これは $e_i \preceq e_j$ と矛盾を引き起こすため, $start(e_j) < start(e_i)$ ではありえない. (証明終り)

**定理 2.2** $e_i, e_j \in S$ について, $pid(e_i) = pid(e_j)$ のとき

$e_i \preceq e_j \Leftrightarrow start(e_i) \leq start(e_j)$

**証明:**

定義 *2.8* および定理 *2.1* より明らか.

**定義 2.9** $n$ 個の逐次実行 $E^i = \{e^i_j | j = 0, \cdots, m^i\} (i = 1, \cdots, n)$ からなる分散プログラム $P$ の実行は各々の実行 $E^i$ の和集合とする. すなわち

[1] $pid(e_i) \neq pid(e_j)$ でも良い. この場合, $e_i$ と $e_j$ の間に遠隔手続きへの call または return が起こると考えられる.

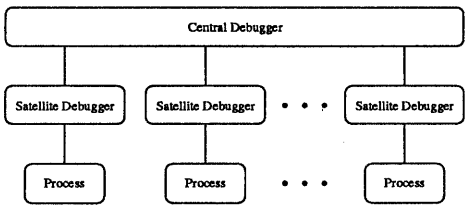


図 1: 分散デバッガ DD の構成

$$P = \bigcup_{i=1}^{n} E^i$$

必要に応じて, $P$から各 $i(=1,\cdots,n)$ について $E^i = \{e^i_j | j = 0,\cdots,m^i\}$ を構成することができる.

## 2.2 デバッギングモデル

この節では本報告で扱うデバッギングモデルを与える. また, ブレークポイントや他の要因に起因する実行の遅延のため, 動作は実行素片 $e$ と同じで, 実行時間が $e$ の実行時間より長いような, 実行素片$\overline{e}$を定める.

**定義 2.10** 分散デバッガ $DD$(図 1) は,
$I = \{pid(e) | e \in P\}$ として

- 各 $p \in I$について衛星デバッガ $SD_p$
- 中央デバッガ $CD$

からなる.

**定義 2.11** 分散プログラムの実行 $P$について, $I = \{pid(e) | e \in P\}$ として各 $p \in I$について,$SD_p$に対するブレークポイントの集合を $B_p$とする.

また, 分散プログラムの実行 $P$のブレークポイントの集合を, $B = \bigcup_{p \in I} B_p$とする. 各ブレークポイント $b \in B$ は 1 つの実行素片に 1 つだけ設定され, かつ, $b$ による実行の遅延は実行素片の本来の実行が終った後に付け加えられるとしても一般性は失われない. 例えば, 1 つの実行素片に複数のブレークポイントを設定した場合は, 各ブレークポイントの終点で実行素片を分割し, 1 つの実行素片に対して 1 つブレークポイントが設定されるように各逐次実行を再構成すれば良い. また, 実行 $P$が与えられ, $E^i = \{e^i_j | j = 0,\cdots,m^i\}$ とすると, $b \in B$が実行素片 $e^i_j \in P$に設定されたブレークポイントであることを, 明示的に示す場合, 特に $b^i_j$のように記す.

衛星デバッガ $SD_p$は $B_p$から定まる実行素片 $e^i_j \in P$ の定められたイベントで実行を中断させる.

**定義 2.12** ブレークポイント $b \in B$による実行の遅延が確定したとき, その遅延時間を, $\overline{susp}(b)$ で与える.

**定義 2.13** 実行素片 $e \in S$と遅延時間$\Delta t$ から定まる遅延された実行素片 $e[\Delta t]$ は実行素片の動作開始から $time(e)$ の間に $e$ と同じイベントの系列を有し, その後 $\Delta t$ だけ時間を消費する. したがって,

$$time(e[\Delta t]) = time(e) + \Delta t$$

である. 必要に応じて $e[\Delta t]$ を$\overline{e}$と記す. このようにして $S$から作られた, 遅延された実行素片の集合を$\overline{S}$と記す. またプロセス $ID$ は保存される. すなわち,

$$e \in S, e[\Delta t] \in \overline{S}\text{について}, pid(e) = pid(e[\Delta t])$$

**定義 2.14** 遅延された実行素片 $e[\Delta t] \in \overline{S}$において, 関数 $susp : \overline{S} \to T$を次のように定める.

$$susp(e[\Delta t]) = \Delta t$$

# 3 アルゴリズム $D$

アルゴリズム $D$は $n$ 個の逐次実行からなる分散プロセスの実行 $P$及び, $p$ 個のブレークポイントからなる集合 $B$の 2 項組 $\langle P, B\rangle$ を入力として, $n$ 個の逐次実行からなる遅延された分散プログラムの実行$\overline{P}$を出力する.

$$\langle P, B\rangle \xrightarrow{D} \overline{P}$$

アルゴリズムは $P$を順に調べ, ブレークポイントによる遅延や伝搬した遅延時間の合計を計算し, 新たな遅延が発生するたびに, 遅延の影響をうける **RPC** の call と return を捜しだし, 遅延時間を伝搬させ, 最終的に $P$と同じ振舞いの$\overline{P}$を計算する.

**定義 3.1** 各逐次実行$\overline{E^i}(i = 0,\cdots,n)$ に対する添字の $n$ 項組 $\mathcal{F} = \langle f^1,\cdots,f^n\rangle$ を分散プログラムの実行前線とする. 特に $f^i$は逐次実行$\overline{E^i}$の実行前線と呼ぶ. 添字 $f^i$の範囲は 0 から逐次実行$\overline{E^i}$の最後の実行素片の添字までで, また最後の実行素片が処理されたときに, 完了を示すマークが代入されるとする.

アルゴリズム $D$の動作の概要を示す.

1. 次に実行される実行素片のうちで, 最も早く終了するものを見つける.
2. 見つけた実行素片までの遅延時間の合計を計算する.
3. (a) その遅延の影響を受ける実行素片を捜し,
   (b) 遅延時間を伝搬させ,
   (c) 以降の実行素片の遅延量を補正する.
4. 実行前線を進める.

### 3.1 アルゴリズム:準備と初期化

**定義 3.2** 逐次実行 $E$が実行される可能性のあるプロセス $ID$ の集合を $servers(E)$ とする. すなわち,

$servers(E) =$

$\{pid(e)|$ すべての $e \in E\}(E$は逐次実行)

**定義 3.3** $\langle P, B\rangle$ が与えられたとき, $P$の各逐次実行 $E^i(i = 1,\cdots,n)$ の $j$番目の実行素片 $e^i_j \in P(j = 0,\cdots,m^i)$ について, 次の条件を満たす最小の集合$\overline{P}$を構成する. すなわち, すべての$e^i_j \in P$について

$e^i_j[\overline{susp}(b^i_j)] \in \overline{P}$ ($e^i_j$に対して$b^i_j \in B$の場合)[2]

$e^i_j[0] \in \overline{P}$ (それ以外)

**定理 3.1**

$e_i, e_j \in P, \overline{e_i}, \overline{e_j} \in \overline{P}$について $e_i \cdot e_j \Rightarrow \overline{e_i} \cdot \overline{e_j}$

**証明:** $\overline{P}$の構成の方法は, 実行素片の実行時間を増加させるに留まるため, $P$に含まれる各逐次実行の実行素片の実行順序は, $P$から構成された遅延された実行素片からなる$\overline{P}$の対応する各逐次実行において保存される.

**定義 3.4** 操作 $\mathcal{D}$は実行素片 $e[\Delta T] \in \overline{P}$ および遅延時間$\Delta x$ の 2 項組 $\langle e[\Delta T], \Delta x\rangle$ を入力として, 次の動作をする. すなわち,

$\overline{P}$から実行素片 $e[\Delta T]$ を取り除き, 代わりに $e[\Delta T + \Delta x]$ を$\overline{P}$の要素とする.

### 3.2 アルゴリズム:本体

$\overline{P}$の各逐次実行$\overline{E^i} = \{\overline{e^i_j}|(j = 0,\cdots,m^i)\}(i = 1,\cdots,n)$ について, 実行前線 $\mathcal{F} = \langle f^1,\cdots,f^n\rangle$ を $\langle 1,\cdots,1\rangle$ から始めて $\langle m^1,\cdots,m^n\rangle$ の処理が完了するまで次の操作 (ステップ 1〜ステップ 4) を繰り返す.

1. 各 $i(i = 0,\cdots,n)$ について
   $front^i_{f^i} = start(\overline{e^i_{f^i}}) + time(\overline{e^i_{f^i}}) - susp(\overline{e^i_{f^i}})$
   を計算し, 最小の $front^i_{f^i}$となる $i$ を決定する.
2. $\Delta T^i_{f^i} = \sum_{u=0}^{f^i} susp(\overline{e^i_u})$ を計算する.
3. $\overline{P}$の各逐次実行$\overline{E^k} = \{\overline{e^k_l}|(l = 1,\cdots,l^k)\}$ に対し, $k = 1,\cdots,n$(ただし $k \neq i$) について次のステップ (ステップ 3a〜ステップ 3c) を繰り返す.
   (a) $l = f^k$から順に $l^k$まで
   $front^i_{f^i} < start(\overline{e^k_l})$ かつ,
   $pid(\overline{e^k_l}) \in servers(\overline{E^i})$ かつ,
   $pid(\overline{e^k_{l-1}}) \neq pid(\overline{e^k_l})$
   を満たす $l$を順に捜す (ただし $f^k$, $l^k$はそれぞれ逐次実行 $E^k$の実行前線および, 最後の実行素片). 見つかれば,
   $$\Delta x_{l-1} = max(0, \Delta T^i_{f^i} - \sum_{u=0}^{l-1} susp(\overline{e^k_u}))$$
   とし3bに進む.$\Delta x_{l-1} = 0$ の場合あるいは, 見つからなければ, k を次の値にしてステップ 3 を繰り返す.
   (b) $\mathcal{D}(\overline{e^k_{l-1}}, \Delta x_{l-1})$ を実行. このとき, $l = f^k$ ならば $f^k$一つ前の添字にする.
   (c) $u = l,\cdots,l^k$について, 次のステップ (ステップ 3(c)i〜ステップ 3(c)ii) を繰り返す.
      i. $\overline{e^k_u}$に対して $b^k_u \in B$ がある場合,
      $\Delta t_u = min(\Delta x_{u-1}, susp(\overline{e^k_u}) - \overline{susp}(b^k_u))$
      そうでないとき
      $\Delta t_u = min(\Delta x_{u-1}, susp(\overline{e^k_u}))$
      とする.
      ii. $\mathcal{D}(\overline{e^k_u}, -\Delta t_u)$ を実行
      $\Delta x_u \leftarrow \Delta x_{u-1} - \Delta t_u$
      $\Delta x_u = 0$ なら k を次の値にしてステップ 3を繰り返す.
4. $f^i$を次の添え字にする. すでに $f^1,\cdots f^n$すべてが完了ならアルゴリズムを終了する.

### 3.3 アルゴリズム:実行例

図 2に示した $P$と, BP で示した 3 箇所のブレークポイントに対して, アルゴリズム $D$が働く様子を示す. 点線で結んだ○印は実行前線, 横線が RPC の call または return, 縦線は実行素片で, そのうち, 点線と破線はぞれぞれブレークポイントおよび, 伝搬による遅延を示す. 実線と点線で囲まれたプロセス名はそれぞれ, サーバおよび伝搬対象となるクライアントを示す. 図3に途中の計算状態を示し, 図4にアルゴリズム $D$により計算された, 遅延された分散プログラムの実行$\overline{P}$を示す.

また, このアルゴリズムは RPC 検出時に衛星デバッガが分散的にプロセスを中断させるため, ブロードキャスト通信を必要としない.

## 4 アルゴリズム $D$の正当性

**定理 4.1** アルゴリズム $D$は分散プログラムの実行素片の半順序関係を保存する. すなわち,

[2] $P$の初期状態としてブレークポイントが設定されている実行素片が遅延されている.

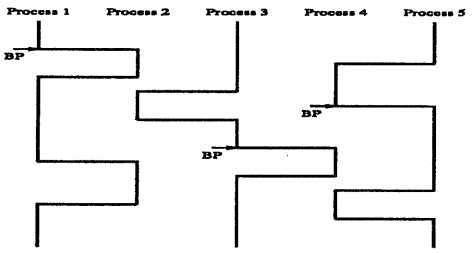


図 2: 分散プログラムの実行 $P$

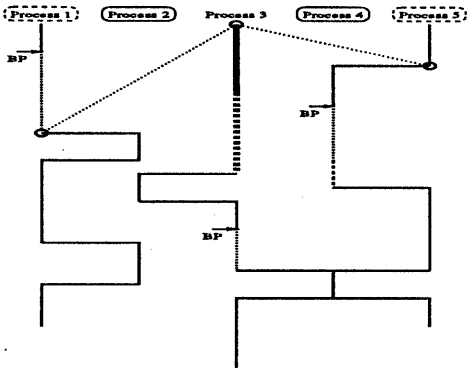


図 3: アルゴリズム $D$ の計算経過

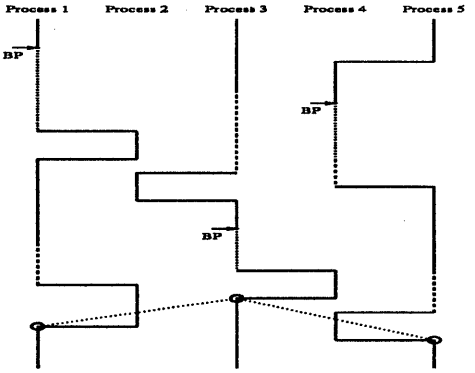


図 4: アルゴリズム $D$ の計算結果$\overline{P}$

$$\langle P, B\rangle \xrightarrow{H} \overline{P}$$

のとき, $P$から定まる $P$上の半順序関係$\preceq_P$および, $\overline{P}$から定まる $P$上の半順序関係$\preceq_{\overline{P}}$について

$$a \preceq b \subseteq \preceq_P \Rightarrow \overline{a} \preceq \overline{b} \subseteq \preceq_{\overline{P}} \ (a, b \in P,\ \overline{a}, \overline{b} \in \overline{P})$$

**証明:** 関係 $\preceq$の定義2.8から, 次の 2 つの場合を証明すれば十分である.

1. $a \cdot b$ かつ $a \preceq b \Rightarrow \overline{a} \cdot \overline{b}$

   定理3.1より $a \cdot b$ かつ $a \preceq b \Rightarrow \overline{a} \cdot \overline{b} \Rightarrow \overline{a} \preceq \overline{b}$ となり明らか.

2. $pid(a) = pid(b)$ かつ $a \preceq b \Rightarrow \overline{a} \preceq \overline{b}$

まず, $pid(a) = pid(b)$ かつ
$start(a) \leq start(b) \Rightarrow start(\overline{a}) \leq start(\overline{b})$
を帰納法で証明する.
$E^i$, $E^k$について, $a = e^i_j \in E^i$, $b = e^k_l \in E^k$とする.
仮定より $pid(e^i_j) = pid(e^k_l)$ である.
また, 定理2.1より,
$e^i_j \preceq e^k_l \Rightarrow start(e^i_j) \leq start(e^k_l)$ である.
ここでは, $b$ の直前で **RPC** が発生した場合のみを考えるが, その他の場合の証明も同様であるため省略する.

**base step:**
定義2.7より,
$time(\overline{e^i_1}) = time(e^i_1) \leq start(e^i_j) \leq start(e^k_l)$
がいえる. ステップ 1で$\overline{e_1}$が選ばれた時,
$front^i_1 = start(\overline{e^i_1}) + time(\overline{e^i_1}) - susp(\overline{e^i_1}) = time(e^i_1)$
とし, ステップ 2で
$\Delta T^i_1 = \sum_{u=0}^{1} susp(\overline{e^i_u}) = susp(\overline{e^i_1})$ が得られる. ここで,
ステップ 3aにおいて, 少なくとも, $\overline{e^k_l}(=\overline{b})$ については,
$front^i_1 < start(\overline{e^k_l})$ かつ, $pid(\overline{e^k_l}) \in servers(\overline{E^i})$
を満足する. 同様の条件を満足する$\overline{e^k_n}(n < l)$ が存在する場合でも以下の証明には影響を与えない.
ステップ 3によって,
$\Delta T^i_1 = \sum_{u=0}^{1} susp(\overline{e^i_u}) \leq \sum_{u=0}^{l-1} susp(\overline{e^k_u})$ が導かれる.

**induction step:**
$1 \leq m \leq j-2$ に対して,
$\Delta T^i_m = \sum_{u=0}^{m} susp(\overline{e^i_u}) \leq \sum_{u=0}^{l-1} susp(\overline{e^k_u})$
が成り立つと仮定して, ステップ 1で$\overline{e_{m+1}}$が選ばれた時,

$$\begin{aligned} & front^i_{m+1} \\ &= start(\overline{e^i_{m+1}}) + time(\overline{e^i_{m+1}}) - susp(\overline{e^i_{m+1}}) \\ &= start(\overline{e^i_{m+1}}) + time(e^i_{m+1}) \end{aligned}$$

$$= \Delta T^i_m + \sum_{u=0}^{m} time(e^i_u) + time(e^i_{m+1})$$
$$\leq \Delta T^i_m + start(e^i_m)$$

となって, 仮定より,

$\Delta T^i_m + start(e^i_m) \leq \sum_{u=0}^{l-1} susp(\overline{e^k_u}) + start(e^k_u) \leq start(\overline{e^k_l})$ である.

従って, 3aにおいて, 少なくとも, $\overline{e^k_l}(=\overline{b})$ については, $front^i_{m+1} < start(\overline{e^k_l})$ かつ, $pid(\overline{e^k_l}) \in servers(\overline{E^i})$ を満足する. 同様の条件を満足する$\overline{e^k_n}(n < l)$ が存在する場合でも以下の証明には影響を与えない.

ステップ3によって,

$\Delta T^i_{m+1} = \sum_{u=0}^{m+1} susp(\overline{e^i_u}) \leq \sum_{u=0}^{l-1} susp(\overline{e^k_u})$

以上より, $m = j-2$ として,

$\sum_{u=0}^{k-1} susp(\overline{e^i_u}) \leq \sum_{u=0}^{l-1} susp(\overline{e^k_u})$

が導かれ, $start(e^i_j) \leq start(e^k_l)$

と合わせて, $start(\overline{e^i_j}) \leq start(\overline{e^k_l})$

が得られる.

結論として,

$pid(a) = pid(b)$ かつ $a \preceq b \Rightarrow$

$start(a) \leq start(b) \Rightarrow start(\overline{a}) \leq start(\overline{b}) \Rightarrow \overline{a} \preceq \overline{b}$

が成り立つ. (証明終り)

## 5 実装

この節では, 第3節で提案した遅延停止アルゴリズムの分散デバッガへの適用について述べる.

第3節で述べたアルゴリズム $D$ の実行順序は, デバッグ時の時間の経過とは無関係なものとなっている. そのため, このままでは逐次的かつ分散して動作分散環境向けデバッガに実装することは困難であろう.

ここで, アルゴリズム $D$ を注意深く調べると, 次のようなことが判明する.

- ブレークポイントによる各実行素片の遅延や, ステップ3で伝搬された遅延は, ステップ2で初めて評価される.
- 各逐次実行の実行素片の遅延が実行順に伝搬される場合, ステップ3cの実行素片に対する遅延の補正は必要ない.

従って, 初期化におけるブレークポイントによる遅延の処理とステップ3での伝搬した遅延の処理をステップ1で行なうようアルゴリズムを変更する. これで, 逐次的かつ分散して動作するアルゴリズムとして分散環境向けデバッガに実装することが可能となる. また, ステップ3での遅延の伝搬をまとめて計算し, クライアントとなる可能性のあるプロセスの最大の遅延時間とすれば良い.

以上の方針で実装したサテライトデバッガとセントラルデバッガのアルゴリズムの詳細を付録Aに示す.

## 6 むすび

ブロードキャストを用いることなく, 1対1の通信のみで分散プログラムの動作を保存する分散環境向けデバッガのアルゴリズムの提案と検証をおこなった.

今後の課題として, タイムアウトを正しく扱う実装と, 実機上でのパフォーマンス等の評価をおこなう必要がある.

## 参考文献


[1] L. Lamport "Time, Clocks, and the Ordering of Events in a Distributed System.", Communications of the ACM, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[2] A. D. Birrell and B. J. Nelson "Implementing Remote Procedure Calls.", ACM Trans. on Computer Systems, vol.2, no. 1, pp. 39–59, Feb. 1984.

[3] K. M. Chandy and L. Lamport "Distributed Snapshots: Determining Global States of Distributed Systems.", ACM Trans. on Computer Systems, vol.3, no. 1, pp. 63–75, Feb. 1985.

[4] T. J. LeBlanc and J. M. Mellor-Crummey "Debugging Parallel Programs with Instant Replay.", IEEE Trans. on Compt., vol. C–36, no. 4, pp. 417–480, Apr. 1987.

[5] R. Cooper "Pilgrim: A Debugger for Distributed Systems.", Proc. of 7th Int. Conf. on Distributed Computing Systems, pp. 458–465, Sep. 1987.

[6] N. Takahashi "Partial Replay of Parallel Programs Based on Shared Objects.", Preprints Work. Gr. for Programming Language 23–10 IPS Japan, Dec. 1989. (In Japanese)

[7] Y. Manabe and M. Imase "Global Condition in Debugging Distributed Programs.", Preprints Work. Gr. for Programming Language 23–11 IPS Japan, Dec. 1989.

[8] W. Hseush and G. E. Kaiser "Modeling Concurrency in Parallel Debugging.", Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 11–20, Mar. 1990.

[9] "The Frontier of the Distributed Computing.", Nikkei Electronics, no. 502, pp. 122–148, 11. Jun. 1990.(In Japaneses)

[10] D. Caswell and D. Black "Implementing a Mach Debugger For Multithreaded Applications.", Proc. of USENIX, Winter '90, pp. 25–39, Nov. 1990.

[11] I. Yamashita and S. Kawabe "RPC-traceable Remote Debugger for Distributed Environment.", Papers '91 Spring Annual Meet. on Computer Systems of the Inst. Electronics, Inf. & Comum. Eng. Japan, pp.6-175(one page), Mar. 1991. (In Japaneses)

# A アルゴリズム

## A.1 共通の関数

```
type
  debugger_id = {CD, SD1, ···, SDn};
  debuggee_id = {P1, ···, Pn};
  command = {SUSPEND, CONTINUE, RPC};
  message = record src: debugger_id; dst: debugger_id;
                   cmd: command end;
  event_type = {BREAKPOINT, CONTIMUE_CMD,
                RPC, RECEIVE_MESSAGE, ALARM};
procedure waite(var event: event_type)
  begin /* イベントが発生するまで待つ */ end
procedure SendMessage
      (src: debugger_id, dst: debugger_id, cmd: command);
  begin /* デバッガにメッセージを送る */ end
function ReadMessage: massage;
  begin /* メッセージを読み出す */ end
```

## A.2 $SD_i$のアルゴリズム

```
program SatelliteDebugger
procedure Suspend;
  begin /* デバッギの実行を中断する */ end
procedure Continue;
  begin /* デバッギの実行を継続する */ end

procedure MAIN
var
  event: event_type;
  msg: message;
begin
  repeat
    wait(event);
    case event of
      BREAKPOINT:
        begin
          Suspend;
          SendMessage(SDi, CD, SUSPEND)
        end
      CONTIMUE_CMD:
        begin
          Continue;
          SendMessage(SDi, CD, CONTINUE)
        end
      RPC:
        begin
          Suspend;
          SendMessage(SDi, CD, RPC)
        end
      RECEIVE_CMD_MESSAGE:
        begin
          msg := ReadMessage;
          if msg.cmd = CONTINUE;
             then Continue;
        end
    end
  end
end
```

## A.3 CD のアルゴリズム

```
program CentralDebugger
procedure StartClock(did: debugger_id);
  begin /* デバッギの時計を動かす */ end
procedure StopClock(did: debugger_id);
  begin /* デバッギの時計を止める */ end
function Time(did: debugger_id): time;
  begin /* デバッギの時刻を返す */ end
function SetAlarm(did: debugger_id, p: pid): time;
  begin /* p の時計のアラームにデバッギの時刻をセット */ end
function ServerID(did: debugger_id): pid;
  begin /* デバッギのサーバの ID を返す */ end
function LatestClientTime(pid: debuggee_id): time;
  begin /* クライアントの中で最も遅れている時刻を返す */ end

procedure MAIN
var
  event: event_type;
  msg: message;
  did: debugger_id;
begin
  wait(event);
  case event of
    RECEIVE_MESSAGE:
      begin
        msg := ReadMessage;
        case msg.cmd of
          SUSPEND:
            begin StopClock(msg.src); end
          CONTNUE:
            begin StartClock(msg.src); end
          RPC:
            begin
              p := ServerID(msg.src);
              if Time(msg.src) > LatestClientTime(p)
                begin
                  StopClock(msg.src);
                  SetAlarm(msg.src, LatestClient(p));
                end
              else
                begin
                  SendMessage(CD, msg.src, CONTINUE);
                  SetClock(msg.src);
                end
            end
        end
      end
    ALARM:
      begin
        did := GetAlarm;
        SendMessage(CD, did, CONTINUE);
        StartClock(did);
      end
  end
end
```