マルチメディア通信と分散処理　52－34
（1991.　9．27）

分散プログラム用デバッガ ddbx-p の試作開発

真鍋 義文　　　青柳 滋己
NTT ソフトウェア研究所

ABSTRACT

分散プログラムのデバッグの困難さの主な原因は (1) 動作の非決定性、(2) プロセスの相互関係理解の困難さにあると考えられる。これらの問題に対処するため、分散プログラムの開発支援用に、(1) 非決定的通信の再演機能、(2) 大域的条件によるブレークポイント・トレース設定機能、(3) 与えられた大域的条件に対する最小限動作機能を持つ、分散プログラム用デバッガのプロトタイプ版 ddbx-p を開発した。通信による動作の非決定性がある場合、再演機能により、同じ状況を繰り返して試験可能である。また、大域的条件によるブレークポイント・トレース設定および条件成立直後での停止が可能なので、プロセスの相互関係に関するバグの発見が容易である。

Distributed program debugger prototype ddbx-p

Yoshifumi Manabe　　　Shigemi Aoyagi
NTT Software Laboratories
3-9-11 Midori-cho, Musashino-shi, Tokyo 180 Japan

ABSTRACT

Debugging distributed programs is more difficult than debugging sequential programs, since (1) distributed program behavior might be nondeterministic and (2) behavior relation among processes is difficult to understand. We develop a prototype debugger ddbx-p for distributed programs. Ddbx-p has (1) replay mechanism for asynchronous communication, (2) breakpoint and selective trace commands using global predicate conditions, and (3) halting mechanism at the first state satisfying the condition. By the first facility, users can test the same execution repeatedly. By the second and third facilities, users can detect errors concerned with the process behavior relation.

1 まえがき

近年、ネットワーク技術の発展に伴い、分散システムが増加しつつある。それにより、分散プログラムの開発機会も増加してきている。しかし一般に、分散プログラムは逐次型プログラムに比べると開発が困難である。それは、分散プログラムは複数のプロセスが協調し合って動作する部分の設計・デバッグが困難なためである。その原因は主に、

- 動作の非決定性
- プロセスの相互関係理解の困難さ

にあると考えられる。これらの問題に対処するため、分散プログラムの開発支援用に、

- 再演機能
- 大域的条件式による停止、トレース機能
- 最小限実行機能

を持つ、分散プログラム用デバッガのプロトタイプ版 ddbx-p(distributed dbx prototype) を開発した。本稿では、ddbx-p の概要およびその評価について述べる。

2 分散プログラム用デバッガに求められる機能

本節では、分散プログラムのデバッグを困難にしている主な要因、

- 動作の非決定性
- プロセスの相互関係理解の困難さ

およびそれに対処するためにデバッガに望まれる機能について述べる。

2.1 動作の非決定性

分散プログラムにおいては、各プロセスのプログラムが決定的であったとしても、プロセス間の通信の非同期性のために、全体的な動作が非決定的になる場合がある。例えば、あるプログラムにおいて、プロセス 0、1 の変数 x の値が 1 である場合にあるリソースを使用する権利が与えられているものとする。二つのプロセスが同時に同じリソースにアクセスすることは従って誤りになる。このプログラムのある実行が図 1(a) のようであったとすると、プロセス 0 と 1 で $x = 1$ が同時に起きている (プロセス 0 と 1 の動作速度によっては同時に起こり得る) ので、このプログラムにはバグがあり、その影響が後で顕在化する。しかし、この顕在化したバグの原因を探るために再度このプログラムを実行した時には、通信の非同期性のため、その実行が図 1(b) のようになり、プロセス 0 と 1 で $x = 1$ が同時に起きず、バグが顕在化しない可能性がある。

逐次型プログラムのデバッグにおいて、ブレークポイントを設定し、ある誤りの状態を調べ、さらにその原因を調べるために別のブレークポイントを設定して再実行を繰り返す手法はサイクリックデバッグと呼び、デバッグの一般的な手法である [3]。分散プログラムに対してサイクリックデバッグを

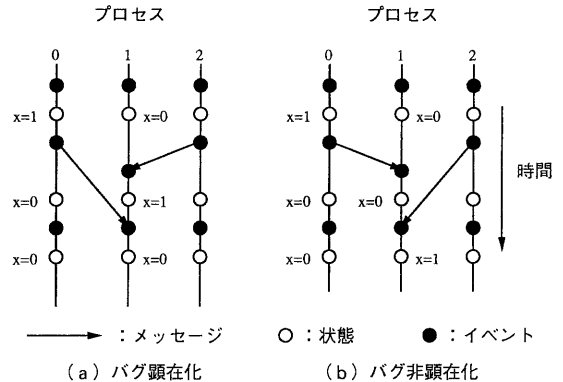

図 1 分散プログラムの動作例

行なうためには、バグの顕在化した実行状況を何度でもテストできることが望ましい。従って、動作の再演性が必要である。

2.2 プロセスの相互関係理解の困難さ

分散プログラムのデバッグにおいて、各プロセスの動作を個別に調べるのであれば、逐次型プログラムに対する従来のデバッガを用いればよい。しかし、上記の例の、2 つのプロセスが同時に同じリソースにアクセスする権利を得る、のように、複数のプロセスの相互関係において誤りが生じる可能性もある。このような複数プロセスに関する誤りは分散システムに特有である。逐次型プログラムに対するデバッガでは複数のプロセスの状態を一箇所で監視することができないので、このような誤りの検出は困難である。従って、分散プログラム用のデバッガではこのような複数のプロセスに関する条件の成立を検出する機能があることが好ましい。上記の例では、『プロセス 0 で変数 $x = 1$ になり、かつ、プロセス 1 で変数 $x = 1$ となる』という事象が成立した場合に全プロセスを停止させる、あるいはその時の状態を表示する、などの機能があることが望ましい。

また、条件式が一部のプロセスのみに関するものであった場合、条件式の存在しないプロセスについてはどの状態で停止させるべきであるかも考慮する必要がある。上記の例『プロセス 0 で変数 $x = 1$ になり、かつ、プロセス 1 で変数 $x = 1$ となった時に停止する』の場合、プロセス 2 については条件が存在しない。条件式がプロセス 0 と 1 のみに関するものであったとしても、その成立にプロセス 2 が寄与している可能性もある。この例では、プロセス 2 からのメッセージによって、プロセス 1 の変数 $x = 1$ となった可能性もある。この場合、プロセス 2 がプロセス 1 にメッセージを送った直後の状態でプロセス 2 を停止させるのが最適である。プロセス 2 をその状態から動作させればさせるほど、変数の内容の変更などによって、なぜおよびどこで、プロセス 2 がプロセス 1 にそのようなメッセージを送信したのかがわかりにくくなる。従って、与えられた条件式に対して、その条件式を満足させる最小の実行を行なった時点で全プロセスを停止させるのが望ましいと考えられる。トレースの場合も同様である。この

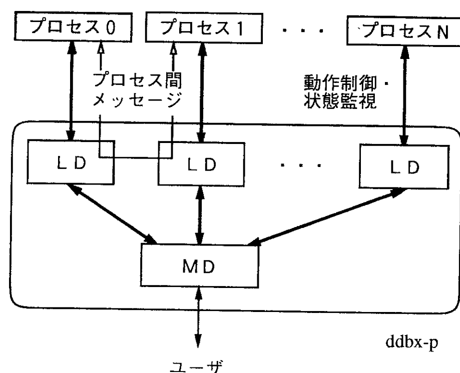

図2 ddbx－pの構成

ような、最小限実行機能を持つことが望ましい。

従って、分散プログラム用デバッガは、

- 再演機能
- 大域的条件式による停止、トレース機能
- 最小限実行機能

を持つことが重要であると考えられる。しかし、既存の分散プログラム用デバッガでこれらの機能をすべて持ち合わせているものは存在しない [9]。再演実行を行なわないデバッガにおいて、複数のプロセスに関する条件式として、'Linked Predicate' と呼ぶ特別な形式の条件式のみを許すもの [10]、複数のプロセスに関する大域的条件式を記述可能なもの [4] が存在するが、最小限実行機能は持たない。また、これらは実行中に条件式の成立を判定するため、条件式判定処理の手間が大きい場合は、デバッガ動作時とデバッガを除いた通常動作時とで動作が異なる場合がある。

再演実行を行なうデバッガについては、単一プロセスに関する条件式のみによる停止機能を持つもの [7] が考えられている。また、このデバッガでは、ある条件式が成立した場合は、該当プロセスのみを停止させ、他のプロセスは停止したプロセスからの通信を待つ状態になるまで実行を継続する。すなわち、最小限実行機能はない。

3 ddbx-p の概要

ddbx-p はメッセージパッシングでプロセス間通信を行なう分散プログラムを対象とした再演をベースにしたデバッガである。対象言語は現バージョンでは C のみである。OS は UNIX 4.2BSD 以上であることが必要である。さらに、現バージョンでは、プロセスの動的な生成 / 消滅は行なわないという制約が存在する。各プロセスは異なる計算機上に存在してもよい。

ddbx-p の構成を図 2 に示す。ddbx-p はユーザとの入出力を行なう Main Debugger(MD) と各プロセスの動作を制御し、状態を監視する Local Debugger(LD) とから成る。LD は被デバッグプログラムのプロセス数だけ存在する。また、再演を行なうため、プロセス間通信は LD を通して行なわれる。

4 ddbx-p の使用法

再演のため、システムが提供する通信プリミティブのみを使用してプログラムを作成する必要がある。ddbx-p では、通信相手の指定方法として、プロセス指定とポート指定の 2 種類を用意している。プロセス指定は通信相手のプロセスに与えられた番号を陽に指定するものである。また、ポート指定は分散アルゴリズムの理論モデル (例えば [5])、すなわち、各プロセスからは通信ポートのみが見え、全プロセス間のトポロジーはプロセスに情報として与えられない限り不明であるというモデルに従った通信方法である。

- `int msinit(&argc, argv)` ：初期化処理
- `int mssnd(process/port, message)` ：指定したプロセス / ポートへの送信
- `int msrcv(mode, process/port, message)` ：指定したプロセス (ポート) から受信
- `int msrcva(mode, message)` ：任意のプロセス (ポート) から受信
- `int msrcvs(mode, process/port_set, message)`：指定したプロセス (ポート) 集合の中から受信
- `int msterm()` ：終了時処理

ここで、mode は`WAITMODE` あるいは`NOWAITMODE` という値をとり、メッセージが受信プロセスにまだ届いていなかった場合、`WAITMODE` の時にはメッセージが届くまでブロックされ、`NOWAITMODE` の場合はすぐに error を返す。

msinit, msterm はそれぞれ、通信の初期化処理と終了時処理を行なうルーチンであり、それぞれプログラムの最初と最後におく必要がある。また、プログラム中では、整数型変数 prno(自分のプロセス番号)、tnpr(全プロセス数: プロセス指定の場合)、degree(接合しているポートの数: ポート指定の場合) が使用可能である。

コンパイル時のオプションを変更することにより、通常実行、履歴保存実行、再演実行を行なう各実行プログラムを得ることができる。また、実行プログラムを起動するには init と呼ぶプログラムを使用することが必要である。init は、複数計算機上で動作させる場合と単一計算機上で動作させる場合で異なる。ここでは、単一計算機上で動作させる場合のみを記す。この時は、

```
init [programfile] [networkfile]
```

を実行する。第一引数は実行プログラム名を記述したファイルの名前である。そのファイルの行数は起動するプロセスの数、各行の内容は各プロセスで実行させるプログラムを表す。[programfile] のファイルの内容が以下のようであったとすると、

```
a.out0 arg01 arg02
a.out1 arg11 arg12
a.out2 arg21 arg22
a.out3 arg31 arg32
```

プロセスを4つ生成し、それらに0、1、2、3というプロセス番号を与え、プロセス0は`a.out0 arg01 arg02` (arg01、arg02はa.out0の引数である)を実行する。プロセス1、2、3はそれぞれ2、3、4行目に書かれたプログラムを実行する。ここで、履歴保存実行、再演実行を行なうプログラム名を記述しておくと、それぞれ履歴保存実行、再演実行を行なうことができる。このときのプロセス番号はプロセス間通信の際に用いる番号であり、実際のUNIXのpidとは異なる。

第二引数はポート指定の通信を行なう場合にのみ用いられる、隣接行列を表したファイルの名前である。[networkfile]のファイルの内容が以下のようであったとする。

```
0 1 0 1
1 0 1 0
0 1 0 0
1 0 0 0
```

1行目はプロセス0がどのプロセスとの間に通信路が存在するかを表す。値が1の場合に通信路は存在する。この例では、プロセス1と3との間に通信路が存在する。従って、プロセス0はdegreeが2であり、2つのポートを持つ(但し、各ポートがどのプロセスと接続されているかは各プロセスには情報として与えられない)。2、3、4行目は同様に、プロセス1、2、3がどのプロセスと接続しているかを表す。例えばプロセス2はdegreeが1であり、プロセス1とのみ接続されている。

5 再演機能の実現法

一般に、非決定的な動作を行なうプログラムの再演を行なうには以下の方法がある[2][11]。非決定的な結果を得る可能性があるシステムコール(など)のすべてに対して、その結果を保存しておく。再演時には、それらのシステムコールに対して、保存しておいた履歴の内容の結果が得られたものとして処理するという方法である。しかし、この方法は履歴の量が膨大であり[11]、また、手間が大きくなるためにprobe effect(状態を監視するための動作により、プログラムの動作自身が変化すること)が大きくなるという問題がある。

それに対して、非決定的な結果を得る部分が少ない場合には、保存する履歴の量を減少させる手法が提案されている。LeBlancらは、共有メモリへのアクセスの遅れにのみ非決定性が存在する分散プログラムの再演法として、Instant replayという手法を提案している[7]。この方法はメッセージ通信のみを行なう分散プログラムにおいても同様に適用することができる[8]。ddbx-pでは、[8]で示した以下の再演法をインプリメントしている。

履歴保存実行を行なった場合、実行中にhistoryX(Xはプロセス番号)という名前の履歴保存ファイルが作成される。その内容は、msrcv/msrcva/msrcvsを実行した際に、受信したメッセージの送信プロセス(ポート)番号の系列である。但し、そのmsrcv/msrcva/msrcvsでどのプロセス(ポート)からも受信しなければ特別な値'-1'を取るものとする。実行例とその際のhistoryの例を図3に示す。

再演実行の際には、msrcv/msrcva/msrcvsを実行する際に、historyの内容を見て、そのプロセス(ポート)からのメッセー

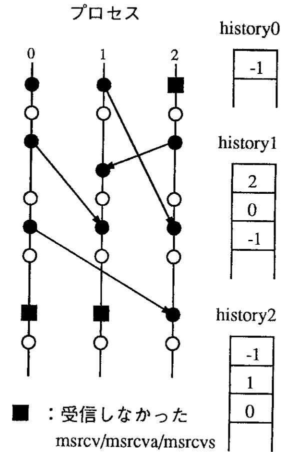

図3 実行例とその際のｈｉｓｔｏｒｙ

ジを受信する('-1'の場合には、メッセージ受信に失敗したかのようにふるまう)。送り手プロセスからメッセージプロセスの動作の非決定性がプロセス間通信の遅れによるものだけである場合には、この方法で再演が可能である。

履歴保存実行を行なった後、再演デバッガを起動する。その場合のコマンドは以下の通りである。

```
ddbx [programfile] [networkfile]
```

引数の意味はinitの場合と同じである。プロセス起動終了後、プロンプトを出力してデバッガのコマンド待ちの状態になる。

6 ddbx-pのコマンド

ddbx-pの主要なコマンドを以下に示す。

- `run` ：再演の開始
- `cont` ：再演の続行
- `rerun` : 再演をもう一度最初から開始
- `stop if [global condition]` : ブレークポイント設定

```
[global condition]::= [conjunctive condition] |
    [conjunctive condition] or [global condition]
[conjunctive condition] ::= [simple condition] |
    [simple condition] and [conjunctive condition]
[simple condition] ::=
    [processno] : [local condition]
[local condition]  ::= [boolean expression] |
    at [lineno]
```

ここで、`[processno]`はプロセス番号、`[boolean expression]`はプログラムの変数などに関する論理式`[lineno]`はプログラムの行番号である。

(例) `stop if 2:i>j and 5:x=0 or 2: at 110`

- `trace [process expression] if [global condition]` : トレースの設定

```
[process expression] ::= [processno] :
    [local expression] | [processno] :
    [local expression] , [process expression]
[local expression] ::= [expression] | source
```

source は、その時点でのソースプログラムを意味する。

(例) `trace 4: source, 2:k if 4:x=0 and 2:y=0`

- `status` :設定された stop, trace コマンド表示
- `delete [commandno]` : stop, trace コマンド取り消し
- `switch [commandno], [predicateno]` : primary の変更
- `print [process expression]` : 式の値の出力
- `list [process lineno]` : ソースプログラム表示

```
[process lineno] ::= [processno] :
    [lineno] , [lineno]
```

- `quit` :デバッガの終了

ddbx-p に特有なコマンドは、大域的条件を用いる stop(ブレークポイント設定)、trace(トレース出力) である。stop は、[global predicate] で指定した大域的条件が成立した時に、全プロセスを停止するという、ブレークポイント設定命令である。また、trace は、[global predicate] で指定した大域的条件が成立した時、[expression] で示す式を出力し、実行を継続する。

ここで、[global predicate] が Conjunctive Predicate(単一プロセスに関する条件式を and で結んだもの) である場合には、条件式が成立する最小限の実行を行なった時点で全プロセスを停止させることが可能である [8]。それに対し、Disjunctive Predicate(単一プロセスに関する条件式を or で結んだもの) である場合には、最小限の実行を行なった地点で停止させることは不可能であることが求められている [8]。従って、一般に stop の条件式に一般の and/or からなる条件式が与えられたとき、および、複数の stop 命令が与えられたとき、そのうちのただ一つの Conjunctive Predicate に対してのみ、最小限実行地点で停止させることが可能であり、それ以外の条件式については、条件式を成立したことを (あとから) 検出して停止させることしかできない。この、ただ一つの Conjunctive Predicate を Primary と呼ぶ。Primary を変更する命令が、switch である。stop, trace コマンドを入力した時に、番号 (commandno) が割り振られる。switch は指定した [commandno] のコマンドの [predicateno] 番目の Conjunctive Predicate を Primary とする。Primary predicate には status コマンドを実行した時に * 印がつけられる。デフォルトでは最初に設定したコマンドの最初の Conjunctive Predicate が Primary となる。

7 ddbx-p の評価

本節では、ddbx-p の評価について述べる。まず、履歴保存のための手間の増加について [12] と同様の測定を行なった。測定を行なった計算機は SUN4-260(OS は SUNOS-4.0.3、メモリ 64MB) である。通常実行において `NOWAITMODE` の msrcv を行なった場合、1 回あたりの CPU time は約 153μs となった。それに対し、履歴保存実行を行なった場合、1 回あたりの CPU time は約 196μs であった。この差約 43μs が履歴保存のための実行時間の増加分である。msrcva、msrcvs の場合も、保存するデータ量および保存手続きは同一であるので、実行時間の増加は受信命令でほぼ同一であった。従って、メッセージ通信の頻度が 4 ～ 5 ミリ秒に 1 回であれば、履歴保存のために実行時間がのびる割合が 1% 程度となる。

また、1 回の受信命令ごとに保存される履歴情報は数バイト (大抵の場合、2 ～ 3 バイト) である。従って、受信命令ばかりを繰り返すプログラムであっても、1 秒あたり 10 数 K バイトの履歴情報が保存される。従って、数十分程度の履歴を保存することも可能である。

また、ddbx-p をグループ内での分散プログラム作成において実際に試用し、評価を受けた。主な意見を以下に記す。

- 一般に、プロセス間通信としてソケットなどを用いる場合、プロセス間通信のために多くの手続きを記述する必要があるが、ここで提供している通信プリミティブは簡単に使用できる。その代わりに、デバッグをしない通常実行のみである場合にも init を用いてプロセス起動を行なわなければならないという欠点を持つ。
- 最小限実行を行なった地点で全プロセスを停止できる機能は、あるメッセージの送受信のあたりに誤りがあり、送信側と受信側のどちらにバグがあるかわかっていない場合に特に有効である。このとき、受信時点にブレークポイントを設定すると、送信側プロセスは対応する送信直後の状態で停止するので、送信側と受信側を対比して見ることができる。
- 本デバッガはプロトタイプ版であるため、プロセスの制御には UNIX 上のデバッガである dbx を使用している。従って dbx と対話的処理を行なうために実行速度は遅い。例えば、ブレークポイントを 1 つ設定して動作させた場合でも、通常実行の場合と比べて実行速度は 10 倍程度遅くなる。ブレークポイントの数が増える、あるいは式が複雑になると速度はさらに遅くなる。
- プロセス数固定のため、プロセスの動的な生成 / 消滅 (fork を行なうプログラム) に対処していない。また、デバッグ対象プロセスと非対象プロセスとが混在した分散プログラム (例えば、デバッグ済みのサーバを使用するクライアントプログラム) のデバッグが不可能である。

また、分散プログラムのデバッグの効率向上のためには、動作状況のユーザへの提示方法も重要である。メッセージの流れに沿ったデバッグ手法、すなわち、ユーザの着目プロセスをメッセージ通信に応じて送り手 / 受け手プロセス間で移動する手法により、分散プログラム動作状況の理解が容易になり、デバッグの効率が向上すると考えられる [1]。

8 あとがき

分散プログラムの開発支援用に、(1) 非決定的通信の再演機能、(2) 大域的条件によるブレークポイント・トレース設

定機能、(3) 与えられた大域的条件に対する最小限動作機能を持つ、分散プログラム用デバッガのプロトタイプ版 ddbx-p を試作し、その評価を行なった。今後の課題としては、プロセスの動的な生成・消滅への対処、デバッグ対象プロセスと非対象 (デバッグ不可能あるいは不要) プロセスが混在した分散プログラムのデバッグへの対処などの機能拡充が考えられる。

[謝辞] 本研究を進めるに当たって御指導頂いた、ソフトウェア研究所尾内理紀夫グループリーダに感謝します。また、再演法などのアルゴリズムについて御討論頂いた今瀬真担当部長、高橋直久主幹研究員、インプリメント手法について御教授頂いた下村隆夫主任研究員に感謝します。さらに、デバッガの試用をして頂いた分散型ソフトウェア研究グループの皆様に感謝致します。

Reference

[1] 青柳、真鍋:“分散プログラミングのためのデバッグ手法の一提案”, 情報科学若手の会シンポジウム (July 1991).

[2] R. Curtis and L. Wittie: “BugNet: A Debugging System for Parallel Programming Environments,” Proc. of 3rd Conf. on Distributed Computing Systems pp. 394–399 (Aug. 1982).

[3] R. E. Fairley: “Software Engineering Concepts”, McGraw-Hill, pp. 288–289.

[4] D. Haban and W. Weigel: “Global Events and Global Breakpoints in Distributed Systems”, 21st Hawaii Int. Conf. on System Sciences, pp. 166–175 (Jan. 1988).

[5] 萩原憲一: “分散環境における問題解法のメッセージ複雑度について”, Proc. of 8th Mathematical Programming Symposium, Japan, pp.41–50 (Nov. 1987).

[6] P. K. Harter Jr., D. M. Heimbiigner and R. King: “IDD: An Interactive Distributed Debugger”, Proc. of 5th Int. Conf. on Distributed Computing Systems, pp. 498–506 (May 1985).

[7] T. J. LeBlanc and J. M. Mellor-Crummey: “Debugging Parallel Programs with Instant Replay,” IEEE Trans. on Comput., vol. C-36, no. 4, pp. 471–480 (April 1987).

[8] 真鍋、今瀬: “分散プログラムのデバッグにおける大域的条件について”, 信学技報 COMP89-99 (Dec. 1989).

[9] C. E. McDowell and D. P. Helmbold: “Debugging Concurrent Programs”, ACM Computing Surveys, Vol. 21, No. 4, pp. 593–621 (Dec. 1989).

[10] B. P. Miller and J.-D. Choi: “Breakpoints and Halting in Distributed Programs,” 8th Int. Conf. on Distributed Computing Systems, pp. 316-323 (June 1988).

[11] D. Z. Pan and M. A. Linton: “Supporting Reverse Execution of Parallel Programs”, Proc. of the ACM Workshop on Parallel and Distributed Debugging, pp. 124–129 (May 1988).

[12] 高橋直久: “データ共有型並列プログラムの部分再演法について”, 信学技報 COMP89-98 (Dec. 1989).