共有アプリケーションのための同期実行モデル

Jacques GARRIGUE† 西村 進 § Susumu NISHIMURA 中島 玲二 † Reiji NAKAJIMA

† 京都大学数理解析研究所
Research Institute for Mathematical Sciences, Kyoto University
{garrigue,reiji}@kurims.kyoto-u.ac.jp

§ 京都大学大学院理学研究科数学教室
Department of Mathematics, Kyoto University susumu@math.kyoto-u.ac.jp

本稿では、インタラクティブな共有アプリケーションの柔軟な構築のための、分散同期実行モデルを提案する。本稿で提案する実行モデルでは、各ユーザ間で共有するデータとそうでないデータを区別し、共有データに関する操作のみを同期実行することによってデータ全体の一貫性を保証すると同時に、通信のオーバーヘッドによる遅延を抑えることができる。このような分散同期実行モデル上でのプログラミングを支援し、かつデータの一貫性が保たれていることを検証できるような Java の言語拡張を提案する。

1 はじめに

本稿は、インタラクティブな共有アプリケーションのプログラミングについて考察する。我々が参加している SOBA プロジェクト [5, 6] は、仮想的な共有空間の中に様々なアプリケーションを介してコミュニケーションを取らせることを目指す。そのためにインタラクティブな共有アプリケーションの開発を簡単にするフレームワークを提供するが、この研究はフレームワークの基礎になる。

インタラクティブな共有アプリケーションとは、ネットワークでつながった異なる計算機で動くアプリケーションを、あたかも複数のユーザが同時に共有して利用しているかのように感じさせる分散アプリケーションのことを言う。ここでは、そのようなアプリケーションの一例としてチャット・アプリケーションを挙げる。アプリケーションの画面を図 1 に示す。

このような共有アプリケーションを開発するに当って、いくつかの問題を解決しなければならない。まず第一に、何が共有されるべきかを明確にしなければならない。以下では、GUI 付きアプリケーションの一般的なモデルである、MVC(モデル=状態・ビュー=表示・コントロール=制御) フレームワークで考える。

すぐにわかるのは、ビューは共有されるべきではないということである。例えば、自分が何かを入力途中のとき、その内容が相手に見えても意味を成さないだろうし、自分の画面をスクロールしたときに相手の画面までスクロールすると邪魔なだけだろう。この観点から、同じ画面を全員に見せるというのは、アプリケーションの共有方法ではあっても、共有アプリケーションの開発方法には適してはいないと考えられる。このような画面共有の例としては、Microsoft NetMeeting のプログラム共有 [4] や Brown 大学の XMX (X protocol multiplexor)[1] が挙げられる。

共有アプリケーションの開発においては、状態と制御のなかで、共有するべきものと共有すべきでないものを区別することが開発のポイントになる。例えば、チャット・アプリケーションの例では、clear ボタンを押すことによって各ユーザは自分のログを消去することができるが、このとき他のユーザのログはそのまま残したい。このことは、状態 (この場合ログの中身) は必ずしもすべてが共有されるべきでないことを示している。また、共有しない状態があるということは、状態の制御の中にも共有されないものが存在することを意味している。

第二の問題は、共有した状態の一貫性をどうやって保持するかである。状態を一ヶ所だけに置くという方法もあるが、インタラクティブなアプリケーションでは様々な観点からそれを避けたい。まず、状態へのアクセスに通信時間がかかり、そのためユーザには反応が遅く見えてしまう。さらに、もしもその状態をある特定のユーザの所に置いた場合、そのユーザが意図的またはネットワークの障害によって共有アプリケーションから離脱した場合、全員がその共有アプリケーションを使えなくなってしまう。

共有アプリケーションを動かしているユーザ全員が共有状態のコピーを持てば、このような問題は避けられる。ただし、状態が変更される度に状態をすべて全員に送信する方法では、通信量や通信の遅れが問題になる。本稿で提案する実行モデルでは、このようなオーバーヘッドを避けるため、分散同期実行モデルにおける並列計算の SPMD (Single Program Multiple Data) 実行方式 [3] からヒントを受けた方法を用いる。具体的には、共有状態の変更を送るのでは

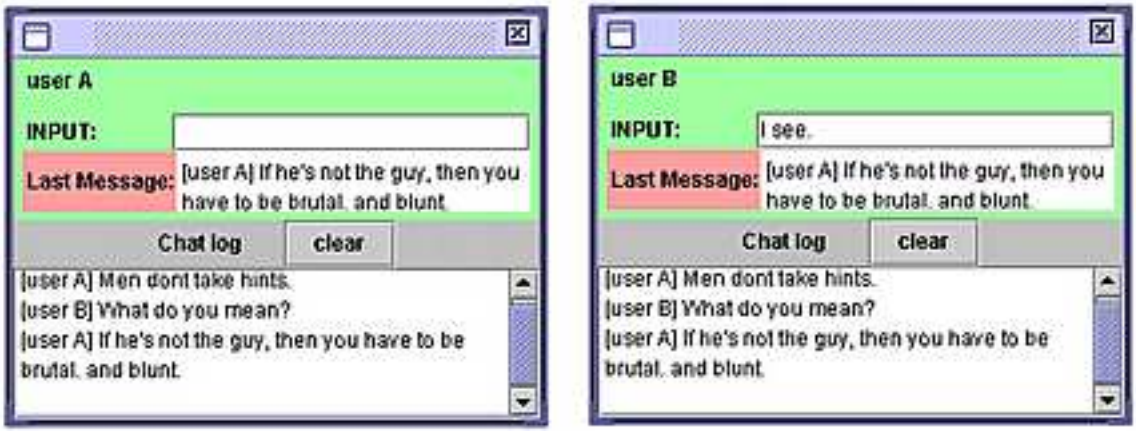

図 1: チャットアプリケーションの画面例

なく、そのきっかけとなったイベントと、それを処理するためのコードだけを共有することによって、状態の同期を保証する。状態の共有を提供している点では、Groove[2] の実行モデルにも似ているが、Groove は共有状態の差分を送信する方式を選んでおり、上記のような通信量の最適化は行われていない。

以下、本稿の構成はこの通りである。2 節で分散同期実行モデルを説明し、3 節で分散同期実行モデル上でのプログラミングを支援するための言語拡張について紹介し、最後にまとめを述べる。

2 分散同期実行モデル

前節の要件を満たすような、GUI 付き共有アプリケーションを構築するための枠組みとなる、**分散同期実行モデル**を提案する。この分散同期実行モデルは、MVC モデルにおけるモデル M が持つ状態を複数のホストで選択的に共有しかつ、共有した状態の一貫性が論理的に保たれるように拡張したものである。ただし、ここで言う MVC モデルとは、図 2 で表されるような構成を持つものとする。ビュー V は外部からのイベント (マウスやキー入力など) をコントロール C に通知する。C は通知されたイベントにしたがってモデル M の状態を変更する。その際、M の状態を読みことも可能である。M は状態の変更を V に通知することによってビューに反映させる。

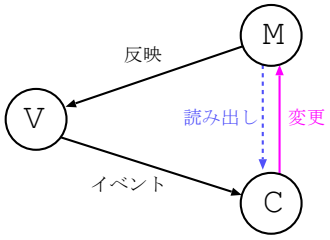

図 2: MVC モデル

2.1 グローバルモデルとローカルモデル

状態を複数のホストで選択的に共有するため、モデル M を、**グローバルモデル**と**ローカルモデル**の 2 つに分割する。(以下、グローバルモデルを GM、ローカルモデルを LM とも書く。) グローバルモデルは、モデルが保持する状態のうち、複数のホストで共有される部分を表す。ローカルモデルは、残りの部分、すなわち各ホスト独自の状態を保持する。グローバルモデルは、下に示す機構により、すべてのホストで常に同一の状態を保持するように一貫性が保たれている。

例えば、前節のチャットプログラムを作るには、Last Message フィールドの GUI のモデルをグローバルモデルで、その他の GUI コンポーネントのモデルをローカルモデルで表せばよい。Last Message フィールドは、そのモデルがすべてのチャット参加者の間で共有されているので、どの参加者の画面でも常に同じメッセージを表示する。一方、INPUT フィールドや Chat log のエリアの GUI に関してはモデルが共有されないので、各参加者が並行してチャットメッセージを入力したり、自分の Chat log だけを消去したりすることができる。

2.2 グローバルコントロールとローカルコントロール

上記のグローバルモデル/ローカルモデルによる表現が意図した通りの実行結果に結び付くためには、グローバルモデルの状態の一貫性を保つ必要がある。MVC モデルにおいて一貫性が崩れる可能性があるのは、コントロール C がローカルモデルの状態に依存してグローバルモデルの状態を変えてしまうときで

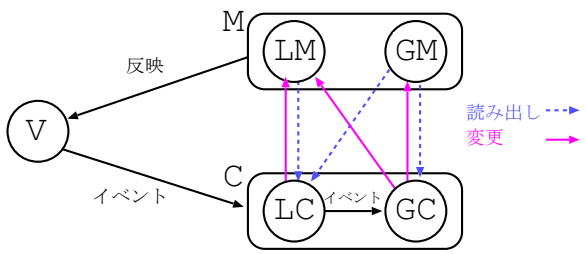

図 3: 拡張された MVC モデル

ある。なぜなら、ローカルモデルの状態は各ホストによって異なっているから、そのような変更を行うとグローバルモデルの状態が各ホスト毎にばらばらになってしまうからである。

このような事態を防ぐため、コントロール C も、モデル M と同様に**グローバルコントロール**と**ローカルコントロール**とに分割する。(以下、グローバルコントロールを GC、ローカルコントロールを LC とも書く。) グローバルコントロールは、すべてのホストで同期実行されるモデル M に対する操作を表し、ローカルコントロールは各ホストで独立に実行されるモデル M に対する操作を表す。これによって、MVC モデルは図 3 のように拡張される。ビュー V にとっては、GM も LM も従来通りひとつのモデル M として認識される。コントロール C に関しても同様である。重要なのは、C と M の関係である。通常の MVC モデルでは、C は M の状態を自由に読み書きすることができたが、拡張された MVC モデルでは、GC は LM の状態を読むことはできず、また LC は GM の状態を変更することはできない。前者の制約は、同期実行中は各ホストに固有のデータに依存した計算が行われないように制限している。後者の制約は、同期実行中でないときのグローバルモデル GM の状態の変更を禁止している。これらの制約によって、グローバルモデル GM の状態の一貫性を保つことができる。

2.3 グローバルイベントによる同期

拡張された MVC モデルにより、共有状態の一貫性を保つことができるが、ではグローバルモデルの状態を変更するにはどうしたら良いだろうか。ビュー V の生成するイベントは各ホストに固有のものであるから、これを直接グローバルモデルの状態を変更するために使うことはできない。ある特定のホストで発生したイベントをそのホストだけでなく、すべてのホストの同期実行によって処理するような機構が必要である。

このような機構を実現するために、本稿では**グローバルイベント**の概念を導入する。グローバルイベントとは、ある特定のホストで発生したイベントをあたかもすべてのホストで同時発生したイベントと解釈したものである。もう少し詳しく言うと、ビュー V(あるいはローカルコントロール LC) がイベントをグローバルコントロール GC に通知したとき、そのイベントはグローバルイベントとして解釈され、GC は同期実行を始める。実際には、グローバルコントロール GC に対するイベントの通知がすべてのホストにブロードキャストされることによって同期実行が開始される。

チャットの例で説明しよう。ユーザ B が INPUT フィールドにメッセージを入力し Return キーを叩くと、入力イベントが発生する。このメッセージを Last Message フィールドに表示するためには、入力イベントをコントロールに通知すればよい。ここで、Last Message フィールドのモデルはグローバルであるから、そこで利用されるコントロールもグローバルでなければならない。したがって、入力イベントはグローバルイベントとなる。すなわち、あたかもその入力イベントがすべてのホストに同時に通知されたように解釈され、その結果、すべてのホストの Last Message フィールドがユーザ B のメッセージに一斉に変更される。グローバルコントロールからローカルモデルへの書き込みも許されているので、Last Message のモデルを変えると同時にそのメッセージを Chat log に追加すれば、全員のログにも表示される。

拡張 MVC モデル及びグローバルイベントの概念は、並列計算機実行モデルのひとつである SPMD 実行方式 [3] から、そのアイディアを得たものである。SPMD 方式においては、並列計算の対象となる配列要素は複数個の並列計算ユニットに分散して配置され、ループ変数などのスカラー変数の値は各計算ユニットにコピーされる。これにより、各計算ユニットは、そのユニットに割り当てられた配列要素の参照やスカラー変数に関する計算は、他のユニットとまったく独立して効率よく行うことができる。ただし、配列要素の値をスカラー変数へコピーするときは、一貫性を保つため、その配列要素が割り当てられている計算ユニットに配列要素の値をブロードキャストさせ、スカラー変数の値をすべての計算ユニット上で更新させる。拡張 MVC モデルにおけるグローバルモデルは SPMD のスカラー値に、ローカルモデルは配列要素におおよそ対応している。

```
class ChatModel {
    ChatListener listener;
    String name;
    global String lastMessage = "";
    global void globalMessage(String sender, String message) {
        lastMessage = "[" + sender + "]" + message;
        listener.messageChanged(lastMessage);
        listener.addToLog(lastMessage);
    }
    void sendMessage(String message) { globalMessage(name, message); }
    void clearLog() { listener.clearLog(); }
}
```

表 1: プログラム例

3 プログラミング

前節で見たように、分散同期実行モデルにおけるグローバルモデルの一貫性が保証されるためには、いくつかの条件が満されなければならない。

1. グローバルモデルの更新は全ホストで実行されなければならない
2. 更新の順番も同じでなければならない
3. 更新を行うコードはローカルモデルに依存してはいけない

(2) に関しては、グローバルイベントが同じ順番で全ホストに届くようなプロトコルを用意さえすれば、更新の順番が変わらないことを保証できる。

しかし、(1) と (3) に関して、プログラムの正しさに依存せざるを得ない。正しいプログラムを書きやすくするためには、プログラミング言語の表現力を高めることが望ましい。

3.1 言語拡張と実行モード

プログラミング言語のレベルでは、まとまったグローバルモデルやグローバルコントロールの概念がないので、それらをもっと細かい単位で定義する。Java をベースとし、新しい修飾子 global に次の意味を与える。

- global をメソッドの修飾子として使えば、そのメソッドがグローバルコントロールに属することになる。ビューやローカルコントロールからそのメソッドを呼ぶだけで、グローバルイベントが発生し、ブロードキャストにより全ホストで同じグローバルメソッドが呼ばれる。
- global をフィールドの修飾子として使えば、そのフィールドがグローバルモデルに属することになる。その中身はローカルメソッドでも読めるけれども、グローバルメソッド以外から書き込むことはできない。

表 1 にはプログラムの例を示した。ビューは省略されているが、モデルは簡単なので、ここではコントロールとモデルが一緒になっている。メッセージを入力すると、ビューは `sendMessage` というローカルイベントを起こす。ローカルコントロールでは、メッセージに (ローカルモデルに含まれる) 発信元を付けて、グローバルイベント `globalMessage` を起こす。そのイベントが全ホストで処理され、`lastMessage` を更新した後、ビューを更新させる。最後に `addToLog` でビューに含まれるログのモデルを更新する。

グローバルコントロールに含まれるメソッドからローカルな値やメソッドを参照することがありうる (例の場合は listener)。それに依存してグローバルモデルが変更されなければ問題にはならない。それを保証するためには、グローバルメソッドの中で実行モードを区別しなければならない。グローバル実行モードでは全員が同じ条件下で同じコードを実行しているが、それ以外の場合はローカル実行モードになる。実行モードの切り替えは次のような流れになる。あるイベントが発生することにより、ローカル実行 (イベントの発生元だけが実行) がまず始まる。グローバルイベントが発生するとグローバル実行モードに変わり、さらにホスト毎にローカルな処理を行うためにローカル実行に戻る。

ローカル $\longrightarrow$ グローバル $\longrightarrow$ ローカル (分散)

表 2 では各実行モードにおける制約を示した。モデルへのアクセス制限は前節で説明したとおりで、グローバル実行のときの通常フィールドの読み出しと、ローカル実行のときのグローバルフィールドの書き込みは禁止されている。メソッド呼び出しにも制限が付く。まず、ローカル実行からグローバルメソッドを呼ぶ場合、ブロードキャストを行うので、シリアライズ可能な引数しか渡せない。また、グローバル実行で通常メソッドを呼ぶためには、そのメソッドが**ピュア** (結果が引数のみに依存する) で、かつ全て

	通常フィールド	グローバルフィールド	通常メソッド	グローバルメソッド
ローカル	読み書き	読み出し	○	○ (ブロードキャスト)
グローバル	書き込み	読み書き	△	○ (直接)

表 2: 実行モードと制約

の引数がグローバルな値でなければならない。

グローバル実行のままでできない処理があれば、一時的にモードをローカル実行に戻すこともできる。ただし、ローカル実行からグローバル実行に戻ったとき、結果や例外を無視しなければならない。

3.2 モード解析

上記の制約が満たされていることを、以下のようなグローバル・ローカルのモード解析によって検証することができる。まず、グローバルメソッドの引数とグローバルフィールドをグローバルな値とする。(グローバルフィールドに関しては定義より。グローバルメソッドの引数はブロードキャスト時にコピーされるので、グローバルな値と見てよい。) また、クラスのローカルフィールドをローカルな値とする。グローバルメソッド中の局所変数やメソッド呼び出しについてはそのモードを以下のように解析する。

1. グローバルメソッドの局所変数がすべてグローバルと仮定して始める。
2. ある変数にローカルな値が書き込まれている場合、その変数のモードをローカルに変える。
3. 変数から読み出した値はその変数と同じモードになる。
4. ピュアな演算子やメソッドあるいはグローバルメソッドに、グローバルな値だけが渡されている場合、その結果もグローバルな値になる。実行モードは変わらない。
5. それ以外のメソッドを呼ぶ場合、あるいは引数にローカルな値が混っている場合、実行モードをローカル (分散) に変える。結果はローカルな値になる。
6. ローカルな値が条件分岐に使われる場合、各分岐の命令と結果がローカルモードになる。分岐の中で return 命令を使ってはならない。
7. 以上を、すべてのモードが定まるまで繰り返す。

簡単な手続型言語なら、これで正しい解析結果が得られる。ただ、Java には状態を持ったオブジェクトと膨大なライブラリがあるので、次の二つの点に気を付けなければならない。

- グローバルなフィールドに状態を持ったオブジェクトが入っている場合、その状態がグローバルでない実行モードで変更されないようにしなければならない。
- 各クラスとメソッドに対して、それがピュアかどうか、あるいはそれが状態を持っているかどうかの情報が必要。

これらの機能を持った Java の拡張では、分散同期実行モデルのプログラムが簡単に書け、安全性も確認できる。

4 まとめ

インタラクティブな共有アプリケーションのための分散同期実行モデルを提案した。SOBA フレームワークではまだデザインのための概念的なモデルとしてしか活用されていないが、これとプログラミング言語の拡張とを組み合せることにより、プログラム解析によるモデルの一貫性の検証を行うことができ、バグの起きにくいアプリケーション開発に役立てることもできるだろう。

謝辞

本研究成果は文部科学省科学技術振興調整費研究「広帯域通信網上の仮想空間応用ソフトの研究」によるものである。

参考文献

[1] John Bazik. XMX - an X Protocol Multiplexor. URL: `http://www.cs.brown.edu/software/xmx/`, Brown University CS Dept, 2002.

[2] Groove developer's reference guide: Platform overview. URL: `http://www.groove.net/devzone/`, Groove Networks, 2002.

[3] P.J. Hatcher and M.J. Quinn. *Data-Parallel Programming on MIMD Computers*. The MIT Press, 1991.

[4] NetMeeting: プログラムの共有. URL: `http://www.microsoft.com/japan/windows/NetMeeting/Features/appshare/default.htm`, Microsoft Corp, 2002.

[5] 篠田直樹, 中島玲二. SOBA で P2P コラボレーション DIY. *Software Design*, January 2003.

[6] 林良生, 角田誠, 篠田直樹, 中島玲二. SOBA フレームワークにおける P2P ネットワーク上の 同 期機構の実現. 情報処理学会第 65 回全国大会, No. 3A-2, March 2003.