B-030

モバイルエージェントシステム AgentSphere における 強マイグレーション機構の改良
Improvement of Strong Migration Mechanism for Mobile Agent System AgentSphere

鈴木 幸祐† Kousuke Suzuki
甲斐 宗徳† Munenori Kai

1 はじめに

ネットワークに接続された複数のコンピュータ環境で効率良く並列分散処理を行うためには、各コンピュータの処理能力や動的に変化する処理負荷を考慮して処理を分散する必要がある。しかしこれを考慮したソフトウェアを作成することは困難である。そこでソフトウェア自体が自律的に必要に応じて適切なコンピュータに移動して処理することが可能な自律型の並列分散処理システムが望まれる。この実現のために、自律的に生成・消滅・移動を行うモバイルエージェントを利用する方法が挙げられる。モバイルエージェントが移動するとき、移動前の処理内容を完全に持っていくことができる移動は強マイグレーションと呼ばれ、移動先でほぼ最初から処理を再開する移動は弱マイグレーションと呼ばれる。途中までの処理結果を無駄にしないためには強マイグレーションの方が望ましい。そこで著者らは強マイグレーションのモビリティを持つ AgentSphere と呼ぶモバイルエージェントシステムを開発している。

Java ではシリアライズ機能を使用することで、実行コードとヒープ領域内の情報の保存が可能であり、これを用いて弱マイグレーションの移動を実現できる。しかし、実行途中のプログラムを移動先で再開することはできない。これはスタック領域内の情報やプログラムカウンタを実行状態として保存する機能が JavaVM ではサポートされていないからである。そのため Java で強マイグレーション方式の移動を実現するためには、スタック領域内の情報とプログラムカウンタに相当する情報を含めて移動することが必要となる。ただしこれを Java で実装するのは容易ではない。 Java を用いた既存の強マイグレーション方式のモバイルエージェントとしては、JavaGo[1]、MOBA、Nomads、Brakes などが挙げられるが、これらの研究は JavaVM の変更かソースコード、バイトコード変換を行っている。JavaVM を変更する場合、JavaVM のバージョンアップごとに修正を行わなければならず、システム開発の負担が大きい。そこで本研究では、JavaVM に手を加えることなく強マイグレーション方式の移動を実現するためのソースコード変換方式を構築した。

2 JPDA（Java Debugger Platform Architecture）を用いた従来の強マイグレーション実現方式

JPDA(Java Platform Debugger Architecture)は Java のデバッグツールの API である。これを使用することによって、スタックに存在する任意の変数とプログラムカウンタの情報を取得することができる。ただし JPDA はあくまでもデバッグツールであるので、取得した変数を復元する機能は存在しない。プログラムカウンタに関しても移動先で再利用することはできない。著者らは強マイグレーションのプログラムを図１の「変換前」に示すように、移動する位置に migrate()という関数を記述することで実現させたい。

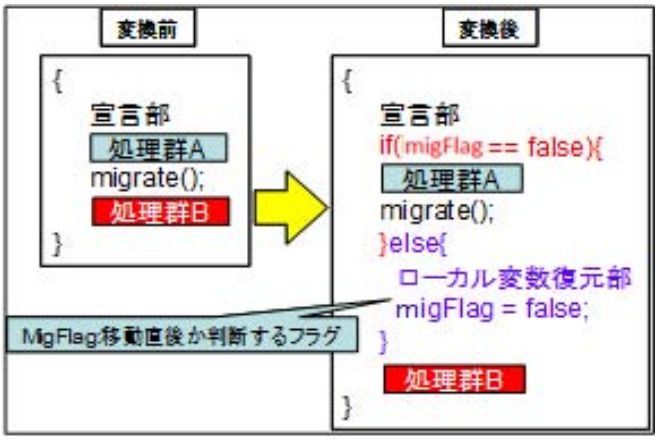

図１ 処理の流れの分割

このようなコードを JavaVM 上で動かすため、同図の「変換後」に示すように、移動に必要な処理を if-else 構造で挿入する方式をすでに提案してきた [2][3]。このとき、ローカル変数の取得や復元については JPDA を利用していた。

しかし、JPDA では参照型をそのまま取得することはできず、プリミティブ型の変数に行き着くまでスタックを探り続けなければならない。従って取得した変数がユーザー定義クラスであれば復元可能であるが、外部ライブラリなどの場合は元のクラスに復元することは不可能となる。例えば、図２のようなコードを移動させようとすることを考える。

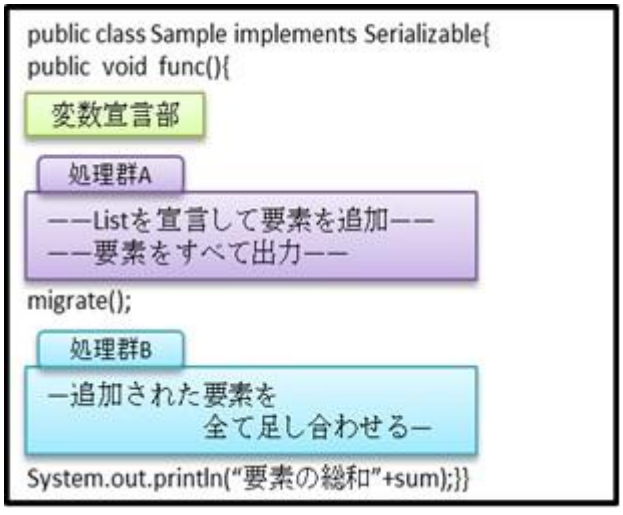

図２ 変換したいコード

† 成蹊大学 Seikei University

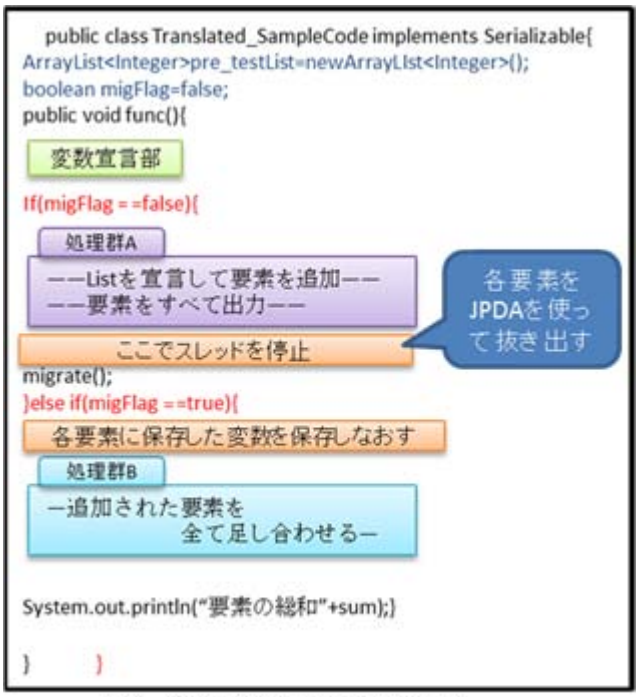

図３　JPDA を利用した変換後コード

ここで宣言されている List は JavaAPI の参照型クラスとする。図３に JPDA を用いた従来方式での変換後コードの形を示す。ここでは、変換後に追加・変更されるコード部分は青字と赤字で示している。赤字部分は移動前の処理と移動後の処理を区別するための if-else の構造であり、青字部分はローカル変数の保存・復元に関する部分である。

このように、JPDA を使いスタックフレームの情報を取得しようとする場合、List の持つ個々の要素に対してプリミティブ型に行き着くまで探し続け、その値を保存していくことになる。しかもこの処理の間、スレッドを強制的に止めるので、オーバーヘッドが生じることになる。さらにこの変数の復元に至っては、プリミティブ型として保存した変数を元のクラス構造に復元しなければならず、図３の例では List クラスなので要素を追加していくことで対応できる。しかし、内部の変数にアクセスできないような参照型クラスであった場合は復元することはできない。このように実行中のスレッドからスタックフレーム内部の情報を抽出する方法では取得できる変数が限定されてしまうという問題が存在していた。

3 ソースコード変換方式の改良

前章で述べた要因により、参照型のローカル変数を使用したプログラムでも強マイグレーション可能な変換方式を新たに構築した。

この変換方式では JPDA を使わずに、そのプログラム中の保存・復元が必要なスタックフレーム内の変数を探し出し、その変数を対象のプログラム内のヒープ領域に保存するコードを生成する。これにより参照型のローカル変数を持つエージェント全体をシリアライズして移動させる事ができる。図４にこの方式で図２のソースコードを変換した例を示す。

図４　新方式の変換後コード

この方式によって、以前の方式では移動対象のスレッドを外部から停止させていたために問題となっていたオーバーヘッドを軽減し、任意の参照型のローカル変数を持つエージェントを強マイグレーション可能にすることができた。

4 終わりに

新たな強マイグレーション機構を導入したことにより、以前では対応できていなかった参照型の変数情報の移動に対応し、さらに JPDA を使うことにより生じていたオーバーヘッドを軽減できる強マイグレーション変換方式を実現することができた。今後の課題としては、スタックフレーム保存部の効率化を検討する必要がある。さらに AgentSphere からの移動要求に応じたエージェントの自律的な移動の実現も挙げられる。

[参考文献]
[1] 米澤明憲,関口龍郎,橋本政朋：「移動コード技術に基づくモバイルソフトウェア」 http://homepage.mac.com/t.sekiguchi/javago/index-j.html, Jul.2011 参照可
[2] 加藤 史彬,田久保雅俊,櫻井康樹,甲斐宗徳「コード変換による強マイグレーション化モバイルエージェントの実現」, FIT2007, B-024, Sep.2007
[3] 加藤 史彬, 近藤敬宏 甲斐宗徳「強マイグレーションモバイルエージェントの自己バックアップ機能とエージェント間通信の実装」, FIT2009, B-010, Sep.2009