Java 言語環境におけるスレッド移送手法と移動エージェントへの応用

首藤一幸　村岡洋一

{shudoh,muraoka}@muraoka.info.waseda.ac.jp

早稲田大学 理工学部

概要

Java^{TM} 仮想計算機をまたいだスレッド移送を実現した。Java ベースの既存の移動エージェントでは移動後実行が継続されないため、プログラマはその不連続性に注意してコードを書かねばならない。スレッド移送では実行コンテクストを含めた移動が可能なため、移動を考慮したプログラム設計を行う必要がない。また、移送システムを負荷分散の機構として利用したり、スレッド外部化でプログラムの耐故障性の向上を図ることが可能である。本稿では、スレッド移送および外部化など要素技術の手法、実装した移送システム、いくつかの応用、および性能評価について報告するとともに、Java 仮想計算機で実行コンテクストを扱う上での諸問題を明らかにする。

Thread Migration Facilities for Java and Its Application to Mobile Agent

Kazuyuki SHUDO　Yoichi MURAOKA

School of Science and Engineering, WASEDA University

Abstract

We have been developing a thread migration system for Java^{TM}. Transferring a mobile agent described with existent libraries for Java destroy its execution context, so the libraries require programmers to pay attention to discontinuation of the execution. With our system programmers can write mobile agents in the same way as codes which run locally because thread migration makes transfer with execution context possible. One can make use of the system as a mechanism of load balancing and improve fault tolerance of one's programs with thread externalization. This paper describes techniques of thread migration and its elements such as thread externalization, an implementation of the system, some applications, the results of performance evaluations and problems that we encountered in dealing with execution context.

1　はじめに

Java^{TM} は 1995 年に Sun Microsystems(Sun) 社が発表した言語環境で、その中心は言語仕様 [4] と、仮想計算機 Java Virtual Machine(JVM) の仕様 [7] である。その最大の特徴のひとつは、JVM で実行される JVM バイトコードがプラットフォーム非依存なことである。このプラットフォーム非依存性に着目しての、Java 言語環境をベースとした分散システム構築基盤研究、開発が盛んである。

実行主体を他の計算機上へ移動可能にする、移動エージェント、移動オブジェクト、あるいは移動コードは、通信先への移動による通信遅延の削減、移動先での計算能力の利用、移動先に特有なサービスの利用などを実現する。Java 用の移動エージェント記述ライブラリもすでにいくつか [3][6][10][1] 公開されている。

これら既存システムでは、エージェントの移動において実行は継続されない。移動までの実行コンテクストは捨てられ、移動後は利用者またはシステム指定のメソッドが起動される。そのため、プログラマは移動にともなう実行の不連続を意識しつつ、ローカルに動作するプログラムとは異なる作法で

エージェントを記述する必要がある。プログラミングインタフェースとして透過性が低い。

本研究では Java のスレッド移送を実現し、スレッド移送システム MOBA を提供している。JVM 上のスレッドを実行中の状態を保ったまま JVM 間で移動させ、移動先で実行を継続させることが可能である。移送スレッドを移動エージェントとみなせば、移動エージェントとローカルに動作するコードの差をメソッド呼び出しひとつとできる。また、Java のプラットフォーム非依存性により、従来のプロセスマイグレーションとは異なり、異種のプラットフォーム間で移送が可能となっている。

スレッド移送およびその要素技術は移動エージェントの記述以外にも、負荷分散、プログラムの耐故障性向上などに応用が可能である。

以降本稿では、プロセス移送との相違、スレッド移送および外部化など要素技術の手法と実装を述べる。続いて、JIT コンパイラと本研究の関係、native methods の使用に関する既存システムと本研究の立場について論じる。移送システムの構成といくつかの応用を述べた後、最後に性能評価の結果を挙げ、考察する。

2　スレッド移送

2.1　移送対象

JVM 上には実行主体としてスレッドがあり、スレッド群を束ねるスレッドグループがある。スレッドグループは、利用することでスレッド群の整理や一括操作が可能になるものの、それ自身は一般的な OS 上のプロセスのようにファイル記述子などの資源は所有しない。スレッド群を束ねる枠でしかない。

つまり、スレッドグループの移送はスレッド移送の上に成り立つ。ゆえに、本研究ではスレッドを移送対象としている。

2.2　プロセス移送との比較

従来移送対象とされてきたのは OS 上のプロセスと、JVM 上のスレッドについて、移送対象としての性質の違いを論じる。

転送が必要な記憶の量　JVM 上スレッドで OS 上プロセスの仮想記憶に当たるのは、スレッドから参照を辿って到達し得るオブジェクト群である (図 1, 図 2)。OS 上プロセスではページが単位で、JVM 上スレッドではオブジェクトが単位である。このため、転送が必要な記憶の総量は、プログラムの性質が同じであっても OS 上プロセスと JVM 上スレッドで変わり得る。

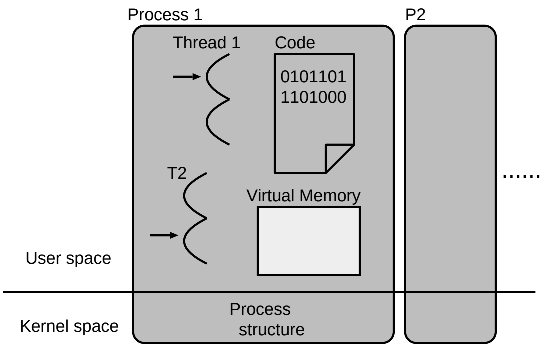

図 1: OS 上のプロセスのモデル

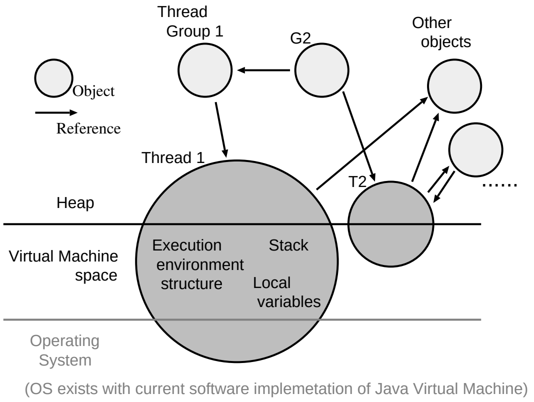

図 2: Java Virtual Machine 上のスレッドのモデル

OS 上プロセスでは dirty なページをまるごと転送する必要があるので無駄が生じ易いが、JVM 上の配列オブジェクトを疎に利用すれば同様のことが起き得るし、lazy copy などの最適化やページの圧縮まで視野に入れると、一概に断じることはできない。

実装したシステムでは、転送するオブジェクトは圧縮していないが、コンパクトな部類のスレッドで 200 バイト前後と、一般的な OS の 1 ページよりもかなり小さくできた。

共有する資源の量　OS 上プロセスと JVM 上スレッドでは、実体間で共有する資源の量が異なる。

通常、プロセス群が仮想記憶を共有するにはシステムコールの明示的な呼び出しが必要であるなど、基本的に所有する資源は分離されている。JVM ではスレッド群がオブジェクトを共有することはごく一般的に行われる。

実装したシステムでは、移送スレッドから到達し得るオブジェクト群すべてを移送先へコピーしている。スレッド群がオブジェクトを共有するプログラムでは移送によってセマンティクスが変わり得るので、それを理解してプログラムを書く必要がある。

3　手法と実装

次の要素技術でスレッド移送を実現している。

- クラス定義の転送
- スレッド外部化
- Object Marshaling
- リフレクション

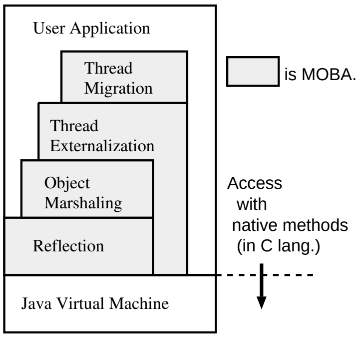

図 3: スレッド移送の要素技術

3.1　スレッド移送

スレッドの移送とは、次の 2 つの移送である。

- スレッドの状態
  スレッド外部化 (3.3節) でバイト列化。
- スレッドが必要とするクラス定義

スレッド移送では、スレッドの状態だけではなく、クラス定義、つまり型定義とコードの転送 (3.2) も必要である。

3.2　クラス定義の転送

移送スレッドが必要とするクラス定義を移送先で得られない場合、実行は続けられない。必要なクラス定義が得られることを保証するために 2 つの方法が考えられる。

- 移送先として考えられる計算機に、利用者がインストールしておく。
- システムが自動的に転送する

利用者によるインストールには実行時に転送が起きないという利点、自動転送には手間がかからないという利点がある。

利用者によるインストールでは、移送スレッドの移送先がインストール済み計算機に限られてしまう。また、複数計算機へのインストールにかかる手間の大きさは致命的と考え、実装したシステムでは自動転送を採用した。スレッドの生成元計算機からスレッドが活動している計算機へ転送する。

自動ロードの場合、転送するタイミングが移送スレッドの実行性能などに影響を与える。

- 静的な転送　スレッド移送時に、移送先で必要となるクラスを列挙し転送する。
- 動的な転送　実行時、移送先で必要になったら転送する。
- 混合　静的、動的転送の混合。

静的に転送するだけでは、クラス名を実行時に得たり文字列で与えるプログラムで、必要なクラス定義が得られないことがある。完璧な転送は不可能である。

実装したシステムでは、動的な転送を採用している。さらに、ある程度のクラス定義を移送の時点で静的に転送してしまうことで、実行中に起こる転送を削減することを検討している。

3.3　スレッド外部化

スレッド移送は、スレッドの外部化、転送、内部化で実現されている。本研究では、スレッドの状態をバイト列化して JVM 外部の実体とすることを外部化、逆にそのバイト列から JVM 内にスレッドを再構成することを内部化と呼ぶ。

外部化によって移送だけでなくファイルへの保存も可能となる。JVM や計算機の異常に備えてスレッドを保存して、プログラムの耐故障性向上に利用することができる。

3.3.1　手順

外部化、内部化の手順を述べる。

自身の外部化のための前処理　外部化対象スレッドは一時停止している必要がある。つまり、自分自身は外部化できない。外部化の過程で自身の状態は必ず変化するからである。自身の外部化は別にスレッドを生成してそのスレッドに任せる。

排他制御, 一時停止　対象スレッドのモニタを取得し、一時停止させる。外部化の途中でスレッドの状態が変化することを防ぐためである。

バイト列化　計算主体をバイト列化する標準的なプロトコルは存在しない。Open Management Group で検討されている Mobile Agent Facility[2] でも Agent System 間のインタフェース策定が目標で、計算主体の表現にまでは立ち入っていない。

スレッドをバイト列化する手順は次の通りである (図 4)。

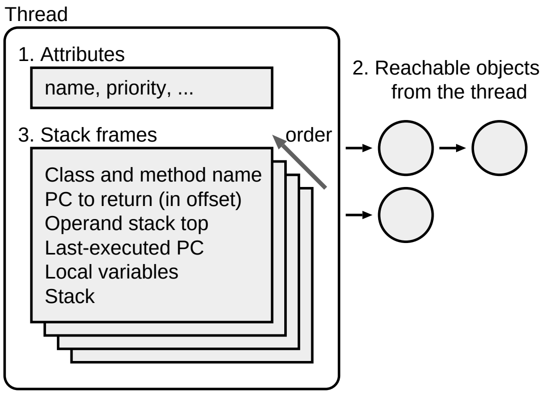

図 4: スレッドの外部化手順

1. スレッドの属性
   生成元計算機のIPアドレス,
   スレッドの名前, 優先度, デーモンスレッドかどうかなど
2. スレッドから辿り得るオブジェクト群
   marshaling(3.4)でバイト列化
3. スレッドの実行コンテクスト
   メソッド呼び出しで作られるフレームの連鎖を、呼び出しが古い方から順に辿る
   - (a) クラス, メソッド名
   - (b) return先 (コード先頭からのoffset)
   - (c) オペランドスタックの先頭位置
   - (d) 最後に実行された命令のアドレス (offset)
   - (e) 局所変数
     メソッドが持つ局所変数表の各要素を辿る
     - i. その時点で該当変数が生存していれば変数の値
       基本データ型であれば即値,
       オブジェクトであればmarshaling結果のバイト列
   - (f) スタック
     オペランドスタック上の各要素を辿る
     - i. 要素の値

3.4 Object Marshaling

スレッド外部化の際に、スレッドから到達し得るオブジェクト群もバイト列化する必要がある。このバイト列化はObject Marshalingで実現されている。Javaの標準APIに含まれるObject Serialization[8]相当の技術である。

3.4.1 位置依存の資源

別の計算機上では意味を持たないオブジェクトが存在する。ローカルファイルシステム上のファイルやファイル記述子、またソケット記述子に相当するオブジェクトなどである。

marshal, unmarshalの際これらに配慮しないと、利用不能になったオブジェクトにメッセージ送信して不都合が起きたり、unmarshalしたオブジェクトが意図せぬ資源と結び付いて不正な操作をしたりし得る。

これら位置依存のオブジェクトに対しては無効化または補整処理を施せばよい。位置依存性についてオブジェクトは次のように分類できる。

- 位置依存
  - 同一スレッドによるmarshal,unmarshalでは特別扱い不要
  - 常に無効化または補整処理が必要
- 位置非依存

システムの現状は次の通り。

- 標準API中のいくつかのクラスについて無効化が可能。
- 補整処理は未実装。
- 移動可能な代替クラスライブラリは未提供。

3.5 リフレクション

次の機能を提供している。

- オブジェクト、クラスのフィールドの読み書き
- クラス定義中の定数表 (constant pool) へのアクセス
- メソッド呼び出し

特にフィールドの読み書きはmarshalingに不可欠の機能である。

4 JITと実行コンテクスト

現在のシステムは、Just-in-Time コンパイラ (JIT) と同時に利用できない。

スレッド移送には実行コンテクストの機種非依存表現が必要である。バイトコードのインタプリタのみ働いている状況ではプログラムカウンタ (PC) はバイトコードを指すので、そのオフセットは機種非依存である。ところがJITコンパイラが働くとPCはプロセッサネイティブなコードを指すので、機種非依存表現を得ることができない。

プロセッサネイティブなコード上のオフセットをバイトコード上のオフセットに変換できれば機種非依存表現が得られる可能性はあるが、JVM の JIT インタフェースにこの変換は含まれていない。また、バイトコードが複数の機械語命令に展開されることを考えると、この変換は現実的ではない。

既存の Java 用移動エージェントで、移動の際に実行コンテクストを保存できないのはこのためである。

5 Native Methods

Native methods は C または C++で記述された Java のメソッドである。実行時には機種依存のライブラリとしておく。Native methods の使用には次のようなデメリットが伴う。

- 実行時に機種依存のライブラリが必要となり、バイトコードの可搬性が損なわれる。
- インストール時にコンパイルが必要となる。
- アプレットで使用しづらくなる。

言語として Java を選択する以上その使用は極力避けるべきである。

MOBA 本スレッド移送システムでは次の目的で native methods を使用している。

スレッド外部化 実行コンテクストへのアクセス

リフレクション クラス、オブジェクトのフィールドへのアクセス

Object Marshaling 基本型配列の読み書き

JIT コンパイラとの同時使用ができないという問題をカバーするため、Object Marshaling の性能を native methods である程度最適化している。それ以外は必要なところで最小限の使用にとどめている。

実行コンテクストへのアクセスには native methods が必要であり、native methods なしにスレッド移送は実現できない。また、実行コンテクストの JVM 内部表現は JVM の実装依存ゆえ、スレッド外部化に関する native methods は JVM の種類に依存してしまう。実装したシステムは Sun の Java Development Kit(JDK) に依存している。

既存システム 既存の Java 用移動エージェントシステムは native methods を使用していない。既存システムが移動の際に実行コンテクストを扱えず、捨ててしまうのはこのためである。また、実行コンテクストを扱おうとすると、4節で述べたようにプロセッサネイティブなコードへの変換系との同時動作で問題が生じる。既存システムではこれらのデメリットを考慮して、実行コンテクストの扱いは切り捨てているものと思われる。

6 スレッド移送システム

実装したスレッド移送システム MOBA の構成を述べる。

6.1 動作環境

本システムは、Sun の提供する Java Development Kit(JDK) 1.1 以上を必要とする。また、UNIX 互換のいくつかの OS でのみ動作確認をしている。

6.2 構成

システムはクラスライブラリと place から構成される。

6.2.1 Place

Place(図 5) は移送スレッドが動作する場である。

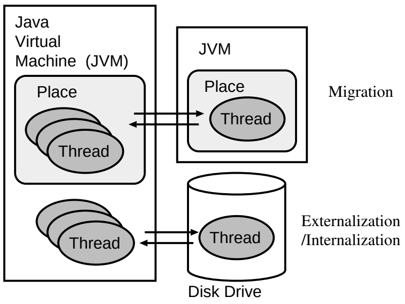

図 5: Place

Place はひとつの JVM を占有するが、同一計算機上には複数存在し得る。同一計算機上の place 群はポート番号で識別される。

ホスト名、ユーザ名ベースのアクセス制限が可能である。

6.2.2 サブシステム

本システムは、次のサブシステムから成る。

スレッド移送系 実行コンテクストの転送を行う

クラスロード系 移送スレッドにクラス定義を提供する

コマンドシェル系 利用者からのコマンドを受け付ける

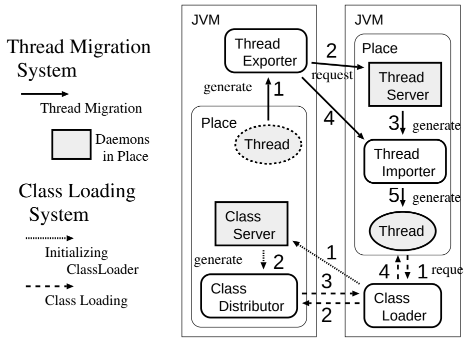

図 6: サブシステム

コマンドシェル　Place はキャラクタベースのユーザインタフェースを持つ。利用者はネットワーク経由で place に接続して指令を与えることができる。指令として次のものがある。

list　Place 上のスレッド一覧を表示する。
start　スレッドを起動する。
stop　スレッドを停止させる。
move　スレッドを移動させる。
fork　スレッドのコピーを別の place 上に作成する。
write　スレッドを外部化する。
read　スレッドを内部化する。
quit　シェルを終了させる。
...

7　応用

7.1　移動エージェント

7.1.1　既存のシステム

移動エージェントを記述できる Java 用ライブラリが、Aglets Workbench[3], Kafka[6], Voyager[10], Odyssey[1] などすでにいくつか公開されている。

これら既存システムでは、エージェントが移動する際に実行コンテクストは保存されない。移動後は利用者やシステム指定のメソッドが起動される。移動によってそれまで実行されていたメソッドとは別メソッドに制御が移るため、プログラマはそれを意識してローカルに動作するプログラムとは異なる作法でエージェントを記述する必要がある。

7.1.2　MOBA

スレッド移送システム MOBA の移送スレッドは移動エージェントとみなせる。移送スレッドは実行コンテクストを含めて移動し、移動先で実行は継続されるので、移動エージェントの記述にローカルなプログラムと異なる作法は不要である。

7.1.3　プログラミングインタフェースの比較

移動エージェントの記述法を、MOBA と ObjectSpace Voyager 1.0 で比較する。

7.1.4　Voyager 1.0

`Agent` クラスのサブクラスとして記述。移動は、移動先と移動先で呼び出したいメソッド名を引数に `moveTo()` メソッドを呼ぶ。

```
import COM.objectspace.voyager.*;

public class MobileAgent extends Agent {
  public void callback(Object arg) {
    System.out.println((String)arg);
  }

  public void go (String address
        /* is <host:port> */) {
    moveTo(address, "callback", "Hello.");
    System.out.println(
      "This statement is never executed.");
  }
}
```

オブジェクトの移動と、移動したオブジェクトに対するメッセージ送信である。移動のために `moveTo()` の呼び出すと、移動後は `moveTo()` の引数として与えたコールバックメソッド `callback()` に制御が移る。

7.1.5　MOBA

通常のスレッドの記述では `Thread` クラスを使用するところ、`MobaThread` クラスを使用する。それ以外は通常のスレッドと同様に記述する。

次のようにスレッドを生成し

```
MobaThread t = new MobaThread(...);
t.start();
```

次のようにスレッド自身で移動する。

```
MobaThread.goTo(移動先計算機);
```

7.2　プログラムの耐故障性向上

スレッド外部化を利用して、実行中のスレッドをファイルやデータベースに保存することが可能である。また、緊急の電源断など計算機の停止や故障をあらかじめ検知できれば、先だって状態を保存したり、他の計算機へ移動することができる。いずれにせよ、計算結果を失わずに済む。

実行に数週間といった長い時間がかかるプログラムには、途中の状態を保存、保存した状態から復帰する仕掛けを用意することが一般的である。スレッド外部化を利用すれば、これら非本質的な作業は不要となる。

プログラムにとっての異常事態には、計算機のシャットダウンといったあらかじめ検知可能な事象も多い。ところが Java のプログラムでそれらの事態を検知する手段がない。方法を検討している。

7.3 負荷分散

利用者が手で、またはスレッドが他スレッドや自分自身を移動させせることで負荷分散が可能である。システムではポリシーは提供していない。利用者に委ねられている。

8 性能評価

スレッド移送のオーバーヘッド、転送性能を評価した。

移送遅延は ObjectSpace 社の Agent-enhanced ORB Voyager と比較、転送性能はスレッド移送で遠隔メソッド呼び出しと同じ動作を行っていくつかの ORB と比較した。比較対象として、MOBA と同じく Java に特化して利便性、機能を追求する方向で開発されている RMI[9],HORB[5] を選んだ。

8.1 実験環境

以降、次の計算機をそれぞれ **WS1**,**WS2** と呼ぶ。

WS1: Sun Ultra 2 (UltraSPARC 168MHz)

WS2: Sun Enterprise 3000 (UltraSPARC 167MHz)

JVM: JDK 1.1.5 (Green Threads 版), Sun が提供する High Performing JIT compiler(`libsunwjit_opt.so`)

ネットワーク: 100Mbps Ethernet (100base-TX, リピータ 1 台を経由)

4節で述べたように、現在の MOBA は JIT コンパイラと同時に利用できない。MOBA だけは JIT コンパイラを使わずに評価した。

Voyager はバージョン 1.0.1 を使用、HORB はバージョンは 1.3b1 を基本に、RMI と同様のバッファリングによって Sun の JIT コンパイラでも性能が出るように改変された unofficial な版を使用した。

8.2 転送遅延

byte 型配列ひとつを除いて余計な転送データをなるべく少なくした移動エージェントを記述し、WS1、WS2 間を往復させて時間を計測、片道の移動時間を算出した (表 1, 図 7)。

MOBA	219($msec$)
Voyager	454($msec$)

表 1: 移送遅延

Native methods(5節) を使用しているとはいえ、実行コンテクストも転送する MOBA の遅延は Voyager の半分以下である。

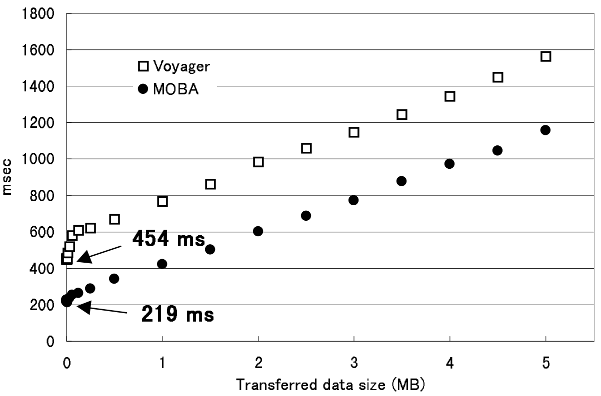

図 7: 移送遅延

8.3 転送性能

MOBA、RMI、HORB について、次の方法で転送性能を調べた。

ORB 2 種: double 型の配列を引数にとり何も返さないメソッドを遠隔呼び出しし、応答時間を測定。

MOBA: double 型の配列を保持して移動、配列を捨てて元の計算機に戻るまでの応答時間を測定。

スレッド移送では遠隔呼び出しをエミュレートして ORB と同等の処理をした。

応答時間を図 8に、応答時間を元に算出したスループットを図 9に示す。

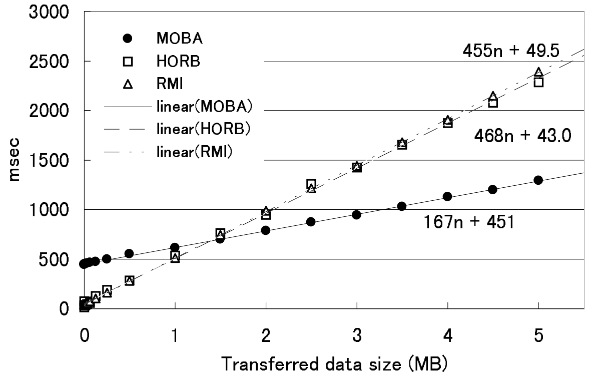

図 8: 応答時間

スレッド移送のオーバーヘッドは遠隔メソッド呼び出しより大きい。そのため、転送量が少ない状況で ORB の方が良い結果を出している。MOBA によるスレッド移送はオーバーヘッドが大きいもののスループットが高いので、転送量が充分に多い状況

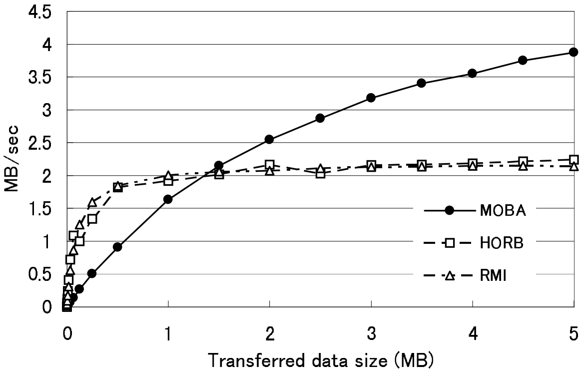

図 9: スループット

では ORB より良い結果を出している。とはいえ、MOBA は転送量が 1.3MB を超えてはじめて ORB より良い結果を出すので、MOBA で遠隔メソッド呼び出しをエミュレートするのは厳しいと言える。

MOBA のスループットが RMI,HORB と比較して高いのは、5節で述べたように基本型配列の読み書きを native methods で最適化しているからである。

スレッド移送の実用には、スループットの高さよりむしろオーバーヘッドの小ささが重要であり、この点はまだ最適化の必要、余地がある。

9 まとめ

実現したシステムについて、スレッド移送およびその要素技術の手法と実装を説明した。続いて、実行コンテクストの扱いと密接に関係する、JIT コンパイラとの関係、native methods の使用に関する既存システムと本研究の立場を論じた。また、システムの構成、スレッド移送の応用可能性を述べた。システムの評価として、既存移動エージェントシステムとプログラミングインタフェースを比較し、ローカルに動作するプログラムに対する MOBA の透過性を示した。性能面では他の移動エージェントより低い移送遅延、ORB より高いスループットが示された。

今後は、最大の問題である JIT コンパイラとの同時動作について可能性を探っていく。また、スレッドから到達し得るオブジェクトのコピーで移送を実現しているためにプログラムのセマンティクスが変わり得る問題について検討していく。

参考文献

[1] Agent technology. http://www.genmagic.com/agents/.

[2] Data Interchange Facility and Mobile Agent Facility RFP. http://www.camb.opengroup.org/RI/MAF/.

[3] IBM Aglets Workbench. http://www.trl.ibm.co.jp/aglets/index-j.html.

[4] James Gosling, Bill Joy, and Guy L. Steele Jr. *Java Language Specification*. Addison Wesley, 1996.

[5] Satoshi Hirano. HORB: Distributed execution of Java programs. In *Proceedings of World Wide Computing and Its Applications*, March 1997.

[6] Kafka: Java 用マルチ・エージェント記述ライブラリ. http://www.fujitsu.co.jp/hypertext/free/kafka/jp/.

[7] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specificatin*. Addison Wesley, 1997.

[8] Sun Microsystems. *Java^{TM} Object Serialization Specification*, 1997.

[9] Sun Microsystems. *Java^{TM} Remote Method Invocation Specification*, 1997.

[10] ObjectSpace: Voyager Overview. http://www.objectspace.com/Voyager/.