成蹊大学理工学研究報告
J. Fac. Sci. Tech., Seikei Univ.
Vol.43 No.1 (2006) pp.23-31

強マイグレーション化モバイルエージェントシステムの実装とそれによる自律分散処理システムの設計

桜井　康樹*1，佐々木　竜介*2，甲斐　宗徳*3

Re-design and Implementation of Strong Migration Mobile Agent System, AgentSphere for Autonomous Distributed Processing Systems

Yasuki SAKURAI*1, Ryusuke SASAKI*2, Munenori KAI*3

ABSTRACT : In this paper, we describe the re-design and implementation of AgentSphere, a mobile agent system based on the strong migration mobility. In order to implement the strong migration, all of execution code, program counter, contents of heap area and stack area have to be moved. To move execution code and contents of heap area, the serializing function of Java is used. We propose a moving method for contents of stack area by using Java Platform Debugger Architecture(JPDA). The contents of stack area are collected into serializable variables and then those variables are serialized together with contents of heap area. In Java, the program counter can not be used to resume any program code, so we propose a resumption method by using code conversion. In this conversion, three kinds of information, statement code, runtime code and statement information, are automatically generated from strong migration program code. Because the generated codes are weak migration codes, they can be executed on conventional Java virtual machines。

Keywords：Mobile agent, Agent systems, Strong Migration, Distributed processing system

(Received March 24, 2006)

１．はじめに

情報技術の進歩に伴い，ハードウェアによる処理能力が向上する一方で，解くべき問題の多様化や複雑化の進み方がより速いことがある。この問題を解決するための手法として分散処理が挙げられる。分散処理を行うことにより，ネットワーク上に接続された複数のマシンに処理を分散させ，単一のプロセッサにかかる負荷を分散させることが可能となり，処理の高速化が期待できる。

一般に分散処理システムでは，システム全体の状況把握や管理が必要となり，プログラム記述者がそれらの制御をすべて含めてアプリケーションプログラムを記述するのは非常に困難である。そこで，そのような困難さをプログラム記述者が意識せずに使えるような分散処理用プラットフォームを開発することにした。開発言語としては一般に広く普及し，OS や機種によらず使用することができる Java 言語を用いた。提案するシステムでは，システムの耐故障性の向上，システムの自律的行動によるシステム管理者とプログラム記述者側の管理負担の軽減，という目標を満足するため，モバイルエージェント技術を用いた。

しかし，多くのモバイルエージェントシステムでは，移動の際に移動前の実行状態は保存されないため，実行状態に応じた移動先の処理をプログラム上で記述する必要性がある。そこで本研究では，このモバイルエージェントシステムに実行状態を保持して移動するための手法と，その手法を用いたモバイルエージェントシステムを提案することにした。

*1 ：経営情報工学科学生
*2 ：情報処理専攻大学院生
*3 ：情報処理専攻教授(kai@st.seikei.ac.jp)
Professor, Dept. of Information Sciences

２．従来の自律分散処理システムの概要

2．1　自律分散処理システム

自律分散処理システムとは，システムの状況の変化にシステム自身が自律的に対応しながら分散処理を行うシステムである。一般的に分散システムにおいて，タスクの負荷分散や停止したタスクの再生処理等の作業にはシステム全体の状況把握や管理が必要であり，それらのコントロールをすべて考慮してプログラム記述者がアプリケーションプログラムを記述するのは非常に困難であると考えられる。この問題の解決策の一つとして，分散システムに自律性を持たせ，システム全体の状況把握や管理を全てシステム側に任せるという方法がある。分散システムの自律性を実現させるためには，システム内で動く各ソフトウェアが，他からのメッセージに応じた行動と，自ら取得した外部の情報に応じて判断を行い，その結果によって行動を計画し，その計画に基づき行動できる必要がある。そこで本研究では，自律分散処理システムを実現するにあたって，モバイルエージェントを利用することにした。

2．2　モバイルエージェントシステム

エージェントとは，人間の代理としてシステム上で自律的に行動するソフトウェアであり，モバイルエージェントとは，ネットワークを通じてマシン間を移動しながらタスク処理を行うことができるエージェントのことを言う。そして，モバイルエージェントシステムとは，モバイルエージェントが存在可能な空間を提供するプラットフォームであり，モバイルエージェントの動作（生成，消去，移動，保存，複製）や，モバイルエージェント同士の通信を提供するシステムのことである。既存のモバイルエージェントとしては，佐藤一郎氏（現国立情報学研究所助教授）が開発した AgentSpace[4]や日本 IBM 社によって開発された Aglets[5]等が挙げられる。

2．3　AgentSpace を使用した自律分散処理システムの概要

「分散処理の手法について詳しくない人でも，システムの自律性に助けられて簡単に分散処理の恩恵を受けられるシステムを構築すること」と，「様々な種類のプログラムを実行可能な汎用性の高いシステムを構築すること」を目的に，筆者らは自律分散処理用プラットフォームを開発している。1 つ目の目的を満たす為に，分散処理の開始や，システムの稼動状態の把握が容易となる様に GUI を用いて操作ができるようにした。また，2 つ目の目的である汎用性を高くするために，SPMD(Single Program Multiple Data) 形式のプログラムと，MPMD(Multiple Program Multiple Data)形式のプログラムのどちらでも自由に記述することができるようにした。

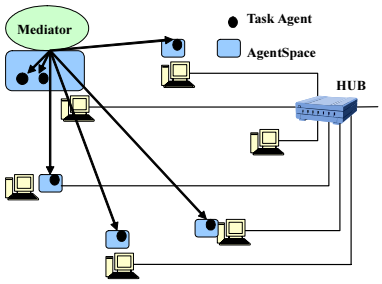

図１　自律分散処理システム構成例

これら 2 つの形式は，それぞれ対象とするアプリケーションによって使い分ける。SPMD 型の方は，ユーザにとって比較的書き易く，分散処理することにより処理時間の短縮が可能である。一方，MPMD 型の方は，モバイルエージェントとしての機能をプログラム記述者が自由に利用して，高度なタスクプログラムの記述が可能となる。[2]

図 1 は，自律分散処理システムの構成の一例を示したものである。

2．4　強マイグレーションの必要性

従来のシステムでは，モバイルエージェントシステムの動作原理の解析やシステムの改造が容易で，エージェント移動時に情報を圧縮して送るため移動が速いという理由から，弱マイグレーションベースのモバイルエージェントシステム AgentSpace を用いて自律分散処理用のプラットフォームの開発を行っていた。AgentSpace では，エージェント移動後の処理や，バックアップファイルを復元したときの処理はプログラム中の常に同じ場所から実行される。そのため，移動のきっかけとなる事象に応じた移動後の処理や，エージェントのバックアップポイントに応じた復元処理など，複雑なコントロールをあらかじめエージェントのプログラムに記述する必要がある。しかし，強マイグレーションベースのモバイルエージェントなら，プログラム記述者はこれらのことを気にすることなくエージェントを設計することができる。また，自律分散処理システムにおいて，ダウンしたタスクエージェントの復元処理の実現には，強マイグレーションのモバイルエージェントが適している[3]。そこで，本研究では，強マイグレーション方式を採用したモバイルエージェントシステムを実装することにした。

3. エージェントの強マイグレーション化

3. 1 Java 言語によるモバイルエージェントの問題点

Java 言語においても，C や C++と同様にプログラミングをする際にプログラムが使用するメモリ領域には，以下のものが挙げられる。

- 恒久変数領域
  static 宣言された変数やグローバル変数を格納する領域
- 実行コード領域
  プログラムの実行コードを格納する領域
- スタック領域
  一般的な変数を格納する領域
- ヒープ領域
  プログラムが使用できるメモリ領域から上記のものを差し引いた領域で，オブジェクトのインスタンス，メソッド・エリアなどのアプリケーション実行時に必要なデータを格納する領域

Java 言語を用いてモバイルエージェントを実装する際に，予め Java 言語に備わっているシリアライズ機能を用いることによって，プログラムコードに加えてヒープ領域内の情報を実行状態として保存が可能となっているが，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を Java 言語で実装するのは，容易ではない。これを行なうためには，Java Virtual Machine(JVM)の変更か，逐次の Java のプログラムコードを強マイグレーション用のコードに変換するための，ソースコード変換を必要とする。既存のモバイルエージェントシステムとしては，JVM の変更を行なっているものとして，産業技術総合研究所 グリッド研究センター首藤一幸氏により開発された MOBA，ソースコード変換を行なっているものとして，東京大学米澤研究室によって開発された JavaGO などが挙げられる[6][7]。JVM を変更する場合，大元の JVM のバージョンアップの都度 JVM を修正しなければならず，システム開発側の負担が大きいとともに，ユーザもその専用の JVM を使用せざるを得ない。そのため，Java 言語を用いたモバイルエージェントシステムの多くが，シリアライズ機能だけを用いて実装された，完全に実行状態の保存されていない弱マイグレーション方式のエージェントを採用していた。

3. 2 強マイグレーション化のための手法

本研究では，従来の自律分散処理システムにおいて使用していた AgentSpace を参考にして，JVM に手を加えることなくシステム開発を行えるように，スタックの取得とソースコード変換を採用する。それにより，強マイグレーション方式のモバイルエージェントシステムの開発を行う。

3. 2. 1 スタック領域の取得

スタックの中身を取得するために，JPDA(Java Platform Debugger Architecture)を使用した。これは，Sun Microsystems 社が提供する JVM に標準的に実装されており，Java アプリケーションのデバッグに使用されるものである[8][9]。実行中のスレッドや実行情報へのアクセスが可能となっている。本研究では，この機能を用いることによって，スタック領域のデータを取得することにする。

エージェントは，スレッド上で実行される。そこで，エージェントの移動の際に JPDA を用いてエージェントが実行されているスレッドにアクセスし，スタック内の情報を取得し，シリアライズ可能な変数に格納する。そして，実行コードやヒープ領域内のデータと共にシリアライズして，移動できるようにする。

3. 2. 2 ソースコード変換[2]

実行状態を完全な形で保存するためには，スタックの中身だけでなく，プログラムカウンタも必要となってくる。JPDA では，このプログラムカウンタに類似したデータを扱っており，これを取得することは可能となっている。しかし，この取得したデータを用いてプログラムを途中から実行する機能は，JPDA でもサポートされていない。そこで，本研究ではソースコードの変換を行うことにより，これをサポートすることにした。

ソースコード変換は，図 2 のようにユーザコードをソースコード変換器にかけることによって，ステートメントコード，ランタイムコード，ステートメントコードの各ステートメントに関する情報を取得することにより行われる。

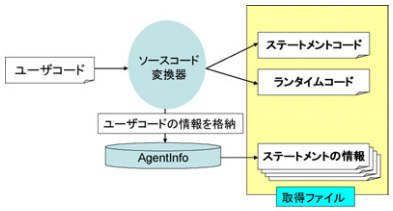

図 2　ソースコード

（1）ステートメントコード

ユーザコードの構造を解析し，ユーザコードから提供されるモバイルエージェントの挙動を指示するプリミテ

ィブによって分割したステートメントを statementN（N＝整数）というメソッドに，if の条件式のステートメントは branchStatementN というメソッドに書き込んだコードのことである。また，ソースコードを解析して繰り返しの回数がわかる for 文に関しては，現時点においてはその for 文をアンローリングすることによって，ステートメントコードを生成する。ソースコードを解析しても繰り返し回数がわからない for 文や, while 文, switch-case 文，再帰関数に関しては，現時点においてソースコード変換器でのステートメントコードへの変換には，対応していない。

図 3 は，ユーザコードのステートメントコードへの変換の例を示している。

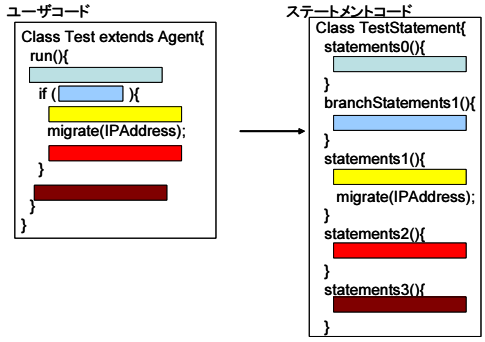

図３　ユーザコートとステートメントコード

また，ユーザコードに対して提供されるプリミティブには，次の 5 つがある。

- エージェントの生成：create(fileName)
- エージェントの消去：destroy()
- エージェントの複製：duplicate()
- エージェントの移動：migrate(IPaddress)
- エージェントの保存：backup()

図 3 のように，ステートメントコードは細かくステートメントごとのメソッドに分割されている。弱マイグレーション方式のモバイルエージェントシステムでは，移動後におけるメソッド単位での継続実行が可能である。そこで，本システムでは，移動後にこのステートメントコードのどのステートメントから処理を行うかを指定することにより，プログラムカウンタの機能をサポートすることにしている。

（2）ランタイムコード

ソースコード変換によって得られたステートメントコードの各ステートメントの情報に基づいて, エージェントの挙動（生成，消去，複製，移動，保存）を管理するコードである。

ランタイムコードの自動生成の実装にあたって，テンプレートを用意し，それを利用して必要な部分を変更しながらランタイムコードを作成するという手段で実装を行った。以下の図 4 はその変換の一部の例である。

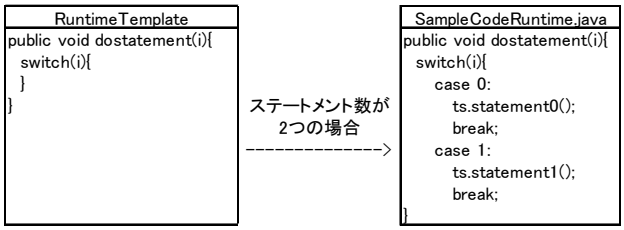

図４　ランタイムコードの自動生成

dostatement 関数は，ステートメントコードに記述されているコードを実行させるための関数であり，ソースコード変換を行った際に生成されたステートメント数に合わせて switch 文の中身を自動的に記述させている。ここで，switch 文内にある変数 ts は，生成されたステートメントコードのインスタンスである。

（3）ステートメントの情報

エージェントの状態を表すデータで，実行時にロードするクラス名やそのオブジェクトや生成したステートメントコードを元にした実行開始ステートメント，実行するステートメントや分岐ステートメント，次に実行するステートメントや分岐先のステートメント等の情報によって構成されている。

3．2．3　強マイグレーション化エージェントの移動

ソースコード変換によって生成されるステートメントコードとランタイムコードは, 既存の JVM のコンパイラにより，クラスファイルを取得することが可能となっている。強マイグレーション化エージェントは，この 2 つのクラスファイルと状態データ，そしてスタックを直列化･圧縮することによって移動する。

4．強マイグレーション方式のモバイルエージェントシステム AgentSphere の提案

前述の，強マイグレーション化モバイルエージェントシステムとして筆者らは AgentSphere[1] という独自のシステムの研究を行っていた。この章では，その旧システムからの変更点を中心にして AgentSphere システムについて述べる。

4．1　Agent 関連のクラス

旧 AgentSphere システムでは，Agent に関連するクラスは Agent クラスの 1 つだけだった。このクラスは，ユーザが記述する分散処理のプリミティブを提供するもので，強マイグレーション方式のモバイルエージェントのみをサポートするクラスとなっている。

新しい AgentSphere システムでは，強マイグレーション方式のモバイルエージェントと弱マイグレーション方式のモバイルエージェントの両方を扱うことが出来るように考慮した。その理由は，分散処理システム内を巡回しながらシステムを管理するようなエージェントを実装しようとした場合，弱マイグレーション方式のエージェントでもこの機能を実現することができるにも関わらず，強マイグレーション方式のモバイルエージェントで実装しようとすると，ソースコード変換によって移動するコードのサイズが大きくなってしまい，移動にかかるオーバヘッドが大きくなってしまうからである。そのため，2 つのモビリティのエージェントを採用するようにした。図 5 において，新 AgentSphere システムにおける Agent 関連のクラスの相関関係を示す。

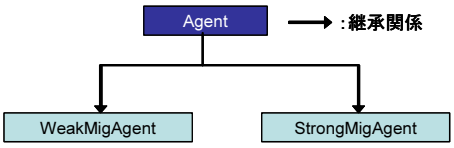

図５　Agent 関連のクラスの相関関係

Agent クラスは，エージェントに関しての最も基礎となるクラスで，エージェントの生成，消去，複製を表すプリミティブ，create(filename)，destroy()，duplicate()の 3 つを提供する抽象化されたクラスである。

また，StrongMigAgent クラスは，Agent クラスを継承したクラスで，強マイグレーション方式のモバイルエージェントを表すクラスである。このクラスが提供するプリミティブには，エージェントの移動と保存を表す，migrate(IPAddress)，backup()の 2 つがある。

そして，WeakMigAgent クラスは，Agent クラスを継承したクラスで，弱マイグレーション方式のモバイルエージェントを表すクラスである。このクラスが持つプリミティブには，移動を表す go(IPAdress)がある。また，ローカルでの処理を表すメソッド create()と，移動先での処理を表すメソッド arrive()を提供する。

4．2　システム関連のクラス

図 6 では，旧 AgentSphere システムのシステム関連のクラスの構成を表している。

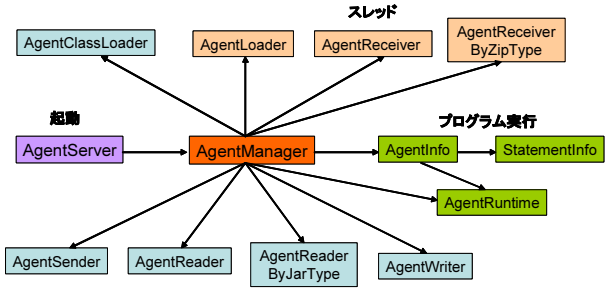

図６　旧 AgentSphere システムのクラス構成

図 6 を見てわかる通り，13 個のクラスから構成されている。これら各クラスの機能を見直してクラスの統合や，機能追加に伴う新しいクラスの追加を行った。図 7 は，更新されたシステム関連のクラスの構成を表している。

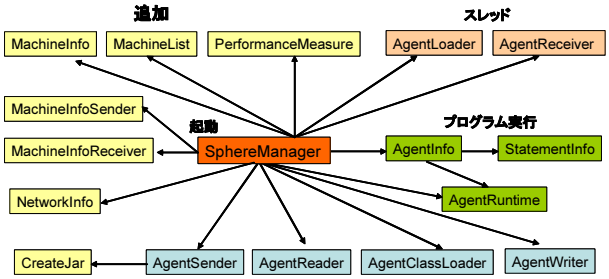

図７　システム関連のクラス構成

図 7 を見てわかるように，新しい AgentSphere システムは，SphereManager クラスを中心にした，17 個のクラスから構成されている。以下に，この構成からの変更点について述べる。

4．2．1　旧システムから変更されたクラス

- SphereManager クラス

  これは旧システムの AgentServer クラスと AgentManager クラスを統合したものである。SphereManager クラスは，AgentSphere システムの中で一番中心となるクラスで，全クラスの管理を行っており，必要に応じて，各クラスを起動し処理を行う。実際にエージェントの実行を行うのも，このクラスである。また，Port 番号の指定や，自ホストの IP アドレスと MAC アドレスの取得とそれらのブロードキャストを行う。

- AgentReader クラス

  実行するエージェントファイルの読み込みを行うク

ラス。これまでは，読み込むファイルの形式によってクラスが分かれていたが，これを統合して１つのクラスで複数のファイル形式に対応できるようにした。

- AgentReceiver クラス
  送信されたエージェントの受信を行うクラス。受信したエージェントが悪意のあるエージェントかの判定を行い，jar 形式のデータの解凍を行う。旧システムでは，AgentReader クラス同様，受信するエージェントの形式によってクラスが分かれていたが，これを統合して１つのクラスにした。

4．2．2　本年度新たに追加されたクラス

旧 AgentSphere システムでは，分散処理システムに適用するには不十分な点が残っていた。新システムでは，分散処理システムに適用するに当たって必要な機能の追加を行った。それらについて，以下に述べる。

- CreateJar クラス
  エージェント送信の際の圧縮を行うクラス。
- MachineList クラス
  システムに参加している全マシンについての情報を保持するクラス。
- MachineInfo クラス
  各マシンに関する情報(IP アドレス, MAC アドレス, 性能値)を保持するクラス。
- MachineInfoSender クラス
  自分以外のマシンに自分のマシンの情報のブロードキャストを行う。
- MachineInfoReceiver クラス
  他のマシンからのマシン情報を受信する
- NetworkInfo クラス
  自分のマシンの IP アドレスや MAC アドレスの取得を行う。
- PerformanceMeasure クラス
  性能値の計算を行う

５．他マシンとの通信

5－1．概　要

旧 AgentSphere システムには，ネットワーク上に存在する他マシンの認識，指定されたマシンへのマイグレーション機能が実装されていなかった。

自律分散処理システムにおいて，他マシンへのマイグレーション機能は必要不可欠なので，新 AgentSphere システムではその機能の実装を行った。

5－2．ネットワーク上にあるマシンの認識，リスト化

他のマシンを認識，区別するために，AgentSpace で実装されていたクラスを AgentSphere に追加し，システムの見直しに伴った修正を行った。以下が修正を行ったクラスである。

- MachineInfo クラス
  マシンの情報を保持するクラス。MAC アドレスの情報を追加。
- MachineInfoReceiver クラス
  他のマシンから送信されたマシン情報を受信するクラス。状況に合わせて自身の情報を送り返す機能を追加。

AgentSphere を立ち上げると，自身の MachineInfo をブロードキャストし，ネットワーク上に存在するマシンがそれを受け取り，各マシンが保持する MachineList へ登録する。

今までの AgentSpace では，起動時での他マシンの情報取得は，巡回型のエージェントの到着を待たなくてはならず，マシン数が多くなると表示に時間がかかってしまう。そこで，他マシンが起動したマシンから情報を受け取った時，自身の情報を返す機能も実装した。

しかし，この機能もマシン数が多くなってしまうと，情報をブロードキャストする時間，返された情報を処理する時間が多くなってしまう。そのため，ネットワーク上のマシンのグループ化が将来的に必要だと考えられる。

5－3．指定したマシンへのマイグレーション

AgentSphere では，ソースコード変換によって生成されたステートメントコード，ランタイムコード，状態データの３つのファイルが揃って初めてコードの実行が可能になるのだが，昨年の AgentSphere ではマイグレーションの際，状態データしか移動させていなかったため，他マシンでの実行ができないでいた。

旧システムの動作検証ではソースコード変換の妥当性の検証が中心だったので，マイグレーションを１台のマシンでしか行っていなかった。自分自身への状態データの移動であれば，そこにはランタイムコード，ステートメントコードがすでに存在しているので，正しく動作する。しかし，それでは他マシンへのマイグレーションが出来る，ということには当然ならない。他マシンには状態データだけ移動させても，ランタイムコード，ステートメントコードがそのマシンにはないので実行できないのである。

そこで，ランタイムコード，ステートメントコードおよびシリアライズ化された状態データの 3 つのファイルを，マイグレーションする直前に jar 圧縮して 1 つのファイルにまとめ，指定されたマシンへと送信し，受信後に解凍することによってマイグレーション後の実行継続が可能となった。

６．評　価

今回このシステムの動作確認をするに当たって，データ数 10 個のバブルソート(昇順)を 2 台のマシン間で行うことにした。図 8 は，今回の評価に用いたユーザコードを，そして図 9 はユーザコードをソースコード変換したステートメントコード, 図 10 にはランタイムコードを示す。

```
import java.io.Serializable;
import agentsphere.*;

public class SortingAgent extends StrongMigAgent implements Serializable {

    private int data[] = ソートを行うデータ;
    private int i, j, temp;

    public void run() {
        for ( i = 0; i < data.length - 1; i++ ) {
            for ( j = data.length - 1; j > i; j-- ) {
                if ( data[j - 1] > data[j] ) {
                    temp = data[j - 1];
                    data[j - 1] = data[j];
                    data[j] = temp;
                }
            }
            enumerate( data );    /*配列表示用の関数*/
            if (i%2==0) migrate("IPAddress2");
            else migrate("IPAdress1")
        }
    }
}
```

図８　ユーザコード

```
import java.io.Serializable;
import agentsphere.*;

public class SortingAgentStatement extends StrongMigAgent implements Serializable {

    transient private AgentRuntime theRuntime = null;
    private int data[] = ソートを行うデータ;
    private int i, j, temp;

    public SortingAgentStatement() {
    }

    public void setRuntime(Object obj){
       theRuntime = (AgentRuntime)obj;
    }

    public void statement0() {
        for ( j = data.length - 1; j > i; j-- ) {
            if ( data[j - 1] > data[j] ) {
                temp = data[j - 1];
                data[j - 1] = data[j];
                data[j] = temp;
            }
        }
        enumerate( data );    /*配列表示用の関数*/
        theRuntime.setTransferAddress("IPAdress2");
    }

    public void statement1() {
        for ( j = data.length - 1; j > i; j-- ) {
            if ( data[j - 1] > data[j] ) {
                temp = data[j - 1];
                data[j - 1] = data[j];
                data[j] = temp;
            }
        }
        enumerate( data );    /*配列表示用の関数*/
        theRuntime.setTransferAddress("IPAdress1");
    }

    public void statement2() {
        /*statement0と同じ処理*/
    }

    public void statement3() {
        /*statement1と同じ処理*/
    }
            ……

    public void statement8() {
        /*statement0と同じ処理*/
    }

    public void statement9() {
        /*statement1と同じ処理*/
    }
}
```

図９　ステートメントコード

```
import agentsphere.*;
import java.io.Serializable;
import java.lang.reflect.Method;

public class SortingAgentRuntime implements Serializable{
   SortingAgentStatement ts = new SortingAgentStatement();

   public void go(Object obj){
      AgentRuntime theRuntime = (AgentRuntime)obj;
      AgentInfo infoList = theRuntime.getInfoList();
      SphereManager theManager = theRuntime.getSphere();
      ts.setRuntime(theRuntime);
    .....;
   }

   public void doStatement(int i){
    switch(i){
         case 0:
            ts.statement0();
            break;
         case 1:
            ts.statement1();
            break;
         .....;
         case 9:
            ts.statement9();
            break;
      }
   }

   public boolean doBranchStatement(int i){
      switch(i){
      }
      return false;
   }
}
```

図 10 ランタイムコード

図を見てわかる通り，ソートを行う配列のデータの中で一番小さい値が確定したら次のマシンに移動し，次に小さい値を確定し，最初のマシンに戻って処理を行うというような処理を行っている。そして，移動前のマシンでの配列の状態が保存されているかの確認を行っている。

表 1 は，今回の評価に用いたマシンである。

表 1　評価に使用したマシン一覧

マシン名	CPU名	クロック数	メモリ
マシンA	PentiumⅣ	3.0GHz	1000MB
マシンB	PentiumⅣ	2.4GHz	768MB

マシン A の 1 台で処理を行った結果を図 11 に示す。

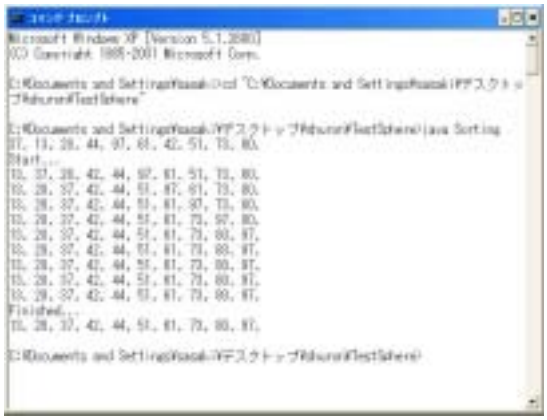

図 11　1 台のマシン（マシン A）での実行結果

次に，2 台のマシンで処理を行った時のマシン A での結果を図 12 に，マシン B での結果を図 13 に示す。

図 12　マシン A での実行結果

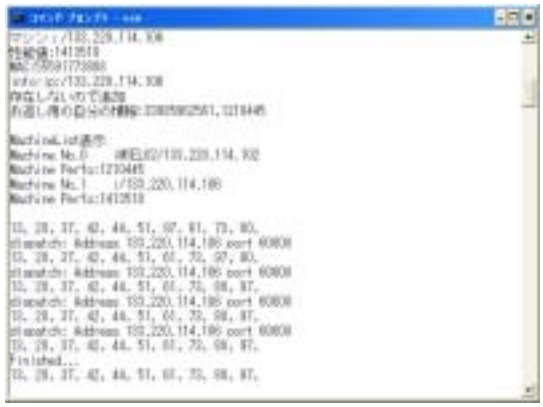

図 13　マシン B での実行結果

図 12 と図 13 が示すように，移動前の配列の状態を保持しながら移動し，移動後ではその続きから処理を行っていることがわかる。

7．おわりに

今回ソースコード変換を行うことによって，簡単なプログラムではあるが，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処理を行うことが出来た。ただし，現時点では次のような制限がある。

- 変数は，全て大域変数
  スタック領域のデータを使用していない
- 繰り返し回数のわかっている for 文の使用
  ソースコード変換を行うのが容易

また，本研究では，次のことが未実装である。

- スタック領域のデータの取得と再利用
- 完全なソースコード変換

そこで，強マイグレーション方式のモバイルエージェントのシステムを完成させるためには，以下の様なことをする必要がある。

- スタック領域のデータの再利用
  JPDA では，スタック領域のデータを取得するための API は提供されているが，そのデータを再びスタック領域にセットするための API は提供されていない。そこで，この機能をサポートするために，ソースコード変換の際に，取得したスタック領域のデータを利用できるようにステートメントコードを生成できるようにする。
- 完全なソースコード変換
  繰り返し回数のわからない for 文や while 文, 再帰関数などにも対応できるように，スタック領域内のデータを取得し，そのデータを利用してソースコード変換を行うようにする。

また，本研究で提案するモバイルエージェントシステムは，強マイグレーション方式と弱マイグレーション方式のエージェントが，システム上で共存している。それは，分散処理システムに参加するマシン間を巡回しながら各マシンの状況を管理するようなエージェントを作ろうとする場合，強マイグレーション方式のエージェントでは，通信のオーバヘッドが大きくなるため，弱マイグレーション方式のエージェントの方が適していると考えるからである。今回，2 つのモビリティのエージェントを実行可能なモバイルエージェントシステムを実現することで，より効率的な自律分散処理システムの構築が可能になると考える。

参考文献

[1] 坂井 功：「自律分散処理用モバイルエージェントシステム AgentSphere の開発」成蹊大学大学院工学研究科情報処理専攻修士論文, Mar.2005

[2] Ryusuke Sasaki, et al.: “Implementation and Evaluation of Autonomic Distributed Processing System Using Mobile Agent”, Proc.of IEEE Pacific Rim conference, No.237, Arg.2005

[3] 坂井 功　他：「自律分散処理用モバイルエージェントシステム AgentSphere の開発」成蹊大学理工学研究報告, Vol.42, No.1, Jun.2005

[4] 佐藤 一郎：「AgentSpace：モバイルエージェントシステム」, 日本ソフトウェア科学会, Dec. 1998

[5] 日本 IBM 東京基礎研究所：
http://www.trl.ibm.com/aglets/ ， Apr. 2006 参照可

[6] 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, Apr. 2006 参照可

[7] 首藤 一幸, 村岡 洋一：「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」, 情報処理学会 研究報告 pp.39-46, Apr 13, 1998

[8] Torsten Illmann, et al.: “Transparent Migration of Mobile Agents Using the Java Platform Debugger Architecture” Mobile Agents: Proceedings of the 5th International Conference, pp.198 – 212, Dec.2001

[9] Sun Microsystems: “Java Platform Debugger Architecture”,
http://java.sun.com/j2se/1.5.0/ja/docs/ja/guide/jpda/,
Apr. 2006 参照可