成蹊大学理工学研究報告
J. Fac. Sci. Tech., Seikei Univ.
Vol.42 No.1 (2005) pp.33-39

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

坂井　功[*1]，甲斐 宗徳[*2]

AgentSphere: A Mobile Agent System for Autonomous Distributed Processing

Isao SAKAI[*1], Munenori KAI[*2],

ABSTRACT：In the design of a mobile agent, it changes a lot by whether about an agent, which of three kinds of mobility, Code Migration, Weak Migration, and Strong Migration, is adopted and implemented. AgentSphere developed by our research is a mobile agent system based on the Strong Migration mobility. Therefore, on the machine by which AgentSphere is running, a mobile agent can move to other machines even from anywhere in a program code, and can resume processing. Although conventional developed systems has realized Strong Migration by changing the JavaVM itself uniquely, our system has the feature that, unlike such past systems, distributed processing program can be executed on the existing JavaVM by using source code conversion. By using AgentSphere, improvement in both throughput and reliability are offered by low cost, and it becomes possible to build the autonomic distributed processing system in which users without technical knowledge could also perform stabilized distributed processing easily

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

(Received March 25, 2005)

１．はじめに

近年，コンピュータネットワークの規模の拡大，情報通信機器の急激な進化に伴い，ネットワーク利用者は急速に増大している。それに比例するかのように，ネットワークの利用方法も高度化して，様々な情報検索の要求と応答や多様なマルチメディアデータがネットワーク上において流通し始めている。このため，ネットワークを介した情報処理の中核となるサーバマシンでは，膨大な計算を高速に処理するため，さらなる性能要求が高まっている。

その解決方法の１つとして，ネットワーク上に接続された複数のマシンに行うべき処理を分散させ，演算結果を統合するという方式を取ることで，全体のスピードを向上させることを実現する分散処理が挙げられる。しかし，分散処理の利用には，分散処理を行う環境を整えることや，分散処理のためのプログラミングなどの専門知識が必要となってくる。

そこで，本研究では，モバイルエージェントを用いて，処理能力・信頼性の向上を低コストで提供し，操作や，コーディングが簡単で，専門知識の無いユーザでも，手軽に分散処理を行えるような環境を提供し，安定した処理を実現することを目的としている。

ここでは，本研究での自律分散処理の特徴をさらに強化させるため，昨年度まで利用していた AgentSpace（お茶の水女子大学，佐藤一郎氏の開発したモバイルエージェントシステム）から，独自のモバイルエージェントシステム（AgentSphere）を導入することによって実現させることを目的としている。

２．従来のシステムの概要

2．1　エージェントのモビリティについて

モバイルエージェントは，ユーザプログラム自体を遠隔ノードに送って実行する遠隔プログラミングという概念を実現している。その設計において，プログラムの記述性に関係するものは，モバイルエージェントシステム

[*1] 情報処理専攻大学院生
[*2] 情報処理専攻助教授(kai@st.seikei.ac.jp)
Associate Professor, Dept. of Information Sciences

がどのようなモビリティを採用しているかに依存する。モビリティとは，プログラムが移動することはどういうことなのかという意味づけである。プログラムのモビリティの分類として大きく分けて次の 3 つが考えられる[1]。

1．遠隔実行（Code Migration）
　プログラムコードのみの移動
2．弱いマイグレーション（Weak Migration）
　プログラムの他に実行途中のデータを伴った移動
3．強いマイグレーション（Strong Migration）
　プログラムコードとデータに加えて，プログラムの実行状態を伴った移動

遠隔実行は，JavaApplet のようなプログラムの実行がノード内で完結するようなもので，プログラムを遠隔ノードに送って実行が開始できればよい。

本システムで利用している AgentSpace[2]や Aglet[3]等の Java ベースのモバイルエージェントシステムは，弱いマイグレーションを採用している。それは，Java に予め備わっている機能（ヒープ領域内のデータの直列化）で容易に実現可能だからである。プログラムの記述法は，エージェントの状態（生成，送信，受信，消滅，複製，保存）に応じて，コールバックするメソッドが決まっており，そのメソッド内にプログラムを記述する(図 1)。

```
create(){
  Local で処理
}
arrive(){
  移動先で処理
}
```

図1　弱マイグレーション

```
run(){
    Local で処理
  migrate(Host A);
    HostA で処理
  migrate(Host B);
    HostB で処理
}
```

図2　強マイグレーション

強いマイグレーションを採用したモバイルエージェントシステムでは，弱いマイグレーションのようなエージェントの状態に応じた処理を考慮せずに，プログラムを記述することができる(図 2)。昨今のエージェントブームのきっかけとなった Telescript[1]は，これを採用したモバイルエージェントシステムであるが，インタプリタで実行される Telescript 言語でプログラムを記述しなければならない。Java 言語ベースのモバイルエージェントシステムでは，MOBA[4]や JavaGO[5]が挙げられる。MOBA は JavaVM の変更やネイティブメソッドの追加によって，プログラムカウンタやスタック領域内の情報を移動することで実現している。しかし，JIT コンパイラを利用することができず，独自の JavaVM をダウンロードする必要がある。JavaGO では移動時にローカル変数を全て保存し，移動先で移動前の処理を再開できるようなソースコードに変換して処理を行う形となっている。しかし，次の 3 つの制限の元にユーザはプログラムを記述しなければならない。

1．複数のクラスで構成されたシステムは利用不可能
2．移動を行うかもしれないメソッドにはあらかじめ宣言しなければならないものがある
3．for 文の初期化部，条件部，繰り返し部の中では移動することができない

さらに，エージェント実行前にソースコード変換をするだけでなく，移動するたびにソースコード変換を行う必要があると言った問題点もある。

モバイルエージェントシステムの構築にあたって，遠隔実行，弱いマイグレーション，強いマイグレーションの中の，どのモビリティを採用するかどうか決めなければならない。

2．2　強いマイグレーションの必要性

本システムでは，弱い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 を採用している。そのため，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常に同じ場所から実行される(図 1)。つまり，これらの機能を弱いマイグレーションベースで設計するとなると，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をあらかじめ記述する必要があるため，エージェントを設計するユーザの負担が掛かる。しかし，強いマイグレーションベースのモバイルエージェントなら，ユーザ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ため，エージェント開発コストを削減することができる。

この節では，本システムの概要と，強い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（AgentSphare）を採用すると，どのような影響があるのかを述べる。

2．2．1　タスク分散用エージェント（Mediator）

Mediator[6]とは，AgentSpace 上で稼動するエージェントベースのアプリケーションである。Mediator は，ユーザからのタスク指定などの入力を受ける。Mediator を起動すると，使用可能なマシンの表示や処理結果が出力される GUI が表示される（図 3)。

その GUI からユーザがタスクを指定し，実行を開始することによって，ネットワーク上の AgentSpace が稼動しているマシンに，性能値を基にして，タスクエージェント（Distributed Agent）を分散させる。そして，各マシンで処理を終了したタスクエージェントは，再び，分散を

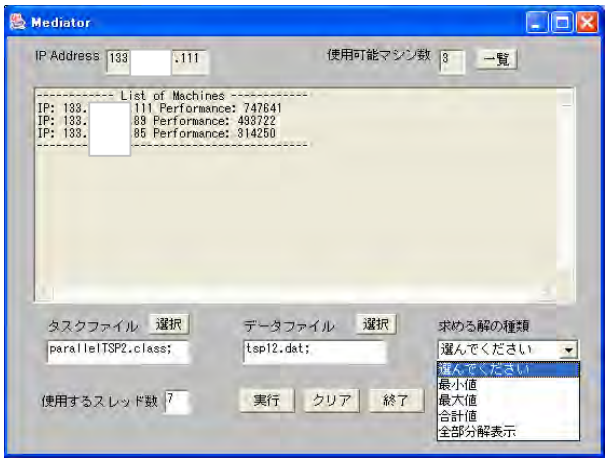

図３　Mediator の稼動

行った Mediator の存在するマシンの AgentSpace に戻り，Mediator によって，これらの処理結果（解）が統合され，最終的な解が Mediator のウインドウに出力される。

このようにタスクの分散処理を行うが，AgentSpace ベースのタスクエージェントは，自身の処理を終了するまで，そのマシンに滞留しなければならない。つまり，実行を開始したタスクエージェントは，外部アプリケーションや他のタスクエージェントによって処理速度が著しく低下した場合や，分散処理に参加したマシンが増減する場合など，状況に応じた動的な移動はできない。強い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ならば，タスクエージェントの実行を開始した場合でも，実行状態を保持した移動ができるため，環境の変化に応じたエージェントの移動が実現する。

2．2．2　管理エージェント

管理エージェント[7]は他の AgentSpace の情報(使用可能なマシンの IP アドレス，マシンの性能値，滞在するタスクエージェントの ID）を受け取り，タスク分散可能なマシンや，Mediator のタスク負荷分散処理，ダウンしたタスクエージェントの発見を行う。管理エージェントがダウンしたタスクエージェントを発見したとき，それを Mediator に伝え，Mediator がそのタスクエージェントを再発行する形となる。しかし，AgentSpace ベースのタスクエージェントでは，始めから処理を行わなければならない。強いマイグレーションベースのタスクエージェントならば，自身を定期的にバックアップし，それを管理エージェントに持たせることで，ダウンしたタスクエージェントを発見した場合，管理エージェント自身からタスクエージェントの再生が可能となる。また，再生されたタスクエージェントは，バックアップした場所から処理を再開することができる。

３．AgentSphere システム

AgentSphere は以下の 4 つの特徴を持つモバイルエージェントシステムである。

１．強いマイグレーションを採用
２．既存の JavaVM で実行可能
３．ソースコード変換
４．コード中のどこからでも移動可能

3．1　ソースコード変換

強いマイグレーションコードを，Java で提供している機能だけで実現させるために，ソースコード変換を使用した。ユーザコードはこのソースコード変換器によってステートメントコード，状態データ，ランタイムコードを取得する（図 4)。

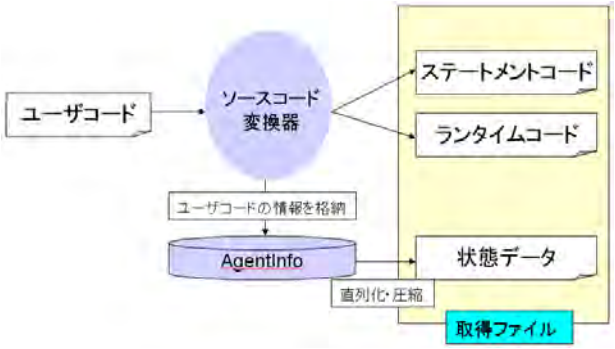

図４　ソースコード変換

ステートメントコードはユーザコードの文の構造を見返して，分割したステートメントを statementN（N＝整数）というメソッドに，for の内部のステートメントや if の内部（条件部）のステートメントは branchStatementN というメソッドに書き込んだものである（図 5)。その際，ユーザコードに提供されるプリミティブは次の 5 つである。

１．エージェントの移動:migrate(address)
２．エージェントの保存:backup()
３．エージェントの生成:create(fileName)
４．エージェントの複製:duplicate()
５．エージェントの消去:destroy()

状態データは，AgentInfo クラスを直列化・圧縮したもので，AgentInfo には，ロードするクラス名やそのオブジェクト，生成したステートメントコードを元に，実行を開始するステートメント，実行するステートメント，実行する分岐ステートメント，次に実行するステートメント，分岐先のステートメント等の情報を格納する。

ランタイムコードは AgentInfo からステートメントの情報を受け取り，ステートメントを実行させたり，次に

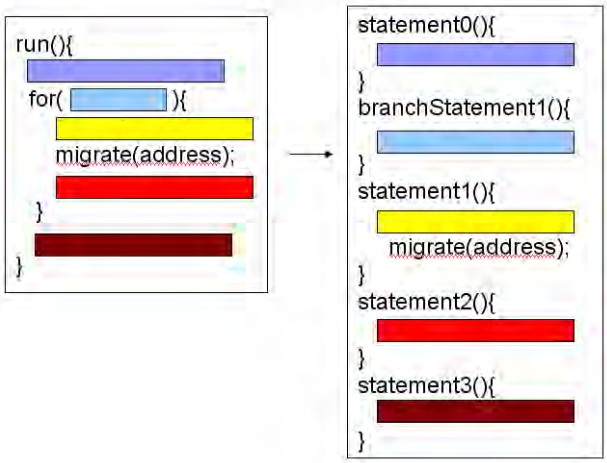

図5　ユーザコードとステートメントコード

実行するステートメントを決定して AgentInfo に渡したり，エージェントの動作（移動，保存，生成，複製，消去）を管理するコードである。

生成されたコードは既存の Java コンパイラにより，クラスファイルを取得できる。この 2 つのクラスファイルと状態データを合わせたものがモバイルエージェントになる。

3．2　AgentSphere

モバイルエージェント（ステートメントクラス・ランタイムクラス，状態データ）を動作させるアプリケーションである。これは以下のクラスで構成されている。

AgentServer

- □　AgentSphere の起動

AgentManager

- □　全てのクラスを管理
- □　エージェントの要求に応じたシステムへの呼び出しを実行

AgentRuntime

- □　エージェントを実行するためのスレッドを継承するクラス

AgentLoader

- □　ユーザからのファイル入力待ちをするクラス

AgentReader

- □　直列化データの復元するためのクラス

AgentClassLoader

- □　AgentInfo クラスからロードするクラス名を取得し，クラスファイルのロードを行うクラス

AgentSender

- □　エージェントの直列化・圧縮を行い，送信する

AgentReceiver

- □　エージェントの受信待ちをする
- □　受信したデータの解凍

AgentWriter

- □　エージェントを直列化しファイルに保存する

AgentSphere を起動することによって，そのマシン上でエージェントが存在することができる。ユーザに提供するプリミティブを元にしたエージェントの状態は，図 6 で表現される。

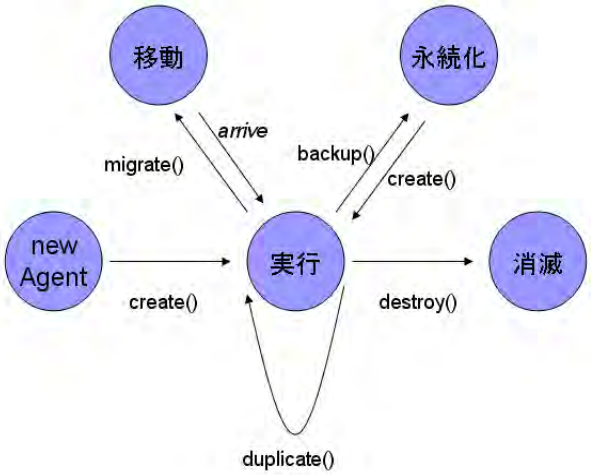

図6　エージェントの状態遷移図

3．2．1　エージェントの生成

AgentSphere がユーザから状態データを受け取ったときに，AgentManager はスレッドを継承するクラス AgentRuntime を生成し，状態データの解凍・復元を行う。そして，状態データから生成するクラスファイル名を受け取り，Java クラスローダを使用してエージェントを実行するクラス（ランタイムクラス）を生成する。そして，ランタイムクラスをインスタンス化してランタイムオブジェクトを取得し，コールバックしてランタイムを実行する。

3．2．2　エージェントの実行

生成されたランタイムオブジェクトは，AgentInfo から実行するステートメントを受け取り，ステートメントクラスの対応するメソッドを呼び出すことによって実行する。そして，分岐したときや分岐しないとき，関数が終了したときなどの条件を元に次に実行するステートメントを決定し，そのステートメントを AgentInfo に格納する。

3．2．3　エージェントの送信・保存

実行するステートメントが migrate メソッドや backup メソッドを呼び出したとき，ランタイムオブジェクトは，ステートメントオブジェクトと自身のオブジェクトを AgentInfo に格納し，AgentSphere がその AgentInfo を直列化し，zip 形式で圧縮を行う。そして，指定されたアドレスへ移動，もしくは，ディスクに保存する。

3．2．4　エージェントの受信・復元

状態データをネットワーク・ファイルから受信したとき，エージェントの生成と同様に，AgentManager はスレッドを継承するクラス AgentRuntime を生成し，状態データの解凍・復元を行う。そして AgentInfo からランタイムオブジェクトを取得し，ランタイムオブジェクトの実行メソッドをコールバックする。AgentInfo は送信・保存する命令を実行した後のステートメント位置を格納しているため，それをランタイムオブジェクトは取得し，対応するステートメントオブジェクトのメソッドを呼び出すことによって，移動・保存前の処理の続きを実行することができる。

4．評　価

4．1　ソースコード変換

1000 万回代入式と四則演算を繰り返すコードをそれぞれ逐次実行したときの処理時間と，2 つのコードをそれぞれ変換し AgentSphere 上で実行したときの処理時間を計測した（表 1）。

表 1　処理性能

	変換前	変換後	オーバーヘッド
代入式	214msec	553msec	339msec
四則演算	438msec	763msec	325msec

上記の 2 つは，共に 1000 万回繰り返して処理をしている。つまり，AgentSphere 上では，1 つのステートメントを 1000 万回実行していることになり，ステートメントを 1 回実行するのに，次に実行するステートメントを決め，その情報を AgentInfo に格納するためのオーバーヘッドが生じる。この処理を AgentSphere 上で実行したときのオーバーヘッドは，二つの処理を平均して 331msec（1 回あたりでは 33.1nsec）生じることを表している。しかし，繰り返し文の中をステートメントに分割するということは，その中でエージェントを移動・保存するメソッドがあるということを表している。通常，代入命令だけして 1000 万回移動したり，バックアップを取るようなプログラムの設計はしない。実際はステートメントの処理量を大きくし，間隔を持ったエージェントの移動や保存を行うようにするのが普通である。

また，変換前のコードと変換後のコードの容量を比較すると 2.5 倍あった。

4．2　モバイルエージェントの動作確認

モバイルエージェントの動作（生成・保存・消去・移動・複製）を確認するために，図 7 のソースコードを利用して処理を行った。図 7 のシステムの要求プリミティブには，エージェントの生成:create(fileName)，エージェントの保存:backup()，エージェントの消去:destroy()，エージェントの移動:migrate(address)，エージェントの複製:duplicate()がある。for 文の条件式部分 N は，繰り返し回数であり，動作確認をするものに応じて変化させた。

```
run(){
   n = 0.0;
   for(i=0;i<N;i++){
        n++;
        System.out.println(n);
         システムの要求プリミティブ
    }
    System.out.println("Answer   = "+n);
```

図 7　動作確認のためのコード

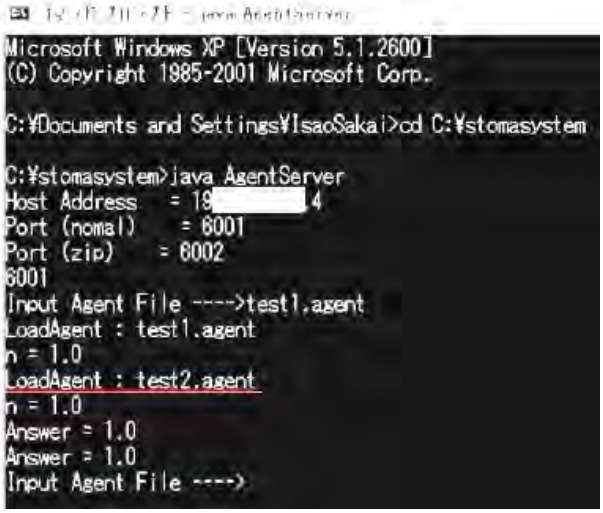

図 8　エージェントの生成:N=1

図 8 は実行中のエージェント（test1.agent）から新しくエージェント（test2.agent）を生成する動作を表している。図 8 の赤線を見ればわかるように test2.agent を test1.agent 実行時に生成することが確認できた。

図 9 はエージェントを保存する動作を表していて，変数 n のインクリメントが実行・出力されるたびに，その実行状態とデータをディスクに保存している。そのファイル名は，「エージェント名@BAK@エージェント ID.agent」という形式でディスクに保存される。ここでは，ディスクに保存するファイル名は test@BAK@0.agent（エージェントの ID は定義されていないのでここでは 0）となっている。実行状態が保存されているか確認するため，test@BAK@0.agent を入力したところ，test.agent が最後に保存した箇所からの処理を再開することが確認できた。また，エージェントの保存 1 回のオーバーヘッドは

2.21msec 生じることがわかった。

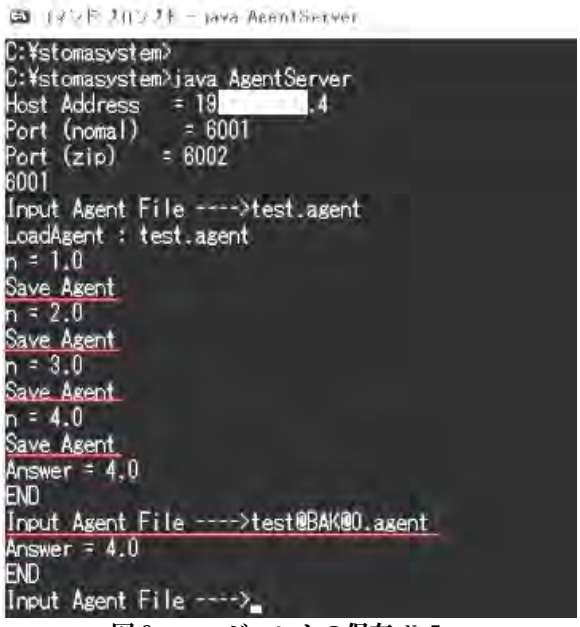

図 9　エージェントの保存:N=5

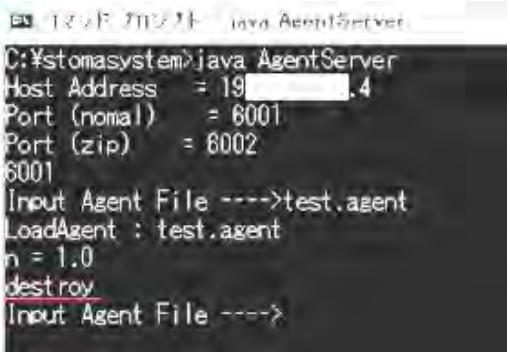

図 10　エージェントの消去:N=5

図 11　エージェントの移動:N=5

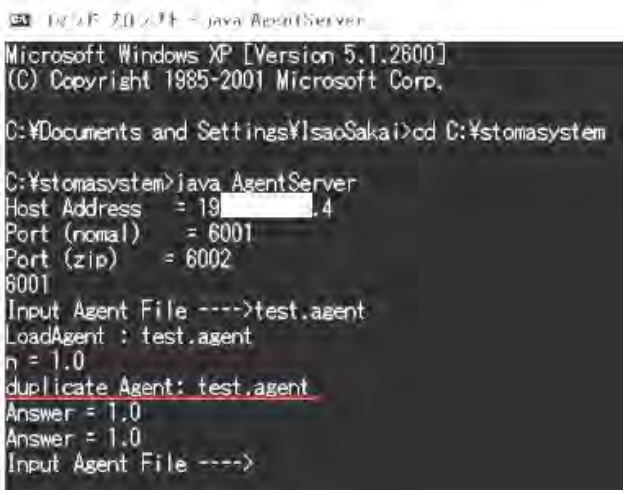

図 12　エージェントの複製:N=1

図 10 はエージェントを消去する動作を表していて，変数 n を 5 回インクリメントし出力するプログラムであるが，途中で処理を終了するプリミティブ（destroy()）を挿入したため，処理が中断してしまった状態を表している。

図 11 は，変数 n をインクリメントし，出力されるたびに，実行状態保持して，ローカルホスト自身への移動を繰り返した様子を表している。

図 12 はエージェントの複製を表していて，変数 n を 1 回インクリメントしたときの値と，その実行状態が複製され，実行しているエージェントの処理結果と，複製されたエージェントの処理結果の出力を確認することができた。

4．3　他のエージェントシステムとの比較

昨年度まで使用されていたモバイルエージェントシステム AgentSpace と AgentSphere を使用して，エージェントがリモートマシンに移動してから，ローカルマシンに戻るまでのオーバーヘッドを計測した。これを実行するためにユーザが記述するコードは AgentSphere では図 13，AgentSpace では図 14 である。

モバイルエージェントシステムにおけるコードの記述性（図 13，図 14）を比較すると，明らかに AgentSphere でコードを記述するほうが簡単に実装できる。このコードを 10 回実行したとき，AgentSpace では，平均 187msec 掛かるのに対し，AgentSphere では平均 168msec で実行することができた。つまり，1 回の送受信におけるオーバーヘッドは，AgentSpace は 93.5msec，AgentSphere は 84msec である。また，AgentSphere の変換されたソースコードに対するクラスファイルのサイズ（通信量）は AgentSpace で記述したソースコードに対するクラスファ

```
run(){
    stime = System.currentTimeMillis();
    migrate(remoteAddress);
    migrate(localAddress);
    ftime = System.currentTimeMillis();
    System.out.println(ftime - stime);
}
```

図13　AgentSphere で記述するコード

```
create(){
    stime = System.currentTimeMillis();
    dispatch(remoteAddress);
}
arrive(){
    if(remote){
        dispatch(localAddress);
    }
    if(local){
        ftime = System.currentTimeMillis();
        System.out.println(ftime - stime);
    }
}
```

図14　AgentSpace で記述するコード

イルのサイズ（通信量）の 2 倍ほど多くなってしまうが，このコードを実行するにあたって，AgentSphere は AgentSpace よりも 9.5msec 早くなる。しかし，この AgentSpace は，自律分散処理を行うための処理や，エージェント情報（エージェント識別子等）の登録，エージェント間通信の準備等のオーバーヘッドがあり，AgentSphere には，現段階では，それらの機能は実装されていない。そのため，AgentSpace よりも高速な移動を実現できたと考えるわけではない。しかし，AgentSpace は弱い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に対し，AgentSphere は強い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であるため，実行状態を保持した移動やバックアップ機能をユーザに低コストで提供することができるという利点はあると考える。

5．おわりに

強い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 AgentSphere を構築したことによって，実行状態を保持した移動やバックアップ機能をユーザに低コストで提供することができた。それ以外にも，エージェントの基本機能である，エージェントの生成，消去，複製等の動作も確認することができた。しかし，本研究での自律分散処理システムには，まだ対応できる段階にはなっていない。そこで，対応することが可能なモバイルエージェントシステムにするためには，エージェント間の通信や，エージェントの ID を決定する機能を実装する必要がある。また，エージェントがファイルやウインドウを使いたいときは，送信時やバックアップ時におけるファイル・ウィンドウの計算リソースの解放，受信時やバックアップファイルを実行させる時は，ファイル・ウィンドウの計算リソースの獲得を行えるようなコード変換が必要となる。本研究の自律分散処理システムに AgentSphere を導入することによって，マシンの性能値の急激な変化や分散処理に参加しているマシンの増減に伴ったタスクエージェントの再負荷分散や，管理エージェントがタスクエージェントのダウンを認識したとき，実行状態を保持したタスクエージェントの再生が実現することができるようになる。

参考文献

［1］ 服部文夫，坂間保雄，森原一郎「わかりやすいエージェント通信」オーム社出版 1998

［2］ 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 1998-12

［3］ IBM Tokyo Research Laboratory http://www.trl.ibm.com/aglets/index.htm

［4］ 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

［5］ 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://web.yl.is.s.u-tokyo.ac.jp/amo/JavaGo/doc/

［6］ 小川大介　他「モバイルエージェントを用いた自律分散処理システムの構築　－タスクエージェントの記述法とタスク分散の効率化－」2003 年度電気学会電子・情報・システム部門大会公演論文集 pp.1116-1120,2003-8

［7］ 坂井功　他「モバイルエージェントを用いた自律分散処理システムの構築　－自律分散処理をサポートする管理エージェントの作成－」2003 年度電気学会電子・情報・システム部門大会公演論文集 pp.1142-1146,2003-8