B-011

強マイグレーションモバイルエージェントシステム AgentSphere の開発
Development of the Strong Migration Mobile Agent System AgentSphere

赤井 雄樹† 若尾 一晃† 横内 貴† 甲斐 宗徳†
Yuki Akai Kazuaki Wakao Takashi Yokouchi Munenori Kai

1. はじめに

本研究で開発しているモバイルエージェントシステムは Java で実装されており、各エージェントがマシンやネットワークの負荷状況を考慮して移動を行うことで、自律的に分散処理を実現しようとするものである。エージェントのモビリティに強マイグレーションを採用し、強マイグレーションを実現するシステムとしては JavaGo[1] などの先例があるが、これらは Java 仮想マシン(JVM)を独自拡張することによって実現されている。本研究では、JVM を変更せずに強マイグレーションを実現するアプローチをとり、ユーザが記述した強マイグレーションのコードを通常の JVM で実行できるように、独自の自動コード変換器を開発してきた[2]。本論文では、エージェントが存在でき、強マイグレーションモバイルエージェントの活動を支援する空間 AgentSphere の設計と実装について報告する。

AgentSphere は、ネットワークに繋がれた複数のコンピュータを用いて効率的に仕事を分散することにより、1 台の性能では時間のかかる処理を、高速に遂行するシステムである。効率的な分散処理のために計算主体にモバイルエージェントを用い、エージェント自身の自律性による負荷分散を実現する。

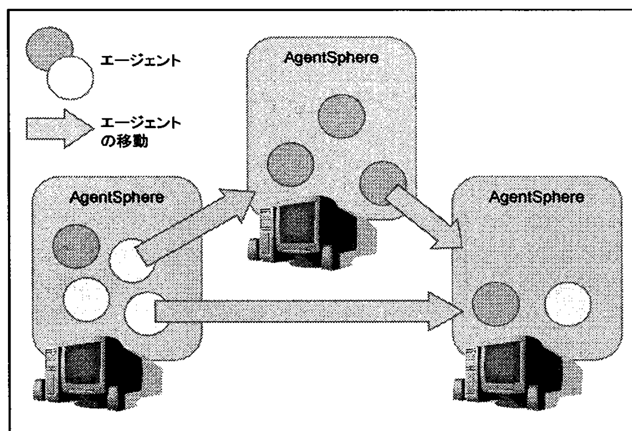

図 1-1：AgentSphere 概要図

図 1-1 は本研究で開発している AgentSphere の概要である。AgentSphere を起動しているマシン上で実行されているエージェントが、実行環境の負荷変動などを検出し、自律的にマシン間を移動をする様子を示している。

AgentSphere は、エージェントが動作するためのプラットフォームを提供するコアシステムと、その上で動作するエージェント群によるサブシステムとで構成される。この構成がとられているのは、サブシステムの更新を、コアシステムを停止させなくとも、動作しているエージェントを取り替えることで実現できるようにするためである。

またサブシステムの一例として、仮想ファイルシステムの実装を行った。仮想ファイルシステムは、エージェントがどこにいたとしても、必要なファイルに正しいアクセスを可能にするものである。このサブシステムも将来の拡張がし易いように、エージェントのみで実装されている。

2. エージェントの起動支援

2.1 ユーザインタフェース

ユーザが AgentSphere を利用するときのインタフェースとして AgentSphere 独自のシェル（SphereShell クラス）を開発した。この SphereShell はコマンドプロンプトからユーザの入力を受け取る。そして、入力を解釈して、ユーザからの命令（コマンド）に対応したエージェントや、システムから要求されたエージェントをロードする。そして、ロードしたエージェントを AgentLauncher（2.3 節で後述）に渡し、起動状態にする。

2.2 エージェントの識別

エージェント同士は連携することにより効率良く仕事ができる。そのためには、エージェント同士がお互いを識別できる必要がある。そこでエージェントの識別子として AgentIdentifier クラスを実装した。これはエージェントの一対一の連携のためだけではなく、1 つのジョブを処理する複数のエージェントのグループとしての識別にも利用できる。

2.3 エージェントの起動

AgentSphere 上でエージェントを活動させるために、エージェントに活動できる環境を提供しなければならない。その環境を提供するには、エージェントにスレッドを割り当てることと、エージェントが活動するのに必要なモジュールの起動が必要である。

AgentLauncher クラスは Thread クラスを継承した内部クラスを持ち、このスレッドをエージェント一つに対して一つ割り当てることでエージェントは活動できるようになる。また、エージェント間通信モジュールの起動も AgentLauncher で行う。ここでのエージェント間通信とは、エージェント同士が互いに連携をとるために行う通信のことである。AgentLauncher 周りの働きを図 2-1 に示す。

† 成蹊大学 Seikei University

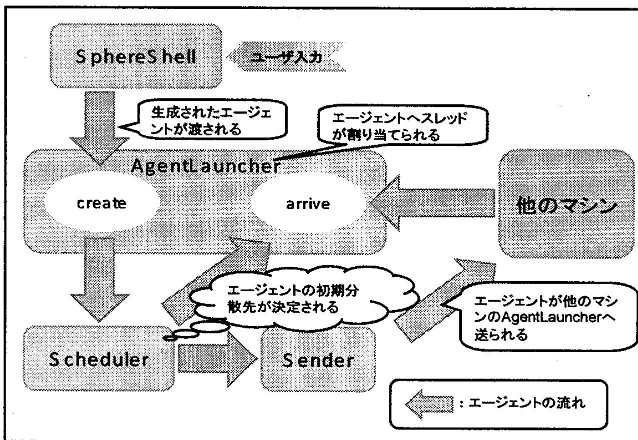

図 2-1：AgentLauncher 周りの流れ

SphereShell クラスでロードされたエージェントが AgentLauncher クラスに渡され、最初にエージェントの create メソッドを実行するためにスレッドが割り当てられる。create メソッドが終わると、常駐型のエージェントは処理が終了し、移動型のエージェントはスケジューラに渡される。移動型のエージェントにとって create メソッドは移動前にすべき処理をするメソッドであり、初期値の設定などが行われる。渡されたエージェントはスケジューラの判断でエージェントをロードしたマシンか他のマシンの AgentLauncher に渡される。そして、受け取った AgentLauncher で到着したエージェントの arrive メソッドを実行するようにスレッドを割り当てる。

また、エージェントのプログラム内部で migrate 命令があった場合、AgentLauncher 内で動いているそのエージェントを停止して、スケジューラに渡し、エージェントが移動する。エージェントが移動した後は、移動先で arrive メソッドが実行される。

3. クラスローダ

Java アプリケーションを実行するシステムにおいて、起動時に存在しないクラスを認識・実行させるためには、専用のクラスローダを実装する必要がある。AgentSphere の為の強マイグレーションエージェント用クラスローダとそれに伴う関連モジュールを設計するにあたり、以下の点を考慮しなければならない。

- 任意に指定されたクラスパスからエージェントのクラスをロードできる事
- 任意のパッケージ名指定を許容する事
- エージェントを構成する独自クラスが複数である事を許容する事

3.1 エージェントのマイナーバージョンアップへの対処

エージェントが記述された Java のクラスファイルが、部分的に更新（マイナーバージョンアップ）され、それがネットワークを通じて各マシンで更新されていくというように実装した。

JVM がクラスをロードするという処理は、Windows 環境で言えば、EXE ファイルが DLL を読み込む事と同じようなもので、この DLL を明示的に破棄及び書き換えることに相当する。

AgentSphere 独自のクラスローダを実装した上で、このクラスローダをエージェントのスレッドに割り当て、クラスデータを新規獲得する機構を組み込むことにより、既に読み込んでいるクラスデータを破棄することを可能とした。この更新処理は Scheduler 及び、AgentLauncher の連携によって実現される。

3.2 設計と実装

クラスローダは、エージェントの移動に伴う直列化とその復帰を完全にサポートしなければならない。そのため、一つのエージェントが複数のクラスによって構成出来る事、そしてパッケージを自由に記述できる事を許可する機構が必要になる。これを実現するために、図 3-1 に示すようなクラスのバイナリデータと正規クラス名を複数保持する「コンテナ型データ」を作成し、それを送受信・ロードするという形でクラスローダを設計した。

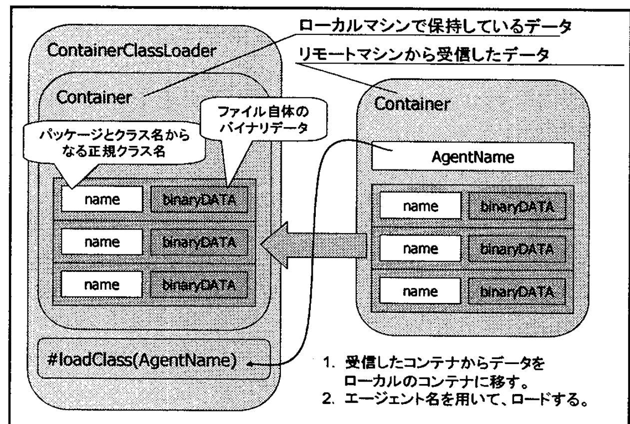

図 3-1：コンテナ型データの概要図

また、一般的にクラスローダを実装する場合は、ファイルへのパスを必要とし、クラスファイル自体へのアクセスが必要となるが、このファイルアクセスを不要にする実装を行った。これにより、エージェントの移動に伴うクラスデータの HDD 書き込みを発生させず、メモリ上で展開構築する事によりオーバーヘッドを減らす事が出来る。さらに、ファイルの実体を操作しないことで、ファイルロックの発生やそれに対する回避などの処置が不要となる。

4. AgentSphere ネットワーク構成モジュール

AgentSphere は、LAN 接続された複数マシンのおのおのに存在し、他の AgentSphere との間でエージェントを送受信するプラットフォームである。複数の AgentSphere がネットワークを構成するにあたり、各マシンの初期化処理が必須である。また、エージェントの移動やマシン情報の送信にかかわるネットワークモジュールの協調した操作が必要な場面があるため、これを統括管理するインタフェースが必要である。

4.1 SphereNetworkInfrastructure(SNI)

通信関連モジュールを統合し、各通信クラスの初期化、及びマシン固有データの生成やユーザ定義などを一元管理させた。またシングルトンパターンとファサードパターンによる実装で、各通信クラスが相互に同じインスタンスを呼べるようにした。これによって、エージェント送信を扱う他のモジュールに対しての通信関連機能の一元的な呼び出しが可能となる。そのためのクラスとしてSphereNetworkInfrastructure(SNI)を実装した(図 4-1)。

通信を利用する各クラスは SNI 以外のモジュールから初期化できず、それを利用するには、SNI を通して取得する必要がある。SNI が提供する通信関連機能を以下に述べる。

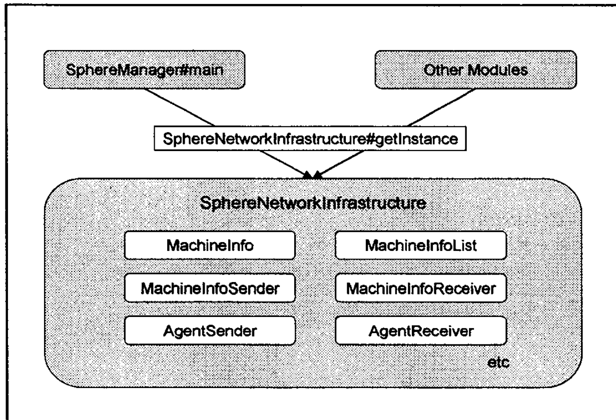

図 4-1：SNI 構成図

4.1.1 SNI の機能

SNI は、通信関連クラスの参照受け渡しの他に、ネットワーク構成のために必要な動作も担う。その機能としては、

・システム起動時のマシン情報の取得
・マシン情報送受信クラスの初期化
・マシン情報の交換によるネットワークへの参加と退場
・エージェント送受信クラスの初期化
・各モジュールへのマシン情報の受け渡し

などがある。

各マシンのネットワーク参加を実現するに当たり、互いの情報を交換しなければならない。その為に 1 台のマシンが SNI を起動した際に、該当マシンのプロパティデータとして MachineInfo を生成する。この MachineInfo には、識別用のデータ(MAC アドレス、IP アドレス、ユーザ情報)が含まれており、このデータをネットワーク中の他のマシンと交換することで互いに認識し合い AgentSphere ネットワークを形成する。

ネットワークは LAN を想定しており、AgentSphere 起動時にログイン情報のパケットをブロードキャストし、それに対して応答することでネットワークに参加している全てのマシンの MachineInfo が各マシンに保持される。

これらの送受信を行うクラス(MachineInfoSender、MachineInfoReceiver)を実装した。

エージェントの送受信クラスについて説明する。

AgentSender はエージェントの送信を行うもので、そのときに必要なデータはエージェント本体のオブジェクト実行状態と、エージェントを構成するクラスデータである。クラスデータは前述のコンテナ型データに格納し、これをエージェントの実行状態と供に送受信する。また、システム上でやり取りされるデータもエージェント互換形式であるため、この場合にはクラスデータは不要であり、実行状態だけを送受信するようにした。

AgentReceiver は受信データをオブジェクト実行状態とコンテナ型データに分解した後、コンテナ型データをクラスローダに渡し、クラスを認識させ、オブジェクト実行状態を AgentLauncher に渡す。

オブジェクト実行状態は、クラスの認識後、エージェントとしてスレッド上で動作することが可能となる。

4.2 通信関連クラス連携動作

システムとしての最低限の機能の他に、通信関連クラスが SNI の保持するデータを共通に利用する事で、以下の連携動作を可能とした。

・AgentSphere の安全な終了動作
・AgentSphere のダウン判定処理

4.2.1 安全な終了動作

AgentSphere を終了するにあたり、そのマシンが適切にネットワークから退場し、該当マシン中のエージェントが適切にネットワークの他のマシンに転送される必要がある。

終了動作の呼び出しはランチャモジュールが行う。具体的な処理は以下の通りである。

・ネットワークに対して、マシンダウン通知を行う。
・AgentReceiver を終了し、エージェントがマシンに来ない様にする。
・動作しているエージェントを他のマシンに移動させる。これは Scheduler が担当する。

4.2.2 ダウン判定処理

AgentSphere 上でエージェントが移動する大前提として、ダウンしているマシンに送ろうとしないこと、マシンがダウンしている事を出来るだけ早く察知すること、が必要となる。

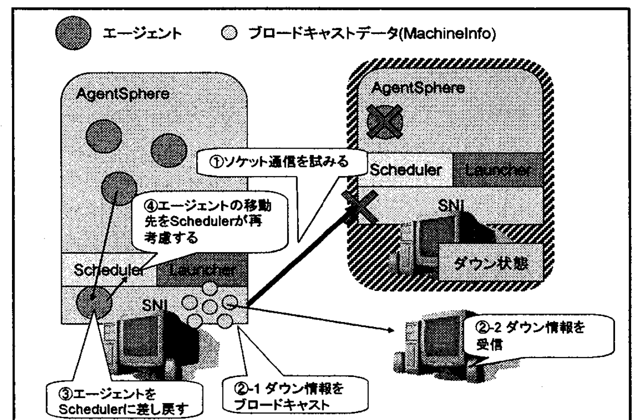

図 4-2：ダウン判定図

マシンのダウンが判明するタイミングは、「AgentSphere の安全な終了処理に際して、ネットワークに自分が終了したという情報を流す」場合と、「エージ

ェントの送信時にソケット通信が失敗した事から判明する」場合の二通りがある。後者の場合は AgentSender クラスが例外を投げる事から判定可能である(図 4-2①)。

一定時間エージェントの移動が起こらなかったマシンのダウン判定が遅延する恐れがあるが、全てのマシンを回る軽量なエージェント（性能値伝達用など）が居れば、退場を通達することが出来る。つまり、安全な終了動作によって、通知されるべき退場事実を、ダウンマシンを発見したマシンによって代わりに送信するのである(図 4-2②)。そしてダウンしたマシンに送信されようとしていたエージェントは、Scheduler によって、別の移動先を再考慮される(図 4-2③)。

5. ファイルシステム

5.1 ファイルシステムの必要性

エージェントが自身で JavaAPI を使い入出力を行う場合、入出力先(一般的にはファイル)の存在するマシン上に移動するか、入出力を専門に行うエージェントを作成しておいて、メッセージを送るしかない。そのため、前者の場合、処理の途中で入出力を行うことはいたずらにオーバーヘッドを増加させることに繋がる。後者はその事態を避けられるが、いずれも他のエージェント(特に他のユーザが起動したエージェント)との間で、ファイルロックの問題やファイル内容の整合性を保つなどの高度な連携が必要となる。

また、エージェントがファイルの状況を把握し、直接操作を行うこと自体に、後述するような問題がある。その問題に対して対処をせずにファイルアクセスが許容されている状態では、AgentSphere が中央集中型システムの欠点を内包することになる。

そのため、入出力を仮想化して、自律的な分散システムとしてあるべきファイルアクセスの手段を提供するサブシステムが必要となる。そのファイルシステムをエージェントを用いて実装した。

5.2 入出力に関する問題点

AgentSphere のエージェントにおいて、特別の配慮を行わずに、Java API を利用した入出力を行うと、大きく分けて以下の三つの弊害が生じる。

① セキュリティ問題

まず、セキュリティ問題について述べる。エージェントは滞在しているマシン上のファイルを自由に操作できる。エージェントは結局のところ AgentSphere というプロセス上のスレッドに過ぎないため、エージェントとマシンを操作しているユーザが実行した Java プログラムの間に、明確な差異は存在しない。そのためユーザのファイルの中身を勝手に読み込んだり、書き変えたりということが、エージェントを使って容易に行なえてしまう(図 5-1)。また、故意にセキュリティを犯そうとしなくても、たまたま自分が要求していたファイルと同じ名前のファイルの中身を間違えて書き換えてしまうということは十分考えられることである。

② ファイルによるエージェントの束縛問題

次に、ファイルによってエージェントが束縛される問題について述べる。JavaAPI ではファイルのリモートアクセスは提供されていないため(URI が与えられているならば、その限りではないが、①で取上げたセキュリティ問題がより悪い形で現れるだろう)、エージェントはファイルが存在しているマシン上に移動して操作しなければならない。そのため、エージェントがファイルによって移動を制限されるという事態が生じる。ファイルを操作している間に移動できないこともあるが、ファイルが存在するマシンから離れてしまうと、実際にファイルアクセスを行う際に移動に大きく時間を割くことになるためである。

図 5-2 はマシン M から離れた場所に性能的にも、混雑状況的にも、利用価値が高いマシンが存在しても、マシン M に必要なファイルが存在するために、エージェントが拘束されて移動できなくなる状況を示している。

これは結局、中央集中的なシステムと同じ状況を作ってしまうということになる。この問題を解消するためには、データがどこにあっても自在に扱えるようになっていなければならない。そして可能であるならば、ファイルを性能的に余裕のあるマシン上に移動させることも考える必要がある。

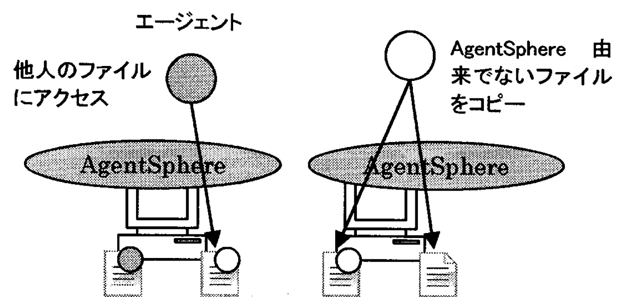

図 5-1：ファイル操作による
セキュリティの問題

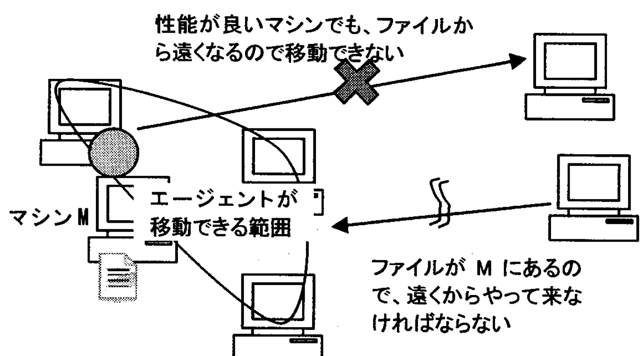

図 5-2：ファイルによるエージェントの束縛

③ 特定ファイルへのアクセス集中問題

最後に、特定ファイルへのアクセス集中問題について述べる。ファイルを所持しているマシンがダウンしてしまえば、ファイルの実体が損なわれていなくとも、ネットワーク上ではファイルが消滅したと考えることができる。このことも、中央集中的なシステムと同じ状況を作る。重要なファイルが 1 台のマシン上にしか存在しなければ、多くのエージェントがファイルを所持していたマシンと共倒れになってしまう。

また、2 つ目の問題点でも述べたように、エージェントはファイルを所持するマシンに群がってしまう可能性がある。そのため、貴重なファイルであればあるほど、ファイルを所持するマシンを含めた周辺のマシンが、一気に過負荷に陥ってダウンしてしまうという可能性がある。

図 5-3 は一つのファイルが原因で、周辺のマシンに被害が及ぶ状況を表している。これによりエージェント自体が失われることによる処理の中断や、連携している他のエージェントに被害が飛び火することによる、連鎖反応が起こる危険性がある。

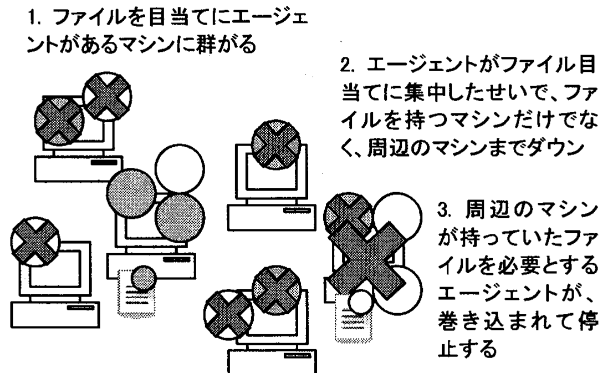

図 5-3：特定ファイルへのアクセス集中によるマシンダウン

以上が、エージェントが自身で入出力を行うことによる問題点であるが、本ファイルシステムはこれらを解決し、エージェントに対して透過的で使いやすいファイル操作機構を提供することを目的に設計・開発されている。

5.3 ファイルシステムの設計

本ファイルシステムは、システムを停止することなく交換可能で、拡張性の高いサブシステムとするため、その中心にエージェントを採用している。そのため、ファイルシステム自体が一種のエージェントプログラムであるという立場を取っている。

ファイルシステムは、図 5-4 に示すような 3 層から成る。

- インタフェース層
- トランザクション層
- プラットフォーム層

これら各層の詳細について以下で述べる。

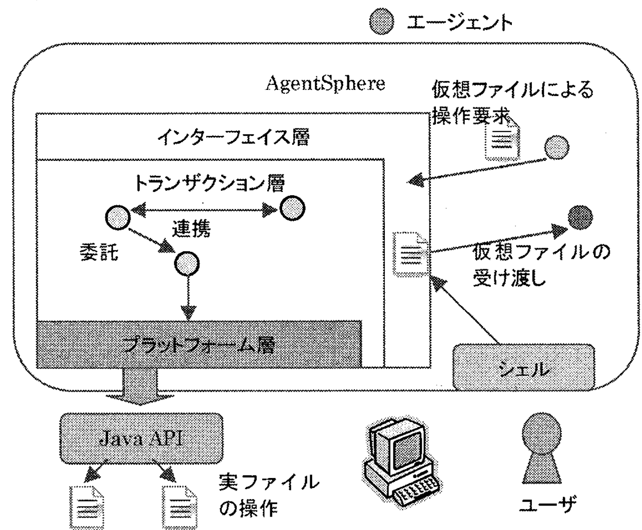

図 5-4：各層のつながり

(1) インタフェース層

インタフェース層は、ユーザやエージェントがファイルシステムを利用するための窓口となる層で、Java API によるファイルアクセス形式に類似した、仮想ファイルとそれを通じたファイルアクセスの手段を提供する。Java API と似させたのは、使用方法を Java API と共通にして、本ファイルシステム独自の使い方を新たに学習する必要を無くすためである。

提供できるファイルを制限することで、前述のセキュリティ問題が解決され、故意か否かに関わらず、他人のファイルにアクセスすることができなくなり、逆に自分のファイルはどこからでもアクセスできるようになる。

(2) トランザクション層

トランザクション層は、インタフェース層からの指示をうけて、ファイル操作を行うエージェント群で構成されている。この層は、ファイル操作のロジックを構築する部分であり、ファイルシステムの効率やファイルの整合性・可用性を向上させる役割を果たす。

ファイルの情報とは、例えばロック状況やファイルのパーミッションなどである。ファイル情報を全てのマシンが持つならば、その情報の整合性を保つのが困難となり、逆に特定のマシンが持つとダウンによる消失が懸念される。そのため、これもエージェントに管理させる。

(3) プラットフォーム層

プラットフォーム層は、トランザクション層に実ファイルを操作するための手段を提供するためのクラス群で、AgentSphere のシステムに組み込まれている。

この層の役割は、各マシン上の実ファイルの管理に限定されており、その機能も最小限に抑えられている。最終的にファイルにアクセスするには Java API を利用する必要がある。これはこのプラットフォーム層に限られており、ユーザエージェントがこれを直接利用することはない。

5.4 ファイルシステムの検証

ファイルシステムの動作確認として、図 5-5 のテストコードを作成した。このコードは、マシン A で仮想ファ

イルを作成してデータを書き込み、マシン B に移動したのち、その仮想ファイルからデータを読み込むものである。

この実行結果を図 5-6 に示す。

図 5-6 では、まずファイルシステムが起動時に認識したファイル群をリストアップし、シェルが入力待ちのプロンプト">"を出力している。ここで、起動時にテストコード上で指定されたファイルが認識されていないことに注意してほしい。

その後、マシン A では図 5-5 のテストコードを持つエージェントが起動され、処理を開始したことが出力され、マシン B では移動して来たエージェントが読み込んだデータを出力している。

```
public class TestAgent1 extends MigrateAgent{
  public void create(Object... args){
  SphereFile file = new SphereFile("virtualfile.dat");
  file.createNewFile();
  SphereFileOutputStream sfos = new
SphereFileOutputStream(file);
  sfos.write("write data".getBytes());

public void arrive(){
  SphereFile file = new SphereFile("virtualfile.dat");
  SphereFileInputStream sfis = new
SphereFileInputStream(file);
  byte[] buffer = new byte[10];
  sfis.read(buffer);
  System.out.println("read data:"+new String(buffer));
  sfis.close();
}
```

図 5-5：テストコード

マシン A

```
Detected files info
path:in kind:SPECIAL
path:out kind:SPECIAL
path:err kind:SPECIAL
path:/ kind:DIR
path:/system kind:DIR
> user.TestAgent1
result : load agent...
```

マシン B

```
Detected system files info
path:in kind:SPECIAL
path:out kind:SPECIAL
path:err kind:SPECIAL
path:/ kind:DIR
path:/system kind:DIR
>
read data:write data
```

図 5-6：テスト結果

6. おわりに

本論文では、強マイグレーションモバイルエージェントを自律的な分散処理システムに適用するために、エージェントの活動を支援する、AgentSphere のコアシステムの実装について報告した。AgentSphere が動作するマシンのネットワークをエージェントが自律的に移動しながら処理を進めるために、実行可能なクラスをダイナミックにロードする専用のクラスローダを実装した。また AgentSphere 間をエージェントが移動するための通信機能に関わるコアシステムについての実装も行った。

さらに自律的な分散処理においてエージェントが安全にファイルアクセスすることができる仮想ファイルシステムをエージェント技術により実装した。このファイルシステムは自律的な分散処理を支援するサブシステムとしての役割を持つ。このファイルシステムを利用することで、エージェントはどこの AgentSphere にいてもアクセス権限を持つファイルに対するアクセスを行うことができる。加えて、ファイルシステムはサブシステムとしての拡張や改良が必要な場合でも、コアシステムと分離したことで、サブシステムの変更のみで対応することができる。

今回実装したコアシステム、サブシステムともに、個々の機能を果たすのに十分に限定的な小さなモジュールやエージェントの集合体として実装されている。

コアシステムのみではエージェントを実際に利用したプログラムを作成するには不便である。それを補う機能を、コアシステムの機能を利用して動作するサブシステムで実現する。

システム全体の機能拡張を行う際に、エージェントを記述し、サブシステムとして実現させるという方法を採用した。コアシステムが提供する機能をバージョンアップした場合でも、その上のサブシステムは元のままで利用可能であり、また、サブシステムへ機能を追加する場合には、コアシステムを変更すること無く拡張やチューニングが可能である。

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.2009 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