B-010

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

Implementation of Self-backup Mechanism and Inter-agent Communication for Strong Migration Mobile Agents

加藤 史彬† 近藤 敬宏† 甲斐 宗徳†
Fumiaki Katoh Takahiro Kondoh Munenori Kai

1. はじめに

モバイルエージェントシステムは大きく分けて弱マイグレーションシステムと強マイグレーションシステムに分けられる。既存の研究として、AgentSpace（佐藤一郎氏）[1]や Aglets（日本 IBM 社）[2]が挙げられるが、これらは弱マイグレーションシステムのため、移動後に移動前の続きから処理を開始するのに十分な情報を持って移動することが出来ない。移動後に移動前の続きからの処理を開始出来る強マイグレーションシステムの研究として JavaGO（東京大学米澤研究室）[3]や MOBA（首藤一幸氏）[4]があるが、特定の位置でしか実行状態の保存や復旧ができないといった問題や、ランタイムシステムの拡張やネイティブメソッドの追加を行っていることにより、移植性が失われてしまうといった問題点を持っている。

本研究で開発しているモバイルエージェントシステム AgentSphere は、自律的に振舞うモバイルエージェントに分割された処理を割り当て、AgentSphere のネットワークに参加しているマシン上で処理を行わせる。さらに、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いながら、自律的な分散を実現するものである。

このシステムでは、Java Virtual Machine（以降 JVM）を変更することなく強マイグレーションが実現できる。Java の直列化機能は、ヒープ領域内の情報を保存することが可能であるが、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提供していない。そこで我々は、それらの情報を含めた実行状態を取得し、移動後に移動前に中断した処理を再開できるようにするための自動コード変換手法を提案した[5]。これにより通常の JVM でエージェントの強マイグレーションを実現している。

ただし、我々の提案した手法でも取得・復元できるローカル変数はプリミティブ型と String 型及びそれらの配列に限定されており、参照型の取得・復元は出来ていなかった。

本論文では、まず従来困難であった参照型のオブジェクトを含んだエージェントが移動できるように migrate 命令を強化したことについて報告する。次に、エージェントの自己バックアップ機能を提案する。そして最後に AgentSphere においてエージェント初期分散を行うスケジューラと、エージェント間通信を行うためのモジュールの実装について述べる。

†成蹊大学 Seikei University

2. migrate 命令に対する強化

2.1 従来の migrate 命令の弱点について

我々が提案する強マイグレーションモバイルエージェントでは、ユーザがエージェントに移動命令である migrate メソッドを記述するだけで、処理の中断、移動、再開が可能となる。処理の中断時には、移動後の再開に必要な実行時データを取得する。従来の migrate メソッドでは、プリミティブ型、String 型そしてそれらの配列型の取得・復元は可能であった。しかし参照型の取得・復元が実現されていなかった。そこで本論文では参照型オブジェクトを取得・復元できるように改良する。

2.2 ローカル変数の取得・復元方法

Java を用いてモバイルエージェントを実装する際、直列化機能を用いることにより、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して移動することが可能である。しかし直列化機能は、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を提供していない。従って移動後に実行状態を復元することができない。強マイグレーションのエージェントには、これらの情報は不可欠である。そこで我々は既にこの問題を解決する方法を提案した[5]。ユーザが記述した migrate 命令は、スタック領域を含めた実行状態を取得し、そして移動し、移動先で実行状態を復元する、という一連のコードに変換される。スタック領域内のローカル変数の情報は、JVM に標準で実装されている Java Platform Debugger Architecture（以降 JPDA）を利用して取得する。具体的には、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、スタック内のローカル変数の値を取得する。しかし、このローカル変数の値は JPDA 特有のクラス(Value 型)となっているため、直列化出来ず、マシン外部へ持ち出し出来ない。そこで、この値を直列化可能な変数に変換し、ヒープ領域内に格納する。そうすることで、JVM を変更することなく、Java に備わっている直列化機能のみを用いて実行コード領域・ヒープ領域そしてスタック領域を保存し、別マシンに送信することができるようになる。図 2-1 にローカル変数の取得のプロセスについて示す。スタックフレームはその中に複数のローカル変数を持っている。

そのスタックフレームから各ローカル変数を取得したものが Value 型のデータである。

ここで、スタックフレームに参照型のオブジェクトが格納されている場合、参照型はプリミティブ型のようにそのままでは直列化できない。そこで参照型の内部を走査し、その中に含まれている変数を１つ１つ調べる。見つかった変数はすべて変数リストに登録するが、内部に

参照型が見つかった場合には、その中をさらに再帰的に走査する。

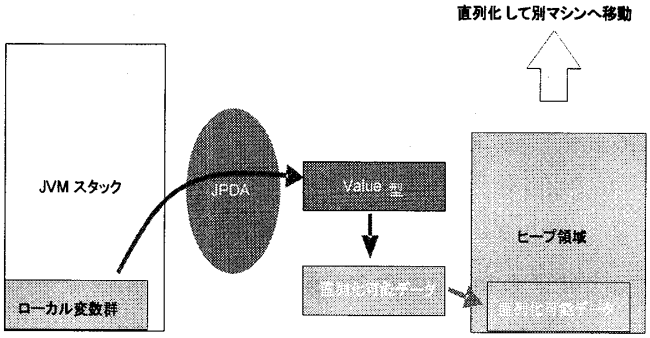

図 2-1　ローカル変数の取得のプロセス

そしてリストに登録された変数がすべて直列化可能な変数だけになったら、それらの変数をヒープ領域に格納して直列化する。このようにして参照型も直列化できるように改良した。エージェントの移動先で参照型を復元する場合には、取得時に作成した変数リストに基づいて、参照型の階層構造に合わせて再帰的に復元することができる。

3. エージェントの自己バックアップ機構の実装

3.1 自己バックアップ機構の概要

通常、分散処理では高信頼性や耐障害性が期待される。しかし分散させて結果を統合するだけでは問題が発生する。それは、初期分散で最適な負荷分散を行っていたとしても、その後のマシン性能が変化して初期分散が最適であるとは限らない。負荷が掛かり続けたマシンが最悪ダウンしてしまい、それまでそのマシンで行っていた計算処理が無駄になることも起こり得る。そこで処理をある程度行ったときにそれまでの実行データも含めてエージェント自身のバックアップを作成し、それを他のマシンへ送り込んでおく。負荷が大きくなったマシンでエージェントが処理し続けるのが困難になった時や、万が一、元のエージェントが失われた時には、Scheduler が別マシンに送り込まれていたバックアップエージェントにバックアップ時点からの実行を再開させることが出来、高信頼性や耐障害性を向上させることが出来る。今回はバックアップを取得するために呼び出される backup メソッドの実装と、その backup メソッドをユーザコードに自動挿入する手法を提案する。

3.2 backup メソッドについて

backup メソッドは呼び出された時点での実行時データをバックアップし、別マシンまたは自マシンへ送り込む。backup メソッドも後での実行再開のために、実行時データの取得を行わなければならない。また実行再開をするために、migrate メソッドと同様に図 4-1 に示すソースコード変換を行う。migrate メソッドと異なるのは、migrate メソッドでは移動直後にそのエージェントの実行が再開されるが、backup メソッドでは、そのエージェントは Scheduler から呼び起こされて処理を再開するという点である。エージェントの中で backup メソッドは複数回呼ばれることが出来、その都度、エージェントのバックアップが生成される。生成されたエージェントのバックアップが直前と同じ AgentSphere に保管されるときには上書きされ、最新のバックアップが残る。現段階では Scheduler に、backup メソッドが記述されたエージェントを管理する機能が用意されていないため、3.3 節の評価では、手動で再開させるコマンドを利用してバックアップエージェントを再生させる。

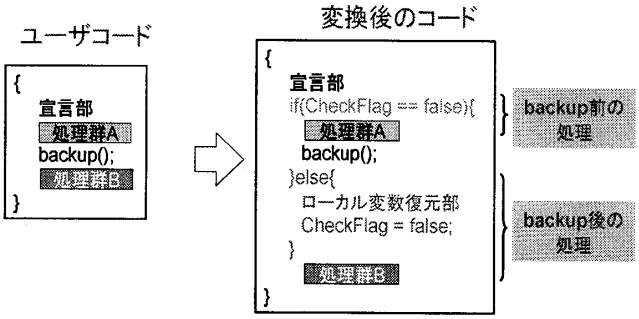

図 3-1　backup メソッドのソースコード変換

3.3 クラス変数の取得・復元と backup メソッドについての動作確認

ここでは backup メソッドの動作確認のため、backup メソッドを手動で入れた図 3-2 のコードを準備した。図 3-2 のコードはクラスを使ったエージェントで、クラスの内部にも別のクラスを持つ。backup メソッドで実行時データをバックアップして別マシンへ送り込み、行き先マシン上でエージェントを再開してローカル変数の取得・復元が出来ているかを確認した。

このコードは、クラス s1 のフィールド i と j 及びクラス s1 のフィールドであるクラス s2 のフィールド d の値を s1 のメソッド valincr()で増加させるプログラムで、増加させるたびに実行時データをバックアップし、別マシンへ送り込む。複数回実行された場合は、新しいバックアップで古いバックアップを上書きする。

図 3-2 は、backup メソッドが挿入された変換前のコードで、ユーザが記述するのはこれだけである。

この後、図 3-2 のコードはソースコード変換器に掛けられる。

変換後のコードを図 3-3 に示す。移動命令（ここでは backup）に関する変換を斜体で示し、さらに backup 用固有の変換を下線部で示してある。

これらのコードを AgentSphere を動作させているマシン A で実行し、i が 6 になったところで AgentSphere を停止し、エージェントを終了させた。そしてここではバックアップされたエージェントが起動後に処理を再開出来るのかを確認するため、マシン B で AgentSphere のバックアップの再生コマンドである bc コマンドを利用し、マシン A での続きから実行されるかについて調べた。マシン B で再開したところ、ループインデックスである i の値及びオブジェクト s1 の値が引き継がれており、マシン A での続きがマシン B で行われ、クラス変数について取得・復元できていることが確認できた。

```
public class BackupSampleTest{
 private String IPaddress = "192.168.1.108";
 private static AgentCoordinator ac;
 public void create (){
  SampleClass s1 = new SampleClass();
  ac = getAgentCoordinator();
  for (int i = 0;  i < 10; i++){
   System.out.println ("backup前の値s1.i:" + s1.i
          + " s1.j:" + s1.getj() + "  s1.s2.d:" + s1.s2.d);
   s1.valincr();
   ac.backup(IPaddress, 0);
   Thread.sleep(2000);
  }
  System.out.println("backup後の値s1.i:" + s1.i
          + " s1.j:" + s1.getj() + " s1.s2.d:" + s1.s2.d);
}
```

```
public class SampleClass {
 int i;
 private double j;
 SampleClass2 s2;
 SampleClass(){
  i=1;  j=3.0;  s2=new SampleClass2();
 }
double getj(){ return j; }

 public void valincr() {
 i++;  j++;  s2.d++;
 }
}
```

```
public class SampleClass2 {
 double d;

 SampleClass2(){
  d = 13;
 }
}
```

図 3-2　backup メソッドを挿入したコード

```
public class Translated_BackupSampleTest{
 private String IPaddress = " 192.168.1.108 ";
 private static AgentCoordinator ac;
 public void create (){
  SampleClass s1 = new SampleClass();
  ac = getAgentCoordinator();
  if(ac.checkflag(0)==false){
   ac = getAgentCoordinator();
  }else{
   s1.buildClassValue(ac,0);
  }
  for (int i = 0;ac.checkflag(0)==true ||  i < 10; i++){
   if(ac.checkflag(0)== false){
    System.out.println (" backup前の値s1.i:"
       + s1.i + " s1.j:" + s1.getj() + "  s1.s2.d:" + s1.s2.d);
    s1.valincr();
    ac.backup(IPaddress, 0);
   }else{
    s1.buildClassValue(ac,0);
    i = ac.getIntegerValue("i",0);
    ac.setFlag(0);
   }
   Thread.sleep(2000);
  }
  System.out.println (" backup後の値s1.i:" :"
       + s1.i + " s1.j:" + s1.getj() + " s1.s2.d:" + s1.s2.d);
 }
}
```

```
public class SampleClass {
 int i;
 private double j;
 SampleClass2 s2;
 SampleClass(){
  i=1;  j=3.0;  s2=new SampleClass2();
 }

 /*クラス変数用の自動挿入メソッド */
 public void buildClassValue(AgentCoordinator ac, int rank) {
  i=(Integer)ac.getClassValue("s1.i", 0);
  j=(Double)ac.getClassValue("s1.j", 0);
  s2.buildClassValue(ac,0);
 }

 double getj(){ return j; }

 public void valincr() {
 i++;  j++;  s2.d++;
 }
}
```

```
public class SampleClass2 {
 double d;

 SampleClass2(){
  d = 13;
 }
/*クラス変数用の自動挿入メソッド */

 public void buildClassValue(AgentCoordinator ac, int rank) {
  d = (Double)ac.getClassValue("s1.s2.d",0);
 }
}
```

図 3-3　図 3-2 の自動変換後のコード

4. backup メソッドの自動挿入手法

4.1 backup メソッドの自動挿入手法の提案

backup メソッドはユーザが任意に挿入出来る。しかしシステムが自動的にユーザコードの適切な位置に backup メソッドを挿入する機能があれば、ユーザはその自動挿入機能を利用するだけで、耐障害性を持つエージェントを作ることができる。そこで、ここでは backup メソッドの自動挿入手法を提案する。

今回提案する backup メソッドの自動挿入手法は、最適な挿入位置を探すために静的分析と動的分析を行う。以下では、それぞれの分析について述べる。

4.2 静的分析

静的分析では、コードを走査し、backup メソッドの予約位置として reserved_backup メソッドを挿入する。reserved_backup メソッドは実際にバックアップを取るメソッドではなく、次の動的分析時に利用されるメソッドである。

エージェントをバックアップするオーバヘッドを出来るだけ少なくするには、バックアップの回数を減らし、また 1 回のバックアップに関わる実行時データを出来るだけ少なくするとよい。スコープが深ければ深いほど、スタックフレーム数が増え、取得する情報量が大きくなり、それらをバックアップして持ち運ぶと、オーバヘッドが大きくなる。そこでスコープから出た位置を reserved_backup メソッドの挿入位置にすることが望ましいと考えた。各 reserved_backup メソッドは自分と他を見分けるため引数としてシーケンシャルに整数値を持っている。

またメソッド呼び出しの直後にも reserved_backup メソッドを挿入する。なぜなら、メソッドが呼び出されると新しいスコープに入り、そのメソッドが終了して戻ってくるときに、必ず 1 つのスコープを出るからである。

もしここで挿入された reserved_backup メソッドが多かったとしても、それらは次の動的分析時に絞り込まれる。

4.3 動的分析

動的分析では、静的分析が終わったコードを実際に実行し、各 reserved_backup メソッドの位置でのエージェント開始からの経過時間とその時点までの変数アクセスの通算回数のデータを取る。

変数のアクセスについては、各変数に対し、実際にエージェントを動作させ JDI（Java Debugger Interface）を利用しどれだけアクセスがあったかを見ることが出来る。

reserved_backup メソッドはあくまで backup メソッドの予約位置であるので、動的分析後に reserved_backup メソッドの絞込みを行う。

今回は最初の試みとして、各 reserved_backup 間の時間で判断するという条件を考えた。その条件とは reserved_backup が短時間に複数回実行されていた場合は不要と判断するものである。動的分析中に通った reserved_backup メソッドの位置で、その番号と通った時間を記録して行く。そしてそのヒストリから適切なタイミングで実行される reserved_backup メソッドを選び出すことが出来る。

従って、reserved_backup メソッドをどのくらいの間隔で残すべきかによって、生成されるバックアップエージェントの数は変わってくるが、現在はユーザがその間隔をパラメータとして指定している。

4.4 backup メソッドの自動挿入手法の評価

backup メソッドの自動挿入の結果が適切かどうかを確認するため、巡回セールスマン問題（以後 TSP）の全探索解法コードに対し backup メソッドの自動挿入を行った。その結果、静的分析によって図 4-1 に示されているように reserved_backup メソッドが挿入された。

続いて動的分析を行った結果、番号が 2 ～ 8 の reserved_backup メソッドは削除され、1 の reserved_backup メソッドのみが残った。3～8 の位置の reserved_backup は再帰メソッドである find の再帰呼び出しが終了する際に短時間で実行される位置であり、バックアップの位置としては適さないと考えられる。しかし 1 の reserved_backup は、再帰が終了し、戻ってきてスタックフレームが減っている場所であり、大量の処理が終わったところである。これは論理的に見ても TSP で 1 つの解候補が得られた直後にバックアップ作業を行うことに相当するので、適切なバックアップであると考えられる。

```
public void create() {
 start_res();
 //Map生成
 for(i=0;i<NUM;i++){
  find(MAP[0][0],1,i);
  reserved_backup(1);
 }
 reserved_backup(2);
}

void find(int sum,int depth,int place){
 if(sum > point){ return;}
 reserved_backup(3);
 check[place] = true;
 for(int i = 1 ; i < NUM ; i++){
  if(check[i] == false){
 find(sum+MAP[place][i],depth+1,i);
 reserved_backup(4);
  }
  reserved_backup(5);
 }
 reserved_backup(6);
 if(depth == NUM - 1){
  sum += MAP[place][0];
  if(sum < point){
   point = sum;
  }
  reserved_backup(7);
 }
 reserved_backup(8);
 check[place] = false;
}
```

図 4-1　静的分析後のコード

5. スケジューラの開発

5.1 スケジューラの役割

スケジューラは、エージェントの移動先を決定するパッケージである。AgentSphere 終了時のエージェントの放出や、緊急時にエージェントの移動を行わせるのもスケジューラの役割である。

分散処理を行う際、特定の PC に大量のエージェントを集めてしまったり、資源に余裕がない PC にエージェントを送ってしまったりすると、局所的に負荷が大きくなり効率が落ちる。この状況を防ぐために、スケジューラはエージェントが移動する宛先の選定を行わなければならない。適切に選定を行うために、AgentSphere 上にいるタ

スクエージェントの数や PC の性能を判断情報として考慮する。スケジューラは、AgentSphere 全体の負荷状況を安定させ効率化を図る。

5.2 パッケージ Scheduler の設計

エージェントが生成され活動を開始しようとするときに、スケジューラはエージェントに対して行き先を選定するものである。

AgentSphere では、参加している全マシンの情報を格納したマシンリストを持っている。また、マシンリストに含まれる各マシンの性能値を更新するエージェントを活動させることができる。これにより参加している全マシンの性能値は、負荷の変化に対応した出来るだけ最新の値となる。スケジューラは、この性能値に基づいて行き先を選定する。

稼働中のエージェントが再分散されるのは、そのエージェントが migrate 命令や backup 命令によって他のマシンに移動するときである。このとき migrate 命令または backup 命令が行き先を明示していない、あるいは行き先をスケジューラに変更されても構わないならば、スケジューラは、マシンリスト中の各マシンの最新の性能値を参照して、宛先を決定する。

ある AgentSphere で負荷が高くなって性能値が他のマシンの性能値の平均値よりも極度に低下した場合には、スケジューラはそこで活動中のエージェントの一部に移動を指示するメッセージを送る。一方、AgentSphere 自体が終了しようとする場合には、スケジューラはそこで活動しているすべてのエージェントに移動を指示するメッセージを送る。移動を指示されたエージェントは、スケジューラに宛先を指定されて移動する。

スケジューラがエージェントの移動先を決定するために参照する情報は、各マシンの最新の性能値の他に、各マシンで活動中のエージェントの数や各エージェントの残りの処理量も考えられるが、それらの情報を得ることは必ずしも簡単とは限らない。そこで現在のスケジューラは、エージェントの処理量はどれも大体同じであることを仮定し、マシンの性能値と活動中のエージェントの個数の関係から移動先を決定する。

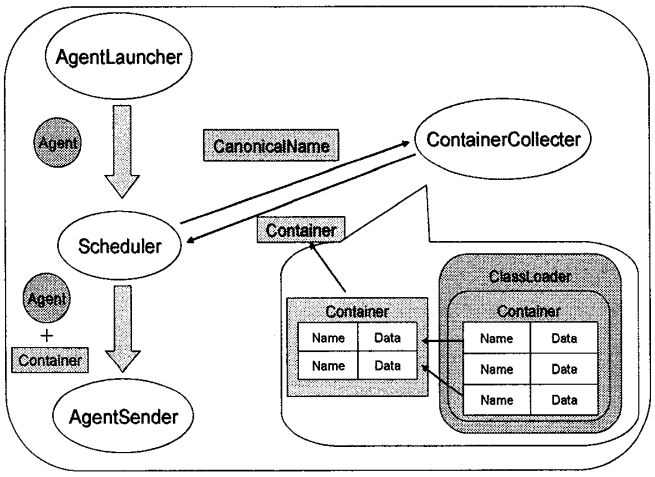

図 5-1 Scheduler の流れ図

図 5-1 は、Scheduler とそれに関係するモジュールを示している。他マシンにエージェントを送る場合は、そのエージェントが必要とするクラスデータも一緒に送らなければならない。Scheduler は ContainerCollecter にエージェントが必要とするクラスの正規名を伝えてクラスデータを取得する。ContainerCollecter は、エージェントの正規名を用いて、必要なデータをもつクラスローダのコンテナから抜き出して、必要最小限なサイズのコンテナにまとめる機能を持つ。

Scheduler は必要なクラスデータが揃ったエージェントを AgentSender に渡し、AgentSender が宛先の AgentSphere にエージェントを送る。

6. エージェント間通信の開発

6.1 エージェント間通信機能の概要

エージェント間通信は、AgentSphere で活動中のエージェント間でメッセージの送受信を可能にする。受信側エージェントが同じ AgentSphere にいる場合はそのままメッセージを渡せるが、受信側エージェントが他のマシンにいる場合には、通信方法によって問題が生じる。例えば、送信側エージェントが自分で受信側エージェントを探しながら AgentSphere を移動してメッセージを渡す方法が考えられるが、送信側エージェントも処理を中断して移動しなければならないので、オーバヘッドが大きい。メッセージを送る場合には、送信側エージェントの処理が止まることなくメッセージを送れることが必要である。そこで、メッセージ配達専用のエージェント Messenger を生成して届けさせる機構を実装した。

6.2 エージェント間通信機能の設計

受信側エージェントも、そのコード内に受信処理を記述すると処理が中断することになるので望ましくない。そこでメッセージが届くと受信側エージェントは別スレッドで受信処理を行う。

届いたメッセージは、各エージェントのキューに保持される。従ってそのエージェントが移動したとしても、届いたメッセージに対応する処理を移動先で行うことができる。

送信側エージェントは、受信側エージェントがどこにいるかを知らなくても、メッセージ送信を行うことができる。

通信命令は以下の 3 種類を準備した。各命令の違いを図 6-1 に示している。

- send ･･･メッセージ送信後、送信側エージェントの実行はブロックされ、返信を要求しない。
- call ･･･メッセージ送信後、送信側エージェントの実行をブロックし、返信を要求して返信が来るまで待つ。
- future･･･メッセージ送信後、送信側エージェントの実行はブロックされないが、返信を要求する。あとで返信要求を行うことにより返信されてきた値が受け取れる。そのときに返信が届いていない場合は処理をブロックして返信がくるのを待つ。

ここで、返信に用いられる値は object 型であり、任意の値を受け取ることができる。

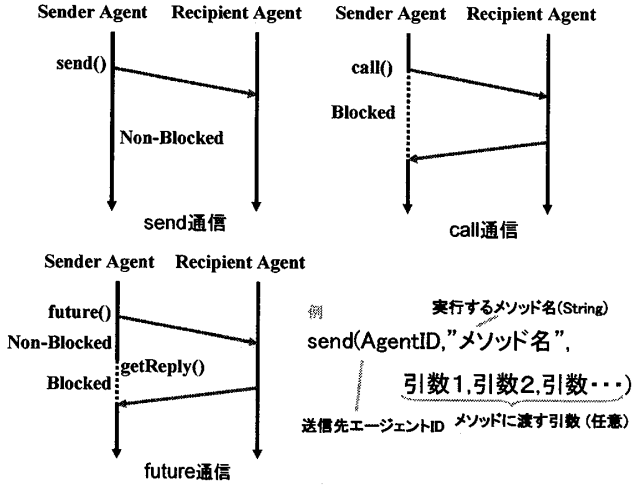

図 6-1 送信方法の種類と例

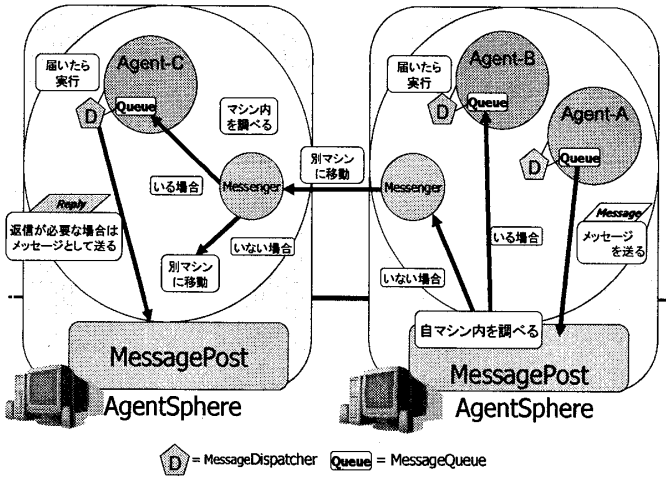

図 6-2 Message の流れ図

エージェント間通信の具体例を図 6-2 に示す。エージェントの生成は AgentSphere が管理するものである。従って、あるエージェントが送信命令を実行すると、そのメッセージはまず MessagePost に渡される。MessagePost はその AgentSphere で活動中のエージェントリストを参照することが出来、受信側エージェントが同じ AgentSphere 内にいることが分かれば、受信側エージェントのメッセージキューにメッセージを入れる。受信側エージェントが見つからなければ、Messenger を作成する。

Messenger は極力軽く最低限の機能しか持たないエージェントであり、できる限り移動のオーバヘッドを抑えている。Messenger はタイムアウトの機能も持っており、相手が見つからないまま、一定の周回数を超えると自動消滅するようになっている。ユーザはこの周回数を自由に設定できる。

MessageDispatcher はエージェントに１つ作られるスレッドで、メッセージの受信検出や実行を行う。従って、受信側エージェントには、受信用の命令を記述する必要はない。

7. おわりに

本研究では、まず強マイグレーションの移動命令 migrate の機能を強化し、エージェントの自己バックアップ機構を実装した。改良されたコード変換により、エージェントは参照型オブジェクトを持って移動することが可能となった。

次に、エージェントの移動先を決定するスケジューラとそれを支援するモジュール群を作成した。加えてエージェント間通信の基本的な送受信動作を実装した。これでユーザはどのようなコードでもエージェントとして記述出来るようになり、様々な処理を行うプログラムを AgentSphere 上で実行できる。

特にエージェントの自己バックアップ機構が実装され、単純な方法であるが backup 命令の自動挿入の方法を提供したことにより、エージェントシステムとしての信頼性・耐障害性を向上させたと考えている。

今後、エージェントの移動先を決めるスケジューラのアルゴリズムの改善や、AgentSphere の安全な終了処理に伴うエージェントの移動方法の選択を行い、動的に変動する負荷を効率的にスケジューリングできるシステムにしていく研究が必要である。

参考文献

[1]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
[2]日本 IBM 東京基礎研究所： http://www.trl.ibm.com/aglets/ , Jul.2009 参照可
[3]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
[4]首藤一幸： http://www.shudo.net/moba/ ,Jul.2009 参照可
[5]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