M-051

コードキャッシングによるモバイルエージェントの高速な移動手法の提案
Proposal of Fast Agent Migration using Code Caching

東野 正幸† 本村 真一†
Masayuki HIGASHINO Shinichi MOTOMURA
川村 尚生† 笹間 俊彦† 菅原 一孔†
Takao KAWAMURA Toshihiko SASAMA Kazunori SUGAHARA

1 はじめに

エージェントとは，人間の代理となって自律的に処理を行うプログラムであり，モバイルエージェントとは，ネットワークに接続されたコンピュータ間を移動可能なエージェントである．

モバイルエージェントが，コンピュータ間を移動する際は，移動元のコンピュータで行っていた処理を，移動先のコンピュータで継続して行えるように，処理に必要なプログラムコードを，移動先のコンピュータへ転送する必要がある．

モバイルエージェントの行う処理が複雑になるほど，モバイルエージェントが必要とするプログラムコード数は増大するが，これは，エージェントの移動時に転送しなければならないプログラムコードの量も増大することを意味する．

このため，モバイルエージェントが行う処理の複雑さに応じて，モバイルエージェントの移動にかかるオーバーヘッドが大きくなってしまう特徴がある．

多数のモバイルエージェントが頻繁に移動するようなアプリケーションにおいては，アプリケーションのシステム内で活動している各々のモバイルエージェントの移動にかかるオーバーヘッドが，アプリケーションのシステム全体のパフォーマンス低下につながるため，モバイルエージェントの移動にかかるオーバーヘッドを小さくすることは非常に重要な課題である．

そこで本論文では，キャッシング（Caching）技術に着目し，各々のモバイルエージェントが必要なプログラムコードを，ネットワークに接続されたコンピュータに設置したキャッシュ装置へキャッシングすることで，モバイルエージェントの高速な移動を実現する手法を提案する．

提案手法では，モバイルエージェントがコンピュータ間を移動する際に，移動元と移動先のコンピュータへプログラムコードをキャッシングする．これにより，モバイルエージェントが過去に訪れたことのあるコンピュータへの移動の際にキャッシュを利用することができるため，プログラムコードを転送する必要が無くなり，モバイルエージェントが高速に移動可能となる．

また，提案手法を実装し，評価した結果，モバイルエージェントの移動を高速化できることを確認したとともに，実際的なモバイルエージェントを用いたアプリケーションでのパフォーマンスを改善できることを確認した．

†鳥取大学大学院 工学研究科 情報エレクトロニクス専攻

本論文では，まず 2 章で既存のモバイルエージェントの移動手法の特徴と課題について述べる．3 章と 4 章では，提案手法の詳細について述べる．5 章では，提案手法の実装について述べる．そして 6 章では評価を行い，最後の 7 章では本論文のまとめを述べる．

2 プログラムコードの移動手法

モバイルエージェントがネットワークに接続されたコンピュータ間を移動する際に，移動元で行っていた処理を移動先で継続して行うためには，モバイルエージェントが処理に必要なプログラムコードを，移動先のコンピュータに転送する必要がある．一般的にプログラムコードの転送手法は，以下の 3 種類に分類できる．

この章では，それぞれの概要と課題を述べる．

2.1 コンピュータに予め配置しておく手法

モバイルエージェントが移動する可能性のある全てのコンピュータへ，プログラムコードを予め配置しておく手法である．

この手法は，モバイルエージェントが移動する際にプログラムコードを転送する必要がないため，モバイルエージェントの移動は非常に高速である．

しかし，モバイルエージェントのプログラムコードの変更が必要になった場合などに，全てのコンピュータに配置されているプログラムコードを更新する必要が生じ，その手間は膨大であるため現実的ではない．

2.2 必要に応じて転送する手法

モバイルエージェントが移動する際に，まずモバイルエージェントの状態を先に移動し，移動先で実際にプログラムコードが必要になった時点で，プログラムコードを転送する手法である．

この手法は，プログラムコードを必要な量だけ転送するため，転送量を最小に押さえることができ，さらに，モバイルエージェントが移動先で処理を開始するまでのレスポンスタイムも優れている．

しかし，この手法は，コンピュータのネットワークが，常時接続でなければならないため，移動元と移動先での非同期実行というモバイルエージェントの特長のひとつが失われてしまう．

2.3 一括して転送する手法

モバイルエージェントの移動の際に，モバイルエージェントが使用する可能性がある全てのプログラムコードを一括して全て転送する手法である．この手法は，一度だ

けの通信で済むため，非常時接続環境での優位性を備えることができる．

しかし，モバイルエージェントは自律的に判断して処理を行うプログラムであるから，移動先で必要になるプログラムコードは移動先のコンピュータの環境に依存する．したがって，移動先で必要になるプログラムコードを移動前に把握することは非常に困難である．この手法を採用している多くのモバイルエージェントシステムでは，移動先で必要になるか，ならないかに関わらず，モバイルエージェントが生涯にわたって使用する全てのプログラムコードを転送しているのが一般的である．このことから，移動先で不必要なプログラムコードまで転送してしまい，モバイルエージェントの移動にかかるオーバーヘッドが大きいという問題がある．

3 提案手法

提案手法では，2.3 節で述べた “一括して転送する手法” にキャッシング技術を組み合わせることで，非常時接続環境での優位性を確保したままモバイルエージェントの移動の高速化をはかる．

3.1 モバイルエージェントの構造

従来のモバイルエージェントの構造は，内部に状態とプログラムコードを保持していることが一般的であった．しかし，この構造は，エージェントが移動する度に，状態の変化しないプログラムコードも転送するため，非常に効率が悪い．状態が変化しないデータを，何度も転送することは非常に無駄である．

そこで，提案手法では，モバイルエージェントとプログラムコードを分離し，プログラムコードの代わりに，プログラムコード名のリストを保持させるようにする．プログラムコード名のリストは，プログラムコードのデータのリストよりも，遥かにデータサイズが小さく，モバイルエージェントが常に保持していてもエージェントのデータサイズが大きく増加することはない．

3.2 モバイルエージェントの移動

提案手法では，コンピュータにプログラムコードをキャッシングするためのキャッシュ装置を設置している．

また，モバイルエージェントが移動する際に，モバイルエージェントが移動先で必要なプログラムコードが移動先キャッシュ装置に全て存在していた場合をキャッシュヒットとよび，逆に，不足しているために移動元から移動先へプログラムコードを転送する必要がある場合をキャッシュミスとよぶ．

以下に，キャッシュミスとキャッシュヒットの場合の処理手順を述べる．

キャッシュミスの場合（図 1）

1 移動元から移動先へモバイルエージェントの状態とプログラムコード名のリストを転送する．

2 プログラムコード名のリストの中で，キャッシュされていないプログラムコードが無いか問い合わせる．

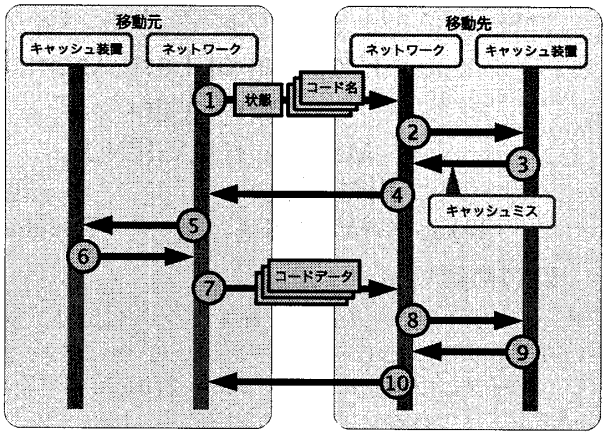

図 1: キャッシュミス

3 キャッシュ装置は問い合わせに対して，キャッシュされていないプログラムコード名のリストを返す（キャッシュミス）．

4 移動先にキャッシュされていないプログラムコード名のリストを移動元に通知する．

5-6 移動先のキャッシュ装置にキャッシュされていないプログラムコードを移動元のキャッシュ装置から取り出す．

7 取り出したプログラムコードを，移動先へ転送する．

8-9 プログラムコードを移動先のキャッシュ装置へ格納する．

10 エージェントの移動処理を終了する．

キャッシュヒットの場合（図 2）

1. 移動元から移動先へ，モバイルエージェントの状態と，プログラムコード名のリストを転送する．
2. プログラムコード名のリストがキャッシュ装置内に全て存在するか問い合わせる．
3. 全て存在している場合は，空リストを返す（キャッシュヒット）．
4. モバイルエージェントの移動処理を終了する．

4 キャッシュ装置

この章では，3 章で述べた提案手法で用いるキャッシュ装置の詳細について述べる．

キャッシュ装置とは，モバイルエージェントが必要とするプログラムコードを一時的に格納しておくための装置である．モバイルエージェントがプログラムコードを取得するために，ネットワークに接続された別のコンピュータから転送して取得するよりも，同じコンピュー

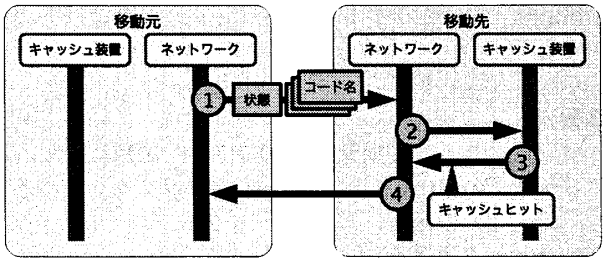

図 2: キャッシュヒット

タにあるキャッシュ装置から読み込んで取得する方が，遥か短時間で取得できることは明白である．

以下に，提案手法の実現に必要なキャッシュ装置の設計について述べる．

4.1 プログラムコードの名前空間

モバイルエージェントを用いて構築したアプリケーションでは，ユーザが任意に作成したモバイルエージェントが参加する場合も考えられる．この場合，キャッシュ装置のデータ構造として，単純にプログラムコード名とプログラムコードのデータを対にしたようなハッシュテーブルを用いただけでは，名前空間の衝突が発生し，モバイルエージェントが予期しない処理を行ってしまう可能性がある．これは，アプリケーション内に悪意を持ったモバイルエージェントを投入された場合に，キャッシュ内の任意のプログラムコードを置き換えられる “キャッシュ汚染” を許すことを意味する．

また，Java や C++などのいくつかのプログラミング言語では，プログラムコードに対する名前空間の機能を提供しているものがあるが，言語仕様として，その機能を使用しなければならないとは定められていないため，名前空間の衝突を発生させることは事実上可能である．したがって，このような既存の名前空間の機能をそのまま利用することはできない．

これらのことから，複数のモバイルエージェントが同名のプログラムコードを持っていても，衝突が発生しないことを保証するデータ構造を設計する必要があることがわかる．次の節で，モバイルエージェント間での名前空間の衝突が発生しないキャッシュのデータ構造について述べる．

4.2 キャッシュ装置

提案手法で用いるキャッシュ装置は，プライベートキャッシュ層と，共有キャッシュ層の２つの階層から構成される．図 3 にその概要図を示す．

プライベートキャッシュ層 モバイルエージェントは，常にプライベートキャッシュ層に対して，プログラムコードを格納または読出する．プライベートキャッシュ層は，モバイルエージェント間での名前空間の衝突を避けるために，モバイルエージェントの種類（クラス）別に論理的に分割されている．この論理的に分割された領域をセグメントと呼ぶ．セグメントの数は，モバイルエージェントの種類数と等しい数だけ作成されることになる．

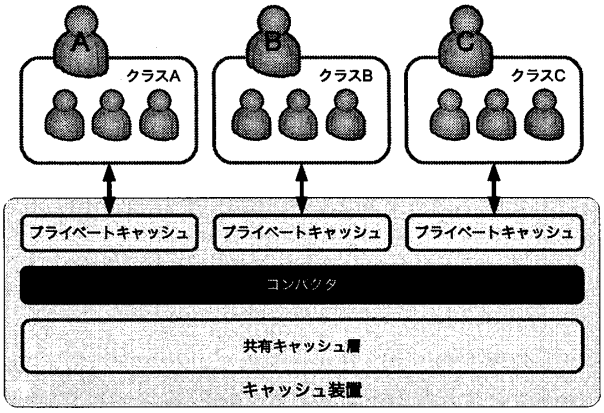

図 3: キャッシュ装置の概要

共有キャッシュ層 実際的なアプリケーションでは，複数種類のモバイルエージェント間で，共通のプログラムコードを利用する場合が考えられる．このような場合，プライベートキャッシュ層だけでは，セグメント間でプログラムコードを重複してキャッシュする場合があるため，キャッシュ装置に用いるメモリ空間を無駄に使用してしまう．

共有キャッシュ層とは，プライベートキャッシュ層のセグメント間で重複したプログラムコードを，共有化して格納し，キャッシュ装置に用いるメモリ空間の使用量を最適化するための層である．

ここでの最適化をコンパクションよび，最適化する装置をコンパクタとよぶ．

4.3 コンパクタ

コンパクタとは，プライベートキャッシュ層のセグメント間で重複して格納されているプログラムコードを探し出し，共有キャッシュ層へ格納するための装置である．

コンパクタを設置せずに，モバイルエージェントがキャッシュ装置へプログラムコードを格納する際に，重複したコードが既に存在するかを確認すれば良いように思えるが，実際はそうではない．ある２つのプログラムコードが完全に等しいことを判別するためには，プログラムコードのデータに対して完全一致が認められなければならない．モバイルエージェントが，キャッシュ装置に対してプログラムコードを格納する際に，キャッシュ装置内に完全一致するプログラムコードが存在するかを判定していたのでは，キャッシュのレイテンシが大きくなってしまい，モバイルエージェントのパフォーマンスが下がってしまう．完全一致ではなく，プログラムコードのデータに対するハッシュ値を用いて比較する方法も考えられるが，大規模なアプリケーションでは，プログラムコード数が膨大になるため，シノニムが発生してしまう可能性が高くなる．

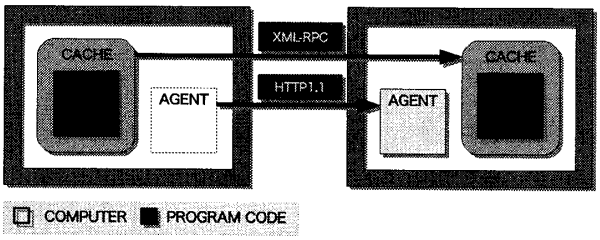

図 4: モバイルエージェントの移動に用いるプロトコル

以上の理由で，コンピュータのプロセス上では，モバイルエージェントのスレッドとは別のスレッドとして駆動するコンパクタを設置している．

4.4 キャッシュの削除

提案手法では，モバイルエージェントとプログラムコードが分離されているため，闇雲にキャッシュを削除してしまうと，モバイルエージェントが必要なプログラムコードを二度と読み込めなくなってしまう．

したがって，キャッシュ装置が設置されているコンピュータ内に，削除しようとするプログラムコードを利用する可能性があるモバイルエージェントが存在する場合は，削除を取りやめなければならないという条件がある．

この条件さえ満たしていれば，キャッシュの削除に LRU や MRU などといった既存のキャッシュアルゴリズムを適用することができる．

5 提案手法の実装

3 章で述べた提案手法を，我々が開発しているモバイルエージェントの構築環境と実行環境を提供するモバイルエージェントフレームワークである Maglog(Mobile AGent system based on proLOG)[1, 2] 上に実装した．

5.1 モバイルエージェント

Maglog は，Java で実装されており，Maglog におけるモバイルエージェントは，MaglogAgent クラスとして実現されている．また，MaglogAgent クラスは，Prolog インタプリタの Java 実装である PrologCafe[3] の Prolog クラスを継承しており，モバイルエージェ

ント自身が Prolog のインタプリタである．PrologCafe は，Prolog のソースコードを Java のソースコードへ変換する機能を持っており，任意の Prolog 述語を Predicate クラスのサブクラスに変換することができる．

Maglog では，1 つのモバイルエージェントに 1 つのスレッドが割り当てられて活動し，Predicate クラスのサブクラスを順次的に実行しながら処理を行う．

Maglog においてキャッシングするプログラムコードは，Predicate クラスのサブクラスのバイトコードである．

5.2 モバイルエージェントの移動

Maglog では，通信プロトコルとしてモバイルエージェントの転送に HTTP/1.1，プログラムコードの転送に XML-RPC[4] を用いている．

Maglog でのモバイルエージェントの移動は以下の要領で行う．

移動元

Maglog では，移動元からの HTTP 接続に java.net パッケージの HttpURLConnection クラスを，モバイルエージェントの送信に java.io パッケージの ObjectOutputStream クラスを使っている．

移動元でのモバイルエージェントの送信手順は以下の通りである．

1. モバイルエージェントのスレッドを停止する．
2. HTTP/1.1 で移動先のコンピュータと接続を確立する．
3. オブジェクト出力ストリームを作成する．
4. オブジェクト出力ストリームに対してプログラムコード名のリストを書き出す．
5. オブジェクト出力ストリームに対してモバイルエージェントの状態を書き込む．
6. オブジェクト出力ストリームを閉じる．
7. HTTP/1.1 接続を終了する．
8. 移動元のモバイルエージェントを削除する．

移動先

Maglog では，移動先の HTTP 接続として javax.servlet.http パッケージの HttpServlet クラスを，モバイルエージェントの受信に java.io パッケージの ObjectInputStream クラスを使っている．

移動元でのモバイルエージェントの受信手順は以下の通りである．

1. HttpServletRequest からオブジェクト入力ストリームを作成する．
2. オブジェクト入力ストリームからプログラムコード名のリストを読み込む．
3. 移動元から受信した上記のプログラムコード名のリストを用いてキャッシュ装置内に格納されていないプログラムコード名のリストを作成する．
4. キャッシュ装置内に格納されていないプログラムコード名のリストを用いて，移動元からプログラムコードを受信する．
5. 受信したプログラムコードをキャッシュ装置へ格納する．
6. オブジェクト入力ストリームからモバイルエージェントの状態を読み込む．

7. オブジェクト入力ストリームを閉じる．

8. モバイルエージェントにスレッドを割り当てて活動を開始する．

5.3 キャッシュ装置

キャッシュ装置は，複数のモバイルエージェントが同時にアクセスするため，スレッドセーフでなければならない．そのため，Java での並列プログラミング機能を提供する java.util.concurrent パッケージにある ConcurrentHashMap クラスを用いて実装している．

キャッシュ装置のデータ構造は，ConcurrentHashMap クラスを用いて実装した 2 種類のハッシュテーブルから構成される．

1 つめは，プログラムコード名をキー，プログラムコードデータを値としたハッシュテーブルで，これをコードテーブルと呼ぶ．2 つめは，節 4.1 で述べたモバイルエージェント間での名前空間の衝突を避けるために，モバイルエージェントのクラスをキー，それに対応するコードテーブルを値とするハッシュテーブルである．

また，キャッシュ装置は XML-RPC による通信機能を持っており，キャッシュ装置間でプログラムコードを送受信することができる．

6 評価

6.1 モバイルエージェントの移動に要する時間

提案手法により，モバイルエージェントの移動がどの程度，高速化するかを評価するために，モバイルエージェントの移動に要する時間を測定した．測定は，Pentium4 3.0GHz，主記憶 1GB のコンピュータ 2 台を 1000BASE-T のイーサネットに接続した実験環境で行った．図 5 は，モバイルエージェントが 2 台のコンピュータを 10 往復するのに要した時間を測定した結果である．横軸では，モバイルエージェントが必要なプログラムコード数を，0 個から 512 個まで変化させている．

キャッシュを用いない従来手法 (without cache) では，プログラムコード数に比例してモバイルエージェントの移動に要する時間が長くなっていることがわかる．これは，モバイルエージェントが移動の際に，毎回必ずプログラムコードも転送しているためである．

これに対して，キャッシンングを用いた提案手法 (with cache) では，プログラムコード数が増加してもモバイルエージェントが 1 0 往復の移動に要する時間がほとんど変わらないことがわかる．これは，モバイルエージェントが 1 回目の往復の際に，移動元と移動先のコンピュータに，プログラムコードをキャッシングしているためである．

2 回目以降の往復では，移動先のコンピュータに設置されているキャッシュ装置のプログラムコードを利用することができるため，無駄なプログラムコードの転送は行わない．

以上の結果から，提案手法は，モバイルエージェントが往復する際に非常に有効的であることがわかった．

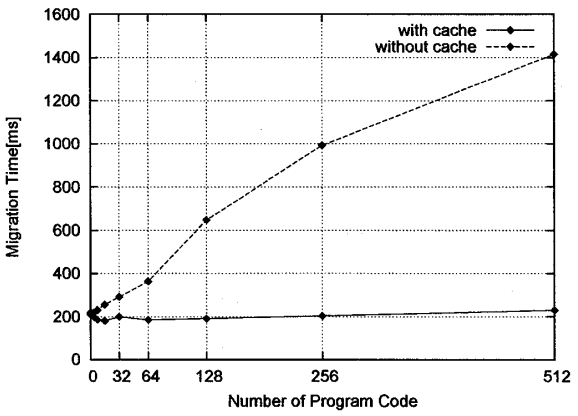

図 5: モバイルエージェントが 2 台のコンピュータを 10 往復するのに要した時間

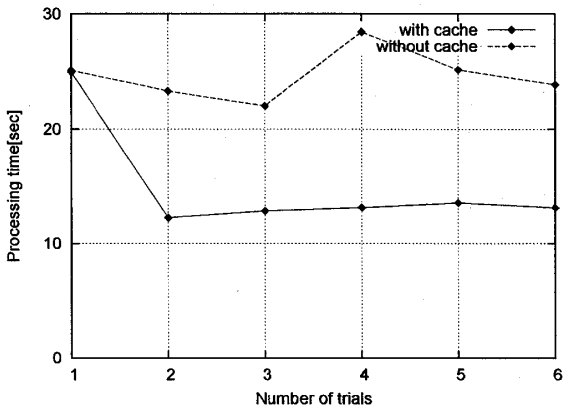

図 6: 会議日程調整に要した時間

6.2 実際的なアプリケーションを用いた評価

実際的なアプリケーションにおける提案手法の有効性を評価するために，我々が Maglog を用いて開発している会議日程調整システム [5, 6] を用いた評価を行った．以下，会議日程調整システムの概要と，提案手法の評価について述べる．

会議日程調整システムの概要

我々は，モバイルエージェント技術に基づく会議日程調整システムを開発している．このシステムでは，“どの日時が空いているか”，“どの日時が空いていないか”，などの予定を，会議の参加者に対してモバイルエージェントが能動的に収集することにより，会議日程調整を円滑に行うことがねらいである．

このシステムにおけるモバイルエージェントは，会議の参加者に対する予定の収集だけではなく，ログイン認証や，参加者の予定が重なった際の交渉，日程調整結果の通知なども行うため，システム内では多数のモバイルエージェントが頻繁に活動している（図 7）．

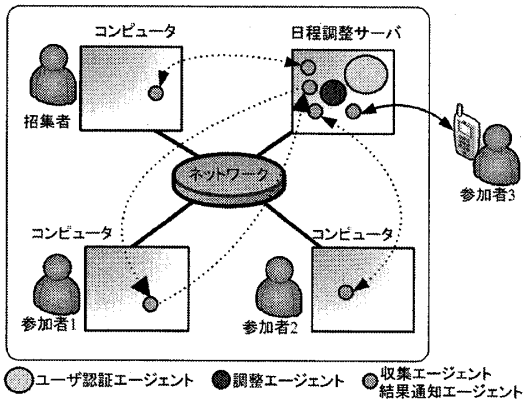

図 7: 会議日程調整システムの概要

評価

前述した会議日程調整システムにおいて，提案手法を用いない場合と用いた場合の会議日程調整に要する時間を比較する実験を行った．

実験環境は，1000BASE-T のイーサネットに接続された計算機 10 台で構築した会議日程調整システムとする．

ただし，本来のシステムでは会議の参加者が行う作業である以下の項目については，即座に自動で行うように会議日程調整システムのプログラムを変更した．

- モバイルエージェントが予定の入力を依頼してきた場合の予定入力
- モバイルエージェントが予定の交渉を行ってきた場合の返答

実験結果を図 6 に示す．縦軸は会議日程調整に要した時間，横軸は会議日程調整の試行回数である．

会議日程調整に要する時間に関して，提案手法を用いない場合（without cache）は，試行回数に関わらず，約 22 秒から 28 秒を要している．これに対して，提案手法を用いた場合（with cache）は，2 回目以降の試行から，約 14 秒まで減っていることがわかる．これは，2 回目以降の試行からモバイルエージェントが必要とするプログラムコードが，各々のコンピュータにキャッシングされたためであると考えられる．

以上の結果より，提案手法を用いた場合に，モバイルエージェントが頻繁に移動するアプリケーションのパフォーマンスを改善できることを確認した．

7 おわりに

本論文では，各々のモバイルエージェントが必要とするプログラムコードを，ネットワークに接続されたモバイルエージェント実行環境にキャッシングすることにより，エージェントの高速な移動を実現する手法の提案を行った．提案手法をモバイルエージェントシステムに実装し，現実的なアプリケーションを用いて実験を行った結果，提案手法を適用することで多数のモバイルエージェントが頻繁に移動するようなアプリケーションにおけるパフォーマンスの改善できることを確認した．

また，本論文では，モバイルエージェントの移動先へプログラムコードを一括して転送する場合について述べたが，移動先でのオンデマンドによる転送を併用したレスポンスタイムを向上させる方法についても，今後検討していく予定である．

参考文献

[1] Motomura, S., Kawamura, T. and Sugahara, K.: Logic-Based Mobile Agent Framework with a Concept of "Field", *IPSJ Journal*, Vol. 47, No. 4, pp. 1230–1238 (2006).

[2] Kawamura, T., Motomura, S. and Sugahara, K.: Implementation of a Logic-based Multi Agent Framework on Java Environment, *Proceedings of IEEE International Conference on Integration of Knowledge Intensive Multi-Agent Systems* (Hexmoor, H.(ed.)), pp. 486–491 (2005). Waltham, Massachusetts, USA.

[3] Banbara, M. and Tamura, N.: Translating a Linear Logic Programming Language into Java, *Proceedings of the ICLP'99 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages* (M.Carro, I.Dutra et al.(eds.)), pp. 19–39 (1999).

[4] Winer, D.: XML-RPC Specification, http://www.xmlrpc.com/spec/.

[5] Kawamura, T., Hamada, Y., Sugahara, K., Kagemoto, K. and Motomura, S.: Multi-Agent-based Approach for Meeting Scheduling System, *Proceedings of IEEE International Conference on Integration of Knowledge Intensive Multi-Agent Systems*, pp. 79–84 (2007). Waltham, Massachusetts, USA.

[6] Kawamura, T., Motomura, S., Kagemoto, K. and Sugahara, K.: Meeting Arrangement System Based On Mobile Agent Technology, *Proceedings of the 2nd International Conference on Web Information Systems and Technologies*, pp. 117–120 (2006). Setubal, Portugal.