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†
Masayuki HIGASHINO
川村 優生†
Takao KAWAMURA

本村 眞一†
Shinichi MOTOMURA
笹間 俊彦†
Toshihiko SASAMA
菅原 一孔†
Kazunori SUGAHARA

1 はじめに

エージェントとは、人間の代理として自律的に処理を
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た処理を、移動
元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、移動元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を増大することを
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
ヘルドが大きくな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ルドが、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ルドを小さく
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があるコンピュー
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
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が確認した。

†鳥取大学大学院 工学研究科 情報エレクトロニクス専攻

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
転送方法は以下の3種類に分類できる。

この章では、それぞれの概念と課題を述べる。

2.1 コンピュータに予め配置しておく方法

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、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が非常に高速である。

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実用的ではない。

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状態だけで転送す
るため、転送量を大きく抑えることができる。さらに、
モバイルエージェントが移動元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、移動先と移動先での
非同期実行というモバイルエージェントの特性の一つが
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" にキャッシュ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、通信用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ュ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上で存在していた場合をキャッシュヒットと解釈し、逆に、必要ない場合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する。

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、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。

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
タにあるキャッシュ装置から読み込んで取得する方が、遅か短時間で取得す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、呼び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応させるようなハッシュテーブルを用いる。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、プログラムコードの格納または書き出し、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、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。

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利用する場合を考えられる。このような場合、プライベートキャッシュ層だけでなく、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および、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
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断するためには、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
装置内部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が、キャッシュのレイアウトが大きく
変更してしまう。モバイルエージェントのバフォーマンス
が低下してしまう、完全一致でなく、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、シンフォニが発生してし
まう可能性が高くなる。

355
（第4分冊）
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、関連すべてを削除してしまった、モバイルエージェントが必要なプログラムコードを再度と読み込む必要がなくなってしまう。

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。

この条件が満たされていれば、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クラクを継承しており、モバイルエージェントが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においてキャッシュ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続き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。

移動元
Maglogでは、移動先のHTTP接続としてjavax.servlet.httpパッケージのHttpServletRequest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続き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
キャッシュ装置では、数のモバイルエージェントが同時にアクセスするため、スレッドセーブでなければならず、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、該当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でき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のオ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が、これにより、モバイルエージェントの移動の際に、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、キャッシュ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10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ュングしているためである。
2回目以降の往復では、移動元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可能で、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往復するために要した時間

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可能である。
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なく、ログイン認証や、参加者の予定が変更された場合の交渉、日程調整結果の通知なども含むため、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）。

357
（第4分冊）
図7 会議日程調整システムの概要

評価
前後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に依頼して来る場合の予定入力
・ 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。横軸は会議日程調整に要した時間、縦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26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ュ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ュすることにより、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、移動先でのオーディオによる転送を併用したレストラン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