モーバイル・メモリセグメントを用いたモーバイルプログラミング言語の非抽出型実現方式

吉田雅年[†] 松原克弥[††] 加藤和彦[††,†††]

モーバイルエージェント機能を持つプログラミング言語システムの研究開発が世界各地で活発に進められている．多くのモーバイルエージェントシステムでは，モーバイルエージェント移動の中心技術としてシリアリゼーション技術を用いている．シリアリゼーション技術とは，転送元のアドレス空間上で互いにアドレス参照し合うデータ構造を，メッセージ転送可能な形式に直列化し，転送先のアドレス空間上で元のデータ構造を復元する技術である．あるプログラミング言語システムにモーバイルエージェントの機能を導入しようとする場合，プログラミング言語システムの実行時システムの実行状態を直列化する必要があるため，それを実装するシステムプログラマには，対象プログラミング言語システムに関する深い知識と多大な作業量が要求される．本論文では，筆者らの研究グループが開発を進めている Planet システムが提供するモーバイル・メモリセグメント機能を活用し，明示的なシリアリゼーション記述をすることなしに，モーバイルエージェントシステムを系統的に実現する方法を提案する．提案方式の実現例として，筆者らの研究グループが開発を進めている Planet システムが提供するモーバイル・メモリセグメント機能を利用し，スクリプト言語 Tcl にモーバイル化機能を導入した例を報告する．

Non-extraction Method of Implementing Mobile Programming Language Systems with Mobile Memory Segment

Masatoshi Yoshida,[†] Katsuya Matsubara[††] and Kazuhiko Kato[††,†††]

Recently, research of programming language systems with mobile-agent functionality attracts many researchers' concern. Most mobile agent systems utilize the serialization technique as a key technique to implement object mobility. With the serialization technique, complex data structures in a virtual address space in a source site are serialized for message communication, and the structure is reconstructed in an address space in a destination site. To introduce mobile-agent functionality into a programming language system, it is required for the implementer to spend a significant amount of time to deeply understand the details of the system's structure and to write explicit complicated serialization codes. In this paper, we describe a novel scheme to introduce a mobile-agent functionality into a programming language system in a systematic way. By using mobile memory-segment, the scheme enables the implementer to implement object mobility without explicit description of serialization. We report our experience to mobilize the Tcl script language with the mobile memory-segment provided by the Planet system, a middleware for mobile objects designed and implemented by the authors' research group.

1. はじめに

近年，モーバイルエージェントシステムの研究開発が世界各地で活発に進められている．モーバイルエージェントとは，ネットワークを介して実行中にホスト間を相互に移動可能なプログラムである．モーバイルエージェントが実行されているホストから別のホストに移動する際には，自発的に自身の実行を一時停止し，目的のホストに移動した後にプログラムの実行を再開

† 筑波大学大学院博士課程工学研究科
Doctoral Program in Engineering, University of Tsukuba
†† 筑波大学電子・情報工学系
Institute of Information Sciences and Electronics, University of Tsukuba
††† 科学技術振興事業団
Japan Science and Technology Corporation

する．

プログラミング言語にモーバイル機能を導入するためには，3つの機能が必要となる．1つは，モーバイルエージェントの実行状態の移動である．モーバイルエージェントの実行状態とは，モーバイルエージェント自身のプログラムコード，属性値，スレッドなどである．これらの実行状態を移動先のホストに再現し，モーバイルエージェントの実行が再開できる必要がある．もう1つの必要な機能は，モーバイルエージェントによる外部資源の操作状態の一貫性を保つことである．外部資源とは，ファイルやプロセス間通信に代表されるエージェント外部に存在する資源である．外部資源の操作状態の一貫性を保持することで，移動後のモーバイルエージェントは移動前の外部資源の操作状態と整合性がとれた状態で実行を再開することが可能となる．3つ目は，モーバイルエージェントの使用環境によっては必要となる，ホストやモーバイルエージェントの不正な行為を防ぐためのセキュリティ機能である．これらの機能を実現するために，現在のモーバイルプログラミング言語にはインタプリタ型言語をベースとしているものが多い[2),8),10)]．インタプリタ型言語では，実行時にモーバイルエージェントを解釈実行するため，モーバイルエージェントの状態を把握しやすく，これらの機能を比較的実現しやすいからである．本論文では，以上に述べた3つの機能のうち，インタプリタ型モーバイルプログラミング言語を実現するうえでの，モーバイルエージェントの実行状態の移動方法についてを扱う．

インタプリタ型言語をベースとした多くのモーバイルエージェントシステムでは，モーバイルエージェントの実行状態の移動を，抽出型実現方式を用いて行っている．抽出型実現方式とは，インタプリタからモーバイルエージェントの実行状態を抽出し，メッセージ転送が可能なように変換した後，移動先のホストに転送し，インタプリタ上に転送された実行状態を復元する方式である．抽出型実現方式では，インタプリタに含まれるモーバイルエージェントの実行状態を抽出する必要があるが，そのためには実行状態のインタプリタ内での位置を特定する必要があり，実装を行うシステムプログラマには，対象プログラミング言語に関する深い知識と多大な作業量が要求される．また，抽出型実現方式では，モーバイルエージェントの移動先にあらかじめインタプリタをインストールしておく必要がある．しかし，インターネットに代表されるオープンなネットワーク環境では，各ホストの管理権限が分散しており，各ホストにモーバイルエージェントのための共通な環境を用意することが実質的に困難である．

本論文では，モーバイルプログラミング言語の実現に，インタプリタから実行状態の抽出を行わずに，インタプリタの内部状態をほぼそのまま転送する方法を提案する．提案方式では，インタプリタの内部状態を転送する方法として，筆者らの研究グループで開発を進めている Planet システム[3)] が提供しているモーバイル・メモリセグメント機能を用いる．モーバイル・メモリセグメントとはネットワークを介してホスト間を移動可能なメモリセグメントで，モーバイル・メモリセグメント上に配置されたコードやデータは，自動的にホスト間を移動可能となる．また，モーバイル・メモリセグメント上のコードにともなうスレッドも同時に移動可能となる．提案方式では，実装コストと実行時パフォーマンスへの柔軟な対応が可能である．実装時コストの削減を最優先にした場合には，インタプリタからモーバイルエージェントの実行状態の抽出をまったく行う必要がないため，抽出型実現方式と比較して容易にモーバイルプログラミング言語の実現が可能となる．また，インタプリタ自体が移動するため，移動先にあらかじめインタプリタをインストールしておく必要がない．つまり，モーバイル化された任意のインタプリタ型プログラミング言語で記述されたモーバイルエージェントが移動可能となる．また，プログラミング言語よりも下の階層でセキュリティ機能を実現可能であるので，各々のプログラミング言語に関してセキュリティ機能を付け加える必要はない．これらの特長により，提案方式は複数言語を統一的にモーバイル化する方法を提供する．

以下，2章ではプログラミング言語をモーバイル化する際の従来の実現法について検討する．3章では，Planet システムが提供するモーバイル・メモリセグメントについて述べ，4章でモーバイル・メモリセグメントを用いたモーバイルプログラミング言語の実現法を提案する．5章で，提案方法によりスクリプト言語 Tcl にモーバイル機能を導入した例を報告し，移動時オーバヘッドに関する実験について述べる．最後に6章でまとめと今後の課題について述べる．

2. 従来型実現方式の検討

モーバイルエージェントを，プログラム実行中にホスト間で移動可能とするためには，モーバイルエージェントのプログラムコード，変数などの属性値，また，スタックやプログラムカウンタなどの実行状態を移動先のホストに再現する必要がある．このモーバイルエージェントの実行状態を移動する方法が，モーバ

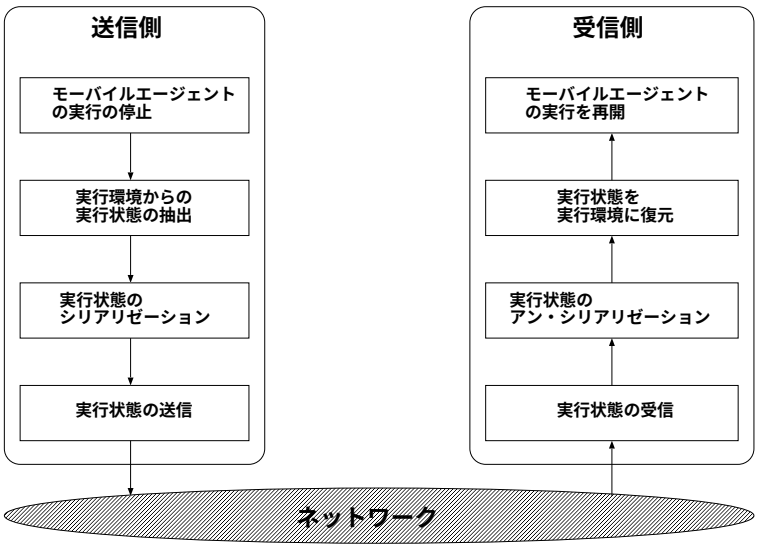

図 1 抽出型実現方式を用いたモーバイルプログラミング言語の実行状態の移動
Fig. 1 Migration of mobile programmming language state using the extraction method.

イルプログラミング言語を実現するうえで重要な技術である．本章では，モーバイルエージェントの実行状態の移動方法として，プロセスマイグレーション技術を用いた方式と，従来モーバイルエージェントシステムの多くで用いられている抽出型実現方式について検討する．

2.1 プロセスマイグレーション方式

プロセスマイグレーション技術とは，実行中のプロセスをホスト間で移動可能とする技術である．プロセスマイグレーション技術は，プロセスのアドレス空間全体をそのまま転送することでプロセスの移動を実現している．プロセスマイグレーション技術を用いてモーバイルエージェントを実行しているインタプリタのプロセスを転送することで，実行状態を含んだモーバイルエージェントをホスト間で移動可能とすることが可能である．プロセスマイグレーション技術を用いた方式の最大の利点は，既存の言語をモーバイル化しようとしたときに，インタプリタにほとんど手を加える必要のないことである．一方，プロセスマイグレーション技術による方式では，モーバイルエージェントの移動がプロセス単位となるため，モーバイルエージェントの移動時オーバヘッドが大きくなるという問題がある．また，同一な仮想記憶空間には 1 つのモーバイルエージェントしか存在できず，複数のモーバイルエージェントが同一ホストに存在し，通信を行うような場合には効率が悪い．

2.2 抽出型実現方式

近年のモーバイルエージェントシステムの多くは，抽出型実現方式を用いている．抽出型実現方式では，モーバイルエージェントの実行状態を表すデータ構造の移動は，インタプリタからの実行状態の抽出，抽出した実行状態の転送，転送された実行状態の復元の 3 つの処理により実現できる（**図 1** 参照）．抽出型実現方式を用いた場合の実行状態の抽出，転送，復元の 3 つの処理は以下のように行う[4]．

(1) モーバイルエージェントの実行状態の抽出
モーバイルエージェント移動のためのプリミティブが呼ばれると，モーバイルエージェントの実行を停止し，その時点でのモーバイルエージェントの実行状態の抽出を行う．モーバイルエージェントの実行状態の抽出とは，モーバイルエージェントの実行状態を表すデータ構造のアドレス空間上の位置を特定し，シリアリゼーション技術によりネットワーク転送が可能な表現形式に変換することである．シリアリゼーション技術とは，転送元のアドレス空間で互いにアドレス参照し合うデータ構造を，メッセージ転送が可能な形式に変換する技術である．

(2) モーバイルエージェントの実行状態の転送
(1) で変換された実行状態を，移動先のホストに転送する．

(3) モーバイルエージェントの実行状態の復元
移動先のホストでは，受信したモーバイルエージェントの実行状態を復元するための実行環境を準備し，そのアドレス空間にアン・シリアリゼーションによりモーバイルエージェントの実

行状態を表すデータ構造を復元する．実行状態の復元後に，モーバイルエージェントの実行を再開する．

抽出型実現方式を用いてモーバイルエージェントの実行状態の移動を実現する場合，最も大きな技術的課題は，モーバイルエージェントの実行状態の抽出である．インタプリタ型言語ではインタプリタがプログラミング言語を実行時に解釈し，処理を行う．そのため，モーバイルエージェントの実行状態は，モーバイルプログラミング言語のインタプリタのアドレス空間上に分散して存在しており，実行状態の抽出を行うためには，実行状態が存在するインタプリタにおけるアドレス空間上の位置を特定する必要がある．しかし，このようなアドレス空間上の位置を特定するためには，対象となるプログラミング言語システムに関しての深い知識と多大な作業量が必要である．また，インタプリタ型プログラミング言語の実装によっては，モーバイルエージェントの実行状態は実行時システムのスタック上に存在する場合があり，実行状態の抽出を行うためには実行時システムのスタック構造を解析する必要がある．モーバイルエージェントにおける実行状態の移動を実現するための実装を行うシステムプログラマは，対象プログラミング言語システムのソースコードを理解し，モーバイルエージェントの実行状態を抽出するための処理を記述する必要がある．また，モーバイルエージェントの実行状態を抽出するために記述した処理は，対象となる処理系に特化した処理であり，別のプログラミング言語にそのまま適用することはできない．

2.2.1 抽出型実現方式の例

従来の多くのモーバイルエージェントシステムでは，モーバイルエージェントの実行状態の移動に抽出型実現法を用いている．抽出型実現法を用いているシステムとして，IBM 社の Aglets[4]，ObjectSpace 社の Voyager[5]，Dartmouth 大学の D'Agents[1),9)]，Kaiserslautern 大学の Ara[7] などがある．以下では，抽出型実現法を用い，かつ複数言語への対応を行っているモーバイルエージェントシステムである D'Agents, Ara について述べる．

D'Agents は，Tcl/Tk，Scheme，Java でモーバイルエージェントが記述可能なモーバイルエージェントシステムである．D'Agents の主な機能は，モーバイルエージェント間の通信，セキュリティ機能，デバッグ/追跡ツールなどである．D'Agents では抽出型実現法によって，モーバイルエージェントの実行状態の移動を実現している．D'Agents でサポートされている Tcl では，スクリプトの解釈をある関数を再帰的に呼び出すことで実現している．そのため，Tcl スクリプトの実行状態は実行時システムのスタック上に存在し，容易には抽出することができない．D'Agents では Tcl のソースに大幅な変更を行い，スタックを明示的にデータとして保持するようにすることで，Tcl スクリプトの実行状態の抽出を実現している．

Ara では，モーバイルエージェントは Tcl，C/C++ で記述可能である．Ara でも，D'Agents の場合と同様にインタプリタに変更を加え，実行状態を抽出型実現方式により抽出することでモーバイルエージェントの実行状態の移動を実現している．Ara では，実行時システムのスタックに分散したモーバイルエージェントの実行状態の移動は，実行時システムのスタック全体をシリアリゼーション技術により転送することで実現している．

3. Planet システムのモーバイルメモリセグメント機構

本章では，提案方式を説明する準備として，Planet システム[3),11)] のシステムモデルと，Planet システムが提供するモーバイル・メモリセグメント機構の概略を説明する．

3.1 Planet のシステムモデル

Planet のシステムモデルは，モーバイルオブジェクト，DSR（Distributed Shared Repository；分散共有格納庫の略），プレース，プロテクションドメイン，オブジェクトポートの 5 つの基本抽象概念を用いて説明することができる（**図 2** 参照）．

計算処理の対象となるデータおよびデータに対する

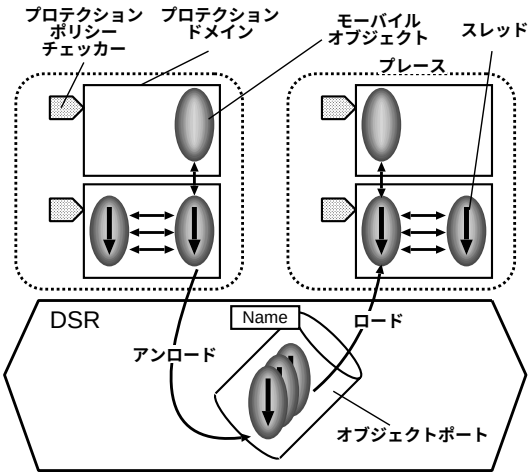

図 2 Planet のシステムモデル
Fig. 2 System model of the Planet.

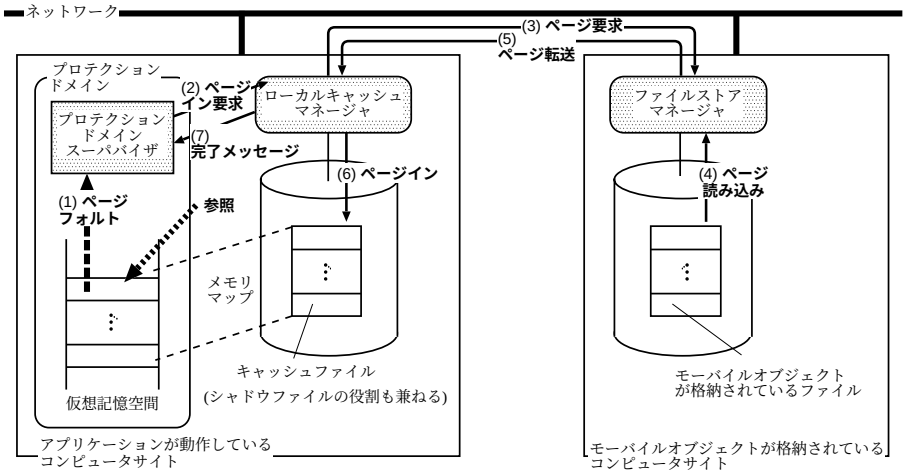

図 3 リモートメモリマップ機構
Fig. 3 Mechanism of remote memory-mapping.

操作を密閉し，仮想記憶領域から分離可能にしたものをモーバイルオブジェクトと呼ぶ．1 つのモーバイルオブジェクトは，内部にデータとプログラムコードに加えてオブジェクトの実行状態を持つことができる．

Planet のモーバイルオブジェクトは DSR またはプレースのいずれかに存在している．DSR は広域ネットワークと永続的記憶空間を抽象化したものであり，プレースはローカルエリアネットワークと揮発的記憶空間を抽象化したものである．モーバイルオブジェクトを DSR からプレースに移動する操作をロード操作，逆にプレースから DSR に移動させる操作をアンロード操作と呼ぶ．ネットワーク環境において DSR 空間は論理的に 1 つしか存在しないのに対し，プレースは数多く存在する．オブジェクトの実行は，オブジェクトがプレース上にあるときにのみ可能であり，DSR 上にあるときは可能でない．

DSR において，モーバイルオブジェクトはオブジェクトポートと呼ばれるキューに格納される．キューは，システム内で一意で位置独立な名前で識別され，モーバイルオブジェクトのロード /アンロード時に指定される．また，キューにはアクセス権が設定されており，アクセス権を持つプレースのみがキューへの操作が可能でなる．

プロテクションドメインは，プレース上の情報保護の単位である．1 つのプロテクションドメインは必ず 1 つのプレースに属し，同時に 2 つ以上のプレースに属することはできない．オブジェクトがあるプレース上にロードされているとき，オブジェクトはそのプレース上の 1 つ以上のプロテクションドメインに必ず関連づけられている必要がある．個々のプロテクションドメインは独立した仮想アドレス空間を持ち，プロテクションドメインを乗り越えた情報アクセスを行うことができないようメモリ管理ハードウェアを用いて厳格に保護されている．各プロテクションドメインに対応して動作しているプロテクションポリシーチェッカーは，ユーザのプロテクション方針に基づいて，システム資源や他のプロテクションドメインへのアクセスを監視する．

3.2 モーバイル・メモリセグメント機構

Planet システムの実装では，DSR とプレース間のモーバイルオブジェクト転送のために，モーバイルメモリセグメント機構を実現している[11]．Planet システムのモーバイル・メモリセグメント機構は，効率的なデータ転送を実現するリモートメモリマップ機構，アドレス空間依存情報の再配置を実現する反復的再配置機構，スレッドのロード /アンロードを可能にする機構の 3 つから構成されている．

リモートメモリマップ機構（**図 3** 参照）とは，クライアントとなるホストの仮想アドレス空間に，異なるホストにある二次記憶装置上のファイルを仮想的にマップする機構である．この機構では，クライアントで参照または変更されたメモリセグメントのみが実際に転送される．そのため，ネットワークを介した転送量を最小化することが可能である．また，その転送はメモリ管理ハードウェアで検出されるページフォールトによって，自動的に行われる．

反復的再配置機構とは，モーバイル・メモリセグメント内のポインタなどのアドレス空間に依存した情報を繰り返し再配置する機構である．この機構により，1 つの仮想記憶空間の任意の位置にモーバイル・メモ

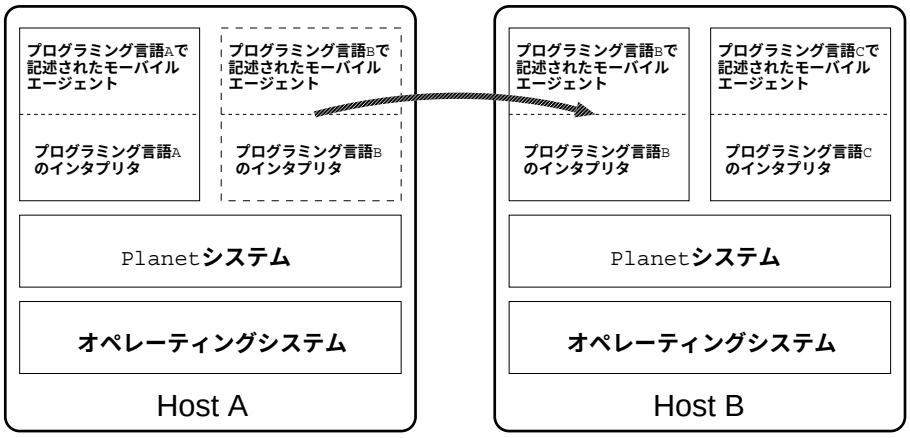

図 4 非抽出型実現方式によるインタプリタの移動
Fig. 4 Migration of language interpreter using the non-extraction method.

リセグメントをロードすることが可能である．また，アンロードしたモーバイル・メモリセグメントを再び任意の仮想記憶空間の任意の位置にロードすることが可能である．

スレッドのロード，アンロードを実現する機構とは，スレッド情報を新たにメモリマップされた仮想記憶領域内に適合するように調整する機構である．

4. モーバイルプログラミング言語の非抽出型実現方式

本章では，モーバイルメモリセグメントを用いることで，モーバイルエージェントの実行状態を抽出せずにインタプリタ型のモーバイルプログラミング言語を実現する方法を提案する．

4.1 非抽出型実現方式

非抽出型実現方式では，モーバイルエージェントの実行状態の移動を実現するために，実行時システム（インタプリタ）の一部または全体をモーバイルエージェントとともに転送する（**図 4** 参照）．本実現方式の最大の特徴の 1 つは，実装コストと実行時性能に対して柔軟な対応ができる点である．インタプリタ全体をモーバイルエージェントとともに移動させる場合には，インタプリタ内のモーバイルエージェントの実行状態を示すデータを特定する必要がなくなり，システムプログラマの知識や実装時の作業量を最小化することができる．しかし，モーバイルエージェントの移動ごとにインタプリタ全体が移動するために，ネットワーク転送オーバヘッドが大きい．逆に，モーバイルエージェントの実行時性能に注目する場合には，インタプリタ内のモーバイルエージェントの実行状態を示すデータを特定することで，モーバイルエージェント移動時のネットワーク転送量を最小化することができる．これを実現するためには，モーバイルエージェントの実行状態を特定するためのインタプリタの実装に対する深い知識を必要とする．

非抽出型実現方式は，インタプリタの一部または全体を実行中にネットワーク間で移動可能とする技術を実現する必要がある．本研究では，Planet システムのモーバイル・メモリセグメント機構を用いてインタプリタの移動を実現する．本機構は，抽出型実現方式でモーバイルエージェントの実行状態を移動する場合に必要だった，実行状態の抽出，メッセージバッファへのコピーとネットワーク転送可能な状態に変換，といった処理が必要なくなり，モーバイルエージェントの実行状態を含むインタプリタのプログラムコード，データと内部実行状態を複雑な変換処理を行わずにネットワーク転送することを可能にする．以下に，インタプリタのプログラムコード，データ，実行状態（スレッド）が移動可能となるために必要な性質について述べ，モーバイル・メモリセグメント機構を用いた実現について述べる．

プログラムコードの動的ロード可能性 非抽出型実現方式では，インタプリタはホストプログラムのアドレス空間内にロードされ，実行される．また，複数の任意の言語のインタプリタが同時に同じアドレス空間内に存在可能とするために，ホストプログラムのアドレス空間内の任意の位置に，インタプリタのプログラムコードをロード可能である必要がある．

データの動的ロード/アンロード可能性 インタプリタにおけるデータも，プログラムコードと同じようにホストプログラムのアドレス空間内の任意の

位置にロード可能である必要がある．データをアドレス空間内の任意の位置にロード可能とする際に問題となるのが，データに含まれるアドレス依存情報（ポインタ）である．このアドレス依存情報をデータがロードされたアドレス空間内の位置に合わせて調整する必要がある．また，動的にロードされたインタプリタ内のデータは，動的にアンロード可能である必要がある．つまり，実行中のインタプリタのデータを，次回の動的ロードが可能な状態でアンロードすることができる必要がある．また，インタプリタが実行中に新たなデータ領域を動的に確保する場合，このデータ領域も動的にアンロードすることができる必要がある．

スレッドの動的ロード/アンロード可能性 インタプリタのスレッドとは，実行されているインタプリタのレジスタ，スタック，プログラムカウンタである．インタプリタのプログラムコードやデータはアドレス空間内の任意の位置に動的ロードされるため，インタプリタのスレッドにはアドレス空間依存情報が含まれる場合がある．スレッドを動的にロード可能とするためには，このアドレス依存情報を，インタプリタのプログラムコードやデータのアドレス空間内の位置に合わせて調整する必要がある．また，動的にロードされたスレッドは，次回の動的ロードが可能な状態でアンロードすることができる必要がある．

以上に述べた性質で，モーバイルエージェントを実行中のインタプリタの一部または全体を，ネットワークを介して移動させることが可能となる．

4.2 モーバイル・メモリセグメントを用いた実現法

本節では，4.1 節で述べた性質を，モーバイル・メモリセグメントを用いて実現する方法を述べる．

プログラムコードの動的ロード可能性を実現するためには，再配置可能コードの技術を用いる．再配置可能なインタプリタのプログラムコードはモーバイル・メモリセグメント機構によってホストプログラムに動的にリンクされる．データの動的ロード/アンロード可能性を実現するためには，モーバイル・メモリセグメント機構が提供する反復的再配置機構を用いる．この機構により，データ内のアドレス空間依存情報を調整する．また，インタプリタの実行中に動的に確保されたデータにも反復的再配置機構を適用可能とするために，モーバイル・メモリセグメント機構が提供する専用のプリミティブを用い，動的に確保されたデータ内のアドレス依存情報の位置を指定する．スレッドの動的ロード/アンロード可能性を実現するためには，モーバイル・メモリセグメント機構が提供するスレッドのロード/アンロード機構を利用する．このスレッドのロード/アンロード機構はインタプリタには透明にスレッドをロード/アンロードすることが可能で，また，原理的に任意の時点でロード/アンロードすることができる．

さらに本実現では，インタプリタのプログラムコード，データ，スレッドの全部分がネットワークを介して転送される場合のネットワーク転送量増加の問題に対して，以下のような対処を行う．

対処 (1) あるホストにおいて，インタプリタでモーバイルエージェントが実行されるとき，プログラムコード，データの全部分が必要となるとは限らない．本実現方式ではこの点に注目し，インタプリタの実行を開始する前には，DSR に存在するインタプリタの実体を仮想的にホストプログラムのアドレス空間にマッピングし，実際に参照や更新が行われた部分のみをメモリセグメント単位でネットワークを介して転送するリモートメモリマップ機構を用いる．リモートメモリマップ機構により，インタプリタのプログラムコードに対する参照や，データに対する参照/更新に局所性がある場合，ネットワーク転送量の削減が可能となる．

対処 (2) プログラムコードに更新を行わないピュアコード方式を用いれば，モーバイルエージェントを実行するインタプリタのプログラムコードに，実行中に変更が行われることはない．また一般に，ある言語のインタプリタのプログラムコードのバージョンアップなどの更新は，そう頻繁には行われないことを期待できる．この点に注目し，一度転送されたインタプリタのプログラムコードは，そのホストのローカルディスクにキャッシュを行う．その結果，同じ言語，同じバージョンのインタプリタ用に記述されたモーバイルエージェントが移動する限りでは，インタプリタのプログラムコードのネットワークを介した転送は行われず，ネットワーク転送量の削減が可能となる．本技法により，モーバイルエージェントが記述されているプログラミング言語に対応したインタプリタがそのホストに未インストールの場合，もしくはバージョンアップが行われた場合のみに，インタプリタのプログラムコードの転送が行われる．

5. 実現例・実験

5.1 実現例：スクリプト言語 Tcl のモーバイル化

提案方式の実現例として，スクリプト言語 Tcl[6] に

おけるモーバイル機能の実現について述べる．提案方式の最も実装コストを少なくする場合の例として，スクリプト言語 Tcl のインタプリタ全体をモーバイル・メモリセグメントに配置する（**図 5** 参照）．インタプリタの移動に必要な処理を**図 6** に示す．この図で示されている各処理の中で，ハッチングされている部分はモーバイル・メモリセグメント機構により自動的に行われる．つまり，各プログラミング言語にモーバイル機能を導入するためにシステムプログラマが記述しなければならない処理は，ハッチングされていない部分のみである．

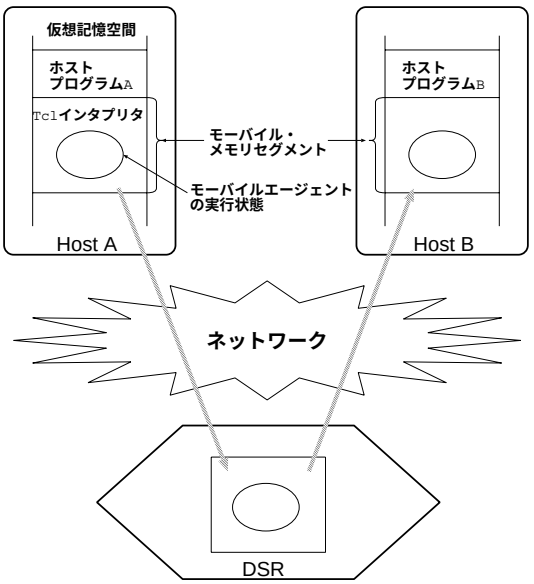

図 5 モーバイル・メモリセグメントによる Tcl インタプリタの移動
Fig. 5 Migration of the Tcl interpreter with mobile memory segment.

本実現では，モーバイル・メモリセグメントに配置するモーバイル Tcl インタプリタと，そのモーバイル Tcl インタプリタを DSR との間でロード/アンロードするホストプログラムを作成する．ホストプログラムとは，モーバイルエージェントが実行されるホストで待機し，モーバイル Tcl インタプリタを DSR との間でロード/アンロードするプログラムである．ホストプログラムは，PLANET システムのプリミティブを使って，ホストプログラムの仮想記憶空間と DSR の間でモーバイル・メモリセグメントをロード/アンロードする処理を記述する．

モーバイル Tcl インタプリタを作成するためには，2 つの作業が必要となる．1 つは，Tcl インタプリタへのモーバイルエージェント記述のためのプリミティブを追加することである．インタプリタへ追加したプリミティブは，モーバイルエージェントが移動するためのプリミティブ agent_move である．このプリミティブはモーバイルエージェント自身の移動をホストプログラムに依頼する．もう 1 つの作業は，インタプリタ内で動的にメモリ確保を行っている部分を，モーバイル・メモリセグメントに確保するように変更すること

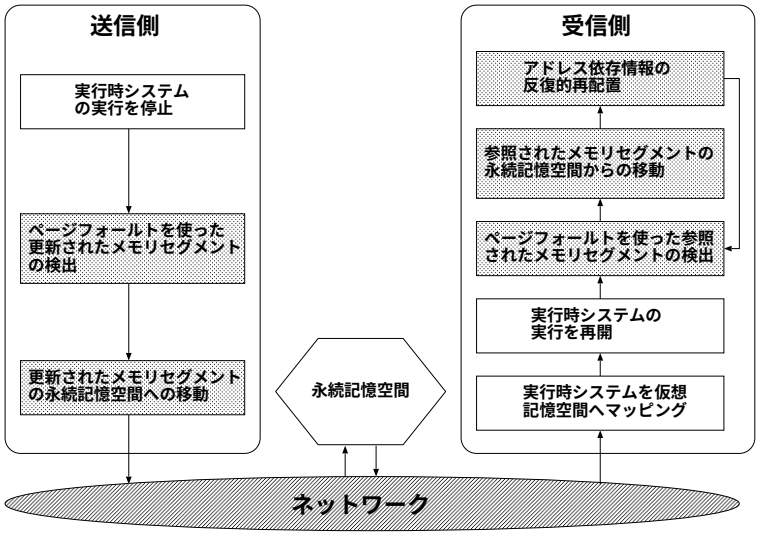

図 6 非抽出型実現方式を用いたモーバイルエージェントの移動（ハッチングされている部分はモーバイル・メモリセグメント機能により，自動的に行われる）
Fig. 6 Migration of the mobile agent using the non-extraction method (Process that is hatched will be automatically processed).

である．具体的には，インタプリタのソースコード内で，malloc() を呼び出している部分を，モーバイル・メモリセグメント機構が提供する pmalloc() を呼び出すように変更する．pmalloc() は，第 1 引数で指定されたサイズのメモリ領域を，モーバイル・メモリセグメント内に確保する．また，第 2 引数以下で，確保したメモリセグメント内のアドレス空間依存情報の位置をオフセットで指定する．

```
void *pmalloc(size_t size, /* offset */ ...)
```

ここで，変更の具体例を示す．まず，対象プログラミング言語のソースコードから malloc を呼び出している場所を探し，そこで確保されるメモリ領域に格納されるデータの型を特定する．たとえば，ソースコード内に

```
ptr = (cell *)malloc(sizeof(struct cell));
```

という部分があれば，確保したメモリ領域に格納されるデータの型（この場合は cell）の型宣言を探し，

```
struct cell {
int a;
char b;
int *c;
struct cell *next;
};
```

もし以上のような宣言がしてあれば，最初の malloc の文を以下のように変更し，格納されるデータ内のアドレス依存情報（ポインタ）の位置をオフセット値で指定する．

```
ptr = (cell *)pmalloc(sizeof(struct cell)
       ,offsetof(cell,c)
       ,offsetof(cell,next));
```

ここで，offsetof は構造体のメンバのオフセット値を返す関数であり，また pmalloc は任意個のオフセット値を示す引数をとることができる．

上記の方法に基づき，既存の Tcl インタプリタの C 言語のソースプログラム（バージョン 7.6p2）に対して変更，追加を行った．その変更は約 56,000 行のソースプログラム中約 1,600 行であり，追加は約 600 行であった．しかし，そのほとんどは動的にメモリ領域を確保する場合にモーバイル・メモリセグメント内に確保するために変更，追加されたものである．この作業は，インタプリタのソースプログラム中で動的にメモリ確保を行っている部分について，確保した領域をどの型として使用するかを調べ，その型に基づいてアドレス依存情報の位置を指定する作業である．これは機械的な作業であり，インタプリタ内部での処理に関する深い知識は必要としない．また，ホストプログラムは単純な機能のみを実現すればよいため，C 言語で約 100 行のプログラムに PLANET システムが提供しているライブラリをリンクしたものである．

5.2 実　　験

提案方式では，モーバイルエージェントの移動時にプログラミング言語の実行時システムを転送する．そのため，従来のモーバイルエージェントシステムで多く用いられているシリアリゼーション技術によるモーバイルエージェントの移動と比較すると，任意のモーバイル化されたプログラミング言語で記述されたモーバイルエージェントが移動可能となる反面，移動時のネットワーク転送量は当然増加する．提案方式では，リモートメモリマップ機構や，インタプリタのプログラムコード部分のキャッシングで，ネットワーク転送量の増加を抑えているが，最終的な移動時のパフォーマンスが実用的な範囲内に収まっていることが重要である．

提案方式の実現例と同じくスクリプト言語 Tcl を用いてモーバイルエージェントを記述可能な D'Agents と，モーバイルエージェントの移動時のネットワーク転送量および，所要時間を比較した．また，提案方式でインタプリタのプログラムコード部分のキャッシングを行った場合と，行わない場合の比較も行った．実験では移動するモーバイルエージェントの例として，移動した先のホストにあるファイルを持ち帰り，そのファイルの内容を表示するモーバイル cat プログラムを作成した．また，提案方式の場合は，モーバイルエージェントは移動元，移動先のホストとは別の DSR サーバを経由して移動する．モーバイルエージェントの移動時のモデルを近づけるために，D'Agents の場合にも別のホストを経由してモーバイルエージェントが移動するようにした．本実験を行った環境を**図 7** に示

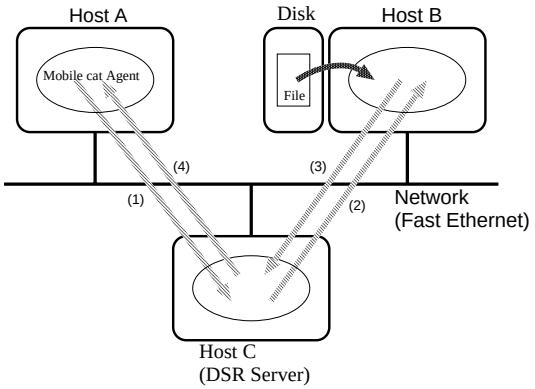

図 7 実験環境
Fig. 7 Environments of the experiment.

す．計算機は転送元ホスト，転送先ホスト，経由ホストとも，Sun Ultra60（CPU: UltraSPARCII 360 MHz, Memory: 640 MBytes，OS: Solaris 2.6）である．各計算機は Fast Ethernet（100 Mbps）で接続されている．本実験で計測したのは，モーバイル cat プログラムが転送元ホストから転送先ホストに移動し，指定した大きさのテキストファイルを持って帰り，表示を行うまでの総ネットワーク転送量および，所要時間である．

本実験の結果を**図 8** および，**図 9** に示す．図 8 の横軸は，モーバイルエージェントが移動先の計算機から持ち帰るファイルの大きさである．縦軸は，モーバイルエージェント実行のために，サーバ・クライアント間で行われたネットワーク転送の量である．図 9 中の「コードキャッシュ付き」とは，実行中に変更が行われないモーバイルエージェント内のプログラムコードをあらかじめローカルディスクにキャッシュしておくことを指す．

本実験に用いたモーバイルエージェントのプログラムコードのサイズは約 464 キロバイトであるが，本実験に用いたモーバイル・メモリセグメント実装システムでは，モーバイルエージェントをディスクに格納している状態とネットワーク転送を行う状態でデータ圧縮を行っている[12] ため，モーバイルエージェント移動時に転送されるコードのサイズは，約 163 キロバイトである．本実験の総ネットワーク転送量には，プログラムコードの 2 回の転送が含まれているため，総ネットワーク転送量中のプログラムコードの転送量は，約 326 キロバイトとなる．図 8 のモーバイルメモリ・セグメントとコードキャッシュ付きモーバイル・メモリセグメントの場合を比較すると，コードキャッシュによる効果が，実際に転送されるコードサイズ以上になる結果が得られた．これは，プログラムコードをあらかじめキャッシュしておくことにより，プログラムコード転送時に必要なモーバイルメモリセグメント機構のセグメント要求メッセージなどの制御メッセージの送信回数が減ったことによるものである．

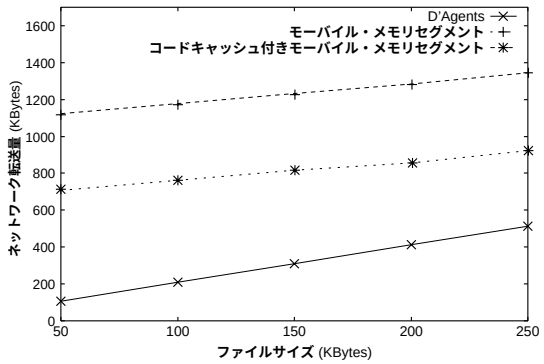

図 8 cat するファイルサイズに対するネットワーク転送量
Fig. 8 Netwark transfer size of each method.

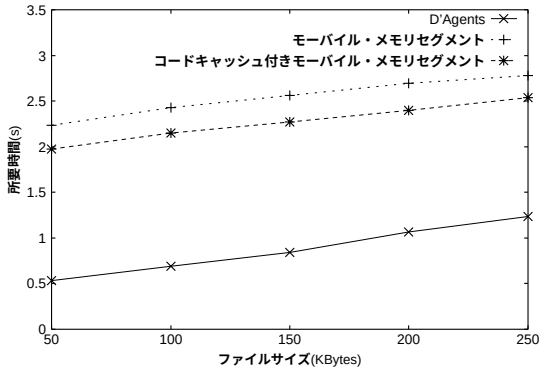

図 9 cat するファイルサイズに対する所要時間
Fig. 9 Necessary time of each method.

D'Agent とモーバイルメモリ・セグメントの結果を比較すると，50 キロバイトのファイルを持ち帰る実験の場合で，コードキャッシュを行う場合も本実現法で D'Agent の約 7 倍の転送量が必要という結果が得られた．これは，D'Agents がモーバイルエージェントのプログラムコードと，モーバイルエージェントの純粋な実行状態のみを転送しているのに対して，本システムではモーバイルエージェントのプログラムコードや実行状態を含んだ，インタプリタのデータ，スレッドを転送しているからである（コードキャッシュによりインタプリタのコードは送信されない）．

D'Agent とモーバイルメモリ・セグメントのネットワーク転送量の差が，エージェントが持ち帰るファイルサイズが大きくなるに従って減少している．これは，モーバイルメモリ・セグメント実装システムに実装されているセグメント圧縮機構において，ファイルサイズが大きくなるほど圧縮の効果が得られているためである．

コードキャッシュを行った場合と，行わない場合の所要時間を比較すると，コードキャッシュを行っても，ネットワーク転送量の削減量に比較して，所要時間の削減の割合が少ない．これは，100 Mbps Fast Ethernet のような，高速なネットワークを用いる場合，ネットワーク転送以外の処理にかかる時間の割合が大きいためである．**図 10** は，所要時間をさらに細かく分けたグラフである．A→C は，インタプリタがホスト A からホスト C にアンロードされるのに消費された時間，C→B とは，ホスト C からホスト B にインタプリタがロードされ，実行されるのに消費された時間，

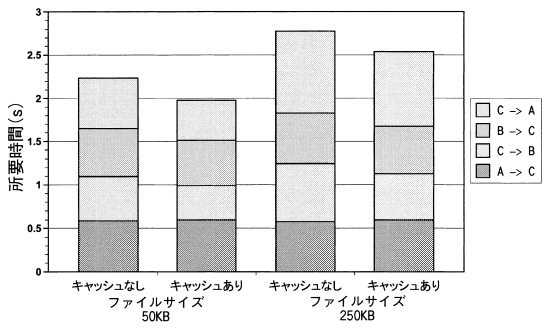

図 10 コードキャッシュの効果
Fig. 10 Effectiveness of code caching.

B→C は，ホスト B からホスト C にアンロードされるのに消費された時間，C→A はホスト C からホスト A にインタプリタがロードされ，実行されるのに消費された時間である．キャッシュを行った場合でも，プログラムコードが転送されるのは C→B と C→A のときだけであり，また C→B と C→A の時間には，モーバイルエージェントの実行時間，つまり，ディスクへのアクセス時間や，画面への出力時間が含まれており，その割合が大きい．そのため，ネットワーク転送量削減による，所要時間への効果が少なくなっている．

6. おわりに

本論文では，モーバイル・メモリセグメントを利用し，インタプリタからモーバイルエージェントの実行状態の抽出を行わずにモーバイルプログラミング言語を実現する非抽出型実現法を提案した．また，提案方式により，スクリプト言語 Tcl に実際にモーバイル機能を導入し，抽出型実現法よりも容易かつ系統的に実現可能であることを示した．

今後の課題としては以下のものがあげられる．第 1 は，提案手法により Tcl 以外のモーバイルプログラミング言語の実現を行い，提案方式が特定の言語に特化した方式でないことの実証を行うことである．

第 2 に，ガベージコレクション機能（以下 GC と略す）とモバイルセグメント機能を調和させる必要がある．提案方式で既存のプログラミング言語にモーバイル機能を導入する場合，インタプリタ内で GC を行っていても，ガーベジコレクションが行われた領域まで転送の対象となってしまい，ガーベジコレクションを行った意味がなくなってしまう．そこで，GC 機能と調和するモーバイル・メモリセグメント機能の研究を行う必要がある．

第 3 は，マルチスレッド言語への対応である．現在の提案方式では，1 つのインタプリタに対して 1 つのモーバイルエージェントが対応している．そのため，同一ホストで複数のモーバイルエージェントが同時に存在する場合には，インタプリタもモーバイルエージェントと同じ数だけ存在する．つまり，マルチスレッド機能を持った言語でも，スレッドとモーバイルエージェントを 1 対 1 に対応づけることはできない．そこで，マルチスレッド言語において，スレッド 1 つに対してモーバイルエージェント 1 つを対応づける研究を行う必要がある．

第 4 は，モーバイルエージェントが移動した際に外部資源の操作状態の一貫性を保つための系統的な手法の確立を行うことである．

参 考 文 献

1) Gray, R.S., Kotz, D., Cybenko, G. and Rus, D.: D'Agents: Security in a multiple-language, mobile-agent system, *Mobile Agents and Security*, Vigna, G. (Ed.), Lecture Notes in Computer Science, Vol.1419, pp.154–187, Springer-Verlag (1998).
2) Karnik, N.M. and Tripathi, A.R.: Design Issues in Mobile-Agent Programming Systems, *IEEE Concurrency*, Vol.July-September, pp.52–61 (1998).
3) Kato, K., Someya, Y., Matsubara, K., Toumura, K. and Abe, H.: An Approach to Mobile Software Robots for the WWW, *IEEE Trans. Knowledge and Data Engineering*, Vol.11, No.4, pp.526–548 (1999).
4) Lange, D.B. and Oshima, M.: *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley (1998).
5) ObjectSpace.: http://www.objectspace.com/.
6) Ousterhout, J.K.: *Tcl and the Tk Toolkit*, Addison Wesley Publishing Company, Reading, Massachusetts (1994).
7) Peine, H. and Stolpmann, T.: The Architecture of the Ara Platform for Mobile Agents, *Proc. 1st International Workshop on Mobile Agents MA '97*, Berlin, Germany, Kurt Rothermel, R.P.-Z. (Ed.), Lecture Notes in Computer Science, Vol.1219, Springer-Verlag (1997).
8) Thorn, T.: Programming languages for mobile code, *ACM Computing Surveys*, Vol.29, No.3, pp.213–239 (1997).
9) White, D.E.: A Comparison of Mobile Agent Migration Mechanisms, Senior Honors Thesis, Dartmouth College (1998).
10) Wong, D., Paciorek, N. and Moore, D.: Java-based mobile agents, *Comm. ACM*, Vol.42, No.3, pp.92–102 (1999).

11) 松原克弥, 加藤和彦：分散永続性を提供するモーバイルオブジェクト・システムの実現, 情報処理学会論文誌, Vol.39, No.8, pp.2494–2508 (1998).
12) 相河　享, 井久保寛明, 松原克弥, 加藤和彦：リモートメモリマップ機構と圧縮機構の統合方式について, 情報処理学会コンピュータシステムシンポジウム論文集, Vol.96, No.7, pp.155–162 (1996).

(平成 11 年 12 月 29 日受付)
(平成 12 年 3 月 28 日採録)

吉田　雅年

1976 年生．1999 年筑波大学第三学群情報学類卒業．現在，筑波大学大学院博士課程工学研究科在学中．分散システム，モーバイルエージェントに興味を持つ．

松原　克弥

1971 年生．1994 年筑波大学第三学群情報学類卒業．1996 年筑波大学大学院修士課程理工学研究科修了．同年同大学院博士課程工学研究科第 3 年次に編入学．1998 年筑波大学電子・情報工学系助手，現在に至る．分散システムの設計と実装に興味を持つ．

加藤　和彦（正会員）

1962 年生．1985 年筑波大学第三学群情報学類卒業，1987 年工学修士（筑波大学大学院工学研究科），1992 年博士（理学）（東京大学大学院理学系研究科）．1989 年東京大学理学部情報科学科助手，1993 年筑波大学電子・情報工学系講師，1996 年同助教授，現在に至る．オペレーティングシステム，プログラミング言語システム，データベースシステム，分散システム，モバイルオブジェクト計算に興味を持つ．電子情報通信学会，日本ソフトウェア科学会，ACM，IEEE 各会員．