ホスト透過型オブジェクト移送システムMogulの実現

中澤 仁[†] 望月祐洋[†] 徳田英幸[†,††]

既存アプリケーションのホスト透過的な移送を実現する，ホスト透過型オブジェクト移送モデルを提案する．本モデルは，プロセス移送や分散ウィンドウシステムにはないプラットホーム独立性を提供し，さらにオブジェクトを単位として移送の粒度を柔軟に制御できるのが特徴である．本研究でJava言語によって実現したホスト透過型オブジェクト移送システムMogulは，既存アプリケーションコードに対する変更をともなわずに，ホスト透過的なアプリケーション移送を実現する．Mogulでは，アプリケーションを移送可能オブジェクトとホスト依存オブジェクトとに動的に分離し，それらの間のメソッド呼び出しをリダイレクトする，動的メソッドリダイレクションによって透過性を確保している．本論文では，アプリケーション移送の実現手法を比較検討したうえで，Mogulの機能および特徴を示し，実装と評価について報告する．

Mogul: Location Independent Object Migration System

JIN NAKAZAWA,[†] MASAHIRO MOCHIZUKI[†] and HIDEYUKI TOKUDA[†,††]

This paper proposes a transparent object migration model which enables existing applications to migrate between hosts without depending on platforms. The key issues are transparency and reusability of existing applications to exploit application mobility for mobile computing. In this research, we have developed an object migration system named Mogul with Java. Using Mogul, existing applications can migrate transparently without changing their code. The novelty of Mogul is Dynamic Method Redirection which provides applications transparency. In this paper, at first, some conventional approaches for migration are presented with discussion of their features. Then we describe design and implementation of Mogul, and finally, usability of Mogul is discussed based on the result of evaluations.

1. はじめに

特定の動作中アプリケーションを選択して，遠隔ホストに移送するアプリケーション移送手法を用いて，ユビキタスコンピューティングや協調作業支援システムを容易に実現できる[1)]．

アプリケーション移送を実現する機構としては，プロセス移送，分散ウィンドウシステムやモバイルコードシステムなどがあげられる．しかしいずれのシステムにおいても，移送対象アプリケーションは移送後も，生成ホスト上の何らかのサービスに対する継続参照の必要性が生じる場合がある．また，複数のホスト上を連続移送する場合にも，初回移送先ホストが提供するサービスを継続して参照する必要性が生じる場合がある．このように，移送対象アプリケーションが特定のホストに依存して動作することを，アプリケーションのホスト依存性と呼ぶ[2)]．また同時に，これらのサービスをホスト依存サービスと呼ぶ．これに対して，移送を隠蔽し，アプリケーションに対して生成ホスト上の単一システムイメージを提供することを，ホスト透過性と呼ぶ[2)]．

筆者らは，既存アプリケーションコードに対する変更をともなわずに，ホスト透過的なアプリケーション移送を実現することを目的として，ホスト透過型オブジェクト移送システムMogul[3)]をJava言語[4)]を用いて開発した．Mogulでは，ウィンドウを含めたユーザインタフェース部品をホスト間移送の対象としている．また，プログラマおよび利用者に対してアプリケーションのホスト透過的な動作を保証するために，動的メソッドリダイレクション機構を構築した．本機構は，アプリケーションの移送時に，特定のホストに依存するオブジェクトを動的に分離し，移送先ホスト上でのメソッド呼び出しをリダイレクトする．

本論文では，2章でアプリケーション移送の実現手法とそれらの問題点に触れたうえで，3章でホスト透

† 慶應義塾大学政策・メディア研究科
Graduate School of Media and Governance, Keio University
†† 慶應義塾大学環境情報学部
Faculty of Environmental Information, Keio University

過型オブジェクト移送モデルを提案する．また，4章では設計と実装について述べ，Mogulを構成する4サブシステムのうち，オブジェクト移送部の実装に関して5章で詳述する．6章で評価に関して述べ，最後に7章で論旨をまとめる．

2. アプリケーション移送の実現手法と問題点

本章では，アプリケーション移送時に発生する問題点を示したうえで，各実現手法を分析する．

2.1 アプリケーション移送時の問題点

単一ホスト上で動作することを前提として構築されたアプリケーションを遠隔ホストに移送し，移送先ホスト上で継続して動作させる機構を実現する際には，以下に示す問題が生じる．

ホスト依存性 移送元ホストや各移送先ホスト上のファイルやソケットをはじめとして，特定のホスト上で提供されるサービスを参照するアプリケーションは，当該サービスが存在するホストに依存して動作する．ホスト依存性を有するアプリケーションは以下に示す各問題を誘発する．

パフォーマンスの低下 ホスト依存性を有するアプリケーションは，サービス提供ホストに対してネットワークを介した参照を必要とする．たとえば移送元ホスト上のファイル参照などがこれに該当する．したがって，当該サービスに対するローカル参照と比較してパフォーマンスが低下する．

障害発生点の増加 ホスト依存性を有するアプリケーションは，サービス提供ホストの障害あるいはネットワークの障害に際して，動作を継続できない．したがって，非移送型アプリケーションと比較して障害発生点が増加する．

プラットホーム依存性 移送対象アプリケーションが特定の計算機機種に依存したコードで提供される場合，移送先ホストは当該計算機機種に限定される．

セキュリティ 情報の改竄，機密情報の漏洩や故意の障害生成などを目的としたアプリケーションから，移送先ホストを防衛する必要がある．

これらの問題点に対して，アプリケーションコードの変更をともなわずにオペレーティングシステムやミドルウエア内で解決する手法が研究されている．これらはそれぞれプロセスマイグレーションや分散ウィンドウシステムとして実現されている．一方，アプリケーションコードを変更し，移送されることを考慮してプログラミングを行う解決法も存在する．

次節では，アプリケーション移送の各実現手法を考察する．

2.2 アプリケーション移送の実現手法

アプリケーション移送とは，**起点ホスト**から**終点ホスト**に対して動作中アプリケーションの**状態**を移動することをいう．またアプリケーション継続移送とは，終点ホストに移動した状態を基にして，アプリケーションを継続して動作させることをいう．本論文では，後者を指して単にアプリケーション移送と呼ぶ[5]．状態とは，アプリケーションの実行状態を意味し，実際にはプログラムカウンタの値，スタックおよびヒープの内容などを含む．ここでは状態を視覚状態と内部状態とに便宜的に分割し，それぞれ以下の定義を与える．

視覚状態 ユーザインタフェースに関連した状態変数の値

内部状態 上記以外の状態変数の値および当該プロセスに関連したカーネルの内部変数

上記の定義を用いて，ホスト間で移動される状態に着目することによって，アプリケーション移送を，全移送，部分移送および視覚移送とに分類できる．以下の各項では，前述した問題点を考慮し各移送手法について考察する．

2.2.1 全 移 送

全移送は，視覚状態および内部状態のすべてを終点ホストに移送する手法で，プロセス移送として実装される．プロセス移送手法は主にカーネル内に実装され，アプリケーションコードを変更することなく，動作中のアプリケーションを終点ホストに移送できることが特徴である．また，移送されたプロセスが発行するシステムコールを起点ホストにリダイレクトすることによって，ホスト依存性を解決している[2),6)]．

一方，プロセス移送は，特定のオペレーティングシステムに依存し，移送対象ホストに制約が生じる．

2.2.2 部 分 移 送

部分移送は，視覚状態，内部状態にかかわらず実行状態の一部を終点ホストに移送する手法であり，オブジェクト移送が典型的な例である．

オブジェクト移送は，移送の粒度をオブジェクトを単位として柔軟に制御できるのが特徴である．Voyager[7)]をはじめとするミドルウエア，あるいはJavaやObliq[1),8)]などのプログラミング言語で実現され，特定の計算機機種やOSに依存しないという利点がある．

オブジェクト移送では，Birrellら[9)]が示したNetwork Objectの概念（本論文では以降スタブオブジェクトと呼ぶ）を用いて，ホスト依存サービスに対する移送後の継続参照を実現している．このとき，ホスト透過性を提供するために，ホスト依存サービスを参照

するメソッド呼び出しを，起点ノードに対してリダイレクトする必要がある．

2.2.3 視覚移送

X Window System[10] をはじめとする分散ウィンドウシステムでは，アプリケーションがライブラリを用いて，任意のホスト上のウィンドウサーバが管理するディスプレイに，ユーザインタフェースを描画できる．したがって，視覚移送では，アプリケーションとウィンドウサーバの双方が視覚状態を共有しているといえる．このとき，XTV[11] や Teleporting システム[12] を用いて，ユーザインタフェースを描画するディスプレイを切り替えることによって，視覚的に移送を実現できる．

上記の各システムは，移送されるアプリケーションとは独立した機構として実装されており，移送対象アプリケーションコードの変更をともなわない．また内部状態は移送されないため，利用者はホスト透過的にアプリケーションを利用できるのが利点である．しかし，ウィンドウの再描画にともなって，ウィンドウサーバとの間で通信が必要となるため，アプリケーションの動作コストが高いという問題点がある．

2.3 移送手法に関する考察

本研究では，任意の計算機をアプリケーションの移送対象とする．既存アプリケーションコードの変更をともなわないという観点では，全移送や視覚移送が優れているが，これらは特定のオペレーティングシステムやウィンドウシステムに依存して動作する．これに対して部分移送は，Java や Obliq といった，プラットホームに対して独立に動作する言語を用いた実装が可能である．したがって，任意の計算機をアプリケーションの移送先ホストとして考慮した場合，部分移送が実現手法として適している．

2.2.2 項に示したように，部分移送はオブジェクト移送機構として実現される．既存のオブジェクト移送システム[1),8),13)] は，移送先ホスト上でファイルやソケットなどを作成する場合，アプリケーションのホスト依存性を増加させる．これに対して Java RMI[14] や HORB[15] を用いて記述したスタブオブジェクトを利用することによって，依存ホストをアプリケーションの生成ホストに制限できる．しかしこれらのシステムでは，アプリケーション構築時に移送を前提としてプログラムを記述する必要があり，単一ホスト上で動作することを前提として構築されたアプリケーションを移送対象として再利用できない．

本研究では，既存のアプリケーションをホスト間で移送することを目的としており，既存のオブジェクト移送システムと異なり，アプリケーションコードの変更をともなわずにホスト透過性を提供する機構を実現する必要がある．

3. ホスト透過型オブジェクト移送モデル

前章 2.3 節では，アプリケーション移送において，任意の計算機を移送先として考慮した場合，部分移送が実現手法として適していることを示した．同時に，既存のオブジェクト移送システムには，ホスト依存性と既存コードの再利用性の観点で問題があることについても言及した．

本章では以上の観点に基づいて，視覚移送と部分移送とを融合した，ホスト透過型オブジェクト移送モデルを提案する．

3.1 機　　能

本モデルでは，ホスト依存性のないすべてのオブジェクト（以下，移送対象オブジェクトと呼ぶ）が，ホスト間での移送機能を持つ．これに対してホスト依存性を持つオブジェクト（以下，ホスト依存オブジェクトと呼ぶ）に関しては，それ自身は移送されず，代替として当該オブジェクトのスタブオブジェクトが動的に生成され移送される．スタブオブジェクトは，移送先ホスト上での同オブジェクトに対するメソッド呼び出しを，移送元ホスト上の対応するホスト依存オブジェクトへリダイレクトするホスト透過機能を有する．これによって，移送先ホスト上の移送対象オブジェクトから，移送元ホスト上のホスト依存オブジェクトへの継続的な参照が可能となる．

以下の各項では，本モデルが提供する上記の各機能に関して詳細を述べる．

3.1.1 移送機能

本モデルでは，アプリケーションを構成する任意のユーザインタフェースオブジェクト 1 つあるいは複数のユーザインタフェースオブジェクトで構成されるグループに対して，ホスト間での移送および複製操作が可能である．アプリケーションの全ユーザインタフェースを移送するには，ウィンドウを構築するユーザインタフェースオブジェクトに対して移送操作を実行すればよい．

移送機能とは，ウィンドウを含めたユーザインタフェースオブジェクトと，ユーザインタフェースオブジェクトから参照されるその他の移送対象オブジェクトをリモートホストへ移送する機能である．このとき，移送対象オブジェクトがホスト依存オブジェクトを参照している場合には，ホスト透過機能を用いて当該オブジェクトのスタブオブジェクトを生成し移送する．

表1 移送手法の類形
Table 1 Classification of migration mechanisms.

	移送前		移送後	
	始点ホスト	終点ホスト	始点ホスト	終点ホスト
全移送	視覚/内部		–	視覚/内部
部分移送	視覚/内部	–	視覚/内部	視覚/内部
視覚移送	視覚/内部	–	視覚/内部	視覚
本モデル	視覚/内部	–	内部	視覚

また当該アプリケーションの再移送時に，移送対象オブジェクトがスタブオブジェクトを参照している場合には，スタブオブジェクト自体を移送する．これによって，再移送後も移送元ホスト上のホスト依存オブジェクトを，ホスト透過的に参照できる．ただし，当該アプリケーションを移送元ホストへ移送する（戻す）際には，再移送処理が適用される．したがって，その後のホスト依存サービスに対する参照もスタブオブジェクトを用いて行われる．

3.1.2 ホスト透過機能

本モデルではアプリケーション移送に際して，ホスト依存オブジェクトは移送せず，当該オブジェクトのスタブオブジェクトを機械的に生成し移送する．ホスト透過機能は，ホスト依存オブジェクトに対応するスタブオブジェクトを生成し，スタブオブジェクトへのメソッド呼び出しを，移送元ホスト上のホスト依存オブジェクトへリダイレクトする機能を提供する．本機能は，プロセス移送におけるシステムコールのリダイレクション[2),6),16)]をアプリケーションレベルで実現するものである．

スタブオブジェクトは，生成対象オブジェクトと同一のメソッドと変数，およびメソッドリダイレクションを実現するシステムメソッドによって構成され，アプリケーション移送時に本機能によって動的に生成される．したがって，プログラマは，単一ホストで動作することを前提とするアプリケーション記述で，移送可能アプリケーションを構築できる．また利用者は，既存のアプリケーションをホスト間で移送可能となる．

またスタブオブジェクトは，当該オブジェクトに対するコンストラクタメソッドの呼び出しを移送元ホストへリダイレクトする．これによって，移送対象オブジェクトが移送先ホスト上でソケットやファイルを新たに作成する場合でも，これらは実際には移送元ホスト上に作成される．したがって，再移送後の，移送対象アプリケーションの各移送先ホストに対する依存性を抑制できる．

3.1.3 動作ホストの保護

移送対象ホスト上のユーザ，同ホスト上で動作している他のプロセスや移送対象ホスト自身を適切に保護する必要がある．本モデルでは，ホスト透過機能によって，移送先ホスト上でのソケットあるいはファイルの生成や参照に関するメソッド呼び出しは，移送元ホストにリダイレクトされる．これによって移送先ホストのファイルシステムやネットワーク資源を保護できる．

これに加えて移送先ホストの管理者に対して，移送対象アプリケーションが利用可能なAPIを制限する機能を提供する必要がある．また，移送対象アプリケーションによって計算資源を占有されることのないよう，移送対象ホストの管理者が移送対象アプリケーションによる計算資源の最大占有率を制限できる必要がある．ただし，次章以降に示す現在の実装には，これらの2機能は含まれていない．

3.2 特　　徴

表1は，各移送手法における状態の移動をまとめたものである．表1の中で視覚移送は，移送前に始点ホストに存在したアプリケーションの視覚状態が，移送後に終点ホスト上のウィンドウサーバによって共有されることを示している．また部分移送は，実行状態の任意の一部を移動できるので，移動後には双方のホストにそれぞれの状態の一部が存在しうる．

本モデルでは，アプリケーションのすべての視覚状態をオブジェクトとして終点ホストに移送することによって，視覚的なアプリケーション移送を実現する．これによって，ユーザインタフェースの描画に際してネットワーク通信が発生する視覚移送と比較してパフォーマンスが向上する．

またホスト透過機能は，移送対象アプリケーションのホスト依存性を抑制する．これによって障害発生点を限定できるが，本モデルでは，スタブオブジェクトが移送元ホスト上のホスト依存オブジェクトに対してネットワークを介して参照するため，移送元ホストへのネットワーク到達性が移送対象アプリケーションの動作前提となる．ただし，移送対象アプリケーションがホスト依存オブジェクトを参照していない場合には上記の前提は存在しない．

本モデルの特徴は，スタブオブジェクトをアプリケーション移送時に動的に生成することである．これによって移送を前提とせずに構築されたアプリケーションを移送対象として扱える．したがって，プログラマは，高度なプログラミング知識を持たずに，ホスト透過的な移送可能アプリケーションを構築できる．また利用者は既存のアプリケーションを移送対象として扱える．

3.3 関連研究

本節では，ホスト間で移送可能なアプリケーションを構築することを目的とした先行研究について，2.1節に示した問題点の解決手法および本研究との関連を記述する．

3.3.1 Obliq

Obliq[1),8)] は，複数ユーザで共有可能な移送型アプリケーションを構築することを目的としたプログラミング言語である．Obliq には *mutable* なデータと *immutable* なデータが存在する．前者は代入によって状態が変化するため移送不可能であり，代替として *network pointer* が移送される．これに対して後者はプログラムコードなどを指し，状態が変化しないため移送可能である．*network pointer* は本研究のスタブオブジェクトと同様であり，*network pointer* への呼び出しは移送元ホストへリダイレクトされる．

しかし Obliq では，ファイルやソケットなどのホスト依存サービスについて，移送後の扱いに関してプログラマによる明示的な記述が必要である．プログラマは，移送先ホスト上でファイルやソケットを再オープンすることも可能であるとされている．したがって，移送対象アプリケーションのホスト依存性が高まる可能性がある．また，移送先ホストのファイルシステムが移送元ホストと共有されていない場合には，移送対象アプリケーションは移送先ホスト上で同一ファイルの参照を継続できない．

3.3.2 Java Serialization および RMI

Java Serialization 機構は *java.io.Serializable* インタフェースを実装しているクラスのインスタンスを，ホスト間で移送する機構を提供する．また，Java RMI 機構は，プログラミング時の型付けと，*rmic* コマンドの実行によってスタブオブジェクトを生成し，リモートメソッド呼び出しを実現できる．両者を利用して移送可能アプリケーションを構築可能であるが，複雑な型付けによって，移送対象オブジェクトとホスト依存オブジェクトをプログラミング時に定義する必要がある．したがって，これらの機構を用いて移送可能アプリケーションを構築する際には，プログラマに対する制約が高くなる．

3.3.3 Voyager

ObjectSpace Voyager は，Java を用いて実装された ORB[17)] であり，本研究と同様に単純なプログラミングで分散アプリケーションを構築することを目的としている．Voyager1.x では，プログラマは移送を意識せずにプログラミングを行い，*vcc* コマンドを用いて任意のオブジェクトの *virtual reference* を構築できる☆．*virtual reference* は，対応するオブジェクトに対してホスト透過的なアクセスを提供するスタブオブジェクトである．本スタブオブジェクトから対応するオブジェクトへメソッド呼び出しがリダイレクトされることを，*forwarding* と呼ぶ．

Voyager では，アプリケーション構築時にプログラマがスタブオブジェクトを明示的に生成する必要がある．さらに，オブジェクトの移送は，スタブオブジェクトの持つ *moveTo* メソッドを呼び出すことによって行われる．このためプログラマは，スタブオブジェクトを生成するオブジェクトを自ら決定する必要があり，プログラマに対する制約が高い．また利用者は，単一ホスト上で動作することを前提として構築されたアプリケーション中のオブジェクトを移送対象にできない．

4. Mogul

前章では，ホスト透過型オブジェクト移送モデルの機能概要と特徴を述べた．本章では，本研究で実現したホスト透過型オブジェクト移送システム Mogul の設計と実装について述べる．

4.1 システム構成と実装環境

Mogul は 3 層から構成されており，最下層にはオブジェクト移送部が位置する．ウィジェット部，ユーザマネージャ部およびコマンド部は，オブジェクト移送部の提供する機能を用いて実装されており，最上層に位置するアプリケーションプログラムをホスト間で制御するための機構を提供する．このうち本章では，全体構成を把握したうえで，オブジェクト移送部を除いた 3 サブシステムの設計と実装について述べる．オブジェクト移送部に関しては次章で詳しく述べる．

実装は JDK1.1.7 を用いて行い，Java 仮想マシンの変更およびプラットホーム依存コードの記述は行っていない．

☆ バージョン 2.0 以降では，スタブオブジェクトの生成は動的に行われるようになった．ただし，*igen* コマンドによって作成したインタフェースを用いた，移送対象オブジェクトの静的な型付けが必要であるため，完全に動的であるとはいえない．

```
public void move(Identifier id)
public void copy(Identifier id)
public void delete()
public Identifier getID()
```

図 1 ウィジェット部の API
Fig. 1 API of distributed widgets.

4.2 ウィジェット部

ウィジェット部は，Java AWT（Abstract Window Toolkit）中のユーザインタフェースオブジェクトの親クラスである *java.awt.Component* クラスに，**図 1** に示すメソッドを追加することによって実装した．これらのメソッドは，後述の各コマンド実行時に，ユーザマネージャによって起動される．またプログラマは，Java が提供する全ユーザインタフェースオブジェクトに対して，必要に応じて図 1 に示した各操作が可能である．

ウィジェット部は，下記に示す識別子を用いた名前空間において一意に識別される．

*widget://*ホスト名/ユーザ名/クラス名/番号

本識別子中でホスト名は当該ウィジェットの存在ホスト名を，ユーザ名は移送先ホスト上での当該オブジェクトを含むアプリケーションのユーザ ID を，クラス名は当該ウィジェットのクラス名を示し，番号は同一クラスから生成された複数のインスタンスを識別するために，順番に割り当てられる整数である．図 1 の中で，*move* および *copy* メソッドの引数である *id* は，ユーザマネージャの識別子であり，上記に示したウィジェット識別子のユーザ名までを指定したものである．

4.3 ユーザマネージャ部

ユーザマネージャ部は，各ユーザに対応して動作するサーバプログラムであり，ウィジェット部に対して，複製，移動，削除の各機能を提供する．ウィジェット部はインスタンス生成時に，ユーザマネージャの持つウィジェット管理テーブルへ登録され，ウィジェット識別子を得る．

ユーザマネージャは起動時に，未使用ポート上に開いたソケットを利用して，オブジェクト移送部を構築する．以降，ウィジェットの移送と，コマンド部によるユーザマネージャへのリクエストがこのポートに対して行われる．

4.4 コマンド部

コマンド部は，ユーザマネージャ部およびウィジェット部を直接操作するための機構を，**表 2** に示す 5 つのユーザコマンドとして提供する．すべてのコマンドはユーザマネージャに対する通信を行い，それぞれの操作要求を行う．

表 2 コマンド一覧
Table 2 A list of commands.

コマンド名	機能と利用例
wcreate	ウィジェットインスタンスを作成 *wcreate jp.ac.keio.sfc.ht.jin.Test*
wcopy	ウィジェットインスタンスをリモートに複製 *wcopy widget://miro.keio.ac.jp/jin/sftest/0* *moma@sfc.wide.ad.jp*
wmove	ウィジェットインスタンスをリモートに移動 *wmove widget://miro.keio.ac.jp/jin/sftest/0* *moma@sfc.wide.ad.jp*
wlist	ウィジェット管理テーブルから鍵を表示 *wlist jin@sfc.wide.ad.jp*
wdelete	指定したウィジェットインスタンスを消去 *wdelete widget://miro.wide.ad.jp/jin/sftest/0*

コマンド部は，シェルスクリプトあるいは MS-DOS 上のバッチファイルとして提供され，コマンド名の先頭を大文字にした Java プログラムとして実装されている．

5. オブジェクト移送部

オブジェクト移送部は，筆者らが開発した動的メソッドリダイレクション（Dynamic Method Redirection. 以下，DMR と呼ぶ）によって実現した．DMR は，ホスト透過型オブジェクト移送モデルにおけるホスト透過機能の実装である．本章では，**図 2** に示すサンプルコードを用いて，DMR の設計と実装について述べる．

5.1 システム構成

DMR は，オブジェクト入出力機構とメソッドリダイレクション機構から構成される．

5.1.1 オブジェクト入出力機構

DMR のオブジェクト入出力機構は，Java の提供するオブジェクト入出力ライブラリ[13]を拡張することによって実装した．本機構は主に，3 つのクラスで構成される．**図 3** は，オブジェクト入出力機構の動作アルゴリズムを UML[18] によるアクティビティ図を用いて示したものである．

WrapperOutputStream クラスは，移送可能なオブジェクトを抽出しストリームに書き込む．図 3 に示すように，移送不可能なオブジェクトに関しては，当該オブジェクトに対するメソッド呼び出しをリダイレクトするためのスタブオブジェクトによって代替する．このとき，*WrapperGenerator* が Java Reflection API[19] を用いた動的コード生成機能によってスタブオブジェクトを生成することにより，プログラマがあ

らかじめスタブクラスを記述する必要はなくなった．

WrapperInputStream は，ストリームからオブジェクトを読み込み，必要であれば読み込んだオブジェクトのバイトコードを，接続先のユーザマネージャからロードする．ただし，一度読み込んだバイトコードをキャッシュすることによって，バイトコード転送のパフォーマンスを向上している．

```
import java.awt.*;
import java.awt.event.*;
public class HelloWindow extends Frame implements
ActionListener {
  private Hello h;
  public HelloWindow(){
    Button b = new Button();
    b.addActionListener(this);
    add(b, BorderLayout.CENTER);
    h = new Hello();
  }
  public void actionPerformed(ActionEvent e){
    h.hello();
  }
}
public class Hello {
  public hello(){
    System.out.println("Hello!");
  }
}
```

図 2 サンプルコード
Fig. 2 Sample code.

5.1.2 メソッドリダイレクション機構

メソッドリダイレクション機構は，Java RMI や HORB などと同様にリモートメソッド呼び出し機構であり，前項で述べたオブジェクト入出力機構を用いて実装した．図 4 は，図 2 に示した *HelloWindow* オブジェクトを移送した際のメソッドリダイレクション手順について，各クラス間の関連を示したシーケンス図である．

図 4 では，移送された *HelloWindow* オブジェクトからのメソッド呼び出しが，始点ホスト上の *Hello* オブジェクトへリダイレクトされている．このとき，*MethodRequest* オブジェクトおよび *MethodReply* オブジェクトが，オブジェクト入出力機構を用いてユーザマネージャ間で授受される．

5.2 新規性のある機能

本研究が実現した DMR は，既存のオブジェクト移送システムにはない機能を提供している．本節では各機能の詳細について述べる．

5.2.1 オブジェクト移送機能

DMR では，オブジェクト移送時に，移送対象とし

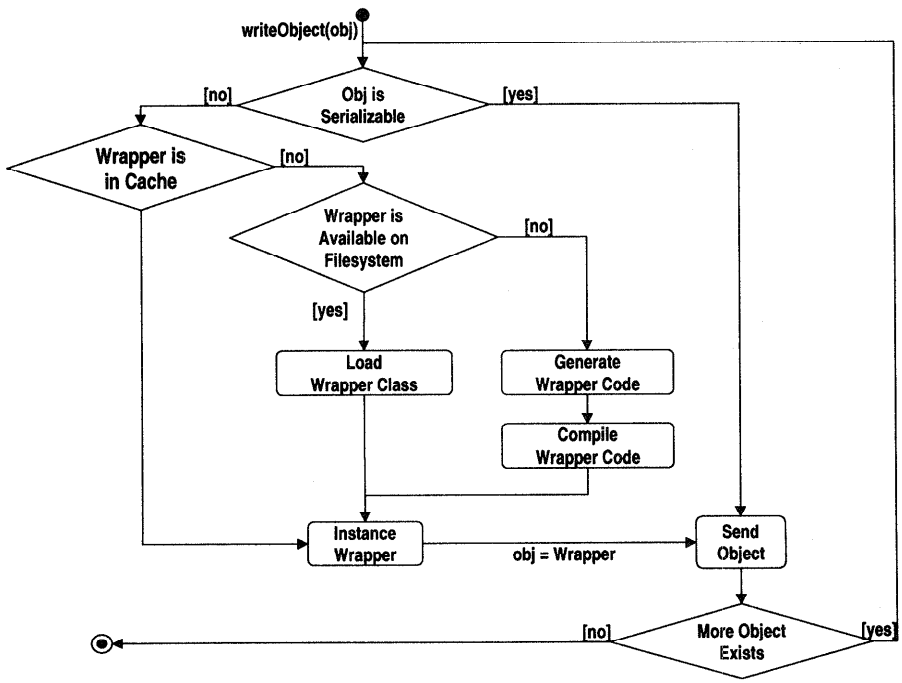


図 3 オブジェクト出力のアクティビティ図
Fig. 3 Activity flow in object output.

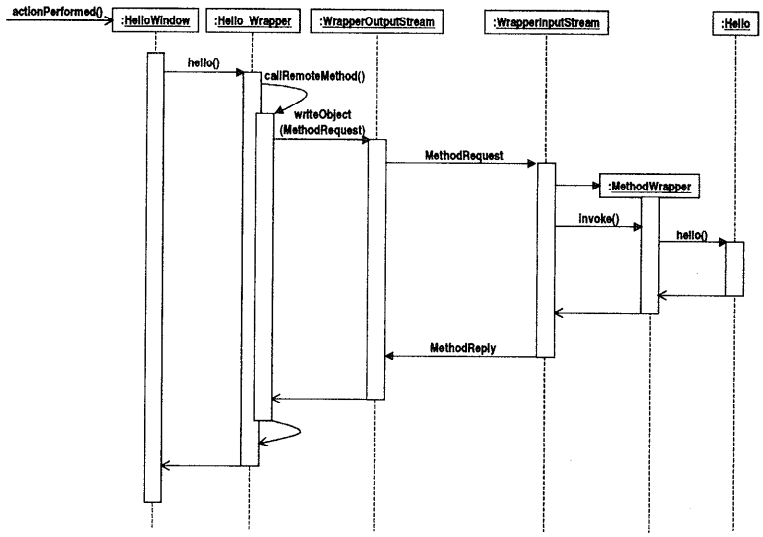


図4 メソッドリダイレクションのシーケンス図
Fig. 4 Message passing flow in method redirection.

て指定されたウィジェットを起点として，変数の参照ツリーをトレースし，移送可能なオブジェクトのみを終点ホストに移送する．トレース中に移送不可能なオブジェクトを検知した場合には，当該オブジェクトのスタブオブジェクトを移送する．このとき，*java.io.Serializable* インタフェースを実装しているオブジェクトを移送可能として扱う．

サンプルコード中の *HelloWindow* オブジェクトが移送対象オブジェクトとして指定された場合には，同オブジェクトを起点として変数の参照ツリーをトレースする．この場合，*HelloWindow* オブジェクトは移送可能であり，図2中の変数 *h* によって参照されている *Hello* オブジェクトは移送不可能である．したがって，オブジェクト移送部は，*HelloWindow* オブジェクトと *Hello* オブジェクトのスタブオブジェクトを終点ホストに移送する．

5.2.2 動的なスタブ生成機能

上記の例において *Hello* オブジェクトのスタブオブジェクトは，終点ホストに移送された *HelloWindow* オブジェクト内では *Hello* オブジェクトとして参照される．したがって，クラスAのインスタンスと，クラスAスタブのインスタンスは透過な型を持つ必要がある．DMRでは，クラスAスタブクラスをクラスAのサブクラスとし，すべてのメソッドをオーバライドすることで型の透過性を保証する．**図5**に，*WrapperGenerator* によって自動生成された *Hello* スタブクラスのソースコードの一部を示す．図5中で，*Hello* スタブクラスが実装している *DMR_Wrapper* インタフェースは，*java.io.Serializable* インタフェースを継承している．したがって，アプリケーションの再移送時には，*Hello* スタブオブジェクトがそのまま移送される．

```
(snipped)
public class Hello_Wrapper extends Hello implements
DMR_Wrapper{
(snipped)
  public void hello(){
    if(!DMR_CONNECTED)
      return;
    Class types[] = new Class[0];
    Object args[] = new Object[0];
    Object reply;
    reply=DMR_callRemoteMethod("hello", args,
    types);
(snipped)
  }
}
```

図5 スタブのサンプルコード
Fig. 5 Sample stub.

5.2.3 メソッドリダイレクション機能

終点ホストに移送された *HelloWindow* オブジェクトからの *Hello* オブジェクトに対するメソッド呼び出しは，同時に移送された *Hello* スタブを経由したリモートメソッド呼び出しによって，始点ホスト上の *Hello* オブジェクトへリダイレクトされる．

メソッドリダイレクションに際して必要となる引数

表 3 測定環境
Table 3 A platform for evaluation.

項目	マシン 1 (miro)	マシン 2 (dali)
CPU	UltraSPARC 168 MHz × 2	UltraSPARC-II 248 MHz × 2
主記憶	128 MB	512 MB
OS	Solaris 2.5.1	Solaris 2.5.1

および，リモートホストから返される返り値あるいは例外オブジェクトは，通常のオブジェクトと同様にオブジェクト入出力機構を用いて移送される．したがって，移送不可能なオブジェクトが引数として渡された場合でも，自動生成した当該オブジェクトのスタブを引数として始点ホストに移送できる．

5.3 動作時の制約

本システムでは，移送対象アプリケーションがホスト依存オブジェクトを参照する場合，同アプリケーションは移送元ホストに依存して動作する．したがって，移送元ホストの障害あるいはネットワークの障害が発生した場合には，同アプリケーションは動作を継続できない．ただし，移送対象アプリケーションがホスト依存オブジェクトを参照しない場合には，上記障害は同アプリケーションの動作に影響しない．

上記の制約以外に，現段階で未解決な動作上の以下の制約が存在する．

- 引数なしコンストラクタを定義しないホスト依存オブジェクトは，スタブオブジェクトを生成できない．
- スタティックイニシャライザおよびスタティックメソッドを持つホスト依存オブジェクトは，スタブオブジェクトを生成できない．
- 本システムと Java RMI や HORB など，他の遠隔メソッド呼び出し機構とを共存させられない．

上記各制約の解消は，今後の課題とする．

6. 評 価

本章では，他の移送手法と比較した定性評価および基本性能評価について述べ，アプリケーション移送実現手法としての Mogul の有用性を実証する．

6.1 基本性能評価評価

Mogul の提供する各機能の所要時間を測定した．測定は，**表 3** に示す 2 ホストを 100Base-T の同一セグメント内に接続して行った．

6.1.1 移送コストに関する測定

Mogul のオブジェクト移送部は，Java が提供するオブジェクト入出力ライブラリを拡張して実現しており，拡張にともなう時間的コストとして以下の 2 点があげられる．

(1) スタブの生成コスト (W)
(2) バイトコードの転送コスト (B)

拡張にともなう上記のコストに加え，各オブジェクトを終点ホストに転送するネットワークコスト N が発生する．したがって，n 個のオブジェクトから構成されるアプリケーションの移送に要する時間的コスト Cn は，上記の各コストの和として，式 (1) で定義できる．ただし，m はスタブ生成を要するオブジェクト数を，また l はバイトコード転送を要するクラス数を表すものとする．

$$Cn = \sum_{i=0}^{n} N + \sum_{j=0}^{m} W + \sum_{k=0}^{l} B \qquad (1)$$

式 (1) において，W および B に関するコストは，初回の移送時にのみ発生する．したがって，同一アプリケーションの 2 度目以降の移送コストは式 (2) によって示される．

$$C'n = \sum_{i=0}^{n} N \qquad (2)$$

表 3 に示した測定環境上では，式 (1) 中の各コストは式 (3)〜(5) の各一次式で近似できる．なお，各式は表 3 に示した測定環境上での各項目に関する実測値を，回帰分析して求めたものであり，単位はミリ秒である．

$$N = 0.12\alpha \quad (\alpha：オブジェクトサイズ) \qquad (3)$$
$$W = 27.80\beta + 1926.00 \quad (\beta：メソッド数) \qquad (4)$$
$$B = 0.02\gamma + 624.59 \quad (\gamma：バイトコードサイズ) \qquad (5)$$

図 2 に示した *HelloWindow* オブジェクトを表 3 に示した測定環境上で，ホスト *dali* からホスト *miro* に対して移送した場合の，理論値と実測値の関係を**表 4** に示した．なお，測定値は 100 回の試行結果の平均値である．

HelloWindow オブジェクトの 2 回目以降の移送コストは，ネットワークコストのみである．したがって，表 4 中の初回移送コストのほとんどは，スタブの生成コストとバイトコードの転送コストで占められている．このうちスタブの生成コストは，スタブオブジェクトのコード生成およびコンパイルに要する時間を含

表 4 移送コストに関する測定
Table 4 A result of evaluation on migrating *HelloWindow*.

理論値（初回）	理論値（2 回目以降）	測定値（初回）	測定値（2 回目以降）
4.15 秒	0.24 秒	4.18 秒	0.28 秒

表 5 RMI と DMR のパフォーマンス比較
Table 5 Comparison of performance between RMI and DMR.

引数の数	0	1	2	3	4	5
RMI	182 ms	228 ms	228 ms	241 ms	251 ms	251 ms
DMR	429 ms	483 ms	519 ms	524 ms	634 ms	633 ms

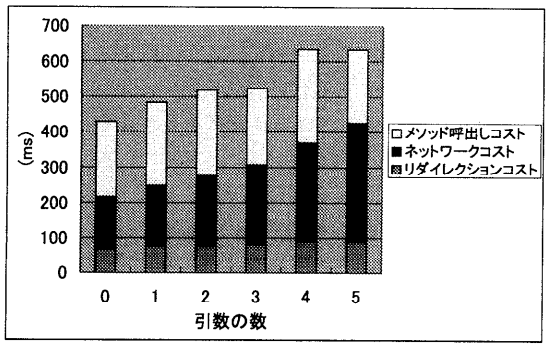


図 6 メソッドリダイレクション所要時間
Fig. 6 A result of evaluation on method redirection.

んでおり，実際の測定では 2.1 秒を要した．以上の結果から，移送対象アプリケーション起動時のスタブクラス生成，およびプログラマによる明示的なスタブクラスの生成が，パフォーマンスの向上に有用であるといえる．

6.1.2 メソッドリダイレクションに関する測定

メソッドリダイレクションが Mogul の性能面に与える影響を把握することを目的として，引数の数とメソッドリダイレクションに要する時間的コストとの関連を測定した．

表 5 は，引数の数が異なる 6 つのメソッドについて，DMR と Java RMI によるリモートメソッド呼び出しの所要時間を示したものである．表 5 より，DMR によるメソッド呼び出しが Java RMI の 3 倍の時間を要しており，実装の最適化の必要性があるといえる．また，**図 6** は，表 5 に示した DMR のパフォーマンスの内訳を示したものである．このうち，ネットワークコストは引数を含んだメソッド呼び出し要求オブジェクトを移送元ホストへ送信するのに要する時間を示す．またメソッド呼び出しコストは，移送元ホスト上で実際にメソッドを呼び出すのに要する時間である．

6.2 既存のオブジェクト移送手法との比較

Mogul や 3.3 節に示した各システムや言語を用いて，分散アプリケーションを構築できる．たとえば，回覧板システム☆の構築では，Mogul や Voyager を用いることによって，回覧板ウィンドウのホスト間移送機構の実装に関するプログラマの負担を取り除ける．

このとき，移送された回覧板ウィンドウから，移送元ホスト上のファイルを参照する場合，各システムや言語によってスタブオブジェクトを生成する必要がある．本システムでは 5.2.2 項で述べたように，スタブオブジェクトを当該オブジェクトの移送時に動的に生成する．このためプログラマはアプリケーション構築時にスタブオブジェクトのクラスを明示的に生成する必要がない．また利用者は，単一ホスト上で動作することを前提として構築されたアプリケーションを移送可能となる．これに対して，3.3 節に示した各システムや言語では，Java RMI では型付けと *rmic* コマンドの実行が必要であり，Voyager では *vcc* コマンドの実行が必要である．たとえば RMI では，**図 7** に示すクラス階層をプログラマが事前に構築する必要があるが，DMR では**図 8** に示すクラス階層が移送時に動的に構築される．

一方，現在の Mogul の実装では，アクセス制御や移送先ホストの保護をはじめとするセキュリティ機構や，移送対象オブジェクトの柔軟な制御機構などを提供していない．これに対して Obliq では，移送対象アプリケーションが実行可能なコードに制約を設けることによる移送対象ホストの保護が可能である．また Voyager および Obliq は双方とも，エージェント指向アプリケーションを構築するためのフレームワークを提供している．Mogul に対するこれらの機能追加は今後の課題とする．

☆ 視覚的実体を持つ何らかの情報が複数の利用者間を順番に移動し，最終的に情報発信者の元へ返却されるシステムを指す．

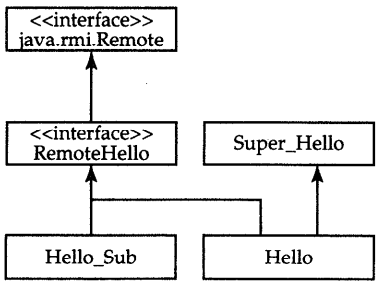


図 7 RMI のクラス階層
Fig. 7 Class hierarchy in RMI.

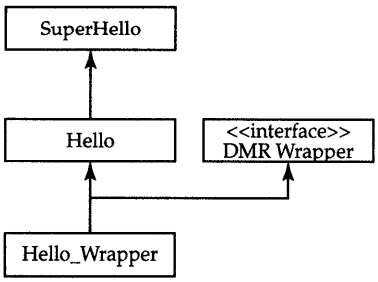


図 8 DMR のクラス階層
Fig. 8 Class hierarchy in DMR.

7. おわりに

本論文では，ユーザインタフェースをともなったアプリケーションのホスト間移送の実現手法を検討し，視覚移送と部分移送とを融合した，ホスト透過型オブジェクト移送モデルを提案した．

本研究で実現したホスト透過型オブジェクト移送システム Mogul は，移送対象アプリケーションを移送可能オブジェクトとホスト依存オブジェクトとに動的に分離する．このため，プログラマがあらかじめホスト間移送を考慮して，アプリケーションを記述する必要はない．また動的メソッドリダイレクションによって，移送対象アプリケーションに対してホスト透過性を提供できる．

Mogul は Java 言語を用いて実装することにより，移送対象アプリケーションのプラットホームに依存しない動作を実現した．また Java RMI や Voyager など，Java を用いた分散プログラミング機構と比較して，既存のアプリケーションコードを変更する必要がない点が Mogul の特徴である．

今後，基本性能の改善を目的として実装の最適化を行い，分散協調作業支援システムや，ネットワーク接続ホストの遠隔モニタリングシステムなど，様々なアプリケーションに対する応用を進めていく．

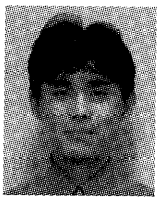
謝辞 本研究を進めるにあたって，ご協力いただいた慶応義塾大学徳田研究会の方々，ならびに R3 プロジェクトの方々に深く感謝いたします．

参考文献

1) Bharat, K. and Cardelli, L.: Migratory Applications, *ACM Symposium on User Interfaces Software and Technology* (1995).
2) Douglis, F. and Ousterhout, J.: Transparent Process Migration: Design Alternatives and the Sprite Implementation, *Software Practice and Experience*, Vol.21, No.8, pp.757–785 (1991).
3) 中澤 仁，望月祐洋，徳田英幸：Mogul：位置透過型分散共有ツールキットライブラリ，電子情報通信学会技術研究報告，Vol.98, No.40, pp.17–24 (1998).
4) Sun Microsystems Inc.: The JAVA Language Overview (1995).
5) Carzaniga, A., Picco, G.P. and Vigna, G.: Designing Distributed Applications with Mobile Code Paradigms, *International Conference on Software Engineering*, pp.22–32 (1997).
6) Douglis, F. and Ousterhout, J.: Process migration in the Sprite operating system, *International Conference on Distributed Computing Systems*, pp.18–25 (1987).
7) Glass, G.: ObjectSpace Voyager: the agent ORB for Java, *2nd International Conference on WorldWide Computing and its Applications*, pp.38–55 (1998).
8) Cardelli, L.: *A language with distributed scope*, MIT Press, Vol.8, No.1, pp.27–59 (1995).
9) Birrell, A., Nelson, G., Owicki, S. and Wobber, E.: *Network Objects*, pp.217–230 (1993).
10) X Consortium: *X consortium welcome document.*
11) Abdel-Wahab, H. and Feit, M.: XTV: A Framework for Sharing X Window CLients in Remote Synchronous Collaboration, *Proc. IEEE TriComm '91*, pp.159–167 (1991).
12) Richardson, T., Bennett, F., Mapp, G. and Hopper, A.: Teleporting in an X window system environment, *IEEE Personal Communications*, Vol.1, No.3, pp.6–12 (1994).
13) Sun Microsystems Inc.: *Object Serialization Specification* (1996).
14) Sun Microsystems Inc.: *Remote Method Invocation Specification* (1996).
15) 竹内かほり，平野 聡：HORB Better for Your Health, http://ring.etl.go.jp/openlab/horb/ (1996).
16) Barak, A., Láadan, O. and Braverman, A.:

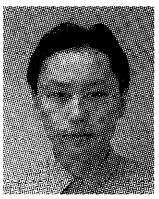
The NOW MOSIX and its Preemptive Process Migration Scheme, *Bulletin of the IEEE Technical Committee on Operating Systems and Application Environments*, Vol.7, No.2, pp.5–11 (1995).
17) Object Management Group: *Common Object Request Broker Architecture and Specification* (1996).
18) Booch, G., Rumbaugh, J. and Jacobson, I.: *Unified Modeling Language Semantics and Notation Guide 1.0* (1997).
19) Sun Microsystems Inc.: *Java Core Reflection API and Specification* (1996).

(平成 10 年 12 月 1 日受付)
(平成 11 年 4 月 1 日採録)



中澤 仁

1998 年慶應義塾大学総合政策学部卒業．同年，同大学院政策・メディア研究科修士課程進学．現在，同修士課程在学中．分散オブジェクト指向システム，オブジェクト移送ミドルウエア等の研究に従事．



望月 祐洋

1994 年慶應義塾大学総合政策学部卒業．1996 年同大学院政策・メディア研究科修士課程修了．現在，同大学院政策・メディア研究科後期博士課程に在学中．分散オブジェクト指向システム，適応支援ミドルウエア等の研究に従事．

徳田 英幸（正会員）

慶應義塾大学より工学修士．カナダ，ウォータールー大学より Ph.D.（Computer Science）．現在，慶應義塾大学常任理事，同大学環境情報学部教授，同 SFC 研究所長．分散リアルタイムシステム，マルチメディアシステム，通信プロトコル，超並列・超分散システム，モバイルシステム等の研究に従事．IEEE，ACM，日本ソフトウェア科学会各会員．