1997 年度　　修士論文

Java 言語環境における
スレッド 移送

指導　　村岡 洋一 教授

提出日　　1998 年 2 月 6 日

早稲田大学 大学院理工学研究科
情報科学専攻
学籍番号　　696P023–1

首藤　一幸

概要

Java 言語環境上のスレッド移送を実現した。従来のプロセス移送とは異なり、OS サポートを必要とせず、異機種間の移動が可能である。また、既存の移動エージェントにはローカルに動作するプログラムと異なる記述作法が要求されたが、実行コンテクストを含めての移動によって特殊な作法を不要とした。他の応用として、スレッド移送で負荷分散を図る、スレッド外部化で耐故障性の向上を図るなどが考えられる。本稿では、スレッド移送および要素技術であるスレッド外部化、Object Marshaling、リフレクションの手法および実装、移送システムの構成、各種応用を述べる。性能評価では、移動エージェントや ORB など他の分散オブジェクトシステムと比較した。本システムの、低い移送遅延、高い転送性能が示された。

目次

図目次

表 目 次

Chapter 1

序論

Java^{TM}[7] 言語環境上のスレッド移送を実現した。従来のプロセス移送とは異なり、OS のサポートを必要とせず、異なるプラットフォーム間でも移動が可能である。また、従来 Java で実現された移動エージェントではオブジェクトの移動、メッセージ送信で移動を実現しているために、移動エージェントの記述に通常のプログラムとは異なる作法が必要だが、スレッド移送の利用で特殊な作法は不要となる。他にも、要素技術、スレッドの外部化によってプログラムの実行状態を保存し耐故障性を向上させる、スレッドを移送して負荷分散を図るなど、様々な応用が考えられる。

1.1 研究の背景

Java^{TM}は 1995 年に Sun Microsystems(Sun) 社が発表した言語環境で、その中心は言語仕様 [7] と、仮想計算機 Java Virtual Machine(JVM) の仕様 [11] である。その最大の特徴のひとつは、JVM で実行される JVM バイトコードがプラットフォーム非依存なことである。このプラットフォーム非依存性に着目しての、Java 言語環境をベースとした分散システム構築基盤、分散オブジェクト技術の研究、開発が盛んである。

分散オブジェクト技術の主流は、ネットワーク上に分散したオブジェクト間のメッセージ送受信を透過的にする [5]Object Request Broker(ORB) であり、Java 用の ORB は急速に充実してきている。

Java 用 ORB のひとつの方向は、Object Management Group(OMG) によって標準化されている Common Object Request Broker Architecture(CORBA)[5] の Java language mapping である。Sun 社自身が Java から CORBA オブジェクトへのアクセスを可能とする JavaIDL[9] を提供している。

もうひとつの方向は、CORBA のような言語非依存性を捨て、代わりに柔軟性やより多くの機能を提供する Java 専用 ORB である。Sun が提供し標準 API に含まれる Remote

Method Invocation(RMI)[13]、他の ORB に先駆けて登場した HORB[8][21]、移動エージェント記述ライブラリでもある ObjectSpace 社の Voyager[17] など、すでにいくつかのソフトウェアが入手可能である。これらは対象を Java に絞ることで、オブジェクトの遠隔生成、pass by value のメッセージ送信、オブジェクトの永続化、Interface Definition Language(IDL) の記述が不要、など、利用し易さ、機能の多さを実現している。

オブジェクトが分散した環境で、ORB によってメッセージはネットワークを越える。メッセージだけでなく、計算主体であるオブジェクト、エージェントがネットワークを越える計算モデル、システムの研究開発もなされている。

オブジェクトの移動を実現する基盤に Java は適している。オブジェクトが移動するということは必然的に、その挙動を決定するクラス定義が移動先で必要となる。だが、クラス定義をあらかじめ移動先マシンで利用可能にしておくことは手間がかかるし、透過的な移動とは言えない。Java では一旦生成されたクラス定義はマシン非依存なので、転送して別マシンで利用することが容易である。

Java で実現された移動エージェントがすでにいくつか公開されている。Aglets Workbench[2]、Kafka[10]、Voyager[17]、Odyssey[1]、どのシステムでも計算主体の移動を記述可能である。

これら既存システムにおいては、オブジェクトを移動させ、そのオブジェクトに対してメッセージ送信することで、計算主体の移動を実現している。移動後にユーザまたはシステム指定のメソッドが起動される。移動において実行は継続されない。そのため、プログラマはローカルに動作するプログラムと移動するプログラムとで異なる作法を要求される。また、既存システムにおける移動法は、Java の標準 API の組合せで容易に実現できる。

1.2 研究の概要

本研究では、Java 言語環境にてスレッド移送 (thread migration) を実現した。移送先での実行の継続 (continuation) が可能である。

Sun 社の Java Development Kit(JDK) 上で利用可能なスレッド移送システム MOBA—Mobile Agent Facilities on Java Language Envirionment—を提供している。このライブラリを利用することで、JVM 上のスレッドを、実行中の状態を保ったまま JVM 間で移送することができる。

移動エージェントの記述に応用すると、移動エージェントと移動しないコードの差は、移動のためのメソッド呼び出しのみとできる。シンプルで透過的なので、移動エージェントのプログラミングインタフェースとして優れている。

また、Java のプラットフォーム非依存性により、従来のプロセス移送 (process migration) とは異なり、スレッドの移送先がプロセッサ、OS の種類に制限されない。異なるプラットフォーム間で移送が可能である。

スレッド移送およびその要素技術は、移動エージェントの記述以外にも次の応用が考えられる。

- プログラムの耐故障性向上
  移送の基礎技術となっているスレッドの外部化によって、スレッドをファイルやデータベースに保存できる。
- 負荷分散従来のプロセス移送の主目的。
- 遠隔メッセージ送信のエミュレーション

1.3 本論文の構成

本論文は以下の各章から構成される。

第 2 章 スレッド移送

なぜ移送対象がスレッドなのか明らかにし、オブジェクト指向言語の計算モデルにおいてスレッド移送の持つ意味を考察する。また、プログラミング言語におけるマルチスレッドサポートを概観し、Java 言語のマルチスレッドサポートの特徴を述べる。

第 3 章 手法と実装

スレッド移送とその各要素技術について、問題、目標とする手法、採用した現手法、そして実装を述べる。

第 4 章 システム

スレッド移送システムの特徴、スレッドの場である place、2 つのサブシステム、そして本システムのプログラミングインタフェースを説明する。

第 5 章 応用

スレッド移送の主な応用である、移動エージェント、耐故障性向上、負荷分散などについて、適用することの利点、各応用のためのさらなる研究課題を述べる。

第 6 章 実験結果

他の移動エージェント、ORB を比較対象として移送のオーバーヘッド、転送性能を測定、本システムを評価する。

第 7 章 結論

本論文をまとめ、さらなる研究課題を述べる。

Chapter 2

スレッド移送

従来、移送の対象とされてきたのは、OS 上のプロセスである。まず、本研究ではなぜ移送対象をスレッドとしているのかを明らかにする。また、プロセス移送とスレッド移送の相違を論じる。

続いて、オブジェクト間のメッセージ送信によって計算が進んでいくという、オブジェクト指向言語の計算モデルにおいてスレッドの移送がどういった意味を持つのかを考察する。

2.1 移送対象

2.1.1 なぜプロセスだったのか

過去、OS 上の計算主体であるプロセスを計算機間で移送させるプロセス移送について、その実現法、高速化手法、非均質環境への対応法、負荷分散への応用などが研究されてきた [6]。

従来研究にて移送対象がプロセスであったのはなぜか。第一の理由として、スレッドライブラリを持つ OS やマルチスレッドプログラミングという概念がまだ一般的ではなく、計算主体といえばプロセスを指していたことが考えられる。

また、移送対象がプロセスであることは技術的に都合がよい。他の主体との関係、依存が多い程、移送は困難となるところ、プロセスが保持する情報、管理する資源がプロセス間で共有されることはほとんどないからである。表 2.1に挙げるように、プロセスは一般的に、仮想記憶、ファイル記述子、プログラムカウンタなどの実行コンテクスト、環境変数やカレントディレクトリなどの実行環境を状態として持つ [6]。これらはプロセスごとに管理され、プロセス間で共有されることはほとんどない。逆にプロセス間で資源、情報を共有するためには、OS が提供するプロセス間通信の機能を明示的に利用する必要があり、これを利用するプログラムは稀である。

virtual memory– code, stack and data segments of a process,

open files– file identifier, displacement pointers, I/O buffers, etc.,

message buffers– sender and receiver information and the contents of the messages,

machine state– current condition codes, PSW, PC, SP, registers, etc.,

environment data– process ID, user ID, current working directory, references to children and parent processes, etc.

Table 2.1: Typical process states

2.1.2 なぜスレッドなのか

本研究の対象が従来研究と同様プロセスではなく、スレッドなのはなぜか。

スレッドと言えば、一般には OS 上のプロセス中の実行コンテクストである。同一プロセス中のスレッド群は、プロセスが保持する、仮想記憶、ファイル記述子などの資源を共有する。

ところが、Java の言語仕様および仮想計算機 Java Virtual Machine(JVM) 上には、計算主体としてプロセスという概念はなく、あるのはスレッドのみである。ユーザのプログラムが起動されると、JVM 上にスレッドがひとつ生成され実行を始める。プログラムは、明示的に指示することで、別のスレッドを生成できる。

スレッドグループという、スレッドおよびスレッドグループを束ねてスレッドの集合を表現する単位はある [7]。しかし、JVM 上のスレッドは、特に設定しない限り、同一 JVM 上の他スレッドへのアクセス、例えば一時停止、再開、停止などの操作、名前、優先度、属性などの取得などを全く制限されていない。アクセス対象スレッドが自身とは別のスレッドグループに属していても、である。スレッドグループは、処理の粒度としてはプロセスに近いものの、プロセスのようにそれを単位として資源へのアクセスが強く制限される枠ではなく、スレッド群を管理するために束ねる単位という性格が強い。

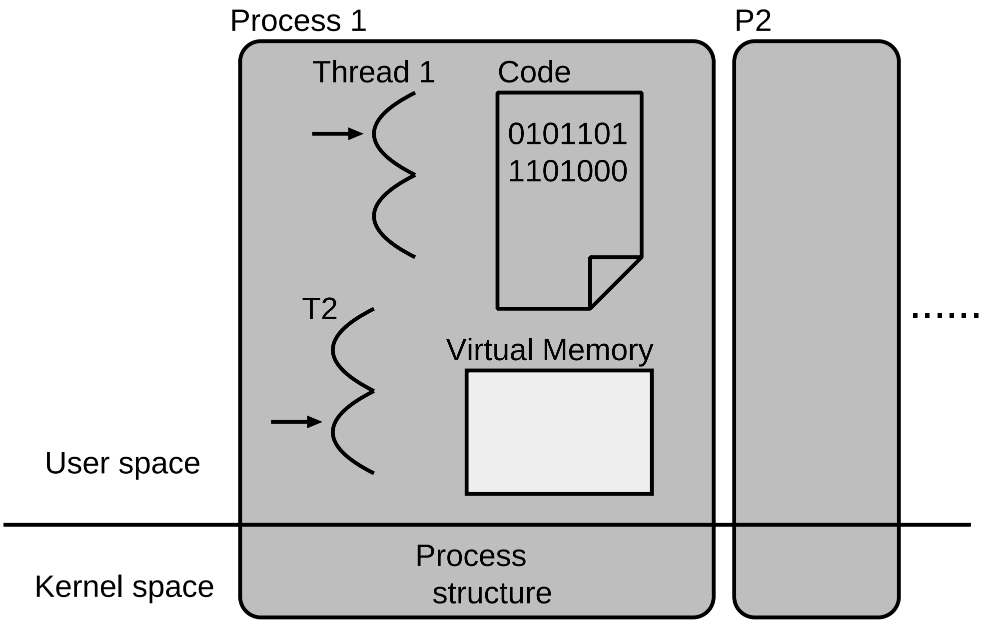

Figure 2.1: OS 上のプロセスのモデル

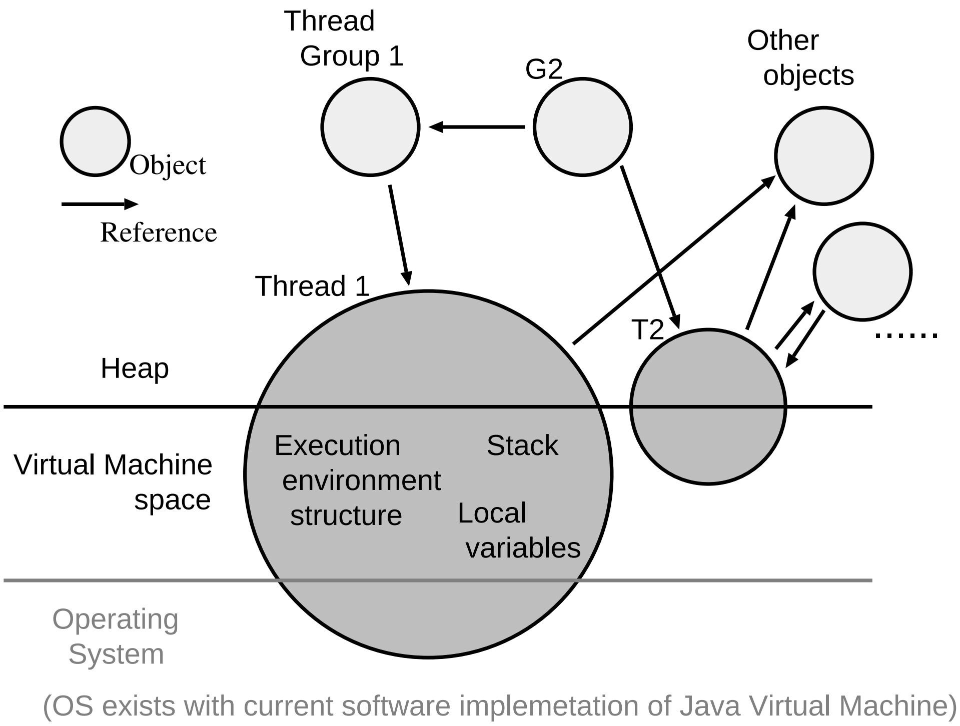

Figure 2.2: Java Virtual Machine 上のスレッドのモデル

2.2　プロセス移送との違い

2.2.1　資源の共有

JVM 上のスレッドでプロセスの仮想記憶に相当するのは、スレッドから参照を辿って到達し得るオブジェクト群である。これらヒープ上のオブジェクトは、特定スレッドに属するわけではなく、同一 JVM 上の任意のスレッドから参照され得る。つまり、JVM 上のスレッドは、プロセスのように明確に分離された仮想記憶を持たなず、仮想記憶に相当するオブジェクトはスレッド群が共有している (図 2.1, 図 2.2)。また、ファイル記述子もオブジェクトであるため、複数スレッドで共有され得る。

このように、JVM 上のスレッドは互いに共有する資源、特にオブジェクトが多いので、移送対象としてあまり都合がよくない。あるスレッドを移送する際に他のスレッドと共有し得る資源、特にスレッドから辿り得るオブジェクト群をどう扱うかを考える必要がある。

2.2.2　資源の量

反面、転送が必要な記憶の総量は、OS 上のプロセスより JVM 上のスレッドの方が少ないと考えられる。

プロセスの仮想記憶はほとんどの場合、ひとつあたり数キロバイトから数十キロバイトのページ単位で管理される。ひとつのページ中で、有効な部分と不要な部分を見分けることはほとんど不可能なので、プロセス移送ではページ単位で転送を行う。dirty なページのみ転送すればよいとはいえ、ページ中の 1 バイトにでも書き込みがなされればそのページは dirty である。実は移送先で不要な部分もまとめてページ単位で転送される。特に、記憶へのアクセスパターンが疎なプロセスにおいて、この無駄は顕著となる。

JVM 上のスレッドで記憶にあたるのはスレッドから辿り得るオブジェクト群なので、スレッド移送にあたって転送するのはオブジェクトである。オブジェクト単位で転送することになるので、ページ単位の転送のような無駄な転送は起きにくい。

もちろん JVM においても巨大な配列を疎に利用することはあるし、プロセスにおいて有効部分の少ないページには圧縮が効き易いだろう。圧縮の処理コストを払っても割に合う可能性がある。ページの圧縮まで視野に入れると、見積もり、比較結果は変わってくるかもしれない。

2.2.3　計算主体の数

プロセスの場合はひとつを移送させれば充分な場合がほとんどなのに対して、スレッドはひとつの目的のために複数動くことが一般的なので、複数スレッドの移送を考える必要

がある。Java では入出力の多重化のためにはスレッドの生成が必要であるなど、ひとつの目的のために複数のスレッドを生成することが一般的なプログラミングスタイルである。

2.3 オブジェクト指向言語における意味

オブジェクト指向言語では、オブジェクトからオブジェクトにメッセージが送られ、レシーバのクラス、セレクタに応じたメソッドが探索、呼び出されてメッセージが処理される。この計算モデルにおいて、スレッドの移送とはどういった意味を持つのか。

2.3.1 スレッド

スレッドとは一般には OS 上のプロセス中の実行コンテクストである。Java においては JVM 上の実行コンテクストである。実行コンテクストとは、計算機、OS 上のスタック、スタックポインタ、プログラムカウンタなど、ある計算主体に固有の資源、情報の総称であり、もともとはプログラミング言語の要素ではない。

もともとは言語の要素ではないスレッドが、オブジェクト指向言語の上ではどういった意味を持つのか。オブジェクト指向言語の計算モデルではオブジェクトが計算主体である。オブジェクトからオブジェクトにメッセージが送られ、レシーバがメッセージを処理することで計算が進んでいく。オブジェクト指向言語において、実行とはすなわちメッセージの処理である。つまり、実行コンテクストとはメッセージ処理中のレシーバ、その状態である。

2.3.2 スレッド移送

スレッドの移送とは、オブジェクト指向言語の計算モデルにおいては、メッセージ処理中のレシーバの移動である。処理中の状態を含めての移動である。第 2.4節で述べるように、Java 言語では、スレッドは一オブジェクトとして見える。Java においてスレッドの移送はスレッドオブジェクトの移動という意味もある。

このように、本来計算機上、OS 上の実体であるスレッドの移送は計算モデルにおいてそれほど大きな意味を持つわけではない。スレッドの移送による恩恵は、計算モデル上よりもむしろ、次のように処理コスト面のものが多い。

- メッセージ処理中のレシーバが移動することで負荷分散を図り得る。
- 頻繁にメッセージを送る対象に近付くことでメッセージ送信のコストを減らし得る。

2.4 プログラミング言語のマルチスレッドサポート

Java は言語仕様としてマルチスレッドプログラミングをサポートしている。言語プリミティブ、また hAPI として提供されているプログラミングインタフェースは、C,C++用スレッドライブラリに近い原始的でシンプルなものである。

以下、他の並列、並行処理向き言語と比較して、Java 言語のマルチスレッドサポートの特徴を述べる。

2.4.1 並行, 並列処理向き言語

プログラムが並列、または並行実行される場合、実行コンテクストは複数存在する。複数の実行コンテクストを共通の目的のために動作させるためには、実行コンテキスト間での

- 情報の共有
- アクセスの排他制御
- 同期

や、場合によってはスケジューリングの制御が必要となる。

並列、並行処理を言語プリミティブでサポートするプログラミング言語では、これら実行コンテクストの制御機構を用意している。例えば、並列オブジェクト指向言語 ABCL[14][18] では future 構文でプロセス、スレッドの生成、touch 構文で同期をサポートしている。GHC[16] や GHC を元に設計された並列論理型言語 KL1[3] では、複数の節、節のボディのゴール群が並行、並列に別プロセスとして動作し得る。プロセス間の同期は、情報の共有手段でもあるストリームを通じた通信によって実現される。他には、スレッドの生成、同期のために fork-join 型の構文を用意している言語もある。

2.4.2 Java 言語

Java はスレッドを一オブジェクトとしてプログラマに見せる。Thread クラスのオブジェクトを生成することがスレッドの生成である。スレッドオブジェクトに対して一時停止、実行再開、停止などの制御が可能である。また、排他制御、同期機構としては唯一モニタが用意されている。スレッド間の同期機構はオブジェクトのモニタに対する wait、モニタを wait しているスレッド、スレッド群に対する notify が用意されている。

このように Java では排他制御、同期についてはモニタが採用されているものの、スレッドの基本的な操作に対して何か高度な処理モデルが採用されているわけではなく、言語プリミティブや API では原始的な操作を提供しているだけである。

Java 言語でのマルチスレッドプログラミングは C,C++でのスレッドライブラリの利用に近い。C,C++では、POSIX threads や OS の用意するスレッドライブラリを利用し、スレッドの作成、制御変数を利用したスレッド間の同期など原始的な処理を関数呼び出しとして記述する必要がある。Java でも同様の原始的な操作を明示的に行うことになる。

Java 言語は汎用であるように設計され、広く使われることが目的とされているので、このような原始的でシンプルな設計は妥当だと考える。

Chapter 3

手法と実装

スレッドの移送 (第 3.1項) は

- クラス定義の移動 (第 3.2節)
- スレッドの外部化 (第 3.3節)
- オブジェクトの marshaling (第 3.4節)
- リフレクション (第 3.5節)
- Class Exportation (第 3.6節)

などの要素技術で実現されている (図 3.1)。

本章ではこれら要素技術の手法、提供しているスレッド移送システムの実装を述べる。

3.1 スレッド移送 (Thread Migration)

スレッドの移送とは次の 2 つの移送である。

- スレッドの状態
  スタックフレームごとに
  - スタックフレームに対応するメソッド
  - プログラムカウンタ (PC) — 最後に実行した命令を指す
  - スタックポインタ (SP)
  - 局所変数
  - スタック上の要素 (即値, オブジェクト)

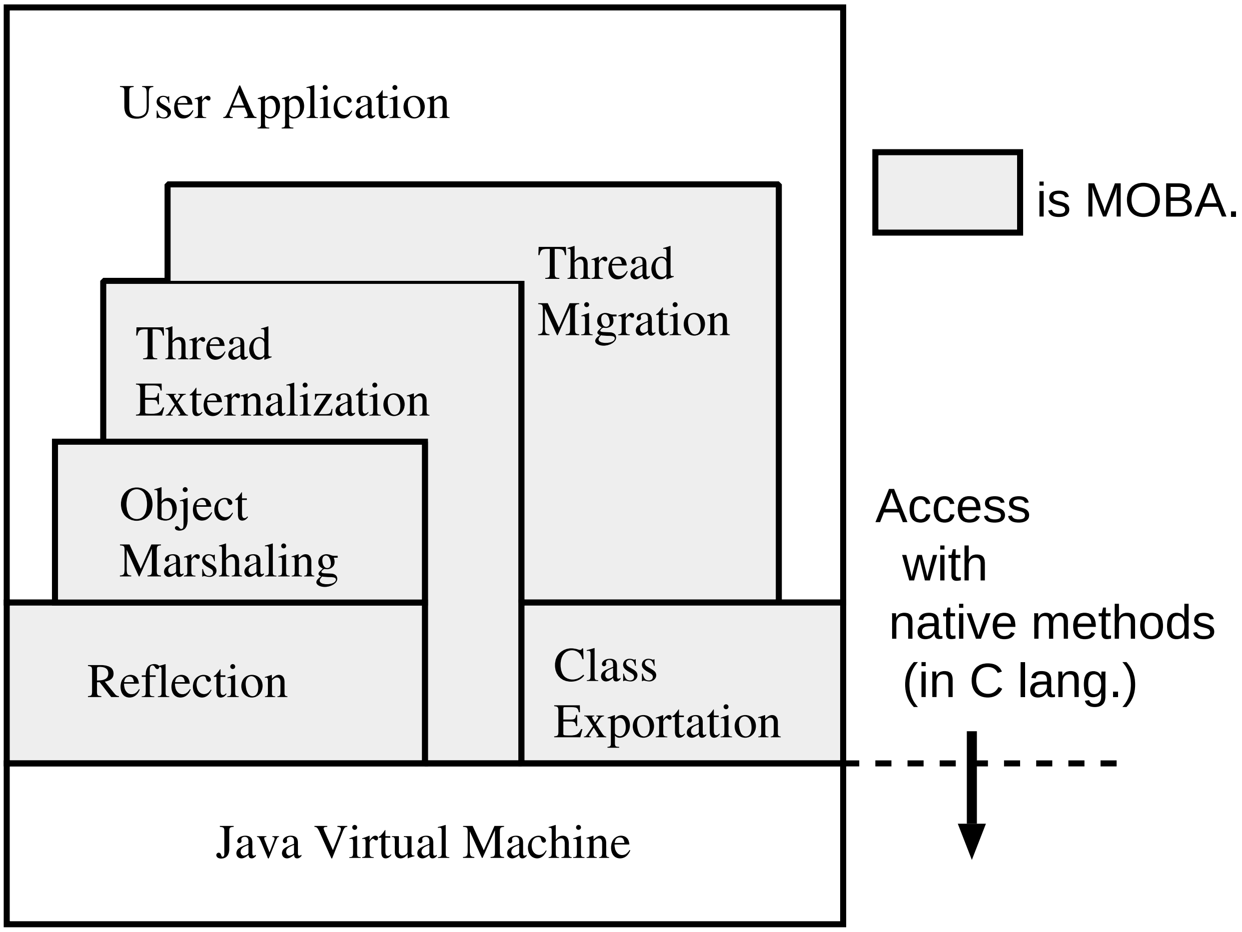

Figure 3.1: スレッド移送の要素技術

- スレッドが必要とするクラス定義

スレッドの移送とは、スレッドの状態の移送だけでなく、クラス定義、つまり型定義およびコードの移動も意味する。

スレッド移送にあたっては、複数スレッド間で共有している資源の扱いに配慮する必要がある。第3.1.3項で、目標とする手法と、現システムで採用している手法を述べる。

3.1.1 スレッドの状態の転送

状態の移送はおよそ次の手順で実現している。(図3.2)

1. 状態をバイト列化 (外部化)
2. バイト列を転送
3. 受け取ったバイト列から、実行中のスレッドを再構成 (内部化)
4. 利用者がスレッドの実行を再開。

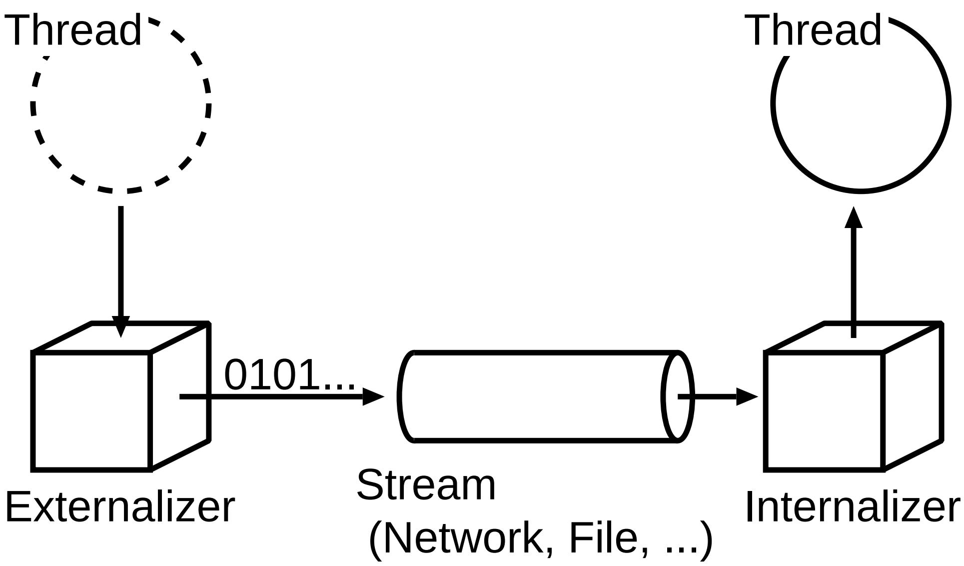

Figure 3.2: スレッドの外部化および内部化

転送はバイト列化と並行して行われ、再構成はバイト列を受け取りつつ行われ、全体としてパイプライニングが行われる。

本研究では、ある時刻のスレッドの状態をバイト列化してJVMの外部の実体とすることをスレッドの外部化 (**Externalization**)、そのバイト列を元にJVM内にスレッドを再構

成することをスレッドの内部化 (**Internalization**) と呼ぶ。スレッドの外部化、内部化については、第 3.3節で述べる。

3.1.2 クラス定義の転送

スレッドの移送には状態の転送だけでは充分ではない。移送元には存在してスレッドが必要とするクラスの定義が移送先には存在しないかもしれない。スレッドが必要とするクラス定義を移送元から移送先に転送、移送先で定義する必要がある。クラス定義の移動手法を第 3.2節で述べる。

3.1.3 スレッド間の共有資源

第 2.2.1項で述べたように、JVM 上のスレッド群はオブジェクトを共有し得る。スレッド移送にあたって、複数スレッドに共有されている、つまり複数スレッドから参照を辿って到達し得るオブジェクトをどう扱うかを考える必要がある。移送されるスレッドと留まるスレッドの双方から到達し得るオブジェクトがあった場合、双方からそれまで通りローカルに参照することはできない。

個々のオブジェクトごとに次の選択肢がある。

1. 実体は移動せず、移送スレッドから遠隔参照する。
2. 実体を移動させ、移送元計算機から遠隔参照する。
3. 実体をコピーする。
    (a) コピー元、先の一貫性に配慮する。(遠隔参照のキャッシング)
    (b) しない。

また、Distributed Shared Memory(DSM) を利用または実現し、その上で JVM を動かすという選択肢もある。

共有オブジェクトを移動させ、移送元計算機でのローカル参照を遠隔参照に切替えることは避けるべきである。JVM、Java 言語は、ローカル参照の置き換えや遠隔参照をサポートしていない。そのため、遠隔参照への切替えを実現するためには JVM 中心部の改造が必要である。このアプローチは Java のプラットフォーム非依存性を大きく損なうため採用できない。

計算のセマンティクスを保てる手法の中では、移送先から移送元計算機上のオブジェクトへの遠隔参照が最も実現し易い。ただ、遠隔参照は実行性能の低下を招く。

実行性能のためには、可能なら移動先でローカル参照にしたい。スレッド間で共有していないオブジェクト、つまり移送スレッドのみから到達し得るオブジェクト (図 3.3) なら、スレッドと共に移動させて何も問題はない。

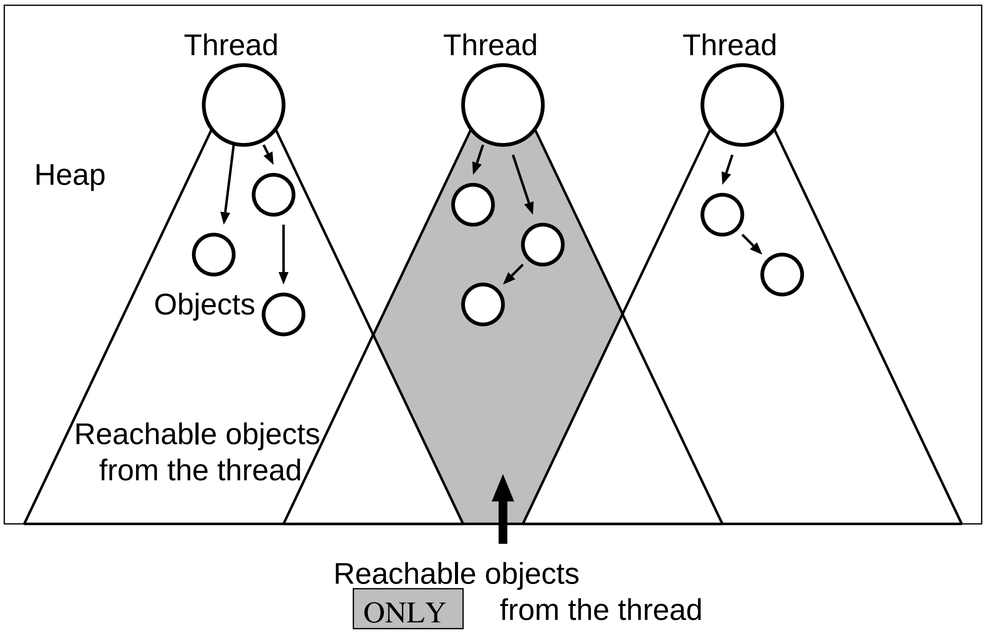

Figure 3.3: 単一スレッドのみから到達し得るオブジェクト集合

では、あるスレッドから到達可能なあるオブジェクトが、他のスレッドから到達可能か否かをどのように判断するか。この判定は原理的には可能である。ネイティブメソッド (native methods)、つまり C 言語で JVM 内に存在するオブジェクトを辿るという手段がある。また、もし JVM 内の Garbage Collecter(GC) の機能を部分的に利用でき、指定オブジェクトからの到達性が試験可能であれば、判定が可能である。しかし、GC の部分的な利用は提供されていない。原理的には可能だが、到達不可能性を示す必要があるため、オブジェクト間の参照関係を葉まですべてチェックする必要がある。少なくとも、GC 一回分の処理コストがかかる。

目標とする手法

移送されるスレッドから到達し得るオブジェクト群は、スレッド移送にあたって次のように扱いたい。

- 移送されるスレッドのみからしか到達し得ないオブジェクト (図 3.3) は、移送先に移動させる。

- 移送スレッド以外のスレッドからも到達し得るオブジェクトは、移送スレッドから移送元への遠隔参照とする。
- 遠隔参照するオブジェクトをキャッシングする。

現在採用している手法

現在採用している手法は次の通りである。

- 移送スレッドから到達し得るオブジェクトはすべて移送先にコピーする。
- 移送元と移送先でのオブジェクトの一貫性は考慮しない。

一旦、スレッド移送によってオブジェクトが移送先にコピーされると、コピーされたオブジェクトは元のオブジェクトとは別の独立したオブジェクトとなってしまう。複数スレッドから到達し得るオブジェクトは、本来単一であるべき実体が複数生成されてしまう。これにより、複数のスレッドがオブジェクトを共有するプログラムでは、単一 JVM 上での動作とセマンティクスが異なり得る。

現在のシステムはその特殊なセマンティクスを意識して利用する必要があるので、単一 JVM との透過性という点で不満が残る。

目標手法の実現には以下が必要である。

- 指定したスレッドからのみ到達可能なオブジェクトへの印付け
  native methods が必要。また、そのコードが JVM の構造に深く依存してしまう。
- 遠隔参照
  RMI などが提供する機構を利用するか、独自に用意するか。

3.2 クラス定義の転送

スレッドが必要とするクラス定義が移送先には存在しないかもしれない。存在することは期待できない。必要なクラス定義が移送先に存在することが保証されない限り、スレッドは移送先で実行を停止する可能性がある。

まず、クラス定義の自動転送、定義と、手動インストールについて、特に安全性に関して論じる。続いて、転送タイミングを、静的転送、動的転送、hybrid と分類し、それぞれの得失を述べる。また、移送スレッドのクラス名空間について考察する。最後に本研究のアプローチをまとめる。

3.2.1 手動, 自動

移送先に必要なクラス定義が存在することを保証するために 2 つの方法が考えられる。移送先として考えられる計算機に

1. プログラマが手でインストールしておく。
2. システムが自動で転送、転送先で定義する。

それぞれの得失を表 3.1に挙げる。

	自動	手動
手間	○ かからない	× かかる
実行時転送コスト	× かかる	○ かからない
安全性に対する配慮	要	不要

○: good, ×: bad

Table 3.1: クラス定義の手動、自動転送それぞれの得失

非均質環境でのクラス定義の自動転送、定義は、Java のプラットフォーム非依存性に負うところが非常に大きい。C++のようにプロセッサネイティブなコードを実行する処理系が一般的な言語では、移送元、移送先の処理系、プラットフォームが非均質であると、プロセス、スレッド移送はほとんど現実的ではない。均質であればクラス定義の移送はオブジェクトコードの転送で実現できる。非均質環境で、移送先でのソースコードからの再コンパイル、移送プログラムの再リンクによって実現するアイディア [15] はあるが、移送コストとメリットの点からあまり現実的ではない。

自動転送、定義は、インストールの負荷を軽減し、プログラマに透過的なスレッド移送を提供する。反面、セキュリティに対して配慮する必要が生じる。クラス定義とは型と振舞いの定義であり、実行コードを含む。クラス定義の転送、転送先での定義とは実行コードの移動である。スレッド移送の受け入れ側で、あらゆる利用者からのあらゆるスレッドを受け入れ、そしてあらゆる行動を許してしまっては、受け入れ側計算機上の情報、計算資源が安全ではない。受け入れ側計算機を安全に保つために、下に挙げる処理を受け入れ側で行うべきである。また、受け入れ側で認証、制限事項を細かく設定できるべきである。

- 受け入れスレッドの認証
  - スレッドの生成元計算機, 移送元計算機, 利用者などを authorization として
- 受け入れスレッドの行動制限

  - 行動時間、ステップ数の制限
  - 資源へのアクセス制限
    * ファイルの読み書き、リンク
    * 他計算機への接続、接続受け付け、マルチキャストの使用
    * 他スレッドの操作、プロセスの起動
    * 実行環境に関する情報へのアクセス
    * クラスライブラリ、動的リンクライブラリの利用
    * 印刷
    * ...
  - 再移送制限

手でのインストールは手間がかかる。スレッドの移送先すべてにインストールが必要なことは、スレッド移送の透過性を著しく下げる。反面、スレッド受け入れ側で安全性に対する配慮は不要である。クラス定義をインストールできるということは、その利用者はもともとそのコードをインストール先計算機で実行できるということなので、スレッドを受け入れることによる安全性低下はない。

インストールの手間がかかることは透過的な利用を妨げる、手動転送の難点である。逆に、安全性に対する配慮が必要な自動転送は、受け入れスレッドの認証、行動制限のための処理が必要となり、処理時間、パフォーマンスの点で不利である。

3.2.2 静的, 動的, 混合

クラス定義を転送するタイミングは、移送したスレッドの実行性能、クラス転送の完璧さに影響を与える。転送のタイミングは次が考えられる。

- 静的な転送 スレッドの移送時に、移送先で必要となるクラスを列挙、転送する。
- 動的な転送 実行時、移送先で必要になったら転送する。
- 混合 静的、動的転送の混合。

静的な転送　静的な転送では、プログラムによっては移送先で必要となるクラスを完璧に列挙できない場合がある。例えば、プログラム中でクラス名を文字列で与えている場合である。文字列がクラス名を含む可能性を考慮して列挙すれば、文字列がプログラム中で静的に与えられている場合には対応できるかもしれないが、文字列が実行時に動的に与えられる場合には対応できない。完璧なクラス転送は難しい。また、列挙して転送したものの、実際は移送先で利用されないといった無駄も生じ得る。

動的な転送　動的な転送では、クラス名が実行時に与えられる場合にも対応できるし、無駄な転送は生じない。反面、移送先からの要求に応じて転送を行うので、要求とそれに対する応答がネットワークを往復する。クラス定義の要求ひとつごとに通信遅延がかさむ。静的な転送とは異なり、プログラムの実行性能に影響を与える。

	静的	動的
完璧さ	×	○
実行性能への影響	○	×

○: good, ×: bad

Table 3.2: クラス定義の静的、動的転送それぞれの得失

静的, 動的転送の混合　実行性能への悪影響のない静的転送、クラス定義の不在を起こし得ない動的転送、それぞれの利点を活かすために、混合手法が考えられる。いくつかのクラスについてはスレッド移送と同時に静的に転送しておき、移送先に存在しないクラスが必要となったら動的に転送を行う。

3.2.3 クラス名空間

Java では、ファイルシステム上に定義が存在するクラスだけを利用している限りは、同一 JVM 上のすべてのスレッドは単一のクラス名空間を共有する。すなわち、どのスレッドから見ても同じ名前のクラスは同じ定義を持つ。

計算機をまたいでオブジェクトが移動する場合、移動元と移動先でクラス名空間が異なると、期待したクラスを発見できなかったり、期待とは異なる定義を得る危険がある。アクセスしようとするフィールドを発見できなかったり、期待とは異なるメソッドを呼び出してしまうかもしれない。

この問題はスレッド移送に限らず、ORB などの分散オブジェクトシステムでは起こり得る。遠隔メッセージ送信が pass by value で行われる場合、メッセージ送信先で引数自身または引数オブジェクトが利用するクラス定義を得られないことが考えられる。多くの ORB では、遠隔メッセージ送信先計算機で期待するクラス定義を利用可能にしておくことは利用者の責任である。利用者がインストールや更新を忘れることで不整合は容易に起こり得る。

以上の考察から、移送スレッドが無条件で他のスレッドと名前空間を共有するわけにはいかないことがわかる。もちろん、共有してよい場合はあるが、適切に個別のクラス名空間を持てる必要がある。

3.2.4 本研究のアプローチ

本節ではクラス定義に関するいくつかの側面と、選択肢、それぞれの得失を挙げてきた。ここで、本研究、実現したシステムでのアプローチを述べる。

自動転送, 定義

クラス定義のスレッド生成元からの自動転送、転送先での自動定義を実現している。スレッドの移送先計算機に手でインストールしておく必要はない。移送先計算機の安全性に対して、本来は 3.2.1項で述べた配慮が必要なところ、現在のシステムでは配慮していない。受け入れスレッドの認証、行動制限は今後の課題である。

利用者がクラス定義を手で各計算機にインストール、必要に応じて更新する必要があることは、分散システムの透過性を著しく下げる。多くの ORB ではメッセージ送信先計算機で目的のクラス定義を利用可能にしておくことは利用者の責任である。とはいえ、ORB においてはインストールの手間があまり問題とならない場合が多い。ORB の応用はほとんどの場合クライアントサーバモデル (client-server model) においてである。クライアントはサーバが輸出するオブジェクトのインタフェース (interface) さえ知れば充分で、その内部構造である実装 (implementation) については関知しない [20]。実装、つまりクラス定義はサーバのみが持てばよい。ゆえに、ORB 利用のたいていの局面ではクラス定義を持つ必要のある計算機はそう多くない。

ところが、スレッド移送においては、移送元と移送先にクライアントとサーバのような上下関係はない。移送先すべてで移送スレッドが必要とするクラスの定義が必要となる。また、ORB とは異なり、スレッド移送の典型的な応用 (表 3.3) ではいくつもの計算機をまたいでの処理を行うことになる。

クラス定義は手でインストールするしかないとすると、いくつもの移送先計算機へのインストール、定義に変更があるたびの更新が必要となる。ORB の場合よりも一層、透過性は低くなる。本システムでは、クラス定義は自動的に転送、定義されるので、利用者は定義のインストールに煩わされない。

動的転送

クラス定義は、スレッド移送先での必要に応じて動的に、スレッド生成元から移送先へ転送される。システムは静的な転送もサポートしている。混合方式は未実装、導入予定である。移送先で確実に必要になるクラスについてはスレッド移送に先だって転送することで、通信遅延による実行性能の低下を抑える計画である。

- 負荷分散
  相対的に負荷の高い計算機から低い計算機へスレッドを移送させる。スレッドは、分散システムを構成する計算機すべてに移送され得る。
- 移動エージェント

  巡回 いくつかの計算機をあらかじめ決められた経路で巡回する。

  自律的な移動先決定 自ら、移動を受け入れる計算機を探したり、移動先を決定したりする。

  …

Table 3.3: スレッド移送の典型的な応用

クラス名空間

クラス名空間は、スレッド生成元計算機一台につきひとつ存在する。つまり、移送スレッドが必要とするクラス定義はスレッドの生成元から取得される。生成元が同一計算機であるスレッド群は同一のクラス名空間を持ち、生成元が異なれば異なるクラス名空間を持つ。この具体的な実現法は、第 4.5 節で述べる。

これにより、別の計算機上で生成されたスレッド群が同一 JVM 上に存在してもクラス名とクラス定義の不整合は起きないことを保証できる。つまり、生成元が別計算機である 2 つのスレッドが同名で異なる定義を期待した場合、双方の要求が満たされる。

ただし、Java の標準 API に含まれる、または標準 API が利用するクラス、すなわち名前が "`java.`" および "sun." で始まるクラスの定義は、すべてのクラス名空間に共通である。標準 API のクラス群はあらゆる JVM が持っていて、定義が等しいことが期待できるので、ローカルのファイルシステムからロードして、すべてのクラス名空間で同一の定義を用いる。また、本システムが提供する API に含まれるクラスも標準 API と同様、ローカルからロードする。

3.3 スレッド外部化 (Thread Externalization)

スレッド移送は、スレッドの外部化、転送、内部化で実現されている (第 3.1.1項)。本研究では、ある時刻におけるスレッドの状態をバイト列化して JVM の外部の実体とすることをスレッドの外部化 (externalization)、逆に、そのバイト列を元に JVM 内にスレッドを再構成することをスレッドの内部化 (internalization) と呼ぶ。

外部化によってネットワーク経由の転送だけでなく、ファイルへの保存も可能となる。仮に JVM や計算機に障害が起きて実行が中断されても、保存したおいた時点からの実行を継続できるので、プログラムの耐故障性向上に利用できる。

本節ではまずバイト列化が必要なスレッドの属性を明らかにする。続いて外部化プロトコルを説明する。外部化においては、ある値が基本型なのかオブジェクトを指すのかの識別が必要となる。完璧な識別が不可能な理由、対策を述べる。最後に位置依存の資源の扱いを述べる。

3.3.1 状態

外部化、内部化後、元のスレッドの実行を継続できなければならない。継続に充分な情報をバイト列化する必要がある。

継続に必要な、スレッドの状態は以下の通りである。

- 実行コンテクスト
  - レジスタ
    プログラムカウンタ, スタックポインタ
  - スタックフレーム
    局所変数, オペランドスタック, $\cdots$
- スレッドから辿り得るオブジェクト群

3.3.2 手順

外部化および内部化の手順を述べる。

排他制御, 一時停止

まず、外部化対象スレッドのモニタを取得して、実行を一時停止させる。外部化の途中でスレッドの状態が変化してしまうと、そのバイト列は内部化できないからである。

一時停止の前にスレッドオブジェクトのモニタを取得するのは、スレッドを排他的に扱うためである。しかし、モニタを取得しても、完全に排他的な操作は実現できない。スレッドの実行開始、停止はモニタを通した操作となっているものの、一時停止、実行再開はモニタを通さなくても可能である。また、スレッドクラスを継承した移送スレッドクラスで一時停止、実行再開をオーバーライドして、モニタを通じた操作とすることも不可能である。スレッドクラスでオーバーライドを禁じているのである。このように、現在のシステムではスレッドの排他制御が不完全である。完全な排他制御は今後の課題である。

自身の外部化のための前処理

外部化対象スレッドの実行は一時停止させる必要があるので、自分自身、つまり現在実行中のスレッドの外部化はできない。自分自身の状態は外部化の過程で必ず変化するからである。自分自身の外部化は、別にスレッドを生成してそのスレッドに任せる。外部化のためのスレッドを生成したら、自身は一時停止する。

バイト列化

バイト列化の手順を示す (図 3.4)。これは本システムのスレッド外部化プロトコル (MOBA Thread Externalization Protocol) である。プロセス、スレッドといった計算主体をバイト列化する標準的なプロトコルは存在しないので、これは独自プロトコルである。

1. スレッドの属性
   そのスレッドが生成された計算機の IP アドレス,
   スレッドの名前, 優先度, デーモンスレッドかどうかなど
2. スレッドから辿り得るオブジェクト群
   Object Marshaling(3.4節) でバイト列化
3. スレッドの状態
   メソッド呼び出しで作られるフレームの連鎖を、呼び出しが古い方から順に辿る
   (a) クラス, メソッド名
   (b) return 先 (コード先頭からの offset)
   (c) オペランドスタックの先頭位置
   (d) 最後に実行された命令のアドレス (offset)
   (e) 局所変数
       メソッドが持つ局所変数表の各要素を辿る
       i. その時点で該当変数が生存していれば変数の値
          基本データ型であれば即値,
          オブジェクトであれば marshaling 結果のバイト列
   (f) スタック
       オペランドスタック上の各要素を辿る
       i. 要素の値

クラス定義中のメソッドは局所変数表を持ち、局所変数それぞれの有効期間をプログラムカウンタの範囲という形で持っている。この情報を参照することで、ある時点でその局所変数が生存しているかどうかを判定できる。

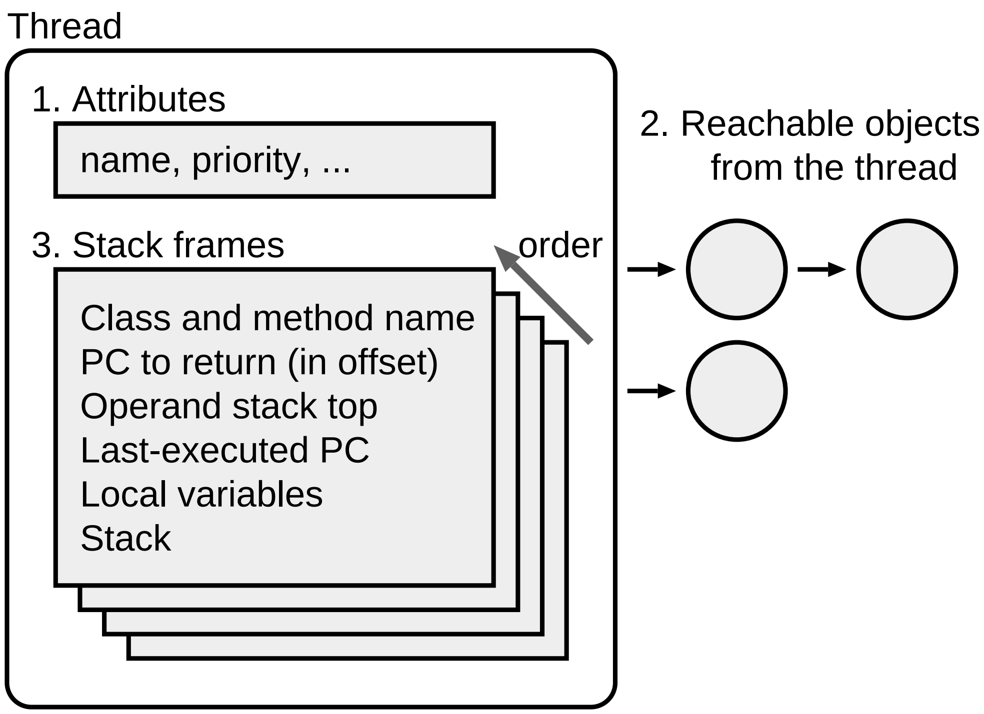

Figure 3.4: スレッドの外部化手順

スレッドの再構成

内部化では、外部化で生成されたバイト列を元に JVM 内にスレッドを再構成する。

まず、JVM 内ヒープ上のオブジェクトとしてのスレッドを再構成する。これはオブジェクトの unmarshaling(第 3.4節) と同様、リフレクション (第 3.5節) で実現できる。

スレッドは実行コンテクストを表す特殊なオブジェクトなので、ヒープ上の実体を初期化しただけでは済まない。少なくとも現在のソフトウェア実装の JVM では、スレッドオブジェクトはシステムレベルのスレッド (以下システムスレッド) と対応付けられている (図 fig:modeljvm 参照)。スレッドライブラリは、OS が提供するものであったり、Sun 社が提供するユーザ空間の Green Threads であったりする。

オブジェクトとしてのスレッドを構成したら次は対応するシステムスレッドを構成する。

システムスレッド上で、C やアセンブラで記述されたプロセッサネイティブなバイトコードインタプリタが走る。Sun 社の JVM では、このインタプリタはプログラムを途中から実行することを想定していない。そのため、本システムではインタプリタに wrapper をかぶせている。この wrapper は、`return` 命令、例外などを、JVM 中の処理と不整合を起こさずに処理できる。

3.3.3　基本型の値とオブジェクトの判定

局所変数の値、オペランドスタック上の要素は

- 整数、浮動小数点数、真偽値などの即値
- オブジェクト

であり得る。スレッド外部化の際、即値はネットワーク上表現に変換するだけで済むが、オブジェクトは marshal する必要がある。従って、局所変数やスタック上の要素を外部化するときは、それが即値なのかオブジェクトなのかを判定しなければならない。

局所変数については JVM 中に型情報があり、それを参照すれば済む。

問題はオペランドスタック上の要素である。少なくとも、Sun 社の JVM は、スタック上に要素の型情報はない。コード実行中のある時点でスタック上の要素の型が何であるかは、通常はコンパイラさえ知っていれば充分である。なので、JVM 内にその情報はない。原理的には、メソッドのコードを先頭から辿ってスタックの状態を追跡することで型は判明する。しかし、この手法は辿るコードの長さに応じた時間がかかるので、ライブラリへの採用は現実的ではない。

現在、本システムでは、OS からの Segmentation Violation シグナル (SIGSEGV) を利用して即値とオブジェクトの判定を実現している。スタック上の 32bit 値をオブジェクトであると仮定し、それに対してオブジェクトのメモリ上の構造に従ってアクセスしていく。も

し 32bit 値が即値であれば SIGSEGV が発生する可能性があり、オブジェクトであれば発生し得ない。発生しなければオブジェクトだと判定する。この手法で確実な判定は不可能だが、判定にある程度の時間をかけることで現実的な精度での判定は可能である。

3.3.4 位置依存の資源

別計算機上では意味を持たない、位置に依存するオブジェクトが存在する。これら位置依存のオブジェクトは、スレッド内部化の際に無効化する。詳細は Object Marshaling に関する節中、第 3.4.3項で述べる。

3.3.5 Just-in-Time コンパイラ対応

現在、スレッド外部化と Just-in-Time(JIT) コンパイラ双方の同時利用は実現できていない。

JIT コンパイルとは JVM による動的コンパイルである。JIT コンパイラを持つ JVM は、実行時にインタプリタの実行コードであるバイトコードをプロセッサの命令に変換できる。これによって実行の高速化が図れる。

スレッドの外部表現がプラットフォームに依存しないためには、プログラムカウンタ (PC) はバイトコード上を指すものでなければならない。プロセッサネイティブな命令を指す PC を外部化しても、全く同一の JVM、JIT コンパイラでしか内部化できない。

スレッド外部化を JIT コンパイラと共に利用するためには、次のどれかが必要である。

- プロセッサネイティブな命令上の PC の、バイトコード上の PC への変換。
- メソッド呼び出しやスレッドごとの、JIT コンパイルの禁止、許可。

どちらも現在は実現されていない。両者の実現可能性を調査中である。

3.4 Object Marshaling

本システムではスレッド移送の際にオブジェクト群を移送先にコピーするので、プラットフォームに依存しない、オブジェクトの JVM 外共通表現が必要となる。そのために、オブジェクト群をバイト列化する Object Marshaling を実現した。

Sun の提供する Object Serialization JDK の標準 API でも、Object Serialization[12] という同等の機能が提供されている。marshaling を独自に実装したのは、スレッド移送のために、標準 API が持つ制約を越える必要があったからである。例えば、Object Serialization の仕様では、インスタンスがバイト列化可能であることを、クラスを記述する時点で明示

的に表明しておく必要がある。ところが、スレッド移送では任意のオブジェクトをバイト列化する必要がある。標準 API Object Serialization の課す制約は一般的な利用においては妥当であるが、スレッド移送では越える必要があり、独自実装が必要となった。

3.4.1 アルゴリズム

オブジェクトは参照関係でグラフをなすので、オブジェクトのバイト列化とはオブジェクトのグラフのバイト列化である。基本アルゴリズムは次の通りである。オブジェクトのフィールドを順次走査し、基本型であれば定められた形式に書き出し、オブジェクトであれば再帰的にバイト列化する。循環グラフを扱うために、バイト列化済みオブジェクトの表を用意しておき、バイト列化対象のオブジェクトが表に見つかった場合は、オブジェクト自体のバイト列化の代わりに表中のインデックスを書き出す。

marshaling のアルゴリズムは次の通り。

```
METHOD marshaling(対象オブジェクト O)
IF（O は marshal 済み）{
  marshal 済みの表中の index を出力
  RETURN
}
marshal 済みの表に登録
IF（O は配列）{
  配列の要素すべてを出力
}
IF（O のクラスが自前の外部化メソッドを持つ）{
  外部化メソッドを呼ぶ
  RETURN
}
FOR クラス C IN O のクラスのスーパークラスを根から順に {
  IF（クラス C が自前の marshaling メソッドを持つ）{
    marshaling メソッドを呼ぶ
    CONTINUE
  }
  FOR フィールド F IN O の C 中のフィールド {
    IF（F はオブジェクトか配列型）{
      CALL marshaling(F)
    }
```

```
        ELSE { // F は基本型
          F の値を出力
        }
      }
    }
```

ここで、自前の外部化メソッド、marshaling メソッドとは次のようなメソッドである。

- 外部化メソッド　自クラスとスーパークラスのバイト列化の責任を負う。
- **marshaling** メソッド　自クラスのバイト列化の責任を負う。スーパークラス、サブクラスには配慮しない。

marshaling の過程で、クラスが持つフィールドの型や属性、値を得たり、unmarshaling の過程でフィールドに値を設定する必要がある。そのためにリフレクション (第 3.5項) を利用する。

プロトコルの頑強性　スレッド外部化 (第 3.3節)、Object Marshaling に関して、現在のプロトコルや内部化部、unmarshal 部は頑強さが充分ではない。プロトコルの頑強性とは、ストリームの様々な違法性を早く確実に検出しやすい性質を指す。プロトコル装置の頑強性とは、違法なストリームに対して誤動作しにくい性質を指す。

現システムは違法なストリームで誤動作し得る。より頑強なプロトコルおよび処理部の構成は今後の課題である。

3.4.2　データの外部表現

フィールドはオブジェクトへの参照か基本型の値を保持するので、marshaling では整数、浮動小数点数、真偽値など基本型を外部表現として書き出して、それを読み込める必要がある。

32bit 整数を 8bit 単位で扱う場合、8bit 要素の順によって 4 の階乗、つまり 24 通りの表現が考えられる。値を変えずに読み書きするためには、書き出し側と読み込み側で何かしらの合意が必要となる。方針はふたつある [19]。

- プラットフォーム独立表現を定める
- 書き出し側と読み込み側で、どの表現を用いるかネゴシエーションする

前者はネゴシエーションが不要で、実装もシンプルになる。後者は外部表現の扱いをより高速に行える可能性がある。例えば、読み書き双方のプロセッサが little-endian であった場

合に、プロセッサ上表現を外部表現としてそのまま採用することに双方が同意すれば、バイトオーダの変換オーバーヘッドはなくなる。
知られた外部データ表現はすでにいくつかある。

- Sun 社の Remote Procedure Call(RPC) ライブラリが利用している External Data Representation(XDR)
- Open Software Foundation(OSF) が採用した Netorking Computing System(NCS) の Network Data Representation(NDR)
- Xerox の Courier
- ISO 標準の Open Systems Interconnection(OSI) Abstract Syntax Notation One(ASN.1), Basic Encoding Rules(BER) and Distinguished Encoding Rules(DER)

NCS の NDR では、書き出し側がデータ表現、つまりバイトオーダを決定、プロトコルによって読み込みがわにデータ表現を伝える。他はすべて、単一のプラットフォーム独立の外部表現を用いる。
本システムの Object Marshaling では各基本型にプラットフォーム独立表現を定めている。Sun XDR と同じ方針である。

- ネットワークバイトオーダ (MSB first) に揃える
- 浮動小数点数の表現は IEEE754 floating-point standard 準拠

3.4.3 位置依存の資源

別計算機上では意味を持たない、位置に依存するオブジェクトが存在する。例えばローカルファイルシステム上のファイルに対応するオブジェクト、ファイル記述子、ソケット記述子に対応するオブジェクト、計算機、JVM に密着した情報に対応するオブジェクトなどがある。
marshal、unmarshal の際、これらに配慮しないと、利用不能になったオブジェクトにメッセージ送信して不都合が起きたり、unmarshal したオブジェクトが意図せぬ資源と結び付いて不正な操作をし得る。これら位置依存のオブジェクトに対しては、内部化の際に無効化または補整処理を施せればよい。
本システムでライブラリとして提供できるクラスであれば、クラス定義自体に、無効化、補整処理を組み込める。ところが、処理系の一部として提供されている標準 API が含むクラスに新たな処理は追加できない。標準 API が含む位置依存クラスについては、marshal、unmarshal の際に存在をチェックし、無効化や補整処理を施す。

無効化、補整が不要な場合もある。marshal 後、同じスレッドがそれを unmarshal すれば、たいていの場合不要である。ただし、ある資源に対応するオブジェクトの存在がひとつしか許されない場合などは、unmarshal 時の補整処理は避けられない。

位置依存性でオブジェクトを次のように分類できる。

- 位置依存
  - 同一スレッドによる marshal、unmarshal では特別扱い不要
  - 常に無効化または補整処理が必要
- 位置非依存

標準 API に含まれないクラスであれば、無効化、補整処理はクラスを記述するプログラマが責任を負うべきである。そのクラスが上記のどの分類に属するかはプログラマのみが知っているからである。標準 API 中のクラスについては、すべてについて本システムで無効化、補整処理を施す必要がある。

現在の実装

現在の実装は次のようになっている。

- 標準 API 中のいくつかのクラスについて無効化が可能。
- 補整処理は未実装。
- 移動可能な代替クラスライブラリは未提供。

課題

標準 API 中の位置依存クラス　現実装では、ファイル記述子に対応するオブジェクトの無効化しか行っていない。不充分である。標準 API 中のクラスについて位置依存性を調べ、位置依存クラスについては同一スレッド内での marshal、unmarshal 時にも特別扱いが必要かどうかを調べなければならない。また、unmarshaling に無効化、補整処理を追加しなければならない。

移送対応クラスライブラリ　移動先では無効化するしかないオブジェクト、例えばソケットなどに対しては、本システムで代替クラスを用意することで、スレッド移送に対応できるかもしれない。プログラマは代替クラスを利用する必要があるが、移送に対応した選択肢があることはよいことである。

3.5 リフクレション (Reflection)

リフレクション (reflection) の定義は

> 計算システムが、自分自身の構成や計算過程・計算方式に関する計算を行うこと

である。

Java では、オブジェクトが自身のフィールドや自クラスのメソッドについて属性を得たり設定することや、自身の挙動を変更することを指す。Java の標準 API もリフレクション API を含む。本システム、標準 API どちらも、提供している機能はフィールド、メソッドの属性操作程度のものである。それほど差はない。

標準 API のみで提供されている機能は

- メソッド、コンストラクタの呼び出し

本システムのみで提供されている機能は

- クラス定義中の定数表 (constant pool) へのアクセス

である。

3.5.1 本システム中の位置

リフレクションは、本システムではオブジェクトの marshaling および Class Exportation に用いられる。marshaling の過程で、クラスが持つフィールドの型や属性、値を得たり、unmarshaling の過程でフィールドに値を設定する必要がある。これらをリフレクションで実現している。Class Exportation では、クラスファイルを JVM 内のクラス定義から再構成するために、フィールドやメソッドの属性を得るため、クラスの定数表にアクセスするために利用している。

標準 API に含まれるリフレクション API を利用せずに独自に実装したのは、marshaling と同様に (第 3.4節)、標準 API が持つ制約を越える必要があったからである。スレッド移送では任意のオブジェクトをバイト列化する必要がある。リフレクションを利用しようが、本来アクセスできないフィールドには当然アクセスできない。標準 API Reflection の課すこの制約は一般的な利用においては妥当であるが、スレッド移送では越える必要があり、独自実装が必要となった。

3.6 Class Exportation

スレッド移送には必須ではない。クラス定義の JVM 外部表現、つまりクラスファイルを JVM 内の情報から再構成する技術、ライブラリである。クラス定義の転送処理の高速化に利用している。

本システムはスレッド移送にあたってクラス定義の転送も行う。転送元で Class Exportation を利用することで、スレッド移送先でクラス定義を得るのにかかる遅延を短縮できる。

3.6.1 応用

JVM は、通常ローカルのファイルシステム上でクラスファイルを探し、ロード、定義する。クラスの定義後クラスファイルのバイト列は捨ててしまう。再びバイト列として必要になったときは、再度ファイルシステム上を走査する必要がある。ファイルシステムへのアクセス時、OS によるキャッシングはある程度期待できるものの、一旦 JVM 内にロードされたクラスについて再びファイルシステムへアクセスするのはアクセス時間の無駄である。

クラスファイルを JVM 内から再構成する時間がファイルシステムからクラスファイルを探すのにかかる時間より短ければ、Class Exportation を利用してクラスファイルを得るための時間を短縮することが可能である。

3.6.2 手法

リフレクション (第 3.5項) を利用して、JVM 内からクラスに関する情報を得てバイト列化し、クラスファイルを復元する。クラスファイルのフォーマットは JVM の仕様 [11] に詳しい。Class Exportation に必要なので、リフレクションにクラスの定数表へのアクセス機能を用意した。

3.6.3 課題

quick 命令

Sun 社の JVM では、あるバージョン以降、動的にコードを書き換えるという最適化を採用している。一度実行された命令がある条件を満たした場合、“<命令名>_quick” という命令 (以下 quick 命令) に変換する。一度目の実行でのみ JVM が必要とする初期化などの処理を省けるといった利点がある。

例えば、定数表中から即値、オブジェクトを取り出す `ldc` 命令、オブジェクトのフィールドを取得する `getfield` 命令、メソッド呼び出しの `invokevirtual` 命令、オブジェクトを生成する `new` 命令などが quick 命令に変換される。

現在の Class Exportation ライブラリは quick 命令に対応していない。quick 命令をそのままクラスファイル中に含めてしまい、これは問題を起こす。そのコードを実行しようとした JVM は、一度目の実行であるにも関わらず quick 命令を実行する。これにより、次の理由で JVM が異常終了することがある。

- 一度目の実行では必ず必要な処理がなされない。

- quick 命令生成時の JVM の状態に依存したオペランドで不正な処理を行う。

quick 命令はその時点の JVM の状態に依存したオペランドを取る。例えばメソッド呼び出しの quick 命令では、JVM 内メソッド表中のインデックスが引数となる。メソッド表の構成は、クラス定義の変更などが理由で実行毎に変化し得る。

上記が理由で、quick 命令は JVM の外に出すべきものではない。Class Exportation では quick 命令は通常命令に変換する必要がある。quick 命令から通常命令への変換は今後の課題である。

Chapter 4

システム

提供しているスレッド移送システム MOBA—Mobile Agent Facilities on Java Language Environment—について述べる。

特徴、よい点、新しい点、そして問題点を挙げ、対策を述べる。続いて、2 つのサブシステム、スレッド移送系、クラスロード系を説明する。最後に、本システムのプログラミングインタフェースを説明する。

4.1 特徴

システムの特徴を挙げる。

4.1.1 長所

- 実行コンテクスト (第 3.3.1項) を含めて、異なる Java Virtual Machine(JVM) 間で移送が可能。
- メソッド呼び出しひとつで移送が可能。
- クラス定義を自動転送。移送先へのクラスファイルのインストール不要。
- 非均質環境で動作。異なるプラットフォーム間で移送が可能。

実行コンテクストを含めた移送

移動エージェントの記述をサポートしたライブラリがすでにいくつか入手可能である [2][10][17][1]。そのどれも計算主体の移動をサポートしているが、どれにおいても移動の際に実行コンテクスト (第 3.3.1項) は保存されない。移動したオブジェクトに対してメッセージが送られ、システムまたは利用者が指定したメソッドが新たに呼び出される。つまり、移

動先で実行は継続されない。これにより、プログラマにはローカルに動作するプログラムと移動エージェントの記述に異なる作法が要求される。

本システムでは実行コンテクスト (第 3.3.1項) を含めてのスレッド移送を実現している。移送先での実行継続が可能なので、ローカルに動作するプログラムを容易に移送スレッドにできる。また、実行コンテクストを含めたスレッド外部化によって、実行中のプログラムを保存して耐故障性向上を図ることが可能となった。

クラス定義の自動転送

ほとんどの ORB、分散オブジェクトシステムでそうであるように、実行が行われる計算機にあらかじめクラス定義をインストールしておく必要はない。すべての計算機上に必要なのは Java の実行系と本システムのみである。

非均質環境で動作

従来のプロセス移送のように、移送元と移送先が同一プロセッサ、同一 OS という均質な環境である必要はない。JVM は基本的にインタプリタで実行コードはプラットフォーム非依存なので、JVM 上のスレッドは中立な外部表現を持ち得る。

4.1.2 難点

逆に、難点を挙げる。

- 複数スレッドがオブジェクトを共有する場合、ローカル実行とセマンティクスが異なり得る。
- Just-in-Time(JIT) コンパイラと同時利用ができない。
- 移送先計算機の安全性に考慮していない。
- アプレットから利用しにくい。

複数スレッドから共有されるオブジェクト

現在の実装では、移送スレッドから到達し得るオブジェクトは移送先にコピーされる。移送先で、移送元にあったオブジェクトとは異なる実体が生成される。複数スレッドから到達し得るオブジェクトは本来単一であるべき実体が複数になってしまう。これにより、場合によってはプログラムのセマンティクスが変わる。

現システムは、この特殊なセマンティクスを知った上で利用するしかない。改善案として第 3.1.3項で述べた手法を検討している。移送先での実行性能を犠牲にせずにローカル実行と同じセマンティクスを保つことが目標である。

JIT コンパイラとの同時利用

JIT コンパイラとの同時利用を実現できていない。第 3.3.5項で述べたように、解決法の実現可能性を調査している。

移送先計算機の安全性

第 3.2.1で述べたように、スレッド移送にあたっては移送先計算機の安全性に考慮して、移送スレッドの認証、移送スレッドの行動制限を検討する必要がある。
現システムではこれらに考慮していない。仕様を定めて実装する予定である。

アプレットからの利用

本システムは、通常の Java のコードには許されない JVM 内への深いアクセスを必要とするため、C 言語で記述したメソッド、ネイティブメソッド (native methods) を使用している。Web ブラウザが持つ JVM では、ネイティブメソッドはローカルのファイルシステム上のものしか利用できない。つまり、本システムはアプレットから利用する場合にもインストールが必要である。よって、アプレットからの利用は現実的ではない。
Sun 社自身が Java の仕様としてスレッド外部化を用意しない限り、スレッド移送にはネイティブメソッドが必要なので、この問題に対策はない。

4.2 動作環境

本システムは

- Sun Microsystems の提供する Java Development Kit(JDK) 1.1 以上
- UNIX 互換 OS

の上で動作する。
リフレクション、スレッド外部化の処理方法は JVM の構造に深く依存する。そのため、現システムを JDK 以外の上で動かすためには移植作業が必要となる。
OS に依存する処理はしていないので、UNIX 互換 OS 以外でも容易に動かせる可能性はある。未確認である。

4.3 構成

システムは次から構成される。

- Place
- クラスライブラリ

4.3.1 Place

place(図 4.1) は移送スレッドが動作する場である。スレッドの移送を受け入れる。

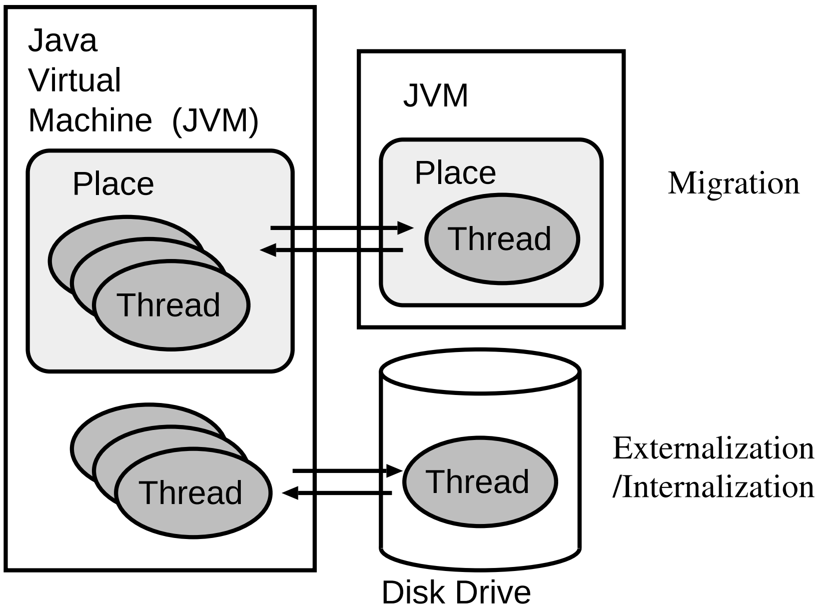

Figure 4.1: Place

place はひとつの JVM を占有するが、同一計算機上には複数存在し得る。同一計算機上の place は、移送を待ち受けるポート番号で識別できるので、place のアドレスは、ホストの識別子、つまりホスト名か IP アドレスとポート番号の組である。

4.3.2 2 つのサブシステム

本システムには、次の 2 つのサブシステムがある (図 4.2)。

スレッド移送系 (Thread Migration System) スレッド移送を実現する系。

クラスロード系 (**Class Loading System**) 移送スレッドにクラス定義を提供する系。

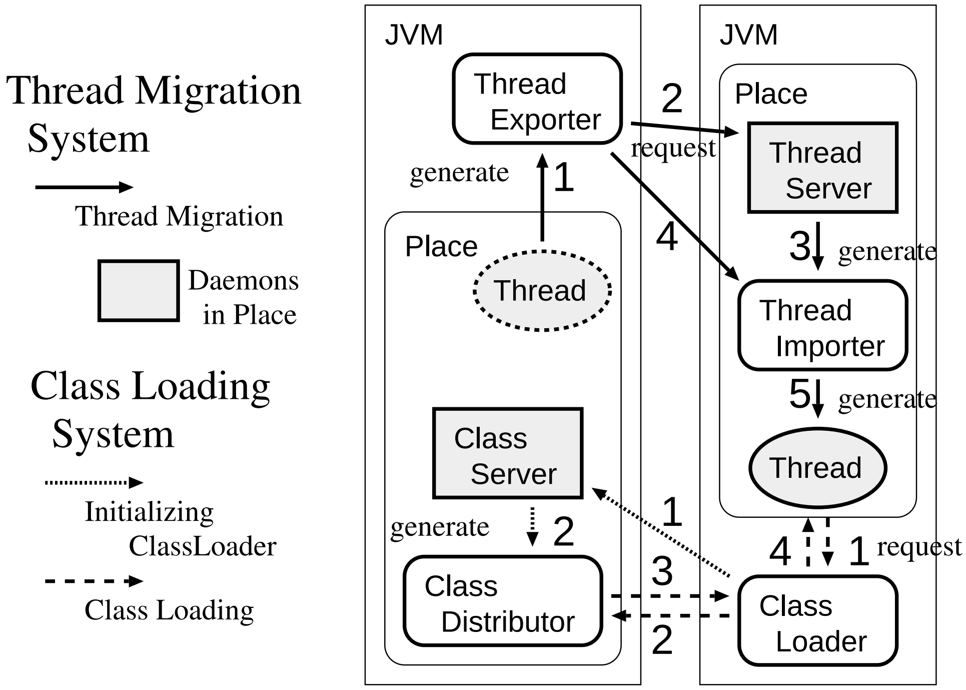

Figure 4.2: 2 つのサブシステム

続く節で、これら 2 つの系について構成要素、動作を述べる。

4.4 スレッド移送系

スレッド外部化、内部化などスレッドを扱う系。次を指す。

- 移送元で使用されるエクスポータ

と、移送先 place 中の

- デーモンであるスレッドサーバ
- スレッドサーバが生成するインポータ

移送時の動作は次の通り。

1. 移送スレッドがエクスポータを生成。

2. エクスポータが移送先に対してインポータの生成を要求。
3. 移送先でインポータが生成される。
4. エクスポータがインポータに、スレッドを外部化して渡す。
5. インポータがスレッドを内部化。
6. 移送元でスレッドを停止。

4.5 クラスロード系

移送スレッドにクラス定義を提供する系。次を指す。

- 移送スレッドが使用するクラスローダ

と、place 中の

- デーモンであるクラスサーバ
- クラスサーバがクラスローダごとに生成するクラスディストリビュータ

移送スレッドは自身の生成元計算機からクラス定義を取得する (第 3.2節) ので、JVM が用意しているものではないクラスローダが必要である。また、クラスローダはネットワーク経由でクラス定義を取得するので、ネットワークの対向にクラス定義を提供してくれる何かが必要である。それがクラスディストリビュータである。クラスローダがクラス提供元のクラスサーバに自分に対応するクラスディストリビュータの生成を依頼し、クラスサーバは依頼に応じクラスディストリビュータを生成する。

Chapter 5

応用

5.1 移動エージェント (Mobile Agent)

エージェントという語の定義は文脈に大きく依存する。ここでは移動エージェントを

　　計算機間を移動可能な計算主体

と定義する。

5.1.1 既存システム

移動エージェントを実現したライブラリは既にいくつか公開されている (表 5.1)。あらゆる既存システムは、移動の際に実行コンテクスト (第 3.3.1項) を保存できない。移動したオブジェクトに対してメッセージが送られ、システムまたは利用者が指定したメソッドが新たに呼び出される。つまり、移動先で実行は継続されない。

Aglets Workbench[2] 日本 IBM による移動エージェントのプログラミング環境

Kafka[10] 富士通研究所によるマルチ・エージェント記述ライブラリ

Voyager[17] ObjectSpace 社の分散システム開発基盤

Odyssey[1] General Magic 社の Mobile Agents

Table 5.1: 既存の移動エージェント

移動によって、それまで実行していたメソッドとは別のメソッドに制御が移るため、プログラマはそれを意識してコードを書く必要がある。このように、既存ライブラリでは、

単一 JVM 上で動作する移動しないコードとは異なる作法が要求される。プログラミングインタフェースの透過性が低い。

5.1.2 MOBA

本システム MOBA の移送スレッドは移動エージェントとみなせる。移送スレッドは実行コンテキストを含めて移動し、移動先で実行を継続できる。よって、移動エージェントの記述に、単一 JVM 上でローカルに動作するスレッドと異なる作法は要求されない。

5.1.3 プログラミングインタフェースの比較

MOBA の API の透過性を示すために、移動エージェントの記述法を MOBA と ObjectSpace 社の Voyager 1.0 とで比較する。

Voyager 1.0

`Agent` クラスのサブクラスとして記述。移動は、移動先と移動先で呼び出したいメソッド名を引数に `moveTo()` メソッドを呼ぶ。

```
import COM.objectspace.voyager.*;

public class MobileAgent extends Agent {
  public void callback(Object arg) {
    System.out.println((String)arg);
  }

  public void go (String address /* is <host:port> */) {
    moveTo(address, "callback", "Hello !");
    System.out.println("This statement is never executed !");
  }
}
```

オブジェクトの移動と、移動したオブジェクトに対するメッセージ送信である。移動のために `moveTo()` の呼び出すと、移動後は `moveTo()` の引数として与えたコールバックメソッド `callback()` に制御が移る。

MOBA

通常のスレッドの記述では Thread クラスを使用するところ、MobaThread クラスを使用する。それ以外は通常のスレッドと同様に記述する。

次のようにスレッドを生成し

```
MobaThread t = new MobaThread(...);
t.start();
```

次のようにスレッド自身で移動する。

```
MobaThread.goTo(移動先計算機);
```

5.2　プログラムの耐故障性向上

スレッド外部化 (第 3.3節) を利用して、実行中のスレッドをファイルやデータベースに保存することが可能である。適当なタイミングで実行中の状態を保存しておけば、計算機の停止や故障時にも保存時の状態から実行を継続できる。また、もし緊急の電源断など計算機の停止や故障をあらかじめ検知することが可能ならば、先だって状態を保存したり、他の計算機上へ移動することで、それまでの計算結果を失わずに済む。

実行に数週間以上かかるプログラムには、不慮の事故に備えて、実行途中の状態を保存する仕掛け、保存した状態から復帰する仕掛けを用意することが一般的である。スレッド外部化を利用することで、このような非本質的な部分に労力を割く必要がなくなる。

課題

プログラムにとっての異常事態のうち、OS のシャットダウンなど、あらかじめ検知することが原理的に可能な事象も多い。ところが、Java のプログラムからそういった事態を検知する手段はない。C 言語ではある種のシグナルをとらえる方法が使えるものの、標準的で広く使える手法ではない。

Java のプログラムから JVM の異常終了を検知する方法を検討している。Java 言語の仕様の一部として提供されるのが望ましいので、よい仕様が出来れば提案していきたい。

5.3　負荷分散

従来研究されてきたプロセス移送 (第 2.1節) の主な応用である。プロセス移送を負荷分散の唯一の手段とするのは非効率だが、タスクの動的配置 (dynamic task placement) に加えてプロセス移送を利用することでさらなる性能向上を図り得る [4]。

課題

負荷分散に利用するためには、現在の MOBA および Java 自体に足りないものが多い。

- 計算機の能力、負荷の表現、取得法。
  様々な側面のある計算能力の表現法、およびその取得、見積り法。
- 負荷分散のポリシー。
  能力、負荷情報から移送対象、移送先を決めるポリシー、アルゴリズム。
- 利用者が手で移送を行うインタフェース。
  手で負荷を分散させるためのユーザインタフェース。

5.4 遠隔メッセージ送信のエミュレーション

スレッド移送で遠隔メッセージ送信と同等の処理が可能である。メッセージ送信を送信先計算機への移動で、return を送信元への移動で代替すればよい。

ただ、意味的には同等の処理が可能とはいえ、スレッド移送はメッセージ送信より処理が重い。実行性能的には代替する理由はない。また、単純に代替するよりむしろメッセンジャスレッドを生成してエミュレーションを軽量化すべきである。

Chapter 6

性能評価

第 5.1節では、移動エージェントの記述に着目し、既存システムと比較して本システム MOBA を評価した。本章では性能を評価する。

MOBA と比較対象について次を測定した。

- スレッド移送のオーバーヘッド
- データ転送性能

移送のオーバーヘッド　スレッド移送のオーバーヘッドは ObjectSpace 社の Voyager[17] と比較した。Voyager は agent-enhanced ORB と名乗り、本システムと同じく移動エージェントの記述を重要な応用としている。本システムではスレッドを移動エージェントとみなし、移動エージェントの移動時間を計測した。

転送性能　転送性能は、スレッド移送で遠隔メッセージ送信をエミュレート (第 5.4節) し、いくつかの Object Request Broker(ORB) と比較した。比較対象として、MOBA と同じく Java に特化して利便性、機能を追求する方向で開発されている RMI[13],HORB[8][21] を選んだ。

6.1　実験環境

以降、次の計算機をそれぞれ **WS1**,**WS2** と呼ぶ。

WS1: Sun Ultra 2 (UltraSPARC 168MHz$\times$2)
JDK 1.1.5 の JVM (Green Threads 版)

WS2: Sun Enterprise 3000 (UltraSPARC 167MHz$\times$6)
JDK 1.1.5 の JVM (Green Threads 版)

ネットワーク: 100Mbps Ethernet (100base-TX, リピータ 1 台を経由)

Voyager はバージョン 1.0.1 を使用、HORB はバージョンは 1.3b1 を基本に、performance patch 有りと無し双方について計測した。

6.2 オーバーヘッド

なるべく転送されるデータを少なくした移動エージェントを記述し、WS1、WS2 間を往復させてかかった時間を計測した。移送に伴うオーバーヘッドとして表 6.1の値を得た。

MOBA	2.7×10^2(msec)
Voyager	4.7×10^2(msec)

Table 6.1: 移送のオーバーヘッド

第 5.1節で述べたように、あらゆる既存システムは実行コンテクスト (第 3.3.1項) を移動できない。MOBA は実行コンテクストも含めて移動させるにも関わらず、オーバーヘッドは Voyager の半分程度である。

他のシステムと比較してオーバーヘッドが小さいとはいえ、応用によっては数百ミリ秒の遅延は大きすぎる。例えば負荷分散 (第 5.3節) では、遅延が小さいほど移送による利益が大きく、またより粒度の細かいタスクの移送が可能になる。逆に遅延が大きいと、大きな遅延に見合うだけの利益を得られるタスクは相応に粒度の粗いものとなってしまう。

6.3 転送性能

6.3.1 計測法

倍精度浮動小数点数 double 型の配列の転送レートを測定。

ORB 2 種: 遠隔メッセージ送信の引数に double 型の配列を渡し、long 型の返り値が返るまでの応答時間を計測。

MOBA: double 型の配列を保持して移動、long 型の値を持って元の計算機に戻るまでの応答時間を計測。

WS1 から WS2 に対してメッセージ送信を行った。

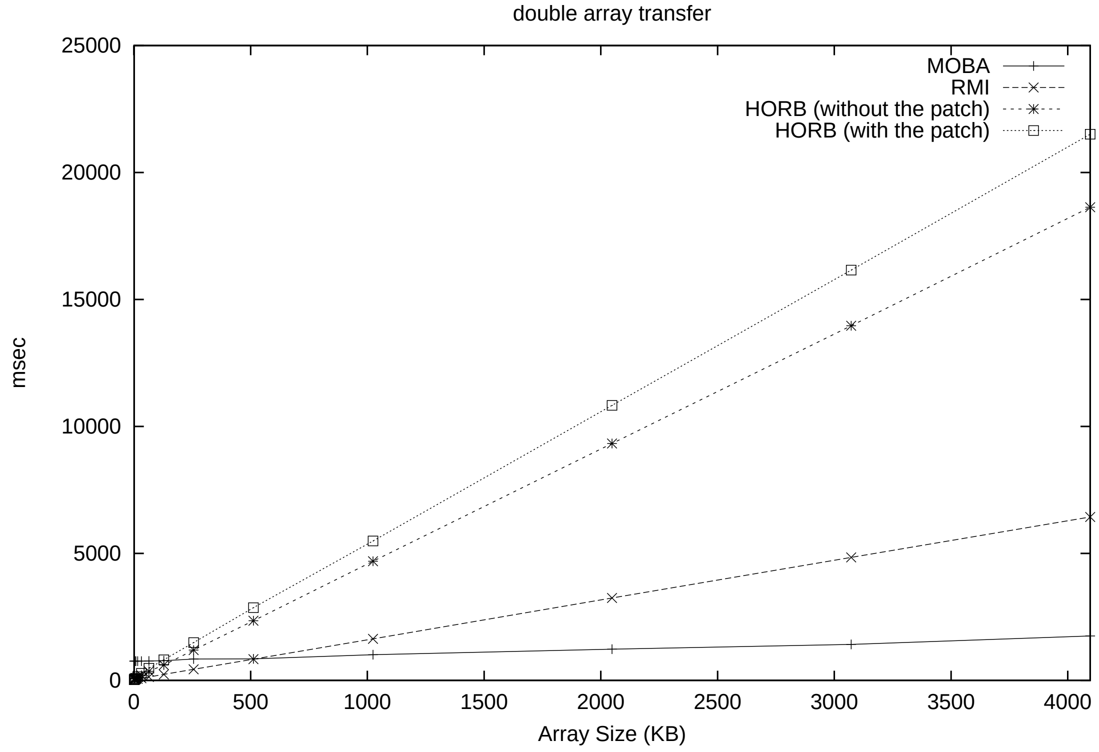

Figure 6.1: 倍精度浮動小数点数配列を転送しての応答時間

6.3.2 結果と考察

応答時間を図 6.1に、応答時間を元に算出したスループットを図 6.2に示す。

スレッド移送のオーバーヘッドは遠隔メッセージ送信よりも大きい。そのため、転送データ量が少ない状況では遅延の小さい ORB が良い結果を出している。データ量を多くするに従って相対的に MOBA の結果が良くなってくる。

MOBA はデータ量が 500KB を越えてはじめて RMI より良い結果を出すので、遠隔メッセージ送信をスレッド移送でエミュレートするのは厳しいと言える。

HORB の結果が悪いが、Microsoft の JIT コンパイラ付き JVM では RMI より高いスループット、小さい遅延を持つという報告がある。Java プログラムの評価結果は JVM およびクラスライブラリの実装に大きく依存することに留意。

6.3.3 参考

応答時間は転送データ量の一次関数であると仮定し、fitting を行った。

転送データ量を n(KB) としたときの応答時間 (msec) の式として表 6.2を得た。

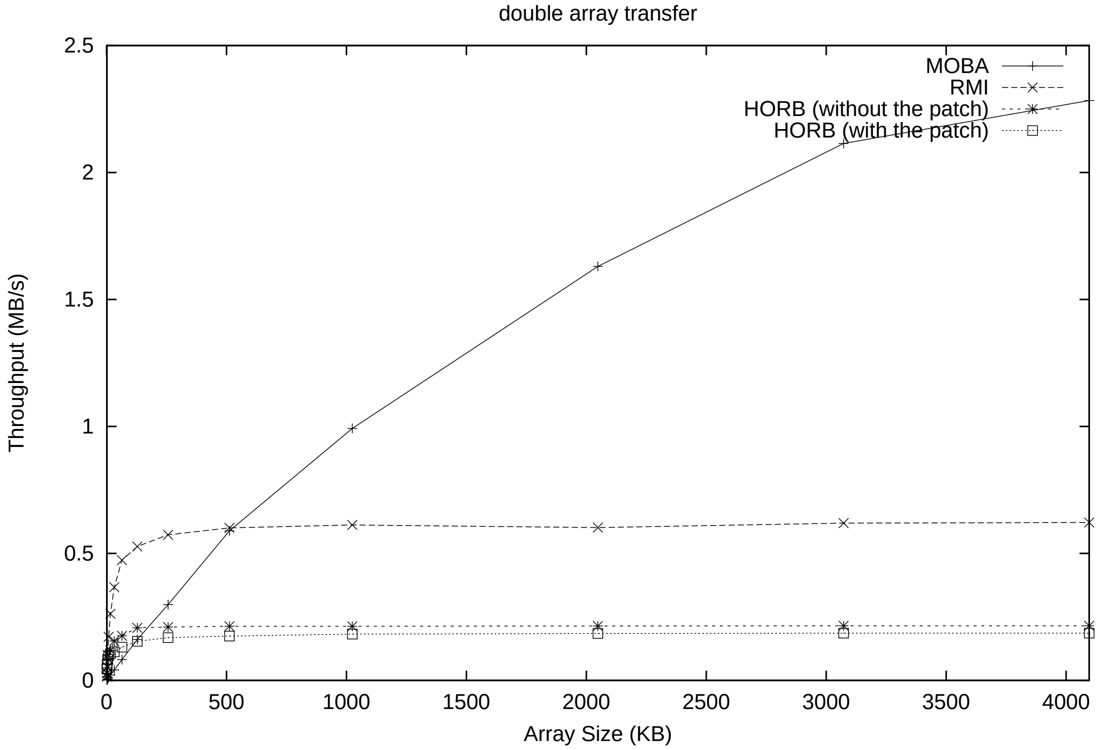

Figure 6.2: 倍精度浮動小数点数配列の転送レート

これは往復時間なので、MOBA の式が移送のオーバーヘッドを表すわけではないことに注意。

スレッド移送と遠隔メッセージ送信なので直接の比較対象とはならないが、比較するとスレッド移送は遠隔メッセージ送信よりも遅延が大きい。

MOBA	$2.3 \times 10^{-1} \times n + 7.5 \times 10^{2}$ (msec)
RMI	$1.6 \times n + 3.4 \times 10$ (msec)
HORB	$4.5 \times n + 5.0 \times 10$ (msec)

転送データ量を n(KB) として

Table 6.2: 転送データ量に対する応答時間の式

Chapter 7

結論

7.1 まとめ

Java 言語環境におけるスレッド移送について、その手法と実装を説明し (第 5章)、実現したスレッド移送システム MOBA—Mobile Agent Facilities on Java Language Envirionment—の特徴と構造を述べた (第 4章)。続いて、各種応用を挙げ、スレッド移送を適用することの利点や、さらなる研究課題を述べた (第 5章)。また、性能評価では、他システムと比較することで、本システムの低い移送遅延、高い転送性能を示した (第 6章)。

システム構築にあたって以下の目標を達成し、それにより透過的でシンプルなプログラミングインタフェース、様々な応用、高い利便性、相互運用性が実現された (第 4.1節)。

- 実行コンテクストを含めた移送
  移動エージェントの記述に、ローカルに動作するプログラムと異なる作法が不要。
- クラス定義の自動転送
  移送先へのクラス定義のインストールが不要。
- 非均質環境で動作
  従来のプロセス移送とは異なり、プラットフォームを選ばない。

7.2 課題

7.2.1 問題

現システムの問題を挙げる。

スレッド間の共有資源

スレッド移送の際、スレッドから到達し得るオブジェクトは移送先にコピーされるので、複数スレッドがオブジェクトを共有するプログラムのセマンティクスが変わり得る。第3.1.3項で改善策を議論している。実行性能をなるべく犠牲にしない手法を構成、実装する。

移送先計算機の安全性

本システムはクラス定義を自動転送するので、移送スレッドを認証する、行動を制限するなど、転送先システムの安全性に対する配慮が必要である (第3.2.1項)。これらは実現されていない (第3.2.4項)。

スレッド操作の排他制御

スレッド外部化などは完全に排他的に行う必要がある。ところが完全な排他制御が実現できていない (第3.3.2項)。実現可能性も含めて検討中である。

外部化,**marshaling** プロトコルの頑強性

スレッド外部化、Object Marshaling に関して、プロトコルやプロトコル処理部の頑強さが充分ではない。違法なストリームで誤動作し得る (第3.4節)。より頑強なプロトコルおよび処理部を構成する。

Just-in-Time コンパイラ対応

Java の中間コードを実行時にプロセッサの命令に動的コンパイルする Just-in-Time(JIT) コンパイラの、スレッド外部化との同時利用を実現できていない (第3.3.5項)。実現可能性を検討中である。

7.2.2 改良

移送操作のユーザインタフェース

負荷分散 (第5.3項)、耐故障性の向上 (第5.2節) に応用するため、スレッド移送を利用者が手で行えるインタフェースを用意する。

Java プログラムからの、**JVM,** 計算機の異常検知

プログラムにとっての異常事態が検出できれば、実行中の状態を保存することで耐故障性の向上が図れる (第5.2節)。異常事態には OS のシャットダウンなど検出可能なものがあ

るにも関わらず、Java のプログラムからそれら検出する方法はない。仕様を検討する。

計算機の能力, 負荷の表現, 取得法

スレッド移送によって計算機の負荷分散を図るには、計算機の能力およびある時点の負荷を知る必要がある (第 5.3項)。これらの表現、取得法を検討する。また、負荷情報から移送対象スレッド、移送先を決めるポリシー、アルゴリズムについて、その表現法、具体的な手法を検討する。

謝辞

本修士論文作成にあたり、日頃より御指導いただいている村岡洋一教授に深く感謝します。

研究の遂行にあたり、3 年間共に研究、議論して頂いたコンパイラ班の菅原健一、浜中征志郎両氏、また、興相正克氏に感謝します。物質、精神あらゆる面から私を支え、遅い帰宅にも辛抱強く耐えてくれた妻の麻里に感謝します。

また、有益な助言、意見をいただいた村岡研究室の諸氏に深く感謝します。

Bibliography

[1] Agent technology. http://www.genmagic.com/agents/.

[2] IBM Aglets Workbench. http://www.trl.ibm.co.jp/aglets/index-j.html.

[3] Kazunori Ueda an T. Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, Vol. 33(6), pp. 494–500, 1990.

[4] Yeshayahu Artsy and Raphael Finkel. Designing a process migratin facility: The Charlotte experience. *Computer*, Vol. 22, No. 9, pp. 47–56, September 1989. Reprinted in *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE Computer Society Press.

[5] CORBA/IIOP 2.1 Specification. ftp://ftp.omg.org/pub/docs/formal/97-09-01.{pdf,ps}.

[6] M. Raşit Eskicioğlu. Design issues of process migration facilities in distributed system. *IEEE Technical Comittee on Operating Systems Newsletter*, Vol. 4, No. 2, pp. 3–13, Winter 1989. Reprinted in *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE Computer Society Press.

[7] James Gosling, Bill Joy, and Guy L. Steele Jr. *Java Language Specification.* Addison Wesley, 1996.

[8] Satoshi Hirano. HORB: Distributed execution of Java programs. In *Proceedings of World Wide Computing and Its Applications*, March 1997.

[9] Java^{TM} IDL Documentation. http://www.javasoft.com/products/jdk/idl/.

[10] Kafka: Java 用マルチ・エージェント記述ライブラリ. http://www.fujitsu.co.jp/hypertext/free/kafka/jp/.

[11] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specificatin.* Addison Wesley, 1997.

[12] Sun Microsystems. *Java^{TM} Object Serialization Specification*, 1997.

[13] Sun Microsystems. *Java^{TM} Remote Method Invocation Specification*, 1997.

[14] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. ABCL/f: A future-based polymorphic typed concurrent object-oriented language – its design and implementation. In *Proc. of the DIMACS workshop on Specification of Parallel Algorithms*, pp. 275–292. American Mathematical Society, 1994.

[15] Marvin M. Theimer and Barry Hayes. Heterogeneous process migration by recompilation. In *Proc. IEEE 11the Int'l Conf. on Distributed Computing Systems*, pp. 18–25. IEEE CS Press, Los Alamitos, Calif., 1991.

[16] Kazunori Ueda. Guarded Horn Clauses. In Eiichi Wada, editor, *Logic Programming '85*, pp. 168–179, 1986. Springer-Verlag Lecture Note in Computer Science No.221.

[17] ObjectSpace: Voyager Overview. http://www.objectspace.com/Voyager/.

[18] Akinori Yonezawa, editor. *ABCL An Object-Oriented Concurrent System: Theory,Language, Programming,Implementation,and Application.* MIT Press, 1990. ISBM 0-262-24029-7.

[19] John Bloomer(森島晃年監訳). RPC プログラミング (原題 Power Programming with RPC). アスキー, 1995.

[20] Jean Bacon(藤田昭平, 篠田陽一, 今泉貴史訳). 並行分散システム (原題 CONCURRENT SYSTEMS). トッパン, 1996.

[21] 平野聡. HORB Home Page. http://ring.etl.go.jp/openlab/horb/.

Appendix A

プログラミングインタフェース

本スレッド移送システムのプログラミングインタフェース (API: Application Programming Interface) を説明する。

A.1 スレッド移送

まず、移送させたいスレッドは、通常スレッドを生成する場合の `Thread` 型ではなく、本 API の `MobaThread` 型のインスタンスとして生成する。

操作は、

- 移送
- 遠隔 fork

のどちらを行うか、移送や fork の対象が

- 自スレッド
- 他スレッド

のどちらなのかによって、4 種類ある。

これら選択肢の組合せ 4 種に対応するメソッドを `MobaThread` クラスに用意している (表 A.1)。これらはどれも次の引数を取る

移送先計算機 `String` 型のホスト名か、`InetAddress` 型の IP アドレス。

移送先ポート番号 `int` 型。省略可。

自身を遠隔計算機上に fork した場合はたいてい、自分が元のスレッドなのかコピーで生成されたスレッドなのかを判別する必要がある。UNIX 互換 OS の `fork(2)` システムコールでは、返り値のプロセス ID によって判別できるようになっている。本 API では `boolean isInternalized()` メソッドで判定できる。

	自スレッド	他スレッド
移送	`static void goTo()`	`void moveTo()`
遠隔 fork	`static void fork()`	`void forkThisThread()`

Table A.1: スレッド移送の API

A.2 Object Marshaling

次の 2 クラスを用意している。

`ObjectWriter` 基本型、オブジェクトのグラフをストリームに書き出す。

`ObjectReader` ストリームから基本型、オブジェクトのグラフを読み込む。

使用法 使用法は、Java の標準 API が含む `ObjectOutputStream`,`ObjectInputStream` とほぼ同じである (表 A.2)。

	コンストラクタ	marshaling, unmarshaling
書き出し	`ObjectWriter(OutputStream out)`	`void writeObject(Object obj`
読み込み	`ObjectReader(InputStream in)`	`Object readObject()`

Table A.2: Object Marshaling の API

A.3 スレッド外部化

object marshaling の 2 クラス (第 A.2項)

- `ObjectWriter`
- `ObjectReader`

を継承した

- `ThreadWriter`
- `ThreadReader`

という 2 クラスを用意している。

使用法　使用法は marshaling のクラスと同じである (表 A.3)。読み書きするオブジェクトが移送スレッド、つまり MobaThread クラスのインスタンスであった場合のみ、marshaling とは異なる動作をする。

	コンストラクタ	外部化, 内部化
書き出し	ThreadWriter(OutputStream out)	void writeObject(Object obj
読み込み	ThreadReader(InputStream in)	Object readObject()

Table A.3: スレッド外部化の API

A.4　リフレクション

主なクラスは次の 4 つである。

MetaClass クラスやインタフェースを表すクラス。

MetaField クラスやインタフェースが持つフィールドに関する情報、アクセスを提供する。

MetaMethod クラスやインタフェースが持つメソッドに関する情報、アクセスを提供する。

ConstantPool クラスが持つ定数表へのアクセスを提供する。

同様のリフレクション API は、定数表へのアクセスを除いて標準 API でも提供されている。
クラスごとに代表的なメソッドを挙げる。

MetaClass

- MetaClass get(String クラス名) or (Class クラスオブジェクト)
  MetaClass クラスのインスタンスを取得する。コンストラクタの代わり。

- MetaField field(String 名前) or (int インデックス)
  フィールドオブジェクトを取得する。引数はフィールド名か、クラスが持つフィールド表中のインデックス。

- MetaMethod method(String 名前, String signature) or (int インデックス)
  メソッドオブジェクトを取得する。引数はセレクタ、つまりメソッド名と signature の組か、クラスが持つメソッド表中のインデックス。

- `Object newInstance()`
  このクラスのインスタンスを生成する。

MetaField

- `String name()`
  フィールド名を得る。
- `String signature()`
  signature を得る。
- `* get*Value(Object obj)`
  オブジェクト `obj` 中のこのフィールドの値を得る。
  “*” には型の名前が入る。`byte`, `char`, `double`, `float`, `int`, `long`, `short`, `boolean`, `Object`。
- `void set*Value(Object obj, * value)`
  オブジェクト `obj` 中のこのフィールドに値を設定する。
  “*” には型の名前が入る。

MetaMethod

- `int maxStack()`
  このメソッドが使う、オペランドスタックの最大深さを得る。
- `int maxLocals()`
  このメソッドが必要とする、局所変数のスロット数を得る。
- `int codeLength()`
  コードの長さを得る。単位はバイト。
- `void writeCode(OutputStream out)`
  出力ストリーム `out` にコードを出力する。

ConstantPool

- `int count(Class clazz)`
  クラス `clazz` の定数表内の要素数を得る。

- int type(Class clazz, int index)
  index 番目の要素の型を得る。
  型は ConstantPool クラス中で定数として定義されている。

- * get*Value(Class clazz, int index)
  クラス clazz の定数表の index 番目の要素を得る。
  "*" には定数表の要素が持ち得る型の名前が入る。
  Utf8, Int, Float, Long, Double, Class, String。

A.5 Class Exportation

次のクラスを用意している。

ClassWriter クラスファイルを JVM 内から再構成、ストリームに書き出す。

使用法　Java の各種出力ストリームと同様に出力ストリームを元にインスタンスを作成、writeClass(Class clazz) メソッドで、クラスオブジェクト clazz に対応するクラスファイルをストリームに書き出す (表 A.4)。

コンストラクタ	exportation
ClassExporter(OutputStream out)	void writeClass(Class clazz)

Table A.4: Class Exportation の API