平成１４年度学士論文

ユビキタスコンピューティングのためのハンドオーバー機能付きRMIの実装

東京工業大学 理学部 情報科学科
学籍番号 ９９－１３４２－３
須永　豊

指導教官
千葉 滋 助教授

平成１５年2月２８日

概要

コンピューティング、ネットワーク技術は驚くべき速度で進化している。近い将来のうちに、我々の身の回りにはコンピューティングデバイス、ネットワークへの接続点、サービスやコンテンツが遍在するユビキタス環境が実現されるであろう。ユビキタス環境においては単一のネットワークリンクや端末でサービスを享受するものではなく、複数の端末やリンクを跨いでサービスが享受されることになる。つまり、ユビキタス環境というのは大規模な分散コンピューティング環境のことである。そのため、ユビキタスを支えるプログラミング技術として、分散処理プログラミングは欠かすことのできないものであると言える。

しかし、そのような分散処理プログラミングで重要となるハンドオーバー機能を支援したユビキタス用の分散処理ミドルウェアの実装は我々の知る限りではまだ行われていない。ハンドオーバーとは主に携帯電話などに使われている機能のことで移動しながら使える仕組みのことである。本稿では、ハンドオーバーという言葉を分散コンピューティングの分野に応用し、移動しながら分散処理を行える機能という意味で用いている。ユビキタスを実現するにあたってハンドオーバー機能が必要だということは以前から言われてきた。ユーザのコンテキストに合わせて、動的にリソースを変化させることのできる機能はユビキタスコンピューティングにとって必要不可欠なためである。しかし、ハンドオーバー機能をサポートするような分散処理ミドルウェアが普及していないので、ユビキタス向けのアプリケーションはそれぞれ個別にハンドオーバー機能を埋め込む他なかった。ところが、アプリケーションごとにハンドオーバー機能を付加しようとすると開発コストが膨大になってしまう。そのようなミドルウェアが存在しないのはユビキタス用の分散処理ミドルウェアの開発が、いまだ発展途上であるためと思われる。

そこで、本稿ではハンドオーバー機能を付加したJava用のRMI(Remote Method Invokation) の提案と実装を行う。ハンドオーバー機能を実現するには主に次の２つの機能が必要となる。

- 使用するサーバの自由な切り替え機能
  分散処理を行っている最中にサーバを自由に切り替えることの出来る機能のこと。

- 通信の切断が起きた場合に自動的にプログラムを待機させ、通信が復旧されたとき処理を再開させる機能

一般的な RMI システムではマスタークラスを元に stub と skelton を生成し、それぞれに埋め込まれた通信プロトコルを用いてリモートメソッド呼び出しを可能にしている。本研究では RMI システムで用いる通信プロトコルに独自の改良を加えてハンドオーバー機能の付加を実現した。そして、RMI を利用するツールとして、それらの通信プロトコルが埋め込まれた stub を自動生成する stub generator の作成と、そこで生成される任意の stub に対応した汎用 skelton である MultiSkelton クラスの作成を行った。

謝辞

本研究を進めるにあたり、研究の方向付けや論文の組立て方についての助言をしていただいた指導教官の千葉先生に感謝いたします。同研究室所属の佐藤芳樹氏には、論文の組み立てからシステムの設計・実装に至るまでの様々な助言を頂き大変感謝致します。また筑波大学の横田大輔氏、東京工業大学大学院の西沢無我氏、中川清志氏、栗田亮氏、宇崎央泰氏には多くの助言を頂きました。東京大学の光来健一氏には論文のスタイルファイルを作って頂きました。以上の方々に重ねて御礼を申し上げます。

目 次

図 目 次

表 目 次

第1章　はじめに

今日、ユビキタス環境の実現に向けてコンピュータ技術、ネットワーク技術の進歩は確実に進歩している。デバイスの小型化、高性能化、省電力化だけではなく、時間や場所を選ばずにインターネット接続ができるようなネットワーク環境など数年前では考えられなかったことが次々と実現されている。そのため近い将来のうちに、我々の身の回りにはコンピューティングデバイス、ネットワークへの接続点、サービスやコンテンツが遍在するユビキタス環境が実現されるだろうと予想される。ユビキタス環境においては単一のネットワークリンクや端末でサービスを享受するものではなく、複数の端末やリンクを跨いだサービスの提供が行われる。将来的にはセンサなどを含めた様々なコンピューティング資源や膨大な量の分散コンテンツに種々のネットワーク資源を介してアクセスを行うことになるだろう。[8, 7] ユビキタス環境とは一種の大規模な分散コンピューティング環境のことであると言える。ユビキタスにおいては、様々なコンピューティング資源同士はネットワークで結ばれ、相互に連携して処理を行っている。そのため、ユビキタス環境の実現のためのプログラミング技術として、分散処理プログラミングは欠かすことの出来ないものであると言える。

しかし、ユビキタス用の分散処理プログラム用支援ツールの開発は今だ発展途上である。そのため、そのような分散処理プログラミングで重要な機能であるハンドオーバー機能をサポートした分散処理用のミドルウェアの開発は我々の知る限り行われていない。ハンドオーバーとは主に携帯電話に用いられている機能のことで、移動中に自動的に基地局の切り替えを行う機能のことである。この機能を分散処理の分野に応用することにより、分散処理中に移動を行っても、問題なく処理の継続ができるシステムが提案できる。ユビキタスコンピューティングにおけるハンドオーバー機能の必要性は以前から言われてきていた。[6] というのも、この機能により、ユーザーのコンテキストに合わせて動的にリソースを変化させることの出来ることになるからである。そこで、このような機

能を持ったアプリケーションの開発がすすめられてきた。しかし、先にも述べたようにそのような機能を付加するような分散処理支援ツールは普及していなかったため、それらのアプリケーションにハンドオーバー機能を付加しようとするとそれぞれ個別にハンドオーバー機能をを埋め込むしか方法がなかった。ところが、アプリケーションごとにハンドオーバー機能を搭載しようとすると開発コストが膨大になってしまう。そのため、そのようなアプリケーションを作成するための支援ライブラリの必要性が生じる。

そこで、我々はハンドオーバー機能を付加した Java 用の RMI（Remote Method Invokation) システムの提案と実装を行う。ハンドオーバー機能は付加するには、RMI システムに以下の 2 つの機能を付加することによって実現できる。

1. 使用するサーバの自由な切り替え機能
   分散処理中に使用するサーバーを自由に切り替えることのできる機能のこと。

2. 通信が切断された際に、自動的に処理を一時待機状態にし、通信が再開されると処理の継続を行えるような機能
   分散処理中に通信が途絶えてしまったら、自動的に処理を一時待機状態にする。その後、通信が再開されるまで処理を待機させておき、サーバ側との通信が再開されたら、処理の続きから行えるような機能。

そこで、本研究ではこの 2 つの機能を埋め込んだ RMI システムの実装を行う。一般的に RMI システムではマスタクラスを元に stub コードと skelton コードを生成し、それらに埋め込まれた通信プロトコルを用いて、リモートメソッド呼び出しを可能にしている。本研究ではその stub コードと skelton コードに埋め込まれる通信プロトコルに独自の改良を施して、ハンドオーバー機能を付加することにした。はじめに基本的な RMI システムを作り、2 の機能を付加することから始めた。具体的には、ネットワーク関連の Exception を用いることによって通信状態のチェックを行い、処理の待機、再開を自動的に行えるような機能を通信プロトコルに埋め込むことを行った。次に、1 の機能を持つ Skelton の作成に取り掛かった。1 の機能を付加するにはサーバ側で何らかのマイグレーション処理を行わなくてはならない。そこで、本研究では処理で用いられてるオブジェクトを全て移動することでサーバの自由な切り替え機能を実現した。ま

た、リモート呼び出しを意識せずに作られたマスタクラスからそのような通信プロトコルを持った stub の自動生成を行う stub generator の作成も行った。なお、本システムではマスタクラスごとに skelton を生成しないで、汎用 skelton である MultiSkelton クラスを作成することにより stub generator で生成される任意の stub に対応するようにした。

第2章　ユビキタスコンピューティングについて

この章ではユビキタスコンピューティングに関する説明を行い、その実現のために必要な技術に関する考察を行う。

2.1　ユビキタスとは

ユビキタスとは「ユビキタスコンピューティング」や「ユビキタスネットワーク社会」などのように使われる、情報化社会の概念的な用語である。ユビキタスは「遍在する、同時にいたるところに存在する」という意味であり、ユビキタスコンピューティングとは日常、どこにいてもコンピュータが使えるという環境である。実際にユビキタス環境が実現されると、日常的な空間の至るところにコンピュータを忍び込ませ、人間の知的活動をサポートするようなことが可能になる。実世界のさまざまな場所に埋め込まれたコンピュータは、それぞれが今どこに置かれているかを理解し、人間がその場所でどのような情報処理を行なっているかを知りさえすれば、人間の行動に合った適切な振舞いをすることができる。このようにユビキタス・コンピューティングでは、コンピュータを実世界にばらまくことによって、パターン認識などの人工知能の技術を導入することなく、状況に依存した高度な情報サービスを提供することができる。

一方、ユビキタスは、大規模な分散コンピューティング環境の事と言い換えることが可能である。なぜならば、それぞれのコンピュータは単独で機能しているのではなく、互いに連携して処理を実現させているからである。もし、それぞれのコンピュータが単独で機能しているのであれば、それはただ単にコンピュータをいたるところに配置しただけにすぎない。つまり、ユビキタスコンピューティングを実現する為には、ただコンピュータを配置するだけではなく、それらのコンピュータをネット

ワークで結び、互いに情報をやりとりして相互に連携しなくてはならない。そのため、ユビキタスを実現するための技術として分散処理プログラミングは欠かせないものとなっている。このような環境を想定すると組み込み機器などの場合でも様々な状況に対応できるような、より複雑な分散処理プログラミングが必要になる。

2.2　ハンドオーバー機能の必要性

2.2.1　ハンドオーバーとは

ハンドオーバーとは、携帯電話や PHS の機能の一つで、携帯電話や PHS での通話中に、電波でデータのやりとりをする基地局を切り替える動作のことである。携帯電話は、基本的には１つの基地局との間で電波を使ってデータのやり取りをしている。このやり取りでの電波の強度を見て、どの基地局とやり取りをするかを決める。例えば、あるセルから移動した場合、基地局からの電波が弱くなるので、ある程度弱くなったらもっと強い電波でやり取りできる基地局がないかを、端末が探しはじめる。近くにもっと強い電波でやり取りできる基地局があった場合、端末と交換機からのネットワークは通信をそちらに切り替えられる。これによって、端末が移動してセルの電波が届く範囲から外れてしまっても、会話を続行することができるわけである。

ようするに、ハンドオーバーとは移動しながら使える仕組みということである。そこで、本研究では**分散処理中に移動しながら使える仕組み**としてハンドオーバー機能という言葉を用いることにする。

2.2.2　ユビキタスに適した分散処理

ユビキタスを実現するためには、いつでもどこでも分散処理を行えるようでなくてはならない。そのため、ユーザーは特定のサーバに捉われずに自分に適したサーバ（通信状態などの条件を加味した）に自由に切り替えられる必要がある。たとえば、例として以下のような状況を考える。

あるユーザーが自分のノート PC を使って無線 LAN で接続されているサーバ A と作業を行っているとする。しかし、そのユーザーがなんらか

の事情により場所を移動しなくてはならなくなった。そこで、ユーザーは自分のノート PC を使ってサーバ A と行っていた作業の続きを移動先のサーバ B を用いて行いたいと思い、ノート PC を持ってサーバ B と接続されている無線 LAN のアクセスポイントまで移動をした。その後、ユーザーはノート PC とサーバ B を用いて作業の続きを続けた。

このような場合はユビキタスが実現されれば非常に多く起こるだろうと考えられる。現状ではユーザー側がノート PC とサーバ A に対し何らかの中断処理を行って、その後にサーバ B の方で処理を再開させなくてはならない。しかし、ユビキタスのような環境を考えるとこのような中断処理の部分は透過的に処理すべきであると考える。移動のたびにユーザーに中断処理を要求することは望ましくない。

そこで、ユビキタスコンピューティングの実現のためにハンドオーバー（ハンドオフ）の必要性が論じられてきた。[6, 8] ハンドオーバーが実現されると、ユーザのコンテキストに合わせて動的にリソースを変化させながら分散処理を行うことが可能になる。このことはユビキタスコンピューティングを実現する上で、大きな進展になるはずである。しかし、現実にハンドオーバー機能を付加したシステムは我々の知る限りでは具体的に実装はされていない。そこで、本研究ではハンドオーバー機能を搭載した RMI システムの具体的な実装を行いたいと思う。

2.3　ハンドオーバー機能を構成する機能

ハンドオーバー機能と、競合するような機能として以下のような研究、技術があげられる。

2.3.1　フォールトトレランス

フォールトトレランスとはコンピュータシステムに障害が発生したときに、正常な動作を保ち続ける能力のことである。言い換えれば、障害発生時の被害を最小限度に抑える能力のことである。また「耐故障性」「故障許容力」などと表現されることもある。

実際のところ、フォールトトレランスというと具体的にハードウェアの 2 重化構成を指すのが一般的である。深刻な故障が発生してもシステムの運用が影響を受けないように、あらかじめハードウェアを 2 重化し

ておくことを意味することが多い。高信頼性を実現するためには、ハードウェアを吟味して品質の高いものだけで構成したり、こまめに診断を行なって重大な障害に発展する前の兆候をつかんで予防的に部品交換を促す、などの手法が取られる。フォールトトレランスを実現するということは、ここからさらに一歩進み、どちらにしても故障は避けられないという観点から、あらかじめ故障に備えて予備を用意しておく、ということである。[2, 3] 分散処理を行う上でのフォールトトレランスを意識した手法として代表的なものに以下のようなものがある。

チェックポイント取得手法

チェックポイント取得手法では、プロセス実行中のある時刻 (これをチェックポイントという) に、そのプロセスの状態を安定な記憶に保存する。実行中に障害が発生し、プロセスが正常な動作を続けられなくなった場合には、保存しておいたプロセスの状態を用いて回復を行い、直前のチェックポイントから実行を再開する。 チェックポイント取得手法は以下の 3 つの手法に分類される。

1. **システムレベルでの取得**
   システムレベル (system-level) のチェックポイント取得では、プロセスのメモリ空間のイメージをチェックポイントとして用いる。プロセスのメモリ空間を保存する機能はライブラリとして提供できるため、プログラムのソースコードをほとんど変更せずにチェックポイント取得を実現できる場合が多い。
   一方、この手法ではメモリ空間を 1 つのイメージとして扱うため、基本データ型 (例えば int 型) などの表現方法が異なるノードからなる異種 (heterogeneous) システムに適用することは不可能である。また、実行再開のために必要のないデータもチェックポイントに含まれるため、チェックポイントのサイズが大きくなるという問題もある。

2. **ユーザ定義による取得**
   ユーザ定義のチェックポイント取得では、ライブラリによってチェックポイント取得のサポート機能を提供する。チェックポイントに含めるデータとチェックポイント取得タイミングを指定するコードを、

ユーザがプログラムのソースコードに付加することにより、チェックポイント取得を実現する。
ユーザは実行再開のために必要な情報を識別できるため、この手法ではチェックポイントのサイズを小さくすることができる。また、ライブラリで提供する機能にデータ表現方法などの変換機能を付加することにより、異種システムでも利用可能なチェックポイント (ポータブルチェックポイント) 取得を実現できる。
しかしながら、チェックポイント取得のためのコードを付加するのは一般に手間のかかる作業であり、大きなプログラムに対しては実際的でない。これはまた、本来のコード開発の妨げとなり、プログラムの品質低下や開発期間の増大を招く要因となる。

3. **コンパイラ支援による取得**
   コンパイラ支援 (compiler-based) のチェックポイント取得では、プログラムをコンパイルする際にコンパイラがチェックポイント取得コードを自動的に生成し、ソースコードに付加する。データ表現などの変換コードを生成することにより、ポータブルチェックポイントの取得も可能である。
   ただし、単純な実装では実行再開に不必要なデータまでチェックポイントに含めてしまう。これを解決するため、必要なデータのみを識別する試みがなされている。しかしながら、完全に必要なデータのみを識別するのは困難であり、一般にデータサイズは必要以上のものとなってしまうことが多い。

レプリケーションを用いた手法

レプリケーションとはプロセスのレプリカを 1 つ以上生成し、何らかのレプリカ管理機構を用意してレプリカを管理する仕組みである。レプリカ管理機構が各プロセスの状態をチェックし、誤りを検出した場合には正常なレプリカを用いて対処する。代表的なものとして以下の 3 つの手法があげられる。

1. **アクティブレプリケーション**
   アクティブレプリケーションでは異なるノードにレプリカを生成する。すべてのレプリカはアクティブであり、各ノードで同じ計算を

同時に実行する。レプリカ管理機構は、多数決によりすべてのレプリカの出力から 1 つの出力を決定する。レプリカに障害が発生し誤った出力を行った場合には、多数決機構によりその障害がマスクされる。アクティブレプリケーションでは、各ノードの出力により障害を検出するため、システムを構成するノードのアーキテクチャの違いは問題とならない。しかしながら、すべてのレプリカが同じ処理を行うために、多くの計算機資源を消費してしまうのが大きな欠点である。

2. **セミアクティブレプリケーション**
   セミアクティブ・レプリケーション手法でも異なるノードにレプリカを生成する。すべてのレプリカはアクティブであり、各ノードで同じ計算を同時に実行する。アクティブレプリケーションとは異なり、1 つのレプリカがリーダ (leader) となり、残りのものがフォロワ (follower) となる。
   出力の決定はリーダが行い、リーダに障害が発生した場合には 1 つのフォロワが新しいリーダとなる。セミアクティブレプリケーションでは、アクティブレプリケーションと同様にノードの出力により障害を検出するため、ノードのアーキテクチャの違いは問題とならない。しかしながら、多くの計算機資源を消費してしまうのが大きな欠点である。

3. **パッシブレプリケーション**
   パッシブ・レプリケーションでは、計算を実行するのはただ 1 つのアクティブなプライマリ・レプリカである。残りのレプリカはバックアップとなる。
   プライマリは定期的に計算の状態を保存し (チェックポイントの取得)、ネットワークを通してそれをバックアップに送信する。バックアップは受信したチェックポイントを保存し、それを用いて自身の状態を更新する。プライマリに障害が発生すると、1 つのバックアップが新しいプライマリとなり、最新のチェックポイントから実行を再開する。
   パッシブレプリケーションでは、ポータブルチェックポイントを用いることで異種システムに対応できる。しかしながら、バックアッ

プの状態を更新していく仕組みが必要になり、レプリカ管理機構が複雑になる。また、チェックポイントの転送によりネットワークのバンド幅を消費してしまう

2.3.2　モバイルエージェント

モバイルエージェントとはデータ+プログラムのオブジェクトであり、ネットワーク上を移動して様々なサーバ上で処理を行い、送り出し元のサーバに処理結果を持ち帰るものである。モバイルエージェントは、内部状態を保持したまま計算機間を自律的に移動し、そのつど処理を行い、移動先での処理結果に応じて、エージェントの行動プラン (次の移動先、処理の内容) は動的に変化する。分散システムを構築する際、モバイルエージェント技術を利用すれば、クライアント/サーバ型分散システムと異なり、ソフトウェアコンポーネントの役割が柔軟なシステムを開発することができる。

モバイルエージェントの移動

モバイルエージェントの移動は以下の手順で行われる。

1. 現在いるホストでモバイルエージェントの実行を停止させる。
2. データを直列化する。
3. 相手先のホストに送信する。
4. 移動先でデータを複合化しモバイルエージェントに変換する。
5. モバイルエージェントの実行を再開する。

移動先で実行を再開するためには、プログラムコードと、プログラムの状態を表すデータを移動しなければならない。モバイルエージェントの移動には以下の 2 種類がある。

1. プログラムコードの移動
2. 実行コンテキストの移動

1 の場合、以下の手法がある。

- 移動先の計算機にも必要となるプログラムコードをインストールしておく手法
  - プログラムコードを転送しなくて良い分効率が良いが、あらかじめインストールしておく手間がかかる。
- 実行中に必要に応じてプログラムコードをダウンロードする手法
  - あらかじめインストールしておく手間が必要ないが、移動時にネットワークに負荷がかかる。また、必要になった時点で動的にクラスをロードするので、移動後も必ずネットワークに接続されている必要がある。

2 の場合、以下の手法がある。

- プログラムコードのみの移動
  - プログラムコードだけが移動し、実行コンテキストは移動しない。移動先では初期状態から処理が再開される。
- プログラムとヒープ領域
  - プログラムコードと、インスタンス変数などのヒープ領域内の情報を移動させる。ヒープ領域内の情報しか転送しないので移動コストが少ない。しかし、プログラムカウンタやローカル変数は対象とならないので、移動直前の状態から実行を再開するのは難しい。
- 全実行コンテキスト
  - プログラムコードとヒープ領域内の情報に加えて、ローカル変数などのスタック領域やプログラムカウンタなども移動させる。全ての実行コンテキストを移送するので、移動後も移動直前の実行個所から処理を継続することができる。移動性に備えた特別な記述が不要で、プログラミングインターフェースの透過性が高い。

既存のモバイルエージェントの分類

既存のモバイルエージェントは以下のように分類することができる

1. プログラムコードのみの移動
    - Java Applet
2. ヒープ領域内の情報を移動
    - Voyager
    - Aglets
3. 実行中のスタック領域内やプログラムカウンタも移動
    - Moba

2.3.3　TCP／IP

TCP/IP とは（Transmission Control Protocol/Internet Protocol）の略で、もともとはアメリカの国防総省（DOD）の主導で DARPA（Defense Advanced Research Project Agency）という機関が設立され、国防のためのコンピュータネットワークの研究から生まれたプロトコルである。ところが、カルフォルニア大学バークレイ校で 1981 年ごろにコンピュータのオペレーティンクシステムである UNIX（4.1BSD）にこのプロトコルを実装したことから、UNIX の標準通信プロトコルとして広がり、最近は UNIX 以外のコンピュータでも標準的に使えるようになってきた。現在では世の中で最も普及した「事実上の標準」プロトコルとしての地位を築くようになっている。TCP/IP の特長は様々であるが、「インターネット」の標準プロトコルであるということはは特に重要な特長と言える。

TCP ／ IP は一種類のプロトコルではなく、Internet Protocol(IP) を中心とした、プロトコルのファミリーからなっている。そのほかのメンバーは、Transmission Control Protocol (TCP)、User Datagram Protocol (UDP)、Address Resolution Protocol (ARP)、Reverse Address Resolution Protocol (RARP)、Internet Control Message Protocol (ICMP) となっている。

TCP ／ IP の階層

TCP ／ IP のプロトコル郡は次の図のような階層をなしている。データを送る側にも、データを受け取る側にも、同じ階層のプロトコルがあり、データを送るときには、高い階層から低い階層にデータが渡され、データを受け取るときには、低い階層から高い階層にデータが渡される。この時、高い階層から低い階層にデータが渡されるにつれ、それぞれの階層のアドレス等の通信制御情報が付け加えられ、逆に、低い階層から高い階層にデータが渡されるときには、これらの制御情報は取り除かれる仕組みなっている。

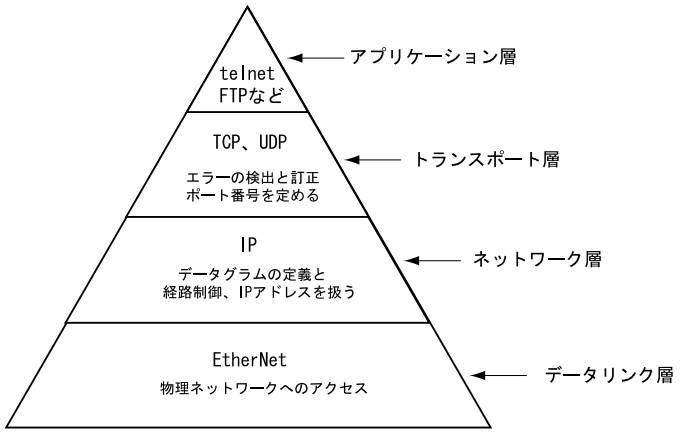

図 2.1: TCP ／ IP の仕組み

- **データリンク層**
  データリンク層は、物理的に接続されたコンピューター間のデータの通信を行う。イーサーネットが標準的なデータリンク層のプロトコルになる。ここでは、物理的な イーサーネット・アドレスが用いられている。この層と次のネットワーク層の中間に、IP アドレスをイーサーネット・アドレスに変換する ARP プロトコルや、この逆の、イーサーネット・アドレスを IP アドレスに変換する RARP プロトコルが含まれている。

- **ネットワーク層**
  IP (Internet Protocol) が、ネットワーク層に属することになる。この層はマシンとマシンの間の通信プロトコルを提供している。IP

は、伝送経路の確立や、IP アドレス・クラスによるネットワークの論理的な管理等を担当している。IP ヘッダーには、様々な情報が含まれるが、その中で最も重要なものは、データの送り手と受け手のホストの、それぞれの IP アドレスである。

- **トランスポート層**
  この層は、別々のマシンのそれぞれのプロセスのあいだの通信を保障する。この層には、TCP や UDP が含まれる。TCP は、通信中にエラーが起こればデータの再送をするなど、確実なデータの伝送を保障している。しかし、その分 TCP は、コネクションの確立などに手間がかかってしまう。UDP では、エラー回復は行われず信頼性は劣るが、コネクションなしのデータグラムの通信を行う。TCP も UDP も、プロセスを特定するのに、ポート番号を使っている。

- **アプリケーション層**
  クライアント側で特定のサービス（例：TELNET,FTP）を使用しているユーザプロセス、サーバ側にある同じサービスを提供するプロセス（例：TELNET,FTP サーバ）はこの層に属している。

TCP の機能

相手のコンピュータの通信処理能力の問題や様々な原因で、パケットが正しく送られなかったり、あるいはデータそのものが壊れてしまう場合、データが正しく送られない可能性がある。そこで必要になるが TCP の機能である。TCP はデータを送信する前に送信するデータの分量を計算し（チェックサム）受信側はそのチェックサムを元に送られてくるデータが正しいかどうかを確認する。また、データが確実に相手に届いていれば、データが届いたという通知のパケット（ACK パケット）を送信元に送り返す仕組みを持っているのだ。もし一定時間以内に相手から ACK パケットが戻ってこない場合は、再びデータを送り出す。

パケット通信では単に分割されたパケットが相手に届くだけでは機能しない。それが組み立てられ、復元されて始めてデータがきちんと送られることになる。データは送られてくる順番どおり組み立てられるので、データの送られてくる順番も非常に大切な役割を果たすことになる。そこで、パケットに正しい並び順（シーケンシャルナンバー）を付与してその順番どおりにパケットが送られるかどうか確認し、同時に必要であれば並び替えが行われるのだ。

TCP はこのように確実にデータが届かなくてはならないものに関しては非常に信頼性が高いがその分スピードは遅くなる。そこでこのような厳密なチェックの必要のない通信のやり取りの場合、TCP ではなく UDP が使われている。

2.4　ユビキタスコンピューティングに必要な関連技術

2.4.1　JavaRMI

JavaRMI とは

RMI（Remote Method Invocation）は他のマシンにある Java で記述されたメソッド（リモートメソッド）をネットワークを介して呼び出す仕組みである。RMI のような分散オブジェクト技術を使用することによりネットワークにまたがる分散アプリケーションの構築が可能になる。JavaRMI には以下のような特長が備わっている。

- 変更に対する柔軟性
  RMI ではサーバとクライアントは Java のインタフェースで結合される。これはインタフェースが変更されない限り、サーバの処理に変更があっても、クライアントを変更する必要がなく、また、サーバの変更なしでクライアントを変更できることを意味する。したがって RMI では、このようにサーバ、クライアントの変更に対し柔軟なシステムを構築できる。

- 記述の容易性
  RMI を用いてサーバ、クライアント・プログラムを記述することは簡単である。サーバ・プログラムでは、それが RMI のサーバであることを宣言するための数行のコードを記述するだけである。クライアント・プログラムではサーバ・プログラム、すなわちサーバ、オブジェクトは、リモート・オブジェクトの参照として得られるが、これは通常のオブジェクトと何ら変わりなく取り扱うことが出来る。

- 可搬性
  RMI は JDK1.1 以降でコアパッケージとして提供されている。したがって、RMI を使用して作成されたプログラムは JDK1.1 以降の環

境があれば、他のどのようなシステムにおいても動作させることが出来る。

- データ送受信の容易性
  RMI ではメソッドを呼び出す際の引数、戻り値として、オブジェクトを受け渡すことが出来る。したがってハッシュテーブルのような複雑なデータ構造でも、ただ１つの引数として取り扱うことができる。例えばソケットを使用してデータを転送する場合と比較して、データの分解、再構成といった、余分なコードは一切不要となる。

- 並列処理
  RMI はマルチスレッド対応であり、複数のクライアントの要求は並列に処理される。

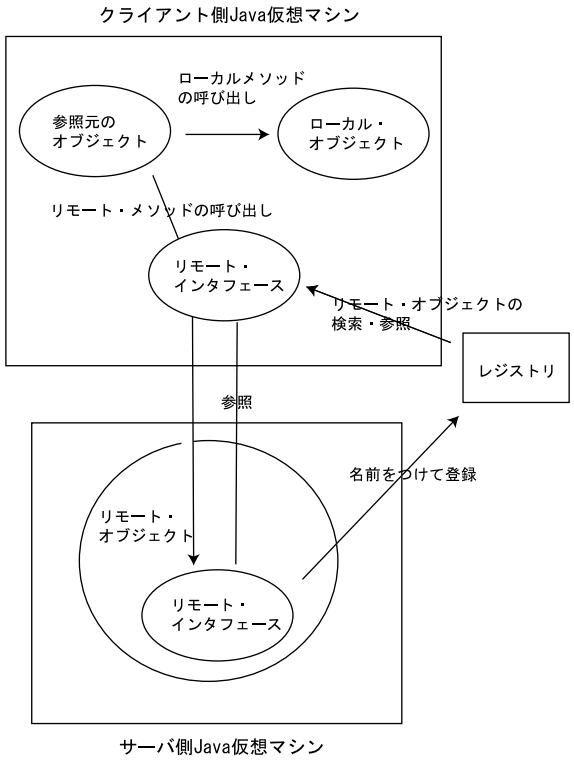

図 2.2: RMI の仕組み

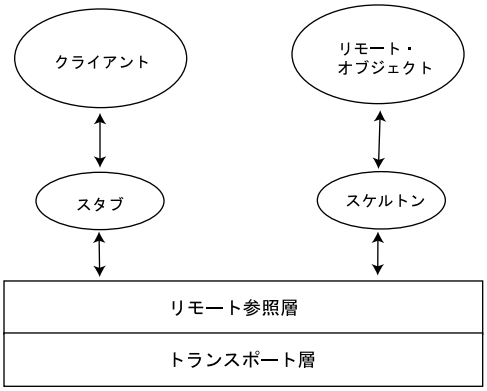

図 2.3: RMI の仕組み 2

RMI の用語の説明

- リモートインタフェース
  Remote インタフェースを直接または間接的に継承したインタフェース。他の Java 仮想マシンから呼び出されるメソッドはここに定義する必要がある。

- リモート・オブジェクト
  リモート・インタフェースを実装したクラスのオブジェクトで、かつ他の Java 仮想マシンからそのリモート・インタフェースが参照可能なもの。

- リモート・メソッド
  リモート・オブジェクトに実装されたメソッドで、かつリモート・インタフェースに定義されているもの。

- ローカルオブジェクト
  参照元のオブジェクトと同一の Java 仮想マシン上にあるオブジェクト。

- ローカル・メソッド
  ローカル・オブジェクトに実装されたメソッド。

- レジストリ
  名前付けされたリモート・オブジェクトの登録、検索を行うネーミングサーバ。サーバはリモート・オブジェクトに名前を付けてレジ

ストリに登録し、クライアントはレジストリに対して名前でリモート・オブジェクトを検索し、その参照を得る。

RMI の具体的な使用例

リモートインタフェース

- public である。
- Remote インタフェースを継承する。
- RemoteException 例外を送出する。

リモート・インタフェース (Hello.java)

```
public interface Hello extends Remote{
     public String getMessage() throws RemoteException;
}
```

サーバ・クラスの記述

次に Hello インタフェースを実装する HelloServer クラスを記述する。

リモート・インタフェース実装クラス (HelloServer.java)

```
import java.rmi.*;
import java.net.*;

public class HelloServer extends UnicastRemoteObject
     implements Hello{

     public HelloServer() throws
     RemoteException{
           super();
     }

     public String getMessage()
     throws RemoteException{
           return "Hello!!";
     }
}
```

レジストリへの登録

ここではセキュリティマネジャの設定と、サーバ・クラスのインスタンス化と登録を行う。

オブジェクト・エクスポートクラス (Launcher.java)

```
import java.rmi.*;
import java.net.*;

public class Launcher{
    public static void main(String[] args){
         //セキュリティマネジャの作成と設定
         System.setSecurityManager(new RMISecurityManager());
         try{
            //サーバ・オブジェクト作成
            HelloServer obj = new HelloServer();
            //リモート・オブジェクトの登録
            Naming.bind("rmi://localhost/HelloServer",obj);
         }catch(Exception e){
            e.printStackTrace();
         }
    }
}
```

クライアント・プログラム

クライアント・プログラムではリモートオブジェクトの取得とリモートメソッドの呼び出しを行う。

クライアント・プログラム (Client.java)

```
import java.rmi.*;
import java.net.*;

public class Client{
    public static void main(String[] args){
         Hello remoteObject;
         try{
          remoteObject=(Hello)Naming.lookup("rmi://localhost/HelloServer");
         }catch(Exception e){
          e.printStackTrace();
```

```
        }
        try{
          String message = remoteObject.getMessage();
        }catch(Exception e){
         e.printStackTrace();
        }
    }
}
```

コンパイルと実行

1. ソースコードをコンパイルする。
2. スタブとスケルトンを生成する (> rmic HelloServer)
3. レジストリプログラムの実行 (> rmiregistry)
4. サーバプログラムの実行 (> java Launcher)
5. クライアントプログラムの実行 (> java Client)

2.4.2　Object Serialization

オブジェクトの直列化とは

Serialization（直列化）はオブジェクトを入出力のストリームを通じてやり取りできるようにする仕組みである。オブジェクトの情報を直接ファイルに保存したり、ネットワークを通じて送ったりすることを可能にする。**Object Serialization（オブジェクトの直列化）**とは、オブジェクトとそこから参照されているオブジェクトを、バイトストリームにコード化することである。このことによって、基本データ型、文字列型の入出力ストリームと同様に、オブジェクトを入出力することが出来る。

例

```
Myclass c = new Myclass();
c.value = 1;
```

というコードを実行したとすると、オブジェクト c が持つ変数 value の値は、実行中は「１」という値を保持するが、実行を終了するとメモリから開放されてしまう。もう一度同じオブジェクトを別のプログラムから実行しても「value=1」という情報は復元されない。しかし、Serialization を使うことにより、この「変数値の復元」が可能になる。

直列化できるオブジェクト

直列化できるオブジェクトは、java.io.Serializable インタフェースを実装するクラスか、それを継承したサブクラスから作られたものである。このインタフェースを実装したクラスから作られたオブジェクトは、ObjectInputStream/ObjectOutputStream によって入出力可能である。こうして作られた直列化可能オブジェクトの入出力では、このオブジェクトが保持している他のオブジェクトへの参照を自動的に辿って、関連するオブジェクトが全て入出力される。オブジェクトを直列化するときに、直列化しないフィールドを宣言しておくこともできる。また、直列化プロセスを明示的に制御することも可能である。この場合は、Serializable インタフェースを継承した java.io.Externalizable インタフェースを実装する。オブジェクトを直列化／復元することで、オブジェクトをファイルに保存して持続的 (persistent) に利用したり、ネットワーキングでソケットストリームを用いてリモートホスト間でやりとりすることが出来る。

Serialization の手順

Serialize 可能なデータは Serializable（Externalizable) インタフェースを実装したクラスと、そのサブクラスのインスタンス、あるいは int,double,long などの基本データ型や配列である。

出力側

- 出力用のストリームを用意
- 通信の場合、ByteArrayStream を使う
- ストリームから、ObjectOutputStream を作る
- ObjectOutputStream に writeObject によって書き込む

入力側

- 入力用のストリームを用意
- ストリームから ObjectInputStream を作る
- ObjectInputStream から readObject によって読み込む

2.4.3　java.lang.reflect

リフレクションとはクラスのフィールド、メソッド、およびコンストラクタに関する情報を検出したり、その検出した情報でそのクラスを利用したりできる API である。リフレクションを用いることにより、以下のようなことが可能になる。

- 新規クラスのインスタンスおよび新規配列の生成
- オブジェクトおよびクラスのフィールドへのアクセスと変更
- オブジェクトおよびクラスに関するメソッドの呼び出し
- 配列要素へのアクセスと変更

アプリケーション

Reflection API を使うと、次のような 2 種類のアプリケーションを作成できるようになる。

1 つは、実行時のクラスに基づくターゲットオブジェクトの、すべての public メンバを見つけ出して使う必要があるアプリケーションの集合である。これらのアプリケーションは、オブジェクトのすべての public フィールド、メソッド、およびコンストラクタに実行時にアクセスする必要がある。このカテゴリに入るアプリケーションには、Java Beans、あるいはオブジェクトインスペクタなどの簡易ツールがあげられる。これらのアプリケーションは、クラス Class のメソッド getField、getMethod、getConstructor、getFields、getMethods、および getConstructors から取得したクラス Field、Method、および Constructor のインスタンスを使う。

もう 1 つは、特定のクラスが宣言したメンバを見つけ出して使う必要のある、複雑なアプリケーションの集合である。これらのアプリケーションは、class ファイルが指定したレベルのクラス実装への実行時アクセスが必要になる。このカテゴリに入るアプリケーションには、インタ

プリタ、インスペクタ、クラスブラウザなどの開発ツールや、オブジェクト直列化などの実行サービスがあげられる。これらのアプリケーションは、クラス Class のメソッド getDeclaredField、getDeclaredMethod、getDeclaredConstructor、getDeclaredFields、getDeclaredMethods、および getDeclaredConstructors から取得したクラス Field、Method、および Constructor のインスタンスを使用する。

第3章　システムの設計と実装

この章ではハンドオーバー機能付き RMI の設計と実装の状況について説明をする

3.1　ハンドオーバー機能付き RMI

本研究ではハンドオーバー機能をもった RMI システムの開発を行っている。ハンドオーバー機能を備えた RMI システムとは通常の RMI システムに以下のような機能を付加したものである

1. プログラム実行中にサーバ・クライアント間で通信が途切れた場合に処理を自動的に待機状態にする機能
    - 中断に関する処理を行わずにサーバとの通信を切った場合、クライアント・サーバの両方の処理を自動的に待機状態にする。サーバとのコネクションが新たに確立されたら、クライアント側、サーバ側で待機状態になっている処理を自動的に再開させる。この機能により特別な処理をユーザー側に要求することなく、自由にサーバ・クライアント間のコネクションを ON／OFF にすることができる。
2. 複数のサーバを用意した場合、サーバを自由に切り替えて処理を継続できる機能
    - あるサーバ A とクライアント X で行っていた処理を一度中断するとする。その後、別のサーバ B とクライアント X でサーバ A との間で行われていた処理の続きを行えるようにする。この仕組みを用いることにより、特定のサーバにとらわれずに分散処理を行うことが用意になる。

以上の 2 点を組み合わせることにより、ハンドオーバー機能が実現できるものと我々は考える。そこで、具体的には以下の手法を用いて、これらの機能を実現することにした。

1. リモートメソッド呼び出しが正常に終了するまで、繰り返す retry 機能

2. サーバ間でのオブジェクトの移動を可能にする機能

１の機能を用いることにより、リモートメソッド呼び出しは通信が正常な限り、確実に行われ、２の機能を用いることにより、使用するサーバの移動が可能になる。

3.2　リモートオブジェクトの参照と実行およびリモートメソッドの呼び出し

RMI では、リモートホストでのオブジェクトの生成、メソッドの呼び出しをローカルホスト上のプロセス内部であるかのように扱うことができる。これらの処理を実行するにあたり、リモートホストへリクエストを送受信するために、Java API の java.io パッケージや java.net パッケージを利用し専用の通信プロトコルを形成している。ハンドオーバー機能を付加させるためには、このプロトコルの部分を独自のものに作り変えなくてはならない。この節では本システムで行っているリモートオブジェクトの生成、参照、リモートメソッドの呼び出しをするために必要なプロトコルと実装の基本的な部分（ハンドオーバー機能を省いた部分）の説明をする。

3.2.1　リモートオブジェクト生成と参照の実装

リモートホスト上にオブジェクトを生成するためには、クラスを参照し、そのクラスのコンストラクタ引数の型と値を送信する機能が必要である。これはリモートホスト上でオブジェクトの生成が、Java の Reflection API を利用しているためである。Java の Reflection API を利用すれば、任意のクラスに対してオブジェクトを生成することができる。もし仮に Reflection API を利用せずにオブジェクトの生成をしていたとすると、オブジェクト生成側のコードは、オブジェクトの元になるクラスに依存

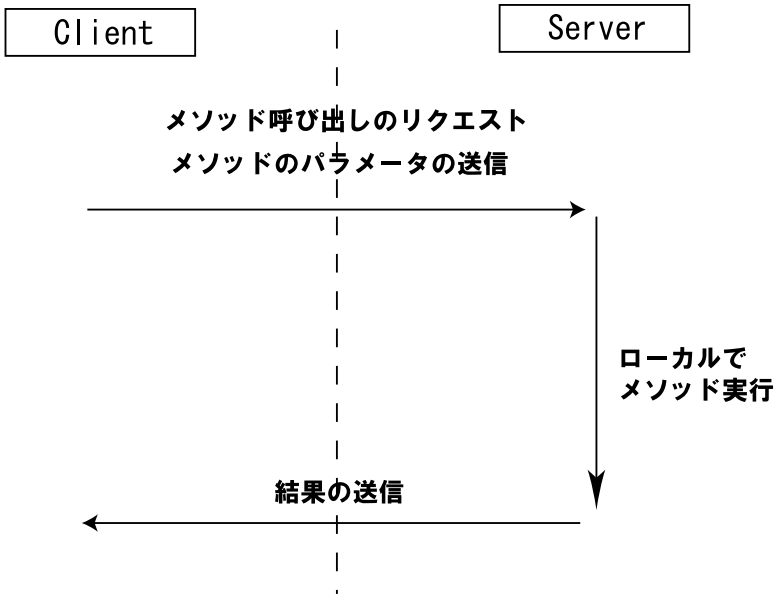

図 3.1: リモートメソッド呼び出しの仕組み

する形となってしまう。このため、違うクラスのオブジェクトを生成するごとにオブジェクト生成側のコードを修正しなくてはならないため、プログラムの保守性が低下してしまうのである。生成されたリモートオブジェクトの参照は、リモートメソッド呼び出しの際に必要になるため、いつでも取り出せるようになっている事が望ましい。そのため、オブジェクトの登録領域を用意しそこに格納しておき、クライアントには格納している場所を表すデータを返しておく必要がある。

objectList の利用によるオブジェクトの保存

本システムでは JavaRMI の rmiregitstry のような特定のオブジェクト管理サーバを使用せずに、オブジェクト参照テーブルを用いることにより生成されたリモートオブジェクトの管理を行っている。具体的にはクライアントがリモートホスト上に新たなオブジェクトを作るように要求を出すと、オブジェクトを生成するプログラムである Skelton サーバは、下記のようなリモートオブジェクト生成のための通信プロトコルを利用してリモートオブジェクトを生成する。そのオブジェクト参照を Skelton サーバは、オブジェクト参照テーブルに登録し、登録に必要な objectID（key）をクライアン

トに送信する。なおオブジェクト参照テーブルは java.lang.util.LinkedList クラスを用いて実装している。以下は、リモートオブジェクト生成とその参照に必要な、クライアント側（stub）の通信プロトコルの実装とリモートホスト側のオブジェクト生成プログラムである、Skelton サーバの通信プロトコル部分である。

```
/** Client **/
    ObjectOutputStream oos
        = new ObjectOutputStream(socket.getOutputStream());
    ObjectInputStream ois
        = new ObjectInputStream(socket.getInputStream());
    oos.writeObject(classname);
    oos.writeObject(paramtypes);
    oos.writeObject(paramvalues);
    int objectid = ois.readInt();
```

クライアントは、クラス名（String 値の classname）、コンストラクタ引数の型（Class[] 値の paramtypes）、コンストラクタ引数の値（Object[] 値の paramvalues）を通信プロトコルとして送信すればよい。クライアントは Skelton サーバから遠隔オブジェクトを参照するために必要な objectid（key）を受信することにより、いつでもその遠隔オブジェクトを呼び出すことができる。次に、リモートホスト上のオブジェクト生成プログラム（Skelton サーバ）の通信プロトコルを以下に示す。

```
/** SkeltonServer (remote object creater) **/
    ObjectInputStream ois
        = new ObjectInputStream(socket.getInputStream());
    String classname = (String) in.readObject();
    Class[] paramtypes = (Class[])ois.readObject();
    Object[] paramvalues = (Object[])ois.readObject();
    Class targetclass = Class.forName(classname);
    Constructor targetcons
        = targetClass.getDeclaredConstructor(paramtypes);
    Object targetobj = targetcons.newInstance(parametervalues);
    int objectid = registierObject(targetobj);
    oos.writeInt(objectid);
```

Skeltonサーバはクライアントから生成するオブジェクトのクラス名（String値のclassname）、コンストラクタ引数の型（Class[] 値のparamtypes）、コンストラクタ引数の値（Object[] 値のparamvalues）を受け取って、Javaの Reflection　API により指定されたオブジェクトを生成している。その後、Skeltonサーバはオブジェクト参照テーブルに生成したオブジェクト参照を登録し、objectid(key) を得る。そして、そこで得られたobjectid（key）をクライアントに送信している。

3.2.2　リモートメソッド呼び出し

リモートメソッドを呼び出すためには、メソッドの引数の型と値をネットワーク越しに送信する機能が必要である。この理由も、リモートオブジェクトの生成の時と同様、メソッド呼び出しにもJavaのReflection API を利用するためである。本システムでは、メソッド呼び出しにReflectionを利用することにより、任意のクラスのメソッドをオブジェクトの元になるクラスに依存することなく、呼び出すことができる。以下で、遠隔メソッド呼び出しに必要なクライアント側の通信プロトコルの実装を記す。

```
/** Client **/
    ObjectOutputStream oos
        = new ObjectOutputStream(socket.getOutputStream());
    ObjectInputStream ois
        = new ObjectInputStream(socket.getInputStream());
    oos.writeInt(objectid);
    oos.writeObject(classname);
    oos.writeObject(methodname);
    oos.writeObject(paramtypes);
    oos.writeObject(paramvalues);
    Object ret = ois.readObject();
```

クライアントは、オブジェクト参照テーブルからオブジェクト参照を取り出すint 値のobjectid（key）、String 値のクラス名classname、String 値のメソッド名methodname、メソッド引数の型（Class[] 値のparamtypes）、メソッド引数の値（Object[] 値のparamvalues）をメッセージプロトコルとして送信し、そのメソッドの結果が戻ってくるのを待つ。一方、リモートメソッドを呼び出すプログラム（Skeltonサーバ側）は以下のようになる。

```
/** SkeltonServer (remote method invocation) **/
    ObjectInputStream ois
        = new ObjectInputStream(socket.getInputStream());
    int objectid = ois.readInt();
    String classname = (String)ois.readObject();
    String methodname = (String)ois.readObject();
    Class[] paramtypes = (Class[])ois.readObject();
    Object[] paramvalues = (Object[])ois.readObject();
    Object targetObject = getObject(objectid);
    Class targetClass = Class.forName(className);
    Method targetmethod
        = targetClass.getDeclaredMethod(methodName, paramtypes);
    Object ret = targetMethod.invoke(targetObject,paramvalues);
    oos.writeObject(ret);
```

Skelton サーバでは、クライアントからのメッセージプロトコルを受信し、java.lang.reflect.Method クラスにより指定のメソッドを呼び出す。その後、メソッドの実行結果をクライアントに送信することでリモートメソッド呼び出しが完了する。

3.3　Skelton サーバの実装

Skelton サーバは、遠隔オブジェクトの生成や参照の保持、遠隔メソッドの呼び出しを実行するためのプログラムであり、リモートホスト上で実行されるサーバとして実装される。また、クライアントからの複数のリクエストにも同時に対応できるように、Skelton サーバはマルチスレッド化されることが望ましい。以下にマルチスレッドを用いて実装した Skelton サーバのプログラムの主要部を示す。

```
/** Skelton Server(MultiThread part) **/
  public class SkeltonServer extends Thread {
    public SkeltonServer (int port) {
      ServerSocket ss = new ServerSocket(port);
      public void run() {
        for( ; ; ) {
          ・
```

```
          .

          final Socket socket = ss.accept();
          new Thread() {
            public void run() {
              while (true) {
                requestHandler(socket);
              }
            }
          }.start();
          .
          .
        }
      }
    }
  }
```

以上のようなプログラムを用いて Skelton サーバはクライアント（stub）からの要求を受け取る。そして、requestHandler において送られてくるデータごとに処理の切り替えが行われる。具体的にはこの requestHandler 内の action_flag によって、通信プロトコルの切り替えを行うようにする。

```
/** requestHandler (in Skelton Server) **/

private void requestHandler(Socket s) {
    ObjectInputStream ois
         = new ObjectInputStream(s.getInputStream());
    ObjectOutputStream oos
         = new ObjectOutputStream(s.getOutputStream());
    int action_flag = ois.readInt();
        switch(action_flag){
            case OBJECT_CREATION:
            .
            .
            break;
            case METHOD_INVOKATION:
            .
```

```
            .
            break;
            .
            .
        }
      .
      .
    }
```

Skelton サーバは始めにクライアント側からヘッダとして送られてくる action_flag の値を読み取る。そして、その読み取った値に応じて通信プロトコルを切り替え、クライアント側の要求に応えたデータの送信を行う。

3.4　分散処理中の切断のタイミングのパターン化

ハンドオーバー機能を実装するには、ユーザーが前処理を行うことなく、通信を切断できなくてはならない。そこで、ハンドオーバー機能付の RMI を実現させるには、ネットワークの切断するタイミングに影響されることなく、処理を継続して行えなくてはならない。つまり、再接続にいたるまでの一連の作業を透過的に RMI 側の方で実現することが必要になる。具体的には切断のタイミングを何パターンか想定し、それぞれのパターンに応じて、専用の通信プロトコルを用いることにより作業の復旧を行うようにする。まずはリモートメソッド呼び出しを用いた分散処理中における切断のタイミングをいくつかに場合分けして説明する。

切断するタイミングについて

1. **リモートオブジェクトの生成に必要なパラメータの通信を行っているときに通信が途切れてしまった場合**

   本システムにおける RMI 部分は Reflection API を用いて任意のメソッド呼び出しを可能にしている。ここで、その Reflection API を使う際に必要な情報（Class[] の引数の型など）を送っている時に通

信が途切れてしまう場合が考えられる

2. **リモートメソッド呼び出しを行うのに必要なパラメータの通信を行っているときに通信が途切れてしまった場合**

   この場合も 1 の場合と同様でリモートメソッド呼び出しをする際にも Reflection　API で必要になるパラメータ等の通信を行わなくてはならない。その為、それらの情報をやり取りする際に通信が途切れてしまう場合が考えられる。

3. **リモートメソッドの戻り値をクライアント側が待っているときに通信が途切れてしまう場合**

   これは、サーバ側が呼ばれているリモートメソッドをローカルで実際に呼び出している時に、通信が途切れてしまう場合のことである。RMI においてはリモートオブジェクトを用いてリモートメソッド呼び出しを行うとき、クライアント側からはあたかもローカルオブジェクトを用いてローカルメソッドを呼び出しているように見えるわけだが、実際はサーバ側でローカルオブジェクトを生成し、ローカルメソッドを呼び出している。そこで、サーバ側はパラメータ等の必要な情報を得ると、Reflection　API を用いてローカルオブジェクトからローカルメソッド呼び出しを行う。そのため、サーバ側でメソッド呼び出しを行っている最中に通信が途切れる場合も考えなくてはならない。

以上の 3 パターンに場合分けをして、それぞれに専用の復旧通信プロトコルを用意する。

3.5　ネットワーク切断時に発生する Exception に関する考察

JavaRMI において、リモートメソッド呼び出しの実行中に発生する通信関連の例外は共通のスーパークラス RemoteException を持っている。

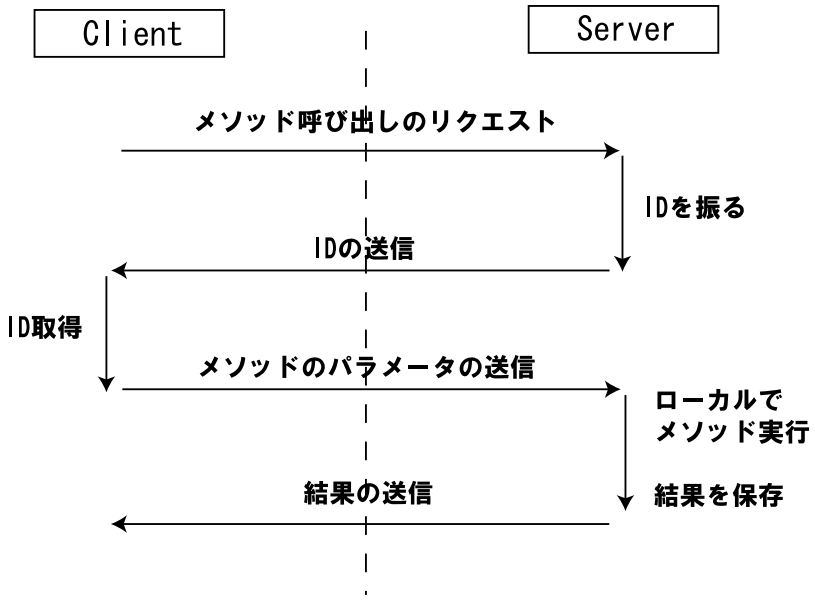

図 3.2: ハンドオーバー機能付き RMI の仕組み

そのため、JavaRMI では通信関連の例外の発生は、大抵の場合この Exception で catch することができる。そこで、本システムでは通信の切断を Exception を検出することで処理中のプログラムに通知する方式を取ることを考えた。本システムの通信プロトコルを用いて、分散処理プログラムを行った場合、通信関連の例外が発生する可能性を考えると、発生しうると考えられる例外は SocketException をスーパークラスに持つ以下の 3 つの例外である。

- BindException
  ローカルなアドレスおよびポートに対してソケットのバインドを試行中にエラーが発生したことを通知する。一般的には、ポートが使用中であるか、要求したローカルアドレス割り当てができないことが原因となる。

- ConnectException
  リモートなアドレスおよびポートに対してソケットの接続を試行中にエラーが発生したことを通知する。一般的には、接続がリモートで拒否された (リモートのアドレスとポートで待機中のプロセスがない) ことが原因となる。

- NoRouteToHostException
  ソケットをリモートアドレスおよびポートに接続しようとしたときにエラーが発生したことを表わす。一般的には、ファイアウォールを仲介しているためにリモートホストに到達できないか、中間ルーターがダウンしていることが原因である。

そこで、本システムにおいては通信関連の例外の発生を SocketException を用いることによって検出することにした。ハンドオーバー機能を実現するには、通信状態に応じてサーバ、クライアント側の処理を待機状態にするかどうかの判断を行わなくてはならない。仮に分散処理中に SocketException が発生した場合、本システムではサーバ、クライアント間で何らかのネットワーク不良が起きたと解釈し、サーバ、クライアント両方の処理を一時待機状態にする。そして、再接続を試みても再び SocketException が発生するようであれば、もう一度待機状態にする。一方、再接続を試みてコネクションが確立されたら処理の継続を行うようにする。以上のような仕組みを用いて、システムを実装しようと考えた。

3.5.1　Exception による処理の待機

Exception を利用することによってプログラムの通信部分を待機状態にする。簡単な例を以下に示す。

```
/** suspend part **/
    while(true){
       try{
            Socket socket = new Socket(host,port);
            ObjectOutputStream oos
                = new ObjectOutputStream(socket.getOutputStream);
            ObjectInputStream ois
                = new ObjectInputStream(socket.getInputStream);
            \\送信・受信を行う部分
                   .
                   .
                   .
            }catch(SocketException se){
```

```
            }
        }
```

以上のような処理を行うことによって、通信関連の Exception が発生したときには、ループが発生し、擬似的に処理は一時待機状態になる。しかし、この方法を用いると、SocketException が検出されるたびにソケットを作り直して、データの再送を行うため、余分なデータを送りなおしてしまう可能性が考えられる。そこで、いくつかのパターンに分けてそれぞれに独自の復旧通信プロトコルを用意しなくてはならない。そのため、前節のような切断のタイミングに関するパターン分けが必要になるのである。

しかし、実装を進めていく内に切断のタイミングによっては SocketException は通信が途切れたとしても発生しないことがあることが分かった。以下で、通信を切断するタイミングにおける SocketException の発生の仕方について説明する。

3.5.2　SocketException の発生に関する問題

SocketException は一般的には通信関連になんらかの障害が起きたときに発生するわけだが、本システムの開発を進めるにあたり、本来検出されるであろうと想定される場合に、発生しないケースもあることが確かめられた。ここでは 3.4 に列挙したそれぞれのケースについてサーバ・クライアントがどのような Exception を検出することが可能なのかを説明していく。
なお、本システムを実装するにあたりネットワークの切断はハンドオーバー機能を念頭において、物理的にネットワークから切断する方法を用いた（LAN ケーブルを抜く）。

1. **リモートオブジェクトの生成に必要なパラメータの通信を行っているときに通信が途切れてしまった場合**

    - サーバ側
      この時、サーバー側では Exception を検出することができなかった。通信は途切れているはずにも関わらず、正常に通信できているかのように扱われる。write は正常に書き込めている

ものと見なされ、read は入力待ちの状態と見なされているようである。

- クライアント側
  この時、クライアント側では Exception が検出される。通信状態に異常が出たものと見なされ ConnectException が発生する。

2. **リモートメソッド呼び出しを行うのに必要なパラメータの通信を行っているときに通信が途切れてしまった場合**

   - サーバ側
     1 の場合と同様にやはりこの場合でも Exception を検出することができない。正常に通信が行われているものと扱われるようである。
   - クライアント側
     クライアント側は 1 の場合と同様に ConnectException が検出される。

3. **リモートメソッドの戻り値をクライアント側が待っているときに通信が途切れてしまう場合**

   - サーバ側
     この場合も Exception が検出されることはない。正常に戻り値を書き込めたものと解釈され、リモートメソッド呼び出しが正常に終了されたかのように扱われる。
   - クライアント側
     通信が切断されると、戻り値の読み込みを待っているところで Exception が発生する。この場合もやはり ConnectException が検出される。

以上の結果で分かるように、分散処理中に物理的にネットワークを切断するという場合を想定するとサーバ側で Exception を検出することは不可能である。
そこで、通信状況の管理はすべてクライアント側で行い、クライアントからの要求によってサーバ側が処理を変えるという方法をとらなければならない。

クライアントで SocketException が検出されたら、待機状態にして新たにコネクションが確立されたらクライアント側から再処理の要求を出すようにする。

3.5.3　OS の違いによるネットワークの切断に対する検証

以上の実験は OS にサーバ側では UNIX、クライアント側では WINDOWS を用いた。というのも UNIX マシンでは OS 稼動中に物理的にネットワークを切断する場合が想定されていないものらしく、LAN ケーブルを抜いたりすると OS の動作自体が著しく不安定になるためである。一方 WINDOWS マシンでは物理的にネットワークの切断しても、OS 自体が不安定になるということはなかった。

そこで、クライアント用に WINDOWS マシンを用い、ネットワークの物理的な切断は主にクライアント側で行う仕様にすることにした。そもそもハンドオーバー機能では目的にしているのはクライアント側の自由な移動なので、クライアント側でネットワークが切断される可能性が高い。そこで、このような仕様にしてもハンドオーバー機能の実現には差し支えないと考える。

3.6　invokecounter の導入

前々節で切断のタイミングを 3 パターンに分けた。ここで 3 の場合の事について少し考えてみたい。このケースはサーバ側がローカルでメソッド呼び出しをしているときに、通信が途切れるわけだが、ローカルでのメソッド呼び出しに時間がかかる場合、このタイミングで切断されるケースは多いと考えられる。

このケースで問題になるのがサーバ側でのローカルメソッド呼び出しまで処理が進んでしまう点である。どういうことかというと、仮にローカルでメソッド呼び出しが行われる前の段階（前節の 2 のケース）であれば、もう一度パラメータ等の情報を再送して、リモートメソッド呼び出しを始めからやり直せばよい。しかし、一旦ローカルでメソッド呼び出しが始まってから切断されてしまうと、もう一度始めからリモートメソッド呼び出しを行うわけにはいかない。一度サーバ側でローカルでメソッド呼び出しがされているため、再接続された際にもう一度リモートメソッド呼び出しの要求を出すと 1 回のリモートメソッド呼び出しのリ

クエストでサーバ側では２回ローカルメソッド呼び出しが行われてしまうためだ。そこで、重複してリモートメソッド呼び出しが行われないように invokecounter を導入する。

3.6.1　invokecounter とは

invokecounter とは３．４節の３のケースの場合でも重複してリモートメソッド呼び出しが行われないようにメソッド呼び出し自体に ID を振り分けるためのカウンターのことである。それと同時に戻り値を常にバックアップしておくためのバックアップリストも用意する。この invokecounter はサーバ側でリモートメソッド呼び出しの要求がされる度にカウントされていく。具体的にはサーバ側のプログラム（Skelton サーバ）の requestHanddler で action_flag に METHOD_INVOKATION が書き込まれるたびに invokecounter を１つずつ増やしていく仕組みである。invokecounter の値はリモートメソッド毎の ID として、クライアント側がメソッド呼び出し用のパラメータの送信等を行う前にクライアント側に送られる。サーバ側では常に戻り値のバックアップしておき、クライアント側から再接続の要求があった場合、その ID を元にバックアップリストからオブジェクトを取り出しクライアント側に値を返すようにする。

3.6.2　invokecounter の為のリスト（invokecounterlist)

この方法を用いると、メソッド呼び出しの重複呼び出しはされなくなる。しかし、サーバ側でのバックアップリストが膨大になってしまう恐れがある。そこで、サーバ側でどのオブジェクトのバックアップを取っておけばいいかの指示をクライアント側から出さなくてはならない。そのために、クライアント側ではこの ID を管理するためにリストを用意し（invokecounterlist)、クラス（Idcheck）を用意する。このクラスでは以下のメソッドを提供する。

次にサーバ側でバックアップオブジェクトを管理するためのメソッドについて説明をする。なお、バックアップオブジェクトは LinkedList クラスを用いた backuptable というリストに格納されている。以上のようなメソッドを用いて、メソッド毎の ID とバックアップリストの管理を行う。以下は invokecounter を用いたクライアント側のプログラム部分である。

```
/** Client **/
```

表 3.1: Idcheck クラスのメソッド

メソッド名	説明
void addList()	リストに ID を加える為のメソッド
void removeList()	リストから ID を削除する為のメソッド
int minNumber()	リストの中の最小の ID を int 型の戻り値にして返すメソッド

表 3.2: サーバ側のオブジェクトのバックアップに関するメソッド

メソッド名	説明
void backupObject(Object obj, int id)	バックアップリストに ID として id を持つオブジェクト obj を 加えるメソッド
void removeObject(int minnumber)	リストから ID が minnumber-1 より小さい ID を持つ オブジェクトを削除する為のメソッド
Object searchbackupObject(int id)	リストの中から ID として id を持つオブジェクトを オブジェクトを取り出すメソッド

```
ObjectOutputStream oos
    = new ObjectOutputStream(socket.getOutputStream());
oos.writeInt(METHOD_INVOKATION);
ObjectInputStream ois
    = new ObjectInputStream(socket.getInputStream());
invokecounter = ois.readInt();
invokecouterlist.addList(invokecounter);
int minnumber = invokecounterlist.minNumber();
oos.writeInt(objid);
oos.writeInt(minnumber);
oos.writeObject(classname);
```

```
    oos.writeObject(methodname);
    oos.writeObject(paramtypes);
    oos.writeObject(paramvalues);
    Object ret = ois.readObj();
    invokecounterlist.removeList(invokecounter);
```

クライアント側では以上のような処理になる。正常に戻り値が戻ってくればクライアント側でリストからそのメソッドの ID が削除される。この方式を用いれば戻り値を待っている時に通信が切断された場合、invoke-coutnerlist から正常にメソッド呼び出しが終了していないメソッドの ID が削除されることはない。つまり、リスト上の最小値の ID を得ればその最小値より小さい ID を持つメソッド呼び出しは正常に終了していることになる。

以下は、サーバ側のプログラム部分である。

```
/** Skelton Server **/
    invokecounter++;
    oos.writeInt(invokecounter);
    int ID = ois.readInt();
    minnumber = ois.readInt();
    removeObject(minnumber);
    Object targetobject = getObject(ID);
    String classname = (String)ois.readObject();
    Class targetclass = Class.forName(classname);
    String methodname = (String)ois.readObject();
    Class[] paramtypes = (Class[])ois.readObject();
    Object[] paramvalues = (Object[])ois.readObject();
    Method targetmethod
        = targetClass.getDeclaredMethod(methodname,paramtypes);
    Object ret = targetMethod.invoke(targetObject,paramvalues);
    backupobject(ret,invokecounter);
    oos.writeObject(ret);
```

サーバ側では以上のような処理が行われる。まずは最初に invokecouter の値をクライアント側に送り、これをこのメソッド呼び出しの ID とする。そして、次にクライアント側からすでにメソッド呼び出しが正常に終了しているメソッドの ID を得る（minnumber)。これを元にして、バック

アップオブジェクトのリストの整理を行い、Reflection を用いてその後の処理につながっていく。

3.7　処理再開の為の通信プロトコルの概要（クライアント側）

切断のタイミングは今まで述べてきたように 3 パターンに分けられている。ここで、問題になるのは前節で述べたようにサーバ側ではネットワークの物理的な切断に対して Exception を検出することができないという点である。その為、クライアントとサーバ間で同時に処理を待機状態にできない。サーバ側では本来、待機状態になっていなくてはならないプロセスが正常に終了したものとみなされてしまうのだ。そこで、サーバ側は常に処理の中断ができないものとして、再開処理のプロトコルも requestHandler メソッド部分に含めてしまうことにする。

以上のようなことを踏まえて、復旧通信プロトコルの概要に触れてみたいと思う。ただし 2 の場合、前節で述べた invokecounter の関係上、さらに細かい場合分けが必要になる。どのように場合分けする必要があるかというと、クライアントがサーバ側の invokecounter からメソッド毎の ID を受け取ってから、パラメータ等を送信している最中に通信が途切れてしまう場合である。この場合、クライアント側では addList によってリストに ID が追加されるわけだが、処理はリモートメソッド呼び出しの始めの部分からやり直しになるので、この ID はリストから永久に削除されることはない。サーバ側のバックアップリストの管理は Idcheck の minNumber を利用してリスト中の最小値を受信して、それより小さい ID を持ったオブジェクト（すでにクライアント側で受け取られている戻り値のオブジェクト）を削除するという方法なので、ここで 2 の場合について場合分けを行わないとバックアップリストが増えつづけてしまう可能性が出てくる。

そこで、 2 の場合はさらに細かく分け、ID 取得前と ID 取得後の 2 パターンに分けることにした。よって、切断のタイミングは 4 パターンに場合分けされる。以下ではそれぞれのパターンについての通信プロトコルの概要について触れる。

1. **リモートオブジェクトの生成に必要なパラメータの通信を行っているときに通信が途切れてしまった場合**
   この場合、パラメータの始めから送りなおせばよいだけなので、特

に工夫なく実装ができる。以下にプログラム例をしめす。

```
/** pattern 1 **/
  while(true){
       try{
            Socket socket = new Socket(host,port);
            ObjectOutputStream oos
                = new ObjectOutputStream(socket.getOutputStream);
            ObjectInputStream ois
                = new ObjectInputStream(socket.getInputStream);
            oos.writeObject(classname);
            oos.writeObject(paramtypes);
            oos.writeObject(paramvalues);
            int objectid = ois.readInt();
            }catch(SocketException se){
            }
        }
```

2. **リモートメソッド呼び出しを行うのに必要なパラメータの通信を行っているときに通信が途切れてしまった場合**
   前にも述べたようにこの場合はさらに 2 つに分類できる。

   - **ID 取得前**
     ID 取得前であれば、invokationcounterlist に ID は書き込まれない。その為、データの再送を始めからやり直しても問題はない。1 の場合と酷似した内容になる。

   - **ID 取得後**
     ID 取得後であると、invocationcounterlist に ID が追加されているため、もう一度 ID を取得しようとすると無駄な ID がリストに残ってしまうことになる。そこで、この場合は ID 取得後のところからの処理が必要になる。

     ```
     /** pattern 2 **/
       try{
         引数の型などの Reflection に必要な情報や
         ID の送受信のみを行う。
     ```

```
.
.
\** pattern 3 **\
ここの部分で戻り値の読み込みを行う
.
.
}catch(SocketException se){
  while(true) {
   try {
      if(ID 取得前) {
        データの再送を始めからやり直す
        /** pattern 3 **/
        .
        .
        return ret;
      }
      if(ID 取得後) {
        ID の取得を行わずにパラメータのみを送って
        メソッド呼び出しを行う。
        なお取得した ID をサーバ側に報告し、
        サーバ側で ID とオブジェクト（戻り値）との
        関連付けを行う。
        /** pattern 3 **/
        .
        .
        return ret;
      }
    }catch(SocketException se2) {
    }
  }
.
.
return ret;
}
```

流れは以上のようになる。ID 取得前と後でサーバ側に送りなおす

データを変える。そのことによって、サーバ側の方も ID を渡す前に通信が切断されたのか渡した後に通信が切断されたかの判断がつくので 1 つのメソッド呼び出しで 2 つの ID を発行してしまうことがなくなる。

3. **リモートメソッドの戻り値をクライアント側が待っているときに通信が途切れてしまう場合**
 この場合は、リモートメソッドの呼び出しが正常にサーバ側には伝わっているため、もう一度パラメータ等を再送すると、重複して呼び出してしまうことが考えられる。そのために invokationcouter を導入したわけだが、以下ではその具体的な使い方について触れてみる。

```
/** pattern 3 **/
  try {
    ret = ois.readObject();
  }catch (SocketException se) {
    while(true) {
      try {
        Socket socket = new Socket(host,port);
        ・
        ・
        oos.writeInt(invokationcounter);
        /** ここで ID をサーバ側に送り、
            その ID を持ったオブジェクトを送り返してもらう**/
        ret = ois.readObject();
      }catch(SocketException se2) {
      }
    }
  }
```

以上のように、この場合では invokationcounter の値を送り、その ID を元にサーバからの戻り値を読み込む処理を繰り返す。これにより、メソッド呼び出しが重複して行われることはなく、このケースではオブジェクトのみをサーバ側から再送する形になる。

3.8　MultiSkelton サーバの実装

ここでは、ハンドオーバー機能に対応した MultiSkelton サーバの実装について述べる。

先にも述べたように、物理的なネットワークの切断に対し、サーバ側では Exception を検出することができない。そのため、通信再開のための処理は主にクライアント側が担当することになる。そこで、Skelton サーバ側ではそのようなクライアントからの要求に応えられるような通信プロトコルをいくつか用意し、クライアントからの指示によってそれらの通信プロトコルの切り替えを行わなくてはならない。そこで、３．３でも説明したとおり Skelton サーバでは requestHandler メソッド内に action_flag という int 型の値を用意し、それのよって通信プロトコルの切り替えを行えるようにしてきた。

3.8.1　サーバ間のオブジェクトの移動

ハンドオーバー機能を実現するには、通信再開の処理だけでは不十分である。クライアントが自由にサーバを選び、さらにどのサーバを選んでも処理の再開ができるようにしなくてはならない。

あるクライアント A がサーバ B と行っている分散プログラムを途中からサーバ C を使って処理を続けるにはサーバ B からサーバ C への何らかのマイグレーション処理が必要になる。そこで、MultiSkelton サーバはサーバ間でのマイグレーション処理を含んだ形で実装されなくてはならない。本システムは Java を用いて開発したため、マイグレーション処理としてオブジェクトの移動が適当であると考えた。そこで、MultiSkelton サーバでは、リモートオブジェクトの移動を行うサーバ・サーバ間の通信プロトコルを用意することにした。

移動すべきオブジェクト

処理を継続して実行するために移動すべきオブジェクトを選抜しなくてはならない。まず、移動しなくてはならないのは objectList である。このリストの中にはリモートオブジェクトが格納されているため、このリス

トをそのまま他サーバに移動することでリモートオブジェクトの状態を変えることなく、他サーバにリモートオブジェクトを移動することができる。その他に objectid（key）や invokecounter、minnumber、backuptable といった通信再開処理に必要なオブジェクト、値の移動も必要になってくる。そこで、以下のようなメソッドを用意し、サーバ間のオブジェクトの移動を可能にした。まずは MultiSkelton 内のリストなどの情報を受信するためのメソッド receiveObjectlists の概要を示す。

```
/** receiveObjectlists (in MultiSkelton) **/
  public void receiveObjectlists
    (ObjectInputStream ois, ObjectOutputStream oos){

    String host = (String)ois.readObject();
    int port = ois.readInt();

    Socket s_1 = new Socket(host, port);
    ObjectOutputStream oos_1
        = new ObjectOutputStream(s_1.getOutputStream());
    ObjectInputStream ois_1
        = new ObjectInputStream(s_1.getInputStream());
    oos_1.writeInt(MOVELIST_INSTRUCTION);
    this.objectList = (LinkedList)ois_1.readObject();
    this.backuptable = (LinkedList)ois_1.readObject();
    this.objectid = ois_1.readInt();
    this.minnumber = ois_1.readInt();
    this.invokecounter = ois_1.readInt();

    int action_flag2 = ois.readInt();
    switch (action_flag2) {
      case OBJECT_CREATION:
      .
      .
      break;
      case METHOD_INVOKATION:
      .
      .
```

```
      break;
      .
      .
    }
  .
  .
  }
```

ここで、receiveObjectlists は処理を再開しようとしているクライアントと以前に処理の行われていたサーバ（簡略化のためサーバ A と表記する）の両方と必要な情報の通信を行う。

はじめに新しいサーバ（以下、サーバ B) はクライアントからサーバ A のホスト名とポート番号を受け取る。そして、そのホスト名とポートをもとにサーバ B はサーバ A に対して必要な情報を送るようにヘッダとして MOVE_INSTRUCTION を送る。この MOVE_INSTRUCTION はサーバ A の requestHandler 内の action_flag に対応していて、別のサーバへリストを移動する通信プロトコルに切り替えることを意味する。これによりサーバ A では新たに処理がされるサーバに対し、必要なオブジェクト、値を送信する。サーバ B ではサーバ A から送られてきたオブジェクト等を用いて、処理の継続を行う。

以下は処理再開のために新しいサーバで必要となるオブジェクトや値を送る sendObjectlists の概要である。

```
/** sendObjectList(in MultiSkelton) **/

  public void sendObjectLists
  (ObjectInputStream ois, ObjectOutputStream oos) {
    oos.writeObject(this.objecttable);
    oos.writeObject(this.backuptable);
    oos.writeInt(this.objectid);
    oos.writeInt(this.minnumber);
    oos.writeInt(this.invokecounter);
     .
     .

  }
```

以上のようにして、サーバ間で必要な情報のやり取りをすることによって、クライアントは自由にサーバを選択することができる。

3.8.2　MultiSkeltonの通信プロトコル

これまで、説明してきたとおり MultiSkelton には状況に応じた様々なプロトコルが用意されている。そこで、これらを整理して requestHandler 内の action_flag と関連付けて説明する。以下、太字は action_flag の値である。

1. **OBJECT_CREATION**
   リモートオブジェクトを生成するための要求がきた場合の action_flag。この場合は通常とおりリモートオブジェクトを生成する通信プロトコルを用いる。objectList を用いて生成したオブジェクトの登録を行う。

2. **METHOD_INVOKATION**
   リモートメソッドを呼び出す要求がきた場合の action_flag。この場合はリモートメソッド呼び出しが行われる。invokecounter のインクリメントやバックアップオブジェクトの管理も行う。

3. **BACKUPOBJECT_TRANSMISSION**
   クライアント側が戻り値を待っている時に通信が切断されたことをサーバ側に伝える action_flag。この要求が来たら、サーバ側ではバックアップリストから適切なオブジェクトを検索し、クライアント側に送信する。

4. **PARAM_TRANSMISSION**
   クライアント側が invokationcouter を受け取ったあと、リモートメソッド呼び出しに必要な情報を送っている最中に通信が切断されたことをサーバ側に伝える action_flag。ここでは、リモートメソッド呼び出しに必要な情報だけ（objectid と Reflection で用いる情報）のみを送受信再び行い、リモートメソッド呼び出しを行う。

5. **RECONNECT_MESSAGE**
   一度、他のサーバと処理を中断したクライアントからの一番初めの要求が来たことを伝える action_flag。ここでは、処理再開のために以前処理を行っていたサーバに対し、下の MOVELIST_INSTRUCTION を送信し、objectlist 等の処理の再開に必要な情報の受信を行う。また、ここではさらに action_flag2 が用意されており、その値によって１～４のどれかの処理を続けて行う。

6. **MOVELIST_INSTRUCTION**
   この action_flag のみクライアントからの送られてくるのではなく、他のサーバから送信されてくる。ここでは、他のサーバに対して、処理再開に必要な情報の送信を行う。

以上が MultiSkelton の通信プロトコルの概要である。

3.9　stub generator の作成

以上のような仕様を満たす、クライアントプログラム（stub プログラム）をプログラムごとにプログラマが用意するのは非常に手のかかる作業になってしまう。そこで、このような MultiSkelton に対応したスタブジェネレータを作成することによって、プログラマの負担が軽減されるはずである。そこで、リモートメソッド呼び出しを意識しないでプログラミングされたマスタクラスから MultiSkelton に対応する stub クラスをジェネレートするような stub generator を作成した。stub generator の作成にあたっては Javassist を利用した。

3.9.1　Javassist を利用した stub の生成

Javassist を用いてマスタクラスの情報を取得し、以下のような作業をすることによりマスタクラスから stub を作成することが出来る。

- マスタクラスの情報の取得
- スタブクラスの作成

- マスタクラスと同じシグネチャのコンストラクタのスタブクラスへの追加
- マスタクラスと同じシグネチャのメソッドのスタブクラスへの追加
- マスタクラスと同じ型のフィールドのスタブクラスへの追加

以上の作業をすることにより stub を作成する。そして、追加されたコンストラクタの本体やメソッドの本体に通信プロトコルを埋め込むことにより、stub の生成が可能になる。

3.9.2　codetranslator の作成

実行プログラムのマスタークラスへの参照部分を stub クラスの参照に置き換える為の codetranslator も作成する。codetranslator を作成することによりユーザーは stub の存在を意識することなくプログラミングを行うことができる。codetranslator で書き換えるのは、実行プログラム中のマスタークラスオブジェクトへの参照部分である。これを新たに作成した stub クラスのオブジェクトへの参照に書き換えるようにする。さらにマスタクラスのオブジェクトはシリアライズされて、サーバ間を移動する可能性があるので、マスタクラスに Java.io.Seriarizable インターフェースを実装する。こうすることにより、プログラマはリモート呼び出しを意識することなく分散処理プログラムを書くことができる。

第4章　実験

ハンドオーバー機能を用いてリモートメソッド呼び出しを行う際の時間を測定するため、実験を行った。実験に用いた計算機は、Sun Blade 1000(UltraSPARC III 750MHz × 2, 1024MB, Solaris 8) Victor InterLink(Pentium3 700MHz , 256MB, WindowsXP HomeEdition)、Nabe International's PC(Pentium3 733HMz , 128MB, Linux(RedHat7.1))、JVM は $JavaHotSpot(TM)ClientVM(build1.4.0_01-b03, mixedmode)$ を用いた。

4.1　リモートメソッド呼び出しにかかる時間比較

JavaRMI と、本研究で用いたハンドオーバー無し RMI と、ハンドオーバー付き RMI の１回あたりのリモートメソッド呼び出しにかかる時間の比較を行った。Solaris、Linux、Windows の３つの OS においても比較できるように、それぞれ実験を行った。この実験では同一マシン内でのリモートメソッド呼び出しによる測定を行っている。結果は、以下の表のようになる。

表 4.1:　１回あたりのリモートメソッド呼び出し回数による時間比較 (msec)

	JavaRMI	ハンドオーバーなし RMI	ハンドオーバー付き RMI
Solaris	4.18	103.30	305.51
Linux	2.19	4.67	40.4
Windows	1.90	4.31	4.70

本システムにおいては JavaRMI とは異なり、それぞれのマスタクラスに個別に stub コードと skelton コードを生成するのではなく、skelton 側に

は汎用 skelton である MultiSkelton クラスを用いている。この汎用 skelton クラスはリフレクションを利用することによって、任意の stub コードに対応している。そのため、リフレクションを利用する回数が多くなれば多くなるほど、オーバーヘッドは大きくなってしまう。表を見てもわかるように、JavaRMI とはメソッド呼び出しの回数が増えれば増えるほど、オーバーヘッドが大きくなっていることが証明されている。また、この表から Solaris と Linux を利用したハンドオーバー付き RMI はオーバーヘッドがかなり大きいものとなってしまっている。これは、リモートメソッド呼び出しの戻り値を受け取る部分の java.io.ObjectInputStream クラスの readObject メソッド部分で、時間がかかってしまうことによる。本システムにおいては ObjectInputStream と ObjectOutputStream を多用しているため、その部分で時間がかかっているものと予想される。

4.2　送信する引数のサイズによる比較

次に JavaRMI と、本システムで用いたハンドオーバー無し RMI と、ハンドオーバー付き RMI を利用して、リモートメソッド呼び出し時に送信する引数のサイズによる 1 回あたりのリモートメソッド呼び出しにかかる時間の比較を行った。この実験では、マシンは上記 Linux マシン（Nabe International's PC) を用いた。表 4.2 は同一マシン内における 1 回あたりのリモートメソッド呼び出しの引数のサイズによる比較である。表 4.3 は同じ Linux マシンを 2 台用いた、異なるマシン間における 1 回あたりのリモートメソッド呼び出しの引数のサイズによる比較である。LAN カードは 100BASE を用いた。

表 4.2: 同一マシン内における 1 回あたりのリモートメソッド呼び出しにおける引数のサイズによる時間比較（msec）

	JavaRMI	ハンドオーバーなし RMI	ハンドオーバー付き RMI
引数なし	2.19	4.67	40.4
1 K バイト	2.24	4.95	40.5
1 M バイト	163	138	140

表 4.3: 異なるマシン間における１回あたりのリモートメソッド呼び出しにおける引数のサイズによる時間比較（msec）

	JavaRMI	ハンドオーバーなし RMI	ハンドオーバー付き RMI
引数なし	2.19	4.54	40.5
１ K バイト	2.76	4.42	42.1
１ M バイト	166	138	141

引数のサイズが大きくなれば大きくなるほど、メソッド呼び出しにかかる時間は大きいわけだが、システムごとの差はそう大きいものではない。表から、引数のサイズが増えるごとに増える時間が各システムとも大差がないことからわかる。つまり、引数の送受信部の実装は、各システムごとに似たようなものになっていることが予測される。また異なるマシン間での呼び出しも同一マシン内での呼び出しも大差がないことが確認された。通常、異なるマシン間での呼び出しの方がマシンの外のネットワークを介する分、時間がかかりそうなものであるが、実験によれば、それは大した時間にならないことがわかった。むしろ、RMI システムの送受信において、時間がかかっている部分というのは、物理的なネットワークによるものではなく、JVM 内の送受信の処理によるものではないかと予想される。

第5章　まとめ

本稿ではハンドオーバー機能を付加する Java 用の RMI システムの提案と実装を行った。本システムを用いることにより、リモートメソッド呼び出しを意識することなくプログラミングされたプログラムをハンドオーバー機能のついた分散プログラムに書き換えることが可能になる。今までは、ユビキタスコンピューティングにハンドオーバー機能が必要ということは言われてきていたが、その機能を組み込もうとするとアプリケーションごとに個々に埋め込むしか方法がなかった。一方、本システムを用いることによって利用者は既存の RMI システムを使う感覚で、ハンドオーバー機能を付加したアプリケーションの製作することができる。本システムにおいてはハンドオーバー機能を以下の 2 つの機能に分類して、それぞれ stub と skelton の通信プロトコルに埋め込むという方法をとって実装を行った。

- サーバの自由な切り替え機能
  サーバのオブジェクトを移動させることにより実現
- 通信状態に応じたプログラムの自動的な待機・再開機能
  リモートメソッド呼び出しが正常に終了するまで retry することによって実現

以上の 2 点の機能により本システムを用いて作成されたアプリケーションを使うユーザーは通信の切断による中断処理を行わないですむ。そのため、ネットワークを気にせずに作業をすることができるようになった。

しかし、本システムでは完全にハンドオーバー機能が実現されているとは言い難い。まず、考えられるのは入・出力の問題に関してである。たとえば、クライアントからの指示に応じて、サーバ側でファイルの書き込み、読み込みをおこなっているようなプログラムを想定する。そのようなプログラムの場合、マイグレーション時に書き込み、読み込みされてるファイルの移動も行わなくてはならない。しかし、現時点ではサーバプログラムで用いられているオブジェクトの移動までしか実現できてい

ないので、このような状況に関する対応は今後の課題となる。また、ネットワークの再接続先は stub プログラムによって決め打ちする方式を取っている。（クライアント側のユーザーに新しいホスト名とポート番号の入力を要求する）そのため、処理の再開のプロセスは自動化されているとは言えない。そこで、ネットワークを検知し、接続先を自動的に選択するような機能を付け加えることも今後の課題となるであろう。また実験においては、OS によってまったく違った速度になってしまった。これは主に ObjectInputStream と ObjectOutputStream の部分に原因があるものと思われる。そこで、Solaris や Linux でも、理想的な値が得られるような送受信部分の最適化も必要になるだろう。

参考文献

[1] Chiba, S.: Load-time Structural Reflection in Java, *Proceedings of the European Conference on Object-Oriented Programming*, pp. 313–336 (2000).

[2] Fabre, J.-C. and Perennou, T.: A Metaobject Architecture for Fault Tolerant Distributed Systems: The FRIENDS Approach, *IEEE Transactions on Computers, Special Issue on Dependability of Computing Systems*, Vol. 47, No. 1, pp. 78–95 (1998).

[3] 高宮安仁, 松岡聡: ユーザー透過な耐故障製を実現する MPI へ向けて, 情報処理学会・電気通信処理学会 並列処理シンポジウム JSPP2002 論文集, Vol. 47, No. 1, pp. 217–224 (2002).

[4] 千葉滋: Javassist Home Page, `http://www.csg.is.titech.ac.jp/~chiba/javassist/index.html`.

[5] 千葉滋, 立堀道昭: Java バイトコード変換による構造リフレクションの実現, 情報処理学会 論文誌, Vol. 42, No. 11, pp. 2752–2760 (2001).

[6] 今井尚樹, 金子晋丈, 森川博之, 青山友紀: ユビキタスアプリケーション実現に向けたサービスハンドオフ機構, 第 4 回 YRP 移動体通信産学官交流シンポジウム (2002).

[7] 南正輝, 杉田馨, 森川博之, 青山友紀: ユビキタス環境に向けたインターネットアプリケーションプラットフォーム, 電子情報通信学会論文誌, No. 12, pp. 1–19 (2002).

[8] 森川博之: ユビキタスネットワークへの道, 情報処理学会高品質インターネット研究会, pp. 4–7 (2002).

付 録A　Javassist

Javassist [1, 5] は、構造リフレクションの機能を提供するクラスライブラリである。Javassist は [4] で配布されている。構造リフレクションとは、クラスや関数などのデータ構造の定義を必要に応じて変更できるようにする機能である。Java は標準 Java API の一部としてリフレクションの機能を提供するプログラミング言語である。しかしながら、提供される機構はプログラム中で用いられるデータ構造、すなわちクラス、の定義を調べる機能（introspection）が主で、プログラムの振る舞いを変更する機能は非常に限られている。Java のリフレクション機能を強化するために、これまでいくつかのシステムが提案されてきたが、そのほとんどは動作リフレクションの機能を提供している。これはメソッド呼び出しのような演算を横取りして、その動作を変更できるようにするものである。

我々はリフレクションの機能拡張を実装するには、動作リフレクションよりも構造リフレクションの方が適していると考える。構造リフレクションは、そのような言語拡張に必要な機能を直接提供するので、実装は容易に実現できる。また、動作リフレクションは構造リフレクションさえ提供していれば、その上に動作リフレクションを実現し、それを使って目的の言語拡張を実装することも可能である。

Javassist は構造リフレクションを実現するにあたり、クラスを JVM にロードする際にバイトコード変換を行なうことで実現する。従来知られている方法では、JVM の内部を変更する必要があったが、可搬性が重要な Java 言語ではこの方法は現実的でない。またソースコード変換器を使って実現する方法では、ソースコードなしでは構造リフレクションが利用できないという問題があった。Javassist で採用したバイトコード変換による方法では、このような問題を回避できる。