JavaRMI と JavaIDL による
分散オブジェクト環境の実現及び評価

五十嵐義人　　片山零士

Evaluation of Distributed Object Environments of JavaRMI and JavaIDL

Yoshito Ikarashi　and　Reiji Katayama

第1章　序論

1.1　研究の背景

メインフレームによる専用システムが中心だった時代は，開発者がそのシステム専用の特別なアプリケーションを開発するまでユーザは自分たちの望む処理を行なうアプリケーションを手に入れることができなかった．しかしその後，分散システムの登場により，いくつかの処理を分散して行なえるようになり，ユーザはより洗練され，かつ柔軟な汎用アプリケーションを自由に用いることができるようになった．

分散システムは主に資源を利用するクライアント（client）と資源を管理し，共有を支援するサーバ（server）という基本的な役割分担がみられる．これは，クライアント／サーバ・システムと呼ばれ，分散システムのアーキテクチャの一つとなっている．

分散システムにおける通信サービス機能の一つに，RPC（Remote Procedure Call）がある．RPC とは，サービスを提供するサーバ・マシンの処理をリモートで呼び出すことができる機能である．RPC を用い，リモート処理を関数化して提供することにより，ネットワークの複雑性が隠蔽でき，便利で分かりやすいものになる．

しかし，サーバ・マシンで提供するサービスが増えてくると，利用者側に提供される関数の使い方が複雑になるという問題がある．

そこで，分散オブジェクト環境が必要となった．分散オブジェクト環境とは，リモートマシン上にあるオブジェクトを自分のマシン上に呼び出し，それを操作することによりリモートマシン上のオブジェクトをあたかも自分のマシン上にあるかのように操作することができるプログラミング技術である．分散オブジェクトを用いれば，関数をクラス化し，クラスの実体（インスタンス）を複数持つことができる．これにより，データをインスタンスごとに管理することが可能となる．さらに，関数のパラメータや戻り値に複雑な構造を渡すときに，その複雑さを隠蔽するためにオブジェクトを使うことができる．これらの機能により，サーバ側のサービスが多くても，クライアント側は容易にサーバ側のサービスを使うことができるようになった．

主な分散オブジェクト環境としては，JavaRMI（Remote Method Invocation），CORBA（Common Object Request Broker Architecture），DCOM（Distributed Component Object Model）が代表的である．現在，分散オブジェクト環境に用いられる言語としては Java が注目されている．Java を使用することにより，オブジェクト指向，プラットフォームからの独立，セキュリティをはじめとする，多くの長所がある．他にも，Java には，ウイルスの拡散やデータの盗難，システムの破壊を心配することなく，プログラムを Web サイトからすばやくダウンロードし，安心して実行できるようにする機能がある．Java のおかげで，これまで何年も構想はありながらサーバベースで要求される処理能力が大きすぎて実現できなかったアプリケーションが，数多く実現されている．これは，Java が処理を

ネットワークに移行するためである.

1.2 研究の目的

Java を用いた分散オブジェクトを実現する環境には，JavaRMI（Remote Method Invocation）と JDK1.2（Java Development Kit）のリリースにより利用できるようになった JavaIDL（Interface Definition Language）がある.

これら 2 つのオブジェクト技術は一見するとかなり類似しているが，どちらも欠かすことのできない技術である．そこで本研究では，それぞれの環境におけるシステムの設計，開発の方法を理解するとともに，実際の動作における性能を比較し，どちらの環境がどの場面で優れているかを評価，検討をすることを目的とする.

第2章　Java と分散オブジェクト環境

2.1　プログラミング言語Java

2.1.1　Java 誕生の背景

1990 年 12 月，米 Sum Microsystems 社にある開発チーム（グリーンコンシューマプロジェクトといい，1991 年 4 月にはグリーンプロジェクトに改名された.）が結成された．このチームの目的は，各種家電製品用の組み込みソフトウェアを作成できる言語を開発することだった．各種家電製品の組み込みプログラムとして以下のような条件が挙げられる．

1) 製品の種類が多用である為，基盤の異なる環境で実行可能なプログラムを作成できる．
2) 記憶媒体といった資源に限りがある為，サイズの小さなプログラムを作成できる．
3) 一度不都合が生じてしまうと，取り外しが非常に困難である為，作成したプログラムの動作が安定している．

これに基づいて開発されたのが，Java の基となった言語 Ork である．しかし，家電用の組みこみソフトウェアを作成する言語であった為，あまり注目をされなかった．しかし，WWW（World Wide Web）とインターネットが爆発的な成長・普及を始めると，徐々に注目を集めるようになる．インターネットでは，種類の異なる多くのコンピュータを互いに接続することができる．コンピュータには，異なる CPU や OS を使用するものも含まれるので，移植可能なプログラムを作成できるという能力，つまり上記の１）の能力は，家電製品だけでなくインターネットにとっても魅力的なものであった．

そこで，Sun Microsystems 社は Ork をインターネット向けの言語に再開発するチームを結成し，様々なプログラミング言語を研究し．1995 年に Java を発表した．

2.1.2 Java の特徴

Java の特徴として，以下のことが挙げられる．

(1) インタプリタ型の言語である．

言語プロセッサという視点でプログラム言語を分類すると，次のように大きく 2 つに分けることができる．（表-2.1 参照）

1) 翻訳プログラムを用いた言語

原始プログラム（ある言語で書かれたプログラム）をまとめて目的プログラム（翻訳されて機械語になったプログラム）に翻訳してから実行するプログラム言語．このとき，翻訳に使われるプログラムを翻訳プログラムといい，アセンブリ言語の翻訳プログラムであるアセンブラ，COBOL，FORTRAN，C などの高水準言語の翻訳プログラムであるコンパイラなどがある．

2) インタプリタ言語

原始プログラムの命令を 1 命令ごとに機械語に直して実行するということを繰り返し行うプログラム言語．実行しながらエラーを発見し，すぐに直せるという利点がある反面，プログラムにループ処理がある場合には，何度も同じ命令を機械語に直すので，コンパイラに比べて処理が遅くなる．インタプリタ言語の種類として，BASIC，APL 等が挙げられる．

表-2.1 インタプリタ言語とコンパイラ言語の比較

	インタプリタ言語	コンパイラ言語
実行効率	△	◎
開発効率	◎	○
柔軟性	◎	○

◎: 優　○: 並　△: 劣

Java はインタプリタ言語に分類されるが，厳密には異なる部分があり，後述する Java のプラットフォーム非依存性と大きく関わってくる．

(2) プラットフォームに依存しない．

プラットフォーム非依存性を説明するため，Java のソースプログラムがどの様な仕組みで実行されているのかを説明する．

プログラムには，前に述べた通り原始プログラム（ソースコード）と目的プログラム（オブジェクトコード）の 2 つの形態がある．オブジェクトコードは，特定の CPU 専

用であるのが一般的なので，異なるプラットフォームでは実行できない．

Java プログラムでもまずソースコードを作成する．しかし，次に行うソースコードのコンパイルで他のコンピュータ言語の違いが出てくる．Java コンパイラは，実行可能コードを作成するのではなく，バイトコードのファイルを作成する．バイトコードは中間コードの一種であり，特定の CPU に依存しない命令体系である．バイトコードは JVM（Java Virtual Machine）によって解釈できるように設計されている．

Java のプラットフォーム非依存性は，同じバイトコードをあらゆるプラットフォーム上の JVM で実行できるという点にある．その環境用に作成された JVM がある限り，どのような Java プログラムでも実行できる．

また，バイトコードの命令を直接実行できるコンピュータハードウェアを作成することも可能であり，現在，Sun Microsystems 社とそのパートナーは，それを目的としたシリコンチップを開発している．

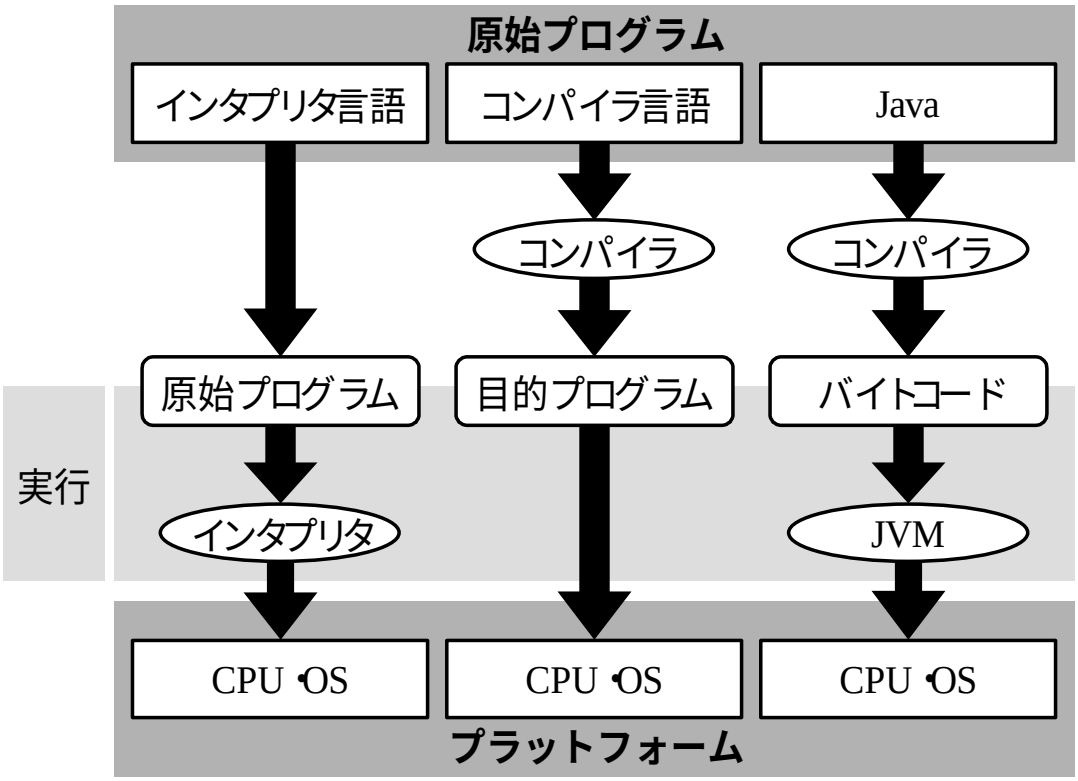


図-2.1　言語プロセッサによるプログラムの実行の違い

(3) オブジェクト指向プログラミング言語

Java はオブジェクト指向のプログラミング言語である．ここでは，オブジェクト指向について説明する．

1) クラスとオブジェクト

クラスとオブジェクトは，全ての Java プログラムの構成要素を形成している．

オブジェクトとは，状態と動作の両方を定義する記憶領域のことである．記憶領域はメモリであることもディスクであることもある．状態は，一連の変数と値によって表現される．動作は，一連のメソッドとそれによって実装されるロジックによって表現される．従って，オブジェクトとは，データとそれを操作するメソッドの組み合わせであると言える．

クラスとは，オブジェクト作成の基盤となるテンプレートのことである．つまり，オブジェクトはクラスのインスタンスである．新しいオブジェクトを作成する仕組みのことをインスタンス化と呼ぶ．これらを図-2.2 に示す．

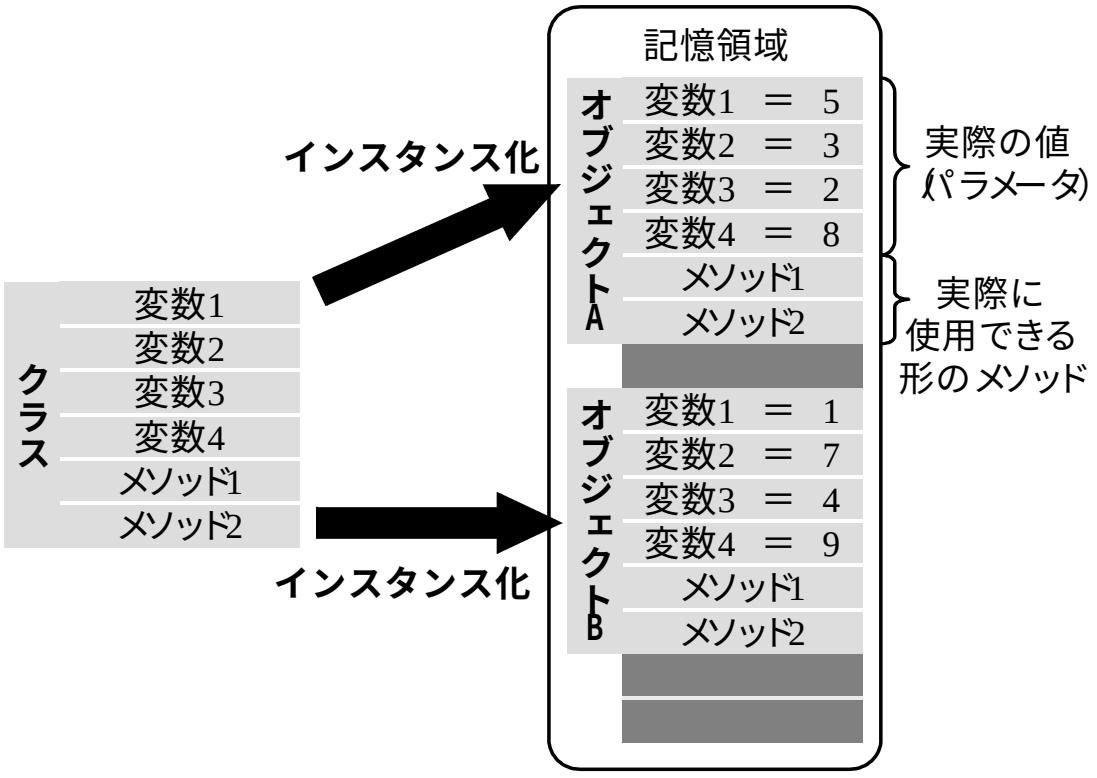


図-2.2 クラスとオブジェクト

2) **オブジェクト指向プログラミングの言語機能**

オブジェクト指向プログラミングの基盤となる言語機能として，以下の 3 つが挙げられる．

a) **カプセル化**

データとデータを操作するコード（メソッド）を関連付けるメカニズムである．他のソフトウェアからカプセル化されたデータに直接アクセスすることはできない（図-2.3 参照）．

カプセル化には重要な利点がある．第 1 に，情報を保存するために使用するデータの形式を簡単に変更することができる．つまり，変更したいデータへのアクセスはそこにカプセル化されたメソッドでのみ許されるため，データ構造を変更する時など，メソッドを呼び出すほかのソフトウェアを修正せず，呼び出されるクラスのメソッドを修正すればよい．第 2 に，機密情報へのアクセスの制御が簡単になる．例えば，データの特定の部分を修正するメソッドでは，ログインとパスワードを受け取るようにすることができる．第 3 に，複数のスレッド（1 つの処理のまとまり）からのデータアクセスを同期化することが可能になる．

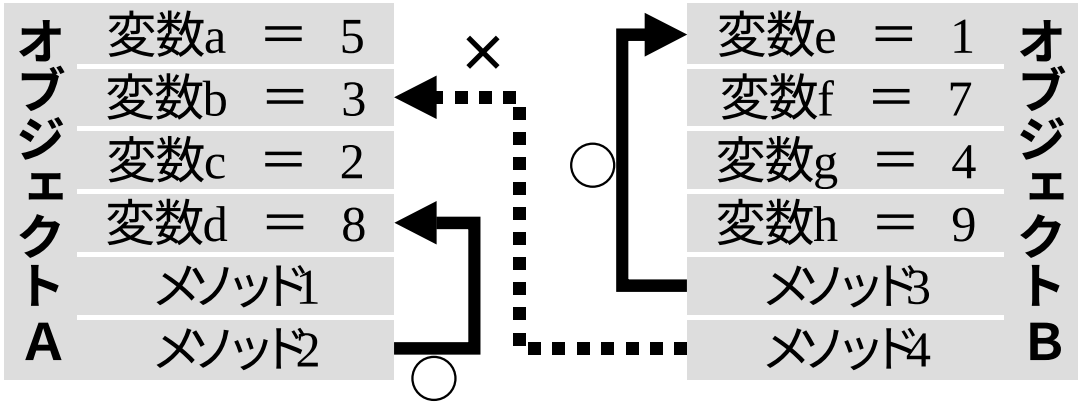


図-2.3 カプセル化

b) **継承（インヘリタンス）**

あるクラスを特化することによって，他のクラスを定義するメカニズムである．1 つのクラスによって既に定義されている構造体や動作を基に別のクラスを定義できるため，ソフトウェアの再利用が促進される．

クラス Y がクラス X を継承する場合，Y を「X のサブクラス」と呼び，X を「Y のスーパークラス」と呼ぶ．また，「Y は X を拡張する」と表現することもある．この様にして，一連のクラスに対して継承の階層構造が形成される．

1 つのクラスは複数のサブクラスを持つことができる．しかし Java では，1 つのクラスは直接的なスーパークラスを 1 つしか持つことができない．継承階層のルートには Object と言う名前のクラスがあり，全ての Java クラスは，この Object クラスを直接的又は間接的に継承している（図-2.4 参照）．

（注）ここでのクラス A～F は，全て Object クラスのメソッドを継承する．

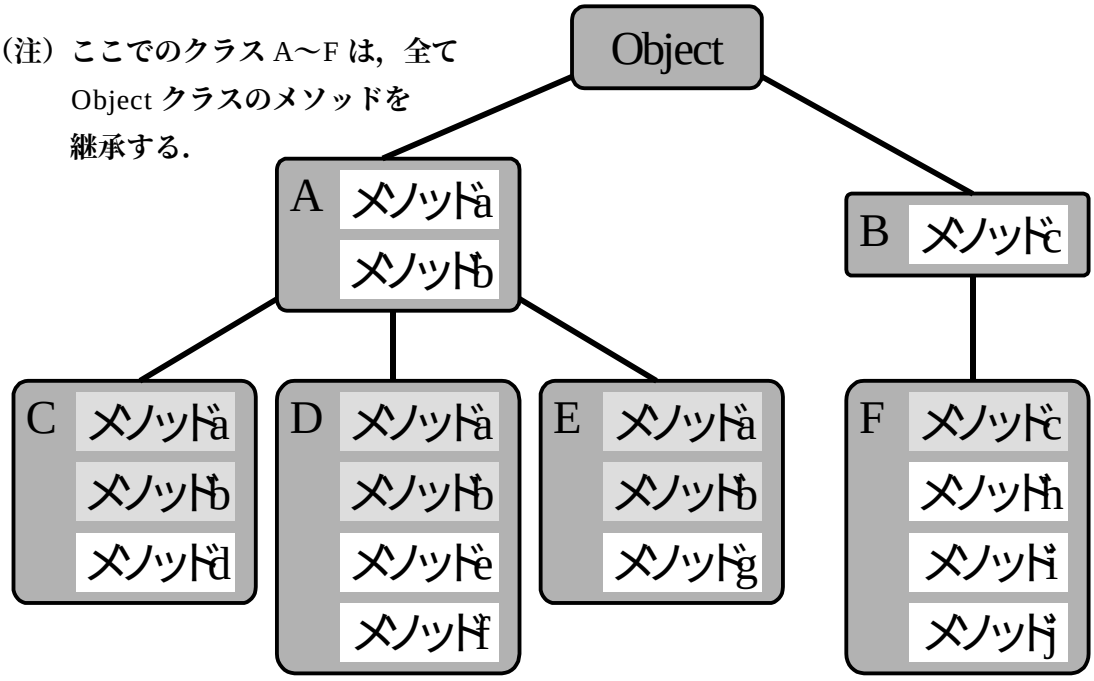


図-2.4　クラスの継承階層

c)　ポリモルフィズム

「1 つのインタフェース，複数の実装」と表現されるメカニズムである．複数のクラスで実装するメソッドを抽象クラスにし，実装したいクラスをそのクラスのサブクラスにすることで継承できる．このとき，抽象クラスはインスタンス化できない．

では，どのような時にポリモルフィズムが役立つのだろうか．例えば，図-2.5 のような図形のクラスがあったとする．ここで，図形の総面積を求めるプログラムを作成するならば，普通のプログラムであれば図形の種類に応じたルーチンを呼び出す必要がある．プログラムの変更も作業が増大する．しか

し，ポリモルフィズムを用いた場合，全て Area メソッドを呼び出すだけで実行でき，プログラムの変更も必要ない．

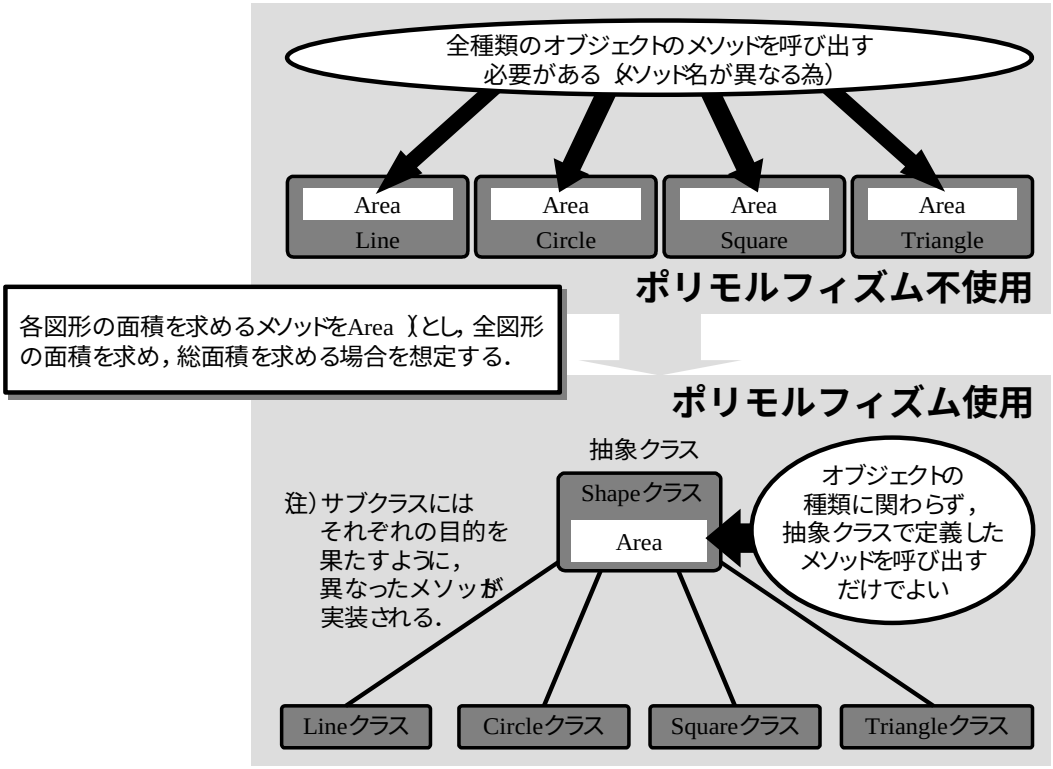


図-2.5　ポリモーフィズム

2.1.3　その他の基礎知識

(1)　アプリケーションとアプレット

Java で作成できるプログラムには，以下のような２つの種類がある．

1) アプリケーション

JVM によって直接実行できるプログラム．

2) アプレット

実行には Web ブラウザが必要．米 Microsoft 社の InternetExplorer では Ver.3.0 以降，米 Netscape 社の NetscapeNavigator/Communicator では Ver.2.0 以降のものであれば，JVM が組み込まれておりアプレットが実行可能である．

しかし，JDK（Java Development Kit）に含まれているアプレットビューアを用

いることで，Web ブラウザなしでもアプレットをテストできる．

本稿で取り上げたプログラムは，全てアプリケーションである．

(2) Java クラスライブラリ

Java 言語そのものに加え，Java では豊富なクラスライブラリが提供されている．これらのクラスライブラリには数多くの標準クラスとメソッドが含まれており，すべての Java プログラムから利用できる．
表-2.2 に良く使われるクラスライブラリをまとめた．また，パッケージとは，クラスの集まりの事である．

表-2.2 Java の主なパッケージ

パッケージ	説明
java.applet	アプレットを作成する
java.awt	GUIを作成する為の Abstract Window Toolkit(AWT)を提供する
java.awt.event	AWTコンポーネントからのイベントを処理する
java.awt.image	画像処理を行う
java.beans	Javaソフトウェアコンポーネントの基盤を形成する
java.io	入出力をサポートする
java.lang	Javaの中枢機能を提供する
java.net	ネットワーキングを可能にする
java.util	ユーティリティ機能を提供する

(3) Java Development Kit（JDK）

JDK は，Sun Microsystems 社がリリースした Java アプリケーションとアプレットの作成･実行ツールである．2000 年 1 月 19 日現在，Windows 用，Solaris 用共に Ver.1.2.2 がリリースされている．

2.2 JavaRMI(Java Remote Method Invocation)

JavaRMI は，プログラマが「分散 Java アプリケーション」を作成するための技術である．「分散 Java アプリケーション」とは，ある仮想マシンから他の仮想マシン（他のホスト上にする場合もある）のオブジェクト（リモートオブジェクト）のメソッドを起動するような処理を含む Java アプリケーションのことである．

2.2.1 JavaRMI の特徴

以下に JavaRMI の特徴を示す．

1) 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．これにより，インタフェースに変更が無ければ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無く，逆も成り立つことを意味する．この特長は 2.1.2 節でも述べた Java のポリモルフィズム機能からくるものである．

2) 記述の容易性

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・プログラムはリモート・オブジェクトへの参照として取得されるが，通常のオブジェクトと扱い方は同じである．

この様に，RMI の実装に他のネットワークの技術・知識を特に必要としない為，プログラムの記述が容易である．

3) 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ので，JDK1.1 以降の環境があれば，どのようなシステムでも動作させることができる．

4) データ送受信の容易性

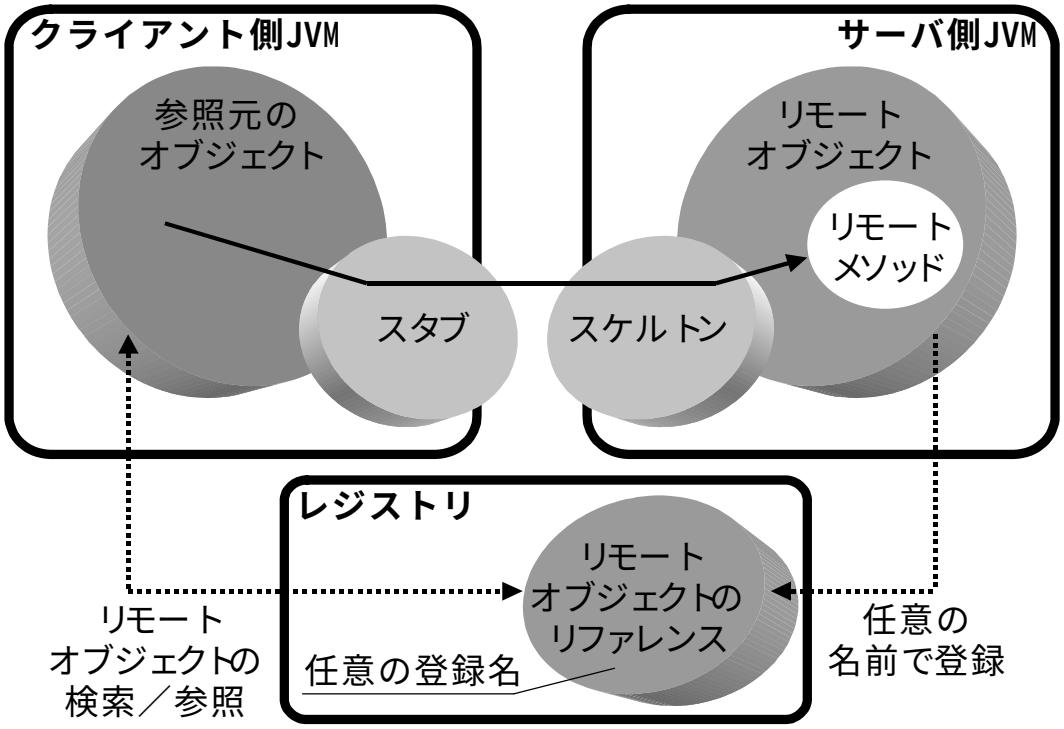
RMI では，メソッドを呼び出す際の引数，返り値としてオブジェクトを受け渡すことができるため，データを送信する形式（バイト列）に変換する必要が無く，送受信が簡単である．それにより，実装も容易になる．

5) 並列処理（マルチスレッド処理）が可能

Java には，プロセス内で複数のスレッドの動作を調節する仕組みが備わっている．スレッドとはプログラムの実行単位であるプロセス内の実行の流れであり，論理的な処理の単位を指す．これによって RMI もマルチスレッドで処理でき，複数のクライアントからの要求は並列に処理される．

2.2.2 JavaRMI の概念

JavaRMI の概念を図-2.6 に示す．実線はアクセスの流れである．この概念図の用語については後述する．



（注）この図では，クライアントからサーバにアクセスする場合を表している．

図-2.6 JavaRMI の概念

2.2.3 JavaRMI システムの 4 層構造

JavaRMI を用いる場合, 異なる JVM 上に存在するオブジェクトのメソッド（リモート・メソッド）をどの様に呼び出すのだろうか.

RMI を使用したシステムは，図-2.7 の様に 4 つの層から構成される.

(1) アプリケーション層

クライアントとサーバのコードが属する層である．この層では，コードの記述時に以下のことに気をつけなければならない.

1) リモート・メソッドを利用するには，インタフェースの中にリモート・メソッドを宣言する必要がある．1 つのメソッドに複数のインタフェースを宣言することも可能だが，Java.rmi.Remote クラス（抽象クラス）のサブクラスとして作成しなけれ

ばならない.

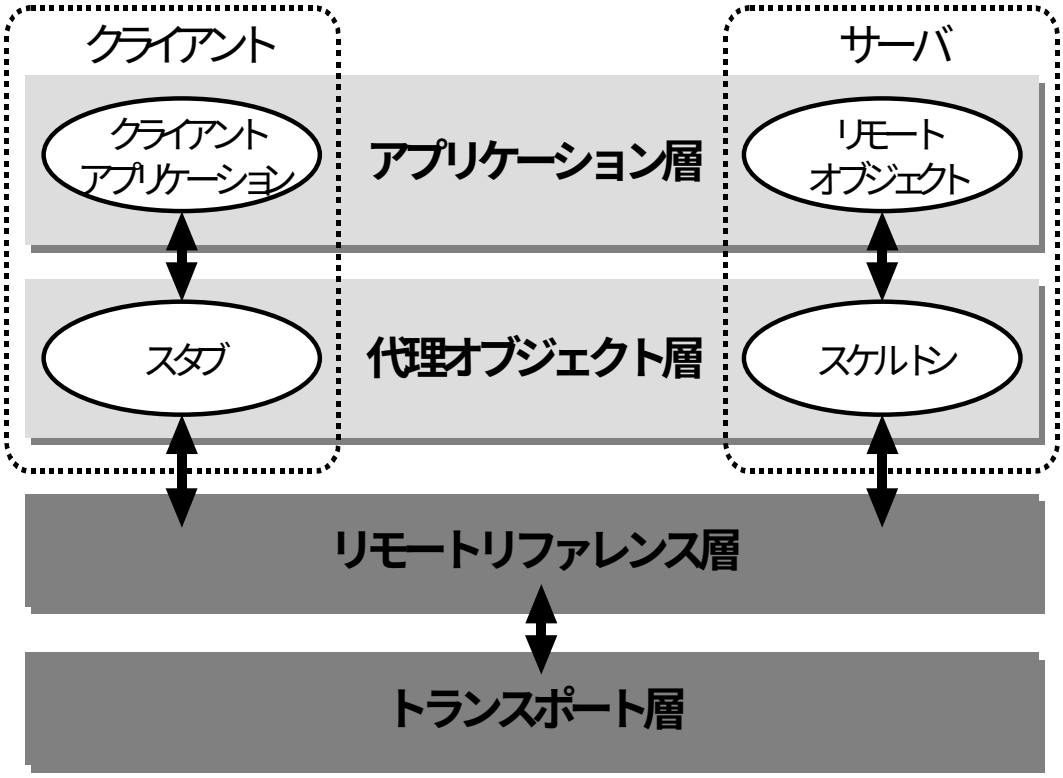


図-2.7 JavaRMI システムの構造

2) リモート・オブジェクトの作成時には，RemoteException オブジェクト（マシン間のやり取りに問題が生じた時発生するオブジェクト）の処理をするコードが必要である.
3) リモート・インタフェースとして宣言したメソッドをインプリメントするオブジェクトには，エクスポート（オブジェクトをリモート・マシンからでも利用できるようにすること）をする必要がある．その為，UnicastRemoteObject クラスのサブクラスにするか，UnicastRemoteObject クラスの exportObject()メソッドを利用する.
4) エクスポートしたオブジェクトは，レジストリと呼ばれるネーム・サーバに登録し，ここからリモート・オブジェクトのリファレンス（参照）を得る．他のオブジェクトのリファレンスを利用するには，最初にレジストリから取得したリファレンスを利用する．リモート・オブジェクトのリファレンスをリモート・インタフェースにキャストすれば，そのリモートインタフェースを利用してリモート・メソッドを起動できる.

アプリケーション層について，図-2.8 にまとめた.

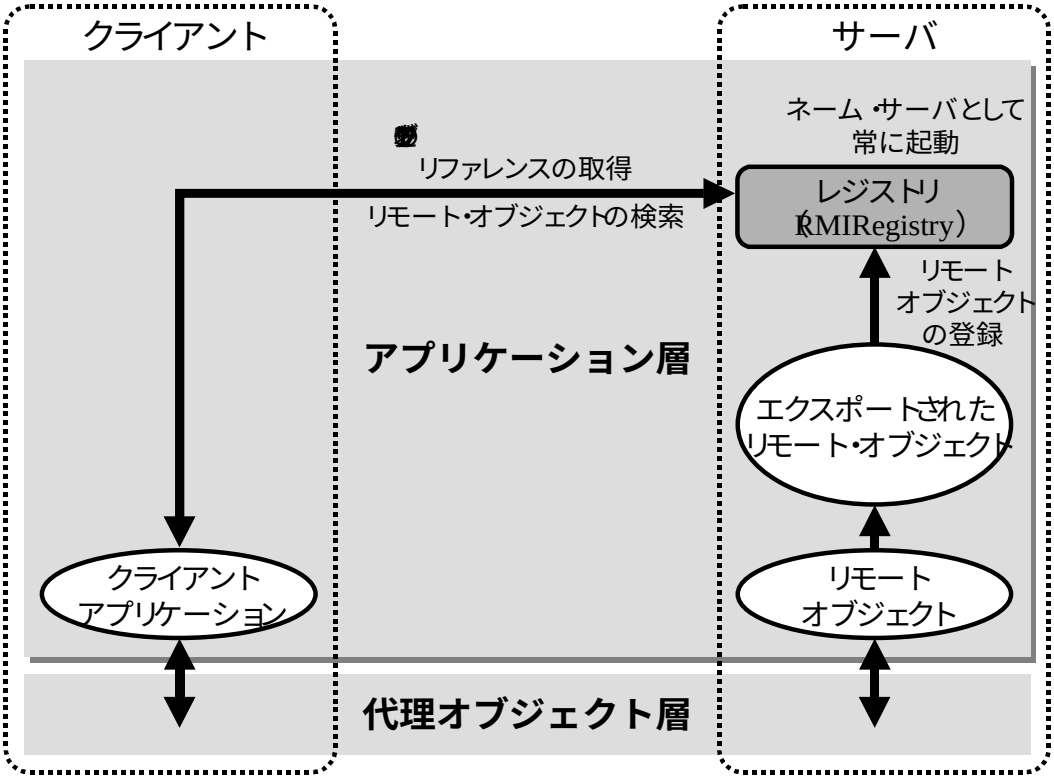


図-2.8　アプリケーション層

(2)　スタブ／スケルトン(代理オブジェクト) 層

スタブやスケルトン（これらを代理オブジェクトと呼ぶことがある）が属する層である．スタブはクライアントと，スケルトンはサーバでリモート・オブジェクトの代わりをする．

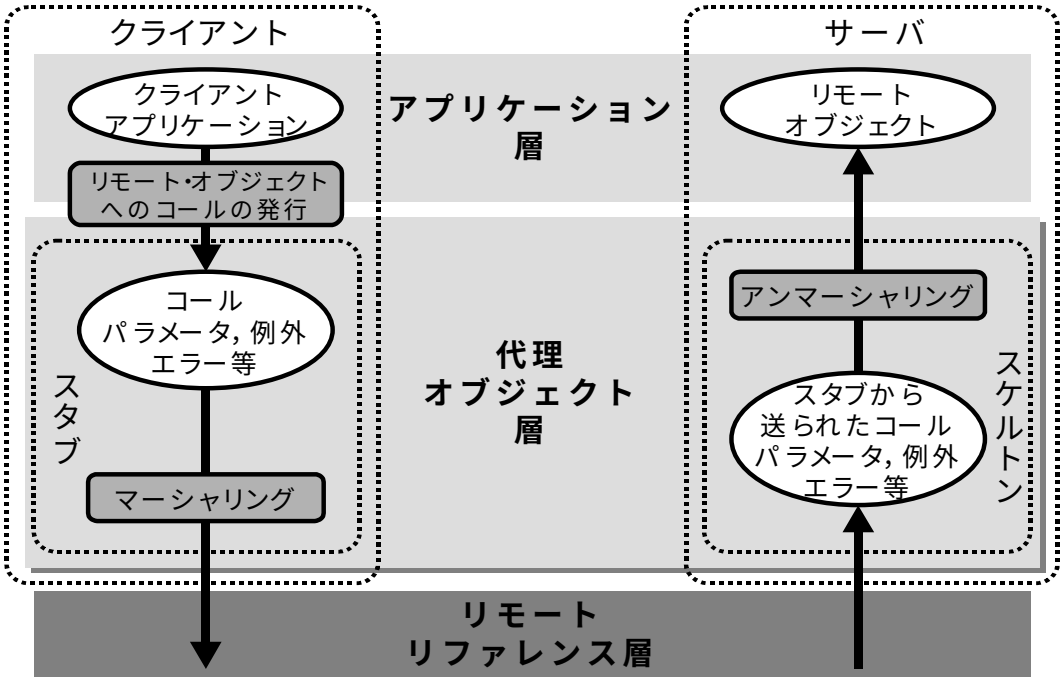
この層では以下のようなことを行うが，システムを開発するときには RMI コンパイラ（後述）を用いて作成するので，実際にこれらのことを考慮する必要はない．

1) スタブ
   a) リモートオブジェクトへのコールの発行を行う．
   b) マーシャリングとアンマーシャリングを行う．ここでは，マーシャリングとは，パラメータ，例外，エラーなどをネットワークで転送できる形に変換することで，アンマーシャリングとは，マーシャリングされて送られてきたリモート・メソッドからの戻り値や例外を復元することである．
   c) リモート・オブジェクトが実装している物と同じリモート・インタフェースを実装する．インタフェースを用いてクラス名，メソッド名を統一することで，あたかもクライアントがリモート・メソッドの参照を得ているようにす

るためである．

2) スケルトン
   a) スタブから送られてきたコールとパラメータを受け取る．
   b) アンマーシャリングとマーシャリングを行う．ここでは，スタブから受け取ったコールとパラメータをアンマーシャリングして，本物のリモート・オブジェクトに送り，メソッドが実行されたら戻り値と例外をマーシャリングしてクライアントに送る．

代理オブジェクト層について，図-2.9 にまとめた．



（注）この図では，クライアントからサーバにアクセスする場合を表している．

図-2.9　代理オブジェクト層

(3)　リモート･リファレンス層

この層では，代理オブジェクト層と，通信処理を行うトランスポート層の仲介をする．トランスポート層とのやり取りでは，ストリーム型（コネクション型）のプロトコルが使用される．この層では以下のことを行う．（図-2.10 参照）

1) オブジェクトの複製が作られたときに，その操作を行う．オブジェクトの複製を作れば，複数のプログラムで実質的に同じオブジェクトをエクスポートするといった

ことも簡単になる.

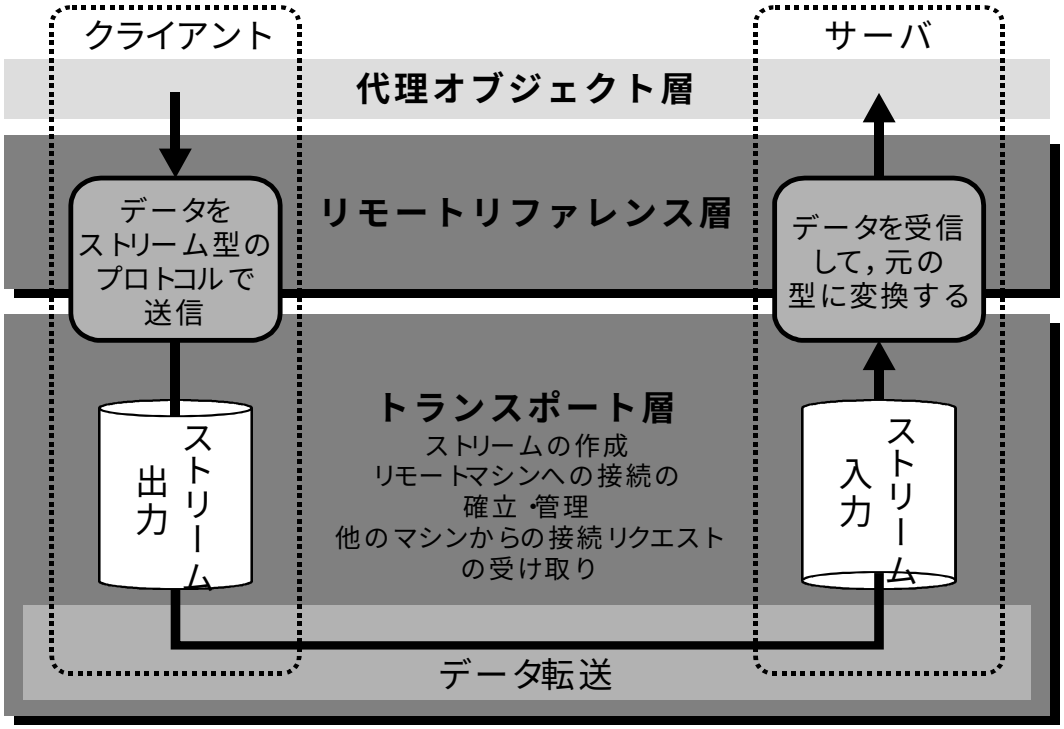
2) オブジェクトを永続化する.
3) 通信障害からの回復を行う.

(4) トランスポート層

この層では，マシン間の通信処理を行う．通信処理はこの層だけで独立して行うことになっている為，たとえ使用する通信プロトコルを変えたり，ストリームの暗号化，データの圧縮といった機能を持つように変更しても，上位層の変更の必要はない.

ストリームとは，データのやり取りを行うときにその源流と宛先を表す抽象的な通信路の概念である（図-2.10 参照）．この層では以下のようなことを行う.

1) ストリームを作成し，マシン間でデータをやり取りする．RMI のデフォルトの通信プロトコルは TCP/IP（Transmission Control Protocol/Internet Protocol）である為，データは基本的にストリームの形で伝送される.
2) リモートマシンへの接続を確立し，その管理（接続が正常に機能しているかどうかの監視も含む）を行う.
3) 他のマシンからの接続リクエストを受け取る.



（注）この図では，クライアントからサーバにアクセスする場合を表している.

図-2.10 リモート・リファレンス層とトランスポート層

2.2.4 JavaRMI のクラスとインタフェース

図-2.11 に，リモート・オブジェクトを作成する為に用いられるクラスとインタフェースの継承関係を示す．このような階層構造になっているのは，java.lang.Object クラスの幾つかのメソッドの機能を様々に変える必要があるからである．

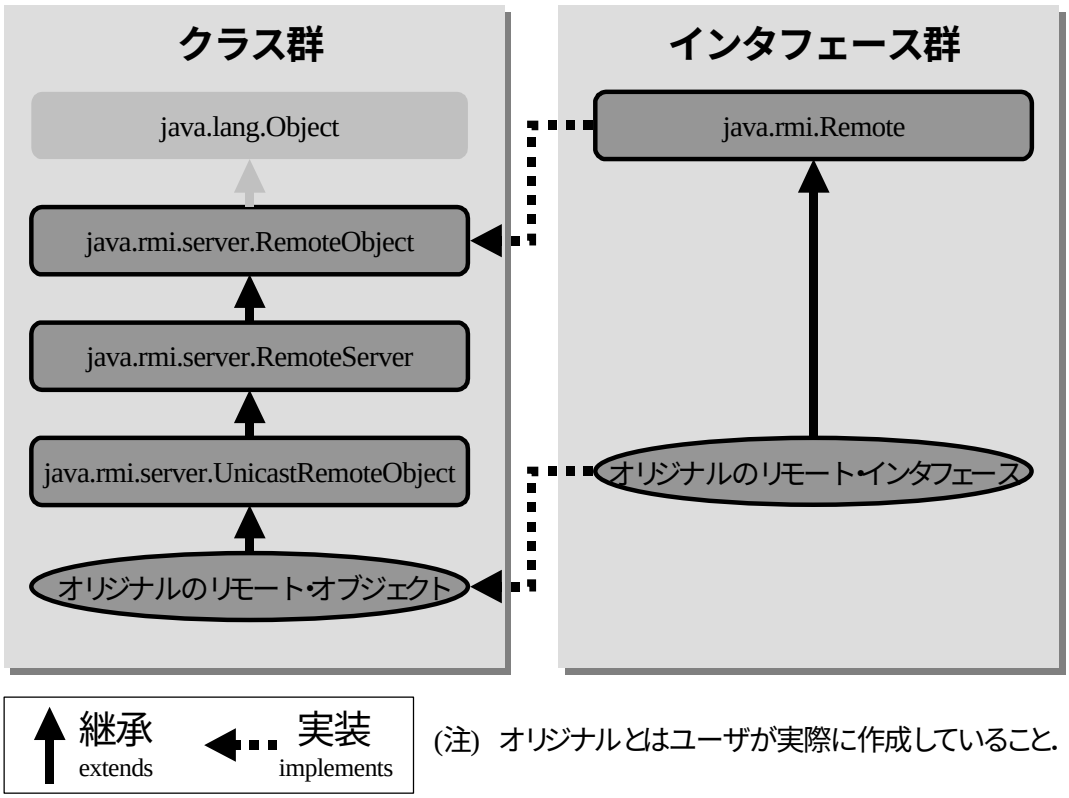


図-2.11 リモート・オブジェクトの階層構造

1) java.lang.Object クラス

あらゆるオブジェクトのベースとなるクラス．実際には，RemoteObject クラスで幾つかのメソッドが分散オブジェクトに対応できるようオーバーライドされているため，事実上，RemoteObject がベース・クラスとなる．

2) java.rmi.Remote インタフェース

Remote インタフェースは，RMI の中でリモート・オブジェクトやスタブ等を Remote 型のオブジェクトとして扱うために用いられる．Remote インタフェースは抽象クラスであるので，メソッドの実装やフィールドを持たない．

3) java.rmi.server.RemoteObject クラス

RemoteObject クラスは，hashCode，equals，toString メソッドをオーバーライドすることにより java.lang.Object クラスが提供している機能をリモート・オブジ

ェクトから利用できるように拡張したクラスである．

4) java.rmi.server.RemoteServer クラス

RemoteServer クラスは，全てのリモート・オブジェクトの共通のスーパークラスである．RemoteServer クラスは抽象クラスであるが，特に実装しなければならないメソッドは持たない．現在サポートされているサブクラスは UnicastRemoteObject クラスのみである．

5) java.rmi.server.UnicastRemoteObject クラス

このクラスのサブクラスとして新たにリモート・オブジェクトを作成すると，コンストラクタによりリモート・オブジェクトが他のマシンから自動的に参照可能な状態になる．

通常，あるクラスを UnicastRemoteObject クラスのサブクラスにし，Remote インタフェースをインプリメントすれば，独自にリモートアクセス可能なオブジェクトを作成することができる．

2.2.5 JavaRMI の開発手順

JavaRMI の開発手順を図-2.12 に，コンパイラ等の実行コマンドを図 2-13 に示した．

図-2.12 からもわかるように，クライアント側ではクライアントのアプリケーションを作成し，java コンパイラで class ファイルを生成するだけである．しかし，サーバ側ではアプリケーションの作成以外に，リモート・インタフェース実装クラス（class ファイルにコンパイルしたもの）から，rmi コンパイラで代理オブジェクト（スタブ，スケルトン）を作成する必要がある．

また，RMI レジストリを起動するときにはポート番号を指定でき，1024 以上 16000 未満であればどの番号でも指定できる．このとき，オブジェクトの登録や，検索を行うメソッドで引数として渡されるレジストリの場所を表す URL 中で，ポート番号を明示的に指定する必要がある．（デフォルト値は 1099 である．）

RMI システムの起動は，RMI レジストリを起動し，リモート・オブジェクトを登録してからクライアントを実行することで行われる．

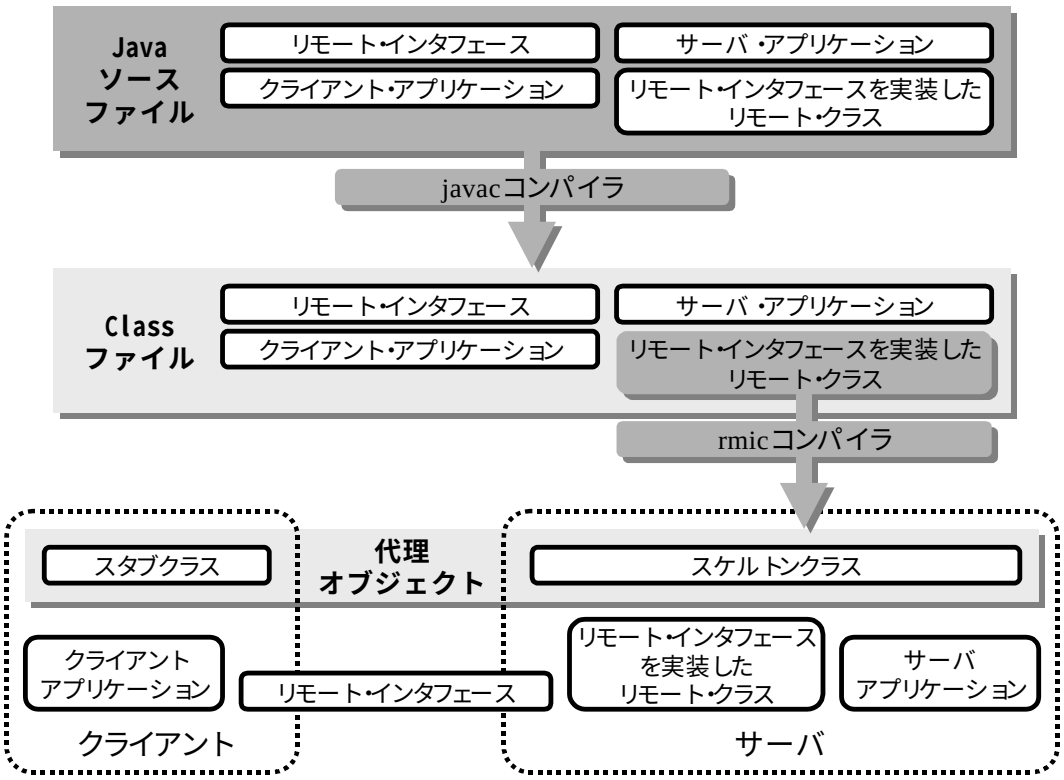


図-2.12　JavaRMI システムの開発手順

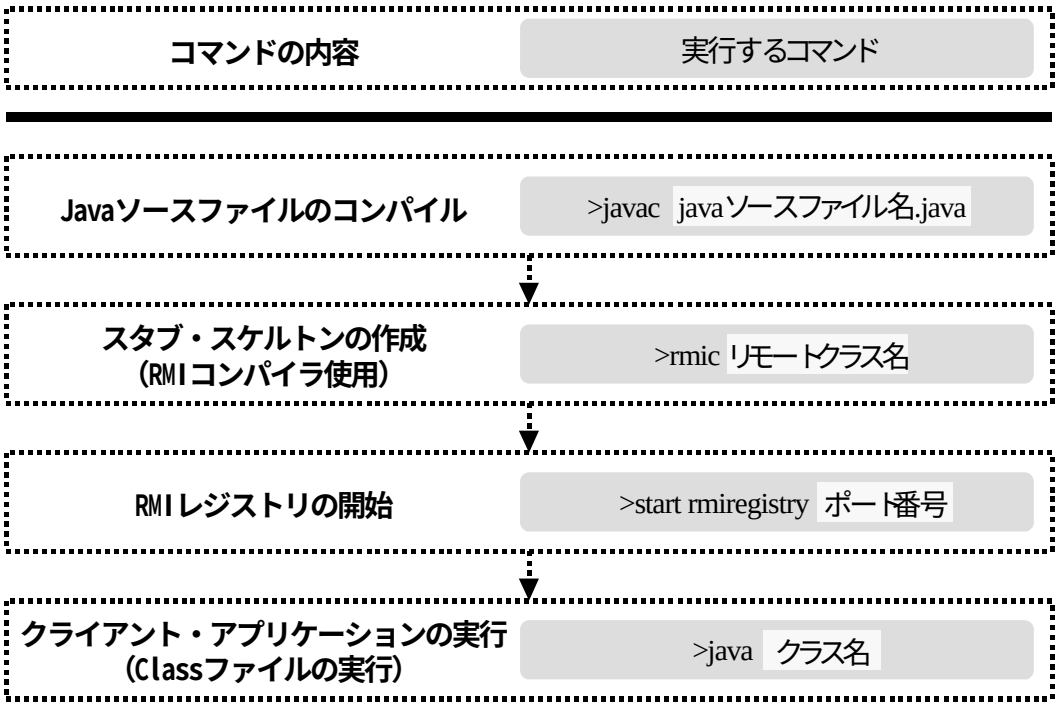


図-2.13　JavaRMI システムの実行

2.3 JavaIDL（Java Interface Definition Language）

2.3.1 CORBA（Common Object Request Broker Architecture）

一般的に，分散オブジェクト環境でのオブジェクトへのアクセスを実現するメカニズムを ORB（Object Request Broker）と呼んでいる．前章で述べた JavaRMI も ORB の 1 つであるといえる．本章で述べる CORBA も ORB の 1 つである．

CORBA は，1989 年にオブジェクト指向の標準化を推進するために設立された OMG（Object Management Group）が，ネットワーク上で異なるマシンや異なる言語で作成されたオブジェクト間で通信を行うための規格として開発したものである．

CORBA は, ネーミングサービスやトランザクションサービス等の共通オブジェクトサービス，よりアプリケーションに近い機能を提供する共通ファシリティ等からなる広範囲の規格である．これに用いることで，クライアントは，呼び出すオブジェクトとそのオブジェクトの提供しているインタフェースを知るだけで，サーバのオブジェクトにアクセスできる．（図-2.14 参照）図-2.15 に，CORBA や CORBAservices から構成される CORBA のアーキテクチャの基本である OMA（Object Management Architecture）の分散オブジェクト環境の概要図を示す．

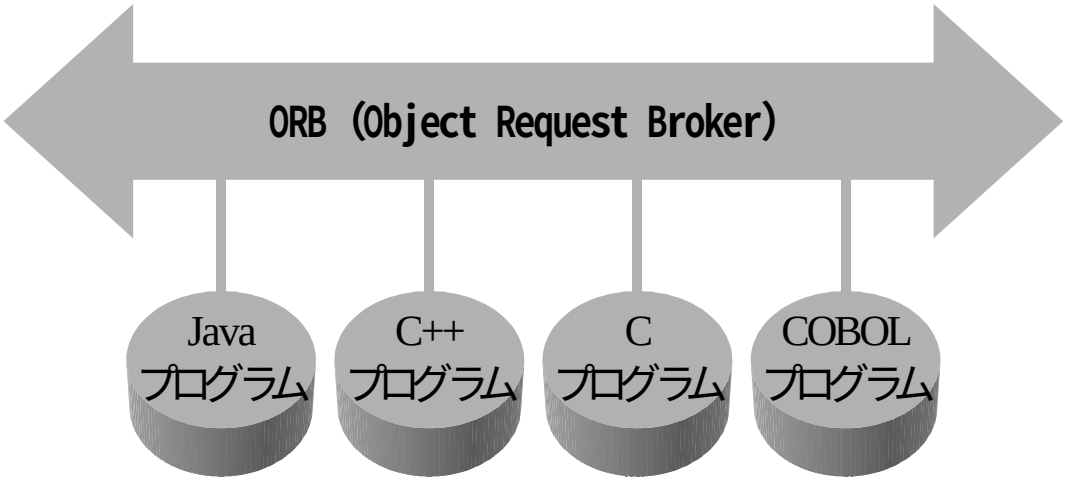


図-2.14 CORBA の異なる言語間での通信

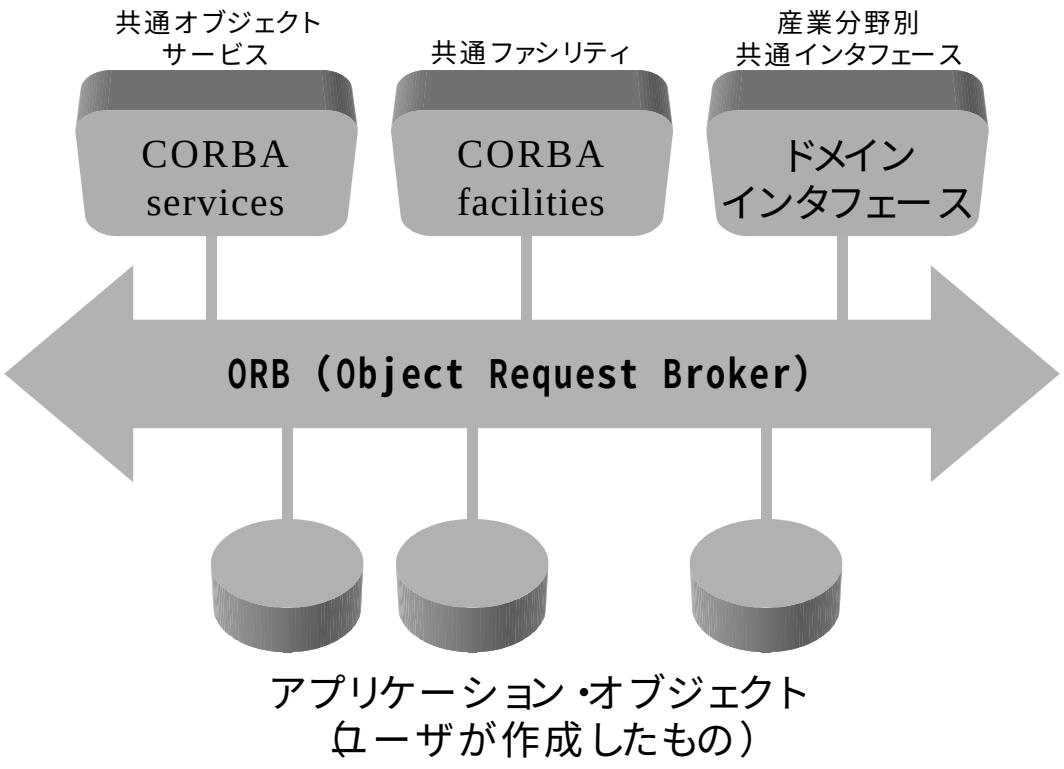


図-2.15　分散オブジェクト環境CORBA の基本アーキテクチャOMA

2.3.2　インタフェース定義言語IDL（Interface Definition Language）

CORBA では，クライアントは呼び出すオブジェクトとそのオブジェクトの提供しているインタフェースを知るだけで，サーバのオブジェクトにアクセスできる．このうちオブジェクトの提供するインタフェースについては，あらかじめクライアントとサーバの間で定義しておく必要がある．インタフェースの定義には，インタフェース定義言語（IDL）という専用の言語が用いられる．

IDL で記述したインタフェース定義は，専用のコンパイラ（IDL コンパイラ）でコンパイルすることにより，クライアントがオブジェクトにアクセスするために必要なコードと，サーバがオブジェクトを実装する際のベースになるコードが生成される．CORBA では，クライアント用の生成コードをスタブといい，サーバ用の生成コードをスケルトンという．

インタフェース定義のコンパイルには，クライアントとサーバのアプリケーションで使用するプログラミング言語の種類に応じて，その言語をサポートしている IDL コンパイラを用いる．例えば，C++でアプリケーションを作成したならば，C++をサポートしている IDL コンパイラを用いなければならない．（図-2.16 参照）

JavaIDL は，Java アプリケーションで CORBA に基づいた ORB の実現で用いられるものである．JavaIDL は，JDK1.2 の Java Enterprise API を構成する要素技術であり，OMG

が規定する CORBA2.0 仕様（1996 年 8 月公開）に準拠した ORB である．JavaIDL を介することで，Java アプリケーションから CORBAIDL や IIOP（Internet Inter-Orb Protocol）に対応したリモート・サービスへ透過的なアクセスが可能となる．

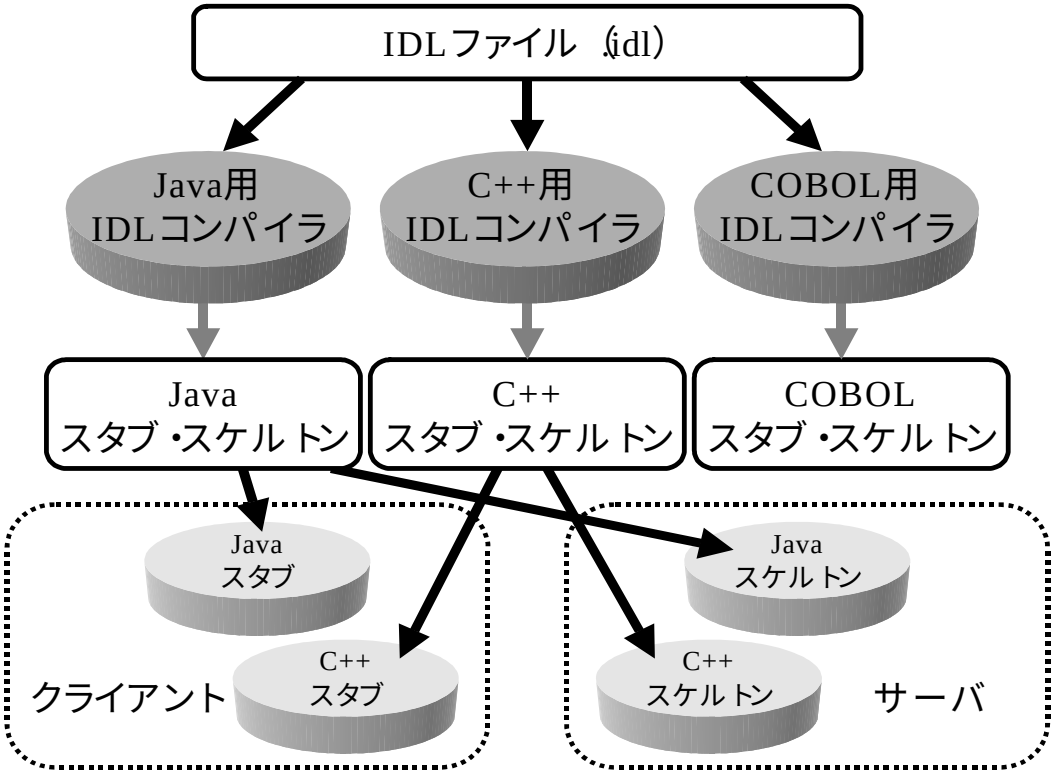


図-2.16 IDL のコンパイル

2.3.3 JavaIDL の特徴

以下に JavaIDL の特徴を示す．

1) 開発言語への非依存性
   異なる言語で開発されたアプリケーションと Java との統合化を実現できる．
2) CORBA2.0 のネーミング・サービス機能を完全にサポート
   CORBA 共通オブジェクト・サービス中のネーミングサービスを完全サポートしている．（詳細は後述する．）
3) 動的起動が可能
   インタフェース・リポジトリからオブジェクトのインタフェース情報を取得することにより実行時にオブジェクト呼び出しの要求を組み立てることができる．（詳

細は後述する.）

4) **オブジェクトの永続性**

CORBA のオブジェクトには 2 種類あり，永続オブジェクトと一時オブジェクトがある．永続オブジェクトは，サーバ・プロセスが停止した場合でも存在することができる.

2.3.4 JavaIDL（CORBA）の概念

JavaIDL（CORBA）の概念図を図-2.17 に示す．大きな矢印はアクセスの流れである.

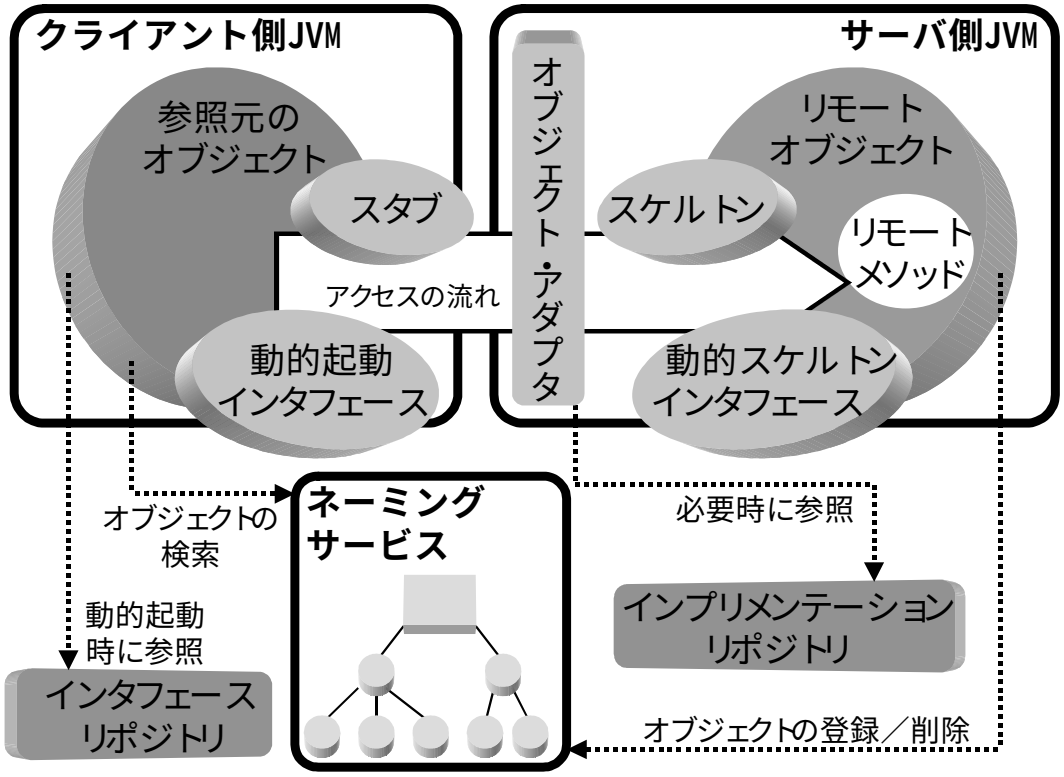


図-2.17 JavaIDL（CORBA）の概念図

それでは，概念図の用語と CORBA のその他の用語について説明する.

(1) 静的スタブと動的起動インタフェースDII（Dynamic Invocation Interface）

クライアントがオブジェクトを呼び出す方法には，IDL コンパイラが生成するスタブを使用する静的な方法と，動的起動インタフェースを使用する動的な方法がある．

1) 静的スタブ

静的スタブは IDL によるインタフェース定義を IDL コンパイラでコンパイルすることで生成される．スタブはローカル・マシン上のリモート・オブジェクトのように動作する．

静的起動の場合，オブジェクト呼び出しの詳細がスタブ・コードに隠蔽されているため，分散を意識せずにリモート・オブジェクトを呼び出すことができる．

2) 動的起動インタフェース（DII）

これに対して，動的方法は，実行時にインタフェース・リポジトリからオブジェクトのインタフェース情報を取得することで，要求を組みたてることができるので，プログラミング時に必ずしもインタフェース定義を入手している必要がない．また，DII に動的スケルトン・インタフェースと組み合わせることで，ファイア・ウォールやゲートウェイを実装することができる．

3) 起動方式（クライアントがオブジェクト呼び出しの応答を受け取る方法）の種類

静的方法で，遅延同期起動，oneway 起動をすることはできない．（oneway 起動は IDL で oneway を指定したオペレーションのみ可能）

a) 同期起動

呼び出し要求を行ってから応答が返ってくるまで，クライアント・アプリケーションが待ち状態になる方式．

b) oneway 起動

クライアント・アプリケーションがサーバからの応答をコールバック・オブジェクトで受け取るような場合に使用．

c) 遅延同期起動

呼び出し要求を行うとすぐにクライアント・アプリケーションに制御が戻る．クライアント・アプリケーションは，後からポーリングを行うことによって応答を受け取る．

(2) インタフェース・リポジトリ

オブジェクトのインタフェース定義情報を格納するためのリポジトリ．（データベース）主に動的起動インタフェースや動的スケルトン・インタフェースを使用する動的なアプリケーションから利用される．CORBA のアプリケーションを開発するための CASE ツールが利用している例もある．

(3) ORB インタフェース

クライアント及びサーバアプリケーションが共通に利用することのできる ORB サービスがまとめられている.

ORB インタフェースの機能として以下のようなことが上げられる.

1) ORB の初期化と初期オブジェクト・リファレンスの取得
2) 動的起動要求の組み立て
3) オブジェクト・リファレンスの文字列への変換，およびその逆変換
4) デフォルトのコンテキスト・オブジェクトの取得

ORB インタフェースは Java では org.omg.CORBA.ORB クラスにマッピングされている.

(4) 疑似インタフェースと疑似オブジェクト

ORB オブジェクトは，実際には CORBA の分散オブジェクトとして実装されているわけではなく，Java，C++などのプログラミング言語で実装されている.

このオブジェクトを疑似オブジェクト（pseudo object）といい，疑似オブジェクトのインタフェース定義を疑似 IDL（PIDL: pseudo-IDL）という.

(5) オブジェクト・アダプタとサーバント

オブジェクト・アダプタは，サーバ側 ORB の構成要素である.

1) サーバント

   インタフェース定義に JavaIDL を用いた場合，サーバ・アプリケーションでは，オブジェクトのインタフェースを Java のクラスとして実装する．このクラスをサーバント・クラスといい，そのオブジェクト・インスタンスのことをサーバントという.
2) オブジェクト・アダプタ

   オブジェクト・アダプタは，主にクライアントからのオブジェクト呼び出しの要求を，適切なオブジェクト・インプリメンテーションの適切なサーバントの適切なメソッドにディスパッチすることである．以下にその他の主な機能を述べる.
   a) オブジェクト・リファレンスとサーバントの作成，及びそれらのマッピング
   b) オブジェクト・インプリメンテーション（リモート・オブジェクト）の登録
   c) オブジェクト・インプリメンテーションの活性化と非活性化（オペレーション実行の為にオブジェクトの準備を行う，もしくはオペレーション実行後の

後始末をすることをいう.）

d) クライアントからの要求に対応するメソッドの起動

e) クライアントの認証情報の取りだし.

3) 基本オブジェクト・アダプタ（BOA）

CORBA の設計段階で，オブジェクトの生存期間，実装スタイルなどによって，ORB に対する要求は多岐にわたることから ORB のサーバ機能を ORB コアとは切り離して，交換可能なコンポーネントとした．これが基本オブジェクト・アダプタである.

初期の CORBA では仕様が曖昧だったため，ORB 製品ごとに異なる API が提供され，アプリケーションのポータビリティが損なわれた.

4) ポータブル・オブジェクト・アダプタ（POA）

CORBA2.2 版で取り入れられ，BOA の機能を詳細に定め，ポータビリティを確立したもの.

(6) 静的スケルトンと動的スケルトン・インタフェース

オブジェクト・アダプタがクライアントからのオブジェクト呼び出し要求をサーバ・アプリケーションに渡す方法には，IDL コンパイラが生成する静的スケルトンを用いる方法と，動的スケルトン・インタフェースを用いる方法の 2 種類がある.

1) 静的スケルトン

静的方法では，インタフェースごとに IDL コンパイラがスケルトン・クラスを生成する．これを継承することで，各オペレーションに対応するメソッドを実装したクラスを作成する．オブジェクト・アダプタは，クライアントが起動したいオブジェクト，オペレーション，そのオブジェクトのインタフェースを判断し，各メソッドをディスパッチするので，アプリケーション開発者は，各メソッドの実装だけに専念できる.

2) 動的スケルトンインタ・フェース

動的方法では，オペレーションや属性が呼び出されると，オブジェクト・インプリメンテーションの invoke メソッドが呼び出され，クライアントからのオブジェクト呼び出し要求の情報が格納された ServerRequest オブジェクトと呼ばれる疑似オブジェクトが引数として渡される．そして，オブジェクト・インプリメンテーション自身で，要求されたオペレーションを判断して，それに応じた処理を実行することができる.

また，動的起動を用いることで，ファイア・ウォールやゲートウェイの実装が可能となる.

(7) インプリメンテーション・リポジトリ

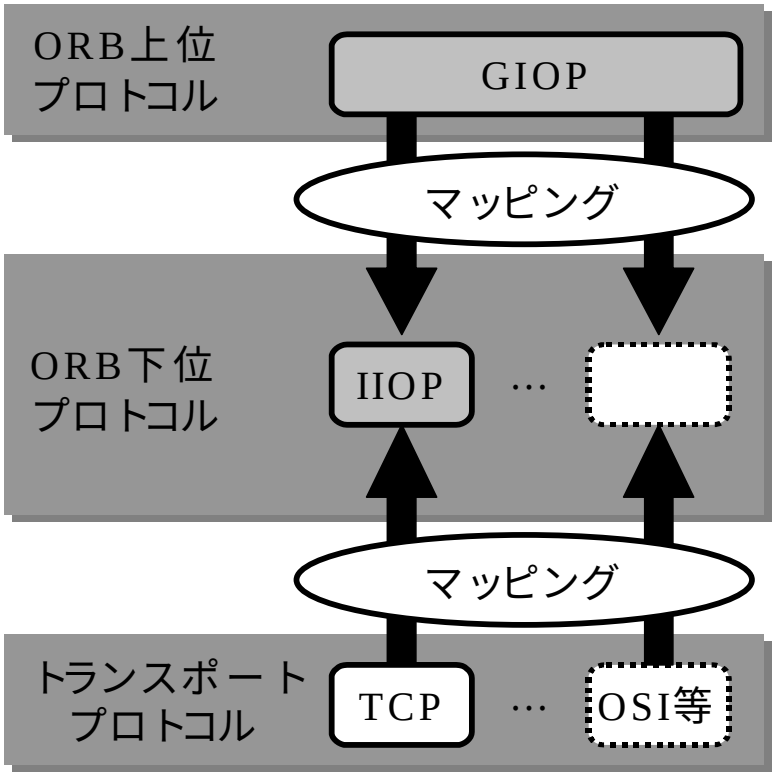
クライアントからの要求受け取り時に，そのオブジェクトを処理すべきサーバ・プロセスが起動されていないとき，オブジェクト・アダプタは，インプリメンテーション・リポジトリから，サーバ名とサーバ起動に必要な情報（Java のクラス名等）を検索し，サーバ・プロセスを自動起動することができる．

このとき，アプリケーション開発者は事前に，インプリメンテーション・リポジトリに，サーバ名とサーバ起動に必要な情報をサーバごとに登録しておく必要がある．

(8) 通信プロトコル

CORBA では，TCP/IP（Transmission Control Protocol/Internet protocol）や OSI（Open System Interconnection）といった個々のトランスポート・プロトコルごとに個別に規則を取り決めるのではなく，あらかじめ ORB 上位プロトコルとして GIOP（General Inter-ORB Protocol）と呼ばれるプロトコルを定め，メッセージ交換規則，データ表現規則等を取り決める．そして，それを各トランスポート・プロトコルにマッピングするというアプローチを取っている．（図-2.18 参照）

例えば，TCP/IP を ORB 下位プロトコルにマッピングしたものが IIOP（Internet Inter-ORB Protocol）である．



これらの必須プロトコルに加えて，オプション・プロトコルも定められている．
（例）ESIOPs

図-2.18 ORB 間プロトコル準拠のために必須な部分

2.3.5 JavaIDL の開発手順

JavaIDL の開発手順を図-2.19 に，コンパイラ等の実行コマンドを図-2.20 に示した．この図では，IDL コンパイラによって生成される静的スタブ，静的スケルトンを用いた静的起動を場合を示している．

まず，JavaIDL を用いてインタフェース定義を行う必要がある．そして，記述したインタフェース定義から idltojava コンパイラでスタブ，スケルトン，ホルダ，ヘルパ等の Java ソースファイルを生成する．

次に，サーバ・オブジェクト実装クラス，クライアント・アプリケーション，サーバ・アプリケーションを作成する．そして，これらの Java ソースファイルを javac コンパイラでクラスファイルに変換する．作成されたクラスファイルは図のようにクライアント，サーバ側に分けられる．これでシステムの起動に必要な全てのクラスファイルが作成された．

次にシステムの起動である．まず，tnameserv コマンドを用いて JavaIDL ネーミングサービスを起動する．ネーミング・サービスを起動するときにはポート番号を指定でき，その場合，クライアント，サーバのアプリケーションを起動する際に，ポート番号を明示的に指定する必要がある．デフォルト値は 900 である．また，ORB 初期化のため，アプリケーション起動時にホスト名やポート番号等のパラメータを指定したり，プロパティとして指定することもできる（図-2.20 参照）．ネーミング・サービスを起動したら，オブジェクト・インプリメンテーションをネーミング・サービスに登録する．

あとは，サーバ・アプリケーション，クライアント・アプリケーションを起動するだけである．

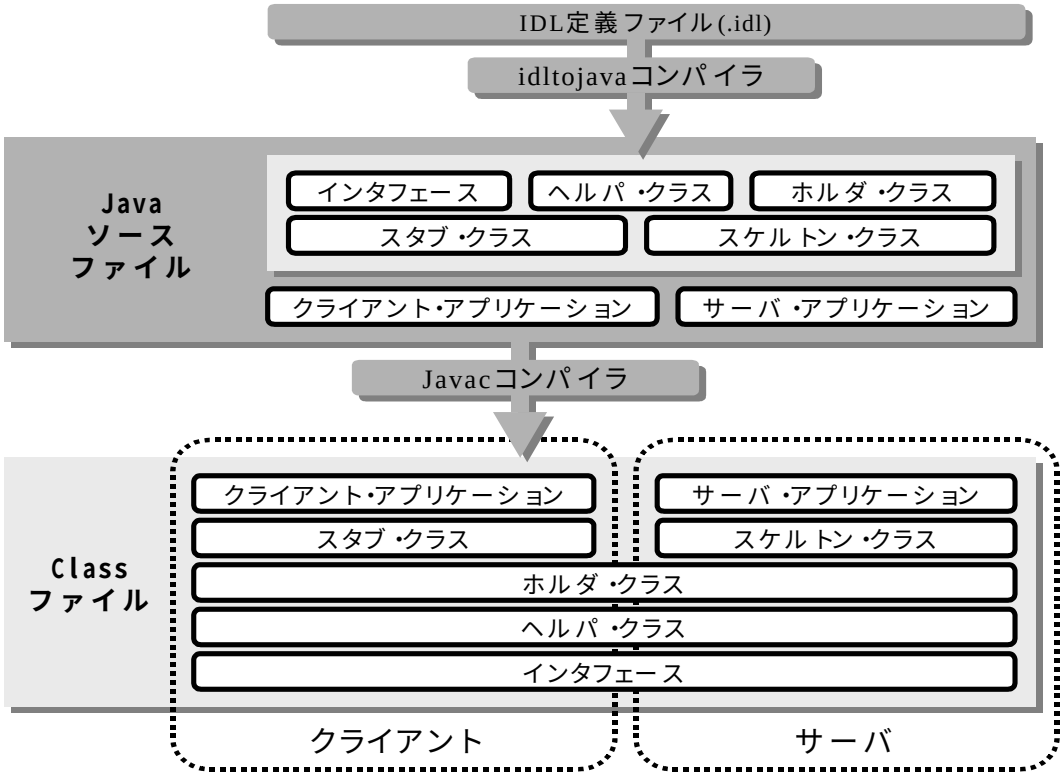


図-2.19　JavaIDL システムの開発手順

コマンドの内容	実行するコマンド
スタブ・スケルトンの作成（idltojavaコンパイラ使用）	>idltojava　IDLファイル名.idl
Javaソースファイルのコンパイル	>javac　javaソースファイル名.java
ネーミング・サービスの起動	>start tnameserv　ポート番号
クライアント・アプリケーションの実行（Classファイルの実行）	>java　クラス名　-ORBInitialPort　ポート番号

ORBの初期化	
プロパティ指定	>java　クラス名　-ORBInitialPort　ポート番号　-ORBInitialHost　ホスト名
オプション指定	>java -Dorg.omg.CORBA.ORBInitialPort=　ポート番号 -Dorg.omg.CORBA.ORBInitialHost=　ホスト名　クラス名

図-2.20　JavaIDL システムの実行手順

第3章　Java による分散オブジェクト環境上のプログラム開発

3.1　MortgageCalc アプリケーション

ここでは，MortgageCalc というアプリケーションを JavaRMI，JavaIDL の両方で作成することで，JavaRMI と JavaIDL の表面的な相違点（サポートする機能の違い等）について見ていく．

3.1.1　アプリケーションの目的

MortgageCalc アプリケーションは，住宅ローンの計算と計算結果の表示を行うもので，まずクライアント起動時に，コマンドラインでサーバ・アプリケーションと，JavaRMI の場合レジストリ，JavaIDL の場合ネーミング・サービスが起動しているサーバ名，家の価格，年当りの金利（単位%）を入力することで，サーバは，頭金（5%，10%，20%）と支払い期間（15 年，30 年）別に月々の住宅ローンの支払い額を計算し，クライアントに結果を返すというものである．ソース・コードは付録にて後述する．

3.1.2　JavaRMI における開発

(1)　開発手順

前述した開発手順に沿って記述するが，JavaIDL と計算部分のコードは変わらないので，PaymentCalc クラスとして作成し，双方で再利用できるようにしておく．こうすることで，アプリケーションが複雑化した場合に双方のアプリケーションに対して同じ修正を加える必要がなくなり，PaymentCalc クラスを拡張するだけで済むようになる．

1) リモート・インタフェースの定義

まず，リモート・インタフェース Calculate を定義する．

Calculate インタフェースには，元々の借り入れ額，年利率，返済期間を引数とし，毎月の支払い額を返す calcPI メソッドと，元々の借り入れ額，年利率，返済期間，年間の地方税/国税，年間保険料を引数とし，毎月の総支払い額を返す calcPITI メソッドが定義される．

リモート・インタフェースに定義するメソッドは以下のルールに従って記述する必要がある．

a) public である．
b) Remote インタフェースを継承する．

c) RemoteException 例外を送出する.

RemoteException を送出するのは，リモート・メソッドの呼び出しはローカル・メソッドの呼び出しとは異なり，ネットワークやサーバの障害に起因する例外が送出される可能性があるからである.

本例では，クライアント・サーバ間でのデータの受け渡しの方法として，計算結果の一覧を Vector オブジェクトに格納して通知するようにしている. インタフェース定義中のメソッドの返り値が Vector になっているが, これはオブジェクトのコンテナとして機能している.

また，計算結果を格納するためのクラスとして，ResultSet クラスも定義する. これにより，Calculate インタフェース中のメソッドを呼び出すだけで，複数の ResultSet オブジェクトを格納した Vector オブジェクトを通知することが可能となる．JavaRMI を使って渡すオブジェクトには，オブジェクト・シリアライゼーションで必要となる java.io.serializable インタフェースを実装しなければならないので注意する.

2) リモート・インタフェースの実装

次に，リモート・インタフェースを実装した CalculateImpl クラスと，それをインスタンス化し，RMI セキュリティ・マネジャを使って RMI レジストリに登録する MortgageCalcServer クラスを作成する.

リモート・インタフェースに定義するメソッドは以下のルールに従って記述する必要がある.

a) 通常，UnicastRemoteObject クラスを継承する.
b) UnicastRemoteObject のコンストラクタが RemoteException 例外を送出する可能性があるため，デフォルトコンストラクタを明示する必要がある.

CalculateImpl クラスでは，CalcPI メソッド，CalcPITI メソッドを実装している．このクラスのコンストラクタ中に Naming.rebind メソッドの呼び出しを行っているが，これは java.rmi.Naming で定義されているクラスであり，RMI レジストリにオブジェクトを登録する際に利用する. 表-3.1 に java.rmi.Naming クラスの主なメソッドを記述する.

MortgageCalcServer クラス内の RMI セキュリティ・マネジャオブジェクトは, クラスが適切にロードされているかを自動的に監視するものである. RMI セキュリティ・マネジャは，JavaRMI でオブジェクトを登録する際に必須である.

表-3.1 java.rmi.Naming クラス

メソッド名	機能
bind(String name, Remote obj)	objで指定されたリモート・オブジェクトに nameで指定された名前を付けて登録する.
rebind(String name, Remote obj)	bindメソッドと同様であるが指定された名前の リモート・オブジェクトがすでに 登録されていれば削除してから登録する.
unbind(String name)	Nameで指定された名前の登録を削除する.
lookup(String name)	Nameで指定された名前に対応する リモート・オブジェクトの参照を返す.
list()	レジストリに登録されている全ての登録を削除する.

3) Java ソース・ファイルのコンパイル

図-2.13 **に記した** javac **コマンドを用いて，**Java **ソース・ファイルをコンパイルする.**

```
>javac *.java
```

（注）カレント・ディレクトリに全ての Java **ソースが入っている場合.**

4) **代理オブジェクトの作成**

図-2.13 **に記した** rmic **コマンドを用いて，クライアント・スタブ，サーバ・スケルトンのクラスファイルを作成する.**

```
>rmic CalculateImpl
```

5) **オブジェクトの登録**

まず，RMI **レジストリを起動する.**

```
>rmiregistry
```

次に，リモート・インタフェース実装オブジェクトの登録を行う．前述した通り，オブジェクトの登録は MortgageCalcServer **を起動すればよい.**

```
>java MortgageCalcServer
```

これで，サーバ側は起動され，クライアントからの接続待ち状態に入る．

6) クライアントの作成

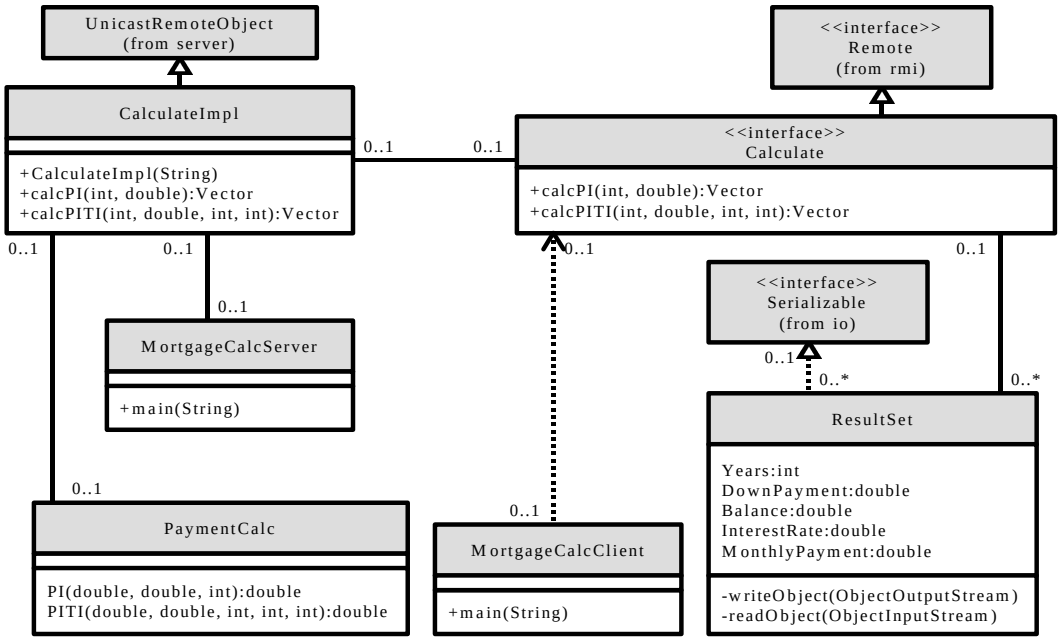
まず，サーバの場合と同様に RMI セキュリティ・マネジャをインストールし，利用したいリモート・オブジェクトを設定する．これで，プログラム内のローカル・オブジェクトと同じ手法でリモート・オブジェクトを利用できる．後は，クライアント・アプリケーションを記述し，コンパイル，起動を行えばよい．クライアントでは，コマンドラインからサーバ名等を入力し，ディスプレイ上に計算結果の表示をする．

オブジェクトのリファレンスを取得する際には，次の例外が送出されることがある．

a) NoBoundException
   指定されたリモート・オブジェクトが，レジストリに登録されていない．
b) UnknownHostException
   指定されたサーバホストが見つからない．
c) RemoteException
   レジストリが見つからない．
d) MalformedURLException
   サーバホスト指定の URL が正しくない．

(2) クラス図とシーケンス図

図-3.1 にクラス図，図-3.2 にシーケンス図を示す．



（注）上図では，左側にサーバ，右側にクライアントが記されている．

図-3.1 MortgageCalc アプリケーションのクラス図（JavaRMI）

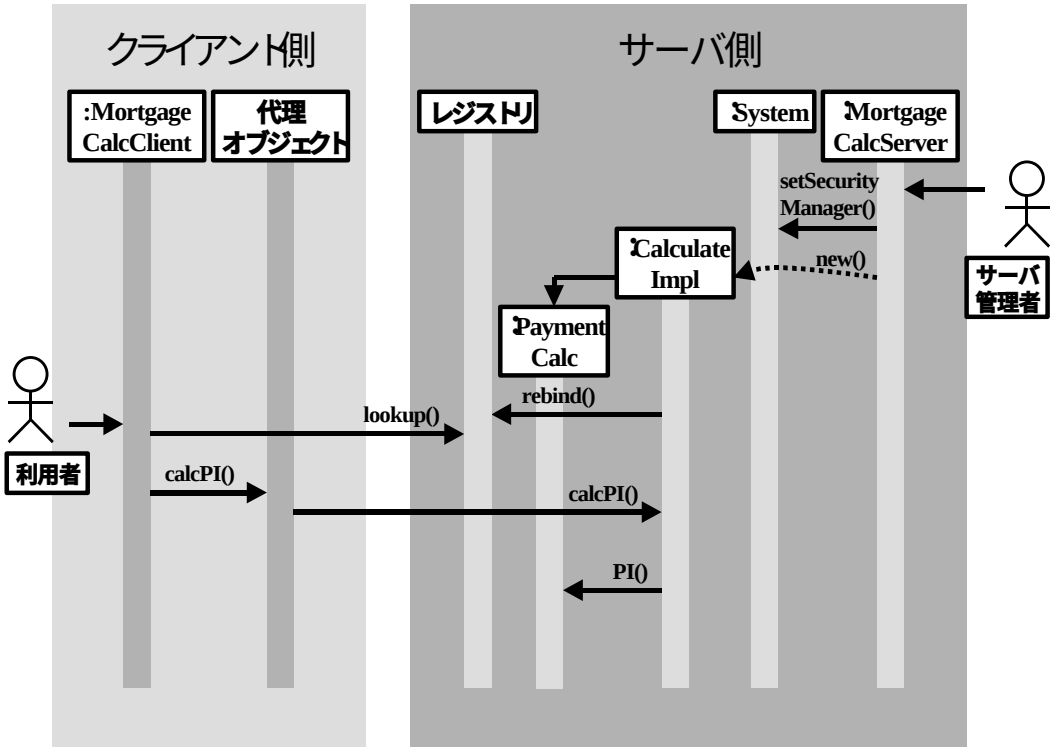


図-3.2 MortgageCalc アプリケーションのシーケンス図（JavaRMI）

3.1.3 JavaIDL における開発

(1) 開発手順

1) リモート・インタフェースの定義

まず，JavaIDL を用いて各オブジェクトを定義する．ソースを見ればわかるように，ResultSet の構造体，データ型の別名の定義（型宣言），インタフェースが記述されている．

ここで注意したいのが，JavaRMI では計算結果を格納するためのクラスを定義し，そのオブジェクトを Java.lang.Vector オブジェクトに格納しているのに対し，JavaIDL では複数のオブジェクトをシーケンス型で送っていることである．Java と IDL 間のマッピング規則を調べてみると，配列型，シーケンス型は共に Java の配列型にマッピングされているので，どちらを使っても問題はない．

ちなみに，配列型とは任意データ型の任意次元の固定長の並びを表す型，シーケンス型とは必要に応じて要素数を可変できる 1 次配列型のことである．

この IDL ファイルを idltojava コンパイラでコンパイルすると，合計 10 個の Java ソース・ファイルが作成される．その内，Calculate インタフェースの定義から生成されるファイルについて説明する．

```
>idltojava Calculate.idl
```

a) Calculate.java

インタフェースのソース．

b) CalculateImplBase.java

スケルトンクラスのソース．

c) _CalculateStub.java

スタブクラスのソース．

d) CalculateHolder

ホルダのソース．インタフェースに記述したオペレーション宣言の中で，各引数の受け渡し方法の宣言が，出力（out），入出力（inout）である場合に必要となるクラス．

e) CalculateHelper

ヘルパのソース．インタフェースやユーザ定義型を利用するときに必要となる補助的なメソッドが格納されているが，ほとんどはスタブ，スケルトン・コードから使用されるもので，ユーザが頻繁に使用するメソッドは，後に説明する narrow メソッドや，any 型から値を取り出す insert メソッドなどが

ある．

2) サーバ・オブジェクトの実装

JavaIDL のドキュメントは，サーバ・アプリケーションを「サーバント」と「サーバ」の 2 つに分けて説明している．この表現手法に従えば，サーバ・オブジェクトに実装部分である CalculateImpl クラスが「サーバント」であり，ORB オブジェクトの初期化や，サーバントの登録，クライアントからの要求処理を行うのが，MortgageCalcServer クラスとなる．

3) サーバ・アプリケーションの作成

JavaRMI では，RMI レジストリを使用してリモート・オブジェクトの参照を取得できたが，JavaIDL では，ネーミング・サービスというシステムを利用してリモート・オブジェクトの参照を取得する．

まず，ORB の初期化を行ったあと，リモート・オブジェクトのインスタンスを作成し，ORB 上に登録（ORB に接続）する．そして，ルート・ネーミング・コンテキスト（ファイル・システムのルートディレクトリに相当する）を取得し，次に NameComponent オブジェクトをルート・ネーミング・コンテキストに追加する．

これで，ネーミング・コンテキストは登録すべきネーミング・コンテキストへのパスを取得したことになるので，Calculator オブジェクトを rebind メソッドで登録する．登録が完了すると，サーバ・アプリケーションはクライアントからの要求を処理できる状態に入る．サーバをクライアントの要求を処理できる状態にすることをサーバの活性化という．逆に，サーバを元の状態に戻すことを非活性化という．

4) クライアント・アプリケーションの作成

まず，ORB を初期化する．そして，ネーミング・コンテキストのオブジェクトを作成し，それを用いて，リモート・オブジェクトの参照を取得する．後の処理は JavaRMI と同様である．

5) アプリケーションの実行

実行する前に tnameserv コマンドを用いて JavaIDL ネーミング・サービスを起動しておく．

```
>tnameserv
```

あとは，サーバ・アプリケーション，クライアント・アプリケーションの順に起動すれば，MortgageCalc アプリケーションを実行できる．

(2) 使用したパッケージ一覧

ここでは，使用したパッケージについて説明する．

a) org.omg.CORBA パッケージ

CORBA モジュールで定義されているインタフェースは，Java でマッピングするとこのパッケージのクラスやインタフェースにマッピングされる．

b) org.omg.CosNaming パッケージ

ネーミング・サービスの CosNaming モジュールを Java にマッピングしたとき，このパッケージのクラスやインタフェースにマッピングされる．

(3) クラス図とシーケンス図

図-3.3 にクラス図，図-3.4，図-3.5 にシーケンス図を示す．

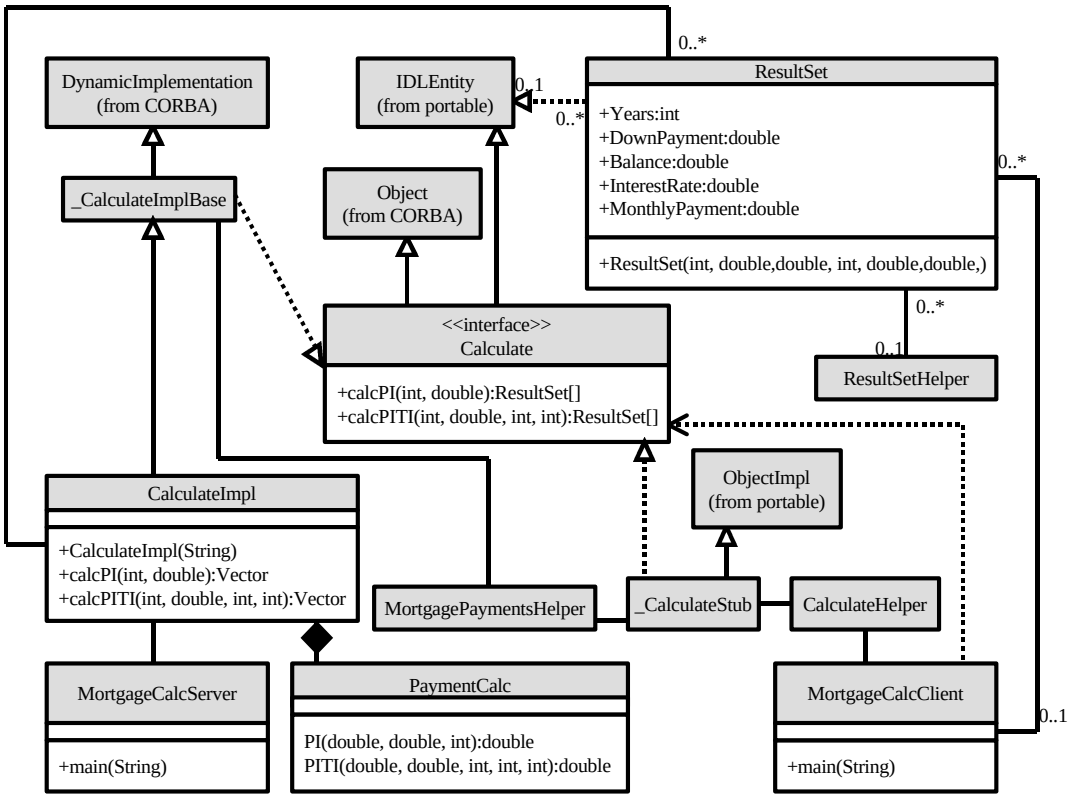


図-3.3 MortgageCalc アプリケーションのクラス図（JavaIDL）

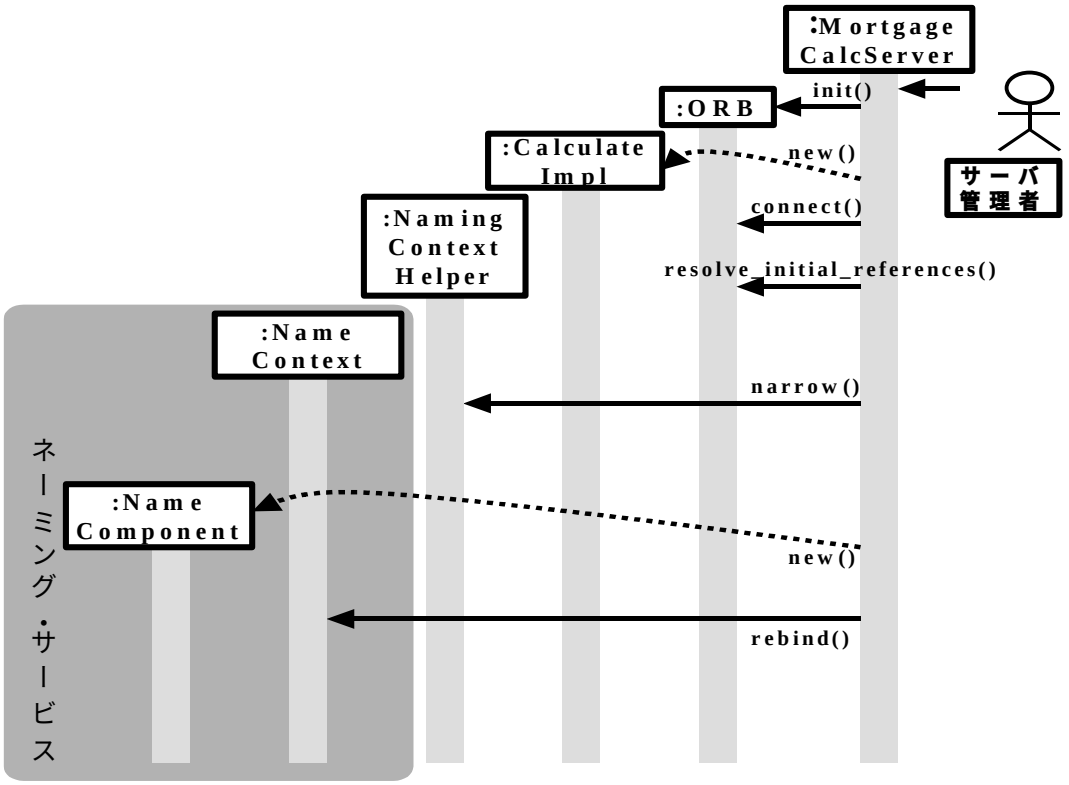


図-3.4　MortgageCalc アプリケーションのサーバ側シーケンス図(JavaIDL)

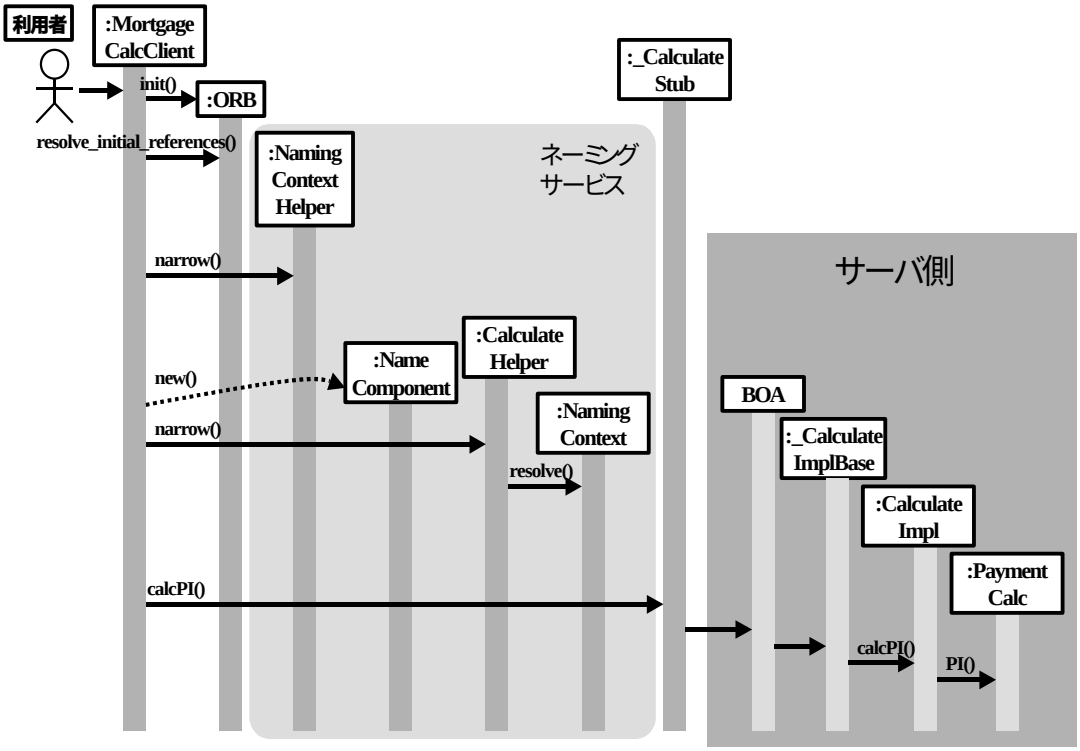


図-3.5　MortgageCalc アプリケーションのクライアント側シーケンス図(JavaIDL)

3.2 データ転送時間測定アプリケーション

ここでは，JavaRMI と JavaIDL のそれぞれの環境におけるリモートマシン間での速度を測定し，比較することを目的としたアプリケーションを作成した.

本プログラムは，クライアント側が指定したデータファイル名を String 型でサーバへ渡し，サーバはその指定されたファイルから 1 行ずつ読み込んで String 型配列に格納し，格納したデータをクライアントへ戻り値として返すプログラムである.

本プログラムにおいて，サーバのメソッドを呼び出してからサーバがデータを返すまでの時間を測定できるように作成した.

3.2.1 JavaRMI による開発

本プログラムの JavaRMI での主な開発部分は次の通りである.

a)RemoteException 例外を送出できるようにサーバ側のインタフェース実装クラスに UnicastRemoteObject クラスを継承
b)RMISecurityManager クラスを用いてセキュリティマネージャを設定
c)Naming クラスを用いてレジストリにオブジェクトを登録，参照
d)Remote インタフェースを継承しているインタフェースを Java で作成し，サーバに実装

これは基本的な JavaRMI の開発方法と同じである.
図-3.6 に JavaRMI を用いて作成したプログラムのクラス図を示す.

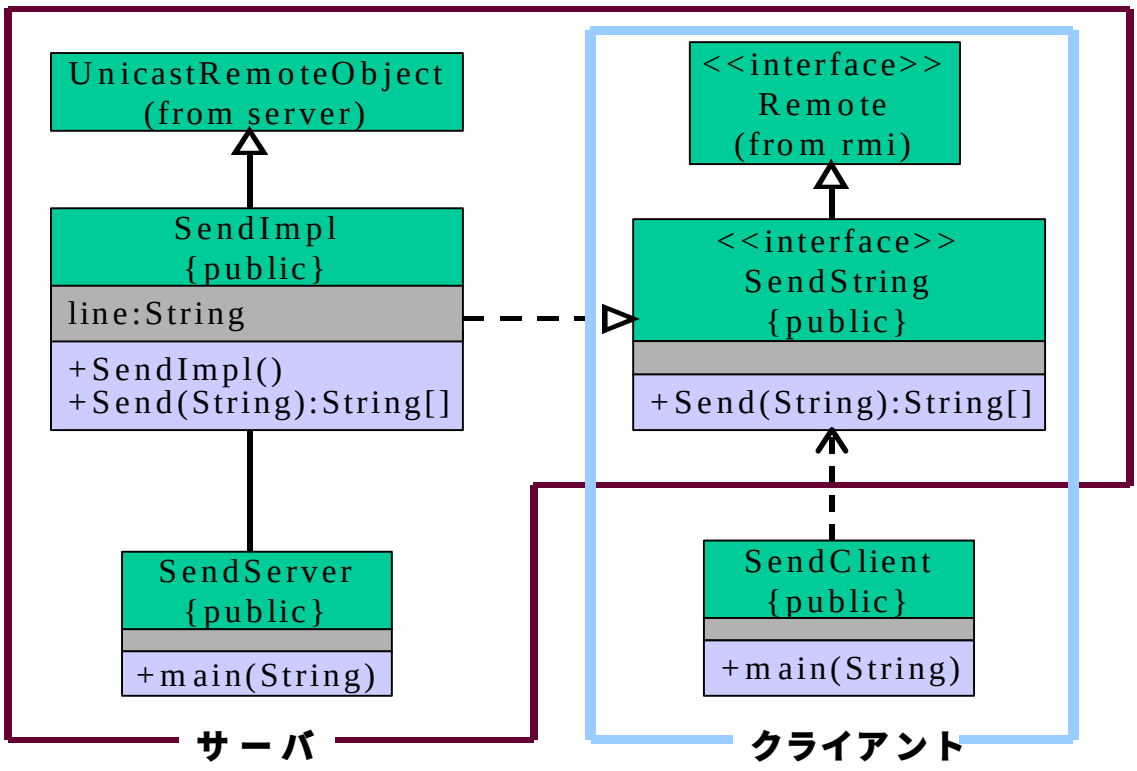


図-3.6　JavaRMI データ転送時間測定アプリケーションのクラス図

3.2.2　JavaIDL による開発

JavaIDL での主な開発部分は次の通りである.

(1)　サーバ側

a) ORB の初期化
b) サーバのオブジェクトを ORB に登録
c) ネーミングサービスの呼び出し
d) ネーミングサービスに登録
e) クライアントからの要求を待つ

(2)　クライアント側

a) ORB の初期化
b) ネーミングサービスの呼び出し
c) サーバオブジェクトの参照

(3)　インタフェース

IDL を用いて作成．配列は，シーケンス型を用いた

図-3.7 に JavaIDL を用いて作成したプログラムのクラス図を示す．なお，ホルダクラスはインタフェースのパラメタに out や inout を用いてないため，使用しない．

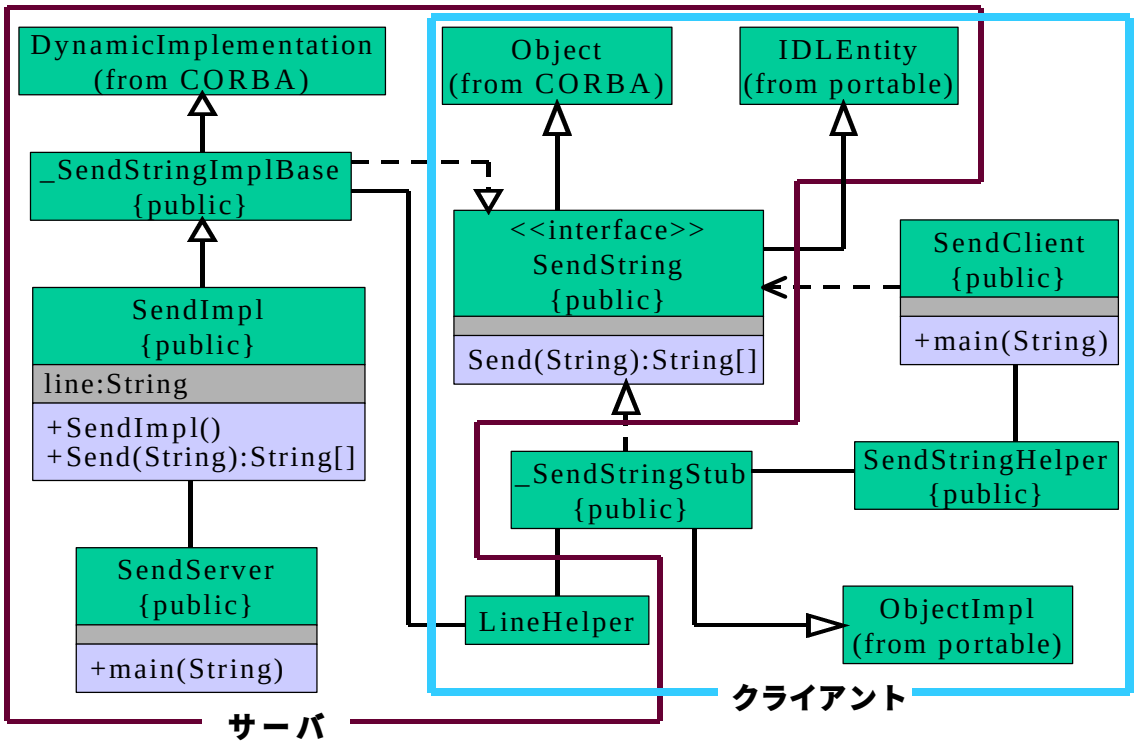


図-3.7　JavaIDL データ転送時間測定アプリケーションのクラス図

第4章　プログラムの評価

4.1　MortgageCalc アプリケーション

4.1.1　評価項目

3.1 で作成した JavaRMI のアプリケーションと JavaIDL のアプリケーションを, ソース，クラス図，シーケンス図から比較し，それぞれの特徴を見ていく．

4.1.2　評価結果

(1)　ソース・プログラムについて

まず Java ソース・コード，クラス・ファイルについて，プログラム行数，ファイル数について記述する．

表-4.1 は，それぞれの値を記述したものである．

表-4.1　ソース・コードのサイズ

	JavaRMI			JavaIDL		
	ソース ファイル	クラス ファイル	ソース 行数	ソース ファイル	クラス ファイル	ソース 行数
a)サーバ本体の記述	3	3	216	4	4	262
b)クライアント本体の記述	2	2	77	2	2	77
c)通信部の記述	1	1	14	9	9	431
d)合計	6	8	307	14	14	744

（注 1）a)，b)はそれぞれに必要な Java ソース・ファイルを指す．c)は，通信に必要なクラスやインタフェースを指す．d)は，サーバ，クライアントで重複する場合は 1 個のファイルとして合計を出したものである．

表からもわかるように，サーバ本体の記述，クライアント本体の記述では JavaRMI，JavaIDL 間に大きな差は見られない．しかし，通信部の記述では，JavaIDL の方が多くのソース・コードの記述を必要とする．

これは，JavaRMI が簡単に分散オブジェクト環境を実現できるのに対し，JavaIDL では，Java 言語以外に IDL を用い様々な設定が可能なため，ソースのサイズが大きくなると考えられる．しかし，JavaRMI は Java 言語以外サポートしておらず，他の言語（C++言語，ＣＯＢＯＬ言語等）を用いた分散オブジェクト環境の実現では CORBA を用いなければならないので，ユーザは，用途に応じてどちらを使うか考えなければならない．

(2) クラス図について

図-3.1，図-3.3 のクラス図を比べると，以下のような相違点が挙げられる．

a) JavaIDL で使われるクラスの数が多い．これについては（1）で述べた．

b) JavaRMI では，rmi コンパイラを用いてリモート・インタフェース実装クラスから代理オブジェクトを生成するが，JavaIDL では，idltojava コンパイラを用いてインタフェース定義からスタブ・スケルトンを生成する．これによって，JavaIDL の記述から，通信関係の全てのクラスを生成できるので，全体の構造が分かりやすくなる．

(3) シーケンス図について

図-3.2，図-3.4，図-3.5 のシーケンス図を比べると，通信部分において以下のような相違点がわかる．

a) リモート・オブジェクトのを登録する前に，JavaRMI では new()メソッドでインスタンスを生成するだけでよいのに対して，JavaIDL では init()メソッドで ORB を初期化し，connect()メソッドで，ORB とリモート・オブジェクトを関連付ける必要がある．

b) リモート・オブジェクトのリファレンスの取得方法が異なる．JavaRMI では RMI レジストリを用い，JavaIDL ではネーミング・サービスを用いる．
RMI レジストリとネーミング・サービスの相違点として，RMI レジストリが，一種のネーミング・サーバであるのに対して，ネーミング・サービスは 1 つのシステムとして働くことが挙げられる．ネーミング・サービスの登録したオブジェクトの名前は，ネーミング・コンテキストというファイルシステムでいうディレクトリに相当するものでツリー構造であるのに対し，レジストリは一定の構造がない．

4.2 データ転送時間測定アプリケーション

3.2 で作成したデータ転送時間測定アプリケーションを用いて，JavaRMI，JavaIDL それぞれの環境のデータ転送時間を測定した．

データ転送時間とは，データ指定のリクエストを出し，サーバからクライアントにデータを受け渡すまでの時間とした．データは String 型のデータを 1 バイトから 2M バイトまで変化させたときの平均転送時間，転送速度を調べた．Java の String 型に格納できるデータが 64k バイトが限界のため，64k バイト以上のデータは 64k バイトごとに配列に格納した．

4.2.1 JavaRMI の転送時間

図-4.1 に JavaRMI によるデータ転送のシーケンスを示す．

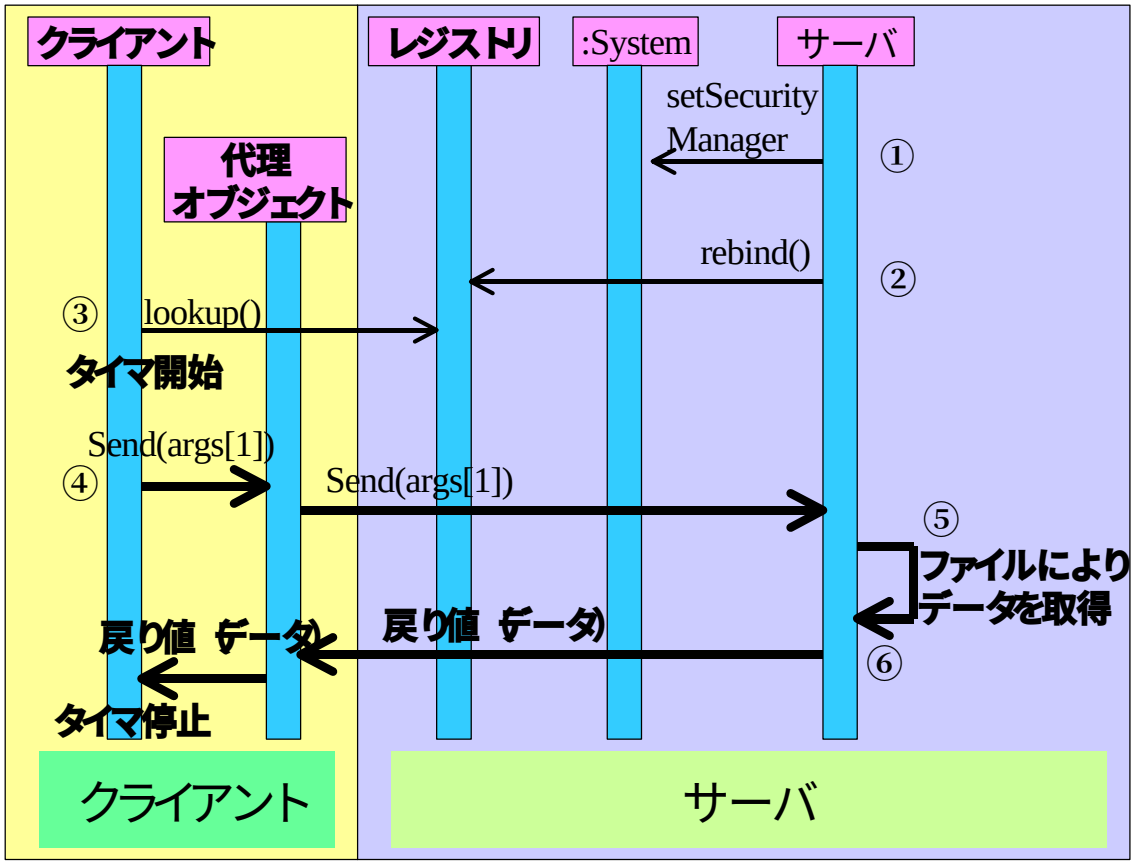


図-4.1 JavaRMI によるデータ転送シーケンス図

シーケンス図における各処理内容の詳細を以下に示す．

① セキュリティマネージャの設定

② リモートオブジェクトの登録
③ リモートオブジェクトの検索
④ リモートメソッドの呼び出し
⑤ 指定されたファイルからデータを読み込み，String 型配列のデータに格納する
⑥ String 型のデータをクライアントへ返す

4.2.2 JavaIDL の転送時間

図-4.2 に JavaIDL によるデータ転送のシーケンスを示す.

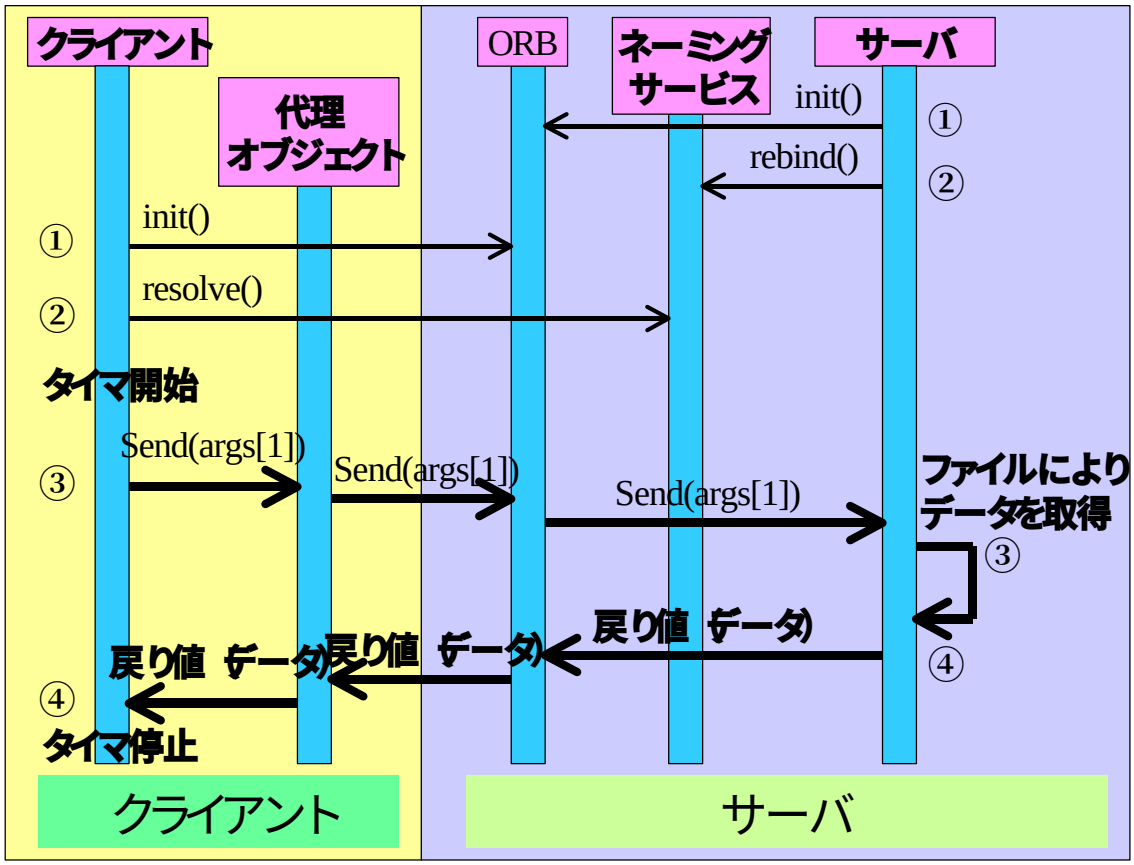


図-4.2 JavaIDL によるデータ転送シーケンス図

シーケンス図による各処理内容の詳細を以下に示す.

(1) サーバ側

① ORB の初期化
② リモートオブジェクトの登録
③ 指定されたデータファイルから読み込み，String 型配列に格納
④ 格納したデータをクライアントへ戻り値として返す

(2) クライアント側

① ORB の初期化
② リモートオブジェクトの検索
③ リモートメソッドの呼び出し
④ データを受け取る

4.2.3 実行環境

計測環境は図-4.3 のとおりである.

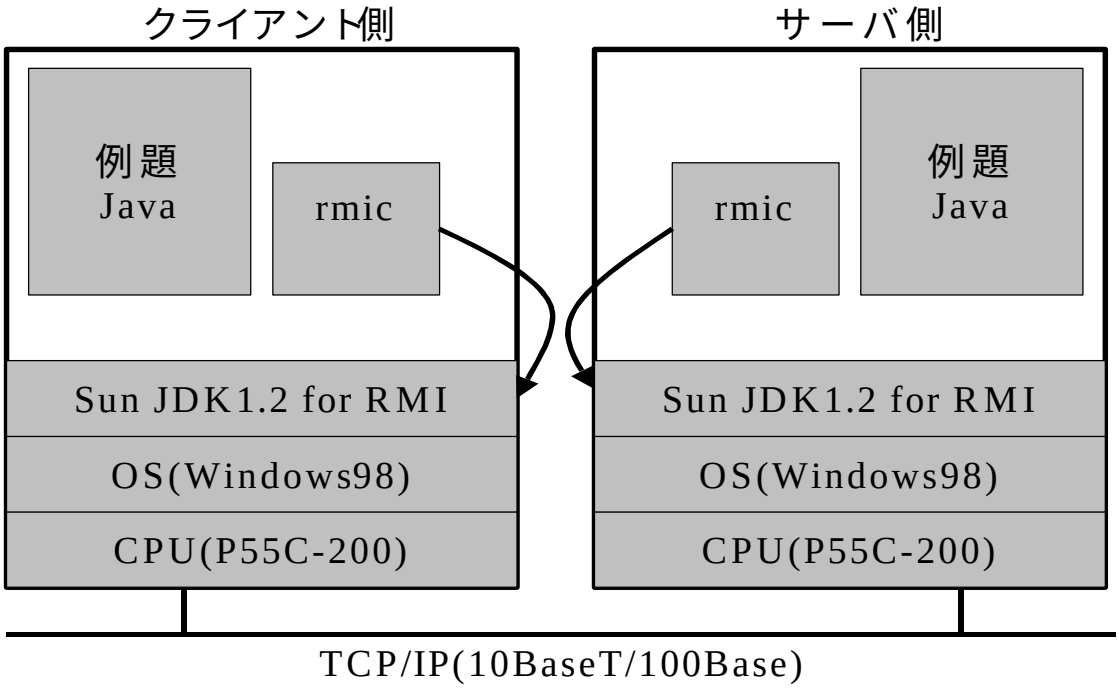


図-4.3 性能評価の計測環境のシステム図

4.2.4 測定方法

(1) アーキテクチャ

JavaRMI と JavaIDL においてクライアントがサーバのメソッドを呼び出す前とサーバからデータを受け取ったときにタイマを設定した.

(2) 通信速度

LAN が 10Mbps と 100Mbps において測定した.

(3) データ長

サーバから受け取るデータを 1byte から 1Mbyte まで増加させて測定した.

(4) データの取り方

データは 1 つの項目につき 10 回計測して平均を求めた.

4.2.5 評価結果

(1) データ転送時間

JavaRMI 及び JavaIDL の環境において，データ量を 1byte から 2Mbyte まで変えたときのデータ転送時間の比較を行なった．通信速度は 10Mbps と 100Mbps の LAN を用いた．

1) 計測結果

a) 10Mbps でのデータ転送速度

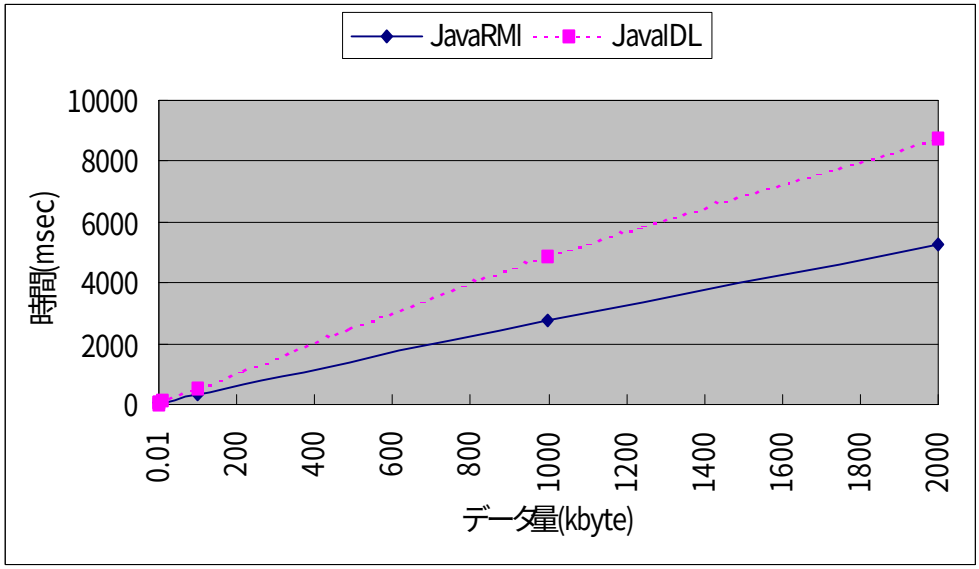


図-4.4 10Mbps でのデータ転送時間

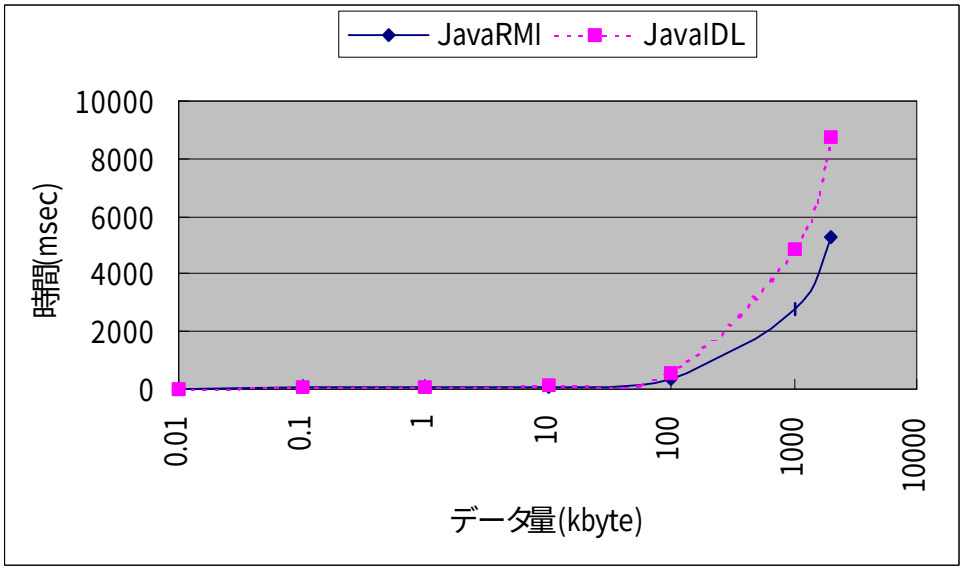


図-4.5 10Mbps でのデータ転送時間の対数グラフ

データ量が 1byte から 10kbyte の間は両者とも時間にさほど変化は見られなかったが，

10kbyte からは両者ともほぼ一直線に急激に上昇しており，両者に差がみられた．

b) 100Mbps でのデータ転送時間

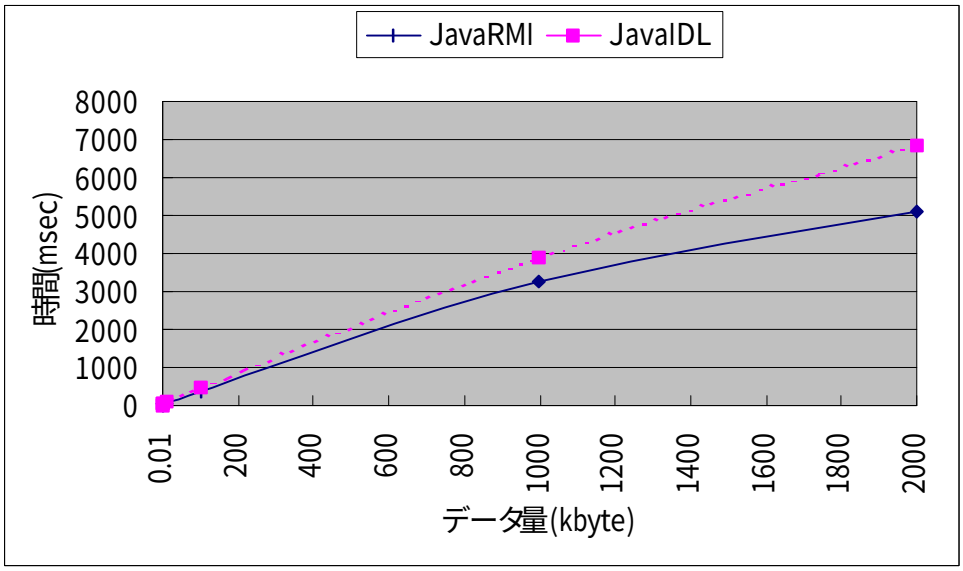


図-4.6　100Mbps でのデータ転送時間

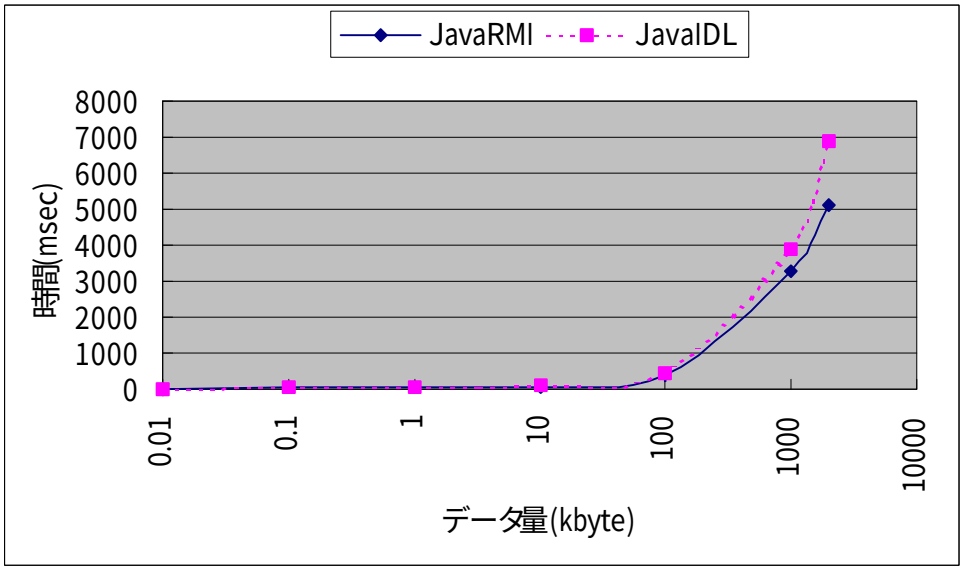


図-4.7　100Mbps でのデータ転送時間の対数グラフ

100Mbps においても，10Mbps と変わらず，データ量が 10kbyte までは時間にさほど変化はみられず，10kbyte からは急激に上昇した．両者に差がみられたが 10MbpsLAN よりも差はみられなかった．

c) 10Mbps と 100Mbps によるデータ転送時間

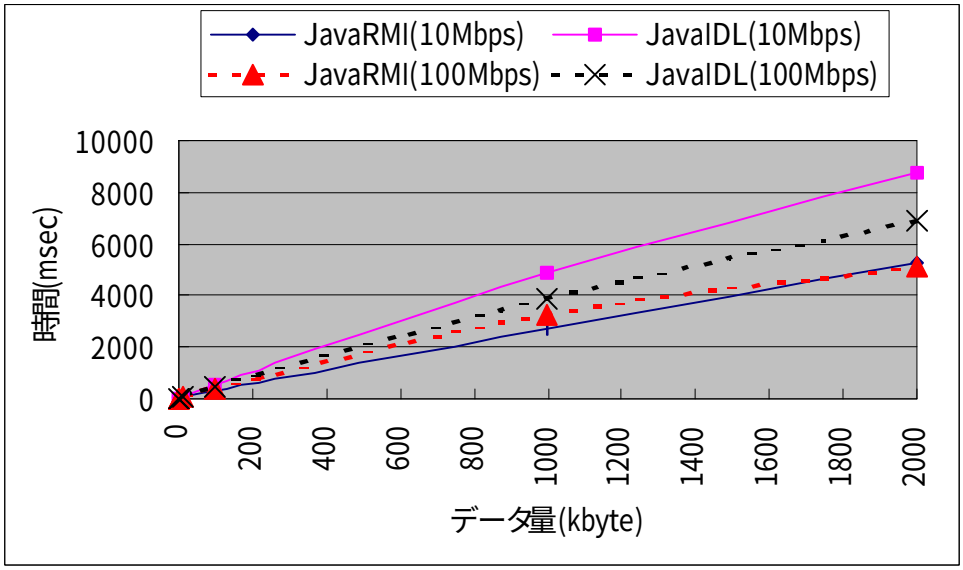


図-4.8　10Mbps と 100Mbps によるデータ転送時間

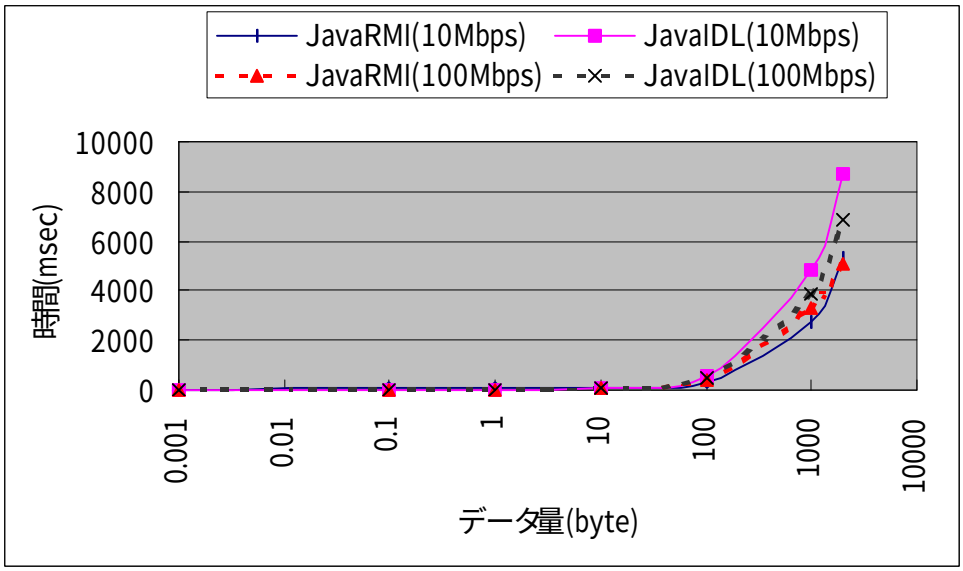


図-4.9　10Mbps と 100Mbps によるデータ転送時間の対数グラフ

全体的にみると 10kbyte まではどれも時間にさほど変化が無い．しかし，10kbyte からは全体的に急激的に上昇した．JavaRMI は LAN の通信速度には影響がみられなかったが，JavaIDL には有意差がみられた．

(2)　データ転送速度

データ転送時間におけるデータ転送速度を 10Mbps，100Mbps のそれぞれにおいて JavaRMI と JavaIDL の比較をした.

1) 計測結果

a) 10Mbps でのデータ転送速度

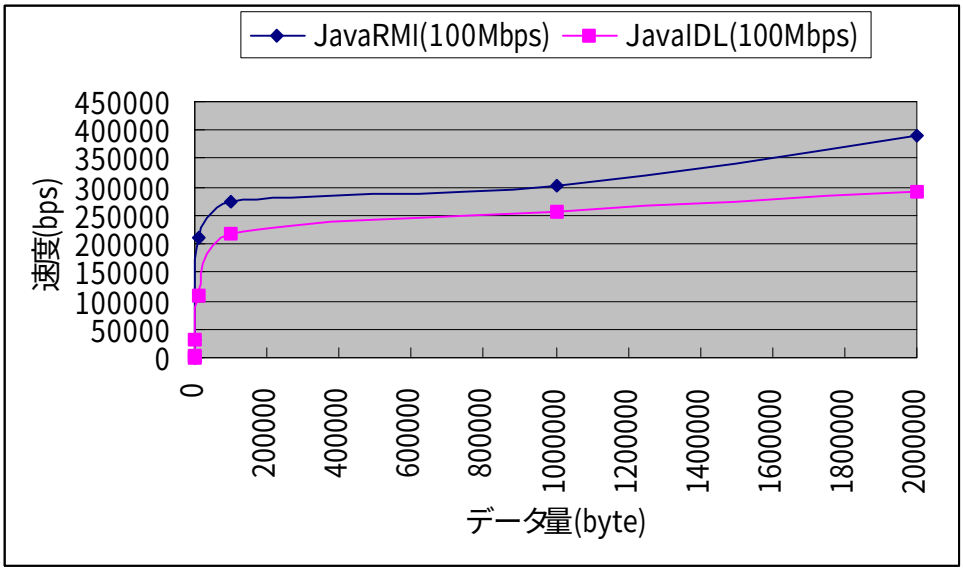


図-4.10　10Mbps でのデータ転送速度

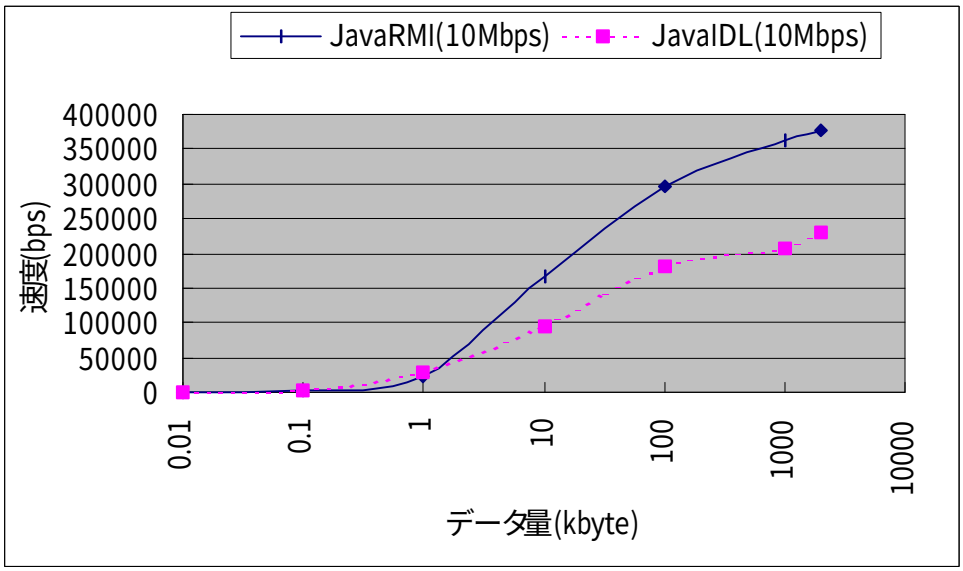


図-4.11　10Mbps でのデータ転送速度の対数グラフ

両者とも 100byte までは速度に大きな差はみられなかったが，1kbyte 以降は速度の大幅

な上昇と JavaRMI，JavaIDL の速度に差がみられた．

b) 100Mbps でのデータ転送速度

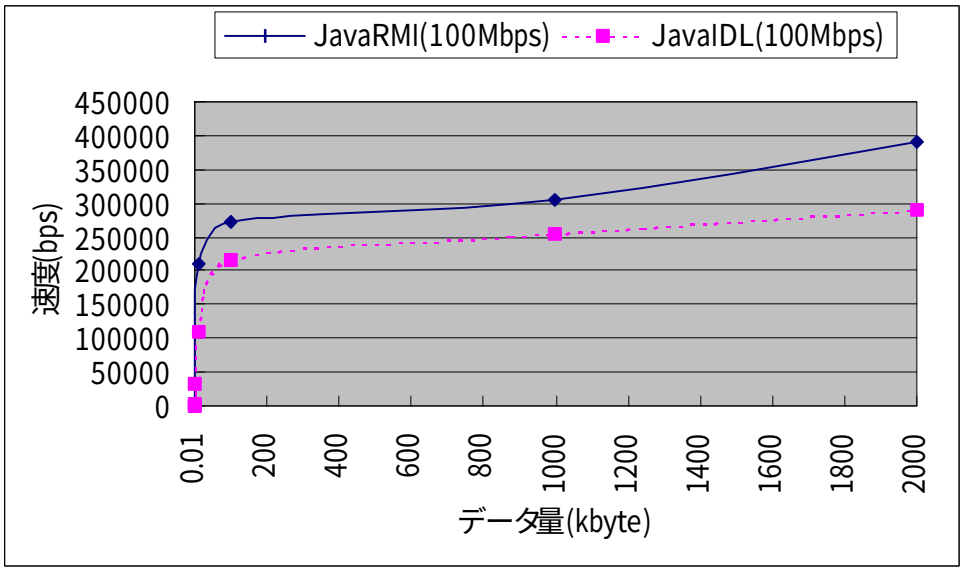


図-4.12　100Mbps でのデータ転送速度

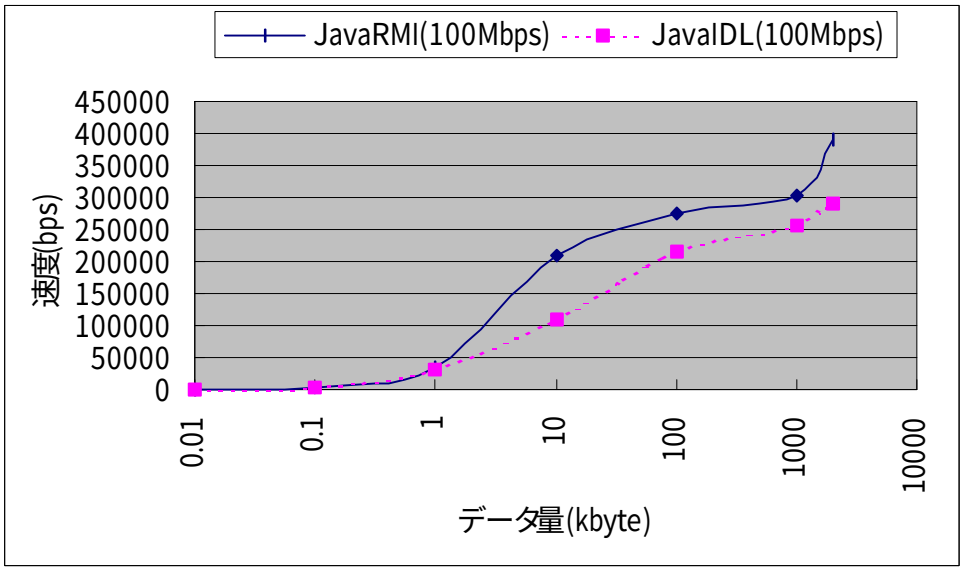


図-4.13　100Mbps でのデータ転送速度

100Mbps においても 10Mbps と同様の結果を得た．しかし 10Mbps より両者の差はみられなかった．

c) 10Mbps と 100Mbps によるデータ転送速度

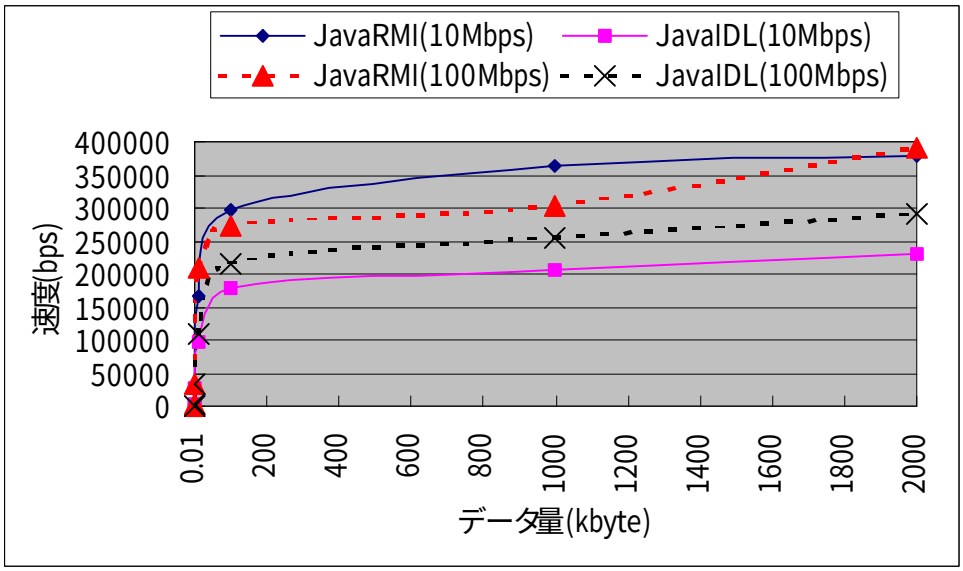


図-4.12 10Mbps と 100Mbps によるデータ転送速度

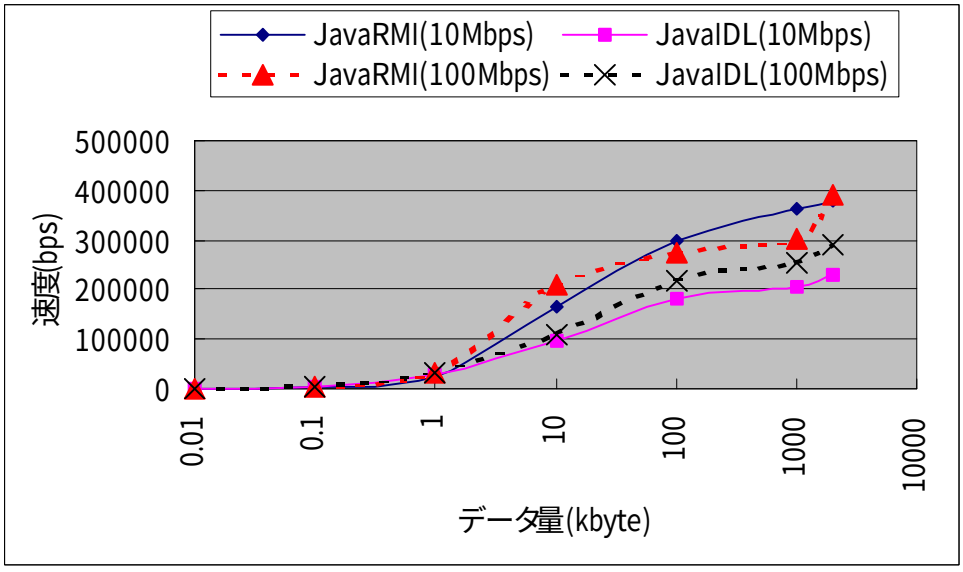


図-4.13 10Mbps と 100Mbps によるデータ転送速度の対数グラフ

JavaRMI，JavaIDL のどちらの環境とも 100byte までは LAN の性能に関係無く，変化はみられなかった．1kbyte 以降は急激に速度が上昇したが両者とも 100kbyte から傾斜が緩やかになった．また，JavaRMI には LAN の影響はほとんどみられなかった．

第5章　考察

5.1　MortgageCalc アプリケーションについて

ここでは，本稿では説明できなかった比較について記述する.

5.1.1　JavaRMI と JavaIDL（CORBA）の比較

JavaRMI と JavaIDL の相違点を図-5.1 に示す.

表-5.1　JavaRMI と JavaIDL（CORBA）との比較

機能	JavaRMI	JavaIDL（CORBA）
値による引数渡し	○	×
動的バインディング	×	○
動的起動	×	○
URLネーミング	○	×（ORB依存）
ファイアウォール・プロキシ	○	×（ORB依存）
言語非依存性	×	○
言語非依存通信プロトコル	×（将来はIIOP経由で？）	○（IIOP経由）
永続オブジェクトのネーミング	×	○
通信レベルでのセキュリティ	○（SSL経由）	○（セキュリティ・サービス経由）
通信レベルでのトランザクション	○（OTS経由）	○（JTS経由）

実験の結果と表-5.1 から，JavaIDL は，インタフェース定義言語 IDL の習得，ソース・コードの複雑化等，分散システム開発の初心者には難解であるが，動的起動やファイアウォールの実装が可能である事など，高度なシステム開発ができ，JavaRMI は，Java 言語のみをサポートし，動的起動などの高度な実装はできないが，分散システム開発者にも，Java 言語をある程度習得できたものであれば，非常に簡単に分散システムを構築できる.

5.2 データ転送時間測定アプリケーション

本稿で計測したデータ転送時間はクライアントからデータ指定のリクエストを出し，サーバからクライアントにデータを受け渡すまでの時間とした.

5.2.1 データ転送時間

図-4.7，図-4.8 に JavaRMI 及び JavaIDL において LAN が 10Mbps と 100Mbps によるデータ転送時間を示す.

Java RMI, JavaIDL ともに 10kbyte まではほとんど変化はみられなかったが, 100kbyte 以降では差がみられた．この理由としては本研究では 64kbyte より上のデータは 64kbyte づつを区切って配列に格納し，送っているためだと考えられる．データ転送速度としては JavaRMI のほうが速く，LAN の影響もみられなかった．しかし，JavaIDL では，LAN による影響がみられた. これは, JavaIDL はリモートメソッドを呼び出すときに BOA や ORB の機能を介して処理を行なうので，処理が遅くなっていると考えられる.

5.2.2 データ転送速度

図-4.12，図-4.13 に JavaRMI 及び JavaIDL において 10Mbps と 100Mbps によるデータ転送速度を示す.

データ転送速度は両者とも 1kbyte までは LAN の通信速度や，環境による速度の違いはみられなかった．1kbyte から急激に速度が上昇し，それぞれに差がみられるようになった．10kbyte からは両者とも速度がほぼ一定に近づいていった.

2Mbyte 時において，JavaRMI には 10Mbps と 100Mbps の差がほとんどみられず, JavaIDL より速かった．JavaIDL は 10Mbps と 100Mbps において差がみられた．これはデータ量を増やすことによって，機能の差が明確に現れてくるため，リモートメソッド起動時に処理が遅い JavaIDL により，LAN の通信速度による転送速度の違いがみられた.

第6章　まとめ

今回の研究では，JavaRMI と JavaIDL においてのサポートしている機能と速度的な面を評価の対象とした．

サポートしている機能として最大の違いは，JavaRMI は Java 言語のみに依存しているためにインタフェースにも Java 言語で記述することができ，Java 言語による開発経験者なら容易に作成することができる．それに対して，JavaIDL は C 言語や COBOL 言語などの多数の言語をサポートしているが，IDL を記述しなければならないため，容易に作成することができない．

他の機能では，RMI にはオブジェクト送受信の容易性，レジストリに登録，参照するのに簡単な記述で済む，少しのクラスファイルだけで分散オブジェクト環境が実現できる，などの長所がある．IDL においてはネーミング・サービス，サーバの動的起動，そして永続オブジェクトがサポートされているという長所がある．

速度的な面では，今回測定した範囲ではほぼ全てにおいて RMI が優れていた．しかし，多種の条件において測定することにより，どちらがどこで何が優れているかを調べる必要がある．

2 つの環境を比較，評価した結果，JavaRMI は作成が容易で速度が速いといった長所が明らかになった．IDL のインタフェース以外は Java 言語だけを用いたため，総合的には RMI のほうが優れていると判断できる．JavaIDL の Java における今後のサポートを期待したい．

今後の課題としては，他の分散オブジェクト環境との比較とともに，Java ベースの分散システムである Jini を理解，実現し，多方面からの評価及び検討したい．

参考文献

[1] R. Orfali and D. Harkey, *Client/ Server Programming with Java and CORBA*, John Wiley & Sons, 1997［並河英二ほか(訳), Java & CORBA C/S プログラミング, 日経 BP 社, 1997］.

[2] 小野沢博之著, CORBA 完全解説　ソフト・リサーチ・センター, 1999.

[3] G. Lewis and S. Barber and E. Siegel, *Programming with Java IDL*, John Wiley & Sons, 1998［日本ユニシス分散オブジェクト研究会(訳), Java IDL プログラミング, カットシステム, 1998］.

[4] 青山幹雄ほか, コンポーネントウェア, 共立出版, 1998.

[5] T. B. Downing, *Java Remote Method Invocation*, IDG Books Worldwide, 1998［夏目大(訳), Java RMI パーフェクトガイド, コンピュータ･エージ社, 1998］.

[6] 鈴木章ほか, Java RMI 分散ネットワークプログラミング, サイエンス社, 1998.

[7] Joseph O'neil, *Teach Yourself Java*, [トップスタジオ(訳), 独習 Java, 翔泳社, 1999].

[8] J. Farley, *Java Distributed Computing*, O'Reilly & Associates, 1998［豊福剛(訳), Java 分散コンピューティング, オライリー･ジャパン, 1998］.

[9] MISCO オブジェクト指向研究会, オブジェクトモデリング表記法ガイド, プレンティスホール出版, 1998.

[10] 吉田裕之ほか, UML によるオブジェクト指向開発実践ガイド, 技術評論社, 1999.

付録

3.1 のアプリケーションで用いたソースプログラム

(1) JavaIDL 版

1) MortgageCalcClient クラスの記述

```
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class MortgageCalcClient
{

	public static void main(String args[])
	{
		if (args.length != 3)
		{
			System.out.println("Usage : java MortgageCalcClient");
			System.out.println("Interest rate must be in demical form!");
			System.exit(0);
		}

		String ServerName = args[0];
		Integer HousePrice = new Integer(args[1]);
		Double InterestRate = new Double(args[2]);

		try
		{
			/* orb 作成とイニシャライズ */
			ORB orb = ORB.init(args, null);

			/* ルートネーミングコンテキストの取得 */
			org.omg.CORBA.Object objRef =
orb.resolve_initial_references("NameService");
			NamingContext ncRef = NamingContextHelper.narrow(objRef);

			/* ネーミングサービスからオブジェクトの参照を解決する */
			NameComponent nc = new NameComponent("MortgageCalc","");
			NameComponent path[] = {nc};
			Calculate calc = CalculateHelper.narrow(ncRef.resolve(path));

			ResultSet[] v = calc.calcPI( HousePrice.intValue(),
InterestRate.doubleValue() );

			for (int i = 0; i < 6; i++)
			{
				ResultSet s = v[i];
				System.out.println
				("Term of Loan =" + s.Years + "years Down Payment = $"
+ s.DownPayment);
				System.out.println
				("Interest Rate =" + s.InterestRate * 100 + "% "
					+ " Monthly Payment =$" + s.MonthlyPayment);
				System.out.println();
			}
		} catch (Exception e) {
			System.out.println("ERROR : " + e);
```

```
                    e.printStackTrace(System.out);
                }
        }
}
```

2) MortgageCalcServer クラス

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class MortgageCalcServer
{

        public static void main(String args[])
        {
                try
                {
                        /* orb の作成とイニシャライズ */
                        ORB orb = ORB.init(args, null);

                        /* 登録したいオブジェクトを orb 上に持っていく */
                        CalculateImpl calculator = new CalculateImpl();
                        orb.connect(calculator);

                        /* ルートネーミングコンテキストを得る */
                        org.omg.CORBA.Object objRef =
orb.resolve_initial_references("NameService");
                        NamingContext ncRef = NamingContextHelper.narrow(objRef);

                        /* ネーミングサービスにオブジェクトの参照を登録 */
                        NameComponent nc = new NameComponent("MortgageCalc","");
                        NameComponent path[] = {nc};
                        ncRef.rebind(path, calculator);

                        /* クライアントからの要求待ちに入る */
                        java.lang.Object sync = new java.lang.Object();
                        synchronized (sync)
                        {
                                sync.wait();
                        }
                } catch (Exception e) {
                        System.err.println("ERROR : " + e);
                        e.printStackTrace(System.out);
                }
        }

}
```

3) CalculateImpl クラス

```
class CalculateImpl extends _CalculateImplBase
{

        private PaymentCalc calc;

        public ResultSet[] calcPI( int HousePrice, double InterestRate )
        {

                ResultSet[] retObject = new ResultSet[6];

                /* 頭金５%，１５年払いの場合 */
                ResultSet set1 = new ResultSet();

                set1.Years = 15;
                set1.DownPayment = HousePrice * .05;
                set1.Balance = HousePrice * .95;
                set1.InterestRate = InterestRate;
                set1.MonthlyPayment = calc.PI(set1.Balance, set1.InterestRate, set1.Years);

                retObject[0] = set1;

                /* 頭金５%，３０年払いの場合 */
                ResultSet set2 = new ResultSet();

                set2.Years = 30;
                set2.DownPayment = HousePrice * .05;
                set2.Balance = HousePrice * .95;
                set2.InterestRate = InterestRate;
                set2.MonthlyPayment = calc.PI(set2.Balance, set2.InterestRate, set2.Years);

                retObject[1] = set2;

                /* 頭金１０%，１５年払いの場合 */
                ResultSet set3 = new ResultSet();

                set3.Years = 15;
                set3.DownPayment = HousePrice * .10;
                set3.Balance = HousePrice * .90;
                set3.InterestRate = InterestRate;
                set3.MonthlyPayment = calc.PI(set3.Balance, set3.InterestRate, set3.Years);

                retObject[2] = set3;

                /* 頭金１０%，３０年払いの場合 */
                ResultSet set4 = new ResultSet();

                set4.Years = 30;
                set4.DownPayment = HousePrice * .10;
                set4.Balance = HousePrice * .90;
                set4.InterestRate = InterestRate;
                set4.MonthlyPayment = calc.PI(set4.Balance, set4.InterestRate, set4.Years);

                retObject[3] = set4;

                /* 頭金２０%，１５年払いの場合 */
                ResultSet set5 = new ResultSet();

                set5.Years = 15;
```

```
                set5.DownPayment = HousePrice * .20;
                set5.Balance = HousePrice * .80;
                set5.InterestRate = InterestRate;
                set5.MonthlyPayment = calc.PI(set5.Balance, set5.InterestRate, set5.Years);

                retObject[4] = set5;

                /* 頭金２０%, ３０年払いの場合 */
                ResultSet set6 = new ResultSet();

                set6.Years = 30;
                set6.DownPayment = HousePrice * .20;
                set6.Balance = HousePrice * .80;
                set6.InterestRate = InterestRate;
                set6.MonthlyPayment = calc.PI(set6.Balance, set6.InterestRate, set6.Years);

                retObject[5] = set6;

                return retObject;
        }

        public ResultSet[] calcPITI ( int HousePrice, double InterestRate, int Insurance, int Taxes )
        {
                ResultSet[] retObject = new ResultSet[6];

                /* 頭金５%, １５年払いの場合 */
                ResultSet set1 = new ResultSet();

                set1.Years = 15;
                set1.DownPayment = HousePrice * .05;
                set1.Balance = HousePrice * .95;
                set1.InterestRate = InterestRate;
                set1.MonthlyPayment =
                        calc.PITI(set1.Balance,  set1.InterestRate,  set1.Years,  Insurance,
Taxes);

                retObject[0] = set1;

                /* 頭金５%, ３０年払いの場合 */
                ResultSet set2 = new ResultSet();

                set2.Years = 30;
                set2.DownPayment = HousePrice * .05;
                set2.Balance = HousePrice * .95;
                set2.InterestRate = InterestRate;
                set2.MonthlyPayment =
                        calc.PITI(set2.Balance,  set2.InterestRate,  set2.Years,  Insurance,
Taxes);

                retObject[1] = set2;

                /* 頭金１０%, １５年払いの場合 */
                ResultSet set3 = new ResultSet();

                set3.Years = 15;
                set3.DownPayment = HousePrice * .10;
                set3.Balance = HousePrice * .90;
                set3.InterestRate = InterestRate;
                set3.MonthlyPayment =
                        calc.PITI(set3.Balance,  set3.InterestRate,  set3.Years,  Insurance,
```

```
Taxes);

                retObject[2] = set3;

                /* 頭金１０%，３０年払いの場合 */
                ResultSet set4 = new ResultSet();

                set4.Years = 30;
                set4.DownPayment = HousePrice * .10;
                set4.Balance = HousePrice * .90;
                set4.InterestRate = InterestRate;
                set4.MonthlyPayment =
                        calc.PITI(set4.Balance,  set4.InterestRate,  set4.Years,  Insurance,
Taxes);

                retObject[3] = set4;

                /* 頭金２０%，１５年払いの場合 */
                ResultSet set5 = new ResultSet();

                set5.Years = 15;
                set5.DownPayment = HousePrice * .20;
                set5.Balance = HousePrice * .80;
                set5.InterestRate = InterestRate;
                set5.MonthlyPayment =
                        calc.PITI(set5.Balance,  set5.InterestRate,  set5.Years,  Insurance,
Taxes);

                retObject[4] = set5;

                /* 頭金２０%，３０年払いの場合 */
                ResultSet set6 = new ResultSet();

                set6.Years = 30;
                set6.DownPayment = HousePrice * .20;
                set6.Balance = HousePrice * .80;
                set6.InterestRate = InterestRate;
                set6.MonthlyPayment =
                        calc.PITI(set6.Balance,  set6.InterestRate,  set6.Years,  Insurance,
Taxes);

                retObject[5] = set6;

                return retObject;
        }

}
```

4) PaymentCalc クラス

```
class PaymentCalc
{

        /* 元金と利息について毎月の支払い額を計算 */
        double PI(double Balance,double AnnualInterestRate,int YearsLength)
        {
                /* 毎月の支払い額の計算 */
                double BAL = Balance;
                double INT = (AnnualInterestRate / 12);
                double MON = (YearsLength * 12);

                double PMT = BAL * (INT / (1 - java.lang.Math.pow(1 + INT, -MON)));

                return PMT;
        }

        /* 元金，利息，税金，保険料について毎月の支払い額を計算 */
        double PITI(double Balance,double AnnualInterestRate,int YearsLength
                ,int AnnualInsurance,int AnnualTaxes)
        {
                /* 毎月の支払い額の計算 */
                double BAL = Balance;
                double INT = (AnnualInterestRate / 12);
                double MON = (YearsLength * 12);

                double PMT = BAL * (INT / (1 - java.lang.Math.pow(1 + INT, -MON)));

                /* 毎月の総支払い額の計算 */
                double INS = (AnnualInsurance / 12);
                double TAX = (AnnualTaxes / 12);

                double TOTPMT = PMT + INS + TAX;

                return TOTPMT;
        }       }
```

5) Calculate.idl ファイル

```
struct ResultSet
{
        long Years;
        double DownPayment;
        double Balance;
        double InterestRate;
        double MonthlyPayment;
};

typedef sequence<ResultSet>
        MortgagePayments;

interface Calculate
{
        MortgagePayments calcPI( in long HousePrice, in double InterestRate );
        MortgagePayments calcPITI( in long HousePrice, in double InterestRate, in long Insurance,
in long Taxes );
};
```

(2) JavaRMI 版

1) MortgageCalcClient クラス

```
import java.util.*;
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

public class MortgageCalcClient
{

	public static void main(String args[])
	{
		if (args.length != 3)
		{
			System.out.println("Usage : java MortgageCalcClient ServerName
HousePrice InterestRate");
			System.out.println("Interest rate must be in demical form!");
			System.exit(0);
		}

		String ServerName = args[0];
		Integer HousePrice = new Integer(args[1]);
		Double InterestRate = new Double(args[2]);

		/* セキュリティ・マネジャの準備 */
		System.setSecurityManager(new RMISecurityManager());

		try
		{
			Calculate calc
				= (Calculate) Naming.lookup("rmi://" + args[0] + "/" +
"MortgageCalc");

			Vector v = calc.calcPI(HousePrice.intValue(),
InterestRate.doubleValue());

			for (int i = 0; i < 6; i++)
			{
				ResultSet s = (ResultSet) v.elementAt(i);
				System.out.println
				("Term of Lorn:" + s.Years + "year(s)  Down Payment:$" +
s.DownPayment);
				System.out.println
				("Interest Rate:" + s.InterestRate * 100 + "%
MonthlyPayment:$" + s.MonthlyPayment);
				System.out.println();
			}
		} catch (Exception e) {
			System.err.println("System Exception : " + e);
		}
		System.exit(0);
	}

}
```

2) ResultSet クラス

```
import java.io.*;

public class ResultSet implements Serializable
{

	int Years;
	double DownPayment;
	double Balance;
	double InterestRate;
	double MonthlyPayment;

	private void writeObject (ObjectOutputStream s) throws IOException
	{
		s.writeInt(Years);
		s.writeDouble(DownPayment);
		s.writeDouble(Balance);
		s.writeDouble(InterestRate);
		s.writeDouble(MonthlyPayment);
	}

	private void readObject(ObjectInputStream s) throws IOException
	{
		Years = s.readInt();
		DownPayment = s.readDouble();
		Balance = s.readDouble();
		InterestRate = s.readDouble();
		MonthlyPayment = s.readDouble();
	}

}
```

3) MortgageCalcServer クラス

```
import java.rmi.*;
import java.rmi.server.*;

public class MortgageCalcServer
{

	public static void main(String args[])
	{
		System.setSecurityManager(new RMISecurityManager());

		try
		{
			CalculateImpl calculator = new CalculateImpl("MortgageCalc");
			System.out.println("MortgageCalc Started!");
		} catch (Exception e) {
			System.out.println("Exception : " + e.getMessage());
			e.printStackTrace();
		}
	}

}
```

4) CalculateImpl クラス

```
import java.util.*;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class CalculateImpl extends UnicastRemoteObject implements Calculate
{

	private PaymentCalc calc;
	public CalculateImpl(String name) throws RemoteException
	{
		super();
		try
		{
			Naming.rebind(name, this);
			calc = new PaymentCalc();
		} catch (Exception e) {
			System.out.println("Exception : " + e.getMessage());
			e.printStackTrace();
		}
	}

	public Vector calcPI (int HousePrice, double InterestRate) throws RemoteException
	{
		Vector retObject = new Vector(6);

		ResultSet set1 = new ResultSet();

		set1.Years = 15;
		set1.DownPayment = HousePrice * .05;
		set1.Balance = HousePrice * .95;
		set1.InterestRate = InterestRate;
		set1.MonthlyPayment = calc.PI(set1.Balance, set1.InterestRate, set1.Years);

		retObject.addElement(set1);

		ResultSet set2 = new ResultSet();

		set2.Years = 30;
		set2.DownPayment = HousePrice * .05;
		set2.Balance = HousePrice * .95;
		set2.InterestRate = InterestRate;
		set2.MonthlyPayment = calc.PI(set2.Balance, set2.InterestRate, set2.Years);

		retObject.addElement(set2);

		ResultSet set3 = new ResultSet();

		set3.Years = 15;
		set3.DownPayment = HousePrice * .10;
		set3.Balance = HousePrice * .90;
		set3.InterestRate = InterestRate;
		set3.MonthlyPayment = calc.PI(set3.Balance, set3.InterestRate, set3.Years);

		retObject.addElement(set3);

		ResultSet set4 = new ResultSet();

		set4.Years = 30;
		set4.DownPayment = HousePrice * .10;
```

```
                set4.Balance = HousePrice * .90;
                set4.InterestRate = InterestRate;
                set4.MonthlyPayment = calc.PI(set4.Balance, set4.InterestRate, set4.Years);

                retObject.addElement(set4);

                ResultSet set5 = new ResultSet();

                set5.Years = 15;
                set5.DownPayment = HousePrice * .20;
                set5.Balance = HousePrice * .80;
                set5.InterestRate = InterestRate;
                set5.MonthlyPayment = calc.PI(set5.Balance, set5.InterestRate, set5.Years);

                retObject.addElement(set5);

                ResultSet set6 = new ResultSet();

                set6.Years = 30;
                set6.DownPayment = HousePrice * .20;
                set6.Balance = HousePrice * .80;
                set6.InterestRate = InterestRate;
                set6.MonthlyPayment = calc.PI(set6.Balance, set6.InterestRate, set6.Years);

                retObject.addElement(set6);

                return retObject;
        }

        public Vector calcPITI (int HousePrice, double InterestRate, int Insurance, int Taxes)
                                                                                        throws
RemoteException
        {
                Vector retObject = new Vector(6);

                ResultSet set1 = new ResultSet();

                set1.Years = 15;
                set1.DownPayment = HousePrice * .05;
                set1.Balance = HousePrice * .95;
                set1.InterestRate = InterestRate;
                set1.MonthlyPayment =
                        calc.PITI(set1.Balance,  set1.InterestRate,  set1.Years,  Insurance,
Taxes);

                retObject.addElement(set1);

                ResultSet set2 = new ResultSet();

                set2.Years = 30;
                set2.DownPayment = HousePrice * .05;
                set2.Balance = HousePrice * .95;
                set2.InterestRate = InterestRate;
                set2.MonthlyPayment =
                        calc.PITI(set2.Balance,  set2.InterestRate,  set2.Years,  Insurance,
Taxes);

                retObject.addElement(set2);

                ResultSet set3 = new ResultSet();
```

```
                set3.Years = 15;
                set3.DownPayment = HousePrice * .10;
                set3.Balance = HousePrice * .90;
                set3.InterestRate = InterestRate;
                set3.MonthlyPayment =
                        calc.PITI(set3.Balance,    set3.InterestRate,    set3.Years,    Insurance,
Taxes);

                retObject.addElement(set3);

                ResultSet set4 = new ResultSet();

                set4.Years = 30;
                set4.DownPayment = HousePrice * .10;
                set4.Balance = HousePrice * .90;
                set4.InterestRate = InterestRate;
                set4.MonthlyPayment =
                        calc.PITI(set4.Balance,    set4.InterestRate,    set4.Years,    Insurance,
Taxes);

                retObject.addElement(set4);

                ResultSet set5 = new ResultSet();

                set5.Years = 15;
                set5.DownPayment = HousePrice * .20;
                set5.Balance = HousePrice * .80;
                set5.InterestRate = InterestRate;
                set5.MonthlyPayment =
                        calc.PITI(set5.Balance,    set5.InterestRate,    set5.Years,    Insurance,
Taxes);

                retObject.addElement(set5);

                ResultSet set6 = new ResultSet();

                set6.Years = 30;
                set6.DownPayment = HousePrice * .20;
                set6.Balance = HousePrice * .80;
                set6.InterestRate = InterestRate;
                set6.MonthlyPayment =
                        calc.PITI(set6.Balance,    set6.InterestRate,    set6.Years,    Insurance,
Taxes);

                retObject.addElement(set6);

                return retObject;
        }

}
```

5) PaymentCalc クラス

```
class PaymentCalc
{

        double PI(double Balance,double AnnualInterestRate,int YearsLength)
        {
                double BAL = Balance;
                double INT = (AnnualInterestRate / 12);
                double MON = (YearsLength * 12);

                double PMT = BAL * (INT / (1 - java.lang.Math.pow(1 + INT, -MON)));

                return PMT;
        }

        double PITI(double Balance,double AnnualInterestRate,int YearsLength
                ,int AnnualInsurance,int AnnualTaxes)
        {
                double BAL = Balance;
                double INT = (AnnualInterestRate / 12);
                double MON = (YearsLength * 12);

                double PMT = BAL * (INT / (1 - java.lang.Math.pow(1 + INT, -MON)));

                double INS = (AnnualInsurance / 12);
                double TAX = (AnnualTaxes / 12);

                double TOTPMT = PMT + INS + TAX;

                return TOTPMT;
        }

}
```

6) Calculate インタフェース

```
import java.util.*;
import java.rmi.*;

public interface Calculate extends Remote
{

        public Vector calcPI(int HousePrice, double InterestRate)
                                                throws RemoteException;

        public Vector calcPITI(int HousePrice, double InterestRate,
                                                        int insurance, int Taxes)
                                                throws RemoteException;

}
```

3.2 のアプリケーションで用いたソースプログラム

(1) JavaRMI 版

1) SendServer.java

```
import java.rmi.*;
import java.rmi.server.*;

public class SendServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        try{
            SendImpl obj = new SendImpl();
            Naming.rebind("SendServer",obj);
            System.out.println("Server ready");
        }catch(Exception e){
            System.out.println("Exception : "+e);
        }
    }
}
```

2) SendImpl.java

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.io.*;

public class SendImpl extends UnicastRemoteObject implements SendString{
    String line;
    String D[] = new String[34];
    public SendImpl() throws RemoteException{
        super();
    }
    public String[] Send(String name) throws IOException,RemoteException{
        int x = 0;
          try{
              BufferedReader dis =new BufferedReader(new FileReader(name));
              while((line = dis.readLine()) != null){
                    D[x] = line;
                    x++;
              }
              dis.close();
        }catch(IOException e){
            System.out.println(e);
        }
          return D;
    }
}
```

3) SendString.java

```
public interface SendString extends java.rmi.Remote{
                        publicString[]Send(Stringname)throws
                    java.rmi.RemoteException,java.io.IOException;
}
```

4) SendClient.java

```
import java.rmi.*;
import java.io.*;

public class SendClient{
    public static void main(String args[]){
        String line[]=new String[34];
        long startTime;
        long stopTime;
        long rtime=0;
        long totrt=0;
          try{
            FileOutputStream fos = new FileOutputStream("data.txt",true);
            DataOutputStream dos = new DataOutputStream(fos);
            PrintStream pos = new PrintStream(dos);
              SendString obj = (SendString)Naming.lookup("rmi://"+args[0]+"/"+"SendServer");
              for(int i=0;i<10;i++){
                    startTime = System.currentTimeMillis();
                  System.out.println("リモート呼び出し前");
                    line = obj.Send(args[1]);

                    stopTime = System.currentTimeMillis();
                    System.out.println("転送終了");

                    pos2.close();
                    rtime=stopTime-startTime;
                    System.out.println("Remote Time : "+rtime+" msec");
                    pos.print("データ量= ");
                pos.print(args[1]);
                pos.print(" ¥t");
                pos.print(rtime);
                pos.print("      ¥t");
                pos.print("¥n");
                totrt=totrt+rtime;
              }
            pos.print("¥n");
            pos.print("TOTAL TIME = ");
            pos.print(" ¥t");
            pos.print(totrt);
            pos.print("      ¥t");
            pos.print("¥n");
              dos.close();
            fos.close();
            pos.close();
          }catch(Exception e){
            System.out.println(e);
        }
    }
}
```

(2) JavaIDL 版

1) SendServer.java

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class SendServer{
    public static void main(String args[]){
        try{
            ORB orb = ORB.init(args,null);
            SendImpl obj = new SendImpl();
            orb.connect(obj);
            org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
            NameComponent nc = new NameComponent("SendServer","");
            NameComponent path[] = {nc};
            ncRef.rebind(path,obj);
            System.out.println("Server ready");

            java.lang.Object sync = new java.lang.Object();
            synchronized (sync)
            {
                sync.wait();
            }
        }catch(Exception e){
          System.out.println("Exception : "+e);
        }
    }
}
```

2) SendImpl.java

```
import java.io.*;

public class SendImpl extends _SendStringImplBase{

    String line;
    String D[] = new String[34];

    public String[] Send(String name){
        int x = 0;
          try{

              BufferedReader dis =new BufferedReader(new FileReader(name));
              while((line = dis.readLine()) != null){
                    D[x] = line;
                    x++;
              }
        }catch(IOException e){
            System.out.println(e);
        }
          return D;
    }
}
```

3) SendString.idl

```
interface SendString
{
    typedef sequence<string> Line;
    Line Send(in string name);
};
```

4) SendClient.java

```
import java.io.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class SendClient{
    public static void main(String args[]){
        String line[] = new String[34];
        long startTime;
        long stopTime;
        long rtime=0;
        long totrt=0;
        try{
            FileOutputStream fos = new FileOutputStream("data.txt",true);
            DataOutputStream dos = new DataOutputStream(fos);
            PrintStream pos = new PrintStream(dos);
            ORB orb = ORB.init(args,null);
            org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
            NameComponent nc = new NameComponent("SendServer","");
            NameComponent path[] = {nc};
            SendString obj = SendStringHelper.narrow(ncRef.resolve(path));
              for(int i=0;i<10;i++){
                    startTime = System.currentTimeMillis();
                  System.out.println("リモート呼び出し前");
                  line = obj.Send(args[0]);

                    stopTime = System.currentTimeMillis();
                  System.out.println("転送終了");
                    PrintWriter pos2 = new PrintWriter(new BufferedWriter(new FileWriter(args[1])));

                    rtime=stopTime-startTime;
                    System.out.println("Remote Time : "+rtime+" msec");
                    pos.print("データ量 = ");
                  pos.print(args[0]);
                  pos.print(" ¥t");
                  pos.print(rtime);
                  pos.print("      ¥t");
                  pos.print("¥n");
                  totrt=totrt+rtime;
              }
            pos.print("¥n");
            pos.print("TOTAL TIME = ");
            pos.print(" ¥t");
```

```
                pos.print(totrt);
                pos.print("       ¥t");
                pos.print("¥n");
                  dos.close();
                fos.close();
                  pos.close();
            }catch(Exception e){
                System.out.println(e);
            }
        }
}
```