ソースコード変換を用いた新たな分散オブジェクトアーキテクチャの提案

小田謙太郎　和田智仁　吉田隆一
九州工業大学 情報工学部

分散オブジェクト技術を用いたシステム開発が普及し始めている。この様な開発では、ユーザプログラムはその技術に強く依存したものとなり、複雑になる。
我々は、ユーザプログラムをシンプルに保ち、リモートオブジェクトやリプリケーションされたオブジェクトなどに対し、単一のアクセス方法を高い透明性をともなって提供することを目的とする。これらの為に我々は、ソースコード変換と ObjectHandler、trap object を新たに導入した。

Yet Another Distributed Object Technology Architecture which employs Source Code Translation

Kentaro ODA　Tomohito WADA　Takaichi YOSHIDA
Department of Artificial Intelligence,
Kyushu Institute of Technology

System development using distributed object technologies have been common. In such development, the user program is so complicated because of its strong dependency on technologies it uses.
Our objectives are to keep the user program simple as possible and to provide uniform access to the special objects such as a remote object or a replicated object with higher transparency. In order to achieve our objectives, we introduce the source code translation method, ObjectHandlers and trap object.

1 はじめに

Java におけるポピュラーな分散オブジェクト技術として電子技術総合研究所の HORB [3]、Sun microsystems 社の Java RMI [5]、ObjectSpace 社の Voyager [4] がある。また C++では、Object Management Group の CORBA [8] や Microsoft 社の DCOM [1] などの技術がある。これらの技術を利用することで、リモートホスト上のオブジェクト(以下リモートオブジェクト)を Socket 等を用いた複雑なネットワークプログラミングを行なうことなしに利用できる。

しかし、これらの技術では、リモートオブジェクトとオブジェクト指向プログラミング言語におけるオブジェクトとの間で様々なギャップがある。これらのギャップは、リモートオブジェクトの性質上、仕方無いものもあるが、プログラマに対してギャップを埋めることを強制しているものもある。つまり、依然としてこれらの技術を用いる場合には、プログラマはオブジェクト本来の機能とは関係ないところでの記述を行なわなければならない。これは、結局のところプログラムを必要以上に複雑にしている。

そこで我々は、オブジェクトの本質的機能とは直接関係ないコードをできるだけユーザプログラムから分離し、ユーザプログラムをシンプルに保ちたいと考える。オブジェクトがリモートオブジェクトであることや、リプリケーションされているといった、オブジェクトのシステム側の側面に依存するコードをユーザプログラムから分離すべきであると考える。

これらの基本理念をもとに我々は、ソースコード変換と ObjectHandler 層、Trap object を導入する。

Trap object は、それ自身へのメソッド呼び出しを捕捉して ObjectHandler 層に処理を依頼する。ORB(Object Request Broker) 層の上に用意された ObjectHandler 層は、ORB 層を介してのリモートオブジェクトへのアクセスや、オブジェクトのリプ

リケーションといったシステム側の機能を提供する。我々は、Trap object を ObjectHandler との組合せにより、代理オブジェクトやリプリケーションされたオブジェクトとして見せている。ユーザプログラムに対して施されるソースコード変換によって、クラスを Trap object としてインスタンス化出来るようにする。

本論文では、記述言語を Java と仮定し話を進めていく。プログラムコードはある程度簡略化して記述してある。

ここでは、ローカルオブジェクトとは、ローカルに直接アクセスできるアドレス空間上にあるオブジェクト、リモートオブジェクトとは異なるアドレス空間上にあるオブジェクトのことを指している。代理オブジェクトとは、リモートオブジェクトと等価なメソッドを持ち、ORB を介してリモートオブジェクトのメソッド呼び出しを行なうローカルオブジェクトのことを指す。

2 既存分散オブジェクト技術の問題点

既存の分散オブジェクト技術では、リモートオブジェクトとオブジェクト指向プログラミング言語におけるオブジェクトとの間で、以下のようなギャップが存在する。

クラスの違い HORB や Voyager では、代理オブジェクトのクラス名は特別な物となる。例えば、HORB では `Server_Proxy`、Voyager では `VServer` となる。代理オブジェクトは、ローカルホスト上においてリモートオブジェクトを表すものである。したがって、利用するオブジェクトがリモートオブジェクトに変更なると、ユーザプログラムは大幅な変更が必要となる。我々は、クラスはオブジェクトの位置を表すものではないと考える

オブジェクトモデルの違い CORBA は、IDL を用いて CORBA オブジェクトのインターフェースを定義しており、IDL の C++や Smalltalk へのマッピングを提供している。IDL により、CORBA は、あらゆる言語に対応できる。しかし、オブジェクトモデルが異なるもの同士でのこのマッピングは、必ずしも自然なものではない。また、実際にはマッピングに関して ORB 製品の実装依存もある。

メソッド呼出しのセマンティクスの違い CORBA や HORB では、本来その言語がサポートしていないメソッド呼出しのセマンティクスをサポートしている。例えば、CORBA では、遅延同期通信や一方向通信、HORB では非同期メソッド呼び出しがサポートされている。

これらのギャップの中には、リモートオブジェクトの性質上や機能拡張のために生まれたものもある。しかし、以上の様なギャップを埋めるため様々なコードを、プログラマはユーザプログラム中に埋め込むことになる。例えば、CORBA の C++マッピングでは、メソッド呼出しを行なった後に必ず例外がどうか確認するコードが必要となる。

一方、Java ベースの技術である HORB や Java RMI は、リモートオブジェクトが Java のオブジェクトであることを前提としているので、リモートオブジェクトとローカルオブジェクトとのギャップは、CORBA と比較して小さい。しかし、クラス名の違いもさることながら、各技術への依存性は CORBA と比較して大きい。

つまり、依然としてこれらのギャップは、プログラマに対して必要以上にシステムを意識させ、それに依存したコードを記述させている。結果としてユーザプログラムは、オブジェクト本来の機能とは関係ないところで不必要に複雑になる。

3 基本理念

我々は、以下のような基本理念のもとに新たなアーキテクチャを設計する。

システムはユーザプログラムから隠蔽されるべき オブジェクトがリモートにあるか、リプリケーションされているかといった、オブジェクトのシステム側の側面に依存するような記述はユーザプログラムに記述するべきはなく、プログラマに対しては意識させるべきではない。

ユーザプログラムをシンプルに オブジェクトの本質とは関係の無いコードは、できるだけユーザプログラムから分離し、プログラムをシンプルなものに保つ

新たな機構の導入を容易に オブジェクトの移送やオブジェクトのリプリケーション、グループ通信などの新しい機構が導入しやすいオープンなアーキテクチャであるべき

パフォーマンス低下は最小に これらの実現のために、パフォーマンスが低下したり互換性がなくなったりすることは最小に抑えなければならない

4 アーキテクチャの構成

4.1 アーキテクチャの概要

ここで我々が提案するアーキテクチャの構成要素を挙げる。

Trap object 従来の Java ベースの技術と同様に Proxy object method [6] を用いる。Proxy object method は、リモートオブジェクトへのメソッド呼び出しを代理オブジェクトへのローカルな呼出しという形で実現する方法である。

我々のアーキテクチャでは、リモートオブジェクトクラスとそれに対応する代理オブジェクトクラスを同一にする。すなわち、同じクラスから、そのクラス本来の機能を持つオブジェクトを生成するか、または代理オブジェクトとして生成するかをインスタンス化する際に選択できるようにする。これは、後述するソースコード変換によって追加されたコードによって実現している。

我々はここでの代理オブジェクトを Trap object と呼ぶ。Trap object は、従来の代理オブジェクトが持つ具体的な実装を持たず、後述する ObjectHandler が持つ。Trap object は他のオブジェクトからのローカルなメソッド呼び出しを受け取り、ObjectHandler 層に知らせる役割しか持っていない。ユーザプログラム中に Trap object を生成するコードは含まれない。システム側のオブジェクトがこれを生成する。

ObjectHandler 層 ORB 層の上に ObjectHandler 層を用意する。この層は、Trap object が捕捉したローカルのメソッド呼び出しを受けて、具体的なシステム寄りの機構を提供する。例えば、メッセージのパッキングを行ないリモートオブジェクトへのアクセスを行なったり、レプリカの一貫性を保ち、レプリケーション機構を提供する。代表的な ObjectHandler としては、ProxyObjectHandler が挙げられる。Trap object と ProxyObjectHandler との組合せにおいて、その Trap object は代理オブジェクトとして動作する。ObjectHandler としての必要十分条件は、共通の ObjectHandler インターフェースを実装することである。

ソースコード変換 ユーザプログラムに対してソースコード変換を行なう。この変換によって、ユーザプログラムの各メソッドの先頭にコードが追加される。この追加コードにより、クラスを Trap object としてインスタンス化出来るようにする。メソッドが呼び出された場合に、追加コードがそのメソッド本来の処理を行なうか、ObjectHandler へと処理が移るかを決定する。

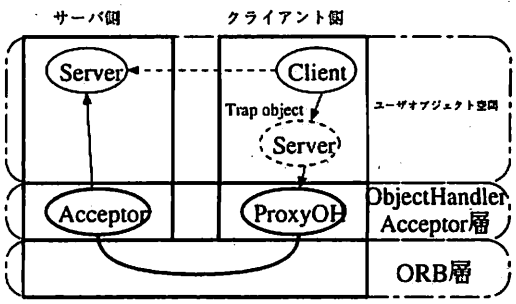

図 1: 本アーキテクチャのモデル

図 1 が本アーキテクチャのモデルである。このモデルでは、リモートオブジェクトとして利用されるクラス (図では Server クラス) に対して、ソースコード変換を行なう。変換されたクラスは従来のプログラムと互換性があり、サーバ側とクライアント側、両方において同じものを利用出来るようにしている。変換されたクラスは、サーバ側において通常どおりインスタンス化することができる。一方、我々はクライアント側における Server オブジェクトは ProxyObjectHandler を引数としてインスタンス化することにより、Trap object とすることができる。Trap object の生成は NameServer といったシステム側のプログラムが行ないプログラマからは隠蔽される。Trap object にて捕捉したメソッド呼び出しを ProxyObjectHandler が処理することで、この Trap object は代理オブジェクトとして動作する。我々は、Server オブジェクトをそれ本来の動作に加えて、代理オブジェクトとして動作させているにも関わらず、そのオブジェクトのクラスを同一にしている。これによって、クライアント側/サーバ側のコードがオブジェクトの位置に依存しないものとしている。さらに我々は ObjectHandler の導入により、ユーザオブジェクトからシステムの機能を分離している。これによって、ユーザプログラムの再変換、再コンパイルなしに新たな ORB の実装

やオブジェクトのリプリケーションといった新たな機構が導入できる。

一方、サーバ側では、Acceptorが要求を待ち受けており、サーバ内のオブジェクトに対してメソッド呼び出しを行なう。

次節からは、本アーキテクチャの各要素について実装も含めて詳しく述べる。

4.2 Trap object、ObjectHandler層、Acceptor層

ObjectHandlerは、それが設定されたオブジェクト(Trap object)のメソッド呼び出しの動作を変えるものである。例えば、ProxyObjectHandlerクラスは、ObjectHandlerの実装の一つで、それと共に生成されたオブジェクトはTrap objectとなり、Proxy objectとして動作するようになる。

ObjectHandlerの実装例として、ProxyObjectHandlerクラスの他にも、通信を暗号化する機能を持つSecureProxyObjectHandlerクラス、オブジェクトのリプリケーションといった機構を実現するReplicationObjectHandlerクラスも考えられる。また、分散した環境ではなく、デバッグやプロファイリングの為のObjectHandlerも考えられる。例えば、これらのObjectHandlerが設定されたTrap objectにおいて、呼び出されたメソッド名を表したり、メソッドの呼び出された回数や処理時間を計測できる。

ObjectHandlerは、ユーザプログラムから独立しているため、ユーザプログラムの再変換、再コンパイルなしに新たなObjectHandlerやORBを導入することができる。また、クラス毎ではなく個々のオブジェクトに対してその特性に応じたObjectHandlerを選択できる。さらには、実行時に動的にObjectHandlerを切り替えることで、オブジェクトの移送といった機構も導入できる可能性がある。

Acceptorは、クライアントからのメソッド呼び出し要求を受け付け、それを実行したのち、返り値をクライアントへ返すのがその役割である。我々はJava Reflection APIを用いることで、リモートオブジェクトクラス毎に個々に必要となるSkeletonクラスを排除している。

4.3 ソースコード変換

本ソースコード変換により、クラスがそれ本来の機能に加えて、ObjectHandlerによって提供される機能が使用できるようになる。

我々は、変換するクラスに対して、Proxyクラスや Skeletonクラスの実装を組み込まない。これにより、ユーザープログラムのシステムへの依存性を小さくし、不必要なコードの増加も防げる。

その変換とは、変換対象のクラスにおいて、ObjectHandlerへのインスタンス変数を追加し、全てのメソッドの冒頭にObjectHandlerへの呼び出しコードを挿入するものである。また、変換は対象のクラスの親クラス全てに対して行なわなければならない。ソースコード変換を施されたクラスのメソッドを疑似的に表した例を以下に示している。

```
void method(引数) {
 if (_oh != null) {
   _oh.invoked(this,
       "メソッド名(引数の型..)", 引数の配列);
          // ObjectHandlerにメソッド呼び出し
          // をパッキングした引数と共に伝える。
 } else {
   ...    // メソッド本来の実装
 }
}
```

_ohは、ObjectHandlerを参照するインスタンス変数であり、オブジェクトがObjectHandlerをコンストラクタの引数として生成されることで設定される。_ohが設定されているならば、そのオブジェクトの動作は、ObjectHandlerによって決定されるようになる。

しかし、_ohがnullであるならばそのオブジェクトは、いままでと同じ様に生成されている。したがって、そのオブジェクトは本来の処理を行なう。

4.4 実行環境

提案するアーキテクチャでは、Trap objectとして利用するクラスのソースコードをトランスレータにより自動的にソースコード変換を施した後、コンパイルを行なう(図2)。

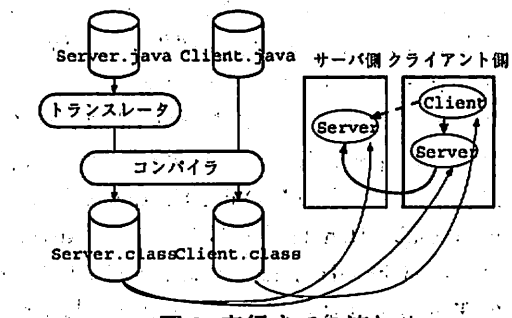

図2: 実行までの流れ

4.5 クラスのソースコード変換例

リスト 1. ユーザプログラムの例

```
class Server {
  int data[];
  Server() {
    data = new int[100];
    // 初期化コード
  }
  Video play(String title) {
    // play メソッドの実行コード
  }
}

オブジェクトの生成：
  Server aServer = new Server();
  NameServer.
        register("server",aServer);

代理オブジェクトまたはローカルオブジェクトの
参照を得る：
  Server aServer =
    (Server)NameServer.
                lookup("server");
```

リスト 2. 変換後のプログラム

```
class Server {
  private ObjectHandler _oh;
  int data[];
  Server() {
    data = new int[100];
    // 初期化コード
  }
  Server(ObjectHandler handler) {
    _oh = handler;
    _oh.instanciated(this);
  }
  Video play(String title) {
  if (_oh != null)  {
    return
     (Video)_oh.invoked(
        this,"play(String)",
        new Object[]{title});
    }  else {
      // play メソッドの実行コード
    }
  }
  void finialize() {
    _oh.finalized(this);
  }
}
```

リスト 1 に、ユーザが記述するプログラムの例を示している。この Server クラスは、リモートオブジェクトとして利用されるにも関わらず、通常の Java プログラムと全く記述は変わらない。

リスト 2 には、リスト 1 の Server クラスをソースコード変換した例を挙げている。変換後のクラスでは、ObjectHandler への参照_oh が加えられ、新たな ObjectHandler を伴うコンストラクタが追加されている。ObjectHandler をコストラクタの引数として生成されると、_oh に ObjectHandler が設定され、このインスタンスは Trap object として利用出来るようになる。また、Server クラスが Trap object として生成されると配列である data は初期化されない。これにより、無駄なメモリの消費を抑えている。

play メソッドが呼び出されると、冒頭の if 文により、ObjectHandler が設定されているかどうか判定される。ObjectHandler が設定されている場合、呼び出されたメソッドのメソッド名とその引数の型、さらに引数の配列をパラメータに、ObjectHandler の invoked メソッドが呼び出される。また、オブジェクトの生成/消滅を ObjectHandler に知らせるために、instanciated メソッドと finalized メソッドを呼び出すようにしている。

リスト 3. NameServer の実装例

```
Object lookup(String name) {
 if (name に結びつけられたオブジェクトが
          同じアドレス空間上に存在する) {
   return オブジェクトへの直接の参照を返す
 } else {
   ObjectHandler oh =
     new ProxyObjectHandler(name);
   Object aProxy = new クラス名(oh);
   return aProxy;
 }
}
```

リスト 3 では、ネームサーバの存在を仮定し、それを利用したオブジェクトへの参照を得るコード例を示している。もはやここでは、Server_Proxy といった記述は必要なく、オブジェクトの位置によらないコードの記述が可能になっている。また、最後の lookup メソッドの例では、サーバオブジェクトとクライアントオブジェクトが同じアドレス空間上に存在した場合に、クライアントに対してサーバオブジェクトへの直接の参照を渡すことによる、最適化の例を示している。

5 考察

提案した方法では、以下の事が可能になった。

1. ObjectHandler 層を導入することで、ユーザープログラムからシステムを隠蔽することができた。ユーザプログラムにシステムの実装を含めず、それを ObjectHandler に移すことで、ユーザプログラムのシステム独立性を高めた。これにより、個々のオブジェクトに応じた ObjectHandler を設定でき、新たな ObjectHandler や ORB を容易に導入できるようになった。さら

に、これらの導入にはユーザプログラムの再変換、再コンパイルは必要無い。また、動的なObjectHandlerの切り替えにより、オブジェクトの移送といった機構が組み込める可能性を提供した。

2. ソースコード変換によって、より高度なオブジェクトの透過性を提供できた。オブジェクトの位置やオブジェクトが複製されているといったことに関わり無く、オブジェクトは通常のプログラムと同じ方法で記述でき利用できようになった。これによって、記述が単一化されプログラムはシンプルなものになった。
3. 変換による通常のメソッド呼出しに対するオーバーヘッドは、if文一つとなり、本アーキテクチャ導入によるパフォーマンス低下を最小限にできた。さらには、目的のオブジェクトが同じアドレス空間上に存在する場合には、代理オブジェクトを返さずに、そのオブジェクトへの直接的な参照を渡すことで最適化が行なえた。
4. Proxyクラス／Skeletonクラスといったクラスの自動生成が必要なくなった。これにより、ユーザにシステムの詳細を意識させることなくなり、バージョン管理といったコストを小さくできたと考える。

以下に考慮すべき点を挙げる。

ソースコード変換による互換性の問題 public宣言された変数への変更が生じた場合、これを検知する手段がないために、この変数の利用はできない。我々は、public宣言された変数は持つべきではないと考える。これを支援するために、変数を検知しトランスレータは変換の際にユーザに対して警告やエラーを出力する。

提案する手法では、リモートオブジェクトクラスのメソッド、つまり呼び出される側がソースコード変換の対象となっているが、クライアント側のメソッド呼び出し部分が変換対象となる変換方法もある。この方法では、変換対象が多く、その範囲があいまいである。さらには、リモートメソッド呼び出しを静的に判定するために、プログラマによる何らかの明示的な宣言が必要となる [2]。明示的な宣言を避け、全てのメソッド呼び出しを変換する方法も考えられる。しかし、この方法ではローカルのメソッド呼び出しに対してもシステムの介在によるオーバーヘッドが発生してしまう。

6 おわりに

ソースコード変換に関する関連研究として、文献 [7] が挙げられる。この文献では、Java言語に新たな自己反映機構を提供するOpenJavaというシステム実現のために、動的な型に基づいたメタクラスの適用を行っている。文献では、この適用において呼出し先のメソッドをソースコード変換する方法を提案している。

現在、我々は提案した方法によりトランスレータの実装を行なっている段階にあり、ORBや`ProxyObjectHandler`を含むランタイムシステムは実装済みである。トランスレータは未完成であるので、手動によるソースコード変換により動作を確認した。今後の課題として、トランスレータの実装が挙げられる。

参考文献

[1] Brown, N., Kindel, C., Distributed Component Object Model Protocol - DCOM/1.0, Internet Draft, 1996

[2] CHIBA, S., TATSUBORI, M., Yet another java.lang.Class. ECOOP Workshops on Reflective Object-Oriented Programming and Systems ,pp.372-373, 1998

[3] HIRANO, S., HORB: a distributed object oriented language for worldwide programming, WOOC'96,1996(In Japanese)

[4] ObjectSpace, VOYAGER Core Technology User Guide, 1997

[5] Sun microsystems, Remote Method Invocation Specification, 1997

[6] Shapiro, M., Structure and Encapsulation in distributed Systems: The Proxy Principle,ICDCS,pp.198-205,1986

[7] TATSUBORI, M., CHIBA, S., OpenJava : Yet another reflection support for Java,In 14th Conference Proceedings of Japan Society for Software Science and Technology, pp.201-204, ISSN 0913-5391, Ishikawa, Japan, September 30 - October 2, 1997. (in Japanese)

[8] Vinoski S.,Distributed Object Computing With CORBA,C++ Report July/August 1993