プログラミング 15－1
（1997. 9. 12）

分散オブジェクト指向言語の設計と実装

芦原 栄登士† 地引 昌弘‡ 山下 雄三†
上田 哲朗‡ 久野 靖‡ 大木 敦雄‡

†筑波大学 経営システム科学専攻
‡筑波大学 企業科学専攻

分散オブジェクト環境において、透過的なオブジェクトアクセスを実現するために、従来より様々なシステムが提供されている。しかし透過性が一面的であったり、静的なクライアントサーバモデルでしかないといった問題点がある。これらを解決する分散仮想マシンを用いたオブジェクト指向プログラミング環境で、透過性や動的なオブジェクトアクセスの提供を実現するための言語 e4 について述べる。型とクラスを分離することによって、どこにどのようなオブジェクトが存在するかを意識せずに利用することができる。

Design and Implementation of Distributed Object Oriented Programming Language

Eitoshi Ashihara, Masahiro Jibiki, Yuzo Yamashita,
Tetsuro Ueda, Yasushi Kuno and Atsuo Ohki

Graduate School of Systems Management, The Univ. of Tsukuba

In the distributed object environment, several systems provide methods to access objects tranparently. But a programmer must know the location of objects he/she wants before hand, or can only describe clinet-server models statically even in these systems. We propose a programming language 'e4', which supports fully tranparent and dynamic object access, for the object oriented programming environment based on distributed virtual machines. In 'e4', semantics of type and class is clearly separeted, making it possible to use objects without knowing their locations.

1 はじめに

分散環境において、透過的なオブジェクトアクセスを実現するために、従来より様々なシステムが提供されている。中でも、OMG が提案している CORBA と呼ばれるシステムが有名であるが、最近では、CORBA と比べて相互運用性が高く、また軽量/手軽なシステムである Java が注目を集めている。また、Java をもとにより自由度の高いリモートオブジェクトアクセスを実現した HORB と呼ばれるシステムも提案されている。

著者らは、従来の分散オブジェクト環境でのオブジェクトアクセスにおける問題点、特に、一面的な透過性、静的なクライアントサーバモデル、といった問題点に注目し、これらを解決するために、分散仮想マシンを用いたオブジェクト指向プログラミング環境を提案している [1]。

本稿では、この環境においてオブジェクトにアクセスするためのプログラム言語の設計とシステムの実装に関して述べる。

以下、第 2 章では、従来の分散環境における問題点を、第 3 章では、その解決策である分散仮想マシンを用いたモデルについて説明する。続く第 4 章では、その環境でオブジェクトにアクセスするための言語 e4 の設計について述べ、第 5 章では、システムの実装について説明する。最後に第 6 章で議論とまとめを行なう。

2 従来の分散オブジェクト環境

2.1 一面的な透過性

分散環境とは、「ネットワークを用いて結合された計算資源が、互いに協力してサービスを提供するシステム」である。ここでは、従来の分散環境における問題点と解決手段について述べる。

従来より、分散環境では、透過的なオブジェクトアクセスを提供するために、多くの努力が払われている。しかしながら多くの場合、オブジェクトアクセスは、ユーザ (オブジェクトサービスの享受者) に対して透過的であるものの、オブジェクト実装者 (オブジェクトサービスの提供者) に対して透過的であるわけではない。例えば、オブジェクト実装者が、他者の実装したオブジェクト B を利用してオブジェクト A を実装した場合を考えてみる。ユーザが A を実行する場合、B の場所を知らなくても、A を介して B のサービスを享受することが可能である。

しかしながら、実装者は、A を実装する際に、B の存在する場所を把握して、ユーザに対して隠蔽された A から B への通信手段を用意する必要がある。この場合、実装者は、自分自身の責任で必要なオブジェクトへのアクセスを提供しなければならず、オブジェクトアクセスは何ら透過的ではない。

例えば、Java アーキテクチャ上で稼働する HORB では、オブジェクトの実装者は次のように、サーバオブジェクトの位置を URL で指定しなければならない。

```
class Client {
 void foo() {
   ...
  host = "gssm.otsuka.tsukuba.ac.jp";
  HorbURL url = new HorbURL(host, null);
  Server_Proxy server =
                new Server_Proxy(url);
  ...
 }
}
```

つまり、透過的なオブジェクトアクセスが提供されていると言えるのは、最終的なユーザに対してのみであり、実装者は、状況に応じてユーザの立場となる場合が存在するにもかかわらず、透過性が提供されていないといえる。

2.2 静的なクライアントサーバモデル

また、従来の分散環境では、多くの場合、サービスを提供するサーバオブジェクトがその場所を移動することはない。したがって、オブジェクトという存在でありながら、オブジェクトアクセスは、基本的に分散アプリケーションにおけるクライアントサーバモデルと大差ないものになってしまっている。

最近の Java のようなシステムでは、オブジェクトを移動させる機能を備えてはいるものの、実行前のオブジェクトを移動できるのみで、実行中のオブジェクトを移動させ、移動した先で実行を再開させ

る機能 (マイグレーション機能) は有していない。

これは、社内 LAN 環境のような、構成する要素 (マシン、ネットワーク) に比較的差がなく静的な環境においては、欠点がクローズアップされることはないかもしれないが、モバイルマシンまで含めたインターネット環境を考えると、以下のような問題が発生する。

- インターネットは、通信網自体が一様ではないために、通信回線の違いによる通信コストに大きな差が生じる。
- 同様に、プラットフォームも多くの種類が存在しており、オブジェクトを実行するプラットフォームの種類によって、実行コストに大きな差が生じる。

したがって、もしオブジェクトが自由に移動可能であれば、通信/実行コストの安いオブジェクトアクセスを提供することが可能となるが、現状のクライアントサーバ型アクセスでは、これらを提供することができない。

2.3 オブジェクトの生存期間

生成されたあるオブジェクトが使われなくなるのはいつだろうか。単一マシン内では、ガベージコレクションによって、使われなくなったオブジェクトを破棄、回収することができるが、分散環境では、生存期間の追跡は難しい。社内 LAN 環境のような比較的静的な環境では、オブジェクトが生成されてから、使われなくなるまでの期間をある程度把握することができる。しかし、インターネットのように大規模で静的でない環境においては、もはやオブジェクトの生存期間を追跡することは不可能である。

単純な解決方法として、(a) 使われなくなったオブジェクトはそのまま放置する、(b) 明示的に解放する、という方法がある。しかし、(a) の方法をとると、いずれは資源を使いはたしてしまう。また、(b) の方法では、M が N を、N が M を利用する、といったアクセスのループが生じると解放されなくなってしまうので、やはり資源を使いはたす恐れがある。アクセスのループが生じないように利用すればいいのだが、後述のオブジェクト探索によって、実際に利用するオブジェクトが、どのオブジェクトかを事前に決定することは少ないので、ループが生じる可能性は高い。また、オブジェクトへのアクセスは透過的に行うのであるから、解放も明示的にではなく、透過的に行いたい。

3 分散仮想マシン

モバイルマシンまで含めたインターネット環境のような、多くの種類のプラットフォームやネットワークが混在する環境で、自由度の高いオブジェクトのマイグレーション機能を実現するには、プラットフォームに依存しないオブジェクト表現や、プラットフォームの違いを吸収するアーキテクチャが必要となる。そこで、オブジェクトアクセスに関する、言語レベルのサポートのために、後述する言語 e4 と仮想マシンを用意した。

オブジェクトアクセスに関する透過性を提供するには、

1. オブジェクトの位置管理
2. オブジェクトの探索、アクセスの仲介

を行う機能が必要となる。

これは、分散環境内の各構成要素 (プラットフォーム) が、内部でオブジェクトに関する情報を交換しあい、外部に対して 1 個の仮想空間をみせることで実現できる。今回は、前述の仮想マシンを分散仮想マシンに拡張し、これらの分散仮想マシンが近隣の分散仮想マシンと通信しあうことで、必要なオブジェクトの探索、仲介を行う。

分散仮想マシン上でのオブジェクトの利用はは次のようになる。オブジェクトのサービスを利用する時は、オブジェクトそのものを指定するのではなく、サービスを指定する。これは、オブジェクトの詳細は知らなくても、受けたいサービスを知っていればオブジェクトを利用できるという利点がある。サービスを指定するには、サービスの種類の他、サービスの属性を指定できる。サービスを指定したときに、そのサービスを提供することのできるオブジェクトがただ 1 つ存在するとはかぎらない。また、同じサービスを提供していても、オブジェクトごとに何らの差があるはずである。そこで、サービスの種類だけでなく、サービスの属性を指定できるようにする。ここでいう属性とは、オブジェクトによって

おのずと決まるものの他に、オブジェクトの存在する環境によって決まるものがある。たとえば、オブジェクトの計算能力が事前に判明していればその計算能力は属性であるし、オブジェクトの (利用者からの) 距離や、ネットワークの負荷なども属性である。よって、利用者は、サービスを指定する時に、「これこれこういうサービスで、計算能力はこれくらい、距離はできるだけちかく」のような指定ができるようになる。

オブジェクトのマイグレーションをサポートするが、移動をユーザが指定するのでは透過的とはいえない。ユーザはオブジェクトがどこにあるかを全く意識しない。同じように、オブジェクトがどこで実行されているのかも意識する必要はない。そこで、マイグレーションは、システムの管理下で行う。システムがより負荷の軽い環境を探し、オブジェクトをマイグレーションさせる。ユーザはサービスの指定さえすればよく、環境の負荷などは気にせずに利用できる。

また、オブジェクトには寿命があるものとする。これは、「オブジェクトには、ある生存期間がある。生存期間をすぎたものは、消滅する (破棄される)。」というものである。

オブジェクトを生成する時に、オブジェクトの生存期間を指定できるようにする。こうすることによって、たとえば、このオブジェクトはすぐに使わなくなるので1時間の生存期間、あのオブジェクトはしばらく使うので1週間、などというように、オブジェクトの種類によって、破棄するタイミングだけを指定できるのである。通常、ある仕事をオブジェクトにまかせるときには、その仕事が十分終了する生存期間をあたえる。しかし、時には、その仕事が終了するまえに生存期間が終ってしまう可能性がある。この場合は仕事が終了せずにオブジェクトが破棄されてしまう。これは問題であるようだが、そもそもインターネットのような信頼性の低い環境においては、いつネットワークが切断されるかわからない。オブジェクトが仕事を完了できずに生存期間が終ってしまうことや、生存期間中でもネットワーク切断によって、オブジェクトとの連絡がとれなくなることがある。そのことに対応したプログラミングが必要となるのである。したがって、仕事を完了せずに消滅するオブジェクトは、生存期間にかかわらず起こり得る事態なので、それに対処するのはユーザの責任と認識できる。これは、ユーザには負担であるが、今回はこの仕様にした。将来は、解決策として、オブジェクトのマイグレーションに加え、仕事の委託を採り入れるという策をとりこんでいきたい。生存期間の切れたオブジェクトが仕事途中であるならば、仕事を他のオブジェクトに委託するのである。

4 プログラム言語 e4

前節で述べたモデルにもとづき、新たなプログラム言語 e4 を設計した。この言語の特徴は、型とクラスを完全に分離したこと、オブジェクトは探索して見つけること、オブジェクトには生存期間があること、などである。

新しい言語を作った理由は、まず第一に、オブジェクトアクセスに関して、言語レベルでのサポートを含めたかったからである。オブジェクトへのアクセスが透過的になっても、言語レベルでのサポートがなければ使いにくいものとなってしまうからである。

新たな言語を作らない場合は、既存の言語を利用し、その言語のライブラリとしてモデルを実現することになる。また、言語をすべて作るのではなく、既存の言語を拡張する方法もある。例えば HORB は、Java 言語を拡張しているとみなすことができる。

ライブラリとして実装することはおそらく新言語を開発するよりも簡単であるだろう。しかし、元の言語には存在しない概念をライブラリのみで実装すると、ユーザの負担が大きくなる。ライブラリ関数のコールする順番など、仮定事項が増えるからである。

既存の言語に新機能を追加する形では、その言語でもともと使用できる機能と追加した機能とで、オブジェクトの利用の仕方に差がでてしまう。すべてを統一的に扱うのが難しくなる。

著者らの設計した言語の第一の特徴である、「型とクラスの分離」という概念を既存の言語に採り入れることは難しい。既存の言語では、クラスは型としても使用できるのが普通である。

また、同期機構として採用している、抽象状態同

期も既存の言語に採り入れるには難しい。

これら1つ1つを既存の言語のライブラリとして実装したり、既存の言語を拡張する形で実装することは不可能ではないが、元の言語とはかけ離れた言語となるか、ライブラリコールだらけの読みにくいソースしか書けない言語となることが予想される。

以上のことから、既存の言語を拡張あるいはライブラリを作るのではなく、新言語を開発することにした。

4.1 型とクラスの分離

著者らの考えるモデルの利点は、一言でいうと、「どこにどんなオブジェクトがあるかを気にせずに利用できること、」である。このモデルを実現するために、オブジェクトをサービスとその実装とに分けて考える。ユーザは、サービスを指定するのであって、実装は意識しないでよい。そのために、サービスと実装は明確に区別する必要がある。

ここでいうサービスとはオブジェクトにできることであり、オブジェクトのインターフェースのことである。つまり、オブジェクトのメソッドには何があるか、ということである。これを、オブジェクトの型と呼んでいる。

そして、クラスはある型の実装であるので、サービスの実装とはクラスのことである。

通常のオブジェクト指向言語では、クラスと型を同一視しているものが少なくない。クラスも型として使用できる言語が多い。例えば、C++やJavaでは、

```
class FOO {
    ...
};
```

のように、あるクラス FOO を定義すると、

```
FOO a;
```

のように、「FOO 型の変数」というものが宣言できる。

これに対し、型とクラスを分離した場合、例えば次のように、

```
型 FOO {
    ...
};
```

```
class CFOO 型 FOO {
    ...
};
```

型 FOO と、型 FOO の実装であるクラス CFOO があり、

```
FOO a;
```

のように、「FOO 型の変数」というものが宣言されるのである。CFOO は、型の実装であるクラスであって、型その物ではないので、

```
CFOO a;
```

のような、「型 CFOO の変数」という宣言は、誤りである。

以上のように型とクラスを分けることにより、分散環境では次のように利用する。

- あるサービスを利用したくなったら、そのサービスを提供するオブジェクトを捜し出して、サービスを利用する。
- プログラム上では、型を指定してオブジェクト生成要求を出す。システムが 捜し出した実装 (クラス) のオブジェクトが生成される。

このときに指定するのはオブジェクトの型だけであるので、システムはどの実装を用いるべきかオブジェクトの探索をする。

言語 e4 では、型の定義を次のように記述する。

```
signature FOO is
  bar();
  baz(a:int):int;
end
```

型 FOO は、メソッド bar と baz を備えている。

型 FOO の実装であるクラスは次のように記述する。

```
class CFOO sig FOO is
  bar() is
    ...
  end
  baz(a:int):int is
    ...
  end
end
```

オブジェクトの生成は型と探索条件を指定する。すなわち、どんなサービスを利用したいかだけを指定するのである。

```
f:FOO;  # 型 FOO の変数 f
f := new FOO<"探索条件">;
```

この場合は、型 FOO の実装はクラス CFOO しか存在しないので、必ず CFOO のオブジェクトが生成されるが、型 FOO の実装が複数存在する場合には、探索条件によってどの実装が利用されるかが決定される。

4.2 オブジェクトの探索

オブジェクトは、探索条件によって探索される。探索条件は、次の2種類がある。

- オブジェクト自身の属性に関する条件
- 実行環境に関する条件

オブジェクトには探索のための属性を持たせることができる。例えば、次のように記述することで、同じ型であっても属性の異なるオブジェクトを記述できる。

```
signature AAA is
  property ppp;
  ...
end

class BBB sig AAA is
  property ppp = "1";
  ...
end

class CCC sig AAA is
  property ppp = "2";
  ...
end
```

オブジェクトの生成時には、探索条件として属性を指定できる。

```
# 属性 "1" を探索
a:AAA := new AAA<ppp = "1">;
```

実行環境に関する条件は、距離や速度など、オブジェクトの実装時には判別できない条件である。

```
# 距離3以下で探索
a:AAA := new AAA<hop = "3">;
```

4.3 オブジェクトの生存期間

言語 e4 では、オブジェクトの生成時に、オブジェクトの生存期間を指定することができる。これは、生存期間中は必ず存在するという保証ではないし、生存期間を過ぎると必ず破棄されるという保証でもない。最低でも生存期間中は破棄しようとはしない、というだけである。生存期間を過ぎてもまだ存在するオブジェクトがあるかもしれないし、生存期間中であるにもかかわらず、存在しなくなるオブジェクトもあるかもしれない。消滅したオブジェクトにアクセスすると、例外が発生する。いずれにせよ、生存期間をすぎてしばらくすればオブジェクトは破棄されていく。

4.4 抽象状態同期

オブジェクトの同期機構には、抽象状態同期 [2] を導入した。抽象状態とは、データ抽象機能を持つ言語において、内部状態を抽象化した形で外部に公開するものである。抽象状態同期とは、その抽象状態に基づく同期機構のことであるが、状態情報がオブジェクトのインターフェースの一環をなすため、選択送信などの機構が実現でき、複数オブジェクトに関わる同期が自然な形で記述できる。

5 システムの実装について

言語 e4 処理系、分散仮想マシンは、現在 unix 上で開発中である。バイトコードは処理系の開発のしやすさからスタックマシンとして C 言語を用いて実装している。e4 のコードは処理系により以下のようなバイナリコードに変換され、1クラスが1ファイルに格納される。[1]

```
CodeFile {
  u4 magic_number;
  u2 lang_version;
```

[1] u1 は 1 バイト,u2 は 2 バイト、u4 は 4 バイト

```
  u4 code_version;
  u2 const_pool_cnt;
  const_info const_pool[const_pool_cnt-1];
  u2 this_class;
  u2 super_class;
  u2 this_type;
  u2 slot_size;
  u2 slot_count;
  slot_info slots[slot_count];
  u2 shared_count;
  shared_info shared[shared_count];
  u2 method_count;
  method_info methods[method_count];
  u2 name_count;
  u1 src_name[name_count];
}
```

各フィールドは次のような情報を持つ。

magic-number
　マジックナンバー
lang-version
　言語仕様のバージョン
code-version
　コードのバージョン
const-pool-cnt
　const-pool のサイズ
const-pool
　このオブジェクトで利用する各定数情報
this-class
　クラス名の const-pool へのインデックス
super-class
　スーパークラスのインデックス
this-type
　型のインデックス
slot-size
　スロット (メンバ変数) の合計サイズ
slot-count
　slots のサイズ
slots
　スロットの情報
shared-count
　shared のサイズ
shared
　スタティックメンバ変数の情報
method-count
　methods のサイズ
methods
　メソッドの情報
name-count
　src-name のサイズ
src-name
　ソースファイル名

次にバイトコードの例をあげる。
次のような型があり、

```
signature FOO is
  bar();
end
```

オブジェクトを生成するコードがある。

```
a:FOO := new FOO<>;
a.bar();
```

このコードをコンパイルすると次のような (疑似) バイトコードになる。

```
; a:FOO := new FOO<>
push attribute  ; 探索のパラメータを push
search FOO      ; 型 FOO を探索
                ; 結果はスタックに入る
invoke init     ; コンストラクタを呼び出す
assign a        ; a に代入
; a.bar()
push a          ; a を push
invoke bar      ; bar を呼び出す
```

各ホスト上の仮想マシンは互いに通信しあい、オブジェクトのやりとりをおこなう。すべての仮想マシンは平等であり、仮想マシンの階層構造は存在しない。仮想マシンを管理する別のプロセスというものも存在しない。

仮想マシンは、次の 5 つのモジュールで構成されている。

- バイトコードインタプリタ : バイトコードを実行する。
- 状態サーバ : 他のマシンの状態サーバと通信し、ネットワークや CPU などの情報を保持する。

- 探索サーバ：オブジェクトを探す。
- オブジェクト通信サーバ：オブジェクト間の通信を行なう。
- サーバマネージャー：状態、探索、通信の３つのサーバを管理する。

オブジェクトを生成し、他のホストマシン上に発見し、メッセージを送信する場合の流れは次のようになる。

1. バイトコードは、まず、バイトコードインタプリタによって実行される。
2. オブジェクト生成の要求があると、バイトコードインタプリタは、サーバマネージャーにその要求を出す。
3. サーバマネージャーは、探索サーバに該当するオブジェクトを探索させる。
4. (オブジェクトが見つかったら、) サーバマネージャーは、状態サーバに状態を問い合わせ、どこで実行するべきかを決定する。
5. オブジェクトの位置が決定すると、バイトコードインタプリタの実行に戻る。
6. オブジェクトにメッセージを送信する要求があると、バイトコードインタプリタは、サーバマネージャーに通信要求を出す。
7. サーバマネージャーは、状態サーバにリソース状況を問い合わせ、マイグレーションの必要があれば、マイグレーションを実行する。
8. サーバマネージャーは、オブジェクト通信サーバに通信要求を出す。
9. オブジェクト通信サーバは、相手ホストのオブジェクト通信サーバと交信し、要求された通信をおこなう。

探索サーバがローカルマシン内に該当するオブジェクトを発見した場合は、他のマシンとの通信は行なわれないので、オブジェクト通信サーバは使用されない。

仮想マシンの各モジュールはそれぞれ別スレッドとして実装する。探索サーバや状態サーバは近接のマシンには UDP で通信を行ない、遠距離は TCP で通信を行う。

6 まとめ

本稿では、従来の分散オブジェクト環境でのオブジェクトアクセスにおける問題点に着目し、オブジェクトアクセスに対する透過性を提供し、また動的なオブジェクトアクセスを実現するために、言語 e4 と分散仮想マシンについて述べた。

これらの機能を実現した分散オブジェクト環境では、この環境下で動作するユーザインターフェースとなるシェルを用意し、またファイルをオブジェクトとすることで、環境自体に対する透過性も提供することが可能となる。

分散仮想マシンの性能評価を行うと共に、ファイルオブジェクトやシェルを用意して環境自体に対する透過性の提供について調べていきたい。

参考文献

[1] 著者：分散仮想マシンを用いたオブジェクト指向プログラミング環境, 情報処理学会プログラミング研究会資料 96-PRO-10-7, pp37-42, 1996

[2] 久野靖, 大木敦雄：抽象状態に基づく並列オブジェクト指向言語 p6, 情報処理学会論文誌 Vol.38 No.3, pp563-573, 1996

[3] K.Arnold, J.Gosling: "The Java Programming Language", Addison-Wesley, 1996