システムソフトウェアと
オペレーティング・システム 73－9
（1996. 8. 26）

オブジェクトの交換を利用した分散サービス利用のためのフレームワーク

西岡 利博*　　塚本 享治
nishioka@oz.ipa.go.jp　　tukamoto@etl.go.jp
三菱総合研究所　　電子技術総合研究所
〒100 千代田区大手町 2-3-6　〒305 つくば市梅園 1-1-4

* 開放型基盤ソフトウェアつくば研究室研究員

分散環境で提供されているサービスを利用する場合、そのサービスの内容や利用方法は個々に異なるので、最適なサーバを探すためのフレームワークが必要になる。トレーディングサービスはこのためのフレームワークであり、ANSA, CORBA などでアーキテクチャモデルが提案されている。これらは、最適なサーバをトレーダが選択し、選択したサーバをクライアントに提示するモデルである。しかし、同種のサービスを提供するサーバが増えるにしたがって、サーバ間の違いをより細かく複雑に表現できる要求が高まり、また、サーバの機能と能力の増大により、クライアント側で、サーバの故障時に複製に切替えたり、通信コストを低減するためにサーバデータの一部をキャッシュしたりといった利用技術も要求されるようになることが予想されるが、CORBA のモデルではこれらに対応しきれない。

筆者らが開発したオブジェクト指向分散環境 OZ++ は、分散環境で安全にオブジェクトをやりとりできる機能を持っている。この機能をトレーディングサービスに応用することで、よりアプリケーションの要求を実現しやすい柔軟なフレームワークが実現できた。

A Framework to use Distributed Servers Based on Exchanging Objects

Toshihiro Nishioka*　　Michiharu Tsukamoto
nishioka@oz.ipa.go.jp　　tukamoto@etl.go.jp
Mitsubishi Research Institute　　Electrotechnical Lab.
2-3-6, Otemachi, Chiyoda-ku, Tokyo, 100 Japan　1-1-4, Umezono, Tsukuba, Ibaraki, 301 Japan

* Researcher, Tsukuba Laboratory, Open Fundamental Software Technology Project

In a distributed environment, a certain framework of searching the most suitable server is required because the service contents and the usage of a server differs in each. The trading service is one of such frameworks and the models have been proposed by ANSA and CORBA. According to the models, the trader selects the best servers and returns them to the client. However, it will be difficult to follow the increasing demands to express distinctions between the servers more strict and complicated way because of the increase of the servers of a same service kind, and to provide higher-level technique to use the servers because of increase of the functionality and the capability of the servers (e.g. switching to an alternative server on a failure, caching a part of server data to deduct communication overhead, etc.)

In OZ++, our object-oriented systems environment, objects can be securely exchanged in a distributed environment. Applying this feature to the trading service, we propose a flexible framework which can satisfy the application demands.

1 はじめに

ネットワークの高速化に伴い、より多くの種類のサービスがネットワーク上で利用可能となっている。それにも関わらず、変更 / 増大するサービスを適切に利用し続け、分散環境のメリットを享受するには、サービスの提供者と利用者を結び付けるためのフレームワークが必要がある。トレーディングサービスは、このためのフレームワークであり、クライアントからのサービス要求をいかにサーバからのサービスオファーに結び付けるかを規定する [1]。図 1は、CORBA のトレーディングサービスのアーキテクチャモデルである [2]。

筆者らは、OZ++ というオブジェクト指向分散環境を研究開発している [5]。これは、未知のクラスのオブジェクトをネットワーク上でやりとりしても、正しく、かつセキュリティ上の問題がなく動作するシステムである [6]。本稿では、OZ++ の特徴を活かしたトレーディングサービスを構築した経験から [7]、そのメリットを明らかにする。その主なものは、以下の通りである。

- サーバやクライアントからトレーダに送られるデータをオブジェクト化することにより、トレーダ内部でサーバとクライアントが直接対話できる。これにより、よりアプリケーションに特化された基準で適切なサーバを選択できる。
- クライアント側に送るサービスオファーもオブジェクト化することで、サーバの利用に際して、そのサーバに固有の機能を効果的に利用できる。

以下、まず 2節で CORBA のトレーディングサービスのモデルを説明する。次に 3節で OZ++ システムの特徴を説明する。続いて 4節で OZ++ 上のトレーディングサービスの概要を説明する。最後に、関連研究事例の紹介とまとめを行う。

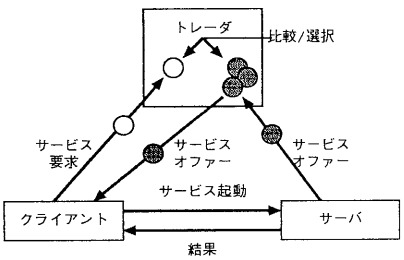

図 1: CORBA のトレーディングサービスのモデル

2 CORBA のトレーダ

この節では、CORBA のトレーディングサービスモデル [2] を説明する。

トレーディングサービスとは、クライアントからのサービス要求を適切なサーバに引き合わせることである。これを**マッチメイク**と呼ぶことがある。図 1は、CORBA のトレーディングサービスモデルを示している。トレーディングサービスの提供者を**トレーダ**と呼ぶ。サーバは、自身の能力や特徴を記した**サービスオファー** (以下単にオファーと言う) をトレーダに送る。オファーは、提供するサービスのインタフェースを表す**サービス型**と、その内容を表すプロパティ (属性 - 値対) の列から構成される。クライアントは、受けたいサービスの内容と特徴を記した**サービス要求**をトレーダに送る。サービス要求は、インタフェースを指定するサービス型と、検索条件を指定する、**制約**、**プリファレンス**、**検索方針**からなっている。トレーダは、サービス要求を受けとると、制約を満たすオファー (複数のこともある) を探し、プリファレンスにしたがってそれらに順序をつけ、クライアントに返す。何をもって最適とするかは、クライアントが指定した検索方針をもとに、トレーダが決定する。

このモデルのメリットは次の点である。

- サービスの提供と利用のフレームワークが示されているので、アプリケーションごとに個別の工夫をしなくてすむ。
- 分散環境で故障や構成変更がある際には、このモデルならばクライアント側の負担は少なくて済む。

しかし、このモデルでは以下の点が十分でない。

- 検索方針はクライアントから指定されても、その方針を提供するのはトレーダである。従って、トレーダを開発した時点で予想できなかった要求には応えられない。
- 検索条件は、制約とプリファレンスという形式しかない。複数のプロパティ間の数値的な制約などの、複雑な制約は、オファーが返ってきてからクライアントで確認するしかない。
- クライアントに返されるのはオファーのリストであり、それらを適切に取捨選択するのはクライアントの責任である。しかし、複製など、簡単には使いこなせない機能を提供しているサーバもあり、サーバへの参照に加えて、効果的な使い方を示せないと利便性に欠ける。

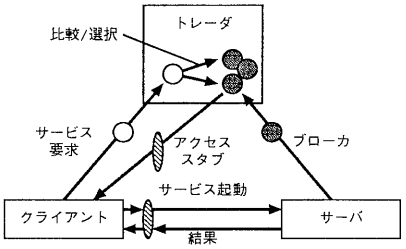

図 2: OZ++ のトレーダのモデル

3 OZ++ の特徴

オブジェクト指向分散環境 OZ++ [5] は、分散システム構築のための環境である。主な特徴は、ネットワーク上のリモートオブジェクトへのメソッド起動が可能であることと、そのメソッド起動の引数や返り値として、あらゆるオブジェクトのコピーを送受信できることである。オブジェクトを動作させるためにはクラスが必要だが、未知のクラスが必要となった場合、ネットワーク上から自動的にロードする機構が実現されている。

未知のクラスをネットワーク上からロードして実行するとセキュリティ上の問題を生じる可能性がある。OZ++ ではこれを、専用のアプリケーションレベルのゲートウェイでオブジェクトに対してマーキングすることと、外来のクラスを再コンパイルして検査し、危険な挙動をする可能性のあるクラスを受け入れないことで解決している [6]。

4 OZ++ のトレーダ

未知のオブジェクトでも安全に動作させられるという OZ++ の特徴を活用し、図 2のような枠組でトレーダを実現した。すなわち、表 1に示すように、オファーとサービス要求をオブジェクト化した。

4.1 ブローカ

サーバからトレーダに送り込むオファーをオブジェクト化したものを**ブローカ**と呼ぶ。ブローカは、自分

送信者	受信者	CORBA	OZ++
Server	Trader	オファー	ブローカ
Client	Trader	サービス要求	サービス要求
Trader	Client	オファー	アクセススタブ

表 1: データのオブジェクト化

```
class SearchEngineBroker : Broker {
    // Broker の subclass である
  constructor: NewSearchEngineBroker;
  public: GetAccessStub,...;  // overridden
  public: CaseInsensitive,WholeText,KeyWord,
          NewsGroups,...;

  global SearchEngine engine;
  // リモートメソッド起動可能なオブジェクトには
  // global 修飾子をつける

  void NewSearchEngineBroker
         (global SearchEngine e) {
    NewBroker(); // スーパクラスのコンストラクタ
    engine = e;
  }
  int CaseInsensitive() {return 1;}
                // case insensitive 検索可
  int WholeText() {return 1;} // 全文検索可
  int KeyWord() {return 1;} // キーワード検索可
  Set<RegularExpression> NewsGroups() {
    // WWW ページの他に検索可能なニュースグループが
    // あれば、それを返す。
  }

  AccessStub GetAccessStub(ServiceRequest req) {
   SearchEngineAccessStub stub;
   return stub=>NewSearchEngineAccessStub(self,req);
   // インスタンス生成には => を使う
  }
}
```

図 3: ブローカの例

が担当するサーバ (群) が何を提供できるか知っている。トレーダは、ブローカが大勢集まる市場のようなものになる。

WWW ページの検索エンジンを例にとる。SearchEngine というサーバを作ったとすると、そのためのブローカ SearchEngineBroker は、例えば図 3のようになる。各種のメソッドは、このブローカが扱う検索エンジンの特徴を表現している。

オファーがオブジェクト化されたことによる利点として、トレーダからサーバ本体が隠蔽され、サーバから独立したオファーを送れることがあげられる。すなわち、特定のサーバ (群) から送り込まれて、そのサービス内容を知らせるのではなく、互いに関係の希薄な複数のサーバに対する透過的なアクセスを提供するブローカも作成できる。図 4は、そのようなブローカの例である。互いに関係のない三つのサーバを持ち、クライアントからの要求をこれらに適切に振り分けて実行させる。このブローカを送ったオブジェクト

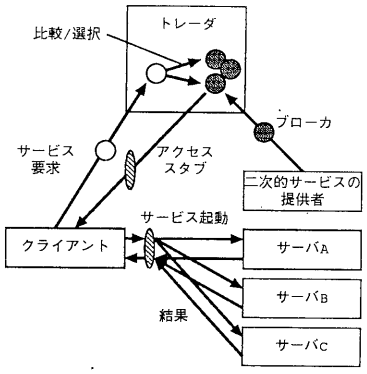

図 4: 異なる種類のサーバを透過的に見せるブローカの例

は、必ずしもサーバでなくてよい。オファーがサーバから独立した結果、他で運用されているサーバを利用した、二次的なサービスも提供できるのである。

4.2 アクセススタブ

クライアントに返ってくるオファーは、ブローカがそのまま返ってきても役に立たないので、ブローカがアクセススタブというオブジェクトを生成して返すこととした。

CORBA では、クライアントに返されたオファーをどう扱うかは、クライアントに任されている。よく知っているサービスであれば任されてもよいが、そうでない場合もある。故障時に適切に複製を選択するようなプログラミングは簡単ではないし、特にクライアントよりも後から開発されたサービスは、うまく扱えない可能性が高い。

OZ++ では、ブローカがアクセススタブを返すことによって、以下の機能を実現できる。

- クライアントが要求しているインタフェースを提供するアクセススタブを返す。
- クライアントの位置やオーナによって、振舞いやインタフェースを変える。
- データの一部をアクセススタブにキャッシュし、通信コストを軽減する。
- サーバの故障時に、別のサーバに切替える。

再び検索エンジンを例にとる。複製を持つ検索エンジンを作ったとする。このとき、この検索エンジンのためのブローカは、故障透過性を要求するサービス要求に対しては故障透過性のあるアクセススタブを、そ

```
class MyAccessStub : SearchEngineAccessStub {
  // 故障透過性のあるアクセススタブ
  ...
  HTML Search(String key) {
    HTML ret;
    while (ret != 0) {
      if (/* CurrentServer が生きているなら */) {
        ret = Delegate(CurrentServer,key);
        // CurrentServer に検索を委譲する
      } else {
        // 故障していたら適切な複製を探し再試行する
        CurrentServer = SelectCurrentServer();
      }
    }
    return ret;
  }
}
```

図 5: 故障透過性のあるサービスを提供するアクセススタブの例

うでなければ普通のアクセススタブを作って返すということもできる。

ここで言う故障透過性のあるアクセススタブとは、例えば図5のようなものを想定している。初めにサーバのリストを受けとっておき、故障を検出したらサーバを切替える。この処理はクライアントからは隠蔽されている。

4.3 サービス要求

サービス要求もオブジェクト化した。

CORBA ではマッチメイキングはトレーダの責任であるが、OZ++ では、トレーダの中でサービス要求とブローカが直接対話する。トレーダ自身は両者を比較 / 選択したりはしない。クライアントの要求を理解したオブジェクトが、その優先順位に従ってサービス内容を比較するので、無駄がない。

図 6は、サービス要求とブローカの対話の様子を図示したものである。最初にサービス要求がブローカを選んだ後、ブローカがサービス要求を参照してアクセススタブを生成する。

図 9はサービス要求のコーディング例である。この例では、 case insensitive な全文検索を、少なくとも fj.lang.oz++ の記事に対して実行できることを条件とし、その条件が満たされているサーバで最初に見つかったものを選択する。MakeMatch の実現を変更することで、任意の検索方針を実現できる。

ブローカは、このサービス要求を受けとると、図3のメソッド GetAccessStub に見るように、アクセ

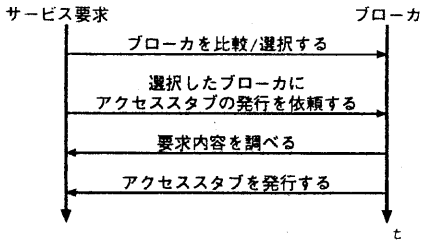

図 6: OZ++ トレーダ内でのマッチメイキング

```
class SearchEngineAccessStub:AccessStub {
  ...
  int CaseInsensitive;
  int WholeText;
  int KeyWord;
  protected Set<String> NewsGroups;
  ...
  void NewSearchEngineAccessStub
        (SearchEngineBroker b,ServiceRequest req) {
    if (/* 検索エンジンのサービス要求であれば */) {
      CaseInsensitive = sreq->CaseInsensitive();
      WholeText = sreq->WholeText();
      KeyWord = sreq->KeyWord();
      NewsGroups = sreq->NewsGroups();
      // 検索条件を記憶しておく
    } else {
      ...
    }
  }
}
```

図 7: アクセススタブの例

ススタブを生成する。アクセススタブのコードは図7のようになっており、サービス要求から得た情報をアクセススタブ内に保持する。クライアントに引き渡されてからは、これらの情報を元に適切に振舞う。

クライアントは、図8のようなコーディングをすれば、これを使える。

サーバによっては、提供するサービスの品質が複雑な制約を持つこともある (画素数を増やすと色数が減る画像サーバなど)。CORBA でこれを扱う場合、全体性能を表す指標を導入するか、さもなければ、ひとつのサーバから何種類かのオファーを提供することになるが、トレーダの実行効率を落すうえ、制約が連続量の場合など、有限個のオファーでは列挙できない場合もある。このような場合、ブローカのように自ら計算能力のあるオブジェクトを送り込み、サービス要求から直接アクセスさせるのが効果的である。

```
MyServiceRequest req=>New();
Trader trader = getTrader();
SearchEngineAccessStub stub
  = trader->RequestService(req);
if (stub != 0) {
  ...
  html = stub->Search ("Trading Service");
  ...
}
```

図 8: クライアントのコーディング例

5 関連事例

Orbix[3] では、スマートプロクシという仕組みを実現している。IDL で生成されたクライアントスタブを、サーバプログラマが自由に継承し、独自のスタブをプログラミングできる。これにより、OZ++ のアクセススタブと同様に、データの一部をクライアント側にキャッシュしたり、故障時にサーバを変更したりできる。しかし、OZ++ と異なり、スタブをクライアントで生成するので、スタブの動作にトレーダに送ったサービス要求を反映するのは難しい。スマートプロクシの機構を活かすためには、スタブの生成のタイミングを、マッチメイキングの時点まで早める必要がある。

6 まとめと今後の課題

オブジェクトを安全に交換できる特徴を持つ分散環境下では、CORBA のトレーディングサービスのフレームワークをより柔軟かつ強力に構築できる。特に、以下のような利点について述べた。

- トレーダ内部でクライアントとサーバが直接対話することにより、よりアプリケーションに特化したマッチメインキングが可能になった。
- サービスオファーをオブジェクト化したことにより、他のサーバに対する二次的なサービスを提供できるようになった
- 知的なアクセススタブを実現できた。
  - 適切なクライアントインタフェースの提供
  - サーバデータの一部のキャッシュによる通信コストの低減
  - 故障時の自動的な複製への切替え

本研究には、以下の課題が残っている。

```
class MyServiceRequest
        : SearchEngineServiceRequest {
  // ユーザの要求を表現したオブジェクト
  ...
  int CaseInsensitive(); {return 1;}
  int WholeText() {return 1;}
  Set<String> NewsGroups() {
    Set<String> set=>New ();
    return s.Add("fj.lang.oz++");
  }

  int IsSatisfiedBy(Broker b) {
    return b->CaseInsensitive() &&
           b->WholeText() &&
           b->NewsGroups()->Match("fj.lang.oz++");
    // 条件を満足しているかどうか調べる
  }
  ...
}

class SearchEngineServiceRequest
        : ServiceRequest{
// 親クラスの定義
  constructor: New;
  public: MakeMatch; // overridden
  public: CaseInsensitive,WholeText,NewsGroups,...;
  ...

  int CaseInsensitive() {return 0;}
  int WholeText() {return 0;}
  int KeyWord() {return 0;}
  Set<String> NewsGroups() {return 0;}
  int IsSatisfiedBy(Broker) {return 1;}
  int IsAcceptable(Broker,Broker) {return 1;}
  // ブローカはこれらのメソッドを用いて要求内容を知る

  AccessStub MakeMatch(Set <Broker> brokers) {
    Broker b;
    Broker currentBest;
    while ((b = brokers->RemoveAny()) != 0) {
      if (IsSatisfiedBy(b)) {
        // 最初に制約を満足したブローカを採用する
        currentBest = b;
        break;
      } else if (IsAcceptable(b,currentBest)) {
        // ひとつもなかった場合は最適のものを選ぶ
        currentBest = b;
      }
    }
    return currentBest->GetAccessStub(self);
  }
}
```

図 9: サービス要求のコーディングサンプル

6.1 インターワーキング

複数のトレーダが協調してトレーディング機能を実現することをインターワーキングと呼ぶ [4]。インターワーキングは、トレーダのスケーラビリティの問題を解消する重要な技術である。

インターワーキングは、単にトレーダどうしが協調すればよいのではない。他のトレーダに登録されているサービスを受ける場合、そのサーバでは異なる組織的 / 技術的ポリシーが採用されている可能性があり、ポリシーの変換が必要になる。ここでも、オブジェクトを送受信できることによるメリットがあると考えられ、詳細を検討している。

謝辞

本研究を通じて熱心な討論をいただいている当プロジェクトのメンバー諸氏に感謝する。

この研究は情報処理振興事業協会 (IPA) が実施している「開放型基盤ソフトウェア研究開発評価事業」の一環として行われたものである。

参考文献

[1] Deschrevel, J-P.: "**The ANSA Model for Trading and Federation**" *ANSA Architecture Report* APM.1005.01, July 15th, 1993.

[2] OMG: "**Trading Object Service**" OMG RFP 5 Submission, *OMG Document* orbos/96-05-06, version 1.0.0, May 10th, 1996.

[3] IONA Technologies: "**The Orbix Architecture**" IONA Technologies, Ltd. January 1995.

[4] Vogel, A., et.al.: "**Enabling Interworking of Traders**" Raymond, K. and Amstrong E. (Ed) *Open Distributed Processing* III, pp. 185–196, Chapman & Hall, London, 1995.

[5] 塚本他: "クラスの共有と配送に基づくオブジェクト指向分散システムの設計と実現" 情報処理学会論文誌, May 1996.

[6] 大西他: "オブジェクト指向分散環境 OZ++ におけるインターネットセキュリティ" マルチメディア通信と分散処理シンポジウム, October 1995.

[7] 大西他: "オブジェクト指向分散環境 OZ++ のトレーディングディレクトリの設計" 情報処理学会第 51 回全国大会, 7L-04, September 1995.