位置透過性をもつコンポーネント構成手法とフレームワーク

鈴木 正人

国立情報学研究所 情報学基礎研究系

コンポーネントは再利用可能で信頼性の高いソフトウェア部品として様々なアプリケーション開発で利用されているが、現状で提供されているほとんどのコンポーネントが単一ホストで動作することを前提として設計されているため分散環境でのアプリケーション開発にはそのままでは利用できない。しかしながら、EJB や DCOM など分散環境用として提供されているコンポーネントは、規模が大きく使用方法も複雑であり、必ずしも全ての分散アプリケーションの開発に必要だとはいえない。本稿では分散非対応のコンポーネントを構成するオブジェクトに位置透過性を持たせるための機構をフレームワークとして実現することで、より多くのコンポーネントを分散環境でのアプリケーション作成に応用する手法を述べる。

A Method and Framework for building location transparent components

Masato Suzuki

Imformaion Science Research Group, National Institute of Informatics

Componentwares provide high reusability and reliability, then they are now widely applied for non-distributed software development. Distributed components such as EJB and DCOM are so complex that are not suitable for all distributed applications. In this paper we propose a mechanism and a framework which allow us to use many of non-distributed components on distibuted environments by providing location transparent connection among them. As the consequence, we can use many of these components for distributed software developments.

1. はじめに

ハードウェアの進化にともなう PC の低価格化とインターネットに代表される通信環境の普及は計算機環境 (プラットフォーム) の多様化と広域化などをもたらしている。このような環境で動作するソフトウェアには、従来のソフトウェアよりも大規模で、複雑で、多様な要求を満たし、かつ高品質なものが必要とされている。

このような要求を満たすための新しいアプローチとしてコンポーネント技術が注目を集めている。開発者は高度に抽象化されたソフトウェア部品 (コンポーネント) を組み合わせることで記述・作成することが可能になる[1)]。コンポーネント技術を利用した開発方法論 CBSE(Component Based Software Engineering) はこれを支援するものであるが、現状では部品であるコンポーネントはすべて単一のホストで動作することを前提として設計実装されており、複数のホストに配置したコンポーネントを連携して動作するようなソフトウェアの開発に十分な効果を与えてはいない。また従来のソフトウェアを再利用性、拡張性、信頼性を向上させるためにコンポーネントを利用して再構成することが行われるが対象を分散環境で動作させるという要求から、従来のコンポーネントを分散環境でも使用するための技術、すなわち物理的位置に依存せずにそれらの接続や統合を行う技術が必要である。

本稿では、単一ホストで動作することを前提として設計されたコンポーネントを接続してアプリケーションを構成する際に、各コンポーネントに位置透過性を与え、複数のホストで動作可能にするための手順と機構を提唱する。まずコンポーネントのソースコードを解析して接続の形態を調査し、必要に応じて遠隔手続き呼び出しのコードを含む部分 (リモートアダプタ) を自動生成する機構を提唱する。コンポーネントとリモートアダプタをセットとして扱うことで、既存のコンポーネントを分散環境で動作するように拡張する。次に接続に関する詳細を抽象化する GIM(General Interaction Model) を導入し、位置透過性の実現部分を

フレームワーク化する。これによりコンポーネント間の情報の授受はすべて Interactor および Port によって隠蔽され、アプリケーションが直接記述する手間を軽減できる。

2. コンポーネントと関連技術

2.1 コンポーネントアーキテクチャ

コンポーネントには様々な定義があるが、本稿では実行環境に依存しない、入れ替え (プラグイン) 可能なソフトウェア部品の単位をコンポーネントと呼ぶことにする[3]。コンポーネントは**インターフェース**をもち、他のコンポーネントとの情報の受け渡しを行う。これをコンポーネントの**接続**と呼ぶ。接続を行うには受け渡しする情報の形式を定義し、必要に応じて変換を行う機構が必要である。これを**コンポーネントアーキテクチャ**と呼ぶ。コンポーネントアーキテクチャの代表例として JavaBeans がある。これはコンポーネントが使用する変数 (プロパティ)、同期型通信 (メソッド)、非同期型通信 (イベントとリスナ) の方法と制約を定義しておりこの環境上で動作するすべてのコンポーネント (Bean) はこの規則に従って作成される。例えば `Foo` という名前のプロパティを操作するメソッド名は `getFoo` および `setFoo` という名前をもち、`BaaEvent` という名前のイベントを受理可能なメソッドは `BaaEventListener` という名前でなければならない。コンポーネント、コンポーネントアーキテクチャの関係とコンポーネント接続の例を図 1 に示す。

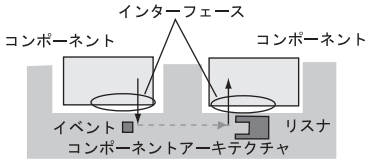

図 1 コンポーネントの接続

2.2 分散ソフトウェア開発

分散環境で動作するソフトウェアは、以下の 2 つの条件を満たす必要がある。

(1) オブジェクトの位置透過性
(2) オブジェクトの参照透過性

(1) はオブジェクトが別のホストで動作する場合でもその意味は変更されないことを保証する。一般には複数のホスト上に共通の名前空間を定義し、物理的位置に依存しない名前 (グローバル名) と特定のホスト上での固有名 (ローカル名) の変換を行う仕組みにより実現される。(2) は参照先のオブジェクトに依存せずにプログラムを記述し、遠隔ホストに参照先が存在する場合には自動的に遠隔呼び出しを行って結果を返す仕組みである。

代表的な分散環境である Java RMI を例にすると、(1) を実現する機構は**ネーミングサービス**、(2) を実現する機構は**スタブ**と**スケルトン**と呼ばれる (図 2)。ネーミングサービスを行うオブジェクトは別プロセス (`rmiregistry`) として動作しており、すべてのオブジェクトは生成時に名前を登録する。オブジェクトの中で遠隔呼び出しを必要とするメソッドを集めてインターフェースを定義し、RMI コンパイラ (`rmic`) に入力すると、サーバオブジェクトの代理として動作するクライアント側オブジェクト (=スタブ) と、クライアントの代理として動作するサーバ側のオブジェクト (スケルトン) が自動生成される。

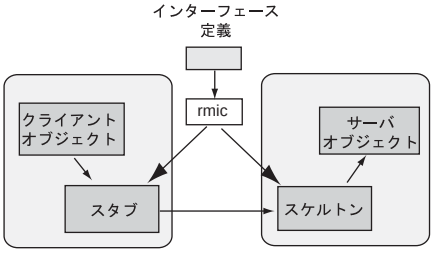

図 2 分散環境上のオブジェクトの通信

2.3 分散環境でのコンポーネント接続の問題点

分散環境で動作するソフトウェアを開発する際に現状のコンポーネント技術を使用するには、以下の 2 つの問題点がある。

(1) 接続 (通信) のためのコードの記述が複雑である
(2) 初期化やオブジェクトの配置の記述が複雑である

(1) に関しては、現状のコンポーネントの接続ではメソッド呼び出しの場合もイベントによる接続の場合も、すべて接続元と接続先が同一のホスト (=名前空間) に存在することが前提となっている。これを分散環境で動作するようにするためには、位置透過性実現のためのネームサーバの呼び出しや参照透過性実現のためのスタブやスケルトンに相当するコードを記述しなければならず、コードが複雑化し、コスト上昇および信頼性の低下の要因となる。

さらにコンポーネントの接続に関しては、双方のインターフェースの違いを吸収するための機構 (コネクタ) が必要となる場合が多い。これは送り手と受け手の間でデータ型や順序の変換を行うものであるが、これらも同一ホスト上でないとうまく動作しない場合が

考えられる。そのような場合、分散環境でも動作するようなコネクタを開発しなければならず、これもコスト上昇要因となる。

(2) は分散環境で動作するプログラムは起動に到るまでの過程が複雑である。最初にオブジェクトの配置を決定し、オブジェクトの初期化を行う。現状のコンポーネントを利用した開発では、オブジェクトの初期化は各コンポーネントを呼び出すプログラム (=メインプログラム) に任されているが、この部分は個別に記述しなければならない。オブジェクトの初期化には定型的な記述も多く、自動化できればコストの削減に効果が期待できる。

3. 位置透過性の実現

これらの問題を解決するために、本稿では以下のようなアプローチを採用する。

(1) コンポーネントを位置透過にするための機構 (アダプタ) の自動生成

(2) 実際に接続を可能にする実行環境 (ランタイム) の提供

(1) はコンポーネント接続に際し、固有なオブジェクトやデータを単一の名前空間上に写像するためのコードを含み、既存のコンポーネントのソースコードが与えられれば自動生成が可能である。なお現状ではコンポーネントにアダプタを適用する際に、そのコンポーネントのソースコードの一部を変更する必要がある☆。

(2) はアダプタが付加されたコンポーネントオブジェクトを初期化する際にネームサーバに登録するためのコードの自動生成と、実行時にオブジェクトの作成消滅に連動してネームサーバを管理するための実行時コード (ランタイム) から構成される。なお現状ではコンポーネントアーキテクチャとしては JavaBeans を前提としているが、スタブやスケルトンに相当する部分の記法が異なるだけで本手法自身はコンポーネントアーキテクチャには非依存である。

本手法によってコンポーネントにより構成されたソフトウェアを分散環境で動作させるまでの手順を図 3 に示す。

3.1 アダプタの作成

アダプタを作成するには、まずコンポーネント相互の接続を示す情報が必要である。これを接続情報と呼び、以下の内容を含む☆☆。

- クラス名

☆ これを避ける方法は 6 節で考察する

☆☆ 実際には XML で記述されている

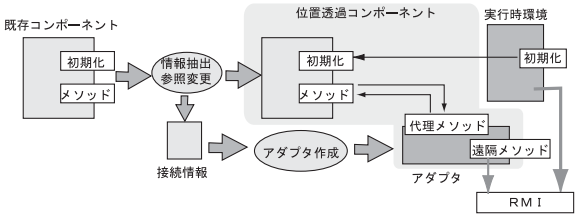

図 3 位置透過性実現手順

- そのクラスが呼び出しているメソッドのリスト
- そのクラスが実装するメソッドの名前とシグネチャ

接続情報はコンポーネントのソースコードのうち、他のコンポーネントに含まれるオブジェクトをレシーバとするメソッド呼び出しを抽出することで作成できる。ただしこのままでは不要な (=透過にする必要のない) メソッドまで抽出される可能性があるため、現在はアダプタ作成前に接続情報を手作業で編集することで不要なメソッドの作成を抑制している。

アダプタは接続情報から以下のように作成される。以降呼び出し側を Caller, 呼び出される側を Callee とする。`X` はクラス名、`Y` はメソッド名である。

- 位置透過の対象とする非分散コンポーネントをプロパティ `body` として持つ
- `invoke_X`、`accept_X` の 2 種類のインターフェースを持つ
- `invoke_X` はクラス `X` が参照しているメソッドに対応した遠隔メソッド (`Remote_Y`) を含む
- `accept_X` はクラス `X` が実装しているメソッドに対応した代理メソッド (`Proxy_Y`) を含む
- 戻り値を格納する変数を Callee 側のアダプタクラスにおいてプロパティとして用意する

アダプタを使用するために、コンポーネントのソースコードのうち、呼び出しに関連する部分を以下のように間接呼び出しに変更する。

- メソッド呼び出しを全てアダプタ内のインターフェース `Invoke` 内の対応する遠隔メソッド呼び出しに変換
- 戻り値を返す部分を全てアダプタ内の対応する `getResult` の呼び出しに変換

(ホスト H1 に存在する) コンポーネント C1 中のメソッド f1 が、(ホスト H2 に存在する) コンポーネント C2 中のメソッド f2 を呼び出す場合を例に説明する。与えられた C1,C2 のコードを図 4 に示す。この例では対象となる呼び出しは C1::f1 から C2::f2 の 1 つのみであり、シグネチャは `String` → `String` である。(キーワード `public` および例外処理は全て省略してある)

```
[コンポーネント 1]
class C1 {
   // プロパティ, イベント省略

   void f1() { // メソッド
     String s1 = ... // 引数用意
     String s2 = c2.f2(s1); // 呼出し
      ...
   }
};

[コンポーネント 2]
class C2 {
   // プロパティ, イベント省略

   String f2(String s) {
     String news = new String(s);
      ...
     return news; // 戻り値
   }
};
```

図 4 与えられたコンポーネント (一部)

生成されたアダプタのコードを図 5 に示す。C1,C2 に接続するアダプタのクラスをそれぞれ RA1,RA2 とする。RA1,RA2 は位置透過なオブジェクトであるため、`RemoteObject` を継承する必要がある。この例では C1 は Caller, C2 は Callee としての役割しか持たないため RA1 は `invoke` のみ、RA2 は `accept` のみを実装しているが、あるクラスが Caller にも Callee にもなりうる場合はアダプタには両方のインターフェースを実装する必要がある。

アダプタを付加するために修正した後のコンポーネント C1 のコードを図 6 に示す。`ra` は各コンポーネントに付随しているアダプタである。この変換は容易なためアルゴリズム等は省略する。

この例では C1 が C2 内のメソッドを直接呼び出していたが、イベントを使用した間接的な呼び出しの場合、以下の点を新たに追加する必要がある。その他の手順は同じである。

- 各イベントを保持するリモートオブジェクトを作成する。その際に情報を取り出す部分をすべてメソッドとして実装する
- イベントリスナ内でのイベントからの情報取り出しに相当する部分をリモートオブジェクトのメソッド呼び出しに変換する

実際の動作には RA1,RA2 を生成する部分を作成してアプリケーションプログラムに付加する必要がある。コードを図 7 に示す。ネームサーバへの登録の際の引数`"RA1","RA2"`は、実際にはインスタンス毎に固有の名前に置換される。

4. フレームワーク化

これまでに述べた手法で非分散なコンポーネントに

```
interface invoke_C1 {
  void Remote_f2(...);
  Object getResult_f2(...);
}

class RA1 extends RemoteObject
     implements invoke_C1 {
  C1 body;
  RA1() {
    C1 body = new C1();
  }

  void Remote_f2(String s1) {
    RA2 ra2;
    ra2 = (RA2)Naming.lookup
         ("rmi::://H2/.../RA2");
    ra2.Proxy_f2(String s1);
  }

  Object getResult() {
    RA2 ra2;
    ra2 = (RA2)Naming.lookup
         ("rmi::://H2/.../RA2");
    return ra2.getResult_f2();
  }
}

interface accept_C2 {
  void Proxy_f2(...);
}

class RA2 extends RemoteObject
    implements accept_C2 {
  C2 body;
  String result_f2;

  RA2() {
    C2 body = new C2();
  }

  void Proxy_f2(String s);
    String news = body.f2(s);
    result_f2 = news;
  }

  void getResult_f2() {...}
}
```

図 5 生成されたアダプタ

```
class C1 {
   RA1 ra;
   void f1() {
     String s1 = ...;
     ra.Remote_f2(s1);
     String s2 = (String)ra.getResult();
      ...
   }
};
```

図 6 変換後のコード

位置透過性を与え、分散環境で動作するソフトウェアの構成に利用することが可能になった。しかし接続の際に

さらに現在のアダプタは、実行時環境により初期化時にすべてのオブジェクトが作成され登録されることを前提としている。サーバを構成するオブジェクトの中にはクライアントからの要求があって初めて作成されるものも多いが、動的なオブジェクト生成を実現す

```
[H1 で実行]
class AppClient {
  static RA1 ra1;

  AppClient(){ // 初期化
     ra1 = new RA1();
     Naming.bind("RA1",ra1);
  }
};

[H2 で実行]
class AppServer {
  static RA2 ra2;

  AppServer(){ // 初期化
     ra2 = new RA2();
     Naming.bind("RA2",ra2);
  }
};
```

図 7 実行時環境

るにはオブジェクトの生成を監視するプロセス☆との交信が必要になり、そのためのコードをアダプタ内に記述する必要がある。また現在のアダプタは作成され登録されたオブジェクト (=コンポーネントの実体) が実行中に別のホストに移動することを考慮していない。

これらの問題を解決するにあたって、コンポーネントの接続の際に従来の直接的な記述に基づいて対応するアダプタをそれぞれ作成するよりは、接続という概念を抽象化してとらえ、必要に応じて型変換や分散化などの機構を付加するような構造にした方が見通しがよくなる。本節では、コンポーネントの接続を抽象化したアーキテクチャ(General Interaction Model, **以下** GIM)[4] を導入することで、コンポーネントの分散化に必要なクラス群をフレームワークにより実現する。なおフレームワークと呼ばれるものにはさまざまな定義があるが、本稿では「相互に関連し、問題の解決に有用な、再利用可能なクラスの集合」[5] の意味でフレームワークという用語を用いることにする。一般にフレームワークはアーキテクチャを決定する不変部 (cold spot) と、問題に応じて (その多くは継承により) カスタマイズ可能な可変部 (hot spot) から構成される。

GIM アーキテクチャの元では、コンポーネント同士を直接接続する代わりに、仲介コンポーネント (GIM コンポーネント) を導入する。接続の詳細はポートのプロパティとして記述される。GIM コンポーネントの中心となるクラスは Interactor と呼ばれる。これは Caller を抽象化したクラス EventFirer と Callee を抽象化したクラス ActionPerformer を接続する。複雑な接続では、EventFirer, ActionPerformer 共に複数存在することもある。各コンポーネントは**ポート**を通じてイベントやアクションを送受信する。通信の種類 (同期/非同期, 単一/同報など) は接続するポートの数および属性として表現できる。

図 4 に対応する呼び出しは EventFirer、ActionPerformer 共に 1 つずつもち、両者を直接接続する Interactor により表現される。これを図 8 に示す。

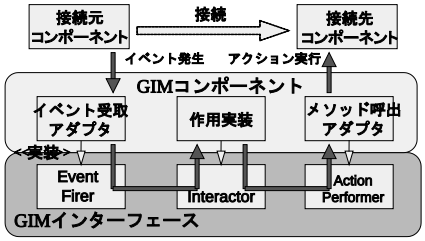

図 8 GIM アーキテクチャ[4]

GIM アーキテクチャでは、すべての接続は内部イベントの発生 (`raiseEvent`) と通知 (`notify`) により実現される。フレームワークの構成を図 9 に示す。インターフェース `Invoke` に含まれる `ra.Remote_Y()` の呼び出しに対応する個数の EventFirer オブジェクトと、インターフェース `Accept` に含まれる `ra.Proxy_Y()` の呼び出しに対応する個数の ActionPerformer オブジェクトが作成される。

`ra.Remote_Y` が呼ばれると `Firer_Y()` は対応する内部イベントを作成し、`ra.Proxy_Y()` が呼ばれた時点で内部イベントは `Performer_Y` に渡される。`ra.Remote_Y` に記述されるべき処理を対応する `Firer_Y` のメソッド `raiseEvent` として、`ra.Proxy_Y` に記述されるべき処理を対応する `Performer_Y` のメソッド `doAction` として実装することにより、図 5 と等価な処理が実現できる。

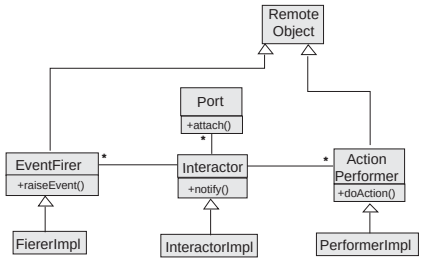

図 9 GIM を利用したフレームワーク

実際には利用者が `Remote_Y`,`Proxy_Y` に相当するコードを記述する必要はなく、例えば図 4 のように 1 対 1 で接続する場合には、フレームワークが提供する以下の Interactor を採用すればよい。

☆ Java RMI の場合は rmid

```
class OnetoOneInteractor
        extends Interactor {
// プロパティ firer, performer を継承

  void init() { ... }

  void notify() {
    Port  f = firer.getPort();
    Event e = f.getEvent();
    Port  p = performer.getPort();
    if (f.getHost() == p.getHost()) {
       performer.doAction(e) // ローカル
    } else {
      String s = "rmi://"+p.getHost()
            + "..." + p.getName();
      Performer rp = (Performer)
         Naming.lookup(s);
      rp.doAction(e); // リモート
    }
  }
}
```

図 10 Interactor の実装

アプリケーションの初期化の際に行うべき処理は以下のとおりである。

(1) 必要な Firer, Performer, Interactor を作成する。
(2) 必要な数の Port を作成し、Interactor と各コンポーネントを接続する (`attach` を使用)
(3) Port の属性を設定する

5. 記述例:分散モニタ

分散コンポーネントを利用したアプリケーションとして複数のホストで動作している計算の状態を監視するモニタを考える。監視対象は時間のかかる計算を並列に行っているものとし、以下のインターフェースを持つ。

- `void StartCompute()` : 計算を実行する
- `void StopCompute()` : 計算を中断する
- `Status Progress()` : 現在の計算状態を報告する

モニタはアプリケーションの実行の中断/再開を制御するコンポーネントとアプリケーションの状態を表示するコンポーネントから構成される。制御コンポーネントは以下のインターフェースを持つ。

- `Status Snap(i)` : i 番目の計算の計算状態を取得する。
- `Status Suspend(i)` : i 番目の計算の計算状態を取得し、中断する
- `Reset(i)` :i 番目の計算を最初からやり直す

分散モニタで使用する Interactor は制御コンポーネントと n 個の計算コンポーネントを接続する。ポート `Snap` からイベントを受理した場合は、イベントのデータ部に格納された i に対応する計算コンポーネントの計算を中断、計算状態を取得した上で実行を継続する。

```
[H1 で実行]
import gim.*;
class C1 extends Component {
  Firer f;

  void init() {
    f = new Firer();
  }

  void f1() {
    String s1 = ...;
    Event e = new Event();
    e.setValue((Object)s1);
  }
}

[H2] で実行
import gim.*;
class C2 extends Component {
  Performer p;

  void init() {
    p = new Performer();
  }

  void f2(Event e) {
    s2 = (String)e.getValue();
  }
}

[メインプログラム]
import gim.*;
class App {
  Component c1,c2;
  Port p1,p2;
  Interactor ir;

  App() { // コンストラクタ
     p1 = new Port();
     p2 = new Port();
     ir = new OnetoOneInteractor();

     p1.attachFirer(ir,c1);
     p2.attachPerformer(ir,c2);
     ir.init();
  }
}
```

図 11 フレームワークの利用

ポート `Reset` からイベントを受理した場合は、対応する計算コンポーネントの計算状態を初期化する。`Snap` に接続される Interactor, Performer, メインプログラムの実装を図 12,13,14 に示す。

6. 考　　察

分散コンポーネントを実現する方法としては、EJB[6],DCOM[7],CORBA Components[8] が知られている。しかしながら、これらのコンポーネントアーキテクチャではコンポーネントの接続の際に Caller は Callee のインターフェースを明示的に import しなければならず、コードが複雑化する。GIM アーキテクチャと分散フレームワークは、接続を抽象的に表現し明示的な import/export を避けることで、接続対象となるコンポーネントやインターフェースの内容、その間で受け渡されるデータの構造が変更された場合でも、コード

```
class MonitoringInteractor
  extends Interactor {
// プロパティ firer, performer を継承

  void notify() {
    Port  f = firer.getPort();
    Event e = f.getEvent();
    Integer i = e.getValue();

    Port  p = performer[i].getPort();
    String s = "rmi://"+p.getHost()
        + "..." + p.getName();
    Performer rp = (Performer)
       Naming.lookup(s);
    rp.doAction(e);
  }
}
```

図 12 分散モニタ用の Interactor

```
class SnapPerformer
  extends ActionPerformer {

   void doAction(e) {
     stopCompute();
     Status st = Progress();
     startCompute();
     e.setValue((Object)st);
   }
}
```

図 13 計算コンポーネントに接続する Performer

```
class MonitorApp
    extends Application {

  ComputeComponent cc[n];
  MonitorComponent mc;
  Port control;
  Port snap[n];
  Interactor mr;

  MonitorApp() {
     ir = new MonitoringInteractor();
     control = new Port();
     control.attachFirer(ir,mc);
     for (i=0;i<n;i++) {
       snap[i] = new Port();
       snap[i].attachPerformer(ir,cc[i]);
     }
     ir.init();
  }
}
```

図 14 アプリケーション本体

の変更を最小限にすることが可能である。

GIM の問題点としては、現状ではコンポーネントのソースコードを必要とすること、Java 言語に実装が限定されていることがあげられる。ソースコードに関しては、現在は接続情報を抽出するために利用しているがこれを別に与えることである程度の抽象化を実現できる可能性がある。コンポーネントの接続の際には、双方のインターフェースでデータの型 (シグネチャ) が一致していたとしても、実際に接続すると正しく動作しない例が存在することが指摘されている。これを解決するには接続の際にデータ型のみでなくそれが属するドメインのなどの情報が必要になるが、その際に分散化に必要な情報を付加する方法を検討中である。

ソースコードが必要なもうひとつの理由は、他のコンポーネントのメソッド呼び出しをアダプタあるいは Firer,Performer を使用した間接呼び出しに書き換えることで、通信の送り手, 受け手, 時刻, サイズなどの必要な情報を得ているためである。ソースコードを変更せず、実行時にこの情報を得るためには ORB の間を流れる低レベルの通信を監視する必要があるが、コンポーネントのレベルでの値やイベントがどのように ORB で表現されるかは ORB の実装に依存する。プロファイリングや負荷分散を目的として ORB を拡張するためのインターフェースがいくつか提案されているが、GIM が提供する抽象度の高い記述をこれらを用いて実装するのは困難である。

現状では Java RMI による実装を考慮しているため Java 言語のみの対応であるが、フレームワークの機能は言語に依存しない。言語依存部は主にデータのイベントへの隠蔽とそれを取り出す部分であり、IDL のようなインターフェース記述言語を利用することでこの部分はある程度自動生成が可能である。

7. おわりに

現在提供されているコンポーネントの多くは単一ホストで動作することが前提となっており、分散環境で動作するプログラムの開発にはそのままでは利用できない。本稿ではコンポーネントに位置透過性を持たせるフレームワークを提案し、元のコンポーネントをソースコードの変更を最小限にしたまま、分散環境での開発に利用する方法を示した。

現在は Firer と Performer は 1 つの Interactor を通じて直接接続されており、複数の Interactor を利用した接続は考慮されていない。これは Interactor と接続可能なポートが Firer か Performer のどちらかに制限されているのが理由である。ここに別の Interactor を接続可能にすることで Interactor の入れ子構造が実現できる。分散モニタの例では、1 つの Firer と 2 つの Performer をもち、Firer が発火する度に Performer で定義された Action を交互に実行する Interactor を実装し、これを MonitorInteractor と直列に接続することでモニタの記述が簡潔になる。また Interactor 同士を接続可能にすることで、より高度な機能を持った Interactor を定義し再利用性することが可能になる。型変換やデータの順序変更を行うアダプタを接続可能にする方法を含めて今後の課題である。

参 考 文 献

1) M. Shaw, D. Garlan: “Software Architechture: Perspectives on an Emerging Dicipline,” Prentics Hall, 1996.
2) Shaw, Garlan, Allen, Klein, Ockerbloom, Scott, Schumacher: “Candidate Model Problems in Software Architecture,” http://www.cs.cmu.edu/afs/cs/project/compose/www/html/modprob
3) J. Q. Ning: “A Component-Based Software Development Model”, Proceedings of COMPSAC 96, August 1996.
4) 當銘 直告, 鈴木 正人: “ソフトウェアコンポーネント間の作用をカプセル化する機構 GIM の提案,” 日本ソフトウェア科学会 第 17 回 大会論文集 D4-3, 2000.
5) M. E. Faid, D. Schmidt, R. Johnson: “Building Application Frameworks,” Wiley 1999.
6) Sun Microsystems: “Enterprise JavaBeans,” http://java.sun.com/products/ejb/docs.html, 1999.
7) M. Horstmann, M. Kirtland: “Distributed Component Object Model Architecture,” http://msdn.microsoft.com/library/en-us/dndcom/html/msdn_dcomarch.htm, 1997.
8) Object Management Groups: “Corba Components,” http://www.omg.org/, Document orbos/99-07-01, 1999.
9) P. Herzum, O. Sims, 長瀬嘉秀 監訳: “ビジネスコンポーネントファクトリ– エンタープライズ領域でのコンポーネント指向開発 –,”, 翔泳社, 2001.