異種分散コンポーネントの連係を支援する機構の設計と実装

2006 年度

名倉 正剛

論文要旨

近年，ネットワークを介して通信を行うアプリケーションを開発する際に，分散コンポーネント技術の利用が増大している．EJB (Enterprise JavaBeans)，CORBA (Common Object Request Broker Architecture)，Web Service などの分散コンポーネント技術を利用して構築されたアプリケーションプログラムは，サービスを提供するサーバプログラムと，ネットワークを介してそれを利用するクライアントプログラムによって構成される． そして特定の分散コンポーネント技術を利用することで，異なるサービスの呼出しを同一の手順で行うことができる．このため，既存の複数のサービスを呼び出し，それらをソフトウェア部品として組み合わせるアプリケーションを，容易に開発できる．しかし，多様なサービスを組み合わせて動作させるためには，まだ解決しなければならない課題がある．

まず，さまざまなベンダによって独自に開発されて公開されているサービスを組み合わせる場合，各コンポーネントの利用する分散コンポーネント技術が異なる場合がある．そのため，それらを呼び出す際の異種性を吸収する必要がある．また一般的に，各コンポーネントの提供者や，それらを実行するコンピュータは異なっている．そのため，アプリケーションプログラムの実行時に，アプリケーションプログラムを実行するコンピュータの管理者が管理できないリソースを利用する．このことは，分散コンポーネントを組み合わせるアプリケーションの信頼性を大きく下げる．さらに，モバイルや組込み機器などのさまざまな形態の端末で運用されているサービスを，実行時に柔軟に利用できることが望ましい．そのためには，分散コンポーネントが動作する機器をネットワークに接続するだけで，ネットワーク上のその他の機器と自動的に連係するという，いわゆる “Plug and Play” を実現する必要がある．

本研究ではこれらの課題を解決し，異種分散コンポーネントの連係を支援するために，“HeteroSOC フレームワーク” を提案する．まず，組み合わせるコンポーネントを開発時に静的に決めることができる場合の課題を解決するために，開発支援システムを提供する．これを利用し，開発者がサービスの組合わせに関するワークフロー情報を記述することによって，この開発支援システムは異種分散コンポーネントを組み合わせるアプリケーションを自動的に生成する，そして信頼性の課題を解決するため，コンポーネントの呼出しに障害が発生した際に，開発時に指定した別のコンポーネントに処理を切り替えられるような仕組みを用意する．次に Plug and Play を実現するために，生成したアプリケーションを実行するプラットフォームを提供する．このプラットフォームは，ローカルネットワークに存在するコンポーネントを実行時に探索し，発見したコンポーネントの異種性を動的に吸収して呼び出すことを可能にする．

HeteroSOC フレームワークによって開発したアプリケーションでは，コンポーネント間の異種性の吸収のために，コンポーネントの呼出し時間が約 30 ms 増加した．また実行時にコンポーネントを動的に探索して利用する場合は，必要なコンポーネントを探索するために，コンポーネントの呼出し時間が約 330 ms 増加した．なお後者は，パラメータの調節により短縮できる．

Abstract

Distributed component technologies are widely used for developing applications that communicate over the network. An application program using such a technology, e.g., EJB (Enterprise JavaBeans), CORBA (Common Object Request Broker Architecture) and Web Service, consists of a server program that provides a service and a client program that uses the service via a network. When using a single component technology, different services can be invoked with the same calling procedure. This enables application development using many components executing on different component servers. Critical issues however remain when orchestrating many components.

First, when combining services provided by different vendors, each service may use a different component technology. Differences between them must be absorbed to enable heterogeneous component invocations. And, when combining distributed components provided by third-parties, each component normally runs on a different computer. Thus, at application runtime, the application uses many resources which is not managed by the administrator of the computer executing the main application itself. This significantly decreases the reliability of the application. Second, callee components are often statically fixed at development time. But services may execute on many types of terminals (such as mobile and embedded appliance), so there are cases where callee components should be flexibly and dynamically fixed at runtime. In other words, distributed components need to be executed through "Plug and Play", so they can automatically work with other components by just connecting the terminal that provides them to the network.

We propose "HeteroSOC framework" to solve these issues, and to support the orchestration of heterogeneous distributed components. In our framework, a development support system which generates program code to absorb the heterogeneity between components. Developers first describe workflow information for statically combining services. The system automatically generates application program codes which combine heterogeneous components. Application reliability is improved through a function that switch processes at runtime to other components specified at development time. Plug and Play is realized by a platform on which generated applications are executed. This platform can discover specified components at runtime and can dynamically absorb the heterogeneity between discovered components at runtime.

Applications developed with the HeteroSOC framework incur about 30 msec overhead for component processing time for absorbing the heterogeneity between components. When applications discover components at runtime, there is an overhead of about 330 msec for discovering specified components. Note that this overhead can be shortened by changing parameter values.

目 次

図 目 次

表 目 次

第1章　序 論

本研究では，異種分散コンポーネントの連係に着目する．本章では，まず分散コンポーネント技術とそれを利用したサービスについて述べる．そしてサービスの連係とそれを構成するコンポーネントの連係についての課題の概要を示し，提案の概略を述べる．

1.1　分散コンポーネント技術とサービス指向アーキテクチャ

オブジェクト指向プログラミングの普及により，ソフトウェアコンポーネントを組み合わせることによるソフトウェア開発が増大している．ソフトウェアコンポーネントは自己完結型のソフトウェアモジュールであり，これを利用することにより，ソフトウェアに対してモジュールの差し替え，修正，再利用をより簡単に行うことができる．この結果，ソフトウェア部品を組み合わせてシステムを開発することができ，ソフトウェアの生産性と品質の向上が期待できる [Aoyama1998].

近年では，インターネットの発達に伴い，分散コンポーネント技術が普及している．これは，ネットワークを介して通信を行うソフトウェアに対してコンポーネント技術 [Brown1996] [Brown2000] [Orso2000] を適用したものである．EJB (Enterprise JavaBeans)，CORBA (Common Object Request Broker Architecture)，Web Service などの分散コンポーネント技術を利用して構築されたアプリケーションプログラムは，サーバプログラムとクライアントプログラムによって構成される．そして，サーバプログラムとしてコンポーネントのセットを実装し，外部から呼び出すことのできる「サービス」として公開する．W3C は，「サービス」を “提供者や利用者から見た際の，一連の処理の集合” と定義している [Booth2004]．そして外部から呼び出すことができ，そのためのインタフェースが定義され公開されていて発見され得るコンポーネントのセットのことや，それらを利用したアプリケーション構築のスタイルのことを，「サービス指向アーキテクチャ (SOA: Service-Oriented Architecture) 」という [Haas2004] [Gold2004] [Mahmoud2005]．サービスの提供者は，サービスを構成するコンポーネントをサーバコンピュータ上に配置し，それらのインタフェースに関する情報を公開する．サービスを利用するためには，まずコンポーネントのインタフェースに関する情報を取得する．そしてそのインタフェースを介して，サービスを構成するコンポーネントを利用するようにクライアントプログラムを開発する．そして利用者は，クライアントプログラムを実行することによって，ネットワークを介してコンポーネントを呼び出し，サービスの処理を実行する．

1.2　サービス指向コンピューティングによるサービスの連係

特定の分散コンポーネント技術を利用することで，コンポーネント呼出しの形式を統一してサービスを提供することができる．このため開発者は，サービスを呼び出すクライアントプログラムを開発する際に，どのような処理を呼び出すかを考えることに専念すれば良い．そして，個々のサービスごとにどのようなコンポーネント呼出しの形式で呼び出すかを考える必要はない．このように同一の手順で異なるサービスの呼出しを行うことができるため，既存の複数のサービスを呼び出し，それらを基本要素として組み合わせることによってアプリケーションを容易に開発できる．このようなコンピューティングパラダイムを，「サービス指向コンピューティング (SOC: Service Oriented Computing) 」と呼ぶ [Papazoglou2003] [Bichler2006]．サービスの組合わせは，サービスを構成するコンポーネントを実行した結果を別のサービスを構成するコンポーネントの入力に利用したり，それぞれのコンポーネントを実行した結果をまとめたりするプログラムコードを生成することで行われる．そして利用者は生成されたプログラムコードを介して，それぞれのサービスを連係させて利用する．

SOC を実現するための基盤技術としては主に Web Service 技術が利用されており，多くの要素技術が標準仕様として定義されている．例えば，サービスの検索，サービスを連係して動作させる際のワークフローの記述，その際のトランザクション管理，サービスのインタフェースの情報の記述，サービスを呼び出すためのメッセージングに関する仕様などが挙げられる [Turner2003]．それらの要素技術を利用することによって，サービス指向によるソフトウェア開発を実現できる [Brown2003]．そして，企業内で情報システムを連係する EAI (Enterprise Application Integration) や，企業間でアプリケーションを連係する B2B (Business-to-Business) を構築し，それらを連係することができる [Chung2004]．

1.3　サービス連係における課題と提案機構

SOA や SOC を実現するためには，1.2 節に挙げたもののみならず，多くの要素技術が提案されている．しかし，色々なコンポーネントを組み合わせて動作させるためには，まだ解決しなければならない課題が残されている．本節では，その中から本研究で取り扱う課題を挙げる．

まず，コンポーネントを利用したアプリケーション開発は，さまざまなベンダによって提供されるコンポーネントのサービスを組み合わせることによって行われる．しかしそれらは独自に開発されており，サービスの性能や質や実装は，プロジェクトチームにより異なる．通常のソフトウェア開発でもこのような状況が発生するが，コンポーネントを組み合わせる際にはより重大な影響を与える [Olsen2006]．アプリケーション開発の際は，コンポーネントの開発者や，その提供者や，実行するコンピュータが異っていることに起因し，各サービスが利用する分散コンポーネント技術が異なっていることを想定する必要がある．その場合は，それぞれのコンポーネントを呼び出す方法や，個々のコンポーネントが利用するデータモデルやコントロールモデルが異なる [Goedicke2000]．そのようなコンポーネントを組み合わせる場合は，開発者はどのような形式でコンポーネントを呼び出すか考慮せざるを得ない．その上で開発者は，コンポーネント間の異種性を吸収するための

プログラムコードを記述する必要がある．

またそれぞれのサービスが独自に管理されているため，全体の性能や信頼性が保証できない [Olsen2006]．そして各コンポーネントを実行するコンピュータが異なっていることに加え，通常はそれらのコンピュータが接続されるネットワークも異なっていることが多い．一般的に，アプリケーションプログラムを実行するコンピュータの管理者は，それぞれのコンポーネントを実行するコンピュータや，それらを呼び出す際に経由するネットワークなどのリソースを管理できない．そのため，それらに障害が発生した際に，アプリケーションプログラムを実行するコンピュータの管理者は対応できない．コンポーネントを組み合わせたアプリケーションの開発では，通常はどのコンポーネントを組み合わせるか，アプリケーションプログラムの設計時にすでに決定している．このため開発者が設計を行った時点では利用できたコンポーネントが，利用者がアプリケーションプログラムを実行しようとする時点では利用できないことがある．このことは，分散コンポーネントを組み合わせるアプリケーションの信頼性を大きく下げる．

さらに，例えばモバイルや組込み機器のような機器をサーバとして利用することを考えると，それらのサーバがどのネットワークに接続されるのか，事前に想定できない場合がある．一般的にコンポーネントを利用したアプリケーション開発は，利用するコンポーネントが開発時に静的に決定していて，それらを組み合わせることによって行われる．しかしそれらのサーバ上で動作するコンポーネントを利用しようとする場合には，ネットワーク上のどこにコンポーネントが存在するのかを事前に特定できない．したがって，それらを利用するようにアプリケーションを開発できない．そのようなさまざまな形態の端末で運用されているサービスも，柔軟に連係できることが望ましく，そのためには前述した課題に加えて，必要なサービスをアプリケーション実行時に自動的に発見して連係する必要がある [Austin2004] [Heuvel2003]．このように，分散コンポーネントが動作する機器をネットワークに接続するだけで，ネットワーク上のその他の機器と自動的に連係するという，いわゆる “Plug and Play” を実現する必要がある．

本研究ではこれらの課題を解決するため，異種分散コンポーネントの連係を支援するために，“HeteroSOC (Service Oriented Computing for Heterogeneous distributed components) フレームワーク” を提案する．分散コンポーネントを組み合わせて動作させる方法としては，必要なコンポーネントを入手しローカルコンピュータに配置して，それらを「統合する」方法と，ネットワークを介してリモートに存在する個々のコンポーネントを呼び出した結果を合成することによって「連係させる」方法の 2 種類が考えられる．分散コンポーネントでは一般的に，外部にインタフェースを公開して，そのロジックを公開することはない．したがって本研究では，後者のようにそれぞれのコンポーネントを外部から呼び出すことによって連係させる場合の課題を取り上げ，それを解決する機構を提案する．

まず HeteroSOC フレームワークは，組み合わせるコンポーネントを開発時に静的に決めることができる場合の課題を解決するために，開発支援システムを提供する．このシステムは，分散コンポーネント間の異種性を吸収するプログラムコードを，自動的に生成する．開発者はまずサービスを組み合わせる際のワークフローに関する情報を記述する．開発支援システムは，そのワークフロー情報を利用し，異種分散コンポーネントを呼び出すことによって一つのアプリケーションプログラムとして連係して動作させるためのアダプ

タと呼ぶプログラムコードを自動生成する．このアダプタがそれぞれのコンポーネントの異種性を自動的に吸収する．このようなシステムにより，異種分散コンポーネントを利用したアプリケーションの開発を容易にする．そして，開発者はこの開発支援システムを利用して，異種分散コンポーネントを連係させるアプリケーションを開発する．利用者は，そのアプリケーションを介して複数の異種分散コンポーネントを呼び出し，それらを連係させることができる．次に，分散コンポーネントを組み合わせるアプリケーションの信頼性の課題を解決するために，アプリケーション実行時にコンポーネントの呼出しに障害が発生した際に，開発時に指定した別のコンポーネントに処理を切り替えられるような仕組みを用意する．この仕組みを実現するために，提供する開発環境では各アクティビティを実現するコンポーネントを複数指定できるようにする．そしてワークフローの開始から終了まで，考えられるコンポーネントのすべての組合わせについてアダプタを生成する．コンポーネント呼出しの障害時には，障害の発生したコンポーネントを利用しないアダプタに処理を自動的に切り替えることにより，信頼性の低下を回避する．

そして Plug and Play を実現するために，実行時に動的にコンポーネントを組み合わせられるようにする．そのために HeteroSOC フレームワークは，生成したアダプタを実行するためのプラットフォームを提供する．このプラットフォームは，ローカルネットワークに存在するコンポーネントを動的に探索して利用するためのミドルウェアを含む．このミドルウェアを利用することにより，分散コンポーネントが動作するサーバやそれを利用するクライアントは，実行時に必要なコンポーネントを自動的に発見し，発見したコンポーネントを動的に連係させる．その際に，それらが動作する端末にはコンポーネントを利用するための設定を行わずに，ネットワークに接続することで，Plug and Play でコンポーネントを利用できるようにする．また分散コンポーネントの異種性の課題に対応するため，提供するプラットフォームに含めるミドルウェアにおいて，アプリケーションの実行時にコンポーネントの異種性を動的に吸収する．

これにより，アプリケーション開発時にネットワーク上のどこに存在するのか想定できなかったコンポーネントでも，利用者がアプリケーションを実行する際に動的に連係することができる．さらに，それらの間の異種性も実行時に動的に吸収できる．このように異種分散コンポーネントにより実現されるサービスを，柔軟に連係させることができる．

1.4　本論文の構成

本論文の構成は，次の通りである．

第 2 章 分散コンポーネントアーキテクチャ
: 分散コンポーネントアーキテクチャの概要と，本研究で対象とする CORBA と EJB と Web サービスを示す．そしてその後で，それらを連係させるための技術を示す．

第 3 章 異種分散コンポーネント連係の問題
: 1.3 節で挙げた異種コンポーネント連係の課題の詳細を論じる．

第 4 章 関連研究
: 既存の関連技術を示す．

第 5 章 HeteroSOC フレームワークの全体設計
本論文で提案する，異種分散コンポーネントの連係を支援する機構について示す．

第 6 章 異種分散コンポーネントを利用するアプリケーションの開発支援システム
提案機構のうち，組み合わせるコンポーネントを開発時に静的に決めることができる場合の課題を解決するための環境について述べ，実施した評価実験について述べる．

第 7 章 異種分散コンポーネントの Plug and Play プラットフォーム
提案機構のうち，実行時に動的に連係を行うためのミドルウェアについて述べ，実施した評価実験について述べる．

第 8 章 実装したシステムの有効性に関する考察
効率的に開発するための条件と，プラットフォームの適用先を述べることによって，実装したシステムの有効性を考察する．

第 9 章 結論
まとめと，今後の展望を述べる．

第2章　分散コンポーネントアーキテクチャ

本章では，コンポーネントの概念と分散コンポーネント技術について，およびそれらを利用したサービス指向コンピューティングについて述べる．

2.1　コンポーネントウェア

コンポーネントウェア (Componentware: Component-based software engineering) とは，“オブジェクト指向に基づき Plug and Play 型ソフトウェア部品を組み合わせてシステムを開発するための技術の体系” である [Aoyama1998]．狭義には，このような開発技術を支援する開発環境の総称としても使われている [Brown1996] [Brown2000]．この時のソフトウェア部品のことをコンポーネント (Component) と呼び，“独立して動作し，提供しているサービスをインタフェースを通して利用できる機能の素片” を指し示す [Brown2000]．外部から呼び出すプログラムを開発する際には，開発者はコンポーネントをブラックボックスとして扱えば良いので，ソフトウェア開発が容易になる．そしてコンポーネントを利用することにより，ソフトウェアに対してモジュールの差し替え，修正，再利用をより簡単に行うことができ，ソフトウェアの生産性と品質の向上が期待できる．

ソフトウェア開発はコンポーネントウェアによって，部品を開発するコンポーネントベンダと，部品を組み合わせてアプリケーションを開発するコンポーネントインテグレータの 2 つの業種に分類されるように発展すると考えられている [Aoyama1996]．さらに，この 2 つの業種間で部品の流通を仲介するコンポーネントブローカが新しい業種として出現している．このことは，ソフトウェア部品市場の出現を意味する．

コンポーネントは，1 個のオブジェクトから，1 つのアプリケーション，あるいはそれを運用するサーバに至るまで，その粒度は幅広い．しかし，それらのコンポーネントは何らかのサービスを提供する，という共通の特徴を持っている．

そしてインターネットの急速な発展に伴い，ネットワークを介してコンポーネントを利用する必要性が高まってきており [Lassing1998]，分散コンポーネント技術へと発展している．

2.1.1　コンポーネントウェア出現の背景

コンピュータの普及に伴い，コンピュータソフトウェア開発に対して，次のような需要が発生している．

(1) エンドユーザによるパッケージソフトの組合わせ
　エンドユーザによってコンピュータ上で業務処理を行う機会が増大した．エンドユー

ザの行う業務により，必要とするソフトウェアの機能が異なる．そのためエンドユーザは，ワードプロセッサや，表計算ソフトウェア，データベース等，色々な機能を持ったソフトウェア群を組み合わせて利用し，自身の業務を遂行する．エンドユーザのニーズはコンピュータの普及により多様化しており，必要とするソフトウェアの機能も，エンドユーザによって多様化していく傾向がある．そのため，既存の色々なソフトウェアを組み合わせることによって，複雑な仕事を効率よく実行できるアプリケーションを，簡単に開発する仕組みの必要性が高まっている．この際の組合わせはそれぞれのユーザによって異なるため，エンドユーザ自身によって開発できることが望ましい．

(2) ソフトウェア開発コストの削減
汎用的なパーソナルコンピュータの普及により，一般的にエンドユーザは，汎用のハードウェアの上で個々の業務に特化したソフトウェアを動作させることで業務を行う．汎用的なハードウェアに関しては同一の製品を大量に製造できるため，コンピュータの普及に伴い，価格も徐々に低下している．一方ソフトウェアに関しては，個々の業務に合わせて開発したりカスタマイズしたりする必要がある．したがって，ハードウェアのコスト低下に比べ，ソフトウェアの価格は比較的高くなっている．ソフトウェアによっては，ハードウェアよりも格段に高価になることも珍しくはないが，一般の利用者からは受け入れられにくい．ソフトウェア開発コストの削減のためには，従来のように一からコーディングせずに，より簡単に開発できる方法が必要になる．

このような従来型のソフトウェア開発方法の限界とともに，組み立て型開発への転換が進み，その結果としてコンポーネントウェアが出現した．コンポーネントウェア出現に至る技術的な背景としては，次のように挙げることができる．

(1) オブジェクト指向技術およびインターネットの発展
オブジェクト指向技術の発展の結果，クラスライブラリやデザインパターンなどの要素技術が発達してきており，これらを利用しソフトウェアを再利用できる．一方でインターネットの発達により，インターネットを利用するさまざまなアプリケーションが開発されている．そしてさまざまなオブジェクトの流通基盤として，インターネットを利用するようになってきている．

(2) Plug and Play 型部品市場の出現
Visual Basic の再利用部品であるカスタムコントロールが，商業的に成功を収めた．そして色々なソフトウェアベンダによってカスタムコントロールを開発し，エンドユーザのコンピュータに組み込むことによって，ソフトウェア部品を簡単に利用することができる．このような性質から，Plug and Play 型ソフトウェア部品と呼ばれるようになった．そしてソフトウェア部品を供給する数多くのベンダが生まれ，ソフトウェア部品を流通するための市場も形成された．

2.1.2　コンポーネントウェアの特徴

コンポーネントウェアを，従来のソフトウェア再利用技術と比較した場合の特徴は，次のようになる．

(1) オブジェクトコードによるソフトウェア再利用
従来のソフトウェア再利用技術では，一般的にソースコードの再利用が行われてきた．このため再利用を行うためには，次の問題がある．

(a) ソフトウェア部品のソースコードを利用するためには，通常は，動作させたいプラットフォーム上で再コンパイルする必要がある．

(b) ソフトウェア部品を利用して機能を追加するためには，部品の内部で行われる処理を知る必要がある．場合によっては，部品の内部のソースコードを変更する必要もある．

これらの問題により，ソースコードによる再利用をエンドユーザが行うことは比較的困難である．したがって，一般的にプログラミングの専門家でないと再利用が難しい．これらの問題を解決するため，コンポーネントウェアではオブジェクトコードによりソフトウェア再利用を行う．これにより，ソフトウェア部品をブラックボックスと捉えて，再利用できる．

(2) 複合オブジェクトの再利用
複数のオブジェクトを組み合わせて新しいソフトウェアを開発した場合に，それらのオブジェクト群をパッケージ化し，まとめて再利用することができる．

(3) インタフェースの標準化
プログラム呼出しだけではなく，データ交換の形式や GUI などのソフトウェアの多様なインタフェースを標準化する．そしてそれによって，部品を交換可能にする．

(4) 多様なアーキテクチャのサポート
あるコンポーネントウェアで動作するコンポーネントが，OS，プロセッサなどの違いを超えて動作する環境を，ミドルウェアやフレームワークの形として提供する．

(5) 部品組み立て型アプリケーション開発モデル
アプリケーションが実現すべき業務を中心に，必要なソフトウェア部品を組み込むことによって，アプリケーション開発が行われる．

2.1.3　コンポーネントウェア実現のための要素技術

コンポーネントウェアによるソフトウェア開発・運用のために必要な要素技術は，次の通りである．

(1) パッケージ化技術
アプリケーションを構成するオブジェクトを，ソフトウェア部品 (コンポーネント) としてパッケージ化するための技術.

(2) 部品組込み技術
外部からのイベントを受け取り，複数の部品を連係させて動作させて応答を返すための実行環境と，その際の制御を記述するためのスクリプト言語.

(3) アプリケーション開発支援環境
コンポーネントを視覚的に組み合わせることのできる，アプリケーション開発支援環境.

一般的にこれらの要素技術は，色々な部品を動作できるようにするために，アプリケーションに特化しない汎用的なフレームワークとして用意される.

2.1.4　コンポーネントの利用範囲による分類

コンポーネントは，部品の再利用の観点から，開発に利用される範囲によって次のように分類できる.

(1) プロダクト／プロダクトライン固有の部品
特定のプロダクトでの開発に再利用されるソフトウェア部品.

(2) ドメイン固有の部品
特定ドメイン内で共通的に利用する機能が含まれ，同一ドメインの他のソフトウェア開発において，再利用されるソフトウェア部品.

(3) ドメイン独立の部品
特定のアプリケーションドメインによらず再利用されるソフトウェア部品.

2.2　ビジネスアプリケーションの変遷

近年，サーバサイドで動作するコンポーネントを利用し，ネットワークを介して動作するビジネスアプリケーションを構築することが多くなっている．この背景として，ビジネスアプリケーションの形態の変遷について述べる.

(1) 2 階層システム
ホストコンピュータによって，すべての処理を行う．クライアント端末からは専用プロトコルによって，ホストコンピュータに接続を行う．ホストコンピュータは処理を行い，その結果をクライアント端末に専用プロトコルによって返す．そしてクライアント端末は受け取った応答をそのまま表示する.

図 2.1 に構成を示す．2 階層システムは，クライアント端末，ホストコンピュータから構成される．アプリケーションで実行すべき処理 (ビジネスロジック) を実行するように実装したプログラムを，ホストコンピュータ上で動作させる．ビジネスロジックによっては，ホストコンピュータに組み込まれたデータベースを利用して処理を行う.

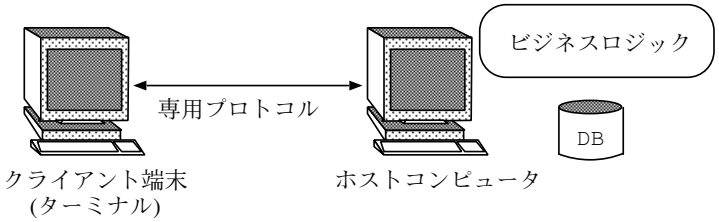

図 2.1 2 階層システム

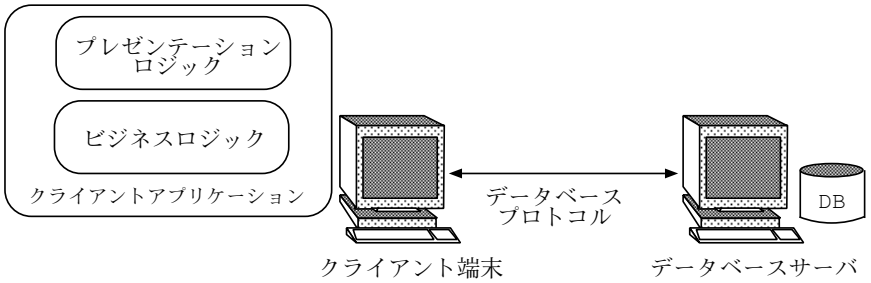

図 2.2 クライアント／サーバシステム

この方式では，端末が処理結果を表現する能力は，サーバの処理能力に制限される．また，ビジネスロジックや，ホストコンピュータに存在するデータの利用は，専用クライアント端末からのみに限定される．

(2) クライアント／サーバシステム

2 階層システムから，情報資源とビジネスロジックを分離させたものである．

図 2.2 に構成を示す．2 階層システムでは，ホストコンピュータにデータベースとビジネスロジックが配置されており，情報資源の利用のためには，専用のアプリケーションを記述し，ホストコンピュータに配置する必要があった．クライアント／サーバシステムでは，データベースなどの情報資源を集中管理するサーバコンピュータを用意する．そして，アプリケーションが実現するビジネスロジックと実行結果を表示するための処理 (プレゼンテーションロジック) を，アプリケーションプログラムに実装し，クライアント端末に配置する．クライアントアプリケーションは，ビジネスロジックにしたがい，データベースプロトコルを介してサーバを利用する． そしてその結果を，プレゼンテーションロジックにしたがって表示する．

この方式ではデータのオープンな利用が可能である反面，ビジネスロジックがクライアント側に分散しているため，アプリケーションの管理が複雑になる．また，クライアントアプリケーションにビジネスロジックが記述されているため，クライアントのプラットフォームごとにビジネスロジックを書き換える必要がある．

(3) 3 階層システム

クライアント／サーバシステムの欠点を克服するために，クライアントからビジネスロジックを分離させたものである．

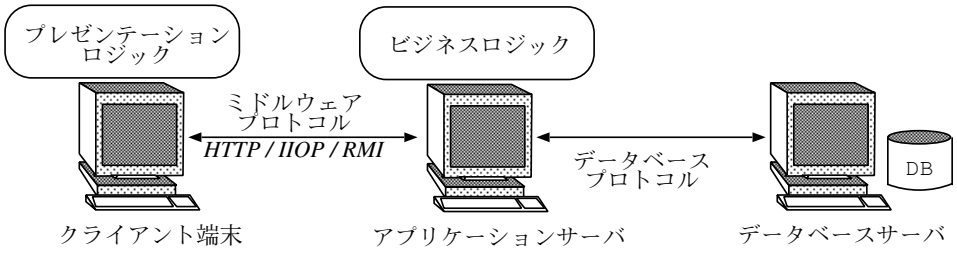

図 2.3 3 階層システム

図 2.3 に構成を示す．クライアント端末上で動作するアプリケーションプログラムには，表示に関するプレゼンテーションロジックのみを実装する．また一般的に，クライアントとアプリケーションサーバ間の通信には，特定のアプリケーションに依存しないミドルウェアプロトコルが用いる．このプロトコルとしては，HTTP (Hyper Text Transfer Protocol)，IIOP (Internet Inter-ORB Protocols)，Java-RMI (Remote Method Invocation) が挙げられる．

この方式ではビジネスロジックのみを分離したため，メインテナンスが容易になる．汎用的なミドルウェアプロトコルを利用しているために，クライアント端末の種類を限定しない．またクライアント端末では，サーバから得た処理結果を，どのように加工して表現するかを，プレゼンテーションロジックに記述できる．

このうち 3 階層システムでは，ビジネスロジックを中間に位置するアプリケーションサーバに記述する．このビジネスロジックを実現するためのアプリケーションを，サーバサイドコンポーネントと呼ぶ．分散コンポーネント技術ではサーバサイドに存在するコンポーネントにロジックを記述し，ネットワークを介してクライアントプログラムから呼び出し，処理を実行する．

2.3　分散コンポーネント技術

3 階層システムを実現するために利用される分散コンポーネント技術は，ネットワークを介して通信を行うソフトウェアに対してコンポーネント技術を適用したものである．そしてネットワーク上に設置したアプリケーションを，従来のコンポーネント技術と同様に，ブラックボックスとして外部から呼び出せるようにしたものである．本研究では，3 階層システムを実現するための分散コンポーネント技術に着目している．本節では本研究において対象とする分散コンポーネント技術について述べる

2.3.1　CORBA (Common Object Request Broker Architecture)

CORBA とは，OMG が標準化を行っている技術仕様であり，サーバサイドコンポーネントを運用するための基盤を提供する．CORBA は，OMG の制定する OMA (Object Management Architecture) と呼ばれるアーキテクチャを基本とする．OMA は，CORBA

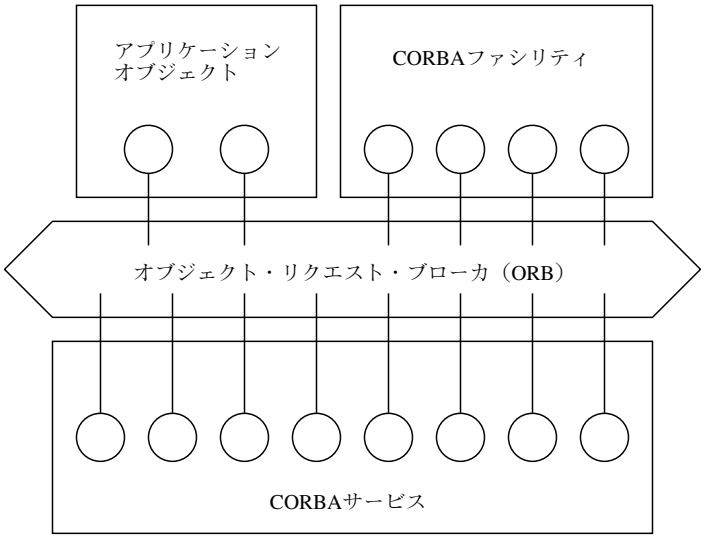

図 2.4 OMG オブジェクト管理アーキテクチャ

オブジェクトと呼ばれるソフトウェアコンポーネントを，ネットワーク経由で提供することを可能にする．

OMA は，次の 4 つの要素から構成される．それらの関係を図 2.4 に示す．

(1) オブジェクトリクエストブローカ (ORB: Object Request Broker)
OMA 参照モデルの中核をなし，CORBA オブジェクトバスと呼ばれるメッセージング機構を提供する．そして，CORBA オブジェクト間のリクエストと，その応答の透過的な送受信を可能にする．

(2) CORBA サービス (共通オブジェクトサービス)
ORB の機能を拡張し，補完するためのさまざまな機能の集合であり，システムレベルで共通的に利用される機能を提供する．代表的なものに CORBA オブジェクトを名前でアクセスするためのネーミングサービス，CORBA オブジェクト間のイベント送受信を行うためのイベントサービスなどがある．

(3) CORBA ファシリティ (共通ファシリティ)
多くのアプリケーションが共通して利用する機能の集合であり，アプリケーションレベルで共通的に利用される機能を提供する．これは，さまざまなアプリケーションに適用可能な水平型と，特定の業種向けに特化した垂直型に分けられる．なお，水平型を共通ファシリティ，垂直型をドメインインタフェースとしてさらに分類する場合もある．

(4) アプリケーションオブジェクト (ビジネスオブジェクト)
各ベンダが開発するユーザアプリケーション．CORBA サービスや，CORBA ファシリティを利用し，ORB を介して他のオブジェクトに作用する．

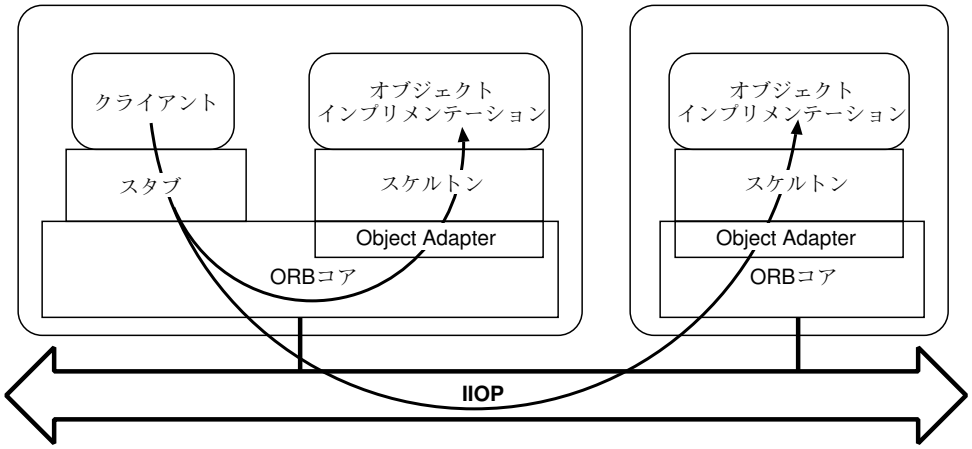

図 2.5 CORBA のオブジェクト間通信

図 2.5 に，CORBA オブジェクト間通信の基本的な構成を示す．矢印はオブジェクトの呼出しを示している．それぞれの構成要素を，次に述べる．

(1) クライアント
クライアントは，CORBA オブジェクトを指し示すオブジェクトリファレンス (CORBA オブジェクトを一意にする名前もしくは識別子) を入手し，そのオブジェクトの持つオペレーションを起動する．CORBA オブジェクトの提供者は，IDL (Interface Definition Language) [OMG2002] と呼ばれるインターフェース記述言語により，オブジェクトリファレンスを予め定義する．CORBA オブジェクトの利用者は，入手したインタフェースの情報にしたがってクライアントプログラムを開発し，それを経由してサーバオブジェクトを呼び出す．

(2) オブジェクトインプリメンテーション
オブジェクトインプリメンテーションとは，CORBA オブジェクトの実装と，そのインタフェースのことを指す．CORBA オブジェクトのインタフェースに関する情報は，前述のように IDL により記述されて公開されている．このインタフェース情報に対応する CORBA オブジェクトの実体をサーバント (Servant) と呼ぶ．そしてこれらをあわせて，オブジェクトインプリメンテーションと呼ぶ．

(3) スタブ (Stub)
スタブとは，IDL 定義から生成されるルーチン群であり，クライアントプログラムからサーバオブジェクトの呼出しを行うためのプログラムである．クライアントはサーバオブジェクト呼出しの前に，IDL で記述されるインタフェース情報にしたがってスタブを生成する．そしてスタブを利用することで，あたかもローカルオブジェクトにアクセスを行うように，リモートオブジェクトを呼び出すことができる．

(4) スケルトン (Skeleton)
スケルトンは，クライアントからの呼出しが発生した際に，サーバ側でオブジェクトアダプタが各オブジェクトインプリメンテーションを呼び出すためのプログラム

である．クライアント側のスタブと同様に，サーバは IDL 定義から，スケルトンを構成するルーチン群を生成する．クライアントからのリクエストを受け取ったオブジェクトアダプタは，スケルトンの持つディスパッチングルーチンを呼び出すことによって，オブジェクトインプリメンテーションの持つ適切なオペレーションを起動する．

(5) ORB コア
ORB コアは，ORB のカーネル機能を提供する．ORB コアが提供する最低限の機能としては，CORBA オブジェクトリファレンスの生成や解釈などオブジェクトの識別機能と，クライアントとオブジェクトインプリメンテーション間の通信機能がある． これらの機能により，クライアントはサーバオブジェクトの位置を意識することなく通信を行うことが可能となる．

(6) オブジェクトアダプタ
オブジェクトアダプタ (Object Adapter) は，ORB コアからオブジェクトインプリメンテーションへの制御の受渡しを行う．そのために，CORBA オブジェクトの生成，削除，CORBA オブジェクトやサーバントの活性化，非活性化，例外情報の設定などを行う．

2.3.2　EJB (Enterprise Java Beans)

EJB は，サーバサイドコンポーネントを Java で開発するための仕様である [SunEJB2006]．

EJB におけるサーバサイドコンポーネントを，エンタープライズ Bean と呼ぶ．そしてサーバには，エンタープライズ Bean を実行するための EJB コンテナを用意する．EJB コンテナは，次のサービスを提供する [Chang1998]．

(1) マルチスレッド制御

(2) トランザクション管理

(3) 状態管理・永続化

(4) セキュリティ

(5) リソースの共有

エンタープライズ Bean は，その用途に応じて，次のような 3 種類に分類することができる．

(A) Session Bean
Session Bean は，呼出しセッションごとの処理結果を永続化しないエンタープライズ Bean である．そのため，単一クライアントからの要求に応じてメソッドを実行して，結果を返す．セッションが持続している間エンタープライズ Bean の状態が保持される Stateful Session Bean と，同一セッション内でもエンタープライズ Bean の状態が保持されない Stateless Session Bean に分別できる．

Session Bean は単一クライアントのために実行され，ライフサイクルが比較的短命になる．また，クライアントからの接続ごとにインスタンスが生成されるので，EJB サーバのクラッシュ時には元の状態を復元できない．

(B) Entity Bean

Entity Bean は，処理結果を永続的に保持する機能を持ったエンタープライズ Bean である．実行した処理結果は，サーバプロセスが終了しても持続する．永続化の処理の実装方法によって，CMP (Container Managed Persistence) と，BMP (Bean Managed Persistence) に分別できる．CMP では EJB コンテナによって永続化が管理されるため，開発者はリソースへのアクセスのためのコードを記述する必要がなく，また異なるリソースを用いる場合であってもコードに変更を加える必要がない．そのため開発者がエンタープライズ Bean に処理させたいビジネスロジックのみを記述すれば，EJB コンテナがリソースへのアクセスを自動的に行う．一方 BMP では開発者自身が永続化に関するコードを記述する必要がある．

Entity Bean は複数クライアントからの共有アクセスを実現しており，ライフサイクルが比較的長期である．また，CMP ではインスタンスの状態が EJB コンテナのデータベースに保持されるので，EJB サーバのクラッシュ時にも元の状態を復元できる．

なお Entity Bean は，EJB 1.1 仕様以降のサーバでサポートされている．また，2006 年 5 月に公開された最新の EJB 3.0 仕様では エンタープライズ Bean の実装を簡略化するため，Entity Bean から永続化の機能が省略されている．永続化は別途 Java Persistence API (JPA) [SunJPA2006] を用いてクライアント側で行う．

(C) Message-Driven Bean

クライアントから送信された JMS (Java Message Service) メッセージ [SunJMS2002] を受信するエンタープライズ Bean である．JMS は非同期メッセージの送受信を行うための API であり，これにより非同期メッセージの送受信を行うことができる．送受信の方法には，JMS キューに送信された一つのメッセージを一つのアプリケーションのみが受け取ることができる方式 (PTP: Point-to-Point) と，複数のアプリケーションが受け取ることができる方式 (Pub/Sub: Publish-Subscribe) の 2 種類がある．

Message-Driven Bean では，非同期処理を実現できる．そして，Pub/Sub の方式を利用することによって，クライアントからのメッセージを複数のアプリケーションで同時に処理することが可能である．

なお Message-Driven Bean は，EJB 2.0 仕様以降のサーバでサポートされている．

エンタープライズ Bean を利用する際に，クライアントがサーバコンポーネントを呼び出す様子を図 2.6 に示す．

それぞれのエンタープライズ Bean には，外部からアクセスできるようにするために，Home と Remote の二つのオブジェクトが用意される．これらのオブジェクトはエンタープライズ Bean の開発者が記述し，クライアントは Java RMI や IIOP を用いてアクセスする．

a) Home オブジェクト
エンタープライズ Bean のインスタンスの初期化 (create) や検索 (find)，削除 (remove) などを管理するためのメソッドを持つオブジェクト．

b) Remote オブジェクト
エンタープライズ Bean のビジネスロジックを実際に呼び出すためのメソッドを持つオブジェクト．

クライアントからは，Home オブジェクトのインタフェース (Home インタフェース) を介して必要なエンタープライズ Bean を検索する．そして，EJB コンテナは，そのエンタープライズ Bean を利用するための Remote オブジェクトをインスタンス化する．そしてクライアントプログラムは，Remote オブジェクトのインタフェース (Remote インタフェース) を介して Remote オブジェクトを呼び出す．これによってエンタープライズ Bean のビジネスロジックを呼び出すことができる．

また，サーバ側で公開するエンタープライズ Bean を事前に登録するためのディレクトリサービスとして，JNDI (Java Naming and Directory Interface) [SunJNDI2003] という仕組みがある．これによって，クライアントから必要なエンタープライズ Bean を検索して利用することもできる．

2.3.3 Web Service

Web Services の定義はさまざまであり，色々な解釈をとる事ができるが，W3C (World Wide Consortium) では次のように定めている．

> “XML によって定義および記述された公開インタフェースとバインディングを持つ，URI (Uniform Resource Identifier) によって識別されるソフトウェアシステムである．その定義を他のソフトウェアシステムから見つける事が可能であり，その定義によって指定された方法で XML ベースのメッセージを用いて相互やりとりを行う．” [Booth2004]

前述の CORBA や EJB では，クライアントとサーバ間のやり取りには，RMI や IIOP を利用していた．これに対して，Web Service では SOAP (Simple Object Access Protocol)

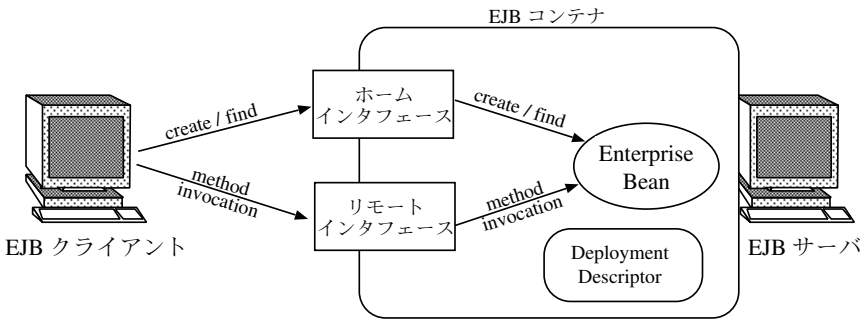

図 2.6 EJB のコンポーネント呼出し

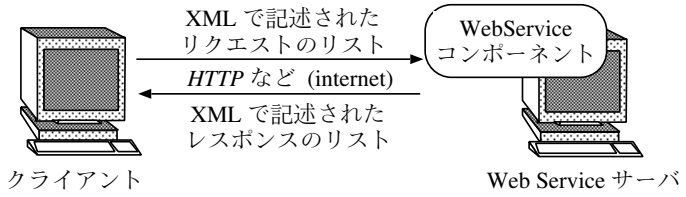

図 2.7 SOAP によるコンポーネント呼出し

[Gudgin2003] を利用する．

SOAP は XML 形式のメッセージをやりとりし，リモートにあるオブジェクト (データ) にアクセスする為のプロトコルである．メッセージトランスポートのためには，特定のプロトコルの利用を規定していない．そして一般的には HTTP が用いられるが，SMTP や FTP などを用いた実装も存在する．メッセージの記述に XML を用いているので，呼出しを行う際の各データに対して型情報などを付与することができ，柔軟にデータ表現を行う事ができる．

Web Service のオブジェクト間通信は，プロトコルに SOAP を用いるという点以外では，他の分散コンポーネント技術と大きな相違は無い．

Web Service では，サーバに存在するコンポーネントのインタフェースに関する情報を，WSDL (Web Service Description Language) [Christensen2001] と呼ばれる XML 形式の記述方法によって記述する．そしてサーバプログラムの提供者は，事前にこのインタフェースを公開しておく．これは CORBA における IDL 定義で記述されたインタフェース情報の場合と同様である．

クライアントの利用者は WSDL によるインタフェース情報の記述を元に，クライアントプログラムを開発する．そしてクライアントプログラムによって SOAP メッセージを組み立て，サーバに対して呼出しを行う．受け取った SOAP メッセージを解釈し，サーバに登録されているオブジェクトを呼び出す．実行結果は再び SOAP メッセージとしてクライアント側に返される．図 2.7 に，この処理の流れを示す．

なお EJB の場合と同様に，公開するコンポーネントを事前に登録するためのレジストリサービスとして，UDDI (Universal Description, Discovery and Integration)[Clement2004] という仕組みがある．UDDI には，WSDL で記述されたコンポーネントに関する情報の他に，サービスの提供者やカテゴリなどの情報を XML 形式で格納することができる．そしてサービス利用者は，その情報を元に自身が望むサービスを見つけることができる．

2.4　分散コンポーネント技術とサービス指向アーキテクチャ

前述のコンポーネント技術を利用して，コンポーネントを開発することによって，サーバサイドで処理を行うアプリケーションを構築することができる．その際には，コンポーネントのセットをサーバプログラムとして実装し，外部から呼び出すことのできる「サービス」として公開する．W3C の定義 [Booth2004] に従うと,「サービス」とは，提供者や利用者から見た際の，一連の処理の集合のことを指す．

2.3 節で取り上げた各コンポーネント技術では，インタフェースを定義し公開することによって，外部から呼び出すことができるようにしている．EJB ではインタフェースの定義は，Java のプログラムによって行うが，CORBA や Web Service では，そのために IDL や WSDL と言った専用の記述言語を用意している．このように外部から呼び出すことができ，そのためのインタフェースが定義され公開されていて発見され得るコンポーネントのセットのことや，それらを利用したアプリケーション構築のスタイルのことを，「サービス指向アーキテクチャ(SOA: Service-Oriented Architecture) 」という [Haas2004] [Gold2004] [Mahmoud2005].

図 2.8 に，SOA の概要を示す．SOA では，サービスの提供者は，サービスを構成するコンポーネントをサーバコンピュータ上に配置する．そしてそれらのインタフェースに関する情報を公開する．サービスを利用するためには，まずコンポーネントのインタフェースに関する情報を取得し．それを利用するようにクライアントプログラムを開発する．そして利用者は，クライアントプログラムを実行することによって，ネットワークを介してコンポーネントを呼び出し，サービスの処理を実行する．

2.5　サービスの連係とサービス指向コンピューティング

既存の複数のサービスを呼び出し，それらを基本要素として組み合わせることによってアプリケーションを開発するコンピューティングパラダイムを,「サービス指向コンピューティング (SOC: Service Oriented Computing) 」という [Papazoglou2003][Bichler2006]. 図 2.9 に，SOC の概要を示す．サービスの組合わせは，サービスを構成するコンポーネントを実行した結果を別のサービスを構成するコンポーネントの入力に利用したり，それぞれのコンポーネントを実行した結果をまとめたりするプログラムコードを開発することで行われる．そして利用者は開発されたプログラムコードを介して，それぞれのサービスを連係させて利用する．

特定の分散コンポーネント技術を利用してコンポーネントを開発することによって，コンポーネント呼出しの形式を統一して複数のサービスを提供することができる．そしてクライアントプログラムの開発者は，個々のサービスごとにどのような形式でコンポーネントを呼び出すかを考えずに，それぞれのコンポーネントを利用できる．サービスを連係させるアプリケーションの開発についても，同様に利用する分散コンポーネント技術が統一している場合には容易に行うことができる．したがって，SOC を実現しサービスを連係させるための要素技術も，特定の分散コンポーネント技術を基本に確立されている．

2.3 節で取り上げた各コンポーネント技術のうち，Web Service についてはプロトコルに HTTP を利用できるため，Web Service の呼出しは，一般的にファイアウォールを通過できる．このような特徴により，現在では Web Service を利用したコンポーネント開発が増大している．そのため，SOC を実現させる要素技術は，主に Web Service 技術を利用して確立されつつある．

Web Service 技術を利用して SOC を実現させる際の要素技術のうち，インタフェースの記述形式として WSDL を利用する標準仕様を，図 2.10 に挙げる [Turner2003].

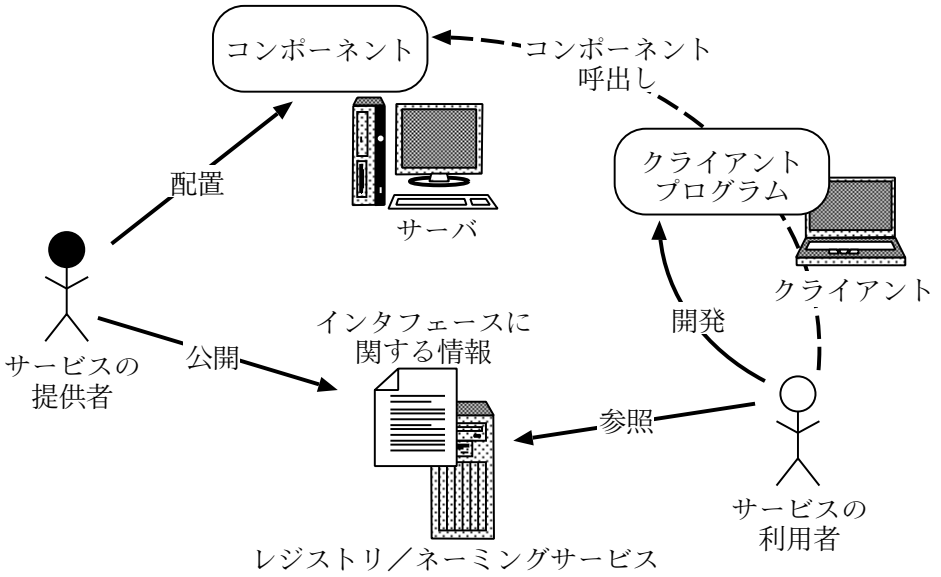

図 2.8 サービス指向アーキテクチャ

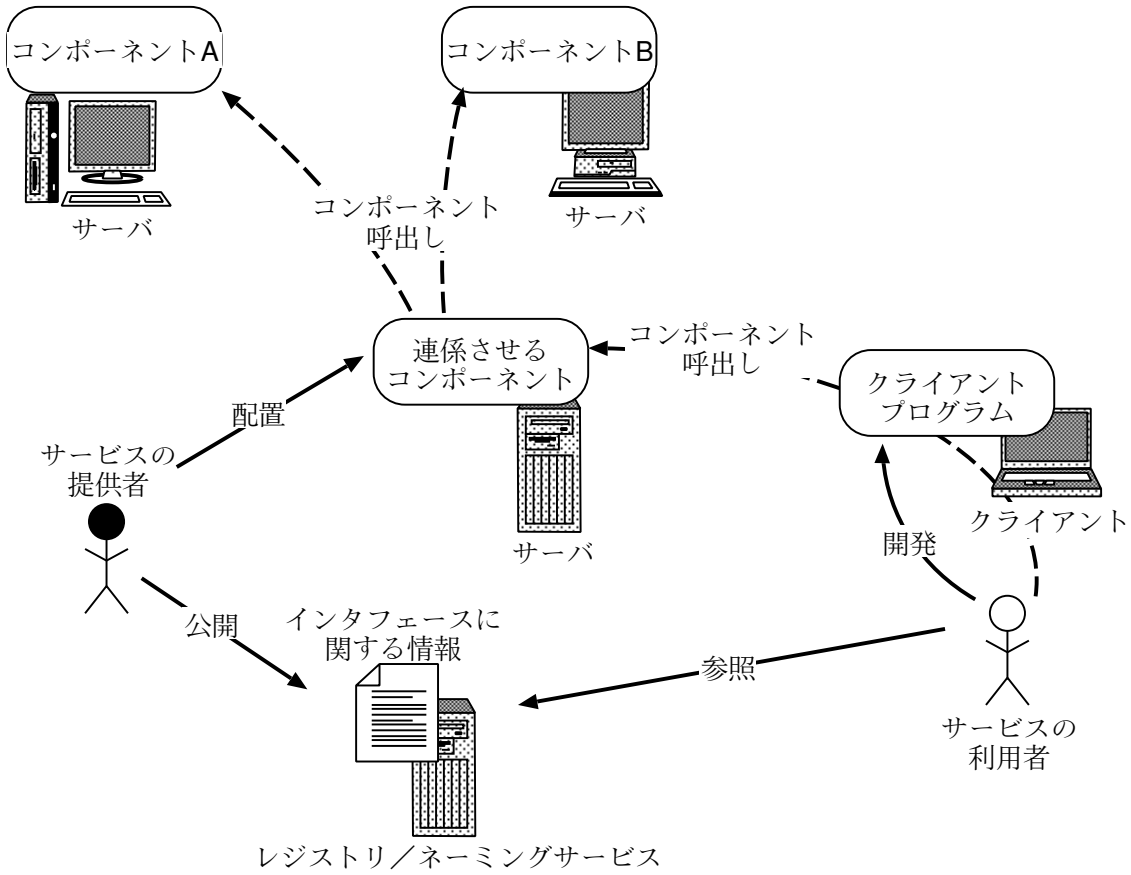

図 2.9 サービス指向コンピューティング

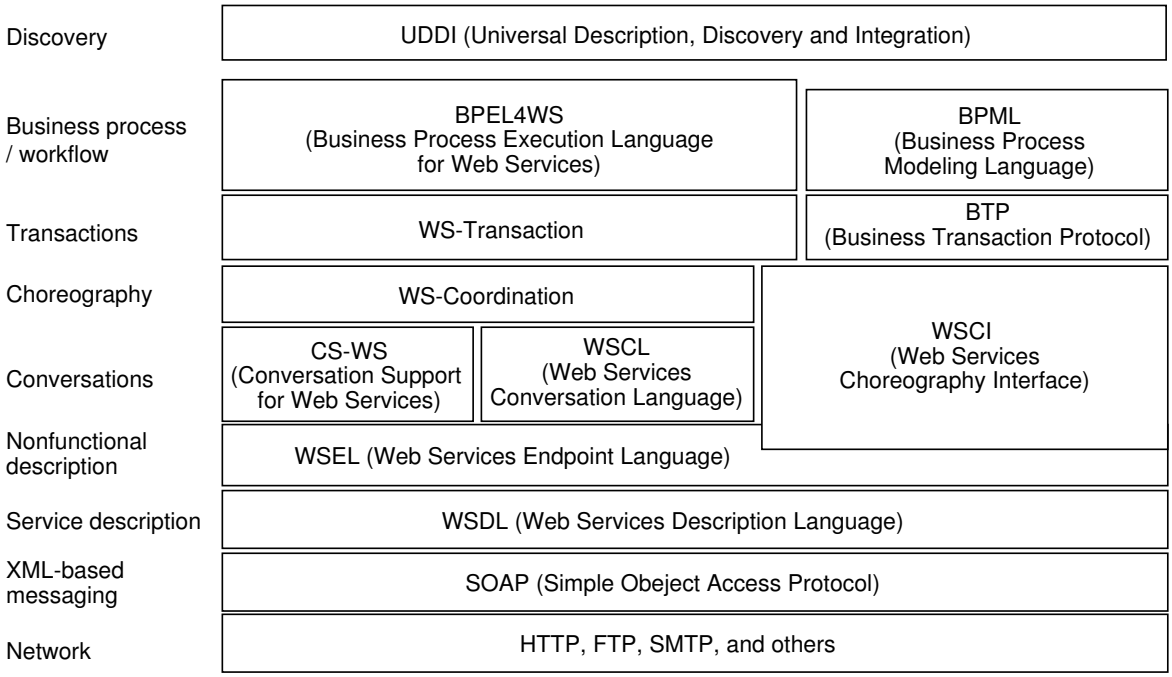

図 2.10 Web Service 関連標準仕様

2.6　サービス指向ソフトウェア開発とワークフロー記述言語

前述のように，既存の複数のサービスを呼び出し，その呼び出した結果を処理するアプリケーションを開発することにより， SOC を実現できる．

組合わせに利用する分散コンポーネントは，それぞれが一つのビジネスロジックを実現していることが多く，比較的粒度が大きい．このため，それらを利用してアプリケーション開発を行う際には，組み合わせる各コンポーネント間に強い関連性がない．したがって，コンポーネントを組み合わせてアプリケーション開発を行う際には，コンポーネントをどのような順番で組み合わせるのかという処理やデータの流れ (ワークフロー) に関する情報が重要になる．具体的には，次の点を明確にする必要がある．

A) どのコンポーネントとどのコンポーネントが対応づけられるのか，呼び出す順番はどうなるのか

B) 引数や返戻値はどのように対応するのか

アプリケーション開発は，開発者がこれらを指定するためのプログラムコードを記述することによって行われる．利用者は外部からそのプログラムコードを直接呼び出したり，ユーザインタフェースを介して呼び出したりすることで，コンポーネントを組み合わせたアプリケーションを利用する．しかしコンポーネントウェアの目的の一つは，この場合での開発者のようなプログラミングの専門家ではなくても，ソフトウェアの再利用を容易に行えるような環境を提供することであった．したがって，もっと容易に分散コンポーネントを組み合わせることのできる仕組みが必要である．

そこで，Web Service のコンポーネントを連係動作させる際のワークフローを記述するための言語が，いろいろな企業から提案されている．一例として，IBM によって提案された WSFL (Web Services Flow Language) [Leymann2001] や，Microsoft によって提案された XLANG [Thatte2001] が挙げられる．そしてこれらを統合した BPEL4WS (Business Process Execution Language for Web Services) 仕様 [Jordan2006] が，標準化団体 OASIS (Organization for the Advancement of Structured Information Standards) によって定義されている．これらは，サービスを連係させる際のワークフローを，XML を用いて記述するためのワークフロー記述言語である．そして，BPEL4WS によって XML で記述したワークフローにしたがって，コンポーネントを呼び出すサーバ実装も登場している [IBM2004] [ActiveBPEL2006]．これらの技術により，プログラムコードを記述せずに，SOC を実現することが可能である．

また別の視点として，ebXML BPSS(ebXML Business Process Specification Schema) [Levine2001] では，それぞれのビジネスパートナー (企業) 間のワークフローを記述することができる．これは，複数のビジネスパートナ間で一種の契約情報としてシステムの利用関係を共有する文書を記述するためのものであり，個々のコンポーネント間のデータフローに関しての表現を規定していない．

第3章　異種分散コンポーネント連係の問題

異なったベンダの提供するさまざまな分散コンポーネントを組み合わせてアプリケーションを開発するためには，まだ多くの課題が残されている．本章では，本研究で着目する課題を，組み合わせるコンポーネントを開発時に静的に決めることができる場合の課題と，実行時に動的にコンポーネントを組み合わせる場合の課題に分けて，分析する．

3.1　異種分散コンポーネント

分散コンポーネントを利用したアプリケーションを開発するために，色々なベンダによってさまざまな分散コンポーネントが提供されている．アプリケーション開発時には，それらのさまざまな分散コンポーネントを組み合わせられる必要がある．しかし，それらの分散コンポーネントは独自に開発されており，性能や質や実装は，プロジェクトチームにより異なる．またそれぞれが独自に管理されているため，全体の性能や信頼性は，保証できない．通常のソフトウェア開発でもこのような状況が発生するが，コンポーネントを組み合わせる際にはより重大な影響を与える [Olsen2006]．

このように，分散コンポーネントを連係させるアプリケーションを開発する際には，それぞれ独自に開発されたものであることを考慮する必要がある．そのため，呼出しの方法や，メッセージ，返戻値の形式が異なる可能性がある．本研究では，このような外部から呼び出す際の呼出し方法が異なる複数のコンポーネントを連係させる．その際の「異種性」は，次の要因により生じる．

(1) 利用するコンポーネント技術の相違
EJB や CORBA や Web Service など，あるビジネスロジックを実現する際に利用するコンポーネント技術が異なる場合．

(2) 利用するサーバ実装の相違
コンポーネントサーバが動作するサーバ実装が異なる場合．同一のコンポーネント技術を利用していても，サーバ実装の相違によりクライアントから呼び出すための API が異なっていたり，クライアントプログラムを実装する際に利用する言語が異なっている場合は，コンポーネントを呼び出すための方法が異なる．例えば，Web Service のサーバ実装としては多種のものが存在するが，利用するライブラリが異なるため，完全に同一の形式のプログラムコードを利用できない可能性がある．

(3) 利用するサーバ実装のバージョンの相違
同一サーバ実装において，サーバのバージョンの相違により，異なる呼出しの方法でクライアントから呼び出す必要がある場合．例えば，2.3.2 節で示したように，EJB

の一部の呼出しは，特定のバージョン以降で追加されている．このバージョン以前から存在する呼出し方法で呼び出す場合とは，呼出しの方法が異なる．

このように規格の異なるコンポーネント技術を利用していたり，異なるサーバ実装を利用して動作していたりすることによって，クライアントプログラムから同一シンタックスを利用した呼出しが不可能な複数の分散コンポーネントを，本論文では「異種分散コンポーネント」と呼ぶ．

3.2　組み合わせるコンポーネントを開発時に静的に決めることができる場合の課題

異種分散コンポーネントを利用して新しいアプリケーションプログラムを開発する場合は，それぞれの実装技術が異なるため，呼出し方法やメッセージや返戻値の形式が異なることを考慮しなければならない．

また，分散コンポーネントを組み合わせたアプリケーションの開発では，それぞれのコンポーネントがネットワーク上に分散して動作しており，開発者が設計を行った時点で利用できたコンポーネントが，利用者がアプリケーションプログラムを実行しようとする時点では利用できないことがある．

このため，異種分散コンポーネントを組み合わせるアプリケーションを開発し，それを運用することは容易ではない．本節では，この課題を分析する．

3.2.1　異種分散コンポーネントを組み合わせるコードの記述

2.5 節で述べたように，一般的に SOC では同一の形式の分散コンポーネントを組み合わせるように，アプリケーションプログラムの開発を行う．

分散コンポーネント技術では，利用者はクライアントプログラムを通してコンポーネントを利用する．したがって，異なる分散コンポーネント技術で実装されているサービスを，組み合わせて利用するように，クライアントプログラムを作成することも可能である．しかしこの場合は，コンポーネント間の実装形式の異種性をクライアントプログラムが吸収する必要がある．また，2.5 節で述べたようなアプリケーションプログラムを開発することによって異種分散コンポーネントを利用する場合は，複数のコンポーネントを連係させるコンポーネントが異種性を吸収する必要がある．そのためそれぞれのプログラムの開発者は，コンポーネントの呼出しの形式の相違を意識しなければならなくなる．また，多くの分散コンポーネント実装では，クライアントから利用するためのライブラリプログラムを提供している．異種分散コンポーネントを呼び出すためには，利用したいすべての分散コンポーネント実装に対応したライブラリプログラムを，同一のクライアントコンピュータで動作させなければならない．それぞれのライブラリプログラムは，必ずしも互いに同一のコンピュータで利用することを考慮していない．このため，同一コンピュータで同時に動作する保証がない．

さらに，コンポーネントの実装形式の異種性を吸収できたとしても，呼び出されるコンポーネント間でパラメータや返戻値の形式や意味が異なる．そのため，異種分散コンポー

ネントを利用して新しいアプリケーションを開発する場合は，コンポーネント間の異種性を吸収し，パラメータや返戻値の対応関係を考慮して，必要なプログラムコードを開発者が記述する必要がある．

したがって，開発者は次に挙げる事柄を熟知していなければならない．

(1) それぞれの分散コンポーネントを呼び出すために必要とされる言語

(2) それぞれのクライアントライブラリの構成，依存関係

(3) コンポーネントに対して呼出しを行うための形式

EAI や B2B によって情報システムを連係させる際の本質的な目的は異なる業務アプリケーションを相互に連係させることなので，アプリケーションプログラムの連係を容易に実現できる必要がある．したがって開発者は，どのようなコンポーネントをどのような順番で組み合わせてアプリケーションを開発するのかのみを考えることが望ましい．そしてどのような形式でコードを記述して，どのような手順で呼出しを行うのかということについては考えなくて済ませられるべきである．

3.2.2　アプリケーションプログラムの設計方法

コンポーネントを組み合わせることによって新しいアプリケーションプログラムを開発しようとする開発者は，どのコンポーネントをどの順番で呼び出すかというワークフローに関する情報を記述する必要がある．

EAI によって情報システムを連係させるために，多種の EAI システムが製品化されている．一般的な EAI システムでは，同一の分散コンポーネント技術を利用した複数のコンポーネントを連係させる．そして EAI ツールと呼ばれる開発環境と，連係するアプリケーションを動作させるためのサーバから構成する．アプリケーション開発者は，開発されるアプリケーションプログラムが処理する業務 (アクティビティ) を組み合わせ，EAI ツールを利用してその結果をワークフローとして記述する．そして記述したワークフローに基づき，それぞれのアクティビティにコンポーネントを対応づけることによって，アプリケーションの開発を行う．

EAI ツールはそれぞれの EAI システムごとに用意されており，開発者は EAI システムに対応するものを利用する．それらは，ワークフローの記述に関してツール間で統一したアプローチを採っておらず，逐次文書として表現することもあれば，フローチャートなどの図を用いて表現することもある．しかし EAI ツールの操作に慣れていない開発者にとっては，独自の表現方法を学習するより，汎用的な表現方法を利用できる方が望ましい．また EAI ツールの操作に慣れている開発者も，異なる EAI ツールを利用してシステムを開発する際には新たに独自の表現方法を学習しなければならない．したがってこの場合も，汎用的な表現方法を利用できる方が望ましい．

また，EAI ツールによって記述したワークフロー情報等の成果物は，EAI ツール独自の形式で表現され保存されることが多い．したがって別の EAI システムで利用しようとすると，同一の形式ではないために利用できない場合がある．EAI システムによって生成したアプリケーションプログラムを色々な EAI システム間で再利用できるようにするた

めに，コンポーネント間のワークフロー情報を保存する際の記述形式を統一することが望ましい．少なくとも，利用する EAI システムで解釈できなくとも開発者がメインテナンスできるように，可読な文書として表されるべきである．

3.2.3　アプリケーションの信頼性

第三者によって公開されている分散コンポーネントを組み合わせて新しいアプリケーションを開発する場合，組み合わせられる各コンポーネントを実行するコンピュータは，一般的にそれぞれ異なっている．特に異なるベンダが提供していることを考えると，それらのコンピュータが接続されるネットワークも異なっていることが多い．したがって，コンポーネントを連係させるアプリケーションを開発した場合は，アプリケーションプログラムを実行するコンピュータの管理者が対処できない次の障害が発生する可能性がある．

A) コンポーネントを実行するコンピュータの障害

B) コンポーネントを実行するコンピュータと，アプリケーションプログラムを実行するコンピュータとの間のネットワークの障害

前者については，それぞれのコンポーネントを実行するコンピュータを多重化することで軽減できる．しかし通常は，コンポーネントを実行するコンピュータの管理者は，コンポーネントを組み合わせようとする開発者や，アプリケーションプログラムを実行するコンピュータの管理者とは異なっている場合が多い．したがって多重化を期待することは，現実的ではない．もし仮に多重化できたとしても，コンポーネントを提供するベンダが多重化されたコンピュータ群を設置することになる．同じベンダによって提供されるため，それらのコンピュータ群は，通常は多重化する対象のコンピュータからネットワーク的に近接した場所に設置されると考えられる．したがって，後者の解決にはならない．

これらの障害は，開発者が設計を行った時点では利用できたコンポーネントが，利用者がアプリケーションプログラムを実行しようとする時点では利用できないことの要因になる．そしてその際の障害を，開発者や，アプリケーションプログラムを実行するコンピュータの管理者，利用者の誰もが解決できない可能性があり，アプリケーションプログラムの運用の信頼性を低下させる．すなわち，[Olsen2006] でも言及されているように，それぞれのコンポーネントが独自に管理されているため，このことが全体の性能や信頼性の低下に繋がる．

3.3　実行時に動的にコンポーネントを組み合わせる場合の課題

現在，さまざまな分野にコンピュータが普及しており，自動車や家電などの新しい分野にもコンピュータを利用したシステムが展開している．さらにインターネットの発達により，それらの新しいタイプのコンピュータを簡単にネットワークに接続することができるようになってきている．したがって，ユーザが端末をいろいろな場所に移動させて，その場所のネットワークに接続するような利用方法が増大している．アプリケーション運用時には，さまざまな形態の端末で運用されているサービスを，実行時に柔軟に利用できることが望ましい．

一般的にコンポーネントを利用したアプリケーション開発は，利用するコンポーネントが開発時に決まっていて，それらを組み合わせることによって行われる．しかし，例えばモバイルや組込み機器のような機器をサーバとして利用することを考えると，それらのサーバ上で動作するコンポーネントがネットワーク上のどこに存在するのかを，事前に想定できない場合がある．その場合は，事前にそれらを利用するようにアプリケーションを開発できない．従来はネットワークとそれに接続される端末の構成を，ある程度静的に決めることができたが，ユーザの利用形態が多様化したため，動的に変更されうる環境での利用を考慮する必要が生じてきている．その際には，必要なサービスを実行時に自動的に発見して連係する必要がある [Austin2004] [Heuvel2003]．例えばモバイルや組込み機器のような機器をサーバとして利用することを考えると，それらのサーバ上で動作するコンポーネントがネットワーク上のどこに存在するのかを，事前に想定できない場合がある．その場合は，それらを利用するように事前にアプリケーションを開発できない．それらを利用するためには，アプリケーションの運用時に必要とするサービスを動的に発見し，自動的に組み合わせる必要がある．そのためには，分散コンポーネントが動作する機器をネットワークに接続するだけで，ネットワーク上のその他の機器と自動的に連係するという，いわゆる “Plug and Play” を実現する必要がある．

Plug and Play の環境を考えると，接続された端末は接続されたネットワークで，まず利用可能な他の端末を探す必要がある．家電品を例にしてネットワークを利用してビデオを配信する環境を想定すると，ビデオプレイヤーのクライアントを利用者が購入して自宅のネットワークに接続した時には，ネットワーク内に存在するビデオサーバを探す必要がある．一方で，映画配信サービスのストリーミングサービスのように，グローバルなネットワークのある特定の場所に存在するサーバについては，クライアントはどこのネットワークに接続しても同一のサーバを利用する．サービスの提供者も，グローバルで公開することを前提としているので，サーバの場所をそれほど頻繁に変更しないだろうし，グローバルなネットワークに存在するディレクトリサービスにその場所に関する情報を登録していることも十分に考えられる．したがって予めサーバの場所を知っている場合は，クライアントにその場所を設定しておけば良く，存在を知らない場合でもグローバルなネットワークに存在するディレクトリサービスに問い合わせれば良い．このため，ネットワークに接続されたクライアントが，ストリーミングサーバの存在を探す必要性は，ローカルネットワークのビデオサーバを探す必要性ほどは高くない．

このように分散コンポーネントとしてこれらの機器の機能を提供することを考えると，特にローカルネットワークにおいて，Plug and Play 環境を実現する必要性があると考えられる．本節の残りでは，ローカルネットワークにおいて Plug and Play 環境を実現する際の問題点について考える．

3.3.1　分散コンポーネントの発見

Plug and Play 環境を実現するためには，クライアントやサーバがネットワークにプラグインした後，他の機器と連係して動作するために，利用したり利用されたりするための設定を自動的に行う必要がある．

SOA では，2.4 節で述べたような手順でコンポーネントの利用が行われる．したがっ

て，クライアントがプラグインした場合は，プラグインしたネットワークで利用したいコンポーネントとそれを提供するサーバを，レジストリやネーミングサービスから探す必要がある．また分散コンポーネントを提供するサーバがプラグインした場合は，自分自身をクライアントから発見できるようにするために，レジストリやネーミングサービスに登録する必要がある．どちらの場合も自動的に他の機器と連係して動作するためには，ネットワーク上のどこにレジストリやネーミングサービスが存在しているかを知らなければならない．また，プラグイン先のネットワークに存在するすべてのサーバが，同一のレジストリやネーミングサービスに登録されているとは限らない．このため，利用したいコンポーネントを，どのレジストリやネーミングサービスが登録しているかも知らなければならない．しかし，プラグインする端末は接続されるネットワークを特定できないので，それらを事前に知ることができない．したがって，連係させるための設定を自動化することは困難である．

さらに，ネットワークへの接続や切断が頻繁に発生することも想定する必要がある．仮にレジストリの存在を把握できても，クライアントはその都度レジストリやネーミングサービスから必要なコンポーネントを検索し，サーバはその都度レジストリやネーミングサービスへ登録や削除を行うことになる．したがって，ネットワークの構成が頻繁に変化する場合は，それらへの負荷が大きくなる．

このように，レジストリのような中央集権的なサーバによってそれぞれの端末や機器を管理するサーバ中心型アーキテクチャ (Server Centralized Architecture) には，色々な問題がある [Nakamura2004]．例えば，サーバにリクエストが集中することにより，信頼性が低下したり，負荷が集中したりする．また管理される対象を拡張した場合に，サーバの機能拡張が必要になる場合もある．その上で，[Nakamura2004] では，ホームネットワークにおいてホームサーバのような仕組みを設けずに，それぞれの機器を Web Service として公開するための仕組みを提供している．同様に Plug and Play を実現する際にも，コンポーネントの存在を管理するための装置を設けずに，コンポーネントを利用時に発見できる必要がある．

3.3.2　発見したコンポーネントの利用

Plug and Play 環境では，どのようなサーバ上で動作するコンポーネントを利用するかを事前に特定できない．そのため，さまざまなコンポーネントを連係させることができる必要がある．それぞれのベンダはそれらを独自に開発しているため，必ずしも同一の分散コンポーネント技術やプラットフォーム実装を利用している保証がない．これらの異種分散コンポーネント間では，呼出しの方法や，メッセージ，返戻値の形式が異なる可能性がある．このように分散コンポーネント間に異種性が存在する場合は，連係させて動作させることができない場合がある．

この問題については 3.2.1 節で述べた開発時での問題と本質的には同一である．しかし，Plug and Play 環境ではネットワークの構成が頻繁に変化するため，連係させるコンポーネントを事前に決定できない．異種性を吸収するためのプログラムコードを，呼び出す可能性のあるすべてのコンポーネントに対して用意すれば，このような環境でもコンポーネントを連係できる．しかし組み合わせる対象となるコンポーネントが多いので，このよう

な方法で異種性を吸収することは困難である．このように，Plug and Play 環境では開発時に組み合わせられるコンポーネントが特定しないため，異種性に関する問題がさらに発生し易い状況であると言える．

第4章　関連研究

本章では，関連研究を示す．まず，一般的なコンポーネント連係のための技術について述べる．そしてその後で，異種コンポーネントを対象にした場合の，コンポーネント間の異種性を吸収するシステムを挙げる．最後に，分散コンポーネントの動的発見や動的連係を行うためのシステムを挙げる．

4.1　コンポーネントを連係させるシステム

アプリケーションプログラムを連係させる既存の EAI システムとしては，BEA Systems の eLink [BEA2004]，Sybase の e-Biz Integrator [Sybase2002]，webMethods の webMethods Glue [webMethods2006] などがある．これらを利用することで，複数のコンポーネントを連係させることができる．これらのシステムでは，開発者がワークフローに関する情報を記述することによって，コンポーネントを呼び出して連係させるようにサーバを動作させる．しかしワークフロー情報指定のための表現や，指定された情報の保存は，それぞれのシステムで独自の形式で行っている．

これらの EAI システムとは別に，統一した形式でワークフロー情報を保存するためのシステムも提案されている．

まず，ワークフローに関する情報を，BPEL4WS により記述される文書として生成するためのエディタが提案されている [Brogi2006A]．しかしこのエディタでは，YAWL (YetAnother Workflow Language) [Hofstede2006] という独自のワークフローの表現形式に基づいて設計を行う必要がある．

さらに，ebXML BPSS により表現される文書として，ワークフローを記述するためのドキュメントエディタを作成するための API も提供されている [Kim2005]．しかし 2.6 節で述べたように，ebXML BPSS をコンポーネント間の連係を表すために利用することは困難である．

なおこれらはすべて，コンポーネントを提供するベンダによってアプリケーションプログラムを連係させることを前提としている．そのため，ネットワークを介して分散コンポーネントを呼び出すことによる信頼性の低下を考慮し，それを回避するための方法を提供していない．

4.2　コンポーネントの異種性を吸収するシステム

異種コンポーネントの相互運用を行う既存のシステムとして，Java プログラムから C++ のコンポーネントを呼び出すことを可能にするためのインタフェースを生成するシステムが提案されている [Kaplan1999]．これは，C++ コンポーネントのインタフェースを解析

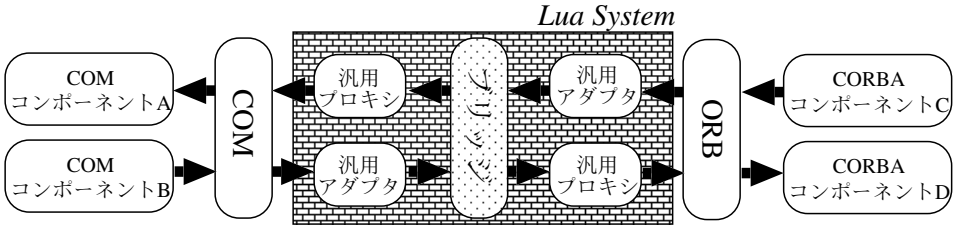

図 4.1 Lua を用いた異種分散コンポーネントの結合

し，Java から呼び出すためのインタフェースを自動生成する．

異種コンポーネント間の結合を行う部分を記述するためのインタープリタ言語として，結合言語 Lua が提案されている [Cerqueira1999]．コンポーネントを結合する際のシステムの構成を，図 4.1 に示す．Lua システムは結合言語 Lua の記述によって動作し，CORBA，COM，Java のコンポーネントを結合する．結合言語 Lua を用いることにより，Lua システムを利用して外部のコンポーネントへ呼出しを行うプログラム (ブリッジ) を作成する．ブリッジを各コンポーネントが呼び出すことで，コンポーネントを結合する．

Lua では異種コンポーネント間に中間言語を用意して，異種コンポーネント間での変換を行っている．しかし分散コンポーネントシステムの場合は前述のように，クライアントサーバシステムを実現するための技術の一つとして利用されるので，コンポーネントを提供するサーバをラッピングすることにより，異なった種類のクライアントから呼び出すための研究も行われている．

まず，既存の色々な種類のサービスを，HTML による Web クライアントや，Java のクライアントプログラムや，CORBA のクライアントプログラムからアクセスできるようにラッピングするシステムが提案されている [Zou2000] ．図 4.2 に，このシステムの構成要素を示す．このシステムは，アプリケーションサーバによって，複数のサービスを連係させる．そしてこの図では，既存のサービスを構成するコンポーネント (ここでは，レガシーコンポーネントと呼ぶ) を連係させている．CORBA サーバ上にレガシーコンポーネントを呼び出すコンポーネントを配置することにより，アプリケーションサーバからレガシーコンポーネントを CORBA のコンポーネントとして呼び出せるようにしている．CORBA サーバ上で動作する CORBA コンポーネントは，アプリケーションサーバからのリクエストに応じて，レガシーコンポーネントへの呼出しを行う．これによって CORBA コンポーネントとしてラッピングを行う．そして，CORBA コンポーネントのインタフェース情報をサービスデータベースに登録することにより，アプリケーションサーバは必要なサービスを探し，連係させることができる．そしてクライアントは，連係させるアプリケーションを，Java アプリケーションや，CORBA アプリケーションとして呼び出す．また Web サーバ経由で Web クライアントからアクセスできる手段も提供している．

同様に，クライアントサーバシステムや，マルチサーバシステム，Peer-to-Peer (P2P) システムを，Web Service としてアクセスできるようにラッピングするシステムが提案されている [Gomes2005] ．図 4.3 に示すように，それぞれのアプリケーションサーバ上に，リクエストを Web Service として変換するための Wrapper を用意している．また，P2P システムについては，動的に Peer を発見し，接続するため，通常のクライアントサーバ

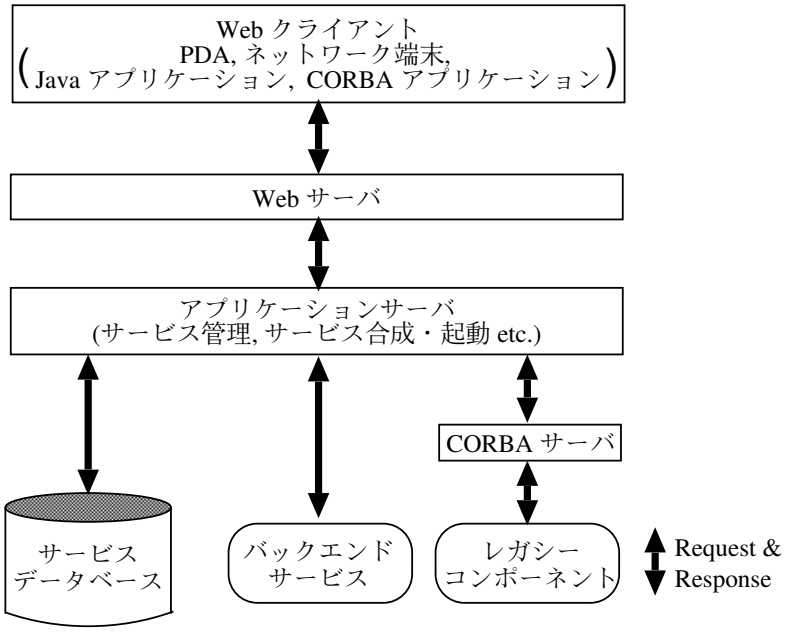

図 4.2 レガシーコンポーネントのラッピング

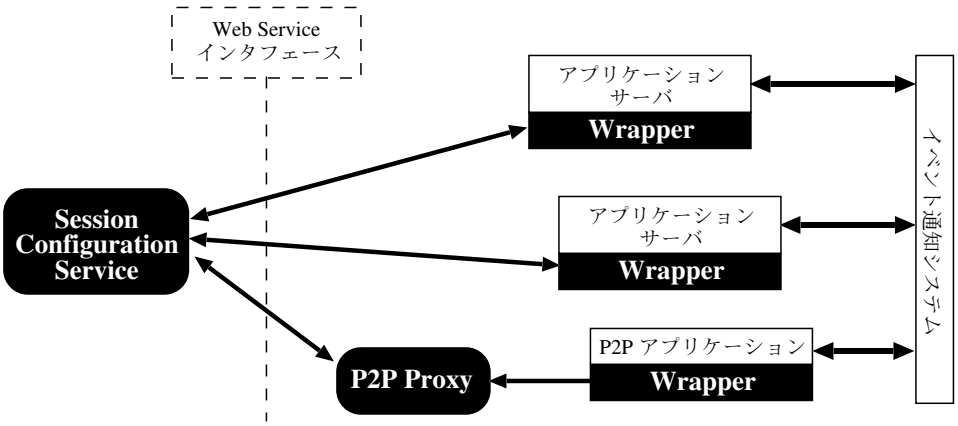

図 4.3 異種システムのラッピング

システムとは呼出し方が異なるが，P2P Proxy によってその際の異種性を吸収する．そして，Session Coniguration Service は，Wrapper や P2P Proxy を Web Service として呼び出すことによって連係させる．

上記の技術により，異種コンポーネントを相互に呼び出すことで新しいアプリケーションを開発したり，既存のシステムを組み合わせたりできるようになる．しかしコンポーネントの実装形式の異種性が吸収できても，コンポーネント間の呼出しの関係を記述したプログラムコードを記述する必要がある．同様に，コンポーネント間の結合に合わせて異なるシグネチャの形式を調整するプログラムコードを記述する必要もある．それらはすべてコンポーネントを組み合わせようとする開発者が行わなければならず，それを支援するための仕組みが提供されていない．

4.3　分散コンポーネントの動的発見

Web Service を利用するクライアントから，利用するコンポーネントの存在を意識させなくするための技術として，DySOA が提案されている [Siljee2005]．これは，利用するコンポーネントを動的に発見し QoS に応じて切替えを行う．DySOA を利用して開発したビデオ配信システムの例を，図 4.4 に示す．Streaming Video Service (以降では，SVS と略記する) と Proxy は，DySOA フレームワークを利用して動作する．SVS はクライアントからのリクエストを受け取った際に，必要なコンテンツを配信するすべてのコンポーネントと，それを提供するサーバをレジストリから探す．そして，Proxy によって QoS に応じて一番適切なサーバへ処理を仲介する．DySOA を利用することで，クライアントから呼出しを行う段階で，適切なコンポーネントをレジストリから探して利用できる．DySOA プラットフォームがコンポーネントを探すので，クライアントはコンポーネントの存在を意識する必要がない．

同様に家電製品を利用して SOA を実現するシステムとして，TOPAZ が提案されている [Kim2006]．これも DySOA と同様に，それぞれのサービスの存在を管理するためのシステムを提供し，それぞれの家電をあらかじめこのシステムに登録しておく．そして，ユーザがこれを利用することによって，必要なものを自動的に発見して利用できるようにしている．

また，ユーザのリクエストに応じて，必要な家電製品を連係させる Web Service を動的に生成して提供する，Home Appliance Integration Unit (HAIU) というシステムも提案されている [Mingkhwan2006]．図 4.5 に，Audio サービスと，Visual サービスと，Player サービスを利用して，映画を再生する Theatre サービスをユーザに提供する例を示す．HAIU では，Service Integration Controller (SIC) と呼ばれる装置を提供する．複数のサービスを連係させる際の手順は，次のようになる．

(1) あらかじめ SIC に，それぞれのサービスの存在を登録する．

(2) ユーザが SIC に対して，Theatre サービスを要求する．

(3) SIC はオントロジを利用し，Theatre サービスの実現に必要なサービスを探す．その結果，Audio サービスと Visual サービスを出力のためのサービスとして利用し，

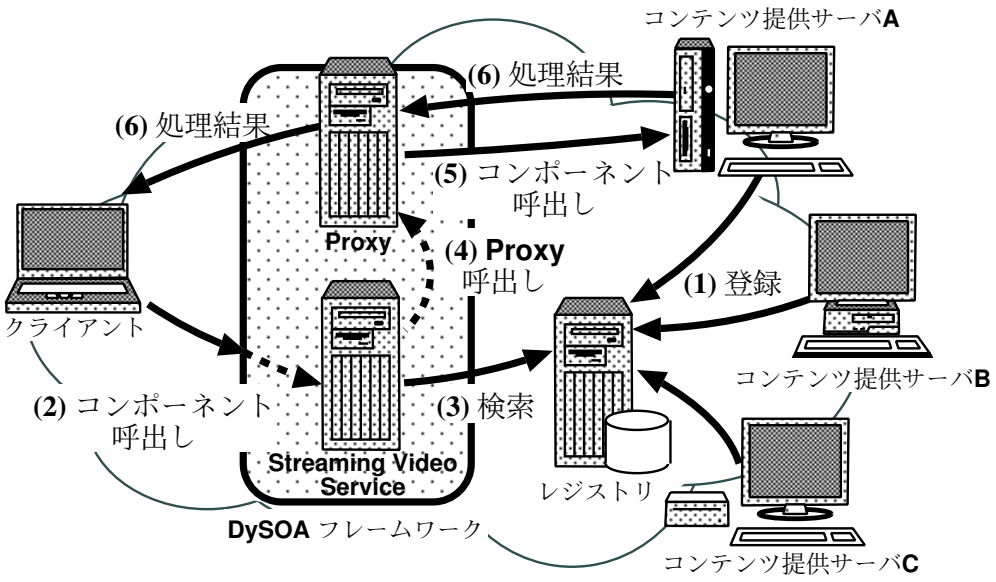

図 4.4 DySOA フレームワークを利用したアプリケーション

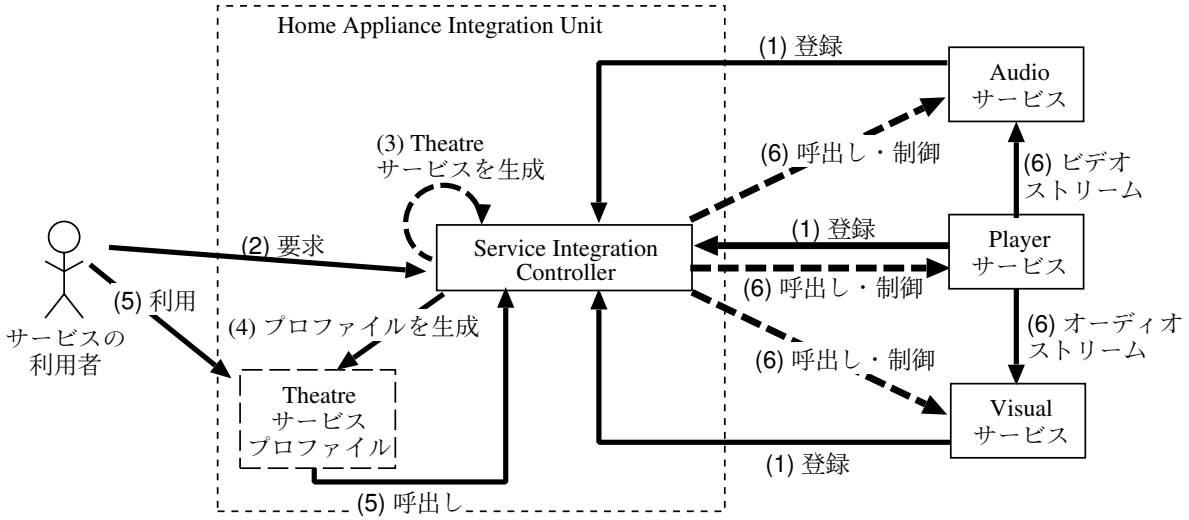

図 4.5 HAIU による家電製品の連係

Player サービスで再生すれば良いことがわかる．そしてそのようなサービスを実現するための Theatre サービスを Web Service として内部的に生成する．

(4) Theatre サービスのプロファイル情報を生成する．

(5) ユーザはプロファイル情報を利用し，Theatre サービスを利用する．

(6) Theatre サービスは，Audio サービスと Visual サービスと Player サービスを連係させて動作させる．

DySOA や TOPAZ や HAIU を利用することによって，Web Service を利用するクライアントは，利用するコンポーネントの存在を意識せずに利用できるようになる．それぞれのシステムでは家電製品を集中管理するための装置を設置し，それを利用して SOA を実現している．例えば DySOA では，DySOA プラットフォーム上で動作するシステム (DySOA の例における SVS) がコンポーネントの存在を管理している．そのため，クライアントはシステム (SVS) の存在を知る必要がある．さらにサーバを追加した場合に，追加されたサーバがレジストリを発見したり，登録したりする手段も必要になる．

しかし前述のように，Plug and Play を実現する際には，コンポーネントの存在を管理するための装置を設けず実現する必要がある．前述の提案 [Nakamura2004] に加え，P2P ネットワークを利用することによって，Web Service を提供するサーバや，それを利用するユーザすべてがサーバレスでデータ共有を行い，UDDI の代わりとして動作するシステムも提案されている [Forster2004]．システムを利用する際の全体の概要を，図 4.6 に示す．

このシステムでは，Web Service を提供するサーバと，それを利用するクライアントが，P2P ネットワークを事前に形成し，それを利用してユーザが必要な Web Service を動的に発見することができる．ユーザがある特定の Web Service を利用したいときに，ユーザの利用するクライアントは，Peer として認識している Web Service のサーバや，他のクライアントへクエリを送信する．クエリの受信側のサーバやクライアントは，誰がその Web Service を提供しているかを知っていれば，提供しているサーバに関する情報を応答する．そうでなければ，さらに他のサーバやクライアントにクエリを転送する，そしてクライアントは必要な Web Service をどのサーバが提供しているかを認識し，そのサーバを利用する．このように，P2P ネットワークを利用し，Web Service の存在を管理するための装置を設けずに，コンポーネントの存在を探索することができるようにしている．これらのことからも，集中管理する装置を設けずに実現する必要があることがわかる．

集中管理するような装置を設けずに探索を行う仕組みとして，汎用的なサービス探索プロトコルが提案されている [Goland1999] [Guttman1999]．これらは，クライアントが IP マルチキャストを利用してサーバ群にリクエストを送信することで，必要なサーバを探索する仕組みを提供している．Web Service にも，同様の方法でサーバを探索するために WS-Discovery が提案されている [Beatty2004]．これは，Web Service を利用するためのプロトコルである SOAP (Simple Object Access Protocol) を用いて探索を行う．

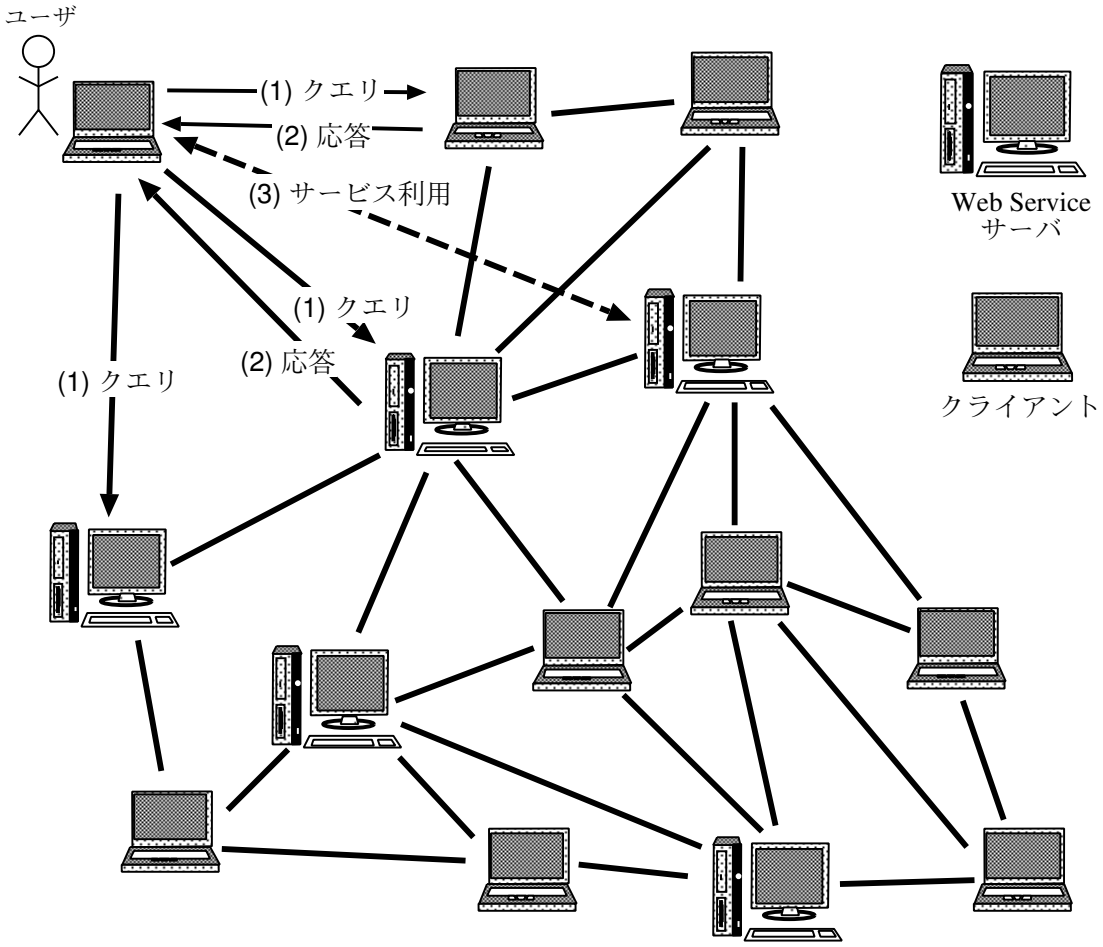

図 4.6 Peer-to-Peer ネットワークを利用した Web Service 連係

第5章　HeteroSOC フレームワークの全体設計

本研究では，3 章で指摘した課題を解決し，異種分散コンポーネントの連係を支援する機構を提案する．

5.1　提案の目的

提案する機構は，いろいろなコンポーネントを組み合わせて動作させる際の，次の課題を解決する．

(1) 組み合わせるコンポーネントを開発時に静的に決めることができる場合の課題
SOC では，特定の分散コンポーネント技術にしたがった単一の形式のコンポーネントのみならば，容易に組み合わせて，それらを連係させる新しいアプリケーションを開発することができる．しかし異なった種類の分散コンポーネント (異種分散コンポーネント) を組み合わせるアプリケーションを開発することは困難である．また，開発時に利用できたコンポーネントを運用時に利用できなくなることもある．

(2) 実行時に動的にコンポーネントを組み合わせる場合の課題
モバイルや組込み機器のような機器をサーバとして利用することを考えると，それらのサーバ上で動作するコンポーネントがネットワーク上のどこに存在するのかを，事前に想定できない場合がある．その場合は，それらを利用するように事前にアプリケーションを開発できない．そのような機器を対象にする場合，実行時に柔軟にコンポーネントを組み合わせて利用できる，いわゆる “Plug and Play” を実現することが望ましい．

本研究では，これらの問題を解決し，いろいろな種類の分散コンポーネントを組み合わせて動作させるために，“HeteroSOC (Service Oriented Computing for Heterogeneous distributed components) フレームワーク” を提案する．

5.2　解決の方針

HeteroSOC フレームワークによって，それぞれの課題を解決する際の方針は，次の通りである．

(1) 組み合わせるコンポーネントを開発時に静的に決めることができる場合の課題
異種分散コンポーネントを組み合わせるコードを開発者が直接記述しなくても，ワー

クフローに関する情報を記述するだけで自動的にアプリケーションプログラムを生成できる仕組みを提供する．またその際の記述の方法も，コンポーネント技術や実装の種類によらず，統一されたアプローチを用いる．
また，開発者が開発時に指定したコンポーネントが，運用時に利用できなくなる可能性がある．この状況に対処するために，呼び出そうとするコンポーネントの処理に障害が発生した時に，同じ機能を持つ他のコンポーネントに自動的に処理を切り替えることにより処理を続行させる方法を提供する．

(2) 実行時に動的にコンポーネントを組み合わせる場合の課題
コンポーネントを，実行時に動的にネットワーク上から探索して組み合わせることのできる実行環境を提供する．
またこの場合も，組み合わせるコンポーネントを開発時に静的に決めることができる場合と同様に，コンポーネントの異種性に起因する課題が発生し得る．しかし組み合わせるコンポーネントが開発時に決定しないので，コンポーネントの呼出しの際に，リクエストを動的に変換できないとならない．提供する実行環境では，このための仕組みも提供する．

5.3　HeteroSOC フレームワークの概要

5.2 節で述べた方針で課題を解決するために，HeteroSOC フレームワークが提供する機能を挙げる．

(a) ワークフローに基づいた開発支援システム
異種分散コンポーネントを開発時に指定することによって，アプリケーションを設計できるようにする．その際に，障害時に他のコンポーネントに処理を切り替えられるように，同じアクティビティに対して複数のコンポーネントを指定できるようにする．

(b) アプリケーションプログラムの生成
開発者による指定に基づき，異種性を吸収するアプリケーションプログラムを，自動生成する．複数のコンポーネントが同じアクティビティに対して指定されている場合には，考えられるコンポーネントのすべての組合わせについて，プログラムコードを生成する．

(c) アプリケーションプログラムの運用
生成したアプリケーションプログラムを実行する．複数のプログラムコードが生成されていた場合は，それらから一つを選択して運用する．

(d) コンポーネント発見・呼出し
アプリケーションプログラム運用中に，必要なコンポーネントを動的に発見し，呼び出す．その際の呼出しの異種性も動的に吸収する．

(e) コンポーネント障害時の切替え
アプリケーション運用時に，コンポーネントが利用できなくなった場合には，別の

コンポーネントの組合わせを利用するプログラムコードに，処理を切り替える．

HeteroSOC フレームワークを利用する際の流れを，図 5.1 に示す．
アプリケーションの開発手順は，次のようになる．

(1) アプリケーションプログラムの開発者が，サービスの組合わせに関するワークフロー情報を記述する．

(2) 開発者は，各アクティビティを実現するコンポーネントを複数指定する．

(3) ワークフロー情報を利用し，プログラムコードを自動生成する．

生成したプログラムコードは，指定された異種分散コンポーネントを呼び出すことによって一つのアプリケーションプログラムとして連係して動作させる．そして，呼び出そうとする対象のコンポーネントの呼出し形式が異なる場合に，それを変換するためのプログラムコードを含む．
また，ワークフロー内の特定のアクティビティに対して複数のコンポーネントが指定された場合は，ワークフローの開始から終了まで，考えられるコンポーネントのすべての組合わせについて，プログラムコードを生成する．
生成したアプリケーションを運用する手順は，次のようになる．

(4) 生成したアプリケーションプログラムのプログラムコードのうち，任意のものをコンパイルし，配置する．

(5) 配置したアプリケーションプログラムを，クライアントプログラムから呼び出す．

(6) アプリケーションは，開発者が指定したワークフローにしたがって，異種分散コンポーネントを呼び出して連係させる．

(7) 開発時に利用するコンポーネントを指定しなかった場合には，利用可能なコンポーネントをネットワーク上から発見して，呼び出す．

なお，手順 (7) のように異種分散コンポーネントを動的に連係させる場合，開発時に利用するコンポーネントを想定できない．そのため，異種性を吸収するためのプログラムコードを事前に生成できない．したがって，呼出しの形式を，発見したコンポーネントに合わせて動的に変換するような実行環境を提供する．
開発時に利用するコンポーネントを静的に指定した場合に，指定したコンポーネントが利用できない場合は，同じワークフローを実現する別のプログラムコードが存在すれば，それに処理を切り替える．その場合の処理手順を，次に示す．

(8) 別のプログラムコードを選択し，コンパイルして配置する．

(9) 途中まで行った処理があれば，新しく配置したものに処理を切り替える．

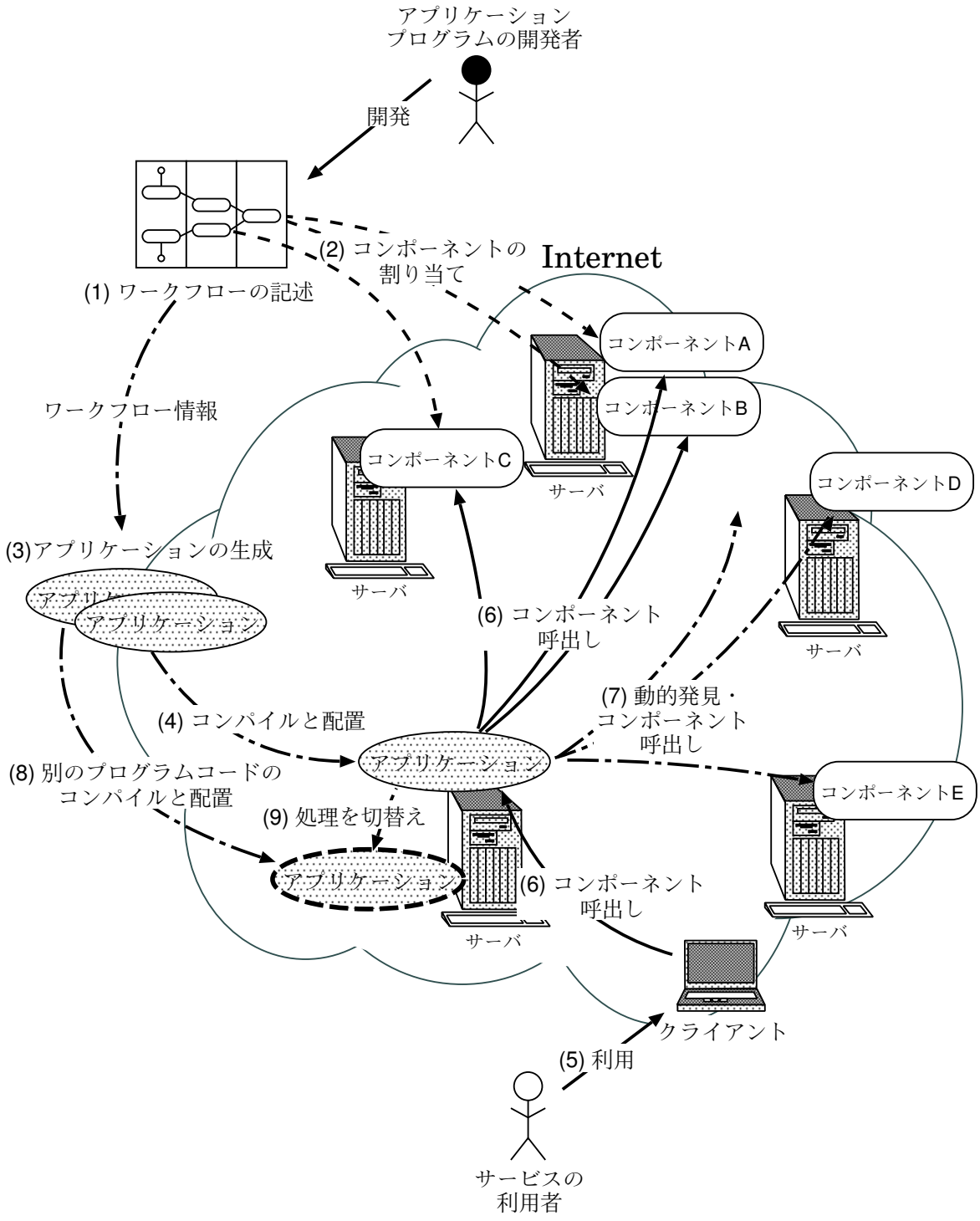

図 5.1 HeteroSOC フレームワークを利用する際の動作の流れ

5.4　想定する利用シナリオ

HeteroSOC フレームワークを構成する機能の動作イメージを明確にするために，想定する利用シナリオを例示する．

いま，ビデオ視聴アプリケーションを開発する場合について考える．このアプリケーションは，クライアントからビデオコンテンツを要求されると，家庭内ネットワークのようなローカルネットワーク内に存在するすべてのビデオサーバから，該当するコンテンツを探し，クライアントに配信する．そして該当するコンテンツがローカルネットワークのビデオサーバに存在しない場合には，要求に応じて適切なビデオ配信会社のサーバからダウンロードして，クライアントに配信する．この際のビデオサーバ，およびビデオ配信会社のサーバが提供するビデオ配信サービスを，分散コンポーネントにより実装しているものとする．

このシナリオにおいて，HeteroSOC フレームワークを利用することによって，ビデオ視聴アプリケーションを開発し，運用することを考える．全体の動作のイメージを，図 5.2 に示す．

まず，ビデオ配信会社のサーバが提供するビデオ配信コンポーネントへの呼出しについては，一般的に利用すべきコンポーネントの存在を想定して開発を行う．したがって，アプリケーションがどのコンポーネントを利用するかは，予め静的に決まっている．そのため，HeteroSOC フレームワークで提供する開発支援システムによってワークフローを記述する際に，利用するコンポーネントを指定できる．また，スポーツや，映画といったジャンルごとに，ビデオ配信会社が異なることもある．クライアントのリクエストに応じて適切なコンテンツを提供するコンポーネントを利用することを考えると，それらの異なるビデオ配信会社を組み合わせて利用する必要もあるかも知れない．そして異なったビデオ配信会社間では，異なった分散コンポーネント技術や実装を利用していることが，十分に考えられる．提案機構ではその場合に，異種性を吸収するためのプログラムコードを自動的に生成する．さらに，同じジャンルの複数のビデオ配信会社と提携していて，どちらかの会社に障害が発生した場合に，別の会社に処理を切り替えるような利用場面も考えられる．この場合も，予め提携しているそれぞれの会社のビデオ配信コンポーネントを指定することによって，障害発生時に切り替えることができるようなアプリケーションプログラムを生成できる．

一方．ビデオ視聴アプリケーションを家庭用の組込み家電機器として開発することを考えると，どの家庭内ネットワークに設置されるかを，開発時に予め想定できない．したがって．家庭内ネットワークのどこに，どのようなビデオサーバが，どれだけ設置されているかということも想定できない．例えば，設置先の家庭によっては，家族それぞれがビデオサーバを持っているかも知れないし，またある日，誰かが購入したビデオサーバを家庭内ネットワークに追加するかも知れない．したがって，予めどのサーバに存在するどのコンポーネントを利用するかを想定して開発を行うことができない．このような場合に，HeteroSOC フレームワークが提供する実行環境を利用することによって，ネットワーク内から該当するコンテンツを配信できるビデオ配信コンポーネントを動的に発見して利用するように，アプリケーションを運用することができる．そしてビデオ視聴アプリケーションは，クライアントからリクエストを受けると，ローカルネットワーク内のすべてのビデオ配信コンポーネントを探し，必要なコンテンツを配信する適切なコンポーネントを利用

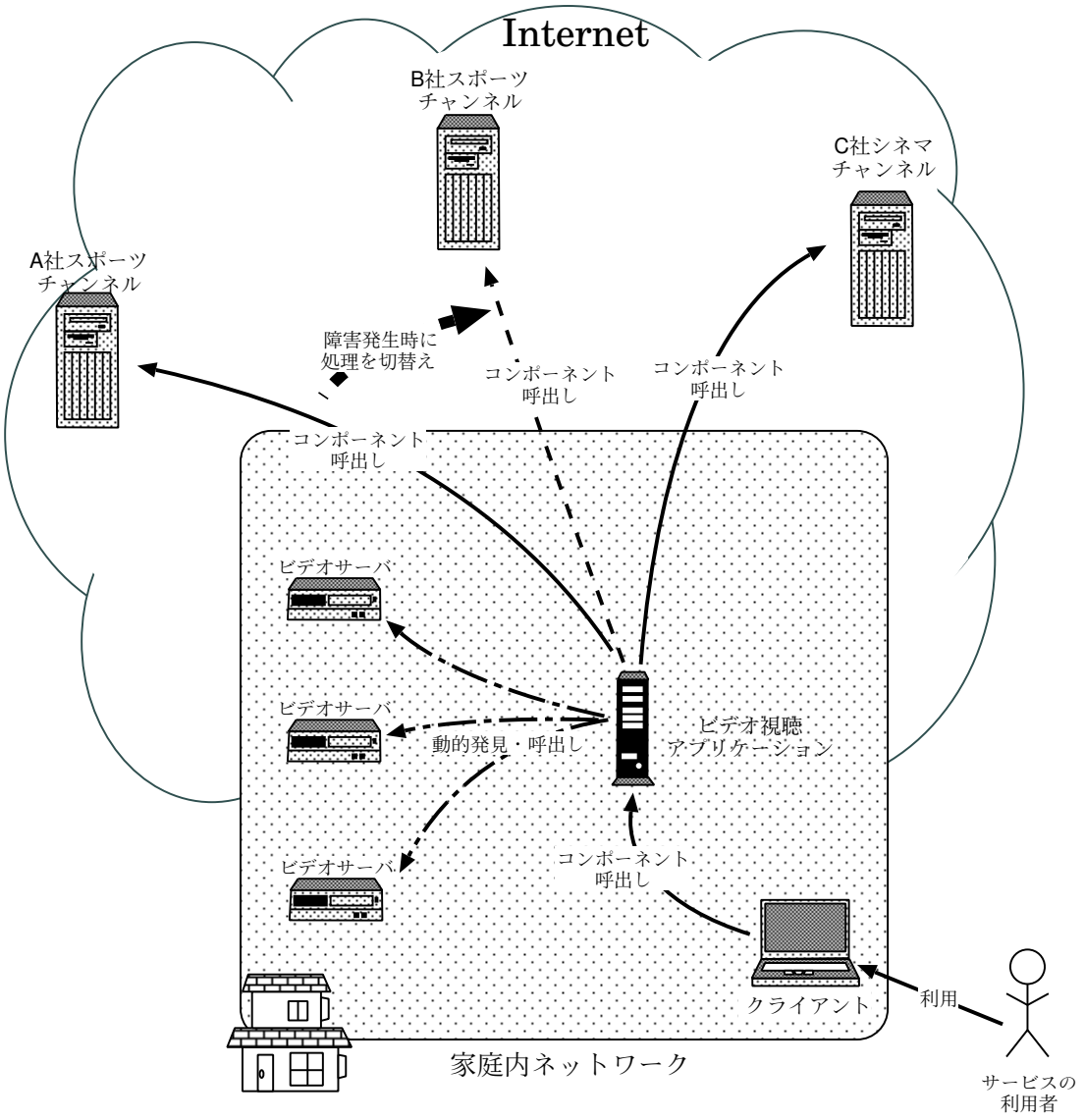

図 5.2 ビデオ視聴アプリケーションの動作

できる．

HeteroSOC フレームワークを利用することにより，このような動作をするアプリケーションプログラムを開発し，運用することができる．

5.5　HeteroSOC フレームワークの構成

本論文の残りでは HeteroSOC フレームワークを 2 つの部分に分けて述べる．

まず，アプリケーション開発時において，組み合わせるコンポーネントを開発時に静的に決めることができる場合に，異種分散コンポーネントを連係するアプリケーションの開発を支援するシステムについて，6 章で述べる．これは 5.4 節のシナリオにおける，ビデオ配信会社のコンポーネントのように，予めどのコンポーネントを利用するかを静的に決定できる場合を対象にしている．そして障害の発生時に，切り替えることができるようなアプリケーションを開発することのできる，開発支援システムである．

そして次に，アプリケーション運用時に，利用できるコンポーネントを動的に発見して，組み合わせることのできるプラットフォームについて，7 章で述べる．これは，5.4 節のシナリオにおける，家庭内ネットワークに存在するビデオサーバのコンポーネントのように，予めどのコンポーネントが存在するかわからないような環境において，Plug and Play でコンポーネントを連係させるための実行環境である．

第6章　異種分散コンポーネントを利用するアプリケーションの開発支援システム

本章では，異種分散コンポーネントを利用するアプリケーションの開発を支援するシステムの詳細を述べる，その後で評価実験を示す．

6.1　概要

本章で提案するシステムは，開発者がワークフロー情報を記述することで，異種分散コンポーネントを一つのプログラムとして連係動作させるためのプログラムコード (アダプタ) を生成する．このアダプタがそれぞれのコンポーネントを呼び出すことによって，全体として開発対象のアプリケーションプログラムとして動作する．アダプタを分散コンポーネントとして動作するように生成し，利用者には他の分散コンポーネントと同様に利用させる．

4 章で挙げた EAI システムも，同様に開発者がワークフローを記述することで分散コンポーネントの連係を行う．しかし提案システムは，3.2 節で述べた課題を解決するために，異種分散コンポーネントを連係して動作させる際に，次の事項を考慮する点で異なる．

A) ワークフローに関する情報の表現方法

B) コンポーネントを結合する方法

C) 運用を行う際に発生した障害の回避

提案システムにおけるアダプタの生成手順を，図 6.1 に示す．

(1) ワークフローの記述

(2) コンポーネントの割り当て

(3) アダプタの生成

(4) アダプタのコンパイルと配置

本章では，まずこれらの手順のうち，ワークフローを記述するための方法と，コンポーネントの割り当ての方法と，生成するアダプタについて述べる．そしてその後で実装について述べ，評価実験について述べる．

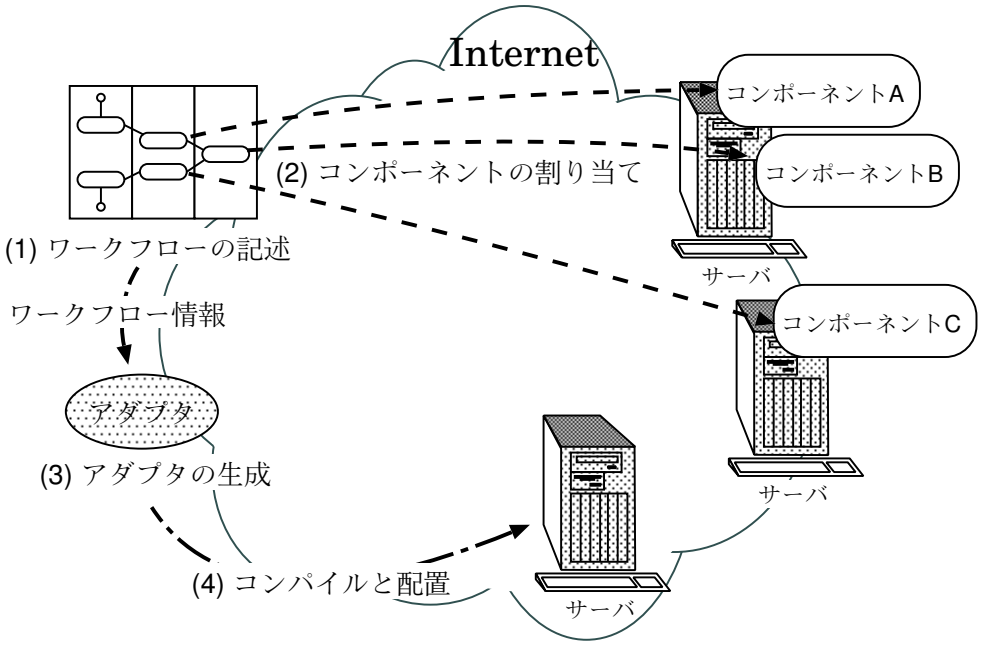

図 6.1 アダプタ生成

6.2　ワークフローを記述するための方法

まず開発者は，異種分散コンポーネントを呼び出すアプリケーションプログラムのワークフローを記述する．3.2.2 節で述べたように，ワークフローの記述は，コンポーネント技術や実装の種類によらず，統一されたアプローチであることが望ましい．提案システムはそのために，UML のアクティビティ図 [Dumas2002] を利用する．

アクティビティ図は，ワークフローを表現するために用いられる UML 図である．アクティビティ図ではひとまとまりの業務や処理を表すために，関連する複数のアクティビティを時間的に順序だてて表現する．アクティビティはそれぞれが業務や処理を構成する一単位に相当する．あるアクティビティの実行終了がトリガーになり，次のアクティビティへの遷移が発生する．このため, UML の状態遷移図の特殊な状態と考えることもできる．図 6.2 にアクティビティ図の例を挙げる.

アクティビティ図を構成する要素は，次の通りである．

A) 初期状態／終了状態
　ワークフローの開始／終了を示す.

B) アクション状態
　アクション状態は業務や処理といったアクティビティを表す．アクティビティの完了にともない次の状態 (終了状態を含む) に遷移する．

C) 遷移
　各状態間を結ぶ実線の矢印によって表される．遷移は複数の状態に対して行われることも，次に述べるガード条件により選択的に行われることもある．

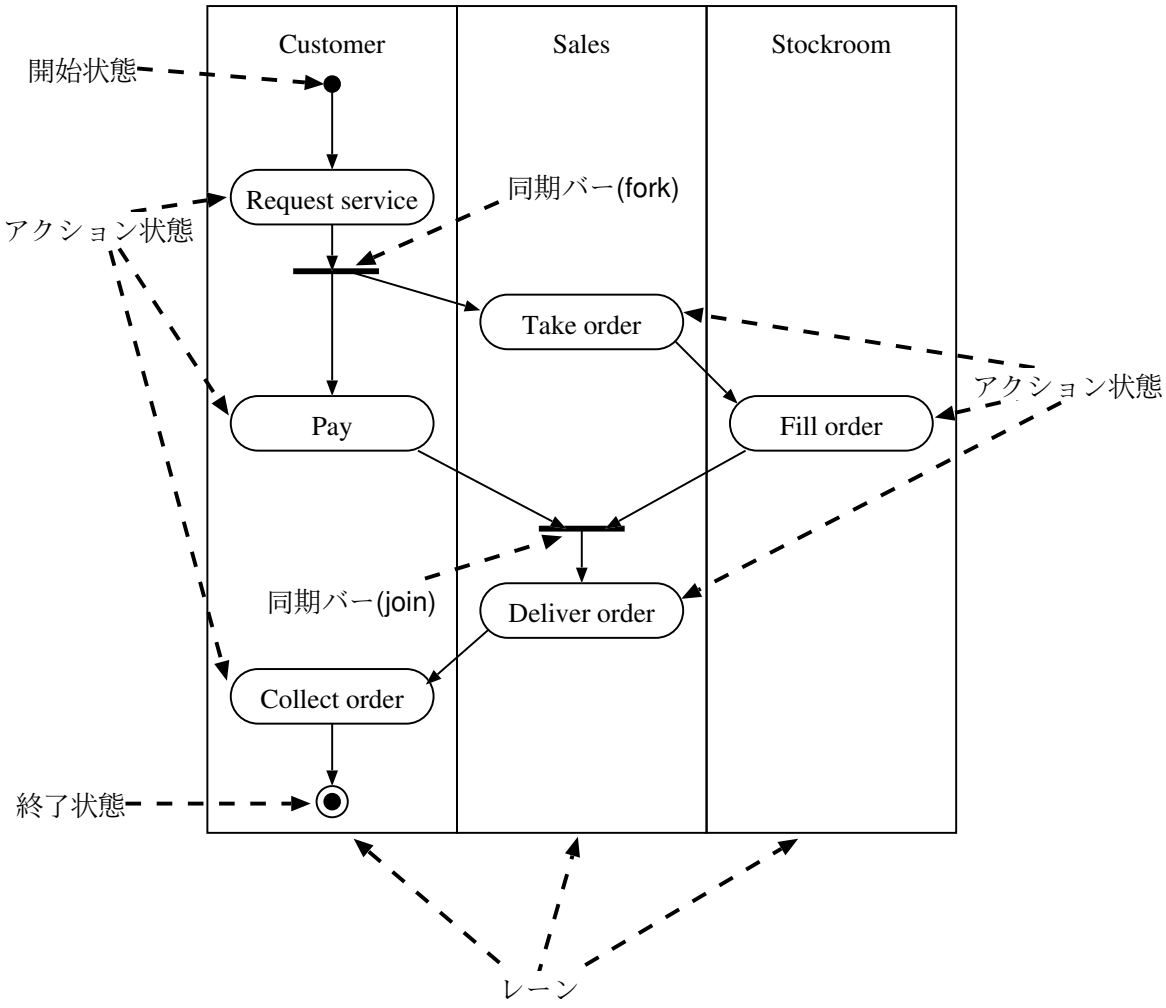

図 6.2 アクティビティ図の例

D) ガード条件
遷移を引き起こすための条件を表す．真偽値式によって表現される．各条件は排他的に記述され，真の値を取る条件に適合した遷移のみが行われる．各ガード条件は遷移を表す矢印に付与される．

E) 条件判断
ガード条件により指定される条件分岐を表す．一つの入力状態と複数の出力状態を持つ．条件判断はひし形の図形により表される

F) 同期バー
遷移の同期を表す．フォーク (fork) とジョイン (join) の二種類がある．フォークは複数状態への同時遷移であり，一つの入力遷移と複数の出力遷移がある．入力遷移がトリガされるとすべての出力遷移が並列に実行される．ジョインはワークフローのある時点での同期を取るための要素であり，複数の入力遷移と一つの出力遷移がある．複数遷移の結果を集めてそれらの値を元に次のアクションへと遷移する場合に用いられる．

G) レーン
各アクティビティの実行責任主体を表す．そしてそれらの間の相互作用を明示するために利用される．レーンは垂直な実線を持って他のレーンと区別される．それぞれのアクション状態は一つのレーンに割り当てられる．

アクティビティ図は，ワークフローを記述するための記法として一般的に普及しているため，開発者は本システムを比較的容易に利用できることが期待できる．提案システムでは，このようなアクティビティ図の要素を用いて記述できるインタフェースを提供する．

6.3　コンポーネントの割り当てとワークフロー情報の表現

ワークフローを記述した後，開発者はそれぞれのアクティビティに，実際のコンポーネントを対応づける．それらの各コンポーネントは，ネットワークを介して呼び出すことができれば，離れた場所のサーバに存在しても良い．

また，各アクティビティはそれぞれ一つのコンポーネントによって実現されるかも知れないし，複数のコンポーネントを連続して呼び出すことによって実現されるかも知れない．そこで，図 6.3 に示すように，一つのアクティビティに複数のコンポーネントの組合わせを指定できるようにする．本稿ではこれを Refinement と呼ぶ *.

コンポーネントの割り当てが完了すると，ワークフローに関する情報を文書として出力する．これは 3.2.2 節で述べたように，汎用的で統一された記述形式で可読文書として表すことが望ましい．本システムでは，開発者によって記述されたワークフローに関する情報を，開発者の選択により，複数の Web Service を連係させる際に，そのワークフローを記述するための XML ベースの言語である WSFL (Web Services Flow Language) 仕様 [Leymann2001]，または BPEL4WS (Business Process Execution Language for Web Services) 仕様 [Jordan2006] のどちらかにしたがった文書として出力する．

*なお，[Pokraev2006] において，同様のアイデアが提案されている．

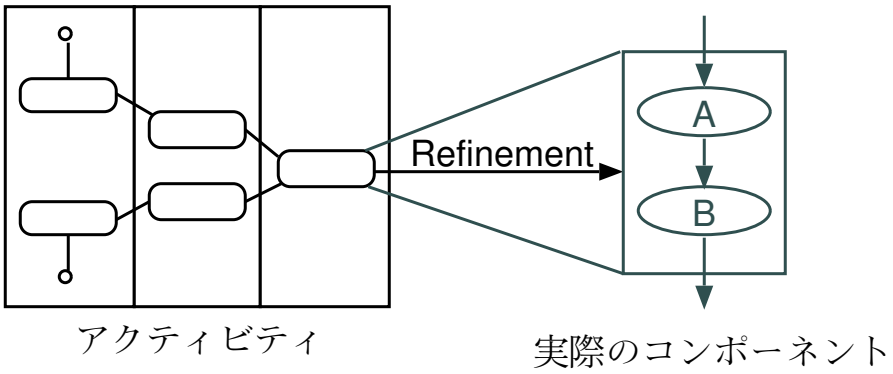

図 6.3 Refinement

6.4　生成するアダプタの構成

コンポーネント間の異種性を吸収し，ワークフローにしたがって各コンポーネントを呼び出すためのアダプタを生成する．

生成するアダプタを，その機能によって次の二つの部分に分ける．

A) Proxy

3.2.1 節で指摘したコンポーネント間の異種性に起因する問題に対処するために，呼出しの異種性を吸収する．そして，各分散コンポーネントごとに，対応する Proxy を生成する．それぞれの Proxy はコンポーネントが利用する分散コンポーネント技術に合わせた呼出し方法でアクセスを行い，各プロセスを実行し，その結果を得る．これにより，異種分散コンポーネントを利用する際に，呼出しごとに異なるアクセス方法を吸収する．連係させる分散コンポーネントの種類を増やす場合は，Proxy の種類を増やすことで対応できる．

B) Manager

与えられたワークフロー情報に基づき，各分散コンポーネントの実行順序やアプリケーションプログラムの実行の遷移を管理する．そして適切なコンポーネントに Proxy を介してアクセスする．異なるコンポーネントを連係させる場合は，対応する複数の Proxy を呼び出す．またその際に，パラメータの順序や型が異なる場合は変換する．

アプリケーションプログラムごとに，ひとつの Manager と呼び出そうとするコンポーネントに対応する複数の Proxy を生成することによって，複数の異種分散コンポーネントを連係させる新しいアプリケーションを開発する．生成されたアプリケーションプログラムにおける Proxy と Manager との関係を図 6.4 に示す．

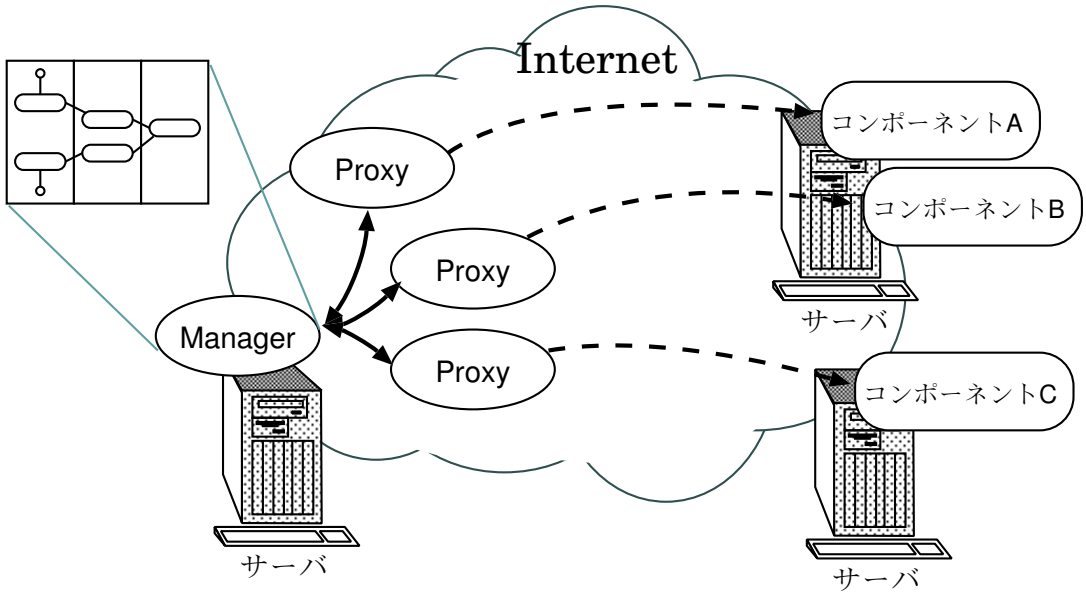

図 6.4 Proxy と Manager の関係

6.5　アプリケーションプログラムの運用の問題を解決する方法

本システムでは，呼び出そうとするコンポーネントの処理に障害が発生した時に，同じ機能を持つ他のコンポーネントに自動的に処理を切り替えることにより処理を続行させる方法を提供する．そのために，まずアプリケーションプログラムの設計時に，開発者がワークフローエディタを用いて各アクティビティを実現するコンポーネント (または Refinement によるコンポーネントの組合わせ) を複数指定する．そしてワークフローの開始から終了まで，考えられるコンポーネントのすべての組合わせについてアダプタを生成する．アプリケーションプログラムの実行時には，このうちから任意のアダプタを実行する．アダプタから呼び出すコンポーネントが利用できなくなった時には，同じワークフローを実現する別のアダプタに実行を切り替える．この手順を，図 6.5 に示す．本稿ではこれを Variation と呼ぶ．このように，実行不可能な場合に代替として動作するアダプタを予め用意し，実行不可能になる確率を低くすることで，信頼性の低下を回避する．

なお，アプリケーションプログラムを設計する時点で利用できるコンポーネントが，代替として実行される時点では利用できなくなっている可能性がある．代替として動作するアダプタの生成は，本来はアダプタの実行に失敗した段階で，その時点で利用できるコンポーネントを呼び出すように行うべきである．しかし，開発者によるコンポーネントの割り当てに基づいて，アダプタ生成が行われるため，アダプタの実行に失敗した段階でアダプタの生成を自動的に行うことはできない．そのため，代替として利用できなくなる可能性があるものの，設計を行う段階で開発者に利用できるすべてのコンポーネントを指定させることによって，代替として動作するアダプタを予め生成している．

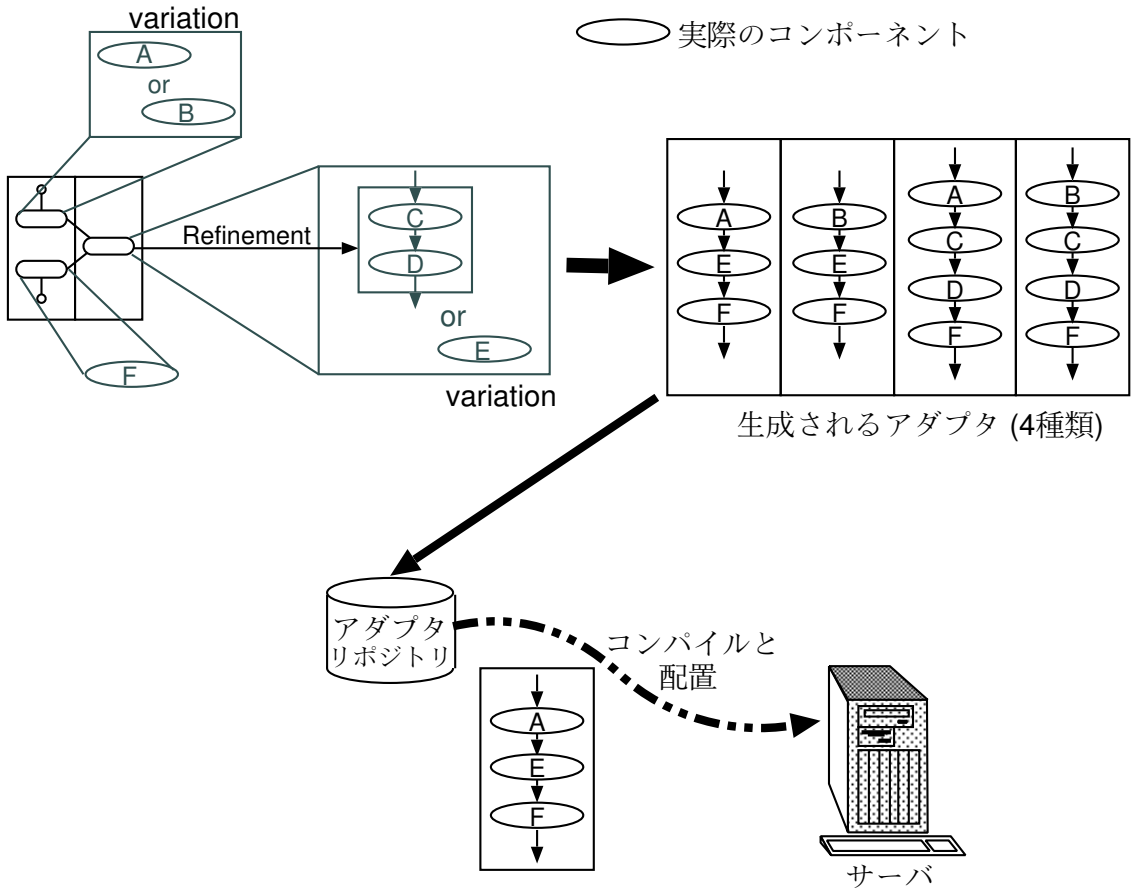

図 6.5 Variation を考慮したアダプタ生成

6.6　実装

本節では，本章で提案する異種分散コンポーネントを利用するアプリケーションの開発を支援するシステムの実装の詳細について，それぞれの構成要素の動作を通して述べていく．

6.6.1　構成

実装したシステムは，EJB と，Web Service と，CORBA の三種類を対象として連係させる．なお他の種類のコンポーネント技術で実装されたコンポーネントに関しても，モジュールを追加することで組み合わせることができるように実装した．

提案システムの実装の構成要素は，次に挙げる．

A) ワークフローエディタ
　開発者がワークフローの記述を行うためのインタフェース

B) アダプタ生成部
　ワークフローエディタで記述されたワークフローにしたがって，アダプタを生成する部分

C) アダプタリポジトリ
　生成したアダプタを登録するリポジトリ

D) アプリケーションプログラム運用部
　生成したアダプタを配置し，運用を行う部分．Web Service のサーバ [Apache2006] や BPEL エンジン [ActiveBPEL2006] と，アダプタを配置するモジュール (配置モジュール) と，サーバの実行状況を監視するモジュール (実行監視モジュール) から構成する．

これらの構成要素と，それを利用する際の流れを，図 6.6 に示す．

本節では，それぞれの構成要素の動作を通して，実装の詳細を述べていく．

6.6.2　ワークフローの記述

6.2 節で示したように，提案システムでは UML アクティビティ図により，統一された記法で設計を行えるようにする．そのためワークフローエディタでは，UML アクティビティ図の要素を用いて設計できるようなインタフェースを提供する．ワーク不ローエディタ上で UML アクティビティ図の要素を用いて設計している様子を，図 6.7 に示す．

コンポーネントを連係させるアプリケーションの開発者は，このエディタ上で UML アクティビティ図のそれぞれの要素を配置することによって，設計を行う (図 6.6-(1))．

そして要素の配置が終わると，それぞれのアクティビティと，実際にネットワーク上に存在するコンポーネントを関連づける (図 6.6-(2))．その際に，本実装では，コンポーネントを検索するための機構 [Zim2005] を利用することで，開発者が入力したクエリを基にコンポーネントをリポジトリから検索できるようにしている．6.3 節や 6.5 節で述べたよ

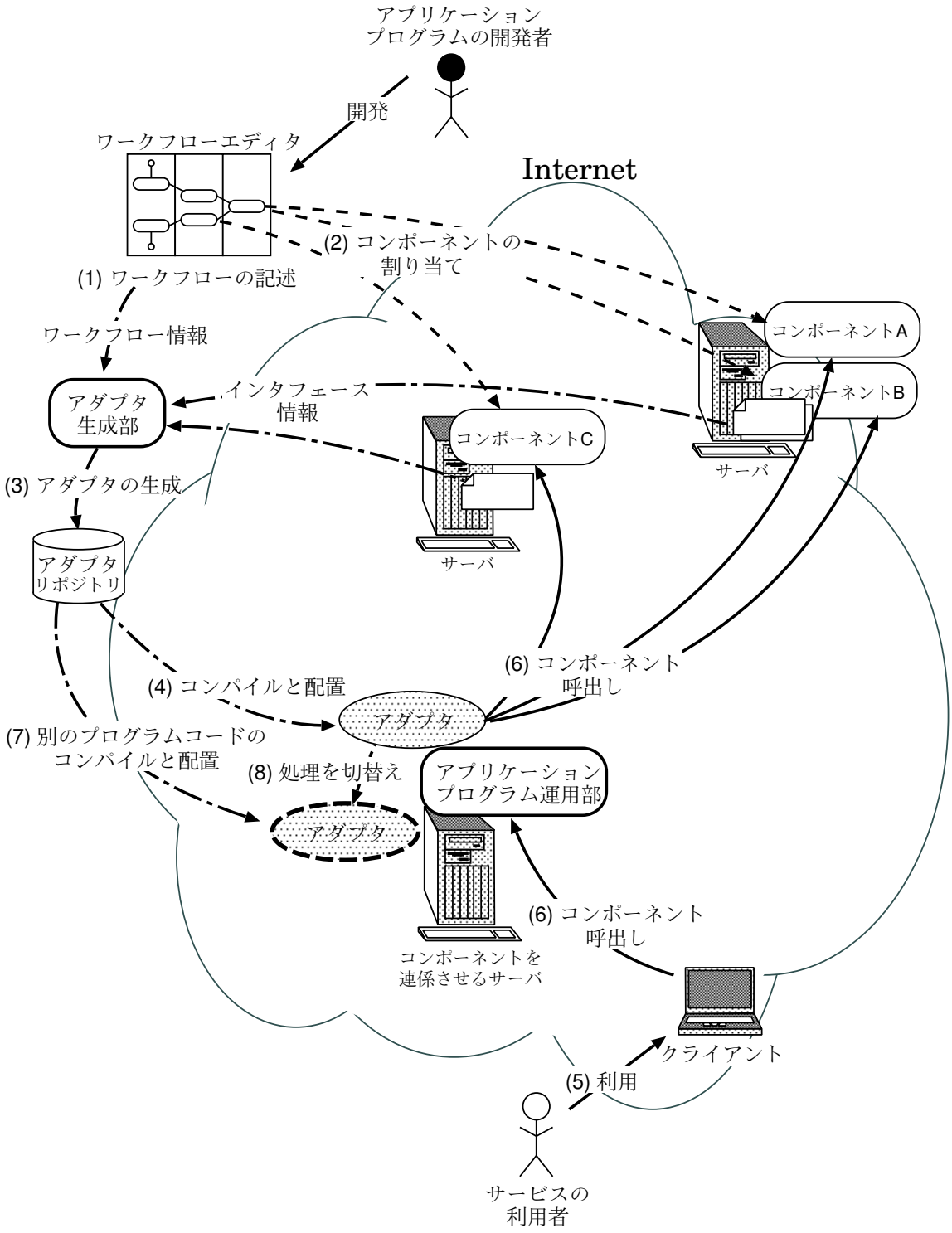

図 6.6 異種分散コンポーネントを利用するアプリケーションの開発を支援するシステム

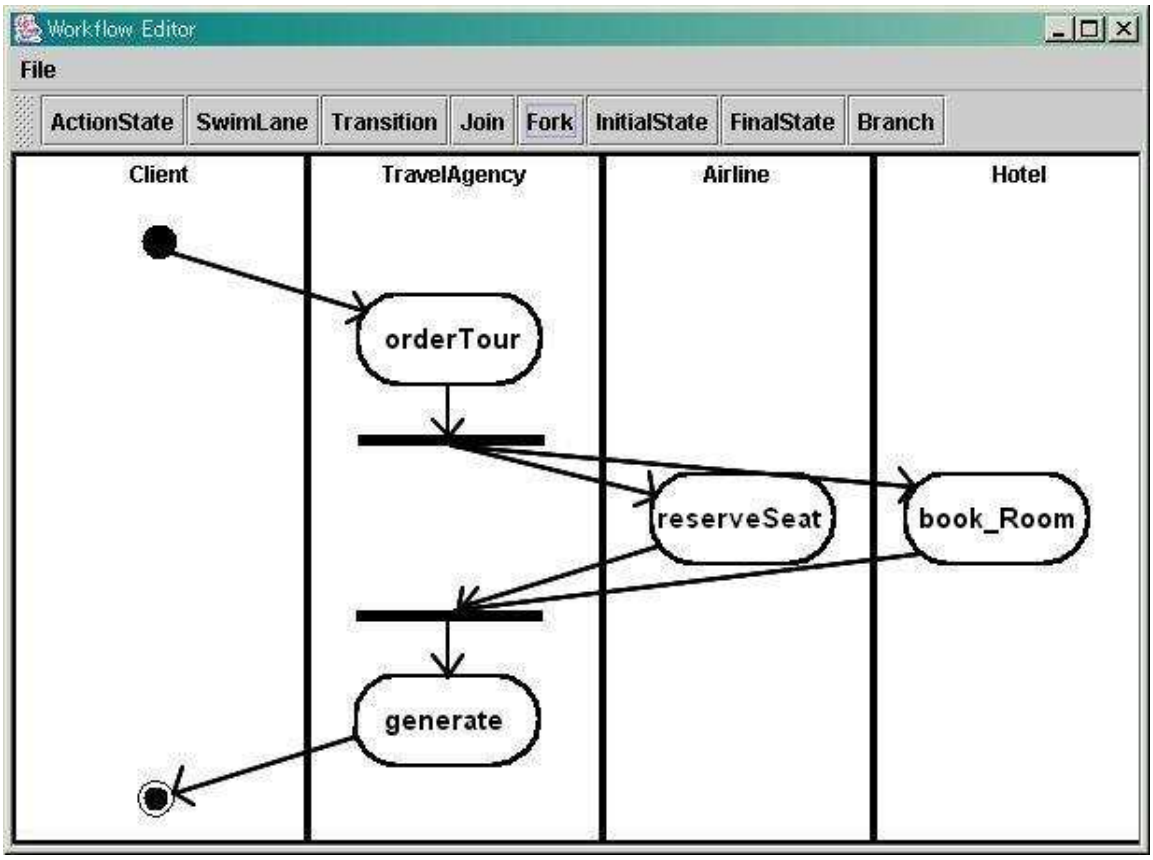

図 6.7 ワークフローエディタ

うに，Refinement や Variation を考慮して，一つのアクティビティに複数のコンポーネントを指定する場合もある．

アプリケーション開発者によって記述されたワークフローを，6.3 節で述べたように，開発者の選択により WSFL や BPEL4WS で記述された文書として保存する．アクティビティ図から WSFL や BPEL4WS への変換の詳細を，付録 A に示す．この際に Variation によって複数のコンポーネントが割り当てられているアクティビティがある場合は，ワークフローの開始から終了まで，考えられるコンポーネントのすべての組合わせについて，保存する．

6.6.3　アダプタプログラムの生成

アダプタ生成部は，開発者によって記述されたワークフローを元に，異種分散コンポーネントを連係させるためのアダプタプログラムを生成する (図 6.6-(3))．既存の EAI システムでは一般的に，ワークフローに関する情報を解釈することによって，複数のコンポーネントを呼び出す独自のサーバを用意している．これに対して提案システムでは，Web Service のサーバ [Apache2006] や BPEL エンジン [ActiveBPEL2006] などの汎用的なコンポーネントサーバのサーバ実装を，拡張せずに運用できるようにする．そのため提案シ

ステムでは，コンポーネント間の異種性を吸収し，それらを連係させる処理を，分散コンポーネントとして動作するプログラムコードの形式で生成する．

A) Proxy の生成
開発者が割り当てたすべてのコンポーネントについて，各コンポーネントのインタフェース情報を入手することによって生成する．インタフェース情報は，Web Service で一般的に利用されている WSDL (Web Service Definition Language) [Christensen2001] を利用して記述する†．そして，コンポーネント間の実装形式の異種性を吸収し，単一の形式の分散コンポーネントとして呼び出せるように，Proxy を生成する．そのために，本実装では Proxy を Web Service として生成する．すなわち Proxy を利用することによって，それぞれのコンポーネントを Web Service 化できるようにする．なお Refinement や Variation により，一つのアクティビティに複数のコンポーネントを割り当てた場合は，それらそれぞれについて Proxy を生成する．

Proxy には，呼び出そうとするコンポーネントのメソッドと同一のシグネチャを持つ Web Service のメソッドを用意する．そしてそれらの中に，それぞれのコンポーネント固有の呼出し手順を記述する．また，各メソッド内でコンポーネントを呼び出す際には，呼出し前に各コンポーネントを実行できるかどうかを確認する．そのために必要なプログラムコードも自動的に生成する．

B) Manager の生成
ワークフローの記述に従い生成される．ワークフローエディタによってワークフローを指定した際に，Variation によって複数のコンポーネントが割り当てられていた場合は，ワークフローの記述も複数生成されている．その場合は，それぞれについて Manager を生成する．そして外部から呼び出すことのできる分散コンポーネントとして生成する．本実装では Proxy と同様に，Manager を Web Service として呼び出すことができるように生成する．そして利用者が Manager を呼び出すことで，複数の異種分散コンポーネントを呼び出した結果を得られるようにする．

ワークフローに関する情報が WSFL で記述されている場合は，記述された順番にしたがって各分散コンポーネントを呼び出す Web Service のプログラムコードを生成する．この際に，実際に各コンポーネントを呼び出すのではなく，対応する Proxy を呼び出すようにする．BPEL4WS で記述されている場合は，BPEL 実行エンジン [ActiveBPEL2006] を用い BPEL4WS の記述をそのまま解釈・実行することで Web Service として動作させることができる．したがって，BPEL4WS によるワークフローの記述を各コンポーネントに対応する Proxy を呼び出すように変更することで，そのまま Manager として利用する．

なおコンポーネント間の連係では，ある処理の結果を別の処理の入力として単純に結びつけることはできず，値自体を加工することが必要になる場合がある．そこで，コンポー

†なお WSDL では，EJB や CORBA のコンポーネントのインターフェース情報を記述できない．このため，付録 B.1 に示すように，拡張を行った．

ネント間で値を変換する必要がある場合は，データのマッピングを行うコードを Manager に記述する．同時に必要なビジネスロジックも，Manager に追加的に記述できる．

また，コンポーネントへの関連付けが行われていないアクティビティに関しては，どのコンポーネントを呼び出せば良いのかがわからないため，Proxy や Manager を生成できない．この場合は，アプリケーションプログラムの運用時に，7.4 節に述べるミドルウェアによって，運用時に動的にコンポーネントを探索して呼び出す．その場合は，コンポーネントのタイプを示す文字列を開発者に指定させ，Manager にその内容を記述する．このコンポーネントのタイプの記述とその扱いについては， 7.4.2 節で述べる．

なお本実装では，Proxy を Web Service として生成し，各コンポーネントを Web Service 化している．Proxy に対して SOAP による HTTP アクセスが可能なので，一般的なファイアウォールを介して呼び出すことが可能になる．ネットワーク的に離れた場所に存在するコンポーネントを呼び出す際に，連係させるアプリケーションプログラムが動作するサーバとコンポーネントの動作するサーバの間にファイアウォールが存在する場合でも，Proxy をコンポーネントを提供するサーバ側に設置することで対応できるようにしている．

そして生成した Proxy や Manager をアダプタリポジトリに登録する．

6.6.4　アダプタのコンパイルと配置

アプリケーションプログラム運用部は配置モジュールを利用して，実行するアプリケーションプログラムに対応するアダプタをアダプタリポジトリから入手する．開発時に Variation により一つのアクティビティに複数のコンポーネントが指定されていた場合は，複数のアダプタから一つを選択する必要があるが，どのような優先順位で選択するかという基準を，開発者が予め配置モジュールに指定できるようにした．これにより，例えば組み合わせて実行するコンポーネントの実行時間や利用するための価格情報などを基に，複数のアダプタから適切なものを順番に選択できるようにしている．

そして配置モジュールは，取得したアダプタのうち Proxy をコンパイルして Web Service のサーバに配置する (図 6.6-(4))．Manager については，Web Service として実装されたコードが生成されている場合は，Proxy と同様にコンパイルして Web Service のサーバに配置する．BPEL4WS で記述されている場合は，BPEL エンジンによって解釈，実行するため，そのまま BPEL エンジンに渡す．BPEL エンジンは渡された文書を解釈して，Web Service のコンポーネントを生成し，自動的に Web Service のサーバに配置する．これにより，利用者がクライアントから利用できるようになる．

6.6.5　アプリケーションプログラムの実行

利用者は，Web Service のクライアントプログラムを介して，通常の Web Service と同様の手順で生成したアプリケーションプログラムを呼び出す (図 6.6-(5)，(6))．クライアントからのリクエストは，BPEL エンジンや Web Service のサーバが受け取り，該当するアプリケーションプログラムの Manager を起動する．Manager は，Proxy を介して，それぞれのコンポーネントを呼び出す．この際に呼出しごとに異なるアクセス方法は Proxy によって吸収し，Manager はすべてのコンポーネントを Web Service として呼び出す．

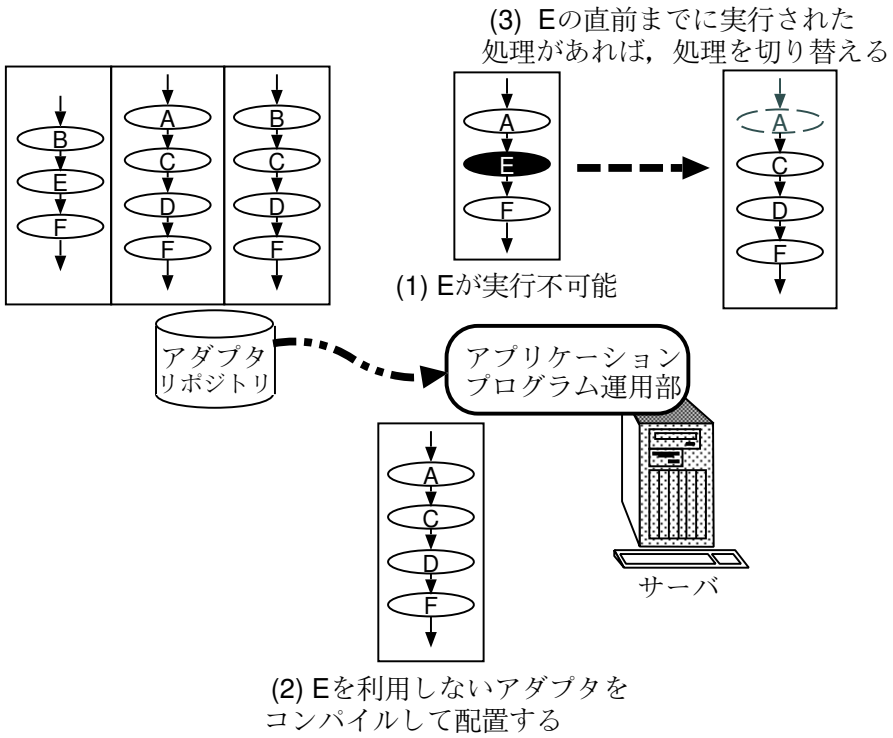

図 6.8 アダプタの切替え

6.5 節に示したように，実装したシステムでは，呼び出そうとするコンポーネントの処理に障害が発生した時に，同じ機能を持つ他のコンポーネントに自動的に処理を切り替えることにより処理を続行させる方法を提供する．そのため，Proxy を介してコンポーネントを呼び出す際に，その呼出しの状況を実行監視モジュールが監視する．そしてコンポーネントの呼出しに障害が発生した場合は，配置モジュールが別のアダプタのプログラムコードをコンパイルし，配置する (図 6.6-(7))．そして実行中の処理が存在すれば，その処理の実行を新しいアダプタに切り替える (図 6.6-(8))．この手順については，次節で述べる．

6.6.6　アダプタの切替え

図 6.5 で生成した 4 種類のアダプタを利用した場合を例に実装したシステムにおける切替えの手順を，図 6.8 に示す．

(1) コンポーネントが実行不可能であることの検知
Manager はコンポーネントを呼び出す前に，呼び出そうとする各コンポーネントを実行できるかどうか確認する．これは Proxy に生成したメソッドを呼び出すことによって行われる．そして実行監視モジュールは Proxy の処理を監視している．確認のためのメソッドの実行や，コンポーネント自体の実行に失敗した場合は，配置モジュールに通知する．

(2) 代わりのアダプタのコンパイルと配置

配置モジュールは，6.6.4 節の手順と同様に，実行しているアプリケーションプログラムに対応するアダプタをアダプタリポジトリから入手する．開発時に Variation により一つのアクティビティに複数のコンポーネントが指定されていた場合は，アダプタリポジトリに代替として動作させることのできるアダプタが存在する．配置モジュールはそのようなアダプタを，6.6.4 節と同様に，開発者が予め指定した優先順位に基づいて，アダプタリポジトリから取得する．その際に，実行できなかったコンポーネントを利用するアダプタを，選択しない．そして選択したアダプタをコンパイルし，今までのアダプタの代わりに配置する

(3) 処理の切替え

Manager は，ワークフローによって指定された順にコンポーネントを呼び出す．このため，クライアントからの一回の呼出しのセッションで，Manager は複数のコンポーネント (実際にはコンポーネントに対応する Proxy) を指定した順番で呼び出す．Manager の処理の過程で，いくつかのコンポーネントを呼び出した後で，あるコンポーネントの呼出しに失敗した場合には，すでに呼び出したコンポーネントの処理を適切に扱わなければならない．

アダプタの実行が不可能になったことを実行監視モジュールが検知すると，実行監視モジュールは該当するアダプタによるすべてのコンポーネントへの呼出しの状況を，クライアントからの呼出しのセッションごとに確認する．そして，呼出しに失敗したコンポーネント以外に，他のコンポーネントへすでに呼出しを行っていたセッションが存在する場合は，その処理を継続できるアダプタを探し，処理を継続させる．

まず実行監視モジュールは，それぞれのセッションごとに，呼出しの状況を配置モジュールに通知する．配置モジュールは，(2) での手順と同様に，実行しているアプリケーションプログラムに対応するアダプタのうち，実行できなかったコンポーネントを利用しないアダプタをアダプタリポジトリから取得する．その際に，実行不可能になった Manager のワークフローの処理の開始から，実行ができなかったコンポーネントの呼出しの直前まで，すべて同じコンポーネントを呼び出すものを選択する．そのようなアダプタが見つからない場合は，さらに一つ前のコンポーネントの呼出しまでが同一であるアダプタを探す．これを繰り返し行い，アダプタが見つかった場合は，そのアダプタからの呼出しの残りの処理を配置モジュールが実行し，処理結果をクライアントに返す．そのようなアダプタが発見できずに，繰り返しによってワークフローの初期状態まで達した場合は，(2) の手順で配置したアダプタを実行し，結果をクライアントに返す．

6.7　評価実験

実装したシステムによって生成されたアプリケーションを，利用する場合に要するオーバヘッドを測定した．また，コンポーネントの障害時にアプリケーションプログラムの切替えを行うが，その際にどのくらいの時間を要するのかを測定した．本節では，それぞれの測定手順とその結果を示し，その後で分析を行う．

6.7.1　アダプタによるコンポーネント呼出しのオーバヘッドの測定

実装したシステムは，異種分散コンポーネントを呼び出すために，アダプタと呼ばれるプログラムコードを生成する．そして，アダプタを介して分散コンポーネントを呼び出す．

いま，コンポーネント間の連係に BPEL4WS を利用することを考えると，通常の Web Service を連係させるアプリケーションと，実装したシステムによって生成したアプリケーションとの相違は，それぞれのコンポーネントを Proxy を介して呼び出す点のみである．そこで，生成した Proxy を介してコンポーネントを呼び出す際のオーバヘッドを計測した．

連係する対象とした Web Service，CORBA，EJB の各コンポーネントについて，同一の処理を行うメソッドを作成し，直接呼び出す時に要する時間と，Proxy 経由で呼び出す際に要する時間を計測した．

実験は，100 BASE-TX の同一ネットワークに接続された端末で行った．それぞれのコンポーネントと，生成した Proxy を動作させた端末の構成を，それぞれ次に挙げる．

A) Web Service:
(CPU) UltraSPARC-IIe 500MHz (OS) SunOS 5.8
(Web Service サーバ実装) Apache Axis 1.1

B) CORBA:
(CPU) UltraSPARC-IIe 550MHz (OS) SunOS 5.8
(CORBA サーバ実装) JacORB 2.1

C) EJB:
(CPU) UltraSPARC-IIe 500MHz (OS) SunOS 5.8
(EJB サーバ実装) Java 2 Platform, Enterprise Edition SDK 1.3.1

D) Proxy (Web Service):
(CPU) Pentium Mobile Processor 1GHz (OS) Windows XP Professional Edition
(Web Service サーバ実装) Apache Axis 1.1

なお，アプリケーションプログラムを実行するコンピュータと各コンポーネントが動作するコンピュータを同一ネットワーク上に設置し，ネットワークトポロジによって呼出し時間へ影響を与えないようにする．また，アダプタからコンポーネントを呼び出す前には，コンポーネントを実行できるかどうかを確認する必要がある．そして Proxy には確認を行うためのメソッドを生成している．アダプタ生成部によりこのメソッドには，色々な方法を実装できる．コンポーネントが実行できることの確認を確実に行うためには，対象のコンポーネントサーバを実際に呼び出すことができるのかを，事前に接続して確認する必要がある．しかしそのような方法では，コンポーネントの呼出し前に名前解決が行われてしまう上に，コンポーネントを実行するサーバの実装によってはサーバ側でコンポーネントがキャッシュされてしまう．これによって呼出し時間にばらつきが生じてしまい，測定結果の比較が困難になるため，ここではコンポーネントを実行できるかどうかを，ネットワーク的な到達性があるかどうかのみによって確認するように，Proxy を生成した．なお，各コンピュータは十分に高速な同一ネットワークに接続されており，到達性の確認のために必要とする時間は非常に短く，測定精度を考慮すると有意な値にはならなかった．

それぞれ 30 回測定し，その平均を表 6.1 に示す．

表 6.1 メソッドの呼出しに要する時間 (単位: ms)

WebService		CORBA		EJB	
直接	アダプタ	直接	アダプタ	直接	アダプタ
3991.6 (σ=8.7)	4023.4 (σ=11.4)	9.8 (σ=0.4)	44.2 (σ=0.7)	132.6 (σ=50.4)	165.2 (σ=57.0)
+31.8		+34.4		+32.6	

6.7.2　アダプタの切替えに要する時間の測定

アプリケーションプログラム運用部は，アダプタが呼び出すコンポーネントが実行不能であることを検知した場合に，同じワークフローを実現する別のアダプタに処理を切り替える．実行不能であることを検知してから切替えが終了するまではリクエストを受け付けられない．その間に要する時間を測定した．

測定のため，同一ネットワークに存在する別々のコンピュータに 4 つのコンポーネントを配置した．構成を 図 6.9 に示す．内訳は，EJB のコンポーネントが 1 つ (E_1) と CORBA コンポーネントが 3 つ (C_1，C_2，C_3) である．ここで E_1 と C_1，および C_2 と C_3 を組み合わせることで，同一のワークフローをそれぞれ実現できる．そのようにコンポーネントを組み合わせるアダプタを，それぞれ AD_1，AD_2 として用意する．これらのコンポーネントとは別の Web Service のコンポーネント W_1，W_2 を利用して，同一のワークフローを実現するアダプタ AD_0 を配備し，AD_0 が利用するコンポーネントを実行不能な状態にした．

実験は，100 BASE-TX の同一ネットワークに接続された端末で行った．それぞれのコンポーネントと，生成したアプリケーションプログラムを動作させた端末の構成を，それぞれ次に挙げる．

A) E_1 (EJB):
(CPU) UltraSPARC-IIe 500MHz (OS) SunOS 5.8
(EJB サーバ実装) Java 2 Platform, Enterprise Edition SDK 1.3.1

B) C_1，C_2，C_3 (CORBA):
(CPU) UltraSPARC-IIe 550MHz (OS) SunOS 5.8
(CORBA サーバ実装) JacORB 2.1

C) W_1，W_2 (Web Service):
(CPU) UltraSPARC-IIe 500MHz (OS) SunOS 5.8
(Web Service サーバ実装) Apache Axis 1.1

D) アプリケーションプログラム (Manager: BPEL4WS，Proxy: Web Service):
(CPU) Pentium Mobile Processor 1GHz (OS) Windows XP Professional Edition
(Web Service サーバ実装) Apache Axis 1.1
(BPEL4WS サーバ実装) ActiveBPEL 1.0.1

クライアントから AD_0 を呼び出すと，AD_0 を運用するアプリケーションプログラム運用部の実行監視モジュールは，(1) 実行不能になったことを検知する，そして，(2) 配置モジュールが適切なアダプタを選択し配置する．この処理に要する時間を測定した．まず配置モジュールは，アダプタリポジトリから同じワークフローを実現するアダプタ AD_1，AD_2 を読み込む (A)．次に各アダプタについて，呼び出すコンポーネントのコストを問い合わせる (B)．このために，各コンポーネントにはコストの問い合わせのためのメソッドを用意する．このメソッドはリクエストに対して，予め設定されたコストの値を返す．コストの問い合わせは呼び出す対象とするコンポーネントに対して行われるため，コストを問い合わせることによって各コンポーネントを呼び出せることも同時に確認する．そしてそれぞれのアダプタに対して呼び出す全コンポーネントのコストの合計値を計算し，コストの合計値が低い方のアダプタを配置する (C)．なおここでは，簡単化のため AD_0 が利用するコンポーネントすべてを実行不能な状態にし，処理結果の一部を AD_1 や AD_2 で継続できないようにした．

(A)，(B)，(C) の各処理に要する時間を 30 回測定した．その平均を 表 6.2 に示す．

表 6.2 アダプタの切替えに要する時間 (単位: ms)

(A) 読み込み		(B) コスト問い合わせ		(C) 配置	合計
AD_1	AD_2	AD_1	AD_2		
1.1 (σ=0.3)	1.1 (σ=0.3)	261.1 (σ=96.8)	112.5 (σ=1.1)	1619.5 (σ=17.1)	1995.3 (σ=95.0)

6.7.3　測定結果の分析

(1) アダプタによるコンポーネント呼出しのオーバヘッド

表 6.1 によると，生成したアダプタを経由して呼び出すことによって，呼出しに要する時間が約 30ms 増加することがわかった．

アダプタを経由して呼び出す際，コンポーネントを実行できるかどうかを確認するための時間と，Manager から Proxy のコンポーネントの名前解決を行うための時間と，実際に Proxy を呼び出すための時間が発生する．これらは呼び出そうとするコンポーネントの処理内容には依存しない．しかしそれぞれのコンポーネントの利用に必要な処理なので，アダプタから呼び出すコンポーネントの数に比例して単純に増加する．

すでに呼び出したコンポーネントと同一のコンポーネントに存在するメソッドを呼び出す場合には名前解決の処理は不要なので，その処理のための時間は発生しない．したがって，アダプタから多くのメソッド呼出しを行っても，少数のコンポーネントに対して行う場合には，ある程度はオーバヘッドが小さくなる．逆に同数のメソッド呼出しであっても，呼び出されるメソッドがそれぞれまったく別のコンポーネントに存在する場合は，オーバヘッドが大きくなる．本質的にはオーバヘッドの多くは，Web Service として実装された Proxy を呼び出す際に，XML で記述された SOAP

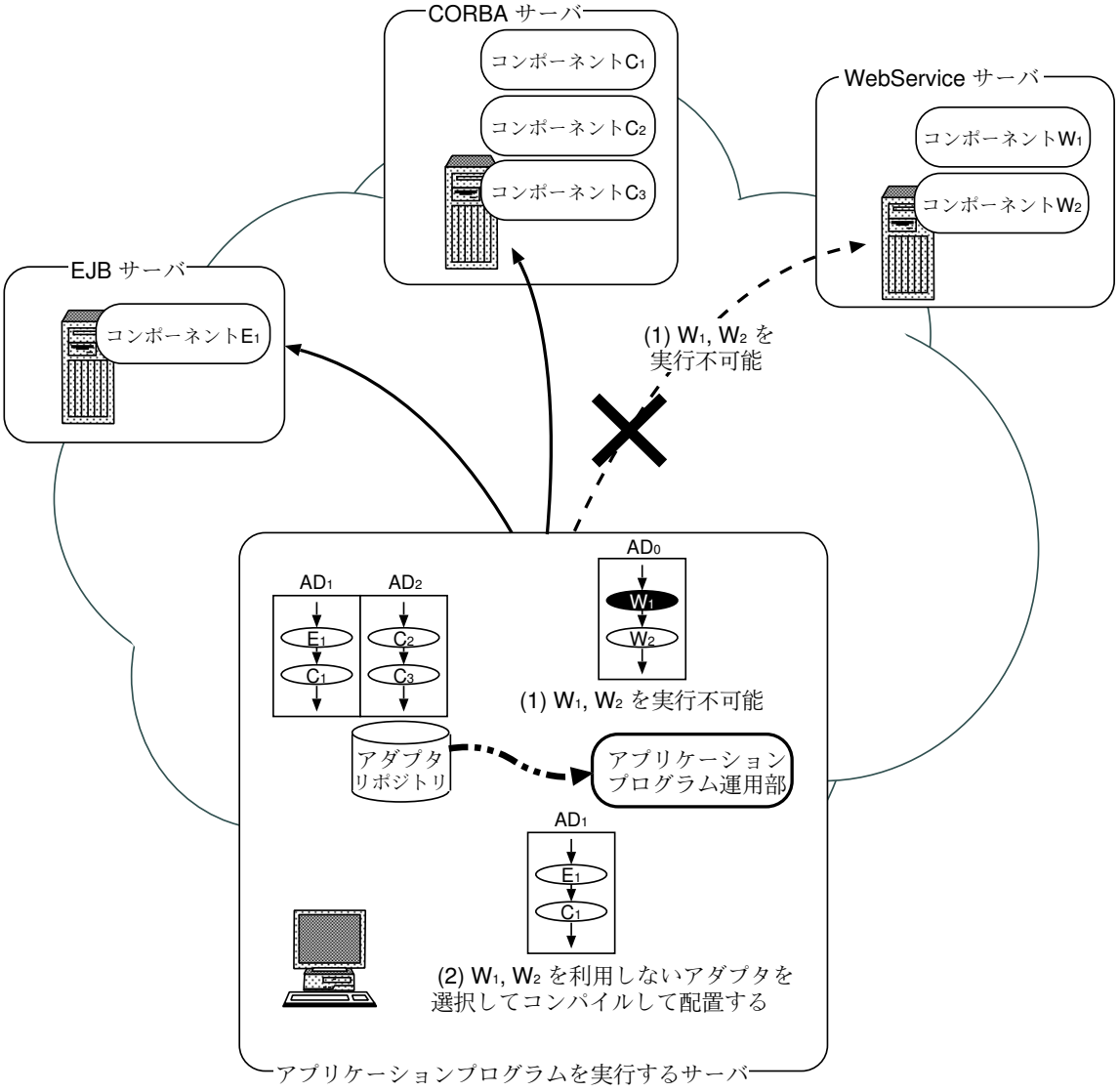

図 6.9 アダプタの切替えの時間を測定する際のシナリオ

メッセージのエンコーディングやデコーディングの処理を行う時間である．今回の実装では，6.6.3 節に述べたように，Proxy を Web Service として生成することによって，SOAP による HTTP アクセスを可能にしている．しかし測定実験では，Manager と Proxy を同一のコンピュータ上で動作させているため，このようなことを考慮する必要がない．このような場合に Proxy と Manager 間の通信に CORBA のような呼出しに要する時間の短い分散コンポーネント技術を利用するようにアダプタを生成することによって，さらにオーバヘッドを小さくできる．

なお表 6.1 において，各コンポーネントの呼出しに要する時間と標準偏差を比較すると，EJB のコンポーネント呼出しでは標準偏差が比較的大きくなっている．その要因は，サーバの JavaVM で散発して発生するガーベージコレクション処理などの動作や，キャッシュミスなどであり，EJB コンポーネントサーバ実装は，これらによる影響を受け易い．その結果，EJB コンポーネントへの呼出しにばらつきが生じており，タイミングが悪いとコンポーネント呼出しに非常に長い時間を要する．ただし 30 回の測定において呼出し時間が長くなっている回数は非常に少なく，EJB についても，ほとんどの場合は比較的安定してコンポーネントを呼び出すことができていた．そして直接呼び出した場合でも，アダプタを介して呼び出した場合でも，呼出し時間が長くなる状況が発生する割合は一定であった．このためメソッド呼出しの時間の比較に関して，結果的にほとんど影響を与えていないので，標準偏差の値は大きいが他のコンポーネント技術と同程度のオーバヘッドが得られている．

(2) アダプタの切替えに要する時間
表 6.2 によると，アプリケーションプログラム運用部が利用不能となったことを検知してから，2 秒程度でアダプタの切替えが終了している．今回の実装では，コストの問い合わせをすべてのコンポーネントに対して逐次行っている．ここでは，AD_1 が呼び出す E_1，C_1 と，AD_2 が呼び出す C_2，C_3 にコストの問い合わせを行う．表 6.1 で示したように，EJB のコンポーネントの方が，CORBA のコンポーネントよりも呼出しに要する時間が長いため，コストの問い合わせに関しても AD_1 に要する時間の方が長くなっている．このようにコストの問い合わせをすべてのコンポーネントに対して行っているため，候補となっているアダプタの数や，アダプタから呼び出すコンポーネントが増加すると，それに伴いアダプタの切替えに要する時間も増加する．したがって，このようなコンポーネントの選択のために必要な処理を，すべてのコンポーネントに対して並行に行うことで，ある程度の一定時間でアダプタの切替えを終わらせることが可能である．

6.8　まとめ

本章では，異種分散コンポーネントを利用するアプリケーションの開発を支援するシステムを提案し，その設計と実装を述べた．

本章で提案したシステムは，開発者がワークフロー情報を記述することで，異種分散コンポーネントを一つのプログラムとして連係動作させるためのプログラムコード (アダプタ) を生成する．その際に開発者によるワークフロー情報の指定のためには，UML ア

クティビティ図による統一された記法を利用できるようにした．そして生成したアダプタがそれぞれのコンポーネントを呼び出すことによって，全体として開発対象のアプリケーションプログラムとして動作する．アダプタを分散コンポーネントとして動作するように生成することによって，利用者はアダプタを他の分散コンポーネントと同様に利用できる．

また，本章で提案したシステムでは，呼び出そうとするコンポーネントの処理に障害が発生した時に，同じ機能を持つ他のコンポーネントに自動的に処理を切り替えることにより処理を続行させるための，Variation と呼ぶ方法を提供した．これは，呼び出そうとするコンポーネントが実行不可能な場合に，同じワークフローを実現する他のコンポーネントを呼び出すアダプタを代替として実行する．これにより，全体として実行不可能になる確率を低くすることで，信頼性の低下を回避している．

提案機構によって開発したアプリケーションでは，コンポーネント間の異種性を吸収するために，アダプタを介してコンポーネントの呼出しを行う．評価実験を行い，このためにコンポーネントの処理時間が約 30 ms 増加することがわかった．

第7章　異種分散コンポーネントの Plug and Play プラットフォーム

本章では, 本研究で提案する異種分散コンポーネントの Plug and Play プラットフォームの詳細を述べる，その後で評価実験を示す.

7.1　概要

分散コンポーネントの存在するサーバやそれを利用するクライアントをネットワークに接続することで，それらを Plug and Play で動作させることのできるアプリケーション実行プラットフォームを提案する．コンポーネントを利用するクライアントや，コンポーネントを連係させるサーバは，このプラットフォームを利用し，必要とするコンポーネントとそれを提供するサーバを利用時に自動的に発見する．そしてコンポーネントの種類によらずに連係させる．したがって，それらのクライアントやサーバのみならず，コンポーネントを提供するサーバも含め，分散コンポーネントシステムを構成するすべての端末を Plug and Play で動作させる．

なお，5 章で述べた全体設計では，コンポーネントを連係させるアプリケーションの運用時に必要なコンポーネントを探すことをシナリオ例として挙げた．ここで提案するプラットフォームはそれだけに留まらず，コンポーネントを利用するクライアントも，コンポーネントを探索して利用できるようにする (7.2 節に詳細を述べる).

Plug and Play プラットフォームは，コンポーネントの存在を管理する特別な機構を設けずに，コンポーネントを利用時に探索することを可能にする．これにより，3.3.1 節で述べた分散コンポーネントを発見する際の問題を解決する．そして，どのようなソフトウェアを連係させるのかを事前に把握できなくても，探索して発見したコンポーネントを，それらの異種性によらずに連係して動作させることを可能にする．これにより，3.3.2 節で述べた分散コンポーネントを利用する際の問題を解決する.

本章では，まず利用シナリオを示すことで，Plug and Play プラットフォームの概観について述べる．その後で，コンポーネントを探索する方法と，連係させる方法を示す．そしてその後で実装について述べ，評価実験について述べる.

7.2　Plug and Play プラットフォームの利用シナリオ

図 7.1 に，Plug and Play プラットフォームの利用シナリオを 2 つ示す．なおこの図において，クライアント α がネットワークにプラグインした時に，利用したいコンポーネントを，サーバ A が提供しているとする．クライアント α からサーバ A を利用する手

順は，次のようになる．

(1) クライアント α は，必要とするコンポーネントを探索し，サーバ A を発見する．

(2) クライアント α は，発見したサーバ A を呼び出す．

7.2.2 節で，この発見と呼出しの詳細について述べる．

また，図 7.1 において，クライアント β がネットワークにプラグインした時に，利用したいコンポーネントを，サーバ C が提供しているものとする．ただし先ほどの例とは異なり，サーバ C にはコンポーネントを連係させるアプリケーションが動作していて，サーバ C が連係させたいコンポーネントをサーバ A とサーバ B が提供しているとする．コンポーネントを連係させるサーバ C は，クライアントから見ればコンポーネントのサーバとして振舞う． クライアント β からサーバ C を利用する手順は，次のようになる．

(3) クライアント β は，クライアント α の場合と同様に，コンポーネントを呼び出す前に必要とするコンポーネントを探索し，サーバ C を発見する．

(4) クライアント β は，発見したサーバ C を呼び出す．

ここでサーバ C は，連係させるべき各コンポーネントを探索して呼び出す．この手順は，次のようになる．

(5) サーバ C は，まずクライアントの場合と同様に，コンポーネントを探索する．その結果サーバ A を発見する．

(6) サーバ C は，発見したサーバ A を呼び出す．

(7) サーバ C は，(5) と同様にサーバ B を発見する．

(8) サーバ C は，発見したサーバ B を呼び出す．

(9) サーバ C は，サーバ A とサーバ B を呼び出し，必要な処理を行った結果を，クライアント β に返す．

7.2.3 節で，(3) 〜 (9) までの手順の詳細を述べる．

7.2.1　コンポーネントの探索

Plug and Play プラットフォームは，コンポーネントを呼び出す時点でコンポーネントを提供するサーバを発見する．そのために次のメッセージを用意する．

(A) コンポーネント探索メッセージ (コンポーネント要求／応答メッセージ)
コンポーネントを探索し，発見するためのメッセージ．

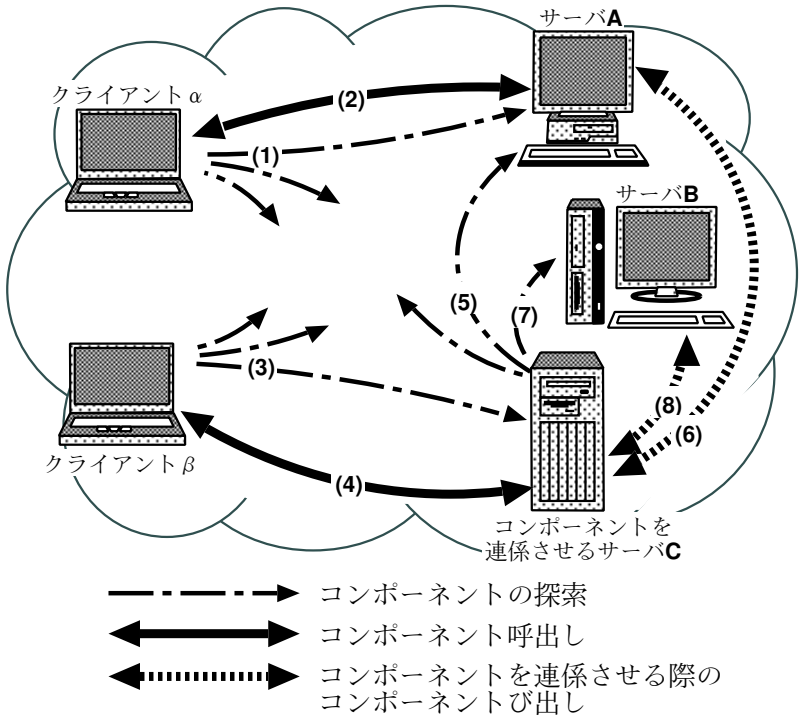

図 7.1 利用シナリオ

(B) パラメータ探索メッセージ (パラメータ要求／応答メッセージ)
コンポーネントを呼び出すためのインタフェースに関する情報を入手するためのメッセージ．なお対象とする分散コンポーネントアーキテクチャによっては，インタフェースに関する情報の取得を独自の方法で行う場合がある．その場合は，インタフェースに関する情報の取得のために必要な情報を応答メッセージに含む．

(C) コンポーネント広告メッセージ
コンポーネントの存在を示すためのメッセージ．

コンポーネントを呼び出すためには，実装されているコンポーネントのアーキテクチャに依存した情報が必要になる．例えば Web Service の場合は，サービスの名前や，WSDL [Christensen2001] で記述されたインタフェースの情報を入手するための URL や，コンポーネントサーバを利用する際の，プログラムのエンドポイントのパスなどである．Plug and Play プラットフォームでは，実装されているコンポーネント技術を特定せずに，色々な種類のサーバで動作するコンポーネントをクライアントから利用できるようにする．そのためクライアントは，同一の処理を提供する色々な種類のサーバを発見し得る．しかし，クライアントはすべての種類のコンポーネントに関する情報を必要としないかも知れない．そこで実装されているコンポーネントのアーキテクチャに依存した情報を，コンポーネント探索とは別個のパラメータ探索メッセージとして設計する．

まず 7.2.2 節で，クライアントがコンポーネントを探索して利用する際の手順を述べる．そして 7.2.3 節で，コンポーネントを連係させるサーバから必要なコンポーネントを探索

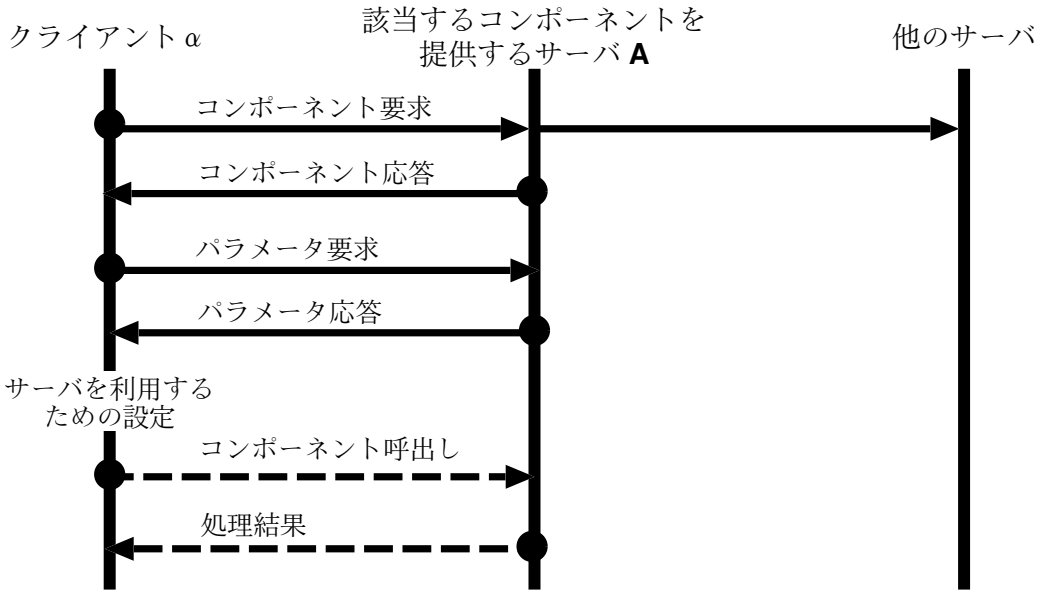

図 7.2 コンポーネントの探索と利用

して利用する際の手順を述べる．

7.2.2　単一のコンポーネントを利用する場合

7.2 節に示したシナリオにおいて，クライアント α がコンポーネントを探索し，利用する場合のメッセージシーケンスを図 7.2 に示す．

まずクライアント α は，利用したいコンポーネントが同一ネットワーク上に存在するかどうかを調べるために，コンポーネント要求メッセージをマルチキャストで送信する．このメッセージには，呼び出したいコンポーネントを指定するための文字列を含む．マルチキャストで送信するため，コンポーネントを提供するすべてのサーバから必要なコンポーネントを探索できる．該当するコンポーネントを提供するサーバ A は，クライアント α にコンポーネント応答メッセージを返す．これにより，クライアント α は必要なコンポーネントを提供するサーバ A を発見する．呼び出したいコンポーネントを提供するサーバを複数発見した場合は，一つを選ぶ．

次にクライアント α は，サーバ A にパラメータ要求メッセージを送信する．サーバ A は パラメータ要求メッセージを受信し，コンポーネントを呼び出すために必要なインタフェース情報を，パラメータ応答メッセージとしてクライアント α に返す．そしてクライアント α はその情報を利用して，コンポーネントを呼び出す．

なお，コンポーネントを提供するサーバがコンポーネント広告メッセージを定期的にマルチキャストで送信することによって，クライアントにコンポーネントの存在を通知することもできる．

7.2.3　コンポーネントを連係させて利用する場合

コンポーネントを連係させるアプリケーションを利用する際には，コンポーネントを連係させるサーバが必要なコンポーネントを探索してから呼び出す．これによって，その時点で利用できるコンポーネントを連係させる．7.2 節に示したシナリオにおいて，クライアント β がコンポーネントを連係させるサーバ C を探索して利用する場合のメッセージシーケンスを図 7.3 に示す．この際，サーバ C は必要なコンポーネントを提供するサーバ A とサーバ B を探索して利用する．

まずクライアント β は，7.2.2 節の場合と同様に，呼び出したいコンポーネントを探索する．そのために，コンポーネント要求メッセージをマルチキャストで送信する．コンポーネントを連係させるサーバ C は，要求の内容が提供するアプリケーションに該当する場合は，クライアント β にコンポーネント応答メッセージを返す．クライアント β は応答メッセージに従い，サーバ C にパラメータの要求を行う．そしてコンポーネントを呼び出すために必要な情報を，パラメータ応答メッセージとして受信する．

次にクライアント β は，受信したパラメータ応答メッセージに含まれる情報を利用して，コンポーネントを連係させるアプリケーションを呼び出す．コンポーネントの連係は，外部の複数のコンポーネントへの呼出しを行った結果を組み合わせることによって行う．サーバ C は，それぞれのコンポーネントへの呼出しの際に，7.2.2 節と同じ手順でコンポーネントを探索して利用する．そして，必要なすべてのコンポーネントを呼び出し，処理を行った結果をクライアント β に返す．

なお 7.2.2 節の場合と同様に，コンポーネントを連係させるサーバは，コンポーネント広告メッセージを定期的にマルチキャストで送信することもできる．さらに，各コンポーネントを提供するサーバからコンポーネント広告メッセージを定期的にマルチキャストで送信することによって，コンポーネントを連係させるサーバが，必要なコンポーネントの存在を予め知ることもできる．

7.3　分散コンポーネント間の異種性の吸収

Plug and Play プラットフォームでは，探索したコンポーネントをその種類によらずに，呼び出せるようにする．

4 章で述べたように，従来の研究 [Cerqueira1999] や 6 章で提案したシステムでは，異種性を吸収するためのプログラムコードを，呼び出そうとするコンポーネントごとに事前に用意しなければならないことが問題であった．3.3.2 節で述べたように，Plug and Play 環境では，どのようなサーバ上で動作するコンポーネントを利用するかを事前に特定できない．そして開発時に組み合わせられるコンポーネントが特定しないため，異種性を実行時に動的に吸収する必要がある．

そこで呼出しの形式の相違を，コンポーネントを連係させるアプリケーションやクライアントのプログラムを実行するプラットフォームで吸収する．そのために，種類の異なるコンポーネントへの呼出しの際に，呼出しの方法やメッセージや返戻値の形式を変換できるようにする．詳細については 7.4.3 節で述べる．

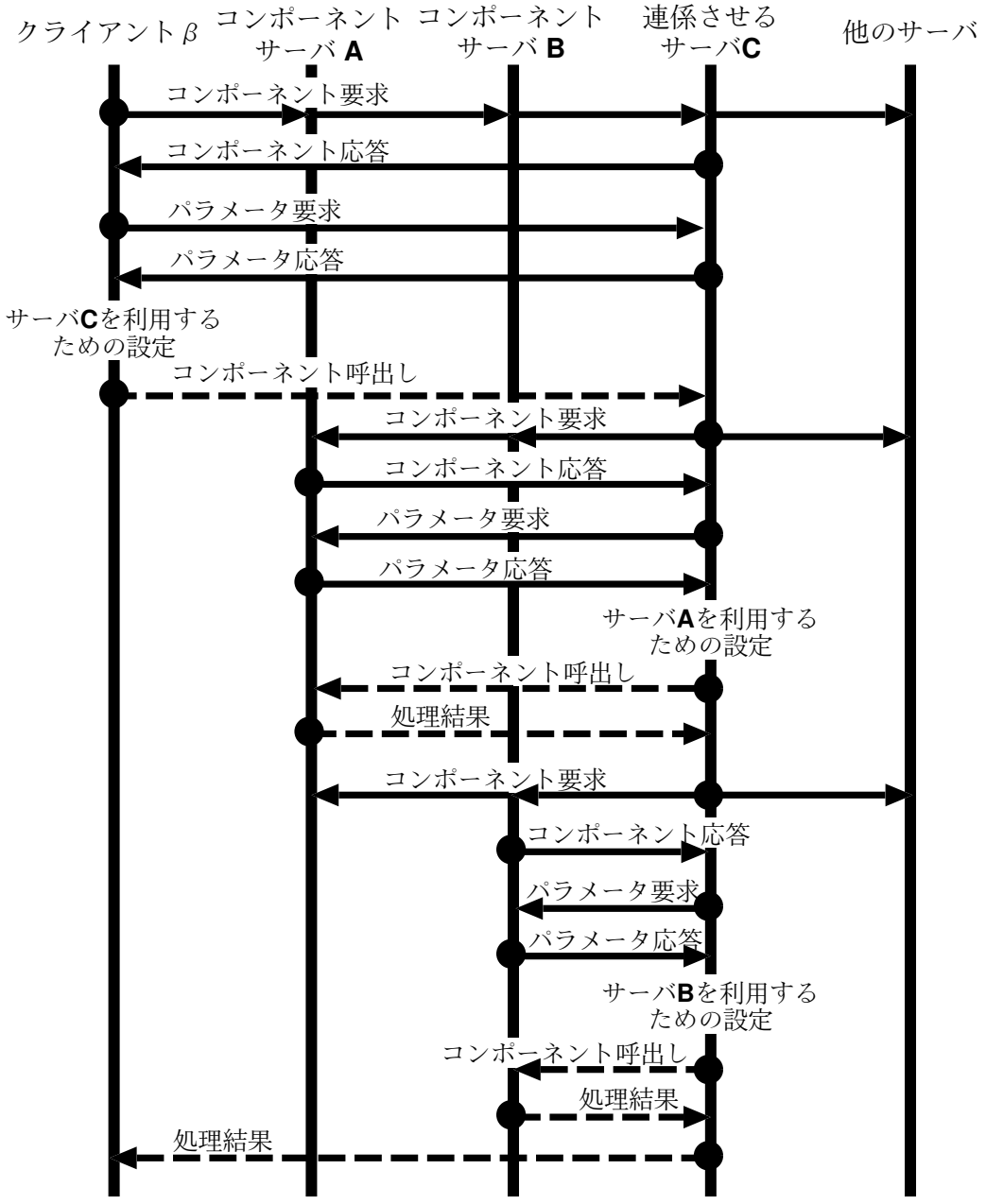

図 7.3 コンポーネントの連係

7.4　実装

本節では，本章で提案する異種分散コンポーネントの Plug and Play プラットフォームについて，まずそれぞれの構成要素を挙げ，それぞれの構成要素の動作について，詳細に述べることによって，実装の詳細を述べていく．

7.4.1　構成

実装したシステムは，6.6 節で実装したシステムと同様に，EJB と，Web Service と，CORBA の三種類を対象として連係させる．なお他の種類のコンポーネント技術で実装されたコンポーネントに関しても，モジュールを追加することで組み合わせることができるように実装した．

提案するプラットフォームを，次の 5 つの部分から構成するように実装した．

A) コンポーネント要求部
　コンポーネント要求メッセージや，パラメータ要求メッセージを送信することによって，コンポーネントを提供するサーバを探索し，インタフェース情報を取得する部分

B) コンポーネント応答部
　コンポーネント要求部からの要求メッセージに対し，コンポーネント応答メッセージやパラメータ応答メッセージを送信することによって応答し，インタフェース情報を提供する部分

C) コンポーネント広告部
　コンポーネント広告メッセージによって，コンポーネントの存在を通知する部分

D) メディエータ部
　探索した結果得られたコンポーネントに対して，コンポーネント技術や実装による異種性を吸収して，呼出しを行う部分

E) アプリケーションプログラム運用部
　コンポーネントを連係させるアプリケーションを運用する部分

なお，コンポーネントの連係の際のワークフローの表現には，BPEL4WS 仕様を利用して行っていることを前提とした．したがって 6.6 節の開発支援ツールで作成したアプリケーションプログラムのうち，BPEL4WS で保存した場合のみ，この Plug and Play プラットフォームで動作させることができる．また，7.2 節で例示したように，このプラットフォームはコンポーネントを連係させるアプリケーションのみならず，コンポーネントを利用するクライアントプログラムにも，動的にコンポーネントを探索して利用できる環境を提供する．

実装したシステムを利用する際の構成を，図 7.4 に示す．この図では，複数のコンポーネントを連係させるサーバをクライアントが利用している．そしてコンポーネントを利用するクライアントと，コンポーネントを提供するサーバと，コンポーネントを連係させるサーバから構成されている．

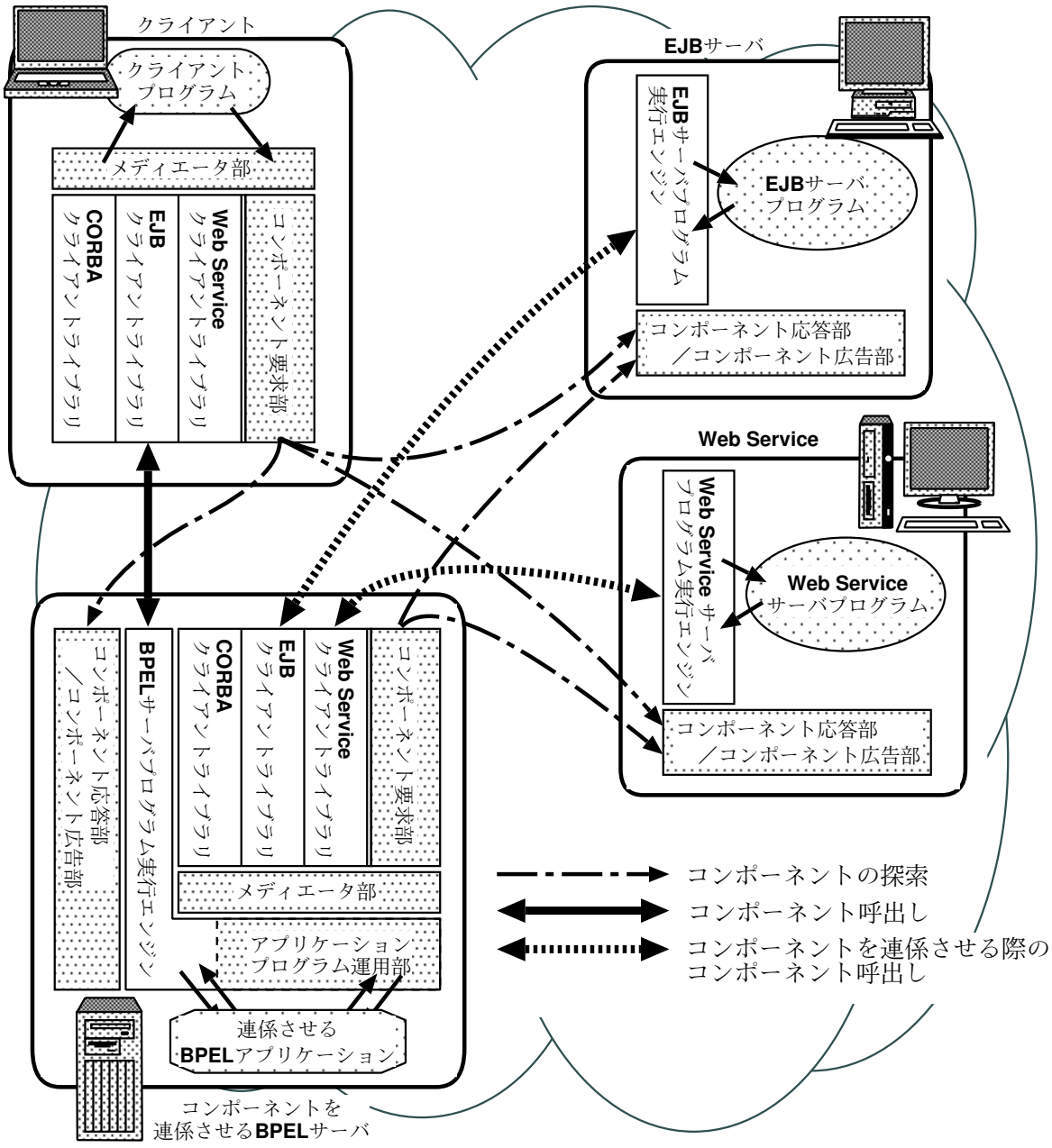

図 7.4 コンポーネントを連係させるシステムの構成

クライアントにはコンポーネントを探索して利用できるように，コンポーネント要求部とメディエータ部を用意する．そしてコンポーネントを提供するサーバやコンポーネントを連係させるサーバには，クライアントから探索できるように，コンポーネント応答部を用意する．またコンポーネントを連係させるサーバは，BPEL4WS に対応した既存のサーバを拡張し，アプリケーションプログラム運用部やメディエータ部を組み込むことによって用意する．

7.4.2　コンポーネント探索プロトコル

提案した実行環境は，それぞれのコンポーネントが利用する分散コンポーネント技術や実装によらずに，連係させることを可能にする．そのためコンポーネントを探索する仕組みも，コンポーネントが実装されているアーキテクチャに依存させずに実現する必要がある．そこで本研究では，7.2.1 節で述べたそれぞれのメッセージをやりとりする独自の探索プロトコルを設計し，それによって探索を行う機構を実装した．なおこのプロトコルの実装の際には，コンポーネント要求のメッセージをやりとりに IP マルチキャストを用いて実装した．

4 章で挙げた WS-Discovery は，利用するプロトコル自体に SOAP を用いており，Web Service が利用するプロトコルに依存している．また同様に 4 章で挙げた汎用的なサービス探索プロトコル [Goland1999] [Guttman1999] は，情報を探索するためのメッセージのやりとりの一般的な方法を規定しており，これを拡張することによって実現することも可能である．しかし拡張する部分が多過ぎ，結果として冗長性の高いものになってしまう．

提案機構でコンポーネントを探索する場合の例を，図 7.5 に示す．

(1) コンポーネントを利用したいクライアントはまず，コンポーネント要求部から，コンポーネント要求メッセージを IP マルチキャストを利用して送信する．コンポーネント要求メッセージには，利用したいコンポーネントのタイプを示す文字列を含む．例えば，3.3 節の例におけるビデオサーバを探索したい場合には，“CompType:VideoServer” のように，提供するサービスを指定する文字列を含む．また必要ならば，動作するプラットフォームや，サーバ実装の種類を指定することもできる．

(2) コンポーネントを提供するサーバや，コンポーネントを連係させるサーバは，コンポーネント応答部によってコンポーネント要求メッセージを受け取る．そしてコンポーネント要求メッセージに含まれるコンポーネントのタイプに一致する場合は，コンポーネント応答部によってコンポーネント応答メッセージを返す．このメッセージには，動作するプラットフォームの種類や，サーバ実装の種類を含む．

    なお，本実装ではクライアントは文字列でコンポーネントのタイプを指定し，サーバは単純にその文字列に一致した場合に応答を返している．これは，予めコンポーネントを規格化しておき，その規格を表す文字列を決めておくことにより探索するような利用方法を想定しているためである．したがってコンポーネントのタイプの文字列の記述に探索の性能が依存する．より柔軟に探索を行うためには，[Constantinescu2004] や [Brogi2006B] や，[Nedos2006] で提案されている，セマンティクスを利用した探索を行うように拡張することが考えられる．

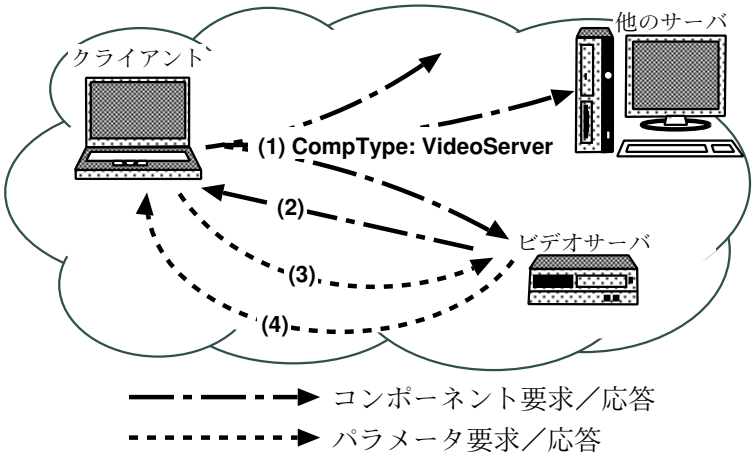

図 7.5 コンポーネントの探索の例

(3) コンポーネント応答メッセージを受け取ったクライアントは，コンポーネント要求部によって，パラメータ要求メッセージを送信する．

(4) 該当するサーバのコンポーネント応答部は，パラメータ要求メッセージを受け取り，実装されているコンポーネントのアーキテクチャに依存した情報を応答として返す．したがってパラメータ応答メッセージの詳細はコンポーネントのアーキテクチャにより異なるが，それぞれのサーバ内でコンポーネントを識別するための名前と，メソッドのシグネチャを含む．

また，コンポーネントを提供するサーバや，コンポーネントを連係させるサーバは，コンポーネント応答部によってコンポーネント広告メッセージを定期的に IP マルチキャストを利用して送信することができる．これによってクライアントにコンポーネントの存在を通知する．コンポーネント広告メッセージは，コンポーネント応答メッセージと同じ内容を含む．

コンポーネント探索プロトコルのメッセージの詳細を，付録 C に示す．

7.4.3　メディエータ部によるコンポーネント呼出し

実装したシステムは，EJB，Web Service，CORBA の 3 種類のコンポーネントを探索し，それらを呼び出すことによって連係させて動作させる．コンポーネントの利用者は，まずコンポーネントを呼び出すためのクライアントプログラムを用意する．コンポーネントの呼出しの際は，メディエータ部がコンポーネントへの呼出しを受け取り，コンポーネント要求部を利用して必要なコンポーネントを探索する．そして発見したコンポーネントの呼出しの手順に従い，リクエストを変換し，コンポーネントを呼び出し，処理結果をクライアントに返す．これにより，コンポーネント技術や実装による異種性を吸収する．

メディエータ部には，Web Service での呼出しの形式に類似した，Java 言語による共通の API を用意した．API の一覧を，付録 D に示す．Web Service の呼出しとの形式的な

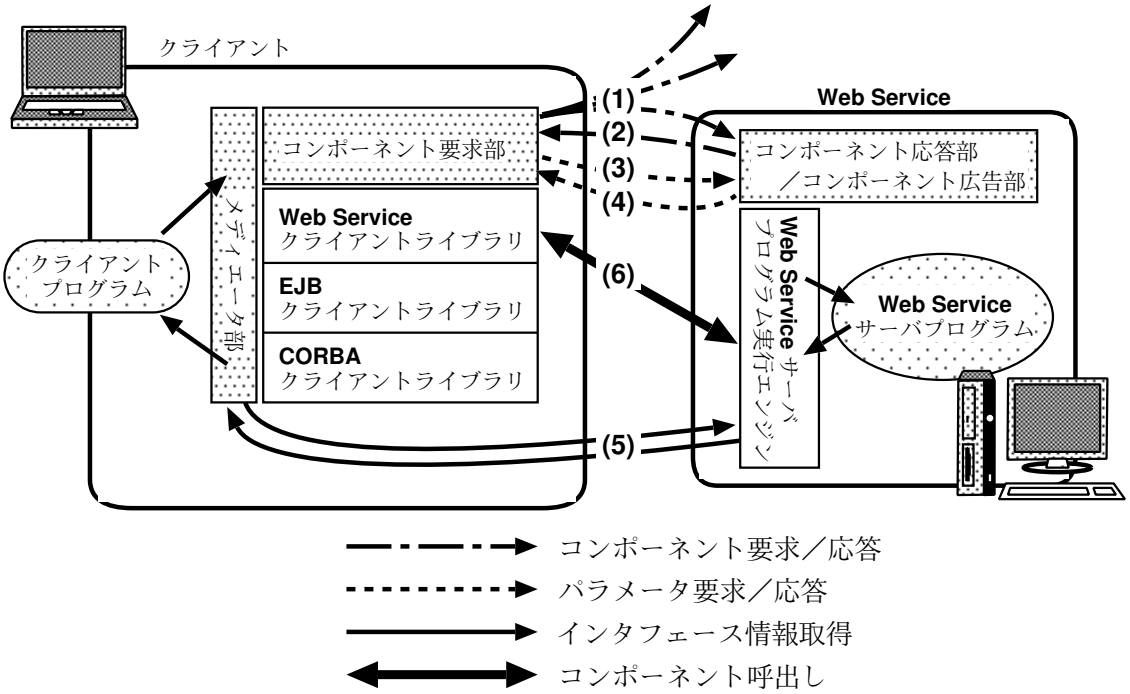

図 7.6 メディエータ部からのコンポーネント呼出し

相違は，コンポーネントの場所を示すエンドポイントの URL を指定するための API の代わりにコンポーネントのタイプを示す文字列を指定するための API を用意した点である．クライアントプログラムにはこの共通 API を利用し，通常のコンポーネント呼出しと同様に，メディエータ部を呼び出すためのコードを記述する． このようにすることで，クライアントプログラムからは意識せずに，メディエータ部によってコンポーネントを探索して呼び出すことを可能にしている．

メディエータ部によるコンポーネント呼出しの手順を，図 7.6 に示す．なおこの図では，Web Service に対しての呼出しの手順を示す．

(1) メディエータ部は，コンポーネント要求部を利用し，必要なコンポーネントを探索する．この際，すでにコンポーネントの存在を知っている場合には，探索を行わない．

(2) 該当するコンポーネントを提供するサーバは，応答を返す．なお同一の内容を提供する複数のコンポーネントが探索された場合にどのような基準で選択するかを，メディエータ部に予め設定しておくこともできる．この基準としては，コンポーネントの提供者によって設定された生存時間や，コンポーネントプラットフォームの種類，コンポーネントの処理に要する時間により選択できる．またメディエータ部にモジュールを追加することで，基準を追加することもできる．

(3) コンポーネント要求部は，探索したコンポーネントを提供するサーバに対してパラメータ要求メッセージにより，パラメータ情報を要求する．

(4) パラメータ要求メッセージを受信したサーバは，パラメータ応答メッセージによって，コンポーネントが実装されているアーキテクチャに依存した情報を応答する．

メディエータ部はパラメータ要求メッセージで取得した情報にしたがって，コンポーネントを呼び出す．この手順の詳細はそれぞれのアーキテクチャによって異なるが，次の処理に大別できる．

(5) 実装されているコンポーネント技術にしたがった方法で，コンポーネントのインタフェースに関する情報を入手する．

(6) 入手した情報にしたがって，コンポーネントの呼出しを行う．

図 7.6 では， Web Service の場合の処理を示している．CORBA の場合も，同様の手順で呼出しを行う．しかし，EJB の場合には，Web Service における WSDL や CORBA における IDL のような，標準的なインタフェースの記述のための言語が用意されていないため，メソッドのインタフェースの情報を，手順 (4) のパラメータ応答メッセージに含めて応答するため，手順 (5) に相当する処理は行わない．
それぞれのアーキテクチャごとの処理を，次に挙げる．

A) Web Service
パラメータ応答メッセージによって，WSDL で記述されたインタフェース情報を入手するための URL を受けとる．そしてその URL からインタフェース情報を入手する．メディエータ部は，クライアントプログラムから受けとったリクエストを該当するメソッドのシグネチャに変換し，探索した Web Service を呼び出す．

B) CORBA
パラメータ応答メッセージによって，CORBA-IDL [OMG2002] で記述されたインタフェース情報を入手するための URL を受けとる．そしてその URL からインタフェース情報を入手する．メディエータ部は入手したインタフェース情報に従い，CORBA のコンポーネントを呼び出すための，スタブプログラムを生成する．メディエータ部は，クライアントプログラムから受けとったリクエストを，スタブの該当するメソッドのシグネチャに変換し，スタブを介して探索した CORBA コンポーネントを呼び出す．

C) EJB
パラメータ応答メッセージによって，該当するメソッドを利用するためのインタフェースに関する情報を受けとる．この情報は，他の 2 つのアーキテクチャにおいて WSDL や CORBA-IDL によって記述されるコンポーネントのメソッドのシグネチャに関する情報に相当する．メディエータ部は，クライアントプログラムから受けとったリクエストを該当するメソッドのシグネチャに変換し，探索した EJB コンポーネントを呼び出す．

これらの呼出しが行われコンポーネントのメソッドの処理結果を受けとると，返戻値としてクライアントプログラムの受けとれる形式に変換して返す．

クライアントからのリクエストの変換，およびコンポーネントからの返戻値の変換のために，マッピングテーブルを利用する．例えば，CORBA-IDL では，文字列は string や wstring という型で表現するが，これを Java の String クラスに変換する必要がある．提案システムの実装では，それらの対応を予めマッピングテーブルとして用意している．

上記において，パラメータ応答メッセージを受けとりその解析を行った後，コンポーネントを呼び出す部分までの手順は，それぞれのコンポーネントアーキテクチャによって異なる．そしてそれぞれのコンポーネントを呼び出すためのライブラリが相互に影響を与えないように，各コンポーネントへの呼出しを個別のスレッドで行っている．新たなコンポーネントアーキテクチャを利用するコンポーネントを連係させる場合は，コンポーネントアーキテクチャに依存した，パラメータ応答メッセージの解析と，コンポーネントの呼出しを行うモジュールをメディエータ部に追加するだけで対応できる．

なお現在の実装では，単純な同期呼出しの変換のみを対象としている．しかし例えば CORBA の保留同期や，Web Service での片方向呼出しのような，一般の同期呼出しと異なる呼出しを行いたい場合にも，それらのセマンティックスを持つモジュールをメディエータ部に追加することで，対応できる．

7.4.4　コンポーネントを連係させるサーバの実装

本実装では，コンポーネントを探索して呼び出すことができるように，BPEL4WS の言語を拡張した．BPEL4WS の定義 [Jordan2006] では，BPEL4WS の記述自体では，個々のコンポーネントの具体的な指定を行わない．実際のコンポーネントサーバの指定は，WSDL によるコンポーネントのインタフェース情報で記述する．そして，BPEL4WS からは，呼び出したいコンポーネントに該当する WSDL のインターフェース情報を指し示す．このように，BPEL4WS によるコンポーネントを連係させるアプリケーションの記述の一部として，WSDL が利用されている．そこで，WSDL の記述を拡張し，コンポーネントのタイプを示す文字列を指定するためのタグを用意した，この文字列の記述については，7.4.3 節で述べた Java API の場合と同様に行う．そしてこのタグを，WSDL において，コンポーネントのエンドポイントの URL を指定するタグ要素の代わりとして利用できるようにした．詳細を，付録 B.2 に示す．コンポーネントを連係させるアプリケーションの提供者は，アプリケーションを BPEL4WS によって記述する際に，コンポーネントのタイプを指定するタグ要素をインタフェース情報として WSDL に記述する．

アプリケーションプログラム運用部は，既存の ActiveBPEL [ActiveBPEL2006] のサーバ実装を拡張することで実装した．BPEL4WS の解釈・実行時には，ActiveBPEL のサーバ実装は，BPEL4WS の記述が参照している WSDL の記述を読み込む．WSDL を解釈する際にこの拡張部分が存在した場合には，メディエータ部を呼び出すように実装した．そして，メディエータ部からは，7.4.3 節で述べた Java API の場合と同様に呼出しを行う．その結果，タグ拡張部分に記述されたコンポーネントのタイプに相当するコンポーネントを，コンポーネント要求部を利用して探索し，動的に発見して利用できるようにしている．また，BPEL4WS の実装では Web Service しか連係できないが，メディエータ部を呼び出すことによって，その他の種類のコンポーネントも呼び出せるようにした．複数の異種コンポーネントを連係させる場合は，異種性をメディエータ部が吸収する．

アプリケーションプログラムの処理は BPEL4WS に記述される．コンポーネント探索の処理や，異種性の吸収のための処理は，アプリケーションプログラム運用部が自動的に行うため，外部からは通常の BPEL アプリケーションと同様に，外部から Web Service のインタフェースで呼び出すことができる．したがって，作成したアプリケーションを組み合わせて，BPEL4WS に対応した他のサーバによって，更に新しいアプリケーションを作ることも可能である．同様に，実装したシステムを利用することにより，作成したアプリケーションを新しい別のアプリケーションが自動的に発見し，組み込むこともできる．

7.5　評価実験

実装したシステムの性能を評価するため実験を行った．本節では，それぞれの測定手順とその結果を示し，その後で分析を行う．

実験は，100 BASE-TX の同一ネットワークに接続された端末で行った．そして，EJB，CORBA，Web Service のコンポーネントを動作させるサーバと，コンポーネントを連係させるアプリケーションプログラムを動作させるための BPEL4WS サーバと，それを利用するクライアントを用意した．各端末には，CPU に Pentium4 3.0GHz を搭載した同一の構成の PC を用いた．それぞれの端末で利用した OS とサーバ実装 (クライアントについては，呼び出すために利用したライブラリ) は，次の通りである．

A) EJB:
　(OS) FreeBSD 5.4 Release
　(EJB サーバ実装) JBoss 4.0.3 SP1

B) CORBA:
　(OS) FreeBSD 5.4 Release
　(CORBA サーバ実装) Java 2 Standard Edition SDK 1.4.2 付属 ORBD

C) Web Service:
　(OS) FreeBSD 5.4 Release
　(Web Service サーバ実装) Apache Axis 1.3

D) アプリケーションプログラム (BPEL4WS):
　(OS) FreeBSD 5.3 Release
　(Web Service サーバ実装) Apache Axis 1.3
　(BPEL サーバ実装) ActiveBPEL 1.2

E) クライアント (Web Service):
　(OS) FreeBSD 5.4 Release
　(Web Service クライアントライブラリ) Apache Axis 1.3

なお，このうち ActiveBPEL 1.2 については，7.4.4 節で述べたように，本実装により拡張を行っている．

7.5.1　コンポーネント呼出しに要するオーバヘッドの測定

7.2 節で述べたように，本研究で実装したシステムでは，コンポーネントを利用するクライアントや，コンポーネントを連係させるアプリケーションは，分散コンポーネントを探索してから呼び出す．またその際に，異種性を吸収するためにリクエストや返戻値を変換する．これらの処理によって，コンポーネントの呼出しの処理を行うために必要な時間が増加する．この影響を調べるため，呼出しの際に発生するオーバヘッドを測定した．

(1) 単一コンポーネントの呼出し

Web Service，EJB，CORBA のそれぞれについて，同一の機能を提供するサーバプログラムと，それを利用するように記述したクライアントプログラムを作成した．また実装したシステムで動作するクライアントプログラムを作成し，各サーバにコンポーネント応答部を配置した．3 種類のコンポーネントそれぞれを利用する 3 つのクライアントプログラムと，実装したシステムで動作するクライアントプログラムの，計 4 種類のクライアントプログラムで，それぞれのコンポーネントを 30 回ずつ呼び出し，それに要する時間を測定した．測定結果の平均を表 7.1 に示す．

表 7.1 単一コンポーネントの呼出しに要する時間 (単位: ms)

処　　理	WS	CORBA	EJB
各コンポーネント技術を利用するクライアントプログラム			
コンポーネント呼出し	382.2 (σ=1,1)	119.4 (σ=1.0)	574.1 (σ=243.6)
実装したシステムで動作するクライアントプログラム			
コンポーネント探索	327.2 (σ=3.2)	326.5 (σ=3.0)	326.3 (σ=3.0)
パラメータ探索	0.9 (σ=0.1)	0.8 (σ=0.1)	0.9 (σ=0.1)
インタフェース情報取得	219.8 (σ=3.1)	20.0 (σ=0.7)	-
スタブ生成	-	1062.2 (σ=4.0)	-
コンポーネント呼出し	382.9 (σ=5.6)	125.4 (σ=1.1)	697.1 (σ=297.3)
パラメータ変換	0.0 (σ=0.0)	0.0 (σ=0.0)	0.0 (σ=0.0)
合　　計	930.8 (σ=7.3)	1535.0 (σ=4.5)	1024.3 (σ=298.3)

なお，7.4.3 節で述べたように，CORBA のコンポーネントの呼出しに関しては，事前にスタブプログラムを用意する必要がある．実装したシステムを利用して呼び出す際には，動的にスタブプログラムを生成するが，通常の呼出しでは，コンポーネントの呼出しとは関係なく，事前にスタブプログラムを用意して呼出しを行う．したがって表 7.1 では，実装したシステムを利用した呼出しの場合にのみ，スタブプログラムを生成する際に要する時間を測定している．

(2) 複数コンポーネントを連係させるアプリケーションの呼出し

まず，コンポーネントを組み合わせるプログラムを，BPEL4WS によって記述した．そしてそのプログラムを実行する際に必要とする Web Service のコンポーネント W_1，W_2 を用意しそれぞれ別のサーバに配置した．BPEL4WS で記述したプログラ

ムを，BPEL4WS に対応したサーバ S_1 と，実装したシステムを組み込んだサーバ S_2 で，それぞれ実行した．なお，サーバ S_1 で動作させるプログラムコードには，サーバ W_1，W_2 を利用するように記述した．またサーバ W_1，W_2 には，コンポーネント応答部を配置した．なお比較対象のサーバ S_1 は Web Service のコンポーネントのみを連係させるため，ここでは Web Service のコンポーネントのみを用意して行った．

クライアントから呼出した時に，それぞれのサーバが各コンポーネントを呼び出すために要する時間を測定した．30 回測定した結果の平均を表 7.2 に示す．

表 7.2 コンポーネント連係の際の呼出しに要する時間 (単位: ms)

処　　理			1 回目	2 回目以降
S_1	BPEL インタプリタ実行		1062.5 (σ=115.7)	426.1 (σ=3.9)
	W_1 呼出し		290.2 (σ=2.9)	48.5 (σ=7.3)
	W_2 呼出し		38.2 (σ=3.2)	47.7 (σ=6.8)
	合　　計		1391.0 (σ=115.6)	522.2 (σ=14.2)
S_2	BPEL インタプリタ実行		1068.9 (σ=15.8)	456.7 (σ=1.7)
	W_1 呼出し			
		コンポーネント探索	331.0 (σ=12.1)	-
		パラメータ探索	1.2 (σ=1.5)	-
		インタフェース情報取得	212.1 (σ=273.9)	-
		コンポーネント呼出し	190.3 (σ=3.1)	43.1 (σ=2.9)
		パラメータ変換	0.0 (σ=0.0)	0.0 (σ=0.0)
	W_2 呼出し			
		コンポーネント探索	330.1 (σ=9.6)	-
		パラメータ探索	0.9 (σ=0.1)	-
		インタフェース情報取得	156.7 (σ=4.2)	-
		コンポーネント呼出し	191.2 (σ=42.1)	45.2 (σ=4.2)
		パラメータ変換	0.0 (σ=0.0)	0.0 (σ=0.0)
	合　　計		2482.4 (σ=277.5)	545.0 (σ=5.3)

7.5.2　利用するコンポーネントの切替えに要する時間の測定

実装したシステムでは，利用していたコンポーネントが利用不可能になった場合は，別のコンポーネントを探索して利用する．このような状況が発生した際に，クライアントからの呼出しにどれだけの影響を与えるかを調べるために，利用するコンポーネントの切替えに要する時間を測定する．

まず実装したシステムを組み込んだサーバが，2 つのコンポーネント W_1 と W_2 を連係させるアプリケーションを実行している．そして W_1 を提供しているサーバをネットワークから切断し，利用不能にする．この場合，W_1 を呼び出す時点で，その呼出しに失敗す

る．その結果，利用できないことを検知し，別のコンポーネントを探索する．そして同じ機能を提供する EJB のコンポーネント E_1 を発見して利用する．

この時に，W_1 を利用できないことを検知し， E_1 を呼び出して応答を得るまでの時間を測定した．30 回測定した結果の平均を表 7.3 に示す．

表 7.3 呼び出すコンポーネントの切替えに要する時間 (単位: ms)

処　　理	時 間
W_1 呼出し	
W_1 を利用できないことの検知	75024.0 (σ=7.3)
E_1 呼出し	
コンポーネント探索	328.2 (σ=35.0)
パラメータ探索	0.6 (σ=0.1)
コンポーネント呼出し	522.4 (σ=52.1)
パラメータ変換	0.0 (σ=0.0)
合　　計	75875.2 (σ=58.7)

7.5.3　スケーラビリティに関する測定

実装したシステムでは，IP マルチキャストを利用してコンポーネントの探索を行う．つまりコンポーネントを探索するクライアントによって，多数のサーバへ同時にリクエストが送信される．またクライアントは複数のサーバから応答を得る可能性がある．そこで，メッセージ送受信の処理の影響を調べるために，コンポーネント探索とパラメータ探索に要するメッセージのサイズと，その処理に要する時間を測定した．

7.5.1 節でオーバヘッドを測定した際に利用した EJB のコンポーネントを，実装したシステムで動作するクライアントプログラムから呼び出した．ここでは同一のコンポーネントに対して，要求するタイプを指定するための文字列を 12 byte，24byte，48 byte の 3 種類で設定し，それぞれについて探索を行った．

まず各メッセージの送受信を行う際のパケットを取得し，IP ヘッダ部分を除いたペイロード部分のサイズを測定した．そしてこれらのメッセージの送受信を行う際に，クライアントの処理に要する時間を 30 回測定した．これらの結果を，表 7.4 に示す．

7.5.4　異種分散コンポーネントを連係させるアプリケーションの呼出しに要する時間の測定

実装したシステムでは，異種分散コンポーネントを自動的に発見し，連係させる．それぞれのコンポーネントを連係させるための手順に要する処理時間を調べるために，実装を行ったシステムで対象とした EJB と Web Service と CORBA の 3 種類のコンポーネントを連係させるアプリケーションを作成し，それらを実行した際の処理時間を測定した．

まず，3 つのコンポーネントを逐次実行し，実行したコンポーネントの出力を次に実行

表 7.4 送受信されるメッセージのサイズと処理に要する時間

	コンポーネント探索		パラメータ探索	
	要 求	応 答	要 求	応 答
文字列の長さ : 12 byte				
サイズ (byte)	21	23	19	97
処理時間 (ms)	0.349 (σ=0.037)	0.097 (σ=0.008)	0.066 (σ=0.009)	0.268 (σ=0.028)
文字列の長さ : 24 byte				
サイズ (byte)	33	23	19	97
処理時間 (ms)	0.343 (σ=0.022)	0.096 (σ=0.001)	0.064 (σ=0.005)	0.262 (σ=0.001)
文字列の長さ : 48 byte				
サイズ (byte)	57	23	19	97
処理時間 (ms)	0.337 (σ=0.021)	0.098 (σ=0.014)	0.063 (σ=0.008)	0.262 (σ=0.003)

するコンポーネントに入力することによって連係させる BPEL アプリケーションを作成した．このアプリケーションには本実装により拡張を行った記述を含んでおり，拡張を行った BPEL サーバ上で動作させることで，必要なコンポーネントを探索してから呼び出す．また， 各サーバのコンポーネント応答部を設定し，7.5.1 節でオーバヘッドを測定した際に利用した各コンポーネントを，作成したアプリケーションが利用できるようにした．ここでは探索の結果，まず最初に CORBA コンポーネントが実行され，次に Web Service が，最後に EJB のコンポーネントが実行されるように，該当するサーバのコンポーネント応答部を設定した．

そしてこの BPEL アプリケーションを本提案により拡張を行った BPEL サーバに配置し，クライアントから呼び出した．なおクライアントからの呼出しの際にも，実装したシステムを利用し，必要なコンポーネントを探索してから実行する．30 回呼出しを行った結果の平均を，表 7.5 に示す．

7.5.5　測定結果の分析

(1) アダプタによるコンポーネント呼出しのオーバヘッド
表 7.1 より，Web Service，CORBA，EJB のいずれの場合も，コンポーネント探索により 330 ms 程度のオーバヘッドが発生していることがわかる．コンポーネント探索は，IP マルチキャストにより行っており，該当する複数のサーバから応答を得る可能性がある．そのため，コンポーネント応答メッセージを受信しても，他のサーバからの応答を待ち続ける．本実装では，コンポーネント要求メッセージを送信してから応答を待ち続ける時間を，300 ms に設定した．これによりコンポーネント探索にオーバヘッドが生じた．

表 7.5 異種分散コンポーネント連係の際の呼出しに要する時間 (単位: ms)

処　　理	1 回目	2 回目以降
コンポーネント探索	326.9 (σ=2.0)	327.0 (σ=3.8)
パラメータ探索	0.6 (σ=0.1)	0.6 (σ=0.0)
インタフェース情報取得	497.9 (σ=11.3)	222.3 (σ=28.1)
BPEL アプリケーション呼出し	3772.1 (σ=275.7)	529.2 (σ=54.9)
(内訳)		
BPEL インタプリタ実行	745.9 (σ=22.1)	418.8 (σ=5.3)
CORBA コンポーネント呼出し		
コンポーネント探索	326.3 (σ=2.8)	-
パラメータ探索	0.8 (σ=0.18)	-
インタフェース情報取得	21.5 (σ=15.9)	-
スタブ生成	909.7 (σ=67.3)	-
コンポーネント呼出し	241.9 (σ=29.7)	29.0 (σ=8.3)
パラメータ変換	0.0 (σ=0.0)	0.0 (σ=0.0)
Web Service 呼出し		
コンポーネント探索	328.2 (σ=0.94)	-
パラメータ探索	0.8 (σ=0.13)	-
インタフェース情報取得	155.5 (σ=17.5)	-
コンポーネント呼出し	196.8 (σ=30.6)	46.9 (σ=6.3)
パラメータ変換	0.0 (σ=0.0)	0.0 (σ=0.0)
EJB コンポーネント呼出し		
コンポーネント探索	326.8 (σ=2.9)	-
パラメータ探索	0.8 (σ=0.15)	-
コンポーネント呼出し	517.1 (σ=255.7)	34.5 (σ=52.9)
パラメータ変換	0.0 (σ=0.0)	0.0 (σ=0.0)
合　　計	4597.5 (σ=268.8)	1079.1 (σ=58.6)

コンポーネント探索を除いたオーバヘッドについては，利用するコンポーネントのそれぞれのアーキテクチャによって異なる．これは 7.4.3 節で述べたように，メディエータ部がそれぞれのアーキテクチャによって，異なった方法でインタフェース情報を取得して，呼出しを行うためである．Web Service のサーバ実装では，クライアントが HTTP を経由してインタフェース情報を記述した WSDL の文書を取得する．サーバはその時点で WSDL の文書を動的に生成し，クライアントに返す．また CORBA コンポーネントでは，インタフェース情報を予め CORBA-IDL で記述して公開しておき，クライアントはこれを HTTP で取得している．そして取得したインタフェース情報にしたがってスタブを生成する．どちらの場合も，これらの処理のために発生するオーバヘッドが大きくなっている．なお EJB の場合は，これらの処理を行わないため，オーバヘッドが発生しない．またどの場合でも，パラメータ探索や，メソッドのシグネチャやパラメータの変換のためには，ほとんどオーバヘッドを発生しない．

また表 7.2 や表 7.5 に示すように，複数のコンポーネントを連係させるアプリケーションを実行する場合も，それぞれのコンポーネントについて同様にオーバヘッドが生じる．

このように通常のそれぞれのコンポーネント呼出しに対して，合計で約 330 ms (EJB の場合) 〜 約 1400 ms (CORBA の場合) のオーバヘッドが生じている．その結果，コンポーネントの呼出しに要する時間が，約 2 倍 〜 約 13 倍に増加している．

発生したオーバヘッドのうち，コンポーネント探索に要する時間は，接続するネットワークのターンアラウンドタイムを参考に調節できる．また複数のサーバからの応答を待ち続けずに，最初に応答が到着したものを利用するようにすれば，300ms の待ち時間が短縮できる．EJB のようにパラメータ応答メッセージに含めてサーバから送信すれば，インタフェース情報の取得によって発生するオーバヘッドを無くすことができる．CORBA では通常はスタブのコードを生成して呼び出すが，メディエータ部が直接 CORBA のコンポーネントを呼び出せば，スタブの生成によって発生するオーバヘッドも無くすことができる．このように，実装を改良することによって，オーバヘッドをかなり減らすことができる．

さらに表 7.2 や表 7.5 に示したように，コンポーネントを連係させるサーバでは，探索を行った結果をキャッシュするため，2 回目以降の呼出しではオーバヘッドを生じない．利用状況にも依るが，これによりある程度のオーバヘッドを軽減できる．

また 2.4 節で述べたように，分散コンポーネントを利用するためには，通常はまずレジストリから必要なコンポーネントを発見し，レジストリや発見したコンポーネントを提供するサーバからインタフェースに関する情報を入手する．そして入手したインタフェースに関する情報にしたがって，コンポーネントの利用者がクライアントプログラムを作成する．一方提案システムでは，コンポーネントを発見する前にクライアントプログラムが作成されており，コンポーネントを発見した後には，発見したコンポーネントを呼び出せるように自動的に調整している．すなわち通常の分散コンポーネントの利用方法では，仮にコンポーネントが発見できたとしても，利用者は発見したコンポーネントを呼び出すようにプログラムコードを開発する必要

があり，発見したコンポーネントを自動的に利用することはできない．提案システムでは長いオーバヘッドが生じたが，通常の分散コンポーネントの利用方法で同様のことを利用者が行うとすると，それよりも更に長い時間を要する．その上，コンポーネントの利用者にアプリケーションの開発知識を求めることになる．

(2) コンポーネントが利用不可能な場合の切替え処理について
コンポーネントを連係させるアプリケーションを実行している際に，呼び出そうとするコンポーネントの実行に失敗すると，他のコンポーネントを探索して処理を切り替える．

呼び出されるサーバをネットワークから切断した場合は，BPEL4WS の実行エンジンが HTTP リクエストによるコネクションがタイムアウトするまで失敗したことを検知できない．そのため表 7.3 のように，非常に長い時間を要した．しかし検知すると，すぐに他のコンポーネント E_1 を発見して処理を切り替えることができた．

提案システムでは，クライアントはコンポーネント探索を行ってからサーバを呼び出す．サーバがネットワークから切断されている場合にはコンポーネント探索に対する応答が得られないので，HTTP によるコンポーネント呼出しを行わない．またサーバがネットワークに接続した状態で，何らかの要因によってコンポーネントが利用不可能になっている場合には，HTTP によるコネクションが切断されるか，HTTP によるコネクションが成功して利用不可能である応答を得るかのいずれかになる．このようにどちらの場合でも，HTTP コネクションのタイムアウトによってコンポーネント実行の失敗を検知するような状況は発生しない．そのような状況は，探索を行っている結果をクライアントによってキャッシュしているときに，該当するサーバが切断された場合にのみ発生する．よって，探索結果をキャッシュとして保持している場合に，該当するサーバのネットワークへの接続性をクライアントから定期的に確認することで，この状況をある程度は回避できる．

(3) メッセージ処理のスケーラビリティについて
コンポーネント探索は，IP マルチキャストを利用して行われる．したがって，多数のサーバとコンポーネント要求やコンポーネント応答のメッセージのやりとりをするため，どれだけの大きさのメッセージがネットワーク上に送出されるかが問題になる．

表 7.4 によると，コンポーネント要求メッセージとコンポーネント応答メッセージは，非常に小さいことが分かる．ネットワーク上に多数のサーバが存在し，多数の要求や応答がやりとりされるとしても，非常に小さなメッセージであるので，ネットワークの負荷を高めるとは考えにくい．なおコンポーネント要求メッセージは必要な機能を指定する文字列を含むため，文字列の長さによってメッセージサイズが線形に変化する．

また，コンポーネント応答メッセージが多数のサーバから送信された場合は，クライアントがコンポーネント応答メッセージを処理するのにどれだけの時間を要するかが問題になる．しかしこれについても，非常に短い時間で終わっているため，クライアントの負荷にそれほどの影響を与えないと言える．なお，実装したシステム

のコンポーネント探索機構では，クライアントが多数のサーバからの応答を得ることを考慮して，なるべくメッセージサイズを小さくするために，パラメータ応答メッセージに含まれるようなコンポーネントの呼出しに関する情報を含まない．そしてそれらの情報を，ユニキャストで行われるパラメータ探索によって得るようにしている．また，クライアントの処理を単純にするために，コンポーネント応答メッセージについては，サーバ内での固定長の番号によってコンポーネントを識別し，メッセージの長さは変化しないようにしている．

7.6　まとめ

本章では, 本研究で提案する異種分散コンポーネントの Plug and Play プラットフォームを提案し，その設計と実装を述べた．

本章で提案したプラットフォームは，分散コンポーネントの存在するサーバやそれを利用するクライアントをネットワークに接続することで，それらを Plug and Play で動作させる．提案したプラットフォームを利用することで，コンポーネントを利用するクライアントや，コンポーネントを連係させるサーバは，提案するプラットフォームを利用し，必要とするコンポーネントとそれを提供するサーバを利用時に自動的に発見する．これによって分散コンポーネントシステムを構成するすべての端末を Plug and Play で動作させる．

提案したプラットフォームは，コンポーネントの存在を管理する特別な機構を設けずに，コンポーネントを利用時に探索することを可能にする．そして，どのようなソフトウェアを連係させるのかを事前に把握できなくても，探索して発見したコンポーネントを，それらの異種性によらずに連係して動作させることを可能にする．

本章で実装したプラットフォーム上でアプリケーションを実行させる際には，コンポーネントの呼出し時には必要なコンポーネントを探索するため，オーバヘッドが発生する．評価実験の結果，コンポーネントの探索のためには，約 330 ms のオーバヘッドを生じていることが計測された．この時間については，パラメータの調節により短縮することができる．またコンポーネントの探索のためには，IP マルチキャストを利用した，したがって，多数のサーバとコンポーネント要求やコンポーネント応答のメッセージのやりとりが発生する．評価実験の結果，それぞれのメッセージのサイズと，その処理時間は非常に小さいことが分かった．ネットワーク上に多数のサーバが存在し，多数の要求や応答がやりとりされるとしても，非常に小さなメッセージであるので，ネットワークの負荷を高める可能性は低いと考えられる．

第8章　実装したシステムの有効性に関する考察

本章では，提案した HeteroSOA フレームワークの有効性を考察するために，まず異種分散コンポーネントを利用するアプリケーションの開発支援システムを利用して効率的に開発するための条件を挙げる．また，異種分散コンポーネントの Plug and Play プラットフォームでは，通常の手順とは異なる手順で，ソフトウェア開発を行うことになる．そのことに起因して，このプラットフォームの適用先についてを述べる．そして，コンポーネント探索のために IP マルチキャストを利用したことに対する有効性についてを述べる．また，異種分散コンポーネントを利用するアプリケーションの開発支援システムと異種分散コンポーネントの Plug and Play プラットフォームでは異種性を吸収するための方法が異なるが，この相違の理由について，利用シナリオとあわせて考察を行う．

8.1　効率的に開発するための条件

HeteroSOA フレームワークが提供するアプリケーション開発支援システムを利用することで開発を効率的に行うためには，開発支援システムの利用者であるアプリケーションプログラムの開発者がある程度の知識や経験を保持していることを前提とする．それらを次に示す．

A) 分散コンポーネントに関する知識
個々の分散コンポーネント技術については深く知る必要がないが，ワークフローエディタを利用して設計を行う際にどの程度の詳細度でアクティビティ図を記述すれば良いかを判断する必要がある．そのため，分散コンポーネントを利用することによって一般的にどの程度の規模のアプリケーションプログラムが開発されているのかを知っている必要がある．

B) UML を利用したソフトウェアの開発経験
ワークフローエディタ上で描画するアクティビティ図の文法を理解している必要があり，さらにアクティビティ図を利用したソフトウェア開発経験があることが望ましい．

C) アクティビティとコンポーネントの対応を判断するための知識
あるアクティビティを実現できるのはどのコンポーネントなのか，アクティビティの入出力は各コンポーネントのどの変数に相当するのかなど，アクティビティとコンポーネントの対応を判断する必要がある．したがって，抽象化されて記述したアクティビティによって，実際にはどのような処理を行うのかを判断できなければな

らず，開発しているアプリケーションプログラムの設計を十分理解している必要がある．

D) アダプタの選択基準に関するドメイン知識
6.6.4 節に述べたように，アダプタを選択する際の基準は，アプリケーションの開発者が予め配置モジュールに指定する．この基準は開発するアプリケーションプログラムの利用目的によって異なるため，開発者は開発したアプリケーションプログラムの利用されるドメインに関する知識を十分に持っていることが望ましい．

8.2　Plug and Play プラットフォームの有効な適用先

異種分散コンポーネントの Plug and Play プラットフォームを利用するためには，コンポーネントを発見する前に，それを利用するアプリケーションを開発することになる．通常のコンポーネントウェアにおけるアプリケーション開発では，利用するコンポーネントが決定してからソフトウェアの開発を行う．しかし前述のように Plug and Play を実現するために，通常とは異なる手順で，ソフトウェア開発を行うことになる．このため，どのような機能のコンポーネントが存在していてどれを利用するかという見当が予めある程度付いている場合には有効に利用できる．逆に，どのような機能のコンポーネントが存在するかまったく想定できず，発見したコンポーネントの機能によってプログラムコードの構造が決定するような場合には利用できない．したがってソフトウェア部品市場が発達し，機能ごとに交換可能なソフトウェア部品群が流通するようになると，Plug and Play プラットフォームを有効に利用できる．例えば，ホームアプライアンスのような組込み製品に展開し，ハードウェアベンダーが提供する製品の機能ごとにコンポーネントを組み込むことを考えてみる．この場合，利用者は必要な機能を提供する製品を購入しネットワークに接続することで，自動的に組み合わせることができるようになる．このような場合にネットワークに設置した装置に対して機能追加が簡単に行えるようになり，Plug and Play プラットフォームを有効に利用できる．

8.3　IP マルチキャスト利用の有効性

Plug and Play プラットフォームの実装では，コンポーネント探索に IP マルチキャストを利用しており，これによって同一ネットワーク上に存在する未知のコンポーネントサーバを発見できるようにしている．Plug and Play プラットフォームでは，3.3 節に示したような比較的狭い環境でのコンポーネント発見・利用を想定しており，グローバルネットワークへのコンポーネント探索を対象にしていない．そのため，ローカルネットワークで IP マルチキャストを利用することによって，コンポーネント探索を実現している．

ここで Plug and Play プラットフォームの対象外ではあるが，より広域に探索を行いたい場合の拡張を考えてみる．なお IP マルチキャストグループを形成するための特別なプロトコル [Fenner1997] を利用することによって，グローバルネットワークへマルチキャストできるようになる．しかしそのためにはルータが対応することが前提になる．Plug and Play 環境では，利用するネットワーク環境を特定できないため，それを仮定することは

困難であるので，ここではそれ以外の方法を挙げる．

広域網を対象に探索を行うためには，ディレクトリサービスのような階層的に情報を管理できる方式が必要になる．その場合には，ディレクトリサービスからコンポーネントを発見したり，ディレクトリサービスにコンポーネントの情報を登録したりすることになる．この際に，ディレクトリサービスを自動発見するために，Plug and Play プラットフォームを利用できる．また，それほど広域ではない複数ネットワーク間での Plug and Play については，Plug and Play プラットフォームに加えて，各ネットワークに単純にリクエストを転送するための装置を設置することによっても実現できる．いずれの場合も Plug and Play プラットフォームを発展させることにより実現できる．

8.4　異種性を吸収する方法の相違について

異種分散コンポーネントを利用するアプリケーションの開発支援システムでは，それぞれの分散コンポーネントによる呼出し方法を，Proxy により Web Service 化している．一方，異種分散コンポーネントの Plug and Play プラットフォームでは，呼出し側のメディエータ部が，それぞれの分散コンポーネントの呼出し方法に変換して呼出しを行っている．

5.4 節で例を挙げたように，前者の場合は，グローバルに公開されているコンポーネントを静的に組み合わせることを想定している．また後者の場合は，家庭内ネットワークのような比較的狭い環境で，コンポーネントを動的に組み合わせることを想定している．

前者の場合には，異なるネットワークに存在するサーバで動作するコンポーネントを組み合わせることを考慮した．Proxy を Web Service として生成しているので，SOAP による HTTP アクセスが可能である．したがってサーバの存在するネットワークに，自動生成した Proxy を配置することにより，該当するコンポーネントを Web Service として外部ネットワークから呼び出すことが可能になる．

一方後者の場合には，ローカルネットワークでの Plug and Play を想定しており，呼出し方法を Web Service に統一する必要性はない．そこで Proxy のようなプログラムによって呼出し方法を統一せずに，それぞれのコンポーネントを提供するサーバ独自の方法によって，コンポーネントの呼出しを行っている．提案する Plug and Play プラットフォームに対応するサーバには，コンポーネント要求に応えるコンポーネント応答部のみを追加すれば良い．サーバに Proxy のような変換機構を用意する場合は，サーバは Web Service のサーバとしても動作する必要があり，このために多くのリソースを要求する．しかしコンポーネント応答部は，表 7.4 に測定したように，非常にサイズの小さいメッセージのやりとりしか行わない．そのため，Proxy を配置する場合に比べ，サーバに対するリソースの要求を抑えることができる．

なおどちらの場合も，呼び出す種類に応じ，それぞれのサーバ実装が提供するクライアントライブラリを介して，それぞれの異種分散コンポーネントを呼び出す．前者の場合は，Proxy を動作させるコンピュータ上に，そして後者の場合はメディエータ部を動作させるクライアントや BPEL サーバに，それらのクライアントライブラリを配置することになる．現段階の実装では，Web Service を利用するためのクライアントライブラリは約 1868 KB，EJB を利用するためのクライアントライブラリは約 3280 KB のファイルサイズである．近年普及している記憶装置のディスク容量と比較すればそれほど大きなものではな

いが，特にクライアントコンピュータから利用する場合には，これらファイルを配置できるディスク空間を有していることが，利用の際の制限になる可能性がある．なお今回の実装ではそれぞれのクライアントプログラムや，BPEL サーバ実装は，Java で動作するものを対象とした．CORBA のコンポーネント呼出しのためのライブラリについては，Java 実行環境に付属しているため，特別に追加する必要がない．

第9章　結 論

本論文では，まず SOC においてさまざまなベンダの提供するサービスを連係させるアプリケーションを開発し，運用する際の課題を分析した．そして，本研究では次の 3 つの課題に着目した．

A) コンポーネントの異種性
さまざまなベンダによって提供されるサービスを組み合わせることを考えると，それらは独自に開発されており，サービスの性能や質や実装がプロジェクトチームにより異なる．このため，サービスを構成するコンポーネントを呼び出す方法や，個々のコンポーネントが利用するデータモデルやコントロールモデルが異なる場合がある．

B) 連係するアプリケーションの信頼性
個々のサービスの性能や質や実装がプロジェクトチームにより異なることに加え，それぞれのサービスが独自に管理されているため，全体の性能や信頼性が保証できない．特に開発者が設計を行った時点では利用できたコンポーネントが，利用者がアプリケーションプログラムを実行しようとする時点では利用できないことがある．このことは，分散コンポーネントを組み合わせるアプリケーションの信頼性を大きく下げる．

C) 実行時における柔軟なサービス連係
一般的にコンポーネントを利用したアプリケーション開発は，利用するコンポーネントが開発時に静的に決定していて，それらを組み合わせることによって行われる．しかし，例えばモバイルや組込み機器のような機器をサーバとして利用することを考えると，それらのサーバ上で動作するコンポーネントがネットワーク上のどこに存在するのかを，事前に想定できない場合がある．しかしそのようなさまざまな形態の端末で運用されているサービスも，柔軟に連係できることが望ましい．そのために，必要なサービスをアプリケーション実行時に自動的に発見して連係する必要がある．

本研究ではこれらの課題を解決するため，異種分散コンポーネントの連係を支援する機構 HeteroSOA フレームワークを提案し，設計・実装を行った．

まず，組み合わせるコンポーネントが開発時に静的に決定している場合の課題を解決するための，開発支援システムを提供した．この環境を利用し，組み合わせるコンポーネントを開発者が指定することにより，異種分散コンポーネントを連係させるプログラムコードを自動的に生成できるようにした．これを利用することにより，開発者は異種分散コンポーネントを連係させるアプリケーションを開発し，利用者はそのアプリケーションを介して，それらを呼び出して連係させることができる．そしてこの開発の際の組合わせの指

定や保存には，UML アクティビティ図や，BPEL4WS 等の標準化された手法を利用できるようにした．そしてこの開発支援システムでは，連係するアプリケーションの信頼性の課題を解決するために，アプリケーション実行時にコンポーネントの呼出しに障害が発生した際に，開発時に指定した別のコンポーネントに処理を切り替えられるような仕組みを用意した．コンポーネント呼出しの障害時には，障害の発生したコンポーネントを利用しないアプリケーションプログラムに処理を自動的に切り替えることにより，信頼性の低下を回避する．

そして，実行時に動的にコンポーネントを組み合わせられるようにするための，プラットフォームを提供した．このプラットフォームを利用し，生成したアプリケーションプログラムを実行する．このプラットフォームには，ローカルネットワークに存在するコンポーネントを動的に探索して利用するためのミドルウェアを含む．分散コンポーネントが動作するサーバやそれを利用するクライアントは，このミドルウェアを利用することにより，実行時に必要なサービスを提供するコンポーネントを自動的に発見し，発見したコンポーネントを動的に連係させることができる．またコンポーネント異種性の課題については，動的に組み合わせる場合にも発生する．この課題に対応するため，実行時に異種性を動的に吸収する仕組みも提供した．これらにより，アプリケーション開発時にネットワーク上のどこに存在するのか想定できなかった異種分散コンポーネントでも，利用者がアプリケーションを実行時に柔軟に連係することができるようにした．

そして，実装したシステムの性能を評価するための実験を行った．HeteroSOA フレームワークの提供する開発支援システムによって開発したアプリケーションでは，コンポーネント間の異種性を吸収するためのプログラムを介してコンポーネントの呼出しを行う．このために，コンポーネントの処理時間が約 30 ms 増加した．また実行時にコンポーネントを動的に探索して利用する場合は，必要なコンポーネントを探索するために，コンポーネントの呼出しの時間が約 330 ms 増加した．なお後者については，パラメータの調節により，短縮することができる．

HeteroSOA フレームワークにより，色々なベンダの提供するさまざまなサービスを連係させるシステムを実現することが可能になる．その際には，利用するコンポーネントを静的に開発者が指定することによって開発を行う方法と，実際に運用を行う時に利用できるものを探索して利用する方法の 2 通りで，システムを構築できる．そして異なった形式の分散コンポーネントにおいても，それらを実現させる方法を示した．これにより，分散コンポーネント技術により提供されるサービスを，より容易に組み合わせて，容易に運用できるようになると期待できる．したがって，今まで分散コンポーネントを利用しなかった分野にも分散コンポーネントを基盤技術として適用し，インターネットをサービス構築のための基盤として利用して新しいシステムを構築できるようになる．

今後の課題を，次に挙げる．

(1) オーバヘッドの短縮

HeteroSOA フレームワークでは，組み合わせるコンポーネントを静的に決定してアプリケーションを開発する場合も，動的にコンポーネントを連係させる場合も，どちらの場合もコンポーネントの呼出しの際にオーバヘッドを生じている．異種性の吸収やコンポーネントの探索のためには，このオーバヘッドを発生させなくすることは不可能であるが，短縮する方法を検討する必要がある．このためには，個々の

オーバヘッドを短縮することはもちろんであるが，[Srivastava2006] のように外部のコンポーネントへの呼出し自体を最適化することも考えられる．そして後者を検討することによって，多数のコンポーネントへの呼出しを行うアプリケーションのワークフローを最適化し，アプリケーション全体のオーバヘッドも短縮できる．

(2) セキュリティに関する考慮
セキュリティに関しては，組み合わせるコンポーネントを静的に決定してアプリケーションを開発する場合には，組合わせを開発者が決定するのでそれほど問題にはならないと考えられる．しかし，動的にコンポーネントを連係させる場合には，セキュリティを考慮する必要がある．現状の実装ではセキュリティに対する考慮をしていないため，どのようなコンポーネントでも発見し，連係させることができる．セキュリティに対しては，次の 3 種類を考慮する必要がある．

(a) 誤った応答を行うサーバの存在
間違えた設定が行われているサーバや，悪意のあるホストが，自分が提供していないコンポーネントに対するコンポーネント要求に応答を返してしまうことを考慮する必要がある．

(b) メッセージの秘匿性
悪意のあるホストが探索メッセージや応答メッセージを盗聴したり，改竄したり，偽造したりすることを考慮する必要がある．

(c) クライアントからのアクセスコントロール
コンポーネントを利用させたくないクライアントには，コンポーネントの探索ができないようにする必要がある．

PKI（Public Key Infrastructure) のような仕組みをつくり，鍵配布のための認証局サーバを用意し，その鍵データを利用して暗号化通信を行うことで，これらすべてを完全に解決できるだろう．しかしそのような方法では，事前に鍵データを配布する必要がある．用途によっては問題にならない可能性はあるが，一般的には Plug and Play に不向きである．このため，これとは異なる手段でセキュリティを確保できる何らかの手段を検討する必要性がある．なお，クライアントからのアクセスコントロールのみに関しては，[Hung2005] にあるような方法で End-to-End のアクセスコントロールを行う仕組みによって，解決することも考えられる．

謝 辞

本研究の機会を与えて下さり，絶えず御指導，御助言を頂いている，高田 眞吾 助教授に，深く感謝致します．

また，私が慶應義塾大学理工学部在学中より御指導をして頂き，御退職後も絶えず御助言を頂き，さらに忙しい中，本論文のチェックを行って頂いた，中央大学 土居 範久 教授に，深く感謝致します．

忙しい中，本論文のチェックを行って頂いた，山本 喜一 助教授，重野 寛 助教授，遠山 元道 専任講師 に，深く感謝の念を表します．

また，修士課程在学中に御助言を頂いていた，飯島 正 助手 に，感謝の念を表します．

本研究の着想に，日本電気株式会社在職中に習得した基礎技術が与えた影響は非常に大きく，そのような機会を与えて下さった，日本電気株式会社 竹内 章平 氏，並びに 北村 浩 氏 に感謝の念を表します．

なお本研究は，文部科学省科学技術振興調整費「環境情報獲得のための高信頼性ソフトウェアに関する研究」の支援により遂行しています．関係諸機関に，深く感謝の念を表します．

そして，日頃より共に研究活動に励んできた高田研究室の皆様に，心より感謝致します．特に同じ領域の研究を行い，論文の共著者に加わって頂いている，日本電気株式会社 日比 洋介 氏と，株式会社 日立製作所 河野 泰隆 氏には，深く感謝の念を表します．

最後に，研究活動を支えてくれた家族と，日頃より色々な相談に乗って頂いている友人達に，感謝致します．

論文目録

論文誌

[1] 名倉 正剛，河野 泰隆，高田 眞吾，土居 範久: “異種分散コンポーネントを利用するアプリケーションの開発を支援するシステム”，情報処理学会論文誌， Vol.47，No.2，pp.484-494，2006 年 2 月．

[2] 名倉 正剛，高田 眞吾，土居 範久: “シームレスコンピューティングのための異種分散コンポーネントの Plug and Play 環境”，情報処理学会論文誌，Vol.48，No.2，2007 年 2 月．

国際会議発表

[1] Masataka Nagura, Shingo Takada, Tadashi Iijima and Norihisa Doi: “Automated Adapter Generation for Gluing Heterogeneous External Components”, Proceedings of 14th International Conference on Software & Systems Engineering and their Applications (ICSSEA 2001), Vol.2, pp.1-8, December 2001.

[2] Masataka Nagura, Yosuke Hibi, Shingo Takada and Norihisa Doi: “Integration of Heterogeneous Distributed Components Using Workflow Information”, Proceedings of the IASTED International Conference on SOFTWARE ENGINEERING (SE 2004), pp.148-153, February 2004.

国内研究会発表

[1] 名倉 正剛，高田 眞吾，飯島 正，土居 範久: “異種コンポーネントの結合を行うアダプタの自動生成”，情報処理学会第 62 回全国大会講演論文集，5Z-05，2001 年 3 月．

[2] 名倉 正剛，高田 眞吾，土居 範久: “異種分散コンポーネントを対象にした Plug and Play 環境の提案”，情報処理学会研究報告，Vol.2004，No.107 (2004-DPS-120)，pp.49-54，2004 年 11 月．

参考文献

[ActiveBPEL2006] Active Endpoints: “ActiveBPEL Engine Overview” (2006).
http://www.active-endpoints.com/active-bpel-engine-overview.htm

[Aoyama1996] 青山 幹雄：“コンポーネントウェア：部品組み立て型ソフトウェア開発技術”, 情報処理, Vol. 37, No.1, Jan. 1996, pp.91-97 (1996).

[Aoyama1998] 青山 幹雄, 中所 武司, 向山 博 編: “コンポーネントウェア”, 共立出版 (1998).

[Apache2006] The Apache Software Foundation: “Apache Axis”, Apache Web Services Project (2006).
http://ws.apache.org/axis/

[Austin2004] Daniel Austin, Abbie Barbir, Christopher Ferris and Sharad Garg: “Web Services Architecture Requirements”, W3C Working Group Note, The World Wide Web Consortium (2004).
http://www.w3.org/TR/2004/NOTE-wsa-reqs-20040211/

[BEA2004] BEA Systems: “BEA eLink Documentation” (2004).
http://e-docs.bea.com/elink/

[Beatty2004] John Beatty, Gopal Kakivaya, Devon Kemp, Brad Lovering, Bryan Roe et al.: “Web Services Dynamic Discovery (WS-Discovery)”, Microsoft Corporation (2004).
http://msdn.microsoft.com/ws/2004/02/discovery/

[Berners-Lee1998] Tim Berners-Lee, Roy T. Fielding and Larry Masinter:: “Uniform Resource Identifiers (URI): Generic Syntax”, RFC 2396, The Internet Engineering Task Force (1998).
http://www.ietf.org/rfc/rfc2396.txt

[Bichler2006] Martin Bichler and Kwei-Jay Lin: “Service-Oriented Computing”, IEEE Computer Society Press, Vol. 39, Issue 3, pp.99-101 (2006).

[Booth2004] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, et al.: “Web Services Architecture”, W3C Working Group Note, The World Wide Web Consortium (2004).
http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/

[Brogi2006A] Antonio Brogi and Razvan Popescu: “Automated Generation of BPEL Adapters”, Proceedings of the 4th International Conference on Service Oriented Computing (ICSOC06), LNCS 4294, Springer-Verlag, pp.27-39 (2006).

[Brogi2006B] Antonio Brogi, Sara Corfini, Jos’e F. Aldana and Ismael Navas: “Automated Discovery of Compositions of Services Described with Separate Ontologies”, Proceedings of the 4th International Conference on Service Oriented Computing (ICSOC06), LNCS 4294, Springer-Verlag, pp.509-514 (2006).

[Brown1996] Alan W. Brown: “Component-Based Software Engineering”, IEEE Computer Society Press (1996).

[Brown2000] Alan W. Brown: “Large-Scale, Component-Based Development”, Prentice Hall PTR (2000).

[Brown2003] Alan W. Brown, Simon Johnston and Kevin Kelly:: “Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications”, A Rational Software White Paper, IBM Corporation (2002).
http://www.oasis-open.org/committees/download.php/17566/06-04-00002.000.pdf

[Cerqueira1999] Renato Cerqueira, Carlos Cassino and Roberto Ierusalimschy: “Dynamic Component Gluing Across Different Componentware System”, Proceedings of the International Symposium on Distributed Objects and Applications (DOA ’99), pp.362-371 (1999).

[Chang1998] Dan Chang and Dan Harkey: “Client/Server Data Access with Java and XML”, Sons Incorporated (1998).

[Christensen2001] Erik Christensen, Francisco Curbera, Greg Meredith and Sanjiva Weerawarana: “Web Service Definition Language (WSDL) 1.1”, W3C Note, The World Wide Web Consortium (2001).
http://www.w3.org/TR/2001/NOTE-wsdl-20010315

[Chung2004] Sam Chung, Lai Hong Tang and Sergio Davalos: “A Web Service Oriented Integration Approach for Enterprise and Business-to-Business Applications”, Proceedings of the 5th International Conference on Web Information Systems Engineering (WISE 2004), LNCS 3306, Springer-Verlag, pp.510-515 (2004).

[Clement2004] Luc Clement, Andrew Hately, Claus von Riegen and Tony Rogers: “UDDI Version 3.0.2”, UDDI Spec Technical Committee Draft, OASIS UDDI Spec Technical Committee (2004).
http://uddi.org/pubs/uddi-v3.0.2-20041019.htm

[Constantinescu2004] Ion Constantinescu, Boi Faltings and Walter Binder: “Type based service composition”, Proceedings of the 13th International World Wide Web Conference (WWW2004), pp.268-269 (2004).

[Dumas2002] Marlon Dumas and Arthur H. M. ter Hofstede: “UML Activity Diagrams as a Workflow Specification Language”, Proceedings of the 14th International Conference on Advanced Information Systems Engineering, pp.76-90 (2002).

[Fenner1997] William C. Fenner: “Internet Group Management Protocol, Version 2”, RFC 2236, The Internet Engineering Task Force (1997).
http://www.ietf.org/rfc/rfc2236.txt

[Forster2004] Florian Forster and Hermann De Meer: “Discovery of Web Services with a P2P Network”, Proceedings of the International Conference on Computational Science 2004 (ICCS2004), LNCS 3038, Springer-Verlag, pp.90-97 (2004).

[Goedicke2000] Michael Goedicke: “Service Integration”, Proceedings of the 2nd International Workshop on Engineering Distributed Objects (EDO 2000), LNCS 1999, Springer-Verlag, pp.231-234 (2000).

[Goland1999] Yaron Y. Goland, Ting Cai, Paul Leach, Ye Gu and Shivaun Albright: “Simple Service Discovery Protocol/1.0”,IETF Internet-draft (*expired*), The Internet Engineering Task Force (1999).
http://www.upnp.org/download/draft_cai_ssdp_v1_03.txt

[Gold2004] Nicolas Gold, Andrew Mohan, Claire Knight and Malcolm Munro: “Understanding Service-Oriented Software”, IEEE Software, March/April 2004, pp.71-77 (2004).

[Gomes2005] Roberta L. Gomes, Guillermo J. Hoyos Rivera. and Jean Pierre Courtiat: “Loosely-Coupled Integration of CSCW Systems”, Proceedings of the 5th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS2005), LNCS 3543, Springer-Verlag, pp.38-49 (2005).

[Gudgin2003] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau and Henrik Frystyk Nielsen: “SOAP Version 1.2 Part 1: Messaging Framework”, W3C Recommendation, The World Wide Web Consortium (2003).
http://www.w3.org/TR/2003/REC-soap12-part1-20030624/

[Guttman1999] Erik Guttman, Charles Perkins, John Veizades and Michael Day: “Service Location Protocol, Version 2”, RFC 2608, The Internet Engineering Task Force (1999).
http://www.ietf.org/rfc/rfc2608.txt

[Haas2004] Hugo Haas and Allen Brown: “Web Services Glossary”, W3C Working Group Note, The World Wide Web Consortium (2004).
http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/

[Heuvel2003] Willem-Jan Van Den Heuvel and Zakaria Maamar: “Moving Toward a Framework to Compose Intelligent Web Services”, Communications of the ACM, October 2003/Vol. 46, No. 10, pp.103-109 (2003).

[Hofstede2006] Arthur ter Hofstede, Lachlan Aldred and Lindsay Bradford: “YAWL (Yet Another Workflow Language)” (2006).
http://yawlfoundation.org/index.php

[Hung2005] Patrick C.K. Hung, Dickson K.W. Chiu, W.W. Fung, William K. Cheung, Raymond Wong, et al.: “Towards End-to-End Privacy Control in the Outsourcing of Marketing Activities: A Web Service Integration Solution”, Proceedings of the IFIP 4th International Conference on Entertainment Computing (ICEC2005), pp.454-461 (2005).

[IBM2004] IBM T. J. Watson Research Center: “IBM Business Process Execution Language for Web Services Java Run Time (BPWS4J)”, alphaWorks, IBM Corporation (2004).
http://www.alphaworks.ibm.com/tech/bpws4j/

[Jordan2006] Diane Jordan, John Evdemon, Alexandre Alves, Assaf Arkin, Sid Askary, et al.: “Web Services Business Process Execution Language Version 2.0”, Public Review Draft, OASIS WSBPEL Technical Committee (2006).
http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.pdf

[Kaplan1999] Alan Kaplan and Bradley Schmerl: “Automating Interoperability For Heterogeneous Software Components”, Proceedings of the International Workshop on Component-based Software Engineering, pp.111-114 (1999).

[Kim2005] Kwanghoon Kim and Ilkyeun Ra: “A Process-Driven e-Business Service Integration System and Its Application to e-Logistics Services”, Proceedings of the 6th International Conference on Web Information Systems Engineering (WISE 2005), LNCS 3806, Springer-Verlag, pp.669-678 (2005).

[Kim2006] Ji Hyun Kim, Won Il Lee, Jonathan Munson and Young Ju Tak: “Services-Oriented Computing in a Ubiquitous Computing Platform”, Proceedings of the 4th International Conference on Service Oriented Computing (ICSOC06), LNCS 4294, Springer-Verlag, pp.601-612 (2006).

[Lassing1998] Nico H. Lassing and Daan B. B. Rijsenbrij and Johannes C. van Vliet: “A View on Components”, Proceedings of the 9th International DEXA Workshop on Database and Expert Systems Applications, pp.768-777 (1998).

[Leach2005] Paul J. Leach, Michael Mealling and Rich Salz: “A Universally Unique IDentifier (UUID) URN Namespace”, RFC 4122, The Internet Engineering Task Force (2005).
http://www.ietf.org/rfc/rfc4122.txt

[Levine2001] Paul Levine, Jim Clark, Cory Casanave, Kurt Kanaskie, Betty Harvey, et al.: “ebXML Business Process Specification Schema Version 1.01”, Business Process

Project Team, UN/CEFACT and OASIS (2001).
http://www.ebxml.org/specs/ebBPSS.pdf

[Leymann2001] Frank Leymann: "Web Services Flow Language (WSFL 1.0)", IBM Corporation (2001).
http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf

[Mahmoud2005] Qusay H. Mahmoud: "Service-Oriented Architecture (SOA) and Web Services: The Road to Enterprise Application Integration (EAI)", Sun Developer Network (SDN), Sun Microsystems (2005).
http://java.sun.com/developer/technicalArticles/WebServices/soa/

[Mingkhwan2006] A. Mingkhwan, P. Fergus, O. Abuelma'atti, M. Merabti, B. Askwith and M.B. Hanneghan: "Dynamic service composition in home appliance networks", ACM Multimedia Tools and Applications, Vol.29 , Issue 3, pp.257-284 (2006).

[Nakamura2004] Masahide Nakamura, Hiroshi Igaki, Haruaki Tamada and Kenichi Matsumoto: "Implementing Integrated Services of Networked Home Appliances Using Service Oriented Architecture", Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC04), pp.269-278 (2004).

[Nedos2006] Andronikos Nedos, Kulpreet Singh and Siobh'an Clarke: "Mobile Ad Hoc Services: Semantic Service Discovery in Mobile Ad Hoc Networks", Proceedings of the 4th International Conference on Service Oriented Computing (ICSOC06), LNCS 4294, Springer-Verlag, pp.90-103 (2006).

[Olsen2006] Greg Olsen: "From COM to Common - Component software's 10-year journey toward ubiquity", ACM Queue, Vol. 4 , Issue 5, pp.20-26 (2006).

[OMG2002] Object Management Group: "OMG IDL Syntax and Semantics (chapter 3 of CORBA v3.0)", OMG Document (2002).
http://www.omg.org/docs/formal/02-06-39.pdf

[Orso2000] Alessandro Orso, Mary Jean Harrold and David Rosenblum: "Component Metadata for Software Engineering Tasks", Proceedings of the 2nd International Workshop in Engineering Distributed Objects (EDO 2000), pp.126-140 (2000).

[Papazoglou2003] Mike P. Papazoglou and Dimitrios Georgakopoulos: "Service-Oriented Computing", Communications of the ACM, October 2003/Vol. 46, No. 10, pp.25-28 (2003).

[Pokraev2006] Stanislav Pokraev, Dick Quartel, Maarten W.A. Steen and Manfred Reichert: "Requirements and Method for Assessment of Service Interoperability", Proceedings of the 4th International Conference on Service Oriented Computing (ICSOC06), LNCS 4294, Springer-Verlag, pp.1-14, (2006).

[Siljee2005] Johanneke Siljee, Ivor Bosloper, Jos Nijhuis and Dieter Hammer: “DySOA: Making Service Systems Self-adaptive”, Proceedings of the 3rd International Conference on Service Oriented Computing (ICSOC05), LNCS 3826, Springer-Verlag, pp.255-268 (2005).

[SunEJB2006] Sun microsystems: “JSR 220 Enterprise JavaBeans 3.0”, Java Specification Requests, Java Community Process (2006).
http://jcp.org/en/jsr/detail?id=220

[SunJMS2002] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli and Kate Stout: “Java Message Service Specification Version: 1.1”, Sun Developer Network (SDN), Sun Microsystems (2002).
http://java.sun.com/products/jms/docs.html

[SunJNDI2003] Sun microsystems: “Java Naming and Directory Interface (JNDI)” (2003).
http://java.sun.com/products/jndi/

[SunJPA2006] Rahul Biswas and Ed Ort: “The Java Persistence API - A Simpler Programming Model for Entity Persistence”, Sun Developer Network (SDN), Sun Microsystems (2006).
http://java.sun.com/developer/technicalArticles/J2EE/jpa/

[Srivastava2006] Utkarsh Srivastava, Kamesh Munaga, Jennifer Widom and Rajeev Motwani: “Query Optimization over Web Services”, Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB’06), pp.355-366 (2006).

[Sybase2002] Sybase: “e-Biz Integrator Release Bulletin” (2002).
http://www.sybase.com/detail?id=1018601

[Thatte2001] Satish Thatte: “XLANG - Web Services for Business Process Design”, Microsoft Corporation (2001).
http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm

[Turner2003] Mark Turner, David Budgen and Pearl Brereton: “Turning Software into a Service”, IEEE Computer, Vol.36, No.10, pp.38-44 (2003).

[webMethods2006] webMethods: “Integration and B2B” (2006).
http://www.webmethods.com/Products/B2B

[Zim2005] Sasiporn Usanavasin, Takahiro Nakamori, Shingo Takada and Norihisa Doi: “A Multi-faceted Approach for Searching Web Applications”, 情報処理学会論文誌, Vol.46, No.5, pp.1256-1265 (2005).

[Zou2000] Ying Zou and Kostas Kontogiannis: “Web-Based Specification and Integration of Legacy Services”, Proceedings of the 2000 conference of the Centre for

Advanced Studies on Collaborative research, IBM Centre for Advanced Studies Conference (2000)

付 録A　ワークフロー情報からのコード自動生成

6 章で述べた異種分散コンポーネントを利用するアプリケーションの開発支援システムでは，ワークフローエディタ上で UML アクティビティ図を利用して描画したワークフロー情報から，WSFL や BPEL4WS で記載されたコードを自動的に生成する．本章では，アクティビティ図の各要素から，WSFL や BPEL4WS の要素へ変換する際の，それぞれの要素間の対応を示す．

A.1　アクティビティ図から WSFL へ変換する際の対応

A) アクション状態
アクション状態はワークフロー中の処理を示し，WSFL の `activity` 要素 に対応づける．また，各アクション状態に対応付けたコンポーネントについては，`activity` 要素 の子要素として，該当するコンポーネントの WSDL で定義されたインタフェースを指し示す．

B) レーン
各レーンは WSFL の `flowModel` 要素 に対応づけられる．そして内包するアクション状態は，`flowModel` 要素 に内包された `activity` 要素 として記述する．

C) 遷移
同一レーン内での遷移の場合，そのレーンに対応した `flowModel` 要素 に内包された `controlLink` 要素 および `dataLink` 要素 として記述する．異なるレーン間でのノード間の遷移の場合，遷移元と遷移先の対応を，`globalModel` 要素 に内包された `plugLink` 要素 として記述する．そして `flowModel` 要素 の内部において異なるレーンへの遷移が発生する箇所に，`export` 要素 を内包した `activity` 要素 を記述する．

D) 開始状態／終了状態
開始状態や終了状態も状態の一種であるため，`activity` 要素 として記述される．ただしこれらの状態はアクション状態とは異なり，実際の処理とは結びつけられる事が無いので，ワークフローの入出力のためのポイントとしか扱わない．

E) 同期バー
`flowModel` 要素 の内部に `controlLink` 要素 を記述することによって，同期を表現する．

F) 分岐
分岐先の状態ごとに controlLink 要素 を用意し，分岐元との対応を表現する．なお分岐条件については，ワークフローエディタでは指定を行わずに，開発者が記述を行う．そのために，分岐条件は controlLink 要素 の transitionCondition 属性 に，空の条件を出力する．

A.2　アクティビティ図から BPEL4WS へ変換する際の対応

A) アクション状態
BPEL4WS では，アクション状態は receive 要素，reply 要素 (receive 要素 と reply 要素 で，同期型呼出しに対応)，invoke 要素 (one-way での非同期型呼出し)，assign 要素 (変数への値の代入) の 4 種類に対応する．ワークフローエディタ上での開発者の記述により，このうちの適切なものに変換する．

B) レーン
各レーンを partnerLink 要素 に対応付ける．

C) 遷移
同一レーン内での遷移の場合，sequence 要素 に内包するように，順に記述する．そして値の対応は，assign 要素 によって記述する．異なるレーン間でのノード間の遷移の場合，sequence 要素 に内包されたアクション状態として，遷移先の対応を指定する．その際には，partnerLink 要素 に記述された，レーンの名前を用いる．

D) 開始状態／終了状態
具体的に対応付けられる BPEL の要素が存在しないため，対応づけを行わない．

E) 同期バー
同期する複数の処理を flow 要素 に内包する．

F) 分岐
switch 要素 で表す．分岐条件については WSFL の場合と同様に，空の条件を出力する．

付 録B　WSDL インターフェース記述の拡張

6 章で述べた異種分散コンポーネントを利用するアプリケーションの開発支援システムでは，WSDL の記述を拡張し，EJB や CORBA のコンポーネントのインターフェース情報を記述している．また 7 章で述べた異種分散コンポーネントの Plug and Play プラットフォームでは，コンポーネントのタイプを指定できるように WSDL の記述を拡張した．本章ではそれぞれの詳細を述べる．

B.1　EJB および CORBA に関するインタフェース情報の記述

6 章で述べた異種分散コンポーネントを利用するアプリケーションの開発支援システムでは，Proxy を Web Service として生成することによって，コンポーネント異種性の吸収を行うようにした．この実装においては，EJB や CORBA のコンポーネントのインタフェース情報を記述するために，Web Service のためのインターフェース記述言語である WSDL の記述を拡張した，

WSDL では，`service` タグで内包される要素に，実際に呼び出すコンポーネントに関する情報を記述する．この `service` タグで内包される部分に，それぞれの実装に依存した呼出しのための情報を記述できる．コンポーネントを呼び出すために必要な情報は，実装しているアーキテクチャにより異なるが，`service` タグで内包される部分に `port` タグを用いて記述する．そして `port` タグで内包される要素として，Web Service の コンポーネントのエンドポイントの URL を，`address` 要素の `location` 属性によって指定する．この `address` 要素の部分を拡張することによって，EJB や CORBA の呼出しのために必要な情報を記述できるようにした．

A) EJB に関するインタフェース情報の記述
EJB のコンポーネントの呼出しの際には，WSDL で標準的に定義されているメソッドのシグニチャに関する情報の他に，コンポーネントの Home/Remote インタフェースのクラス名と，コンポーネントの名前を登録してある JNDI サーバの URL，JNDI サーバからコンポーネントを検索するための名前が必要になる．これらを `service` タグで内包される `address` 要素に記述できるように拡張した．なお，この拡張部分を明示するために，ここでは `wsdlejb:address` と表記する．

```
<wsdlejb:address location="JNDI Server URL"
                 jndiName="JNDI lookup name"
                 homeClassName="EJB Home class name"
                 remoteClassName="EJB Remote class name"/>
```

B) CORBA に関するインタフェース情報の記述
CORBA のコンポーネントの呼出しの際には，WSDL で標準的に定義されているメソッドのシグニチャに関する情報の他に，コンポーネントのクラス名やパッケージ名と，CORBA ネーミングサービスを提供するサーバの URL と，ネーミングサービスからコンポーネントを検索するための名前が必要になる．これらを `service` タグで内包される `address` 要素に記述できるように拡張した．なお，この拡張部分を明示するために，ここでは `wsdlcorba:address` と表記する．

```
<wsdlcorba:address nameServiceRef="Naming Service URL"
                   ncName="Naming Service lookup name"
                   className="Server Object class name"
                   packageName="Server Object package name"/>
```

B.2　コンポーネント探索に必要な情報の記述

7 章で述べた異種分散コンポーネントの Plug and Play プラットフォームでは，コンポーネントを連係させるサーバは複数のコンポーネントを探索して呼び出すことにより，連係させる．この際にコンポーネントを連係させるプログラムを BPEL4WS で記述する．BPEL4WS では，WSDL を利用することによって実際に呼び出すコンポーネントの詳細を記述する．この際にコンポーネントの詳細として，どのようなコンポーネントを探索して利用するかを記述できる必要がある．そこで WSDL の記述を拡張し，コンポーネントのタイプを示す文字列を指定できるようにした．

具体的には，B.1 節で行った拡張と同様に，`port` タグで内包される要素に，実際に呼び出すコンポーネントに関する情報を記述できるようにした．B.1 節では `address` 要素を拡張したが，コンポーネントの探索ではエンドポイント URL を指定しないので，この `address` 要素の代わりにコンポーネントのタイプを指定する要素を記述するタグを用意した．なお，この拡張部分を明示するために，ここでは `wsdlcdp:compdisc` と表記する．

```
<wsdlcdp:compdisc type="Component type string"
                  selPolicy="Selection policy class name"/>
```

コンポーネントのタイプを指定する文字列を，`type` 属性に記述する．また，コンポーネント探索の結果，同一の内容を提供する複数のコンポーネントが発見された場合には，メディエータ部がどれを利用するのかを選択する必要がある．`selPolicy` には，この際の選択の基準を提供するモジュールのクラス名を記述する．

付 録C　コンポーネント探索プロトコル

7 章で述べた異種分散コンポーネントの Plug and Play プラットフォームでは，コンポーネントを探索し発見するために独自のプロトコル (Component Discovery Protocol : CDP) を用意した．本節では，そのために定義したメッセージの詳細を述べる．

C.1　概要

コンポーネントを探索するために，次のメッセージを定義した．

(1) コンポーネント要求メッセージ／コンポーネント応答メッセージ
コンポーネントを利用するクライアントや，コンポーネントを連係させる BPEL サーバが，同一ネットワークに接続されているコンピュータ上で動作するコンポーネントを発見するためのメッセージ．クライアントや BPEL サーバはコンポーネント要求メッセージをマルチキャストで送信し，該当するコンポーネントを提供するサーバが，ユニキャストでコンポーネント応答メッセージを送信する．

(2) パラメータ要求メッセージ／パラメータ応答メッセージ
コンポーネントを利用するクライアントや，コンポーネントを連係させる BPEL サーバが，発見したコンポーネントに対し，その呼出し方法を問い合わせるためのメッセージ．クライアントや BPEL サーバは，コンポーネント応答メッセージの送信元に，パラメータ要求メッセージをユニキャストで送信する．該当するサーバは，パラメータ応答メッセージによって，ユニキャストで応答する．

(3) エラー応答メッセージ
コンポーネントを提供するサーバや，コンポーネントを連係させる BPEL サーバが，コンポーネント要求メッセージやパラメータ要求メッセージを受信して処理を行った結果，何らかのエラー状態になったことを通知するためのメッセージ．要求メッセージの送信元にユニキャストで送信する．

(4) コンポーネント広告メッセージ
コンポーネントを提供するサーバや，コンポーネントを連係させる BPEL サーバが，コンポーネントの存在を定期的にマルチキャストで通知するためのメッセージ．

次節で，これらのメッセージの全体の構造の概要を示し，その後で個々のメッセージの詳細を述べる．

C.2　メッセージの全体の構造と CDP ヘッダ部

図 C.1 に，送受信されるメッセージの構造を示す．

一回に送信されるメッセージに，複数のコンポーネントに対する同一の種類の要求メッセージや応答メッセージを含めることができるように，CDP のメッセージを設計した．そして CDP ヘッダ部にメッセージ共通の内容を記述し，その後に複数の要求メッセージや応答メッセージを記述する．

CDP ヘッダ部の構造を，図 C.2 に示す．なお，以降で示す図では，それぞれの一行分で 4 バイト (32 ビット) を表す．

CDP ヘッダ部に含まれる情報は，次の通りである．なお，それぞれの項目の後に，メッセージ中に含まれる長さを記載する．

(1) Version (8 ビット)
CDP のプロトコルのバージョン番号．今後改版する際の互換性のために用意している．

(2) フラグ (8 ビット)
CDP メッセージ全体に共通の属性情報を表すフラグ．現段階の実装では，次の 3 つを用意しており，後半部 5 ビットは使用していない．

(a) R (Request MCAST): マルチキャストやブロードキャストで送信していることを表す．

(b) E (Error): 要求メッセージが不正なフォーマットであることを返答するためのメッセージであることを示す．

(c) N (Not Found): 要求によって探索を行った結果が見つからなかったことを明示的に返していることを示す．

(3) XID (Exchange Identification) (16 ビット)
要求メッセージと応答メッセージの対応関係を識別するために，付与する番号．

そして CDP ヘッダ部の後に，C.1 節に示した要求メッセージや応答メッセージの内容を記述する．メッセージの詳細については C.3 節で述べるが，最初の 1 バイトの部分 (Function-ID) に，それぞれのメッセージの種類を識別するための番号を記述する．またそれぞれのメッセージには，複数のオプションを含めることができ，追加的な情報を記述できる．それぞれのメッセージの最後には，最後であることを示すための Null Option を含む．各オプションの詳細については，C.4 節に示す．

C.3　メッセージ詳細

本節では，送受信されるメッセージの詳細を示す．

なお，C.2 節で述べたように，本節で述べる各メッセージの最初の 1 バイトの部分には，Function-ID と呼ぶ識別番号を記述する．それぞれのメッセージに，次のように付加した．

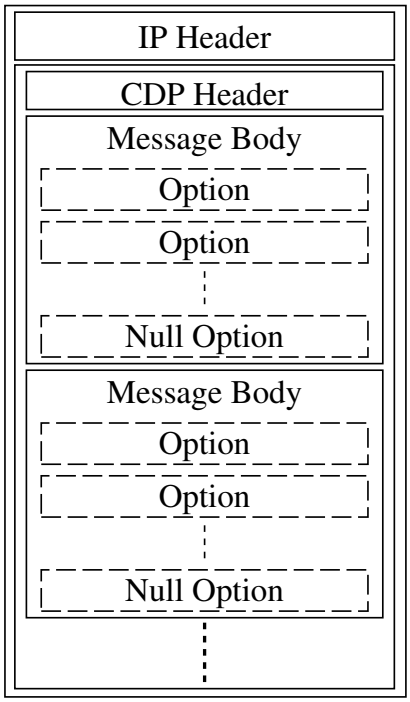

図 C.1 CDP メッセージ構造

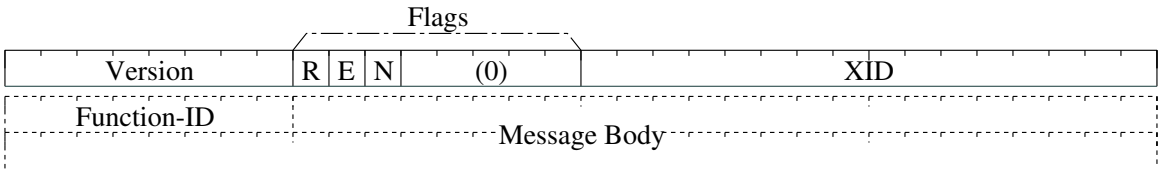

図 C.2 CDP ヘッダ部

(1) コンポーネント要求メッセージ: 0x01

(2) コンポーネント応答メッセージ: 0x02

(3) パラメータ要求メッセージ: 0x11

(4) パラメータ応答メッセージ: 0x12

(5) エラー応答メッセージ: 0xD2

(6) コンポーネント広告メッセージ: 0x03

C.3.1　コンポーネント要求メッセージ

コンポーネントを利用するクライアントや，コンポーネントを連係させる BPEL サーバは，コンポーネントを探索するために，コンポーネント要求メッセージをマルチキャストで送信する．そのため，CDP ヘッダ部のフラグのうち，R フラグをセットする．なお，現段階の実装では意図的にユニキャストで要求を送信することもでき，その場合には R フラグをセットしない．また，XID にはユニークな ID を生成し，記述する．

コンポーネント要求メッセージの構造を，図 C.3 に示す．それぞれのフィールドが示す情報は，次の通りである．

(1) Function-ID (8 ビット)
コンポーネント要求メッセージであることを示す (0x01).

(2) フラグ (8 ビット)
コンポーネント要求メッセージの受信者に対して，追加的な動作を要求するためのフラグを記述する．現段階の実装では，次の 1 つを用意しており，後半部 7 ビットは使用していない．

- P (with Parameter Request): パラメータ要求を同時に行うことを示す.

(3) コンポーネント技術や実装に関する情報
利用するコンポーネント技術や，その実装を限定する必要がある場合に，記述することができる．

(a) Platform Type (8 ビット): コンポーネント技術の種類 (EJB，CORBA，Web Service などを示す)

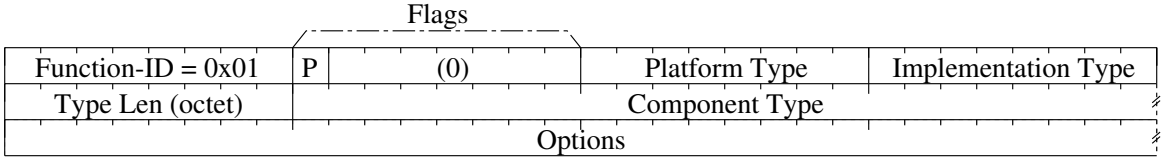

図 C.3 コンポーネント要求メッセージ

(b) Implementation Type (8 ビット): コンポーネントサーバ実装の種類

(4) コンポーネントのタイプを示すための情報
要求するコンポーネントのタイプを示すための文字列を記述する．

(a) Type Len (8 ビット): Component Type フィールドの長さ (バイト単位)

(b) Component Type (可変長): コンポーネントのタイプを指定する文字列

(5) Options
オプションを記述するための部分．現段階の実装ではオプションを用意していない．そのため，メッセージの最後であることを示すための Null Option のみを付加する．

C.1 節で述べたように，コンポーネントの呼出し方法に関する情報は，パラメータ応答メッセージによって取得する．しかしメッセージのやり取りの回数を減らすために，利用場面に応じて，それらの内容をコンポーネント応答メッセージに含めて取得できるようにしている．その場合は，コンポーネント要求メッセージの P フラグをセットして送信する．

また，あるタイプのコンポーネントに対するコンポーネント要求として送信されるメッセージに，他のタイプのコンポーネントに対するコンポーネント要求のメッセージも含むことができる．その場合は C.2 節に述べたように，一つ目のコンポーネント要求メッセージの終端を示す Null Option の後に，次のコンポーネント要求メッセージを記述する．

C.3.2　コンポーネント応答メッセージ

コンポーネントを提供するサーバや，コンポーネントを連係させる BPEL サーバは，コンポーネント要求メッセージの送信元にコンポーネントの存在を通知するために，コンポーネント応答メッセージをユニキャストで送信する．コンポーネント要求メッセージを受け取ったサーバは，コンポーネント要求メッセージに含まれるコンポーネントのタイプに一致するコンポーネントを提供する場合には，コンポーネント応答メッセージを返す．コンポーネントのタイプが指定されなかった場合には，すべてのコンポーネントに関する情報を返す．この際に，コンポーネント要求メッセージに，コンポーネントアーキテクチャや実装が記述されている場合には，それに該当する実装を利用しするコンポーネントを提供している場合にのみ応答メッセージを返す．

コンポーネント応答メッセージでは，CDP ヘッダ部のフラグはセットしない．また，XID には，受け取ったコンポーネント要求メッセージの XID をコピーする．これによりコンポーネント応答メッセージの受信者は，どの要求に対する応答かを識別できる．

コンポーネント応答メッセージの構造を，図 C.4 に示す．それぞれのフィールドが示す情報は，次の通りである．

(1) Function-ID (8 ビット)
コンポーネント応答メッセージであることを示す (0x02)．

(2) フラグ (8 ビット)
コンポーネント応答メッセージの受信者に対して，コンポーネントの動作に関する

追加的な情報を通知するためのフラグを記述する．現段階の実装では，次の 2 つを用意しており，後半部 6 ビットは使用していない．

(a) O (Other Component): 他のコンポーネントを呼び出して利用していることを示す．

(b) S (Suspended): 何らかの理由で，コンポーネントの動作が停止していることを示す．

(3) コンポーネント技術や実装に関する情報
コンポーネントが動作するために利用しているコンポーネント技術や，その実装を記述する．

(a) Platform Type (8 ビット): コンポーネント技術の種類 (EJB，CORBA，Web Service などを示す)

(b) Implementation Type (8 ビット): コンポーネントサーバ実装の種類

(4) Lifetime (16 ビット)
コンポーネントの提供者によって設定された生存時間

(5) Options
オプションを記述するための部分．詳細は後述する．

オプションについては，次のものを含む．各オプションの詳細については，C.4 節に示す．

A) Component ID Option: コンポーネント応答メッセージの受信者がコンポーネントを識別するための識別子．コンポーネント応答メッセージは，Component ID Option を必ず含む．

B) IP Address Option: コンポーネント応答メッセージの送信者と，コンポーネントを提供するサーバが異なる場合に付加する．そして，コンポーネントを実際に提供するサーバの IP アドレスを記述する．このオプションを含めることにより，コンポーネントを提供するサーバの代わりに，コンポーネントに関する情報を応答することができる．複数のコンポーネントサーバに関する情報を，一台のサーバに予め登録しておくことにより，簡易的なディレクトリサービスとして利用することができる．

C) Type Option: タイプの指定されていないコンポーネント要求メッセージに対する応答を返す場合に付加する．そして Type Option によって，どのタイプのコンポーネントに対する情報なのかを示す．また，一つのメッセージに複数のコンポーネント

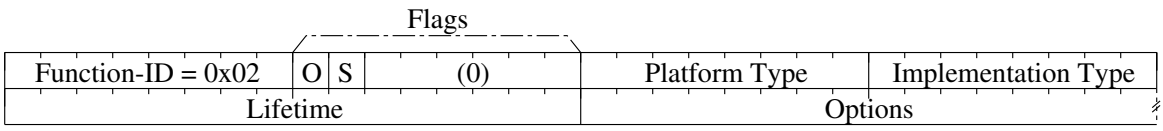

図 C.4 コンポーネント応答メッセージ

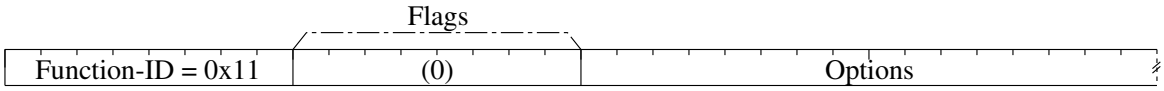

図 C.5 パラメータ要求メッセージ

要求メッセージが含まれていた場合も，同様に Type Option によってタイプを明示する必要がある．

D) Parameter Option: P フラグのセットされたコンポーネント要求メッセージに対する応答の場合は，パラメータ応答メッセージで送信する呼出しに関する情報を，Parameter Option に記述して付加する．

これらのオプションをコンポーネント応答メッセージの後に付加し，オプション列の最後に Null Option を記述する．

また，コンポーネント要求メッセージに該当する複数のコンポーネントを提供する場合や，タイプの指定されていないコンポーネント要求メッセージを受信した場合には，複数のコンポーネントに関する情報を返す必要がある．この場合はそれぞれのコンポーネントに関するコンポーネント応答メッセージを作成し，C.2 節に述べたような形式でそれらを繋げて，一つのメッセージとして応答する．

なお，コンポーネント要求メッセージに該当するコンポーネントを提供しないノードは，コンポーネント応答メッセージを送信しない．ただし C.3.1 節で述べたように，コンポーネント要求メッセージが意図的にユニキャストで送信されていた場合には，明示的に応答メッセージを送信する．その場合は，CDP ヘッダ部に N フラグをセットしたエラー応答メッセージを送信する．この詳細については，C.3.5 節で述べる．

C.3.3　パラメータ要求メッセージ

コンポーネントを利用するクライアントや，コンポーネントを連係させる BPEL サーバは，発見したコンポーネントに対し，その呼出し方法を問い合わせるために，パラメータ要求メッセージを送信する．このメッセージは，コンポーネント応答メッセージの送信元に対して，ユニキャストで送信する．

パラメータ要求メッセージでは，CDP ヘッダ部にはフラグをセットしない．また，XID にはユニークな ID を生成し，記述する．

パラメータ要求メッセージの構造を，図 C.5 に示す．それぞれのフィールドが示す情報は，次の通りである．

(1) Function-ID (8 ビット)
パラメータ要求メッセージであることを示す (0x11).

(2) フラグ (8 ビット)
今後改版するために確保したフラグ部分．現段階の実装では使用していない．

(3) Options
　オプションを記述するための部分．詳細は後述する．

オプションについては，次のものを含む．各オプションの詳細については，C.4 節に示す．

A) Component ID Option: 要求対象のコンポーネントの識別子．コンポーネント応答メッセージに含まれていたものと同一のものを指定する．パラメータ要求メッセージは，必ず Component ID Option を含む．

B) IP Address Option: 要求対象のコンポーネントを提供するサーバの IP アドレス．パラメータ要求メッセージはコンポーネント応答メッセージの送信元に対して送信するが，コンポーネント応答メッセージに IP Address Option が含まれていた場合は，コンポーネント応答メッセージの送信元とコンポーネントを提供するサーバが異なる．そのため，この場合はパラメータ要求オプションに IP Address Option を付加し，対象コンポーネントを提供するサーバを指定する．

これらのオプションをパラメータ要求メッセージの後に付加し，オプション列の最後に Null Option を記述する．

コンポーネント要求メッセージと同様に，複数のパラメータ要求メッセージを一つのメッセージに含むことができる．その場合は C.2 節に述べたように，一つ目のパラメータ要求メッセージの終端を示す Null Option の後に，次のパラメータ要求メッセージを記述する．

C.3.4　パラメータ応答メッセージ

コンポーネントを提供するサーバや，コンポーネントを連係させる BPEL サーバは，パラメータ要求メッセージの送信元に，パラメータ応答メッセージをユニキャストで送信する．これによって，コンポーネントの呼出しに関する情報を通知する．パラメータ要求メッセージを受け取ったサーバは，Component ID Option で指定される識別子のコンポーネントを提供する場合には，パラメータ応答メッセージを送信元に返す．パラメータ要求メッセージに IP Address Option が含まれている場合には，コンポーネントの IP アドレスについても比較を行い，該当するコンポーネントについての呼出しに関する情報を送信する．

パラメータ応答メッセージでは，CDP ヘッダ部のフラグはセットしない．また，XID には，受け取ったパラメータ要求メッセージの XID をコピーする．これによりパラメータ応答メッセージの受信者は，どの要求に対する応答かを識別できる．

パラメータ応答メッセージの構造を，図 C.6 に示す．それぞれのフィールドが示す情報は，次の通りである．

(1) Function-ID (8 ビット)
　パラメータ応答メッセージであることを示す (0x12).

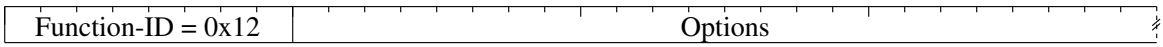

図 C.6 パラメータ応答メッセージ

(2) Options
　オプションを記述するための部分．詳細は後述する．

オプションについては，次のものを含む．各オプションの詳細については，C.4 節に示す．

A) Parameter Option: それぞれのコンポーネント技術やサーバ実装に依存した呼出し形式に関する情報を記述する．

B) Component ID Option: この応答メッセージにより呼出し形式が示されるコンポーネントの，コンポーネント ID を記述する．複数のパラメータ要求メッセージが，一つの メッセージに含まれていた場合に，付加して応答する．パラメータ応答メッセージの受信者は，どのコンポーネントに関する呼出し形式に関する情報であるかを，Component ID Option により判断する．

これらのオプションをパラメータ応答メッセージの後に付加し，オプション列の最後に Null Option を記述する．

受信したメッセージに複数のパラメータ要求メッセージを含む場合には，複数のコンポーネントのパラメータに関する情報を応答する．そしてそれぞれのコンポーネントに関するパラメータ応答メッセージを作成し，C.2 節に述べたような形式でそれらを繋げて，一つのメッセージとして応答する．その際には，それぞれのパラメータ応答メッセージに Component ID Option を付加する．これにより，それぞれのパラメータ応答メッセージを受信側で識別できるようにしている．

なお，パラメータ要求メッセージによって指定された識別子のコンポーネントを提供しない場合には，コンポーネントを提供していないことを通知する．その際には C.3.2 節の場合と同様に，パラメータ要求メッセージの送信元に対して，CDP ヘッダ部に N フラグをセットしたエラー応答メッセージを送信する．この詳細については，C.3.5 節で述べる．

C.3.5　エラー応答メッセージ

コンポーネントを提供するサーバや，コンポーネントを連係させる BPEL サーバは，コンポーネント要求メッセージやパラメータ要求メッセージ受信時に行った処理にエラーが発生した場合，エラー応答メッセージをユニキャストで送信する．

エラー応答メッセージでは，CDP ヘッダ部のフラグは，発生したエラーに応じてセットする．これについては，発生したエラーの種類を記述するための ErrorCode と一緒に後述する．また，XID には，受け取った要求メッセージの XID をコピーする．これによりエラー応答メッセージの受信者は，どの要求に対する応答かを識別することができる．

エラー応答メッセージの構造を，図 C.7 に示す．それぞれのフィールドが示す情報は，次の通りである．

(1) Function-ID (8 ビット)
　エラー応答メッセージであることを示す (0xD2).

(2) Error Code (16 ビット)
　エラーの詳細を記述するための部分．現在の実装では，次のいずれかを指定する．

(a) 0x0000 (OK): 処理が正常に終了したことを示す.

(b) 0x0001 (Not Found): 要求メッセージに該当するコンポーネントを見つけられなかったことを示す.

(c) 0x0002 (Parse Error): 要求メッセージのフォーマットを解釈できなかったことを示す.

(d) 0x0003 (Not Implemented): 要求メッセージにしたがってサーバが応答を返すために必要な機能が，実装されていないことを示す.

(e) 0x0004 (Network Timeout): 応答メッセージ送出や，その他のネットワークを利用する操作をした際に，何らかの原因によってタイムアウトをしたことを示す.

(f) 0x0005 (Memory Allocation Failed): 要求メッセージにしたがってサーバが処理を行う過程で，メモリ確保に失敗したことを示す.

(g) 0x0006 (Malformed Parameter): 要求メッセージに記述されたパラメータの値が，想定していないものであったことを示す.

(h) 0x0007 (Network Error): メッセージ送受信の処理の過程で，ネットワーク上になんらかの障害が発生したことを示す.

(i) 0x0008 (Internal System Error): (b)〜(h) に該当しない障害が発生したことを示す.

(a) については，処理が正常に行われたことを特に明示的に応答する必要がある場合のために用意している．このため CDP ヘッダのフラグはセットしない．なお，現在の実装では利用していない.

(b) については，C.3.2 節や C.3.4 節で述べたように，該当するコンポーネントが見つけられなかったことを応答する必要がある場合に，CDP ヘッダの N フラグをセットして送信する.

(c)〜(i) については，それぞれ対応するエラーが発生した場合に，CDP ヘッダの E フラグをセットして送信する.

(3) Options
オプションを記述するための部分．詳細は後述する.

受信したメッセージに要求メッセージを一つしか含まない場合は，エラー応答メッセージには Null Option 以外のオプションを付加しない．複数の要求メッセージを含んでいた場合は，それぞれの要求メッセージごとにエラー応答メッセージを作成する．そして，C.2 節に述べたような形式でそれらを繋げて，一つのメッセージとして応答する．その際にど

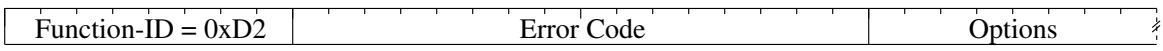

図 C.7 エラー応答メッセージ

の要求メッセージの処理に関連するエラーであるのかを識別するために，次のオプションを記述する．各オプションの詳細については，C.4 節に示す．

A) Component ID Option: コンポーネントを識別するための識別子．要求メッセージで指定されているものをそれをそのままコピーして付加する．

B) IP Address Option: コンポーネントサーバのアドレス．要求メッセージで指定されているものをそれをそのままコピーして付加する．

C) Type Option: コンポーネントのタイプ．異なったのタイプのコンポーネント要求メッセージが複数含まれていた場合に，それぞれのエラー応答メッセージに付加する．

これらのオプションをエラー応答メッセージの後に付加し，オプション列の最後に Null Option を記述する．

C.3.6　コンポーネント広告メッセージ

コンポーネントを提供するサーバや，コンポーネントを連係させる BPEL サーバは，コンポーネントの存在を通知するために，コンポーネント広告メッセージをマルチキャストで定期的に送信することができる．

コンポーネント広告メッセージでは，CDP ヘッダ部のフラグのうち，R フラグをセットする．また，XID にはユニークな ID を生成し，記述する．

コンポーネント広告メッセージの構造を，図 C.8 に示す．ほとんどのフィールドは，C.3.2 節で述べたコンポーネント応答メッセージと同様である．それぞれのフィールドが示す情報は，次の通りである．

(1) Function-ID (8 ビット)
コンポーネント広告メッセージであることを示す (0x03).

(2) フラグ (8 ビット)
コンポーネント応答メッセージの受信者に対して，コンポーネントの動作に関する追加的な情報を通知するためのフラグを記述する．現段階の実装では，次の 1 つを用意しており，後半部 7 ビットは使用していない．

- O (Other Component): 他のコンポーネントを呼び出して利用していることを示す．

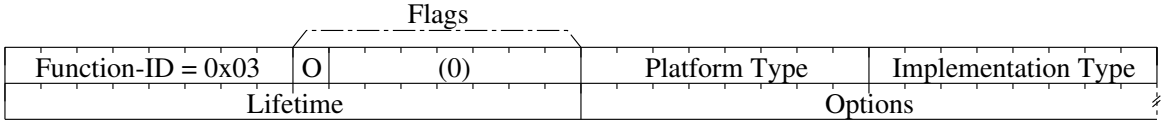

図 C.8 コンポーネント広告メッセージ

なお，C.3.2 節で述べたコンポーネント応答メッセージには，停止状態を示す S フラグを用意した．しかしコンポーネントの処理を停止してる場合には，サーバはコンポーネント広告を送信しないため，S フラグは用意しない．

(3) コンポーネント技術や実装に関する情報
コンポーネントが動作するために利用しているコンポーネント技術や，その実装を記述する．

(a) Platform Type (8 ビット): コンポーネント技術の種類 (EJB，CORBA，Web Service などを示す)

(b) Implementation Type (8 ビット): コンポーネントサーバ実装の種類

(4) Lifetime (16 ビット)
コンポーネントの提供者によって設定された生存時間

(5) Options
オプションを記述するための部分．詳細は後述する．

オプションについては，次のものを含む．各オプションの詳細については，C.4 節に示す．

A) Component ID Option: コンポーネント広告メッセージの受信者がコンポーネントを識別するための識別子．コンポーネント広告メッセージは，Component ID Option を必ず含む．

B) IP Address Option: コンポーネント広告メッセージの送信者と，実際にコンポーネントを提供するサーバが異なる場合に付加する．このオプションを含めることにより，コンポーネントを提供するサーバの代わりに，コンポーネントに関する情報を広告することができる．

C) Type Option: どのタイプのコンポーネントに対する情報なのかを示す．C.3.2 節で述べたコンポーネント応答メッセージと異なり，コンポーネント広告メッセージでは，必ず Type Option を付加しタイプを明示する．

これらのオプションをコンポーネント広告メッセージの後に付加し，オプション列の最後に Null Option を記述する．

また，一つのメッセージで複数のコンポーネントに関する情報を広告する場合には，それぞれのコンポーネントに関するコンポーネント広告メッセージを作成し，C.2 節に述べたような形式でそれらを繋げて，一つのメッセージとして送信する．

C.4　メッセージオプション詳細

C.3 節で述べたように，各メッセージには種類に応じてオプションを付加できる．本節では，その詳細を述べる．なおそれぞれのオプションの先頭の 8 ビットは，オプションの種類を表す．この識別番号を，Option Type と呼ぶ．それぞれのオプションに，次のように割り当てた．

図 C.9 Null Option

(1) Null Option: 0x00

(2) Component ID Option: 0x01

(3) IP Address Option: 0x02

(4) Type Option: 0x81

(5) Parameter Option: 0x31

C.4.1 Null Option

各メッセージに付加するオプション列の終わりを示すためのオプションである．メッセージにオプションを付加しない場合でも，Null Option を付加する必要がある．

Null Option の構造を，図 C.9 に示す．これは，Option Type 以外のフィールドを持たない．

C.4.2 Component ID Option

サーバ内でのコンポーネントの識別子を記述するためのオプションである．コンポーネントの情報を送受信するサーバやクライアントが，どのコンポーネントに関する情報であるのかを識別するために利用する．

Component ID Option の構造を，図 C.10 に示す．

(1) Option Type (8 ビット)
Component ID Option であることを示す (0x01)．

(2) Sub Type (8 ビット)
Component ID がどのような形式で作成されているかを示す．現段階の実装では，次の値のみが用意されている．

- UUID Based Component ID: 0x01

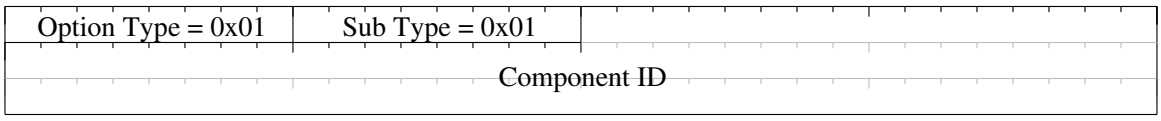

図 C.10 Component ID Option

この値については後述する．

(3) Component ID (80 ビット)
コンポーネントの識別子．このフィールドは，Sub Type の値により，変化する．図 C.10 では，Sub Type に 0x01 が指定された場合の，Component ID Option の構造を表している．これについては後述する．

コンポーネントの識別子の割り当てには，色々な形式を選択できる．そのため，Sub Type フィールドに，コンポーネント識別子の割り当て形式の種類を指定する．そして，その後に続くフィールドは，割り当て形式に合わせて変更することができる．

図 C.10 は，UUID (Universally Unique IDentifier) [Leach2005] を元に識別子を生成した場合の Component ID Option である．UUID では，128 ビットの URN (Uniform Resource Name) [Berners-Lee1998] を生成する際に，物理ネットワークデバイスの MAC (Media Access Control) アドレスを利用する．MAC アドレスを 48 ビットのフィールドとして記述し，残りの 80 ビットのフィールドをノード内での識別のために利用している．この方法を利用し，UUID 識別子から MAC アドレスを取り除いた 80 ビットの識別子を，コンポーネントの識別子として利用している．

Component ID Option では，このような形式の識別子を記述する際に，Sub Type フィールドに 0x01 を指定する．これによって，その後に続く 80 ビットが，コンポーネントの識別子であることを示している．現段階の実装では，コンポーネントの識別子として，この形式のみを用意している．将来的に他の識別子の記述方法を利用した場合には，図 C.10 とは Sub Type フィールドの後のフィールドの構成が異なる．

C.4.3　IP Address Option

コンポーネントサーバの IP アドレスを記述するためのオプションである．プロトコルファミリとしては，IPv4 と IPv6 を想定し，それぞれにメッセージを用意している．IPv4 の場合の IP Address Option の構造を図 C.11 に，IPv6 の場合の IP Address Option の構造を図 C.12 に示す．

(1) Option Type (8 ビット)
IP Address Option であることを示す (0x02).

(2) Protocol Family (8 ビット)
プロトコルファミリを記述する．次のいずれかの値になる．

(a) IPv4: 0x04

(b) IPv6: 0x06

(3) Component ID (32 ビットもしくは 128 ビット)
IP アドレスを記述する．Protocol Family が 0x04 の場合は，32 ビットの IPv4 アドレスを，Protocol Family が 0x06 の場合は，128 ビットの IPv6 アドレスを記述する．

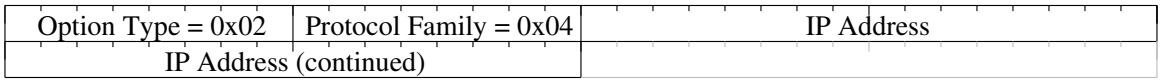

図 C.11 IP Address Option (IPv4)

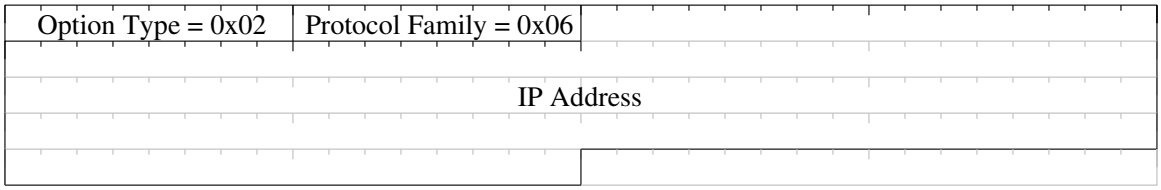

図 C.12 IP Address Option (IPv6)

C.4.4　Type Option

コンポーネントのタイプの文字列を記述するためのオプションである．メッセージの構造を，図 C.13 に示す．

(1) Option Type (8 ビット)
Type Option であることを示す (0x81)．

(2) TypeLen (8 ビット)
Component Type フィールドの長さ (バイト単位)．

(3) Component Type (可変長)
コンポーネントのタイプを指定する文字列．

C.4.5　Parameter Option

コンポーネントの呼出しの方法を記述するためのオプションである．メッセージの構造を，図 C.14 に示す．

(1) Option Type (8 ビット)
Parameter Option であることを示す (0x91)．

(2) Sub Option (可変長)
呼出しの方法を記述する．

図 C.13 Type Option

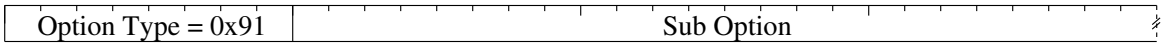

図 C.14 Parameter Option

Sub Option には，コンポーネントの実装に応じて，コンポーネントの呼出しに必要な情報の詳細を記述する．Sub Option の最初の 1 バイトの部分には，Sub Type と呼ぶ識別番号を記述する．それぞれのメッセージに，次のように付加した．

(1) WS HTTP Parameters Sub Option: 0x81

(2) CORBA Parameters Sub Option: 0xA1

(3) EJB Parameters Sub Option: 0x91

なお同一のコンポーネントに対して複数の呼出しの方法がある場合には，パラメータ応答メッセージに，必要なだけ Parameter Option を付加する．
本節の残りは，これらの Sub Option の詳細を述べる．

(1) **WS HTTP Parameters Sub Option**
Web Service を HTTP 経由で呼び出す際に必要となる情報を記述する．メッセージの構造を，図 C.15 に示す．

(a) Sub Type (8 ビット)
WS HTTP Parameters Sub Option であることを示す (0x81).

(b) フラグ (8 ビット)
今後改版するために確保したフラグ部分．現段階の実装では使用していない．

(c) Platform ID (16 ビット)
コンポーネントを実行しているサーバの実装に関する情報を記述する．

(d) コンポーネントを提供するサーバに関する情報

i. Hostname Len (16 ビット)
Hostname フィールドの長さ (バイト単位)．Hostname フィールドを省略する場合は，0 を指定する．

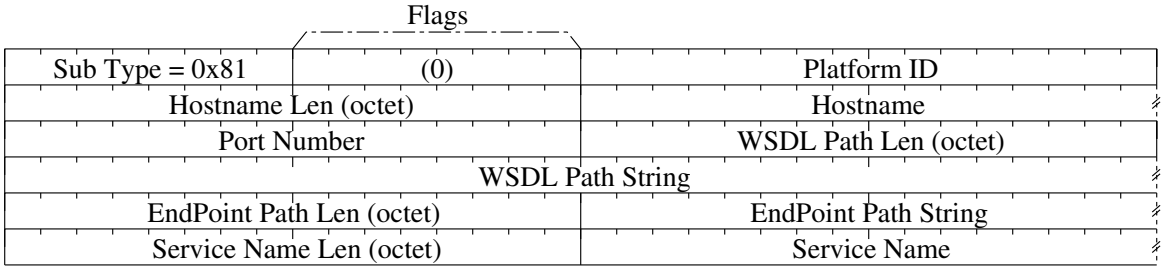

図 C.15 WS HTTP Parameters Sub Option

ii. Hostname (可変長)
コンポーネントが動作しているサーバのホスト名を指定する文字列．省略した場合は，パラメータ応答メッセージの送信元の IP アドレスの情報を利用する．

iii. Port Num (16 ビット)
コンポーネントにアクセスするためのポート番号．

(e) WSDL により記述されるインターフェース情報を取得するための情報

i. WSDL Path Len (16 ビット)
WSDL Path String フィールドの長さ (バイト単位)．

ii. WSDL Path String (可変長)
WSDL により記述されるインタフェース情報を取得するための URL のうち，サーバ内でのパスの部分の文字列．

(f) コンポーネントの指定に関する情報

i. EndPoint Path Len (16 ビット)
EndPoint Path String フィールドの長さ (バイト単位)．

ii. EndPoint Path String (可変長)
コンポーネントに実際にアクセスするための URL のうち，サーバ内でのパスの部分の文字列．

iii. Service Name Len (16 ビット)
Service Name フィールドの長さ (バイト単位)．

iv. Service Name (可変長)
コンポーネントにアクセスするために必要な，サービス名．

(2) **CORBA Parameters Sub Option**
CORBA を呼び出す際に必要となる情報を記述する．メッセージの構造を，図 C.16 に示す．

(a) Sub Type (8 ビット)
CORBA Parameters Sub Option であることを示す (0xA1)．

(b) フラグ (8 ビット)
今後改版するために確保したフラグ部分．現段階の実装では使用していない．

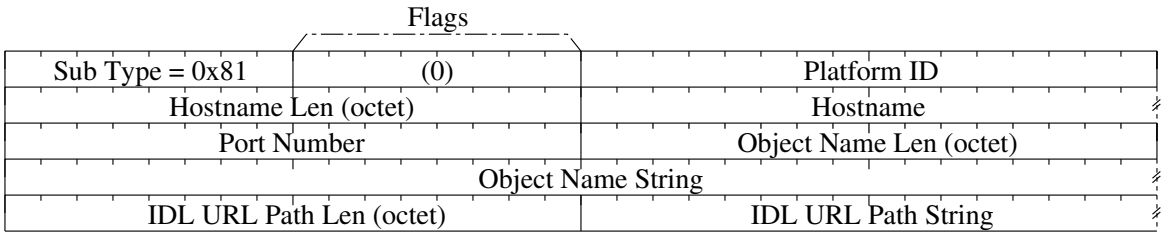

図 C.16 CORBA Parameters Sub Option

(c) Platform ID (16 ビット)
コンポーネントを実行しているサーバの実装に関する情報を記述する．

(d) コンポーネントを提供するサーバに関する情報

i. Hostname Len (16 ビット)
Hostname フィールドの長さ (バイト単位)．Hostname フィールドを省略する場合は，0 を指定する．

ii. Hostname (可変長)
コンポーネントが動作しているサーバのホスト名を指定する文字列．省略した場合は，パラメータ応答メッセージの送信元の IP アドレスの情報を利用する．

iii. Port Num (16 ビット)
コンポーネントにアクセスするためのポート番号．

(e) コンポーネントの指定に関する情報

i. Object Name Len (16 ビット)
Object Name フィールドの長さ (バイト単位)．

ii. Object Name (可変長)
コンポーネントにアクセスするために必要な，オブジェクト名．

(f) IDL により記述されるインターフェース情報を取得するための情報

i. IDL URL Path Len (16 ビット)
IDL URL Path String フィールドの長さ (バイト単位)．

ii. IDL URL Path String (可変長)
IDL により記述されるインタフェース情報を取得するための URL．

(3) **EJB Parameters Sub Option**
EJB を呼び出す際に必要となる情報を記述する．メッセージの構造を，図 C.17 に示す．

(a) Sub Type (8 ビット)
EJB Parameters Sub Option であることを示す (0x91)．

(b) フラグ (8 ビット)
今後改版するために確保したフラグ部分．現段階の実装では使用していない．

(c) Platform ID (16 ビット)
コンポーネントを実行しているサーバの実装に関する情報を記述する．

(d) コンポーネントを提供するサーバに関する情報

i. Protocol Len (8 ビット)
Protocol String フィールドの長さ (バイト単位)．

ii. Protocol String (可変長)
呼出しの際に利用するプロトコル．

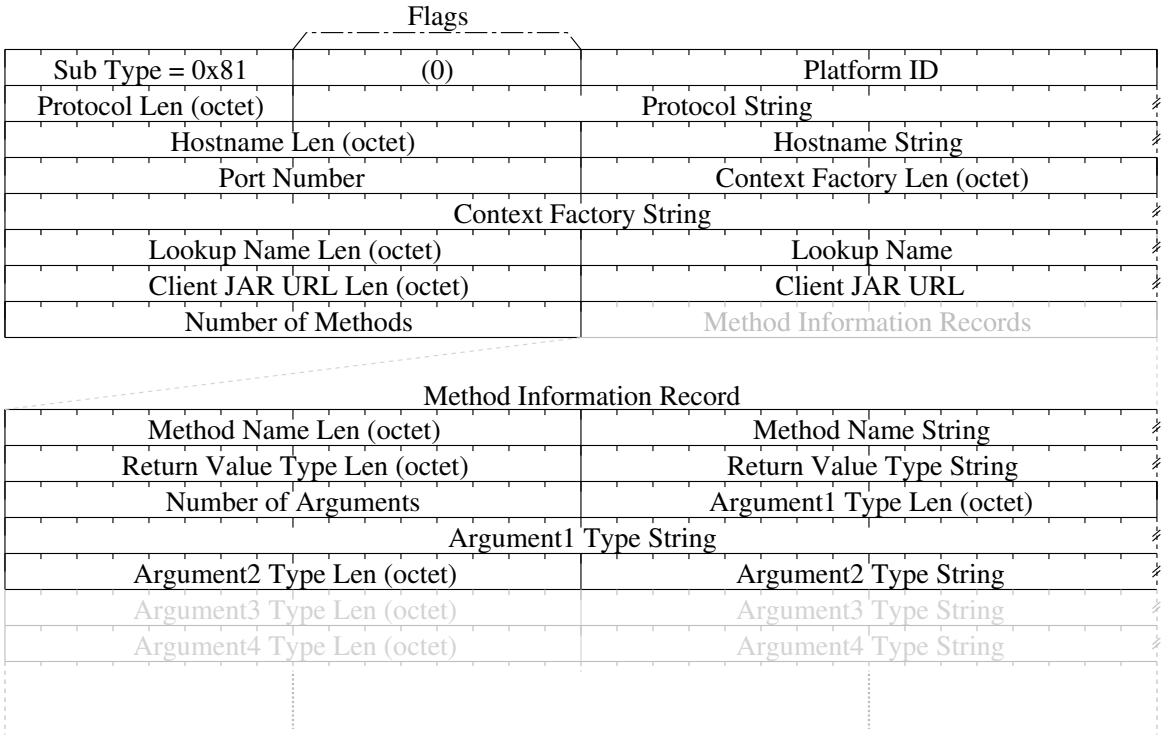

図 C.17 EJB Parameters Sub Option

iii. Hostname Len (16 ビット)
Hostname フィールドの長さ (バイト単位)．Hostname フィールドを省略する場合は，0 を指定する．

iv. Hostname (可変長)
コンポーネントが動作しているサーバのホスト名を指定する文字列．省略した場合は，パラメータ応答メッセージの送信元の IP アドレスの情報を利用する．

v. Port Num (16 ビット)
コンポーネントにアクセスするためのポート番号．

(e) コンポーネントの指定に関する情報

i. Context Factory Class Name Len (16 ビット)
Context Factory Class Name フィールドの長さ (バイト単位)．

ii. Context Factory Class Name (可変長)
コンポーネントにアクセスする際の Naming Context を実装しているクラス．

iii. Lookup Name Len (16 ビット)
Lookup Name フィールドの長さ (バイト単位)．

iv. Lookup Name (可変長)
サーバにおけるコンポーネントの識別名．

(f) コンポーネントを利用する際のライブラリの取得に関する情報
EJB コンポーネントを呼び出す際には，Client JAR ライブラリコードが必要になる．その URL を指定する．EJB コンポーネント呼出し時には，クライアントはここで指定された URL から Client JAR ライブラリコードを取得して，動的にロードし，呼出しを行う．なお，EJB サーバ実装によっては，Client JAR ライブラリコードを利用せずに呼出しを行うことができる．その場合には，このフィールドは省略される．

i. Client JAR URL Len (16 ビット)
Client JAR URL フィールドの長さ (バイト単位)．Client JAR URL を省略する場合は，0 を代入する．

ii. Client JAR URL: コンポーネントを呼び出すために必要な Client JAR ライブラリコードを取得するための URL.

(g) 提供するメソッドのリスト

i. Number of methods (16 ビット)
提供するメソッドの個数.

ii. Method Information Records
メソッドのインタフェースに関する情報．後述する Method Information Record を記述する．複数のメソッドを提供する場合は，Method Information Record を順に繋げて記述する．

EJB では，Web Service における WSDL や CORBA における IDL のような，標準的なインタフェースの記述のための言語が用意されていない．そのため，メソッドのインタフェースの情報を，EJB Parameter Sub Option に記述する．具体的には，Method Information Records フィールドに，メソッドに関する情報を記述する．個々のメソッドに関する情報を Method Information Record に記述し，それらを繋げて Method Information Records に記述する．

個々のメソッドに関する情報を記述するための Method Information Record は，次のフィールドから構成される．

(a) Method Name Len (16 ビット)
Method Name String フィールドの長さ (バイト単位).

(b) Method Name String (可変長)
メソッド名.

(c) Return Value Type Len (16 ビット)
Return Value Type String フィールドの長さ (バイト単位).

(d) Return Value Type String (可変長)
メソッドの返戻値の型の名前.

(e) Number of arguments (16 ビット)
メソッドが必要とする引数の個数.

(f) メソッドの引数に関する情報
メソッドの引数に関する情報を，引数の順番に Number of arguments フィールドで指定した個数だけ，繰り返して記述する．
引数に関する情報のフィールドとしては，次のものを記述する．

i. Argument Type Len (16 ビット)
Argument Type String フィールドの長さ (バイト単位)．

ii. Argument Type String (可変長)
引数の型の名前．

付 録D　動的探索のための API 一覧

7 章で述べた異種分散コンポーネントの Plug and Play プラットフォームでは，メディエータ部に Web Service での呼出しの形式に類似した，Java 言語による共通の API を用意した．本節ではこれを列挙する．なお，アプリケーションプログラム運用部も，この API を利用してメディエータ部を呼び出すように実装している．

コンポーネント呼出しの際のメディエータに対する呼出しは，`compdisc.CompCall` クラスに実装した．メソッドは次の通りである．

A) コンストラクタ
引数でコンポーネントのタイプを指定することにより，コンポーネントの探索を行う．

B) `select` メソッド
コンストラクタによって指定されたタイプで探索を行った結果，複数のコンポーネントが発見された場合は，適切なものを選択する必要がある．その場合の選択基準を指定する．この際に選択基準として，`compdisc.AbstractSelectPolicy` クラスを継承して実装したものを，引数として渡す．

C) `addparameter` メソッド
探索を行ってコンポーネントのメソッドを呼び出す際の，引数を指定する．

D) `setReturnType` メソッド
探索を行ってコンポーネントのメソッドを呼び出した結果得られる返戻値の型を指定する．実際に呼び出されるコンポーネントと異なる場合は，メディエータが変換する．

E) `setOperationName` メソッド
探索を行ってコンポーネントのメソッドを呼び出す際の，メソッド名を指定する．

F) `invoke` メソッド
探索を行って発見したコンポーネントのメソッドを，実際に呼び出す．コンポーネントのメソッドの返戻値を適切な型に変換し，`invoke` メソッドの返戻値として返す．

G) `isInvokeErr` メソッド
コンポーネントの呼出しの際に，エラーが発生したかどうかを返す．

H) `removeAllParameters` メソッド
`addparameter` メソッドで指定した引数を，クリアする．

I) `destroy` メソッド
終了処理を行う．