

修士論文

# 複合 Web サービス実行の効率化

指導教員 石田 亨 教授

京都大学大学院情報学研究科  
修士課程社会情報学専攻

藤代 祥之

平成 18 年 2 月 9 日

## 複合 Web サービス実行の効率化

藤代 祥之

### 内容梗概

本論文では、複合 Web サービス記述から同等のサービスを実現するプログラムを生成することで、複合 Web サービスを高速に効率的に実行する手法を提案する。

複合 Web サービスとは複数の Web サービスを構造的に組み合わせることで作成された Web サービスのことである。複合 Web サービスは通常、専用の記述言語を用いて記述され、実行エンジンに Web サービスとして配備される。複合 Web サービスはそれ自体も Web サービスであり、ある複合 Web サービスを別の複合 Web サービスのコンポーネントとして利用することで機能の集積が可能である。

複合 Web サービスはその柔軟な拡張性と相互接続性から企業システムなどで多く利用されている。しかし最近では、道路混雑情報 Web サービスや地図 Web サービスなどを組み合わせて旅行プランを生成する複合 Web サービスなど、一般利用者向けのサービスにおいても複合 Web サービスが利用され始めている。

より利用者に近い部分での複合 Web サービス利用が増加するにつれて、複合 Web サービスの実行速度の改善は大きな課題となっている。例えば、アプリケーション内部で利用される場合には、複合 Web サービスの処理時間がアプリケーションの処理速度に直接影響を与える。また、複合 Web サービスを再帰的に複合した場合には、単一の複合 Web サービスにおいては少しの処理時間の改善が、それを複合したサービスでは大きな違いとなって現れる。

本研究では複合 Web サービスを高速により効率的に実行することを目指す。特に、複合 Web サービスの実行速度や処理能力の低下を招く以下の問題に取り組む。

- 複合 Web サービス記述言語は多くの機能を実現するためより複雑になり、それ故実行時に多量の計算機資源が必要となり、処理能力の低下を招いている
- 複合 Web サービスは専用のエンジン上で実行されるが、実行において全ての制御とデータが実行エンジンに集中するため処理のボトルネックとなっ

てしまう

これらの問題を解決するために、本研究では複合 Web サービス記述からその複合 Web サービスの実行に最適化されたプログラムを生成する手法を提案する。この手法ではプログラムの生成時に、その複合 Web サービスを実行するために必要な機能を判別し、無駄なスレッドや CPU リソースの利用を抑制する。また、複合 Web サービスの制御フローをプログラムに置き換えることで実行時に必要な処理を減らし、コンパイラによる最適化の恩恵を受けやすくする。条件分岐やループなどの構造化アクティビティについては、対応する Java の制御構造に直接変換することで処理の負荷を抑える。このように最適なプログラムを生成することにより、システム全体の処理能力を向上させる。

また、この手法により生成されたプログラムは専用のエンジン無しで実行が可能となる。通常の Web サービスエンジンやクライアントマシン上でも利用することができる。これにより実行環境が分散でき実行エンジンが処理のボトルネックとなる問題を回避できる。

本研究では、BPEL4WS 記述から Java のプログラムを生成する BPEL2Java を開発し、従来の実行エンジンを用いる場合と生成されたプログラムを実行する場合について評価を行った。評価結果より、生成されたプログラムは実行エンジンによる実行に比べて高負荷時における単位時間当たりの処理件数が 1.5 倍になることを確認した。生成されたプログラムは、実行に必要な計算機資源が少なく高負荷時において処理件数の増加、平均応答時間の短縮に効果を発揮することが分かった。また、生成したプログラムを分散した環境で実行することにより、複合 Web サービスエンジンの持つボトルネックの問題を解消できることを確認した

# Improving Runtime Efficiency of Composite Web Services

Yoshiyuki FUJISHIRO

## Abstract

A composite web service is a web service that is constructed from multiple component web services. It is usually written in description languages such as BPEL4WS and executed on execution engines. Composite web service in itself can become a component of another composite web service. This is a great feature of composite web services.

Composite web service is often used in industrial arena because of its high interoperability and flexibility. But recently, it has begun to be used in consumer services such as travel planning web service, which is composed using map service, weather forecast service, and traffic information service.

The more often composite web service is used in the consumer scene, the more important the runtime performance of a composite web service becomes. This is because if it is used in the application, its response time directly affects the application's response time. In addition, if it is composed recursively, a little enhancement of execution speed in one component service will affect manyfold in the speed of the whole composite web service.

The purpose of this study is to improve runtime efficiency and performance of composite web service. In particular, I work on the following problems which deteriorate the performance and execution speed of composite web service.

- Description language for composite web services is too complex and has many functions. As a result, the engine needs more resource for execution, and this has ill effects on system performance.
- Composite web services require special engine for execution. Because all data and process about services are handled by this engine, it easily creates a bottleneck during the execution process.

To solve these problems, I propose the following two approaches.

The first is to generate program source codes from the description of a composite web service. Before generating the source codes, the generator system judges what functions are needed to execute this composite web services and

suppresses the use of needless threads and CPU resources. The system converts the process flow of composite web services into a program in order to reduce the runtime instructions and to make it optimized by the compiler of the programming language. In this manner, the generator enables the composite web service to run more efficiently.

The second is to perform the generated program on the client-side. The generated program does not require particular engines for execution so that it can be run on many environments such as client machines and web service engines. This removes the root cause of bottleneck in composite web service engines.

I developed BPEL2Java, which is the Java source code generator from BPEL4WS description and evaluated the proposed methods. From the experimental results, 150

# 複合 Web サービス実行の効率化

## 目次

第 1 章	序論	1
第 2 章	複合 Web サービス	4
2.1	Web サービス	4
2.1.1	SOAP	4
2.1.2	WSDL	5
2.2	複合 Web サービス	5
2.2.1	Orchestration と Choreography	6
2.2.2	BPEL4WS	7
2.3	複合 Web サービス例	7
2.4	複合 Web サービスの実行過程	9
第 3 章	複合 Web サービス実行の問題点	11
3.1	多機能化による処理能力低下	11
3.2	複合 Web サービス実行エンジンの処理ボトルネック	12
3.2.1	複合 Web サービス実行における処理ボトルネック	12
3.2.2	先行研究:Decentralized Orchestration	13
3.2.3	Decentralized Orchestration の問題点	14
第 4 章	複合 Web サービス記述からのプログラム生成	16
4.1	プログラム生成の利点	16
4.1.1	実行時リソースの減少と処理速度の向上	16
4.1.2	プログラミング言語の特徴の継承	17
4.2	BPEL2Java	18
4.2.1	アーキテクチャ	18
4.2.2	実装手法	20
4.2.3	効率化・高速化の手法	24
4.2.4	実装上の課題	26
第 5 章	生成されたプログラムの分散環境実行	28
5.1	生成されたプログラムの利用形態	28

5.1.1	Web サービスとしての利用	28
5.1.2	ライブラリとしての利用	29
5.2	生成されたプログラムの分散環境実行例	31
<b>第 6 章</b>	<b>Web サービスキャッシュ</b>	<b>33</b>
6.1	対象環境	33
6.1.1	キャッシュ空間	33
6.1.2	キャッシュ機構の設置箇所	34
6.1.3	キャッシュ可否の判断	35
6.2	Web サービスキャッシュの特徴	36
6.2.1	Web サービスの局所性	37
6.3	キャッシュアルゴリズムの要件	38
6.3.1	LNC-R アルゴリズム	38
<b>第 7 章</b>	<b>評価</b>	<b>40</b>
7.1	評価実験 1:生成プログラムによる実行の高速化	40
7.1.1	設定	40
7.1.2	評価結果	41
7.2	評価実験 2:生成プログラムによる実行の効率化	43
7.2.1	設定	43
7.2.2	評価結果	44
7.3	評価実験 3:プログラムの分散環境実行	45
7.3.1	設定	45
7.3.2	評価結果	46
<b>第 8 章</b>	<b>結論</b>	<b>47</b>
	謝辞	48
	参考文献	49

## 第1章 序論

Web サービスの大きな特徴の1つはインターネット上で相互に情報とその処理とをやり取りできることである。このようなりモートマシン上の処理呼出は、CORBA や RMI など既存の技術を用いても可能である。既存技術と Web サービスとの大きな違いは XML を基盤にした SOAP[1] や WSDL[2] といった標準仕様を利用することにより、OS やプログラミング言語などの実行環境に左右されることなく相互にメッセージをやり取りできることである。また、インターネットという開かれたネットワークにおいて情報をやり取りするために、通信プロトコルに依存していない点も Web サービスの利点である。

複合 Web サービスとは複数の Web サービスをコンポーネントとして構造的に組み合わせることで作成された Web サービスのことである。複合 Web サービスは通常 BPEL4WS[3] に代表されるような複合サービスの記述言語を用いて記述され、専用の実行エンジン上で実行される。複合 Web サービスには次のような特徴がある。

**高い相互接続性** 複合 Web サービスのインタフェースは WSDL により定義され、SOAP プロトコルを用いて通信が行われる。即ち外部から見れば通常の Web サービスと区別することはできない。このため、Web サービスの特徴である相互接続性の高さをそのまま受け継ぐ。

**コンポーネントの柔軟な組合せ** 複合 Web サービスの記述は Web サービスのインタフェース記述言語である WSDL を利用して行われる。WSDL は Web サービスのインタフェースと実装を分離して定義できるという特徴を持っている。このため、実装や環境に依存せず、コンポーネントとなる Web サービスを柔軟に組み合わせることができる。

**再帰的な複合** 複合 Web サービスはそれ自身も Web サービスである。そのため、別の複合 Web サービスのコンポーネントとして利用することができる。産業界では、この複合 Web サービスを利用したサービス指向アーキテクチャ (SOA) に注目が集まっている。これはシステム内の個々の機能を Web サービスとして実装し、それら Web サービスを組み合わせることで柔軟で大規模な企業システムを構築するというアプローチである。SOA に関連した Web サービスの標準仕様も数多く策定されており、SOA の基盤技術である複合 Web サービスの重要性が高まっている。

さらに、複合 Web サービスは企業内用途に留まらず、一般利用者向けのサービスに対しても利用され始めている。例えば、iPlat プロジェクトは利用者が旅行プランを作成するためのシステムを複合 Web サービスを用いて実現し実証実験を行っている [4]。このシステムでは道路交通情報サービス、地図サービス、気象情報サービスなど 13 ものサービスから成る複合 Web サービスを利用している。Web サーバ上のアプリケーションから複合 Web サービスが呼び出されるため、利用者はブラウザを利用して間接的に複合 Web サービスを利用できる。また、言語グリッドプロジェクトでは言語資源の構築と集積に複合 Web サービスを利用している [5]。このプロジェクトでは、個々の言語資源を Web サービス化し、それらを複合 Web サービス化して新たな言語資源 Web サービスを作成している。複合化された言語資源 Web サービスは、様々なアプリケーションでライブラリとして呼び出されることになる。

このように、より利用者に近い部分での複合 Web サービス利用が増加するにつれて、複合 Web サービスの実行速度の改善は大きな課題となっている。アプリケーション内部で利用される場合には、複合 Web サービスの処理時間がアプリケーションの処理速度に直接影響を与える。また、複合 Web サービスを再帰的に複合した場合には、単一の複合 Web サービスにおいては少しの処理時間の改善が、それを複合したサービスでは大きな違いとなって現れる。

本研究では複合 Web サービスを高速により効率的に実行するための手法を提案する。特に、複合 Web サービスの実行速度や処理能力の低下を招く以下の問題に取り組む。

- 複合 Web サービス記述言語は多くの機能を実現するためより複雑になり、それ故実行時に多量の計算機資源が必要となり、処理能力の低下を招いている
- 複合 Web サービスは専用のエンジン上で実行されるが、実行において全ての制御とデータが実行エンジンに集中するため処理のボトルネックとなってしまう

本研究では、上記の問題を解決し、複合 Web サービスを高速により効率的に実行するための手法を提案する。提案手法の一つは、複合 Web サービス記述から必要な機能だけを盛り込んだ最適化されたプログラムを生成することである。これにより実行時に必要な計算機資源を抑え、システム全体の処理能力の向上を期待できる。また、生成されたプログラムの実行に専用のエンジンが不要と

なり，通常の Web サービスエンジンやクライアントマシン上で利用することができるようになる．これにより実行環境が分散でき処理のボトルネックを低減できる．また，Web サービスに対するキャッシュについて考察を行う．Web サービスには，内部に状態を持たず，同じ呼出引数に対しては常に同じ結果を返すものが多く存在する．データベースを内部に持ちその中からデータを検索して返すようなサービスは，この良い例である．このような Web サービスに対する呼び出しは，呼出引数と結果の対応をキャッシュしておくことで，同じ引数による 2 度目以降の呼出では，時間のかかる処理を省略することができる．Web サービスの持つ特徴から Web サービスに適したキャッシュの仕組みを考察する．

以下，2 章では複合 Web サービスとそこで利用される標準仕様について述べる．3 章では複合 Web サービスの持つ問題点について論じ，4,5,6 章においてそれらに対する手法を提案し，7 章ではこれらの手法の有効性についての評価を行う．最後に 8 章では，結論と共に本手法の貢献をまとめる．

## 第2章 複合 Web サービス

本章では複合 Web サービスの基盤をなす Web サービスの規格について説明をし、複合 Web サービス記述と実行時の処理過程について述べる。

### 2.1 Web サービス

一般的な Web サービスは WSDL によりインタフェースが定義され、そのインタフェース定義を元に SOAP メッセージを作成し通信を行う。

#### 2.1.1 SOAP

SOAP は XML を基盤とした分散環境におけるメッセージ交換プロトコルである。SOAP では送受信するデータを XML 言語を利用して表現することで、言語やプラットフォームに依存しない通信を可能にしている。また、HTTP や SMTP などのインターネットの一般的通信プロトコルを下位プロトコルとして利用することでネットワーク環境に対する汎用性を高めている。

以下は、日英翻訳を行う機械翻訳 Web サービスを利用した場合の SOAP リクエストの例である。

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <translate
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <sourceLang xsi:type="xsd:string">JA</sourceLang>
      <targetLang xsi:type="xsd:string">EN</targetLang>
      <source xsi:type="xsd:string">こんにちは</source>
    </translate>
  </soapenv:Body>
</soapenv:Envelope>
```

また、それに対する SOAP レスポンスは以下のようになる。

```

<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <translateResponse
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <translateReturn xsi:type="xsd:string">Hello.</translateReturn>
    </translateResponse>
  </soapenv:Body>
</soapenv:Envelope>

```

XML を利用することで汎用性と可読性が高まっているが、日英翻訳という単純な Web サービス呼出を行うだけでも大きなサイズのデータをやり取りする必要があることがわかる。

### 2.1.2 WSDL

WSDL(Web Services Description Language) は Web サービスのインタフェース情報を記述するための記述言語である。WSDL では Web サービスがどのような機能をもつか、その機能を利用するためにはどのような要求を行えばいいのかなどを XML 言語を利用して記述することができる。WSDL の特徴としては、Web サービスの機能が実行のための通信プロトコルから分離して抽象的に記述されているため、ネットワーク環境や通信方法が変更された場合にもその Web サービスの機能に関する定義はそのまま利用できる点が上げられる。また、いくつかの記述要素を拡張してシステム固有の情報を WSDL に付加することができる点も WSDL 仕様の大きな特徴である。

## 2.2 複合 Web サービス

Web サービスの複合化とは、Web サービスをある機能を提供するコンポーネントと見なし、コンポーネントを複数組み合わせることで新たなサービスを作ることを行う。また複合化により作成されたサービスが Web サービスとして提供される場合には、これを複合 Web サービスという。

一般に複合 Web サービスは BPEL4WS や WS-CDL といった複合 Web サービ

スの記述言語を利用して記述される．このように記述された複合 Web サービスの実行には専用の実行エンジンが必要であり，複合 Web サービス記述は実行エンジン上に配備されることで実行可能となる．複合 Web サービスはそれ自身も Web サービスとして公開されるため，サービス利用者は SOAP など Web サービスの呼出プロトコルを利用して呼び出すことでサービスを実行できる．

### 2.2.1 Orchestration と Choreography

Web サービスの複合化の記述には，視点の違いから Orchestration と Choreography の 2 種類が存在している [6] ．

Orchestration とは特定の機能を持つサービスを実現するためのフローを記述したものである．Orchestration では，フローを実行する複合 Web サービスの実行主体の存在が仮定されており，その視点から複合 Web サービスを実行可能な詳細さで記述することになる．Orchestration は直接的に実行フローを表現してあるため，専用の実行エンジン上に配備することでそのまま実行が可能である．Orchestration による複合 Web サービス記述言語としては BPEL4WS が良く知られている．

一方 Choreography とは Web サービス間の関係を大域的な視点から記述したものである．Choreography では，ある Web サービスとその利用者との間の観測可能なインタラクションを定義する．ここでサービスの利用者とは広義の意味の利用者であり，別のサービスやクライアントプログラムや人を含む．複数のコンポーネント Web サービスを利用して新たなサービスを実現しようとするとき，コンポーネント Web サービスは一定の順序や制約に従って交互にメッセージをやり取りすることになる．Choreography ではこのようなメッセージ交換のインタラクションやその制約を定義することにより，Web サービスの複合化方法を間接的に表現する．Choreography には Orchestration に見られるような複合 Web サービスの実行主体は存在しない．そのため，Choreography 記述そのものは実行可能なものではなく，記述を元に Orchestration などの実行可能なサービス記述を生成することで実行が可能となる．この意味では Orchestration と Choreography は相互補完的なものである．Choreography による複合 Web サービス記述言語としては，WS-CDL が挙げられる．

現在，複合 Web サービスの記述には Orchestration の視点に基づいた BPEL4WS が広く利用されている．BPEL4WS で記述された複合 Web サービスを実行するための環境はオープンソース製品や，商用製品などが多数存在している．そこ

で、本論文においては以降 Orchestration の視点から複合 Web サービスについて考えていく。また、特に BPEL4WS を記述言語として利用し考察を行うが、得られた知見は必ずしも BPEL4WS に限定されるわけではなく、広く Orchestration 型の複合 Web サービスに対して適用できる。

### 2.2.2 BPEL4WS

BPEL4WS(Business Process Execution Language for Web Services) は複数の Web サービスを連携させるためのワークフロー記述言語である。実行可能なビジネスプロセス (Executable Business Process) を記述することと、抽象的なビジネスプロトコル (Business Protocol) を記述することが可能である。構造は XML をベースにしており、WSDL で記述された Web サービスのインタフェースを利用し、サービスの呼出やデータの操作などの簡単な原子アクティビティを記述することができる。また、順次実行、条件分岐、繰り返し、並列実行などの構造化アクティビティを利用し原子アクティビティと組合せることで、複雑なプロセスを記述できる。現在標準化団体 OASIS において標準化の作業が行われており、最新バージョンは BPEL4WS 1.1 である。

BPEL4WS では構成された複合プロセスを実行するためのインタフェースは WSDL で定義される。そのため外部から該当 Web サービスを実行する際に、その Web サービスが BPEL4WS を利用して構成された複合プロセスであるのか、それとも通常の Web サービスであるのかを意識することなく実行できる。この特徴を利用することで、複合 Web サービスをコンポーネント Web サービスとして利用し、より複雑な複合プロセスを簡単に記述することが可能となる。

## 2.3 複合 Web サービス例

複合 Web サービスの 1 例として、図 1 のような複合 Web サービスを考える。このサービスは入力として日本語の文を受け取り、それを中国語と英語へと機械翻訳し、それぞれの結果を 1 つの変数に代入して返す。図 1 の図はこの複合 Web サービスの動作を UML アクティビティ図を用いて表現したものと、この複合 Web サービスを実現する BPEL4WS 記述の一部である。

この複合 Web サービスでは、2 つの Web サービスを内部で呼び出している。1 つは日本語から中国語への機械翻訳 Web サービスで、図中の”invoke JtoC-trans”がこのサービスに対する呼出のアクティビティをあらわしている。またもう 1 つは日本語から英語への機械翻訳サービスで、図中の”invoke JtoE-trans”

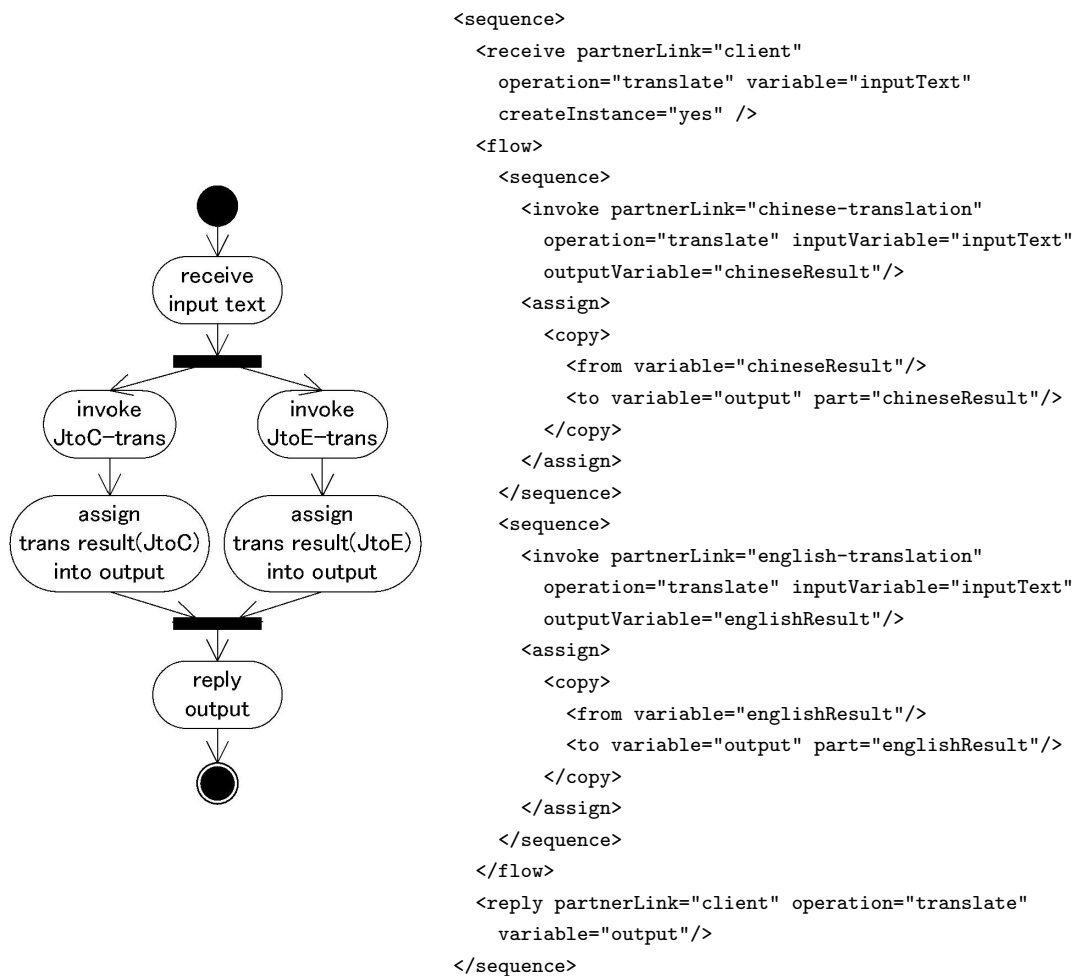


図 1: Activity diagram and BPEL4WS description of the example service

がこのサービスに対する呼出アクティビティである。この2つのアクティビティ直前での制御フローの分岐は、ワークフローの And-Split に相当し、2つのフローが並行して実行されることを表している。また、2つのフローの結合は And-Join であり、前のアクティビティが全て完了後、次のアクティビティが実行される。BPEL4WS においては *flow* タグにて囲まれたアクティビティは並列して実行される。また *sequence* タグで囲まれたアクティビティは順番に実行される。

つまりこのサービスでは、日本語の入力文を *receive* すると、日中機械翻訳サービスの *invoke* とその結果の変数への *assign* 処理、日英機械翻訳サービスの *invoke* とその結果の変数への *assign* 処理が並行して実行される。並列処理が両方とも完了した時点で、処理結果が呼出元に対して *reply* されることになる。

## 2.4 複合 Web サービスの実行過程

図 2 は典型的な複合 Web サービスの実行過程を表したものである。図の Composite Web Service 中の丸い要素は複合 Web サービスを構成する各アクティビティを示している。receive はクライアントからのメッセージの受け取りを、invoke はコンポーネント Web サービスの呼出を、reply はクライアントへの結果の返送を行うアクティビティである。また、各アクティビティに出入している矢印はデータのフローを表現している。

この図ではクライアントからの複合 Web サービス実行要求が実行エンジンに渡ると、実行エンジンは予め用意されている複合 Web サービス記述をもとに実行を開始する。実行エンジンはコンポーネント Web サービスの呼び出しを 2 回行い、サービス記述に基づいてクライアントに結果を返却する。

図 2 では、このような複合 Web サービス実行の過程を図中縦の点線により (A),(B),(C) の 3 つの部分に分けて考えている。

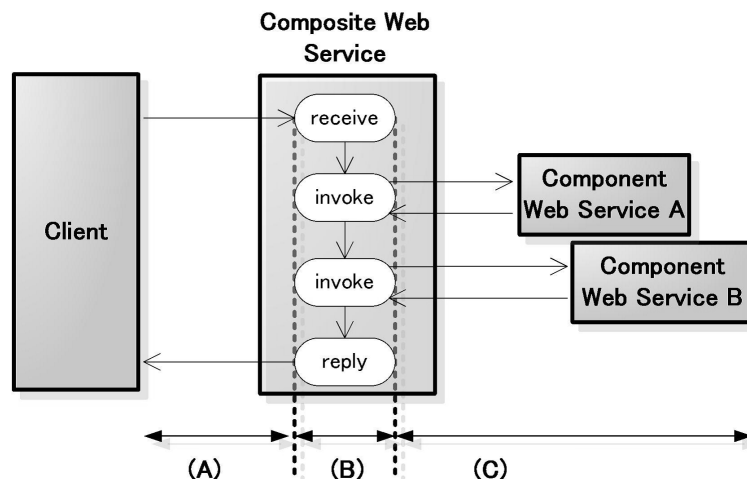


図 2: Three partitions in the process of executing a composite web service

(A) 複合 Web サービスへの SOAP 呼出 クライアントからの SOAP を用いた複合 Web サービス実行処理である。クライアントの視点にたてば、呼出先の Web サービスが複合 Web サービスであるのかそれとも単体の Web サービスであるのかは判断できない。そのため処理は、複合 Web サービスに対して通常の SOAP 通信が行われる。この部分は具体的には以下のような処理が含まれる。SOAP 要求時には、

1. クライアントマシン上でサービス要求データを XML 形式へ変換
2. クライアントと複合 Web サービスサーバ間でデータ通信
3. 複合 Web サービスサーバ上で受信した XML データを解析しサービス要求データを生成

の3つの処理が行われる。また、それに対する SOAP 応答として、上記と逆の手順でサーバからクライアントへとサービス応答データが返される。ただし、この複合 Web サービスが呼出だけの一方向サービスの場合には、応答時の処理は行われない。

- (B) 複合 Web サービスのプロセス処理 複合 Web サービス記述にしたがって、サービスを実行する部分である。条件分岐や繰り返し、例外捕捉、変数への代入など指定されたサービス記述を解析し、対応する処理を実行する。ただし、コンポーネント Web サービスの呼出処理は後述の (c) に含まれる。
- (C) コンポーネント Web サービス呼出 複合 Web サービス内で、コンポーネント Web サービスを呼び出す処理である。この部分はネットワーク通信、サービス要求/応答データの XML データとの相互変換、コンポーネント Web サービスのサービス部分の実行処理を含む。

## 第3章 複合 Web サービス実行の問題点

本研究の目的は、複合 Web サービスの実行をより高速かつ効率的にすることである。すなわち、複合 Web サービスの実行時における処理効率の向上、応答時間の短縮を目指す。

複合 Web サービス実行時の問題点として、処理効率、応答時間を低下させる次の3つの問題を取り上げる。

- 複合 Web サービス記述言語は多くの機能を実現するためより複雑になり、それ故実行時に多量の計算機資源が必要となり、処理能力の低下を招いている
- 複合 Web サービスは専用のエンジン上で実行されるが、実行において全ての制御とデータが実行エンジンに集中するため処理のボトルネックとなってしまう
- 複合 Web サービスは外部 Web サービスとの通信処理が存在するため多くの時間がかかると共に、処理速度がネットワーク状態に影響されてしまう

### 3.1 多機能化による処理能力低下

複合 Web サービスは元々産業界の貢献により発展普及してきた技術である。そのため BPEL4WS に代表されるような複合 Web サービスの記述言語は、ビジネス利用における様々な要求に対応するため高度な機能が組み込まれているのが一般的である。そして、実行エンジンは高度な機能を実現するためにより複雑な処理を要求され、実行時に必要なリソースが増加してしまう。

例えば、BPEL4WS の持つ高度な機能の1つとしてプロセス実行中のイベント捕捉がある。捕捉できるイベントには、プロセス中のある時点から一定時間経過した際に発生するアラームイベントと、プロセス実行中に外部から予め決められたメッセージを受信した際に発生するメッセージイベントの2つが存在する。イベントにはアクティビティを関連付けられ、イベントを捕捉すると実行中のメインフローと並行して、関連付けられているアクティビティが実行される。この機能の実現のために、イベントの発生を監視するスレッド処理や、現在実行中のプロセスの管理、プロセスの並列実行などに処理時間とリソースが消費されることになる。特に実行要求が集中する状況においては多くのリソースが消費され、処理効率の低下を招いていしまう。

このような高度な機能は複合 Web サービスを用いてビジネスフローを実現する場合には有用かもしれない。しかし、いくつかの Web サービスを組み合わせで新たな複合 Web サービスを作成するといった単純な複合 Web サービスを実行する場合には、実行時のオーバーヘッドとなってしまう。

企業利用のために高度な機能の提供を行うと共に、一般利用のために単純な複合 Web サービスを高速に効率的に実行することが重要となる。

## 3.2 複合 Web サービス実行エンジンの処理ボトルネック

### 3.2.1 複合 Web サービス実行における処理ボトルネック

通常、Orchestration 型の複合 Web サービスの実行では全ての制御が実行エンジン上で処理される。図 3 は複合 Web サービスの一般的な実行方法を表したものである。図ではクライアント X, クライアント Y, ホスト Z の 3 つがホスト A の実行エンジン上で動作する複合 Web サービス A のクライアントである。一方、ホスト B, ホスト C は複合 Web サービス A において利用されるコンポーネント Web サービスを提供している。クライアント X ではアプリケーション X が、クライアント Y ではアプリケーション Y が複合 Web サービスを SOAP 経由で呼び出す。一方、ホスト Z 上では、別の複合 Web サービス Z が動作しており、複合 Web サービス A はこのサービスのコンポーネント Web サービスとして SOAP 経由で呼び出される。なお、図中の矢印は制御およびデータのフローを、図中の丸は複合 Web サービス記述のアクティビティを表している。

図を見ると、クライアントからの複合 Web サービス呼出要求も、コンポーネント Web サービスへの呼出要求も全て実行エンジン上にて処理されていることが良く分かる。制御が全て実行エンジン上に集中しているため、行き交うデータも全て実行エンジンを経由することとなる。

つまり、複合 Web サービスの実行のために、実行エンジン上に多くの記憶領域と計算時間が必要になる。このため、複合 Web サービス呼出が頻発する場合や、複合 Web サービスの記述が複雑な場合には、実行エンジン上での処理がボトルネックとなり、複合 Web サービスエンジンの処理能力を下げってしまう。特に BPEL4WS に代表されるような複合 Web サービス記述言語は、様々な要求に対応するため多くの機能を有しており、実行エンジン上での処理が複雑になりやすい。

高負荷時に実行エンジンが処理のボトルネックになりやすい事は、Orchestra-

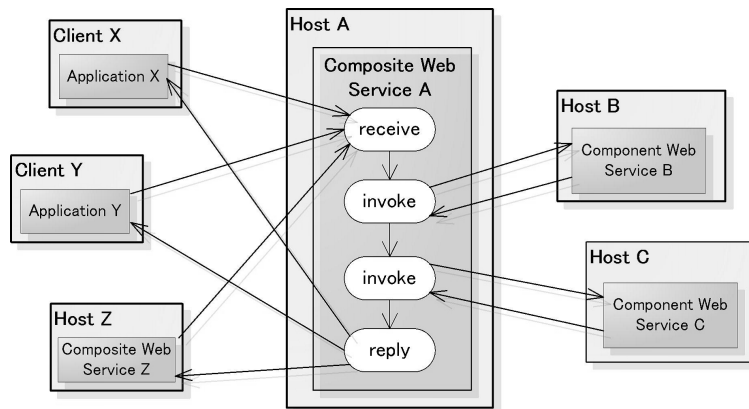


図 3: centralized orchestration

tion 型複合 Web サービスの潜在的な問題である。

### 3.2.2 先行研究:Decentralized Orchestration

この問題の解決策として Decentralized Orchestration という手法が提案されている [7, 8]。Decentralized Orchestration では、複合 Web サービス記述から各コンポーネント Web サービス間のデータと制御の依存を解析し、1 つの複合 Web サービス記述を同等のサービスを実現する複合 Web サービス記述の集合に分割する。分割された複合 Web サービス記述はネットワーク上の分散された場所に配備され、相互に呼出を行いながら 1 つのサービスを提供する。

図 4 は図 3 を分割して、Decentralized Orchestration の形式に変換したものである。ホスト A, B, C に配備されている複合 Web サービス A-0, A-1, A-2 は図 3 における複合 Web サービス A を分割したものである。

図 3 においては、複合 Web サービスは 1 つの実行エンジンで実現されており、これは Decentralized Orchestration と対比して Centralized Orchestration と呼ばれている。一方、図 4 では分割された 3 つの制御フローが存在しており、分散された実行エンジン上でそれぞれが動く。制御フローの中で別の制御フローに対して一方向の呼び出しを行うことで、3 つの制御フローが連携して利用者に対して 1 つの機能を実現している。

Centralized Orchestration では複合 Web サービスの実行に必要な全てのデータが実行エンジンを經由してコンポーネント Web サービスに渡されていたが、Decentralized Orchestration では必要なデータのみを必要な相手へと送信することが可能になる。例えば、図 3 において、コンポーネント Web サービス B の実行結果をコンポーネント Web サービス C が呼出引数として利用しているとす

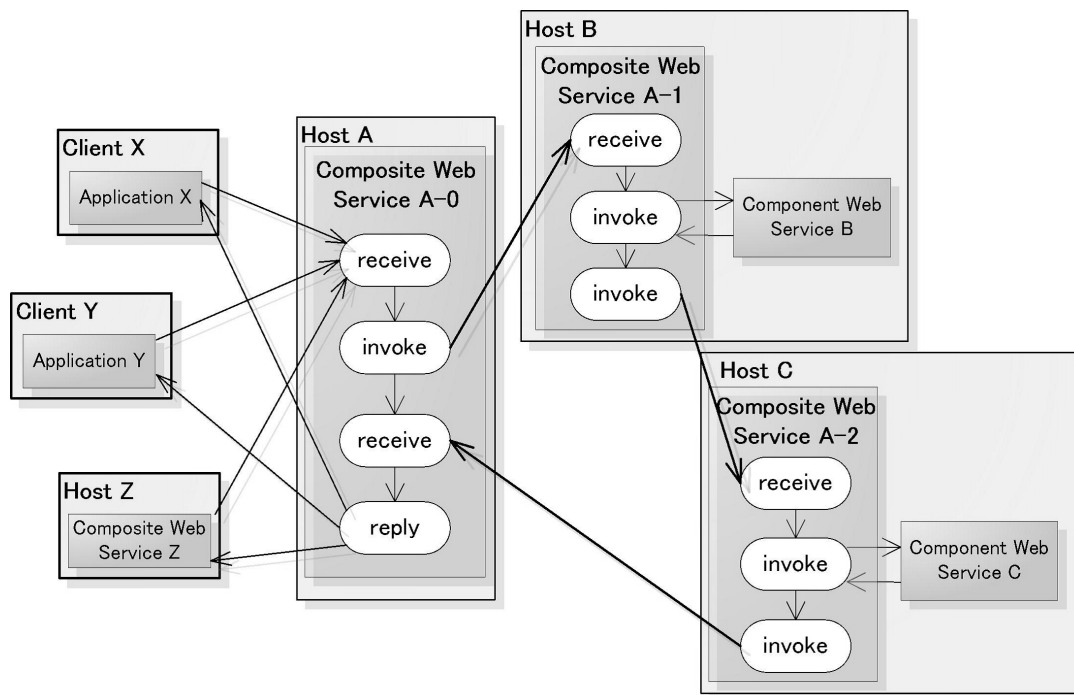


図 4: decentralized orchestration

る。Centralized Orchestration においては、コンポーネント Web サービス B の実行結果は、一度実行エンジンを經由してコンポーネント Web サービス C へと渡される。

一方、Decentralized Orchestration においては、コンポーネント Web サービスの B の実行結果は、分割された複合 Web サービス記述 A-1 によって、複合 Web サービス記述 A-2 が実行され、ホスト A を介さずにコンポーネント Web サービス C に渡される。このように Decentralized Orchestration では、ボトルネックとなる処理を分散させ、通信データサイズを減らすことで、複合 Web サービスの実行効率を高めている。また、Decentralized Orchestration には、送受信する情報を限定することができ、セキュリティに観点においてより安全である。情報のコントロールは企業利用においては非常に重要である。

### 3.2.3 Decentralized Orchestration の問題点

このように Decentralized Orchestration には利点が存在するが、しかしいくつかの問題も存在している。

Decentralized Orchestration では分割された複合 Web サービス記述を実行するために、複数の実行エンジンを用意しなければならないという問題がある。特

に Decentralized Orcehstration による処理能力向上を最大限発揮するためには、利用するコンポーネント Web サービスと同一のホスト上で分割された複合 Web サービス記述を実行する必要がある。企業内や企業間において複合 Web サービスを実現する場合には、このような環境を用意することは比較的容易である。しかし、インターネット上で公開されている Web サービスをコンポーネントとして利用し複合 Web サービスを作成しようとする場合には、複合 Web サービス実行エンジンをコンポーネント Web サービスの近くに用意することは難しい。このような外部の Web サービスを組み合わせた複合 Web サービスにおいては、Decentralized Orchestration の実現が難しく、何らかの別の方法が必要となってくる。

また、Decentralized Orchestartion には、実行エンジンの潜在的なボトルネックが残ったままであるという問題がある。複数の実行エンジンにより処理の分散を行っても、なおクライアントからの実行要求は主となる実行エンジンが全て処理を行っている。この主実行エンジンは Centralized Orcehstration における実行エンジンに対応し、Decentralized Orcehstration においてはクライアントからの要求を受け取る実行エンジンとして機能している。主実行エンジンは、クライアントからの要求を受け取ると分散された実行エンジンに処理の実行を依頼する。その後、主実行エンジンは処理の実行結果を受け取ると、クライアントに対して処理結果を返す。Centralized Orchestration と比較すると主実行エンジンにおける処理量は減少してはいるが、それでもなおこの部分がボトルネックとなる可能性はなくなっていない。クライアントからの要求は全て主実行エンジンを経由して処理されるため、Decentralized Orchestration は多くの処理要求が発生すると主実行エンジンが処理のボトルネックとなってしまう。

## 第4章 複合 Web サービス記述からのプログラム生成

3.1 節で述べた多機能化による処理能力低下の問題を解決するため、複合 Web サービス記述から必要な機能だけを盛り込んだ最適化されたプログラムを生成する手法を提案する。つまり、高度な機能を利用した複合 Web サービス記述からはそれを実現する複雑なプログラムを、単純な機能のみを利用した複合 Web サービス記述からは簡潔なプログラムを生成する。

多機能化による処理能力低下を回避するためのアプローチとしては、一般的な利用に適した単純で簡潔な複合 Web サービス記述と実行環境を作成することも考えられる。複合 Web サービス記述には、ビジネスフローでしか利用しないような複雑な機能は盛り込まず、外部 Web サービス呼出や変数の代入などの基本的アクティビティと、それらを構造化するための条件分岐や繰り返しなどを用いることで、オーバーヘッドのない高速な実行が可能となる。

しかし、独自仕様の利用は複合 Web サービスの既存資産が利用できないという欠点を併せ持つ。例えば、BPEL4WS を容易に記述するための GUI ツール、BPEL4WS に関する様々なドキュメント、また BPEL4WS を用いた既存の複合 Web サービス記述などが存在することは、BPEL4WS の持つ大きな利点の 1 つである。このためここでは既存の記述言語をそのまま利用できるプログラムの生成というアプローチをとる。

### 4.1 プログラム生成の利点

複合 Web サービス記述からのプログラム生成には以下のような利点がある。

#### 4.1.1 実行時リソースの減少と処理速度の向上

複合 Web サービスの実行のために必要なリソースを減少させることは重要である。実行要求が少ない場合には、利用可能なリソースは潤沢に余っており、少しの実行時リソースの差によって処理効率や処理速度に違いが発生することは無いかもしれない。しかし、実行要求が集中した場合には、事情は異なってくる。集中した実行要求により、マシン上のリソースを多く消費すると、それ以上のリソースの確保が難しくなる。確保できる場合でも、ディスクへのアクセスが発生するなど、パフォーマンスの急激な低下が発生しやすい。このような状況では、複合 Web サービスの処理効率や応答時間も低下してしまう。しかし、実行に必要なリソースが少ない状況では、システムのリソースが枯渇する状態

になりやすく、実行要求が集中しても処理効率の低下に陥りにくいといえる。

複合 Web サービス記述からプログラムを生成すると、以下のような理由で実行時リソースの低減、処理速度の向上に貢献する。

- 生成されるプログラムには、入力された複合 Web サービス記述の実行に必要な機能のみを含める。複合 Web サービス記述を読み込み後、実行にはどの機能が必要かの判別を行う。プログラム生成時には、判別結果にしたがいできるだけ簡潔なプログラムを生成する。これにより、余計なスレッドの実行などを抑えることができる。
- 複合 Web サービスの制御フローをプログラムに書き換えることで、実行時の処理を減少させる。一般の複合 Web サービス実行エンジンは、複合 Web サービスがエンジンに配備されると、渡されたファイルから複合 Web サービス記述をメモリ空間に読み込み待機する。そして実行要求が来た際に、メモリ内の記述を元に次に実行するアクティビティを参照し処理を行う。一方、本手法では渡されたファイルから複合 Web サービス記述を読み込んで、ソースコードを生成する。コンパイル型のプログラミング言語の場合、生成されたソースコードはコンパイルにより実行可能な状態となる。そして実行要求が来た際には、コンパイルされたバイナリプログラムが実行される。実行フローはマシン語や中間言語としてバイナリ表現されており、実行時にメモリ内に複合 Web サービス記述を持つ必要が無く、また制御フローの参照処理も必要ない。
- 複合 Web サービス記述にはプログラミング言語の構文を用いて置き換えることが可能なアクティビティが存在している。例えば、繰り返し処理や条件分岐処理の構造化アクティビティは、一般的なプログラミング言語の `for(while)` や `if` を用いて、直接記述することができる。また、例外の捕捉や、例外のスローなどもプログラミング言語の構文を用いて記述可能である。このようにアクティビティをプログラミング言語の構文に置き換えることで、より高速な実行が期待できる。また、コンパイル時の最適化もされやすくなる。

#### 4.1.2 プログラミング言語の特徴の継承

また別の利点としては、生成されるプログラミング言語の持つ特徴を継承することができるということがある。

例えば、プログラミング言語として Java を利用することで、Java のメリッ

トである移植性の高さを手にすることができる。生成されたプログラムは Java の実行環境が存在すればどこでも実行できる。手元のクライアントマシンや Java が動作可能な携帯でも実行が可能となる。プログラミング言語として C++ を利用すれば、実行環境に最適化されたバイナリを生成し、より高速に実行するという C++ の特徴を享受できる。

## 4.2 BPEL2Java

本研究では、複合 Web サービス記述からプログラムを生成するためのシステム BPEL2Java の実装を行った。ここからは、BPEL4WS による複合 Web サービスの記述を読み込み、同等のサービスを実現する Java のソースコードを生成するシステム BPEL2Java を元に、実装の詳細について述べる。

### 4.2.1 アーキテクチャ

図 5 は BPEL2Java の動作を表すブロック図である。

BPEL2Java は入力として BPEL4WS ファイルと WSDL ファイルを受け取り、出力として Java のファイル群を生成する。

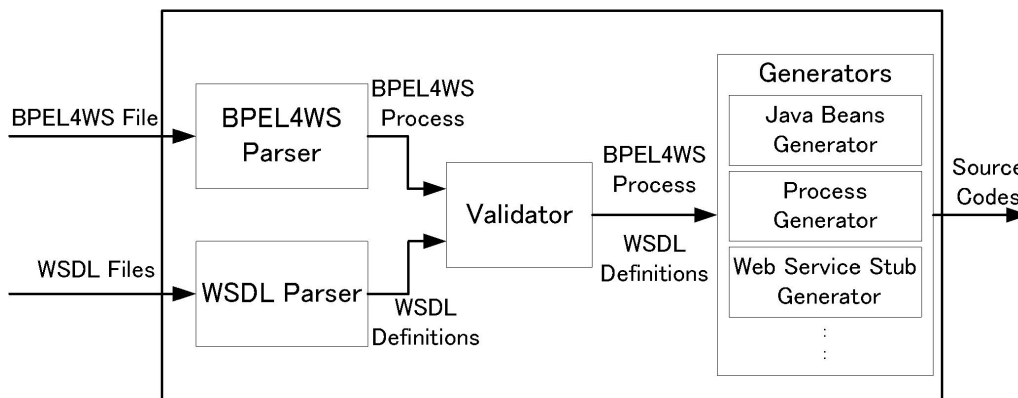


図 5: The block diagram of BPEL2Java

BPEL2Java は Parser, Validator, Generator の 3 つの部分に分けられる。Parser には BPEL4WS を処理するパーサと WSDL ファイルを処理するパーサの 2 つがある。

BPEL4WS における複合 Web サービス記述は、1 つの BPEL4WS ファイルと複数の WSDL ファイルによって定義される。BPEL4WS ファイルには、複合 Web サービスを定義する中心のファイルであり、複合 Web サービスの制御フ

ローやプロセス内で利用する変数が記述されている。一方、WSDL ファイルには、プロセス内で利用するコンポーネント Web サービスに関する情報が記載されている。これには、コンポーネント Web サービスと SOAP 通信するための情報や、プロセスで利用する変数型の XML スキーマによる定義などが含まれる。

入力ファイル群は最初にパーサに渡される。BPEL4WS パーサは BPEL4WS ファイルを読み込み、複合 Web サービス情報を格納した BPEL4WS プロセスオブジェクトを生成する。一方、WSDL パーサは複数の WSDL ファイルを読み込み、複数の WSDL オブジェクトを生成する。

パーサにより生成されたオブジェクトは、次にバリデータに渡され記述に誤りや矛盾している点がないか検証される。問題がない場合には、オブジェクトがそのままジェネレータに渡される。

BPEL2Java には複数のジェネレータが登録しており、それぞれのジェネレータが BPEL4WS プロセスオブジェクトや、WSDL オブジェクトを元に多くのファイルを生成する。

ここで生成されるファイルには、以下のようなものがある。

**プロセスクラス** 複合 Web サービスで記述されたプロセスに対応する Java クラスである。生成されるファイル群の中で最も重要なファイルといえる。BPEL2Java では 1 つの BPEL4WS ファイルに対して、1 つのプロセスクラスが生成される。

**コンポーネント Web サービス用スタブクラス** プロセスで利用されるコンポーネント Web サービスに対しての SOAP 呼出処理を簡単に行うための Java クラス群である。スタブクラス内で煩雑な SOAP 処理の全てが行われる。そのため、スタブクラスのメソッドを実行することで簡単に外部サービスの呼び出しが行えるようになる。

**ユーザ定義型用変数クラス** WSDL 内で XML スキーマを利用して複雑な型を定義することができる。このようなユーザ定義型をプロセス内で変数として利用するための Java クラスである。

**ビルドファイル** 生成される Java ファイル群はソースコードであり、実行にはそれらをコンパイルする必要がある。ビルドファイルを使うことで、コンパイルを容易に行える。

全てのファイルを出力すると、これで BPEL2Java の処理は終了する。

BPEL2Java の利用者もしくは利用ソフトウェアは、生成されたビルドファイ

ルを元にソースコードをコンパイルする。コンパイルされたクラスファイルを実行環境に配備することで、複合 Web サービスプログラムを実行することができる。

#### 4.2.2 実装手法

BPEL2Java においては、様々な Java のクラスが生成される。ここではそれらの生成方法について説明する。

プロセスクラスの生成 プロセスクラスは、複合 Web サービスのプロセスフローを表した Java クラスである。

```
<process name="ProcessName" targetNamespace="uri" ... >
  <partnerLinks>
    <partnerLink name="ncname" ... />      1) component web services
  </partnerLinks>

  <variables>
    <variable name="ncname" ... />        2) variables
  </variables>

  ... (activities)                          3) activities
</process>
```

図 6: The basic structure of BPEL4WS language

図 6 は BPEL4WS の基本的な構造を表したものである。図 6 に示されているように、BPEL4WS による複合 Web サービス記述は主に 3 つの要素から成り立っている。

1. プロセス内で利用するコンポーネント Web サービスの定義
2. プロセス内で利用する変数
3. プロセスの実行フロー（アクティビティ）

プロセスクラスの生成においては、この 3 つの要素を Java の要素に変換する。

プロセス内で利用される変数は、このプロセスクラスのインスタンス変数として宣言される。また、BPEL4WS における実行フローは、外部から呼出可能な公開メソッドとして、このプロセスクラスに実装される。ここでは、この公開メソッドをプロセスのサービスメソッドと呼ぶことにする。つまり、生成さ

れたプロセスクラスのサービスマソッドの実行は，BPEL4WS プロセスの実行に対応するようになる．生成されるプロセスクラスの基本的構造を表したのが図 7 である．

```
public class ProcessName {
    // some declarations of variables here
    ....

    //constructor
    public ProcessName(){
        super();
    }

    // the public method corresponding to activities here
    public Object serviceMethod(Object arg){
        ....(activities)
    }
}
```

図 7: The basic structure of Process class

図 7 ではサービスマソッド *serviceMethod* は，Object 型の引数 *arg* を受け取り，Object 型の値を返すメソッドとして定義されている．しかし実際には，サービスマソッドのメソッド名，引数の型と数，戻り値の型は，WSDL ファイル内に含まれる複合 Web サービスのインタフェース情報から決定される．

また，サービスマソッドの内部処理は，BPEL4WS の制御フローを構成するアクティビティから対応するソースコードが生成される．表 1 は，BPEL4WS の主要な各アクティビティに対する説明とその機能の BPEL2Java での実装方法を記述した表である．BPEL4WS では，制御フローは構造化アクティビティと基本アクティビティの組合せにより表現されている．

サービスマソッドの内容を出力する際には，アクティビティの構造関係を維持したまま，対応する Java の処理へと置き換えていく．

図 8，図 9 は，このような方法で BPEL4WS の処理フローから，サービスマソッドを生成した例である．図 8 は実行要求を受け取ると，コンポーネント Web サービスの呼出を行い，結果を呼び出し要求発行元に返却するという単純

BPEL4WS activity	Description	Implementation in BPEL2Java
Structured Activities		
sequence	perform contained activities sequentially	do nothing
switch	perform a given activity if the given condition is true	replace into if,elseif,else syntax
while	perform a contained activity repeatedly until the given condition no longer holds true	replace into while syntax
flow	performe contained activities concurrently	use threads and execute concurrently
Basic Activities		
invoke	invoke a component web service	call the method of generated stub classes
receive	receive a message from clients or other web services	assign the service method parameters into the specified variable
reply	reply a message to clients	use return syntax to reply the specified variable data
assign	copy data from one variable to another	clone the data of the variable and assign to the specified variable
throw	throw an exception	replace into throw synatax
wait	wait for a certain period of time or until a certain deadline	use the sleep method in Thread class

表 1: BPEL4WS activities and those implementation in BPEL2Java

な複合 Web サービスを表している。receive, invoke, reply で表現される各処理が BPEL4WS 記述と Java のソースコードにおいて対応していることがわかる。一方 BPEL4WS 記述において、最も外側のアクティビティである sequence は、サービスメソッドに対応する部分はサービスメソッド内に生成されていない。これは、sequence アクティビティは包含するアクティビティを順番に実行することを意味しているが、Java のソースコードでは特別な記述をしなくとも命令が上から順番に実行されていくためである。BPEL2Java ではこのように、BPEL4WS のアクティビティから対応する Java ソースコードを生成する。

コンポーネント Web サービス用スタブクラスの生成 プロセスにて利用されるコンポーネント Web サービスに対しては、SOAP を用いて呼び出すためのスタブクラス群を生成する。このファイル群の生成には Web サービスエンジン (Apache Axis) に添付されているツールを利用する。このツールは対象となる Web サービスの WSDL ファイルを元に、該当の Web サービスを SOAP 経由で呼び出すための Java プログラムを生成することができる。対象 Web サービスの WSDL は BPEL4WS ファイルと共にシステムに渡されるため、この WSDL ファイルとツールから Java のスタブクラスを生成する。これら生成されたファイル群は、公開メソッドの内容を生成する時点で利用される。

```

<sequence>
  <receive partnerLink="client" portType="impl:Demo"
    operation="echo" variable="demoRequest" createInstance="yes" />
  <invoke partnerLink="echo" portType="impl:SleepEchoService" operation="echo"
    inputVariable="demoRequest" outputVariable="demoResponse"/>
  <reply partnerLink="client" portType="impl:Demo"
    operation="echo" variable="demoResponse"/>
</sequence>

```

☒ 8: Activities written in BPEL4WS

```

public java.lang.String echo(java.lang.String input)
  throws java.rmi.RemoteException{
  // <sequence>
  demoRequest.put("input", input); // <receive>

  try { // <invoke>
    SleepEchoService sei =
      new SleepEchoServiceServiceLocator().
        getSleepEcho(
          new java.net.URL(
            partnerLink_echo_partnerRole.getAddress().toString()));
    demoResponse.put("echoReturn",
      sei.echo(
        (String)(demoRequest.get("input"))));
  } catch (javax.xml.rpc.ServiceException e) {
    throw new RemoteException("ServiceException", e);
  } catch (java.net.MalformedURLException e) {
    throw new RemoteException("MalformedURLException", e);
  }

  return (String)demoResponse.get("echoReturn"); // <reply>
  // </sequence>
}

```

☒ 9: A generated source code from BPEL4WS

ユーザ定義型用変数クラスの生成 プロセス内でユーザ定義型の変数が利用される場合には、WSDL 内に含まれる XML スキーマ定義により変数クラスを生成する必要がある。Java スタブクラスを生成するのと同様のツールを利用することで、WSDL 内の XML スキーマから Java クラスを生成する。

#### 4.2.3 効率化・高速化の手法

BPEL2Java ではプログラム生成時に不必要な機能の実装を避け、必要な機能のみを実装することで、実行時リソースの減少と実行の高速化を行う。その手法の一つとして、実行スレッドの削減を取り上げる。

プロセス実行スレッドの削減 BPEL4WS はビジネスでの利用を念頭に作成された記述言語であるため、単純な手続きだけでなく、複雑なビジネスプロセスを記述することが可能である。手続き処理とビジネスプロセス処理の違いの一つとして、終了条件の違いが挙げられる。

通常の手続き処理においては、呼出元への値の返却は手続き処理の終了を意味する。一方、ビジネスプロセスにおいては、呼出元へ値を返却してもプロセスは終了せず、もし次のアクティビティが定義されていればそれを実行する。プロセスは実行する必要がある全てのアクティビティの実行が終わった時点で初めて終了することになる。このためビジネスプロセスでは非常に生存期間の長いプロセスを表現することが可能である。

このような値返却後も実行を続けるビジネスプロセスを実装するには、ビジネスプロセスの呼出を行う現在のスレッドとは別に、ビジネスプロセスを実行するためのスレッドを用意し新たに用意したスレッド上でビジネスプロセスを実行させる必要がある。

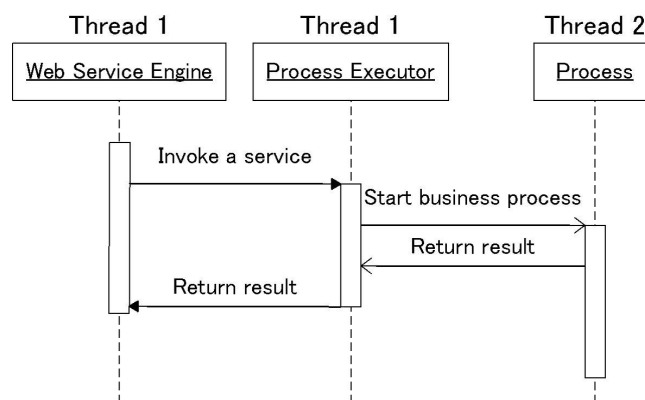


図 10: The sequence diagram of executing a long-lived process

図 10 はビジネスプロセスを実行する際のシーケンス図である．ここでは，ビジネスプロセスを起動するためのクラス *Process Executor* を新たに用意し，Web サービスエンジンはこの起動クラスを呼び出す．起動クラスはビジネスプロセスを起動実行し，ビジネスプロセスから値返却の通知が来るまで待つ．起動されたビジネスプロセスはアクティビティの処理を行い，値の返却を行うアクティビティに到達すると，実行結果の値を起動クラスに渡す．ただし，この後もビジネスプロセスクラスは終了せずに実行を続ける．起動クラスは受け取った値を Web サービスエンジンへと返却する．この場合プロセス実行のために 2 つのスレッドが実行されることになる．

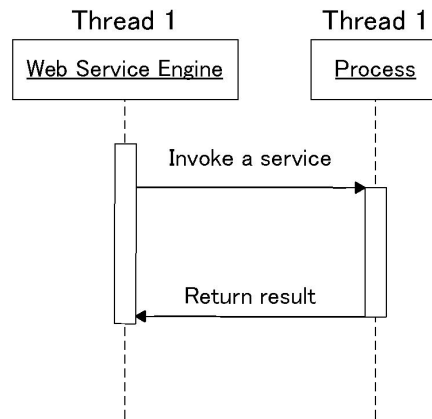


図 11: The sequence diagram of executing a short-lived process

しかし，一般的な複合 Web サービスは長時間生存し続けるようなビジネスプロセス型ではなく，入力を受け取り実行し結果を返すような手続き型である．手続き型のプロセスでは，値の返却によりプロセスの実行も終了する．そのためビジネスプロセスを別スレッドにて実行する必要がなくなる．図 11 はこのような手続き型プロセスのシーケンス図である．ビジネスプロセスが同一スレッド上の Web サービスエンジンから呼び出され値を返却するため，実行には 1 つのスレッドで十分である．

BPEL2Java では，与えられた BPEL4WS プロセスが手続き型である場合には，プロセスを同一スレッド上で実行することにより，スレッド数を減少させる．  
 Java 制御構造の利用 BPEL4WS における構造化アクティビティは，Java 言語における制御構造として実装が可能である．例えば，条件分岐を実現するアクティビティ switch は，Java 言語上では *if else* 構文に変換することができる．

繰り返しのための `while` アクティビティについても、制御構造である `while` または `for` を利用して表現することができる。

また構造化アクティビティではないが、プロセスに対する例外捕捉ハンドラも Java 言語の制御構造 `try-catch` に変換することができる。

BPEL4WS の要素を Java 言語の制御構造に変換することにより、生成されるプログラムを簡潔にしサイズを縮小することができる。またコンパイラによる最適化の恩恵を受けることができるようになる。

#### 4.2.4 実装上の課題

BPEL4WS プロセス、特に高機能なビジネスプロセスからの Java プログラム生成には、実装上の課題が存在する。

最大の課題は、プロセス実行中の外部からのメッセージ受信である。手続き型の単純な BPEL4WS プロセスでは、複合 Web サービスが外部から呼び出されるためのポートはただ 1 つのみで、そのポートに対して SOAP 呼び出しを行い、複合 Web サービスの実行結果を取得する。しかし、高機能なビジネスプロセスでは、1 つのプロセスが外部からの呼出ポートを複数持つことができる。これは、あるプロセスの実行中に外部から別のポート経由で呼び出しを行うことで、実行プロセスへメッセージを送信するためである。例えば、プロセス強制終了メッセージを受信するためのポートを用意してあれば、プロセス実行中に該当ポート経由で強制終了用の SOAP メッセージを送信することで、外部からプロセスの強制終了が可能になる。他にも、プロセス内で非同期型の Web サービス呼出を行った場合には、実行結果は実行プロセスに対する一方向 SOAP 呼出によってプロセスに返却される。

今、1 つの複合 Web サービスに対して複数の実行要求が来、複数のプロセスインスタンスが実行されている状況を考える。このとき、外部から SOAP メッセージが来た際には、その SOAP メッセージがどのプロセスインスタンスに対するメッセージなのかを判断して、適切なプロセスインスタンスにメッセージを渡す必要がある。BPEL4WS では、この対応付けのために `correlation` という仕組みが用意されている。この仕組みは Relational Database の `primary key` の概念に似ている。BPEL4WS ではプロセスインスタンスとの対応が一意に決まるように、受信するデータのフィールドの組合せを `correlation` として登録しておく。実行時には、この `correlation` をキーにしてプロセスインスタンスと受信メッセージの対応付けを行う。

このようにプロセス実行中にメッセージを受信するアクティビティを `correlated receive` と呼ぶ。 `correlated receive` を含むプロセスはスレッドを利用して実行される。通常このスレッドで `correlated receive` アクティビティが実行されると、スレッドは外部からメッセージを受信するまで待ち続けることになる。

そのため、 `correlated receive` を持つプロセスインスタンスが複数実行されている場合には、外部からのメッセージ受信を待ち続けるスレッドが複数動作することになる。これでは、プロセスインスタンスが増加するにつれて使用スレッドがどんどん増えていき、最終的にはスレッドが作成できなくなったり、メモリが不足したりし、システム全体が停止する可能性もある。

この問題に対しては、 `correlated receive` アクティビティでは、処理の待ち受けを行わずに実行中のプロセスインスタンスの情報全てをデータベースなど外部保存領域に格納することが対策として考えられる。外部からメッセージが送信されてきた場合には、これら `correlated` メッセージを処理するための Manager を別途用意しておき、その Manager が対応するプロセスインスタンスをデータベースから復元した上で、プロセスインスタンスにメッセージを渡すのである。

この方法を用いれば、 `correlated receive` を含むプロセスのインスタンスが複数実行されている場合にもシステムに対する問題は起きない。しかし、プロセスインスタンスの外部領域への保存および復元処理に処理時間がかかってしまうという重大な欠点がある。

幸い、ある Web サービスと別の Web サービスを組み合わせる新しい複合 Web サービスを作るといったような一般的利用においては、 `correlated receive` が必要となるような状況はあまり発生しないと思われる。しかし、 `correlated receive` の効率的な処理については今後の検討課題である。

## 第5章 生成されたプログラムの分散環境実行

複合 Web サービスからのプログラムの生成には、もう1つの利点が存在している。

通常の複合 Web サービスは実行するために専用の実行エンジンが必要である。しかし、生成されたプログラムを実行するためには、このようなエンジンは不要となる。そのため、複合 Web サービスはクライアントマシンなど多くの環境においても実行可能となる。

生成されたプログラムのこの特徴を利用することで、複合 Web サービスを分散してそれぞれの環境で実行することができる。これにより、多くの実行要求を集中して処理するような実行エンジンが存在しなくなる。実行エンジンに相当する複合 Web サービスプログラムは各環境に分散されるため、単一の実行環境に対して過度に多くの実行要求が集中するような事態は起こりにくい。即ち、3.2節で述べたようなボトルネックの問題が起こる可能性が限りなく小さくなる。

また前章で述べたように、プログラムは実行に必要なリソースが大きくなりすぎないように生成されている。そのため、様々な環境においてプログラムを容易に実行することが可能であると共に、高負荷時においてもより良い実行効率を發揮できる。

### 5.1 生成されたプログラムの利用形態

複合 Web サービス記述を元に生成されたプログラムの利用形態は以下の2つがある。

#### 5.1.1 Web サービスとしての利用

生成されたプログラムは、Web サービスエンジンに配備することで、Web サービスとして利用できるようになる。

生成されたプログラムは、複合 Web サービス記述と同等の動作をする単なるプログラムであり、これだけでは Web サービスではない。外部ホストから Web サービスとして SOAP 通信で利用できるようにするには、外部に対してサービスを公開するための環境が必要になる。この環境を提供するのが Web サービスエンジンであり、Web サービスエンジンに生成されたプログラムを配備することで、初めて Web サービスとして外部から利用できるようになる。また、これにより、別の複合 Web サービスなどからコンポーネント Web サービスとし

て利用することが可能となる。

複合 Web サービスを外部へ公開するためには Web サービスエンジンが必要であるならば、結局 Web サービスエンジン上に配備された複合 Web サービスプログラムに実行要求が集中し、専用の実行エンジンで複合 Web サービスを実行することと大差が無いのではないかと思われるかもしれない。

しかし、Web サービスエンジンで複合 Web サービスプログラムを実行する場合と、複合 Web サービスの実行エンジンで複合 Web サービス記述を実行する場合は、以下の点で異なっており、プログラムの生成という手法はボトルネックの解消に有効である。

- Web サービスエンジンは複合 Web サービスエンジンに比べて、非常に広く利用されている。また、Web サービスエンジンは複合 Web サービスエンジンに比べて、導入が容易である。このため、複合 Web サービスプログラムを多くの環境に分散して配備することが可能となり、処理要求が集中する可能性を抑えられる。
- 生成されたプログラムを Web サービスエンジン上で実行する場合の方が、複合 Web サービス実行エンジンで実行する場合よりも、実行に必要なリソースが少ない。そのため、高負荷時にボトルネックとなりにくい。

#### 5.1.2 ライブラリとしての利用

生成されたプログラムの2つ目の利用形態は、生成されたプログラムを他のプログラムからライブラリとして利用することである。例えば、あるアプリケーションで複合 Web サービスを利用したい場合には、生成されたプログラムの API を通して直接実行することができる。この方法には、Web サービスとして利用する方法にはない大きな利点が存在している。

複合 Web サービスプログラムをライブラリとして利用する場合には、利用元のアプリケーションと利用されるライブラリの間は、そのプログラム言語の関数呼出として実行される。そのため、時間のかかる通信処理や、XML とデータの変換処理を省略することができる、複合 Web サービス全体の実行速度短縮に大きく貢献する。これは、複合 Web サービスプログラムをライブラリとして利用することの非常に大きな利点である。

しかし、残念ながら一部の複合 Web サービスはライブラリとしては利用できない。

複合 Web サービスの中には、サービスの実行中に外部ホストからの SOAP 呼

出を受け付けるものが存在する。外部ホストからの呼出を受けるには、複合 Web サービス自体が外部に公開されている状態でなくてはならない。そのため、外部からの呼出を受け付ける場合にはライブラリとしては利用できず、Web サービスとして配備し利用する必要がある。

複合 Web サービスをライブラリとして利用できないのは、以下のような場合である。

- 非同期型の Web サービス呼出を行う場合

Web サービスの呼出には以下の 2 つのタイプが存在している。

**request/response** Web サービス要求を出した際にそれと同じコネクションで Web サービス応答が帰ってくるもの

**one-way** Web サービス要求を出しても際に、Web サービス応答が帰ってこないもの

一般に Web サービスは request/response 型で呼び出す場合が多い。しかし、ビジネス分野などで処理完了に時間のかかる Web サービスに対しては、one-way 型の Web サービス呼出を行い、サービスの処理が完了したら、サービスから自分に対しサービス実行結果を one-way 型の Web サービス呼出でコールバックしてもらう場合がある。このように one-way 型を組み合わせた Web サービス呼び出しは、非同期型の Web サービス呼出と呼ばれている。

非同期型の Web サービス呼出では、実行結果をコールバックしてもらうため、サービスが外部から呼出可能である必要がある。

- 外部からのイベントを捕捉する場合

イベントハンドラーなどで、外部からの Web サービス呼出をイベントとして捕捉するためには、外部からの呼出が必要である。

- 複数のホストからの Web サービス呼出が行われて初めて、処理を開始する場合

一般の Web サービスは最初に呼出が行われた時点で処理を開始するが、Multiple Start と呼ばれる種類の複合 Web サービスは、異なるいくつかの呼び出しが全て行われた時点で処理を開始するものがある。このような複合 Web サービスは一般に複数のホスト上から呼び出される必要がある。

ただし、ここで挙げられたようなケースは、複合 Web サービスをビジネスにおいて活用する際に必要となる機能であり、一般的な利用においては多くの場

合，生成されたプログラムをライブラリとして利用できる。

## 5.2 生成されたプログラムの分散環境実行例

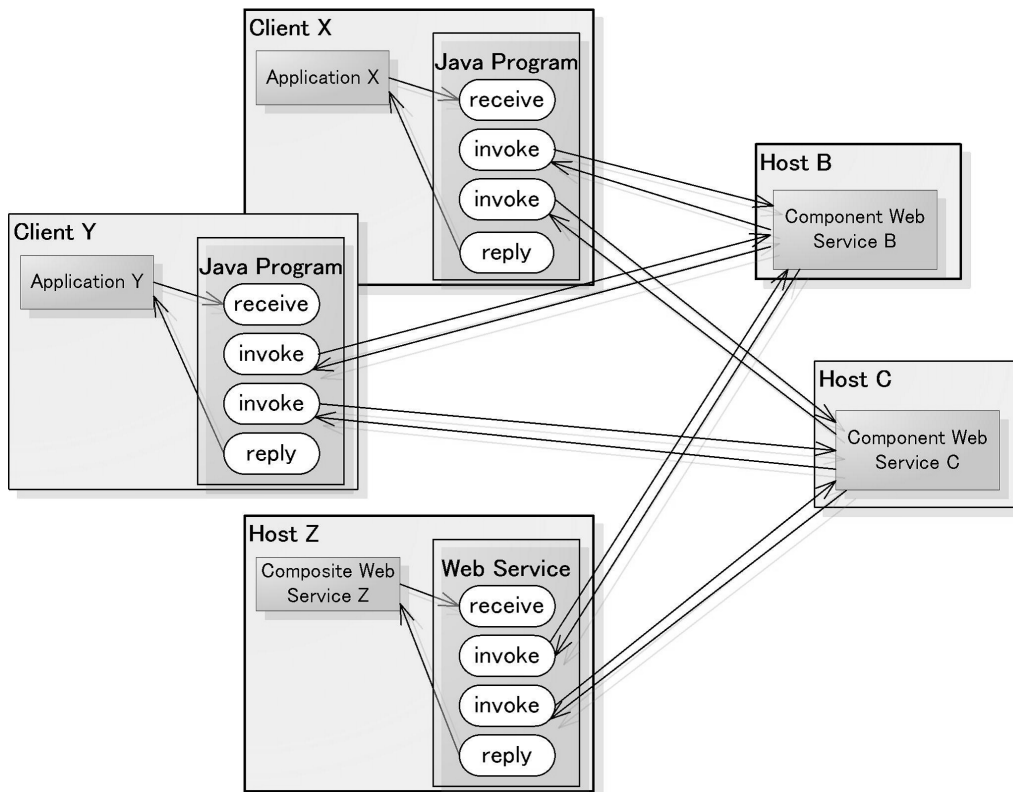


図 12: Client-side execution

図 12 は，このような分散環境でのプログラムの実行を表した図である．図ではクライアント X, クライアント Y, ホスト Z の 3 台のマシンの上で，生成されたプログラムが実行される．ホスト B とホスト C は生成されたプログラムでコンポーネントとして呼び出される Web サービス B と Web サービス C を公開している．クライアント X ではアプリケーション X が動作し，その内部で複合 Web サービスプログラムがライブラリとして呼び出される．クライアント Y でも同様にアプリケーション Y が複合 Web サービスプログラムをライブラリとして利用している．一方ホスト Z においては，実行エンジン上で別の複合 Web サービス Z が動作しており，ホスト Z 上に Web サービスとして配備された，複合 Web サービスプログラムを SOAP 経由で呼び出すことになる．なお，図中の矢印は制御とデータのフローを，図中の丸は複合 Web サービス記述にお

けるアクティビティを表している。ただし、図中では各アクティビティは Java プログラムとして生成されているため、厳密にはアクティビティに相当するものといえる。

Centralized Orchestration においては、複合 Web サービスの実行エンジンが動作するホストが存在し、コンポーネント Web サービスの実行は全てそこで処理されていた (図 3)。Decentralized Orchestration においては、複数の実行エンジンを用意し、1 つの複合 Web サービス記述を分割して分散実行することでボトルネックの緩和を図っている。しかし、やはりクライアントとコンポーネント Web サービスの間に位置し、ハブとして機能するようなホストが存在し、これがボトルネックとなる可能性があった (図 4)。

一方、図 12 ではハブとして機能するホストは存在せず、ボトルネックが発生しにくいことがわかる。

## 第6章 Web サービスキャッシュ

ここでは Web サービスの実行結果をキャッシュすることを考える。データ検索や機械翻訳などに代表されるような Web サービスはキャッシュを有効に活用することでその実行時間を短縮することができる。

キャッシュについては既に多くの分野で研究がなされている。データベースの分野においては、低速なディスクアクセス処理を避けるため頻繁に利用されるページをメモリ上にキャッシュすることが一般的である [9, 10]。また World Wide Web 上においても、Web プロキシサーバやサーチエンジンなどの分野で Web ページのキャッシュが利用されている [11, 12, 13]。キャッシュを利用することで、サービスのボトルネックをなくし、ネットワークの通信データ量を軽減し、結果的に利用者のアクセス遅延を最小限にとどめることができる。本章では、Web サービスの持つ特徴を考慮し、Web サービスのキャッシュアルゴリズムに求められる要件について考察を行う。

### 6.1 対象環境

#### 6.1.1 キャッシュ空間

本研究では、複数の Web サービス呼出を単一のキャッシュテーブルを用いて統一的にキャッシュする方法について考察するものとする。

キャッシュを行いたい Web サービスについて事前に十分な情報が得られているならば、そのサービスに特化したより効率的なキャッシュを作成することができる。しかし、本稿で対象とするような複合 Web サービス環境を考えると、このような特定の Web サービスに特化したキャッシュ機構を Web サービスごとに個別に管理することは現実的ではない。なぜなら、複合サービス実行エンジン上で実行される Web サービスは非常に多岐に渡り、各 Web サービスについてあらかじめ十分な情報を取得することは難しいからである。また、Web サービス毎にキャッシュ空間を個別に持つと局所的に特定の Web サービスが頻繁に呼出された場合などには全体からみれば一部のキャッシュ空間しか利用されていないことになり、効率が悪いという問題も存在する。このため複数の Web サービス呼出を単一のキャッシュ空間を用いて効率的にキャッシュすることを考える。

### 6.1.2 キャッシュ機構の設置箇所

Web サービスの呼び出しは通常，ネットワーク上のあるホストから別のホストへのリモート呼出である．Web サービス呼出時のデータフローと送受信されるデータを表現したのが図 13 である．

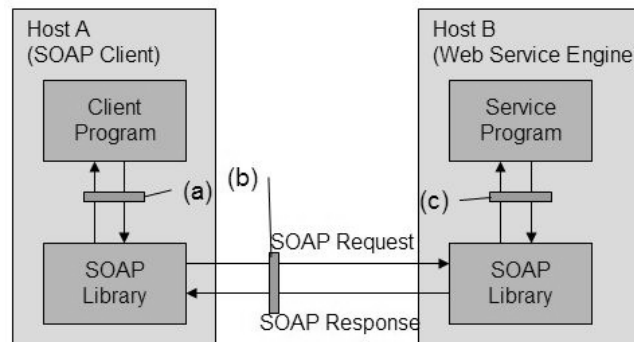


図 13: Data flow of SOAP Request/Response

Web サービスの呼び出しではまず，呼出元プログラムから Web サービス呼出に必要な情報がプログラミング言語の変数に格納されて SOAP ライブラリへと渡される．SOAP ライブラリは情報を XML に変換し，それを呼出先ホストの SOAP ライブラリへと HTTP などのインターネットプロトコルを利用して送信する．呼出先ホストの SOAP ライブラリは受け取った XML データを該当プログラミング言語で扱える情報へと変換する．得られた情報を元にサービスプログラムが実行され，結果が今までと逆の手順で呼出元プログラムへと返却される．

プロキシとしてキャッシュ機構を設置可能なポイントとしては，図中の (a)，(b)，(c) の 3 つが挙げられる．

- (a) は呼出元プログラムと SOAP ライブラリの間にはキャッシュ機構を設定する方法である．利点は，キャッシュがヒットした場合には，ネットワーク上の通信処理 XML データへの変換処理の両者を省くことができるため，実行時間短縮に対して非常に効果的なことである．一方欠点は，これを實現するには既存クライアントプログラムに修正を加える必要があることである．
- (b) は呼出元ホストと呼出先ホストの間に設置する方法である．利点としては複数の呼出元ホストが単一のプロキシサーバを利用することで，より

よいキャッシュヒット率を得られる可能性があることや、キャッシュ専用のマシンを設置することが可能なため大規模なキャッシュ空間を提供できることである。欠点としては、XML データは一致の検証が難しいことがあげられる。一般に XML においては、要素内容や要素内属性間の空白文字が無視されることなどを元に、同一のモデルを表現しているが文字列としてはまったく異なるデータを無限に生成することができる。このことが2つの XML データの表現しているモデルが同一のものかどうか、簡単に判断できなくしている。

- (c) は呼出先ホストにおいて SOAP ライブラリとサービスプログラムの間  
に設置する方法である。この方法ではサービス実行時間のみが短縮される  
ことになり、Web サービスのキャッシュとしてはあまり有効ではない。

これらより、キャッシュの効用を最大化するという点で考えれば (a) の方法が最も優れているといえる。特に、BPEL4WS 記述からプログラムを生成する場合には、キャッシュ機構を組み入れたプログラムを生成することが可能で、(a) の方法の持つクライアントプログラムへの修正が必要という欠点を無視できる。

また、生成されたプログラムをクライアントマシンにおいて実行することで、データの局所性が高まり、キャッシュの恩恵が高まると期待される。ただし、クライアントマシンはサーバと比べメモリ容量なども小さいため、大規模なキャッシュ空間を用意できないという問題点もある。

### 6.1.3 キャッシュ可否の判断

Web サービスにはキャッシュ可能なものと、キャッシュが不可能なものがある。キャッシュが可能な Web サービスとは、同じ引数を用いたサービス呼出に対しては常に同じ結果を返すような Web サービスである。例えば、ある単語についての意味を返す辞書検索 Web サービスや、入力文を他の言語の文へと翻訳する機械翻訳 Web サービスが挙げられる。一方、キャッシュが不可能な Web サービスとは、同じ引数を用いて呼出を行っても、そのときの環境やサービスの内部状態により返却される値が異なるものがある。例えば、ある場所の明日の天気を返すような天気予報 Web サービスや、ホテルの予約状況を返すような Web サービスがこれにあたる。

通常の SOAP 通信で送受信される情報のみから、ある Web サービスがキャッシュ可能かどうかを判断することは非常に難しい。ある Web サービスがキャッシュ可能かどうかを判断するには、そのサービスの実行結果にサービスの内部

状態や外部環境が関係するのかわかる必要がある。つまり、サービスの内部ロジックを把握する必要があるのである。このため、ある Web サービスがキャッシュ可能かどうかを正しく判断できるのは、そのサービスを提供しているユーザしかいないといえる。

本研究では、ある Web サービスがキャッシュ可能かどうかについては予めユーザが判断し、システムに知らせておくこととする。

具体的には、コンポーネント Web サービスがキャッシュ可能かどうかを WSDL の拡張要素を用いて記述する。複合サービス記述においては、コンポーネント Web サービスに関する情報は WSDL ファイルにおいて表現されている。WSDL 仕様では、今回のような外部情報を Web サービスに付加するために拡張要素を記述することが可能となっている。この仕組みを利用して、システムにキャッシュ可能かどうかの情報を Web サービスの持つ情報と共に通知する。

## 6.2 Web サービスキャッシュの特徴

キャッシュの対象として Web サービスを見た場合、Web サービスには次のような特徴がある。

キャッシュのキー キッシュを実現するためには、データに対して一意となるキャッシュのキーを定める必要がある。Web サービスキャッシュにおけるキャッシュのキーは、既存のキャッシュと比べて随分複雑である。

既存のキャッシュ分野では、一意性を持ちキャッシュのキーとなる単純なデータが存在している。例えば Web プロキシにおけるキャッシュでは URL がキーとして利用される。また、ディスクの IO 処理ではノード ID がキーとして利用される。

一方、複数の Web サービスが単一のキャッシュ空間を利用する場合には、キャッシュのキーが一意性を確保するためには、どの Web サービスをどういう引数の値で呼び出したかという情報を含む必要がある。

対象となる Web サービスを一意に決めるには、Web サービスの呼出先の URL、Web サービスのポート名、オペレーション名、呼出時の各引数の型、返り値の型の情報が揃う必要がある。さらにキャッシュのキーとして利用するには、この情報に各引数の値を組み合わせる必要がある。今回は単純に、これらの情報の文字列表現を連結することで一意性を保証するキー文字列を利用する。

サービス実行時間 複合 Web サービス内で利用されるコンポーネント Web サービスの複雑さや粒度は一定ではない。例えば、単純な原子 Web サービスをコンポーネントとして利用している場合もあれば、複雑な複合 Web サービスをコンポーネント Web サービスとして利用している場合もある。そのため、コンポーネント Web サービスの実行にかかる時間も Web サービスにより様々である。さらに、ネットワーク通信などの不確定要素も実行時間には影響してくる。

### 6.2.1 Web サービスの局所性

キャッシュを行う上で重要になるのはその局所性である。局所性が高ければ、キャッシュのヒット率が高まり、キャッシュを利用する効果が高くなる。

Web サービスの利用に関する研究はこれから発展していく分野であり、利用時の特性などについてのデータはあまり多くない。その中で、Heらは内部の大規模データベースから情報を取得して返すような Web サービスにおいて、空間的局所性が見られることを報告している [14]。ここでの空間的局所性とは、データベースの特定の領域が他の領域と比べて頻繁なアクセスを受けることを指している。この研究ではアクセスの多い2つの Web サービス(宇宙空間内の任意の座標に関する情報を取得できる Web サービスと、アメリカ合衆国内の任意の緯度経度の衛星写真画像を取得できる Web サービス)を対象に、そのアクセスログからデータアクセスの局所性を検証している。前者のサービスでは84.04%のリクエストが全領域中30%の部分にアクセスしており、また後者のサービスでは99.94%のアクセスが10%の領域に集中している。

この研究を元にすれば、データベースからのデータ検索サービスでは呼出時の引数の値には空間的局所性があると推測されるが、しかしこれを一般の Web サービスにそのまま適用することは難しい。一般的に Web サービスの引数の値は、とりうる値の範囲が広い。このため、上記の研究のように大量の実行要求がある状況ではデータの空間的局所性が表れるが、実行要求が多くない場合には引数の値は分散し、十分なデータ局所性が表れない可能性が高いと推測される。また、たとえ多くの実行要求が存在したとしても、キャッシュ空間が分散したデータを格納するのに十分な大きさが無ければ、データの空間的局所性を有効に利用することはできない。

一方、Web サービスを単なるサービス呼出と捉えると、呼出引数の値には時間的局所性が存在していると直感的に考えられる。直接的には関連しないが、Web におけるサーチエンジン利用における局所性の研究がある [15]。

## 6.3 キャッシュアルゴリズムの要件

Web サービスのキャッシュアルゴリズムが満たすべき要件の1つは、局所性の有効利用についてである。前節にて述べたように、一般的な Web サービスではキャッシュ効率を大きく向上させるような空間的局所性を得られる可能性は低いといえる。そのため、空間的局所性よりも時間的局所性に注目したほうが良いと考える。

一般的によく使われるキャッシュアルゴリズムとして LRU(Least Recently Used) と LFU(Least Frequently Used) がある。キャッシュのアルゴリズムは多く存在しているが、ほとんどはこの2つのいずれかの考え方を元としている。2つの違いは、キャッシュ空間が一杯になりさらに要素を追加する必要がある際に、取り除く要素を決めるアルゴリズムの違いである。LRU では、最も長い間利用されていない要素が取り除かれるが、LFU では最も利用回数が少ない要素が取り除かれる。即ち、LRU は時間的局所性を持つデータに対して有効であり、LFU は空間的局所性を持つデータに対して有効である。Web サービスのキャッシュでは時間的局所性を利用した LRU タイプのアルゴリズムがより適している。

また、2つ目の要件は、サービスの実行時間のバラつきをうまく考慮に入れることである。例えば A と B という2つの Web サービスがあり、A の実行には平均1秒、B の実行には平均5秒かかるとする。A と B が同じ頻度で呼び出されているのならば、B の方を優先してキャッシュした方が良いことは明らかである。次に、A が B の2倍の頻度で呼ばれている状況だとする。この場合 A を優先してキャッシュした方がキャッシュのヒット率は高くなるが、B がキャッシュにヒットした場合にはより多くの実行時間を短縮できることを考えると B を優先してキャッシュした方が良い。このように、Web サービスのキャッシュアルゴリズムは実行時間のバラつきまでを考慮に入れる必要がある。

### 6.3.1 LNC-R アルゴリズム

LRU アルゴリズムの派生の1つに LNC-R アルゴリズムがある [16, 17]。このアルゴリズムは、キャッシュの利用による効用を最大化することを目指したものである。ここでの効用は、キャッシュヒットにより削減される実行時間のキャッシュシステム全体での合計である。

キャッシュ空間に格納されている要素  $E_i$  があるとする。つまり  $E_i$  は、ある Web サービスに対してある引数値を用いて Web サービス呼出を行った際の、実

行結果を表している．このとき，LNC-R アルゴリズムでは  $E_i$  の効用関数を

$$profit(E_i) = (l_i * d_i) / s_i$$

と定義する．ただし， $l_i$  は要素  $E_i$  を返す Web サービス呼出の平均実行率， $d_i$  は要素  $E_i$  を返す Web サービス呼出の平均応答時間， $s_i$  は要素  $E_i$  のサイズである．また， $l_i$  をある定数  $K (K \geq 1)$  を使って，

$$l_i = K / (t - t_K)$$

と定義する．ただし， $t$  は現在の時刻， $t_K$  は現在から  $K$  回前の  $E_i$  を返す Web サービス呼出が行われた際の時刻である．つまり， $l_i$  は直近  $K$  回の呼出を基にした平均呼出率をあらわしている．効用関数は，ある Web サービスに対するある引数値を使った呼出が，最近頻繁に実行されていればいるほど，呼出に時間のかかる処理であればあるほど，また結果データのサイズが小さければ小さいほど，その要素  $E_i$  をキャッシュしておく効用が高いということを表している．

LNC-R アルゴリズムでは，キャッシュから要素を取り除く必要がある場合には，この効用関数の値が最小となる要素を取り除く．効用関数の式に  $d_i$  を導入することで Web サービスの実行時間のバラつきを考慮し，また直近  $K$  回の呼出率  $l_i$  を導入することで，時間的局所性を重視してあるといえる．このため，LNC-R アルゴリズムは Web サービスのキャッシュに適しているといえる．

Web サービスの利用はこれから増加していく．現在のところ，一般的な様々な Web サービスに対する利用ログデータや研究があまり存在しない．そのため，一般的な Web サービス利用に対してこのアルゴリズムが十分有効であるかどうかを検証することは非常に難しい．今後，Web サービス利用に関するデータや研究が増えることで，Web サービスキャッシュについて更なる提案が可能になると考える．

## 第7章 評価

本章では，第3章にて述べた3つの問題に対して，本研究の提案手法がどれだけ有効であるか，それぞれの場合に分けて検証を行う．

### 7.1 評価実験 1:生成プログラムによる実行の高速化

#### 7.1.1 設定

ここでは，複合 Web サービス実行エンジン上で複合 Web サービスを実行した場合と，BPEL2Java によって作成された Java プログラムを通常の Web サービスエンジン上に配備して実行した場合について，平均処理時間の比較を行う．BPEL2Java の導入による直接的な高速化効果を検証する．

評価実験の対象として次の2つのプロセスを用意した．

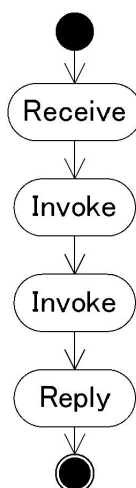


図 14: Sequential Process

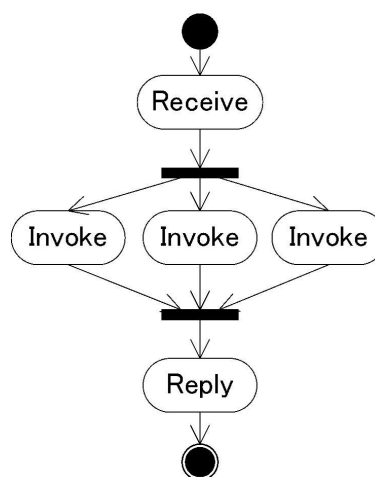


図 15: Concurrent Process

図 14 はメッセージを受け取り，2つの外部サービス呼出を順番に行い，結果を返却する，Sequential な複合 Web サービスである．この形は例えば，日英翻訳 Web サービスと英中翻訳 Web サービスを利用して，日中翻訳 Web サービスを作成するといった場合を想定している．一方，図 15 はメッセージを受け取り，3つの外部サービス呼出を並行に行い，結果を返却する Concurrent 複合 Web サービスである．この形は例えば，日本語を受け取り，日英翻訳 Web サービス，日中翻訳 Web サービス，日韓翻訳 Web サービスを利用し，日本語から英中韓の3ヶ国語への多言語翻訳 Web サービスを作成するといった場合を想定している．

2つのプロセスに対してそれぞれ BPEL2Java による Java プログラム版と、BPEL4WS エンジン版を用意した。それぞれの複合 Web サービスをクライアントから呼出、その処理にかかる平均時間を取得した。

各プロセス内で利用されるコンポーネント Web サービスは、呼び出されると 500ms だけ待ち結果を返す専用のスリープサービスを作成し、両方のプロセスともにこれを使用した。ネットワーク状態など不確定要因をできるだけ排除するため、クライアント、BPEL4WS エンジンあるいは Web サービス、プロセスで利用されるコンポーネント Web サービスは全て同一のマシン上に設置し、SOAP プロトコルで通信をおこなった。

BPEL4WS エンジンとしては、ActiveEndpoints 社が提供するオープンソースの BPEL4WS エンジン ActiveBPEL2.0 を利用した。Web サービスの実行エンジンには、広く利用されている Apache Axis1.3 を利用した。また、HTTP サーバとして Apache Tomcat5.5 を用意し、この環境上で BPEL4WS エンジンおよび Web サービスエンジンを動作させた。

#### 7.1.2 評価結果

表 2 は 2 つのプロセスを、BPEL4WS エンジン上で動かした場合と、Java プログラムとして動作させた場合の、平均処理時間を示した表である。BPEL4WS と書かれているのが BPEL4WS エンジン上で動かした場合で、Java と書かれているのが BPEL2Java により生成した Java プログラムを Web サービスエンジン上で動かした場合である。どちらのプロセスにおいても、BPEL2Java を利用して生成したプログラムの方が高速に動作していることが分かる。

	BPEL4WS	Java
Sequential Process	1071 ms	1034 ms
Concurrent Process	566 ms	542 ms

表 2: The time to execute a process

この表の値には、コンポーネント Web サービスの処理時間も含まれている。そのため、BPEL2Java による高速化効果がどれくらいあるのかを見るため、複合 Web サービス全体の処理時間のなかの BPEL4WS プロセス自体の処理にかかっている時間を調べる。そこで、コンポーネント Web サービス呼出に必要な

時間の計測をおこなったところ、これには平均して 512ms かかることが分かった。コンポーネントサービス単体での実行は 500ms であるので、これとの差の 12ms が SOAP 通信処理にかかる時間であると推測できる。

Sequential Process においては、コンポーネント Web サービスを順番に 2 つ実行しているため、この値を 2 倍した約 1024ms がコンポーネント Web サービス呼出にかかった時間だといえる。Concurrent Process においては、コンポーネント Web サービスが並行で実行されるため、おおよそ 512ms がコンポーネント Web サービスの呼出にかかっていると考えられる。全体の平均処理時間から、コンポーネント Web サービス呼出に必要な時間を引いた時間、つまり、複合 Web サービスに対する SOAP 通信時間とプロセス自体の実行に必要な時間の和をグラフにしたのが図 16 である。グラフの縦軸は実行に必要な時間 (ms) を表している。

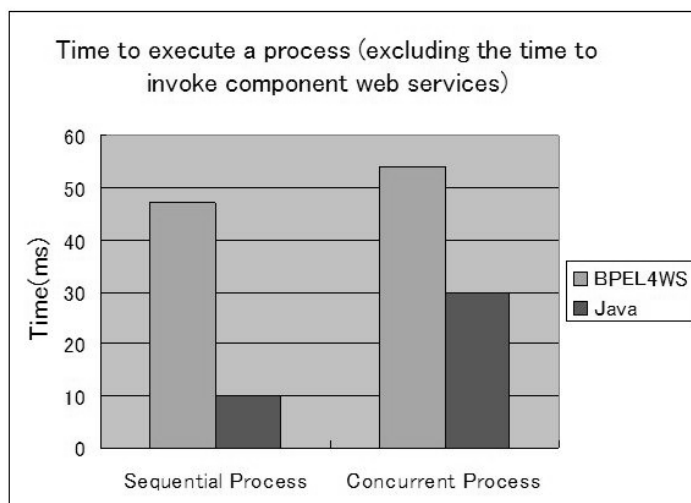


図 16: Time to execute a process excluding the time to invoke component web services

どちらのプロセスにおいても、かなりの割合で処理時間が削減されていることが分かる。特に、SOAP 通信にかかる時間おおよそ 10ms を引いて考えると、いずれの場合も 50%以上処理時間が削減されているといえる。しかし、残念ながらプロセス部分の処理時間は複合 Web サービス全体の処理時間に比べてかなり小さいため、全体の処理時間としてみた際にはそれほど高速化されているとは言い難い。

## 7.2 評価実験 2:生成プログラムによる実行の効率化

### 7.2.1 設定

生成されたプログラムの実行と BPEL4WS エンジンでの実行について、負荷が高い状況での実行効率とボトルネックの発生状況を確認する。多数のリクエストが集中する状況において、単位時間当たりのリクエスト処理件数と平均処理時間を計測しプログラムの生成が効率的な実行に寄与しているかどうかを確認する。

負荷が高い状況として多国語翻訳を利用したチャットアプリケーションを利用する状況を想定する。利用する複合 Web サービスとして、入力を受け取ると、5つの機械翻訳サービスを並行で実行し、その結果をまとめて返すという多国語翻訳複合サービスを考える。多国語間チャットではこの多国語翻訳サービスが頻繁に呼ばれる。とくに、ユーザの文字入力に反応しリアルタイムに翻訳を行う場合などには、1秒間に何度もリクエストが発生する。

ここでは、多国語翻訳サービスに対応するものとして、前回の図 15 のプロセスの Web サービス呼出部分の並列度を 5 に増加させたものを利用する。コンポーネント Web サービス前回と同様に 500ms だけ待ち結果を返すスリープサービスを利用する。

評価実験には図 17 のように 3 台のマシンを用意した。コンポーネントとして利用される Web サービスを提供するマシン、複合 Web サービスを実行するために BPEL4WS エンジンと Web サービスエンジンを動作させるマシン、複合 Web サービスの実行要求を出すためのクライアントマシンである。コンポーネント Web サービスを提供するホストと複合 Web サービスを実行するホストは同一の LAN 上におき、クライアントマシンは外部ネットワークから接続する。

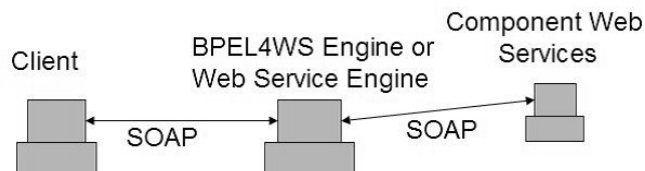


図 17: Settings for Experiment2

この環境でクライアントマシンでスレッドを作成し、複合 Web サービスの実行要求を順次発行していく。クライアントマシン上でのスレッドの数を 1-60 ま

での間で変化させ、処理効率と平均処理時間を計測するとともに、ボトルネックの発生について確かめる。

### 7.2.2 評価結果

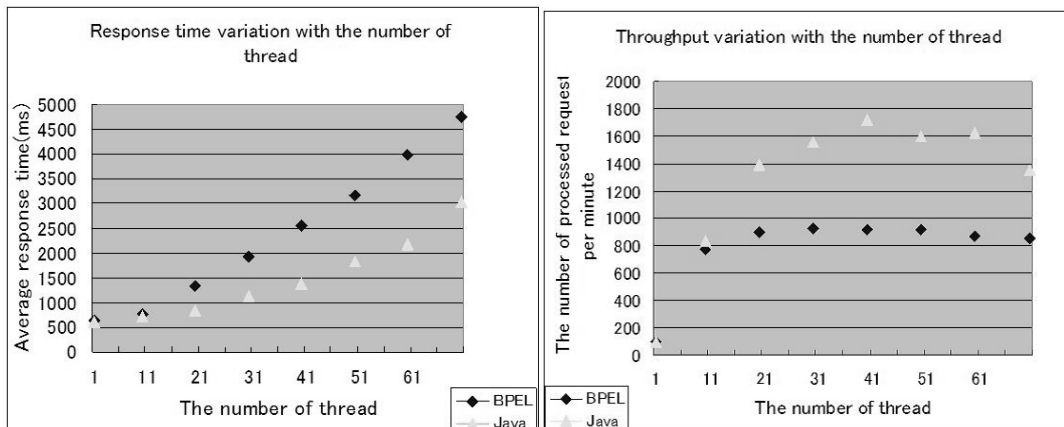


図 18: Average response time variation with the number of thread 図 19: Throughput variation with the number of thread

図 7.2.2 はクライアントの並列スレッド数を増加させていったときの複合 Web サービスの平均応答時間を表したグラフである。横軸がクライアントの並列度、縦軸が平均応答時間となっている。BPEL と書かれているものが BPEL4WS 記述を実行エンジン上で実行した際の値で、Java と書かれているものが同様の BPEL4WS 記述から生成した Java プログラムを Web サービスエンジンで実行した際の値である。

並列度が 1 の場合には、BPEL 版と Java 版で平均応答時間にグラフに表れるような大きな差はない。これは前節の結果の通りである。

一方、並列度が 20 ぐらいから BPEL 版と Java 版で平均応答時間に差が開き始めている。だが並列度が増しても、BPEL 版 Java 版共、平均応答時間はほぼまっすぐに増加している。これより、BPEL 版と Java 版の差は両者の実行効率の違いが並列度が高くなるにつれて顕著化したと考えることができる。これは、次の図 7.2.2 を見ることでよりはっきりする。

図 7.2.2 はクライアントの並列スレッド数を増加させていったときの一分間あたりの処理件数を表したグラフである。横軸がクライアントの並列度、縦軸が処理件数となっている。図 7.2.2 では、並列度が小さい時には BPEL 版の実行

も Java 版の実行も処理件数に違いは無い。これは実行要求が少ない場合には、CPU の処理能力もメモリ容量も十分余っており差が発生する要因がないからである。逆に並列度が 20 近くから、BPEL 版と Java 版の処理件数に大きな違いが出ている。特に BPEL 版は 900 件/分近くで処理件数が停滞してしまっており、高負荷時の処理ボトルネックが発生していることが確認できる。一方、Java 版でも 1600 件/分で処理件数が停滞している。

並列度が 20 以上で、処理件数が停滞している状況において、実行時の CPU 使用率とメモリ使用率を調べたところ、CPU 使用率は BPEL 版、Java 版とも 99% であった。また、並列度が 10 の場合の CPU 使用率を調べたところ、Java 版では 50-60% で終始推移していたが、BPEL 版では 99% であった。いずれの場合にもメモリ使用率には十分な余裕があった。このことから、Java 版では BPEL 版に比べて複合 Web サービスの実行処理が軽い、即ちプロセス実行に必要な CPU リソースが少ないということが出来る。この実行効率の違いは、利用スレッドの削減や、プログラムがコンパイルされていることに起因していると考えられる。

## 7.3 評価実験 3: プログラムの分散環境実行

### 7.3.1 設定

BPEL2Java により生成されたプログラムを複数のクライアントマシン上で動作させることで、BPEL4WS エンジンの持つボトルネックを回避し、より効率的な実行が可能かどうかを調べる。

ここでは、評価実験 2 と同様の複合 Web サービスを利用して実験を行う。評価実験 2 では、クライアントマシンが 1 台のみで、実行要求を複合 Web サービス実行マシンに送り実行していた。一方、今回はクライアントマシンを 2 台用意し、そのクライアントマシン上で直接複合 Web サービスプログラムを実行させる。クライアントマシンは 1 台を今までどおり外部ネットワークへ、また新しい一台はコンポーネント Web サービスと同一 LAN 内におく。今回は BPEL4WS エンジンを実行していたマシンは使用しない。

この評価実験では、2 つのクライアントでスレッドを複数作成し、2 台のクライアントマシンで同時に複合 Web サービスを実行させる。動作スレッドの数を順次上げていき、ボトルネックの発生の有無や単位時間当たりの処理件数、平均応答時間を計測する。ただし、今回はクライアントマシンが 2 台あるため、スレッドの並列度は各マシンの動作スレッドの合計で表すものとする。また、処

理件数や平均応答時間は 2 台の結果を平均する。

### 7.3.2 評価結果

図 20 は、2 台のクライアントマシン上で複合 Web サービスを動作させた場合の、各並列度に対応した処理件数である。グラフの並列度は各クライアントマシンの並列度を合計してある。即ち、グラフで並列度が 10 とは、2 台のクライアントマシンをそれぞれ並列度 5 で動作させていることを表している。複合 Web サービスを実行するプロセスが評価実験全体では何個並列に動いているかを表していると考えればよい。

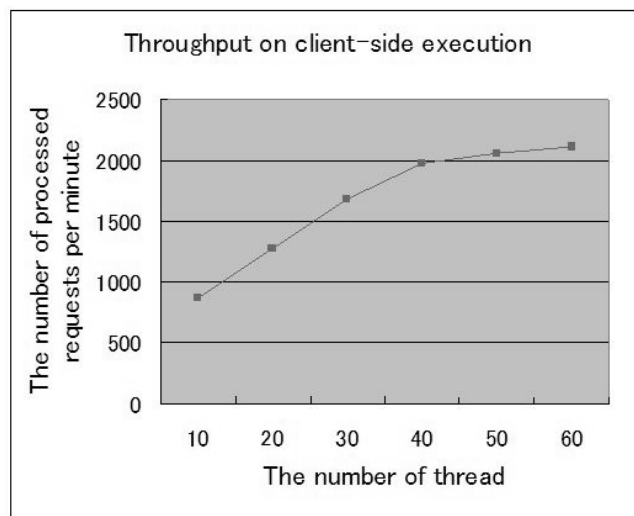


図 20: Throughput on client-side execution

図 20 を図 7.2.2 と比較すると、クライアント上で分散して複合 Web サービスを実行した際のスループットが、サーバ上で複合 Web サービスを実行した際よりも上回っていることが分かる。特に、サーバ上での実行では 1600 件/分で上限に達していたが、分散実行では 2000 件/分を越えている。今まで 1 台で動作させていたものを 2 台で動作させたことになるため、理論上は 1600 件/分の 2 倍まで処理可能であるが、図 20 では 2000 件/分を越えたあたりで処理件数の増加が鈍化している。この並列度が 60 で実行中に、コンポーネント Web サービスの CPU 使用率を調べるたところ 90%を越えていることを確認した。これは、複合 Web サービス部分の処理ボトルネックが解消されたため、次は多数のリクエストを受け取るコンポーネント Web サービス部分がボトルネックとなっていると考えられる。

## 第8章 結論

本研究では，複合 Web サービスを効率よく実行するために，複合 Web サービス記述からプログラムを生成することを提案した．また BPEL4WS から Java プログラムを生成する BPEL2Java を開発し，評価を行った．評価の結果より，次のことが確認された．

- 生成されたプログラムは，プロセスの実行処理部について処理時間の短縮が観察されたが，これは複合 Web サービス全体の処理時間に比べると僅かである
- 生成されたプログラムは，高負荷時において通常の BPEL4WS 実行よりも単位時間あたりの処理件数が 150% 向上した
- 生成されたプログラムを分散して実行させることにより，複合 Web サービス実行エンジンの持つボトルネックを解消できる．

また，一般的な Web サービスに対するキャッシュについて，アルゴリズムが考慮に入れるべき要件について考察した．

本研究の貢献は，以下の 3 点である．

- 複合 Web サービス記述からプログラムを生成する手法を提案した
- この手法が，高負荷時において複合 Web サービスの処理能力を向上することを確認した
- 生成されたプログラムを分散実行させることで，複合 Web サービスエンジンの持つボトルネックの問題を解決した

本研究では，簡単な複合 Web サービスの場合は，複合 Web サービス記述からプログラムを生成する方法が処理能力の向上や，ボトルネックの解消に有効であることを確認した．今後複合 Web サービスがさらに普及すれば，このような簡単な複合 Web サービス記述がますます増加すると考える．そのためにもさらなる最適化は今後の課題である．さらに，複雑な複合 Web サービス記述からの高機能な実行プログラムの生成も技術的に興味を引かれる課題である．今後は，これらの課題を中心に BPEL2Java の開発をおこなっていく．

## 謝辞

本研究の機会を与えて下さり、熱心に御指導を賜りました石田 亨教授に深甚の謝意を表します。本研究を進めるに当たり多大なご協力をいただいた村上陽平氏をはじめとする石田研究室の各位に感謝いたします。また、BPEL2Javaプログラムの開発においては、独立行政法人情報通信研究機構けいはんな情報通信融合研究センター言語グリッドプロジェクトの皆様に変にお世話になりました。この場を借りて御礼申し上げます。

## 参考文献

- [1] World Wide Web Conference: *Simple Object Access Protocol 1.2* (2003). <http://www.w3.org/TR/soap/>.
- [2] World Wide Web Conference: *Web Services Description Language 1.1* (2001). <http://www.w3.org/TR/wsdl.html>.
- [3] BEA Systems, International Business Machines Corporation, Microsoft Corporation, SAP AG, Siebel Systems: *Business Process Execution Language for Web Services Version 1.1*. <http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/>.
- [4] MATSUYAMA, N. and OBA, M.: Experiment of Compound Web Services Using Traffic Information Web Service, Technical Report IPSJ-DD05051006, IPSJ (2005).
- [5] Ishida, T.: Language Grid: An Infrastructure for Intercultural Collaboration, *2006 International Symposium on Applications and the Internet (SAINT 2006)*, IEEE Computer Society (2006).
- [6] Peltz, C.: Web Services Orchestration and Choreography, *IEEE Computer*, Vol. 28, No. 10, pp. 46–52 (2003).
- [7] Chafle, G. B., Chandra, S., Mann, V. and Nanda, M. G.: Decentralized orchestration of composite web services, *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, New York, NY, USA, ACM Press, pp. 134–143 (2004).
- [8] Nanda, M. G., Chandra, S. and Sarkar, V.: Decentralizing execution of composite web services, *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, New York, NY, USA, ACM Press, pp. 170–187 (2004).
- [9] O’Neil, E. J., O’Neil, P. E. and Weikum, G.: The LRU-K page replacement algorithm for database disk buffering, pp. 297–306 (1993).
- [10] Whang, K. and Park, C.: A Cost-Based Object Buffer Replacement Algorithm for Object-Oriented Database Systems (1996).
- [11] Wang, J.: A Survey of Web Caching Schemes for the Internet, *ACM Com-*

- puter Communication Review*, Vol. 25, No. 9, pp. 36–46 (1999).
- [12] Arlitt, M., Friedrich, R. and Jin, T.: Performance Evaluation of Web Proxy Cache Replacement Policies, *Lecture Notes in Computer Science*, Vol. 1469, pp. 193+ (1998).
  - [13] Jin, S. and Bestavros, A.: Popularity-Aware GreedyDual-Size Web Proxy Caching Algorithms, Technical Report 1999-009 (1999).
  - [14] He, C. and Karamcheti, V.: An Analysis of Usage Locality for Data-Centric Web Services, Technical Report TR2005-866, Computer Science Department of New York University (2005).
  - [15] Xie, Y. and O’Hallaron, D. R.: Locality in Search Engine Queries and Its Implications for Caching., *INFOCOM*, IEEE (2002).
  - [16] Scheuermann, P., Shim, J. and Vingralek, R.: A case for delay-conscious caching of Web documents, *Computer Networks and ISDN Systems*, Vol. 29, No. 8–13, pp. 997–1005 (1997).
  - [17] Shim, J., Scheuermann, P. and Vingralek, R.: A Unified Algorithm for Cache Replacement and Consistency in Web Proxy Servers, *International Workshop on the Web and Databases*, pp. 1–13 (1998).