國立交通大學

資訊科學與工程研究所

碩 士 論 文

延伸 Globus Toolkit Java WS Core 並提供可靠的格網訊息服務

Extending Globus Toolkit Java WS Core to Support Reliable Grid Messaging Services

研 究 生：陳勇宇

指導教授：袁賢銘 教授

中 華 民 國 九 十 五 年 六 月

延伸 Globus Toolkit Java WS Core 並提供可靠的格網訊息服務

Extending Globus Toolkit Java WS Core to Support Reliable Grid Messaging Services

研 究 生：陳勇宇　　Student：Yung-Yu Chen

指導教授：袁賢銘　　Advisor：Shyan-Ming Yuan

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年六月

延伸 Globus Toolkit Java WS Core 並提供可靠的格網訊息服務

研究生：陳勇宇　　指導教授：袁賢銘

國立交通大學資訊科學與工程研究所

摘要

近年來，隨著格網運算的熱潮，企業也漸漸地採用格網技術來整合新舊系統。Globus Toolkit 是目前業界通用地用來建構格網環境的主要工具之一，最新版的 Globus Toolkit (GT4) 採用了服務導向架構，提供了以網路服務為基礎的格網環境。然而，Globus Toolkit 卻沒有保證在網路服務訊息互相傳遞的情況下能可靠地傳送與接收訊息，一旦在訊息在傳遞過程中，發生了系統癱瘓或是網路問題，則會造成訊息的流失，對於企業而言，將會造成非常嚴重的影響。此外，Globus Toolkit Java Web Services core 所提供的訊息傳遞機制是基於資源特性的改變，意即它把訊息當作是資源特性，以程式設計師的角度來看是不太合理的。

由於網路的快速發展，訊息導向中介軟體成為企業間傳遞訊息最普遍使用的工具，昇陽公司制定了 Java 訊息服務應用程式設計介面 (JMS API)，提供一個統一的標準介面，讓建立於訊息導向中介軟體之上的應用程式具有可移植性。我們實驗室所開發的 Persistent Fast Java Message (PFJM) 即是一套基於 JMS 所開發的產品，擁有可靠的訊息傳遞機制，並且加強了永續訊息與效能等特性。在這篇研究當中，我們將整合 PFJM 與 GT4 Java WS core，將 PFJM 包裝成網路服務，並設計合理的程式邏輯，以及提供方便有用的工具讓開發網路服務的使用者使用。最後我們測試了分別以 GT4 Java WS core 與 PFJM WS 來傳遞訊息的應用程式，由測試結果可以看出，透過我們的 PFJM WS 來傳遞訊息擁有比 GT4 Java WS core 傳送訊息有較好的效能。

Extending Globus Toolkit Java WS Core to Support Reliable Grid Messaging Services

Student：Yung-Yu Chen Advisor：Shyan-Ming Yuan

Institute of Computer Science and Engineering
National Chiao Tung University

Abstract

In the recent years, with the upsurge of the grid computing technology, enterprises adopt the grid technology to integrate legacy and new systems gradually. Globus Toolkit is one of the most important tools in the industry to construct grid environment. The newest version of Globus Toolkit (GT4) adopts Services Oriented Architecture (SOA) to provide grid environment based on Web Services. However, Globus Toolkit does not guarantee to reliably send and receive messages during messages passing between Web Services. Once systems crash or network fails during messages communication, it will cause the messages to be lost. For enterprise, this will make very serious effect. Furthermore, the messages communication mechanism which Globus Toolkit Java Web Services core provides is based on the changes of resource properties. In other words, it regards messages as resources properties but that is unreasonable for the perspectives of the programmers.

With the rapidly growth of Internet, Message Oriented Middleware (MOM) has become the widespread used tool for delivering messages between enterprises. Sun Corporation has defined the Java Message Service Application Programming Interface (JMS API) to provide a unified interface for portability of the programs

developed on the Message Oriented Middleware. Persistent Fast Java Message (PFJM) is a JMS compliant Message Oriented Middleware developed by our laboratory and it has a reliable messages passing mechanism and some improved features such as persistent message and high performance. In this research, we will integrate PFJM and GT4 Java WS core to design reasonable programming styles and provide convenient and useful tools for Web Services development users by wrapping PFJM into PFJM Web Services (PFJM WS). Finally, we give a throughput test of messages communication respectively for GT4 Java WS core and PFJM WS. In the report, we can see that PFJM WS has a higher performance than GT4 Java WS core.

Acknowledgements

首先感謝我的指導教授 袁賢銘教授 教導分散式系統上的基礎以及給予我論文重要的意見。在分散式系統實驗室的博士後研究學長葉秉哲，及博士班學長蕭存喻、吳瑞祥、鄭明俊、高子漢，感謝他們在自己研究之餘能為我的論文給予不少的意見，祝福他們能順利地獲得博士學位。另外也恭喜實驗室的其他碩二生，林良彥、陳俊元、李杰葳、林家鋒、蔡紀暘以及范志歆能順利地畢業。

此外，秀玲，謝謝妳能給予我精神上的鼓勵與支持，陪伴我順利地過完兩年碩士生涯。

最後感謝我的父母 陳三郎先生 和 林里女士，由於你們的辛勞付出，才能讓我進入國立交通大學就讀，並且能無後顧之憂地完成學業，真的非常感激你們。

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Preface

Due to the improvement of the technology or the new demands of the long-standing companies which have a large number of legacy systems, they may like to purchase or develop new systems to enhance services providing by those legacy systems. Nevertheless, for example, companies do not want to drop the legacy database storing bulky datasets and indeed throwing away a system may have unexpected risks. So what they want is to retain the legacy systems and the new ones simultaneously. If there is a fine mechanism to integrate those legacy and new systems, enterprises will save a lot of efforts and money when emerging systems appear frequently.

As a result of the importance of communication between systems and systems, Message Oriented Middleware (MOM) [1] has been brought up. MOM is the term for software that connects separate systems in a network by carrying and distributing messages between them. Unlike Remote Procedure Call (RPC) [2] it is an asynchronous form of communication, i.e. the sender does not block waiting for the recipient to participate in the exchange. If the message service offers persistence and reliability then the receiver need not be up and running when the message is sent.

However, there are so many MOM venders in the world and each vender provides its own proprietary API. When application developers who develop applications to communicate with each other using MOM move from this MOM to another, the codes can not be used again, i.e. developers must spend extra effort to rewrite new codes

applied to the new MOM. For portability, Sun Microsystems and its partners wrote the Java Message Service (JMS) spec [3]. One objective of JMS is to minimize the learning curve for writing messaging applications and to maximize the portability of messaging applications. And the Advantage of the standard interface is that applications written by JMS API can run on every JMS compliant MOM.

Nowadays there are many MOM providers such as SonicMQ [4], FioranoMQ [5], and OpenJMS [6] which support the JMS 1.1 standard finalized at 2002 at least. Persistent Fast Java Messaging (PFJM) [7] [8] is a Message-oriented Middleware designed by our laboratory. In addition to support JMS 1.1 standard, we enhance the persistent message and high performance features for increasing the scalability greatly.

1.2 Motivation

In recent years, with the rise of grid technology [29] [30] enterprises are willing to take grid technology as the opportunity to integrate their resources inside such as computing powers, storages, and so on. In traditional enterprise environment, for example, the specific service enterprises provide may reside on specific server such as Figure 1-1 which depicts the traditional enterprise system model. If the capability of the server reaches its limit, what enterprises can do is to buy another more powerful server to replace with the old one. This is a very wasted solution since other servers which users access seldom may still have a lot of computing power or resources left.

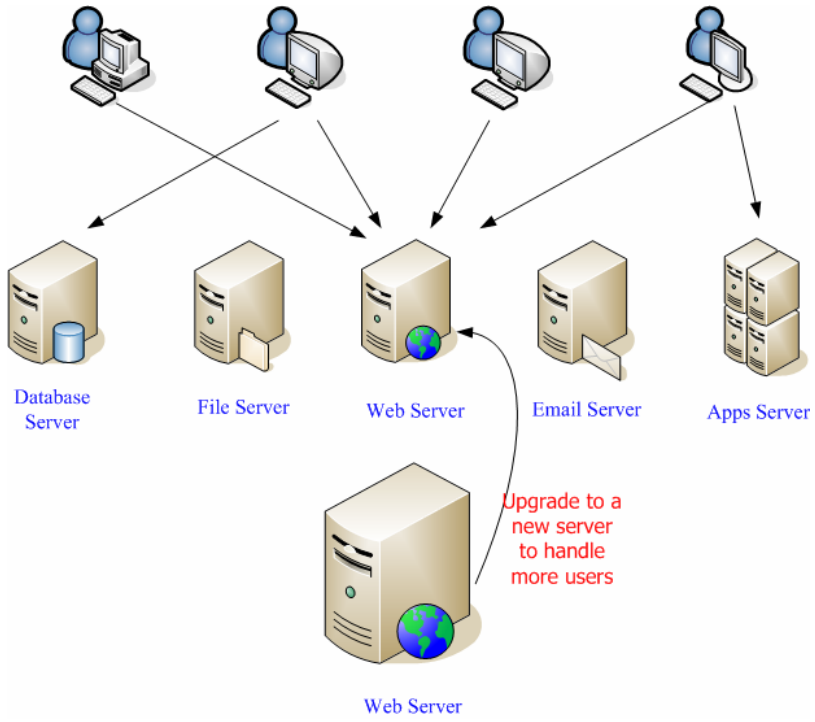


Figure 1-1 Enterprise Traditional Model

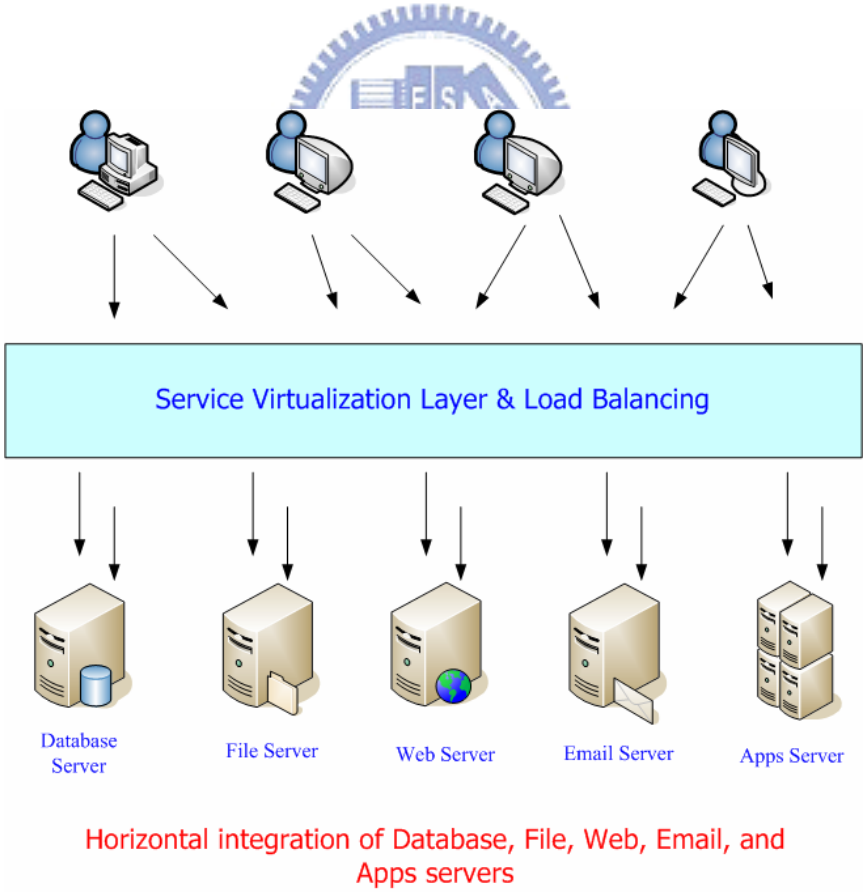


Figure 1-2 Enterprise Grid Based Model

Nowadays enterprises have considered adopting the grid solution to integrate their overall resources. By means of grid technology depicted in Figure 1-2, users access a service virtualization layer instead of accessing to the specific server directly. The service virtualization layer can abstract the underlying all kinds of services to a single, large virtual world so every server can exhaust himself to reach the most effective avail.

The Globus Toolkit (GT) [9] [32] [33] is an open source software toolkit used for building Grid systems and applications. It is being developed by the Globus Alliance and many others all over the world. Java WS Core [10] is one of GT common runtime components and it provides APIs and tools for developing Grid services and offers a run-time environment capable of hosting them. The Java WS Core in GT4 implements the Web Services Resource Framework (WSRF) [11] [12] [13] [31] and the Web Service Notification (WSN) [14] family of standards. However the communication mechanism between web services does not guarantee reliable messaging. In other words, the notification consumer can not reliably receive the messages sent by notification producer since the unreliable network. Furthermore communication between applications in enterprise environment is expected to be reliable because the messages lost may case a very serious consequence. It stands to reason that we extend Globus Toolkit WS Core to support reliable messaging.

Fortunately JMS fits the features, Web Service Notification which GT Java WS Core implements, and it is more important that JMS provides the guarantee of reliable messaging. So in this research we will integrate PFJM, a JMS compliant product developed by our lab, into GT Java Web Core to provide useful web services for reliable messaging.

1.3 Research Objectives

In this research, we discuss the necessity of reliable messaging and the defective GT4 Java WS core. There are two objectives in this research including reliable messaging and reasonable programming styles.

1.3.1 Reliable messaging

Since the GT4 Java WS core lacks reliable messaging which is important for grid applications, we integrate a JMS compliant product, PFJM, into GT4 Java WS core to provide reliable messaging mechanism. The features are described in the following：

1. Persistent messaging：Persistent messages are guaranteed to survive through JMS provider failure. If a message is set as persistent, before it is sent to the network, it must be stored in a persistent storage.
2. Durable subscription：Durable subscribers are guaranteed to receive persistent message published during their registration and de-registration, even they are not always active.

1.3.2 Reasonable programming styles

Since messages in GT4 Java WS core are always marked as a resource property discussed more detailed in Section 4.2, this is unreasonable from the programmer's point of view. For programmers, they expect to use Topic as the message destination to send and receive messages. Through our system, PFJM WS, clients can use JMS-like programming style to send and receive messages reliably.

1.4 Organization

This research is organized as following: In Chapter 2, the background will be reviewed. We will talk about JMS including its architecture, message delivery model,

and its main feature, reliable messaging. Then we will introduce PFJM which is a JMS compliant product developed by our lab. Also the Globus Toolkit will be introduced in details including its architecture, the specifications it implements, and the leading role GT4 Java WS core. In Chapter 3, we will describe my system, PFJM WS, including architecture, the service portTypes we provide, and the mechanism of communication. In Chapter 4, we will give scenarios respectively for PFJM WS and for GT4 Java WS core and discuss the execution sequence from a programmer' point of view in details. In Chapter 5 we will experiment the throughput of PFJM WS and GT4 Java WS core based on the scenarios described in Chapter 4. Finally in Chapter 6, there is a conclusion of this research and we also give an idea for the future work.

Chapter 2 Background

2.1 Java Messaging Service (JMS)

The Java Message Service (JMS), which is designed by Sun Microsystems and several other companies under the Java Community Process as **JSR 914** [15], is the first enterprise messaging API that has received wide industry support. The Java Message Service (JMS) was designed to make it easy to develop business applications that asynchronously send and receive business data and events. It defines a common enterprise messaging API that is designed to be easily and efficiently supported by a wide range of enterprise messaging products.

2.1.1 Design Architecture

The design architecture of JMS can be divided into two categories：

1. Central Architecture：Figure 2-1 depicts the central architecture of MOM design. JMS Server is responsible for message delivery. JMS Client is a so-called publisher or a subscriber in JMS field. Each application uses JMS API to be a publisher or a subscriber to send or receive messages. Once JMS Server receives the messages from a publisher, it then sends the messages to a subscriber. The shortcoming of central architecture is the server-bottleneck problem. If the central server gets low performance or even becomes failure, the overall message exchange system will becomes unavailable. However, it has the advantages of easy management and uncomplicated design.

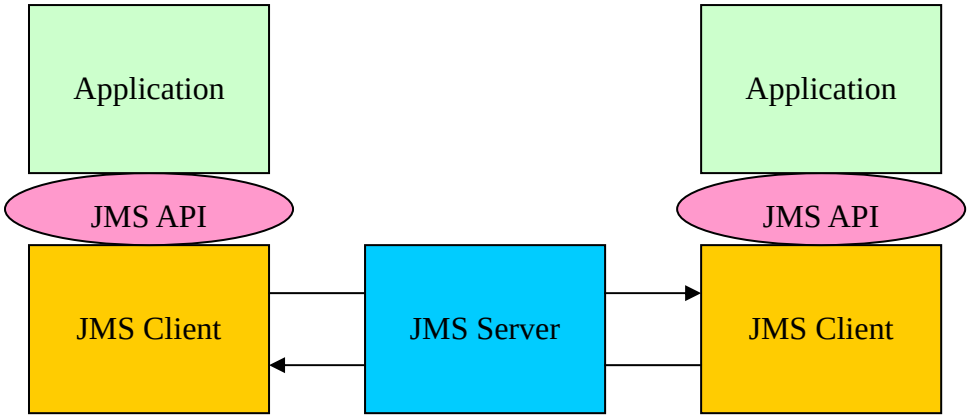


Figure 2-1 Central Architecture of MOM Design

2. Distributed Architecture：Figure 2-2 depicts the distributed architecture of MOM design. Under this architecture there is no central server any more and messages delivery jobs are distributed to every JMS client participating in the messages communication. Because there is no longer a server, every client must be aware of some information of other clients, for example, IP and Port. The advantage of distributed architecture is the loading of original central server is divided and distributed to every client. So the single-point-failure problem does not exist. On the contrary, resource management will be complicated and difficult.

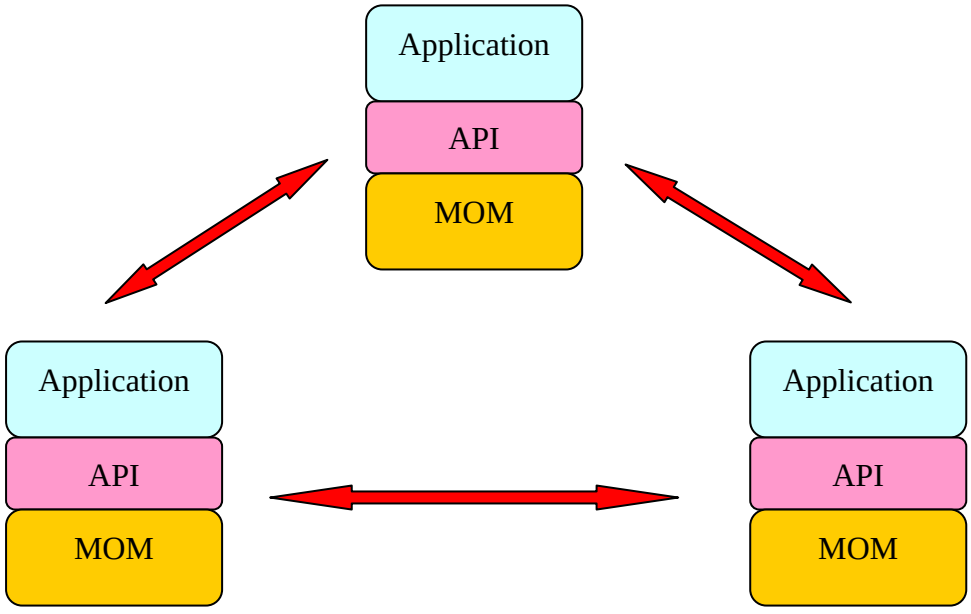


Figure 2-2 Distributed Architecture of MOM Design

2.1.2 Message Delivery Models

JMS supports two different message delivery models:

1. Point-to-Point (Queue destination): In this model, a message is delivered from a producer to one consumer. The messages are delivered to the destination, which is a queue, and then delivered to one of the consumers registered for the queue. While any number of producers can send messages to the queue, each message is guaranteed to be delivered, and consumed by one consumer. If no consumers are registered to consume the messages, the queue holds them until a consumer registers to consume them.
2. Publish/Subscribe (Topic destination): In this model, a message is delivered from a producer to any number of consumers. Messages are delivered to the topic destination, and then to all active consumers who have subscribed to the topic. In addition, any number of producers can send messages to a topic destination, and each message can be delivered to any number of subscribers. If there are no consumers registered, the topic destination doesn't hold messages unless it has **durable subscription** for inactive consumers. A durable subscription represents a consumer registered with the topic destination that can be inactive at the time the messages are sent to the topic.

2.1.3 Reliable Messaging

JMS defines two reliability-related specifications：

1. Persistent messaging: "*The PERSISTENT mode instructs the JMS provider to take extra care to insure the message is not lost in transit due to a JMS provider failure. A JMS provider must deliver a PERSISTENT message once-and-only-once. This means a JMS provider failure must not cause it to be lost, and it must not deliver it*

twice."

2. Durable subscription: "*A durable subscriber registers a durable subscription with a unique identity that is retained by JMS. Subsequent subscriber objects with the same identity resume the subscription in the state it was left in by the prior subscriber."*

2.2 Persistent Fast Java Messaging (PFJM)

Persistent Fast Java Messaging is a JMS compliant product designed by our laboratory. PFJM adopts the distributed architecture and implements the message delivery protocol using IP multicast technology. Figure 2-3 illustrates the overall architecture of PFJM.

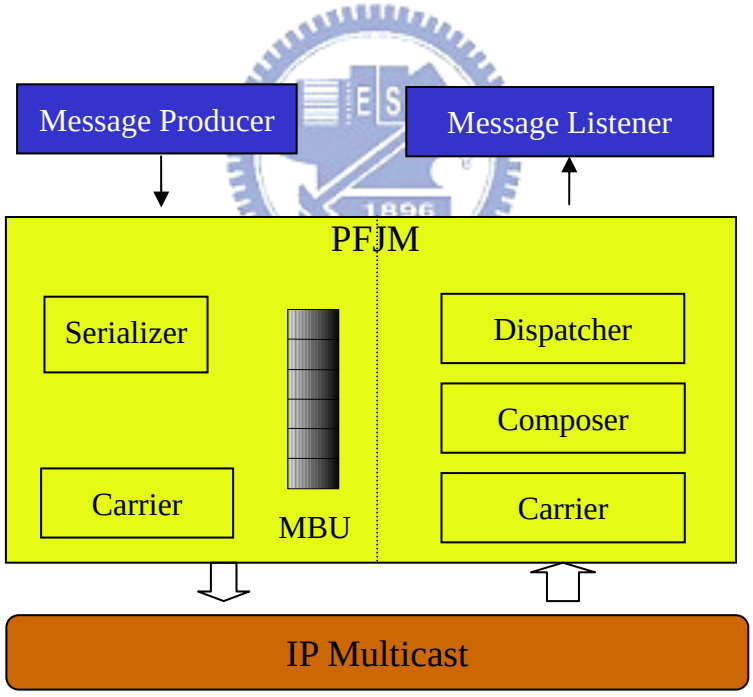


Figure 2-3 Architecture of PFJM

In a distributed computing environment that uses PFJM, every participated program (or host) has to execute the required PFJM run-time components, which are different due to the role of hosts. In the publisher part, the activated components are **Serializer**

and **Carrier**. In the subscriber part, the activated components are **Dispatcher**, **Composer**, and **Carrier**.

In publisher, Serializer serializes all messages into sequence of bytes, and then divides the bytes stream into appropriate-sized Memory Block Unit (MBU). Theses MBUs are put into a sending queue. Then, carrier delivers MBUs in sending queue to subscribers in order. Subscriber's Carrier receives MBUs which were sent by publisher and put them into Input Memory. If message listener is set, once all MBUs of one JMS message arrived, the message will recomposed and send to Dispatcher. Dispatcher will dispatch the message to participate application.

In addition to following JMS standard, PFJM also emphasizes on some features such as persistent message and high performance.

2.3 Globus Toolkit

The Globus Toolkit (GT) has been developed since the late 1990s to support the development of distributed computing applications and infrastructures. The objective of the GT is to provide a set of libraries and programs that address common problems which occur when building distributed system services and applications. The following sections will introduce the details of Globus Toolkit.

2.3.1 Architecture

GT4, the latest version, is composed of several components such as **Security** [16], **Data Management** [17], **Information Services** [18],**Execution Management** [19], and the **Common Runtime** [20] as shown in the Figure 2-4. With the popularity of

Web Services GT4 has already focused on Web Service's tool development. Even so GT4 also provide both tools including WS and non-WS components.

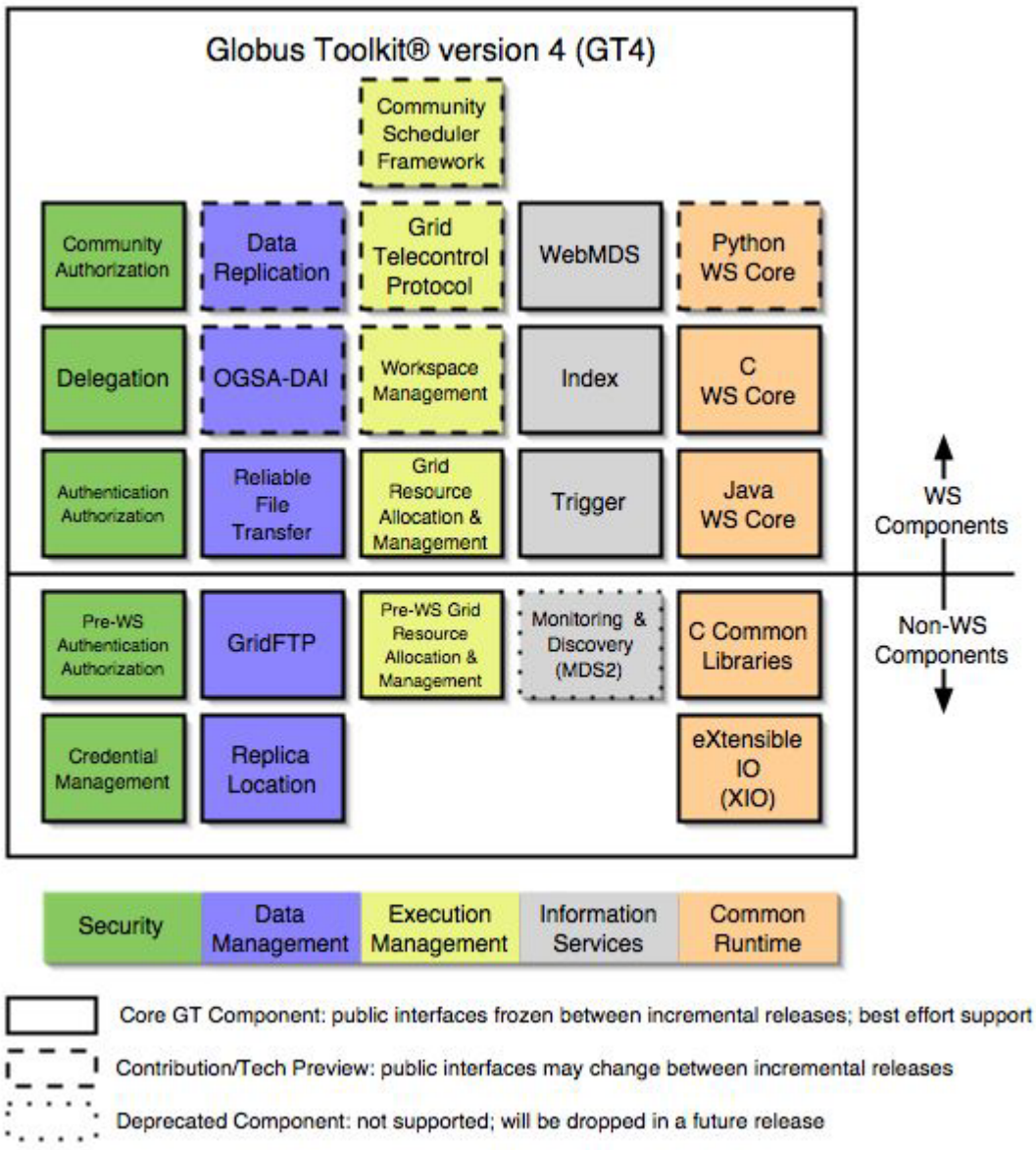


Figure 2-4 GT4 Architecture

Let us take a look at what these five components can provide：

1. **Common Runtime**：The Common Runtime components provide fundamental libraries and tools which are needed to build WS and non-WS services.
2. **Security**：The Security components can ensure communication being secure based on Grid Security Infrastructure (GSI).

3. **Data Management**：The Data Management components can allow us to manage large sets of data in our virtual organization.
4. **Execution Management**：The Execution Management components deal with the initiation, monitoring, management, scheduling and coordination of executable programs, usually called **jobs**, in a Grid.
5. **Information Services**：The Information Services components can be used to discover and monitor resources in a virtual organization.

2.3.2 Service Oriented Architecture (SOA)

GT4 is a set of software components for building distributed systems in which diverse and discrete software agents interact via message exchanges over a network to perform some tasks. GT4 is, more specifically, a set of software components that (with some exceptions) implement Web services mechanisms for building distributed systems. Web services provide flexible, extensible, and widely adopted XML-based mechanisms for discovering, describing, and invoking network services; in addition, its document-oriented protocols are well suited to the loosely coupled interactions. These mechanisms facilitate the development of service-oriented architectures – systems and applications structured as communication services in which services are discovered, interfaces are described, operation are invoked, and so on, all in uniform ways.

So, what is **Service Oriented Architecture** [21]? SOA can be characterized by the following aspects：

1. Logical view：The service is abstracted from actual programs, database, business logic, and etc. It defines what it does and carries out business operations.

2. Message orientation：The service is defined in terms of messages exchanged between service provider and service requester but not the properties of the service itself. Using SOA one does not know the actual implementation of the service including language, architecture, and etc. So the legacy systems can be used by means of the message passing mechanism and hidden implementation details from service requesters.
3. Description orientation：The service is described by machine-readable metadata. Only the details exposed to the public and important for the use of the service are should be included in the description.
4. Granularity：Services are composed of a small number of operations with relatively large and complex messages.
5. Network orientation：Services tend to be deployed over network.
6. Platform neutral ： Due to XML technology, messages are sent in a platform-neutral and standardized format

2.3.3 Web Services

As mentioned in previous section, GT4 is set of components that implement Web Services [22]. Now let us take a look at what is Web Service and how does it work：A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (WSDL [23]). The interface defines the operations in which clients can use that to interact with Web Services by SOAP [24] messages conveyed by HTTP.

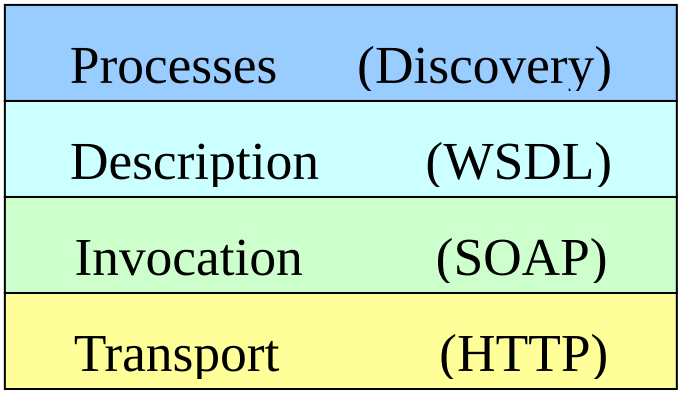


Figure 2-5 The Web Services Architecture

Figure 2-5 depicts the Web Services Architecture and the most popular protocol or function used in each layer：

1. Service Processes：The highest layer usually contains many Web Services. From Figure 2-5 the discovery service can allow us to locate one particular service from a collection of Web Services.
2. Service Description：Once you have located a Web Service, you can ask it to "describe itself" and tell you what operations it supports and how to invoke it. This is handled by the Web Service Definition Language (WSDL).
3. Service Invocation：SOAP (Simple Object Access Protocol) specifies how we should format requests to the server and how the server should format responses to requesters.
4. Transport：Finally all the messages must be transmitted by HTTP.

So how does the invocation of Web Services work？Figure 2-6 depicts the process of the invocation between clients and Web Services. Let us supposes that we have already located the Web Service we want to use and the stubs have been generated from WSDL files. The following steps describe the overall process：

1. Whenever our program calls the Web Service, it in fact calls the client stubs. The

client stubs will turn the “local invocation” into a proper SOAP message. This is called **marshaling** or **serializing** process.

2. The SOAP request is sent to Server via internet. After receiving the SOAP request, the server stubs will convert that into something the Web Service implementation can understand (this is typically called **unmarshaling** or **deserializing** process).
3. Once the SOAP request has been deserialized, the server stubs call the Web Service implementation and ask it to do the proper operation.
4. After the Web Service handles the request operation, it returns the result of the operation to server stubs which turn the result into SOAP response.
5. The SOAP response is set to Client via internet. After receiving the SOAP response, the client stubs convert that into something our program can understand.
6. Finally our program receives the result of Web Service invocation and uses it.

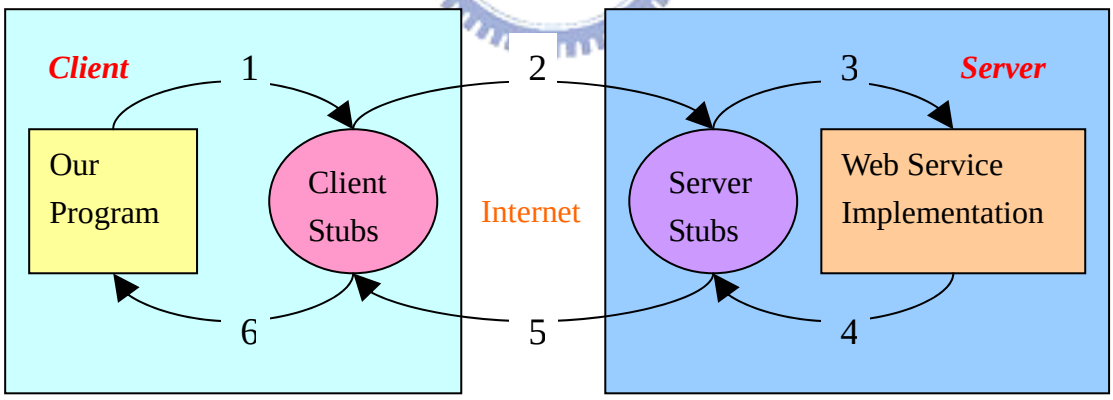


Figure 2-6 A Typical Web Service Invocation

2.3.4 Web Services Resource Framework (WSRF)

So far we have discussed the Web Services which are the technology for building internet-based loosely coupled applications. That makes them the natural choice for building next generation of grid-based applications. However Web Services have some limitation that is inadequate for grid applications. For example, plain Web Services are usually stateless. This means that Web Services cannot remember information from one invocation to another. As shown in Figure 2-7, this is an integer accumulator Web Service handling requests from clients to execute add operation. Due to the integer accumulator Web Service cannot keep the state which previous invocation brings out, the Web Service always returns the same response to clients.

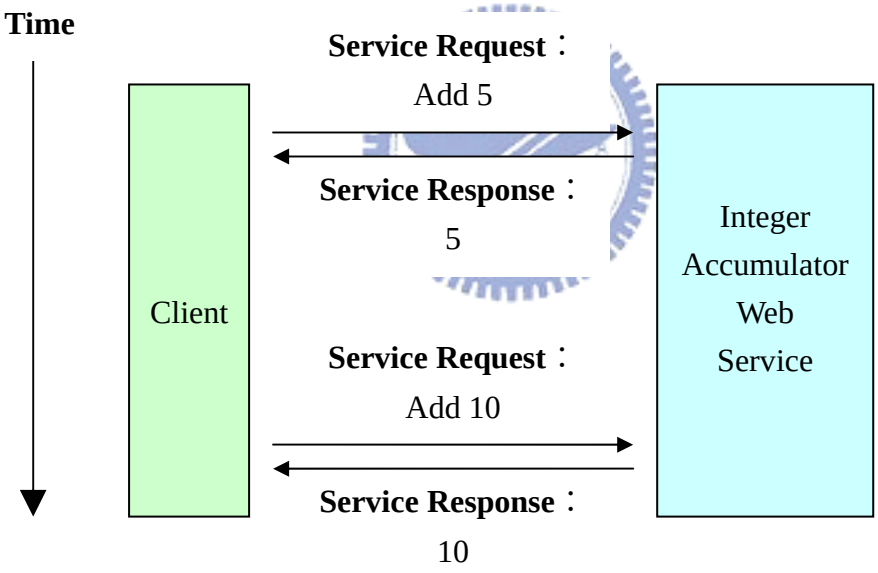


Figure 2-7 A Stateless Web Service Invocation

Although the Web Service cannot keep state, this is not a bad thing for certain applications such as Weather Web Service returning information of weather. However, **Grid** applications do generally require statefulness. From Figure 2-8 we can see how an integer accumulator Web Service using **State** works.

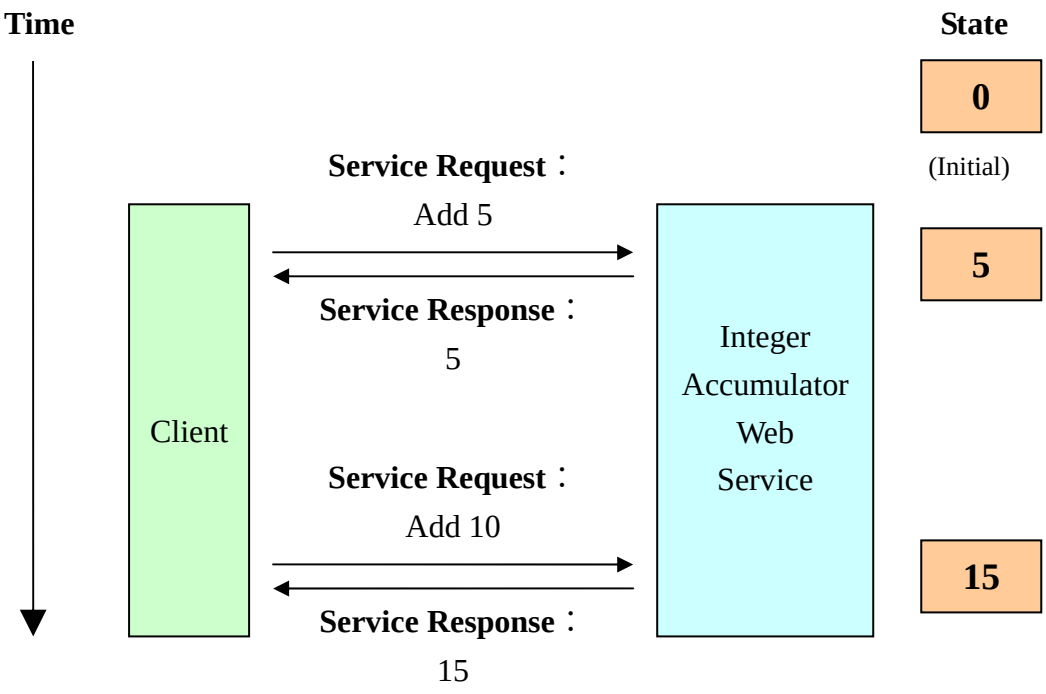


Figure 2-8 A Stateful Web Service Invocation

Now let us take a look at how GT4 implements WSRF to support stateful Web Services. Instead of putting the state in the Web Service, GT4 keeps state in a separate **resource** which stores all the state information of Web Services. Each resource has a unique key so whenever we want to interact with a Web Service, we simply instruct the Web Service which resource we want to use.

As shown in Figure 2-9, we specify the resource B that we want to execute the add operation. When the Web Service receives the request, it will try to retrieve resource B and perform add operation on that resource. The resources themselves will be stored in memory, database, or secondary storages depending on different implementations.

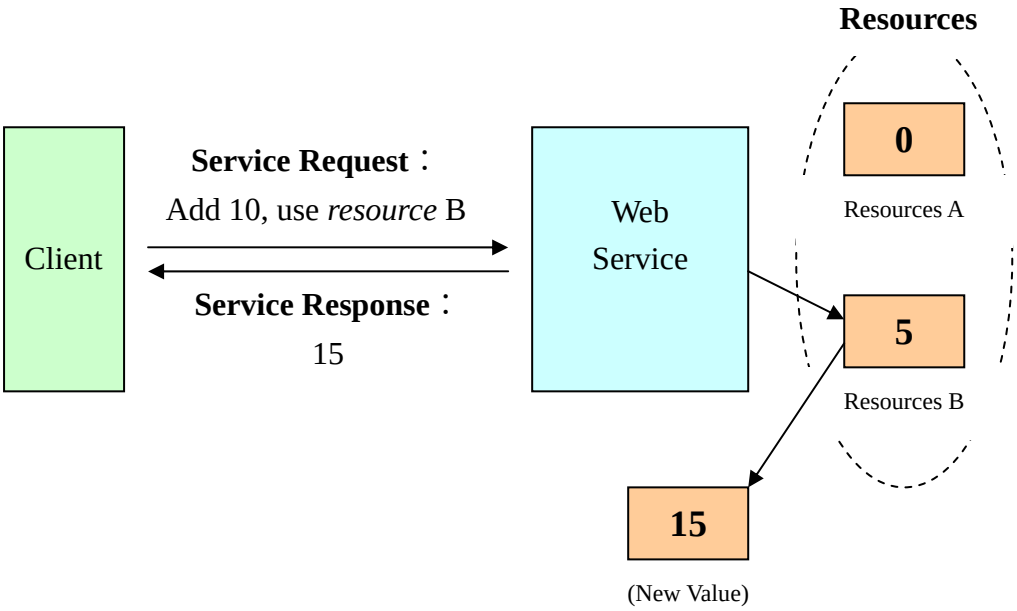


Figure 2-9 The Resource Approach to Statefulness

Of course, resource can come in different shapes and can have multiple properties as shown in Figure 2-10.

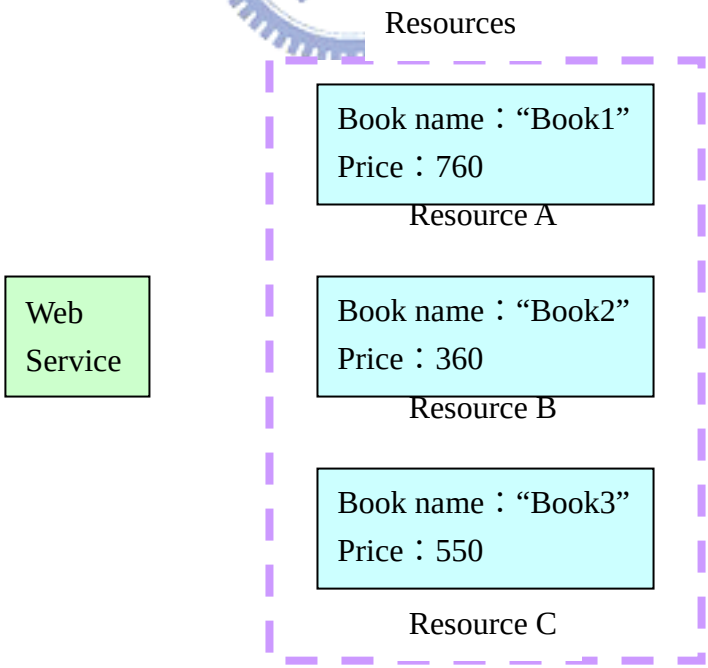


Figure 2-10 A Web Service with Three Resources and Each Resource Has Two Properties

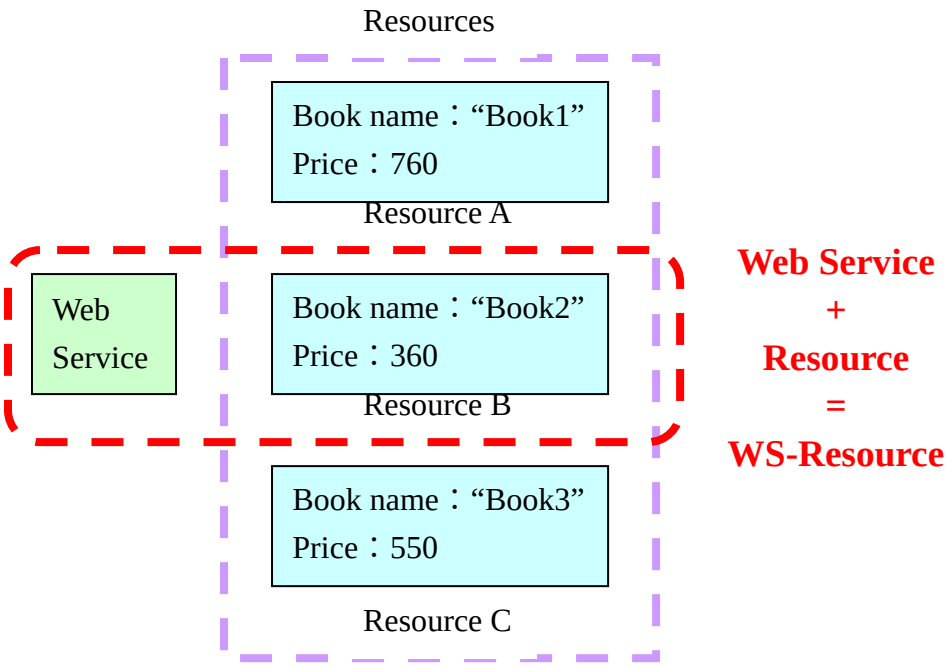


Figure 2-11 WS-Resource

Figure 2-11 depicts the relation between Web Service and Resource. **WS-Resource** [25] is a combination of a stateful resource and a Web Service and by the way **WS-Resource Properties** [26] is the properties in the resource.

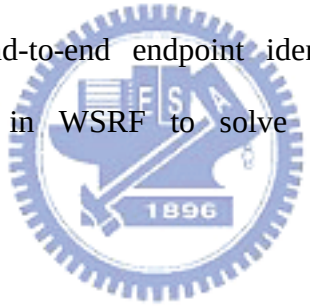
In brief, WSRF defines a number of operations that can be performed on a WS-Resource, from getting and setting its properties to adding it to a **ServiceGroup** with other similar services. It is also possible to use WSRF to destroy a WS-Resource or to set an "expiration date" for it.

2.3.5 Web Services Addressing (WSA)

From the previous section we have known what WS-Resource is. But how does a client locate a specific WS-Resource. Once upon a time, it was easy to specify the address of a Web service. All you really needed was the URL. Nowadays, with Web

service applications getting increasingly complex, it's not always that simple. What if you want the reply to be sent somewhere other than the original requestor？What if you simply need to attach to a particular instance of a Web service?

The **Web Services Addressing** [27] comes to solve this problem. WS-Addressing provides a way to specify information about a location other than just a simple Universal Resource Identifier (URI) or URL. WS-Addressing introduces the concept of an **EndpointReference**. The EndpointReference is a way to specify the information needed to get a message to the right place. In WS-Addressing, the address of a particular WS-Resource is called an endpoint reference. Specifically, this specification (WS-Addressing) defines XML elements to identify Web service endpoints and to secure end-to-end endpoint identification in messages. GT4 implements WS-Addressing in WSRF to solve the issues of addressing of WS-Resource.



2.3.6 Web Services Notifications (WSN)

We have talked about WS-Resources which are the combination of a stateful resource and a Web service. One common situation in WSRF is the need for one resource to know when the properties of another have changed. For example as shown in Figure 2-10, a client may want to know the price of the book has changed when a publisher changes its price. WSN lets it easy. We can create a structure in which client components can "subscribe" to a particular topic such as a change in the book’s price and when the event takes place, those components get a notification message. In short, WSN enables us to emulate an event-based system using Web services.

GT4 currently doesn't implement the WS-Notifications family of specifications completely. For example, no support for brokered notification is included. However, GT4 does allow us to perform effective topic-based notification. One of the more interesting parts of the GT4 implementation of WS-Notifications is that it will allow us to effortlessly expose a resource property as a topic, triggering a notification each time the value of the resource property changes.

2.3.7 Java WS Core

The Java WS Core is an implementation of the Web Services Resource Framework (WSRF) and the Web Service Notification (WSN) family of standards. It provides APIs and tools for building stateful Web services. More specifically, GT4 provides three Web Services containers including Java, C, and Python to deal with such issues as message handling, resource management, and security, thus allowing the developers to focus their attention on implementing application logic.

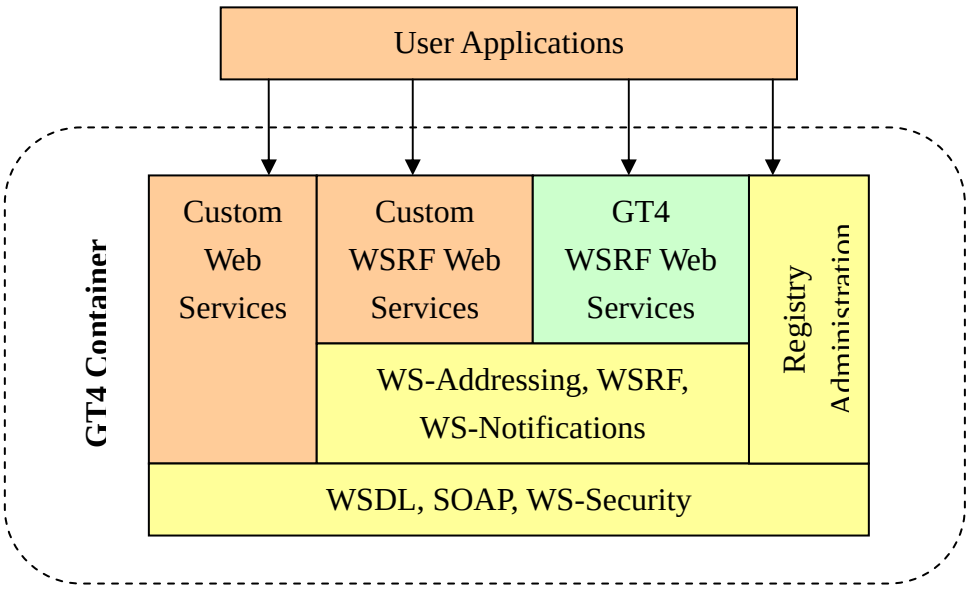


Figure 2-12 Capabilities of A GT4 Container

As illustrated in Figure 2-12, GT4 containers implement many WS specifications such as WSDL, SOAP, WS-Security, WS-Addressing, WSRF, and WS-Notifications to support basic Web Services functionality and support services that want to expose and manage state associated with services, back-end resources, or application activities.

In general, the Java container provides the most advanced programming environment, the C container the highest performance, and the Python container the nicest language. In this paper, we will focus on the Java WS core because JMS is a spec based on Java language.

Chapter 3 System Architecture and Design

3.1 Overview

In order to solve the reliable messaging problem which Globus Toolkit does not support, we integrated PFJM, a JMS compliant product developed by our lab, with Globus Java Web Services Core. From previous chapter we know Globus Java WS Core is an implementation of WSRF, WSN, and other relevant Web Services family of standards. It provides an environment and tools to help develop plain Web Services and stateful Web Services. So the solution we use is to utilize these components Java WS core provides to wrap PFJM into Web Services. Whenever a client wants to communicate with others using reliable messaging, it can simply exploit the Web Services we provide to easily achieve its goal.

In Section 3.2, we will show the system architecture and introduce the basic operation and relationship of each component in the architecture. Then we will introduce the individual service portType we support in Section 3.3 and the mechanism of communication in Section 3.4.

3.2 System Architecture

Figure 3-1 depicts the simplified system architecture. Whenever a client wants to be a message sender or a message receiver, it firstly must locate the persistent JMSFactoryService. A persistent service is a service which resides in the Web Services container when the container starts. After locating the JMSFactoryService, the client can use that to create a transient JMSPublisherService or

JMSSubscriberService depending on what the client wants to be. Compared to a persistent service, a transient service is a service which can be created and destroyed dynamically. Then the client can use JMSPublisherService or JMSSubscriberService to create a PFJM instance, a publisher or a subscriber, and finally use the PFJM instance to do publish or subscribe operation via reliable messaging.

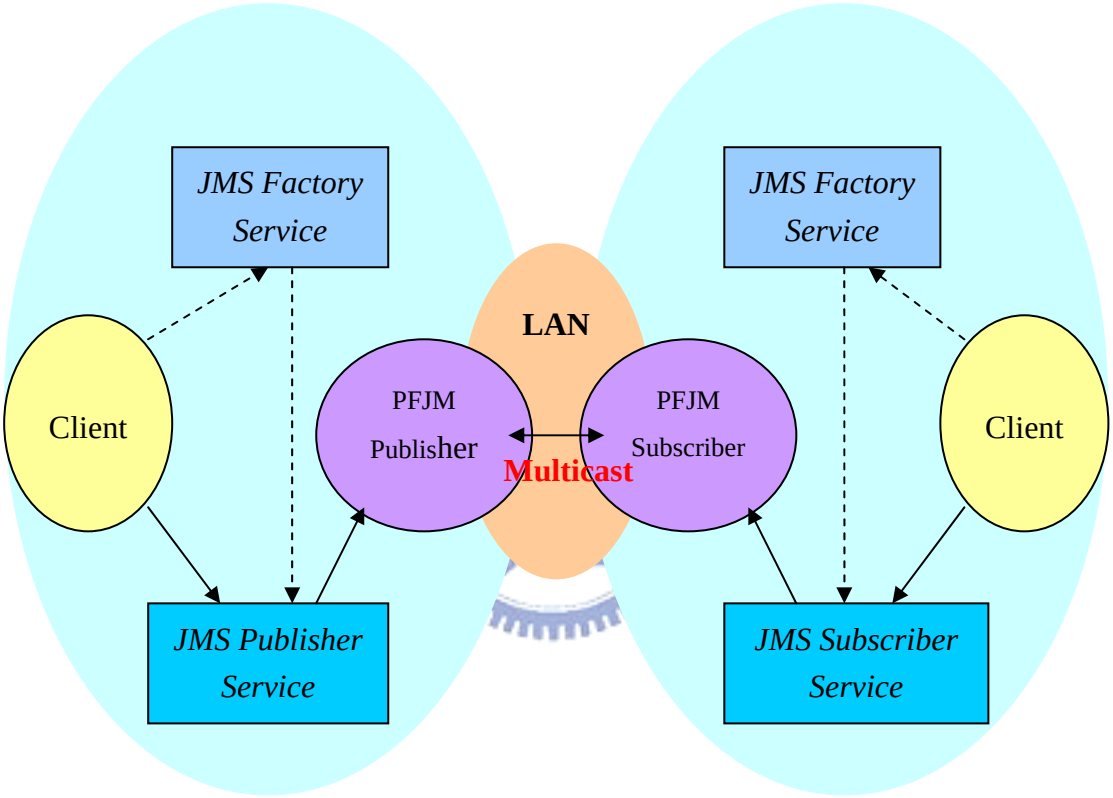


Figure 3-1 The System Architecture

3.3 Service PortTypes

3.3.1 JMSFactory PortType

First of all, as mentioned in Section 2.3.5 a Web Service can be addressed by a so-called EndpointReference. By passing the EndpointReference to a ServiceAddressingLocator, a client can get the service's portType implementation defined in the WSDL file. As well as JMSFactory portType which we provide using

Factory design pattern, the client can use JMSFactoryAddressingLocator to locate the JMSFactory service.

The position of the JMSFactory is to create a JMSPublisher service or a JMSSubscriber service. Now let us take a look at how the JMSFactory works. For the purpose of managing created services, we provide two auxiliary managers, JMSPublisherManager and JMSSubscriberManager respectively in charge of JMSPublisher and JMSSubscriber services. When a client asks the JMSFactory to create an instance service, the JMSFactory passes the job to the manager. The service manager takes care of actually creating a new JMSPublisher or JMSSubscriber service and receives an object of type ResourceKey returned from a service resource home which implements ResourceHome interface provided by GT WS core. The ResourceKey is the identifier which we need to create the endpoint reference returning to the client. Figure 3-2 depicts the relationship of the service manager and the service resource home.

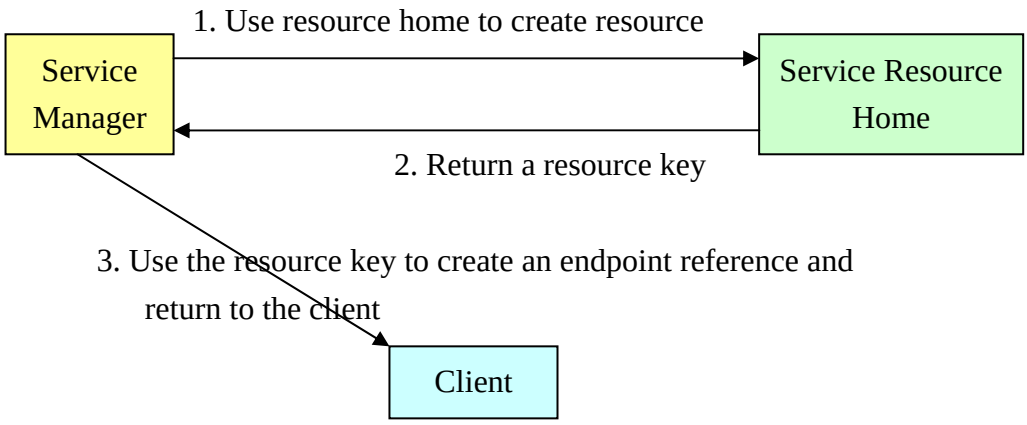


Figure 3-2 The Relationship of Service Manager and Resource Home

The following two figures, Figure 3-3 and Figure 3-4, depict the sequence diagrams

of creating a JMSPublisherService and a JMSSubscriberService. Both of them use JMSFactoryAddressingLocator to get the JMSFactory portType and then acquire the JMSPublisher or JMSSubscriber endpoint reference from the service manager.

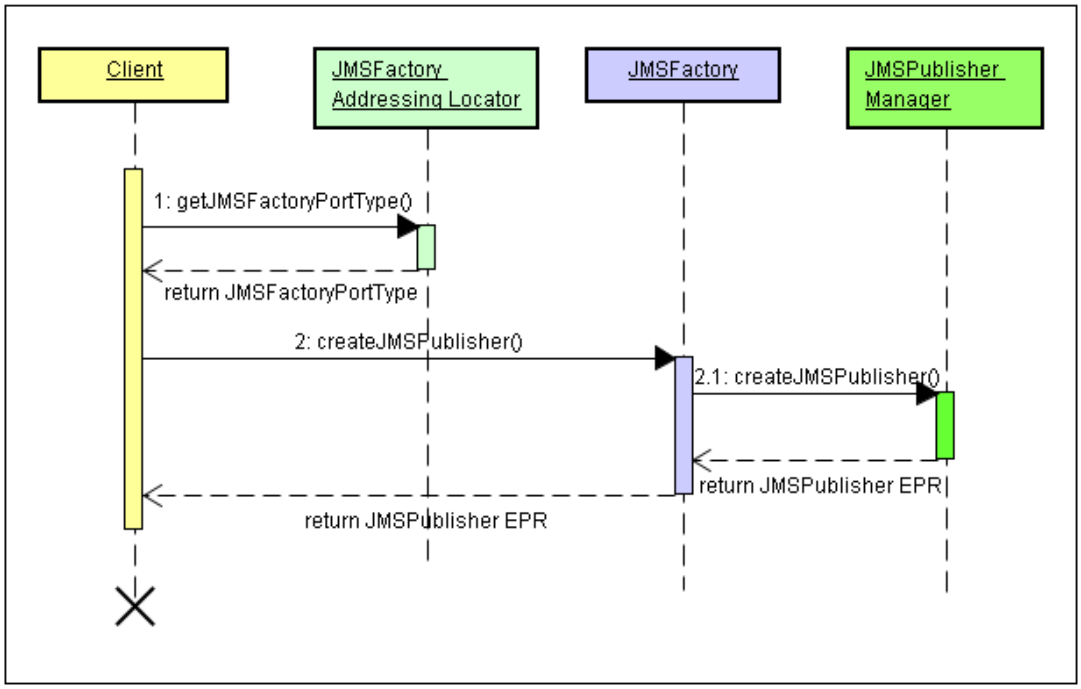


Figure 3-3 The Sequence Diagram of Creating A JMSPublisherService

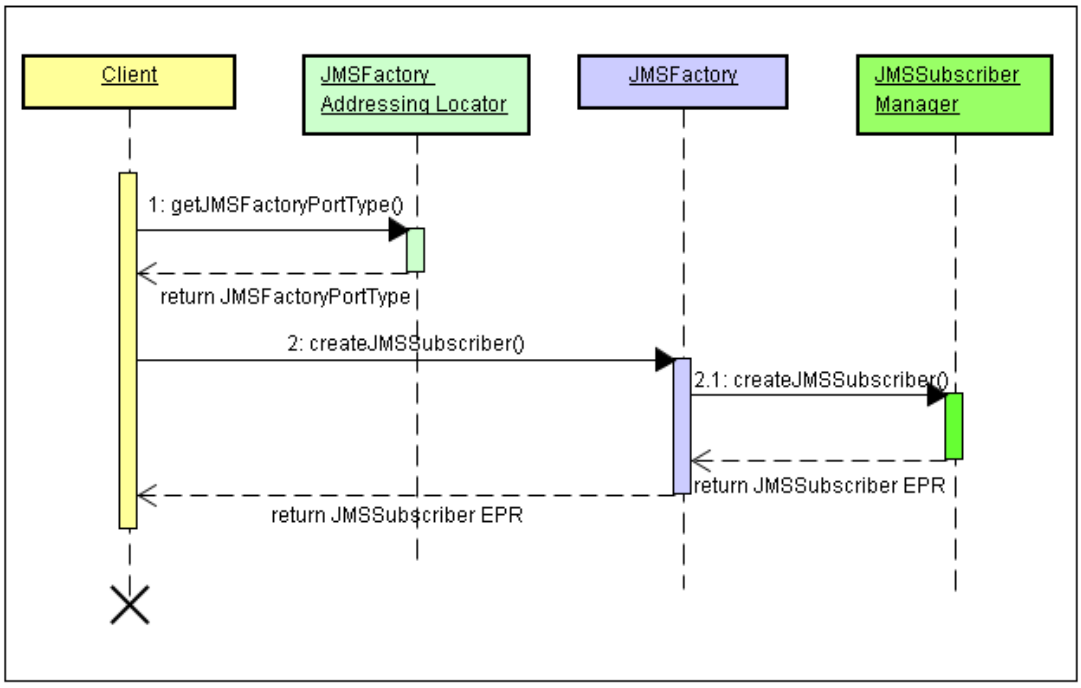


Figure 3-4 The Sequence Diagram of Creating A JMSSubscriberService

3.3.2 JMSPublisher PortType

JMSPublisherService created from JMSFactoryService is a transient service responsible to publish messages to a specific Topic. Whenever clients want to communicate to each other with reliable messages, the message sender can create a JMSPublisherService by passing specified topic name to JMSFactoryService and then utilize the created JMSPublisherService to publish messages. More precisely speaking, the JMSPublisherService is a PFJM instance which actually handles message sending.

The JMSPublisherService exposes only one operation, **publish**, to the public. And the publish operation has one parameter which is a String object indicating the sending messages. Figure 3-5 shows the simplified WSDL file of JMSPublisherService.

```
<types>
<xsd:schema>
   <xsd:element name="publish" type="xsd:string"/>
</xsd:schema>
</types>

<message name="PublishInputMessage">
   <part name="parameters" element="tns:publish"/>
</message>

<portType name="JMSPublishPortType">
   <operation name="publish">
      <input message="tns:PublishInputMessage"/>
      <output message="tns:PublishOutputMessage"/>
   </operation>
</portType>
```

Figure 3-5 The Simplified WSDL File of JMSPublisherService

3.3.3 JMSSubscriber PortType

As described in Section 3.3.2, JMSSubscriber is also created from JMSFactoryService and responsible to subscribe to the specific Topic. When a message receiver has created a JMSSubscriberService from JMSFactory Service by passing topic name, it then could use the JMSSubscriberService to do subscribe operation. In addition, the message receiver must implement a **NotifyCallback** interface which defines one function called **deliver** and pass itself to the subscribe operation of JMSSubscriberService. The deliver function is the callback function which the message receiver wants to be called back asynchronously when messages arrive. Figure 3-6 depicts the simplified example of a message receiver.

```
public class Subscriber implements NotifyCallback{
    public void deliver(Object message)
    {
        ...
    }

    public static void main(String[] args) {
        JMSSubscribePortType jmsSubscribe =
            instanceLocator.getJMSSubscribePortTypePort(
                                            instanceEPR);
        jmsSubscribe.subscribe(this);
    }
}
```

Figure 3-6 The Simplified Example of A Message Receiver

The same as JMSPublisherService, JMSSubscriberService is a PFJM instance which actually handles message receiving. In addition to expose subscribe operation to the

public, the JMSSubscriberService must also implement an interface, **MessageListener**, which JMS spec defines for asynchronously receiving messages.

3.4 Mechanism of Communication

3.4.1 Publish Mechanism

Whenever a client wants to publish messages, in the beginning it must locate the JMSPublisherService by passing endpoint reference got from JMSFactoryService to JMSPublisherAddressingLocator. After locating the JMSPublisherService, it can call the publish function exposed by JMSPublisherService to send messages to a topic. Then JMSPublisherService activates the real publish operation provided by PFJM instance. Figure 3-7 depicts the sequence diagram of publish mechanism.

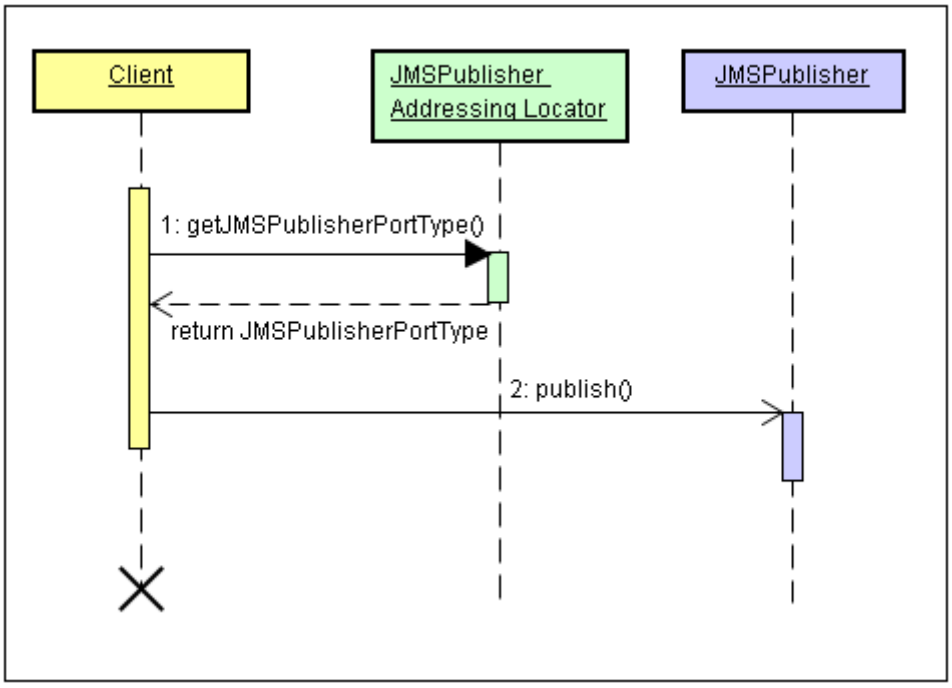


Figure 3-7 The Sequence Diagram of Publish Mechanism

3.4.2 Subscribe Mechanism

Whenever a client wants to subscribe to a topic, it firstly locates the JMSSubscriber by passing endpoint reference got from JMSFactoryService to JMSSubscriberAddressingLocator. After locating the JMSSubscriberService, it passes itself implementing NotifyCallback interface as a parameter to the subscribe function exposed by JMSSubscriberService. Then the JMSSubscriberService activates the real durable subscribe operation provided by PFJM instance. Figure 3-8 depicts the sequence diagram of subscribe mechanism.

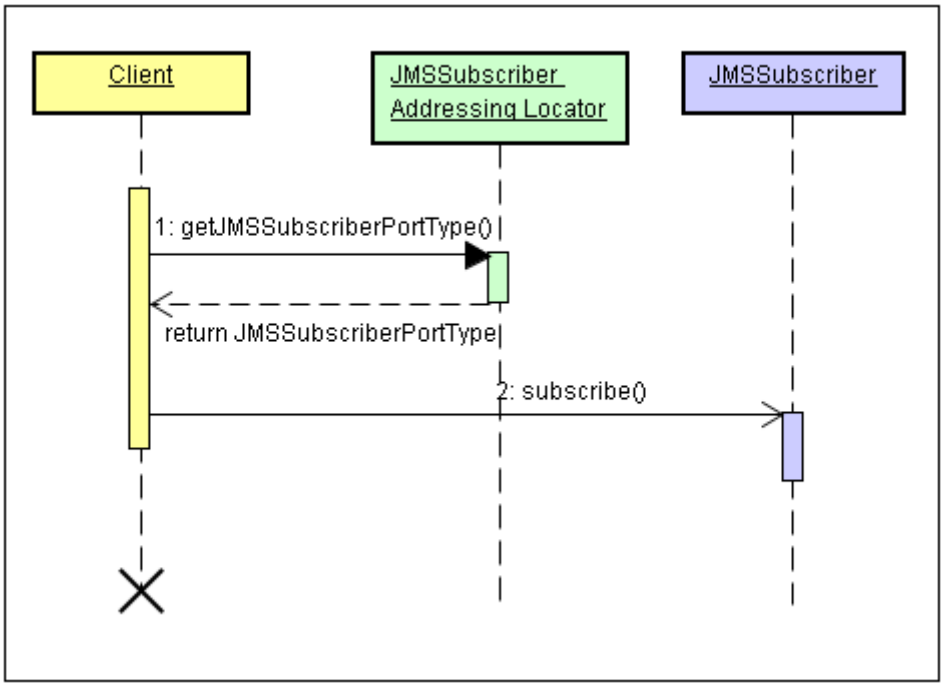


Figure 3-8 The Sequence Diagram of Subscribe Mechanism

3.4.3 Delivery Mechanism

As long as messages are sent to Topic, the PFJM core will invoke the **onMessage** method which JMSSubscriberService implements. Then JMSSubscriberService will invoke the callback function **deliver** implemented by the client. Finally the client will receive the messages. Figure 3-9 depicts the sequence diagram of delivery mechanism.

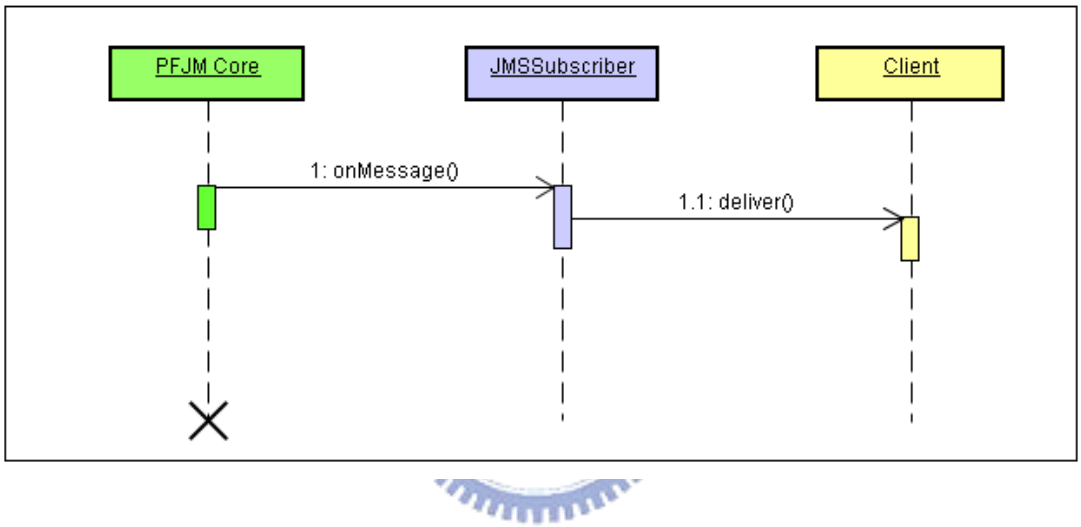


Figure 3-9 The Sequence Diagram of Delivery Mechanism

3.4.4 Recovery Mechanism

Now let's take a look at how the recovery mechanism works when the receiver crashes or the network fails and later the receiver revives again. In JMS lingo, when a client wants to receive reliable messages, it must register a durable subscription with a unique identity also known as subscription name that is retained by the JMS provider. Subsequent subscriber objects with the same identity resume the subscription in the state in which it was left by the previous subscriber. If a durable subscription has no active subscriber, the JMS provider retains the subscription's messages until they are received by the subscription or until they expire.

In PFJM WS, the receiver also passes a subscription name in addition to a topic name to the JMSFactoryService to register itself. After getting the EPR of JMSSubscriberService returning from JMSFactoryService, the receiver can use the EPR to locate JMSSubscriberService and then subscribe to the specific topic with the subscription name. Then the JMS provider will be in charge of sending messages reliably to the receiver.

If the network fails, the JMS provider will store the messages published to the topic. Until the receiver revives with the same topic and subscription name, the subscription will be reactivated, and the JMS provider will deliver the messages that are published while the subscriber is inactive.

Chapter 4 Comparison of Programming Styles

In this chapter, we will discuss GT4 Java WS core and the integrated system, PFJM WS, which we provide to support reliable messaging. We will give a grid service application in Section 4.1. Based on the example described in Section 4.1, we will give scenarios respectively for PFJM WS in Section 4.2 and for GT4 Java WS core in Section 4.3. In Section 4.4 and Section 4.5, we will discuss the detail implementation about using messaging mechanism respectively for PFJM WS and for GT4 Java WS core. Finally, we will give a discussion about the comparison of programming styles between PFJM WS and GT4 Java WS core.

4.1 A Gird Service Application

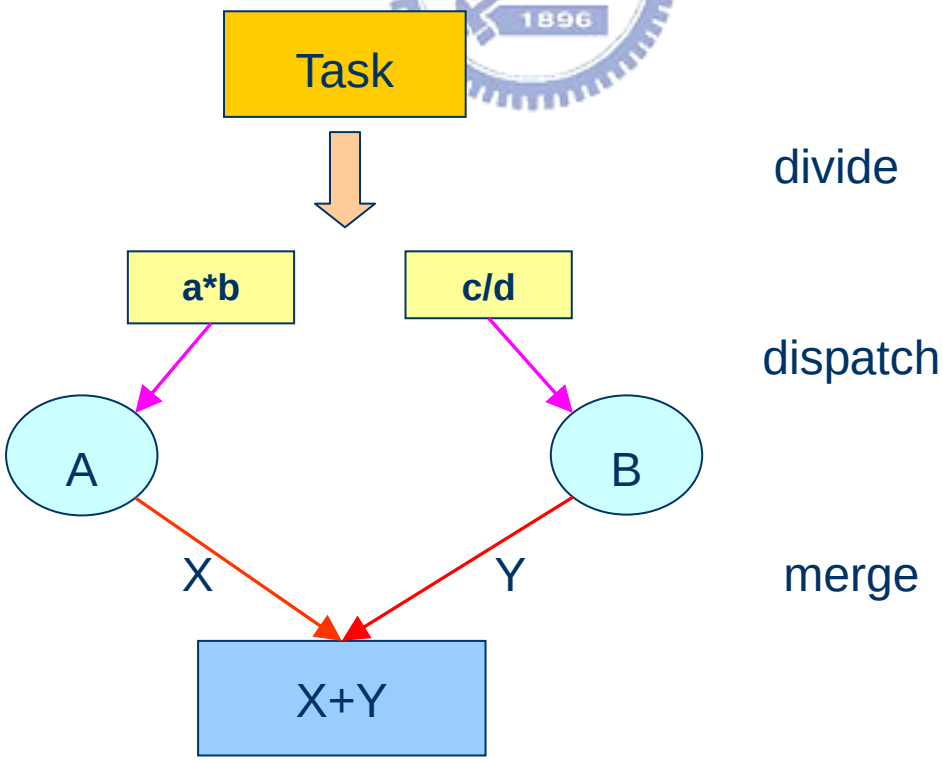


Figure 4-1 A Grid Service Application

In Grid Computing domain, there is a famous example called divide-and-conquer [34]. As shown in Figure 4-1, when the arithmetic grid service responsible for the four fundamental operations of arithmetic receives a task, it then **divides** the task into two subtasks, a multiplication subtask and a division subtask. Afterward, the arithmetic grid service will locate two other grid services respectively responsible for multiplication and division and **dispatch** the divided subtasks to those. After the multiplication and division grid service finish their operation, they will pass the results to the arithmetic grid service. Finally the arithmetic grid service will **merge** the results and return to the client.

4.2 The Scenario for PFJM WS

Here we will use the example in Section 4.1 to describe how to implement the scenario from PFJM WS.

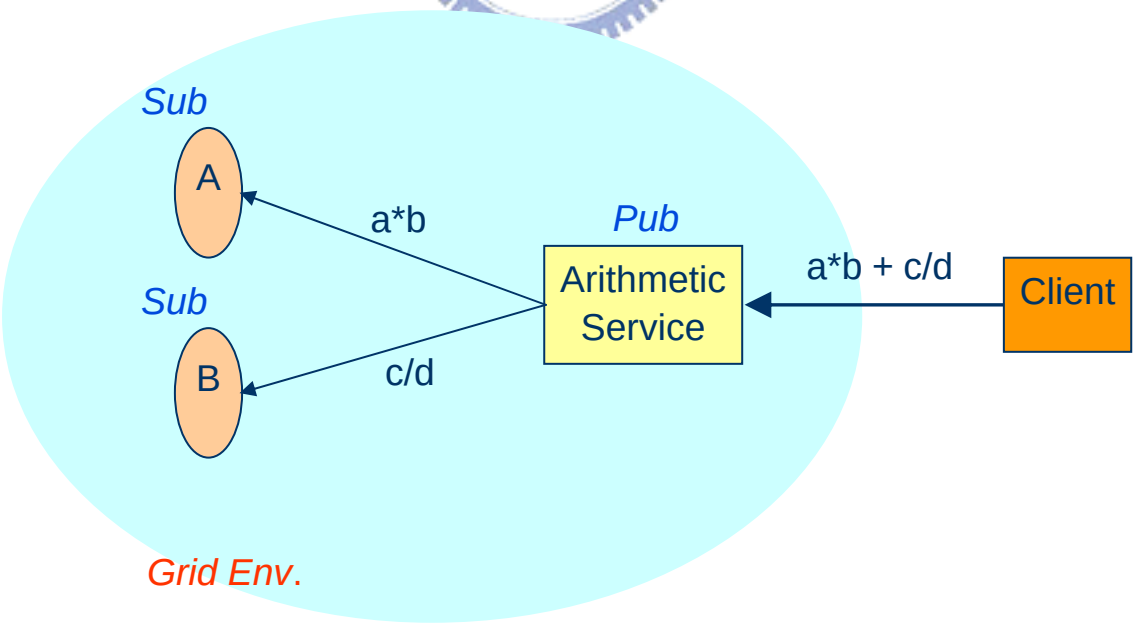


Figure 4-2 The Scenario for PFJM WS - 1

As shown in Figure 4-2, after a client submits a task to the Arithmetic Service, the service internally divides the task into two subtasks and locates two other grid services responsible for multiplication and division services. Then the Arithmetic Service being a publisher dispatches the two subtasks to the multiplication service and division service being subscribers.

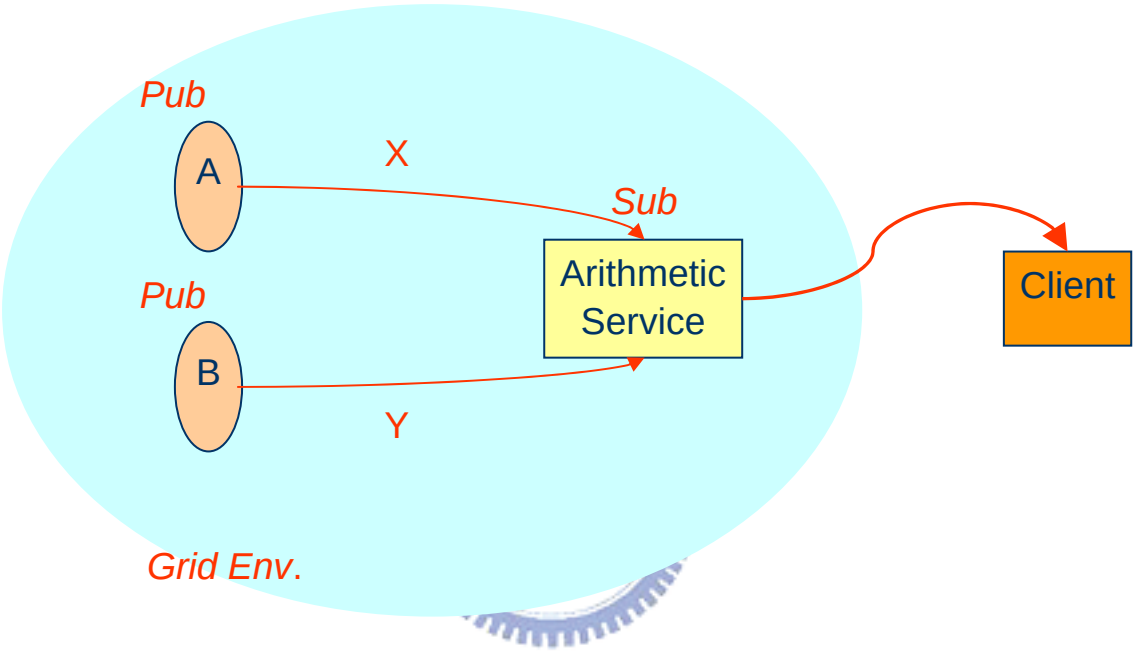


Figure 4-3 The Scenario for PFJM WS – 2

Once the multiplication and division services finish their work, they will become publishers and return the results to the Arithmetic Service being a subscriber. And the Arithmetic Service will merge the results and finally return to the client. Figure 4-3 depicts the scenario.

4.3 The Scenario for GT4 Java WS Core

Here we will also use the example in Section 4.1 to describe how to implement the scenario from GT4 Java WS core.

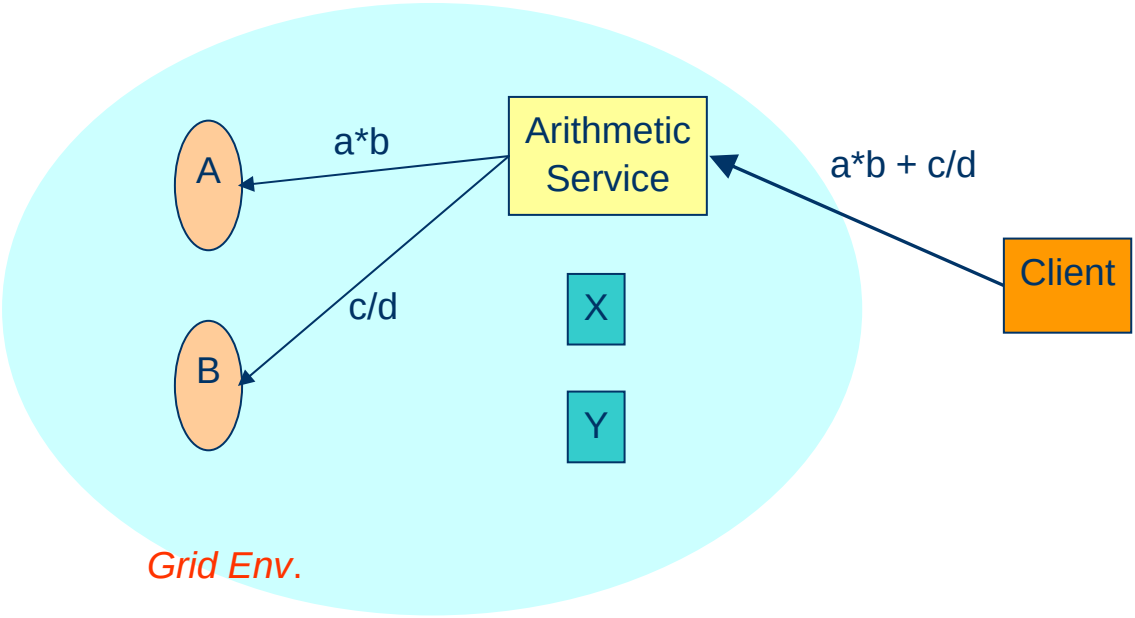


Figure 4-4 The Scenario for GT4 Java WS Core – 1

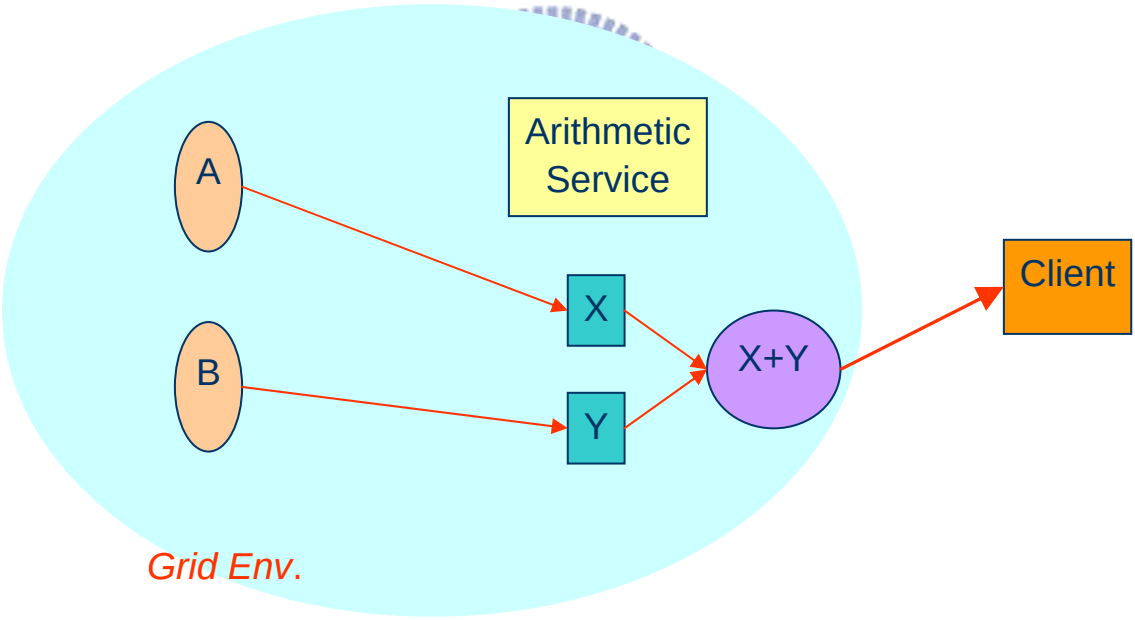


Figure 4-5 The Scenario for GT4 Java WS Core - 2

As shown in Figure 4-4, after a client submits a task to the Arithmetic Service, the service internally divides the task into two subtasks and declares two resources, X and Y, standing for the results of the two subtasks. The Arithmetic Service also locates two other grid services responsible for multiplication and division services and

dispatches the two subtasks to them.

Once the multiplication and division services finish their work, they will utilize the operation provided by the Arithmetic Service to update the properties of the resources. While the Arithmetic Service receives the notification about changing of resources properties from Java WS core, it finally merges the results and returns to the client. Figure 4-5 depicts the scenario.

4.4 The Detail Implementation for PFJM WS

As described in Chapter 3, we introduce the architecture of my system called “PFJM WS”, and show how the components of PFJM WS work in details. Here we give implementation details for PFJM WS about messaging. Figure 4.1 demonstrates the scenario for users sending and receiving messages with PFJM WS.

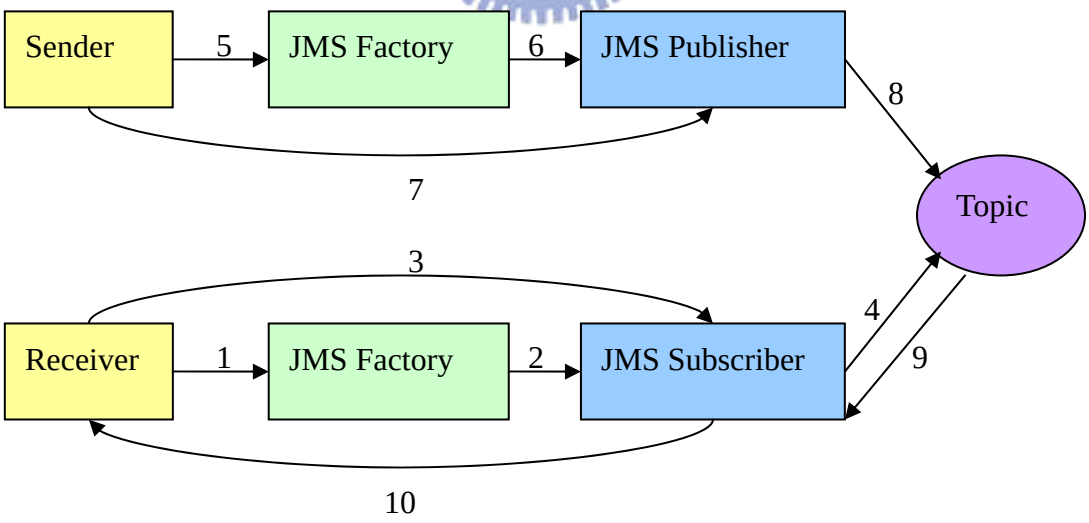


Figure 4-6 The Detail Implementation for PFJM WS

The following describes the process step by step：

1. First the receiver uses EPR of JMSFactory to locate JMSFactoryService.

```
JMSFactoryServiceAddressingLocator instanceLocator =
    new JMSFactoryServiceAddressingLocator();
EndpointReferenceType instanceEPR;

// First argument contains a URI
String serviceURI = args[0];
// Create endpoint reference to service
instanceEPR = new EndpointReferenceType();
instanceEPR.setAddress(new Address(serviceURI));

// Get PortType
JMSFactoryPortType jmsFactory =
    instanceLocator.getJMSFactoryPortTypePort(instanceEPR);
```

Figure 4-7 The Detail Implementation for PFJM WS - Step 1

2. After locating JMSFactoryService, the receiver uses JMSFactoryService to create a JMSSubscriberService and gets the EPR of JMSSubscriberService.

```
EndpointReferenceType subscriberEPR;
// Use topic name and subscription name to register a durable
// subscription
subscriberEPR = jmsFactory.createJMSSubscriberService(topic ,
   subscriptoinName);
```

Figure 4-8 The Detail Implementation for PFJM WS - Step 2

3. The receiver uses the EPR of JMSSubscriberService to locate JMSSubscriberService.

```
JMSSubscribeServiceAddressingLocator instanceLocator = new
    JMSSubscribeServiceAddressingLocator();

// Get PortType
JMSSubscribePortType jmsSubscribe = instanceLocator
    .getJMSSubscribePortTypePort(subscriberEPR);
```

Figure 4-9 The Detail Implementation for PFJM WS - Step 3

4. The receiver uses the JMSSubscriberService to subscribe to the specific topic.

```
jmsSubscribe.jmsSubscribe(this);
```

Figure 4-10 The Detail Implementation for PFJM WS - Step 4

5. On the other side, the sender also uses EPR of JMSFactory to locate JMSFactoryService.

```
JMSFactoryServiceAddressingLocator instanceLocator =
    new JMSFactoryServiceAddressingLocator();
EndpointReferenceType instanceEPR;

// First argument contains a URI
String serviceURI = args[0];
// Create endpoint reference to service
instanceEPR = new EndpointReferenceType();
instanceEPR.setAddress(new Address(serviceURI));

// Get PortType
JMSFactoryPortType jmsFactory =
    instanceLocator.getJMSFactoryPortTypePort(instanceEPR);
```

Figure 4-11 The Detail Implementation for PFJM WS - Step 5

6. After locating JMSFactoryService, the sender uses JMSFactoryService to create a JMSPublisherService and gets the EPR of JMSPublisherService.

```
EndpointReferenceType publisherEPR;
// Use topic name to create a publisher
publisherEPR = jmsFactory.createJMSPublisherService(topic);
```

Figure 4-12 The Detail Implementation for PFJM WS - Step 6

7. The sender uses the EPR of JMSPublisherService to locate JMSPublisherService.

```
JMSPublishServiceAddressingLocator instanceLocator = new
    JMSPublishServiceAddressingLocator();

// Get PortType
JMSPublishPortType jmsPublish = instanceLocator
    .getJMSPublishPortTypePort(publisherEPR);
```

Figure 4-13 The Detail Implementation for PFJM WS - Step 7

8. The sender uses the JMSPublisherService to publish messages to the specific topic.

```
jmsPublish.jmsPublish(mesg);
```

Figure 4-14 The Detail Implementation for PFJM WS - Step 8

9. When messages arrive to the specific topic, the callback function of JMSSubscriberService will be invoked.

```
public void onMessage(Message mesg) {
    String text = "";
    try{
        TextMessage textMessage = (TextMessage) mesg;
        text = textMessage.getText();
    } catch(JMSException jmse) {
        jmse.printStackTrace();
    }

    NotificationConsumerServiceAddressingLocator consumerLocator =
        new NotificationConsumerServiceAddressingLocator();
    try{
        Consumer consumerPort =
            consumerLocator.getConsumerPort(consumer);

        org.oasis.wsn.Notify notification =
           new org.oasis.wsn.Notify();
        NotificationMessageHolderType[] message = {
            new NotificationMessageHolderType()};

        EndpointReferenceType producerEndpoint =
            new EndpointReferenceType();
        Address producerAddress = new
            Address("http://localhost/client");
        producerEndpoint.setAddress(producerAddress);

        TopicExpressionType topic = new TopicExpressionType(
           WSNConstants.SIMPLE_TOPIC_DIALECT,
           PSQNames.TOPIC_1);
```

Figure 4-15 The Detail Implementation for PFJM WS - Step 9.1

```
            message[0].setProducerReference(producerEndpoint);
            message[0].setMessage(text);
            message[0].setTopic(topic);
            notification.setNotificationMessage(message);
            consumerPort.notify(notification);
        } catch (Exception e){
            e.printStackTrace();
        }
}
```

Figure 4-16 The Detail Implementation for PFJM WS - Step 9.2

10. The JMSSubscriberService will finally notify the receiver that the messages have already arrived. And the receiver can handle these messages now.

```
public void deliver(List topicPath, EndpointReferenceType producer,
                Object message) {
    if (Num_Message == 1)
        stime = System.currentTimeMillis();
    else if(Num_Message == Total_Message) {
        etime = System.currentTimeMillis();

        // output
        System.out.println("Throughput:\t" + (Data_Size *
           (Total_Message-1) *1000.0)/(etime-stime));

        // reset
        Num_Message = 1;
        return;
    }
    Num_Message++;
}
```

Figure 4-17 The Detail Implementation for PFJM WS - Step 10

4.5 The Detail Implementation for GT4 Java WS Core

From Chapter 2, we know that the existence of Web Services Notifications in GT4 Java WS core is to notify changes of resource property to clients who need that info. As shown in Figure 4-2, for example, there is a grid service, SystemHealthService, returning to users if the system is healthy. In this example, the SystemHealthService also needs other grid services including CPUService returning if the CPU usage is greater than 95% and StorageService returning if the storage usage is greater than 90%. Here “CPU usage” and “Storage usage” are regarded as resource properties. When the resource properties change, the change info will be sent to SystemHealthService and SystemHealthService will use that info to construct its service’s logistic.

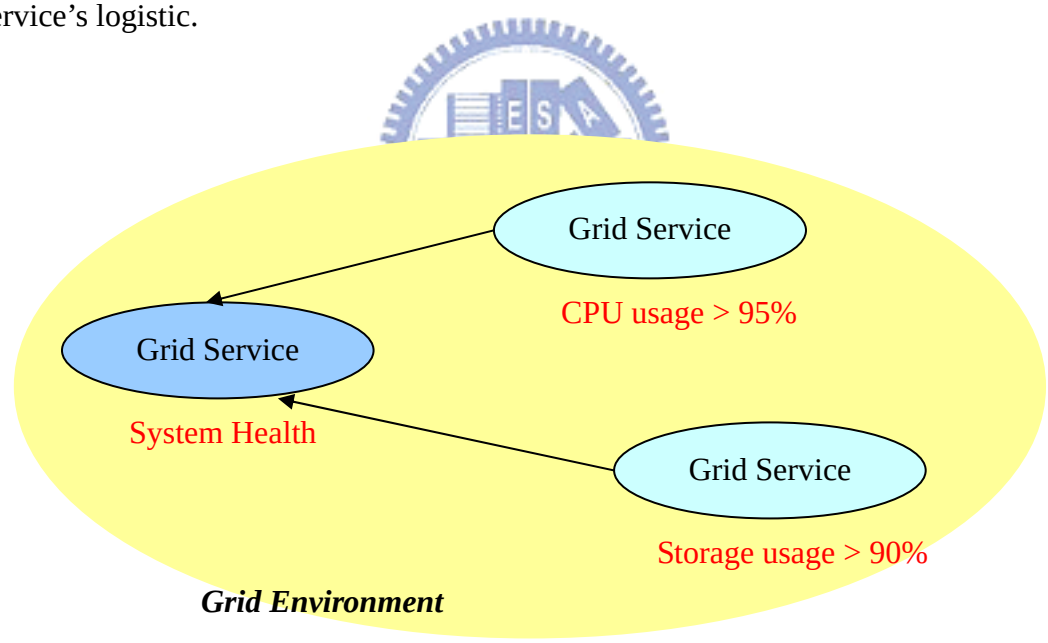


Figure 4-18 WSN in GT4 Java WS Core

Now let us take a look at how users utilize GT4 Java WS core to send and receive messages. Figure 4.18 demonstrates the scenario for users sending and receiving messages with GT4 Java WS core. It deserves to be mentioned that the resource property is the so-called Topic in JMS and sending messages to Topic means to change the value of the resource property.

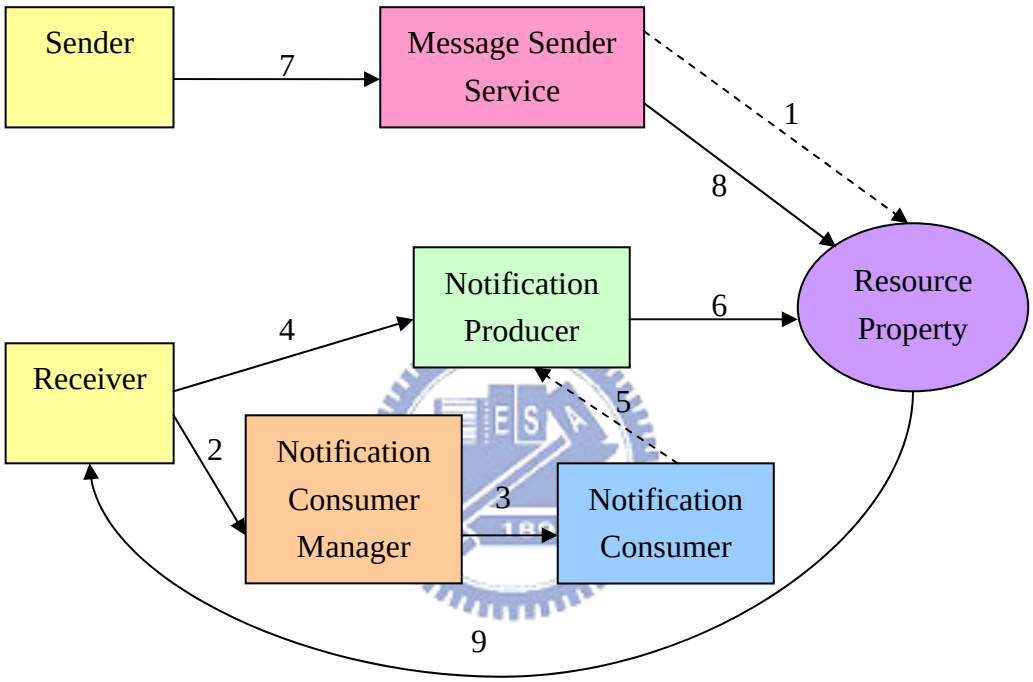


Figure 4-19 The Detail Implementation for GT4 Java WS Core

The following describes the process step by step：

1. First, when the service container starts, the MessageSenderService developed by a user will declare a resource property which is regarded as a Topic and used for message communication.

```
/* Resource Property set */
private ResourcePropertySet propSet;

/* Resource properties */
private ResourceProperty mesgRP;

/* Topic list */
private TopicList topicList;

/* Create RP set */
this.propSet = new SimpleResourcePropertySet(
        PubSubQNames.RESOURCE_PROPERTIES);

/* Initialize the RP's */
try {
    mesgRP = new SimpleResourceProperty(PubSubQNames.RP_MESG);
    mesgRP.add("no mesg");
} catch (Exception e) {
    throw new RuntimeException(e.getMessage());
}

/* Configure the Topics */
this.topicList = new SimpleTopicList(this);

mesgRP = new ResourcePropertyTopic(mesgRP);
((ResourcePropertyTopic) mesgRP).setSendOldValue(true);

this.topicList.addTopic((Topic) mesgRP);
this.propSet.add(mesgRP);
```

Figure 4-20 The Detail Implementation for GT4 Java WS Core – Step 1

2. The receiver is going to act as a notification consumer. This means that the receiver will have to expose a callback function that will be invoked by the notification producer. For this to happen, the receiver has to act as both a client and a server. Fortunately, thanks to a Globus-supplied class called NotificationConsumerManager, it will help us to do so. In this step, the receiver creates a NotificationConsumerManager.

```
NotificationConsumerManager consumer;

consumer = NotificationConsumerManager.getInstance();
consumer.startListening();
```

Figure 4-21 The Detail Implementation for GT4 Java WS Core – Step 2

3. The receiver uses the NotificationConsumerManager created in step 2 to create a NotificationConsumer and assigns itself implementing ***NotifyCallback*** interface as a parameter to the NotificationConsumer which will act as a client. And then the receiver will get the EPR of NotificationConsumer since the EPR will be used by the NotificationProducer to deliver the messages.

```
EndpointReferenceType consumerEPR = consumer
                .createNotificationConsumer(this);
```

Figure 4-22 The Detail Implementation for GT4 Java WS Core – Step 3

4. The receiver uses the EPR of NotificationProducer to locate the NotificationProducerService.

```
// Get a reference to the NotificationProducer portType
WSBaseNotificationServiceAddressingLocator notifLocator =
        new WSBaseNotificationServiceAddressingLocator();
EndpointReferenceType endpoint = new EndpointReferenceType();
endpoint.setAddress(new Address(serviceURI));
NotificationProducer producerPort =
    notifLocator.getNotificationProducerPort(endpoint);
```

Figure 4-23 The Detail Implementation for GT4 Java WS Core – Step 4

5. The receiver passes the EPR of NotificationConsumer gotten in step 3 as a parameter to the located NotificationProducer.

```
// Create the request to the remote Subscribe() call
Subscribe request = new Subscribe();

// Must the notification be delivered using the Notify operation
request.setUseNotify(Boolean.TRUE);

// Indicate what the client's EPR is
request.setConsumerReference(consumerEPR);

// The TopicExpression specifies what topic we want to subscribe
to
TopicExpressionType topicExpression = new TopicExpressionType();

topicExpression.setDialect(WSNConstants.SIMPLE_TOPIC_DIALECT);
topicExpression.setValue(PubSubQNames.RP_MESG);
request.setTopicExpression(topicExpression);
```

Figure 4-24 The Detail Implementation for GT4 Java WS Core – Step 5

6. In this step the receiver actually sends the subscription request to GT4 Java WS core by invoking the method ***subscribe*** of NotificationProducer. GT4 Java WS core will then monitor if the resource properties change or not.

```
// Start the ball rolling...
producerPort.subscribe(request);
```

Figure 4-25 The Detail Implementation for GT4 Java WS Core – Step 6

7. The sender uses the EPR of MessageSenderService to locate the MessageSenderService.

```
PubSubServiceAddressingLocator instanceLocator = new
    PubSubServiceAddressingLocator();
try {
    EndpointReferenceType instanceEPR;
    if (args[0].startsWith("http")) {
        String serviceURI = args[0];
        // Create endpoint reference to service
        instanceEPR = new EndpointReferenceType();
        instanceEPR.setAddress(new Address(serviceURI));
    } else {
        // First argument contains an EPR file name
        String eprFile = args[0];
        // Get endpoint reference of WS-Resource from file
        FileInputStream fis = new FileInputStream(eprFile);
        instanceEPR = (EndpointReferenceType)
        ObjectDeserializer.deserialize(new InputSource(fis),
                                EndpointReferenceType.class);
        fis.close();
    }
    // Get PortType
    PubSubPortType pubsub = instanceLocator
        .getPubSubPortTypePort(instanceEPR);
```

Figure 4-26 The Detail Implementation for GT4 Java WS Core – Step 7

8. The sender executes the operation provided by MessageSenderService to change the value of the resource property declared initially. Actually changing the value of resource property means sending messages to the specific topic.

```
    // Perform operation
    for (int i = 0; i < Total_Message; i++)
        pubsub.publish(sb.toString());
```

Figure 4-27 The Detail Implementation for GT4 Java WS Core – Step 8

9. If GT4 Java WS core detects that the resource property is changed, the receiver will be notified and received the changed value which is the message sent by the sender.

```
    /* This method is called when a notification is delivered */
    public void deliver(List topicPath, EndpointReferenceType
        producer, Object message) {
        if (Num_Message == 1)
            stime = System.currentTimeMillis();
        else if(Num_Message == Total_Message) {
            etime = System.currentTimeMillis();

            // output
            System.out.println("Throughput:\t" + (Data_Size *
(Total_Message-1) *1000.0)/(etime-stime));

            // reset
            Num_Message = 1;
            return;
        }
        Num_Message++;
    }
```

Figure 4-28 The Detail Implementation for GT4 Java WS Core – Step 9

4.6 Discussion

From Section 4.1 to Section 4.5, we can know that if a client wants to use messaging mechanism based on GT4 Java WS core, it can achieve this goal by means of modifying the resource properties and receiving the notification. This is not intuitive from the point of view of a programmer since messages are received passively. Besides, the API of GT4 Java WS core is complex for a programmer and a lot of processes must be done by the client himself. On the contrary, if a client uses PFJM WS to do message communication, it is not only reasonable for the point of the view of a programmer, but the programmer also can use the API provided by PFJM WS more easily than GT4 Java WS core.

Chapter 5 Experiment

In this chapter, we will introduce the performance test between original GT4 Java WS core and the integrated system, PFJM WS, we provide to support reliable messaging. The experiment environment will be described in Section 5.1 and the experiment results are described in Section 5.2. Finally we will give some discussion about the experiment in Section 5.3.

5.1 Experiment Environment

During the experiment, we use the FX-05EA 5 Ports 10/100M Switching Hub to form a local area network (LAN) with one notebook and three PCs connecting to a 100 Mbps fast Ethernet. The hardware and software specifications of these PCs are depicted by the Table 5-1.

	Notebook	PC1	PC2	PC3
CPU	PM 1.6GHz	P4 2.40GHz	P4 2.40GHz	P4 2.40GHz
Memory	760MB	512MB	512MB	512MB
NIC	Intel(R) PRO/100 VM Network Connection	ASUSTeK/Broadcom 440x 10/100 Integrated Controller	ASUSTeK/Broadcom 440x 10/100 Integrated Controller	ASUSTeK/Broadcom 440x 10/100 Integrated Controller
OS	Windows XP Service Pack 2	Windows XP Service Pack 2	Windows XP Service Pack 2	Windows XP Service Pack 2
Java	Sun JDK 1.4.2-b28	Sun JDK 1.4.2_03-b02	Sun JDK 1.5.0-06-b05	Sun JDK 1.4.2_08-b03

Table 5-1 The Hardware and Software Specifications

5.2 Experiment Results

We divide the experiment into two categories: The first is one-to-one communication using notebook as a message sender and PC1 as a message receiver. The second is one-to-many communication also using notebook as a message sender and PC1~PC3 as message receivers.

5.2.1 The Throughput for One-to-One Communication

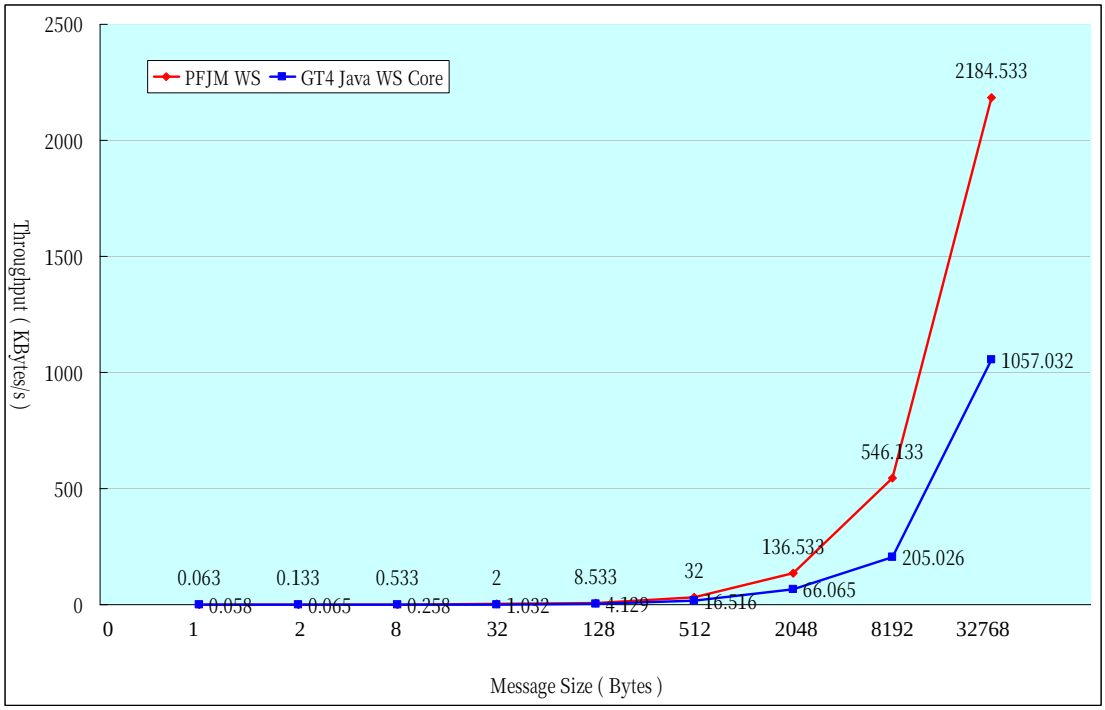


Figure 5-1 The Throughput for One-to-One Communication

5.2.2 *The Throughput for One-to-Many Communication*

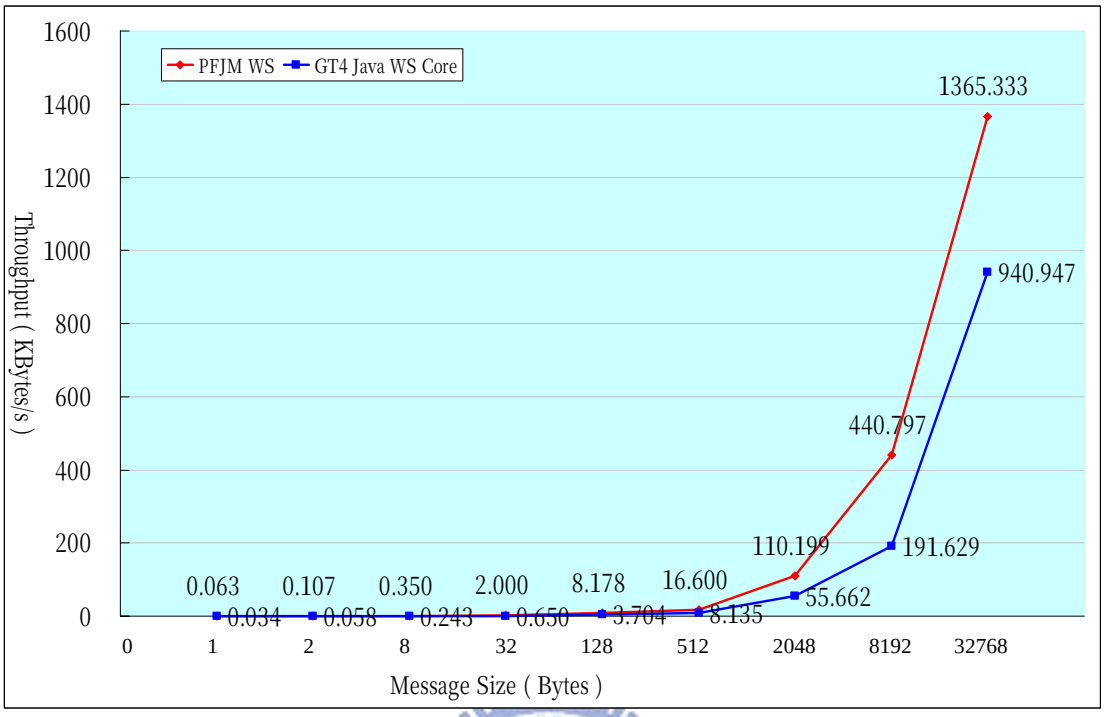


Figure 5-2 The Throughput for One-to-Many Communication

5.3 Discussion

The benchmark is performed on 2 PCs for one to one communication and 4 PCs for one to three communications in May 2006. We use the default setting for PFJM and measure throughput of different data sizes for GT4 Java WS core and PFJM WS. For each data sizes, we perform one hundred times message transmission and calculate the average throughput in bytes per second. As shown in Figure 5-1 and Figure 5-2, the x-axis stands for data size in bytes and the y-axis stands for throughput in Kbytes per second. We can see the experiment results from Section 5.2. The throughput of PFJM WS is better than GT4 Java WS core in both cases. If a user wants to use reliable messaging in GT4 Java WS environment, our PFJM WS providing a convenient manner and having nice efficiency is a good choice.

Chapter 6 Conclusion and Future Works

6.1 Conclusion

With the rising of grid technology, GT4 has become the most popular tool in the industry. However the Java WS core in GT4 has a serious drawback that is not support reliable messaging. But in grid environment reliable messaging is expectable since lose messages may cause critical effects. Fortunately Java Messaging Service (JMS) designed by Sun Microsystems and several other companies has an excellent advantage that does not support by GT4 Java WS core. The advantage is messages are guaranteed to be successfully consumed once and only once.

Due to the superiority of JMS, we adopt PFJM, a JMS compliant product developed by our lab, as the internal communication framework in GT4 Java WS core. Thus we wrap PFJM into PFJM WS to provide useful tools and reasonable programming styles than GT4 Java WS core about sending and receiving messages discussed in Chapter 4 since messages in GT4 Java WS core are always resource properties. And we can see using PFJM WS to send and receive messages is more effective than GT4 Java WS core discussed in Chapter 5. In a short word, the PFJM WS not only provides a better programming style and a reliable messaging mechanism but also is more effective than GT4 Java WS core about message communication.

6.2 Future Works

Web Services Reliability (WS-Reliability) [28] is an OASIS standard and is announced on 15 November 2004. The purpose of the OASIS WSRM TC is to create a generic and open model for ensuring reliable message delivery for Web Services. WS-Reliability is a SOAP-based protocol for exchanging SOAP messages with guaranteed delivery, no duplicates, and guaranteed message ordering. WS-Reliability is defined as SOAP header extensions and is independent of the underlying protocol.

Although our developed PFJM WS is noted for reliable messaging but it doesn't follow WS-Reliability yet. We know that following standard specification brings many advantages such as portability so in the future we may exploit PFJM WS and follow WS-Reliability to extend GT4 Java WS core to have reliable messaging.

Now we give some analysis about the benefit and the influence of implementing WS-Reliability by means of PFJM WS. According to OASIS, WS-Reliability 1.1 supports "guaranteed delivery". In other words, it ensures that a message is delivered at least once. It also eliminates duplication, certifying that a message was delivered just once. And it provides message delivery ordering, which guarantees that messages in a sequence are delivered in the order sent, according to OASIS. Fortunately, JMS 1.1 also defines reliability of messaging. As shown in Table 6-1, if a client adopts "Durable Subscriber" and "PERSISTENT" options simultaneously to do message communication, it can achieve "once-and-only-once" effect the same as WS-Reliability. Moreover, JMS defines that messages sent by a session to a destination must be received in the order they were sent. From the characteristics of JMS, it seems that JMS has the natural instinct to implement WS-Reliability.

How Published	**Non-Durable Subscriber**	**Durable Subscriber**
NON_PERSISTENT	at-most-once (missed if inactive)	at-most-once
PERSISTENT	once-and-only-once (missed if inactive)	once-and-only-once

Table 6-1 Pub/Sub Reliability

Besides, due to WS-Reliability is a SOAP-based protocol, everything we have to do is to design SOAP header and messaging models according to WS-Reliability specification (as shown in Figure 6-1 and Figure 6-2) and modify the SOAP processor in GT4 Java WS core to handle SOAP message. From the point of view of a client, he still uses the same operation PFJM WS provides to do message communication and does not need to understand how the WS-Reliability is implemented into PFJM WS.

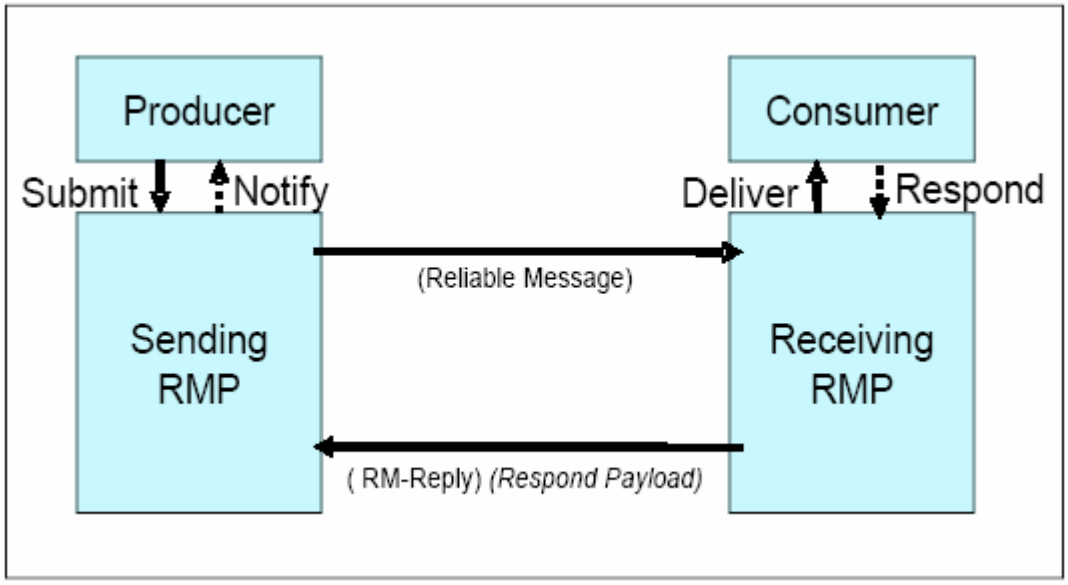


Figure 6-1 Messaging Model

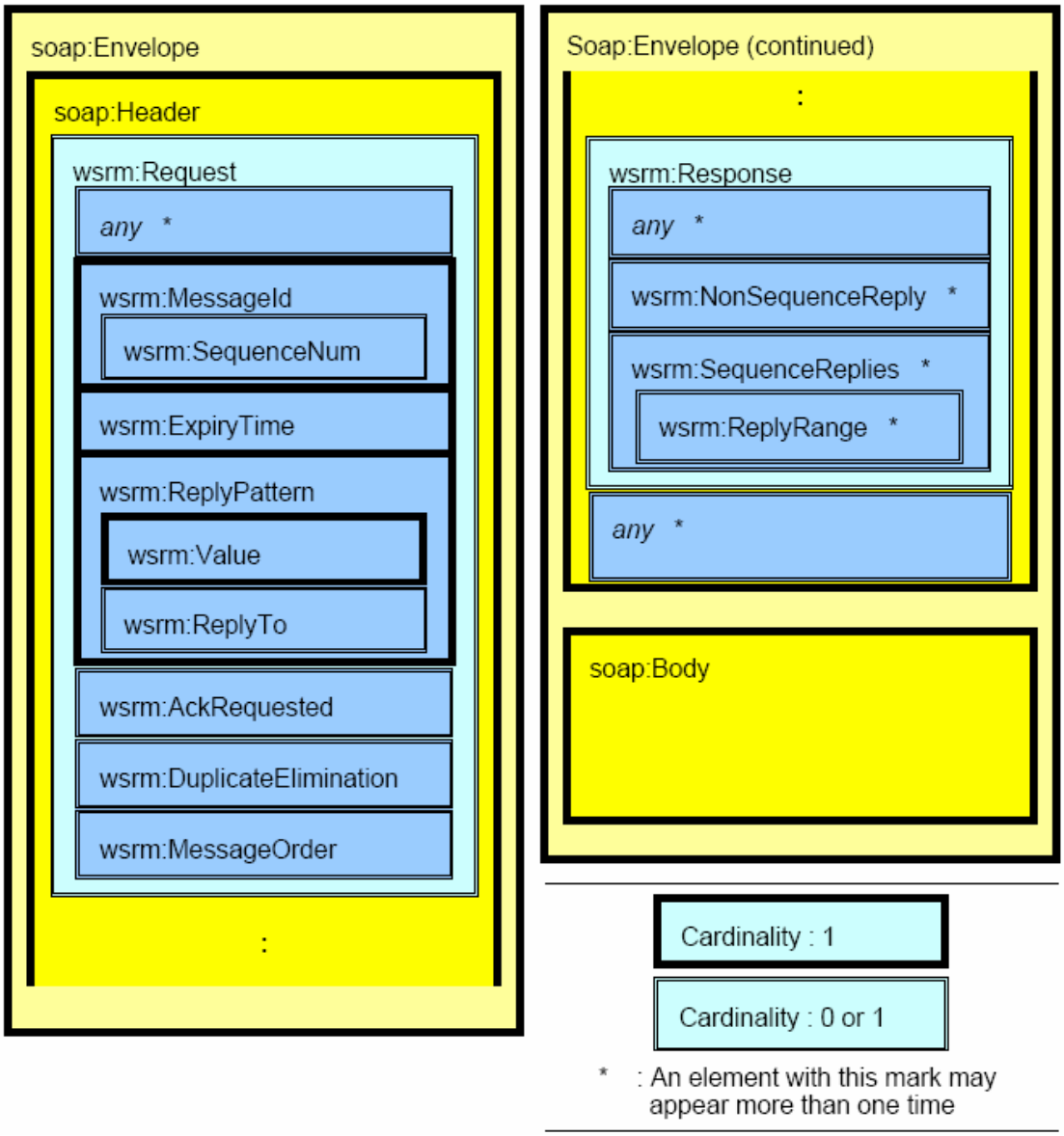


Figure 6-2 Structure of WS-Reliability elements

Bibliography

[1] Markku Korhonen, “Message Oriented Middleware”, Tik-110.551 Internetworking Seminar, Department of Computer Science, Helsinki University of Technology, http://www.tml.tkk.fi/Opinnot/Tik-110.551/1997/mqs.htm

[2] “Remote procedure call specification”, Sun Microsystems, Mountain View, CA, Jan. 1985.

[3] Sun Microsystems. “Java Message Service”, Version 1.1, April 2002.

[4] “SonicMQ”, http://www.sonicsoftware.com

[5] “FioranoMQ”, http://www.fiorano.com

[6] “OpenJMS”, http://openjms.sourceforge.net

[7] Chuan-Pao Hung, Hsin-Ta Chiao, Yue-Shan Chang, Tsun-Yu Hsiao, Tzu-Han Kao, Shyan-Ming Yuan , “FJM: A Fast Java Message Delivery Mechanism based on IPMulticast”, Third International Conference on Communications in Computing (CIC 2002), Monte, June 24 – 27, 2002.

[8] Yu-Fang Huang, Tsun-Yu Hsiao, Shyan-Ming Yuan. “A Java Message Service with Persistent Message”, Proceeding of Symposium on Digital Life and Internet Technologies 2003.

[9] “Globus Toolkit”, http://www.globus.org/toolkit/

[10] “GT 4.0: Java WS Core”, http://www.globus.org/toolkit/docs/4.0/common/javawscore/

[11] Globus Alliance, IBM, and HP, “Web Services Resource Framework”, http://www.globus.org/wsrf/, January 20, 2004.

[12] Vladimir Silva, Contractor, Pervasive Systems Development, “Globus Toolkit 4 Early Access: WSRF”,

http://www-128.ibm.com/developerworks/cn/grid/gr-gt4early/, October 26, 2004.

[13] K. Czajkowski, DF Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S.Tuecke, and W. Vambenepe, “The WS- Resource Framework”, http://www.globus.org/wsrf/specs/ws-wsrf.pdf, March 2004.

[14] Mike Weaver, “Web Service Notification”, http://www-128.ibm.com/developerworks/cn/grid/gr-ws-not/, February 06, 2005.

[15] “JSR-000914 Java™ Message Service”, http://www.jcp.org/aboutJava/communityprocess/final/jsr914/index.html

[16] Frank Siebenlist, Von Welch, “Overview of GT4 Security”, http://www.globus.org/toolkit/presentations/GlobusWorld_2005_Session_3c.pdf

[17] Bill Allcock, Ann Chervenak, Neil P. Chue Hong, EPCC, “Overview of GT4 Data Management”, http://www.globus.org/toolkit/presentations/GlobusWorld_2005_Session_1c.pdf

[18] Ben Clifford, “GT4 Monitoring and Discovery”, http://www.globus.org/toolkit/presentations/GlobusWorld_2005_Session_9c.pdf

[19] Karl Czajkowski, “Overview of GT4 Execution Management”, http://www.globus.org/toolkit/presentations/GlobusWorld_2005_Session_2c.pdf

[20] “Common Runtime”, http://www.globus.org/toolkit/common/

[21] I. Foster, “Globus Toolkit Version 4: Software for Service-Oriented Systems.”, IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779, pp 2-13, 2005.

[22] “Web services”, http://www.w3.org/2002/ws/

[23] W3C Note, “Web Services Description Language 1.1”, http://www.w3.org/TR/wsdl, March 15, 2001.

[24] W3C Note, “Simple Object Access Protocol 1.1”, http://www.w3.org/TR/soap/, May 08, 2000.

[25] OASIS Working Draft 02, “WS-Resource 1.2”, http://docs.oasis-open.org/wsrf/2004/11/wsrf-WS-Resource-1.2-draft-02.pdf, December 9, 2004.

[26] Steve Graham, Karl Czajkowski, Donald F Ferguson, Ian Foster, Jeffrey Frey, Frank Leymann, Tom Maguire, Nataraj Nagaratnam, Martin Nally, Tony Storey, Igor Sedukhin, David Snelling, Steve Tuecke, William Vambenepe, Sanjiva Weerawarana, “WS-Resource Properties 1.1”, http://www.ibm.com/developerworks/library/ws-resource/ws-resourceproperties.pdf, March 15, 2003.

[27] W3C Member Submission, “WS-Addressing”, http://www.w3.org/Submission/ws-addressing/, 10 August 2004

[28] OASIS Standard, “WS-Reliability 1.1”, http://docs.oasis-open.org/wsrm/ws-reliability/v1.1/wsrm-ws_reliability-1.1-spec-os.pdf, 15 November 2004.

[29] I. Foster, C. Kesselman, J. Nick, S. Tuecke, “The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration.”, Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002.

[30] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, “The Open Grid Services Architecture, Version 1.0.”, J. Von Reich. Informational Document, Global Grid Forum (GGF), January 29, 2005.

[31] M. Humphrey, G. Wasson, K. Jackson, J. Boverhof, M. Rodriguez, Joe Bester, J. Gawor, S. Lang, I. Foster, S. Meder, S. Pickles, and M. McKeown, “State and Events for Web Services: A Comparison of Five WS-Resource Framework and

WS-Notification Implementations.”, 4th IEEE International Symposium on High Performance Distributed Computing (HPDC-14), Research Triangle Park, NC, 24-27 July 2005.

[32] “GT4_Primer_0.6”, http://www.globus.org/toolkit/docs/4.0/key/GT4_Primer_0.6.pdf

[33] “GT4 Admin Guide”, http://www.globus.org/toolkit/docs/4.0/admin/docbook/admin.pdf

[34] I-Chen Wu , H. T. Kung, “Communication complexity for parallel divide-and-conquer”, Proceedings of the 32nd annual symposium on Foundations of computer science, p.151-162, September 1991, San Juan, Puerto Rico.