A Study on Methods and Tools for Developing Service-Oriented Grid Application

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2008

Kohei ICHIKAWA

Author's Publications for Doctoral Degree Application

A. Journal Paper

1. Kohei Ichikawa, Susumu Date, Takeshi Kaishima and Shinji Shimojo. "A framework supporting the development of Grid portal for analysis based on ROI," *Methods of Information in Medicine*, vol. 44, no.2, pp. 265–269, June 2005.

B. International Conference Papers

1. Kohei Ichikawa, Susumu Date and Shinji Shimojo. "A framework for meta-scheduling WSRF based services," in *Proceedings of 2007 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pp. 481–484, August 2007.

2. Kohei Ichikawa, Susumu Date, Sriram Krishnan, Wilfred Li, Kazuto Nakata, Yasushige Yonezawa, Haruki Nakamura and Shinji Shimojo. "Opal OP: An extensible Grid-enabling wrapping approach for legacy applications," in *Proceedings of 3rd Workshop on Grid Computing and Applications*, pp. 117–127, June 2007.

Summary

The application of service-oriented architecture to Grid technologies is leading a new mode of wide-area distributed computing in scientific and engineering areas such as life science, high-energy physics, and earth science, and a new computing approach has gathered concerns and interests from experts with various research backgrounds. The new computing approach allows such experts to perform loosely-coupled large-scale simulation by seamlessly and flexibly federating Grid services, each of which is built from a computer program and delivers its own function. The application realized by federating multiple Grid services is referred as "service-oriented Grid application" in this dissertation. The service-oriented Grid application has gathered a lot of attention especially in the research fields of the recently emerged multi-scale, multi-physics simulations composed of multiple different simulations, because it complements the missing part of the current computing approach which is a single tightly-coupled large-scale simulation. In reality, however, even scientists with knowledge on Grid technologies have difficulty in developing a Grid service from an existing program due to the complicated configuration and implementation accompanying Grid service development. Also, both the deployment of Grid services onto multiple organizations and the development of a Grid application composed of these Grid services are not a piece of cake. For the reason, methods and tools for facilitating the development of service-oriented Grid application is demanded today.

This dissertation focuses on addressing some of the issues described above. In the area of application development, a new wrapping method that would make an existing program into a service-oriented Grid application with minimal effort is proposed. To facilitate the development based on the new method, a tool called Opal Operation Provider (Opal OP) is developed. The Opal OP allows an application developer to import the functions for handling an existing program into his/her Grid service by utilizing a plug-in technique for Grid service or operation provider. This tool is demonstrated in real use to be effective in providing the application developer with the ease-of-use and flexibility that is not available in other conventional wrapping methods.

Another contribution of this dissertation is the proposal of Meta-Scheduling Services

Architecture (MSSA) and the development of a meta-scheduler as a Grid service to operate in the service-oriented Grid environment. This architecture focuses on providing an interface transparent way for selecting a Grid service among multiple Grid services deployed on the service-oriented Grid environment. The architecture takes advantage of the factory pattern technique, which is used for state management of the Grid service. The experiment in this dissertation proves that MSSA is fault-tolerant to failures of Grid services in execution and the additional overhead for achieving the fault-tolerance is minimal.

This dissertation is organized as follows. In Chapter 1, the background and goal of this study is described. After that, Chapter 2 clarifies the technical issues to achieve in this study through the consideration of the difficulties and problems in Grid application development process. In particular, the developing stage of a Grid service from an existing program and the developing stage of a Grid application composed of multiple Grid services deployed on wide-area computing environment are highlighted.

In Chapter 3, a new method that facilitates the development of a Grid service from an existing program is proposed. The new method reduces the difficulties and problems in Grid application development process. Also, a new tool named Opal OP which helps the development along with the proposed method is proposed and implemented. After that, the usefulness and effectiveness of the proposed method and the tool is discussed through the actual example of the system developed based on the proposed method and tool.

In Chapter 4, a new method named MSSA and the corresponding tool that simplifies the development of service-oriented Grid application composed of multiple Grid services deployed on computing service-oriented Grid environment from standpoints of performance improvement, load balancing, and fault-tolerance enhancement are proposed. Discussion in this dissertation focuses on the usefulness of the proposed MSSA by showing an actual scientific Grid application.

Finally, Chapter 5 concludes this study and discusses directions for future research.

Contents

Chapter 1

Introduction

1.1 Research Background

The computer usage pattern of researchers has been changing since the emergence of the computer. In the era of the mainframe computer, a single large computer was shared with multiple users. Today, conversely, people utilize multiple computers simultaneously to solve their computational problems. Network Of Workstations (NOW) [1] and Beowulf [2-5] are typical examples of such computer usage patterns. The change in computer usage pattern has been driven by the users' infinite pursuit of computational performance and throughput. This trend is supposed to continue from now on.

According to Moore's Law, the number of transistors that can be inexpensively placed on an integrated circuit increases exponentially, and as a result doubles approximately every two years [6]. Similarly Gilder's Law, which pertains to network bandwidth, says that wide-area network capacity doubles every nine months [7]. In fact, processor and networking technology have been advancing following these rules. Figure 1.1 illustrates such a development situation. These two facts mean that the cost of transmitting a bit over a network decreases faster than the performance increase of a processor. Taking these development situations into consideration, much effort has focused on the development of distributed computing systems linking multiple computers on a high-speed network rather than the development of a single large computer.

Distributed computing technologies have also made dramatic improvement in the last few decades. During these decades, a variety of software technologies have been proposed and implemented. RPC (Remote Procedure Call) [8] is a representative example of such technologies. RPC allows a computer program to call a subroutine or procedure of the program executed on a remote computer. MPI (Message Passing Interface) [9] is another typical example of such technologies. MPI is mainly used today for high performance

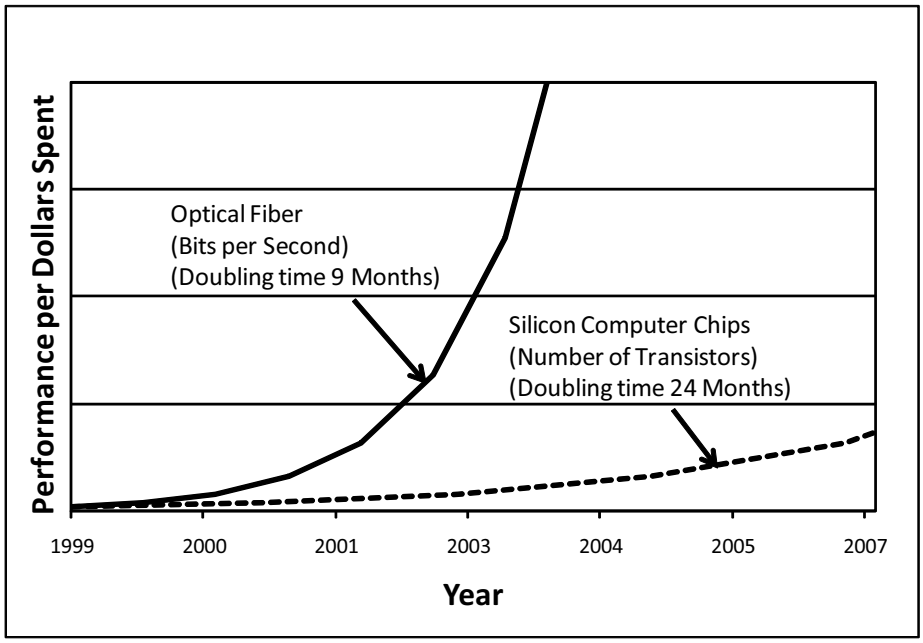

Figure 1.1: Moore's law and Gilder's law

computing on a computing cluster system, and it allows users to easily write program with communication among multiple processes, each of which runs on a computing node of the cluster system. These days, many scientific programs have been developed with MPI for high-performance computing on a cluster system.

Furthermore, the emergence of object-oriented architecture has been accelerating the advancement of distributed computing technologies. In object-oriented architecture, a program is composed of independent components or objects, each of which encapsulates a certain function. Examples of object-oriented architecture include CORBA [10, 11] and DCOM [12]. In this architecture, various components or objects communicate with each other in an architecture-independent manner. Very recently, this distributed object-oriented architecture has transformed to a new architecture inspired by a more sophisticated concept, that is, service-oriented architecture represented by Web service technologies.

The service-oriented architecture puts more focus on the interoperation among different computers. The software components in this architecture are defined as Web services which communicate with each other using XML messages following the SOAP standard. In addition, the interfaces of the Web service are written in a machine readable description, Web Services Description Language (WSDL).

Today, this service-oriented architecture typified by Web service is playing an important role in terms of not only data and information exchange but also business process devel-

opment. The success of the service-oriented architecture typified by Web service in the business area stimulates the necessity of the counterpart in scientific research area. For the reason, the Grid computing technologies, which had been developed as high-performance computing technologies for scientific research, is transforming to new high-performance computing technology that allows us to loosely couple multiple computational resources, by integrating the concept of service-oriented architecture. Furthermore, the transformation has been further accelerated by the recent scientists' expectation to the collaboration with different multiple organizations on the Internet. From these backgrounds, a new architecture of Grid, that is, service-oriented Grid architecture [13] has emerged. As a result, the latest Grid middleware, that is, Globus Toolkit 4 (GT4) [14-16], has combined service-oriented architecture and the traditional high-performance computing Grid to two new concepts, Open Grid Services Architecture (OGSA) and Web Services Resource Framework (WSRF) [17].

The technological development described above is changing how researches are performed in various scientific fields such as social science [18], high-energy physics [19, 20], earth sciences [21, 22] and bioinformatics [23]. Today, scientists and researchers can obtain enormous amount of data and information through the Internet due to the development of network technology, even if data source are located in different organizations. Furthermore, scientists and researchers can take advantage of the enormous computational power by aggregating multiple computational resources. Moreover, scientists can control scientific measurement devices such as microscopes [24] and medical devices [25, 26] on the Internet. These facts mean that the geographical distances of research organizations is being reduced and a new research environment where scientists at multiple research organizations can perform their research in a collaborative manner for their common research purpose is increasingly demanded.

The recently emerged service-oriented Grid is considered a building block technology for establishing such a collaboration environment on the Internet since it allows computational and data resources to be loosely coupled in an on-demand way. From this consideration, scientists and researchers in various research fields are attempting to re-develop their own scientific programs to Grid services these days. This service-oriented Grid has gathered a lot of attention especially in the research fields of multi-scale and multi-physics simulation federating multiple simulations in terms of collaborative research on such environment. In practice, however, many difficulties still exist despite the recent maturity of distributed computing technologies when scientists develop a Grid application composed of multiple Grid services. For example, the development of a Grid service from an exist-

ing program requires a change in mind set and in-depth knowledge on the service-oriented Grid. Also, the development of Grid applications utilizing such Grid services distributed over multiple organizations is hard for computational scientists without detailed knowledge and techniques of information technologies.

The author believes these difficulties in the development of service-oriented Grid application as major reason hampering the advancement and promotion of computational science. In fact, scientists' expectations of an easy method for facilitating the development of service-oriented Grid application are increasingly growing with the maturity of Grid technologies. More specifically, a method for exposing existing programs as software components or Grid services and utilizing these Grid services over multiple organizations is now demanded.

1.2 Research Objective

From the research background described in section 1.1, this research proposes easy methods for developing a service-oriented Grid application. At the same time, new user-supporting tools that facilitate the development of Grid applications based on the proposed methods are designed and implemented.

In particular, this dissertation focuses on the inefficiency of the following two main stages composed of service-oriented Grid application development:

1. Developing stage of a Grid service from an existing scientific program

2. Developing stage of a service-oriented Grid application from multiple Grid services

For the first stage, this study focuses on the wrapping service which allows the application developer to build a Grid service encapsulating an existing scientific program without writing additional codes. This study proposes a technique to implement the functions of the wrapping service as a software module or operation provider, and allows the application developer to import the functions into his/her Grid service.

For the second stage, this study proposes a new meta-scheduling architecture which allows the application developer to develop a meta-scheduler to handle multiple Grid services. The problem here is how to handle the Grid service which may has different interface each other. To tackle this problem, the proposed scheduling architecture focuses on providing an interface transparent way for selecting a Grid service by taking advantage of the factory pattern technique, which is used for state management of the Grid service.

The ultimate goal of the study summarized in this dissertation is the promotion of service-oriented Grid technologies by realizing new and easy methods and the corresponding supporting tools for developing service-oriented Grid application. Moreover, this research hopefully contributes to the advancement of computational science through the promotion of service-oriented Grid.

1.3 Outline of the Dissertation

This dissertation is organized as follows. Chapter 2 reviews the general procedure of Grid application development and the foregoing related researches. Through the review, the technical issues to solve in this dissertation are clarified. After that, as a solution to technical issues clarified in chapter 2, chapter 3 proposes a new wrapping service model and method for encapsulating an existing program to a Grid service. Also, a tool for facilitating the development of Grid service based on the proposed method is shown. Subsequently, chapter 4 proposes a new meta-scheduling model for building service-oriented Grid application and the corresponding tool for supporting this method. Finally, chapter 5 concludes this dissertation with summary of achievements and directions for future research.

Chapter 2

Technical Issues in Developing Service-Oriented Grid Application

2.1 Introduction

This chapter focuses on the technical issues faced by the application developer in developing a service-oriented Grid application from existing programs. For this purpose, the general procedure of Grid application development is first reviewed. Then, difficulties at the development are particularly focused. Second, a couple of approaches possibly available to develop a Grid application are investigated. Finally, the technical issues to achieve in this study are clarified. Before going into the detail of discussion, this introduction explains the service-oriented Grid architecture and Web Services Resource Framework (WSRF) as a basic knowledge for discussion later in this chapter.

Figure 2.1 shows the general architecture of service-oriented Grid [27, 28]. Today, the term of "Grid" is used to refer to various types of wide-area distributed computing . Campus Grid and Desktop Grid are some examples. This dissertation uses the term of "Grid" to refer to the service-oriented Grid which conforms to the architecture as shown in Fig. 2.1. The service-oriented Grid has a three-tier architecture composed of resource, service, and presentation tiers. The resource tier includes applications, computing resources, data storage resources, and instruments. The access from a user to these resources is taken in the presentation tier, and then virtualized through Web/Grid services in the service tier. Portal frameworks such as GridSphere [29] and Jetspeed [30] provide the user with intuitive Web interfaces of the Grid applications. To process the response to the user request, the portal accesses the appropriate resources via Web/Grid services.

Figure 2.2 shows the overview of WSRF. The outstanding feature of WSRF is the *stateful Web service*. In WSRF, the stateful Web service is prescribed to hold its states as *Resource Properties* (a set of variables) so that it allows the client program to reuse the

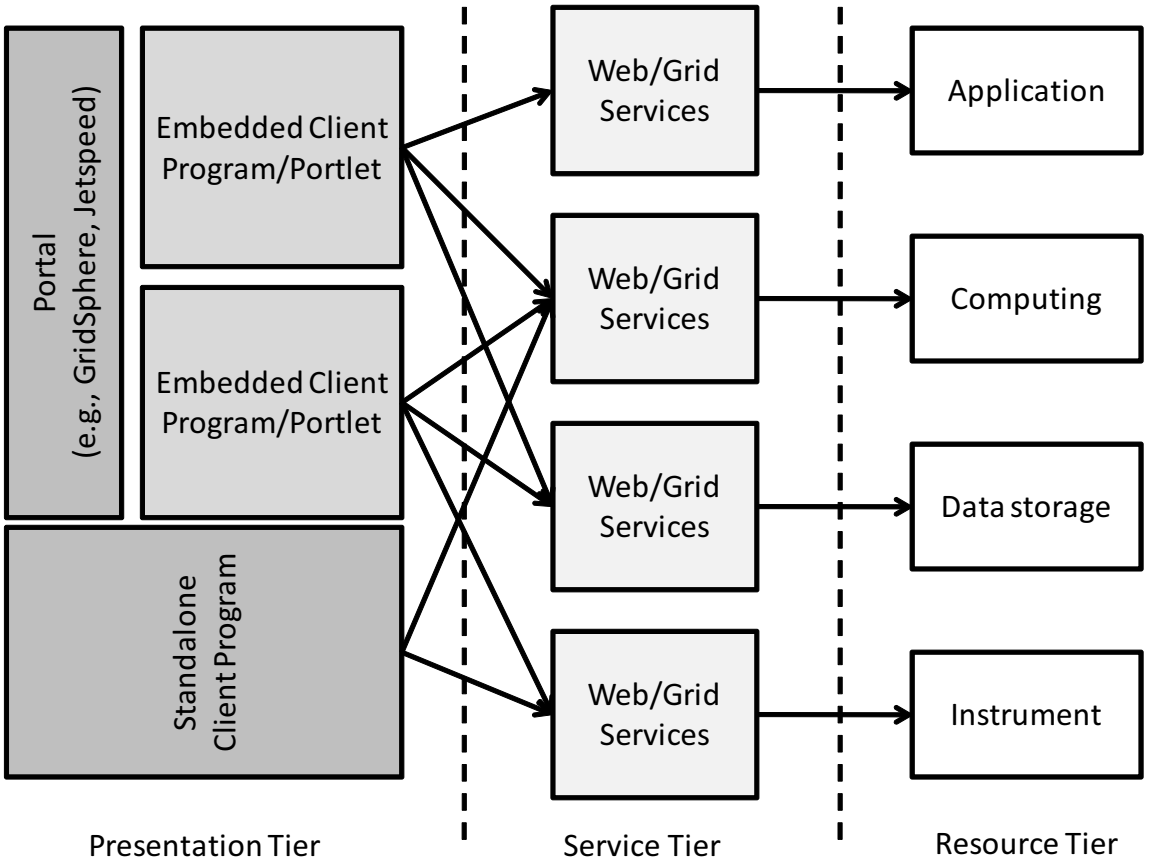

Figure 2.1: General architecture of service-oriented Grid

previous computing results, check the progress of computation and so on. The Resource properties can be defined in Web Services Definition Language (WSDL) as well as the interface of the service. Therefore, they can be accessed through the use of SOAP messages in the same way as the access to the interface of the service.

As described in Chapter 1, the development of Grid application which conforms to this newly emerged service-oriented Grid architecture and WSRF is increasingly demanded. However, there are still many difficulties which developers encounter in developing a service-oriented Grid application conforming to WSRF. In the following sections, such difficulties are investigated through the careful review of the general procedure of Grid application development. After that, the technical issues to achieve in this study are derived.

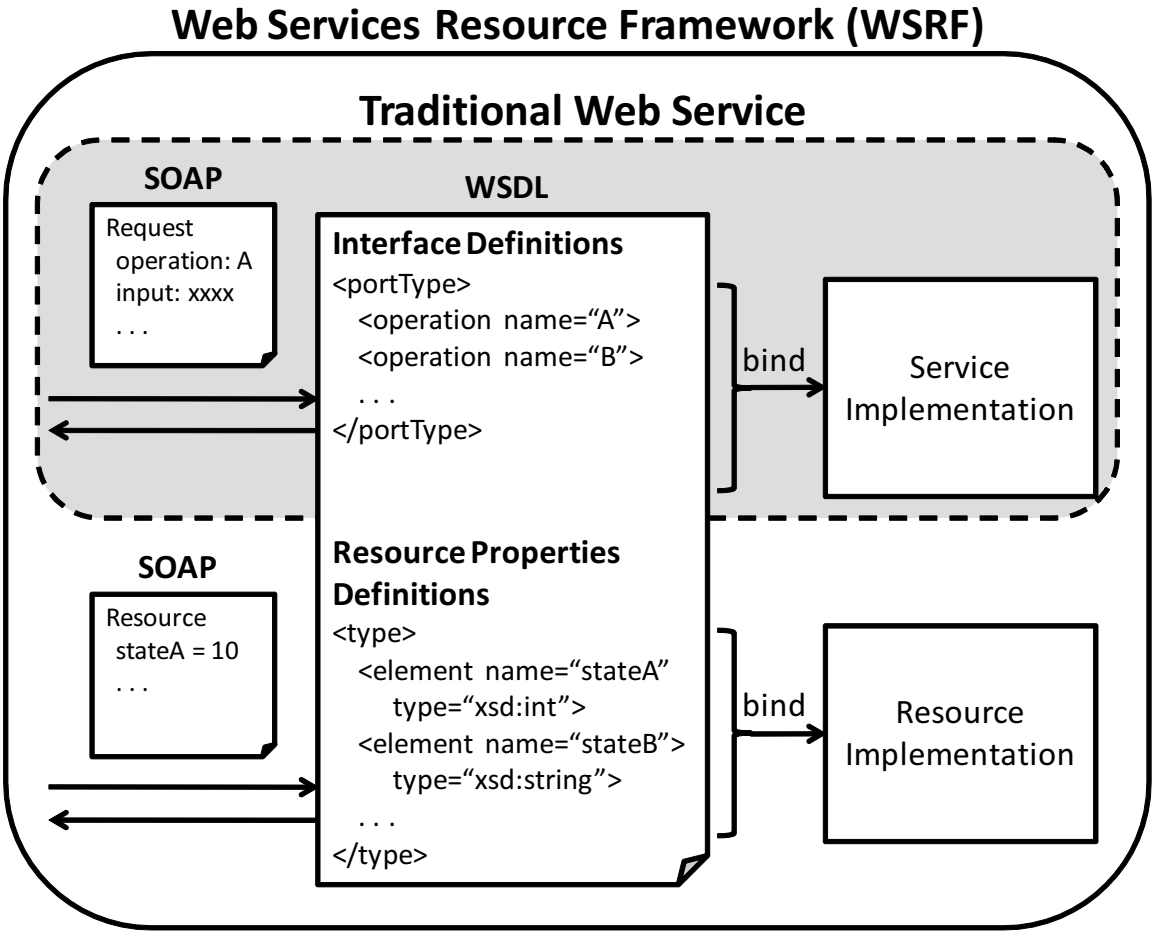

Figure 2.2: Concept of WSRF

2.2 Grid Application Development Procedure and Difficulties

As mentioned in sections 1.1 and 2.1, the application developer encounters many difficulties when he/she develops a service-oriented Grid application from existing scientific programs. This section investigates such difficulties and analyzes the causes of them.

Figure 2.3 illustrates the procedure of Grid application development. In general, the procedure is composed of four stages. The four stages are (1) writing stage of target program, (2) developing stage of WSRF-based service, (3) deploying stage of WSRF-based service, and (4) developing stage of a Grid application composed of multiple WSRF-based services. Each stage of development is reviewed in the following subsections to analyze what kind of difficulties exists and what causes the difficulties.

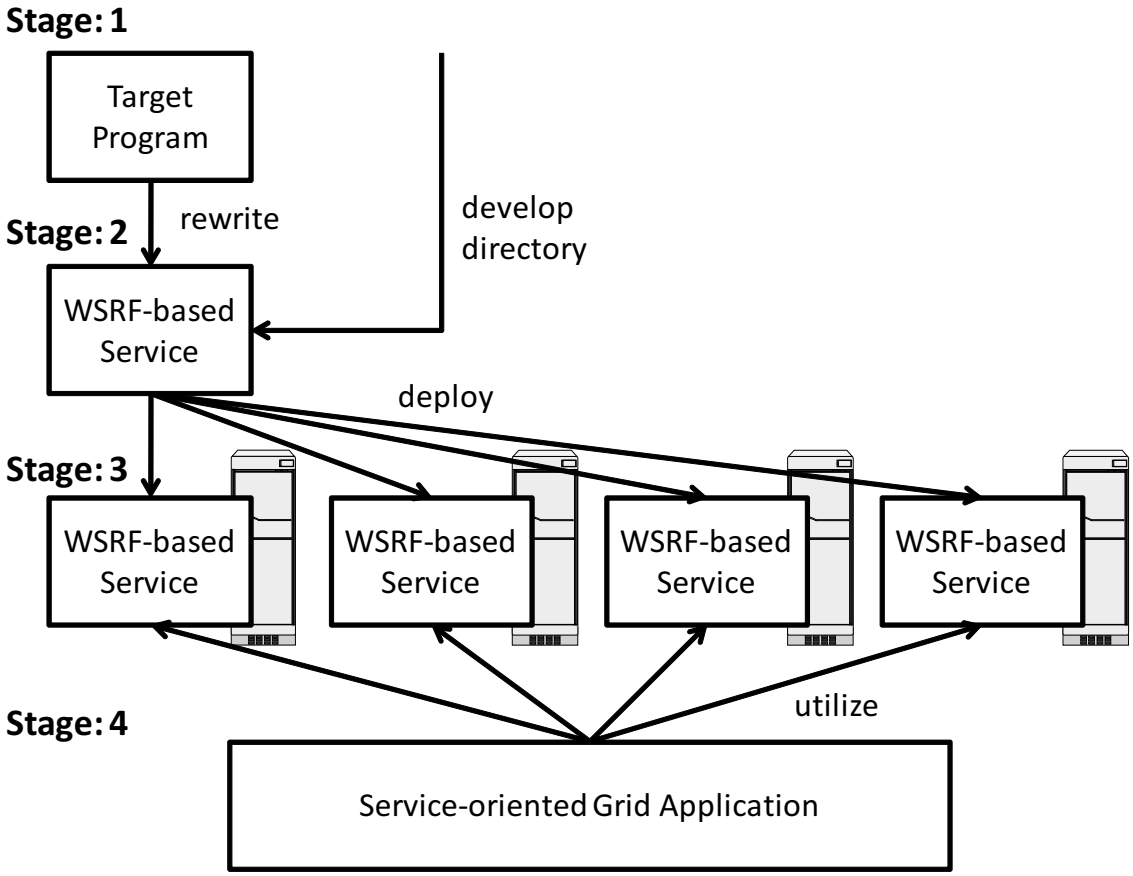

Figure 2.3: Procedure of Grid application development

Stage 1 : **Writing Stage of Target Program**

The first stage is the development of a target program which a scientist as an application developer wants to build as a WSRF-based service. At this stage, the application developer may reuse an existing program or write a program from scratch. In the former case, the application developer can skip this stage. On the other hand, in the latter case, the scientist writes a target program by modeling his/her scientific problems based on his/her professional experience. The works occurred at this stage are mostly derived from the scientist's research fields. In the area of life sciences, for example, the scientist has to consider how chemical compounds as drug candidates are modeled and expressed on a computer, how accurate the model is and so on for writing a target program. This dissertation does not cover these kinds of science-oriented works observed at this stage. Rather, this dissertation assumes that the scientist has an existing program that he/she wants to benefit from Grid technologies.

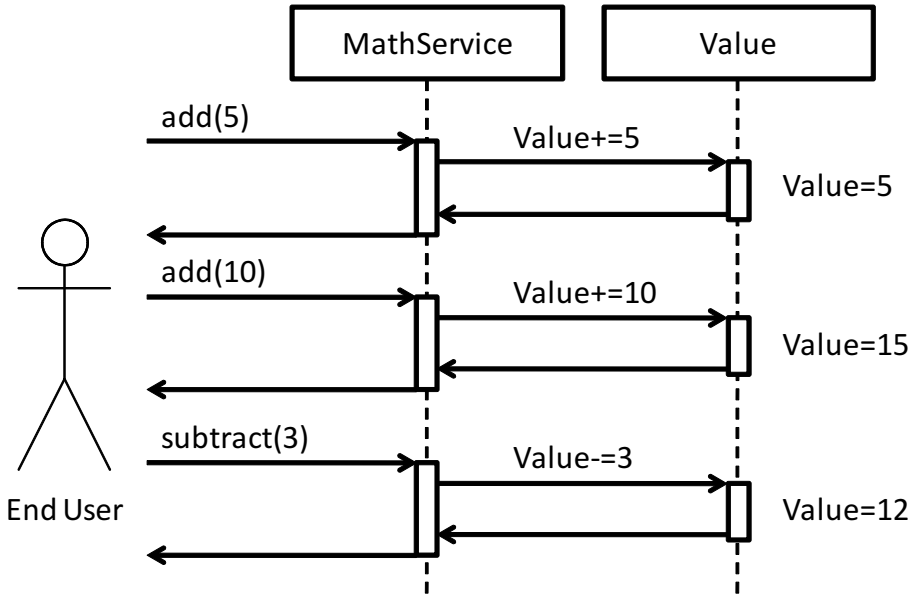

Figure 2.4: Behavior of MathService

Stage 2: Developing Stage of WSRF-based Service

The second stage is the development of a WSRF-based service from a target program. At this stage, the application developer rewrites the target program to a WSRF-based service. More technically, the operations and the corresponding interfaces of the WSRF-based service are designed and implemented based on the consideration on how the target program should be accessed by end users. At this stage, the implementation and configuration works become the hardest ones to the application developer.

At this stage, alternatively, the application developer can develop a WSRF-based service from scratch without going through the first stage. In this case, however, the application developer encounters more programming difficulties than the development case using the target program, since the detailed knowledge and techniques on the Grid technologies as well as in his/her own scientific area are simultaneously required for the development of a WSRF-based service from scratch.

For further understanding of difficulties related to implementation and configuration of the WSRF-based service, how hard the implementation and configuration of the WSRF-based service is analyzed through the following step-by-step review of the development example of *MathService*.

MathService is a simple service from GT4 tutorial documentations [31] and just performs simple two types of calculations (*add()* and *subtract()*) by repeatedly receiving the inputs from the user (Fig. 2.4). Table 2.1 summarizes the amount of configuration and

Table 2.1: Configuration files and implementation for a WSRF-based service example

Configuration files		Implementation files	
File name	Lines	File name	Lines
Math.wsdl	119	MathFactoryService.java	51
Factory.wsdl	71	MathQNames.java	16
deploy-server.wsdd	27	MathResource.java	73
deploy-jndi-config.xml	38	MathResourceHome.java	20
		MathService.java	50

```
/* Remotely-accessible operations */
  public AddResponse add(int a) throws RemoteException {
    value += a;
    lastOp = "ADDITION";
    return new AddResponse();
  }

  public SubtractResponse subtract(int a) throws RemoteException {
    value -= a;
    lastOp = "SUBTRACTION";
    return new SubtractResponse();
  }
```

Figure 2.5: Implementation of MathService operations

implementation required for developing this MathService. As the table indicates, the application developer has to prepare total 255 lines for configuration files and total 210 lines for implementation files even for MathService whose substantial operations are realized by only approximately 10-line codes as shown in Fig. 2.5. This large amount of configuration and implementation required for building a WSRF-based service becomes a big obstacle for the development at this stage.

The developing stage of a WSRF-based service is, in general, further divided to the following seven steps. Figure 2.6 shows the overview of WSRF-based service development composed of the seven steps. At the first step, the application developer designs the interfaces of the WSRF-based service. In this step, the application developer decides the specification of operations (functions provided by the service) and resource properties (a set of variables to hold the state of the service) which the WSRF-based service provides to the end user. At this step, the application developer designs add() and subtract() operations and its interfaces as well as variables as resource properties.

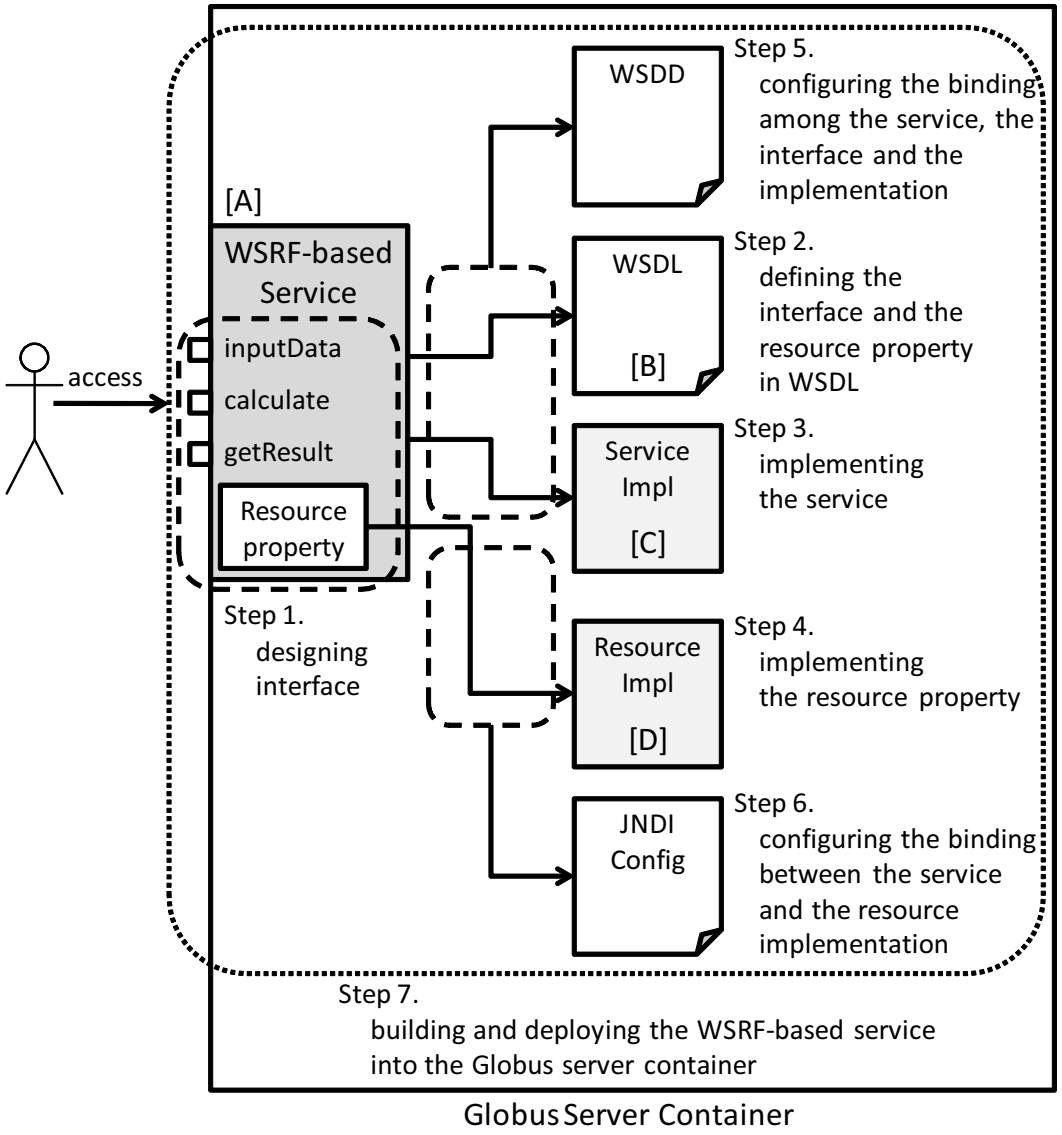

Figure 2.6: Overview of WSRF-based service development

At the second step, the application developer defines the interfaces and resource properties in the WSRF-based service in WSDL to allow the client program to know what kind of interfaces and resource properties are available for the access to the WSRF-based service. Figure 2.7 shows a part of the actual definition of operation “void add(int a)”. As the figure indicates, the application developer has to prepare at least more than 15-line configuration in WSDL just for defining add() operation and making it accessible from the end users. As easily imagined, in the case of the WSRF-based services which are used for practical scientific computation, the work at this step becomes time-consuming and error-prone work due to the increase in the amount of implementation and configuration which the application

```
<xsd:element name="add" type="xsd:int"/>
<xsd:element name="addResponse">
  <xsd:complexType/>
</xsd:element>
...
<message name="AddInputMessage">
  <part name="parameters" element="tns:add"/>
</message>
<message name="AddOutputMessage">
  <part name="parameters" element="tns:addResponse"/>
</message>
...
<operation name="add">
  <input message="tns:AddInputMessage"/>
  <output message="tns:AddOutputMessage"/>
</operation>
```

Figure 2.7: Part of interface definition of add() operation

developer has to prepare.

At the third and fourth steps, the application developer implements the operations and resource properties of the WSRF-based service. At these two steps, the operations and resource properties are actually implemented so that the WSRF-based service provides the actual functionalities of add() and subtract() operations using resource properties (a set of variables). In the example of MathService, add() and subtract() operations can be easily implemented because of the small number of operations. However, in the case of the development of the WSRF-based service from the existing practical scientific application, the amount of program codes which the application developer has to rewrite becomes large because the WSRF-based service for such practical scientific application usually requires many operations to be available via service interfaces. Moreover, the knowledge on and skills of WSRF-based service development are heavily required at these stages. Thus, the reduction of program codes which the application developer has to rewrite is an important technical issue to achieve in these steps.

At fifth and sixth steps, the application developer writes the configuration files to bind the WSRF-based service ([A] in Fig. 2.6) with the interface definition ([B] in Fig. 2.6), the implementation of the service ([C] in Fig. 2.6) and the resource properties ([D] in Fig. 2.6). For this purpose, the application developer himself has to prepare for *deploy-server.wsdd*, and *deploy-jndi-config.xml* configuration files. Figure 2.8 shows the actual examples of

```
<service name="examples/core/rp/MathService" provider="Handler"
      use="literal" style="document">
  <parameter name="className"
      value="org.globus.examples.services.core.rp.impl.MathService"/>
  <wsdlFile>
    share/schema/examples/MathService_instance_rp/Math_service.wsdl
  </wsdlFile>
  . . .
</service>
```

deploy-server.wsdd

```
<service name="examples/core/rp/MathService">
  <resource name="home" type="org.globus.wsrf.impl.ServiceResourceHome">
    . . .
  </resource>
</service>
```

deploy-jndi-config.wsdd

Figure 2.8: Examples of configuration files for a WSRF-based service

these configuration files for MathService: deploy-server.wsdd, and deploy-jndi-config.xml. As easily imagined, writing this configuration is cumbersome, time-consuming and error-prone.

At the last step, the application developer builds and deploys the WSRF-based service into the Globus container which controls the behavior of the WSRF-based service, just by executing ant-based building tools provided by GT4.

As mentioned above in detail, much larger amount of configuration files and implementation in comparison with the operations provided by the WSRF-based service is a big obstacle of the second stage of the development of Grid application. In reality, the preparation for such large amount of configuration files and implementation is time-consuming and error-prone. Furthermore, the fact that the configuration for the WSRF-based service cannot be written in a single file makes the configuration work more inefficient. What is worse, every time the application developer adds an operation to the WSRF-based service, all seven steps must be repeated step by step.

The reason that these complicated configuration and implementation are required is explained from the fact that the WSRF standard provides flexibility when the application developer develops a WSRF-based service. By separating the interface definition and the implementation of the service and the resource properties, the developer can prepare sev-

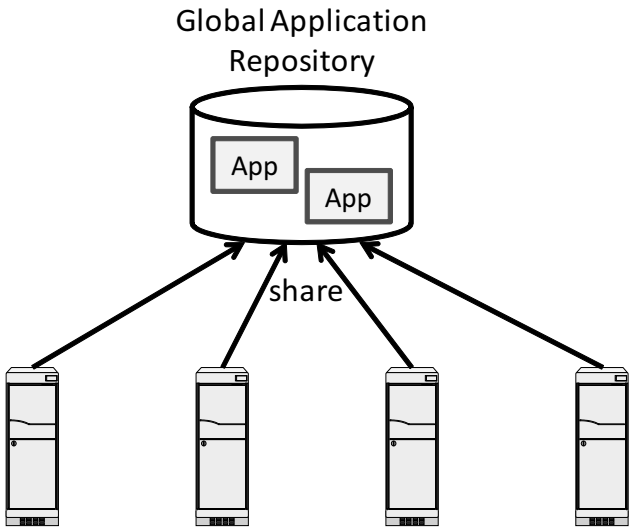

Figure 2.9: An example of global application repository

eral implementations for the service and the resource properties under a same interface definition. This flexibility is particularly useful for porting a WSRF-based service from a legacy implementation to a new implementation, because the developer can handle the legacy implementation and the new implementation under a uniform interface. However, from the aspect of developing a WSRF-based service which reutilizes the existing program prepared at the first stage, this implementation flexibility is not fully necessary. Therefore, a method to reduce the application developer's works of writing configurations and implementations should be proposed based on an assumption that the WSRF-based service reutilizes the existing program.

Stage 3: Deploying Stage of WSRF-based Service

The third stage is the deployment of the WSRF-based service developed at the second stage to multiple resources so that they can be simultaneously used for fault-tolerance, workload distribution and so on. In this stage, the application developer copies and sets up the WSRF-based service to appropriate directories on multiple remote computing resources. The difficulties at this stage mostly come from not only the development itself, but also from cumbersome installation works. In fact, the administrators of remote computing resources have to install and deploy the WSRF-based services so that they can launch and work correctly to the access from the end users. Examples of such installation works include the setup of the programs on every remote site, and the management of program version and license. For the alleviation of the difficulties at the third stage, an application

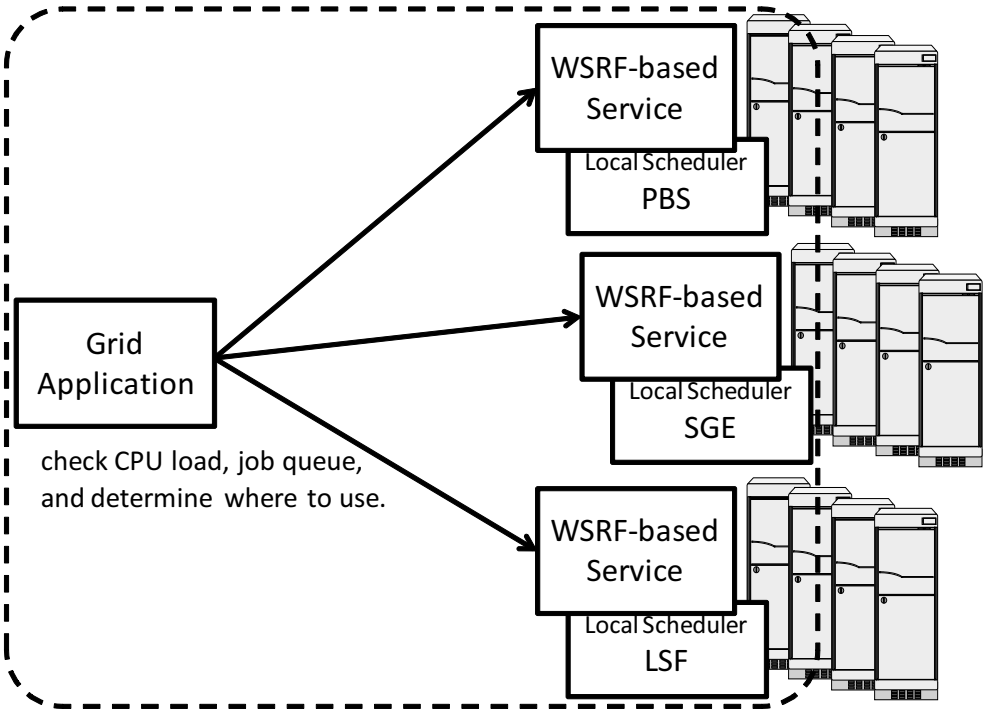

Figure 2.10: An example of a Grid application utilizing multiple WSRF-based services

repository with a global-shared file system such as Gfarm [32] and Global File System [33] can be used as shown in Fig. 2.9, so that multiple remote computing resources can share the directories where the WSRF-based service is deployed. The difficulties at this stage lie in the deployment work rather than the development work. In addition, the deployment work is tightly related to the administration policies in remote sites. Thus, this dissertation does not cover these difficulties.

Stage 4: Developing Stage of Grid Application Composed of Multiple WSRF-based Services

At the final stage, the application developer develops a service-oriented Grid application by making maximum use of the WSRF-based services deployed at the third stage, in hope that the Grid application improves performance, balances loads, and enhances fault-tolerance. To utilize multiple WSRF-based services for the hope, the application developer currently has to develop a Grid application as shown in Fig. 2.10 which checks each job queue of remote sites to determine where to run jobs, and then submits the jobs to remote sites from scratch. The development of such Grid application from scratch is the hardest at the last stage. The reason can be explained from the heterogeneity of administration policies among multiple computing resources.

In general, each site has its own administration policy of computational resources. For example, some remote sites may use Portable Batch Scheduler (PBS) as a local scheduler of computational resources and others may use Sun Grid Engine (SGE) and Load Sharing Facility (LSF). In this case, the application developer has to know how to submit, monitor and control such local scheduling systems in advance and then develop a Grid application that is capable of handling this heterogeneity of administration policies. However, the development of such Grid application is very hard. As imagined easily, this kind of work is also time-consuming and error-prone. These situations mean that an easy-to-use method for building a Grid application from WSRF-based services deployed on the Internet is demanded.

As reviewed above, the large amount of configuration and implementation required for the development of a WSRF-based service makes the development of Grid application at the second stage of the Grid application development inefficient, and the complicated work for handling multiple WSRF-based services prevents the application developer from efficiently developing a Grid application composed of multiple WSRF-based services. On the other hand, the difficulties at the first stage of the Grid application development are mostly derived from scientific research fields, and the difficulties at the third stage lies in the deployment work rather than the development work. Therefore, this study focuses on the second stage and the last stage of Grid application development. In the following sections, the author investigates the conventional approaches possibly available for building a Grid application by solving the difficulties at the second and last stages of Grid application development.

2.3 Review of Conventional Approaches

This section reviews conventional approaches which can be possibly available for the development of a Grid application from an existing application. For this purpose, three classes of conventional methods are first discussed for the second stage of developing a Grid application. Second, conventional meta-schedulers are investigated for the last stage. Finally, through the review of the conventional approaches, this section reveals the technical issues on the development of service-oriented Grid applications.

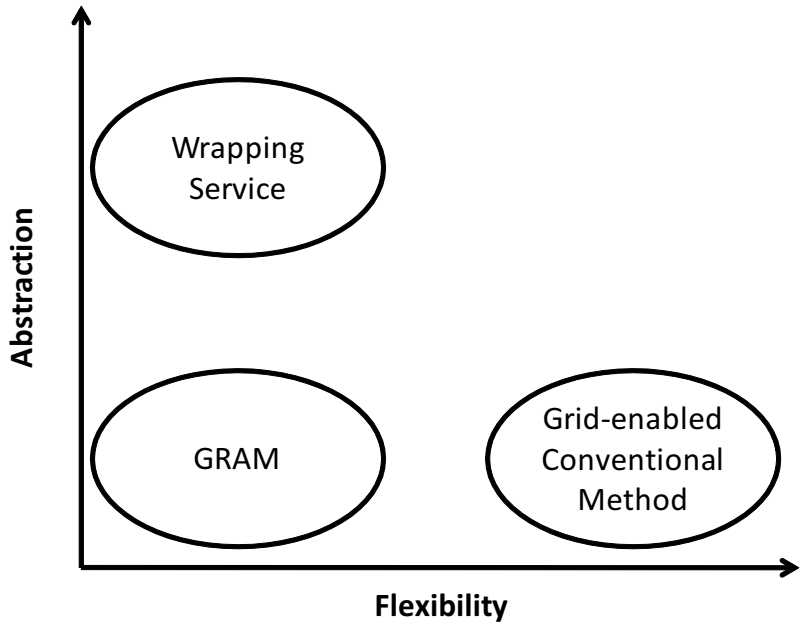

Figure 2.11: Classification of conventional method

2.3.1 Development of a WSRF-based Service from an Existing Program

For developing a WSRF-based service from an existing program, the program has to become accessible on the Grid environment at the second stage of Grid application development. To do this, the following classes of methods can be available.

- Grid Resource Allocation Manager (GRAM)
- Grid-enabled distributed communication method (GridMPI/GridRPC)
- Wrapping service

GRAM and Wrapping service provide a WSRF-based service interface to execute a program on a computing resource. On the other hand, GridMPI/GridRPC extends the interfaces for traditional distributed computing to the ones for Grid computing.

Figure 2.11 roughly summarizes the features of these three methods. The X-axis in the graph represents flexibility, showing how flexible the development style provided by the method is. The Y-axis represents abstraction, showing how much the method hides the heterogeneity of the computational resources and programs. From the standpoint of service-oriented Grid application development, high abstraction and high flexibility are desirable.

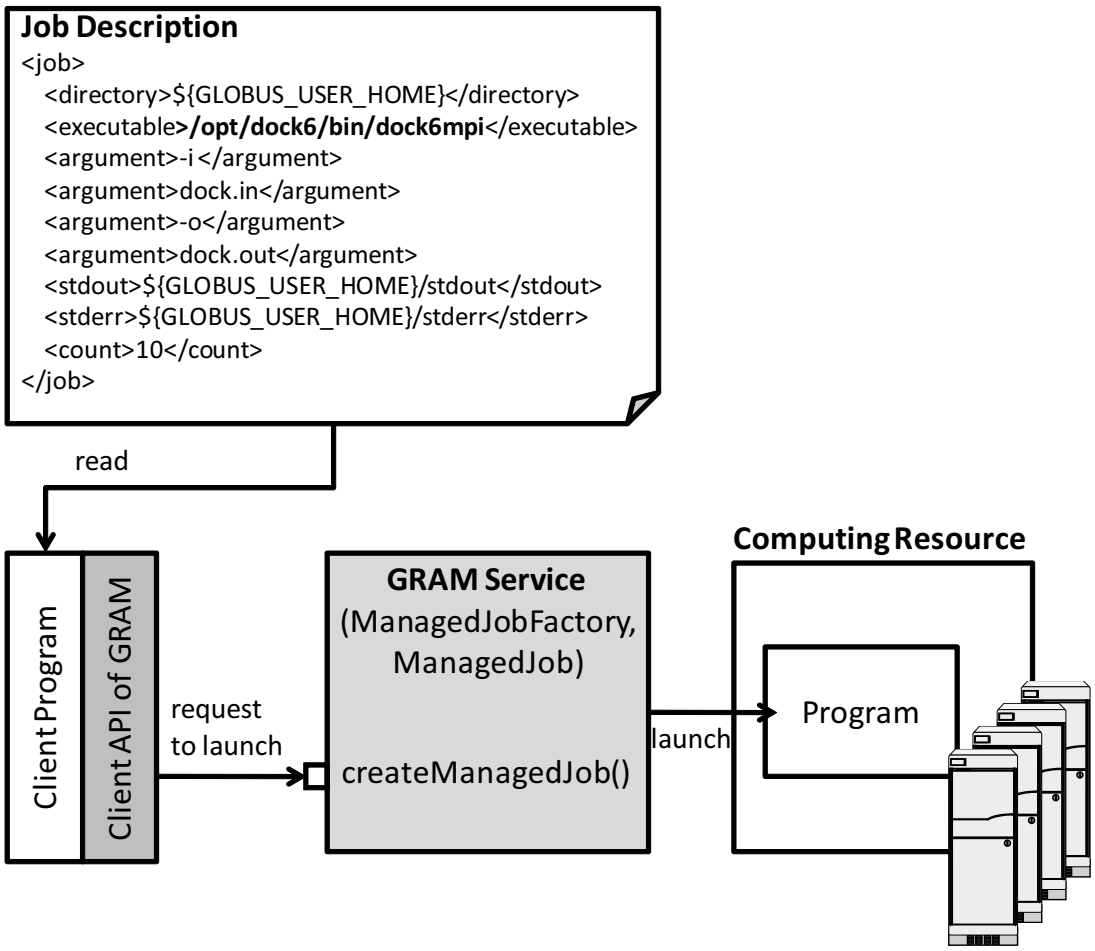

Figure 2.12: GRAM architecture

The following subsections explain each of these methods in detail in order to clarify the technical issues in later discussion.

1. Grid Resource Allocation Manager (GRAM)

The GRAM [34] provides a single WSRF-based interface for requesting and using remote resources for job execution on the Grid environment. The most common usage of GRAM is remote job submission and control. It is designed to provide a uniform and flexible interface to local schedulers. By using GRAM, the user can execute any arbitrary commands as far as the local site policy permits.

Figure 2.12 shows the GRAM architecture. The access to a remote program through GRAM is realized as follows: 1) writing a job description file, and 2) requesting to launch through the interface of GRAM, *createManagedJob()*. Assuming that a program, *dock6mpi*, is installed into */opt/dock6/bin* on a remote site. At this time, the user needs to write the job

description file as shown in Fig. 2.12. In the job description file, the user needs to specify the location where the program is deployed, command-line arguments and so on. To start the program, the client program reads the job description file, and requests to launch a program by accessing the createManagedJob() with this job description file. As shown above, the user can start a program remotely via GRAM.

The advantage of utilizing the GRAM is that the application developer does not need to write complicated WSRF-based service interface to execute his/her existing program, since it already provides WSRF-based interfaces for the execution of any programs. However, the GRAM has the following three problems. The first problem is that GRAM does not hide the heterogeneity of the deployment location of a program. To execute a program through GRAM, a user needs to specify the path where the program is deployed. Generally, the path where a program is deployed differs among remote sites. A user, therefore, needs to write an appropriate job description file for each site. The second is that GRAM has little flexibility in developing Grid applications. Because the GRAM just provides an interface for starting a program on remote sites, the programs started via GRAM do not have any interfaces to interact each other. The application developer therefore cannot develop sophisticated Grid applications that allow loosely-coupled and federating programs executed on different sites only with the GRAM. The third problem is that GRAM allows the user to execute arbitrary commands. Considering from the aspect of exposing a program on the Grid environment, the user should be allowed to start only the exposed program for security reasons. This functionality may cause a serious security problem.

For the reasons above, in case that each site develops a WSRF-based service for a target program with this GRAM, it manages to provide the information on where the program is deployed within the site. Furthermore, if a remote site allows the user to use GRAM, the site administrator has to design site policy carefully to prevent the user from producing an unexpected result.

2. Grid-enabled Distributed Communication Method

The second class of methods possibly available for building a WSRF-based service from an existing program is Grid-enabled traditional distributed communication technologies such as GridMPI [35, 36] and GridRPC [37, 38]. As introduced in Chapter 1, MPI and RPC have emerged in the history of distributed computing. As these technologies facilitate the coding of communication among processes, the application developer can write high-performance distributed programs on the distributed environment composed of multiple computers. Today, these technologies are widely accepted in the development of distributed programs

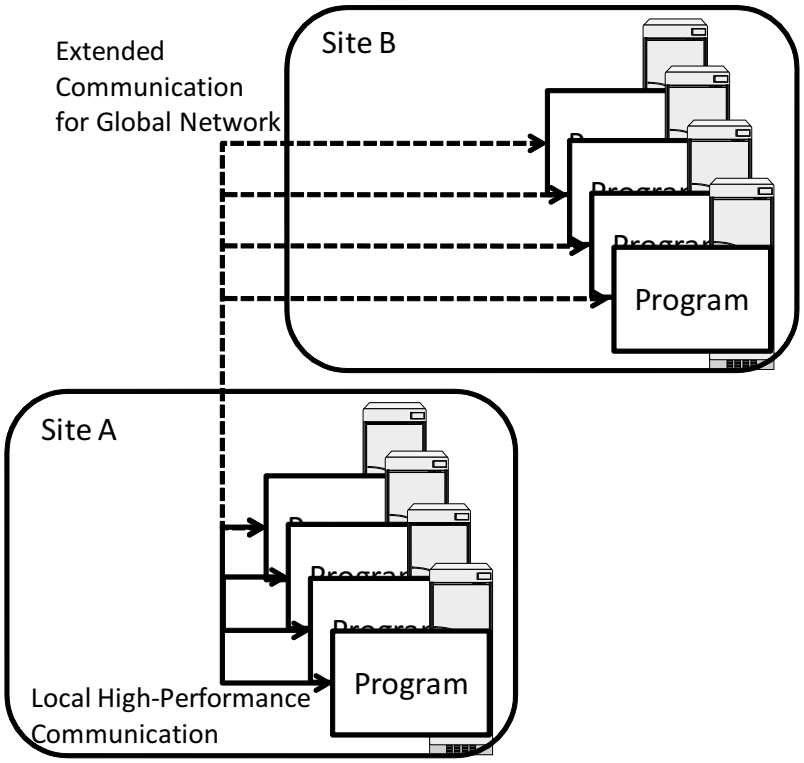

Figure 2.13: Concept of Grid-enabled distributed communication

within a single organization. Since Grid computing technologies emerged, these conventional technologies naturally have been extended to meet Grid computing technologies needs.

The advantage of these two technologies is that the new Grid-enabled communication interfaces are the same as traditional communication interfaces provided by original MPI and RPC. These two technologies hide the details of global communication among the Grid environment behind the traditional communication interfaces. For this reason, the application developer can use traditional tightly-coupled flexible communication interfaces on the Grid environment. In this way, these technologies make it easy for the application developer to rewrite his/her programs for the Grid environment. Figure 2.13 shows the concept of these Grid-enabled distributed communication methods.

However, these two technologies lack the abstraction capability to implement a program as a software component (a WSRF-based service). These technologies focus on the extension of traditional tightly-coupled communication technologies into the Grid environment, while the service-oriented Grid focuses more on the loosely-coupling of programs, organizations and researchers. Therefore, the use of this class of methods is not inherently appropriate for the development of WSRF-based service.

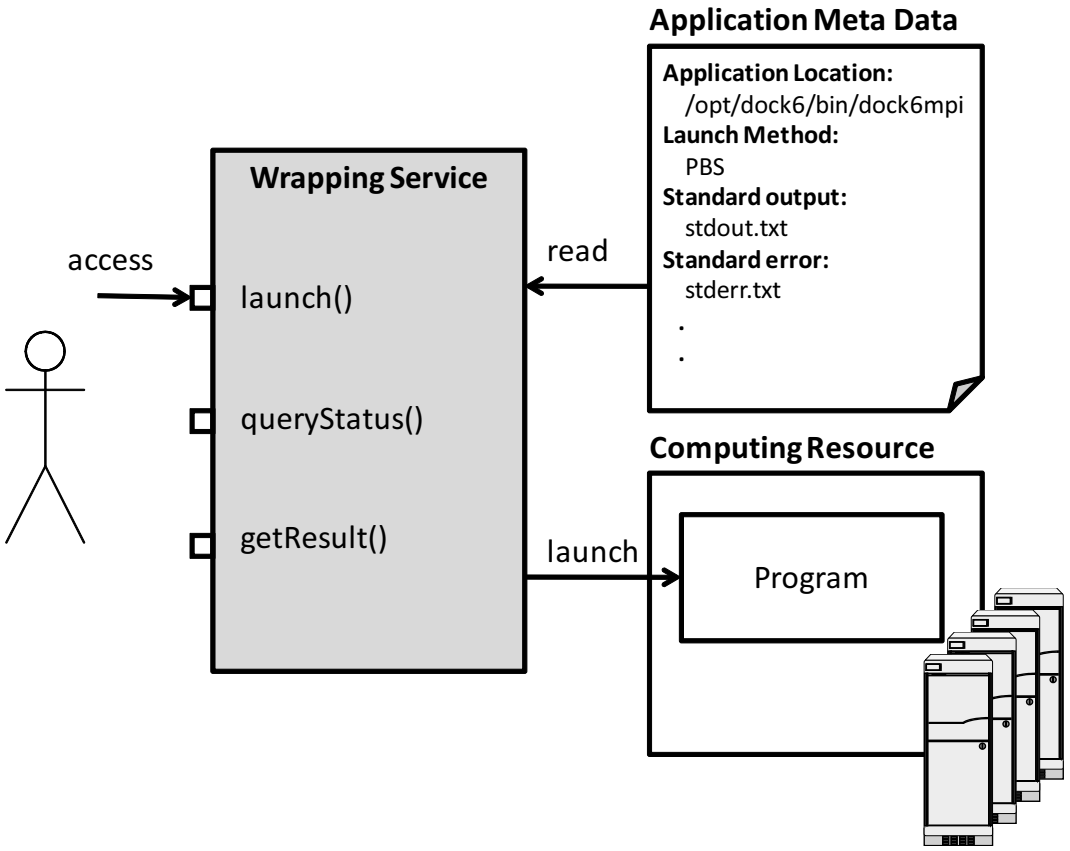

Figure 2.14: Basic design of wrapping service

3. Wrapping Service

Another type of method possibly available for building a WSRF-based service from an existing program is wrapping service. In this solution, a wrapper encapsulates a program as a Web/Grid service. The service encapsulating the program executes the program on a certain computing resource and then provides a Web service interface with which allows the user to request program execution, obtain the result, and so on.

This method could also be explained as a method that removes the disadvantages of using GRAM directly. As mentioned previously, each remote site has to provide the information where a program is exactly deployed to enable the invocation of the program in the case of GRAM. Furthermore, whereas the GRAM allows the user to execute any commands that are not related to the target program, the wrapping service provides an interface to execute only the target program. Thus, the wrapping service is better than GRAM from the aspect of exposing a program as a Web/Grid service.

There are several wrapping tool implementations that construct wrapping services. Al-

though the implementations are different among the wrapping tools, the basic design of these tools is very similar. Figure 2.14 shows the basic design behind these wrapping services. In the basic design, the wrapping service provides the interfaces for launching applications, querying about application status, and acquiring the result to the user. If a user requests starting the program, the wrapping service reads application metadata which describes program location, the way to start the program, and so on. After that, the wrapping service launches the program appropriately on a computing resource.

The wrapping service allows the application developer to expose his/her program as Web/Grid service with minimal effort. In fact, for building such services, all the application developer has to do is write a single or a couple of a few line-configuration files with most wrapping tools. Therefore, with the wrapping tools, the application developer is relieved from the large amount of configuration and implementation works observed at the second stage. In practice, however, there is little flexibility and extensibility in developing practical Web/Grid services with this wrapping service. Sometimes, a scientist as an application developer needs to extend the service constructed by these wrapping services for the purpose of federating multiple Web/Grid services. In this case, the application developer has greater difficulty in extending the service by reutilizing it, because the wrapping service is specially designed and implemented so that not the application developer but the user of the service constructed by the wrapping service can easily use the service.

2.3.2 Development of a Grid Application Utilizing Multiple WSRF-based Services

To develop a service-oriented Grid application from multiple WSRF-based services, an easy way to allow the application developer to utilize WSRF-based services on multiple resources is necessary at the last stage of Grid application development. In order to relieve the application developer from the difficulties in utilizing WSRF-based services on multiple resources, a single logical resource aggregating computing power from multiple resources is needed. A meta-scheduler could be a solution for that. The meta-scheduler relieves the user from the complicated management of multiple resources, and allows the user to submit jobs to cluster systems composing the Grid environment. The term “meta-scheduler” is used as compared to the term “local scheduling system” such as PBS and SGE set up in a cluster system. A meta-scheduler is an upper-layer scheduler that schedules jobs to multiple local scheduling systems. When a user submits a job through a meta-scheduler, the meta-scheduler determines where the job should run by checking the state of multiple resources.

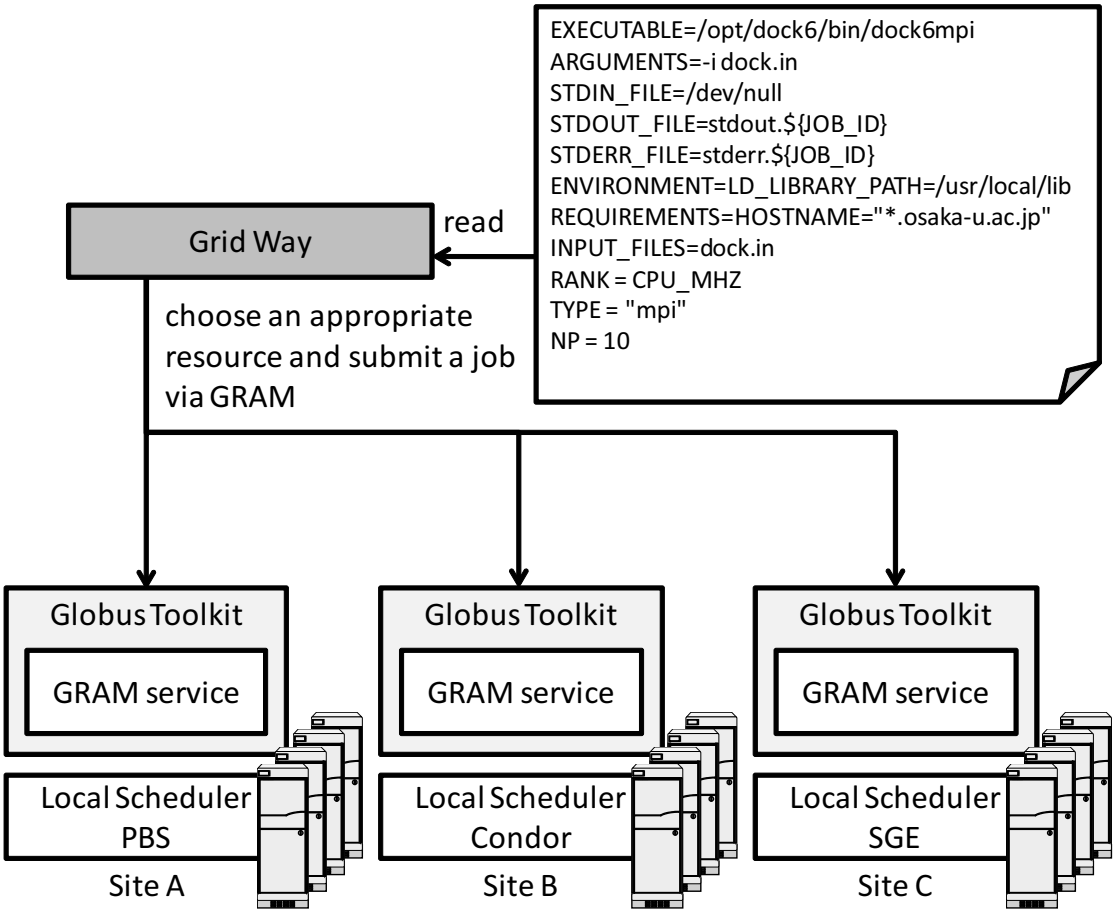

Figure 2.15: Overview of GridWay architecture

There are several implementations of the meta-scheduler. Major examples are Condor-G [39], CSF4 [40] and GridWay [41]. All of these meta-schedulers utilize the resource management service of Globus Toolkit, GRAM, to submit jobs into underlying computing resources, since GRAM hides the heterogeneity of local schedulers and provides a uniform interface for submitting jobs. In submitting a job to a meta-scheduler, therefore, a job description file is required as well as a job submission at GRAM. Figure 2.15 shows an example of a job description file (GWJT) of GridWay and its architecture. In the job description, the user needs to specify almost the same parameters as the job description in case of GRAM. In addition, the user can specify some parameters to control global scheduling policy. In this example, the description requests a job to be submitted into sites only matched with **.osaka-u.ac.jp* and also requests the fastest-ranked site based on the clock frequencies of CPUs (CPU_MHZ). When a job description is given, the meta-scheduler selects an appropriate resource among multiple sites which best matches the request and submits a job via the GRAM service of the selected site.

However, these meta-schedulers are basically extension of local schedulers and do not assume WSRF-based services as scheduling targets. These meta-schedulers are designed to provide an upper job submission service over existing job submission services, GRAM. The specification of the GRAM service interface has been fixed with Globus Toolkit. Therefore, preparing an upper job submission service over such fixed service is not difficult essentially. Such meta-schedulers are applicable only for scheduling of GRAM services. The meta-schedulers which have been developed until today are, thus, not applicable for WSRF-based services developed by application developers.

2.4 Technical Issues in Developing Service-oriented Grid Application

2.4.1 At Development of a WSRF-based Service from an Existing Program

From the aspect of exposing an application on the Grid environment, to allow the application developer to use APIs provided by the resource management service of Globus Toolkit or GRAM is not a good solution, because the GRAM-based method does not hide the heterogeneity of the application deployment environment completely. In addition, it may cause critical security problems because it allows the user to execute arbitrary programs. Grid-enabled traditional distributed computing technologies such as GridMPI and GridRPC allow an application developer to easily write very flexible programs because they provide the functionalities of explicitly writing codes of communication among multiple computers. Taking the abstraction of a program as a WSRF-based service into consideration, however, the use of conventional tightly-coupled distributed computing technologies as-is over global network is not efficient.

Compared to the above two classes of technologies, the wrapping service approach is the most possible way to expose a program on a Grid environment. The wrapping service method abstracts the access to the program as a Web/Grid service, while GRAM does not hide the application deployment environment. Also, since the wrapping service approach can reduce the amount of configuration and implementation, the wrapping service approach is superior to other two classes of methods. However, the problem on the inflexibility of the wrapping service approach has to be taken into account. The wrapping service just provides a set of interfaces to execute a program on a computing resource. In most cases, the application developer may be satisfied with just executing his/her program. However, to develop a more sophisticated Web/Grid service which interacts and federates with other

Web/Grid services, the wrapping service should be designed and implemented so that the application developer can extend the wrapping service to meet his/her demands.

By relieving this disadvantage of the inflexibility, the author believes that the wrapping service can become a new method for the development characterized by high abstraction and high flexibility. The establishment of a flexible and extensible wrapping service method is, therefore, an important technical issue to achieve, in order to encourage the development of Grid applications.

2.4.2 At Development of a Grid Application Utilizing Multiple WSRF-based Services

To facilitate the management and utilization of multiple computing resources, the meta-scheduler is useful for the submission of jobs to multiple resources. However, meta-schedulers which have been implemented so far can take only GRAM services as scheduling targets. In other words, such meta-schedulers provide an interface only for executing a program on a remote site through the GRAM.

In order to develop a Grid application composed of multiple WSRF-based services developed by the application developer, a new meta-scheduler suitable for the service-oriented Grid architecture is essential. In other words, the establishment of a new meta-scheduling method which can utilize the multiple WSRF-based services as scheduling targets is demanded for facilitating the development of Grid application composed of multiple WSRF-based services.

2.5 Concluding Remarks

For the purpose of establishing methods for service-oriented Grid application development, this chapter clarified technical issues to be solved in this dissertation. Through the reviews of the Grid application development procedure, this chapter clarified the difficulties in developing service-oriented Grid applications. In particular, the developing stage of a WSRF-based service from an existing program, and the developing stage of a Grid application utilizing WSRF-based service deployed over multiple resources were discussed. In this chapter, the author considered conventional approaches possibly available for solving the difficulties at these two stages, and then clarified the following technical issues.

- Establishment of a new flexible and extensible wrapping service method
- Establishment of a new meta-scheduling method suitable for WSRF-based services

To tackle these technical issues and relieve difficulties at the two developing stages of the Grid application development procedure is important in encouraging the development of service-oriented Grid applications.

Chapter 3

Extensible Grid-Enabling Wrapping Method

3.1 Introduction

Despite the maturity of Grid computing middleware, the methods for exposing an existing application as a service have not been well developed. The wrapping method [42-45] that executes a command-line program on a computing resource and provides interfaces for accessing the result has been studied as a possible way for easily exposing existing applications such as the Web/Grid service. In practice, however, there is little flexibility and extensibility for the application developer to further develop the wrapped application. In other words, the application developer cannot implement application specific interfaces in the service realized by such traditional wrapping tools.

Considering this situation and the issues revealed in Chapter 2, a new wrapping method, the "Extensible Grid-enabling Wrapping method", and its tool, Opal Operation Provider (Opal OP), which allows the application developer to easily build a WSRF-based service from an existing application, are proposed in this chapter. The proposed wrapping method is based on the new Extensible Wrapping Service model.

This chapter is organized as follows. Section 3.2 attempts to find a common model behind the traditional wrapping methods available today, and then clarifies the problems. In section 3.3, the Extensible Wrapping Service Model is proposed to solve the problem mentioned in section 3.2. Section 3.4 describes a tool, Opal Operation Provider (Opal OP), which implements the new model. Section 3.5 introduces three examples including bio-molecular simulation system using Opal OP and discusses the usability of Opal OP. Section 3.6 concludes this chapter.

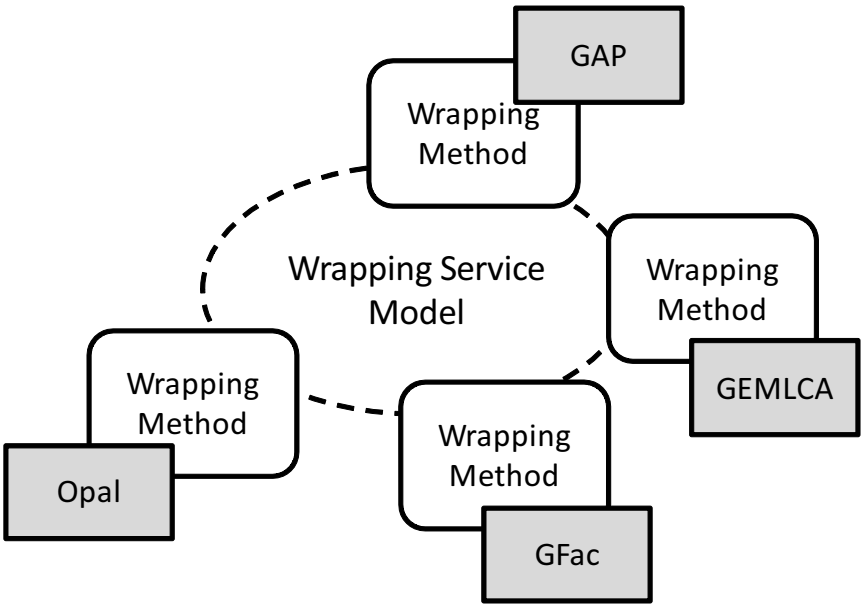

Figure 3.1: Wrapping tools available today

3.2 Conventional Wrapping Methods and Problems

Figure 3.1 shows the author's view to the current situation of wrapping methods. This section first finds a common model behind the conventional wrapping methods by reviewing them. After that, the problem encountered in the common model is described.

3.2.1 Conventional Methods and Tools

Until recently, several wrapping methods, which allow the application developer to easily build a Web/Grid service from a legacy program, have been proposed, and the corresponding tools have been implemented. Figure 3.1 shows some examples of such methods and tools discussed later in this subsection. As the name indicates, the wrapping method generally wraps a program and then exposes the program as a Web/Grid service. However, the wrapping method cannot encapsulate all kind of programs. The target program to be wrapped is supposed to be a simple command-line program which inputs data from files, command-line arguments, and standard input, and outputs data to files, standard output, and standard errors (Fig. 3.2). The current wrapping method cannot support programs with Graphical User Interface (GUI), network server programs (e.g., Web server), OS and so on. However, in most cases, this is not a disadvantage in developing Grid applications, because most scientific programs are implemented as a command-line program.

In general, a command-line program has the following states: 1) starting, 2) reading

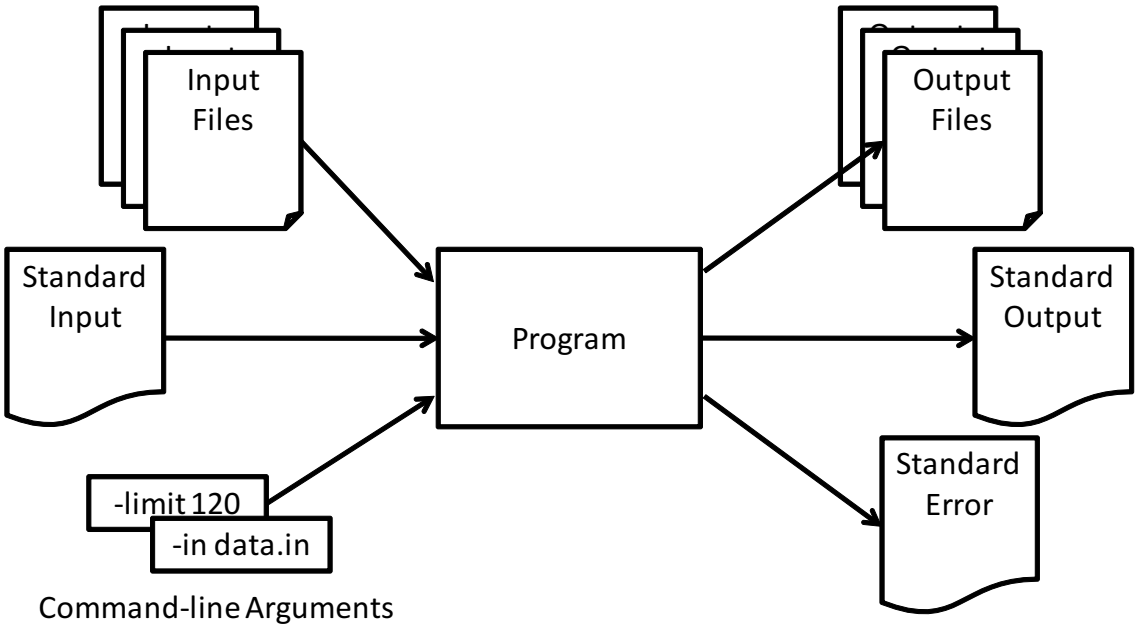

Figure 3.2: Model of command-line program

input data, 3) computation, 4) writing output data, and 5) finishing. The wrapping tool, therefore, has only to control these five states of the command-line program by providing the interface of Web/Grid services. In order to govern these behaviors of the command-line program, the wrapping tool offers a suite of interfaces for transferring input data, starting the program, retrieving output data, and monitoring the program status.

The following are typical examples of wrapping tools facilitating the development of Web/Grid service based on the corresponding wrapping methods.

- Opal
- GEMLCA (Grid Execution Management for Legacy Code Architecture)
- GAP Service (Generic Application Service)
- Gfac (Generic Application Service Factory)

Opal is a tool that allows the application developer to encapsulate a command-line program to a Web service [42]. The second example is GEMLCA [43]. It encapsulates a command-line program to a Grid service, utilizing Globus Toolkit 3. The third example is GAP. This GAP encapsulates a command-line program to a visualization component used in the visualization middleware named In-VIGO framework [44]. Gfac encapsulates a command-line program to a Web service [45].

```
<appConfig xmlns="http://nbcr.sdsc.edu/opal/types"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <metadata>
  <usage>
    <![CDATA[./babel [-i<input-type>] <name> [-o<output-type>] <name>]]>
  </usage>
  <info xsd:type="xsd:string">
  <![CDATA[
  ...
  Currently supported input types
            alc -- Alchemy file
            prep -- Amber PREP file
            ...
  Currently supported output types
  ...
            caccrt -- Cacao Cartesian file
            cacint -- Cacao Internal file
  ...
  Additional options :
  ...
  ]]>
  </info>
 </metadata>
 <binaryLocation>/Users/sriramkrishnan/bin/babel</binaryLocation>
 <defaultArgs></defaultArgs>
 <parallel>false</parallel>
</appConfig>
```

Figure 3.3: An example of a configuration file used by Opal

All of these tools encapsulate a command-line program as a Web/Grid service. Although the details of the implementations of these wrapping tools differ, the concept behind these tools is almost the same. Specifically, these tools commonly control the behavior of the program by governing the five program states through a Web/Grid service.

In addition, in order to allow a scientist as an application developer to wrap his/her program as a Web/Grid service with minimal effort, all of these wrapping tools force him/her to write only a configuration file. In other words, the application developer does not have to write any additional program code in wrapping their programs.

The configuration files for these wrapping tools available today contain the meta-data of the target program. Where the target program is deployed on a system, how to start the program, and what the command-line arguments are, are the example descriptions required for configuration. By referring this meta-data, these wrapping tools can automatically

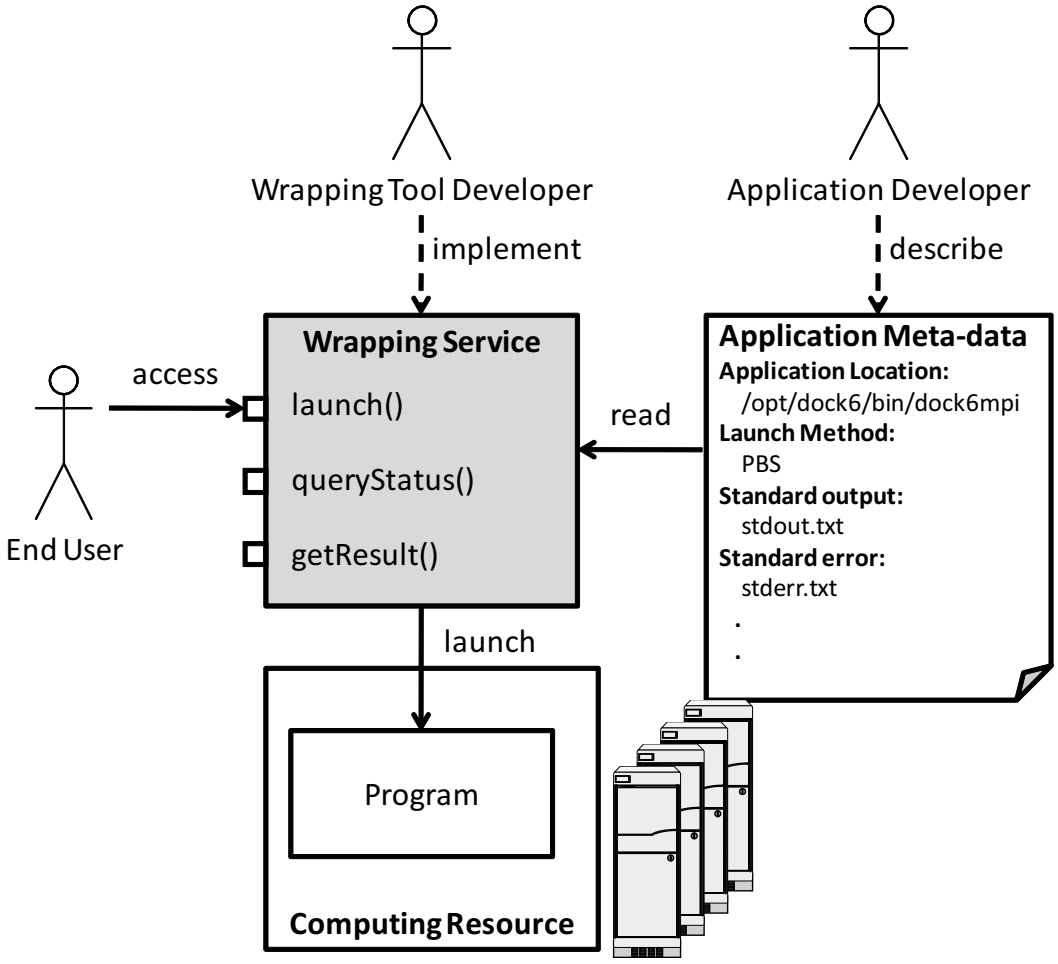

Figure 3.4: Wrapping service model

encapsulate the program. Figure 3.3 shows an example of a configuration file used by Opal. *binaryLocation* specifies the path of target program where the program is deployed. By *defaultArgs*, default command-line arguments are specified. *parallel* specifies whether the target program needs to be started as a parallel computing program (e.g., MPI) or not. This example indicates that the target program is deployed at */Users/sriramkrishnan/bin/babel*, needs no default argument, and does not need to be started as a parallel computing program.

3.2.2 Wrapping Service Model behind Existing Wrapping Tools

In section 3.2.1, several conventional wrapping methods and tools were reviewed. Through this review, the author has come up with a common model which lies behind these conventional wrapping methods available today for encapsulating a command-line program to a Web/Grid service. Figure 3.4 shows the common model and its architecture behind the wrapping methods available today. In this dissertation, the author refers to this model

as the "Wrapping Service Model". There are three actors: the application developer, the wrapping tool developer, and the end user in this model. The application developer is the actor who develops a program encapsulated with the wrapping tool to a Web/Grid service, the wrapping tool developer is the actor who develops the wrapping tool encapsulating application developer's program, and the end user is the actor who uses the Web/Grid service encapsulating the application developer's program. In this model, all the application developer has to do is to describe a configuration file specifying the meta-data of the target program. All the wrapping methods do not force the application developer to write additional program codes to encapsulate a program as a Web/Grid service. *Wrapping service* in Fig. 3.4 is a Web/Grid service encapsulating the target program, which is built using wrapping tools such as Opal and Gfac. In response to a request from the end user, the wrapping service starts a program job on the computing resources based on the meta-data in the configuration file.

The wrapping service method on the model shown in Fig. 3.4 is helpful to the application developer who just wants to convert his/her own program to a Web/Grid service. However, this wrapping method is not useful and effective to the application developer who wants to extend the Web/Grid service wrapping his/her own application because it cannot satisfy all requirements from such a developer. For example, a program, which an application developer wants to run on a Grid environment, may sometimes need to interact with other programs during program execution. In this case, the application developer may need to implement additional specific functions for these purposes on the wrapping services in an extensible manner. However, the conventional wrapping tools such as Opal and GEMLCA do not allow the application developer to easily develop the wrapping service in such an extensible way. Exceptionally, only Gfac allows the application developer to extend the implementation of the wrapping service. However, the mechanism provided by Gfac just allows the application developer to add simple pre-process and post-process routines to each interface provided by the wrapping service. In other words, the application developer is only allowed to extend the functions such as launchJob() provided by Gfac. The mechanism does not allow the application developer to design and add application specific functions freely. As stated above, as far as the conventional wrapping methods are used, many difficulties take place when the application developer attempts to further develop the wrapped application. The reason is that these wrapping tools never assume the application developer's further extension of the wrapping service built through their use, and therefore, only a limited suite of interfaces for using the wrapping service is provided.

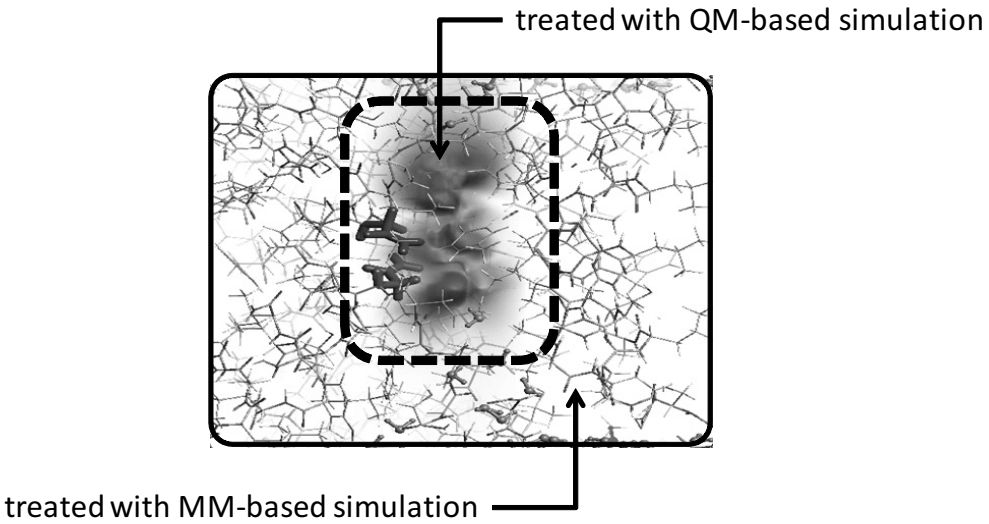

Figure 3.5: QM/MM hybrid simulation

3.2.3 Inextensibility and Inflexibility in Development based on Wrapping Service Model

For further understanding of this inextensibility and inflexibility in the existing wrapping service model, this subsection briefly introduces an actual problem in the development of a Grid service based on the wrapping service model by reviewing a development example, a bio-molecular simulation called QM/MM hybrid simulation composed of multi-scale and multi-physics simulations.

Multi-scale/multi-physics simulation is a simulation that attempts to simulate complex phenomena by integrating multiple simulations across different viewpoints. Until today, many simulation programs solving a simple problem have been developed based on a single viewpoint. However, understanding of complex scientific phenomena (e.g., life scientific phenomena, and global climatic phenomena) cannot be achieved with a single viewpoint. Therefore, a new simulation trend for integrating several simulations to understand scientific phenomena from multiple viewpoints has emerged recently. This trend is believed to lead to a new paradigm shift in computational sciences [46].

As an example of such multi-scale simulations, a bio-molecular simulation called QM/MM hybrid simulation can be considered. The QM/MM hybrid simulation attempts to integrate a molecular orbital simulation and a molecular dynamic simulation. These two simulations calculate and predict the behavior of the molecule, based on different scale physics of Quantum Mechanics (QM) and Molecular dynamic Mechanics (MM), respectively. In order to accurately understand the dynamic behavior and detailed chemical reactions of

bio-molecules, performing these two types of simulations in an integrated and simultaneous manner is essential (Fig. 3.5). To this end, the exchange of intermediate data (e.g., coordinates of molecules, force between molecules) between the two programs during the execution of these simulations is required.

In this actual development case, in order to execute each of these two simulation programs as a single simulation individually, each of the simulations can be easily developed as a Web/Grid service through the use of existing wrapping methods. Importantly, however, to develop the integrated QM/MM hybrid simulation, the further development of these two Web/Grid services built through the use of the wrapping method is required so that two Grid services synchronize and exchange data.

3.3 Extensible Wrapping Service Model

This section proposes a new model to overcome the inflexibility and the inextensibility in further developing the wrapped application. Figure 3.6 shows the new wrapping model. The author calls this model the "Extensible Wrapping Service Model". The concept behind the proposed model is the modularization of the target application as a program module, which allows the application developer to easily integrate the module into his/her WSRF-based service. The difference from the model shown in Fig. 3.4 is that the target program is first encapsulated as a *wrapping module* and subsequently imported into the WSRF-based service developed by the application developer. In this model, the application developer can develop his/her WSRF-based service by him/herself whereas he/she could only write application meta-data in the previous model.

In this model, there are three actors as in the previous model: the application developer, the wrapping tool developer, and the end user. The wrapping tool developer provides a wrapping tool that allows the application developer to encapsulate the target program into a wrapping module. The application developer designs and implements his/her WSRF-based service, and imports the wrapping module into his/her service. If the end user accesses to an interface related to the wrapping modules, the request of the end user is delegated to the wrapping module implementation. On the other hand, if the end user accesses to the application specific functions, the request is dealt within the WSRF-based service.

The advantage of this model is that the implementation of the wrapping module is separately performed from an application specific implementation. The application developer, therefore, can design and implement the WSRF-based service flexibly and extensively without being aware of the implementation restrictions of the conventional wrapping

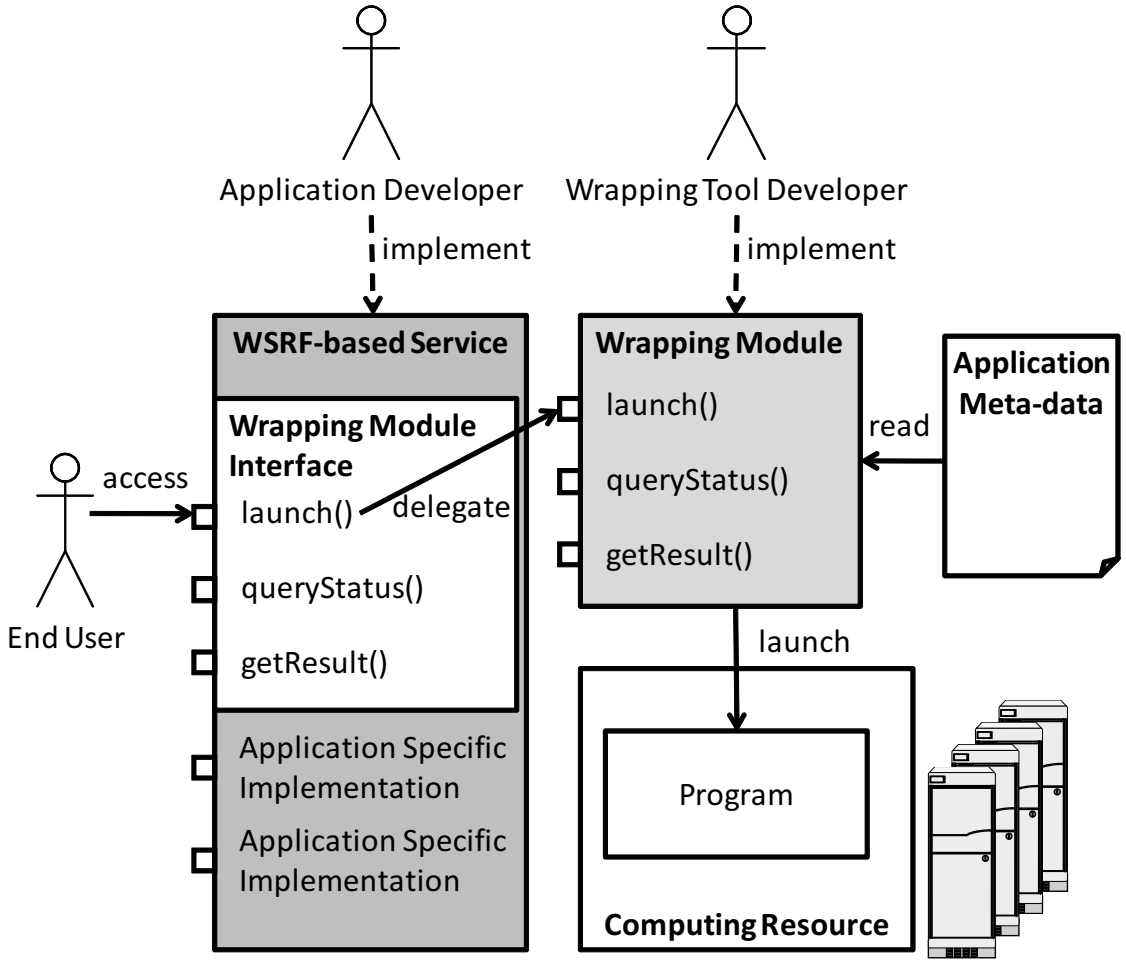

Figure 3.6: Extensible wrapping service model

service. In other words, the wrapping tool developer can concentrate on the design and implementation of wrapping tool, while the application developer can concentrate on solving application-specific problems.

In this model, the wrapping module provides a set of functions (launch(), queryStatus(), and getResult()) for encapsulating an existing program to a WSRF-based service. If these functions are not sufficient for the development of WSRF-based service, the application developer can further develop new interfaces of his/her WSRF-based service by him/herself. In addition, if the names of the functions of the wrapping module (i.e., “launch”, “queryStatus”, “getResult”) are not appropriate for the WSRF-based service, the application developer can prepare his/her own interfaces instead of them.

In order to execute and manage an existing program on a computing resource, this model reutilizes the same idea as the wrapping service model. Therefore, the program handled in this model has to be a command-line program. However, unlike the wrapping

service model, the application developer can handle the intermediate status and data of the program with application specific implementation under the proposed model. Thus, this proposed model is applicable to not only a simple command-line program but also a program which outputs intermediate data needed to be transferred to other programs.

However, there is a big demerit although the proposed model facilitates the application developer's development of a WSRF-based service. The demerit is that the proposed model requires the application developer to build a WSRF-based service using the wrapping module by him/herself in addition to the preparation of application meta-data whereas the conventional wrapping methods just require the application developer to prepare an application meta-data. For avoiding this demerit, a tool which reduces the application developer's implementation and configuration works is proposed in the section 3.4.4.

3.4 Opal Operation Provider (Opal OP)

The idea of modularizing a target application as a wrapping module is not a new idea because this idea is similar to the one of making some functions a program library. Importantly, the challenge here is how to technically establish the extensible Grid-enabling wrapping method using the extensible wrapping service model, and how to provide a tool which facilitates the development of WSRF-based service from the existing application.

To this end, the author considers *operation provider*, which is one of implementation techniques used in GT4 to build plug-in modules, as a building block in establishing the extensible Grid-enabling wrapping method based on the extensible wrapping service model. Also, the combination of the operation provider technique and Opal, which is a conventional wrapping tool for building a wrapping module, is considered to be a good solution. From this consideration, the author has come up with a new wrapping tool named "Opal Operation Provider (Opal OP)". The Opal OP allows the application developer to develop a WSRF-based service from an existing program, and the Opal OP is composed of two important components of the *Opal OP module* as a wrapping module, and the *Opal OP toolkit* which facilitates the development of WSRF-based service from an existing program.

3.4.1 Overview of Opal OP

Figure 3.7 shows the software stack of Opal OP on top of the Globus Toolkit architecture. The Opal OP module and Opal OP toolkit were developed in this research. The dark gray boxes show software components implemented in the Globus Toolkit. The Opal OP module implements a wrapping module for Opal functions as one of the operation providers on the

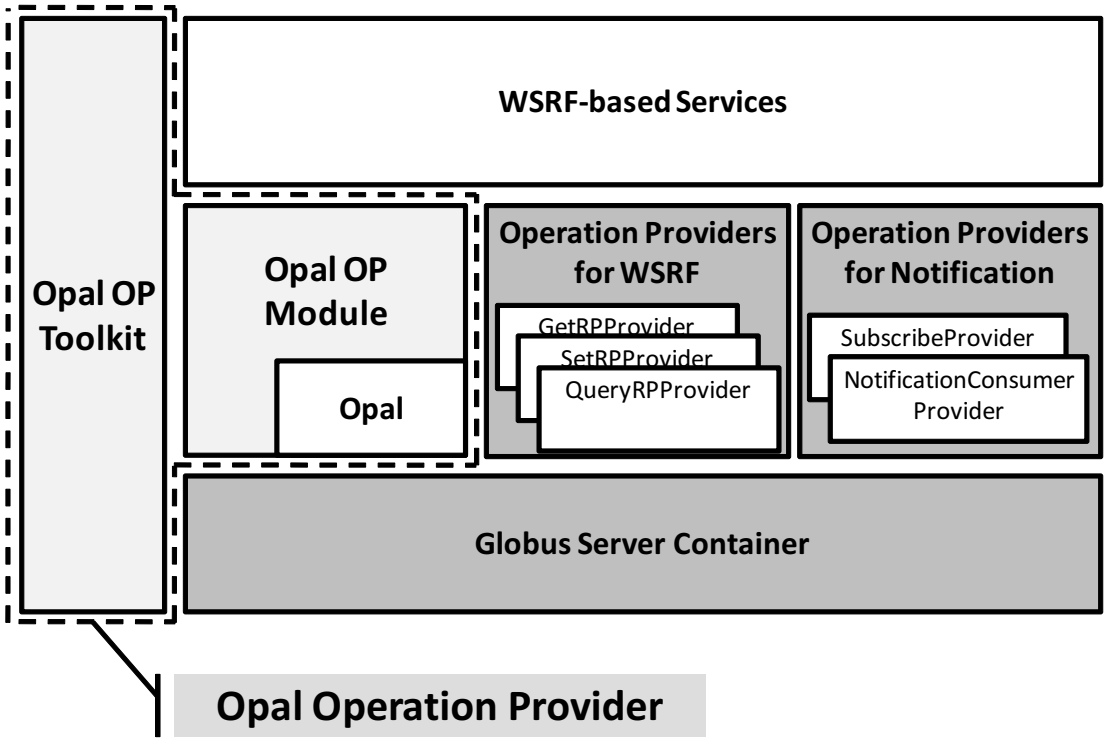

Figure 3.7: Software stack of the Opal OP in Globus Toolkit architecture

Globus Toolkit so that the application developer can use the wrapping module in the same manner as other operation providers of the Globus Toolkit. The application developer imports the functions of Opal OP into his/her WSRF-based service, and also he/she is allowed to extend and further develop the WSRF-based service. Opal OP toolkit supports the procedure from the generation of the WSRF-based service until deployment, and also provides common command-line tools to access the WSRF-based service.

This section explains how Opal OP is implemented in detail. For this explanation, the operation provider as an implementation technique for Opal OP module is described first. Next, the detail implementation of Opal OP and Opal OP toolkit is described.

3.4.2 Operation Provider

Under the GT4 architecture, operation provider is believed to be a good solution to implement common features that can be shared with various WSRF-based services. GT4 has a software stack as shown in Fig. 3.7. In the software stack, WSRF-based services are handled on top of the Globus server container. Operation providers are pluggable modules, from which the WSRF-based services can import the necessary operations. For example, the primary features of GT4 such as the management operation of resource properties

(i.e., a prominent feature of WSRF) and the asynchronous communication operation (WS-Notification [47]) are realized as operation providers.

The advantage of the operation provider is ease-of-use. The WSRF-based service can import the set of operations of the operation provider without any modification of the WSRF-based service. The implementation method of operation provider is very similar to the implementation method of Web service. In fact, an operation provider is composed of an interface definition described in WSDL and an implementation of the interface, similar to a Web service. On the other hand, as a possible disadvantage of operation provider, the configuration and implementation works increase.

As described above, taking the importance of the operation provider on GT4 architecture and its advantage into consideration, operation provider is an appropriate technique for implementing Opal OP module although the increase of configuration and implementation works is considered to be a disadvantage. Therefore, this study adopts the operation provider technique as an implementation technique for the Opal OP module.

In the following, how to use operation providers for a WSRF-based service is introduced for further discussion later on the implementation of the Opal OP module. The operation provider is used based on the following three steps:

- **[Step 1]** specifying the operation provider's interfaces to import into a WSRF-based service in the WSDL file of the WSRF-based service
- **[Step 2]** generating a unified WSDL definition which imports all operations from operation providers to the WSRF-based service
- **[Step 3]** binding the implementation of the operation providers whose operations are imported to the WSRF-based service

[Step 1] The first step is to specify the operation provider's names to the *extends* attribute in the WSDL file of the WSRF-based service. The top of Fig. 3.8 shows an example of a WSDL definition. In this example, the WSRF-based service named *SampleService* imports two operation providers: *GetResourceProperty* and *NotificationProducer*. These two are operation providers for the basic functions of GT4 for WSRF and WS-Notification. The bottom of Fig. 3.8 shows a *portType* definition of the first operation provider imported in *SampleService*. This operation provider has the operation named *GetResourceProperty*. Based on this configuration, the *SampleService* imports this operation, *GetResourceProperty* from WS-ResourceProperties.wsdl. For the first step, the application developer has to prepare for these configurations by hand.

Figure 3.8: sample_service.wsdl

[Step 2] The second step is to generate a unified WSDL definition containing all operations from operation providers whose operations are imported to the WSRF-based service. This process is called *flatten* and is executed by the building tool shipped with GT4. A generated large WSDL is called *flatten WSDL*. Figure 3.9 diagrams the *flatten process*. In the example shown in the figure, a WSRF-based service extends two operation providers. One of the operation providers has two operations, operationA, and operationB. The other operation provider has an operation, operationC. The WSRF-based service has two operations, appspecificD, and appspecificE. All of these operations are aggregated into a large flattened WSDL. The application developer has to manage generating a flattened WSDL.

[Step 3] The last step is to bind the implementation of operation providers and the

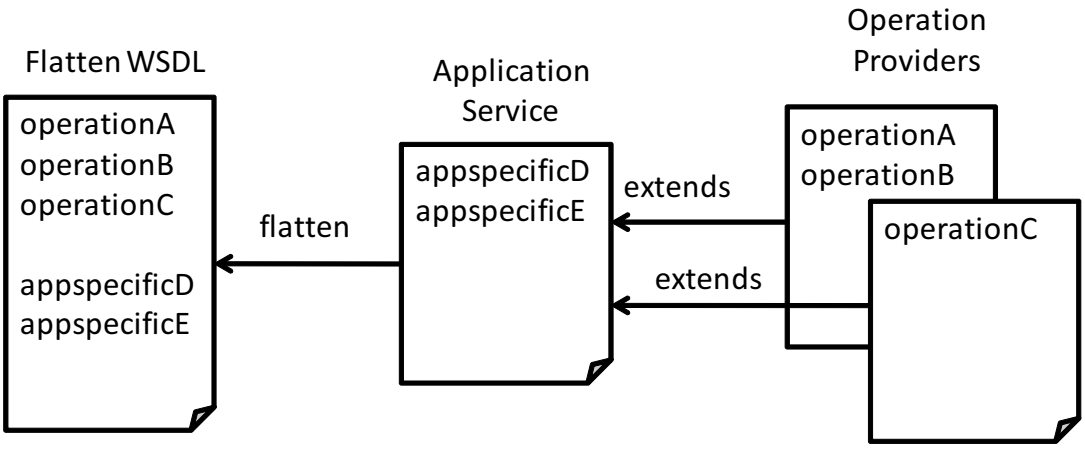

Figure 3.9: Flatten process

```
<service name="examples/SampleService“
        provider="Handler"  use="literal"  style="document">

   <wsdlFile>
      share/schema/samples/sample_service.wsdl
   </wsdlFile>

   <parameter  name="className“
         value="sample.services.impl.SampleService"/>

   <parameter  name="providers“
      value="GetRPProvider
               SubscribeProvider
               GetCurrentMessageProvider"/>
   . . .
</service>
```

Figure 3.10: deploy-server.wsdd

WSRF-based service. In building the Web service, the developer needs to describe the Web Services Deployment Description (WSDD) file to specify the binding between service interface (i.e., WSDL file) and the corresponding service's actual implementation. Likewise, in case of the WSRF-based service, the binding information between WSRF-based service interface and the operation provider's implementation is specified in a WSDD file. Figure 3.10 shows an example WSDD file of a WSRF-based service, *deploy-server.wsdd*. In this example, the WSDD file binds a WSDL file, *sample_service.wsdl* and a Java class implementation, *sample.services.impl.SampleService*. Furthermore, three operation providers of GetRPProvider, SubscribeProvider, and GetCurrentMessageProvider are specified in the WSDD file. These three are operation providers which implement the basic functions of GT4, WSRF and WS-Notification. The Globus container refers these elements of the WSDD file to load Java class implementations, and binds the implementations with the WSRF-based service.

In this way, the application developer can import the functions of several operation providers into his/her WSRF-based service. The important advantage of using the operation provider technique is that the application developer needs to write no additional program codes in order to integrate the operations defined as an operation provider into a WSRF-based service although the application developer still needs to prepare several configuration files.

3.4.3 Design and Implementation of Opal Operation Provider Module

This subsection describes the design and implementation of the Opal OP module. Figure 3.11 illustrates how the Opal OP module as a wrapping module works. In this illustration, any WSRF-based service can import the operations of the wrapping module in a plug-in manner. To realize this Opal OP module, Opal has been extensively reutilized. In this subsection, how the functions of Opal were reutilized and ported into a new wrapping module, the Opal OP module, is described.

The reason why Opal has been reutilized as an implementation technology to establish the extensible Grid-enabling wrapping method is explained from the following two reasons. The first reason is that Opal has been implemented as a standard Web service for wrapping existing application, and the functions of Opal are very simple and limited for wrapping a target program as a Web service. As mentioned in section 3.4.2, the implementation method of an operation provider is very similar to an implementation method of a Web service. Therefore, Opal is considered to be easily ported as an operation provider due to the implementational similarity between Opal and the operation provider. The second

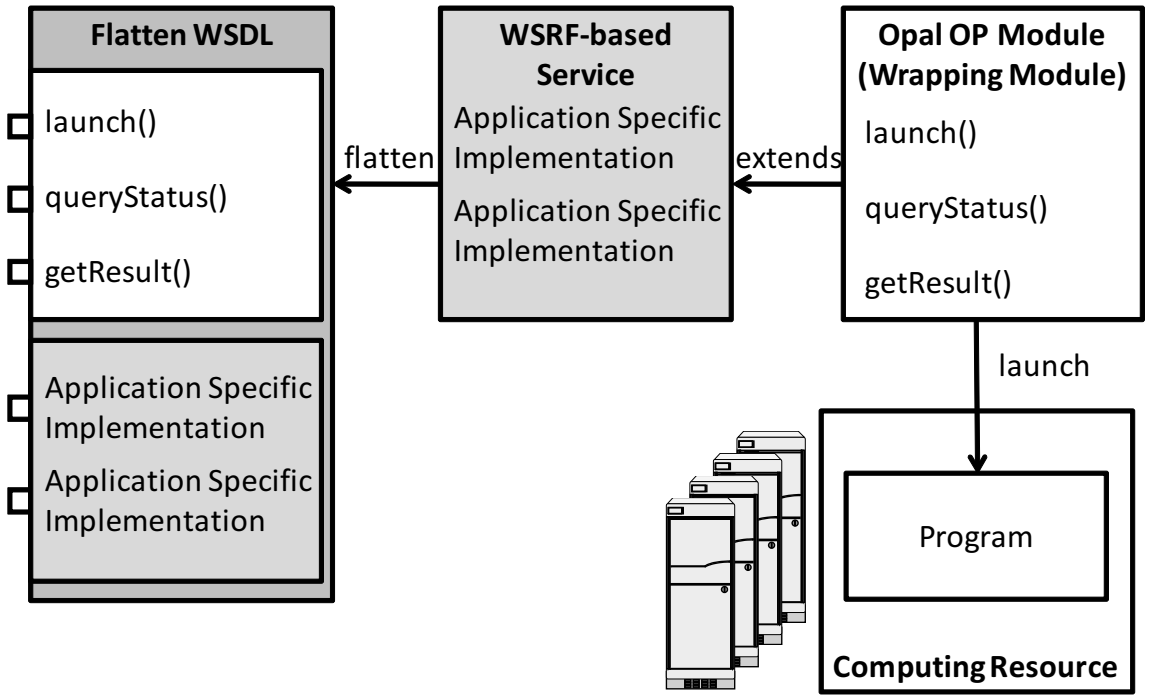

Figure 3.11: Overview of Opal OP

reason is that the simple implementation of Opal is suitable to the implementation of Opal OP module. As described in section 2.4.1, the wrapping module should not have any unnecessary functions except for wrapping a target program for security reasons. Therefore, the simple implementation of Opal is advantageous.

The important factor to consider in porting the functions of Opal into the Opal OP module as a operation provider is the state management functionality of the job. If the state management functionality was ported in a way not following the WSRF standard, the implementation of the Opal OP module would prevent the interoperation among WSRF-based services. Thus, the functionality is ported to the Opal OP module while conforming to the WSRF standard.

Opal provides the following interfaces and functions for state management to allow the end user to control the five states of the command-line program described in section 3.2.1.

- **[launchJob]**: launching a job into computing resources
- **[queryStatus]**: querying the job status
- **[getOutput]**: retrieving the result of the job

The mechanism and usage for handling the states of a job with these functions is shown in Fig. 3.12. When an end user requests starting a job with input files and arguments, Opal

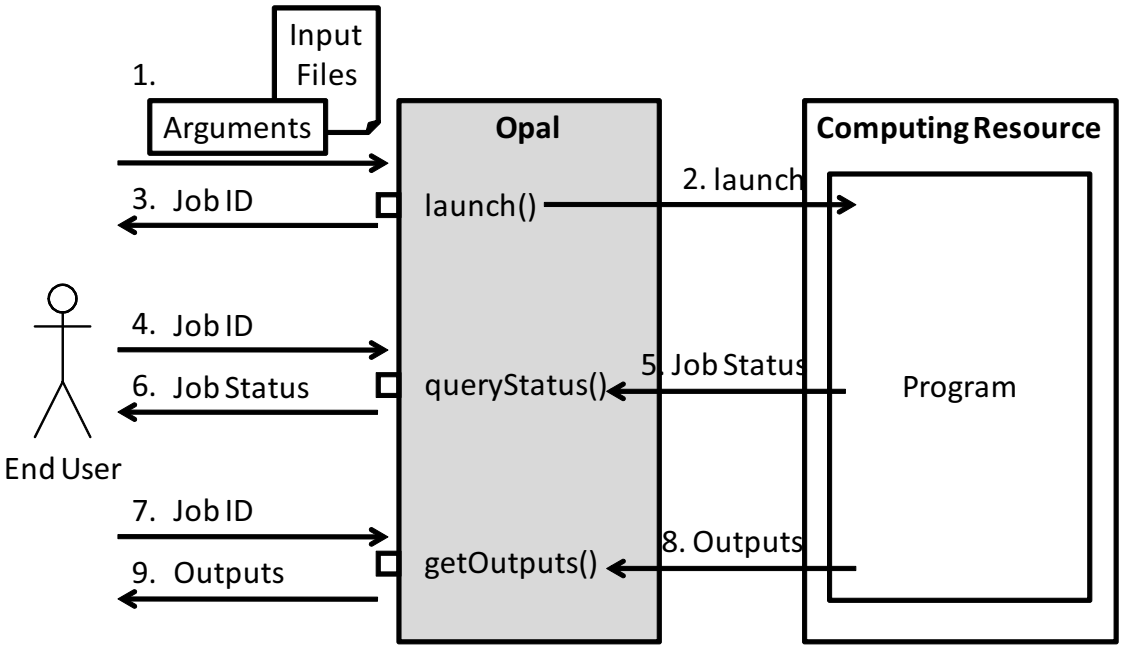

Figure 3.12: Usage of Opal

launches a job for the target program on a computing resource and returns the job ID to the end user. Once a job is launched, the end user can query the status of the job (e.g., running, done, abort), and retrieve the result with the job ID.

Description of how each of these functionalities for state management in Opal is ported to a new Opal OP module so that it conforms to the WSRF standard is as follows:

[launchJob] To handle job status, the launchJob function of Opal returns a unique ID as a job ID corresponding to the job status in response to a request from the end user. The Opal allows the end user to access the job status by specifying the job ID in subsequent requests. As mentioned in section 2.1, in the WSRF standard, the states of a WSRF-based service must be defined as resource properties in a WSDL file, and exposed in an architecture-independent way. The author, therefore, has replaced the original implementation of Opal with resource properties according to the WSRF standard as follows.

The author has newly defined a resource property representing the job ID and the job status as shown in Fig. 3.13 to expose job status as a resource property defined in the WSRF standard. The resource property is named OpalOPRP, and defined in the WSDL file of the Opal OP module. This definition is imported into each WSRF-based service WSDL files during the flatten process. By defining the states as resource properties, the states are automatically bound to subsequent requests of the end user. The end user, therefore, no longer needs to specify the job ID explicitly every time the end user accesses the service.

```
.
.
<!– Definition of  OpalOP Resource Properties-->
<xsd:element name="JobID" type="xsd:string"/>
 <xsd:element name="Status" type="types:StatusOutputType"/>
 <xsd:element name="OpalOPRP">
   <xsd:complexType>
     <xsd:sequence>
       <xsd:element maxOccurs="1" minOccurs="1" ref="types:JobID"/>
       <xsd:element maxOccurs="1" minOccurs="1" ref="types:Status"/>
     </xsd:sequence>
   </xsd:complexType>
 </xsd:element>
.
.
 <wsdl:portType name="AppServicePortType"
    wsrp:ResourceProperties="types:OpalOPRP">
.
.
</ wsdl:portType>
.
.
```

Figure 3.13: Definition of Opal OP resource property

[queryStatus] The job status is defined as a resource property so that it conforms to the WSRF standard. Furthermore, the WSRF standard prescribes that the resource property can be accessible through a common interface named *getResourceProperty*. As the Opal OP module has been developed based on WSRF, the Opal OP module does not need to provide another interface to access the resource property. In the Opal OP module, the function of queryStatus is kept to maintain backward-compatibility with the original Opal.

[getOutput] In the case of the original implementation of Opal, the end user needs to specify a job ID when the end user requests results through the getOutput interface. In the Opal OP module, the job ID is implemented in a resource property, and automatically bound to the end user's request. The end user, thus, does not need to specify a job ID anymore. For the implementation of getOutput, the author has removed the argument for the job ID from the getOutput interface.

3.4.4 Design and Implementation of Opal OP Toolkit

This subsection describes the implementation of the Opal OP toolkit which facilitates the development of WSRF-based services with the Opal OP module. As mentioned in section 3.3, under the proposed extensible wrapping service model, the application developer has to prepare a WSRF-based service implementation and configurations as well as an application meta-data, whereas the conventional wrapping methods just require writing the application meta-data. In addition, to import the wrapping module implemented as an operation provider into his/her WSRF-based service, the cumbersome configuration works described in section 3.4.2 are still necessary.

The Opal OP toolkit has been developed to minimize these cumbersome implementation and configuration works imposed on the application developer when the application developer sets up and deploys the wrapping module which encapsulates a target application. In summary, the Opal OP toolkit attempts to allow the application developer to develop a WSRF-based service by just writing a single configuration file. This means that the Opal OP toolkit aims to minimize the cumbersome works to the same level of works required in utilizing the conventional wrapping methods. Furthermore, the Opal OP toolkit also provides the end user with a simple command-line tool to access the WSRF-based service developed with the Opal OP module so that the end user can access the WSRF-based service without writing codes.

The Opal OP toolkit alleviates the burdens that come from the following two works. The first work alleviated by the Opal OP toolkit is the implementation and configuration works during the development, building, and deployment process of a service using the Opal OP module. The Opal OP toolkit generates the template implementation codes and configuration files for the service, which results in the application developer not needing to write any codes to use the Opal OP module. Also, the Opal OP toolkit automates the process from the building to the deployment of the service into a Globus toolkit container.

The second work alleviated by the Opal OP toolkit is the developing work of the client program to the service. The Opal OP toolkit provides a suite of command-line tools for accessing the WSRF-based service developed with the Opal OP module. Specifically, the suite of command-line tools provided by the Opal OP toolkit includes the tools to request launching a job, querying the job status, and retrieving a URL for the result of the job. Without this suite of command-line tools, the end user would need to develop a client program that uses WSRF-based SOAP API to access the service. Even if the end user is not familiar with WSRF-based services technologies, the end user can easily use the

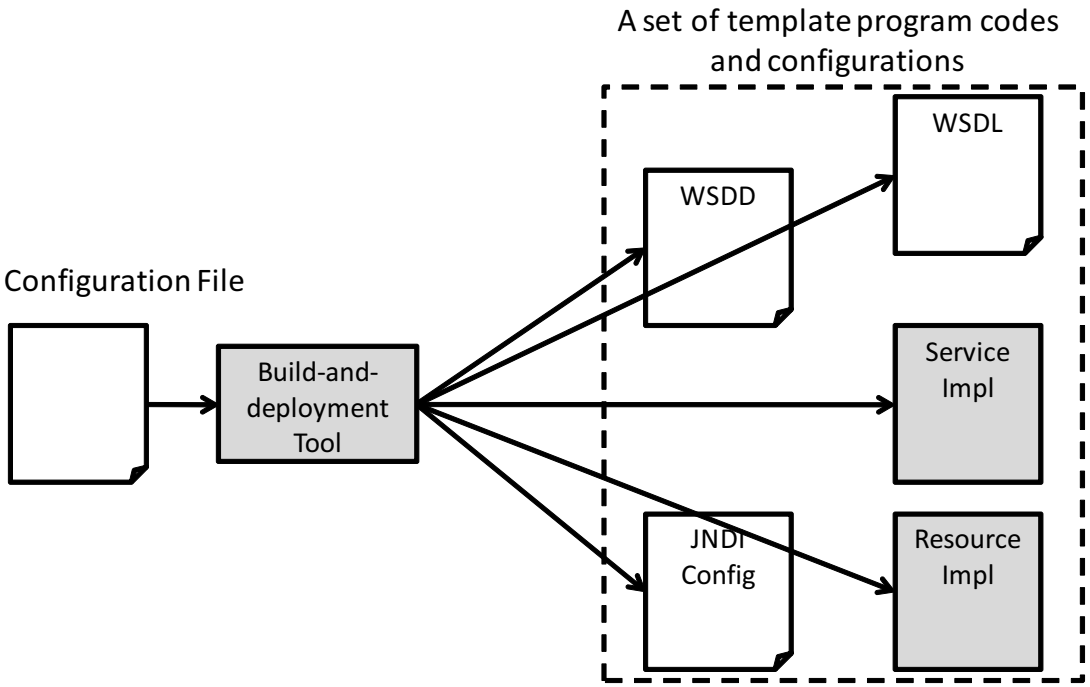

Figure 3.14: Concept of the build-and-deployment tool

```
$ ant -f build-opal.xml -propertyfile configuration-file new-service
$ cd services/Dock
$ ant deploy
```

Figure 3.15: An example command-line sequence for developing a WSRF-based service from an existing program

wrapping services through these command-line tools.

Below, how these works are alleviated by the Opal OP toolkit is explained in detail.

1. Build-and-deployment Tool for Opal OP

The basic idea of the build-and-deployment tool for the Opal OP is to generate a set of templates for implementation files and configuration files of a WSRF-based service utilizing the Opal OP module from as small the number of configuration files as possible. From this consideration, the author has designed and implemented the build-and-deployment tool as shown in Fig. 3.14 so that the toolkit automatically generates a set of template implementation files and configuration files from a single configuration file prepared by the application developer by hand.

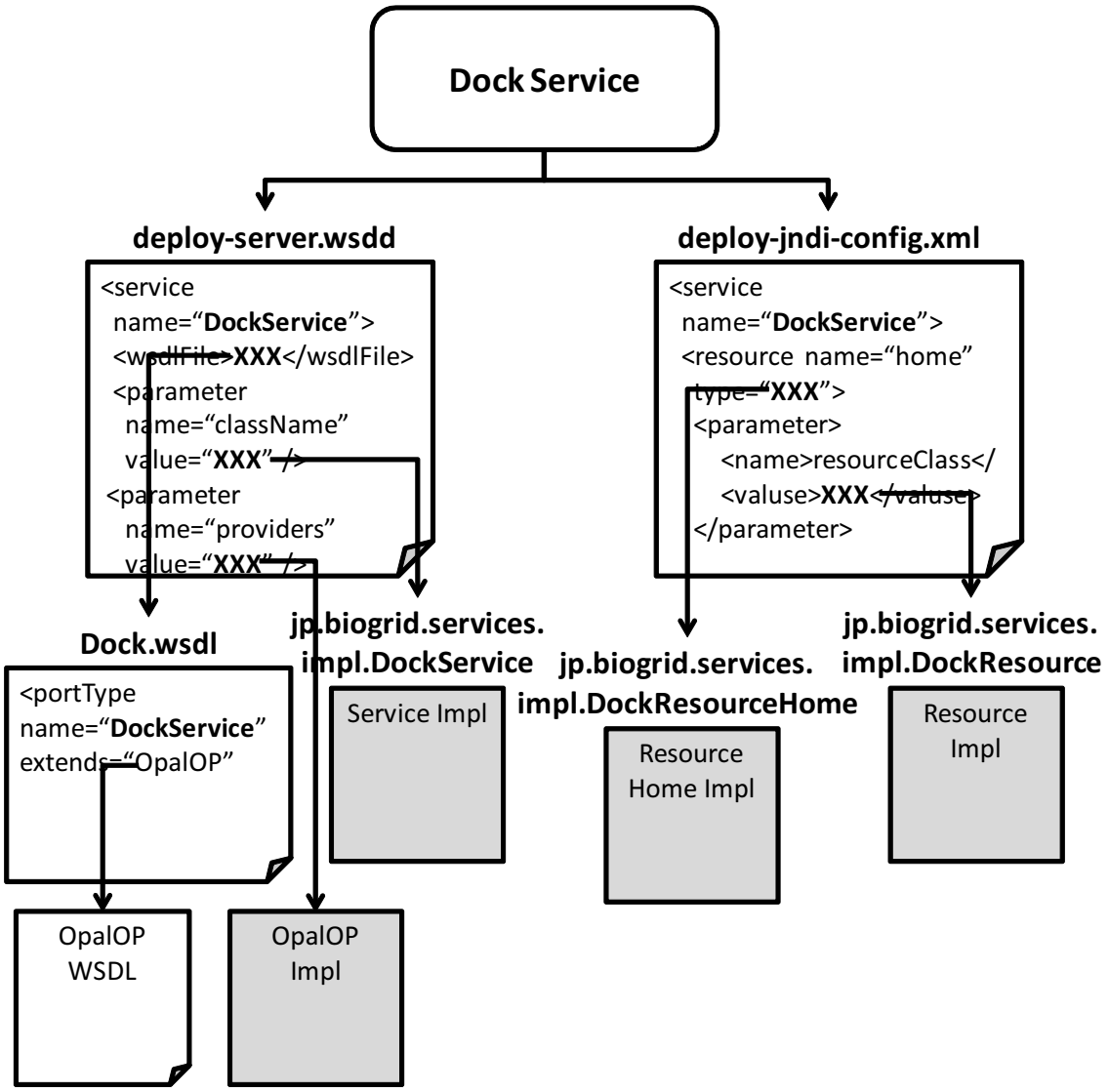

Figure 3.16: Overview of dependency among a WSRF-based service, interfaces and implementations

```
interface.name=Dock
binary.location=/opt/dock6/bin/dock6mpi
target.namespace=http://biogrid.jp/namespaces/DockService
package=jp.biogrid.services.dock
stubs.package=jp.biogrid.stubs.dock
prefix.publish.path=example/dock
factory.target.namespace=http://biogrid.jp/namespaces/DockFactoryService
```

Figure 3.17: DockService.properties

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="@SERVICE_NAME@"
  targetNamespace="@TARGET_NAMESPACE@"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="@TARGET_NAMESPACE@“
.
.
<portType name="@INTERFACE_NAME@PortType"
  wsdlpp:extends="wsrpw:GetResourceProperty opal:AppServicePortType"
  wsrp:ResourceProperties="tns:@INTERFACE_NAME@ResourceProperties">
.
.
```

Figure 3.18: An example of template source for WSDL

With this build-and-deployment tool, the development of a WSRF-based service from an existing program is performed as follows: 1) creating a configuration file used in the code generation of the WSRF-based service, 2) executing a toolkit command to generate the WSRF-based service, and 3) executing a toolkit command to build and deploy the generated WSRF-based service into a Globus server container. The example command-lines which the application developer needs to execute are shown in Fig. 3.15. These works are believed to be same level of the works required in utilizing the conventional wrapping methods, which just require writing an application meta-data. In the following, how the build-and-deployment tool realizes this work reduction is explained.

To realize this mechanism, the author has leveraged parameter dependency among a set of implementation files and configuration files necessary for developing a WSRF-based

service. Figure 3.16 illustrates an example of parameter dependency among interface definition files and implementation files necessary for a WSRF-based service for a Docking simulation program [48]. These interfaces and the actual implementations are bound with the WSRF-based service in configuration files. To define a WSRF-based service, there are two important configuration files: *deploy-server.wsdd* and *deploy-jndi-config.xml*. The former specifies interface definitions (WSDL) and service implementations, while the latter specifies resource implementations.

Another point the author focuses on is to make sure that these configuration files have the same or redundant descriptions with each other. Only a few parameters in the configuration files specify the binding among interface definition files and implementation files. Furthermore, such parameters can be automatically determined if the name of service and implementation files are determined. Through this investigation, the author has found that the parameters as shown in Fig. 3.17 are the smallest ones for generating a set of templates for implementation and configuration files.

Figure 3.18 is the example of the template WSDL file. The build-and-deployment tool first extracts parameters from properties file shown in Fig. 3.17, and then replace keywords starting with " @ ", like @*SERVICE_NAME*@ in Fig. 3.18 with the corresponding parameters. Table 3.1 shows the total lists of keywords used for generating a set of template implementation files and configuration files. The actual values corresponding to these keywords are extracted from the properties files provided by the application developer (configuration file in Fig. 3.14), and a set of template implementation files (WSDD, JNDI, and WSDL files in Fig. 3.14) and configuration files (service impl and resource impl in Fig. 3.14) are generated from these extracted values automatically. For further development of the WSRF-based service, the application developer has to extend the generated implementation files by him/herself.

More specifically, this build-and-deployment tool assumes that the generated WSRF-based service encapsulates a command-line program through the Opal OP module and provides the fixed set of interfaces. Based on this assumption, the build-and-deployment tool can have a set of templates as shown in Fig. 3.18. Also, the number of items required for the configuration file which the application developer has to prepare can be reduced to seven. Therefore, the build-and-deployment tool can generate actual template files for configuration and implementation from the 7-line properties file as shown in Fig. 3.17 by replacing the keywords with the actual values in the template files. With this mechanism, the tool helps the generation of a set of template implementation files and configuration files necessary for building a WSRF-based service. This build-and-deploy tool has been

Table 3.1: Description of the keywords required for generating templates

Keyword	Description
@GLOBUS_LOCATION@	Where the Globus Toolkit is installed. This keyword is determined from a system environment value, GLOBUS_LOCATION (e.g., /opt/globus).
@BINARY_LOCATION@	The installation path of the target program wrapped with Opal OP. The value of “binary.location” properties is used (e.g., /opt/dock6/bin/dock6mpi).
@INTERFACE_NAME@	The interface name of the generated service. The value of “interface.name” is used (e.g., Dock).
@SERVICE_NAME@	The service name of the generated service. This keyword is determined from “interface.name” (e.g., DockService).
@PACKAGE@	The package name for the generated service source codes. The value of “package” is used (e.g., jp.biogrid.services.dock).
@STUBS_PACKAGE@	The package name for the generated stub source codes. The value of “stubs.package” is used (e.g., jp.biogrid.stubs.dock).
@PACKAGE_DIR@	The name of directory where the generated source codes are stored. This keyword is determined from “package” (e.g., /jp/biogrid/services/dock).
@TARGET_NAMESPACE@	The namespace of XML file for the WSDL of service. The value of “target.namespace” is used (e.g., http://www.biogrid.jp/namespaces/dock/DockService).
@SCHEMA_PATH@	The name of directory where the generated WSDL file is stored. This keyword is determined from “interface.name” (e.g., DockService).
@FACTORY_INTERFACE_NAME@	The interface name of the generated factory service. This keyword is determined from “interface.name” (e.g., DockFactory).
@FACTORY_TARGET_NAMESPACE@	The namespace of XML file for the WSDL of factory service. The value of “factory.target.namespace” is used (e.g., http://www.biogrid.jp/namespaces/dock/DockFactoryService).
@FACTORY_SCHEMA_PATH@	The name of directory where the generated WSDL file is stored. This keyword is determined from “interface.name” (e.g., DockFactoryService).
@PREFIX_PUBLISH_PATH@	The URI to publish the service on the Globus server container. The value of “prefix.publish.path” is used (e.g., dock).
@GAR_FILENAME@	The name of the archive file which contains the service implementation. This keyword is determined from “package” and “interface.name” (e.g., jp_biogrid_dock_services_Dock.gar).

developed as an *ant*-based tool [49].

The assumption described above restricts the targets which the proposed method and tool can cover to command-line programs. However, in terms of developing a WSRF-based service for an existing scientific program, the restriction derived from this assumption is not a problem. Most programs treating a scientific problem run for a long time without the interaction with end users, and then they are implemented as command-line programs. Such programs therefore can be taken as WSRF-based services by the proposed method.

However, the WSRF-based service can be covered with Opal OP is limited than the general WSRF-based services built without Opal OP. Generally, the WSRF technologies realize a Grid environment composed of various types of WSRF-based services which communicates each other with XML messages standardized by SOAP. For example, a Grid environment includes WSRF-based services handling account management and database management system. These kinds of WSRF-based services are studied as Grid Account Management Architecture (GAMA) and OGSA Data Access and Integration (OGSA-DAI), respectively. The proposed method does not cover these kinds of WSRF-based services built from programs other than command-line programs used for scientific computation.

2. Tools to Access WSRF-based Services Developed with Opal OP

To use the WSRF-based service, the end user has to write program codes with WSRF-based service API. The provision of command-line tools for accessing the WSRF-based service minimizes this end user's work. The Opal OP toolkit provides the following command-line tools for launching a job and querying the job status: *opalop-jobrun* and *opalop-jobquery*, respectively.

In practice, the implementation of each WSRF-based service generated by the Opal OP toolkit is individually different from each other. On the other hand, all WSRF-based services extend a common interface of the Opal OP module in WSDL. With these tools, the end user can access the WSRF-based service developed with the Opal OP module without writing any client program codes.

3.5 Evaluation and Discussion

This section discusses the usability and effectiveness of the Opal OP from two aspects. The first aspect is how much the Opal OP reduces the work in developing a WSRF-based service from an existing program. The second aspect is how the Opal OP has been utilized for development of service-oriented Grid applications.

Table 3.2: Generated templates from Opal OP toolkit

Implementation files	lines
XXXFactoryService.java	53
XXXQNames.java	14
XXXResource.java	80
XXXResourceHome.java	20
XXXService.java	37

Configuration files	lines
XXX.wsdl	95
XXXFactory.wsdl	55
deploy-server.wsdd	29
deploy-jndi-config.xml	38
namespace2package.mappings	15
build.properties	8
opal_config.xml	8

3.5.1 Application Developer's Work Reduction

To discuss the usability of the proposed Opal OP, this section reviews the works for developing a WSRF-based service based on the proposed extensible wrapping service model. In this section, how the proposed Opal OP reduces the works for developing a WSRF-based service is discussed.

To reduce the works of developing a WSRF-based service from scratch, the proposed Opal OP provides the Opal OP module as an operation provider encapsulating an existing program. However, at least the set of implementation files and configuration files shown in Table 3.2 are still required in utilizing the Opal OP module. Table 3.2 shows the total lines of configuration and implementation files for which the application developer has to prepare in the case of developing a WSRF-based service of a typical command-line program. As this table shows, development using the Opal OP module requires many works related to the setup and configuration of the developed WSRF-based service based on the Opal OP module. However, the Opal OP toolkit, which was explained in section 3.4.4, reduces this amount of works imposed on the application developer to only a 7-lines configuration, by taking advantage of parameter-dependency and by removing the inherent redundancy among configuration files for setting up a WSRF-based service. This reduction is realized by the assumption that the WSRF-based service just encapsulates an

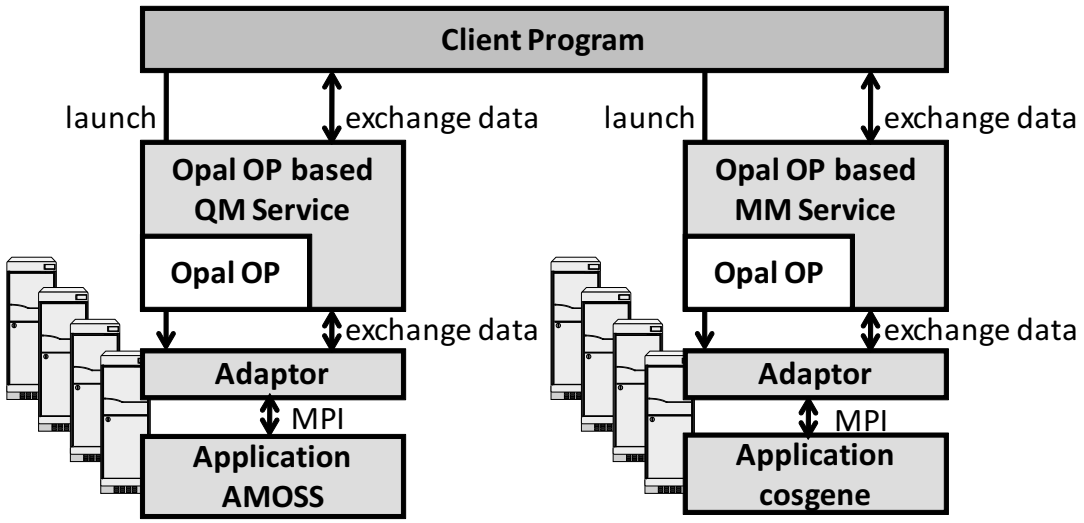

Figure 3.19: Overview of the QM/MM hybrid simulation system

existing program through the Opal OP module. Importantly, this 7-lines configuration file can work for the development of a WSRF-based service using the Opal OP module from any type of command-line program, and the application developer does not have to write any program code and configuration file in this process.

3.5.2 Case Studies

The author has been developing several simulation services using Opal OP. In this subsection, the author introduces a bio-molecular simulation system as an example of Grid applications using Opal OP. Also, this subsection shows a protein structure similarity search system and a drug docking simulation system as examples of easy service development.

QM/MM Hybrid Simulation System

The author has been developing a QM/MM hybrid simulation system to simulate bio-molecular behaviors. This system calculates forces interacting among atoms at short time steps (e.g. 0.5 femto-sec), and then simulates molecular behavior by repeating these calculations tens of thousands of times. The QM/MM hybrid simulation system consists of two simulation programs. One is AMOSS [50], based on Quantum Mechanics (QM), and the other is cosgene [51], based on Molecular Dynamics Mechanics (MM). The QM-based calculation is time-consuming, but has high accuracy. The MM-based calculation is fast, but has low accuracy. This research tries to develop a highly accurate and large-scale bio-molecular simulation by adopting QM-based calculations to the important part of the

simulation and MM-based calculations to the remaining parts.

To calculate molecular behavior, these two simulations have to exchange data at every time step while the two applications are running. This kind of requirement in computation cannot be handled by the traditional wrapping approaches. To handle this kind of requirement, a mechanism like Opal OP is needed that can handle application-specific problems. The author has implemented a QM/MM hybrid simulation system using the following three steps. Figure 3.19 shows an overview of the architecture of the system implemented by the Opal OP.

First, in order to hide the complexity of the dynamic process creation of QM and MM programs from a local scheduler and then enable the synchronization between the two programs, the author has developed adapter programs. Both QM and MM programs consist of several program modules, and the modules are executed and combined by a dynamic process-creation method (spwan) of MPI-2 [52-54]. The adapter programs hide such dynamic process creation and make the applications simple MPI programs. Also, these adapter programs help to synchronize QM and MM programs to exchange data using a traditional file-locking mechanism. When calculations of each simulation step start, the adapter creates a lock file, and then removes it after the step is completed. Each of QM and MM service checks the existence of the lock file on the computing resource where the service running, and determines the timing for transferring data.

Second, the author wrapped the adapter programs using Opal OP. For this process, the author did not need to write any codes. The author configured these services as parallel applications in the Opal's application meta-data file.

Finally, the author added operations to exchange data into the services generated by Opal OP. Also, the author added operations to check the lock files for synchronization. For this work, the author did not need to consider how the Opal OP wrapped the application.

Generally, application developers find it difficult to develop this kind of service from scratch. Using Opal OP, the author concentrated only on how to synchronize two applications in the process of development. This is the advantage brought about by Opal OP.

Protein Structure Similarity-search System and Drug-docking Simulation System

This subsection describes a protein structure similarity search system and a drug-docking simulation system developed by utilizing Opal OP.

The protein structure similarity-search system was developed as a Web portal with Java servlet technologies, and used a protein structure similarity searching program as a backend program [55]. To distribute the processes of the backend program, the system

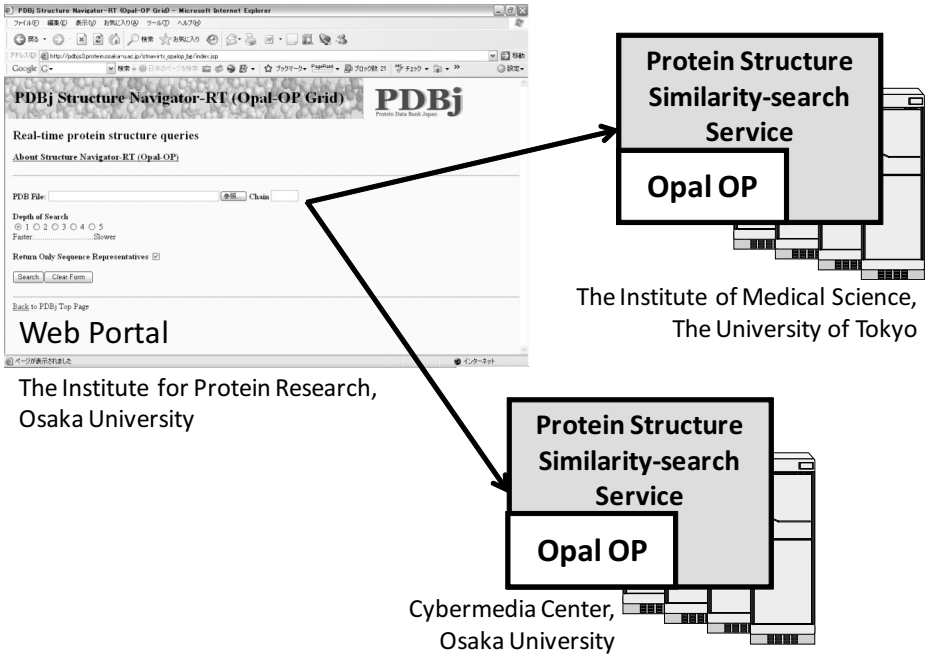

Figure 3.20: Protein structure similarity-search system

needed to use Grid technologies. A researcher of the Institute for Protein Research, Osaka University, who was not familiar with Grid technologies, had developed protein structure similarity-search services using Opal OP and a Web portal which accesses the services via the interfaces with SOAP (Fig. 3.20). In the development of this system, Opal OP helped to develop a WSRF-based service to handle the protein structure similarity searching program. The researcher had no knowledge of Grid technologies at the time of development, but it took approximately three weeks to build such a system. This system was developed on two cluster systems, and currently provides the protein structure similarity-search service as a part of services of Protein Data Bank Japan (PDBj) at the following site: `http://pdbjs3.protein.osaka-u.ac.jp/stnavirtx_opalop/`.

Another example utilizing Opal OP is the drug-docking simulation system which has been developed by a UCSD undergraduate student whose major is bio-engineering. The system uses DOCK, a docking program for drug discovery. To benefit from a large amount of computing resources, the Opal OP-based docking services, which wrap DOCK, were developed and deployed into Grid resources (Fig. 3.21). Although the student was not even familiar with computer science as well as Grid technologies, he was able to use the common command-line tools provided in the Opal OP toolkit to access the services. He used with ease the command-line tools in Perl scripting to build the distributed drug-docking simulation system. It took approximately three weeks to build this system, with most of

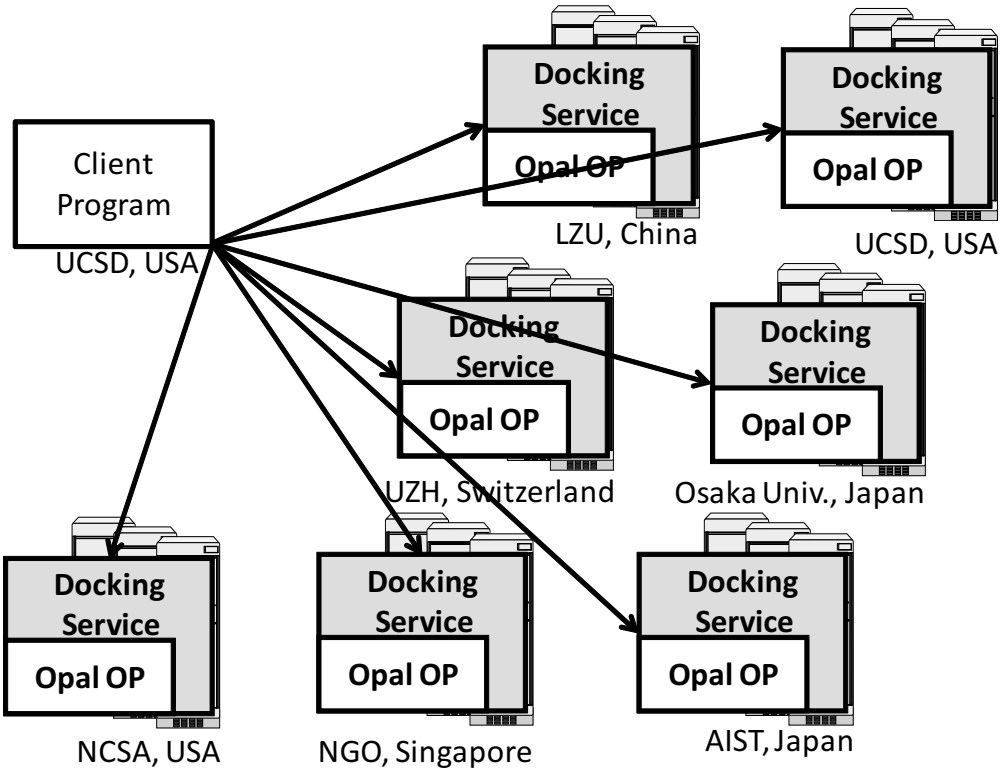

Figure 3.21: Drug-docking simulation system

the time spent learning Perl programming. In the development of this system, Opal OP helped the student to develop a WSRF-based service to handle the docking program and also supported to develop the client program. This system was developed on seven cluster systems, and used for screening approximately 2 million drug candidates in a week.

Table 3.3 compares these two examples. In both cases, the researcher and the student spent just ten percents of the overall development time for the WSRF-based services. On the other hand, they had difficulties and spent much time to construct the environment for their system because they are not experts of Grid technologies and information technologies. To construct these systems, they needed to install Globus Toolkit and deploy their WSRF-based services on each of computing resources. This deployment work was mentioned as the third stage of developing a Grid application in section 2.2. Although this dissertation introduced that this study did not cover this stage, this stage is needed to be taken into account from the viewpoint of the total development procedure.

These two examples show how the development of services is made simple by using Opal OP in comparison with the development of service using traditional wrapping meth-

Table 3.3: Summary of two examples using Opal OP

System name	System type	System environment	Reduced works with Opal OP	Weeks worked	Construction of environment	Learning technologies	Development of Grid service
Structure Navigator-RT	Web Portal	2 cluster systems (60 CPUs)	Development of a WSRF-based service	approx 3 weeks	70% (14 days)	20% (4 days)	10% (2 days)
Drug-Docking Simulation	Perl script	7 cluster systems (350 CPUs)	Development of a WSRF-based service and a client program	approx 3 weeks	60% (12 days)	30% (6 days)	10% (2 days)

ods. The Opal OP enables the building of services that wrap existing programs without knowledge of Grid technologies.

3.6 Concluding Remarks

This chapter established a new, flexible, and extensible wrapping method listed as a technical issue in Chapter 2. For this purpose, the author proposed a new wrapping model, the extensible wrapping service model and the corresponding extensible Grid-enabling method based on the model. Whereas traditional wrapping service methods such as Opal and GEMLCA provide a fixed implementation of the wrapping service, the method allows the application developer to extend and further develop his/her WSRF-based service. The advantage of the extensible Grid-enabling method is that the method enables the separate implementation of WSRF-based service and the implementation of a wrapping module. In other words, the extensible Grid-enabling wrapping method based on the model allows the application developer to concentrate on the further development of WSRF-based application from the existing application.

To realize the development based on the extensible Grid-enabling method, the author has developed a new wrapping tool named the Opal Operation Provider (Opal OP), which allows the application developer to develop a WSRF-based service from an existing program. Opal OP is composed of an Opal OP module and an Opal OP toolkit. The former helps the application developer to import the features of Opal into his/her WSRF-based services, and the latter helps the application developer to develop a WSRF-based service from an existing program without writing any additional program codes. To develop this tool, the author combined the operation provider technique and the Opal technology.

In the evaluation in this chapter, the author reviewed the Opal OP from two different aspects to verify the usability and effectiveness of the proposed Opal OP. Specifically, how

the Opal OP reduced the work involved in developing a WSRF-based service was investigated. The result indicates that the Opal OP reduced the configuring and implementing works into just writing a single configuration file in developing a WSRF-based service based on the extensible wrapping service model. Second, the author reviewed three actual examples of Grid applications developed with the proposed Opal OP as case studies. This evaluation and review showed that the application developer was able to develop his/her application as a Grid application with ease.

Chapter 4

Transparent Meta-Scheduling Architecture for Grid Applications

4.1 Introduction

Many scientific institutions and universities are attempting to redevelop the scientific applications they have developed so far as WSRF-based services so that their research collaborators can use such applications from remote sites. The Opal OP presented in Chapter 3 provides an easy way to build up an application as a WSRF-based service, thus helping the developer.

However, Opal OP does not provide sufficient tools to dramatically make the application developers work efficient from the standpoint of developing a service-oriented Grid application composed of multiple WSRF-based services. The remaining issue or the provision of a meta-scheduling method suitable for a service-oriented Grid application still must be achieved. As described in Chapter 2, the application developers deploy WSRF-based services to multiple sites for the demands of enhancing performance, balancing loads, and increasing fault tolerance. This fact means that end users have to selectively use a WSRF-based service among multiple ones from the standpoint of meeting the above demands.

In this chapter, the author proposes a new meta-scheduling architecture (MSSA: Meta-Scheduling Services Architecture) that allows the application developer to easily develop a Grid application composed of multiple WSRF-based services deployed on the Internet. This architecture focuses on providing a transparent interace to select a WSRF-based service from multiple WSRF-based services. For this purpose, this architecture takes advantage of the *factory pattern* technique, which is used in typical WSRF-based service for handling resource properties. Then, how the proposed scheduling architecture provides the transparent scheduling mechanism is detailed.

The rest of this chapter is organized as follows. In section 4.2, the requirements on

the meta-scheduler, which takes the important role of meta-scheduling WSRF-based services, are analyzed. After that, the meta-scheduling architecture is proposed in section 4.3, and evaluation and discussion of the architecture is presented in section 4.4. Section 4.5 concludes this chapter.

4.2 Requirement Analysis of Meta-Scheduler

This section first describes the different structures of the Grid application and the role of the meta-scheduler. Next, the requirements on the meta-scheduler are analyzed.

4.2.1 Meta-Scheduler

Figure 4.1 illustrates the different structures of the Grid application. The first case (a) shows the easiest way to use the WSRF-based service. An end user utilizes a WSRF-based service, which is deployed on a computing resource from a client program. The Opal OP allows the application developer to build the WSRF-based service from an existing program and provides a program template for the client program, as described in Chapter 3. In this case, therefore, most of development works necessary for building this environment (Fig. 4.1 (a)) is covered by Opal OP.

The second case (b) happens when each end user wants to pursue the improvement of performance and throughput or the enhancement of fault-tolerance independently of other end users. For this structure of Grid application, a WSRF-based service is first deployed in multiple computing resources by the application developer, and the WSRF-based services are shared among multiple users. In this structure, the end user has to check the availability and usability of the computing resources on which the WSRF-services are deployed in prior to the use of the WSRF-based services. Also, the end user must know the locations and URLs of his/her target WSRF-services in advance.

The third case (c) is the advanced mode of the second case (b). In this case, a meta-scheduler plays an important role in helping the end user's selection of a WSRF-based service. The meta-scheduler is expected to aggregate the end user's access requests to the WSRF-based services and then select an appropriate set of WSRF-based services on behalf of the end user. Recently, the development of a Grid application based on this structure has been increasingly demanded. However, there has been little research exploring the technical solutions which facilitate the development of a Grid application based on this structure composed of multiple WSRF-based services, whereas conventional meta-schedulers focus on just scheduling GRAM services. For this reason, this chapter focuses on how to techni-

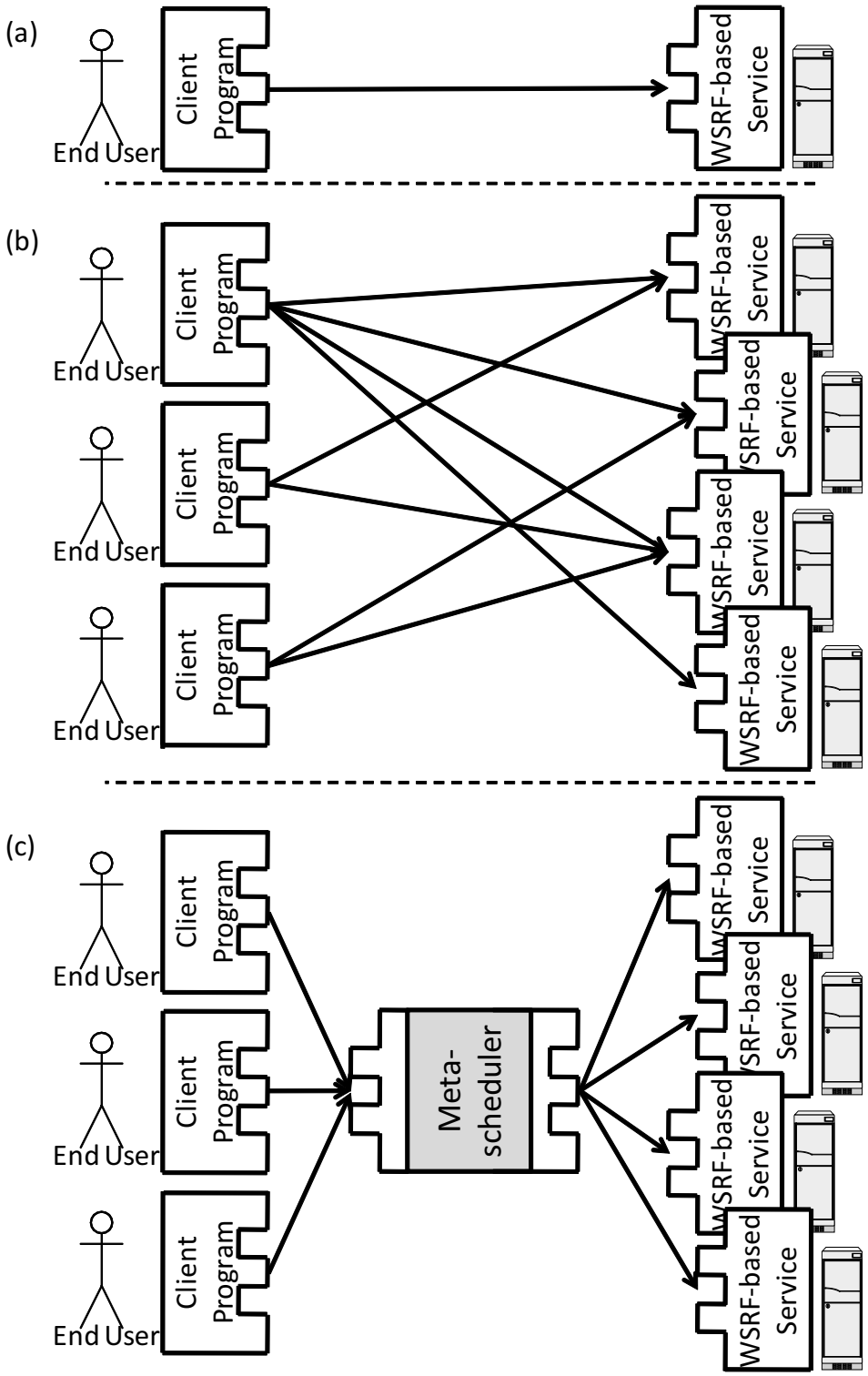

Figure 4.1: Different types of Grid application structures

cally realize this kind of meta-scheduler that allows the application developer to develop a Grid application composed of multiple WSRF-based services.

4.2.2 Requirement to Meta-Scheduler for WSRF-based Service

As shown in Fig. 4.1, the meta-scheduler mediates the communication between the WSRF-based service and its client program. For this reason, the interfaces provided by the meta-scheduler should be transparent to both the client program and the WSRF-based service. In other words, the meta-scheduler should be built so that the client program communicates with the meta-scheduler in the same manner it does with the WSRF-based service and vice versa. Otherwise, the client program and the WSRF-based services deployed on multiple sites must be modified so that they communicate with the meta-scheduler. Thus, the transparency of the meta-scheduler interface must be achieved to avoid the additional development work in introducing the meta-scheduler.

This consideration is lacked in the conventional meta-scheduler approach. As mentioned in section 2.3.2, the conventional meta-scheduler is designed to provide a global batch queuing service over only the job submission service, GRAM. Therefore, the conventional meta-scheduler does not need to assume to schedule various interface designs of WSRF-based service developed by the application developer. Moreover, in the batch queuing system, the ways for job submission and acquisition of the result are primary focused on, but little attention has been paid to the interaction between services and end users. Thus, the interface transparency is not an important issue in the conventional meta-scheduler.

Therefore, in order to address the meta-scheduler for WSRF-based services, the discussion on the conventional meta-scheduler is missing the point. For the purpose to achieve the interface transparency of meta-scheduler, a new meta-scheduling mechanism must be realized.

4.3 MSSA: Meta-Scheduling Services Architecture

Through the requirement analysis described in the previous section, the author has proposed a new meta-scheduling architecture named, “Meta-Scheduling Services Architecture (MSSA)” and MSSA toolkit that facilitates the development of a Grid application composed of multiple WSRF-based services based on MSSA. The primary feature of the MSSA is that the MSSA uses the factory pattern technique, utilized in WSRF-based service.

Before explaining the proposed MSSA, this section first explains the factory pattern

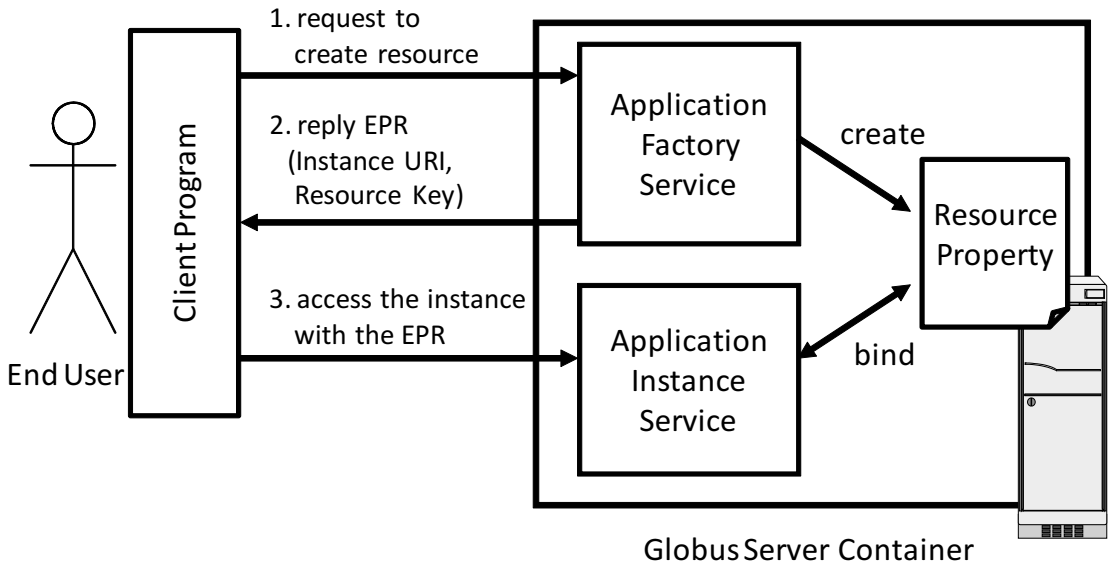

Figure 4.2: Factory pattern in WSRF-based service

technique in WSRF-based service. Next, this section introduces the proposed meta-scheduling architecture, MSSA, and explains the design and implementation of the MSSA in detail.

4.3.1 Factory Pattern in WSRF-based Service

As described in section 2.1, the WSRF standard realizes a stateful Web service. The WSRF standard prescribes a resource property in order to allow a Web service to hold its state and expose this state in the standard way. To create an instance of a resource property and then bind the instance of the resource property with a WSRF-based service, the WSRF-based service uses the factory pattern technique [17, 56].

Figure 4.2 shows the factory pattern in a typical WSRF-based service. In the factory pattern, services are roughly categorized into factory services and *instance services*. A factory service creates a resource property and binds the resource property with an instance service. The instance service is a service that provides actual functions of the WSRF-based service to end users.

This factory pattern works as follows. An end user first needs to send a "create" request to the factory service of interest on a Globus server container (step (1) in Fig.4.2). The factory service creates a resource property and returns an End Point Reference (EPR) (step (2) in Fig.4.2), which contains a URI of the instance service and a resource key. At this point, the end user can access the instance service via this EPR (step (3) in Fig.4.2). The

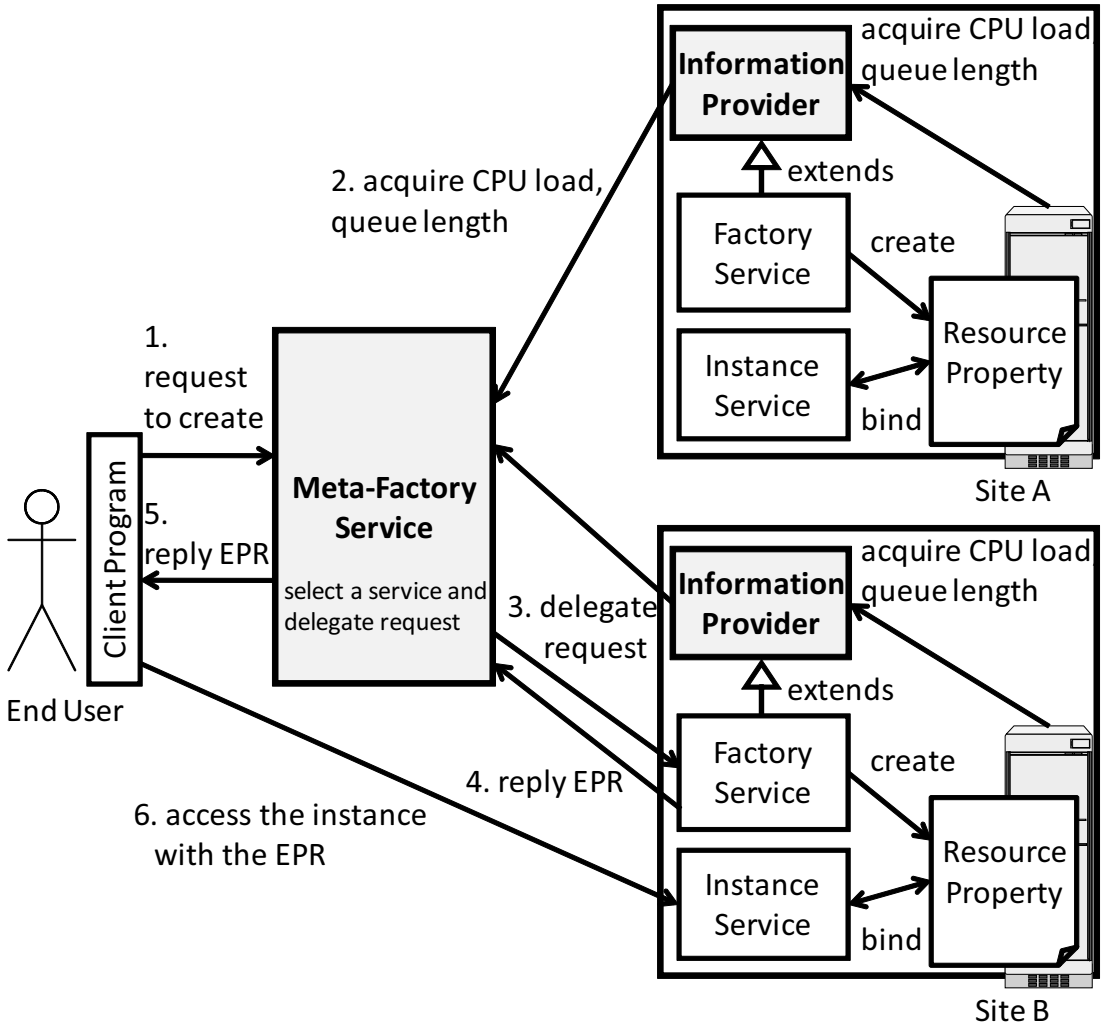

Figure 4.3: Meta-factory service in MSSA

Globus server container binds the end user request with appropriate resource property by referring the resource key contained within the end user request.

4.3.2 Overview of MSSA

The proposed MSSA makes use of the factory pattern technique to build a meta-scheduler that mediates the communication between the client program and WSRF-based services. Specifically, the author focuses on the fact that the end user always needs to access a factory service from the client before obtaining a URI of the corresponding instance service. In more detail, the proposed MSSA leverages this switching mechanism in the factory pattern technique from factory service to actual instance service to achieve the interface transparency.

Figure 4.3 shows the overview of the proposed MSSA. The MSSA is composed of two important components, "Meta-Factory Service" and "Information Provider". The meta-factory service mediates the communication between the client and a factory service of the selected WSRF-based service. Basically, this meta-factory service is a WSRF-based service whose role is as a meta-scheduler that receives a request from the end user via the client program and then delegates the request to the factory service of the WSRF-based service selected from multiple WSRF-based services. On the other hand, the information provider is a component implemented as an operation provider that provides functions to expose the information on computing resource (e.g., CPU load, queue length) where the WSRF-based service runs to the meta-factory service.

The author has designed this MSSA architecture to be performed as follows: 1) an end user sends a "create" request to the meta-factory service as a meta-scheduler instead of the factory service of the target WSRF-based service, 2) the meta-factory service acquires the necessary information on computing resource from information providers, 3) the meta-factory service determines which WSRF-based service should be used based on its own scheduling policy and then delegates the "create" request to the actual factory service of that WSRF-based service, 4) the actual factory service creates a resource property and then replies with an EPR (End Point Reference) to the meta-factory service, 5) the meta-factory service forwards the EPR to the end user, and 6) the end user can finally access the actual instance service selected by the meta-factory service.

Importantly, the factory pattern used in WSRF-based service has been originally leveraged for handling resource properties of the WSRF-based service. Usually, a factory service is supposed to create a resource property and reply the EPR of an instance service which is deployed on the same Globus server container where the factory service is deployed. On the other hand, in MSSA, the meta-factory service replies the EPR of the instance service which is selected as the result of scheduling process. This fact means that the MSSA uses the factory pattern to hide the scheduling process from the end user, and provides interface transparency to the end user.

4.3.3 Design and Implementation of Meta-factory Service

This section describes the design and implementation for meta-factory service as a meta-scheduler. The meta-factory service is a key component of the proposed MSSA. More technically, a common factory interface named "MssaFactoryInterface" introduced to the inside of the meta-factory service is an important element in designing the meta-factory service. The MssaFactoryInterface has been designed to satisfy the requirement for the

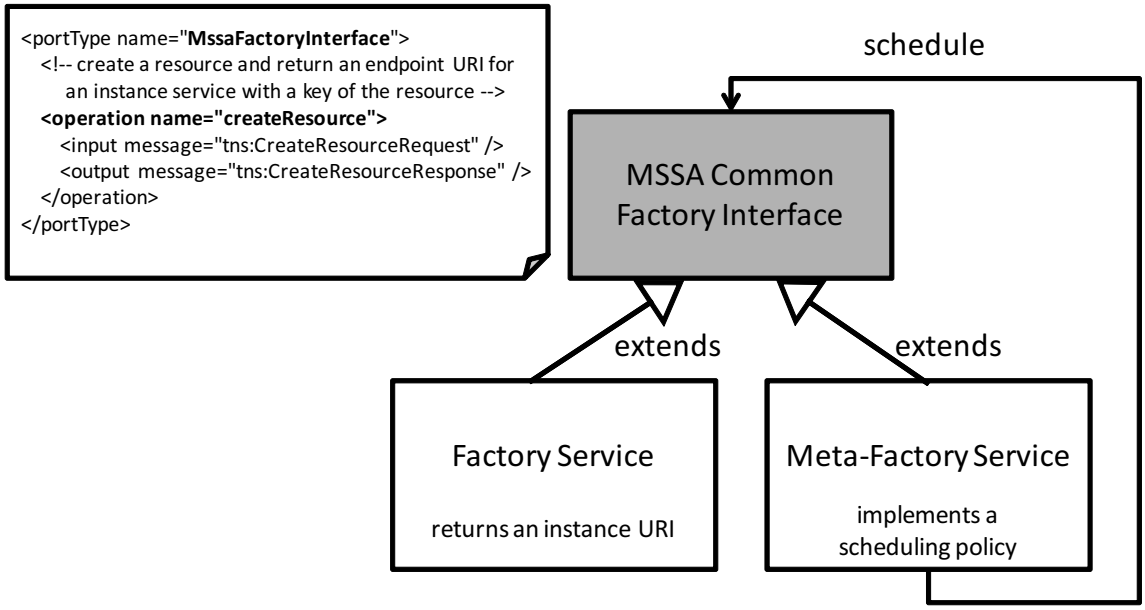

Figure 4.4: Design of MSSAFactoryInterface

interface transparency.

Figure 4.4 shows the design of the common factory interface, MSSAFactoryInterface, in WSDL. Both the actual factory service of the WSRF-based service and the meta-factory service implement the common factory interface. The MSSAFatoryInterface is provided so that the end user can use the both services transparently. The idea behind this design is that the meta-factory service as a meta-scheduler and the actual factory service of the WSRF-based service use the common interfaces through the inheritance mechanism. If a meta-factory service and an actual factory service are developed to have the interfaces inherited from MSSAFactoryInterface, the client program cannot distinguish these two services. In other words, it makes no difference to the end user whether an EPR of the instance service is obtained via the meta-factory service or via the actual factory service. In this way, transparency of meta-factory service interface is achieved.

In practice, in order to inherit from MSSAFactoryInterface, both the actual factory service and the meta-factory service have to import the operations from the definition of MSSAFactoryInterface described in WSDL. In this study, the flatten process mentioned in section 3.4.2 is leveraged again for this purpose. As mentioned in section 3.4.2, the flatten process is used for importing operations of an operation provider into a WSRF-based service. Here, it is used to import the operations of the MSSAFactoryInterface.

For designing the MssaFactoryInterface, the following two points also has been taken into account: hierarchical deployment of meta-scheduler and flexibility of scheduling pol-

icy. In the following, how the MssaFactoryInterface has been designed is explained through the discussion of these two points.

1. Hierarchical Deployment of Meta-scheduler

The MSSAFactoryInterface has been designed so that multiple meta-factory services form the hierarchical structure for achieving the scalability of the meta-scheduler deployment. It is not realistic that only a single meta-scheduler is in charge of all WSRF-based services, since a single point of failure at the meta-scheduler leads to the whole Grid application failure. Moreover, since the deployment configuration of all WSRF-based services flocks to the single meta-scheduler, every change of WSRF-based services deployment within an organization requires changing the configuration of the single meta-scheduler, and this fact prevents scalable expansion of the WSRF-based service deployment. From this consideration, multiple meta-schedulers should be deployed so that they can cover the WSRF-based services deployed over multiple organizations.

Taking the physical setup and deployment of the computing resources, the author has come up with the conclusion that the hierarchical structure of meta-factory services is suitable. It is because that most cluster systems are organized and managed with the hierarchical structure on the Grid environment. Moreover, a cluster system itself is managed through local scheduling systems such as SGE [57] and PBS [58] in a hierarchical manner.

For this reason, the MSSAFactoryInterface has been designed so that multiple meta-factory services form the hierarchical structure for achieving the scalability of the meta-scheduler deployment. Again, Fig. 4.4 illustrates how the MSSAFactoryInterface takes the hierarchical structure of meta-schedulers. The important point is that the MSSAFactoryInterface was designed so as to permit its recursive call. By the recursive call, the actual factory services of the WSRF-based services and the meta-factory services which inherit MSSAFactoryInterface can take a hierarchical structure.

Figure 4.5 diagrams how meta-schedulers are practically deployed over multiple organizations. In this example, there are three organizations: A, B, and C. Each organization has some factory services of WSRF-based services on the computing resources. Organization A has two factory services and a meta-factory service which controls the two factory services. To schedule the WSRF-based services deployed on organizations A and B, there is an upper meta-factory service AB for collaborative scheduling between organization A and B. Between organization B and C, there is also a research community and an upper meta-factory service BC. Through the use of the MSSAFactoryInfterface, this kind of hierarchical structure can be realized.

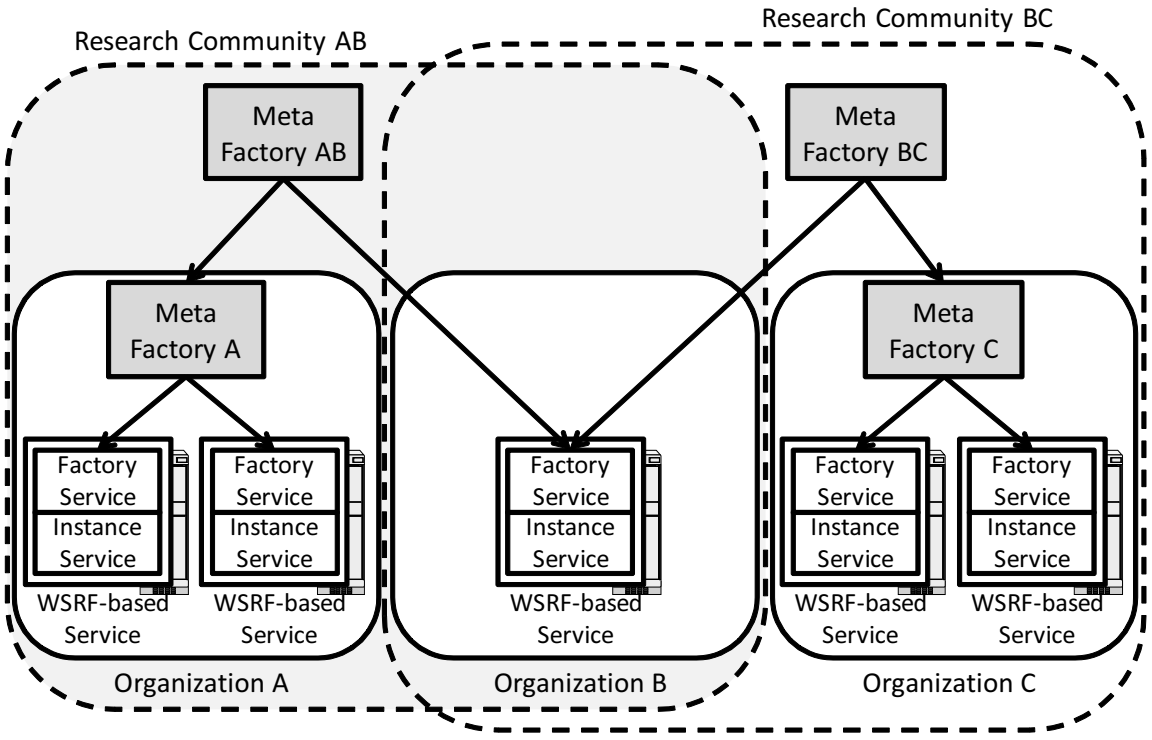

Figure 4.5: A realistic example of hierarchical meta-scheduling environment

2. Flexibility of Scheduling Policy

As described in section 4.1, a meta-scheduler is deployed for performance, throughput and fault-tolerance. Which factors are most important depends on application type, Grid administration perspective, and so on. For example, the applications requiring high-throughput processing, such as drug-docking simulations, need many computing resources simultaneously. On the other hand, the applications requiring high-performance, such as QM/MM hybrid simulation, need a few high-performance computing resources with heavy communications. To satisfy various requirements to the scheduling policy, the meta-scheduler should be able to flexibly accommodate various scheduling policies depending on application type, Grid-administration perspective, and so on.

As described in section 4.1, the end user wants to select a single or a set of WSRF-based services in terms of performance, load-balancing, fault-tolerance, and for other reasons. This means that the meta-factory service as a meta-scheduler can satisfy these requirements for the selection of WSRF-based services. Inherently, the Grid is composed of computing resources of multiple research organizations which have its own administration policies regarding the computing resources. Based on this consideration, the author therefore believes that the best way to achieve flexibility for scheduling policies is the provision

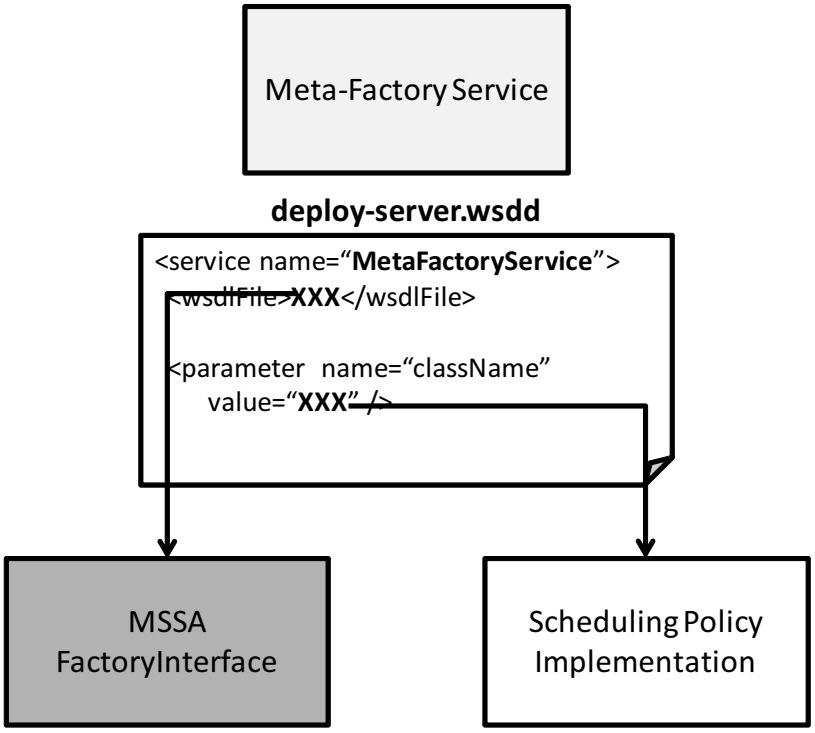

Figure 4.6: Mechanism for configuration of scheduling policy

of a mechanism that allows the administrator to manage scheduling policies in a plug-in manner.

In MSSA, a meta-factory allows the administrator to manage scheduling policies in a plug-in manner. Usually, a Web service is composed of an actual service implementation and its interfaces. As shown in Fig. 4.6, the interface definition is bound to the implementation in a deploy-server.wsdd configuration file. Taking advantage of this feature of Web service deployment, the administrator can flexibly switch his/her scheduling policy just by modifying the WSDD configuration file if he/she has the corresponding implementation of a scheduling policy.

Examples of such scheduling policies include round-robin and high-throughput scheduling. The author has developed a simple round-robin scheduling policy, *SimpleRoundRobinScheduler*, as a prototype. This scheduling policy just dispatches a request from the client program to one of available factory services one by one. Figure 4.7 shows an example configuration of a meta-factory service utilizing this round-robin scheduling policy. In the example, *mssa_factory_service.wsdl*, which defines the common factory interface, is specified in the element of *wsdlFile*, and the implementation of *SimpleRoundRobinScheduler* is specified in the parameter *className*.

```
<service name="sample/SampleMetaFactoryService“
   provider="Handler" use="literal" style="document">
     <wsdlFile>
        share/schema/mssa/mssa_factory_service.wsdl
     </wsdlFile>
     <parameter name="className“
        value="jp.biogrid.mssa.impl.SimpleRoundRobinScheduler"/>
     .
     .
</service>
```

Figure 4.7: An example of meta-factory service utilizing SimpleRoundRobinScheduler

4.3.4 Design and Implementation of Information Provider

This section describes the design and the implementation of the information provider. In MSSA, the information provider has the role of collecting the information on the computing resource (e.g., CPU load, job queue length) necessary for meta-scheduling at the meta-factory service. In order to have this information provider smoothly built into MSSA, the author has developed this information provider as an operation provider so that it can be easily imported to WSRF-based services for scheduling. This mechanism achieves the interface transparency to the WSRF-based services.

As described in Chapter 3, the operation provider technique allows a WSRF-based service to import a set of functions from an operation provider in a plug-in manner. To import the functions of an operation provider into a WSRF-based service, the WSRF-based service had to complete the following configurations: 1) specifying the operation provider name in the “extends” attribute in a WSDL file and 2) specifying the implementation name of the operation provider in the “providers” parameter in a WSDD file. For this advantage of the operation provider technique, it is not required to modify the WSRF-based service implementations, and the author has developed the information provider as an operation provider so that it can be built into the factory service of the WSRF-based service. By utilizing the information provider as an operation provider, the WSRF-based service as a scheduling target can import the functions of information provider needed to expose the information on their computing resources without writing any additional codes.

Figure 4.8 shows how the information provider works. Each information provider notifies the availability of the computing resource of the meta-factory service. Next, the

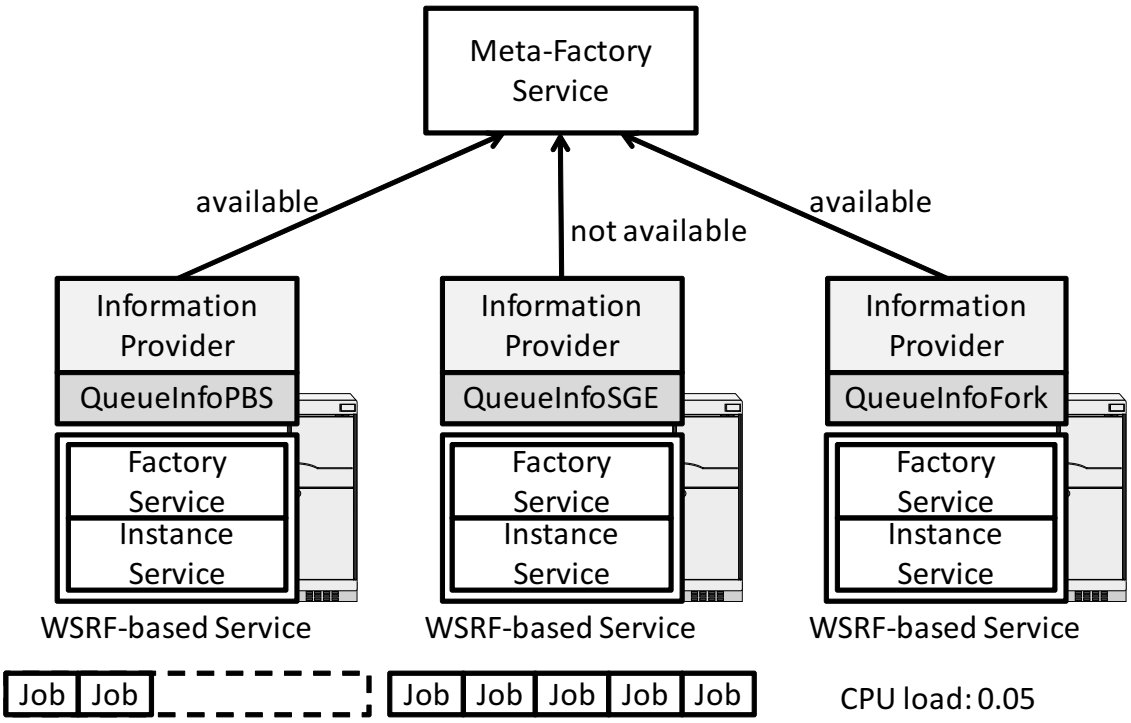

Figure 4.8: Concept of information provider

meta-factory service determines which computing resource is appropriate to be used. The availability of the computing resource can be decided based on information such as job queue length and CPU load. The way to decide the availability of computing resource also depends on what kind of local scheduler (e.g., SGE, PBS) is installed on the computing resource.

However, there is no standard implementation and way for aggregating the information such as job queue length and CPU load from different local schedulers at the moment, although a standard API (DRMAA: Distributed Resource Management Application API) for the submission and control of jobs to various local schedulers is being explored recently [59]. Considering this situation, to make the information provider usable at the realistic environment, the author has developed three types of information collector mechanisms aggregating the information on the computing resource: *QueueInfoSGE*, *QueueInfoPBS* and *QueueInfoFork*. SGE and PBS are the most widely deployed as local schedulers. QueueInfoSGE and QueueInfoPBS are for SGE and PBS, respectively. QueueInfoFork is the information collector mechanism for fork process generation. The author has designed this information provider so that it automatically selects an appropriate mechanism among these three mechanisms depending on the local scheduler system. Specifically, when the Globus server container starts on the cluster where WSRF-based services are deployed, the

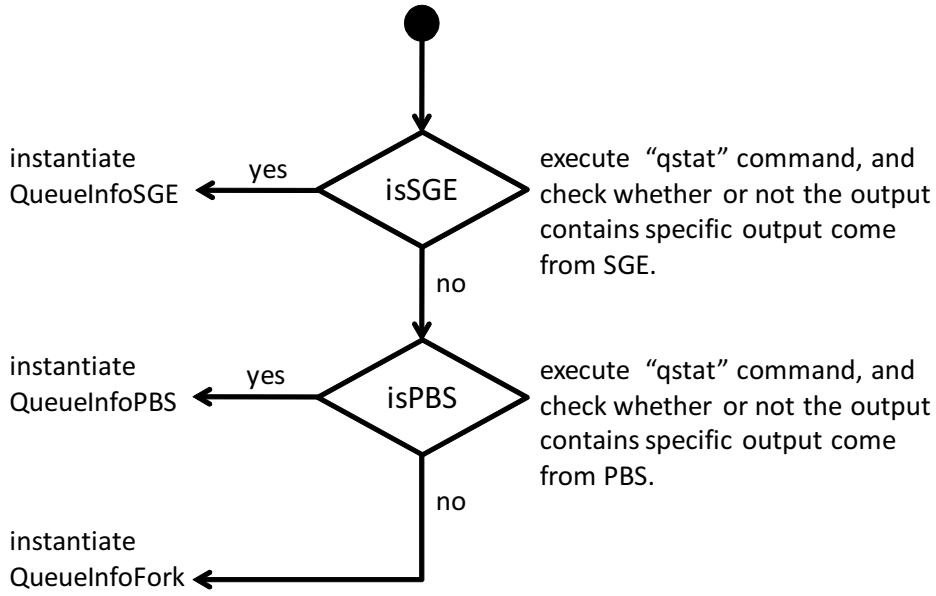

Figure 4.9: Detection process of QueueInfoXXX

information provider checks the type of local scheduler, and then chooses an appropriate mechanism from the above three mechanism. Figure 4.9 shows the detection process of local scheduler. Each implementation of information collector mechanism is detailed in the following.

QueueInfoSGE and QueueInfoPBS

SGE and PBS are very similar local schedulers. QueueInfoSGE and QueueInfoPBS, therefore, share a common design. When QueueInfoSGE and QueueInfoPBS receive a request from a meta-factory service about the availability of computing resources where a SGE and a PBS are in charge, QueueInfoSGE and QueueInfoPBS works as follows. They first check whether the end user's jobs are already running on the computing resource. If there are the end user's jobs, they return the message showing "not available". Next, they checks job queue length. If there is no free CPU to launch a job on, they return the message showing "not available". If there is any free CPU, they return "available". At the first step, both QueueInfoSGE and QueueInfo execute *qstat* command to check the existence of the end user's jobs. At the second step, QueueInfoSGE executes qstat command, and QueueInfoPBS executes *pbsnodes* command to check the number of available CPUs.

Table 4.1: Configuration files for a meta-factory service

Name	lines
deploy-server.wsdd	40
deploy-jndi-config.xml	4
namespace2package.mappings	7
build.properties	8

```
interface.name=Dock
package=jp.biogrid.services.dock
prefix.publish.path=example/dock
```

Figure 4.10: An example configuration for MSSA toolkit

QueueInfoFork

The information collector mechanism of QueueInfoFork decides the availability of the computing resources only from the CPU load, because there is no local scheduler. When QueueInfoFork receives a request from a meta-factory service about the availability of computing resources, QueueInfoFork executes *uptime* command to check CPU load, and returns "available" if the CPU load is less than 0.50.

4.3.5 Design and Implementation of MSSA Toolkit

To develop a meta-factory service, configuration files shown in Table 4.1 are required. These configuration files bind the common interface of the meta-factory service, MSSAFactoryInterface, and the implementation of scheduling policy. Writing these configuration files by hand is also time-consuming and error-prone work.

The MSSA toolkit is a toolkit that facilitates the above work in developing a meta-factory service. This MSSA toolkit is designed and implemented to reutilize the Opal OP toolkit design treated in Chapter 3, and it is integrated with the Opal OP toolkit. Whereas Opal OP toolkit generates a set of configuration and implementation files for a WSRF-based service named like XXXService, MSSA toolkit generates a set of configuration files for a meta-factory service named like MetaXXXFactoryService. The MSSA toolkit requires the configuration as shown in Fig 4.10. The MSSA toolkit picks parameters from this configuration file, and generates configuration files in the same mechanism of Opal OP

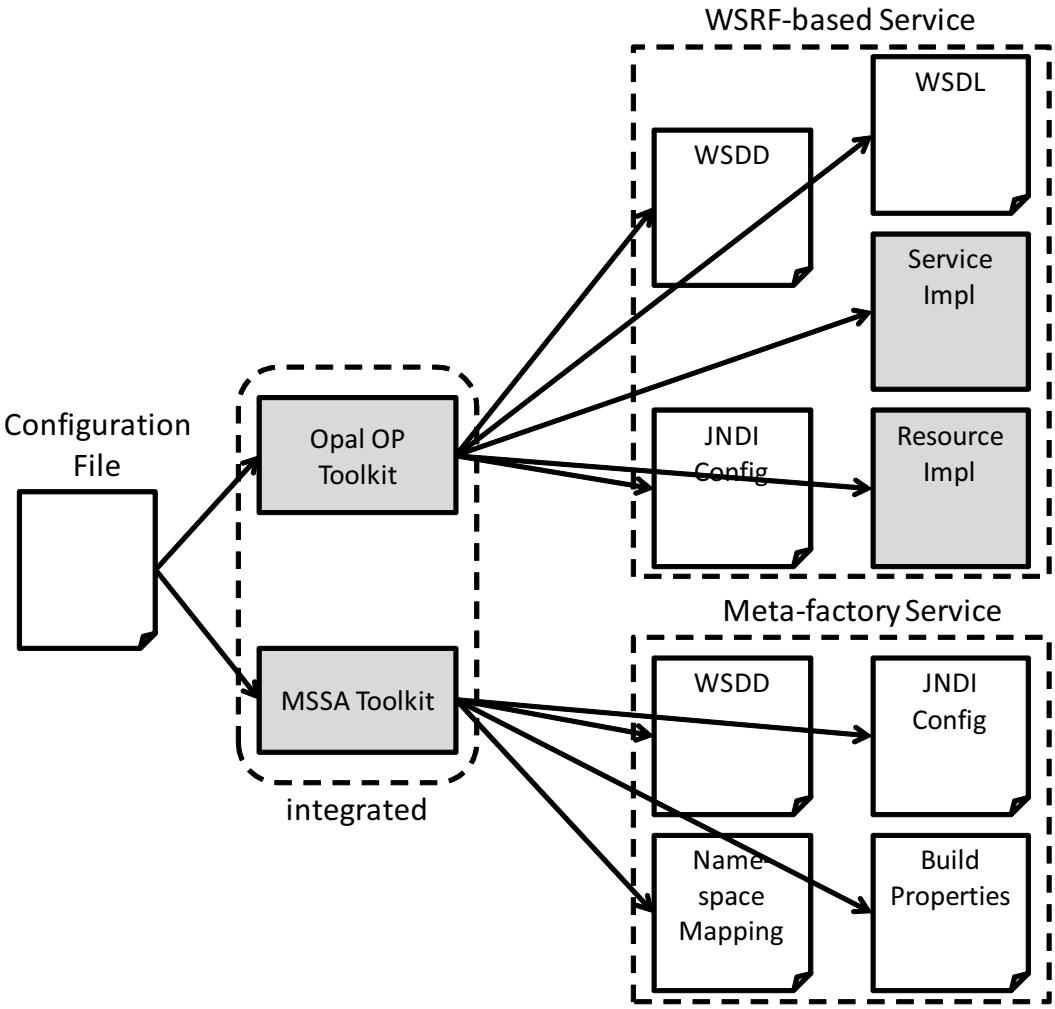

Figure 4.11: MSSA toolkit and Opal OP toolkit

```
$ ant -f build-opal.xml -propertyfile dock.properties new-service
$ cd services/Dock
$ ant deploy
$ cd ../..
$ ant -f build-opal.xml -propertyfile dock.properties
                                        -Dfactories=”site1 site2. . .” new-metaservice
$ cd services/DockMetaFactory
$ ant deploy
```

Figure 4.12: An example command-lines for developing a WSRF-based service and a meta-factory service

toolkit. This configuration is subset of the configuration required by Opal OP toolkit. The application developer, therefore, can reutilize the configuration file used in the developing his/her WSRF-based service with Opal OP.

As shown in Fig. 4.11, by the integration of the MSSA toolkit and Opal OP toolkit, the application developer can build his/her WSRF-based service and then develop a meta-factory service for the WSRF-based service with the command-line execution shown in Fig. 4.12. The upper 3-line command sequence goes to Opal OP toolkit for the development of a WSRF-based service. The bottom 3-line command sequence goes to MSSA toolkit for the development of a meta-factory service for the built WSRF-based service.

In the current implementation of MSSA toolkit, SimpleRoundRobinShceduler, which the author developed as a prototype scheduler, is selected as a default scheduling policy. By using MSSA, the procedure of the developing stage of a WSRF-based service from existing program discussed in Chapter 3 and the developing stage of a Grid application composed of multiple WSRF-based services are performed in an integrated manner.

4.4 Evaluation and Discussion

This section evaluates and discusses the proposed MSSA. For this purpose, a prototype system for drug-docking simulation deployed based on the MSSA over multiple cluster systems is discussed as an actual development example. Through the example, the author verifies that a scalable Grid application composed of 175 CPUs can be developed. After that, the more detailed behavior of the actual Grid application based on the proposed MSSA is analyzed.

4.4.1 Prototype System for Drug-docking Simulation

This section discusses a drug-docking simulation system which was developed and deployed over multiple cluster systems. Table 4.2 shows the cluster systems used for this prototype system.

Development Procedure

In building this prototype system, the Opal OP toolkit and MSSA toolkit were used to reduce the building works of the WSRF-based services for the drug-docking simulation program and the meta-factory services as meta-schedulers. The *Dock services* shown in Fig. 4.13 are WSRF-based services which encapsulate the drug-docking simulation program. These Dock services were built up with the Opal OP toolkit and deployed into all

Table 4.2: Cluster systems used in prototype system for drug docking simulation

Cluster name	CPU architecture	Number of worker CPUs	Local scheduler
cafe	Intel Xeon 2.8GHz	38	SGE
tea	Intel Pentium III 1.4GHz	80	SGE
sibbs	Intel Pentium III 1.4GHz	9	SGE
tdws	AMD Opteron Processor 252 2.6GHz	22	N/A
rocks-52	Intel Xeon 2.4GHz	26	SGE

cluster systems shown in Fig. 4.13. On the other hand, the *MetaDockFacotry services* are meta-factory services built up with the MSSA toolkit. In this prototype system, the author deployed MetaDockFactory A, MetaDockFactory B, MetaDockFactory Root on the *cafe*, *sibbs* and *tea* cluster system, respectively. These three cluster systems, therefore, run not only as Dock services, but also MetaDockFactory services as meta-schedulers.

The development of the prototype system was realized as the following three steps. First, the author wrote a configuration file for Opal OP toolkit described in Chapter 3. The configuration file was used for building a WSRF-based service encapsulating the drug-docking simulation program and also used for building a meta-factory service. Figure 4.14 shows the configuration file. Second, to generate a set of template files of the WSRF-based service and the meta-factory service, the author executed the command-line sequentially as shown in Fig. 4.12. By the command-line sequence, the WSRF-based service and the meta-factory service were generated, built and deployed into the Globus server container installed on where the machine on the author's desk. At the last, the author succeeded to construct the prototype system as shown in Fig. 4.13 by copying the generated services into each of cluster systems, and starting up the Globus server container. Importantly, all the author had to do was the only three steps: 1) writing a single configuration file, 2) executing command-line tools and 3) deploying the generated services into cluster systems.

With the interface transparency provided by the MSSA, the MetaDockFacotry services have the same interfaces with original Dock services. For this reason, the command-line tools, which are available to the WSRF-based service which is composed of a factory (e.g. Dock Factory) and the corresponding instance services (e.g. Dock Service), such as opalop-jobrun and opalop-jobquery are also available to the meta-factory service (e.g. MetaDockFactory Service). Thus, the end user can easily access multiple Dock services through meta-factory service Root by utilizing the command-line tools provided by the Opal OP toolkit.

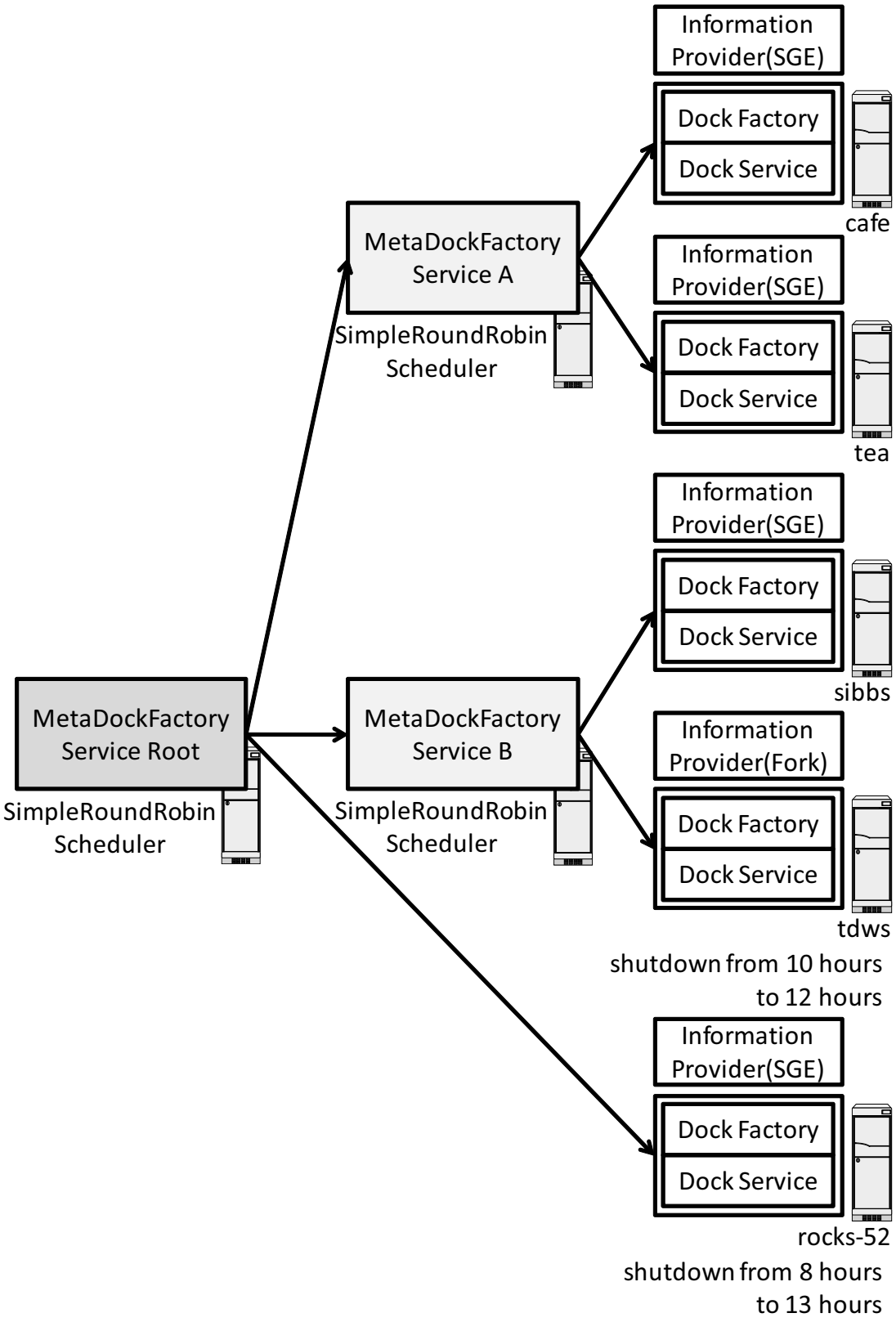

Figure 4.13: The environment for the prototype system

```
interface.name=Dock
binary.location=/opt/dock6/bin/dock6mpi
target.namespace=http://biogrid.jp/namespaces/DockService
package=jp.biogrid.services.dock
stubs.package=jp.biogrid.stubs.dock
prefix.publish.path=example/dock
factory.target.namespace=http://biogrid.jp/namespaces/DockFactoryService
```

Figure 4.14: Properties file used to develop the prototype system

Table 4.3: Comparison of the codes between the client program introduced in section 3.5.2 and this section

Client program	Language	Lines
Client program introduced in section 3.5.2	Perl script	143
Client program introduced in this section	Shell script	26

In the case of developing this kind of Grid application from scratch, the application developer is required to have a deep knowledge to handle multiple WSRF-based services. Also, the application developer has to do many configuration and implementation works. For example, the drug-docking simulation introduced in section 3.5.2 has been developed without the proposed MSSA. In this case, the end user had to have the client program code choosing and accessing appropriate services among multiple WSRF-based services by hand. Table 4.3 shows the comparison of the client program codes necessary for the scheduling of and the access to the WSRF-based services for drug-docking simulation with and without MSSA. According to the comparison, 143 lines of perl script were necessary when the application developer had to develop the client program in the case of the development without MSSA. On the other hand, when the application developer develops the client program in the case of the development with MSSA, only 26 lines of shell script was required. Although the language used in each case was different, the development with MSSA is considered to reduce the implementation work of the client program.

Execution Profile

Figure 4.15 shows the number of the CPUs used during an experiment screening 100,000 drug candidates on this prototype system. In this experiment, each job performs the screening of 500 drug candidates. In short, 200 jobs in total were submitted. To investigate the

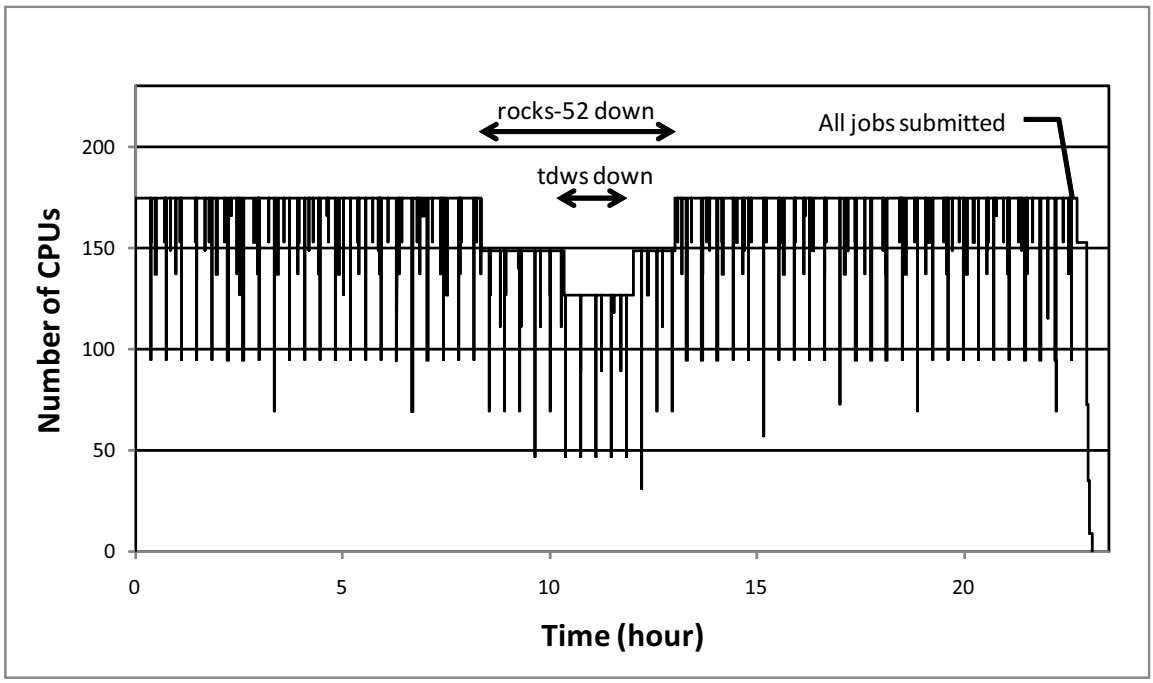

Figure 4.15: Number of CPUs used during docking experience

strength of fault-tolerance to the sudden change of the computational environment, two cluster systems of *rocks-52* and *tdws* were separately shut down on purpose during the experiment. The *rocks-52* cluster system was shut down for 5 hours from 8 to 13 hours, and the *tdws* cluster system was shut down for 2 hours from 10 to 12 hours since the start of the experiment.

According to the result shown in Fig. 4.15, all 175 CPUs were used for computation except during the *rocks-52* and *tdws* cluster system's shutdown. Importantly, this result indicates that the meta-factory services built into the prototype docking simulation system were tolerant to the sudden computational environmental change. While *rocks-52*, *tdws*, or both were down, MetaFactroyDock services recognized the situation, and succeeded not to dispatch any jobs to these two cluster systems. Also, when these cluster systems came up again, the MetaFactroyDock service succeeded in restarting the dispatching jobs to these two cluster systems again. This means that the meta-factory service as a meta-scheduler built into the prototype system can perform the meta-scheduling of the underlying WSRF-based services without the end user's awareness of the failure of *rocks-52* and *tdws* cluster systems.

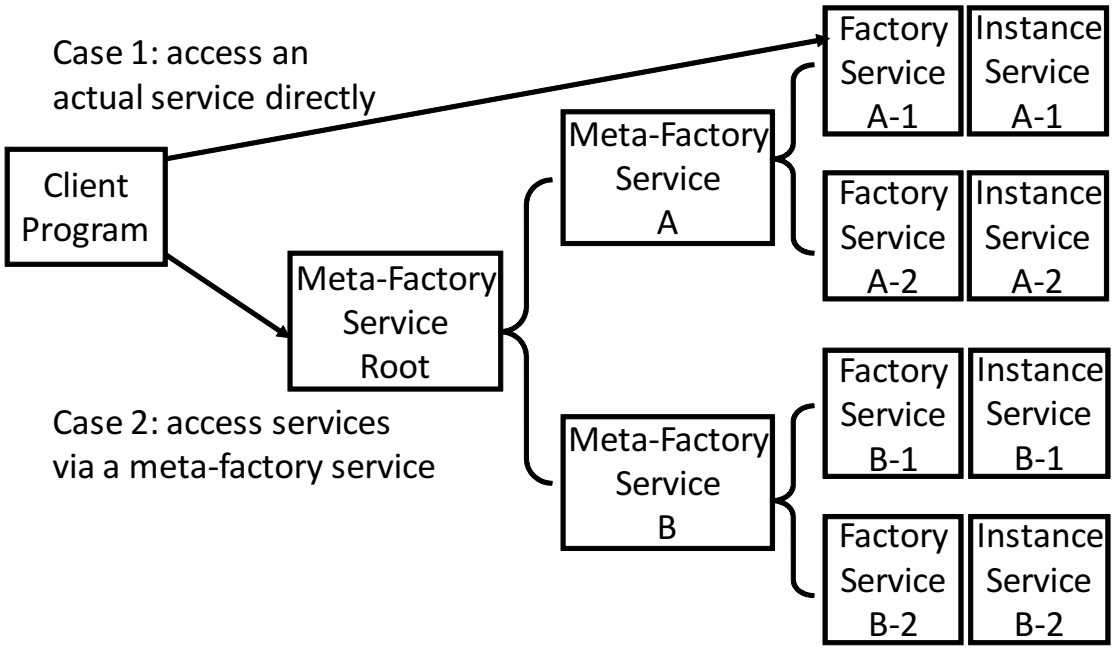

Figure 4.16: Meta-scheduling environment for measurement

4.4.2 Detailed Behavior of MSSA-based System

The previous subsection shows that the prototype system which adopts the proposed MSSA is tolerant to the change in the computational environment and can perform the meta-scheduling of WSRF-based services according to the computational environment. In addition, the Opal OP toolkit and MSSA toolkit were practically used to facilitate the author's development of the prototype system. In this section, the author analyzes the characteristics of the general system based on the proposed MSSA through a simple experiment.

To analyze the detailed behavior of the MSSA, the author developed a meta-scheduling environment shown in Fig. 4.16 and measured the overhead of scheduling process and the behavior under highly-loaded situation. This environment was constructed within a local area network connected with a 1Gbps connection, and used four machines with dual Intel Xeon 2.8GHz processors. There are four actual WSRF-based services in the environment: A-1, A-2, B-1 and B-2. All of these WSRF-based services do not perform anything. These WSR-based services are prepared for just reviewing the behavior of the MSSA. In this environment, services of A-x and B-x are duplicated for the purpose of load balancing and fault tolerance. The WSRF-based services are scheduled in two layers of meta-factory services. Meta-factory service A governs the factory services A-1 and A-2, and the meta-factory service B governs B-1 and B-2. Further, meta-factory service Root governs meta-factory

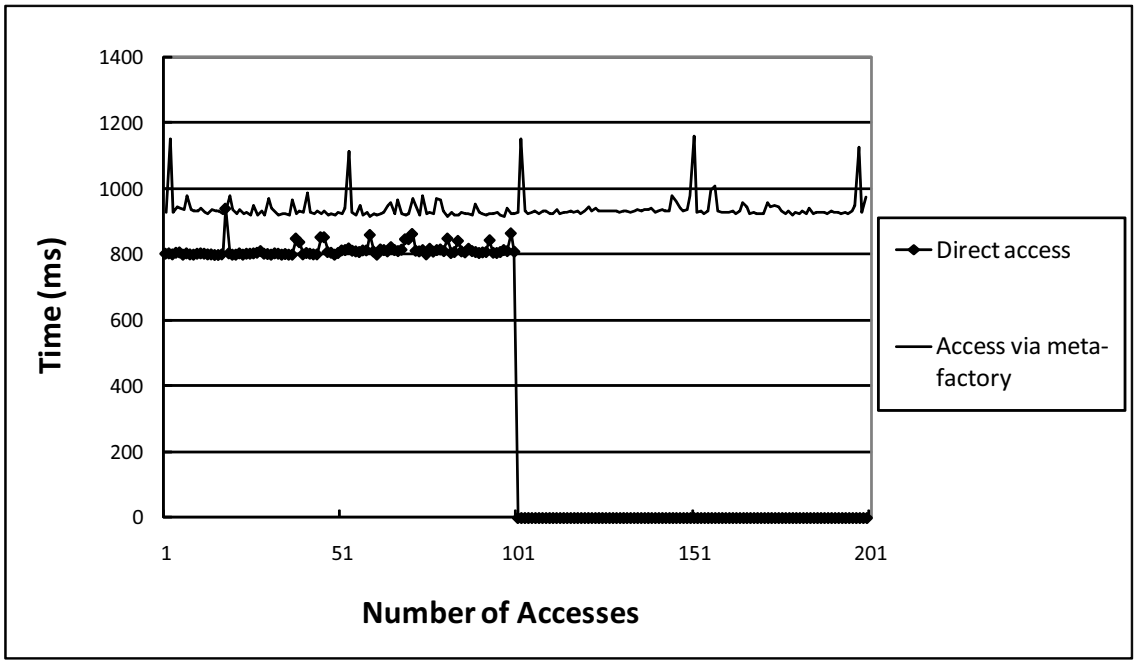

Figure 4.17: Time to get URIs of instance services

services A and B. All meta-factory services of Root, A and B implement the SimpleRoundRobinScheduler.

Evaluation of Overhead

In this evaluation, the time from when a client program sends a request to a factory service until when the client program obtains a URL of an instance service were measured in the following two cases to review the overhead of scheduling process. First, the time was measured in the case where the client program accesses factory service A-1 directly without using any meta-factory service as a meta-scheduler. Second, the time was measured in the case where the top of the meta-factory service (Meta-Factory service Root) was accessed from the client program. The measurement was performed 200 times for each case. After the 100th access in either case, A-1 and B-1 services were shutdown on purpose to see how the system behaves and how the overhead changes.

Figure 4.17 shows the measurement result. According to the measurement result, the direct access case is better than the access via meta-factory service. Simultaneously, the result indicates that the overhead of using the meta-factory service is approximately 120 msec. In case of the access via meta-factory service, the request from the client program must be handled by two meta-factory services. Taking this fact into consideration, the overhead of a single meta-factory service is around 60 msec.

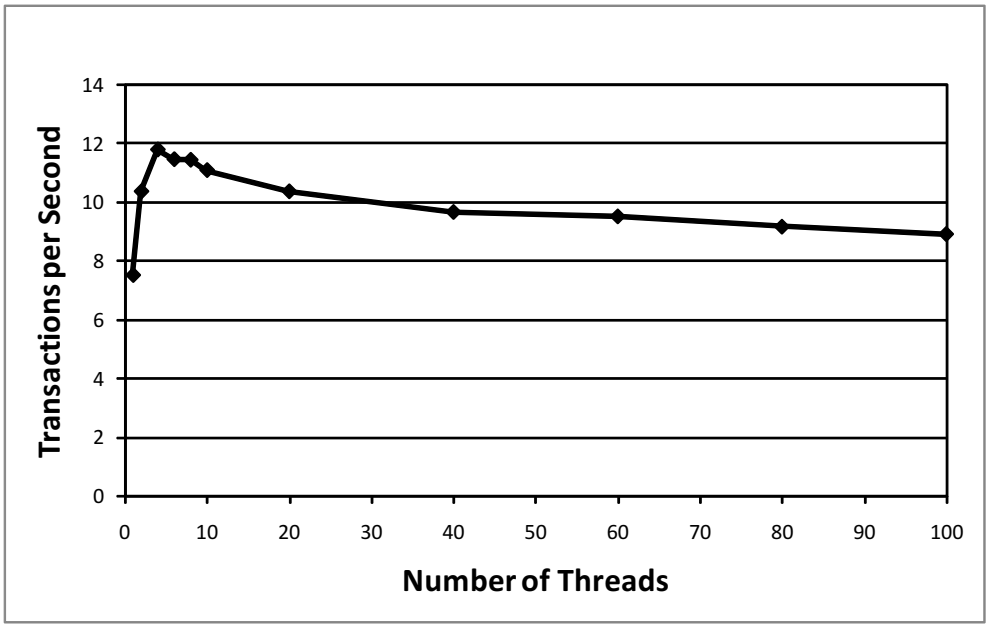

Figure 4.18: Number of SOAP transactions executed per second

Also, the measurement result indicates that the system developed based on the proposed MSSA continued running in spite of the sudden death of A-1 and B-1 without affecting the time for obtaining the URL from the meta-factory services whereas the client program stopped in the first case. Importantly, the client program did not know the change of the computational environment, meaning that the system based on the proposed MSSA could perform meta-scheduling according to the computational environment in a transparent manner. The client program used in this measurement was exactly the same in both cases.

Evaluation of Behavior Under Highly-loaded Situation

In this evaluation, to measure the performance of meta-factory service under the highly-loaded situation, a benchmark tool for Web service, WSTest [60], was utilized. Specifically, it was reviewed how the number of SOAP transactions executed per second was changed while multiple threads were accessing a meta-factory service. In order to allow WSTest to access the top of the meta-factory service (Meta-Factory service Root) in Fig. 4.16, the implementation was modified so that it conformed to the WSRF standard.

Figure 4.18 shows the measurement result. According to the measurement result, accessing with 4 threads takes maximum performance, approximately 12 transactions per second. After that, the performance dropped with the increase of threads accessing the meta-factory service simultaneously. However, even when 100 threads were used, the

meta-factory service could succeed in performing 9 transactions per second. This fact means that the meta-factory service could dispatch 9 requests from clients to different 9 WSRF-based services in a second. Thus, if there are a hundred of WSRF-based services as scheduling targets, the meta-factory service could dispatch the requests from clients to all WSRF-based services in approximately 11 seconds. This performance is considered to be enough in most cases of the usage.

4.5 Concluding Remarks

In this chapter, the author proposed a new scheduling architecture, Meta-Scheduling Services Architecture (MSSA). The MSSA provides a transparent meta-scheduler interface to WSRF-based services developed by application developers. The transparency of the meta-scheduler interface allows the use of client program as-is for accessing the target WSRF-based services via this scheduling architecture. The advantages of the MSSA are: the hierarchical scheduling structure, and the flexibility of scheduling policy. The hierarchical scheduling structure allows the application developer to build a scalable and fault-tolerant Grid application composed of multiple WSRF-based services. In addition, the site administrator can flexibly switch the scheduling policy of the meta-factory service as a meta-scheduler based on the site-administration policy.

To technically realize the proposed MSSA, the author has leveraged the factory pattern utilized in WSRF-based service. Under the factory pattern mechanism, the end user always has to access the factory service composing the target WSRF-based service before accessing the instance service which provides the actual function of the target WSRF-based service. The author has focused on this factory pattern mechanism and realized a technical solution that facilitates the development of a Grid application based on the proposed MSSA. The technical solution provides a common interface named MSSAFactoryInterface, with which the application developer can easily develop the meta-factory service as a meta-scheduler.

Through the review of the development case of a prototype system for drug-docking simulation, this chapter showed that the application developer could build a scalable and fault-tolerant Grid application composed of WSRF-based services through the use of the Opal OP toolkit and the MSSA toolkit. Also, the measurement result shown in this chapter indicates that the system developed based on the proposed MSSA can obtain fault-tolerance with a small overhead of 60ms also take high performance under highly-loaded.

Chapter 5

Conclusion

5.1 Concluding Remarks

Distributed computing technologies have made dramatic improvement in the last decades. The integration of service-oriented architecture to Grid technologies is leading a new mode of wide-area distributed computing in scientific and engineering areas. Despite the recent maturity of Grid computing technologies represented by WSRF technologies, many difficulties still exist in the development procedure from an application program to a service-oriented Grid application composed of multiple WSRF-based services. For this reason, the author has proposed methods and tools to facilitate the development of a service-oriented Grid application in this dissertation.

Chapter 2 clarified the technical issues to achieve in this study. To understand the difficulties in developing the service-oriented Grid application, the author first reviewed the Grid application development procedure. In this review, the developing stage of a WSRF-based service from an existent program and the developing stage of a Grid application utilizing WSRF-based services deployed over multiple organizations were particularly focused on. Next, through the discussion on conventional approaches available today for solving the difficulties at these two stages, the author has come up with the conclusion that the wrapping method and meta-scheduler are possibly available solutions that facilitates the development for these two stages, respectively. In the conventional wrapping method, however, the author pointed out that inflexibility and inextensibility became a serious problem in the process of developing a WSRF-based Grid service from an existent application program. Also, the author revealed that the existent meta-scheduler targeting only GRAM services does not satisfy the application developers' requirements to the meta-scheduling of multiple WSRF-based services composing a service-oriented Grid application. To solve these problems, the author has concluded that two technical issues must be solved. The

establishment of a new flexible and extensible wrapping service and a new meta-scheduler suitable for service-oriented Grid application are the two issues that need to be solved.

In Chapter 3, as a solution to the first issue, the establishment of a new flexible and extensible wrapping service, the author has proposed a new wrapping model, the "Extensible Wrapping Service Model" and its corresponding development method based on the model. Also the corresponding tool named Opal Operation Provider (Opal OP), which allows the application developer to easily build a WSRF-based service from an existing application, was developed. The basic idea of the new model was the provision of a set of functions for encapsulating an existing program as a module so that the application developer can import the functions into his/her WSRF-based service with minimal efforts. For the purpose of modularizing the functions for encapsulation, the operation provider technique was investigated. The author then developed the Opal OP module as an operation provider by combining the operation provider technique and Opal technology. In addition, to support the total development of a WSRF-based service from an existent program with the Opal OP module, the author proposed the Opal OP toolkit. The Opal OP toolkit has leveraged the dependency and redundancy among configuration files and implementation files for a WSRF-based service to generate a set of template codes for a WSRF-based service automatically from only a single small configuration file which the application developer has to prepare. More specifically, the Opal OP toolkit assumes that the generated WSRF-based service encapsulates a command-line program through the Opal OP module and provides a fixed set of interfaces. This assumption restricts the targets of the proposed method to command-line program whereas generally WSRF technologies realize various types of Grid services (e.g., account management service and database management service). However, in terms of developing a WSRF-based service for an existing scientific program, the restriction derived from this assumption is not a problem because most scientific programs are implemented as command-line programs. The usefulness of the proposed method and tool were shown through the review of three actual examples of Grid applications developed with the Opal OP.

In Chapter 4, the second technical issue revealed in Chapter 2, namely, the establishment of a new meta-scheduler suitable for service-oriented Grid application was treated. For this purpose, the author has proposed a new meta-scheduling architecture (MSSA: Meta-Scheduling Services Architecture) and MSSA toolkit for facilitating the development of Grid application utilizing WSRF-based services deployed over multiple computing resources. The MSSA provides a transparent meta-scheduler interface to WSRF-based services developed by application developers. The transparency of the meta-scheduler

interface allows the use of client program as-is for accessing the target WSRF-based services via this scheduling architecture. The MSSA also has the following two advantages: scalable hierarchical scheduling structure, and flexibility of a scheduling policy. In order to realize transparent meta-scheduler interface, the author has leveraged the factory pattern utilized in WSRF-based service, and proposed a mechanism of meta-factory service implementing scheduling process behind this factory pattern. For the implementation of the meta-factory service, the author has defined a common factory interface, MSSAFactoryInterface. With this interface, the application developer can develop the transparent meta-scheduler that has the same interface as the factory service of the WSRF-based service. It also allows the administrator to switch the scheduling policy in a plug-in manner. Evaluation and discussion showed how the development work for a scalable and fault-tolerant Grid application composed of WSRF-based services was facilitated with Opal OP toolkit and MSSA toolkit through the review of a prototype system of drug-docking simulation. Also, the measurement result of the system developed based on the proposed MSSA indicates that such system can select a WSRF-based service from multiple WSRF-based service with small overhead of 60 msec, and perform up to 12 transactions per second.

5.2 Future Directions

In this study, the author proposed Opal OP and MSSA for solutions of technical issues clarified in Chapter 2. The total development procedure of a service-oriented Grid application from an existing program was facilitated with the proposed Opal OP and MSSA. Opal OP has been already utilized in several research communities, and highly evaluated as a flexible method for facilitating the development of a WSRF-based service from an existing program. However, MSSA has not been tested and evaluated in realistic developments. Therefore, the author would like to actively promote the use of MSSA through the research communities which has already introduced Opal OP. The author hopes that the achievement of the dissertation contributes to the enhancement and advancement of computational science with service-oriented Grid. Four directions toward the promotion of this study are considered below.

1. Supporting Application Specific Implementation for a WSRF-based Service Utilizing Opal OP

In order to further facilitate the development of a WSRF-based service, the proposed Opal OP should support the preparation of application specific implementation for a WSRF-

based service realized by Opal OP. The current Opal OP allows the application developer to extend the WSRF-based service generated by Opal OP by adding his/her application specific implementation, whereas the conventional wrapping methods provide the fixed implementation of a wrapping service. However, the Opal OP does not provide any way to develop application specific implementation itself because the application specific implementation such as intermediate data exchange and synchronization between multiple Grid services is dependent on the individual implementation of the program. For the purpose of facilitating the development of a WSRF-based service, a way to facilitate the development of the application specific implementation, such as a common library extracted from various use cases of Opal OP, is required.

2. Supporting Construction of Grid Environment

As reviewed in section 3.5.2, the application developer has difficulties and spends long time to construct a Grid environment for his/her Grid application (i.e., the work mentioned as the third stage of developing a Grid application). For example, he/she needs to install Globus Toolkit and his/her WSRF-based service on each of remote resources. For facilitating the total works of developing a Grid application, this construction work should be taken into account as a future issue.

3. Extension of Resource Property in Opal OP

As described in Chapter 3, the author has implemented a resource property to expose the job status as the resource property defined in the WSRF standard. Currently this resource property is available only from the client program so that it checks the job status including job termination and aborting. For more sophisticated integration and federation among multiple WSRF-based services (e.g., autonomous coordination among interdependent WSRF-based services), the resource property should be utilized and shared by not only the client program but also multiple WSRF-based services composing a Grid application. The WSRF standard prescribes a standard method, WS-Notification, for asynchronous communication between WSRF-based services which is triggered by the change in resource property. For the purpose of realizing more sophisticated integration among WSRF-based services built with Opal OP, the implementation of resource property conforming to WS-Notification is required.

4. Implementation and Verification of Scheduling Policies

In this study, the author has developed a round-robin scheduling policy implementation, SimpleRoundRobinScheduler, as a prototype. However, to further discuss the availability and usefulness of the proposed meta-scheduling architecture from practical aspects, the author considers that more complicated scheduling policies, such as performance-intensive policy and data-intensive policy which requires more complicated interaction among meta-factory services as meta-schedulers, must be implemented and verified to be practically deployed on the proposed MSSA.

Acknowledgments

I would like to thank Professor Shinji Shimojo of the Cybermedia Center at Osaka University for his overall and primary supervision, countless suggestions, and constructive comments on my research activities and on writing this dissertation.

I am heartily grateful to Professors Shojiro Nishio, Toru Fujiwara, Fumio Kishino, Norihisa Komoda, Kiyoshi Kogure, and Norihiro Hagita of the Department of Multimedia Engineering in the Graduate School of Information Science and Technology at Osaka University for their support and numerous suggestions for revising this dissertation.

I would also like to express my gratitude to Specially Appointed Associate Professor Susumu Date of the Department of Bioinformatic Engineering in the Graduate School of Information Science and Technology at Osaka University for his advice to the dissertation. His expertise and insightful comments have been most beneficial.

My sincere appreciation also goes to the faculty members of Shimojo Laboratory; Associate Professor Ken-ichi Baba, Assistant Professor Toyokazu Akiyama, Specially Appointed Instructors Eisaku Sakane and Shingo Okamura, Educational Affairs Office Employee Kazunori Nozaki, and Specially Appointed Researcher Yoshimasa Ishi for their sincere advice and support. Also, Dr. Kaname Harumoto of the Graduate School of Engineering at Osaka University, Dr. Yuuichi Teranishi of the Department of Multimedia Engineering, Dr. Seiichi Kato of Hyogo University of Health Sciences, and Dr. Susumu Takeuchi of the Department of Multimedia Engineering gave me invaluable encouragement and constructive comments on my research. I would also like to thank the colleagues of the Shimojo Laboratory.

I am also heartily grateful to Dr. Sriram Krishnan, Dr. Wilfred Li, Mr. Marshall Levesque, Ms. Ellen Tsai, Dr. Peter Arzberger of the University of California San Diego, Professor Haruki Nakamura, Dr. Yasushige Yonezawa, Dr. Reiko Yamashita of The Institute for Protein Research at Osaka University and Dr. Kazuto Nakata of NEC Soft, Ltd. for technical supports. I would also like to express my gratitude to Dr. Bu-Sung (Francis) Lee of the School of Computer Engineering at Nanyang Technological University in Singapore for his careful proofreading of this dissertation and invaluable advice.

Finally I would like to show my deepest gratitude to my family members, to my mother Hiromi and father Shunichi, who raised me with courteous parental ship and emotional support; to my older sister Keiko, and to my younger brother Michihiro.

Bibliography

[1] R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson, "The interaction of parallel and sequential workloads on a network of workstations," in *Proceedings of ACM SIGMETRICS'95/PERFORMANCE'95 Joint International Conference on Measurement and Modeling of Computer Systems*, pp. 267–278, May 1995.

[2] M. K. Gobbert, "Configuration and performance of a Beowulf cluster for large-scale scientific simulations," *Computing in Science and Engineering*, vol. 7, no. 2, pp. 14–26, March 2005.

[3] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer, "BEOWULF: A parallel workstation for scientific computation," in *Proceedings of the 24th International Conference on Parallel Processing*, vol. 1, pp. 11–14, August 1995.

[4] T. Sterling, D. J. Becker, J. Salmon, and D. F. Savarese, *How to Build a Beowulf*. MIT Press, May 1999.

[5] T. L. Sterling, *Beowulf Cluster Computing With Linux*. MIT Press, October 2001.

[6] R. R. Scheller, "Moore's law: Past, present and future," *IEEE Spectrum*, vol. 34, no. 6, pp. 52–59, June 1997.

[7] G. Gilder, *TELECOSM: How Infinite Bandwidth will Revolutionize Our World*. Free Press, September 2000.

[8] J. E. White, "A high-level framework for network-based resource sharing." `http://tools.ietf.org/rfc/rfc707.txt`.

[9] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable parallel programming with the message-passing interface*. MIT Press, October 1994.

[10] S. Vinoski, "CORBA: Integrating diverse applications within distributed heterogeneous environments," *Communications Magazine, IEEE*, vol. 35, no. 2, pp. 46–55, February 1997.

[11] R. Marvie and P. Merle, "CORBA component model: Discussion and use with OpenCCM," in *Technical Report, LIFL*, June 2001.

[12] M. Horstmann and M. Kirtland, "DCOM architecture." `http://msdn2.microsoft.com/en-us/library/ms809311.aspx`.

[13] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, "The physiology of the Grid: An open Grid services architecture for distributed systems integration." `http://www.globus.org/research/papers/ogsa.pdf`, June 2002.

[14] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *High Performance Computing Applications*, vol. 11, no. 2, pp. 115–128, Summer 1997.

[15] "The Globus Alliance." `http://www.globus.org/`.

[16] I. Foster, "Globus Toolkit version 4: Software for service-oriented systems," in *Proceedings of the IFIP International Conference on Network and Parallel Computing, NPC 2005*, pp. 2–13, November 2005.

[17] K. Czajkowski, D. F. 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.

[18] *Political Multi-Agent Simulation with Grid Computing*, vol. 17 of *RCSS Discussion Paper Series*, Kansai University, August 2004.

[19] H. Stockinger, F. Donno, E. Laure, S. Muzaffar, P. Kunszt, G. Andronico, and P. Millar, "Grid data management in action: Experience in running and supporting data management services in the EU DataGrid project," in *Proceedings of Computing in High Energy Physics (CHEP 2003)*, March 2003.

[20] "The Particle Pyhsics Data Grid (PPDG) home page." `http://www.ppdg.net`.

[21] I. Zaslavsky and A. Memon, "GEON: Assembling maps on demand from heterogeneous Grid sources," in *Proceedings of the 2004 ESRI User Conference*, August 2004.

[22] "GEON: Geoscience network." http://www.geogrid.org.

[23] "BIRN: Biomedical informatics research network." http://www.nbirn.net.

[24] A. W. Lina, L. Dai, K. Ung, S. Peltier, and M. H. Ellisman, "The telescience project: Applications to middleware interaction components," in *Proceedings of the 18th IEEE International Symposium on Computer-Based Medical Systems*, pp. 543–548, June 2005.

[25] S. Date, Y. Mizuno-Matumoto, Y. Kadobayashi, and S. Shimojo, "An MEG data analysis system using Grid technology," *Transactions of Information Processing Society of Japan*, vol. 42, no. 12, pp. 2952–2962, December 2001.

[26] K. Ichikawa, S. Date, T. Kaishima, and S. Shimojo, "A framework supporting the development of Grid portal for analysis based on ROI," *Methods of Information in Medicine*, vol. 44, no. 2, pp. 265–269, June 2005.

[27] J. Alameda, M. Christie, G. Fox, J. Futrelle, D. Gannon, M. Hategan, G. Kandaswamy, G. von Laszewski, M. A. Nacar, M. Pierce, E. Roberts, C. Severance, and M. Thomas, "The open Grid computing environments collaboration: Portlets and services for science gateways," *Concurrency and Computation: Practice & Experience*, vol. 19, no. 6, pp. 921–942, April 2007.

[28] M. Pierce and G. Fox, "Making scientific applications as Web services," *Computing in Science & Engineering*, vol. 6, no. 1, pp. 93–96, January 2004.

[29] J. Novotny and O. W. M. Russell, "GridSphere: A portal framework for building collaborations," *Journal of Concurrency and Computation: Practice and Experience*, vol. 16, no. 5, pp. 503–513, April 2004.

[30] I. Kelley, J. Novotny, M. Russell, and O. Wehrens, "Jetspeed evaluation." http://www.gridsphere.org/gridsphere/docs/jetspeed-eval.pdf, June 2002.

[31] B. Sotomayor, "The Globus Toolkit 4 programmer's tutorial." http://gdp.globus.org/gt4-tutorial/, November 2005.

[32] O. Tatebe, N. Sonoda, and S. Sekiguchi, "Gfarm v2: Design and implementation of global virtual file system," *IPSJ SIG Notes*, vol. 2004, no. 81, pp. 145–150, July 2004.

[33] S. R. Soltis, T. M. Ruwart, and M. T. O ' Keefe, "The global file system," in *Proceedings of the 5th NASA Goddard Conference on Mass Storage Systems and Technologies*, pp. 319–342, September 1996.

[34] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke, "A resource management architecture for metacomputing systems," in *Proceedings of the 4th Workshop on Job Scheduling Strategies for Parallel Processing, LNCS 1459*, pp. 62–82, Springer-Verlag, March 1998.

[35] I. Foster and N. Karonis, "A Grid-enabled MPI: Message passing in heterogeneous distributed computing systems," in *Proceedings of Supercomputing Conference 98*, pp. 46–46, ACM Press, November 1998.

[36] T. Kudoh, Y. Ishikawa, and M. Matsuda, "Evaluation of MPI implementations on Grid-connected clusters using an emulated wan environment," in *Proceedings of the third IEEE/ACM International Symposium on Cluster Computing and the Grid*, pp. 10–17, May 2003.

[37] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova, "Overview of GridRPC: A remote procedure call API for Grid computing," *GRID COMPUTING - GRID 2002, LNCS 2536*, pp. 274–278, November 2002.

[38] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka, "Ninf-G: A reference implementation of RPC-based programming middleware for Grid computing," *Journal of Grid Computing*, vol. 1, no. 1, pp. 41–51, June 2003.

[39] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke, "Condor-G: A computation management agent for multi-institutional Grids," *Cluster Computing*, vol. 5, no. 3, pp. 237–246, July 2002.

[40] X. Wei, Z. Ding, S. Yuan, C. Hou, and H. Li, "CSF4: A WSRF compliant meta-scheduler," in *Proceedings of 2rd Workshop on Grid Computing and Applications (GCA2006)*, pp. 61–67, June 2006.

[41] E. Huedo, R. S. Montero, and I. M. Llorente, "A modular meta-scheduling architecture for interfacing with pre-WS and WS Grid resource management services," *Future Generation Computing Systems The International Journal of Grid Computing: Theory, Methods and Applications*, vol. 23, no. 2, pp. 252–261, February 2007.

[42] S. Krishnan, B. Stearn, K. Bhatia, K. K. Baldridge, W. Li, and P. Arzberger, "Opal: Simple Web services wrappers for scientific applications," in *Proceedings of IEEE International Conference on Web Services (ICWS'06)*, pp. 823–832, September 2006.

[43] P. Kacsuk, T. Kiss, A. Goyeneche, T. Delaitre, Z. Farkas, and T. Boczko, "High-level Grid application environment to use legacy codes as OGSA Grid services," in *Proceedings of 5th IEEE/ACM International Workshop on Grid Computing*, pp. 428–435, November 2004.

[44] V. Sanjeepan, A. M. Matsunaga, L. Zhu, H. Lam, and J. A. B. Fortes, "A service-oriented, scalable approach to Grid-enabling of legacy scientific applications," in *Proceedings of The 2005 IEEE International Conference on Web Services (ICWS 2005)*, pp. 553–560, July 2005.

[45] G. Kandaswamy, L. Fang, Y. Huang, S. Shirasuna, S. Marru, and D. Gannon, "Building Web services for scientific Grid applications," *IBM Journal of Research and Development*, vol. 50, no. 2/3, pp. 249–260, March 2006.

[46] H. Nakamura, S. Date, H. Matsuda, and S. Shimojo, "A challenge towards next-generation research infrastructure for advanced life science," *New Generation Computing*, vol. 22, no. 2, February 2004.

[47] "OASIS Web Services Notification (WSN)." `http://www.oasisopen.org/committees/tc_home.php?wg_abbrev=wsn`, October 2006.

[48] T. J. Ewing, S. Makino, A. G. Skillman, and I. D. Kuntz, "DOCK 4.0: Search strategies for automated molecular docking of flexible molecule databases," *Journal of Computer-Aided Molecular Design*, vol. 15, no. 5, pp. 411–428, May 2001.

[49] E. Hatcher and S. Loughran, *Java Development with Ant*. Manning Pubns Co, August 2002.

[50] T. Sakuma, H. Kashiwagi, T. Takada, and H. Nakamura, "Ab initio MO study of the chlorophyll dimer in the photosynthetic reaction center. i. a theoretical treatment of the electrostatic field created by the surrounding proteins," *International Journal of Quantum Chemistry*, vol. 61, pp. 137–151, December 1997.

[51] Y. Fukunishi, Y. Mikami, and H. Nakamura, "The filling potential method: A method for estimating the free energy surface for protein-ligand docking," *Journal of Physical Chemistry B.*, vol. 107, pp. 13201–13210, November 2003.

[52] "MPI-2: Extentions to the Message-Passing Interface." `http://www.mpiforum.org/`, July 1997.

[53] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir, "MPI-2: Extending the message passing interface," in *Proceedings of Euro-Par '96 Parallel Processing, LNCS 1123*, pp. 128–135, August 1996.

[54] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, November 1999.

[55] D. Stamdley, H. Toh, and H. Nakamura, "Detecting local structural similarity in proteins by maximizing the number of equivalent residues," *PROTEINS: Structure, Function, and Bioinformatics*, vol. 57, no. 2, pp. 381–391, November 2004.

[56] Gamma, Erich, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, January 1995.

[57] W. Gentzsch, "Sun Grid Engine: Towards creating a compute power Grid," in *Proceedings of 1th IEEE International Symposium on Cluster Computing and the Grid Workshop*, pp. 35–36, May 2001.

[58] R. L. Henderson, "Job scheduling under the portable batch system," in *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 279–294, April 1995.

[59] P. Troger, H. Rajic, A. Haas, and P. Domagalski, "Standardization of an API for distributed resource management systems," in *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid 2007 (CCGrid 2007)*, pp. 619–626, May 2007.

[60] "WSTest 1.0 - Web Services Performance in Java and .NET." `http://java.sun.com/performance/reference/codesamples/`, July 2004.