# Ubiquitous Grid Service Interoperation Protocol Through Polyarchical Middleware

Mengjie, Yu A.Taleb-Bendiab D.Reilly Wael Omar

School of Computing and Mathematical Science
Liverpool John Moores University
Byrom Street Liverpool, L3 3AF UK
cmsmyu@livjm.ac.uk a.talebbendiab@livm.ac.uk d.reilly@livjm.ac.uk cmpwomar@livjm.ac.uk

## Abstract

Next generation software applications will be required to run on globally distributed heterogeneous assemblies of disparate resources including: emerging computing grids. Such application calls for seamless integration and interoperation between varieties of service standards and architectures developed and deployed using existing service middleware standards and architectures such as; DCOM, COBRA, Jini, Web service UPnP. Whilst such middleware adequately provide different APIs, programming models for distributed components and services integration and interoperation at both design and runtime. There is still need for additional middleware service to support runtime services invocation regardless of the components/service standards and type of middleware used. Based on an ongoing research focusing on self-adaptive software for adaptive middleware, this paper will describe a proposed on-demand (runtime) service invocation mechanism, and the associated service interoperation protocol. This will be followed by a brief introduction of a proposed polyarchical middleware, which is capable of discovery/lookup service components, dynamic service invocation adaption, and service interoperation management of given end-user applications.

## Keywords
Middleware, Service-Oriented, Service Interoperation Protocol, Dynamical Adaptation

## 1. Introduction

Over the recent years, grid computing [1] has been promoted as the new revolution promising to offer end-users large-scale virtual computing infrastructure including: hardware, software and data, which can be physically dispersed over clusters of different computers running on a range of platforms, operating systems and even with different standards and protocols.

In addition, other web developments led many advocate of service-oriented model to speculate that in the near future end-users need not be anymore tied to a particular server and/or service providers, but can on-demand discover and subscribe to a given service of choice. However, as attractive vision as it is it requires addressing many technical issues including: the development of universal open standards for grid and web services development, deployment and integration. Also their seamless usage and interoperation with "legacy" grid/software services already published using a variety of standards and middleware such as; DCOM, COBRA, Jini, Web Services, UPnP[2].

Recent re-engineering initiatives of grid toolkits such as Globus into Open Grid Service Architecture (OGSA[1]) to adopt the web service standards are now underway. However, the middleware community recognizing the requirements of cross middleware integration has taken to different approaches to provide developers with the "glue" technology to interoperate distributed components and services developed using a range of middleware and service standards including; J2EE, CORBA and web services.

However, so far these initiatives cater well for design-time integration and interoperation, but lack a uniform abstraction and/or programming, interaction and control models to support runtime self-adaptive services interchangeability including runtime service invocation be they "legacy" services or not.

To this end, we describe an extension to core middleware services' model that is capable of dealing with dynamic behaviur and accommodating different programming technologies at runtime. We feel that service interoperation and integration will become an essential requirement for low-cost life-time management of grid-based and service-oriented architecture enabling infrastructure, components and/or services to undergo dynamic changes due to failure, upgrades, replacement

and/or evolution. Following such changes, the component may still provide the same service, but may use a different protocol or have different bindings. Client should still be able to access this service through *adapters*, whose code will deal with switching of invocation calls from one standard to another. For example, a service-providing component may have been deployed to use Java Remote Method Invocation (RMI) and to Java-enabled, the service can be directly accessed via Java's RMI APIs. However, if a non-Java client needs to access the service via a Web service protocol it may do so through an adapter, which switches the invocation semantics between Java RMI and Web service protocols.

The remainder of the paper will outline a solution to the service invocation problem based a proposed model of polyarchical middleware. This architecture is based on a service interoperation protocol, which serves to mediate runtime those inconsistencies and hide any service adaptation from clients. For the remainder of this section, we briefly review several related works, which focus on the same problem but adopt different solutions to those of our own. The paper is structured as follows: Section 2, describes several ongoing related research and development themes. Section 3, describes the service interoperation protocol, which we use to support our service interoperation framework. Section 4, considers the development of the Ubiquitous Middleware architecture, which mediates service invocation calls between different existing service middleware. Section 5, describes a recent case study conducted to evaluate and test our framework. Finally, Section 6 draws overall conclusions and mentions directions for future work.

## 2. Relate Work

The connector model has been widely used by the component-based software engineering community model and abstract inter component interaction and invocation. This has been further developed into the open connector to enable software component invocation and interaction using different standards.

OpenWings [3] was first set up to meet the military use to discover replacement services in a critical and volatile environment, which request lease manual configuration, auto-discovery by another system and standard. Later it develops several components to consist the whole framework. One of the components is Connector, which is used to represent the inter-communication between two different standards. By creating a protocol abstract layer, both sides need to implement a special APIs to translate to and back from that layer, where all the data and information will be transmitted.

Web Service Invocation Framework (WSIF) [4] is designed to supply users with simple APIs to invoke Web services through abstract representations of the services. The framework allows developers/users to adopt their own programming models without having to worry about the details relating to the implementation and access method of the service. So far, WSIF provides APIs that allows users to use different SOAP packages to provide a degree of protocol-independence, but it does however assuming as the underlying protocol.

Open Grid Services Architecture (OGSA), still in the early stages of development, has been introduced by IBM to allow Grid services discovery, sharing and collaboration in a dynamic distributed environment. It is a distributed architecture, which ensures the service discovering, sharing and collaboration among different standards / protocols.

## 3. Service Interoperation Protocol

In this section, we describe the proposed Service Interoperation Protocol to support the service interoperation and integration in this architecture through the proposed polyarchical middleware.

### 3.1 Interoperation Protocol Scenario

We begin by considering two applications, based on different programming model (e.g. Client-Server / Peer 2 Peer) that wish to communicate over a network. For such communication, an application protocol [5], must be agreed and/or negotiated by both sides to enable the data exchange and hence communication. In this Polyarchical Middleware case, they are likely to be end-users and service-providing components. In order to facilitate interoperation, clients should not be tied to one particular programming technology so that they may use services, which are based on different technologies. The Service Interoperation Protocol is introduced to address this very need and enable the inter-communication between the applications by accommodating different programming technologies and protocols (discovery/lookup and invocation). This protocol sits above Hypertext Transfer Protocol (HTTP), which is widely considered as a general purpose application protocol because it is easy and familiar for most applications to use. The service interoperation protocol also reuses a number of the ideas and implementation details of HTTP [5].

Essentially, the Service Interoperation Protocol is used to dynamically support code adaptation by providing the necessary extra files at runtime, that are required to convert from one standard direct to another, without translating service calls into a common standard layer.

### 3.2 Service Adaptation Solution

In conjunction with the interoperation protocol, our approach makes use of service adapters. The adapters dynamically provide the necessary code to handle service interoperation and integration issues among different standards and these adapters set our work aside from that conducted by Openwings and WSIF. The service adapter files are essentially designed to facilitate code adaptation from one standard to another. The adapters make use of a template file, in which the invocation code for each individual standard is already specified. The following diagram shows the anatomy of service interoperation protocol.

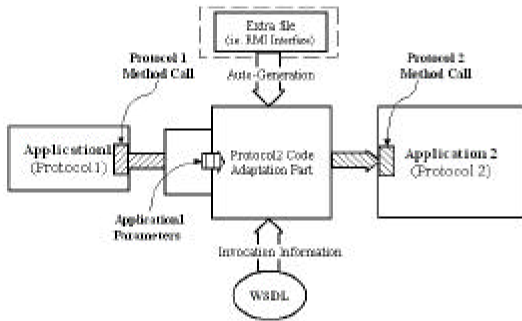


**Figure 1. Service Interoperation Protocol**

The protocol 1 method call can be reset to the normal protocol 2 mechanism after using the code adaptation. Application 2 can still be invoked by the protocol 2 method call in normal data encoding (i.e. serialization / deserialization in Java RMI or marshanlling / unmarshalling in SOAP-RPC of parameters and results ), which the other proposes including OpenWings intend to translate method calls into a standardized protocol transmission.

Invocation information is retrieved by parsing a service Web Services Description Language (WSDL [6]) and any extra file such as stub and/or proxy classes (i.e. Java RMI Interface [7]) is also auto-generated at runtime.

A Polyarchical Middleware framework, discussed in the next section, is introduced to hide the technical details of service invocation adaption from end-users at runtime.

## 4. Architecture of the Framework

This section considers the "on-going" development of the framework, which has been implemented using Java language.

### 4.1 Polyarchical Middleware Scenario

The term *Polyarchical* has been first coined by Sun™ Open Network Environment (Sun ONE [8]) initiative, which offers application developers and end-users a software component view of the future. Service interoperation and integration are intended to help end-users set up software applications by allowing the same service to be accessed through more than one binding. Application-level software components may be developed using different standards and connectors are used to facilitate inter-communication between components of different standards.

Our own polyarchical middleware is based on service-oriented architecture, where developers and end-uses may access service interoperation and integration utilities through a common standard i.e. HTTP. The framework is designed to dynamically lookup and invoke matching services through runtime code generation, thus abstracting service discovery and invocation and simplify the service interoperation and integration. Through the framework, end-users may concentrate specifically on their applications without having to worry about inconsistencies or mismatches between component standards. The framework architecture (Fig. 2) is based on a three-layer model, which is described below.

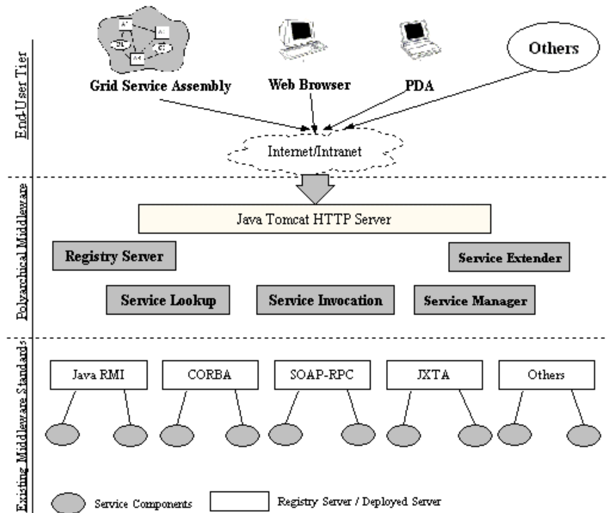


**Figure 2. Framework Architecture**

1. End-Users: refers to those service consumers/producers access the polyarchical middleware. Typically they could be normal Web browsers, PDAs (or other wireless devices), or even commercial distributed applications (SOAP-RPC, Java RMI/Jini, CORBA, Web services etc).
2. Polyarchical Middleware: the middleware framework, which mediates service invocation calls and hides technical details from end-users.
3. Existing Middleware Standards: refer to the middleware technologies that provide normal middleware discovery/lookup and invocation protocols. Through the polyarchical middleware, service components developed using these

middleware standards may be integrated and interoperated by end-users.

It is highly likely that new service component technologies will arise in the future and with this in mind, the polyarchical middleware is designed to accommodate new the adapter template files. The framework provides this functionality through a Template Markup Language[1] (TML).

## 4.2 Framework Architecture

As shown in Figure 2, the architecture contains a number of components/services including:

- HTTP server: This provides access to the framework via Internet/Intranet connections. More particularly, Java Tomcat HTTP server was used to provide HTTP (TCP/IP) access the polyarchical middleware framework. A Java Servlet container hosted by Tomcat server is used to retrieve service invocation calls from end-users and return any result from the invocations back to end-users.
- Service Lookup Process: After the Java Servlet retrieves service invocation calls through the HTTP server, which switches/dispatches any service request to a background Java file, which handles the service discovery/lookup issues. The Service Lookup Process may be used to allow an end-user to lookup service components that match its needs among several different service bindings. The Service Lookup process involves two separated stages: 1) it first uses keywords, such as service name, to search inside the local UDDI Registry server and returns an array of matched service; 2) In order to avoid user intervention, it checks the availability of each matched service individually. The check is conducted to detect and react to potential service failures by automatically initializing query adapter template files (for different bindings). If failures are detected, the template files are appended with information from an alternate service component, which provides the same semantics as the failed component.
- Registry Server: To support the service lookup interoperation problem a Universal Description, Discovery and Integration (UDDI) compliant registry server was introduced, which holds description of all published/available services (the same service may provide different accessible bindings) encoded in Web Service Definition language (WSDL). The WSDL includes service location, the service invocation mechanism and even the server from which the service has been deployed. WSDL was used, since it has already been widely accepted as a common standard service description language.
- Service Invocation Process The Service Invocation Process initializes the invocation adapter template files to access / invoke the matched service components. Similar to the Service Lookup process, the invocation process needs JWSD APIs to get the invocation information from service WSDL files, which is appended to the adapter templates files. At this stage, some relevant / extra files, which are used to assist the service invocation, are automatically generated. For example, the service interface [7] will be required by a Java RMI-based service/connector. The polyarchical middleware will generate at runtime the associated RMI Interface by parsing the service description encoded using the WSDL specifications[2]. Failure due to changes in service components will also be detected by the polyarchical middleware at this stage and alternate components and connector will be sought.
- Extended Adapter: To make this polyarchical middleware more extendable in service interoperation and integration, we intend to introduce a Template Markup Language (TML) into the framework design. The TML will be used to add / describe more adapters as and when new service component standards are introduced in future. New service adapter templates can be added into this architecture after storing relevant information in the file described using the TML in XML-based format.

## 5. Illustrative Example

As indicated above the framework is still under ongoing development. To date, several service adapter templates and components of the framework have already been implemented and tested on an existing case study, from which this illustrative example has been adapted (Fig. 3).

[1] Which is briefly outlined below though the full description of the TML and its usage is beyond the scope of this paper

[2] Which describes the service method names and parameters

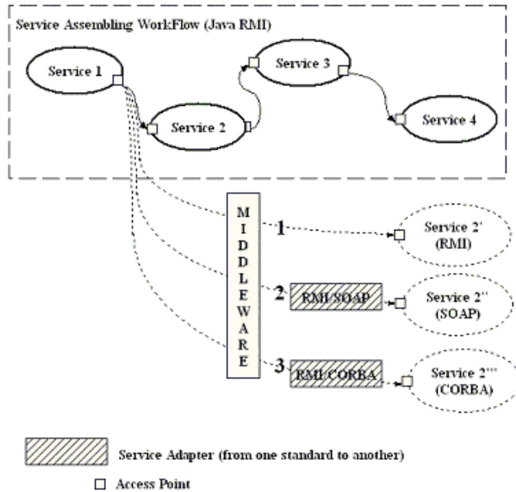


**Figure 3. Illustrative Example**

Figure 3 shows a simplified service assembly (representing the Grid Service shown in Figure 2) of four services that represent a given user application. The services are engaged in synchronous communication and Java RMI is used as the underlying protocol for the connector model between the four service-providing/using components. It assumed that component 2 (service 2) has failed. At runtime, the middleware enables the discovery of service 2' and the replacement of service 2 by service 2’. The polyarchical middleware has maintained the service workflow of lookup, configuration and invocation of alternate services dynamically at runtime, subject to the availability of alternate services and the requirements of that service workflow.

For the first stage of the process, the polyarchical middleware retrieved the service request information from service 1 by catching the Java exception (java.lang.Exception) to detect the failure of service 2. The service request information is then used to locate a service, with a name ‘HelloService’ and a method ‘sayhello()’. This information is gathered by a Java Servlet file (as mentioned previously in section 4), which passes the request to the Service Lookup component so that it may discover an alternative.

The second stage involves the use of service lookup and invocation components to maintain service workflow by mediating service requests from Java RMI to another bindings or the same binding, assuming Java RMI is available. The service lookup component first queries the alternative services on the local UDDI registry server using the keyword ‘HelloService’, retrieved from service1. The UDDI registry server returned an Iterator object (java.util.Iterator), which contains a list of different bindings for the matched HelloService’ service. After checking the availability of matched services individually, the service invocation component incorporates the service invocation code by auto-generating the extra files that may be needed at runtime.

So far, two ways of invoking alternative services have been implemented and tested: 1) The first uses alternative an alternative service of the same binding (Java RMI-Java RMI); 2) The second tries an alternative service with a different binding (Java RMI -SOAP-RPC. In the Java RMI test, the new RMI registry server Uniform Resource Locator (URL) ‘rmi://cmsmyu/HelloService’ and the codebase URL ‘http://cmsmyu:8080/stubcodebase/’, are obtained from the RMI servicebinding object returned by the UDDI server. These are required to configure the adapter template file at runtime. The Java interface file (MyServiceInterface.java) was also auto-generated after parsing the Java RMI WSDL specification. Java Reflection [9] method was then used to return a reference object for RMI remote object, which may be used for service invocation. In the SOAP-RPC test, the apache-SOAP rpcrouter URL and service method name along with the target object URL were configured into the SOAP-RPC adapter template file by parsing the WSDL as for the java RMI case. Finally, the alternative service (whether Java RMI or SOAP-RPC) was invoked using the code adapter files or extra file (Java interface).

These steps to achieve this are shown in the flow diagram of figure 4:

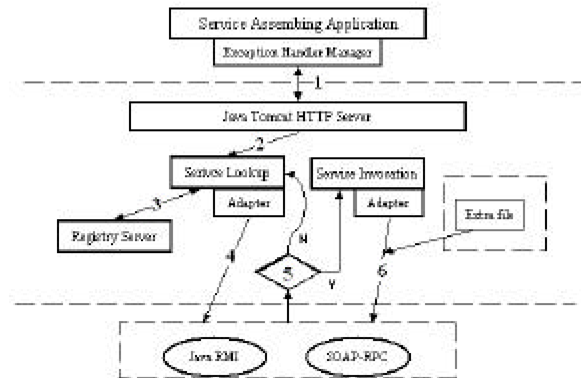


**Figure 4. Flow Diagram**

In this test, the Polyarchical Middleware maintained the service workflow by finding / invoking alternative services after auto-generating and configuring the necessary files at runtime. The service assembly workflow was able to continue working without having to halt or wait for the recovery of service2 as it was possible to invoke service2’ as an alternative without changing any code. The following figure shows the SOAP-RPC adapter code. At runtime, a ‘ParseWSDL.java’ file is initialised by setting the parameter WSDLURL for the parser of the SOAP-RPC rpcrouter URL and serivceName().

```
public class SoapAdapter {

  public String soapConnect(String wsdlURL ) throws Exception
  {
   // inital stage
     .....
     ....
   // Parse WSDL at runtime after get the wsdlURL from Register Server
       ParseWSDL parseWSDL= new ParseWSDL(wsdlURL);
   // Get the SOAP rpcrounter URL
       url=new URL(parseWSDL.soapURL());
   // Build a Call object
      .....
   // Get ServiceName
       call.setMethodName(parseWSDL.serviceName());
      ...
   // Set the parameters
      ....
   // Invoke the call.
      Response resp =null;
   // normal SOAP-RPC code
     .....
  }
}
```

**Figure 6. Parts of SOAP-RCP Adapter Code**

## 6. Conclusions and Further Work

Service interoperation based on the Polyarchical Middleware appears to be a promising approach. In this paper Service Interoperation Protocol outlined the use of code adapter templates with extra files (generated at runtime) instead of creating a protocol abstract layer. While, the Polyarchical Middleware enabled end-users to discover/invoke alternative service components at runtime to accomplish critical tasks, hiding the technical details behind.

Through this paper we have described an approach to service interoperation based on the polyarchical middleware that has delivered promising results. The paper has outlined a Service Interoperation Protocol, which makes use of code adapter templates with extra files (generated at runtime) instead of creating a protocol abstract layer. The polyarchical middleware enables end-users to discover/invoke alternative service components at runtime and accomplish critical tasks without having to worry about the technical details.

In our future work, we intend to extend the polyarchical middleware to increase its service interoperation functionality. In particular, development is underway to extend the proposed Polyarchical middleware including:

1. To offer an extensible code adapter software factory for more component standards and styles. This will include a runtime Java file generation, compilation and class loading to achieve runtime invocation and interoperation of a given service assembly. In addition, the proposed Template Markup Language (TML) will be used to define components styles, patterns and invocation standard properties, which are required during runtime code generation and/or software self-adaptation.
2. To support asynchronous mode of services' invocation, which is often used through message/event-oriented middleware. As indicated above (Sec. 5) currently only synchronous service invocation mode has been implemented and tested.
3. To develop a mechanism, similar to Jini Lease Manager [10]: this will be used to manage and update registered service on the local UDDI and deal with state registrations.
4. Best-fit matches: the use of service characteristics in order to select a "best-match" service from those services returned by the local UDDI server.
5. Multicast lookup: to enable the lookup of multiple UDDI registers.

## 7. References


[1] Liang-Jie Zhang, J.-Y.C., Researcher, IBM T.J. Watson Research Centre, "*Developing Grid Computing Application, Part1" ,* http://www1.ibm.com/grid/grid_what_is.shtml

[2] E.Grishikashvili, N.B., D.Reilly,M.Allen,M. Yu and A. Taleb-Bendiab;School of Computing and Mathematical Sciences Liverpool John Moores University, "*Automatic Computing: A Service-Oriented Framework to Support the Developement and Management of Distributed Applications",3rd Annual PostGraduate Symposium, PGNET2002*

[3] Wade Wassenberg, S.E., Motorola ISD, "*Protocol Independent Programming Using Openwings Connector Services", http://www.openwings.org/download.html*

[4] Matthew J. Duflter, N.K.M., Aleksander Slominski and Sanjiva Weerawarana;IBM T.J. Watson Research Center, "*Web Service Invocation Framework (WSIF)", http://www.research.ibm.com/people/b/bth/OOWS2001/duftler.pdf,* August 9, 2001.

[5] Marshall T. Rose, "*BEEP, The Definitive Guide", published by O'Reilly & Associates, Inc.,* March 2002 *ISBN: 0-596-0024-0.*

[6] Erik christensen, M.F.C., IBM Research; Greg Meredith, Microsoft;Sanjiva weerawarana, IBM Research, "*Web Services Description Language (WSDL) 1.1.",* http://www.w3.org/TR/wsdl.

[7] Microsystems., S., "*Designing a Remote Interface, Java RMI,Java tutorial", http://java.sun.com/docs/books/tutorial/rmi/designing.html.*

[8] Shin, S., "*Sun One: Now and Future",* http://www.plurb.com/webservices/SunONENowAndFuture_V4.pdf

[9] McCluskey, G., "*Using Java Reflection", http://developer.java.sun.com/developer/technicalArticles/ALT/Reflection/,* January 1998.

[10] Edwards, W.K., "*Core Jini (2nd Edition)",* published by Prentice Hall PTR, December 28, 2000, ISBN:0130894087.