Available online at http://www.journalcra.com

International Journal of Current Research
Vol. 3, Issue, 8, pp.082-087, August, 2011

INTERNATIONAL JOURNAL OF CURRENT RESEARCH

ISSN: 0975-833X

RESEARCH ARTICLE

FRAMEWORK FOR INTEGRATING SOFTWARE APPLICATIONS IN GRID COMPUTING ENVIRONMENT

Adesina, O.O*. and Aremu, D.R.

Department of Computer Science, Faculty of Communication and Information Science, University of Ilorin, Ilorin, Nigeria

ARTICLE INFO

Article History:
Received 12th May, 2011
Received in revised form
18th June, 2011
Accepted 28th July, 2011
Published online 5th August, 2011

Key words:
Grid computing,
Heterogeneity issues,
Distributed system.

ABSTRACT

Grid is a computing and data management infrastructure whose goal is to provide electronic underpinning for a global society in business, government, research, science and entertainment. Being a distributed system, grid is complex due to the heterogeneous nature of the underlying software and hardware resources forming it. The heterogeneous nature of grid will hinder interoperation of grid applications. In this paper, we present a framework for integrating grid applications in spite of its distributed and heterogeneous nature. To realize this, we perform extensive review of similar implementation solutions for managing and integrating heterogeneous distributed applications. Similarly, we have developed a model for integrating heterogeneous grid applications. Moreover, we discussed the implementation strategy for the model designed. Finally, we summarized this paper and stated future thoughts to realize a fully operational grid computing environment.

INTRODUCTION

Buyya *et al.* [1] defined grid as a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed `autonomous' resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality-of-service requirements. Grid computing integrates networking, communication, computation and information to provide a virtual platform for computation and data management in the same way the Internet integrates resources to form a virtual platform for information. Dubitzky [2] enumerated the promises of sharing resources with grid. Grid computing, being a distributed system is complex due to the heterogeneous nature of the underlying software and hardware resources forming it. According to Michael Stal [3], documented issues of heterogeneity embattling distributed systems such as grid include differences in: (i) network technologies, devices, and operating systems; middleware solutions and communication paradigms; (ii) programming languages; (iii) services and interface technologies; (iv) domain and machine architectures; and (v) data and document formats. However, grid computing will fulfill its promises, if and only if the issues of heterogeneity can be managed. The aim of this paper is to develop a software model for integrating software applications in the grid computing environment.

***Corresponding author:** opeyemi.adesina@gmail.com;draremu2006@gmail.com

We used the following objectives to achieve the aim stated for this paper: (i) to analyze the existing implementation solutions for distributed systems; (ii) to design a model/framework for implementing the grid systems and to implement the model designed. In the literature, various implementation solutions have been developed for integrating heterogeneous applications. These include CORBA, DCOM, GLOBE, and Java RMI. However, these solutions are either programming language-dependent or platform-specific. This has led us to the design of a model for software collaboration in the grid environment. The model designed is based on the specifications and standards of web services framework. The paper also presented an implementation prototype of the model designed. The rest of the paper is organized as follows: Section 2 presented the related work by discussing various implementation solutions that can be adopted for the development of a blueprint for integrating grid applications. In section 3, we discussed the Software Infrastructural model for integrating grid applications. Section 4 presented the internal structure of the grid systems, while in section 5, we presented a prototype implementation of the model designed. Section 6 concluded the paper.

Related Work

In the literature, various implementation solutions have been developed to manage heterogeneity in distributed systems. These include CORBA, DCOM, GLOBE, and Java RMI.

Common Object Request Broker Architecture (CORBA)

Common Object Request Broker Architecture (CORBA) [4,5] is an industry defined standard for distributed systems. An important goal of the Object Management Group, OMG with respect to CORBA was to define a distributed system that could overcome many of the interoperability problems, with integrating networked applications. CORBA's global architecture adheres to a reference model of the OMG. The reference model consists of four groups of architectural elements connected to the Object Request Broker (ORB). ORB forms the core of any CORBA distributed system; it is responsible for enabling communication between objects and their clients while hiding issues of heterogeneity. CORBA adopts the remote-object model. In its remote-object model, implementation of an object resides in the address space of a server. While CORBA server object and service are specified in Interface Definition Language (IDL). This IDL provides a precise syntax for expressing methods and their parameters. CORBA interface is a collection of methods, and objects specify which interfaces they implement. These interfaces are binary in nature and independent of programming languages. On the other hand, client applications usually have a proxy available that implements the same interface as each object it is using. A proxy is a client-side stub that merely marshals an invocation request and sends that request to the server. A response from the server is unmarshaled and passed back to the client. Server-side proxy can be statically compiled from CORBA IDL specification or dynamically available as a skeleton. When using dynamic skeleton, an object will have to provide proper implementation of the invoke function as offered to the client. To allow dynamic construction of invocation requests, it is important that a process can find at runtime what an invocation look like. This is addressed by CORBA interface repository. CORBA interface repository stores all interface definitions. In CORBA interface repository is view as a part of CORBA that assist in runtime type checking. The interface repository stores all information needed for the implementation and activation of objects. CORBA uses Portable Object Adapter (POA) for its objects activation. Many interoperability issues in early versions of CORBA systems were addressed by standard communication protocol, known as General Inter-ORB Protocol (GIOP). GIOP is actually a framework for a protocol; it assumes that an actual realization is executed on top of an existing transport protocol. However, it is essential that transport protocols reliable, connection-oriented, and provide notion of a byte stream, along with a few other features. TCP satisfies these requirements. The realization of GIOP running on top of TCP is called the Internet Inter-ORB Protocol, IIOP [4].

The adoption of CORBA as a middleware for the integration of distributed applications has some benefits and limitations. One of its benefits is that CORBA implementation naturally accommodates extensions. Being an effort of committees, it has features and facilities in abundance. It is flexible in architectural models. CORBA is programming language, operating systems, and machine independent. It offers facility to find services that are available to a process. CORBA allows dynamic construction of invocation requests. Its flexibility in assigning interface identifiers, allows uniqueness in interface definitions within interface repository. It is a better platform for reusing legacy systems. CORBA is suitable for large web-enabled applications where performances under heavy client load are crucial. Although CORBA is programming language independent, however it is necessary to provide exact rules concerning the mapping of IDL specifications to existing programming languages. Till date, only few of these rules are available. It also illustrates that making a simple distributed system may be somewhat overwhelmingly difficult exercise.

Distributed Component Object Model (DCOM)

Distributed Component Object Model (DCOM) [4,6] originated from Component Object Model (COM). COM is the underlying technology of various Windows operating systems produced by Microsoft starting with Windows '95. Like all object-based systems, DCOM adopts remote-object model. DCOM object is an implementation of an interface which can either be placed in the same process, as client on the same or remote machine. It has only binary interfaces, and each interface is essentially table of pointers to the implementations of the methods that are part of the interface. To define these interfaces, DCOM uses Microsoft IDL (MIDL). The standard layout for binary interfaces is generated from the IDL. These binary interfaces are programming language independent. And each interface in DCOM has a unique 128-bit identifier, called its Interface identifier (IID). Objects in DCOM are created as an instance of a class. To do so, it is necessary to have that class available. For this reason, DCOM has class objects. Formally, such an object can be anything that implements the IClassFactory interface. This interface contains the method CreateInstance, which is comparable to the new operator in Java. By invoking CreateInstance on a class object has the implication of creating DCOM object, containing implementation of the interfaces associated with the class object. A class object is a collection of objects that implement the same set of interfaces. Objects of the same class differ only with respect to their current state. By instantiating an object from a given class objects, it becomes possible to invoke the methods contained in those interfaces. Objects invocation in DCOM can either be dynamic or static. All objects that implement IUnknown interface can be invoked statically; while objects for which an invocation request can be constructed at runtime are required to implement IDispatch interface. The equivalent of CORBA interface repository in DCOM is called a type library. The type library is generally associated with an application or other component consisting of various class objects. The library itself can be stored in a separate file, or included as part of an application. In any case, a type library is primarily used to find out exact signature of a method that is to be invoked dynamically. Object activation in DCOM is supported by the combination of Windows registry and Service Control Manager (SCM). On the client side, a process is given access to the SCM and the registry to help look up and set up a binding to a remote object. The client will be offered a proxy implementing the object's interface. Every server object has a stub for marshaling and unmarshaling invocations, which are passed to the actual object. Communication between the client and server is normally done by means of RPC.

DCOM is a widely accepted middleware solution, with tens of millions of people using windows daily in networked environment [4]. It is programming language independent. DCOM also supports dynamic invocation of objects. It offers interface repository for storing and retrieving interfaces. To facilitate object activation, DCOM offers Service Control Manager (SCM) in conjunction with the Window registry. Due to the transient nature of DCOM's objects, garbage collection is less an issue. In spite of these benefits, DCOM has its problems. One of these is that DCOM is not an effort of a committee. Based on this, it offers minimal set of core elements from which components and services are built. DCOM is an intricate system, because similar things can be done in different ways, and such that coexistence of different solutions is sometimes even impossible. It is platform dependent (i.e. Windows platforms). Passing object references, to another process in DCOM demands special measures, because its objects are transient by virtue of its object model.

Global Object-Based Environment (GLOBE)

Global Object-Based Environment (GLOBE) [4,7] is an object-based system in which scalability plays a central role. All aspects that deal with constructing a large-scale wide-area system that can support huge numbers of users and objects drive the design of Globe. Like other object-based systems, objects in GLOBE are expected to encapsulate state and operations on that state. An important difference with other object-based systems is that objects are also expected to encapsulate the implementation of policies that prescribe the distribution of an objects state across multiple machines. Objects in Globe describe how, when, and where their state should be migrated and replicated. Unlike most other object-based distributed systems, Globe does not adopt remote-object model. Instead, the state of an object can be distributed and replicated across many processes. Any process that is bound to a distributed shared object is offered a local implementation of the interfaces provided by the object. Such a local implementation is called a local representative or object. Each local object implements a standard object interface called SOInf. Local objects are assumed to implement binary interfaces that essentially consist of tables of function pointers. The specification of interfaces is supported by an Interface Definition Language (IDL). Local objects consist of at least four sub-objects. These are semantics, communication, replication, and control sub-objects. Each of the sub-objects is used for special purpose. Semantic sub-object implements the functionality provided by a distributed shared object. Communication sub-object is used to provide a standard interface to the underlying network. The most important sub-object to all Globe objects is the replication sub-object. It implements the actual distribution strategy for an object. Control sub-object is used as an intermediate between user-defined interfaces of the semantics sub-object and standardized interfaces of the replication sub-object. In contrast to CORBA and DCOM, Globe does not provide an interface repository, nor does it have equivalent of an implementation repository. This is a result of the object model adopted by Globe. In the same vein, binding a process to an object in Globe involves loading the specific local object into its address space as indicated by the distributed shared object to which it is binding. A complete binding starts with provision of human-readable name to the DNS-based naming service provided by Globe; and the service returns a globally unique and location-independent object handle. The globally unique object handle is given to the Globe location service; and a set of contact addresses for the given object is returned. From the set of contact addresses returned, a process will select a contact address using a selection criterion such as the distance to an address or expected QoS when binding to a specific address. Each contact address specifies exactly the local object that the process should load. Local objects are loaded and instantiated from a class repository. Finally, binding ends with the initialization of local objects. Through these objects, clients subsequently contact the local objects that form part of the distributed shared object.

Using Globe as an infrastructure for integrating distributed software applications has great benefits as well as disadvantages. Its benefits are that, it can be used to support a huge number of users and objects spread across the internet, which is contrary to most other object-based distributed systems. Globe objects make decisions on how, when, and where its state should be migrated? They may also determine the security policies and implementation. Because the location service may return many contact addresses for an object, it does give options to select a contact address based on any selection criterion, such as distance or expected QoS. Objects contact addresses are flexible in specifications. This empowers clients to use any implementation, provided it obeys the rules guiding the protocol. However, the flexibility in contact address specifications comes with a price of having to make implementations for different local objects, and possibly for different operating systems and machine architectures.

Java Remote Method Invocation (JRMI)

The major goal of introducing distributed objects with Java Remote Method Invocation (JRMI) [4,8] was to keep as much of the semantics of non-distributed objects as possible. That is to maintain high degree of distribution transparency. JRMI adopts remote objects model as only form of distributed objects. By remote object, we mean a distributed object whose state always resides on a single machine, but whose interfaces can be made available to remote processes. Remote objects in JRMI are built from two different classes. A class contains an implementation of server-side code, known as the server class. The class contains an implementation of that part of the remote object that will be running on a server. The server-side stub, otherwise called skeleton, is generated from the interface specifications of the object. The other class contains an implementation of the client-side, which we refer to as the client class. This class contains an implementation of a proxy. Similar to the skeleton, this class is also generated from the object's interface specification. The main function of proxy is to convert each method call into a message that is sent to server-side implementation of the remote object, and convert a reply message into the result whenever a method is called. For each call, it sets up a connection with the server, which is subsequently terminated at the end of the call. For this purpose, the proxy needs the server's network address and endpoint. This information and the local identifier of the object at the server, is always stored as part of the state of a proxy. In principle, proxy marshaling involves conversion of its complete implementation to a series of bytes. However, marshaling code like this is inefficient and may lead to large references. Hence, in JRMI proxy marshaling involves generation of implementation handle, specifying the classes needed for the construction of proxy. This approach reduces references to remote objects to a few hundred bytes. The approach is flexible and it is one of the distinguishing features of JRMI; and it allows for object-specific solutions. Binding a remote object by the client involves copying the entire state to client machine. Each time the client invokes a method, it operates on a local copy. To ensure consistency, each invocation checks for change of state at the server side, in which case the local copy is refreshed. Therefore, the developer of the remote object will only have to implement the necessary client-side code and clients dynamically download it during binding. Passing proxy in JRMI as parameter works only because each process is executing the same virtual machine. That is each process is running in the same environment. A marshaled proxy is simply unmarshaled at the receiving side, after which its code can be executed.

Benefits of using JRMI as a software infrastructure for integrating distributed resources are enormous. The distinction between local and remote objects is hardly visible at the language level. It also hides most of the differences during a remote method invocation. Java RMI makes distribution apparent where a high degree of transparency is simply too inefficient, difficult, or impossible to realize. The complexity associated with marshaling proxy by converting its complete implementation into series of bytes, was addressed by generating implementation handle, specifying precisely the classes needed for constructing proxy. This makes Java RMI the most efficient of all object-based distributed systems. Also, JRMI can hide most of the differences during a remote method invocation. However, primitive or objects involved in this process must be serializable; but platform-independent objects such as file descriptors and sockets can not be serialized.

Model for Integrating Grid Applications

This section presents a model for integrating software applications in the grid computing environment. The model (Figure 1), is composed of a set of clients and servers systems; resource broker; Wide Area Networks (WAN)/Local Area Network (LAN); and Application Programming Interfaces (e.g. inquiry and publisher APIs). A Client in the context of this model refers to a computer system housing a software application or a process that accesses service(s) on other computer system(s), known as servers, via a network. A server in similar context is a computer system with computer programs or software running as services, and serves the need or request of other application programs (clients).

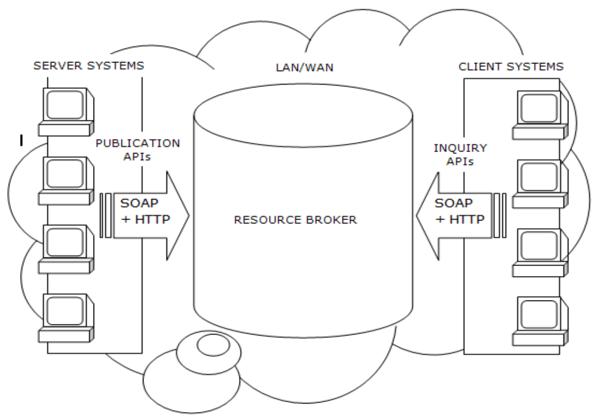

Figure 1: Model for integrating software applications in grid

Resource broker is agent software in grid environment that acts as middleman between clients and servers. It provides interfaces for service providers (servers) to store and update information of services they offer in the grid computing environment. These interfaces are provided through the Publication API. Similarly, clients on can look-up or access published service(s) information through the Inquiry API in the model presented (figure 1). In other words, the resource broker is software that implements the mechanism needed for discovery, description and integration of services, in order to overcome all forms of interoperability problems characterizing distributed community. Grid as a distributed system demands that its systems must be interconnected. Hence, the block labeled WAN or LAN represents a backbone linking all computer resources in grid community. This implies that grid may be set up within a confined geographical location (LAN) or the internet (WAN). In order to hide completely, the differences in grid resources, we have considered various technological infrastructures in the design of this model. These include eXtensible Markup Language (XML), Web Service Description Language (WSDL) [9,10], Simple Object Access Protocol (SOAP) [10,11], Universal Description, Discovery and Integration (UDDI) [12,13], and Hypertext Transport Protocol (HTTP) [14].

XML [15] is a document processing standard that is officially recommended by the World Wide Web Consortium (W3C). The use of XML in the design of our model is based on the facts that it solves all problems of heterogeneity in a distributed community. For the capability of XML to solve all forms of heterogeneity, WSDL, SOAP, and UDDI (i.e. web services specifications) are built on XML. Similar to the GIOP in CORBA, SOAP has been adopted as a standardized packaging protocol for messages shared between software applications in grid. Software interfaces are usually defined using an Interface Definition Language. In the design of this model, we adopted WSDL as a description language for expressing interface of services. This is similar to CORBA IDL and DCOM MIDL. Once the WSDL of a service has been created, a client must be able to find it, in order to be able to use it. This is referred to as service discovery. Like interface repository in CORBA and Window registry in DCOM, UDDI has been adopted as a resource discovery mechanism in the design of our model. Furthermore, to enable direct application-application on the network layer requires a standardized transport protocol. Transport protocols such as TCP, Jabber, SMTP and HTTP has been developed. However, we have adopted HTTP in the design of our model, because it provides the most ubiquitous firewall support.

Internal Structure of Grid Systems

The figure 2, shows the internal structure of the grid systems. The figure is made of three parts: the client, the server, and the resource broker. The client is divided into four basic parts. These include client application, stub or proxy, dynamic invocation interface, and operating system. Client application refers to computer program on client's system. Its input is an output from the service execution in the server's address space. Programs on client system can only access service implementation remotely via stub or dynamic invocation interface. Stub handles marshaling of client's requests and unmarshaling of server responses. The stub is generated by

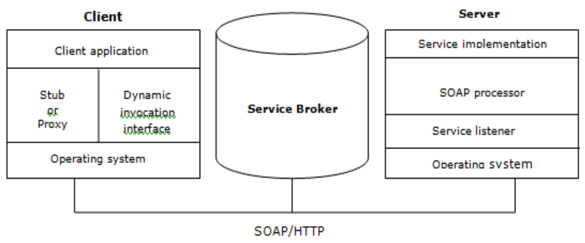

Figure 2: *The internal structure of systems in grid computing*

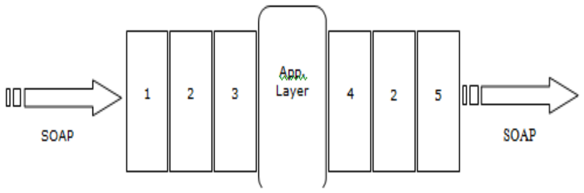

Figure 3: *Cross-sectional view of Service implementation vs. SOAP Processor*

downloading and compilation of WSDL file of the service implementation provided by the server. This is similar to the client-side proxy in CORBA and JRMI. Binding remote services by grid clients can not only be done through proxy, but also dynamic invocation interface. Dynamic binding allows client application to invoke service whose data types were unknown at the time the client was compiled. The operating system is software that houses computer programs and data. It also manages computer hardware resources and provides common services for efficient execution of application software. The structure of server system presented in figure 2 is divided into four components. These are service implementation, SOAP processor, service listener and operating system. Service implementation is the actual software application offered as a web service in the presented grid computing environment. In the same vein, similar to the server-side stub in CORBA and skeleton in JRMI is the SOAP processor. The SOAP processor handles marshaling and unmarshaling of server response and client request respectively. A relationship between service implementation and SOAP processor is presented in figure 3 below. Furthermore, the service listener refers to software port. This acts as a unique address for locating service implementation of servers on the grid. Finally, the operating system component on grid's server is similar to that described under client.

Resource broker presented in figure 2 above is similar to interface repository and type library in CORBA and DCOM respectively. It is designed as a mechanism for describing, discovering and integrating software resources in the grid community. Although, the internal structure is not included in figure 2, we have divided the broker into two basic parts. These include the service endpoint and the broker software. The service endpoint comprises of the operating system, service listener, and SOAP processor in the server presented in Figure 2. While the broker software represents the implementation of the UDDI core data structures. In other words, broker software is a service implementation which can be consumed by clients and servers for inquiry and publication respectively. However, the SOAP processor shown in figure 3 is divided into three different layers. These include message, application, and processing layers. The message layer contains incoming and outgoing messages. Incoming messages are SOAP messages forwarded by the client to the server. On the other hand, the outgoing SOAP response messages by the server. The application layer is the actual service implementation. And the processing layer is divided into five different phases. These are: (1) de-serialization, (2) transformation, (3) disassembling, (4) assembling, and (5) serialization.

At the de-serialization phase, the incoming SOAP message is validated for conformation to XML, and parsed against external SOAP schema. After parsing, the message is transformed into domain-specific XML representation. This process involves removing the envelope for the SOAP message received. In other words, it implies gathering relevant information from the SOAP message received. Furthermore, the domain-specific XML representation is disassembled into in-memory tree that can be modified by the service implementation (i.e. list of arguments required for the execution of the service implementation). The list of arguments obtained from the message received is passed to the application layer as arguments. At the application layer, the business logic embedded in the target method is applied to the arguments and response is generated as a result of the method execution. The response obtained from the application layer is assembled into XML at phase (4) in figure 3 above. The output of phase 4 is further transformed to SOAP by inserting the output into a SOAP envelope. Finally, the SOAP envelope generated at phase 2 is validated against external SOAP schema. This is essential in order to ensure that the response is a valid SOAP message that can be processed at the client side.

Implementation

The prototype implementation of the model presented in section 3 has been realized with Java API for XML Web Services (JAX-WS) available in the NetBeans Integrated Development Environment (IDE). JAX-WS allows creation and consumption of web services. In the model presented, there are three major components. These are resource broker, server, and client. The detail implementation exercise of these components is discussed as follow. We implemented the resource broker (i.e. specifications of UDDI project) as a web service. The resource broker implemented provided interfaces for publication of information about service implementation by servers and inquiry of published service information by clients. Furthermore, we generated WSDL file for the resource broker and make it available on the home page of the web service for accessibility of client and server systems in grid community. Hence, we deployed the resource broker to the local server provided by the IDE. By deploying the resource broker, we make available the interfaces of the web service for consumption. In the same vein, we implemented the server in the following manner. First and foremost, the server system compiled the WSDL file provided at the home page of the resource broker. The compilation generates the proxy for communicating with the broker. Any system in the presented grid environment can play roles of server and client simultaneously. For this reason, the WSDL file compiled by the server system makes open interfaces for publication and inquiry. Furthermore, we developed a calculator web service as a service implementation to be consumed in grid. The information for describing and categorizing the calculator web service developed is documented as an entity and published into the resource broker via a publication API. Finally, we implemented the client in the grid as follow. Similar to the server system implemented, the client also compiled the WSDL file provided at the home page of the resource broker. The client can also play the roles of client and server simultaneously. After the compilation of the broker's WSDL file, the stubs or proxies necessary for communication with the broker are created. Therefore, we search the calculator web service using various categorization systems. The essence of this is to gather the information required demanded for binding. We then compile the WSDL of the calculator web service to generate client-side stubs. We bind the calculator web service via the stubs generated.

Conclusion and future thoughts

The unprecedented growth of data and information in a wide range of knowledge sectors [2,16] is an indication for the need of efficient computer resources to store, analyze and process these data in order to justify their existence and maximize their use. Grid computing has emerged as a standardized computational infrastructure for such demands. Its emergence is in line with its ability to integrate heterogeneous computer resources across the globe in a manner similar to the internet. However, pooling grid's resources together is a complex task. The major complexity arises from the heterogeneous nature of the underlying software and hardware resources forming it. In order to foster the adoption of grid in various sectors, it is essential to develop a framework for integrating the heterogeneous software applications in grid. The literatures reviewed include CORBA, DCOM, Globe, and JRMI as solutions that can be adopted for integrating grid resources. However, platform and programming language specific natures of these solutions render them unusable in grid computing. We have designed a model for integrating software applications in grid irrespective of the issues of heterogeneity characterizing the existing solutions. The model designed is based on the standards and specifications of web services. A prototype implementation of the designed model has been developed with JAX-WS. As grid becomes the most appropriate solution to solving grand challenge problems [2,17], we expect our framework to support some mechanisms. These mechanisms include: a resource matching mechanism for matching available resources to clients' requests; a multi-agent intelligent system to replicate service information published within brokers in the registry; an efficient fault-tolerant mechanism to handle faults during job execution; and a security mechanism for authenticating servers and clients.

REFERENCES

[1] Rajkumar Buyya, Chee Shin Yeoa, Srikumar Venugopala, James Broberg, Ivona Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Computer Systems 2008:25 599-616.

[2] Dubitzky Werner. Data Mining Techniques in Grid Computing Environment. West Sussex: John Wiley and Sons; 2008.

[3] Stal M. Web Services: Beyond Component-Based Computing. Communications of the ACM 2002:45(10):71–76.

[4] Tanenbaum AS, van Steen M. Distributed Systems Principles and Paradigms. NJ: Prentice-Hall; 2002.

[5] Object Management Group. The Common Object Request Broker: Architecture and Specification, Revision 2.4.2. Framingham: Object Management Group; 2001.

[6] Platt D. The Essence of COM and ActiveX: A programmers Workbook. NJ: Prentice Hall; 1998.

[7] Homburg P,van Doorn L, van Steen M, Tanenbaum AS, de Jonge W. An Object Model for Flexible Distributed Systems. In *Proceedings 1st Annual ASCI Conference 1995:* 69-78.

[8] T. Java RMI: Remote Method Invocation. IDG Books Worldwide Inc.; 1998.

[9] Christensen, E., Curbera, F., Meredith, G., Weerawarana, S. Web Services Description Language (WSDL) 1.1. W3C Note, Available at http://www.w3.org/TR/WSDL, 2001.

[10] Dough T, James S, Pavel K. Programming Web services with SOAP. O'Rielly; 2001.

[11] Box E, Ehnebuske D, Kakivaya G, Layman A, Mendelsohn N, Nielsen H, Thatte S, Winer D. Simple Object Access Protocol (SOAP) 1.1. Available at http://www.w3.org/TR/SOAP; 2002.

[12] Bellwood T, Clement L, Ehnebuske D, Hately A, Hondo M, Husband Y, Jaruszewski K, Lee S, Mckey B, Munter J, Reigen C. UDDI Version 3.0 Technical Report. Available at http://uddi.org/pubs/uddi-v3.00-published-20020719.htm; 2002.

[13] McGovern J, Tyagi M, Stevens M, Matthew S. Java Web Services Architecture. Morgan Kaufmann Publishers; 2003.

[14] Fielding R, Gettys J, Mogul J, Frystyk H, Masinter L, Leach P, Berners-Lee T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616; 1999.

[15] Eckstein R, Casabianca M**.** XML Pocket Reference. 2nd Edition. O'Rielly; 2001.

[16] Wright A. Glut: Mastering Information through the Ages. Washington D.C.:Henry; 2007.

[17] Wah, B. Report on workshop of high performance computing and communications for grand challenge applications: computer vision, speech and natural language processing, artificial intelligence. IEEE Transactions on Knowledge and Data Engineering 1993:5 (1): 138-154.

****