Proceedings of the 3rd International Conference on Grid and Cooperative Computing, Springer-Verlag Lecture Notes in Computer Science Series, vol. 3251, pp. 17-25, China, October 21-24, 2004.

State Management Issues and Grid Services

Y. Xie[1] and Y.M. Teo[1,2]
[1]*Singapore-Massachusetts Institute of Technology Alliance, Singapore 117576*
[2]*Department of Computer Science, National University of Singapore, Singapore 117543*
email: xieyong@mit.edu

Abstract

Defining the ways for components around the world to collaborate with each other to execute applications over the internet is one of the biggest challenges for computer scientists and engineers. Recent development on Web/Grid services have achieved some success and showed very promising future. However, there still exist several open issues, such as state management. In this paper, we analyze the requirements for state management. We compare different ways of achieving state management in *Web Service*, *Grid Service* and the recent *Web Service Resource Framework*, and articulate the costs and benefits of each approach in terms of fault tolerance, communication cost, etc. We propose a new approach for grid service state management. A prototype is implemented using the Globus Toolkit as a proof of concept.

1. Introduction

Since the invention of the World Wide Web, one of the biggest challenges for computer scientists is to define the ways how components collaborate to carry out applications distributed over the internet. Generations of software systems have been proposed, such as CORBA, Java RMI and so on, but none prevailed.

The concept of Web Service is proposed in the year 2000, together with a group of technologies, such as Simple Object Access Protocol (SOAP) [12], Web Services Description Language (WSDL) [18], and Universal, Description, Discovery and Integration (UDDI) [2]. The W3C web services architecture working group defines web service as "*a software system designed to support interoperable machine-to-machine interaction over a network, and it has an interface described in a machine-processable format (specially WSDL), and other systems interact with the web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other web-related standards*" [11]. There has been some success in deploying Web Service in the industry, such as IBM's WebSphere, BEA's WebLogic, and Microsoft's .NET platform.

A group of researchers lead by Ian Foster introduced the concept of "Grid" computing as a coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations [3]. Recent works [4, 6] on standardizing of Grid computing architecture focus on the idea of *Grid Services*, which is defined by the Open Grid Service Architecture (OGSA) [9], and is specified by Open Grid Service Infrastructure (OGSI) [15]. The Grid service extends the Web service by introducing the concept of a service factory to differentiate service instances, so that states can be encapsulated inside the instance. But this approach has not been widely accepted in the industry yet, and there are some detailed issues in the implementation. For example, in the Globus Toolkit [4], deploying new Grid services to the hosting environment requires the restart of the service container, which causes all ongoing services and persistent services to be stopped.

Most recently, there has been some collaboration between the Web service community (mainly industry) and the Grid service community (mainly academia) in the evolution of grid standards: Web Service Resource Framework (WSRF) [14], which separates the state management from the stateless Web service.

In this paper, we first define services' state, and analyze the requirements for state management. We compare different models and state management such as in Web service, Grid service (OGSI) and WSRF, and analyze the costs and benefits in terms of fault tolerance, scalability, etc. We propose a new approach for state management and implemented a prototype demonstrate the functionality in managing states using the Globus Toolkit Version 3.

The rest of the paper is organized as follows. Section 2 describes the fundamental concepts in the Web/Grid service. We also articulated the requirements of state management in the lifecycle of services and service instances. In Section 3, we survey existing approaches in modeling and managing states, and compare them in terms of targeted applications, fault tolerance, communication cost, etc. We then propose a new approach in Section 4 and compare it with existing approaches. Section 5 presents the prototype we

implemented. Our concluding remarks are in Section 6.

2. Fundamentals

In Web/Grid service, there are two main entities, namely *service requestor (client)* and *service provider*. A *service requestor* or *client* is the software component that consumes a Web/Grid service supplied by the *service provider*. . Figure 1 shows the lifecycle of web/grid services. After the service provider deploys the service, a service requestor can create an instance (session) of the service and start execution of the application. The instance/session of service can be destroyed either by a destroy request from the requestor or by the expiry of its lifetime. The service provider is in charge of maintaining the service once it is deployed. Undeploying takes it out of service and ends the service lifecycle. In the event of failure, recovery mechanisms are needed for both the service provider and service requestor for restoring state information, etc.

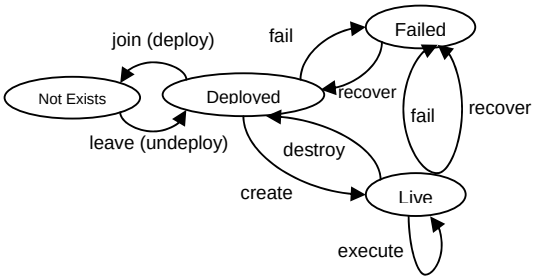

Figure 1. Web/Grid Service Lifecycle

2.1 State

There have been lots of discussion on the "state" and related "stateless/stateful web services", but the definition of state is still elusive, which is one of the major causes of the confusion in today's Web/Grid service community regarding state management. In fact, the state of a web service is very general, and it comprises of one or more of the following:

- *Service's internal data or attributes which are needed to persist across multiple invocations*. In the example of a simple calculator example 2, the state of the calculator is the internal integer's value which is being used for addition and subtraction in every invocation of the service. Details of such example can be found in the Math service example in [16].
- *Web service's internal data or attributes which are needed to maintain for multiple clients*. This is typically related to the clients' account information, like client's bank account number in a banking web services. Such kind of state information can also be found in the calculator's example in [7].
- *Web service's internal data or attributes which are needed to deal with the dependencies among multiple services*. Web services can be standalone services, as well as a composite of several other services, which involves inter-dependencies among services. Information such as other services' address, status, and version numbers are needed to be kept persistent.
- *Service's status at system level*. Information such as service's life cycle, system workload, is needed for monitoring and maintenance purposes. These information are not related to the business or application logic, but it is important for management purposes.

These four categories can be further divided into two main groups based on their relationship with the service requestor and provider. The first two categories are service requestor specific (service instance specific), while the last two are mainly controlled by the service provider (service specific). Based on the definition of state, we found that existing discussions of stateless/stateful service [7, 14, 16, 19] only address certain aspects of the state, but not all. In general, most of the web services today are stateful, meaning that state information is kept and used during the execution of applications.

2.2 Requirements for State Management

State management deals with how state information is stored, retrieved and maintained. State management is required throughout the life cycle of services.

- *Join (deploy)*: when a new service is deployed, other services should not be affected.
- *Create*: when an instance of a grid service is created, state information needs to be initialized and linked to the software component that provides the function of the service.
- *Execute*: during the consuming of the service, state information are being accessed, updated and stored into other system components if necessary.
- *Destroy*: Service requestor should have the right to choose when to destroy the state information either explicitly by a destroy message or implicitly by previously negotiated lifetime of the service instance, or to renew the lifetime during execution.
- *Leave (undeploy)*: when an existing service is undeployed, other services should not be affected.

- *Fail (instance failure and service failure)*: when the execution of a service instance fails due to software bugs, I/O exceptions, etc, other service instances should continue to function so long as there is no dependency between them.
- *Recover:* recovery mechanisms are needed to restore states after failure.

Moreover, in order for a state management framework to work well on the Internet, the following properties are crucial as well:

- *Naming*: as the state information may be shared by other related services, a unique identifier is required.
- *Scalability*: as the web service applications become more and more complex, any state management framework should be scalable to support the increase in both the number and the size of the states.
- *Communication costs (transferring of states)*: as web service applications could involve communications across the entire Internet, the latency in communication could have a significant impact on the performance of the applications.
- *Heterogeneity (state models and management schemes)*: as difference web services could require different ways to model and manage states, the state management framework should be generic in this regard.
- *Security (state repository and access to state)*: as different service providers could be under different administrative domains, and the client could have their specific requirements on where the state is stored, and how states are being accessed. The state management framework should support states' security while preserving the clients' rights and flexibility in controlling their applications' states.

3. Analysis

In this section, we analyze the existing approaches for modeling and managing state information, and compare them in terms of target applications, state repository (where state is stored), communication cost, scalability, fault tolerance and so on. The comparisons in this section are summarized in Table 1.

3.1 Web Service

Web service has been widely regarded as *stateless* and *non-transient*. "*Stateless*" web service means that it cannot remember the results from one invocation to another. "*Non-transient*" means that they outlive all their clients. After one client finishes a Web service, all the information the Web service is remembering could be accessed by the next clients. In fact, while one client is using the Web Service, another client could access the Web Service and potentially affect the first client's operations.

However, some Web service vendors actually work around the "stateless" problem and proposed the idea of "stateless implementation, stateful interfaces" [8]. As a consequence, any state needed for a given message-exchange execution must be provided either explicitly within the request message to recreate the context at the service provider side, or maintained implicitly at some other system components such as database for web service to retrieve. Typical implementation requires the use of client side HTTP cookie/session and/or a dispatcher-like software component at the Web service site to decide which internal service component to invoke, and where the saved state can be retrieved.

So far, Web services are targeted mainly at industrial applications, which are usually short-lived and require little state with simple data structure or even none [7]. For a given application, states are typically stored at a fixed location, either at requestor side or Web service site. If the requestor is responsible for the state, then the requestor's own server is used as a state repository for keeping its own state. Alternatively, the server determines the state repository such as a database or file system as shown in Figure 2.

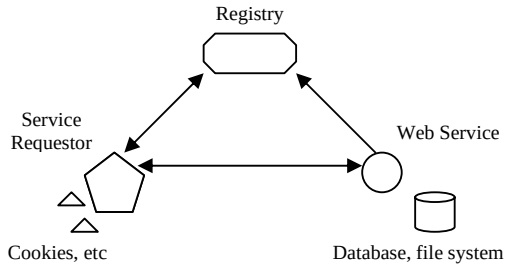

Figure 2. Web Service Model

With the stateless implementation of Web service, the join of a new service should not affect existing services. After deploying the Web service and registering itself with the UDDI registry, clients can discover its URI by contacting UDDI [2]. In a typical stateless Web services, the communication overhead is just the round trip cost between the requestor and the service provider. The cost for transferring states depends on where the state information is kept. If the state information is with the client, the cost is dependent on the number and size of the parameters in the remote invocation of the services. When the state information is being controlled by the service provider, the communication cost between provider and the requestor will not be affected significantly by the change in the number and size of states, as the state information are stored in the service provider's

database or file system, but a central database or file system would still be a potential bottleneck, because there are additional costs in accessing, updating, storing and maintaining the state information. A stateless Web service can be restarted after failure without concerning its history of prior interactions. Even with the stateful interface, as state information is being communicated through invocation of the service or accessing of some other system components, failure of the service should not cause any loss of state information. The leaving of a service is similar to the joining process, which does not affect other services.

If state information is maintained by the requestor, the state is secure in the sense that the requestor has full control. However, if the state information is stored at the service provider side, extra work by the provider is needed to ensure the security, such as checking of id and password. Both ways of maintaining state information would limit the scalability for the requestor and service provider. The requestor could be overloaded with the amount of state information it needs to keep. Similarly, the service provider may face the same scalability problem if the number of requestor, the number of invocation per requestor, the size of state information all increase. To our knowledge, so far, there is no standard naming scheme for states to support sharing of states.

3.2 Grid Service

Several standards for Grid service have been developed such as OGSA [9] and OGSI [10]. Based on the Web services technology, OGSA defines a uniform exposed service semantics (the *Grid service*), standard mechanisms for creating, naming, and discovering transient Grid service instances, and it also provides location transparency and multiple protocol bindings for service instances, and supports integration with underlying native platform facilities. OGSI is the technical specification of extensions and specializations to the Web Services technology for Grid deployment, as required by OGSA. Grid service is considered as an extension of web service by providing state management and so on. The Globus Toolkit 3 (GT3) [6] is a reference implementation of the OGSI specification, and has become an emerging standard for Grid middleware.

Grid Services overcome the *stateless* and *non-transient* limitations of web services by using a *factory* approach. Instead of having a stateless service shared by all users, a Grid service factory is used to create multiple instances of the Grid service. One Grid service instance could be shared by multiple requestors and a requestor could have access to multiple Grid service instances. Moreover, the Grid service instances are transient as they have a limited lifetime and can be destroyed explicitly by requestor after use [1]. The lifetime is specified at the creation of instance, and can be renewed later. When the lifetime is expired, the service instance will self destruct without the need for a synchronous destroy request.

The Grid service focuses on the long-lived applications with many states implemented using complex data structure. States are maintained separately by the service instances at the service provider site, and such kind of encapsulation of states inside a service instance provides certain level of security.

When a new service joins, it is deployed on the service provider's host, and registered with the runtime environment. In GT3's implementation, this requires a restart of the service container, which will affect all existing services in the same container. The service provider publishes its service factory information to a global registry (1) for requestors to look up (2). Then the requestor can send a request to the service provider's service factory to create a new instance of service (3). After the factory creates the service instance (4), it returns the instance's URI called Grid Service Handle (GSH) to the requestor (5). After that, the requestor can interact with the instance using the GSH obtained (6). Detailed steps are shown in Figure 3.

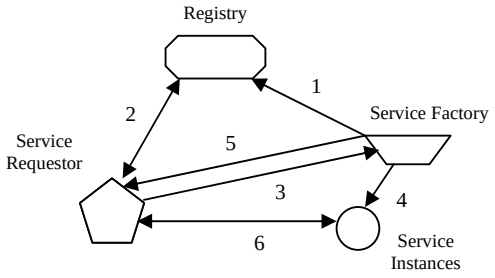

Figure 3. Grid Service Model

As the state information that is associated with the instance itself, the communication cost in Grid service is smaller comparing to the Web service case. However, all state information will be lost together with the instance when the service fails. The leaving of a service is similar to the joining process, which involves update to the runtime environment.. In the case of Globus Toolkit (GT3), this involves restart of the service container. Naming of the state information can be achieved by using the service instance's Grid Service Handle (GSH), but it is not supported in GT3. Although the OGSI specifies that the service instance can be remotely created on the Grid, such feature is not supported in the current version of Globus Toolkit, which would limit its scalability.

3.3 WS-Resource Framework (WSRF)

The WS-Resource framework was proposed as a refactoring and evolution of OGSI aiming at exploiting new Web services standards by a team composed of major players from both Grid service and Web service communities [14]. It is a set of Web services specifications that define a rendering of the WS-Resource approach in terms of specific message exchanges and related XML definitions. A WS-Resource [8] is defined to have a specific set of state data expressible as an XML document, have a well-defined lifecycle, and be known to, and acted upon, by one or more Web services.

To create a new stateful service, besides the stateless Web service providing the function, a WS-Resource factory, which is a Web service to create the instance of a stateful resource, is also required. This factory can be the stateless Web service itself, as long as it is capable of bringing a WS-Resource into existence. Similar to the grid service case, WSRF also requires a registry for provide to publish the service and WS-Resource factory information (1), and for requestor to lookup those information (2). Upon a service request, an endpoint reference [7, 17] is returned to the requestor. The creation of an endpoint reference includes the following steps [8]: (3) requestor sends a request to the WS-Resource factory; (4) the resource factory creates a new stateful resource, assigns the resource a unique identity (resource id) within the service context, creates the association between the resource and its corresponding Web service and (5) returns an endpoint reference to the requestor. The steps are shown in Figure 4. The endpoint reference contains two important components: the reference to the stateless Web service, and an XML serialization of a stateful resource identifier. The service requestor would use the endpoint reference to send messages to the identified Web service. The Web service then uses the identifier in the endpoint reference to access/modify the stateful resource.

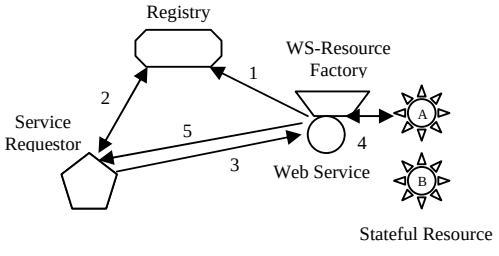

Figure 4 WS-Resource Framework Model

A WS-Resource can be destructed in two ways: (immediate destruction) client explicitly sends a destroy request message to the resource; (scheduled destruction) when the lifetime of the resource defined at creation expires, the WS-Resource is self destructed. Similar to Grid service, this lifetime can be renewed as well.

WSRF aims to fulfill the requirements from both industry and academic applications in an interoperable way [14]. As both the Web service and the WS-Resource factory are stateless Web services, the joining and leaving of a service is the same as it in Web services, which will not affect other services. When a service is being consumed, the stateful resource is accessed through the resource id provided by the requestor. The additional communication cost depends on how the stateful resource is implemented, such as using database, file systems or EJB, etc. As the resources can exist independently of the service, failure of Web services will not cause loss of the state information. The uniqueness of the state id can be achieved by using the Web service end point reference [14].

Access control to resources is required to enhance the security. However, the detail is not specified in the WSRF draft. Moreover, as the WS-Resource factory is specific to a given web service, all states are managed by the provider. As a result, the service provider must be trusted, and it should not be compromised. The tight coupling between the factory and service may lead to some scalability issues. For example, there is limit to the number of states that a service provider can handle. It is not clear whether the requestor has the flexibility to change the policy of state management dynamically for reasons like the requestor no longer trusts the service provider to store its states.

4. Proposed Grid Service Model

From detailed analysis, we note that although WSRF separates state management from the stateless web service and provided a mechanism for state management, it still has some limitations. Firstly, the service-specific WS-Resource factory and state management increases the complexity of developing a new service. The tight-coupling between the resource factory and the service restricts the scalability. More importantly, in the WSRF, it is the service provider who decides where to store the stateful resource and where the state management is carried out, which may introduce security problems, as the requestor and provider may have different security policies. For example, in a critical application involving a company's confidential information, the requestor of a web service does not want to expose the state information to others. In other words, although the service itself is trusted by the requestor, the security policy (like access control) of the service provider under a different administrative domain may not be trusted. Moreover, the flexibility of choosing the location of the state repository is not supported in WSRF either.

		Web Service	OGSI Grid Service	WSRF	Proposed Model
Target Application		Industry, simple state, short-lived	Academic, complex state, long-lived	Industry and academic	Industry and academic
State management schemes	**State Repository**	Client or Server	Service Instance	WS-Resource	Stateful Resource
	Service Join	No effect on others	Restart Container	No effect on others	No effect on others
	Service Failure	No loss of State	Loss of State	No loss of State	No loss of State
	Service Leave	No effect on others	Restart Container	No effect on others	No effect on others
Other issues	**Communication Cost**	Client – service, depending on where state is stored	Client - Service	Client – Factory, Client – Service, Service – WS-Resource	Client – Service (for initialization), Client – State Management Service, Client – Service (for service interaction), Service – State Management Service – Resource
	State Security	• Non-transient for stateless service • State on client: secure • State on server: require extra work from dispatcher	Encapsulated in service instance: secure	secure, given access control to resources and server is trusted by requestor; otherwise, not secure as server can access state resource	Client controls where to store state information: secure with access control
	Scalability of State Information	Both ways of maintaining state information have restriction on scalability.	Remote deploy service instance: scalable (GT3 does not support)	Tight coupling between factory and service, only allow one copy of factory: not scalable	Loose coupling between state management service and web service, allow multiple copies of state management service: scalable
	Naming of states	Not supported	Using GSH	WS end point reference	Many ways: e.g. combine service id, client id, and application id.

Table 1. Comparison of State Management Schemes

To overcome these limitations, we propose a new state management framework based on the WSRF. In our framework, we extend the idea of "separation of state management from Web service" by introducing the concept of a generic state management service. This service takes in the initial state in the form of a XML document, stores it as a stateful resource, and returns the URI of the resource to the requestor. The generality makes the state management and its service loosely-coupled by setting the identifier of the stateful resource to be universally unique. Different implementation of the generic state management service could support different mechanisms for handling states such as security policy. Also interoperability can be achieved by standardizing the interface for state management service to interact with the requestor, service provider and other state management services as well.

The process of using the Web service in our framework is illustrated in Figure 5, and two steps are involved, namely the initialization step and the interaction step. The initialization step is to create the initial state resource: (1) a requestor sends the request to the desired Web service; (2) the Web service returns the initial state to the requestor; (3) the requestor combines with the initial state information provided by its own, then contacts the state management service and passes the initial state with an optional parameter to specify the type of stateful resource (e.g. XML file); (4) the State Management Service stores the initial information as a stateful resource; (5) URI of the stateful resource is returned to the service requestor. The interaction step involves: (1) the Web service takes in the URI of the stateful resource from the requestor, (2) – (4) executes the service by accessing/updating the stateful resource though the state management service, (5) returns the service result if applicable.

Besides the reduction of work for service developers, this framework gives the control of state management to the service requestor by allowing the requestor the ability to deploy the state management service at the desired place (such as requestors' local host). Inter-dependencies of state information can be handled by the interactions between state management services, but the detailed design is out of the scope of this paper. In addition, scalability is enhanced by the flexible deployment of the state management service. Unique naming of the states can be achieved by combining the client id, service id and application id similar to WSRF.

As compared with WSRF, an additional round trip communication is needed in our framework between the requestor and the Web service at the initialization step. Moreover, the access of state information is done through the interaction between the web service and the state management service. Though additional communication overhead is incurred, this cost can be reduced by utilizing locality for deploying the state management service. (Analysis of the proposed framework is summarized in the Table 1).

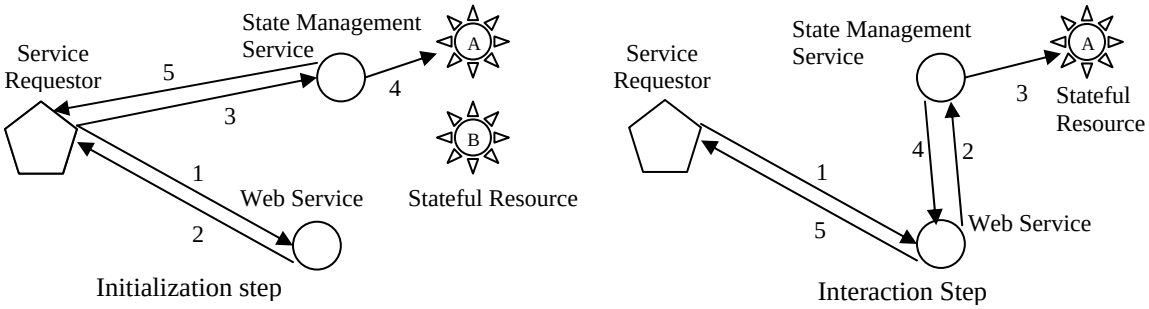

Figure 5. Proposed Grid Service Framework

```
public interface State {
    public String create(String doc, int cid);
    public void set (String uri, String doc);
    public String get(String uri);
}
```

(a) Interface for the state management service.

```
public interface Calc {
    public String start();
    public int add(String uri, int b);
    public int subtract(String uri, int b);
    public int multiply(String uri, int b);
    public int divide(String uri, int b);
}
```

(b) Interface of the sample calculator web service.

Figure 6. Interfaces for Web Services in the Prototype

5. Implementation

A prototype is implemented to illustrate our idea of separating state management from web service by using a generic state management service. We have developed two persistent services (Web Services) using the Globus Toolkit version 3. One is a calculator service (*CalcService*) providing the computation functionalities and the other is a general state management service (*StateService*). The interfaces for these two services are shown in Figure 6. In this prototype, simple state information is maintained as the current computation result.

A client program is implemented to use the *CalcService*. It first calls the '*start*' method in *Calc* to get the initial state (e.g. 1), and then invokes the '*create*' operator in *StateService*, passing the state and its client id. The *StateService* creates a XML file holding the state information and returns its URI. Here we model the URI as: *host name* + *storage directory* + *'state'* + *client-id* + *'.xml'* to ensure the uniqueness. Then the client is able to call the computation functions provided by *CalcService*, such as '*add*', by providing the URI of the state information. The information can be retrieved or updated by *CalcService* through the '*get*' and '*set*' operations in *StateService* respectively.

6. Conclusion and Future Work

In this paper, we have clearly defined the state of services, and articulated the requirements for state management. We analyzed different ways of modeling and managing states in Web services, Grid services and WS-Resource Framework, and compared them in terms of target applications, fault tolerance, etc. We proposed a new framework for state management, and compared it with existing state management schemes. We have also described our prototype based on the Globus Toolkit 3.

More work needs to be done to reduce the additional communication cost in our framework such as to utilize locality of the state management service and the state repository. Other details such as standardizing the naming or addressing of the stateful resource and access control can be further improved in future as well.

Acknowledgment
The authors like to express their thanks to Ai Ting and Wang Caixia for discussion and comments in an earlier draft of this paper.

References

[1] K. Czajkowski, etc. From Open Grid Services Infrastructure to WS-Resource Framework: Refactoring & Evolution.

[2] (discovery) Discovery Universal, Description and Integration (UDDI). http://www.uddi.org/.

[3] I. Foster. "The anatomy of the Grid: Enabling scalable virtual organizations". Lecture Notes in Computer Science, 2150:1–12, 2001.

[4] I. Foster and C. Kesselman. "Globus: A metacomputing infrastructure toolkit". The International Journal of Supercomputer Applications and High Performance Computing, 11(2):115–128, Summer 1997.

[5] I. Foster, C. Kesselman, J. Nick, and S. 15. "The physiology of the grid: An open grid services architecture for distributed systems integration", 2002.

[6] Globus Toolkit version 3. http://www.globus.org/.

[7] Implement and access stateful Web services using WebSphere Studio. http://www-106.ibm.com/developerworks/webservices/library/ws-statefulws.html.

[8] Modeling Stateful Resources with Web Services. http://www.globus.org/wsrf/.

[9] Open Grid Services Architecture. http://www.globus.org/ogsa/.

[10] Open Grid Services Infrastructure. https://forge.gridforum.org/projects/ogsi-wg.

[11] Public draft The W3C Web Services Architecture working group. http://www.w3.org/tr/2003/wd-wsarch-20030808/.

[12] Simple Object Access Protocol (SOAP). http://www.w3.org/tr/soap.

[13] T. 13, D. Ferguson, B. Lovering, J. Shewchuk. Reliable Transacted Web Services. http://www-106.ibm.com/developerworks/webservices/library/ws-securtrans/

[14] The WS-Resource Framework. http://www.globus.org/wsrf/.

[15] S. 15, K. Czajkowski, I. Foster, J. Rey, F. Steve, and G. Carl. Grid service specification, 2002.

[16] Tutorial of Globus Toolkit version 3 (GT3). http://www.casa-sotomayor.net/gt3-tutorial/.

[17] Web Services Addressing. ftp://www6.software.ibm.com/software/developer/library/wsadd200403.pdf.

[18] Web Services Description Language (WSDL). http://www.w3.org/tr/wsdl.

[19] Web Services are not Distributed Objects. http://weblogs.cs.cornell.edu/allthingsdistributed/archives/000343.html.