Grid Business Process: Case Study

Asif Akram[1], Sanjay Chaudhary[2], Prateek Jain[2], Zakir Laliwala[2] and Rob Allan[1]
[1]CCLRC Daresbury Laboratory, Daresbury, UK
{a.akram, r.j.allan}@dl.ac.uk
[2]Dhirubhai Ambani Institute of Information and Communication Technology,
Gandhinagar 382 007, Gujarat, India.
{ sanjay_chaudhary, prateek_jain, zakir_laliwala }@daiict.ac.in

1. Introduction

The emerging Grid paradigm (Joseph & Fellenstein, 2003) can be described as "coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations". Grid design should ensure cross-platform interoperability, discovery, re-usability and integration of systems in a heterogeneous context; the key requirement of the Business Process. Business processes of large enterprise systems interact with various stack holders i.e. business partners and customers to integrate distributed heterogeneous services. These long running and dynamic business processes require support for state persistence, monitoring, transaction management, and event notifications. Existing grid computing technologies take advantage of underused computing capacity and can be applied to solve business problems and provide IT-level infrastructure to support business applications. The business grid will provide a virtualized infrastructure to support the transparent use and sharing of business functions on demand in an orchestrated manner. It must also be able to build a business solution from multiple grid services and resources (Liang-Jie, Haifei & Herman, 2004). The scope of Business Process Grids covers business process provisioning and outsourcing; integration; collaboration; monitoring; and management infrastructure.

This chapter presents a 'Case Study' based on the distributed market. The requirements of this Grid Business Process are more demanding than any typical business process deployed within a single organization or enterprise. Recently different specifications built on top of Web service standards have originated from the Grid paradigm to address limitations of stateless Web services. These emerging specifications are evaluated in the first part of the chapter to capture requirements of a dynamic business process i.e. Business Process Grid. In second part of the chapter, a case study with different use cases is presented to simulate various scenarios. The abstract discussion and requirements of the case study is followed by the actual implementation. The implementation is meant for the proof-of-concept rather than fully functional application. The implementation has different possibilities of improvements which are discussed before the conclusion to give reader future directions.

2. Grid Related Specifications and Standards

Web services architecture lacks support for the state, event and notification, and resource lifecycle management to share and coordinate diverse resources of real life in dynamic 'virtual organizations' (Foster, 2002). Recent convergence between Web services and the Grid computing community (Czajkowski, Ferguson, Foster, & et al., 2004) towards the re-factoring and evolution of Grid standards aimed at aligning OGSI functions with the emerging consensus on Web services architecture (Booth, Haas, McCabe, & et al, 2004). This effort has produced two important sets of specifications: WS-RF (WS-ResourceFramework, 2005) and WS-Notification (WS-Notification, 2005). These specifications essentially retain all the functional capabilities present in OGSI, and at the same time built on broadly adopted concepts of Web services.

2.1 Web Services Addressing

The WS-Addressing specification defines a standard for incorporating message addressing information into SOAP (SOAP, 2000) messages. SOAP does not provide a standard way to specify where a message is going, how to return a response, or where to report an error. WS-Addressing introduces two new constructs for Web services vocabulary: *Endpoint References* and *Message Addressing Properties*.

2.1.1 Endpoint Reference

Endpoint References are a new model for describing the Web service destination and service-specific attributes within an address for routing the message to a service or for use by the destination service itself. An endpoint reference is a data structure that is defined to encapsulate all the information required to reach a service endpoint at runtime.

A significant aspect of an endpoint reference is the ability to attach data from any XML namespace via Reference Properties or Reference Parameters. Both of these elements are collections of properties and values used to incorporate elements from different XML namespace into the endpoint reference. The key distinction between a Reference Property and a Reference Parameter is not the format but the intended usage. The reference properties help to identify the resource to be used during service invocation. The reference parameters wrap the information required for successful invocation of the service which is not required to identify the resource.

The following example shows an endpoint reference for a service that simulates the personal address book. The service's URI is specified in the Address element. A reference property indicates the type of the resource i.e. family, friend, colleague etc. A reference parameter specifies the information required i.e. address, home number, mobile number etc.:

```
<wsa:EndpointReference xmlns:wsa="..." xmlns:example="...">
   <wsa:Address>http://example.com/contact</wsa:Address>
   <wsa:ReferenceProperties>
       <example:contactType>Family</example:contactType >
   </wsa:ReferenceProperties>
   <wsa:ReferenceParameters>
       <example:detail>Mobile</example: detail >
   </wsa:ReferenceParameters>
</wsa:EndpointReference>
```

Listing 1. Example of Endpoint Reference

2.1.2 Message Addressing Properties

WS-Addressing introduces a set of message headers providing information about a message by incorporating delivery, reply-to, and fault handler addressing information into a SOAP envelope necessary to support a rich bidirectional and asynchronous interaction. Most of the fields are optional; the only required fields are the 'To' and 'Action' fields, each of which specifies a URI. The 'To' header identify the destination of the message and 'Action' header provides additional information how to process the message. The value of Action URI is related to the WSDL operation to which the message is related. In the absence of Action URI in the WSDL, the Action URI in the EPR is target namespace and the operation called. The following example illustrates a typical SOAP message using WS-Addressing:

```
<S:Envelope xmlns:S="http://www.w3.org/2003/05/soap-envelope"
                xmlns:wsa="http://www.w3.org/2004/12/addressing">
    <S:Header>
        <wsa:MessageID>
            http://......
        </wsa:MessageID>
        <wsa:ReplyTo>
            <wsa:Address>http://client.application</wsa:Address>
```

```
        </wsa:ReplyTo>
        <wsa:FaultTo>
            <wsa:Address>http://logging.application</wsa:Address>
        </wsa:FaultTo>
        <wsa:To>http://service.to.invoke</wsa:To>
        <wsa:Action>http:// method.to.invoke</wsa:Action>
    </S:Header>
    <S:Body>
         <!-- The message body of the SOAP request appears here -->
    </S:Body>
</S:Envelope>
```

Listing 2. SOAP message

When a service receives a message addressed using WS-Addressing, it will also include WS-Addressing headers in the reply message. The *Message ID* of the original message becomes a *RelatesTo* element in the reply's address. At present, the only supported relationship type is "Reply." If a client is sending multiple web services requests and receiving asynchronous responses, possibly over different transports, the *RelatesTo* element provides a standard way to associate incoming replies with their corresponding requests.

2.2 Web Service Resource Framework (WS-RF)

Web Service Resource Framework specifies various aspects related to stateful Web services. It defines message exchange pattern as to how stateful Web services should be created, addressed, and destroyed. (Foster, Czajkowski, & et al, 2005). WS-RF defines conventions within the context of established Web Services standards for managing 'state' so that applications can reliably share changing information, and discover, inspect and interact with stateful resources in a standard and interoperable way. WS-RF is a collection of four specifications, namely WS-ResourceProperties, WS-ResourceLifetime, WS-ServiceGroup, and WS-BaseFaults. It also refers to two related specifications, namely WS-Notification and WS-Addressing.

2.2.1 WS-ResourceProperties (WS-ResourceProperties, 2006)

WS-ResourceProperties describes properties of stateful resources and its association with Web services as the *Implied Resource Pattern*. In the *Implied Resource Pattern*, SOAP messages include a component that identifies a stateful resource to be used in the execution of the message exchange. The composition of a stateful resource and a Web service under the Implied Resource Pattern is termed as a WS-Resource. The specification standardizes the definition of the properties of a WS-Resource as a part of the Web service interface in terms of a resource properties document. The declaration of the WS-Resource's properties represents a view on the resource's state in XML format.

This specification standardized set of message exchanges for the retrieval, modification, update and deletion of the contents of resource properties and supporting subscription for notification when the value of a resource property changes. The set of properties defined in the resource properties document associated with the service interface, defines the constraints on the valid contents of these message exchanges.

2.2.2 WS-ResourceLifetime (WS-ResourceLifetime, 2006)

The lifetime of a WS-Resource is defined as the period between its instantiation and its destruction. The WS-ResourceLifetime specification standardizes the means by which a WS-Resource can be destroyed either immediately or at a scheduled time. The scheduled destruction of the WS-Resource means that a resource may be destroyed after a certain period of time. When that time expires, the WS-Resource may self-destruct without requiring an explicit destroy

request message from a client. The specification also supports extension of the scheduled termination time of a WS-Resource at runtime.

WS-ResourceLifetime defines a standard message exchange by which a service requestor can destroy, query, establish and renew a scheduled termination time for the WS-Resource. To support the standard message exchange pattern WS-ResourceLifetime declares different methods which are normally implemented by the WS-RF engine. The specification also supports the notification to interested parties when resource is destroyed.

2.2.3 WS-ServiceGroup (WS-ServiceGroup, 2006)

The WS-ServiceGroup provides a description of a general-purpose WS-Resource which aggregates information from multiple WS-Resources or Web Services for domain specific purposes. The aggregated information can be uses as a directory in which the descriptive abstracts of the individual WS-Resources and Web Services can be queried to identify useful entries. The WS-ServiceGroup itself is a stateful Web Service that is a collection of other Web Services or WS-Resources and the information that pertains to them.

The specification standardized message exchange for registration or addition of new members, membership rules, duration of the membership, contents advertised by the members and support for notification when new member is added or when details of the existing member are modified. The membership in the ServiceGroup is flexible and it can be either through ServiceGroupRegisteration defined by specification or through any other means. Details of each member in the ServiceGroup are in the form of WS-ResourceProperties; which wraps the EndpointReference and the contents of the member.

Membership in the group can be constrained, controlled through policies. Controlled membership enables requestors to form meaningful queries against the contents of the WS-ServiceGroup. The membership of the service group can be restricted to only those members which implement any particular interfaces or declare any specific WS-Resource model. The ServiceGroup resource property document may contain zero MembershipContentRule child elements. When no MembershipContentRule elements are specified, the members of the ServiceGroup are unconstrained. The element MembershipContentRule in the resource property document of the ServiceGroup has following two attributes:

MemberInterfaces: This attribute is optional and declares the list of interfaces which must be implemented by each “entry” in the ServcieGroup.

ContentElements: This attribute declares the list of WS-ResourceProperties, which must be part of the WS-Resource model for each entry. ContentElements is mandatory attribute in the element MembershipContentRule but can have no value.

<wssg:MembershipContentRule MemberInterface="ns2:X" ContentElements=""/>

<wssg:MembershipContentRule MemberInterfaces="ns3:Y" ContentElements="ns3:RP1 ns3:RP2” />

MembershipContentRule in the first statement expects members to implement “ns2:X” portType; where as second MembershipContentRule expects not only implementation of “ns3:Y” but also exposing two ResourceProperties. Multiple MembershipContentRule elements have the “or” relation which means the members should fulfil at least one the membership criteria completely. A member fulfilling different membership criteria can appear multiple times in the ServiceGroup.

2.2.4 WS-BaseFaults (WS-BaseFaults, 2006)

A typical Web services application often uses interfaces defined by others. Fault management in such an application is more difficult when each interface uses a different convention for representing common information in fault messages. Web services fault messages declared in a

common way improves support for problem determination and fault management. It is also more likely that common tooling can be created to assist in the handling of faults described uniformly. WS-BaseFaults defines an XML Schema type for a base fault, along with rules for how this fault type is used by Web services. It standardizes the way, in which, errors are reported by defining a standard base fault type and procedure for use of this fault type inside WSDL. WS-BaseFault defines different standard elements corresponding to the time when the fault occurred (*Timestamp*), the endpoint of the Web service that generated the fault (*OriginatorReference)*, error code (*ErrorCode*), error description (*Description*), the cause for the fault (*FaultCause*) and any arbitrary information required to rectify the fault.

2.2.4.1 Use of Base Faults in WSDL 1.1

The WS-BaseFaults specification recommends that each custom fault type must extend the base fault. Each distinct type of base fault associated with a WSDL [WSDL 1.1] operation should be listed as a separate fault response in the WSDL operation definition. The extended faults must follow following rules:

1 There must be a distinct XML Schema complexType that extends WS-RF bf:BaseFaultType, which represents this fault's distinct type. This extended fault complexType can contain additional attributes and/or elements.

2 An element must be defined for this distinct fault, whose type is the complexType of the distinct fault as defined in step 1. This is the requirement of Document/literal style Web Services which are compliant to WS-I Basic Profile.

3 A WSDL message must be defined for this distinct fault. This message must have one part. The WSDL part must have an 'element' attribute and this must refer by fully qualified name to the element of this distinct fault as defined in step 2. This is once again the requirement of Document/literal style Web Services which are compliant to WS-I Basic Profile.

4 The WSDL operation must have a fault element for this distinct fault. The value of the WSDL fault element's name attribute should be the same as the NCName of the fault element defined in step 2, although it may ignore this rule (for example to avoid NCName collisions between fault elements defined in different namespaces). The value of the WSDL fault element's message attribute must refer by fully qualified name to the WSDL message element of this distinct fault as defined in step 3.

2.3 WS-Notification

The Event-driven, or Notification-based, interaction pattern is a commonly used pattern for inter-object communications. Different domains provide this support in varying extent; 'Publish/Subscribe' systems provided by Message Oriented Middleware vendors; support for 'Observable/Observer' pattern in the programming languages; "Remote Eventing" in the RMI and CORBA. Due to stateless nature of Web Services; Web Service paradigm has no notion of Notifications; which limited the applicability of Web Services for complicated application development. WS-RF defines conventions for managing 'state' so that applications can reliably share changing information, discover, inspect, and interact with stateful resources in standard and interoperable way; bringing Notification-based interaction pattern in Web Services domain. WS-Notification (WSN) [14] is set of three separate specifications (WS-BaseNotification, WS-BrokeredNotification, and WS-Topics) but its usefulness beyond WS-RF is limited.

2.3.1 WS-BaseNotification (WS-BaseNotification, 2006)

The WS-BaseNotification is the base specification on which all the other specifications in the family of WSN depend. It defines the normative Web services interfaces for two of the important

roles in the notification pattern, namely the NotificationProducer and NotificationConsumer roles. Strictly speaking this specification defines many different roles and any single entity can fulfill the criteria of different roles. This specification includes standard message exchanges to be implemented by service providers that wish to act in these roles, along with operational requirements expected of them. Latest WS-BaseNotification specification supports 'Pull' based notification for resource constrained devices; but yet none of the WS-RF framework supports those recommendations and is not discussed.

2.3.1.1 NotificationProducer

A NotificationProducer is an entity which monitors the state of different resources and detects various types of events. Whenever there is any change in the state of a resource or occurrence of any new event; which may qualify for certain actions the NotificationProducer notify the relevant entities. These entities may be only interested in the changes or may initiate the series of events to accommodate changes.

The NotificationProducer must support any appropriate mechanism that lets a potential Subscriber to discover which resources and events are monitored by a NotificationProducer. These resources and events monitored by the NotificationProducer are called Topics (discussed in section 2.3.2). For this purpose, each NotificationProducer must support resource properties (Listing 3) defined in the specification other than any custom resource properties. Out of these resource properties, the 'TopicSet' is the collection of topics supported by the NotificationProducer expressed, as a single XML element as described in [WS-Topics].

```
<xsd:element ref="wsnt:TopicExpression" minOccurs="0" maxOccurs="unbounded" />
<xsd:element ref="wsnt:FixedTopicSet" minOccurs="0" maxOccurs="1" />
<xsd:element ref="wsnt:TopicExpressionDialect" minOccurs="0" maxOccurs="unbounded" />
<xsd:element ref="wstop:TopicSet" minOccurs="0" maxOccurs="1" />
```

Listing 3. Resource Properties supported by Notification Producer

The WS-BaseNotification specification standardized various message exchanges for NotificationProducer to support subscription for notification, querying the last message, renewal, cancellation, pause and resumption of subscription. These methods heavily rely on Topics declared in the TopicSet.

2.3.1.2 NotificationConsumer

An entity which may have interest in the elements monitored by the NotificationProducer for appropriate actions is called NotificationConsumer. A NotificationConsumer can subscribe to receive notifications directly from a NotificationProducer, supporting only direct and point to point notifications. A NotificationConsumer discovers the NotificationProducer and browse the Topics for subscription (i.e. sending Subscribe request or invoking Subscribe operation). A NotificationConsumer must implement the call back methods to receive the notification. Normally the NotificationConsumer implements *Notify* call back operation; which is invoked by the NotificationProducer. The NotificationConsumer must support one of the two or both NotificationMessage format; or at least should be in position to handle the form of Notification it has requested for the given Subscription.

2.3.1.3 NotificationMessage

The NotificationProducer must also clarify; the supported formats for the notification messages. When NotificationProducer has a notification to distribute, it matches the notification against the subscription list and issues the notification to the subscriber which is registered for the

notification of such event. WS-Notification allows a NotificationProducer to send a NotificationMessage to a NotificationConsumer in one of two ways:

1 The NotificationProducer may simply send the raw NotificationMessage (i.e. the application-specific content) to the NotificationConsumer.

2 The NotificationProducer may send the NotificationMessage data using the Notify message, which means wrapping the application-specific content in the Notify element along with additional information i.e. the source of notificatioin; time; last value etc.

When a Subscriber sends a Subscribe request message, it indicates which form of Notification is required (the raw NotificationMessage, or the Notify Message). The NotificationProducer must observe this Subscription parameter, and use the form that has been requested.

2.3.2 WS-Topics (WS-Topics, 2006)

The 'WS-Topics' defines a mechanism to organize and categorize items of interest for subscription knows as 'topics'. WS-Topics defines an XML model for describing metadata associated with topics, and three topic expression dialects that can be sued as subscription expressions in subscribe request messages. The specification aims at categorizing topics in different categories under which the topics are clubbed.

2.3.2.1 Topics and Topic Namespaces

The WS-Notification specifications allow the use of Topics as a way to organize and categorize a set of Notifications messages that relate to a particular type of information. For example a stock ticker NotificationProducer application might set the Topic of the NotificationMessages it produces to the stock symbol with which the information is associated - for example "stock/BP". The Topics mechanism provides a convenient means by which subscribers can reason about Notifications of interest. A Topic is the concept used to categorize Notifications and their related Notification schemas. These Topics are used as part of the matching process that determines which (if any) subscribing NotificationConsumers should receive a Notification. When Topic generates a Notification, a NotificationPublisher can associate it with one or more Topics.

The mechanism for achieving this collision avoidance is normally determined by the application developer - in one pattern an application developer defines a namespace for use by a related group of applications. This leaves the developer free to use whatever topic structure they see fit within that namespace. To continue the example above the application developer could define one TopicNamespace for notification messages published in French, and a different one for notifications published in English. A subscribing application then specifies the namespace/topic in which they are interested (e.g. "english:stock/BP") to ensure they receive notification messages in the appropriate language. In this way you can use the 'same' topic structure (with different namespaces) to ensure that the application does not receive incompatible notifications. The set of Topics associated with a given XML Namespace is termed a Topic Namespace. Each Topic in a Topic Namespace can have zero or more child Topics, and a child Topic can itself contain further child Topics. A Topic without a parent is termed a root Topic. A particular root Topic and all its descendents form a hierarchy called a Topic Tree.

2.3.2.2 Topic Expression Dialects

Topics are referred to by TopicExpressions. TopicExpression is a query mechanism to reach one particular Topic in unambiguous manner in the Topic Tree. Section 8 of the WS-Topics specification defines example topic expression dialects that are recommended for use by WS-Notification applications. Note that the WS-Notification specifications provide an extensibility mechanism to allow vendors to define their own topic expression dialects if they wish. The

different default topic expression dialects supported by different Application Server are *Simple TopicExpressions, Concrete TopicExpressions, Full TopicExpressions* and *XPath TopicExpression Dialect*.

2.3.2.3 Topic Set

The Topic Set is a collection of Topics supported by a NotificationProducer. Topics from a single Topic Namespace can be referenced in the Topic Sets of many different NotificationProducers. Moreover the Topic Set of a NotificationProducer may contain Topics from several different Topic Namespaces. A NotificationProducer can support an entire Topic Tree, or just a subset of the Topics in a Topic Tree. The set of Topics supported by the NotificationProducer may change over time.

2.3.3 WS-BrokeredNotification (WS-BrokeredNotification, 2006)

The WS-BrokeredNotification specification defines the interface for the NotificationBroker. A NotificationBroker is an intermediary between message Publishers and message Subscribers. A NotificationBroker decouples NotificationProducers and NotificationConsumers and can provide advanced messaging features such as demand-based publishing and load-balancing. A NotificationBroker also allows publication of messages from entities that are not themselves service providers. The NotificationBroker interface specifies standardized message exchange to be implemented by NotificationBroker along with operational requirements expected of service providers and requestors that participate in brokered notifications. A NotificationBroker is capable of subscribing to notifications, either on behalf of a NotificationConsumers, or for the purpose of messaging management. It is also capable disseminating notifications on behalf of Publishers to NotificationConsumers. Thus NotificationBroker aggregates NotificationProducer, NotificationConsumer, and RegisterPublisher interfaces.

3. WS-Resource

The WS-Resource represents the state in a Web services context. This state has atomic components, called *Resource Properties* elements, which can be updated and queried. A set of resource property elements are gathered together into a resource property document: an XML document that can be queried by client applications using XPath or any other query languages. WS-RF supports dynamic insertion and deletion of the resource property elements of a WS-Resource at run time according to the XML Schema of resource property document.

A WS-Resource itself is a distributed object, expressed as an association of an XML document with a defined type attached with the Web service portType in the WSDL. Each WS-Resource has a unique identity and distinguishable handler; which can be serialized in XML format to embed it in the message before sending across the network. Although WS-Resource itself is not attached to any Uniform Resource Locator (URL), it does provide the URL of the Web service that manages it. The unique identity of the resource and the URL of the managing Web service is called an Endpoint Reference (EPR), which adheres to Web Services Addressing. Instances of a resource have a certain lifetime, which can be renewed before they expire as specified by WS-ResourceLifetime specification. They can also be destroyed pre-maturely as required by the application. The lifetime of an instance of a resource is managed by the client itself or any other process interacting as a client, independent of the Web service and its container.

WS-Resources are not bound to a single Web service; in fact multiple Web services can manage and monitor the same WS-Resource instance with different business logic and from a different perspective. Similarly, WS-Resources are not confined to a single organization and multiple organizations may work together on the same WS-Resource, which leads to the concept

of collaboration (Akram). Resource sharing is extensively used for load balancing. Semantically similar or cloned Web services are deployed independently for multiple client access. At run time, appropriate EPR's of the WS-Resource are generated with the same unique Resource identity but with different URLs of managing Web services.

Resources are composed of Resource Properties, which reflect their state. These can vary from simple to complex data types and even reference other Resources. Referencing other Resources through Resource Properties is a powerful concept, which defines and elaborates inter-dependency of the Resources at a lower level.

3.1 Implied Resource pattern

The WSA defines the relationship between Web services and stateful resources, which is the core of the WS-Resource Framework. The WS-RF specifications recommend the use of the Factory/Instance pattern i.e. Implied Resource pattern. The term implied is used because the identity of the resource isn't part of the request message, but rather is specified using the reference properties feature of WSA. The endpoint reference provides means to point to both the Web service and the stateful resource in one convenient XML element.

This factory pattern is well understood in computer science: the notion of an entity that is capable of creating new instances of some component. A Factory service is responsible for creating the resource, assigning it an identity, and creating a WS-Resource qualified endpoint reference to point to it. An Instance service is required to access and manipulate the information contained in the resources according to the business logic. Implied Resource Pattern can be extended in various ways to effectively capture the requirements of the application. Different possible extensions are discussed below.

3.1.1 Factory/Instance Pair Pattern

The Factory/Instance Pair Model is the simplest model, in which for each resource there is a Factory Service to instantiate the resource and corresponding Instance Service to manage the resource. In a typical application different Factory Services are independent of each other and can work in isolation. This is the simplest approach: repeating the similar resource instantiating logic in multiple Factory Services or even the same Factory Service can be deployed multiple times.

In this model the user manually interacts with different Factory Services; which instantiate the appropriate resources and return the corresponding EPR's to the client. The client application contains the logic of deciding when and which resources should be instantiated.

3.1.2 Factory/Instance Collection Pattern

The Factory/Instance Collection Model is an extension of the Factory/Instance Pair Model. The difference being that a single Factory Service instantiates multiple WS-Resources managed by different Instance Services. In any Business Process various entities can be tightly coupled and due to this inter-dependency all of these WS-Resources must co-exist before a client may interact with them successfully.

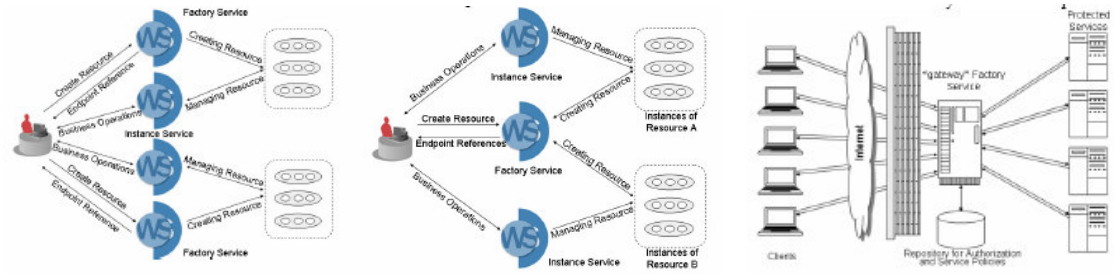
3.1.3 Master-Slave Pattern

In a Grid application with unpredictable request traffic, different security and load balancing measures are required for smooth execution. Business Processes are frequently protected by a firewall. It has to be anticipated that firewall policies will limit direct access from external clients to WS-Resources (i.e. it is most likely that these Resources will be located inside private firewalls, and can only be accessed via known gateway servers). Consequently, an extensible Gateway model is required for accessing these resources. This model mandates that all client

requests are sent to an externally visible Gateway Web Service before being routed through the firewall to the actual requested service. In addition, firewall administrators may implement additional security measures such as IP-recognition between gateway server and service endpoint in the form of Web Services handlers. The client interacts only with the Master Factory Service without knowing the inner details of the application. The Master Factory Service performs authentication and authorization of the client before invoking respective Factory Services (Slaves) which are behind the firewall and restricted by strict access polices.

3.1.4 Hybrid Approach

The best approach for any complicated Business Process is to combine these variations of the Implied Resource Pattern as follows. The client still interacts with a single Factory Service which instantiates all mandatory WS-Resources and returns a single EPR. Subsequent client interactions invoke the 'create' operation of the Factory Service with different 'parameters' with the Factory Service instantiating the corresponding WS-Resources according to those parameters. Optional WS-Resources are supported using a Factory/Instance Pair model due to their limited usage.



(a) (b) (c)

Figure 1. Various forms of Factory/Instance pattern (Akram, 2006)

3.2 Notification Pattern

The Event-driven or Notification-based interaction pattern is a commonly used pattern for inter-object communications. WS-RF and WSN bring notification-based interaction pattern in Web Services domain. This notification pattern can be extended in various ways to meet the application requirements.

3.2.1 Client as Notification Consumer

In this approach client application acts as a Notification Consumer, which is notified of any change in the "state" of subscribed WS-Resource instance. The client processes the notification messages and updates instances of other related Resources through corresponding Instance Service(s). Client application exposes "notify" operation to receive asynchronous notification messages. Client also implements complicated logic of inter-relating dependent WS-Resource instances together, whereas Enterprise Application is simple and independent of notification.

3.2.2 Service as Notification Consumer

At application level, different Resource instances are associated with each other. Due to inter-dependency of WS-Resources, the managing services have interest in the state of other Resource instances. Thus handling the notifications at service level is more appropriate without involving client applications. This is the situation where automatic and quick actions are required. The client does not manage actions, and it does not have a role in the decision-making. Actions required are related to the main functionality of the application and not with a specific client.

3.2.3 Resource as Notification Consumer

The two notification approaches discussed above (Sections 3.2.1 and 3.2.2) have their own limitations and benefits. A third notification model can provide the best of both approaches with an even cleaner design. In this approach WS-Resource itself is a notification consumer, yet may also act as a producer to have one-to-one associations. Each instance of the WS-Resource can subscribe to 'state' changes of specific WS-Resource instances whilst broadcasting notification messages related to its own 'state'. Overall this mechanism gives tighter control on the business logic without interference from the client side.

Implementing a WS-Resource as a notification consumer or a consumer-producer can result in large numbers of messages which can overload the Subscription Manager Service, thus affecting the overall performance of the application. The more inter-related instances, the worse the problem becomes. This model should be applied with caution.

4. Trading scenario

To demonstrate the application of WS-RF specifications in the Business Process Grid, an application to demonstrate Agricultural Marketing Process in India has been developed (Sorathia, Laliwala, & Chaudhary, 2005). In any typical Asian Agro-market, wholesale spot markets and derivative markets are emerging hubs of agricultural marketing business. Agro trade in these markets is heavily influenced by local, socio-economic and cultural characteristics. This leads to variation in the crop prices of same crop in different markets. The producers have little choice to search for the best available price and are forced to sell their products in a local market. Inhibitive transport and storage cost can also play a pivotal role apart from urgency to sell perishable products. Buyers and wholesalers experience difficulties in purchasing desired quality of products at competitive prices. A typical trade in a Business Process Grid can span across different markets located at various places. Privately owned markets, food processing and other related industries are allowed to trade directly with the farmers. Trading in such a competitive market will be more complex than that in the conventional trading scenario constrained to single geographical market.

4.1 Execution of Trade Workflow

A seller or a farmer joins the market place with intention to start a trade of agricultural produce. An authorized market functionary carries out measurement and grading of the produce and collect fees for any optional service such as transportation or storage. Price of the produce can be set by tender bid, auction or any other transparent system. In case of direct sale, a seller is exempted to pay any market fee or commission; whereas in case of indirect sale the market fee is imposed on the seller for using different utility services. Once the agreed upon price is received, it is published for the traders. Only license holders are allowed to carry out any trade in the market area. If trader fails to pay any fees to the Trade Market or fail to pay the agreed-upon price of the purchased good, the system may cancel the license of the trader. This is the situation where management of the whole market is required; independent of the main functionality of the Grid market. If the trade is carried out by license holders in a manner explained above, the bill will be issued and the transaction will be recorded in the Trade Market database. Figure 2 represents a simplified workflow capturing few aspects of a typical trade.

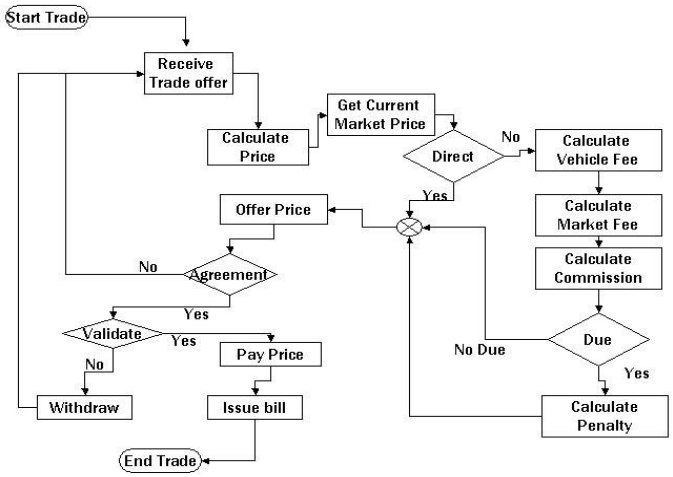


Figure 2. Workflow of Marketing of Agro-produce

5. System design

The Grid computing utilizes the Service Oriented Architecture (SOA) to meet requirements of the Grid Business Process. The goal of SOA is to achieve loosely coupled interoperable integration among scattered and heterogeneous services and clients. Principles of SOA influence the business logic of services by encouraging modular design and component reuse through dynamic discovery of existing services. Web services have been established as a popular technology for design and implementing SOA (Laliwala, Jain, & Chaudhary, 2006) based business process. A Web service has five essential attributes: (Gottschalk, Graham, Kreger, & Snell, 2002) it can be described using a standard service description language, Web Service Description Language (WSDL) (WSDL, 2005); it can be published to a registry of services, Universal Description, Discovery and Integration (UDDI) (UDDI, 2004) registry; it can be discovered by searching the service registry; it can be invoked, usually remotely, through a declared API; and, it can be composed with other Web services. Interaction Diagram of components of Agricultural Marketing System is shown in figure 3.

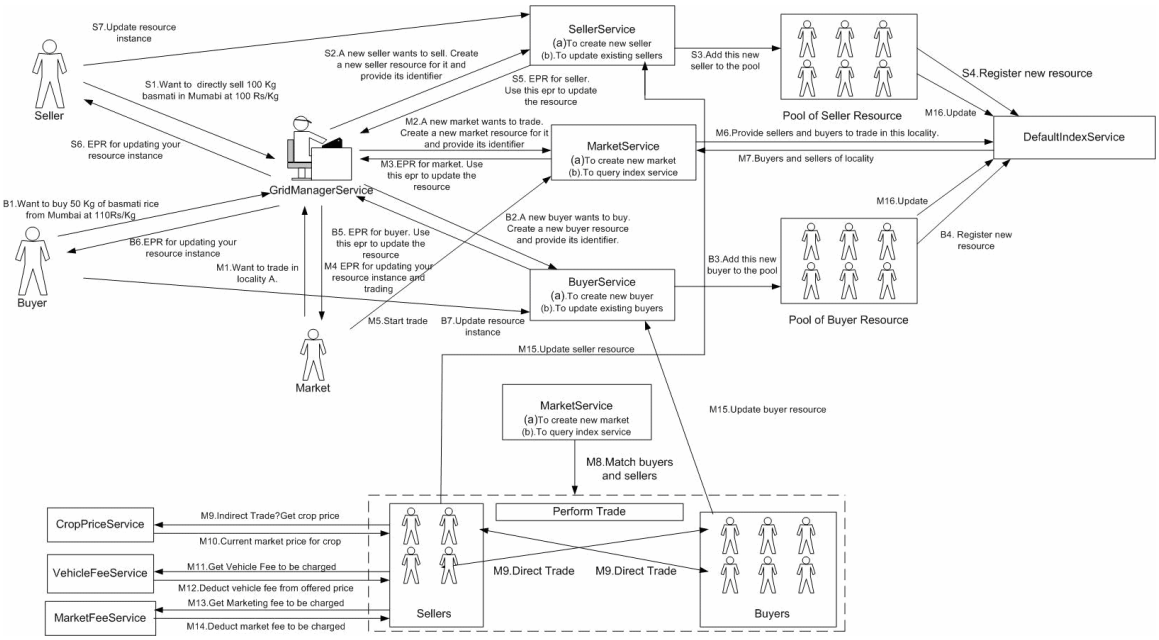


Figure 3. Interaction Diagram of Agricultural Marketing System

5.1 Components of Grid Application

The case study is designed following the service-oriented grid architecture to provide state, notification, execution and monitoring, and scalability. The role of the targeted system is to automate the business process with interoperable integration of scattered services. The design is upon various WS-* specifications, specifically set of WSRF and WS-Notification (WSN) specifications to achieve required compliance. The distributed architecture implemented in the use case comprises of various components and resources; the role of these resources and components in the application is discussed below (implementation details are in the section 6, 7, and 8):

5.1.1 Grid Manager:

Grid Manager Service is implementation of the Factory/Instance Collection Pattern (Section 3.1.2). It initiates different type of resources i.e. Seller, Buyer, Crop and Market according to the client request; hosted by different Grid Nodes. The Grid Manager orchestrates different components of the system in predefined manner for smooth working of the whole application. The Grid Manager is the first point of the contact with the system.

5.1.2 Grid Nodes

Grid Nodes are the hosting environment which host different vanilla or stateful Web Services. The services hosted on different nodes are used in various combinations to capture the requirement of the application. Each initialized WS-Resource has Endpoint Reference (EPR) attached to it, which identify the resource itself and the URL of the managing Instance Service. Ideally these Grid Nodes should be geographically dispersed across various enterprises.

5.1.3 Grid Registry

Grid Registry is a special Service based on the WS-ServiceGroup specification, which keeps track of various resources initialized during the course of application. The Grid Client (seller or buyer) can update and query the Grid Registry to search appropriate resources.

5.1.4 Grid Client

Grid Client is an external application which either request for initialization of the resource/s or interact with the existing resource/s to query, destroy or modify them. The client adheres with WSA specification to work with WS-Resources and corresponding services through their EPR. In this case study the Grid Client has three flavors, for buyers, sellers and administrator which instantiate the market instance for the trade.

5.2 Interactions among Components

In the Grid architecture for our agrocultural case study, various WS-Resources i.e. Seller, Buyer, Market and Crop etc are involved. During the lifecycle of business process there can be various instances of similar WS-Resource instantiated during different phases. For example at any time there can be various buyers and sellers in a single market and similarly the same buyer or seller can participate in different transactions initiated in different markets. The following sub-sections illustrate steps involved the instantiation of different WS-Resources and interactions among different components through sequence diagram.

5.2.1 Instantiation and Interaction with the Seller entity

The first interaction of the seller with the system results in the creation of the new seller WS-Resource. This resource captures all details of the specific seller and is used for the future

interaction with that seller. The Figure 4 shows the sequence of events, which allow registration of the buyer with intention to purchase the required produce in the market.

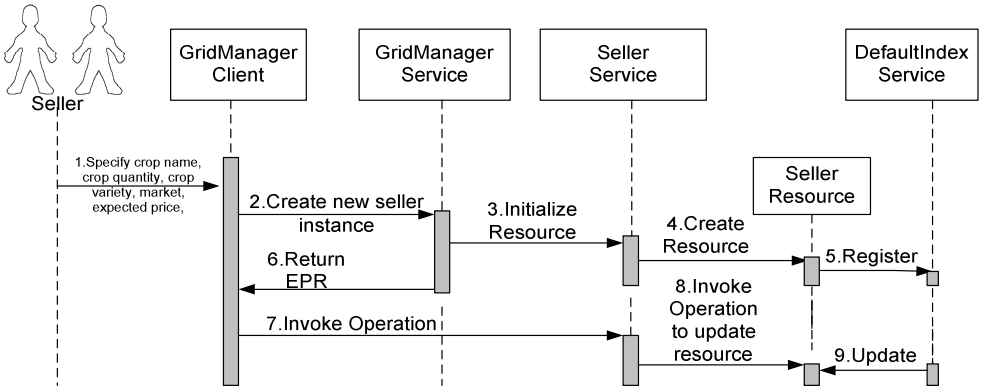


Figure 4: Seller Service Sequence Diagram

1. The trading for a seller begins with the expression of intent to sell the available crops by supplying crop name, crop quantity, crop variety, expected price/kg. of crop, and market city for trade by using the GridManagerClient.
2. The GridManagerClient sends a request to GridManagerService, which looks up the instance service of a Seller to create an instance of SellerResource.
3. Newly created Seller resource, registers with the DefaultIndexService (i.e. Grid Registry) exposed by Globus container to register the details of the newly created seller resource.
4. The GridManager returns back the EPR of a newly created SellerResource to the GridManagerClient.
5. The GridManagerClient, utilizes the EPR to invoke different operations to update the newly created resource.

5.2.2 Instantiation and Interaction with the Buyer entity.

Similarly the first interaction of the buyer with the system results in the creation of the new buyer WS-Resource. This resource captures all details of the specific buyer and is used for the future interaction with that seller. The Figure 5 shows the sequence of events, which allow registration of the buyer with intention to participate in the trade.

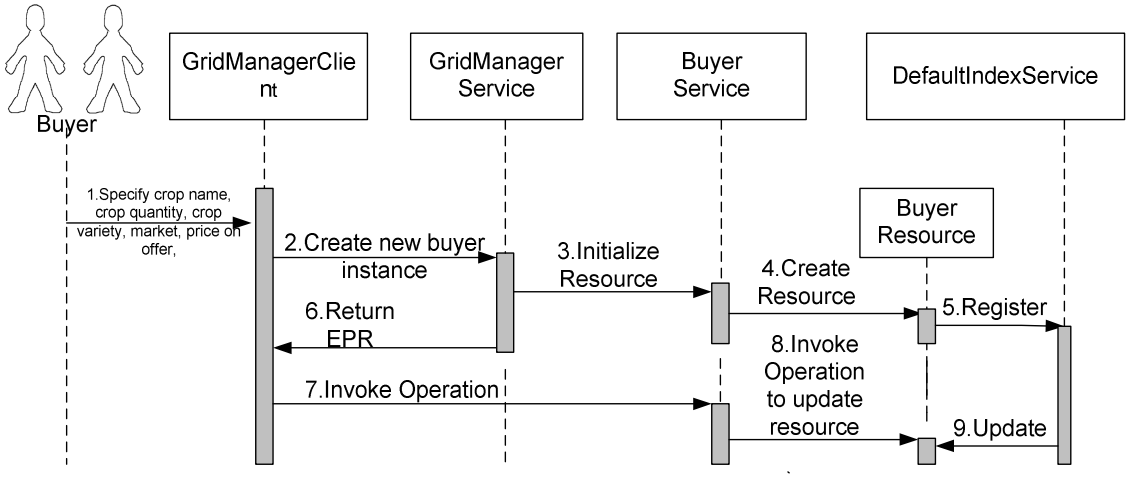


Figure 5: Buyer Service Sequence Diagram

1. The trading for a buyer begins with the expression of intent to buy the available crops by supplying crop name, crop quantity, crop variety, offered price/kg. of crop and market city for trade by using the GridManagerClient.
2. The GridManagerClient sends a request to GridManagerService, which looks up the instance service of a Buyer to create an instance of BuyerResource.
3. Newly created Buyer resource, registers with the DefaultIndexService (i.e. Grid Registry) exposed by Globus container to register the details of the newly created buyer resource
4. The GridManager returns back the EPR of a newly created BuyerResource to the GridManagerClient.
5. The EPR is utilized by the GridManagerClient to invoke different operations to update the newly created resource.

5.2.3 Market Service for Direct Trading

In Direct Trading the system matches resources for different sellers and buyers which are trading in the same location. Once any such match is found then it means there is atleast one buyer interested in the produce of a seller. To simulate the successful trade the resource properties for seller and buyers are updated. Below is the sequence diagram of events involved in direct trading.

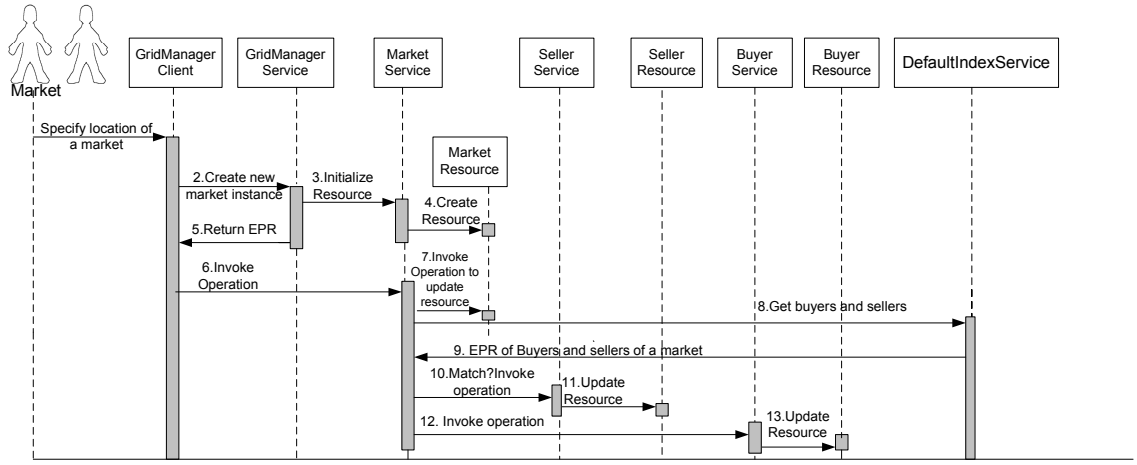


Figure 6: Sequence Diagram of Market Service for Direct Trading

1. The first step for successful trade is the instantiation of the MarketResource by passing the required location. The market only starts its operation once MarketResource is instantiated through GridManagerClient.
2. The GridManagerClient invokes the GridManagerService to create a new instance of Market Resource. The GridManagerService, triggers the creation of a new MarketResource using the Market instance service and returns the EPR to the GridManagerClient.
3. The newly created EPR is utilized to invoke operations to start trade.
4. The market service queries the DefaultIndexService (i.e. Grid Registry) by supplying the location to obtain the information about sellers and buyers registered with preferences to trade at particular location for which this market service instance is running.
5. The DefaultIndexService, returns the EPR of Buyer and Seller resources interested in trading at a given location.
6. The MarketService compares for the resource properties of the buyer and seller resources by utilizing retrieved EPR's. If the trading mode for a seller is direct and if its resource properties matches with that of required by a buyer, trading is performed by invoking

operations on the seller and buyer resources to adjust the crop quantities and earning for a seller, crop quantity and amount spent for a buyer.

5.2.4 Market Service for Indirect Trading

The sequence of events for indirect trade is quite similar to the direct trade. The initial matching of seller and buyer resources in the market is followed by invoking operations for the services specific to indriect trading and calculation of the final price after deducting appropriate charges. Below is the sequence diagram for the Indirect Trading:

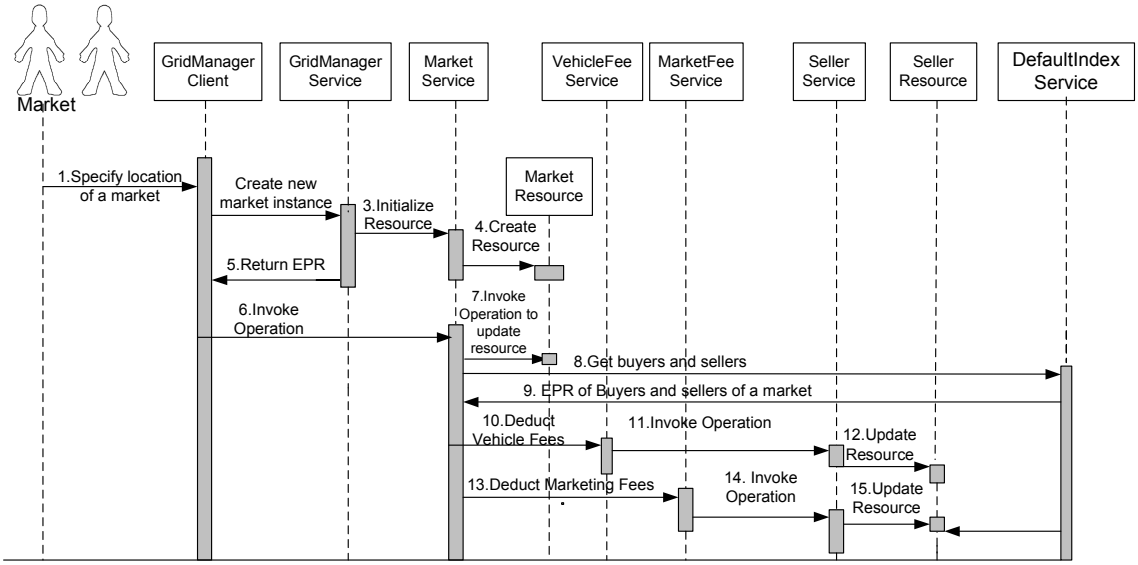


Figure 7: Sequence Diagram of Market Service for Indirect Trading

1. The market starts its operation by expressing its intent to trade. Market uses the GridManagerClient and informs about the location, where it wants to trade.
2. The GridManagerClient invokes the GridManagerService to create a new instance of Market Resource. The GridManagerService, triggers the creation of a new MarketResource using the Market instance service and returns the EPR to the GridManagerClient.
3. The newly created EPR is utilized to invoke operations to start trade.
4. The MarketService queries the DefaultIndexService (i.e. Grid Registry) by supplying the location to obtain the information about sellers and buyers registered with preferences to trade at particular location for which this market service instance is running.
5. The DefaultIndexService, returns the EPR of Buyer and Seller resources interested in trading at a given location.
6. The MarketService, compare for the resource properties of the seller and buyers trading within the same location. On any successful match, the trade preference of the seller is quried for indirect trading. The indirect trading is simulated by deducting transportation and marketing fees from the asked price.
7. As a result of successful transaction the seller resource is updated to reflect the new values i.e. crop still available to sell and total earnings.
8. Similarly the buyer resource is updated to reflect new quantity of crop required and amount spent in single year.

6. Implementation

The dynamic nature of the agro-marketing case study and requirements to monitor the state of different resources during the lifecycle of business process, are difficult to manage through vanilla Web services. Different built in features of the WS-RF family specifications naturally maps the demanding requirements of the case study and are the first choice for the implementation. Different commercial and open source implementation of the WS-RF specifications are available, and selecting any one of them is more or less personal choice. During this development Globus Toolkit 4.0.1 (GT4) on Red Hat Linux 9 is used to develop and deploy different Web services. Globus implementation of WS-RF comes with embedded container (i.e. Tomcat) and has extensive support for different level of security. The working details of the GT4 are very specific and thorough understandings of them is very crucial to develop any real world application. GT4 is based on the Axis SOAP engine and makes maximum use of different tools and feature available in the Axis toolkit.

6.1 Implied Resource Pattern

As discussed in the section 3.1, in the Implied Resource Pattern the Factory service instantiate the appropriate resource and returns the EPR of the resource to the client. The EPR includes the unique ID of the resource and the URL of the Instance service managing the resource.

6.1.1 Factory Service

The role of the Factory service is to instantiate the appropriate resource through its resource home. The Factory service retrieves the information related to the resource and its resource home through the JNDI configuration file. In the JNDI configuration file discussed in the previous section, the Factory service doesn't have any resource associated to it but it do have a link to the resource home for the AddressBookService. The AddressBookFactoryService will retrieve the information about the resource which in fact it is sharing with the AddressBookService, and instantiate it appropriately. Below is the java code to retrieve the resource home and returning the resulting EPR to the client:

```
try {
            Context initialContext = new InitialContext();
            ctx = ResourceContext.getResourceContext();
            String name = Constants.JNDI_SERVICES_BASE_NAME + ctx.getService() +
                          "/AddressBookHome";
            home = (AddressBookResourceHome) initialContext.lookup(name);
            key = home.create();         }
```

Listing 4. Code snippet of Factory Service to retrieve Resource Home

"/AddressBookHome" is the name of the resource link declared in the Factory service. The above code retrieves the resource home object and calls the create() method of the resource home object. The create() method instantiate the resource and returns the unique ID of the resource. The normal practice is to keep the resource instantiation code in the private utility method of the Factory service and this convention is followed during the implementation of the case study. This unique ID is used to create EPR, which is returned to the client. Below is the remaining code which creates the EPR of the resource:

```
try {
            URL baseURL = ServiceHost.getBaseURL();
            String instanceService = (String) MessageContext
                                     .getCurrentContext().getService().
                                     getOption("instance");
            String instanceURI = baseURL.toString() + instanceService;
            // The endpoint reference includes the instance's URI and the resource key
            epr = AddressingUtils.createEndpointReference(instanceURI, key);
            response.setEndpointReference(epr);         }
```

Listing 5. Code snippet for EPR creation of resource

The String parameter "instance" is declared in the "wsdd" which deploys the Factory service.

6.1.2 Resource Home

The resource home is simple class with only one create(); which initialize the corresponding resource. The resource home normally extends the GT4 specific ResourceHomeImpl class. The ResourceHomeImpl is very useful class which shields user from lot of implementation. This supper class locates the implementation of the resource through the JNDI configuration file and instantiates it appropriately. The ResourceHomeImpl also provides the private hash table to store all the resources created through the custom tailored resource home. The business logic of searching and updating the private hash table is automatically available in the custom resource home by extending the ResourceHomeImpl. Below is the code for the sample resource home.

```
public class AddressBookResourceHome extends ResourceHomeImpl {
  private String instanceServicePath;
  public ResourceKey create(){
  ResourceKey key=null;
  try{
    // Retrieves the resource class from the JNDI configuration file
    AddressBookResource addressBookResource = (AddressBookResource)
this.createNewInstance();
    addressBookResource.initialize();
    key=new SimpleResourceKey(this.getKeyTypeName(),addressBookResource.getID());
    this.add(key, addressBookResource);  }
```

Listing 6. Code snippet of AddressBookResourceHome

The most important bit in the method is to create the instance of the resource through createNewInstance() method. This method is implemented in the super class i.e. ResourceHomeImpl; which returns an object instance of respective resource by retrieving the implementation details of the resource from JNDI configuration file. Other than instantiating the resource, the resource home creates the instance of the SuperResourceKey. The SuperResourceKey couples the unique ID of the resource with its fully qualified name. This fully qualified unique ID is used in the EPR. At anytime container can be managing many different types of resources which may have the same unique ID; it is the fully qualified name and the unique ID of the resource which makes any specific instance of the resource unique from other instances.

```
key=new SimpleResourceKey(this.getKeyTypeName(),addressBookResource.getID());
this.add(key, addressBookResource);
```

Listing 7. Coupling of unique resource ID and resource

The implementation details of each resource home developed during the course of the case study are very similar and it will not be discussed further with actual implementation details of that Web service.

6.2 Grid Registry

The role of the Grid Registry is to monitors all resources instantiated during the life cycle of the application; irrespective of its nature and type. The resource home only monitors and manage resources of specific type, so they can't be used directly as Grid Registry. Although it is possible to develop Grid Registry on the top of all different resource homes; but this requires lot of effort to implement the business logic of searching, querying and modifying these individual resources. The GT4 implementation provides the DefaultIndexService built according to the specifications of WS-ServiceGroup discussed in the section 2.2.3. This Index Service maps to our concept of the Grid Registry; which aggregates the information related to all available resources at one place. The Index Service can be built on top other Index Services and each child Index Service

mapping specific type of resources. This concept of child Index Services leads to better management and organization of resources in the hierarchical manner. Each Grid Node i.e. GT4 WS-RF container has one Index Service supporting all the operations required by WS-ServiceGroup specification. By default the resources instantiated through resource homes are not registered with the Index Service. The resource home explicitly registers new creates resource instance with the Index Service; which aggregates all the resources at a single point. The registration process is done through ServiceGroupRegistrationClient. Below is the Java code snippet from the resource home to register the resource with the Index Service:

```
protected void add(ResourceKey key, Resource resource) {
……
EndpointReferenceType epr=null;
try {
   URL baseURL = ServiceHost.getBaseURL();
   String instanceService = getInstanceServicePath();
   String instanceURI = baseURL.toString() + instanceService;
   epr = AddressingUtils.createEndpointReference(instanceURI, key); }
……
// Location of the configuration file
String regPath = ContainerConfig.getGlobusLocation()
  + "/etc/org_sws_examples_services_buyer/registration.xml";

try {
   AddressBookResource buyerResource = (AddressBookResource) resource;
ServiceGroupRegistrationParameters params = ServiceGroupRegistrationClient
      .readParams(regPath);
params.setRegistrantEPR(epr);
Timer regTimer = regClient.register(params);
addressBookResource.setRegTimer(regTimer);
//regClient.register(params);
} ……..
```

Listing 8. Code snippet demonstrating registration of resource with Index Service

The resources have tight control the amount of information they want to advertise in the Index Service, how frequently the Index Service query their states to update itself and the polling interval. The Index Service is distributed registry; which keeps on updating itself regularly after specific intervals. Any change in the state of the resource or its deletion results in the notification of Index Service; which adjusts itself accordingly. This information is provided in the form configuration file '*registration.xml*' for each resource. Below is the sample configuration file.

```
<ServiceGroupRegistrationParameters
xmlns:sgc="http://mds.globus.org/servicegroup/client"
…….
<RefreshIntervalSecs>30</RefreshIntervalSecs>
   <Content xsi:type="agg:AggregatorContent"
xmlns:agg="http://mds.globus.org/aggregator/types">
      <agg:AggregatorConfig xsi:type="agg:AggregatorConfig">
         <agg:GetMultipleResourcePropertiesPollType
            xmlns:bs="http://www……./AddressBookService" >
            <agg:PollIntervalMillis>20000</agg:PollIntervalMillis>
<agg:ResourcePropertyNames>….</agg:ResourcePropertyNames
<!—Resource Properties specific to the AddressBookService -->
  …….
</agg:GetMultipleResourcePropertiesPollType>
      </agg:AggregatorConfig>
    <agg:AggregatorData/>
   </Content>
</ServiceGroupRegistrationParameters>
```

Listing 9. XML code snippet of registration.xml

7. Web Services Developed

In the section 5.1 different components required in any Grid applications are discussed. In the agro-marketing use case along with discussed components, different stake holders are also involved such as buyer, seller, market etc. These components and stake holders are implemented are either implemented as stateful Web services or simple stateless Web services based on its role in the overall functioning of the application. Below is the list of different Web services with their role and responsibilities, developed during the prototype.

1. **Seller Service:** The Seller Service implements the business logic necessary for the seller. This service expose different operations related to the seller resource such as instantiation of seller resources, monitoring and modification of different properties for seller resources. These operations are either executed manually by the seller or invoked automatically by other components during the course of the application as discussed in the different sequence diagrams.
2. **Buyer Service:** The Buyer Service is the implementation of our buyer stake holder. It instantiates the resource for a buyer, and provides different operation on the buyer resources. Instantiation of a buyer resource result in the registration of the Buyer with our system for future interactions. The service after receiving the preferences instantiates a new resource instance for the buyer. The service also registers these preferences with the DefaultIndexService of GT4 and provides operations to be invoked on this resource instance.
3. **Market Service:** This service acts as an intermediary between buyers and sellers. The role of the Market Service is more or less like a broker service among different stake holders. It queries the Grid Registry (i.e. Index Service provided by the Globus Toolkit) to obtain the list sellers and buyers trading in the same city/market and continuously monitors the trading preferences of buyers and sellers to perform trade. Market Service instantiate the new trade and transaction once it successfully matches the seller and buyer preferences. Initiation of the trade triggers sequence of events which results in the update of buyer and seller resources involved in the trade and notification to interested parties.
4. **GridManager Service:** The GridManager Service acts as a common factory service to instantiate different resources involved in the case study. The GridManager Service is implemented according to the Factory/Instance Collection Pattern and Master/Slave Pattern as discussed in the section 3.1. GridManager Service instantiate different resources through the specific service i.e. seller resource is instantiated through Seller Service. Interacting.
5. **VehicleFee Service:** VehicleFee Service is responsible for calculating any transportation charges incurred in case of indirect trade. The transportation charges are deducted from the final selling price and are paid by the seller.
6. **CropPrice Service:** CropPrice Service is used for retrieving the current market price of the crop involved in the trade. The CropPrice Service calculates the price of the crop based on different factors such as the type of the crop, season, its availability in the market, and its demand etc. This service is only invoked when the mode of trade is indirect.
7. **MarketingFee Service:** The MarketingFee Service is used for deducting any marketing fee incurred during the course of indirect trade. Similar to the transportation charges, marketing fee is deducted from the final selling price and is paid by the seller.

In the subsequent sections, details about various implemented services are given.

7.1 GridManagerService

The "GridManager" service as mentioned above is an interface provided to the Client application to instantiate resources related to different stateful Web services. The GridManager Service is implemented according to the Factory/Instance Collection Pattern and Master/Slave Pattern; which instantiate appropriate resource according to the client preference. The intended service whose resource instance has to be created is specified by the client program.

7.1.1 Implementation of GridManagerService

The WSDL document is a contract between the service and the client. Implementation of the service strictly follows this contract and it has one-to-one mapping of all functionality advertised through the WSDL. The implementation of the GridManager implements the only operation exposed by the WSDL document. The implementation can have any number of utility operations to support the mandatory operation. The mandatory method in the GridManager is the createResource() method, which instantiates the appropriate resource and returns the corresponding EPR. The main points related to the implementation are discussed below with reference to relevant implementation.

1. The createResource operation retrieves the name of the service from the client request for which the resource has to be instantiated. The serviceName is a private attribute of type String in the CreateResource class.
2. The getInstanceResourceHome is a private utility method to obtain the home of an appropriate resource. Each resource is instantiated through its corresponding home.
3. The object of ResourceHome is utilized to invoke create method of resource home object. The create() method creates a new resource instance and returns its unique identifier or the resource key. Notice that key is an object of the ResourceKey class. The properties related to resource key like the type of key, name of the key etc are defined in the JNDI file of the instance as discussed earlier.
4. Once the create method returns a key, then this key is used to construct the EPR of the WS-Resource. Remember

   EndPointReference=URL of the instance Service + Unique Resource Identifier.
5. The next step is to create the URL of the instance service. The URL of the instance service is created by adding the name of the instance service to the base URL of the container. The Factory service obtains the name of the appropriate instance service from the "wsdd" (Web service Deployment Descriptor) file.

```
String instanceURI = baseURL.toString() + instanceService;
```

   The "wsdd" file of GridManager service has an entry corresponding to the instanceService,

```
<parameter name="buyerInstance" value="sws/examples/BuyerService"/>
```

6. Once URL and resource key are obtained, the remaining step is to create the EPR to be return back to the client using the generated stub classes from the WSDL file.

```
epr = AddressingUtils.createEndpointReference(instanceURI, key);
CreateResourceResponse response = new CreateResourceResponse();
response.setEndpointReference(epr);
return response;
```

This utility method getResourceInstanceMethod() retrieves the information from the JNDI file associated to locate the appropriate resource home.

7.1.2 Resource Home and Resource

The GridManager service is stateless Web service which only instantiate different resources managed by other services. It doesn't manage any resource, thus there is no specific resource and resource home attached with the GridManager service.

7.2 BuyerService

Once the resource associated with the BuyerService is instantiated and the corresponding EPR is returned to the GridManagerClient. The GridManagerClient invokes different operations of the instance service referenced in the EPR; which may result in modification and update of the resource properties of the resource associated with the instance service.

7.2.1 Implementation of BuyerResource

The resource declared in the WSDL document is implemented as a separate class. It is important, that the Factory service can instantiate the resource and the Instance service can locate and update the resource properties of the corresponding resource. The container locates any particular resource through its unique ID and its qualified name as declared in the WSDL document. The namespaces in the Web services are always error prone and these are normally declared in a separate interface for reusability and maintenance purposes.

```
public interface BuyerConstants {
public static final String NS =
"http://www...../namespaces/examples/BuyerService";
public static final QName RP_BUYERCROPNAME = new QName(NS, "BuyerCropName");
……
```

Listing 10. Code snippet of Java interface for namespace declaration

Resource properties and resources are referenced by their fully qualified name. The fully qualified name includes the namespace to which the resource property and resource belongs and its local name as shown in the above code snippet.

The namespace and local name of the resource and resource properties in the interface must be in accordance with the fully qualified name of the resource and resource properties in the WSDL document. It is possible to hard code these qualified names in the Web service and resource implementation class which can lead to unnecessary typo errors. However, use of separate interface is an elegant approach since this allows all declaration in a single place. In case of any change, only interface needs update, which considerably minimize the possibilities of errors.

Each resource is implemented as a separate class and the name of the resource class is declared in the JNDI configuration file. The resource class is very simple class only instantiating different private resource properties and providing corresponding accessor methods.

```
// BuyerResource class.
public class BuyerResource implements Resource, ResourceIdentifier,
    ResourceProperties, TopicListAccessor {
private String buyerCropName;
public Object initialize() {
this.key = uuidGen.nextUUID();
this.propSet = new SimpleResourcePropertySet(
      BuyerConstants.BUYERRESOURCE_PROPERTIES);
    try {
buyerCropNameRP=new SimpleResourceProperty(BuyerConstants.RP_BUYERCROPNAME);
setCropName("NULL");
buyerCropNameRP.add(buyerCropName);
this.propSet.add(buyerCropNameRP);
….
```

Listing 11. Code snippet from BuyerResource class

The code snippet above shows the declaration of a resource property set (i.e. the resource property set wraps all resource properties related to a single resource) and a particular resource property the "buyerCropNameRP". The resource property set maps to the resource declaration in the WSDL document and thus shares the same fully qualified name.

The buyerCropNameRP is declared as an object of a SimpleResourceProperty type, provided by the GT4 implementation of the WS-RF. The SimpleResourceProperty eases the efforts to manage, monitor and update individual resource properties. The objects of SimpleResourceProperty can also be advertised as WS-Notification topics without extra efforts. The container monitors the states of these WS-Notification topics and generates notification messages on any state change.

The SimpleResourceProperty has two constructors; one takes the qualified name of the resource property and other takes the object of ResourcePropertyMetaData. The ResourcePropertyMetaData is an interface and SimpleResourcePropertyMetaData is its concrete implementation. By default GT4 WS-RF implementation ignores the maximum and minimum occurrence of any element declared in the WSDL. The ResourcePropertyMetaData provides developer tight control on the cardinality of different resource properties within the resource. The utility interface *BuyerConstants* explained earlier is utilized to initialize resource property set and different member resource properties.

As mentioned earlier the resource property set is the wrapper around different resource properties, use of GT4 provided API makes it easy to transform these objects in the corresponding XML document. This XML document is part of SOAP messages exchanged between a client and the service. In the implementation, different resource properties are initialized without any default values. It is possible to assign the default values directly to the resource properties but the more flexible approach is to declare private variables for each resource property and these variables are used to manage the state of resource properties. Each private variable has corresponding accessor methods which are indirectly used to modify and query values of resource properties at run time. For example, the resource property BuyerCropName declared in the WSDL has corresponding utility variable buyerCropName of type String.

```
private String buyerCropName;
public void setCropName(String cropName) {
  this.buyerCropName = cropName; }
```

Listing 12. Accessor method for setting resource property

The buyerCropNameRP; which is the actual resource property is passed the utility variable buyerCropName. However, it is worth mentioning that the getters and setters are not strictly required when resource properties are instances of SimpleResourceProperty. We can directly assign the value to the resource property but this approach makes it difficult to manage the state of the resource property and the source of change. Below is the code assigning the value to the resource property buyerCropNameRP through the utility variable buyerCropName.

```
buyerCropNameRP.add(buyerCropName);
```

Listing 13. Assignment of Resource Property value using utility variable

Once the resource properties are initialized properly they are added to the resource property set. The same process has to be followed for initialization of all other resource properties, which collectively represent the state of the resource.

```
this.propSet.add(buyerCropNameRP);
```

Listing 14. Addition of resource property to resource property set

7.2.2 Implementation of BuyerResourceHome

The BuyerResourceHome is a typical resource home implemented on the lines discussed earlier and its implementation details are not repeated here. The main functionality of the BuyerResourceHome is outlined below.

- Creates a new instance of BuyerResource and passing the EPR to the GridManager
- Provides the querying and searching functionality of the resource based on the resource key
- Registers the newly created resources in the DefaultIndexService of Globus Toolkit 4.

7.2.3 Implementation of BuyerService

The BuyerService provides various operations to manage and alter the state of already created buyer resources through the EPR. The BuyerService is the interface between buyer resources and entities interested in its state i.e. client, GridManager Service, Market Service etc. The BuyerService is a stateful Web service exposing various business operations and few operations related to the management of the resources as specified in the WS-RF specification. The BuyerService does not implement WS-RF specific operations as they are provided by the WS-RF container; but it has to provide the signature of WS-RF specific methods supported by the service in the WSDL file. Any service deployed in the WS-RF container is not stateful service unless it manages any resource and supports few of the WS-RF specific operations. The stateful service does not need to support all of the WS-RF specific operations and it can only supports subset of operations appropriate for its business logic. The implementation details of the BuyerService are pretty trivial, and most of the logical details are related to the deployment descriptor and the WSDL document. The "wsdd" file is specific to the Axis framework. The service informs the WS-RF container of its dependency on the WS-RF specific operations; which are implemented by the container through the parameter "providers". The parameter "providers" take space separated list of Java classes implementing various WS-RF specific operations. These classes are already registered with the WS-RF container and the container knows which class implements which operation. Below is the fragment from the "wsdd" highlighting the use of parameter "providers":

```
<parameter name="providers" value="GetRPProvider GetMRPProvider DestroyProvider"/>
```

The GT4 provides extension of the <wsdl:portType> in the form of attribute 'wsdlpp'. The 'wsdlpp' stands for 'WSDL pre processor'; which is precise way to add the details of the WS-RF specific operations in the WSDL supported by the service. Below is the portion of the WSDL document from the BuyerService showing the use of 'wsdlpp'.

```
<portType name="BuyerPortType"
  wsdlpp:extends="wsrpw:GetResourceProperty wsrpw:GetMultipleResourceProperties"
        wsrp:ResourceProperties="tns:BuyerResourceProperties">
```

Listing 15. Usage of WSDL pre processor (wsdlpp)

The developers should remember that this extension property is a utility feature only provided by GT4, for the convenience. In the final flattened file generated by GT4, the actual signature of the (i.e. messages and elements) related to wsrpw:GetResourceProperty will be inserted in the WSDL. It can be understood as a copy paste operation of picking elements from one WSDL file and placing them in another WSDL. The use of 'wsdlpp' restricts the reuse of the WSDL document across different implementations of WS-RF so it should be used with care. The actual signature of the 'wsrpw:GetResourceProperty' is shown below:

```
<operation name="GetResourceProperty">
  <input name="GetResourcePropertyRequest" message="wsrpw:GetResourcePropertyRequest"
    wsa:Action="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties/GetResourceProperty"/>
```

```
  <output name="GetResourcePropertyResponse" message="wsrpw:GetResourcePropertyResponse"
    wsa:Action="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties/GetResourcePropertyResponse"/>
  <fault name="InvalidResourcePropertyQNameFault" message="wsrpw:InvalidResourcePropertyQNameFault"/>
  <fault name="ResourceUnknownFault" message="wsrpw:ResourceUnknownFault"/>
</operation>
```

Listing 16. Signature of the wsrpw:GetResourceProperty

The GetResourceProperty and GetMultipleResourceProperties allow the clients to query and retrieve the current values of the resource properties of the resources. These portTypes are also utilized by the Index Service to regularly query the resources for updated information regarding their state.

The business operations implemented by the BuyerService are pretty trivial. Each method looks up the resource referenced in the EPR through the resource home and then after locating the resource it invokes the appropriate action on it. Below is the code for updating the amount spent by the buyer in a single year; which can be used for statistical, accounting or taxation purposes (not implemented in the case study).

```
public AmountSpentResponse setAmountSpent(float amount) throws RemoteException
  {
    BuyerResource buyerResource = getResource();
    System.out.println("Old amount spent:"+ buyerResource.getAmountSpent());
    amount=amount+buyerResource.getAmountSpent();
    System.out.println("New amount spent:"+ amount);
    Float amountSpeant=new Float(amount);
    buyerResource.setAmountSpent(amountSpeant);
    return new AmountSpentResponse();  }
```

Listing 17. BuyerService operation setAmountSpent()

The reader can now appreciate the use of accessor methods for the utility private variables corresponding to each resource property, which were implemented in the BuyerResource class. These utility methods provide more elegant solution to update the resource properties. These utility methods can validate the values before assigning to the resource properties. This validation can be of various types, such as the appropriate range for numerical values, decimal rounding for floating points, and regular expressions for the String values.

BuyerProperties is a complex data type which is declared in the WSDL document of the BuyerService. A wrapper Java class is generated for this data type when the tool “wsdl2Java” is used. The object of this class is used within the actual implementation of the case study and by the client application rather than working with raw XML documents through DOM or SAX API.

7.3 SellerService

The SellerService manages and monitors the resources related to the seller and its implementation is very similar to the BuyerService. Hence, the explanation of the SellerService is not repeated here. The complete WSDL document and the implementation of the SellerService is included in the downloadable source code.

7.4 MarketService

The MarketService is an intermediate between the ByerService and the SellerService. The MarketService provides the required platform to carry out any trade. The MarketService orchestrate different services in the pre-set manners for successful transactions, updating the involved resources, notifying the interested parties.

7.4.1 Implementation of MarketResource and MarketResourceHome

The implementation of the MarketResourceHome has exactly the same business logic and nearly the same operations as discussed in the BuyerService section 7.2. The MarketResource is instantiating the resource property set and different resource property elements on the same line as discussed in the BuyerResource. The implementation of these classes are not repeated here, but the source code is available for download.

7.4.2 Implementation of MarketService

The MarketService implements few business operations to manage resource but the most important methods in the MarketService are the fireXpathQuery() and storeEntries() method. In this section only these two methods are discussed. Familarity of these methods will help to understand the working of IndexService i.e. Grid Registry provided by the GT4.

7.4.2.1 The fireXpathQuery() Method

The fireXpathQuery() method is used to construct and fire appropriate XPath query, to query the IndexService (i.e. Grid Registry). The main purpose of this operation is to match trading preferences of buyers and sellers registered in the same market.

The fireXpathQuery() operation is bit complicated method and good understanding of its implementation is important for the understanding of the role played by the MarketService. Below are the main points related to the fireXpathQuery() operation.

1. Firstly the fireXpathQuery() operation retrieves the URL of the IndexService. This URL is used to create the EPR of the IndexService. This new created EPR is used to invoke different operations on the IndexService.
2. The fireXpathQuery() invokes different operations on the IndexService through Axis generated utility classes, i.e. WSResourcePropertiesServiceAddressingLocator and QueryResourceProperties_PortType, which serialize and de-serialize SOAP request and response messages before transforming down the wire.
3. The next step is to create an appropriate XPath query to be fired. However, the version of XPath query being used is to be specified as there are two different versions of XPath. The XPath version used for the query should match the XPath supported by the WS-ServiceGroup. This purpose is achieved by using the dialect.
4. The XPath query is used to create an appropriate request message. This newly created request message is passed to the queryResourceProperties(..) operation of the QueryResourceProperties_PortType class. The QueryResourceProperties_PortType the local stub for the remote Web service.
5. The queryResourceProperties(..) returns the result of the query wrapped in the QueryResourcePropertiesResponse object.

7.4.2.2 The storeEntries() Method

After receiving the response from the queryResourceProperties(..) in the fireXpathQuery, the next step is to get the entries matching the query crieria. Remember that the DefaultIndexService of GT4, is actually an Aggregrator Service i.e. its function is to aggregrate several resource properties together according to the WS-ServiceGrooup specifications. All entries are retrieved from the response by calling queryResponse.get_any() method; which returns an array of MessageElement type.

```
MessageElement[] buyerEntries = queryResponse.get_any();
```

The MessageElement is the Axis framework specific Java mapping of data type xsd:any. The data type xsd:any is used to pass raw XML between services particularly when the contents of the

XML are dynamic and changes at run time. Hence, we need to de-serialize the MessageElement entries to invoke different operations on them and particularly to retrieve the EPR of the resources for further point-to-point communication. The business logic of retrieving the information from individual entries is implemented in the storeEntries(..) operation.

First an array of the MessageElements is de-serialized into an object of EntryType class. From the object of EntryType different operations are invoked to retrieve information related to that the particular entry i.e. EPR. AggregatorContent class is used to retrieve the aggregated data within a particular "entry" object of EntryType class. Since, the actual data i.e. resource properties are required for match making; which is the key to initiate the trade.

```
AggregatorContent content = (AggregatorContent) entry.getContent();
AggregatorData data = content.getAggregatorData();
```

The "data" object obtained, is actually an array of the information advertised by the resource within the Index Service in the form of String. Hence, each entry in the array is reference to the specific resource property. The registration.xml file is used to set the resource properties to be advertised in the IndexService; and is discussed in the section 6.2

String cropName = data.get_any()[0].getValue(); is utilized

The data.get_any()[0].getValue() retrieves the very first entry of the array which has been returned, which is of String type. Similarly, the process has to be repeated to obtain the values for other entries. Since the position of the element can change; which requires the appropriate change in the 'index' passed to the get_any() operation. Once all entries are retrieved they are stored in a Vector, whose elements are accessed later to mach if the requirements of any buyer are satisfied by any seller. Operation performTrade() uses this Vector in which the buyer and seller entries were stored and then performs the trade contingent on the following conditions being met

- Crop required by any buyer is being offered by any seller
- Quantity is appropriate
- Crop Variety is matching
- Offered price of buyer is higher or equal to expected price of seller.

If these conditions are met, then the performDirectTrade0 function is invoked by passing the seller & buyer EPR, the quantity to be purchased and the amount, which will be offered to seller. Operation performDirectTrade(), is a normal function which utilizes the EPR's of the buyer and the seller to adjust the quantity and the amount spent by buyer and amount earned by the seller. Thus, by adjusting these resource properties simulate the successful trade and transaction.

For example if a seller has 60Kg. of rice offered at 10 Rs./Kg. and its preference match that of a buyer which needs this 60 Kg. at 12 Rs./Kg., the SellerCropQuantity ResourceProperty for this instance will be set to 0 and BuyerCropQuantity will be also set to 0, since the seller has sold as much as it wanted to and buyer has purchased as much as it wanted to. However, the SellerEarning ResourceProperty will be set to Rs. 60 x 12= Rs. 720 and the BuyerAmountSpent ResourceProperty will be set to Rs. 60 x 12 = Rs. 720.

7.5 Services for Indirect Trading

The market service also takes care of performing trading for sellers interested in indirect trading. This is achieved with the help of simple services by invoking operations on them by passing suitable parameters. These services manipulate a resource properties associated with the EPR of the seller for whom the trade is being performed. Hence, they have functions which are invoked by passing EPR of the seller. Crop quantity, crop name, crop variety etc are also passed depending upon the provided functionality.

7.5.1 CropPriceService

CropPriceService is a simple Web service, which returns the price to be offered to any farmer which is interested in indirect mode of trade. For any crop name which is provided as input, it provides the price per kg. which is offered. It does not have a resource factory with it, as the function is invoked by passing input parameters. It is based on singleton resource instance pattern. This multiplied by quantiy returns the total amount for this crop variety. The service further invokes the setEarning() method of SellerService to set the earning for this crop. The code of the service is easy to understand and comprises of just one java file namely CropPriceService.java along with other additional files related to container deployment.

7.5.2 MarketFeeService

MarketFeeService is based on similar pattern as CropPriceService. It deducts the marketing fee, which is imposed on the earning of the seller. As explained above, this also takes seller EPR as input. It utilizes this EPR to obtain the earning of the seller and deducts the market fee as applicable and invokes operation setEarning() of SellerService to set the earning for the seller after deduction of market fee.

7.5.3 VehicleFeeService

VehicleFeeService imposes vehicle fee levied on the seller for using vehicle inside market area for transport of crop. It takes seller EPR and crop quantity transported as input and then invokes setEarning() operation to set the earning for this seller after deducting vehicle fee.

7.6 Client

The end user interacts with the developed Web services with the help of a Java based Client program.The end user executes this client program by providing certain command line arguments. These command line arguments are different for a seller,buyer and market. Hence, from the common code related to intializing some paramters, the discussion for the client will be dealt in three different sections with perspective for seller, buyer and market.

7.6.1 SellerPerspective

(a) An end user interested in executing the client as seller has to provide following command line arguments:

1. URL of GridManager Service
2. Trader type namely seller in this case.
3. Crop name offered for sale
4. Crop variety
5. Crop quantity
6. Expected price
7. Trade mode like if seller wants to do direct trading with a buyer or wants to offer crop to amrket service for sale
8. Preferred trading location.

(b) Using values assigned to these variables, an object of the SellerProperties is created.

```
sp=new SellerProperties(cropName,cq,cropVariety,op,market,tradeMode);
```

(c) Next a reference to the GridMangerPortType is obtained to create a new instance of the Seller Resource. This is achieved by using the EPR of the GridManager service, which can be created by using the URI of it.

```
GridManagerPortType gridManager;
SellerPortType seller;
```

```
gridManagerEPR = new EndpointReferenceType();
gridManagerEPR.setAddress(new Address(gridManagerURI));
gridManager = gridManagerLocator.getGridManagerPortTypePort(gridManagerEPR);
```

Listing 18. EPR retrieval of GridManagerService

(d) Security related parameters are to be initialized in order to encrypt the exchange of SOAP message between the Client and the GridManagerService. Here, Message Level Security is being utilized. Hence, only the body of the message is encrypted leaving the header untouched.

```
((Stub)gridManager)._setProperty(Constants.GSI_SEC_CONV,Constants.ENCRYPTION);
((Stub) gridManager)._setProperty(Constants.AUTHORIZATION,
NoAuthorization.getInstance());
```

Listing 19. Initialization of security related parameters for SOAP message encryption

(e)The GridManagerService acts as a common factory for the Seller, Buyer and Market Service. Its create method will be invoked by passing an object of the stub class CreateResource, which was discussed in the section related to GridManagerService. The constructor of CreateResource class expects a String argument namely serviceType, to identify which stake holders Resource instance has to be created and initialized. In this case the serviceType argument will have value as "seller" to indicate that a new Seller Resource has to be created.

```
try {
createResponse = gridManager.createResource(new CreateResource(serviceType));}
```

The EPR of a Resource instance is to be returned after creating it. The EPR returned will be stored in an object of createResponse. The client needs to inform the GridManagerService about the stakeholder of a newly created resource instance namely the seller, buyer or the market.

(f) Once the EPR is returned and stored in the object createResponse, operations can be invoked on this Seller Resource to modify the default values and set the trading parameters as indicated by the command line arguments provided by the end user. This is done, by first obtaining a reference to the SellerService port type, setting of security related parameters to secure the conversation, and then invoking operations exposed by the SellerService namely setSellerProperties() by passing the object of SellerProperties class created and initialized above.

```
if(serviceType.equals("seller")){
  sellerInstanceEPR= createResponse.getEndpointReference();
  notificationEPR=sellerInstanceEPR;
seller = sellerInstanceLocator.getSellerPortTypePort(sellerInstanceEPR);
((Stub)seller)._setProperty(Constants.GSI_SEC_CONV,Constants.ENCRYPTION);
((Stub) seller)._setProperty(Constants.AUTHORIZATION, NoAuthorization.getInstance());
Try {
seller.setSellerProperties(sp); }
```

Listing 20. Invocation of operation setSellerProperties of SellerService

(g) The EPR of the Seller Resource is created and written in a file, will be utilized by a client responsible for providing notification related information when changes take place in the value of SellerResourceProperties; i.e the sellerStatus. After writing EPR in a file, the client NotificationListenerClient responsible for listening to notifications is invoked by invoking its operation startSubscription by passing serviceType and filename which contains the EPR of this newly created Resource.

7.6.2 BuyerPerspective

(a) An end user interested in executing the client as buyer has to provide following command line arguments:

1. URL of GridManager Service
2. Trader type namely buyer in this case
3. Crop name interested in purchasing
4. Crop variety

5. Crop quantity
6. Offered price
7. Preferred trading location

(b) Using values assigned to these variables, an object of the BuyerProperties is created.

```
buyerProperties=new BuyerProperties(cropName,cq,cropVariety,market,op);
```

(c) Next a reference to the GridMangerPortType is obtained to create a new instance of the Buyer Resource. The explanation and the steps are as similar as we have seen for Seller Perspective till step (e), where the serviceType will be 'buyer" instead of "seller".

(d) The EPR, which will be returned, will be of Buyer Resource, hence operations exposed by BuyerService, namely setBuyerProperties() will be invoked to replace the default initialized value of the Buyer Resource with the arguments specified by the end user. After this the explanation related to writing EPR in a file and notification is same as that of SellerResource.

7.6.3 Market Perspective

(a) An end user interested in executing the client as market has to provide following command line arguments:

1. URL of GridManager Service
2. Trading location for which it wishes to trade.

(b) Most of the explanation related to execution of a client from Market perspective remains same as Seller Perspective and Buyer Perspective. The only noticible difference being that after obtaining the EPR of the Market Resource, the operation invoked will be setMarketProperties(marketProperties); marketPortType.setMarketProperties(marketProperties);

Once the property is set, then trading will begin for the specified trading location. Sellers and buyers registered with the DefaultIndexService will be matched for similar trading preferences and trade will be done. In case of indirect trade mode for some sellers, suitable amount will be offered and its Earning Resource Property will be modified to reflect its earning. WS-RF offers functionality related to Notifications using WS-Notification, whereby a client is delivered messages whenever there is a change in the Resource Property to which it subscribes.

7.7 Notification Listener Client

A client to subscribe and receive notification messages is developed.

1. The client is a notification consumer. It will receive notifications, which will be sent by the services, BuyerService and SellerService. The services acting as notification producer will invoke Notify operation on this client. The client runs in a daemon mode to receive the notifications at any time. Thus it acts like a server on which operations can be invoked any time. In order to attain this property NotificationConsumerManager class is utilized.

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

2. The client starts listening to the incoming notifications. In order to identify the end point at which the notification has to be delivered by the Seller or Buyer service, a proper address has to be assigned to this client. This is achieved by creating an EPR of the client.

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

3. Next we send a request to the services to start sending the notification by using the standard Notify operation. The consumer EPR is passed to identify the end point at which the notification is to be delivered.

```
Subscribe statusRequest = new Subscribe();
statusRequest.setUseNotify(Boolean.TRUE);
```

```
statusRequest.setConsumerReference(consumerEPR);
```

4. The client needs to inform the Topic to which it wishes to subscribe. In this case the client subscribes to different resource properties depending upon the service to which it subscribes. As for example, if the client program is executed as a seller, then the Resource Properties will be SellerStatus.

```
TopicExpressionType statusExpression = new TopicExpressionType();
  statusExpression.setDialect(WSNConstants.SIMPLE_TOPIC_DIALECT);
 if(serviceType.equals("seller"))       {
statusExpression.setValue(SellerConstants.RP_SELLERSTATUS);
  }       else       {
statusExpression.setValue(BuyerConstants.RP_BUYERSTATUS);       }
```

5. A reference to the notification producer is obtained for sending subscription request by obtaining reference to the NotificationProducerportType, implemented by the remote service like the BuyerService or SellerService. This is done by using extends tag in the WSDL, where the portType of these service extend the NotificationProducerPortType. The notification to be delievered is for a service based on factory pattern. Thus, EPR of the resource instance to be subscribed is to be obtained. This is achieved by reading from a file, which is created by the Client.java program. After reading EPR subscription a request is sent to the remote service.

```
WSBaseNotificationServiceAddressingLocator notifLocator =
          new WSBaseNotificationServiceAddressingLocator();
         try    {
     FileInputStream fis = new FileInputStream(fileName);
     File file=new File(fileName);
     file.deleteOnExit();
     endpoint=null;
endpoint = (EndpointReferenceType) ObjectDeserializer.deserialize(new InputSource(fis),
             EndpointReferenceType.class);
     if(serviceType.equals("seller"))     {
       endpointString = ObjectSerializer.toString(endpoint,SellerRESOURCE_ENDPOINT);
     }      else     {
       endpointString = ObjectSerializer.toString(endpoint,BuyerRESOURCE_ENDPOINT);
     }      System.out.println("\nEND POINT STRING:"+ endpointString);
   }
   …….
       producerPort = notifLocator.getNotificationProducerPort(endpoint);
```

6. For every subscription request, a unique EPR is created. This EPR can be used to subscribe, pause and terminate subscription. This is achieved by

```
EndpointReferenceType statusSubscriptionEPR =
producerPort.subscribe(statusRequest).getSubscriptionReference();
```

7. The client starts listening to subscriptions till the time key is pressed. Incoming notifications are handled by a different method called deliver. The deliver method uses three parameters namely the topicPath used for creating subscription, the EPR of a producer of notification and message which is sent by the service.

Our Resource classes used ResourcePropertyTopic for notification in ResourceProperties. Hence, the notification messages are of ResourcePropertyValueChangeNotificationType. This further embeds message of ResourcePropertyValueChangeNotificationType in itself. The object of this class contains the new resource property value.

8. Execution scenario

In this section the trading process is displayed by execution of various services depending on the preferences indicated by a client.

8.1 Execution of Seller Grid Service

Now when the GridManagerClient is executed for each of the above given entities to perform trade, following situation will emerge:

- By looking at the data above it can be inferred that for direct trade, seller and buyers interested in trading in Bhopal will trade with each other. Similarly, trading will take place for sellers interested in indirect trading irrespective of location of their trading.
- A seller expresses his intention to offer his crop for sale by running the client program, Client.java by providing command line arguments.These arguments are in the following order:
  - URL of the GridManager service
  - seller: This word is provided to indicate client wants to have an instance of a seller service
  - Crop Name: The crop seller wishes to sell
  - Crop Quantity: The quantity of a crop, which seller is interested in selling
  - Crop Variety: The variety of a crop
  - Trade Location: The location where the seller wishes to trade
  - Expected Price/Kg.: The amount which a seller wishes to obtain per kg. of a crop
  - Trade Mode: If seller directly wants to sell directly to a buyer then direct mode is adopted, else indirect mode is adopted.

In the following example, an intention of a seller is shown, who is ready to sell 50 kg. of wheat at Bhopal location, and expecting minimum of Rs. 60 per kg.

```
$ java -DGLOBUS_LOCATION=$GLOBUS_LOCATION org.sws.examples.clients.GridManager.Client
http://10.100.64.65:8080/wsrf/services/sws/examples/GridManager seller wheat 50 kamla
Bhopal 60 direct
```

Execution of seller grid service will start and will wait for appropriate offer to come from a potential buyer. Seller grid service will receive notification, once an appropriate offer is floated by a buyer. Initially, following output will be generated:

```
END POINT STRING:<ns1:SellerServiceEndpoint xsi:type="ns2:EndpointReferenceType"
xmlns:ns1="http://www.securingwebservices.org/namespaces/examples/SellerService"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ns2="http://schemas.xmlsoap.org/ws/2004/03/addressing">
 <ns2:Address
xsi:type="ns2:AttributedURI">http://10.100.64.65:8080/wsrf/services/sws/examples/SellerSe
rvice</ns2:Address>
 <ns2:ReferenceProperties xsi:type="ns2:ReferencePropertiesType">
  <ns1:SellerResourceKey>a1662350-1628-11db-9878-ce21816644d9</ns1:SellerResourceKey>
 </ns2:ReferenceProperties>
 <ns2:ReferenceParameters xsi:type="ns2:ReferenceParametersType"/>
</ns1:SellerServiceEndpoint>
Waiting for notification. Ctrl-C to end.
Waiting for notification. Ctrl-C to end.
```

8.2 Execution of Buyer Grid Service

Similarly a buyer can express his intention to purchase crop by running Client.java but with a different set of command line arguments:

- URL of the GridManager service
- buyer: This word is provided to indicate that a client would like to have an instance of a buyer service
- Crop Name: The crop buyer wishes to buy
- Crop Quantity: The crop quantity buyer is interested in purchasing
- Crop Variety: The crop variety of the crop

- Trade Location: The location where the buyer wishes to trade
- Offered Price/Kg.: The amount which a buyer would like to obtain per kg. of a crop.

As shown in the following screen, one buyer is ready to purchase 10 kg. of wheat at Rs. 70 per kg. Thus, buyer grid service will also wait for appropriate match to happen in the market located at Bhopal.

```
$java -DGLOBUS_LOCATION=$GLOBUS_LOCATION org.sws.examples.clients.GridManager.Client
http://10.100.64.65:8080/wsrf/services/sws/examples/GridManager buyer wheat 10 kamla
Bhopal 70
```

Here, execution of buyer grid service will start and it will wait for appropriate match to be identified in a market located at Bhopal.

```
END POINT STRING:<ns1:BuyerServiceEndpoint xsi:type="ns2:EndpointReferenceType"
xmlns:ns1="http://www.securingwebservices.org/namespaces/examples/BuyerService"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ns2="http://schemas.xmlsoap.org/ws/2004/03/addressing">
 <ns2:Address
xsi:type="ns2:AttributedURI">http://10.100.64.65:8080/wsrf/services/sws/examples/BuyerSer
vice</ns2:Address>
 <ns2:ReferenceProperties xsi:type="ns2:ReferencePropertiesType">
  <ns1:BuyerResourceKey>3136b260-1629-11db-9878-ce21816644d9</ns1:BuyerResourceKey>
 </ns2:ReferenceProperties>
 <ns2:ReferenceParameters xsi:type="ns2:ReferenceParametersType"/>
</ns1:BuyerServiceEndpoint>
Waiting for notification. Ctrl-C to end.
Waiting for notification. Ctrl-C to end.
```

8.3 Execution of Market Grid Service

Whenever the buyers and sellers express their intentions to trade, their resource instances are also registered with the DefaultIndexService. When MarketService start its operations, it will query DefaultIndexService to obtain the buyers and sellers interested in trading. The Client.java program is executed with a different set of command line arguments.

- URL of the GridManager service.
- Market: This word is provided to indicate that the client would like to have an instance of a market service.
- Trade location: The location of a market where the actual trade will occur.

```
$ java -DGLOBUS_LOCATION=$GLOBUS_LOCATION org.sws.examples.clients.GridManager.Client
http://10.100.64.65:8080/wsrf/services/sws/examples/GridManager market Bhopal
```

Once market service of a particular location starts executing, it tries to match potential buyers and sellers. If an appropriate buyer is found for a seller, approproate notification is sent to both of them to inform them about success of entering into a trade deal. Otherwise both of them will wait for a potential purchaser of a crop. In case of indirect trade mode, the seller receives various types of notifications, which are computed for this trade.

8.4 Notification to Seller Service

After initiation of a market service for Bhopal, matching will performed for buyer(s) and seller(s) having preference for Bhopal. Based on the match found for a seller and a buyer for wheat, following notification will be generated for a seller service:

```
A new notification has arrived
Your crop sold till now is 10.0Kg
A new notification has arrived
Your crop has been sold recently
A new notification has arrived
Your crop quantity still remaining to be sold is:40.0Kg
A new notification has arrived
Amount Offered to you per Kg is Rs70.0
A new notification has arrived
```

```
Crop Quantity Which will be purchased by a buyer is 10.0Kg
A new notification has arrived
Your Earning is Rs700.0
Waiting for notification. Ctrl-C to end.
```

8.5 Notification to a buyer service

Following notification will be generated for a buyer service:

```
A new notification has arrived
The Crop Quantity which you will purchase from a seller is 10.0Kg
A new notification has arrived
Amount Offered by you per Kg is:Rs70.0
A new notification has arrived
Amount spent by you in current purchase is Rs700.0
```

9. Management of Grid Market

In the Grid environment, heterogeneous nature of resources, services and their functionality makes it difficult to manage them in a uniform way. Querying the functionality of any node and service is one of the hardest processes and Grid management system often relies on the static WSDL documentation available to them even before the start interacting with the node, rather than determining its capabilities on the fly.

In our case study, their can be different type of markets specializing in different domains, sellers, trade scenarios and buyer interests etc. which makes the management very difficult if not possible. The situation becomes more complicated as the heterogeneity of the system can't be predicted at anytime and it should cope with any future situation. WSDM provides the uniform mechanism to interact with the heterogeneous resources for effective management. As discussed earlier WSDM specification describes what information any service or node may expose for uniform management. This information can be set of resource property elements, WSDM capabilities i.e. caption, description and version, manageability characteristics specific to its domain, operational status, metrics, notification of WSDM events, metadata for resource properties elements, relationship of different resources and resource properties elements.

Although we haven't discussed different type of markets and different trade scenarios in our case study, but still we have to monitor the activities of sellers and buyers. To manage different stake holders involves exposing new set of resource property elements and corresponding operations. The implementation of these manageable resources and operations is very similar to the any ordinary resource and has not discussed in the case study. The possible manageable resource for the seller is discussed briefly, so that readers can appreciate the usage and role of WSDM in any Grid application. The possible resource properties elements for the seller manageable resource can be which can be modified only by the authentic management client:

- *Number of previous transactions:* This is metric metadata monitored for certain time period such as last 6 months. This resource property is updated for each successful transaction.
- *Number of items to be sold:* This is simple counter which monitors the item still available for sale.
- *Pending Fee:* This resource property is of data type boolean; which can have value false if the user has failed to pay any previous market fee for using any utility service.
- *Ranking:* This resource property indicates the ranking of the seller based on seller's previous activities i.e. successful transactions, activeness in the market, any complaints against the seller, his promptness to pay market fee etc.

Similarly their can be few management operations for the seller which depends on the manageable resource, update these resources, or take appropriate actions in case of any notification. Few of such operations are listed below:

- *suspendMembership:* This operation temporarily suspends the membership of the seller may be due to any on going dispute.
- *resumeMembership:* This operation lets seller to continue trading in the market once the issues related to its suspensions are resolved.
- *cancelMembership:* This operation cancels the membership of the seller permanently.
- *updateRanking:* After any transaction, sellers intentions to sell any new thing or any complaint against seller updateRanking operation is called automatically.

These manageable resources and corresponding operation are important for efficient queries. The buyer can search the seller based on the rankings, management system can offer incentives to the seller based on its activeness in market trading. This brings lot more confidence in the Grid Market as only management clients have complete access to these resource properties values and only few read-only resource properties are available to users of the system. On the same lines their will be manageable resource properties for the buyer and market.

10. Summary

Business processes of large enterprise systems interact with various stack holders i.e. business partners and customers to integrate distributed heterogeneous services. In this chapter, different specifications originating from the Grid paradigm are evaluated to capture the requirements of dynamic business process which is called Business Process Grid. These long running and dynamic business processes require support for state monitoring, transaction management, and event notifications.

We focused our attention towards the utilization and integration of existing technologies to achieve loosely coupled integration of services having support for state, notification, execution monitoring and scalability. Principles of Grid computing utilizing the Service Oriented Architecture (SOA) are followed and implemented to satisfy the requirements of the business process. To achieve these objectives, WS-RF, WSN, and relevant specifications and standards in this area are followed here. We have shown the use of WS-RF to achieve state in Web Services and WSN for notification and integration of distributed heterogeneous services. We have demonstrated the use of grid services to provide required middleware support. Convergence of grid and Web services standards and specifications for the complete execution of a business process is discussed in this chapter. The future work is diverted into several paths: implementation of policy, agreement, negotiation, and use of semantic web to provide meaningful integration and coordination of resources in a grid environment.

11. References

Akram, A. Web Services Resource Framework Resource Sharing, Tutorial 9 and Tutorial 10, http://esc.dl.ac.uk/WOSE/tutorials/.

Akram A., Kewley J., Allan R., (2006), Modelling WS-RF based Enterprise Applications, The Tenth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2006)

Birman, K. (2005). Can Web Services Scale Up? *Computer, 38*(10), 107-110.

Booth, D., Haas, H., McCabe, F., & et al. (2004). Web Services Architecture, W3C Working Group Note 11 February 2004. Available at http://www.w3.org/TR/ws-arch/.

Czajkowski, K., Ferguson, D., Foster, I., & et al. (2004). From Open Grid Services Infrastructure to WS-Resource Framework: Refactoring & Evolution.

Foster, I. (2002). *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, www.globus.org/alliance/publications/papers/ogsa.pdf*.

Foster, I., Czajkowski, K., & et al. (2005). *Modeling and managing State in distributed systems: the role of OGSI and WSRF*. Paper presented at the Proceedings of the IEEE.

Gottschalk, K. D., Graham, S., Kreger, H., & Snell, J. (2002). Introduction to Web services architecture. *IBM Systems Journal, 41*(2), 170-177.

Joseph, J., & Fellenstein, C. (2003). *Grid Computing*. Upper Saddle River, NJ, USA: Prentice Hall PTR.

Junginger, M. O. (2003). *High Performance Messaging System for Peer-to-Peer Networks.* University of Missoury, Kansas.

Laliwala, Z., Jain P. & Chaudhary S. (2006), *Semantic based Service-Oriented Grid Architecture for Business Processes*, 2006 IEEE International Conference on Services Computing (SCC 2006), SCC 2006 Industry track

Modal Act. (2003). The State Agricultural Produce Marketing (Development & Regulation Act, 2003), http://agmarknet.nic.in/reforms.htm.

Pallickara, S., & Fox, G. (2005). *An Analysis of Notification Related Specifications for Web/Grid Applications.* Paper presented at the ITCC (2).

Papazoglou, M. P. (2003). Web Services and Business Transactions. *World Wide Web, 6*(1), 49-91.

Schmit, B., & Dustdar, S. (2005). *Systematic Design of Web Service Transactions*.

SOAP. (2000). Simple Object Access Protocol (SOAP) 1.1, http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

Sorathia, V., Laliwala, Z., & Chaudhary, S. (2005). *Towards Agricultural Marketing Reforms: Web Services Orchestration Approach.* Paper presented at the SCC '05: Proceedings of the 2005 IEEE International Conference on Services Computing, Washington, DC, USA.

UDDI. (2004). Universal Description, Discovery and Integration, UDDI Version 3.0.2, http://uddi.org/pubs/uddi_v3.htm.

Vinoski, S. (2004a). More Web services notifications. *Internet Computing, IEEE, 8*(3), 90-93.

Vinoski, S. (2004b). WS-nonexistent standards. *Internet Computing, IEEE, 8*(6), 94-96.

Wang, H., Huang, J. Z., Qu, Y., & Xie, J. (2004). Web services: Problems and Future Directions. *Journal of Web Semantics, 1*(3).

Wilkes, L. (2005). The Web Services Protocol Stack, http://roadmap.cbdiforum.com/reports/protocols/index.php.

WS-Addressing. (2004). Web Services Addressing, W3C Member Submission 10 August 2004, http://www.w3.org/Submission/ws-addressing/.

WS-BaseFaults. (2006). Web Services Base Faults 1.2, http://docs.oasis-open.org/wsrf/wsrf-ws_base_faults-1.2-spec-os.pdf.

WS-BaseNotification. (2006). Web Services Base Notification 1.3, http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-pr-03.pdf.

WS-BrokeredNotification. (2006). Web Services Brokered Notification 1.3, http://docs.oasis-open.org/wsn/wsn-ws_brokered_notification-1.3-spec-pr-03.pdf.

WS-Eventing. (2006). Web Services Eventing (WS-Eventing), http://www.w3.org/Submission/WS-Eventing/

WS-Notification. (2005). Web Services Notification, www.oasis-open.org/committees/wsn/

WS-ResourceFramework. (2005). Web Services Resource Framework, www.oasis-open.org/committees/wsrf/

WS-ResourceLifetime. (2006). Web Services Resource Lifetime 1.2, http://docs.oasis-open.org/wsrf/wsrf-ws_resource_lifetime-1.2-spec-os.pdf.

WS-ResourceProperties. (2006). Web Services Resource Properties 1.2, http://docs.oasis-open.org/wsrf/wsrf-ws_resource_properties-1.2-spec-os.pdf.

WS-ServiceGroup. (2006). Web Services Service Group 1.2, http://docs.oasis-open.org/wsrf/wsrf-ws_service_group-1.2-spec-os.pdf.

WS-Topics. (2006). Web Services Topics 1.3, http://docs.oasis-open.org/wsn/wsn-ws_topics-1.3-spec-pr-02.pdf.

WSDL. (2005). Web Services Description Language (WSDL) 1.1, http://www.w3.org/TR/2001/NOTE-wsdl-20010315

Liang-Jie Zhang, Haifei Li, and Herman Lam (2004). Toward a Business Process Grid for Utility Computing. IT Professional, IEEE 6(5), 64 – 63.