Modelling WS-RF based Enterprise Applications

A. Akram, J Kewley and R. Allan

CCLRC e-Science Centre, Daresbury Laboratory, Warrington, WA4 4AD, UK
{a.akram, j.kewley, r.j.allan}@dl.ac.uk

Abstract

The Web Service Resource Framework (WS-RF) specifications originated from the Grid paradigm which has no widespread programming methodology and lacks established design models. The flexibility and richness of WS-RF specifications are ideal for the complex, unpredictable and inter-dependent components in an Enterprise Application. This paper presents a Model-Driven approach for WS-RF to meet the requirements of Enterprise Applications (EAs) spread across multiple domains and institutes. This Model-Driven approach addresses cross-platform interoperability, quality of service, design reuse, systematic development and compliance to user requirements at the design level.

1. Introduction

Modular software is designed to avoid failures in large enterprise systems, especially where there are complex user requirements. A Services Oriented Architecture (SOA) is an architectural style whose goal is to achieve loose coupling among interacting software agents (services and clients). A service is a function that is self-contained and immune to the context or state of other services. These services can communicate with each other, either through explicit messages (which are descriptive rather than instructive), or by a number of 'master' services that coordinate or aggregate activities together, typically in a workflow. An SOA can also define a system that allows the binding of resources on demand using resources available in the network as independent services.

In recent years, Web Services have been established as a popular "connection technology" for implementing SOAs. The well-defined interface required for a service is described in a WSDL file (Web Service Description Language [1]). Services exposed as Web Services can be integrated into complex workflows which may span multiple domains and organizations.

There is growing interest in the use of stateless Web Services for scientific, parallel and distributed computing [2]. Web Services Resource Framework (WS-RF) [3] specifications built on top of existing Web Services standards address the limitation of stateless Web Services through the concept of WS-Resources by defining conventions for managing a 'state' so that applications can reliably share the information, and discover, inspect and interact with stateful resources in a standard and interoperable way [4]. The lack of a recognized Model Driven strategy, standard patterns and reusable concepts for stateful resources generally results in ad-hoc solutions which are tightly coupled to specific problems, and are not applicable outside the targeted problem domain. This code driven approach is neither reusable nor does it promote dynamic adaptation facilities as it should do in an SOA.

Section 2 discusses the abstract concepts related to WS-RF and WS-Resources and we have envisioned various possible interaction mechanisms for the WS-Resources. In the Section 3, concrete design modelling approaches are presented with respect to the WS-Resource instantiation and Section 4 covers different Notification Models for the WS-Resource state change.

2. Web Services Resource Framework

Web Services lack the notion of state, stateful interactions, resource lifecycle management, notification of state changes, and support for sharing and coordinated use of diverse resources in dynamic 'virtual organizations' [5]: issues that are of central concern to the developers of distributed systems. To address these problems, two important sets of specifications: WS-Resource Framework and WS-Notification [7], built on the broadly adopted Web Services architecture [6] and compliant with the WS-Interoperability Basic Profile [13], were proposed.

WS-RF originates from the Grid paradigm which can be described as "coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations". Grid design should ensure cross-

platform interoperability and re-usability of systems in a heterogeneous context; although being a recent computing discipline, Grid Computing lacks established programming practices and methodologies.

WS-RF specifications are based on the Extensible Markup Language (XML) schemas, and Web Services Definition Language (WSDL) interfaces for the properties and ports common to all WS-RF resources. WS-RF comprises four inter-related specifications; which define how to represent, access, manage, and group WS-Resources:

- *WS-ResourceProperties* [8] defines how WS-Resources are described by XML documents that can be queried and modified;
- *WS-ResourceLifetime* [9] defines mechanisms for destroying WS-Resources;
- *WS-ServiceGroup* [10] describes how collections of Web Services can be represented and managed;
- *WS-BaseFaults* [11] defines a standard exception reporting format.

2.1 WS-Resources

WS-Resources model the state of a Web Service by wrapping atomic/composite data types called WS-Resource Properties. A Resource Property is a piece of information defined as part of the state model, reflecting a part of the WS-Resource's state, such as its meta-data, manageability information and lifetime.

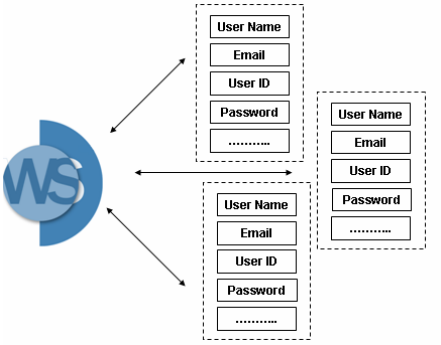

Figure 1: A WS-Resource with Resource Properties

The WS-RF specification supports dynamic insertion and deletion of the Resource Properties of a WS-Resource at run time. Customer details in a Trading System are a single WS-Resource with multiple Resource Properties like name, address, card details and trading history. The address Resource Property can have multiple entries such as billing address and shipping address. Trading history is a dynamic Resource Property, which is added for every new order and may automatically be deleted after a given period. 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. 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 WS-Resource and the URL of the managing Web Service is called an Endpoint Reference (EPR), which adheres to Web Services Addressing (WSA) [12]. WS-RF avoids the need to describe the identifier explicitly in the WSDL description by instead encapsulating the identifier within its EPR and implicitly including it in all messages addressed through it.

WS-Resources instances have a certain lifetime which can be renewed before they expire; they can also be destroyed pre-maturely as required by the application.

2.2 WS-Resource Sharing

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 leading to the concept of collaboration. Passing a unique identity of the WS-Resource instance between partner processes and organizations results in minimum network overhead and avoids issues of stale information. The WS-Resource EPRs are generated dynamically and can be discovered, inspected and monitored dynamically via dedicated Web Services. In Figure 2, Resource B is shared between two different Web Services each of them exposing possibly different sets of operations on the same WS-Resource, for instance to provide both an administrators' and users' perspective (where an administrator can modify the data and a user can only query the data).

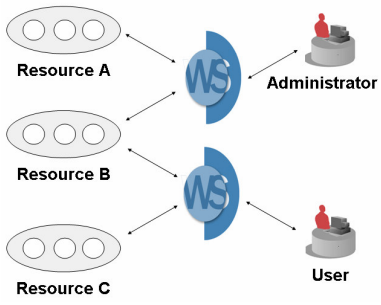

Figure 2: Web Service managing multiple WS-Resources and two Web Services sharing same WS-Resource.

WS-Resource sharing is used extensively for load balancing by deploying semantically similar or cloned Web Services for multiple client access. At run time, appropriate EPRs of the WS-Resource are generated with the same unique WS-Resource identity but with different URLs of managing Web Services.

2.3 Managing Multiple WS-Resources

In EAs, WS-Resources related to different entities can be very similar. Seller and Buyer details, for instance, are different WS-Resources in the sample Trading Application, but the majority of operations executed on these WS-Resources are either queries or minor updates. It is more effective to manage these similarly natured operations on different WS-Resources with the single Instance Service. In Figure 1, different instances of the same WS-Resource are being managed by a single Web Service; whereas in Figure 2, multiple Web Services are managing different WS-Resources which can have any number of instances. A single Web Service managing multiple WS-Resources could be deployed as a *Gatekeeper* Service which creates instances of different WS-Resources, returning the corresponding EPR. It could also be deployed as a *Monitoring* Service which monitors the state of different but inter-dependent WS-Resources. For example, when a particular stock level drops below a threshold value, the WS-Resource related to the order is either created or updated. Monitoring services have different applications like managing Quality of Service (QoS), recording usage for statistical analysis or enforcing WS-Resource dependencies and resolving conflicts: features which can be crucial for Enterprise Applications.

2.4 WS-Resource Referencing

WS-Resources are composed of Resource Properties which reflect their state. These can vary from simple to complex data types and even reference other WS-Resources. Referencing other WS-Resources through Resource Properties is a powerful concept which defines inter-dependency of the WS-Resources at a lower level. This eliminates complicated business logic in a similar way to the mapping of Entity Relationships in a Relational Database through primary and foreign keys. In EAs entities do not exist in isolation but inter-communicate and are dependent on each other's state. Similarly, WS-Resources are not only dependent on the state of other WS-Resources but can even query and modify them. In a Trading System (see Figure 3), a single User may reference multiple Orders placed by that User on different occasions and each Order references varying numbers of items purchased as a part of a single Order.

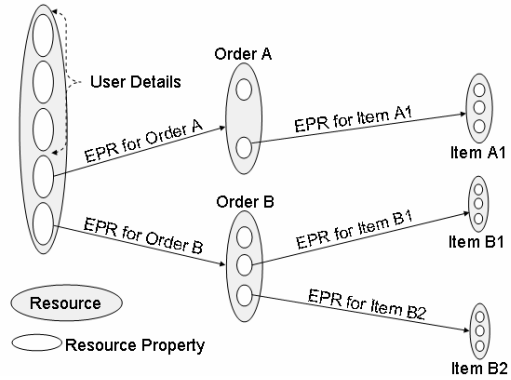

Figure 3: WS-Resources referencing other WS-Resources through their EPRs

3. Modelling the Implied Resource Pattern

The WS-RF specifications recommend the use of the Implied Resource pattern (Figure 4) to describe views on state and to support its management through associated properties. The Implied Resource pattern has a single *Factory* Service to instantiate the resources and an *Instance* Service to access and manipulate the information contained in the resources according to the business logic.

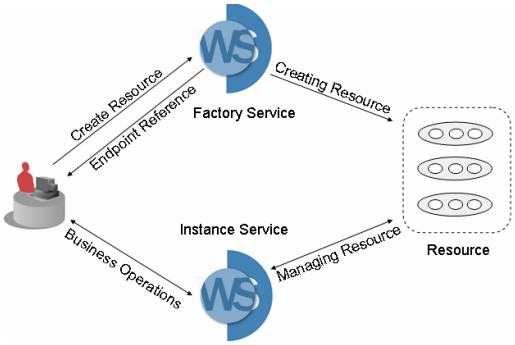

Figure 4: The Implied Resource pattern

During the prototype development we investigated variations of the Implied Resource pattern to model varying enterprise requirements.

3.1 Factory/Instance Pair Model

The Factory/Instance Pair Model (Figure 5) 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 EA 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 which can be deployed multiple times.

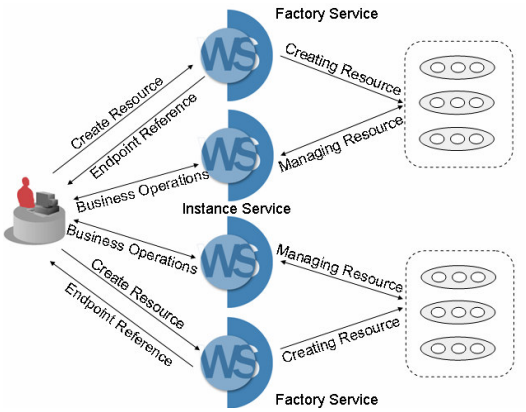

Figure 5: Factory/Instance Pair Model

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

The Factory/Instance Pair Model is preferable in many different scenarios. In many cases, WS-Resources may be instantiated for a limited duration only when required. This *late binding* is crucial to maximize the effective usage of scarce physical resources like memory or storage media. The loose coupling of different WS-Resources within the application is easier to achieve in a dynamic environment where each WS-Resource is managed by a separate pair of Factory/Instance Services. The late binding and loose coupling provides the mechanism of re-usability of WS-Resources and dynamic replacement of WS-Resources at run time. To optimize the performance the server can instantiate the pool of semantically equivalent WS-Resources and share them among different clients for scalability and meeting the high-performance requirements for EAs. Each create WS-Resource request fetches the available WS-Resource from the pool. In other instances there are a few optional WS-Resources which may only be used by certain users, e.g. in a certain university, not all students live in Halls of Residence, so such a WS-Resource (with corresponding Student Resource EPR) would only be created when a room is allocated to the student.

During the development and testing phase it is easier to test a set of Factory/Instance Services and WS-Resources in isolation using a black box testing methodology. This approach requires some fairly complicated logic for the client who must interact with the Factory Services to instantiate different WS-Resources and update the mandatory WS-Resource to reference every optional WS-Resource. The server side implementation of this model is quite simple. The Server-Client implementation is tightly integrated, leaving minimum flexibility in design change without altering the client application.

3.2 Factory/Instance Collection Model

The Factory/Instance Collection Model (Figure 6) 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. Enterprise Applications process various entities which are tightly coupled and due to this inter-dependency all of these WS-Resources must co-exist before a client may interact with them successfully. This model is suitable for core WS-Resources which should be instantiated as early as possible during the application lifecycle. For example, in a Banking System, when a user opens a bank account, three WS-Resources are created immediately: User, Current Account and Saving Account; the User WS-Resource will reference the other two WS-Resources through their EPRs.

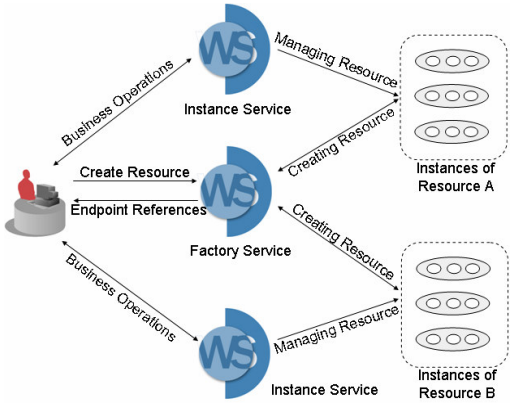

Figure 6: Factory/Instance Collection Model

The implementation of this Factory Service is likely to be more complicated than the previous, depending on the requirements of the application, and can be implemented in two possible ways:

Returning an Array. This is the simplest scenario where a Factory Service instantiates multiple Resources

and returns the array of EPRs corresponding to each Resource. The client application parses an array of EPRs and manages each of them accordingly, putting more workload on the client. This solution is "Fragile" as all clients need to know how to handle each Resource and its inter-dependencies.

Returning a Single EPR. In this recommended approach, a single Factory Service implements the business logic and inter-dependency of the different Resources. The Factory Service instantiates all the Resources yet returns only a single EPR which contains references to other Resources; this may or may not be changed by the client. The Banking example quoted above falls in this category: a Client can't modify his Account since any change would have to be authorized by the bank administration. Returning a single EPR requires a more complicated Factory Service but a much easier to implement client, resulting in a comparatively robust application.

The Factory/Instance Collection Model results in less interaction between the client and the server with minimum network overhead. The Server-Client implementation is quite flexible with minimum inter-dependency, providing the opportunity to update the server business logic without changing the client application. The biggest disadvantage of the Factory/Instance Collection Model approach is the larger initial processing overhead with its correspondingly longer initial latency, especially when the WS-Resources are geographically distributed. Due to the fundamental nature of WS-Resources' late binding, loose coupling and reusability are fairly limited in this model.

3.3 Master-Slave Model

In a security dominated era with its unpredictable request traffic, different security and load balancing measures are required for an application to run smoothly. These measures should be planned at design time, irrespective of the technologies to be used for implementation. EAs are frequently protected by a firewall. It has to be anticipated that firewall policies will limit direct access from external clients to 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.

One approach is to use a *Gateway* Service to manage multiple Factory Services in the Master-Slave format; 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.

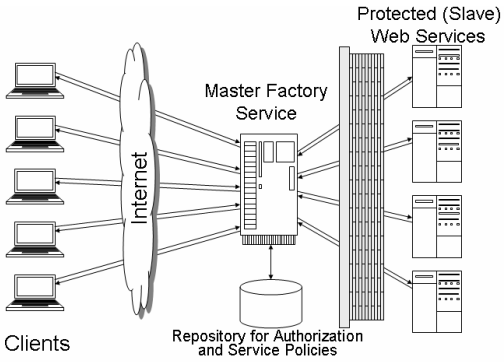

Figure 7: Master-Slave Model

Services expecting higher volumes of user traffic can be cloned and deployed on different nodes in the cluster. The Master Factory service can use any monitoring mechanism to monitor each node related to its service load, quality of service and availability of external resources and thus redirect the clients' requests to the most appropriate node (ultimate service endpoint). The main advantage of this approach is that if at any time any service is overloaded or even unavailable, then that service can easily be replaced by a compatible or cloned service on another node with minimal effort. The implementation of the corresponding client is quite simple since the client interacts only with the single Master Factory Service and is independent of the location of other protected services. The business model can be modified, updated and refined without affecting the client application as long as the Master Factory Service still provides the same interface.

3.4 Hybrid Model

We propose that the best approach 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. The core WS-Resources which are to be shared among different applications are also instantiated through the Factory/Instance Pair model. The Factory Service is an extension of the Factory/Instance Collection Model with request parsing capabilities, utilising advanced features of XML Schema in its WSDL interface. Since WSDL specifications prohibit the overloading of methods, the Factory Service implements a single "create" operation wrapping all the parsing logic in a few private utility methods. Our experience in using WS-RF for different distributed applications has shown that the "Hybrid Approach" is more manageable and easier to maintain. This is due to having a single Factory Service, shorter response times, inter-dependency logic confined to the server and the facility to upgrade or add WS-Resources.

4. WS-RF and Notification Model

The Event-driven, or Notification-based, interaction model is commonly used for inter-object communications. Different domains provide this support to various degrees: "Publish/Subscribe" systems provided by Message Oriented Middleware vendors; support for the "Observable/Observer" pattern in programming languages; "Remote Eventing" in RMI and CORBA. Due to the stateless nature of Web Services, the Web Service paradigm has no notion of Notifications. This has limited the applicability of Web Services to complicated application development. WS-RF defines conventions for managing 'state' so that applications can reliably share information as well as discover, inspect, and interact with stateful resources in a standard and interoperable way. WS-Notification (WSN) [7] is a set of three separate specifications (WS-BaseNotification, WS-BrokeredNotification, and WS-Topics), but its usefulness beyond WS-RF is limited.

The WSN specification defines the Web Services interfaces for Notification-Producers and Notification-Consumers. It includes standard message exchanges to be implemented by service providers (producers) and clients (consumers) that wish to act in these roles, along with the operational requirements expected of them. Notification Consumers subscribe with Notification Producers to request asynchronous delivery of messages. A subscribe request may contain a set of filters that restrict which notification messages are delivered. The most common filter specifies a message topic using one of the topic expression dialects defined in WS-Topics (e.g., topic names can be specified with simple strings, hierarchical topic trees, or wildcard expressions). Additional filters can be used to examine message content as well as the contents of the Notification Producer's current Resource Properties. Each subscription is managed by a Subscription Manager Service (which may be the same as the Notification Producer). Clients can request an initial lifetime for subscriptions, and the Subscription Manager Service controls subscription lifetime thereafter. Clients may unsubscribe by deleting their subscription through the Subscription Manager Service. When a Notification Producer generates a message that is sent wrapped in a <Notify> element (though unwrapped "raw" delivery is also possible) to all subscribers whose filters evaluate to 'true'. WS-BrokeredNotification provides for intermediaries between Notification Producers and Notification Consumers. These intermediaries receive messages from Notification Producers and broadcast them to their own set of subscribers, allowing for architectures in which Notification Producers do not want to, or even cannot, know who is subscribed. In any notification model a Web Service, or other entity, disseminates information to a set of other Web Services or entities, without prior knowledge of them.

4.1 Client as a Notification Consumer

In this approach the client application acts as a Notification Consumer; which is notified of any change in the "state" of the subscribed WS-Resource instance. The client processes the notification messages and updates instance/s of other related WS-Resources through corresponding Instance Services. It is the client's responsibility to inter-relate dependent Resource instances. The client application exposes a 'notify' operation to receive asynchronous notification messages and must implement the complex logic of associating different Resource instances. An EA, on the other hand, is simpler to maintain and independent of notification. There can be many scenarios where the client receives optional notifications which are independent of core functionality of application. Notification processing at the application level can be an overhead due to enormous amount of messages. The notification and subscription at the application level can result in a cyclic notification chain. In a Travel

application, clients subscribe for a particular type of deal (e.g., deals related to South Asia, family packages, multi-city tours or budget deals). With various categories and possible subcategories, managing the notifications can be a significant overhead at the application level since most do not result from a client's subscription. The client as a notification consumer is only applicable for low priority notifications where immediate action is not required. This notification model does not assume that the client application is continuously executing and any delay in response should not affect the core functionality of the Enterprise Application.

The client can also delegate any other service as a notification consumer on its behalf, provided that the service fulfils the criteria of being a 'Notification Consumer' by implementing the appropriate WSN interface. Such delegated services are independent of the business logic of the EA, although they may be provided as utility services. These are provided for the clients who do not want to be notification consumers, are not available all the time, are behind a firewall, or who do not have a reachable notification interface (e.g., desktop clients).

4.2 Service as a Notification Consumer

At the application level, different services managing different WS-Resource instances can have inter-dependencies. These services may have an interest in the state of other WS-Resource instances. It is therefore better to handle notifications of these state changes at the service level without any client interaction. This is the situation where automatic and quick action is required; these actions are not initiated by the client. The client has no role in these decisions and the actions required are related to the core functionality of the application and not with any specific client. This approach is generic where certain types of notifications are processed for a certain clients (e.g., changing the overdraft limit for all international students, a discount offer for all loyal customers, upgrading broadband speed for all customers in central London). These are more or less management and application level policies and updating the appropriate WS-Resource instances should be handled at the application level.

The client application may not even be aware of any such WS-Resource instances and their relationships. The client applications are therefore simplified with most of the processing logic residing on the server side. These server implementations can be extended so that users can subscribe and unsubscribe to certain types of notification; this requires business level support for such filtering.

Overall this approach results in a cleaner design with an easier to manage and maintain EA. Since the notification processing logic is confined to the server, the client application is immune to the 'state' changes and there is therefore no need to update the client logic whenever the business logic changes. The main drawback in this approach is that of the loose associations between WS-Resource instances which are of a one-to-many or many-to-many linkage rather than one-to-one.

4.3 Resource as a Notification Consumer

The two notification approaches discussed above (Sections 4.1 and 4.2) have their own limitations and benefits. A third notification model can provide the best of both approaches with an even cleaner design. Applications are still easy to manage and maintain and WS-Resource instances can have one-to-one associations. In this approach WS-Resource itself is a notification consumer, yet may also act as a producer. 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 (e.g., if the outstanding balance in a customer's current account is insufficient to pay a bill, funds are either transferred from the customer's savings account). 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 and WS-Resources serving as notification consumers should obey the following guidelines:

- There is a controlled number of instances of each WS-Resource at any given time;
- Each WS-Resource has limited dependency on other WS-Resources and is not involved in complicated association linkage;
- At least one of each WS-Resource instance is available all the time, Brokered Notification is required for persistent WS-Resources;
- Producer-consumer WS-Resources should be avoided if possible to avoid cyclic notification chains.

4.4 Hybrid Approach

There is no clear rule when to recommend the use of one or another of the Resource-Notification models and when they should be avoided. The application of any specific model depends on the overall role of the WS-Resources in the application; their availability and inter-dependencies; the context in which each WS-Resource is used; and the type of notification to be used (either point-to-point or broker-based notification). Distributed applications spanning multiple domains comprise large number of data entities (WS-Resources), each of them having different roles. This requires mixing different patterns to have the best of both worlds (a clean design with a manageable and maintainable implementation). This suggests a Hybrid Approach; which requires a clear understanding of the limitations and advantages of each approach. A typical EA client or delegated service will act as a notification consumer for optional Resources (which don't participate in core activities of the application and have low priority in the overall Resource hierarchy). The generic and application level notification messages will be processed by the services, whereas the critical and one-to-one association notification messages will be handled at the WS-Resource level.

5. Conclusions

WS-RF specifications are designed on top of Web Services specifications to address the implementation of heterogeneous and loosely-coupled distributed applications. WS-RF provides the missing concepts of stateful interactions, state change notification, and support for the sharing and coordinated use of diverse resources in an easy and standard way. Being a comparatively new programming paradigm, Grid computing does not have established programming methodologies. In this paper we have investigated a Model-Driven approach to encapsulate different applications and user requirements to design Enterprise Applications. This Model-Driven approach can be used from design to deployment of an application in a standard way. MDE is a useful paradigm to develop reusable, loosely coupled, scaleable and efficient systems. The use of models eases the transformation from design to the implementation of the system in a robust and flexible manner. We have presented different models based on the Implied Resource Pattern to seek a common strategy for the functional and non-functional requirements of the mandatory and optional WS-Resources.

WS-RF based applications can take advantage of many Web Services specifications to implement any level of complexity with flexibility, portability and interoperability. Migration to WS-RF is conceptually easier if the application and user requirements can be modelled earlier in the design phase. The paper further discusses different possible notification models for event-driven systems with their advantages and disadvantages in conjunction with WS-Resources and concluded that the different design models are complementary and can be combined to form a Hybrid Model.

6. References

[1] *Web Services Description Language (WSDL) 1.1,* Available at http://www.w3.org/TR/wsdl.

[2] Wolfgang Emmerich, Ben Butchart, Liang Chen and Bruno Wassermann, "*Grid Service Orchestration using he Business Process Execution Language (BPEL)*", to be published.

[3] Web Services Resource Framework, Available at http://devresource.hp.com/drc/specifications/wsrf/index.jsp

[4] Globus Alliance, Web Service Resource Framework, Available at http://www.globus.org/wsrf/

[5] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration.

[6] D. Booth, H. Haas, F. McCabe, and et al. Web Services Architecture, W3C Working Group Note 11. Available at http://www.w3.org/TR/ws-arch/, 2004.

[7] S. Graham, I. Robinson, and et al. Web Services Resource Framework Primer, Draft 5. Available at http://www.oasis-open.org/committees/wsrf, 2005.

[8] Web Services Resource Properties 1.2 (WS-ResourceProperties), http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-04.pdf.

[9] Web Services Resource Lifetime 1.2 (WS-ResourceLifetime), http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-1.2-draft-03.pdf.

[10] Web Services Base Faults 1.2 (WS-BaseFaults), http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-BaseFaults-1.2-draft-02.pdf.

[11] Web Services Service Group 1.2 (WS-ServiceGroup), http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ServiceGroup-1.2-draft-02.pdf.

[12] D. Box, E. Christensen, F. Curbera, and et al. Web Services Addressing, W3C Member Submission 10 August 2004, Available at http://www.w3.org/Submission/ws-addressing, 2004.

[13] Keith Ballinger, David Ehnebuske, Martin Gudgin, Mark Nottingham,Prasad Yendluri, Available at http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html