GT4 WS Java Core Design

May 24, 2004

Jarek Gawor {gawor@mcs.anl.gov}
Sam Meder {meder@mcs.anl.gov}

Table of Contents

1 Web Service Implementation and Design ... 3
2 Resource Discovery ... 3
3 Operation Dispatching ... 3
4 Client API ... 4
5 Resources ... 4
  5.1 Lifecycle ... 4
    5.1.1 Creation ... 4
    5.1.2 Destruction (Removal) ... 5
    5.1.3 Scheduled Destruction ... 5
    5.1.4 Loading and Storing Resources ... 5
6 Resource Properties ... 6
  6.1 Resource Properties Interfaces ... 6
    6.1.1 ResourceProperties ... 6
    6.1.2 ResourcePropertySet ... 7
    6.1.3 ResourceProperty ... 7
    6.1.4 QueryEngine ... 7
    6.1.5 ExpressionEvaluator ... 7
7 Notifications ... 8
  7.1 Subscriptions ... 8
  7.2 Notifications ... 8
  7.3 GetCurrentMessage ... 9
  7.4 Topics ... 9
    7.4.1 Creating a new topic ... 9
    7.4.2 Internal representation of a concrete topic path ... 10
    7.4.3 Topic references ... 10
    7.4.4 Demand-based creation of topics ... 10
    7.4.5 Topics and resource properties ... 10
    7.4.6 Topics and TopicListeners ... 11
  7.5 TopicExpressionEngine and TopicExpressionEvaluators ... 11
  7.6 Performance ... 11
  7.7 Lifetime ... 12
  7.8 Features we won't implement for the 3.9.0 release ... 12
8 Background Tasks ... 12
  8.1 Timer ... 12
  8.2 WorkManager ... 12
9 The Container Registry ... 13
10 Unresolved Issues ... 13

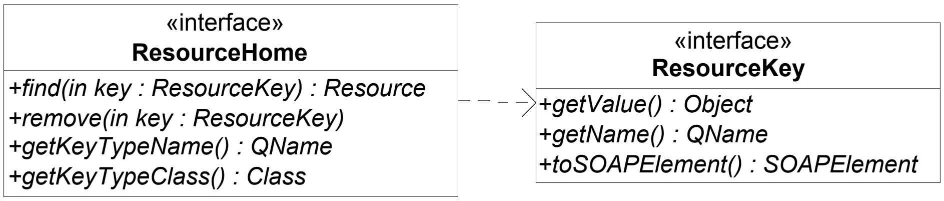
10.1 Accessing the same resource via different web services ... 13
  10.1.1 Synchronization ... 13
  10.1.2 Resource Properties ... 13
10.2 ResourceProperty Type Checking ... 13
11 Miscellaneous Issues ... 13
11.1 WSDL bindings ... 13
11.2 Issues Not Addressed ... 14

1 WEB SERVICE IMPLEMENTATION AND DESIGN

In Java, a simple web service is just a plain Java object. In our design, we allow a web service to be composed of one or more reusable Java objects called **operation providers**. Each operation provider can implement one or more web service functions.

This allows one to, for example, implement one generic *GetResourceProperty* operation provider and reuse it in multiple services. The operation provider concept in this design follows the same idea as in GT3.

2 RESOURCE DISCOVERY


«interface»
ResourceHome
+find(in key : ResourceKey) : Resource
+remove(in key : ResourceKey)
+getKeyTypeName() : QName
+getKeyTypeClass() : Class
«interface»
ResourceKey
+getValue() : Object
+getName() : QName
+toSOAPElement() : SOAPElement

A resource key is represented by a *ResourceKey* interface. It is a combination of a key name and the actual key value.

Resources are managed by an object that implements the *ResourceHome* interface. The *ResourceHome* interface provides methods for finding and removing resources as well as methods for identifying the SOAP header element and class for the resource key. In addition to the methods specified by the interface, *ResourceHome* implementations will generally provide an implementation-specific `create()` call or any other methods that operate on a set of resources.

When an operation on a WS-Resource is invoked, the WS-Addressing SOAP headers need to be resolved into an actual resource instance:

1. The first step in the resolution process is to look up the ResourceHome instance associated with the service in the container registry.

2. The lookup is followed by discovering the QName of the SOAP header element containing the resource key from the resource home and deserializing the header into an instance of the resource key class.

3. The resulting resource key instance is then used to look up the resource using the `find()` operation of the *ResourceHome*.

3 OPERATION DISPATCHING

When the service utilizes operation providers for some of its operations, the container will need figure out which object to use for invoking a given operation. This step is currently handled in the **dispatcher**. The dispatcher looks up the web service properties in the Axis service descriptor and uses that information to find the right operation provider class for the requested operation.

4 CLIENT API

The client-programming model will be very close to the programming model in GT3. For example, instead of passing *HandleType* or *LocatorType* to get a client stub for a given service, a client will be able to pass a WS-Addressing *EndpointReferenceType* to get the stub. Appropriate client JAX-RPC handlers will be used to automatically put the right WS-Addressing SOAP headers in the request.

Clients will also able to start an embedded hosting environment for the purpose of running a notification consumer service. This will allow clients to easily receive notifications.

5 RESOURCES

A **resource** is an entity that encapsulates the state of a stateful web service. Generally, each resource is a separate object but in certain cases it might be a singleton. A resource may just be a front end for state kept in an external entity, such as a file in a file system, a row in a database or an entity bean in a J2EE container.

5.1 Lifecycle

5.1.1 Creation

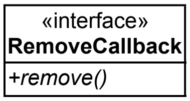
Resources may, as previously mentioned, be created by invoking an implementation-specific `create` method on a *ResourceHome* instance.

Resources may also be created two other ways:

- Resources may be instantiated on-demand (for example, the creation of a temporary representation of an external resource as a result of a *ResourceHome* `find()` call.)

- Resources may be created when the *ResourceHome* is instantiated (for example, the population of the *ResourceHome* with resources previously check-pointed to permanent storage when recovering from a container crash.)

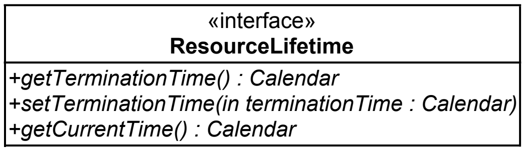
5.1.2 Destruction (Removal)



Resource instances are destroyed through the *ResourceHome* `remove()` operation. Resources **may** implement the *RemoveCallback* interface, which allows resources to clean up connected state prior to their removal.

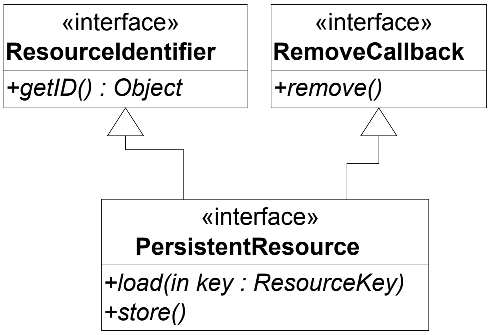
ResourceHome implementations **must** call the *RemoveCallback* `remove()` operation **prior** to removing the resource if the resource implements this interface.

5.1.3 Scheduled Destruction


«interface»
ResourceLifetime
+getTerminationTime() : Calendar
+setTerminationTime(in terminationTime : Calendar)
+getCurrentTime() : Calendar

Web services that utilize the operations defined in the WS-ResourceLifetime *ScheduledResourceTermination* port type should implement the *ResourceLifetime* interface. It allows for generic mechanisms for removing expired resources.

5.1.4 Loading and Storing Resources

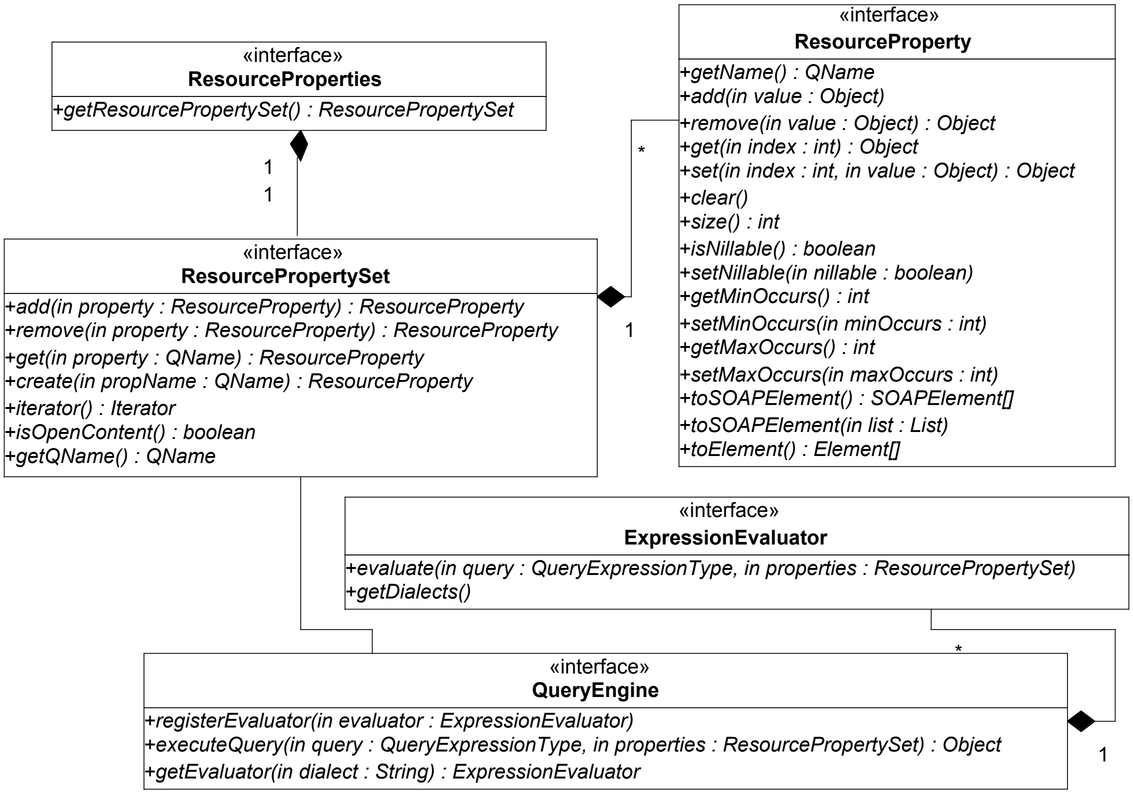


If a resource needs to be persisted to and loaded from permanent storage, then it should implement the *PersistentResource* interface. This interface provides methods for loading and storing the resource and inherits the *RemoveCallback* (discussed previously) and *ResourceIdentifier*, an interface for obtaining the resource identifier.

A resource that implements the *PersistentResource* interface must have a default constructor. It must also define at least one `create()` operation. The `create` operations are used to create new resource instances while the `load()` operation is there to retrieve the resource state from disk. The resource implementation is responsible for making sure its state in memory is synchronized with the state on disk.

Currently, only the PersistentResourceHome implementation can be used with resources that implement the PersistentResource interface.

6 RESOURCE PROPERTIES


«interface»
ResourceProperties
+getResourcePropertySet() : ResourcePropertySet
1
1
«interface»
ResourcePropertySet
+add(in property : ResourceProperty) : ResourceProperty
+remove(in property : ResourceProperty) : ResourceProperty
+get(in property : QName) : ResourceProperty
+create(in propName : QName) : ResourceProperty
+iterator() : Iterator
+isOpenContent() : boolean
+getQName() : QName
*
1
«interface»
ResourceProperty
+getName() : QName
+add(in value : Object)
+remove(in value : Object) : Object
+get(in index : int) : Object
+set(in index : int, in value : Object) : Object
+clear()
+size() : int
+isNillable() : boolean
+setNillable(in nillable : boolean)
+getMinOccurs() : int
+setMinOccurs(in minOccurs : int)
+getMaxOccurs() : int
+setMaxOccurs(in maxOccurs : int)
+toSOAPElement() : SOAPElement[]
+toSOAPElement(in list : List)
+toElement() : Element[]
«interface»
ExpressionEvaluator
+evaluate(in query : QueryExpressionType, in properties : ResourcePropertySet)
+getDialects()
*
«interface»
QueryEngine
+registerEvaluator(in evaluator : ExpressionEvaluator)
+executeQuery(in query : QueryExpressionType, in properties : ResourcePropertySet) : Object
+getEvaluator(in dialect : String) : ExpressionEvaluator
1

Resources may have **resource properties**. Resource properties are declared in the WSDL of the service as elements of a resource property document. The content of the document may potentially be open.

Resources that expose resource properties are required to implement the *ResourceProperties* interface and to register their resource properties with the *ResourcePropertySet* interface implementation returned by their implementation of the *ResourceProperties* `getResourcePropertySet()` method.

6.1 Resource Properties Interfaces

6.1.1 ResourceProperties

The *ResourceProperties* interface contains a single accessor method for retrieving the *ResourcePropertySet* from a resource. It must be implemented by **all** resources that want to expose resource properties.

6.1.2 ResourcePropertySet

The *ResourcePropertySet* is the representation of the resource property document associated with the resource. It contains methods for managing the set of resource properties, e.g. adding and removing resource properties, and for discovering properties of the document itself, e.g. its name.

6.1.3 ResourceProperty

The *ResourceProperty* interface needs to be implemented by all resource properties. It contains methods for:

- managing the set of values associated with the resource property.
- discovering properties of the resource property element.
- serializing the resource property to a array of SOAP or DOM elements.

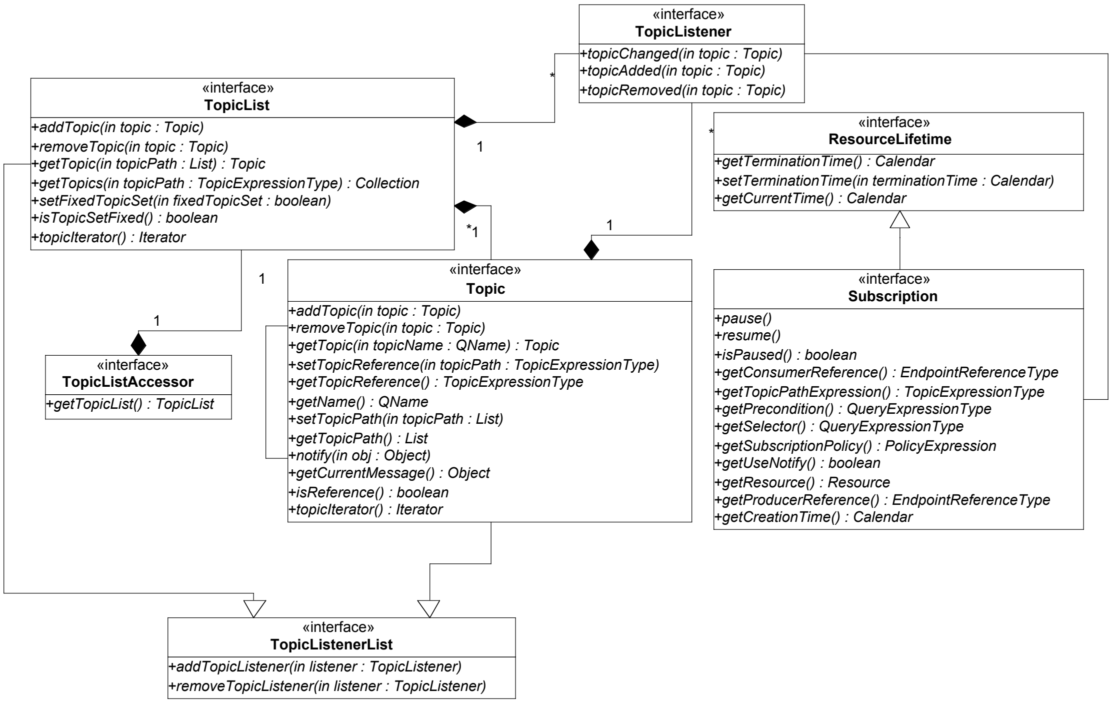
6.1.4 QueryEngine

The *QueryEngine* interface provides a dialect-neutral mechanism for evaluating queries on a *ResourcePropertySet*. It determines the dialect from the query expression and calls the *ExpressionEvaluator* registered for the dialect to perform the dialect-specific query operation.

6.1.5 ExpressionEvaluator

The *ExpressionEvaluator* interface is used for implementing dialect-specific query operations. Implementations of this interface are meant to be registered with an implementation of the *QueryEngine* interface.

7 NOTIFICATIONS


«interface»
TopicListener
+topicChanged(in topic : Topic)
+topicAdded(in topic : Topic)
+topicRemoved(in topic : Topic)
«interface»
TopicList
+addTopic(in topic : Topic)
+removeTopic(in topic : Topic)
+getTopic(in topicPath : List) : Topic
+getTopics(in topicPath : TopicExpressionType) : Collection
+setFixedTopicSet(in fixedTopicSet : boolean)
+isTopicSetFixed() : boolean
+topicIterator() : Iterator
«interface»
ResourceLifetime
+getTerminationTime() : Calendar
+setTerminationTime(in terminationTime : Calendar)
+getCurrentTime() : Calendar
*
1
*
*1
1
1
1
«interface»
Topic
+addTopic(in topic : Topic)
+removeTopic(in topic : Topic)
+getTopic(in topicName : QName) : Topic
+setTopicReference(in topicPath : TopicExpressionType)
+getTopicReference() : TopicExpressionType
+getName() : QName
+setTopicPath(in topicPath : List)
+getTopicPath() : List
+notify(in obj : Object)
+getCurrentMessage() : Object
+isReference() : boolean
+topicIterator() : Iterator
«interface»
Subscription
+pause()
+resume()
+isPaused() : boolean
+getConsumerReference() : EndpointReferenceType
+getTopicPathExpression() : TopicExpressionType
+getPrecondition() : QueryExpressionType
+getSelector() : QueryExpressionType
+getSubscriptionPolicy() : PolicyExpression
+getUseNotify() : boolean
+getResource() : Resource
+getProducerReference() : EndpointReferenceType
+getCreationTime() : Calendar
«interface»
TopicListAccessor
+getTopicList() : TopicList
«interface»
TopicListenerList
+addTopicListener(in listener : TopicListener)
+removeTopicListener(in listener : TopicListener)

7.1 Subscriptions

The following describes the intended flow of events:

1. A client invokes the `subscribe` operation, which gets routed to the `subscribe` operation provider.

2. The `subscribe` provider discovers the resource home for subscription resource from the registry and creates a subscription resource (i.e. an object that implements the *Subscription* and *ResourceProperties* interfaces.)

3. It then associates the created subscription resource with a *TopicListener* instance and registers the *TopicListener* with the topics the subscription applies to (via the *TopicListenerList* interface.)

4. The set of topics a subscription applies to is generated by resolving the subscription topic expression to a set of concrete topics via a call to the `getTopics(topicExpression)` method on an object implementing the *TopicList* interface.

7.2 Notifications

A situation (for example, a resource property change) is represented by an object that implements the *Topic* interface. Now assume that the situation occurs. Notification of the situation may be sent either:

1. **implicitly** by automatically invoking the `notify()` method from within the method that caused the situation.

   We leave it up to the implementer of the method that causes a situation to support for the implicit `notify()` call.

2. **explicitly** by invoking the `notify()` method after the situation occurred.

   The explicit mechanism will always be provided as part of the *Topic* interface. Also, an explicit `notify()` can be used to send a notification even if the state has not changed.

The `notify()` call itself is responsible for:

1. Traversing the list of *TopicListeners* associated with a *Topic* interface.

2. Calling the `topicChanged()` method on each listener.

3. This triggers the actual sending of a notification message, either immediately or after certain (configurable) events (referred to as *non-immediate*.)

   Non-immediate notification only sends a message after either a given number of situations have occurred or a given amount of time has passed. This option should be used in latency insensitive scenarios, where it is more efficient to aggregate notification messages before sending them.

7.3 GetCurrentMessage

The current message for a given topic/resource may be obtained by invoking the `getCurrentMessage()` method on the topic in question.

7.4 Topics

7.4.1 Creating a new topic

Service implementers may create a new topic by:

1. Getting an instance of an object that implements the *Topic* interface.

2. Then, adding that topic either:

   a. to an existing topic tree (also represented by an object that implements the *Topic* interface) or

    b. by registering it as a new root topic with an object implementing the *TopicList* interface.

We expect that not all topic objects represent actual situations; some may be used for the sole purpose of building a topic hierarchy.

We may in the future support child topics with the name '*' to indicate that while we support subscription to any child topic, we are unable to instantiate all of these child topics in the topic space structure. Real child topics would only be created upon subscription to said child topic.

7.4.2 Internal representation of a concrete topic path

Concrete topic paths are represented using a list of QNames, only the first of which is fully qualified.

7.4.3 Topic references

A topic reference is equivalent to a named topic path expression. Topic references are objects that implement the *Topic* interface and that contain a topic path expression set via the `setTopicReference()` method. One can determine whether a given topic is a reference by invoking the `isReference()` method.

7.4.4 Demand-based creation of topics

Even though we made the assumption that we will not support on-demand creation of topics, this should still be achievable by implementing a *Topic* that automatically instantiates child topics when queried for specific child topics via the `getTopic()` method.

7.4.5 Topics and resource properties

Resource properties can be exposed as topics in several ways. The approach we have taken to date is to combine implementations of the *Topic* and *ResourceProperty* interfaces using delegation. Other approaches, such as inheriting and extending, are equally possible.

7.4.6 Topics and TopicListeners

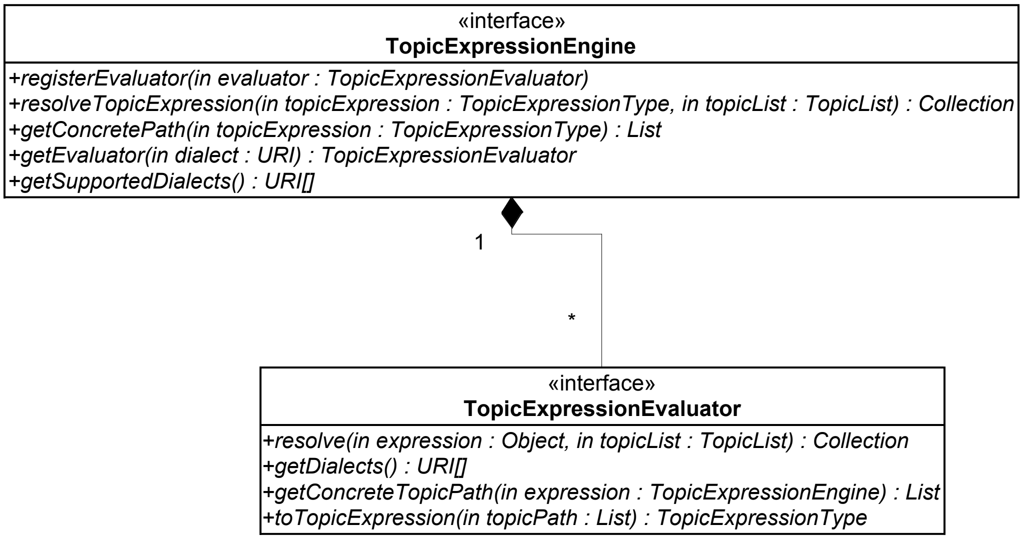
Topics as well as the *TopicList* allow for a list of *TopicListeners*. *TopicListeners* are used for:

- notification of changes in the topic hierarchy (for example, when child and root topics have been added or removed.)

- changes in the topic "value" (such as the occurrence of a situation.)

TopicListeners currently connect topics with subscription resources as well as construct and update the *Topic* resource property. In the future, they may also be able to automatically associate existing subscriptions with newly added topics.

7.5 TopicExpressionEngine and TopicExpressionEvaluators


«interface»
TopicExpressionEngine
+registerEvaluator(in evaluator : TopicExpressionEvaluator)
+resolveTopicExpression(in topicExpression : TopicExpressionType, in topicList : TopicList) : Collection
+getConcretePath(in topicExpression : TopicExpressionType) : List
+getEvaluator(in dialect : URI) : TopicExpressionEvaluator
+getSupportedDialects() : URI[]
1
*
«interface»
TopicExpressionEvaluator
+resolve(in expression : Object, in topicList : TopicList) : Collection
+getDialects() : URI[]
+getConcreteTopicPath(in expression : TopicExpressionEngine) : List
+toTopicExpression(in topicPath : List) : TopicExpressionType

We previously mentioned that topic expressions are resolved to a set of topics by invoking the `getTopics()` method defined in the *TopicList* interface. In practice, this method delegates the actual resolution to a *TopicExpressionEngine* implementation.

The *TopicExpressionEngine* interface provides a mechanism for delegating the resolution to a dialect-specific *TopicExpressionEvaluator* implementation. This allows for the easy and transparent addition of new topic expression dialects.

7.6 Performance

We may want to cache topic path expression evaluations, especially once we start supporting more complex ones. The problem there is figuring out an efficient mechanism for triggering invalidation. One could conceivably add a change timestamp scheme or something similar.

Other performance improvements under consideration are the previously mentioned mechanism for throttling and aggregating notifications and implementing notifications that send only partial messages rather than the full notification message.

7.7 Lifetime

The lifetime of the subscription resource is independent of the lifetime of the WS-Resource producing the notifications.

7.8 Features we won't implement for the 3.9.0 release

- WS-BrokeredNotification
- Dialects other than the simple topic dialect. This implies that we will initially only support a flat topic space.
- Support for raw notification messages
- The optional elements in the `subscribe` operation, such as `selector` and `precondition`.

8 BACKGROUND TASKS

Services commonly need to run background tasks outside of the execution context of a specific web service operation. To accommodate this need and to provide the environment with a mechanism for controlling the resource used by background tasks. we provide implementations of two APIs (*Timer* & *WorkManager*) proposed by BEA and IBM for use in J2EE environments.

8.1 Timer

The *Timer* API provides the same schedule methods as the *java.util.Timer* API with the added ability of suspending and resuming all timer tasks. The *Timer* API is implemented using a configurable pool of *java.util.Timer* objects. A default *TimerManager* instance is discoverable in the registry.

8.2 WorkManager

The *WorkManager* API provides methods for running non-periodic background tasks. It provides for short as well as long running tasks by allowing the user to either schedule the task using threads from a thread pool (for short tasks), or start a new daemon thread (for long running tasks.)

It also provides several methods for waiting on task completion, both synchronous and asynchronous. Similar to the *TimeManager*, the container provides a default *WorkManager*, which is discoverable in the registry.

9 THE CONTAINER REGISTRY

Parts of the Core design rely heavily on the concept of a container registry for discovering preconfigured resources, such as *ResourceHome* instances, the default *WorkManager*, *QueryEngine*, etc. implementations. The registry must accessible through JNDI APIs.

JNDI provides the service developer and administrator with a convenient way of configuring both simple resources (e.g. configuration information) as well as complex resources (e.g. a database connection cache).

Our registry implementation is based on the JNDI implementation found in Apache Tomcat. We attempted to keep the configuration format for specifying JNDI entries as similar to the format used in Tomcat as possible; however, minor modifications were required due to the different architectures.

10 UNRESOLVED ISSUES

10.1 Accessing the same resource via different web services

10.1.1 Synchronization

It is up to the resource implementer to make sure that the state encapsulated by the resource object is properly synchronized.

10.1.2 Resource Properties

When the services sharing the resource expose different closed content resource properties documents, there needs to be some sort of filtering when executing operations against those resource properties. We aim to support this filtering in the operation providers for these operations.

10.2 ResourceProperty Type Checking

In the current design, there is no specified way to get information on a *ResourceProperty* (such as its type.) We should consider adding *ResourcePropertyMetaData* to *ResourceProperty* to allow type checking, etc.

11 MISCELLANEOUS ISSUES

11.1 WSDL bindings

The new WSRF specifications use standard document/literal binding. The OGSI specification used wrapped/literal binding. A wrapped/literal binding is a subset of document/literal binding with certain naming conventions in the WSDL.

We used the wrapped/literal binding in OGSI because .NET supports it by default and also API generated by the tooling is much more developer-friendly then plain document/literal. For example if an operation takes two parameters, with a wrapped/literal binding, a generated method signature would look like the following:

```
foo(int a, int b);
```

With document/literal the generated method signature would look like the following:

```
foo(FooIn in);
```

Where `FooIn` is an object that would have two getters for each of the parameters.

Obviously, the wrapped/literal signature is more logical and friendly. The problem with the wrapped/literal conventions is that they are not standardized.

Because OGSI and WSRF are using different WSDL binding styles, the existing OGSI clients and services might need to be updated to conform to the document/literal binding and method signatures.

11.2 Issues Not Addressed

This document does not address any transitioning issues such as GWSDL to WSRF WSDL translation tools.