Transactional Support for Web Service Orchestrations

October 25, 2007

1 Document Structure

This document describes the implementation of a transactional model for web services based on the Web Service-Composite Application Framework specifications (WS-CAF).

The first section introduces the final goal of the WS-CAF specifications. The second and third sections describes the base specifications of the WS-CAF and its implementation, the Web Services Context and Web Services Coordination Framework. The fourth section introduces the transactional model selected: the Long Running Action model. Finally, in the last section, we give a detailed explanation of the WS-CAF Implementation project and hints about how it must be used.

2 Introduction to WS-CAF

The WS-CAF is a set of three related specifications: Web Services Context (WS-CTX), Web Services Coordination Framework (WS-CF) and Web Services Transactions (WS-TXM). As a whole, these specifications provide a stack of functionality for supporting applications or business processes that involve multiple web services (WSs) and that need transactional support.

Particulary,

1. WS-CTX allows WSs to share a common context structure, which is designed to be used independently of the other two specs. It is the base spec. of the WS-CAF.

2. WS-CF complements WS-CTX. It defines a coordinator to guarantee the notification of messages to the WSs that share a particular context.

3. WS-TXM defines several transaction protocols for ensure that a common outcome is agreed between the WSs participating in an application or business process.

3 WS-CTX

3.1 Introduction

The ability to scope work is needed in many aspects of distributed applications such as business interactions or workflows. In order to correlate the work of participants in a business activity, it is needed to propagate additional information known as the context. In this way, the participant knows surely in wich business activity is participating.

The main goal of the WS-CTX spec. is to propose an extensible basic context and the operations to manage it. The intention is that this basic context could be the shared basis from which other WS specs. (like WS security, WS coordination, WS transactions...) could be constructed. These specs. will augment the basic context with their specific information.

WS-CTX provides the structuring mechanisms (operations) for organizing context information by means of which the callers (applications, WSs...) may create, update and finalize a basic context. Other specs. may use and extend this basic context.

3.2 Context description

The context is propagated in the SOAP header when an operation on a WS is called. To find a context in the SOAP header indicates that the callee task (the WS operation) is part of the activity that the context represents.

A context may be passed by reference of by value. The spec. leaves to the implementation to choose either if the context is propagated by reference or by value. Passing the context by value implies to add an snapshot copy of all the context items to the SOAP header. Passing the context by reference, implies that only the context identifier is included in the SOAP header and the URL to a dereferencer service is needed.

A basic context is composed of the following elements.

context-identifier: This is the unique mandatory element. It must contain a unique URI that will identify the context.

context-manager: A reference to a context manager service.

context-service: A reference to a context service.

parent-context: To support nested activities, contexts may be nested as well. This element contains the contents or a reference to its parent context.

expires-at: An activity may have a timeout. This element contains the expiration time for the context.

Arbitrary elements: Apart from the elements described above, a context may contain whatever other element that can be expressed in XML. This permits the referencing specifications to extend the basic context to create new types of contexts.

3.3 Main components

The WS-CTX service implementation consists of two basic components that correspond to two porttypes of the WSDL document provided in the spec.

Context Service This service provides the operations to demarcate contexts (begin, complete), and to get/set information about it (getStatus, get/setTimeout). The begin operation creates a new context and returns its context identifier. Each invocation on these operations (except maybe in the begin method) must be contextualized with this context identifer in order to look for the target context where to act. If the begin operation is called with no context, it will create a root context. If a context is found (either by value or by reference), the begin operation will create a context that will be nested with the context found in the SOAP header.

Context Manager Service This element provides the necessary operations to get and set the entire contents of a previously created context. Each invocation on these operations must be contextualized in order to look for the target context where to act.

3.4 Notes on the implementation

The standarization process is still being carried out. Therefore, there are mismatchs between the spec. and the WSDL files.

The WSDL files taken from the official WS-CAF page (www.oasis-open.org) define ports and bindings to implement asynchronous WS. They have been modified in order to specify also synchronous WSs (RPC style). That way we support synchronous invocations for the SOAP engines that doesn't support asynchronous invocations (like Axis 1.X) while we levereage the asynchronous invocation mechanism used in Axis2. Also the traditional asynchrous invocation based on callbacks is supported through WS-Addressing.

The framework has been implemented using Apache AXIS2 as the SOAP engine to support the interactions with WSs.

3.5 Context Manager

The Context Manager service has two operations:

getContents: Returns the content of the context reference passed in the incoming SOAP message.

setContents: Changes to contents of the context reference passed in the incoming SOAP message with the information passed in the body.

The sequence of messages from an invoking service to a Context Manager service is shown in figure 1.

When it receives an invocation, it invokes a singleton class that holds the contexts collection for the Context Service and Context Manager services. It

has operations to add contexts, retrieve a context or change the contents of a given context.

3.6 Context Service

The Context Service service has the following operations:

begin: Creates a new context instance. If it detects a context in the incomming SOAP message, the new context will be a child of the given context.

complete: Completes a context. That marks the end of the activity that the context represents. If a context has handlers registered, then all of them will be invocated.

getStatus: Gets the status of the given context.

setTimeout: Sets the timeout of the given context.

getTimeout: Gets the timeout of the given context if any.

Upon invocation, it invokes a singleton class called ContextService that will serve the operations. That class will invoke internally the ContextManager class.

3.7 Limitations

Due to the Context Service and Context Manager services sharing the singleton ContextManager class, both services must be deployed in the same Axis2 installation.

The setContents operation adds and changes elements based on the received information, but it can't delete elements from a context.

4 WS-CF

4.1 Introduction

WS-CF extends the WS-CTX specification to add a generic coordination service. The specification permits the web services to register in an activity extending the basic context with structures representing participants. It also defines a service to manage the participants enroled in activities.

4.2 RegistrationContext description

WS-CF extends the basic context with the following elements:

registration-service: A reference to a Registration Service service.

sub-protocol: Some sub-protocols in which the activity may be involved

participants: A list of protocols in which the activity may be involved and a set of participant addresses involved in every protocol.

extension: Apart from the elements described above, a context may contain whatever other element that can be expressed in XML. This permits the referencing specifications to extend the basic context to create new types of contexts.

4.3 Main components

Registration Service represents a service that another service may use to add, remove or replace participants involved in an activity.

4.4 Notes on the implementation

WS-CF specification is based on WS-CTX 1.0 specification, however a new version of WS-CTX has been released (1.0 Standard). The WSDL and XSD files have been changed in order to support this new WS-CTX version. Also the same notes as in section 3.4.

4.5 Registration Service

The Registration Service service has the following operations:

addParticipant: Register a set of services in a set of protocols for the activity.

removeParticipant: Remove a set of services previously enroled in an activity.

replaceParticipant: Replace the participants passed as argument with some others.

getParticipants: Returns a list of participants for a given set of protocols.

getStatus: Returns the current status of the activity represented by the given context.

replaceRegistration: This operation must be implemented by a registering service. When a Registration Service fails and recovers it may change the address in which it is listening. To inform the registered services of that event, it may invoke this operation in every participant found in every context registered when the Registration Service recovers from a crash.

The sequence of messages from an invoking service to a Registration Service service is shown in figure 2.

Every invoked operation must be contextualized including a valid context reference in the incomming SOAP header. That context represents the activity that the invoking service is refering to.

Upon invocation, it will use the singleton ContextManager class described above to modify the contents of the involved contexts, adding, removing or replacing participants.

To support protocol termination tasks, every time a participant is added to a new protocol, a class implementing the CompletionHandler interface is added to the involved context class. That class will have one operation called complete which receives the completing Context class and the completion status. For example, given a context class with no CompletionHandler elements, if a addParticipant request is performed over that context for protocol "LRA", a new LRACompletionHandler object will be added to the list of CompletionHandlers that the involved context has. When a context is completed through the Context Service (calling the complete operation), all of the Completion Handlers for that context are executed.

4.6 Limitations

As it shares the ContextManager class with the Context Service and Context Manager services, the Registration Service must be deployed to the same Axis2 instance that the Context Manager and Context Service are deployed.

5 Long Running Actions (LRA)

5.1 LRA Model

The LRA model is designed to be used in business interactions that occur over a long periods of time. A business interaction (e.g. A seat reservation on an airline) is represented by an LRA activity. The LRA activities can be nested. The work performed within an LRA activity is required to be compensatable and must remain ready for compensation until the enclosing environment informs that it is no longer needed. Therefore, the activity's work is either performed successfully or undone using a compensator when the LRA activity is going to be terminated and the LRA protocol is triggered. The WS-LRA spec. states that a compensator is an LRA participant that operates on behalf of a service to undo the work performed within the scope of a LRA or to compensate a piece of work that could not be completed.

The spec. does not define how the participant services perform their work and how they ensure that the completed work can be undone.

5.2 LRAContext description

WS-LRA extends the RegistrationContext with the following elements:

lra-id: Id which identifies the current LRA activity.

timelimit: A time limit to complete the activity

coordinator-hierarchy: A set of URIs representing a Coordinator hierarchy.

5.3 Main components

WS-LRA spec specifies the interface that a compensator service must implement in order to compensate its work, complete it or free the allocated resources needed to compensate. It will be invoked by the Context Service when an activity is completed and there are participants registered for the LRA protocol in that activity's context.

5.4 Notes on the implementation

The last WS-LRA dates from 2003 while the last WS-CTX and WS-CF specifications date from 2005. To adapt the WS-LRA to the new specifications released later, the WSDL and associated schema file has been heavily modified. Also the same notes as 4.4 applies to this.

5.5 LRACompletionHandler

This class implements the LRA completion protocol. Whan a participant is registered in a Context with the LRA protocol for the first time, a new LRACompletionHandler is added to the context's completion handlers.

When the Context Service receives a complete message, it will execute every completion handler found in the context passing it the context content and the completion status. Based on this, the completion hanler will:

- Invoke the compensate operation of every participant if the completion status is FAIL.
- Invoke the complete operation of every participant if the completion status is SUCCESS.
- Invoke the forget operation of every participant otherwise.

5.6 Compensator interface

Every compensator service must implement the following operations:

compensate: When called, a compensator service must use the information it has to compensate the work done.

complete: When called, the compensator is notifier of a successfus execution and can free the resources needed to compensate.

forget: If a previous call to compensate failed, a compensator must maintain the compensation information until a call to forget is executed.

6 WS-CAF Manual

6.1 WS-CAF Service

The WS-CAF services implementation (Context Manager, Context Service and Registration Service) is composed of two main parts: A module implementation and the three main services' implementation.

6.1.1 Module

This implementation uses the Axis2 modules mechanism[1] to read and write contexts to the message headers, as well as to provide ways for a service to be compensable. All the module code is in the module directory and is composed of the following parts:

binding contains the XML to Java binding for the different contexts (Basic Context, Registration Context and LRA Context). It's based on JAXB 2.0[2].
The content in this directory is not intended to be modified by hand because a regeneration of the databinding (executing ant generate) will delete the directory and recreate it with the new generated code.

config contains the module.xml file with the module configuration.

schemas contains the wsctx, wscf and wslra schemas used to generate the databindings.

src contains the core implementation of the module (handlers and message receivers).

utils contains helper classes for the module and databinding.

To build the module, there is a build.xml file for ant as well as a build.properties file with some necessary routes defined.

build.properties a valid route to an Axis2 1.3 repository is needed in order to generate and compile the module. To deploy the module and libraries automatically, also the webapp.dir property must point to the location where the Axis2 war file is deployed.

build.xml has some targets to build and deploy the module. The main target, jar-module, will build the .mar file in the dist directory and will leave a file named wscaf-module-common.jar in the lib directory. To deploy the module successfully, the wscaf.mar file must be copied to modules directory in the Axis2 deploy dir. Also, the wscaf-module-common.jar file must be copied to the lib directory in the Axis2 deploy dir as well.
If the webapp.dir has been set in build.properties, then the deploy target will copy those files automatically.

[1] http://ws.apache.org/axis2/1_3/modules.html
[2] https://jaxb.dev.java.net/

In order to use the module, some phases must be defined in te axis2.xml file. A sample file is included in the config directory.

6.1.2 Services

The services folder contains the implementation for the Context Service, Context Manager and Registration Service services. They use the above module so in order to use them the module must be deployed previously. It has the following structure:

codegen contains the Axis2 generated code for all the services. This includes databinding code to transform XML elements to Java elements and the stubs needed to invoke the three main services, as well as a generic compensator stub that is used by the Context Service to invoke compensators when a LRA context is completed. It also contains the services deployment descriptor and generated WSDLs.
The content in this directory is not intended to be modified by hand because a regeneration of the service (executing ant generate) will delete the directory and recreate it with the new generated code.
Due to the way the faults are thrown, its necessary to modify the service's skeleton interface (classes *.<ServiceName>SkeletonInterface) to add a *throws AxisFault* to every operation every time the code is regenerated. Using the Java Eclipse IDE will help in this task as it will detect errors in the skeletons and suggest the solution.

test contains test cases for all the three main services.

utils contains static classes that helps converting elements from Axis2 XML model (Axiom) to XMLBeans model used in the services and the other way around. It also containt helper classes to manipulate Contexts in Axiom format, to generate request messages as well as some miscellaneous utilities used in the main services.

middleware contains the classes implementing the interface of the main services. For every service it has a <ServiceName>Skeleton class that implements the service's interface. It also contains the completion handlers for every protocol. For the moment only LRA is implemented in LRACompletionHandler.

core contains the true implementation of the main services. It has the referred ContextManager, ContextService and RegistrationService classes that implements the logic necessary to manage contexts.

lib contains the jar files that the service needs in order to work.

wsdl contains the wsdl files and associated schemas used to generate the service's interface, databindings and stubs.

The project is built with Apache Ant through the build.xml project descriptor. It has the following main targets:

generate regenerates the codegen directory. This is useful when updating the axis version.

build-all generates binaries for the services and utilities jars. It generates a dist/wscaf-all.aar file that contains the services implementation and must be deployed in Axis2. It also generates lib/wscaf-codegen.jar with the code in the codegen directory and lib/wscaf-utils.jar with the content of the utils directory. These two jars are intented to be used for clients that will invoke the wscaf services, as they contain the stubs needed as well as some useful utilities.

deploy-all-tomcat deploys the generated service archive in a servlet container if the webapp.dir property is defined.

As with the module, some properties must be set in the build.properties file:

axis2.dir must point to a local Axis2 1.3 repository.

xmltask.dir must point to a place where the xmltask jar can be found. XMLTask is not included but can be downloaded from http://www.oopsconsultancy.com/software/xmltask/

webapp.dir must point to the place where Axis2 war is deployed. It's only necessary to use the automatic deployment function .

6.2 Context reading and writing

The provided module will handle the Context serialization and deserialization when enganged to a service. Once enganged, it will leave the incomming context in the following properties:

ContextModuleConstants.CurrentContextProperty if the incomming context was a basic context.

ContextModuleConstants.CurrentRegistrationContextProperty if the incomming context was a registration context.

ContextModuleConstants.CurrentLRAContextProperty if the incomming context was a LRA context.

So, for example, the following code:

```
org.oasis.wscaf.bindings.wsctx.ContextType ctx =
  (org.oasis.wscaf.bindings.wsctx.ContextType)MessageContext.
    getCurrentMessageContext().getProperty(
      ContextModuleConstants.CurrentContextProperty);
```

will return in ctx the incomming basic context that was in the message header.

An outgoing context can be put in a header using this method:

```
MessageContext outMessageContext =
  MessageContext.getCurrentMessageContext().
    getOperationContext().getMessageContext(
      WSDLConstants.MESSAGE_LABEL_OUT_VALUE);
outMessageContext.setProperty(
  ContextModuleConstants.OutContextProperty, newCtx);
```

As with the incomming context, the property name varies depending on the context type passed as parameter:

ContextModuleConstants.OutContextProperty for a basic context;

ContextModuleConstants.OutRegistrationContextProperty for a registration context;

ContextModuleConstants.OutLRAContextProperty for a LRA context;

6.3 Compensator services

In order to be compensable, a service's skeleton must implement the interface org.oasis.wscaf.module.utils.Compensator and have the wscaf module enganged.

6.4 Clients

For the clients not implementing the compensator interface, using the wscaf-codegen.jar and optionally wscaf-utils.jar classes should be enough to execute the operations in the three wscaf main services.

6.5 WS-CAF core implementation

The figure 3 shows the relationship between the main classes implementing the WSCAF core specification.

The main class is ContextManager which holds a Hash Table with all the contexts created so far by the Context Service and indexed by context identifier. The ContextService class is invoked by the Context Service service to create new contexts and it puts them in the table through the ContextManager class. It also serves the other operations that the Context Service service exposes, using the ContextManager instance to work over the existing contexts.

The RegistrationService class will be invoked by the Registration Service service and it will use the ContextManager class to add, remove and replace participants from contexts.

To support different context types, a class named ContextWrapper holds a context and the completion handlers associated to that context. When a participant is registered in a protocol for the first time through the addParticipant

operation, the RegistrationContext class will obtain the target ContextWrapper class (the one indexed by the identifier of the operation's context) using the ContextManager instance and it will add a CompletionHandler object to it based on the participant's protocol. Only LRA is implemented for the moment. When this context is completed through the complete() operation in the ContextService instance, it will obtain the same ContextWrapper instance and will execute its registered Completion Handlers.

The information a context has is saved in a class called Context or RegistrationContext depending on the type of context and the content it has. A context is always created as a Context instance but once the first element of a RegistrationContext is added (being it a Registration Service endpoint reference or a participant group) it is converted to a RegistrationContext instance.

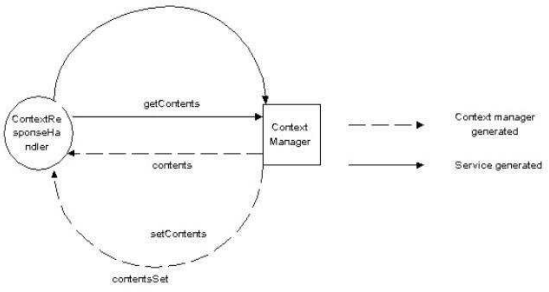

Figure 1: Context Manager messages

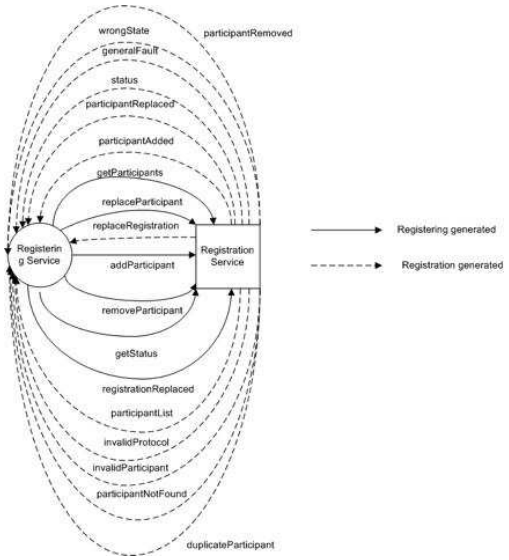

Figure 2: Registration Service messages

Figure 3: WSCAF core implementation classes