Web Services ACID Specification (WS-ACID)

Editors draft version 0.2

Version created 5 July 2005

Editors

Mark Little (mark.little@arjuna.com)

Eric Newcomer (eric.newcomer@iona.com)

Greg Pavlik (greg.pavlik@oracle.com)

Abstract

An increasing number of applications are being constructed by combining or coordinating the execution of multiple Web services, each of which may represent an interface to a different underlying technology. The resulting applications can be very complex in structure, with complex relationships between their constituent services. Furthermore, the execution of such an application may take a long time to complete, and may contain long periods of inactivity, often due to the constituent services requiring user interactions. In the loosely coupled environment represented by Web services, long running applications will require support for recovery and compensation, because machines may fail, processes may be cancelled, or services may be moved or withdrawn. Web services transactions also must span multiple transaction models and protocols native to the underlying technologies onto which the Web services are mapped.

A common technique for fault-tolerance is through the use of atomic transactions, which have the well know ACID properties, operating on persistent (long-lived) objects. Transactions ensure that only consistent state changes take place despite concurrent access and failures. However, traditional transactions depend upon tightly coupled protocols, and thus are often not well suited to more loosely-coupled Web services based applications, although they are likely to be used in some of the constituent technologies. It is more likely that traditional transactions are used in the minority of cases in which the cooperating Web services can take advantage of them, while new mechanisms, such as compensation, replay, and persisting business process state, more suited to Web services are developed and used for the more typical case.

WS-TXM provides a suite of transaction models, each suited to solving a different problem domain. However, because WS-TXN leverages WS-CF, it is intended to allow flexibility in the types of models supported. Therefore, if new models are required for other problem areas, they can be incorporated within this specification.

Table of contents

1 Note on terminology

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC2119 [2].

Namespace URIs of the general form http://example.org and http://example.com represents some application-dependent or context-dependent URI as defined in RFC 2396 [3].

1.1 Namespace

The XML namespace URI that MUST be used by implementations of this specification is:

```
http://docs.oasis-open.org/wscaf/2005/03/wsacid
```

1.1.1 Prefix Namespace

Prefix	Namespace
wscf	**http://docs.oasis-open.org/wscaf/2005/02/wscf**
wsctx	**http://docs.oasis-open.org/wscaf/2004/09/wsctx**
wsacid	**http://docs.oasis-open.org/wscaf/2005/07/wsacid**
ref	**http://docs.oasisopen.org/wsrm/2004/06/reference-1.1**
wsdl	**http://schemas.xmlsoap.org/wsdl/**
xsd	**http://www.w3.org/2001/XMLSchema**
wsu	**http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd**
tns	**targetNamespace**

Comment: Kevin, can you check these are right (dates)?

1.2 Referencing Specifications

One or more other specifications may reference the WS-ACID specification. The usage of optional items in WS-ACID is typically determined by the requirements of such as referencing specification.

A referencing specification generally defines the protocol types based on WS-ACID. Any application that uses WS-ACID must also decide what optional features are required. For the purpose of this document, the term *referencing specification* covers both formal specifications and more general applications that use WS-ACID.

2 Architecture

Atomic transactions are a well-known technique for guaranteeing consistency in the presence of failures [10]. The ACID properties of atomic transactions (Atomicity, Consistency, Isolation, and Durability) ensure that even in complex business applications consistency of state is preserved, despite concurrent accesses and failures. This is an extremely useful fault-tolerance technique, especially when multiple, possibly remote, resources are involved.

WS-ACID leverages the WS-CF and WS-Context specifications. Figure 4 illustrates the layering of WS-ACID onto WS-CF. WS-ACID defines a pluggable transaction protocol that can be used with the coordinator to negotiate a set of actions for all participants to execute based on the outcome of a series of related Web services executions. The executions are related through the use of shared context. Examples of coordinated outcomes include the classic two-phase commit protocol, a three phase commit protocol, open nested transaction protocol, asynchronous messaging protocol, or business process automation protocol.

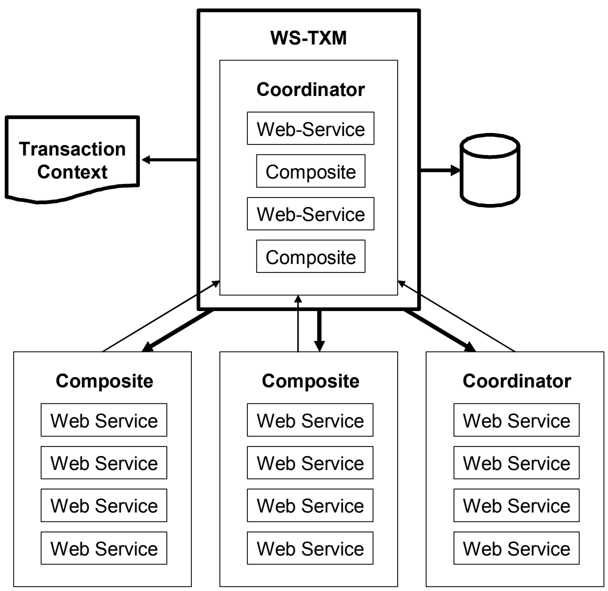

Figure 1, Relationship of transactions to coordination framework.

Coordinators can be participants of other coordinators, as shown above. When a coordinator registers itself with another coordinator, it can represent a series of local activities and map a neutral transaction protocol onto a platform-specific transaction protocol.

2.1 Invocation of Service Operations

How application services are invoked is outside the scope of this specification: they MAY use synchronous or asynchronous message passing.

Irrespective of how remote invocations occur, context information related to the sender’s activity needs to be referenced or propagated. This specification determines the format of the context, how it is referenced, and how a context may be created.

In order to support both synchronous and asynchronous interactions, the components are described in terms of the behavior and the interactions that occur between them. All interactions are described in terms of correlated messages, which a referencing specification MAY abstract at a higher level into request/response pairs.

Faults and errors that may occur when a service is invoked are communicated back to other Web services in the activity via SOAP messages that are part of the standard protocol. To achieve this, the fault mechanism of the underlying SOAP-based transport is used. For example, if an operation fails because no activity is present when one is required, then the callback interface will receive a SOAP fault including type of the fault and additional implementation specific information items supported the SOAP fault definition. WS-Context specific fault types are described for each operation. A fault type is communicated as an XML QName; the prefix consists of the WS-Context namespace and the local part is the fault name listed in the operation description.

> Note, a transientFault message is produced when the implementation finds it cannot successfully execute the requested operation at that time from some *temporary* reason. This reason may be implementation or referencing specification specific. A receiver of a transientFault is free to retry the operation which originally generated it on the assumption that eventually a different response will be produced. Sub-types of transientFault MAY be further defined using the fault model described which can allow for the communication of more specific information on the type of fault.

As long as implementations ensure that the on-the-wire message formats are compliant with those defined in this specification, how the end-points are implemented and how they expose the various operations (e.g., via WSDL [1]) is not mandated by this specification. However, a normative WSDL binding is provided by default in this specification.

> Note, this specification does not assume that a reliable message delivery mechanism has to be used for message interactions. As such, it MAY be implementation dependant as to what action is taken if a message is not delivered or no response is received.

2.2 Relationship to WSDL

Where WSDL is used in this specification it uses one-way messages with callbacks. This is the normative style. Other binding styles are possible (perhaps defined by referencing specifications), although they may have different acknowledgment styles and delivery mechanisms. It is beyond the scope of WS-ACID to define these styles.

> Note, conformant implementations MUST support the normative WSDL defined in the specification where those respective interfaces are required. WSDL for optional components in the specification is REQUIRED only in the cases where the respective components are supported.

For clarity WSDL is shown in an abbreviated form in the main body of the document: only portTypes are illustrated; a default binding to SOAP 1.1-over-HTTP is also assumed as per [1].

2.3 Referencing and addressing conventions

There are multiple mechanisms for addressing messages and referencing Web services currently proposed by the Web services community. This specification defers the rules for addressing SOAP messages to existing specifications; the addressing information is assumed to be placed in SOAP headers and respect the normative rules required by existing specifications.

However, the Coordination Framework message set requires an interoperable mechanism for referencing Web Services. For example, context structures may reference the service that is used to manage the content of the context. To support this requirement, WS-CAF has adopted an open content model for service references as defined by the Web Services Reliable Messaging Technical Committee [5]. The schema is defined in [6][7] and is shown in Figure 1.

```
<xsd:complexType name="ServiceRefType">
    <xsd:sequence>
      <xsd:any namespace="##other" processContents="lax"/>
    </xsd:sequence>
    <xsd:attribute name="reference-scheme" type="xsd:anyURI"
        use="optional"/>
  </xsd:complexType>
```

Figure 2, service-ref Element

The ServiceRefType is extended by elements of the context structure as shown in Figure 2.

```
<xsd:element name="context-manager" type="ref:ServiceRefType"/>
```

Figure 3, ServiceRefType example.

Within the **ServiceRefType**, the reference-scheme is the namespace URI for the referenced addressing specification. For example, the value for WSRef defined in the WS-MessageDelivery specification [4] would be http://www.w3.org/2004/04/ws-messagedelivery. The value for WSRef defined in the WS-Addressing specification [8] would be http://schemas.xmlsoap.org/ws/2004/08/addressing. The reference scheme is optional and need only be used if the namespace URI of the QName of the Web service reference cannot be used to unambiguously identify the addressing specification in which it is defined.

Messages sent to referenced services MUST use the addressing scheme defined by the specification indicated by the value of the reference-scheme element if present. Otherwise, the namespace URI associated with the Web service reference element MUST be used to determine the required addressing scheme. A service that requires a service reference element MUST use the mustUnderstand attribute for the SOAP header element within which it is enclosed and MUST return a mustUnderstand SOAP fault if the reference element isn't present and understood.

> Note, it is assumed that the addressing mechanism used by a given implementation supports a reply-to or sender field on each received message so that any required responses can be sent to a suitable response endpoint. This specification requires such support and does not define how responses are handled.

To preserve interoperability in deployments that contain multiple addressing schemes, there are no restrictions on a system, beyond those of the composite services themselves. However, it is RECOMMENDED where possible that composite applications confine themselves to the use of single addressing and reference model.

Because the prescriptive interaction pattern used by WS-ACID is based on one-way messages with callbacks, it is possible that an endpoint may receive an unsolicited or unexpected message. The recipient is free to do whatever it wants with such messages.

3 WS-ACID

The *ACID transaction* model recognizes that Web Services are for interoperability as much as for the Internet. As such, interoperability of existing transaction processing systems will be an important part of Web Services Transaction Management: such systems already form the backbone of enterprise level applications and will continue to do so for the Web Services equivalent. Business-to-business activities will typically involve back-end transaction processing systems either directly or indirectly and being able to tie together these environments will be the key to the successful take-up of Web Services transactions.

Although ACID transactions may not be suitable for all Web Services, they are most definitely suitable for some, and particularly high-value interactions such as those involved in finance. As a result, the ACID transaction model has been designed with interoperability in mind. Within this model it is assumed that all services (and associated participants) provide ACID semantics and that any use of atomic transactions occurs in environments and situations where this is appropriate: in a trusted domain, over short durations.

In the ACID model, each activity is bound to the scope of a transaction, such that the end of an activity automatically triggers the termination (commit or rollback) of the associated transaction. The coordinator-type URI for the ACID transaction model is http://www.webservicestransactions.org/wsdl/wstxm/tx-acid/2003/03

Comment: Update.

3.1 Restrictions imposed on using WS-CF

As a Referencing Specification, the WS-ACID transaction model imposes the following restrictions on using WS-CF:

- It is illegal to attempt to remove a participant from a transaction at any time. When the transaction terminates, participants are implicitly removed. As such, any attempt to call *removeParticipant* will result in the *wrongState* error message being returned.

3.2 Two-phase commit

The ACID transaction model uses a traditional two-phase commit protocol [2] with the following optimizations:

- *Presumed rollback*: the transaction coordinator need not record information about the participants in stable storage until it decides to commit, i.e., until after the prepare phase has completed successfully.
- *One-phase*: if the coordinator discovers that only a single participant is registered then it SHOULD omit the prepare phase..
- *Read-only*: a participant that is responsible for a service that did not modify any transactional data during the course of the transaction can indicate to the coordinator during prepare that it is a *read-only participant* and the coordinator SHOULD omit it from the second phase of the commit protocol.

Participants that have successfully passed the *prepare* phase are allowed to make autonomous decisions as to whether they commit or rollback. A participant that makes such an autonomous choice *must* record its decision in case it is eventually contacted to complete the original transaction. If the coordinator eventually informs the participant of the fate of the transaction and it is the same as the autonomous choice the participant made, then there is obviously no problem: the participant simply got there before the coordinator did. However, if the decision is contrary, then a non-atomic outcome has happened: a *heuristic outcome*, with a corresponding *heuristic decision*.

The possible heuristic outcomes are:

- *Heuristic rollback*: the commit operation failed because some or all of the participants unilaterally rolled back the transaction.
- *Heuristic commit*: an attempted rollback operation failed because all of the participants unilaterally committed. This may happen if, for example, the coordinator was able to successfully prepare the transaction but then decided to roll it back (e.g., it could not update its log) but in the meanwhile the participants decided to commit.
- *Heuristic mixed*: some updates were committed while others were rolled back.
- *Heuristic hazard*: the disposition of some of the updates is unknown. For those which are known, they have either all been committed or all rolled back.

3.2.1 State transitions and relationship to WS-Context

WS-ACID is a referencing specification for WS-CF and hence leverages the *activity group* concept. When an application creates a new activity group (by sending a **wsctx:begin** message to the relevant Context Service), an associated WS-ACID coordinator MAY be created in the Active state, as shown in Figure 4.

Deleted: Figure 45

> Note, participants enlisted with a WS-ACID activity group progress through the same state transitions.

The coordinator has the lifetime period associated with the activity: if the activity timeout elapses before the activity has terminated, then the transaction will be terminated in the RolledBack state.

A transactional activity can be terminated via the wsctx:complete message in one of two ways:

- Committed: the transaction commits.

Comment: We need to decide what the commit/rollback data within the complete message looks like.

- Rollback: the transaction rolls back.

If the transaction is instructed to commit then the application sends an appropriate **wsctx:complete** message to the Context Service. If there is only a single participant enrolled with the transaction then the coordinator SHOULD use the one-phase commit optimization. As such, the coordinator begins the *OnePhaseCommit* protocol and either transits to the *RolledBack* or *Committed* state, depending upon the result returned by the participant. **The activity completion status is either *Failure* or *Success* respectively.**

Comment: Open issue: how are errors communicated back to the application via the wsctx:completed message?

If there are multiple participants enrolled with the transaction, the coordinator transits to the *Preparing* state and begins to execute the two-phase commit protocol by sending the **wsacid:prepare** message to each participant. If all of the participants indicate that the services they represent performed no work (i.e., are read only) then the transaction is complete and the coordinator transits to the *Committed* state.

Any failures from a participant or indication that it cannot prepare cause the coordinator to rollback (move to the *RollingBack* state) and send wsacid:rollback messages to all of the participants. It then transits to the *RolledBack* state.

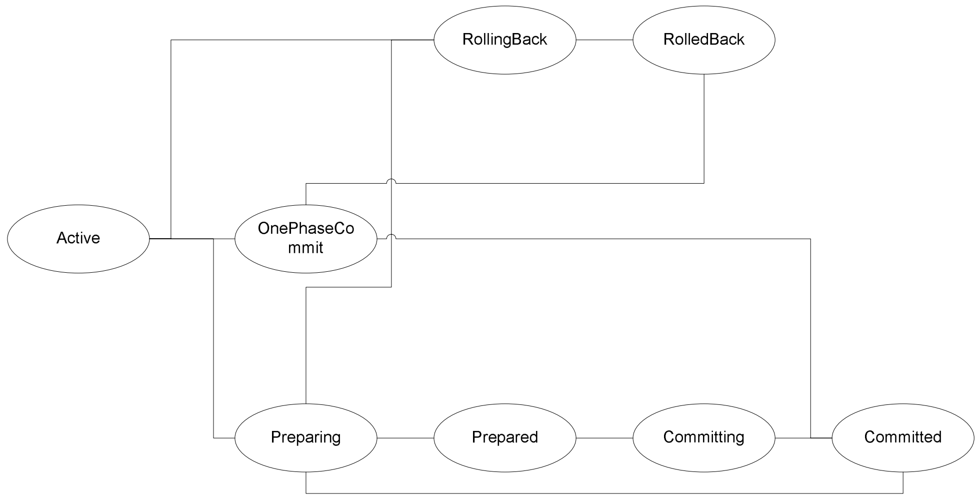

Figure 4, Transaction coordinator two-phase status transition.

Assuming all participants have prepared successfully, the transaction coordinator makes the decision as to whether to commit or rollback and must record sufficient information on stable storage to ensure this decision can be completed in the event of a failure. It is then in the *Prepared* state. When the coordinator starts the second phase of the commit protocol it is in the *Committing* state and ultimately moves to the *Committed* state.

3.2.2 Two-phase commit message interactions

In this section we shall describe the message exchanged between the coordinator and the participants. Although the text refers to the coordinator soliciting responses from participants, in some cases participants MAY send unsolicited responses to the coordinator;.where this is the case it will be explicitly stated.

The ACID transaction model supports two styles of participant service implementation: the *singleton* approach, whereby one participant service (end-point) is implicitly associated with only one transaction, and the *factory* approach, whereby a single participant service may manage participants on behalf of many different transactions. Therefore, all operations on the participant service are associated with the current context, i.e., it is propagated to the participants in order to identify which transaction is to be operated on. The unique participant identification is also present on each message.

The two-phase commit sub-protocol URI is http://www.webservicestransactions.org/wsdl/wstxm/tx-acid/2pc/2003/03 and this is used in the *addParticipant* message. An enlisted Participant Service should expect to receive the following messages (illustrated in Figure 5):

- prepare: The coordinator is preparing. The participant can respond with a *voteReadonly*, *voteCommit* or *voteRollback* messages indicating whether or not it is willing to commit. **If *voteCommit* is used then optional Qualifiers may be sent back to augment the protocol**. The voteReadonly and voteRollback messages MAY be sent autonomously by the participant, i.e., before any **wsacid:prepare** message is received. However, the participant SHOULD be able to deal with a subsequent **wsacid:prepare** message. If an unreliable transport mechanism is used, then there may be an arbitrary number of these messages. If the participant is a subordinate coordinator and finds that it cannot determine the status of some of its enlisted participants then an error message with the **wsacid:HeuristicHazardFault** error code will be returned. Alternatively, if a subordinate

Comment: Need to update.

Deleted: Figure 58

Comment: Issue 275

coordinator finds that some of the participants have committed and some have rolled back then it must return the **wsacid:HeuristicMixedFault** error message.

Comment: SOAP faults

- rollback: The coordinator is rolling back. If the participant is receiving this message after a **wsacid:prepare** message, then any error at this point will cause a heuristic outcome. If the participant is a subordinate coordinator and cannot determine how all of its enlisted participants terminated then it must return an error message with the **wsacid:HeuristicHazardFault** fault code. If the participant is a subordinate coordinator and some of its enlisted participants committed then it must return the **wsacid:HeuristicMixedFault** fault code. If the participant commits rather than rolls back then it must return the **wsacid:HeuristicCommitFault** message. Otherwise the participant sends the *rolledback* message. The **wsacid:rolledback** message MAY be sent autonomously by the participant, i.e., before any **wsacid:rollback** message is received. However, the participant SHOULD be able to deal with a subsequent **wsacid:rollback** message. If an unreliable transport mechanism is used, then there may be an arbitrary number of these messages.
- commit: The coordinator is top-level and is committing. Any error at this point will cause a heuristic outcome. If the participant is a subordinate coordinator and cannot determine how all of its enlisted participants terminated then it must return an error message with the **wsacid:HeuristicHazardFault** fault code. If the participant is a subordinate coordinator and some of its enlisted participants committed then it must return the **wsacid:HeuristicMixedFault** fault code. If the participant rolls back rather than commits then it must return the **wsacid:HeuristicRollbackFault** fault code. Otherwise the participant returns a *committed* message.
- onePhaseCommit: If only a single participant is registered with a two-phase coordinator then the coordinator SHOULD optimize the commit stage by not executing the prepare phase. If the participant is a subordinate coordinator and cannot determine how all of its enlisted participants terminated then it must return an error message with the **wsacid:HeuristicHazardFault** fault code. If the participant is a subordinate coordinator and some of its enlisted participants committed then it must return the **wsacid:HeuristicMixedFault** fault code. If the participant rolls back rather than commits then it must return the **wsacid:HeuristicRollbackFault** fault code. Otherwise the participant returns either the *committed* or *rolledback* message.
- forgetHeuristic: The participant made a post-prepare choice that was contrary to the coordinator's outcome. Hence it may have caused a non-atomic (heuristic) outcome. If this happens, the participant *must* remember the decision it took (persistently) until the coordinator tells it via this message that it is safe to forget. Success is indicated by sending the *heuristicForgotten* message. Any other response is assumed to indicate a failure.

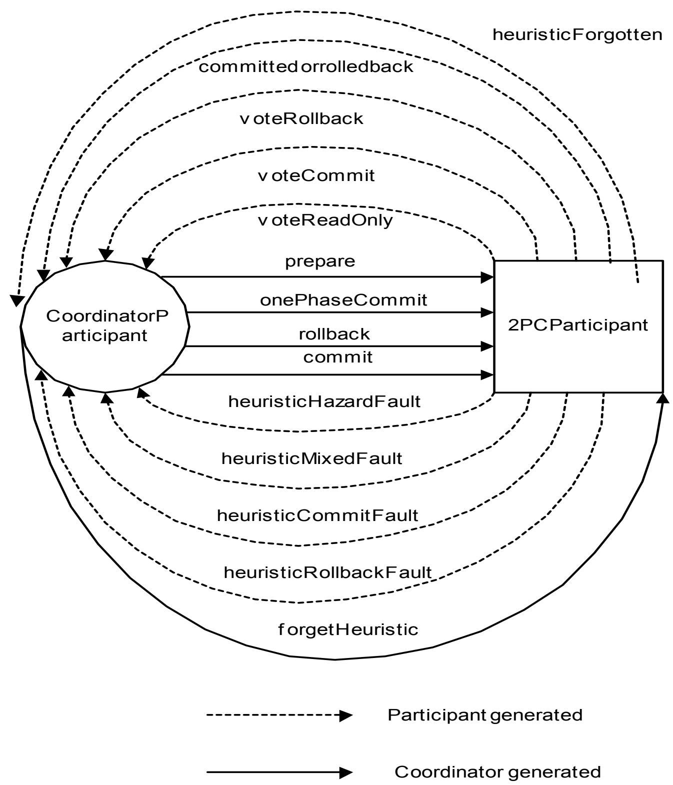

Figure 5, AT coordinator-to-participant message exchanges.

The WSDL portType declarations for the CoordinatorParticipant and twoPCParticipant roles are shown in Figure 6.

Comment: Need to change the names.

Deleted: Figure 69

```
<wsdl:portType name="twoPCParticipantPortType">
  <wsdl:operation name="prepare">
    <wsdl:input message="tns:PrepareMessage"/>
  </wsdl:operation>
  <wsdl:operation name="onePhaseCommit">
    <wsdl:input message="tns:OnePhaseCommitMessage"/>
  </wsdl:operation>
  <wsdl:operation name="rollback">
    <wsdl:input message="tns:RollbackMessage"/>
  </wsdl:operation>
  <wsdl:operation name="commit">
    <wsdl:input message="tns:CommitMessage"/>
  </wsdl:operation>
  <wsdl:operation name="forgetHeuristic">
    <wsdl:input message="tns:ForgetHeuristicMessage"/>
  </wsdl:operation>
</wsdl:portType>
<wsdl:portType name="CoordinatorParticipantPortType">
  <wsdl:operation name="committed">
    <wsdl:input message="tns:CommittedMessage"/>
  </wsdl:operation>
  <wsdl:operation name="rolledBack">
    <wsdl:input message="tns:RolledBackMessage"/>
  </wsdl:operation>
  <wsdl:operation name="vote">
```

```
      <wsdl:input message="tns:VoteMessage"/>
    </wsdl:operation>
    <wsdl:operation name="heuristicForgotten">
      <wsdl:input message="tns:HeuristicForgottenMessage"/>
    </wsdl:operation>
    <wsdl:operation name="heuristicFault">
      <wsdl:input message="tns:HeuristicFaultMessage"/>
    </wsdl:operation>
  </wsdl:portType>
```

Figure 6, WSDL portType Declarations for Coordinator and 2PCParticipant Roles

Note, although an application Web Service may play the role of a participant, it is not required to.

3.3 Pre- and post- two-phase commit processing

Most modern transaction processing systems allow the creation of participants that do not take part in the two-phase commit protocol, but are informed before it begins and after it has completed. They are called Synchronizations, and are typically employed to flush volatile (cached) state, which may be being used to improve performance of an application, to a recoverable object or database prior to the transaction committing; once flushed, the data will the be controlled by a two-phase aware participant.

The sub-protocol URI for the synchronization protocol is http://www.webservicestransactions.org/wsdl/wstxm/tx-acid/sync/2003/03 and this is used in the *addParticipant* invocation.

Comment: Needs updating.

The message exchanges (ignoring the normal WS-CF coordinator-to-participant message exchanges, including failures) are illustrated in Figure 7:

Deleted: Figure 710

- beforeCompletion: A Synchronization participant is informed that the coordinator it is registered with is about to complete the two-phase protocol and in what state, i.e., committing or rolling back. The failure of the participant at this stage will cause the coordinator to cancel if it is not already doing so.
- afterCompletion: A Synchronization participant is informed that the coordinator it is registered with has completed the two-phase protocol and in what state, i.e., committed or rolled back (via the associated Status). Any failures by the participant at this stage have no affect on the transaction.

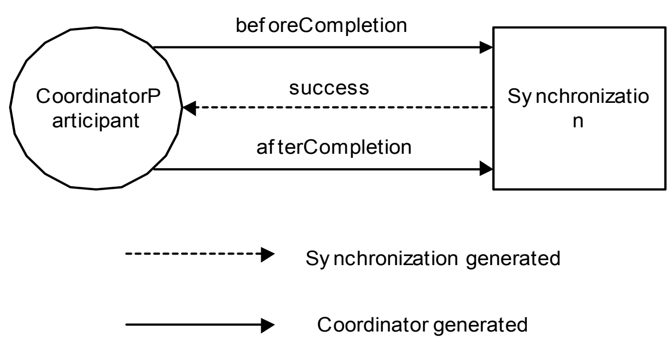

Figure 7, AT coordinator-to-synchronization message exchanges.

The WSDL portType declarations for the CoordinatorParticipant and Synchronization roles are shown in Figure 8.

Deleted: Figure 811

```
<wsdl:portType name="SynchronizationPortType">
  <wsdl:operation name="beforeCompletion">
```

```
    <wsdl:input message="tns:BeforeCompletionMessage"/>
  </wsdl:operation>
  <wsdl:operation name="afterCompletion">
    <wsdl:input message="tns:AfterCompletionMessage"/>
  </wsdl:operation>
</wsdl:portType>
<wsdl:portType name="CoordinatorParticipantPortType">
  <wsdl:operation name="beforeCompletionParticipantRegistered">
    <wsdl:input
message="tns:BeforeCompletionParticipantRegisteredMessage"/>
  </wsdl:operation>
  <wsdl:operation name="afterCompletionParticipantRegistered">
    <wsdl:input
message="tns:AfterCompletionParticipantRegisteredMessage"/>
  </wsdl:operation>
</wsdl:portType>
```

Figure 8, WSDL portType Declarations for Coordinator and 2PCParticipant Roles.

Note, the participant is registered for both beforeCompletion and afterCompletion.

3.3.1 State transitions for synchronization protocol

The state transitions for the transaction coordinator which has enrolled Synchronizations is shown in Figure 12. In this scenario we assume the transaction is committing: if it were to rollback, then only the *AfterCompletion* message will be sent from the coordinator to the Synchronization participants.

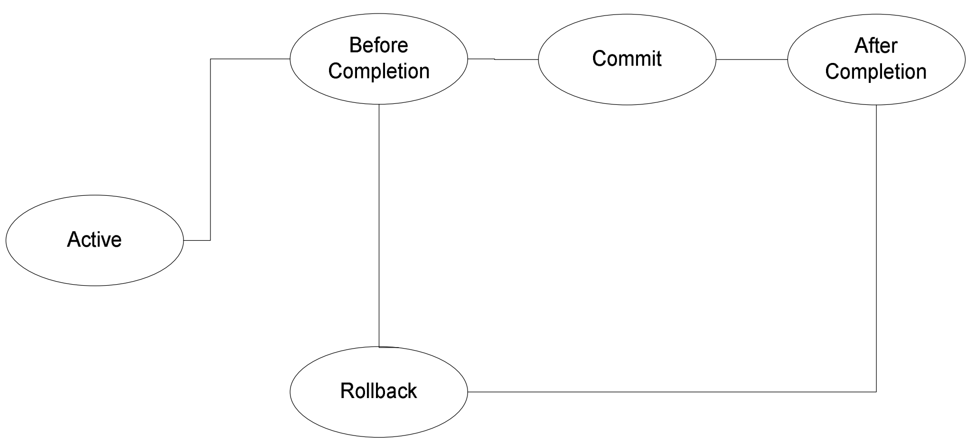

Figure 9, Transaction coordinator Synchronization state transitions.

The coordinator moves into the *BeforeCompletion* state and sends each enrolled Synchronization the *beforeCompletion* message. Any error received by the coordinator from a Synchronization at this stage will force the transaction to rollback. Assuming no errors occur, the two-phase commit protocol is executed, as detailed previously. Once the protocol has completed, the coordinator transits to the *AfterCompletion* status and sends the *afterCompletion* message to all Synchronizations; any errors at this stage do not affect the transaction outcome and how they are dealt with is implementation dependant.

3.4 Recovery and interposition

Because WS-ACID is a Referencing Specification of WS-CF, interposition is allowed though not required. Individual participants may be subordinate coordinators to improve performance or to federate a distributed environment into separate domains (possibly managed by different organizations or transaction management systems).

Each participant or subordinate coordinator is responsible for ensuring that sufficient data is made durable in order to complete the transaction in the event of failures. *Recovering participants or coordinators use the recovery mechanisms defined in WS-CF to determine the current status of a transaction/participant and act accordingly.* Interposition and check pointing of state allow the system to drive a consistent view of the outcome and recovery actions taken, but allowing always the possibility that recovery isn't possible and must be logged or flagged for the administrator.

Although enterprise transaction systems address the aspects of distributed recovery, in a large scale environment or in the presence of long term failures, recovery may not be automatic. As such, manual intervention may be necessary to restore an application's consistency.

3.5 The context

```
<xs:complexType name="ContextType">
  <xs:complexContent>
    <xs:extension base="wstxm:ContextType"/>
  </xs:complexContent>
</xs:complexType>
<xs:element name="context" type="tns:ContextType"/>
```

Figure 10, Transaction Context.

3.6 Statuses

The following extensions to the WS-Context Status type MAY be returned by participants and the Context Service to indicate the outcome of executing relevant parts of the protocol and are also used to indicate the current status of the transaction:

Comment: Issue – need to add a getStatus to the Participant Service WSDL?

- RollbackOnly: the status of the coordinator or participant is that it will rollback eventually.
- RollingBack: the coordinator or participant is in the process of rolling back.
- RolledBack: the coordinator/participant has rolled back. This may be a transient and in fact, because the protocol uses a presumed-abort optimisation, the NoActivity status can be used to infer that the coordinator cancelled.
- Committing: the coordinator/participant is in the process of committing. This does not mean that the final outcome will be Committed.
- Committed: the coordinator/participant has confirmed.
- HeuristicRollback: all of the participants rolled back when they were asked to commit.
- HeuristicCommit: all of the participants committed when they were asked to rollback.
- HeuristicHazard: some of the participants rolled back, some committed and the outcome of others is indeterminate.
- HeuristicMixed: some of the participants rolled back whereas the remainder committed.
- Preparing: the coordinator/participant is preparing.
- Prepared: the coordinator/participant has prepared.

These are specified in the schema, as per Figure 11.

Deleted: Figure 1114

```
<xs:simpleType name="StatusType">
  <xs:restriction base="wstxm:StatusType">
    <xs:enumeration value="activity.status.tx-acid.ROLLBACK_ONLY"/>
    <xs:enumeration value="activity.status.tx-acid.ROLLING_BACK"/>
    <xs:enumeration value="activity.status.tx-acid.ROLLED_BACK"/>
    <xs:enumeration value="activity.status.tx-acid.COMMITTING"/>
    <xs:enumeration value="activity.status.tx-acid.COMMITTED"/>
    <xs:enumeration value="activity.status.tx-
acid.HEURISTIC_ROLLBACK"/>
    <xs:enumeration value="activity.status.tx-acid.HEURISTIC_HAZARD"/>
    <xs:enumeration value="activity.status.tx-acid.HEURISTIC_MIXED"/>
    <xs:enumeration value="activity.status.tx-acid.PREPARING"/>
    <xs:enumeration value="activity.status.tx-acid.PREPARED"/>
  </xs:restriction>
</xs:simpleType>
```

Figure 11, StatusType.

4 References

[1] WSDL 1.1 Specification, see http://www.w3.org/TR/wsdl

[2] "Key words for use in RFCs to Indicate Requirement Levels," RFC 2119, S. Bradner, Harvard University, March 1997.

[3] "Uniform Resource Identifiers (URI): Generic Syntax," RFC 2396, T. Berners-Lee, R. Fielding, L. Masinter, MIT/LCS, U.C. Irvine, Xerox Corporation, August 1998.

[4] WS-Message Delivery Version 1.0, http://www.w3.org/Submission/2004/SUBM-ws-messagedelivery-20040426/

[5] WS-Reliability latest specification, http://www.oasis-open.org/committees/download.php/8909/WS-Reliability-2004-08-23.pdf. See Section 4.2.3.2 (and its subsection), 4.3.1 (and its subsections). Please note that WS-R defines BareURI as the default.

[6] Addressing wrapper schema, http://www.oasis-open.org/apps/org/workgroup/wsrm/download.php/8365/reference-1.1.xsd

[7] WS-R schema that uses the serviceRefType, http://www.oasis-open.org/apps/org/workgroup/wsrm/download.php/8477/ws-reliability-1.1.xsd

[8] Web Services Addressing, see http://www.w3.org/Submission/ws-addressing/

[9] Web Services Security: SOAP Message Security V1.0, http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf

[10] J. N. Gray, “The transaction concept: virtues and limitations”, Proceedings of the 7th VLDB Conference, September 1981, pp. 144-154.

Appendix A. Acknowledgements

The following individuals were members of the committee during the development of this specification:

Appendix B. Notices

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS's procedures with respect to rights in OASIS specifications can be found at the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementors or users of this specification, can be obtained from the OASIS Executive Director.

OASIS invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to implement this specification. Please address the information to the OASIS Executive Director.