Web Services Transaction (WS-Transaction)

9 August 2002

This version:
http://www.ibm.com/developerworks/library/ws-transpec/ (available in PDF)

Authors:
Felipe Cabrera, Microsoft <cabrera@microsoft.com>
George Copeland, Microsoft <gcope@microsoft.com>
Bill Cox, BEA <wcox@bea.com>
Tom Freund, IBM <tjfreund@uk.ibm.com>
Johannes Klein, Microsoft <joklein@microsoft.com>
Tony Storey, IBM <tony_storey@uk.ibm.com>
Satish Thatte, Microsoft <satish@microsoft.com>

Abstract

This specification describes coordination types that are used with the extensible coordination framework described in the WS-Coordination specification. It defines two coordination types: Atomic Transaction (AT) and Business Activity (BA). Developers can use either or both of these coordination types when building applications that require consistent agreement on the outcome of distributed activities.

Composable Architecture

By using the SOAP [SOAP] and WSDL [WSDL] extensibility model, SOAP-based and WSDL-based specifications are designed to work together to define a rich Web services environment. As such, WS-Transaction by itself does not define all features required for a complete solution. WS-Transaction is a building block used with other specifications of Web services (e.g., WS-Coordination, WS-Security) and application-specific protocols that are able to accomodate a wide variety of coordination protocols related to the coordination actions of distributed applications.

Status of this Document

WS-Transaction and related specifications are provided for use as-is and for review and evaluation only. Microsoft, BEA, and IBM will solicit your contributions and suggestions in the near future. Microsoft, BEA, and IBM make no warranties or representations regarding the specification in any manner whatsoever.

Acknowledgements

The following individuals have provided invaluable input into the design of the WS-Transaction specification:

Francisco Curbera, IBM

Gert Drapers, Microsoft

Don Ferguson, IBM

David Langworthy, Microsoft

Frank Leymann, IBM

Jagan Peri, Microsoft

John Shewchuk, Microsoft

Sanjiva Weerawarana, IBM

We also wish to thank the technical writers and development reviewers who provided feedback to improve the readability of the specification.

Table of Contents

---

1. Introduction

The current set of Web service specifications [WSDL][SOAP] defines protocols for Web service interoperability. Web services increasingly tie together a large number of participants forming large distributed applications. The resulting activities can be complex in structure, with complex relationships between their participants.

The WS-Coordination specification defines an extensible framework for defining coordination types. A coordination type can have multiple coordination protocols, each intended to coordinate a different role that a Web service plays in the activity.

To establish the necessary relationships between participants, messages exchanged between participants carry a CoordinationContext. The CoordinationContext includes a Registration service PortReference of a Coordination service. Participants use that Registration service to register for one or more of the protocols supported by that activity.

This specification provides the definition of two coordination types including their respective protocols for:

- An atomic transaction (AT) is used to coordinate activities having a short duration and executed within limited trust domains. They are called atomic transactions because they have an "all or nothing" property. The Atomic Transaction specification defines protocols that enable existing transaction processing systems to wrap their proprietary protocols and interoperate across different hardware and software vendors.
- A business activity (BA) is used to coordinate activities that are long in duration and desire to apply business logic to handle business exceptions. The long duration prohibits locking data resources to make actions tentative and hidden from other applications. Instead, actions are applied immediately and are permanent. The Business Activity specification defines protocols that enable existing business process and work flow systems to wrap their proprietary mechanisms and interoperate across trust boundaries and different vendor implementations.

A Web services application can include both atomic transactions and business activities.

To understand the protocols described in this section, the following assumptions are made:

- The reader is familiar with existing standards for two-phase commit protocols and with commercially-available implementations of such protocols. Therefore this section includes only those details that are essential to understanding the protocols described (see reference section for a list of available specifications).
- The reader is familiar with the WS-Coordination specification that defines the framework for the WS-Transaction coordination protocols.
- The reader is familiar with the WSDL specification, including its HTTP and SOAP binding styles. All WSDL port type definitions provided here assume the existence of corresponding SOAP and HTTP bindings.

1.1. Notational Conventions

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 [KEYWORDS].

Namespace URIs of the general form "some-URI" represents some application-dependent or context-dependent URI as defined in RFC2396 [URI].

2. Part I: Atomic Transaction (AT) - AT1 Introduction

Atomic transactions have an all-or-nothing property. The actions taken prior to commit are only tentative (i.e., not persistent and not visible to other activities). When an application finishes, it requests the coordinator to determine the outcome for the transaction. The coordinator determines if there were any processing failures by asking the participants to vote. If the participants all vote that they were able to execute successfully, the coordinator commits all actions taken. If a participant votes that it needs to abort or a participant does not respond at all, the coordinator aborts all actions taken. Commit makes the tentative actions persistent and visible to other transactions. Abort makes the tentative actions appear as if the actions never happened. Atomic transactions have proven to be extremely valuable for many applications. They provide consistent failure and recovery semantics, so the applications no longer need to deal with the mechanics of determining a mutually agreed outcome decision or to figure out how to recover from a large number of possible inconsistent states.

Atomic Transaction defines protocols that govern the outcome of atomic transactions. It is expected that existing transaction processing systems wrap their proprietary mechanisms and interoperate across different vendor implementations.

This specification leverages WS-Coordination by extending it to define a coordination type to support atomic transactions. It does this by defining the behavior and messages required by the respective coordination protocols.

The WS-Coordination protocols are used for transactions to do the following:

- Create a new atomic transaction CoordinationContext that is associated with a coordinator.
- Add an interposed coordinator to an existing transaction.
- Propagate the CoordinationContext in messages between Web services.
- Register for participation in coordination protocols, depending on the participant's role in the activity. Each of these protocols is one of the coordination protocols of the atomic transaction coordination type defined in this specification. These protocols include Completion, PhaseZero, 2PC (two-phase commit) and OutcomeNotification.

The CoordinationContext and messages defined in this specification include an extensibility element that allows implementation-specific and application-specific information to be included.

The relationship of Atomic Transaction and WS-Coordination are described in Section 2. The coordination protocols for atomic transactions are described in Section 3.

Terms introduced in Part I are explained in the body of the specification and summarized in the **8. AT Glossary**.

2.1. AT1.1 Namespace

The XML namespace [XML-ns] URI that MUST be used by implementations of this specification is:

```
http://schemas.xmlsoap.org/ws/2002/08/wstx
```

The namespace prefix "wstx" used in this specification is associated with this URI. This is also used as the CoordinationContext type for atomic transactions.

The following namespaces are used in this document:

Prefix	Namespace

S	http://www.w3.org/2001/12/soap-envelope
wsu	http://schemas.xmlsoap.org/ws/2002/07/utility
wscoor	http://schemas.xmlsoap.org/ws/2002/08/wscoor
wstx	http://schemas.xmlsoap.org/ws/2002/08/wstx

If an action URI is used then the action URI MUST consist of the wstx namespace URI concatenated with the '#' character and the operation name. For example:

```
           http://schemas.xmlsoap.org/ws/2002/08/wstx#Commit
```

2.2. AT1.2 XSD and WSDL Files

The following links hold the XML schema and the WSDL declarations defined in this document.

http://schemas.xmlsoap.org/ws/2002/08/wstx/wstx.xsd

http://schemas.xmlsoap.org/ws/2002/08/wstx/wstx.wsdl

3. AT2 Using WS-Coordination

This section describes how Atomic Transaction uses WS-Coordination.

3.1. AT2.1 CoordinationContext

An atomic transaction uses the WS-Coordination CoordinationContext with the CoordinationType set to the following URI:

```
   http://schemas.xmlsoap.org/ws/2002/08/wstx
```

The CoordinationContext allows elements to be added via extensibility elements.

The following is an example atomic transaction CoordinationContext. An IsolationLevel element has been added as an example of a proprietary extension.

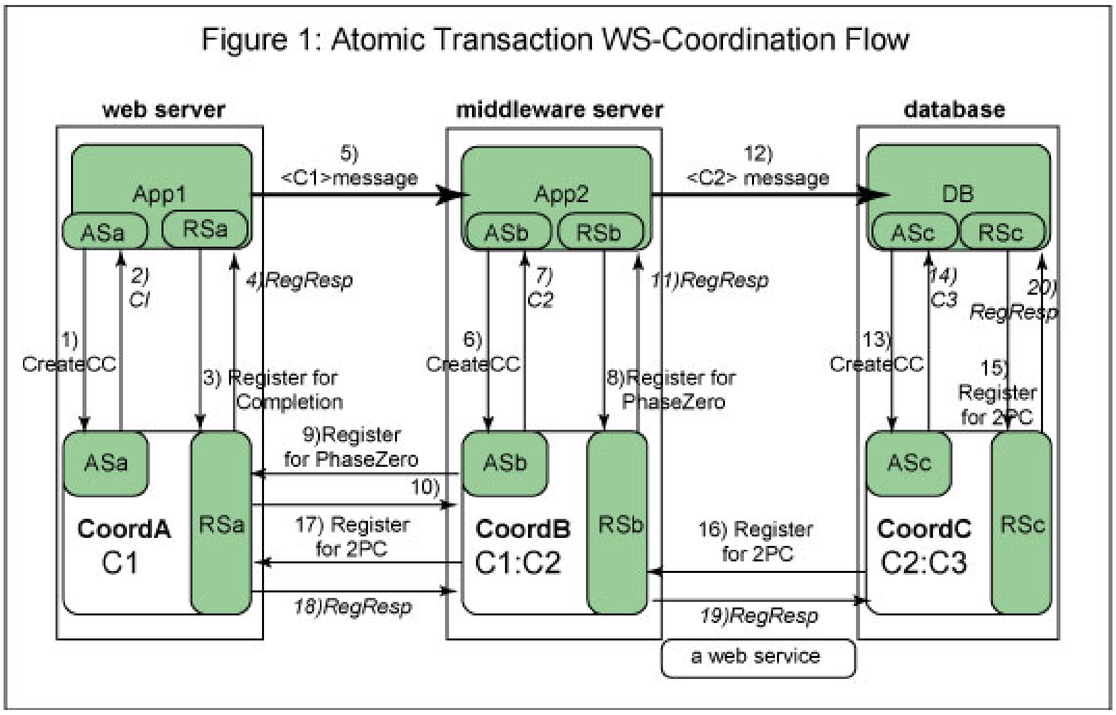
```
   <?xml version="1.0" encoding="utf-8"?>
   <S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope"
       <S:Header>
           . . .
           <wscoor:CoordinationContext
               xmlns:wscoor="http://schemas.xmlsoap.org/ws/2002/08/wscoor"
               xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility"
               xmlns:myApp="http://www.w3.org/2002/08/myApp">
               <wsu:Identifier>http://foobaz.com/SS/1234</wsu:Identifier>
               <wsu:Expires>2002-08-31T13:20:00-05:00</wsu:Expires>
               <wscoor:CoordinationType>
                   http://schemas.xmlsoap.org/ws/2002/08/wstx
               </wscoor:CoordinationType>
               <wscoor:RegistrationService>
                   <wsu:Address>
                   http://myservice.com/mycoordinationservice/registration
                    </wsu:Address>
                   <myApp:BetaMark> ... </myApp:BetaMark>
                   <myApp:EBDCode> ... </myApp:EBDCode>
               </wscoor:RegistrationService>
               <myApp:IsolationLevel>RepeatableRead</myApp:IsolationLevel>
           </wscoor:CoordinationContext>
           . . .
       </S:Header>
       . . .
   </S:Envelope>
```

3.2. AT2.2 CreateCoordinationContext Operation

The CreateCoordinationContext operation semantics depend on the arguments passed in the CreateCoordinationContext message. The cases are:

- When a CurrentContext is not included, a new transaction and its associated protocols are created. The returned CoordinationContext represents the new transaction.
- When a CurrentContext is included, the target coordinator is interposed as the subordinate to the current coordinator. The returned CoordinationContext represents the same transaction but has the PortReference of the interposed coordinator's RegistrationService.

Figure AT1: Atomic Transaction WS-Coordination Flow



4. AT3 Coordination Protocols

This specification defines commit protocols for atomic transactions, which are WS-Coordination protocols. The Coordination protocols for atomic transactions are summarized below with names relative to the base http://schemas.xmlsoap.org/ws/2002/08/wstx:

- **Completion:** One participant (generally the application that created the transaction) registers for the completion protocol, so that it can tell the coordinator either to try to commit the transaction or force a rollback. A status is returned to indicate the final transaction outcome.
- **CompletionWithAck:** Same as Completion, but the coordinator must remember the outcome until receipt of an acknowledgment notification.
- **PhaseZero:** A participant that wants the coordinator to notify it just before the 2PC protocol begins registers for this. A typical example is an application that caches data and needs a notification to write outstanding updates to a database. This is executed prior to the 2PC protocol.
- **2PC:** A participant such as a resource manager (e.g., database) registers for this, so that the coordinator can manage a commit-abort decision across all the resource managers. If more than one 2PC participant is involved,

a PhaseOne and then PhaseTwo are executed. If only one 2PC participant is involved, a OnePhaseCommit is used to delegate the commit-abort decision to the participant.

- **OutcomeNotification:** A transaction participant that wants to be notified of the commit-abort decision registers for this. Applications use outcome notifications to release resources or perform other actions after commit or abort of a transaction.

A participant can register for multiple of these protocols by sending multiple Register messages.

4.1. AT3.1 Example Atomic Transaction Message Flow

Figure AT1 illustrates the interaction flows for the WS-Coordination protocols for an atomic transaction using a common scenario. There are three hosts, each containing part of the application (App1 is a web server that builds web pages, App2 is a middleware server containing business logic and cached data, and DB is a database server). Each has its own coordinator containing the coordinator's side of an Activation service (AS) and a Registration service (RS). Each application service also supports the participant end of the Activation and Registration services.

Figure AT2 illustrates the coordination protocol flows, including the protocol PortReferences that are established in Figure AT1 but not shown in Figure AT1.

App1 begins by doing the following:

- It sends a CreateCoordinationContext message (message 1 in Figure AT1) to its local coordinator's Activation service ASa to create an atomic transaction T1, and gets back in a CreateCoordinationContextResponse message (2) a CoordinationContext C1 containing the transaction identifier T1, the atomic transaction coordination type and CoordA's Coordination PortReference RSa.
- It sends a Register message (3) to RSa to register for the Completion protocol and gets back a RegisterResponse message (4), exchanging protocol service PortReferences for the coordinator and participant sides of the two-way protocol (Ca-cp and Pa-cp are shown in Figure AT2).
- It sends an application message to App2 (5), propagating the CoordinationContext C1 as a header named wscoor:CoordinationContext in the message.

App2 does the following:

- Instead of using CoordA, it decides to interpose its local coordinator CoordB in front of CoordA, which acts as a proxy to CoordA for App2, so that CoordA is the superior and CoordB is the subordinate. It does this by sending a CreateCoordinationContext message (6) to the Activation service of CoordB (ASb) with C1 as input, and getting back (7) a new CoordinationContext C2 that contains the same transaction identifier (T1) and coordination type, but has CoordB's Coordination PortReference RSb.
- It registers with CoordB for the PhaseZero protocol (8 and 11), because it caches data from the database DB, exchanging protocol service PortReferences for the coordinator and participant sides of the two-way protocol.
- CoordB registers with CoordA for the PhaseZero protocol (9 and 10), exchanging protocol service PortReferences for the coordinator and participant sides of the two-way protocol.
- It sends a message to DB (12), propagating CoordinationContext C2.

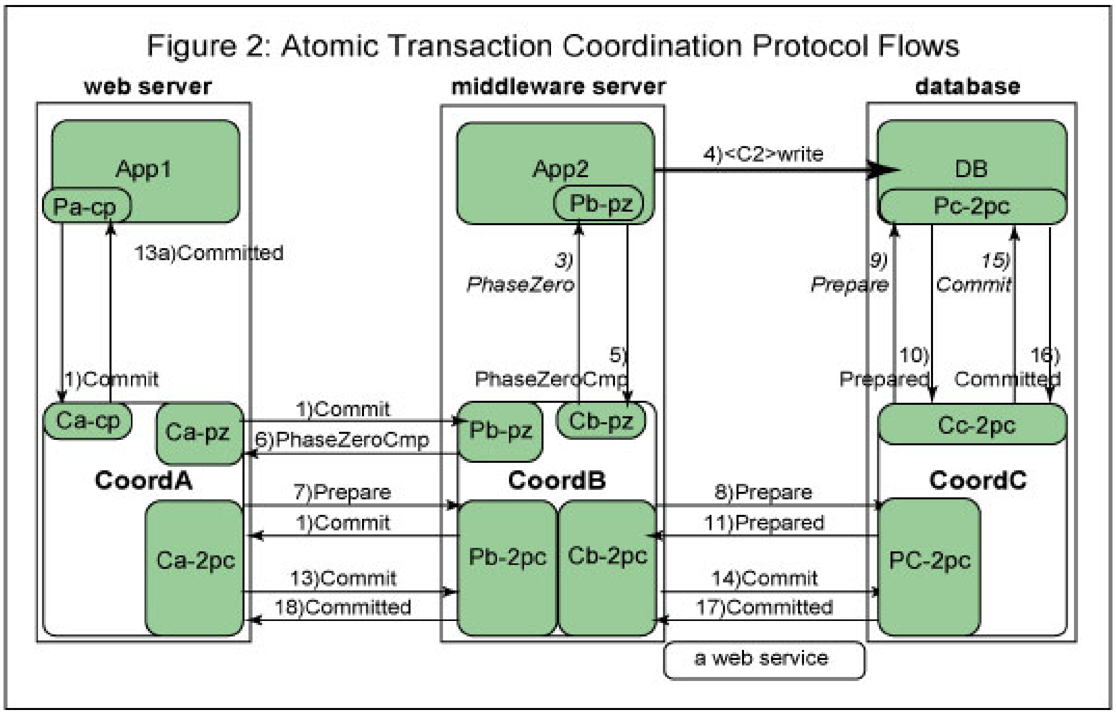
DB does the following:

- Instead of using CoordB, it decides to interpose its local coordinator CoordC by sending a CreateCoordinationContext message (13), further extending the superior-subordinate chain. When it makes this association, it gets back (14) a new CoordinationContext C3 that contains the same transaction identifier (T1) and coordination type, but CoordC's Registration service PortReference RSc.
- It registers with CoordC for the 2PC protocol because it is a resource manager (15 and 20), exchanging protocol service PortReferences for the coordinator and participant sides of the two-way protocol.
- This causes CoordC to register with CoordB for the 2PC protocol (16 and 19), exchanging protocol service PortReferences for the coordinator and participant sides of the two-way protocol (Cb-2pc and Pc-2pc are shown in Figure AT2).
- This causes CoordB to register with CoordA for the 2PC protocol (17 and 18), exchanging protocol service PortReferences for the coordinator and participant sides of the two-way protocol (Ca-2pc and Pb-2pc are shown in Figure AT2).

At this point the coordinators know all the participants, what Coordination protocols they expect to use, and the protocol PortReferences needed by the Coordination protocols.

Figure AT2 illustrates the coordination protocol flows for normal completion of the application.

Figure AT2: Atomic Transaction Coordination Protocol Flows



App1 triggers the Coordination protocols:

- It tries to commit the transaction using the Completion protocol (message 1 in Figure AT2).

CoordA subsequently executes the Coordination protocols. The PhaseZero protocol is first.

- CoordA has 1 participant registered for PhaseZero (CoordB), so it sends a PhaseZero message (2) to CoordB's PhaseZero Participant protocol service Pb-pz.
- CoordB relays the PhaseZero message to App2 (3), so it can flush outstanding cached updates for T1 to DB.
- App2 sends its cached updates to DB. The application message (4) propagates the CoordinationContext C2, so that the updates are made under the same transaction. When App2 knows this has completed, it sends a PhaseZeroCompleted message (5) to CoordB.

After returning successfully from PhaseZero (6), CoordA begins the 2PC protocol:

- CoordA has 1 participant registered for 2PC (CoordB), so it sends a Prepare message (7) to CoordB's 2PC Participant protocol service Pb-2pc.
- CoordB has 1 participant registered for 2PC (CoordC), so it sends a Prepare message (8) to CoordC's 2PC Participant protocol service Pc-2pc.
- CoordC tells DB to prepare (9), which means that DB will be capable of either committing or aborting later, depending on the outcome decision.

After returning successfully from PhaseOne (10, 11 and 12), CoordA commits and makes the decision durable. At this point, the transaction is committed, and the participants need to be notified, which includes both 2PC and Completion participants. The commit notification travels through the same path as the prepare flow:

- CoordA sends the Commit message (13) to CoordB. The Committed notification to App1 (13a) can also be sent at this point.
- CoordB sends the Commit message (14) to CoordC

- CoordC tells DB to commit T1. When DB receives the Commit message (15), it commits.
- When the Committed message returns (16, 17 and 18), the two-party protocol has ended.

If DB had not been able to prepare, the flows would be the same example as in Figure AT2, except for the following:

- The Prepared notification messages (10, 11 and 12) would be replaced by Aborted notification messages.
- Since the participant reported Aborted, the status notification messages for that participant (13 through 18) would be unnecessary. The coordinator would send Abort notification messages to other participants who voted Prepared.

Figure AT2 does not use the OnePhaseCommit optimization. If this optimization were used in all three coordinators, OnePhaseCommit messages would be sent instead of a Prepare messages (7, 8 and 9), delegating the commit-abort decision all the way to the DB. When DB decided to commit or abort, it would make the decision durable and return the status through the full return path to App1.

If a Web service is not a resource manager but wants to vote on the outcome of a transaction, it can register for the 2PC protocol and respond to PhaseOne with either ReadOnly (vote for commit) or Aborted (vote for abort).

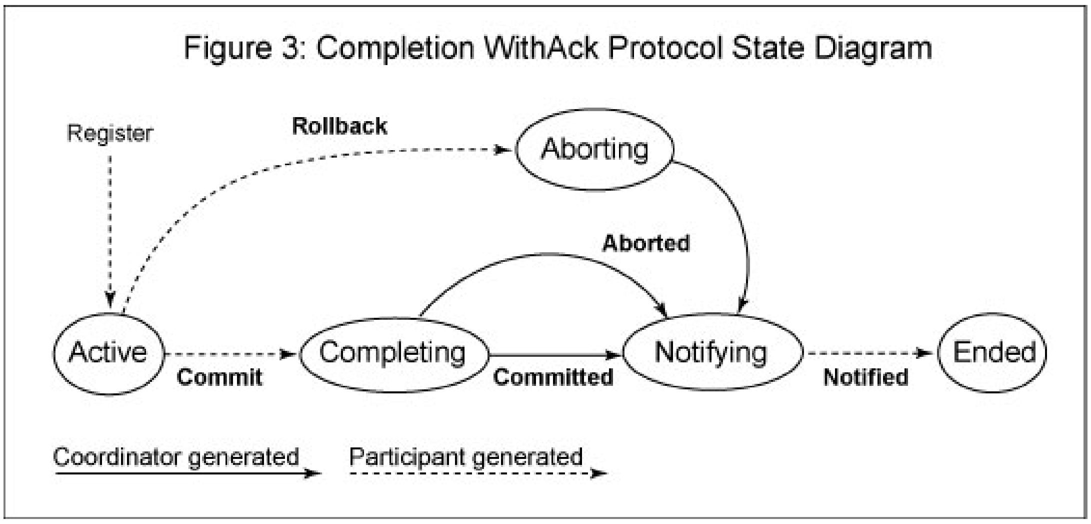
The problem of deciding that all the actions requested as part of a transaction have completed is not part of this specification. Instead, it is the responsibility of the application to determine this prior to attempting to commit or rollback the transaction.

4.2. AT3.2 CompletionWithAck Protocol

The CompletionWithAck protocol is used by an application to tell the coordinator to either try to commit or abort an atomic transaction. After the transaction has completed, a status is returned to the application.

The state diagram in Figure AT3 specifies the behavior of the protocol between coordinator and one of its participants. The state reflects what both sides know of their relationship. Omitted are details such as resending of messages or the exchange of error messages due to protocol error.

Figure AT3: CompletionWithAck Protocol State Diagram



The participant sends the **Commit**, **Rollback** and **Notified** messages.

The coordinator sends the **Committed** and **Aborted** outcome status messages.

The participant begins by sending a **Commit** or **Rollback** request, putting the protocol in the Completing or Aborting state. The only outcome of the Aborting state is for the coordinator to send an **Aborted** status message. While in the Completing state, the coordinator can decide on either commit or abort, returning a **Committed** or **Aborted** status message. Regardless of the outcome, a **Notified** acknowledgement is needed before the coordinator can forget about the transaction.

The coordinator's CompletionWithAck service is defined as:

```
        <wsdl:portType name="CompletionWithAckCoordinatorPortType">
            <wsdl:operation name="Commit">
                <wsdl:input message="wstx:Commit" />
            </wsdl:operation>
            <wsdl:operation name="Rollback">
                <wsdl:input message="wstx:Rollback" />
            </wsdl:operation>
            <wsdl:operation name="Notified">
                <wsdl:input message="wstx:Notified" />
            </wsdl:operation>
            <wsdl:operation name="Unknown">
                <wsdl:input message="wstx:Unknown" />
            </wsdl:operation>
            <wsdl:operation name="Error">
                <wsdl:input message="wstx:Error" />
            </wsdl:operation>
        </wsdl:portType>
```

The participant's CompletionWithAck service is defined as:

```
    <wsdl:portType name="CompletionWithAckParticipantPortType">
            <wsdl:operation name="Committed">
                <wsdl:input message="wstx:Committed" />
            </wsdl:operation>
            <wsdl:operation name="Aborted">
                <wsdl:input message="wstx:Aborted" />
            </wsdl:operation>
            <wsdl:operation name="Error">
                <wsdl:input message="wstx:Error" />
            </wsdl:operation>
        </wsdl:portType>
```

A party should be prepared to receive duplicate notifications and respond back to the source in a manner consistent with the current state of the target.

If a party receives a notification protocol message for an Unknown transaction, it should transmit an **Unknown** notification back to the source.

A complete definition of all messages is provided at the end in .

4.3. AT3.3 Completion Protocol

The Completion protocol is the same as the CompletionWithAck protocol, except that the Notifying state is bypassed and the **Notified** message is not used.

The coordinator's Completion service is defined as:

```
        <wsdl:portType name="CompletionCoordinatorPortType">
            <wsdl:operation name="Commit">
                <wsdl:input message="wstx:Commit" />
            </wsdl:operation>
            <wsdl:operation name="Rollback">
                <wsdl:input message="wstx:Rollback" />
            </wsdl:operation>
            <wsdl:operation name="Unknown">
                <wsdl:input message="wstx:Unknown" />
            </wsdl:operation>
            <wsdl:operation name="Error">
                <wsdl:input message="wstx:Error" />
            </wsdl:operation>
        </wsdl:portType>
```

The participant's Completion service is defined as:

```
<wsdl:portType name="CompletionParticipantPortType">
        <wsdl:operation name="Committed">
            <wsdl:input message="wstx:Committed" />
        </wsdl:operation>
        <wsdl:operation name="Aborted">
            <wsdl:input message="wstx:Aborted" />
        </wsdl:operation>
        <wsdl:operation name="Error">
            <wsdl:input message="wstx:Error" />
        </wsdl:operation>
    </wsdl:portType>
```

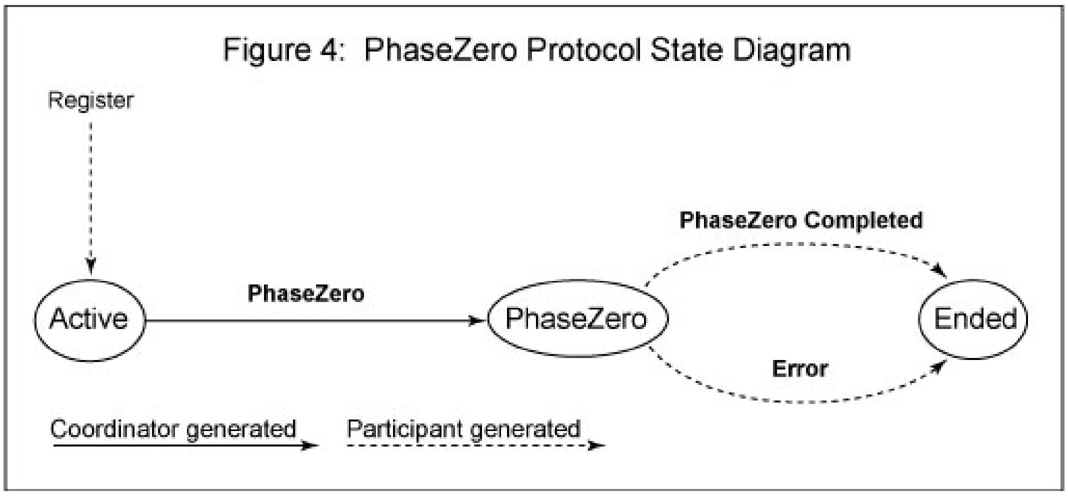
A party should be prepared to receive duplicate notifications and respond back to the source in a manner consistent with the current state of the target.

If a party receives a notification protocol message for an Unknown transaction, it should transmit an **Unknown** notification back to the source.

4.4. AT3.4 PhaseZero Protocol

The PhaseZero protocol is used by applications that need a notification to prepare for the completion of a transaction prior to the 2PC protocol. A typical example usage is an application that caches updated information in-memory and needs a **PhaseZero** notification, so that it knows when to flush cached updates to a database before the 2PC protocol is started. The PhaseZero protocol is needed since Resource managers typically cannot perform additional work for a transaction after 2PC has begun.

Figure AT4: PhaseZero Protocol State Diagram



The state diagram in Figure AT4 specifies the behavior of the two-way protocol as the exchange of messages between a coordinator and one of its participants. The state reflects what both sides know of their relationship. Omitted are details such as resending of messages or the exchange of error messages due to protocol error.

The coordinator sends the **PhaseZero** notification.

The participant sends the **PhaseZeroCompleted** notification or **Error** message.

The coordinator begins the protocol by sending the **PhaseZero** notification, so both sides know the protocol is in the PhaseZero state. After attempting a **PhaseZero**, the participant can respond with either the **PhaseZeroCompleted** or an **Error** with a wstx:PhaseZeroFailure Errorcode, and both sides know the protocol between them has Ended. The coordinator will only initiate the 2PC protocol after all PhaseZero participants have returned the **PhaseZeroCompleted** notification.

The participant's PhaseZero service is defined as:

```
<wsdl:portType name="PhaseZeroParticipantPortType">
    <wsdl:operation name="PhaseZero">
        <wsdl:input message="wstx:PhaseZero" />
    </wsdl:operation>
    <wsdl:operation name="Error">
        <wsdl:input message="wstx:Error" />
    </wsdl:operation>
</wsdl:portType>
```

The coordinator's PhaseZero service is defined as:

```
<wsdl:portType name="PhaseZeroCoordinatorPortType">
    <wsdl:operation name="PhaseZeroCompleted">
        <wsdl:input message="wstx:PhaseZeroCompleted" />
    </wsdl:operation>
    <wsdl:operation name="Unknown">
        <wsdl:input message="wstx:Unknown" />
    </wsdl:operation>
    <wsdl:operation name="Error">
        <wsdl:input message="wstx:Error" />
    </wsdl:operation>
</wsdl:portType>
```

A party should be prepared to receive duplicate notifications and respond back to the source in a manner consistent with the current state of the target.

If a party receives a notification protocol message for an Unknown transaction, it should transmit an **Unknown** notification back to the source.

4.5. AT3.5 2PC Protocol

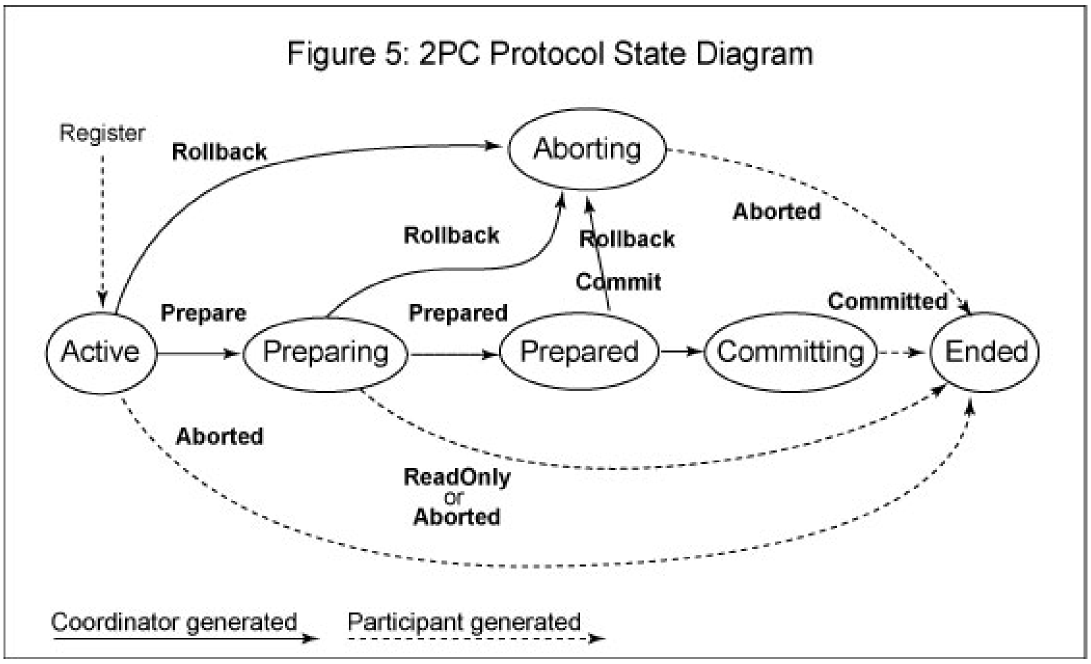
The 2PC (two-phase commit) protocol is a Coordination protocol that defines how multiple participants reach agreement on the outcome of an atomic transaction.

The state diagram in Figure AT5 specifies the behavior of the two-way protocol as the exchange of messages between a coordinator and one of its participants. The state reflects what both sides know of their relationship. Omitted are details such as resending of messages or the exchange of error messages due to protocol error.

The coordinator sends the **Prepare**, **Rollback** and **Commit** messages.

The participant returns the **Prepared**, **ReadOnly**, **Aborted** and **Committed** messages.

Figure AT5: 2PC Protocol State Diagram



The 2PC protocol makes a "presumed abort" assumption to minimize work for normal commit case. Presumed abort means that no knowledge of a transaction implies it is aborted, which allows the following optimizations:

- A coordinator can delay logging anything about the transaction until the commit decision.
- A participant can terminate the protocol and forget all knowledge of it after sending an **Aborted** or **ReadOnly** status to the coordinator during PhaseOne. No acknowledgement message from the coordinator is needed.
- After a **Prepared** status is received during PhaseOne, an abort outcome allows a coordinator to forget the transaction after sending the **Rollback** message to its participants. No acknowledgement message from the participant is needed.
- Only after a **Prepared** status is received, a commit outcome requires a coordinator to remember the transaction until all **Committed** acknowledgement messages have been received from its participants.

The coordinator initiates the protocol and requests participants to vote by issuing a Prepare message. While processing the **Prepare** message, both sides are in the Preparing state. The participant can do either of the following:

- It can reply with a **ReadOnly** status, which indicates that it votes to commit and does not need to participate further in the 2PC protocol. In this case, the two-party protocol is in the Ended state.
- It can reply with an **Aborted** status, which indicates that it votes to not commit and does not need to participate further in the 2PC protocol. In this case, the two-party protocol is in the Ended state.
- It can reply with a **Prepared** status, which indicates that it votes to commit. In this case, the two-party protocol is in the Prepared state. A **Prepared** status also indicates that the participant has reliably stored information needed to either commit or abort even if it subsequently fails.

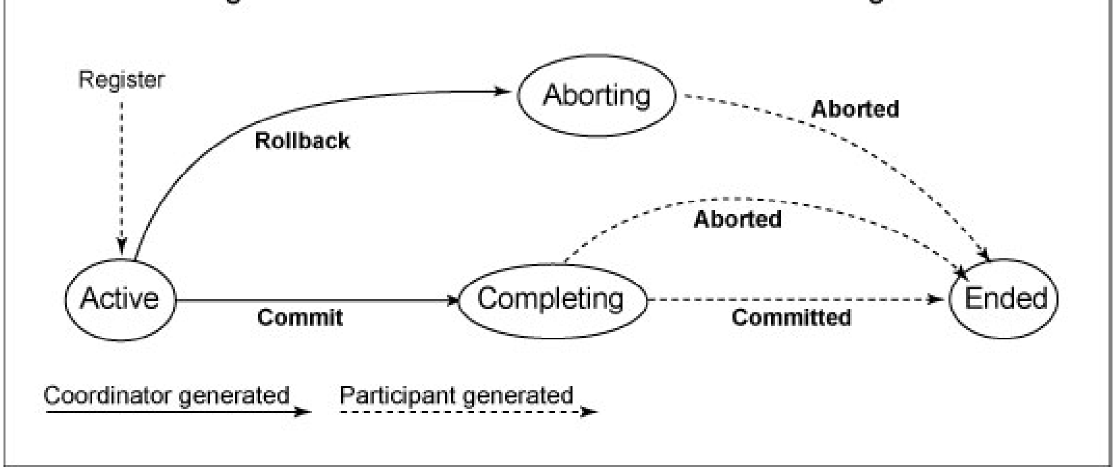
If during the Preparing state the coordinator sends the **Rollback** message, the participant enters the Aborting state.

For the overall transaction, once all **Prepared** reply messages have returned, the coordinator decides whether the outcome for the overall transaction is to commit or abort. It permanently records the decision on stable storage and sends the **Commit** or **Rollback** to all participants, leaving each of the two-party protocols in the same Committing or Aborting state. When each participant has finished committing or aborting, it replies with a **Committed** or **Aborted** acknowledgment.

The state diagram in Figure AT6 specifies the behavior of the protocol between a coordinator and one of its participants. The state reflects what both sides know of their relationship. Omitted are details such as resending of messages or the exchange of error messages due to protocol error.

Figure AT6: OnePhaseCommit Protocol State Diagram

Figure 6: One Phase Commit Protocol State Diagram



The coordinator sends the **Commit** and **Rollback** messages.

The participant sends the **Committed** and **Aborted** outcome status messages.

The coordinator begins by sending a **Commit** or **Rollback** request, putting the two-party protocol in the Completing or Aborting state. The only outcome of the Aborting state is for the participant to send an **Aborted** status message. While in the Completing state, the participant can decide on either commit or abort, returning a **Committed** or **Aborted** status message.

The above state diagram did not discuss failure semantics. The **Replay** message can be use by a participant to solicit the transaction outcome from the coordinator after failure, where the participant provides its protocol service PortReference. An acknowledgement is not needed, because the coordinator will begin sending protocol messages.

The participant's 2PC service is defined as:

```
       <wsdl:portType name="2PCParticipantPortType">
           <wsdl:operation name="Prepare">
               <wsdl:input message="wstx:Prepare" />
           </wsdl:operation>
           <wsdl:operation name="OnePhaseCommit">
               <wsdl:input message="wstx:OnePhaseCommit" />
           </wsdl:operation>
           <wsdl:operation name="Commit">
               <wsdl:input message="wstx:Commit" />
           </wsdl:operation>
           <wsdl:operation name="Rollback">
               <wsdl:input message="wstx:Rollback">
           </wsdl:operation>
           <wsdl:operation name="Unknown">
               <wsdl:input message="wstx:Unknown" />
           </wsdl:operation>
           <wsdl:operation name="Error">
               <wsdl:input message="wstx:Error" />
           </wsdl:operation>
       </wsdl:portType>
```

The coordinator's 2PC service is defined as:

```
       <wsdl:portType name="2PCCoordinatorPortType">
           <wsdl:operation name="Prepared">
               <wsdl:input message="wstx:Prepared" />
           </wsdl:operation>
           <wsdl:operation name="Aborted">
               <wsdl:input message="wstx:Aborted"/>
           </wsdl:operation>
           <wsdl:operation name="ReadOnly">
               <wsdl:input message="wstx:ReadOnly"/>
           </wsdl:operation>
           <wsdl:operation name="Committed">
               <wsdl:input message="wstx:Committed"/>
           </wsdl:operation>
           <wsdl:operation name="Replay">
               <wsdl:input message="wstx:Replay"/>
           </wsdl:operation>
           <wsdl:operation name="Unknown">
               <wsdl:input message="wstx:Unknown" />
           </wsdl:operation>
           <wsdl:operation name="Error">
               <wsdl:input message="wstx:Error"/>
           </wsdl:operation>
       </wsdl:portType>
```

A party should be prepared to receive duplicate notifications and respond back to the source in a manner consistent with the current state of the target.

Replay informs a party that the sender is recovering. In response the party should respond back to the source in a manner consistent with the current state of the target.

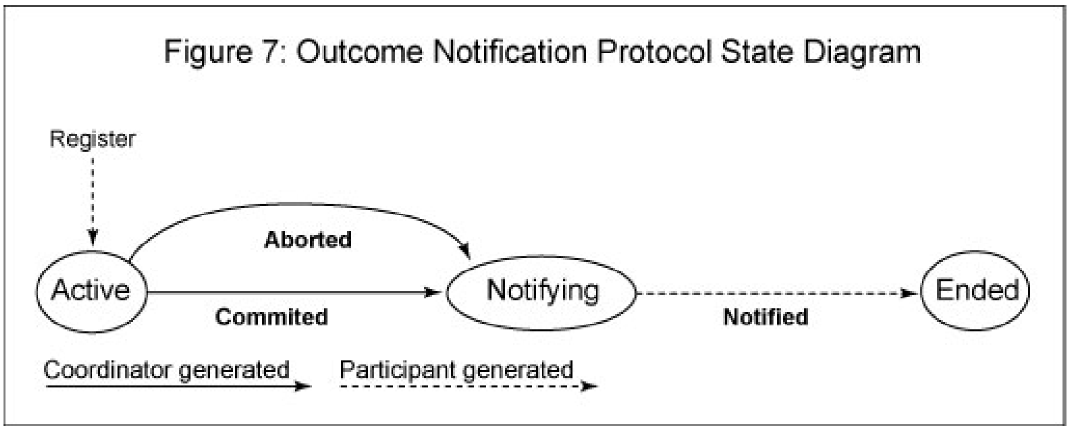
If a party receives a notification protocol message for an unknown transaction, it should transmit an Unknown notification back to the source.

4.6. AT3.6 OutcomeNotification Protocol

The outcome notification protocol is used by applications to find out when a transaction has completed and what the outcome is.

The state diagram in Figure AT7 specifies the behavior of the protocol between a coordinator and one of its participants. The state reflects what both sides know of their relationship. Omitted are details such as resending of messages or the exchange of error messages due to protocol error.

Figure AT7: OutcomeNotification Protocol State Diagram



The coordinator sends the outcome notification indicating either a **Committed** or **Aborted** status. The **Notified** message is a simple acknowledgement that the outcome notification was received. The coordinator must remember the outcome until this acknowledgement is received.

The participant's OutcomeNotification service is defined as:

```
       <wsdl:portType name="OutcomeNotificationParticipantPortType">
           <wsdl:operation name="Committed">
               <wsdl:input message="wstx:Committed"/>
           </wsdl:operation>
           <wsdl:operation name="Aborted">
               <wsdl:input message="wstx:Aborted"/>
           </wsdl:operation>
           <wsdl:operation name="Unknown">
               <wsdl:input message="wstx:Unknown" />
           </wsdl:operation>
           <wsdl:operation name="Error">
               <wsdl:input message="wstx:Error" />
           </wsdl:operation>
       </wsdl:portType>
```

The coordinator's OutcomeNotification service is defined as:

```
       <wsdl:portType name="OutcomeNotificationCoordinatorPortType">
           <wsdl:operation name="Notified">
               <wsdl:input message="wstx:Notified"/>
           </wsdl:operation>
           <wsdl:operation name="Replay">
               <wsdl:input message="wstx:Replay"/>
           </wsdl:operation>
           <wsdl:operation name="Unknown">
               <wsdl:input message="wstx:Unknown" />
           </wsdl:operation>
           <wsdl:operation name="Error">
               <wsdl:input message="wstx:Error"/>
           </wsdl:operation>
       </wsdl:portType>
```

A party should be prepared to receive duplicate notifications and respond back to the source in a manner consistent with the current state of the target.

Replay informs a party that the sender is recovering. In response the party should respond back to the source in a manner consistent with the current state of the target.

If a party receives a notification protocol message for an unknown transaction, it should transmit an **Unknown** notification back to the source.

5. Security Considerations

Because messages can be modified or forged, it is strongly RECOMMENDED that business process implementations use WS-Security to ensure messages have not been modified or forged while in transit or while residing at destinations. Similarly, invalid or expired messages could be re-used or message headers not specifically associated with the specific message could be referenced. Consequently, when using WS-Security, signatures MUST include the semantically significant headers and the message body (as well as any other relevant data) so that they cannot be independently separated and re-used.

The protocols defined in this specification are subject to various forms of replay attacks. In addition to the mechanisms list above, messages SHOULD include a message timestamp (as described in WS-Security [WSSec]). Recipients can use this timestamp information to cache the most recent messages for a context and detect duplicate transmissions and prevent potential replay attacks.

It should also be noted that services implementing this protocol are subject to various forms of denial-of-service attacks.

Implementers should take this into account when building their services.

6. Relationships to Other Web Services

AT depends on WS-Coordination.

7. Interoperability Considerations

In order for two parties to communicate, both parties will need to agree on the protocols provided. This specification facilitates this agreement and thus interoperability.

8. AT Glossary

[Definition: **Abort** - Make the tentative actions taken during an atomic transaction appear to never have happened.]

[Definition: **Atomic Transaction** - A WS-Coordination activity type that provides an all-or-nothing property. Actions during the transaction are tentative (i.e., neither made persistent nor seen outside the transaction). If all parts of the intended goal are achieved, the actions are all committed. If something goes wrong, the actions are all aborted. Some applications increase concurrency using weaker isolation levels that allow some tentative actions to be seen by other transactions.]

[Definition: **Commit** - Make the tentative actions taken during an atomic transaction persistent and visible to other atomic transactions.]

[Definition: **Completion protocol** - A protocol that allows a transaction participant to either try to commit an atomic transaction or to force it to abort.]

[Definition: **OutcomeNotification protocol** - A Coordination protocol that allows a transaction participant to find out when a transaction has completed and what the outcome is.]

[Definition: **PhaseOne** - The first half of the 2PC Coordination protocol, where the coordinator tells each of its 2PC participants to prepare (see prepare) to either commit or abort.][Definition: **PhaseTwo** - The second half of the 2PC Coordination protocol, where the coordinator notifies its 2PC participants of the outcome to either commit or abort.]

[Definition: **PhaseZero** - A Coordination protocol for atomic transactions intended for caches. The coordinator notifies its PhaseZero participants that it should do what it needs to do prior to 2PC (e.g., sync its updates to a database). This protocol is preformed prior to beginning the 2PC protocol.]

[Definition: **Prepare** - Make it possible to either commit or abort after the decision is made, even if a failure occurs, and then vote on the transaction outcome.]

[Definition: **OnePhaseCommit** - An optimization of the 2PC Coordination protocol. When a coordinator has a single participant, it can delegate the commit-abort decision to that participant.]

[Definition: **2PC (two-phase commit) protocol** - A Coordination protocol for atomic transactions intended for resource managers (e.g., database). The coordinator first tells all 2PC participants to prepare for either a commit or abort decision, decides whether to commit or abort depending on the vote of all the 2PC participants, and then notifies the 2PC participants of the decision.]

9. AT References

KEYWORDS
- S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, RFC 2119, Harvard University, March 1997.

SOAP
- W3C Note, *SOAP: Simple Object Access Protocol 1.1*, 08 May 2000.

URI

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

XML-ns
- W3C Recommendation, *Namespaces in XML*, 14 January 1999.

XML-Schema1
- W3C Recommendation, *XML Schema Part 1: Structures*, 2 May 2001.

XML-Schema2
- W3C Recommendation, *XML Schema Part 2: Datatypes*, 2 May 2001.

WSCOOR
- Microsoft & IBM, *Web Services Coordination (WS-Coordination)*

WSDL
- W3C Note, *Web Services Description Language (WSDL) 1.1*

WSSec
- Microsoft, IBM, &VeriSign, *Web Services Security (WS-Security)*

10. Part II: Business Activity (BA) - BA1 Introduction

The Business Activity protocols handle long-lived activities and the desire to apply business logic to handle business exceptions. These coordination protocols are required when interoperating across vendor implementations and provide support for a variety of business process behaviors such as those found in the BPEL language specification [BPEL].

While atomic transactions are important building blocks for these activities, they are not sufficient for the overall coordination. Atomic transaction implementations typically assume short time duration and higher trust, so they hold data resources (e.g., locking) and physical resources (e.g., connections, threads, memory). Business activities require resources to be shared prior to completion. Even if the business activity is not expected to need a long time to execute, another trust domain (e.g., another company) may not allow resources to be held.

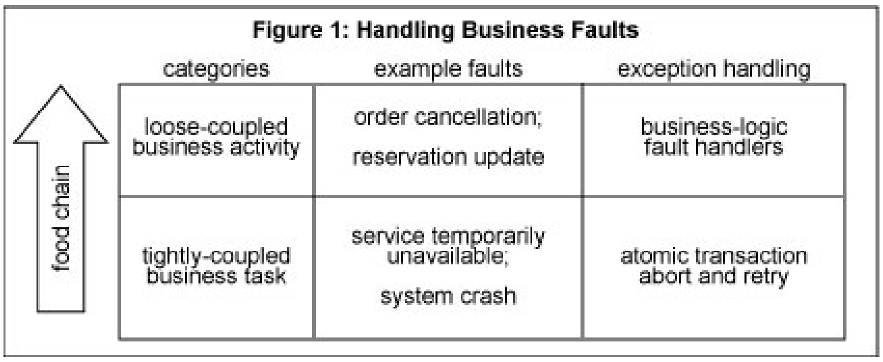


Figure 1: Handling Business Faults

categories	example faults	exception handling
loose-coupled business activity	order cancellation; reservation update	business-logic fault handlers
tightly-coupled business task	service temporarily unavailable; system crash	atomic transaction abort and retry

Some of the above limitations of atomic transactions could be overcome by different implementation designs by using flexible isolation policies or compensations instead of locking. However, there is an even more fundamental rationale for introducing business activity coordination. Figure BA1 illustrates the food chain of tasks, from individual task to overall activity. When low-level tasks go wrong, atomic transaction abort and retry is a good solution, because business logic is not needed for retry. As a result, a business activity is typically designed as an activity that consists of a sequence of tasks, where each task satisfies the constraints of an atomic transaction. However, abort is typically not the desired course of fault handling for business level exceptions. Because locks are not held between tasks that comprise an activity, each task becomes part of the business history. Exception handling mechanisms such as compensation can be used to reverse the effects of a completed business tasks, and the business activity contains logic that keeps the overall activity moving forward in time.

It is relatively inexpensive to abort and retry an atomic transaction. Implementations have a design point that exploits this characteristic that includes a "presumed abort" assumption designed to reduce costs for transactions that complete without problems.

The characteristics of business activities and atomic transactions differ in the following ways:

- A business activity may consume lots of resources. There may be a significant number of atomic transactions

involved.
- More importantly, loss of state of a business activity has huge repercussions, because important historical information is lost. Loss of customer history can cost not only a sale, but a customer relationship. Loss of financial history can cause legal and accounting problems.
- Because individual tasks within a business activity can be seen prior to the completion of the business activity, their results may have an impact outside of the computer system that is potentially quite costly. It is worth spending computer resources to detect problems as early as possible to avoid unnecessary tasks.
- Responding to a request may take a very long time. Human approval, assembly, manufacturing or delivery may have to take place before a response can be sent.

These characteristics lead to a different design point, with the following assumptions:

- All state transitions are reliably recorded (e.g., on stable storage or redundant systems), including application state and coordination metadata.
- All request messages are acknowledged, so that problems are detected as early as possible. This avoids executing unnecessary tasks. It can also detect a problem earlier when fixing it is easier and less expensive to do.
- A response is defined as a separate operation instead of the output of the request. Message input-output implementations will typically have timeouts that are too short for some business activity responses. If the response is not received after a timeout, it is resent. This is repeated until a response is received. The request receiver discards all but one identical request received.

These are the fundamental principles assumed throughout this specification.

Business Activity defines protocols for business activities for Web services applications that enable existing business processing and work flow systems to wrap their proprietary mechanisms and interoperate across vendor implementations and business boundaries.

This specification leverages WS-Coordination by extending it to support business activities. It does this by adding constraints to the protocols defined in WS-Coordination and by defining its own Coordination protocols.

A business activity uses the WS-Coordination protocols to do the following:

- Create a business activity, which is defined as a WS-Coordination coordination type. This returns a BusinessActivity CoordinationContext.
- Propagate the CoordinationContext in messages between Web services, so that receiving Web services can participate in the activity.
- Interpose a coordinator as a subordinate to the current coordinator.
- Register for participation in a Coordination protocol, depending on the participant role. The set of available coordination protocols depends on the WS-Coordination coordination type. The coordination protocols supported by the BusinessActivity coordination type includes the BusinessAgreement and BusinessAgreementWithComplete protocols.
- Optionally, create a business scope, which is part of an overall business activity. This create updates the existing BusinessActivity CoordinationContext to indicate the relationship of the business activity and business task.

The CoordinationContext and messages defined in this specification include an extensibility element that allows implementation-specific and application-specific information to be included.

The constraints that Business Activity puts on WS-Coordination protocols are described in Section 2. The Business Activity Coordination protocols are defined in Section 3.

Terms introduced in Part II are explained in the body of the specification and summarized in the .

10.1. BA1.1 Model

Business Activity Coordination protocols provide the following flexibility:

- It allows a business activity to be partitioned into scopes. A scope is a business task consisting of a general-purpose computation carried out as a bounded set of operations on a collection of Web Services that require a mutually agreed outcome. There can be any number of hierarchical nesting levels. The relationship between a parent and child scope needs to be understood. Nested scopes:

  - Allow a parent activity to select which child tasks are included in the overall outcome processing. For example, a business activity might solicit an estimate from a number of suppliers and choose a quote or bid based on lowest-cost.
  - Allow a parent to catch an exception thrown by its child, apply an exception handler and continue processing even if something goes wrong. When a child completes its work, it is associated with a compensation that the parent may call (e.g., from its exception handlers).

- It allows a task within a business activity to specify that it is leaving a business activity. Providing the ability to exit a business activity allows business programs to delegate processing to other scopes. In contrast to atomic transactions, the participant list is dynamic and a participant may exit the protocol at any time before voting on the outcome.
- It allows a task within a business activity to specify its outcome directly without waiting for solicitation. Such a feature is generally useful when a task fails so that the notification can be used by business activity exception handler to modify the goals and drive processing in a timely manner.

10.2. BA1.2 Namespace

The XML namespace [XML-ns] URI that MUST be used by implementations of this specification is:

```
http://schemas.xmlsoap.org/ws/2002/08/wsba
```

The namespace prefix "wsba" used in this specification is associated with this URI. This is also used as the business coordination type identifier.

The following namespaces are used in this document:

Prefix	Namespace
S	http://www.w3.org/2001/12/soap-envelope
wsu	http://schemas.xmlsoap.org/ws/2002/07/utility
wscoor	http://schemas.xmlsoap.org/ws/2002/08/wscoor
wsba	http://schemas.xmlsoap.org/ws/2002/08/wsba

If an action URI is used then the action URI MUST consist of the wsba namespace URI concatenated with the '#' character and the operation name. For example:

```
  http://schemas.xmlsoap.org/ws/2002/08/wsba#Complete
```

10.3. BA1.3 XSD and WSDL Files

The following links hold the XML schema and the WSDL declarations defined in this document.

http://schemas.xmlsoap.org/ws/2002/08/wsba/wsba.xsd

http://schemas.xmlsoap.org/ws/2002/08/wsba/wsba.wsdl

11. BA2 Using WS-Coordination

This section describes the Business Activity usage of WS-Coordination protocols.

11.1. BA2.1 CoordinationContext

A business activity uses the WS-Coordination CoordinationContext with the CoordinationType set to the following URI:

```
http://schemas.xmlsoap.org/ws/2002/08/wsba
```

A CoordinationContext can have additional elements for extensibility.

The following is an example business activity CoordinationContext:

```
<?xml version="1.0" encoding="utf-8"?>
<S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope"
    <S:Header>
        . . .
        <wscoor:CoordinationContext
            xmlns:wscoor="http://schemas.xmlsoap.org/ws/2002/08/wscoor"
            xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility"
            xmlns:myApp="http://Adventure456.com/myApp">
            <wsu:Identifier>
                http://Fabrikam123.com/SS/1234
            </wsu:Identifier>
            <wsu:Expires>2002-08-31T13:20:00-05:00</wsu:Expires>
            <wscoor:CoordinationType>
                http://schemas.xmlsoap.org/ws/2002/08/wsba
            </wscoor:CoordinationType>
            <wscoor:RegistrationService>
                <wsu:Address>
                    http://Schedule456.com/mycoordinationservice/registration
                </wsu:Address>
                <myApp:Myapp:BetaMark> ... </myApp:Myapp:BetaMark>
                <myApp:EBDCode> ... </myApp:EBDCode>
            </wscoor:RegistrationService>
        </wscoor:CoordinationContext>
        . . .
    </S:Header>
    . . .
</S:Envelope>
```

11.2. BA2.2 CreateCoordinationContext Operation

The CreateCoordinationContext operation semantics depend on the arguments passed in the CreateCoordinationContext message. The cases are:

- When a CurrentContext is not included, a new business activity is created. The returned CoordinationContext represents the new business activity.
- When a CurrentContext is included, the target coordinator is interposed as the subordinate to the current coordinator. The returned CoordinationContext represents the same business activity but has the PortReference of the interposed coordinator's RegistrationService.

As described in the WS-Coordination specification, implementations MAY extend the CreateCoordinationContext operation by including additional elements in the CreateCoordinationContext message.

For example, some implementations MAY provide support for capturing additional information related to parent-child relationships between scopes in their respective CoordinationContexts. An implementation MAY place, for example, a `<myService:NestedCreate wsu:MustUnderstand="true">` element within the CreateCoordinationContext message in order to cause the respective service to add information specific to the nesting relationship to newly created CoordinationContext returned in the response.

To ensure that all implementations properly recognize such requests the wsu:MustUnderstand="true" attributed MUST be added to the extension element. If and implementation does not support `myService:NestedCreate` the implementation MUST return a standard `SOAP Misunderstood` fault referring to the NestedCreate element.

The overall coordination framework described in WS-Coordination and Business Activity also enables extensibility by the creation of a new CoordinationType; for example, it is also possible to define a nested business activity that would have additional semantics on the create.

12. BA3 Coordination Protocols

The Coordination protocols for business activities are summarized below with names relative to the wsba base name:

- **BusinessAgreement:** A nested scope participant registers for this protocol with its parent scope coordinator, so that its parent scope can manage it. A nested scope must know when it has completed all work for a business activity.
- **BusinessAgreementWithComplete:** A nested scope participant registers for this protocol with its parent scope coordinator, so that its parent scope can manage it. A nested scope relies on its parent to tell it when it has received all requests to perform work within the business activity.

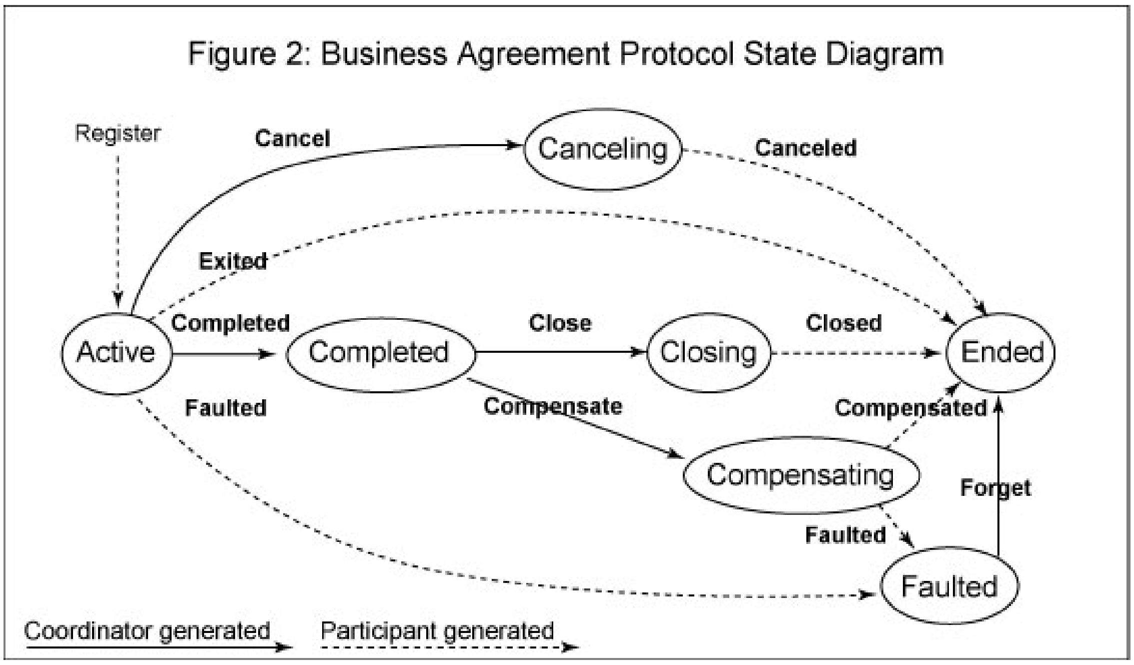
12.1. BA3.1 BusinessAgreement Protocol

The state diagram in Figure BA2 specifies the behavior of the protocol between a coordinator and a participant. The state reflects what both sides know of their relationship. Omitted are details such as resending of messages or the exchange of error messages due to protocol error.

A coordinator sends the **Close**, **Cancel**, **Compensate** and **Forget** messages.

A participant sends the **Completed**, **Faulted**, **Compensated**, **Closed**, **Canceled** and **Exited** messages.

Figure BA2: BusinessAgreement Protocol State Diagram



When a parent sends an application message containing a business CoordinationContext to a child, the child Registers with the parent as a participant for the BusinessAgreement protocol. After registering, the two-party protocol is in the Active state and can have one of the following happen:

- If the child finishes and the nature of the task requires no more participation in the business activity (e.g., read-only, irreversible, suppressed internal fault), the child sends an **Exited** message to its parent, putting the two-party protocol in the Ended state in the state diagram of Figure BA2.
- If the child finishes and wants to continue participation in the business activity, the child notifies the parent. This requires the ability to compensate for the tasks it has taken so far, and then sending an unsolicited **Completed** message to its parent, putting the two-party protocol in the Completed state in the state diagram of Figure BA2.

After the parent receives the **Completed** message from a child, the child scope does not end the two-party protocol. Instead, it lives on until the parent sends a **Close** or **Compensate** message to the child. The compensation is not the same as an atomic transaction abort, which makes it look like the original task never happened, because the original task actually did happen and is part of recorded history. Instead, a compensation is another forward task that makes an adjustment to reverse the original task in some practical way. For example, the compensation for making a reservation might be to drop the reservation, although a fee might be involved. An installed compensation is executed if the parent sends a **Compensate** message to the child. When a compensation is executed, both the original task and its compensation are part of recorded history.

- If the child fails while active or compensating, it sends a **Faulted** message to its parent, putting the two-party protocol in the Faulted state, where the child abandons its work in some appropriate way. The parent replies with a **Forget** message, putting the two-party protocol in the Ended state.
- If the parent tells the child to **Cancel**, it puts the protocol in the Canceling state, where the child abandons its work in some appropriate way. The child responds with a **Canceled** message, putting the protocol in the Ended state.

Since a transition from the Active state can be initiated by either the parent or the child, there are race conditions possible:

- It is possible for the child's **Completed** message and the parent's **Cancel** message to pass one another, creating a race condition. In this case, the **Completed** message wins, so the two-party protocol goes to the Completed state. Both parent and child ignore the **Cancel** message. The parent can tell when this happens, because a **Completed** message is not a correct response to its **Cancel** message. The child can tell when this happens, because a **Cancel** message is not a correct response to a **Completed** message.
- It is possible for the child's **Faulted** message and the parent's **Cancel** message to pass one another, creating a race condition. In this case, the **Faulted** message wins, so the two-party protocol goes to the Faulted state. Both parent and child ignore the **Cancel** message. The parent can tell when this happens, because a **Faulted** message is not a correct response to its **Cancel** message. The child can tell when this happens, because a **Cancel** message is not a correct response to a **Faulted** message.
- It is possible for the child's **Exited** message and the parent's **Cancel** message to pass one another, creating a race condition. In this case, the **Exited** message wins, so the two-party protocol goes to the Ended state. Both parent and child ignore the **Cancel** message. The parent can tell when this happens, because an **Exited** message is not a correct response to its **Cancel** message. The child can tell when this happens, because no response is expected for an **Exited** message.

A party should be prepared to receive duplicate notifications and respond back to the source in a manner consistent with the current state of the target.

Replay informs a party that the sender is recovering. In response the party should respond back to the source in a manner consistent with the current state of the target.

If a party receives a notification protocol message for an unknown business activity, it should transmit an **Unknown** notification back to the source.

12.2. BA3.2 BusinessAgreementWithComplete Protocol

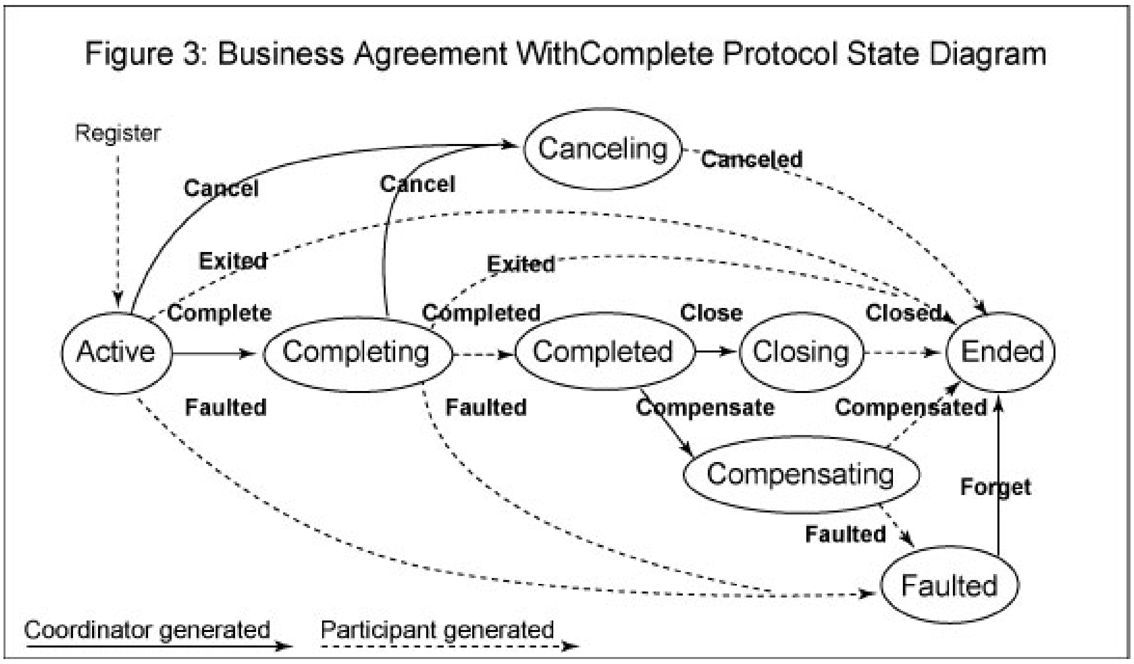
The state diagram in Figure BA3 specifies the behavior of the protocol between a coordinator and a participant. The state reflects what both sides know of their relationship. Omitted are details such as resending of messages or the exchange of error messages due to protocol error.

A coordinator sends the **Complete**, **Close**, **Cancel**, **Compensate** and **Forget** messages.

A participant sends the **Completed**, **Faulted**, **Compensated**, **Closed**, **Canceled** and **Exited** messages.

The BusinessAgreementWithComplete protocol is the same as the BusinessAgreement protocol, except that a nested scope relies on its parent to tell it when it has received all requests to do work within the business activity by receiving the **Complete** message. When it receives the **Complete** message, the two-party protocol goes to the Completing state. While in the Completing state, the Completed message takes the protocol to the Completed state. While in the Completing state, the **Cancel**, **Exited** and **Faulted** messages take the protocol to the same states as they do while in the Active state.

Figure BA3: BusinessAgreementWithComplete Protocol State Diagram



13. Security Considerations

Because messages can be modified or forged, it is strongly RECOMMENDED that business process implementations use WS-Security to ensure messages have not been modified or forged while in transit or while residing at destinations. Similarly, invalid or expired messages could be re-used or message headers not specifically associated with the specific message could be referenced. Consequently, when using WS-Security, signatures MUST include the semantically significant headers and the message body (as well as any other relevant data) so that they cannot be independently separated and re-used.

The protocols defined in this specification are subject to various forms of replay attacks. In addition to the mechanisms list above, messages SHOULD include a message timestamp (as described in WS-Security [WSSec]). Recipients can use this timestamp information to cache the most recent messages for a context and detect duplicate transmissions and prevent potential replay attacks.

It should also be noted that services implementing this protocol are subject to various forms of denial-of-service attacks. Implementers should take this into account when building their services.

14. Relationship to Other Web Services

BA depends on WS-Coordination.

15. Interoperability Considerations

In order for two parties to communicate, both parties will need to agree on the protocols provided. This specification facilitates this agreement and thus interoperability.

16. BA Glossary

[Definition: **Cancel** - Back out of a business activity.]

[Definition: **Close** - Terminate a business activity with a favorable outcome.]

[Definition: **Compensate** - A message to a Completed scope from a coordinator to execute its compensation. This message is part of both the BusinessAgreement and BusinessAgreementWithComplete protocols.]

[Definition: **Complete** - A message to a scope from a coordinator telling it that it has been given all of the work for that business activity. This message is part of the BusinessAgreementWithComplete protocol.]

[Definition: **Completed** - A message from a scope telling a coordinator that the scope has successfully executed everything asked of it and needs to continue participating in the protocol. This message is part of both the BusinessAgreement and BusinessAgreementWithComplete protocols.]

[Definition: **Exited** - A message from a scope telling a coordinator that the scope it does not need to continue participating in the protocol. This message is part of both the BusinessAgreement and BusinessAgreementWithComplete protocols.]

[Definition: **Faulted** - A message from a scope telling a coordinator that the scope could not execute successfully.]

[Definition: **BusinessAgreement protocol** - A business activity coordination protocol that supports long-lived business processes and allows business logic to handle business logic exceptions. A participant in this protocol must know when it has completed with its tasks in a business activity.]

[Definition: **BusinessAgreementWithComplete protocol** - A business activity coordination protocol that supports long-lived business processes and allows business logic to handle business logic exceptions. A participant in this protocol relies on its parent to tell it when it has received all requests to do work within a business activity.]

[Definition: **Scope** - A Web service that acts as a unit for exception handling and recovery as part of the BusinessAgreement protocol. A scope integrates coordinator and application logic. A Web services application can be partitioned into a hierarchy of scopes, where the parent scope coordinates its child scopes.]

17. BA References

BPEL
Microsoft & IBM, *Web Services Business Process Execution Language*

KEYWORDS
- S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, RFC 2119, Harvard University, March 1997.

SOAP
- W3C Note, *SOAP: Simple Object Access Protocol 1.1*, 08 May 2000.

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

XML-ns
- W3C Recommendation, *Namespaces in XML*, 14 January 1999.

XML-Schema1
- W3C Recommendation, *XML Schema Part 1: Structures*, 2 May 2001.

XML-Schema2
- W3C Recommendation, *XML Schema Part 2: Datatypes*, 2 May 2001.

WSCOOR
- Microsoft & IBM, *Web Services Coordination (WS-Coordination)*

WSDL
- W3C Note,*Web Services Description Language (WSDL) 1.1*