Web Services Atomic Transaction (WS-AtomicTransaction)

November 2004

Authors

Luis Felipe Cabrera, Microsoft
George Copeland, Microsoft
Max Feingold, Microsoft
Tom Freund, IBM
Jim Johnson, Microsoft
Chris Kaler, Microsoft
Johannes Klein, Microsoft
David Langworthy, Microsoft (Editor)
Anthony Nadalin, IBM
David Orchard, BEA Systems
Ian Robinson, IBM
Tony Storey, IBM
Satish Thatte, Microsoft

Copyright Notice

Abstract

This specification provides the definition of the atomic transaction coordination type that is to be used with the extensible coordination framework described in the WS-Coordination specification. The specification defines three specific agreement coordination protocols for the atomic transaction coordination type: completion, volatile two-phase commit, and durable two-phase commit. Developers can use any or all of these protocols when building applications that require consistent agreement on the outcome of short-lived distributed activities that have the all-or-nothing property.

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-AtomicTransaction by itself does not define all features required for a complete solution. WS-AtomicTransaction 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 accommodate a wide variety of coordination protocols related to the coordination actions of distributed applications.

Status

WS-AtomicTransaction 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 warrantees or representations regarding the specification in any manner whatsoever.

Acknowledgments

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

Francisco Curbera, IBM
Sanjay Dalal, BEA Systems
Doug Davis, IBM
Gert Drapers, Microsoft
Don Ferguson, IBM
Kirill Garvylyuk, Microsoft
Frank Leymann, IBM
Thomas Mikalsen, IBM
Jagan Peri, Microsoft
John Shewchuk, Microsoft
Alex Somogyi, BEA Systems
Stefan Tai, IBM
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 number of participants forming large distributed applications. The resulting activities may have complex structure and relationships.

The WS-Coordination specification defines an extensible framework for defining coordination types. This specification provides the definition of an atomic transaction coordination type used to coordinate activities having an "all or nothing" property. Atomic transactions commonly require a high level of trust between participants and are short in duration. 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.

To understand the protocol described in this specification, 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.

- The reader is familiar with the WS-Coordination [WSCOOR] specification that defines the framework for the WS-AtomicTransaction coordination protocols.
- The reader is familiar with WS-Addressing [WSADDR] and WS-Policy [WSPOLICY].

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 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.

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].

1.2 Namespace

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

```
http://schemas.xmlsoap.org/ws/2004/10/wsat
```

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/2003/05/soap-envelope
wscoor	http://schemas.xmlsoap.org/ws/2004/10/wscoor
wsat	http://schemas.xmlsoap.org/ws/2004/10/wsat

If an action URI is used then the action URI MUST consist of the wsat namespace URI concatenated with the "/" character and the element name. For example:

```
http://schemas.xmlsoap.org/ws/2004/10/wsat/Commit
```

1.3 XSD and WSDL Files

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

http://schemas.xmlsoap.org/ws/2004/10/wsat/wsat.xsd

http://schemas.xmlsoap.org/ws/2004/10/wsat/wsat.wsdl

Soap bindings for the WSDL documents defined in this specification MUST use "document" for the *style* attribute.

2. Atomic Transaction Context

Atomic Transaction builds on WS-Coordination, which defines an activation and a registration service. Example message flows and a complete description of creating and registering for coordinated activities is found in the WS-Coordination specification [WSCOOR].

The Atomic Transaction coordination context must flow on all application messages involved with the transaction.

Atomic Transaction adds the following semantics to the CreateCoordinationContext operation on the activation service.

- If the request includes the CurrentContext element, the target coordinator is interposed as a subordinate to the coordinator stipulated inside the CurrentContext element.
- If the request does not include a CurrentContext element, the target coordinator creates a new transaction and acts as the root.

A coordination context may have an Expires attribute. This attribute specifies the earliest point in time at which a transaction may be terminated solely due to its length of operation. From that point forward, the transaction manager may elect to unilaterally roll back the transaction, so long as it has not transmitted a Commit or a Prepared notification.

The Atomic Transaction protocol is identified by the following coordination type:

```
http://schemas.xmlsoap.org/ws/2004/10/wsat
```

3. Atomic Transaction Protocols

This specification defines the following protocols for atomic transactions.

- **Completion:** The completion protocol initiates commitment processing. Based on each protocol's registered participants, the coordinator begins with Volatile 2PC then proceeds through Durable 2PC. The final result is signaled to the initiator.
- **Two-Phase Commit (2PC)**: The 2PC protocol coordinates registered participants to reach a commit or abort decision, and ensures that all participants are informed of the final result. The 2PC protocol has two variants:
  - **Volatile 2PC:** Participants managing volatile resources such as a cache should register for this protocol.
  - **Durable 2PC:** Participants managing durable resources such as a database should register for this protocol.

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

3.1 Preconditions

The correct operation of the protocols requires that a number of preconditions MUST be established prior to the processing:

1. The source MUST have knowledge of the destination's policies, if any, and the source MUST be capable of formulating messages that adhere to this policy.
2. If a secure exchange of messages is required, then the source and destination MUST have a security context.

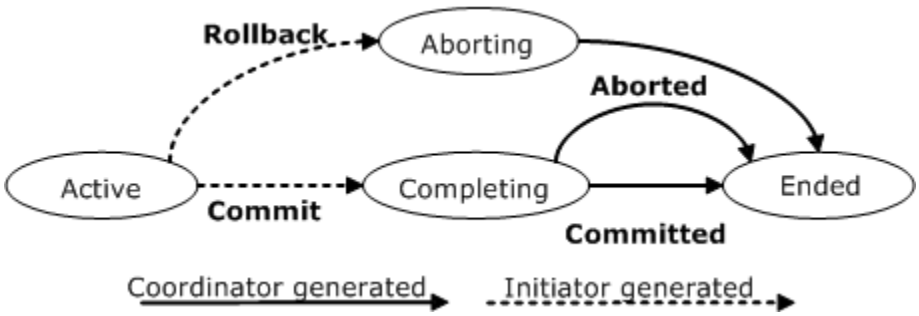
3.2 Completion Protocol

The Completion 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.

An initiator registers for this protocol using the following protocol identifier:

```
http://schemas.xmlsoap.org/ws/2004/10/wsat/Completion
```

The diagram below illustrates the protocol abstractly:



The coordinator accepts:

Commit
: Upon receipt of this notification, the coordinator knows that the participant has completed application processing and that it should attempt to commit the transaction.

Rollback
: Upon receipt of this notification, the coordinator knows that the participant has terminated application processing and that it should abort the transaction.

The initiator accepts:

Committed
: Upon receipt of this notification, the initiator knows that the coordinator reached a decision to commit.

Aborted
: Upon receipt of this notification, the initiator knows that the coordinator reached a decision to abort.

Conforming implementations must implement Completion.

3.3 Two-Phase Commit Protocol

The Two-Phase Commit (2PC) protocol is a Coordination protocol that defines how multiple participants reach agreement on the outcome of an atomic transaction. The 2PC protocol has two variants: Durable 2PC and Volatile 2PC.

3.3.1 Volatile Two-Phase Commit Protocol

Upon receiving a Commit notification in the completion protocol, the root coordinator begins the prepare phase of all participants registered for the Volatile 2PC protocol. All participants registered for this protocol must respond before a Prepare is issued to a participant registered for Durable 2PC. Further participants may register with the coordinator until the coordinator issues a Prepare to any durable participant. A volatile recipient is not guaranteed to receive a notification of the transaction's outcome.

Participants register for this protocol using the following protocol identifier:

```
http://schemas.xmlsoap.org/ws/2004/10/wsat/Volatile2PC
```

3.3.2 Durable Two-Phase Commit Protocol

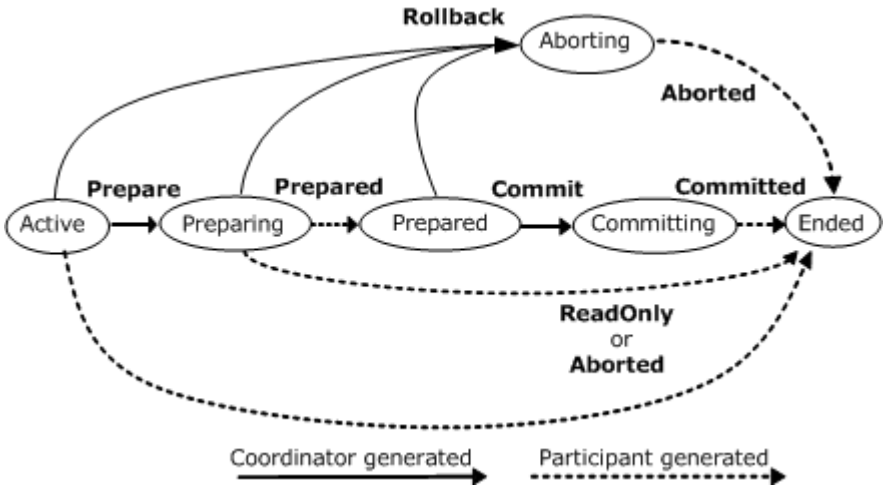
After receiving a Commit notification in the completion protocol and upon successfully completing the prepare phase for Volatile 2PC participants, the root coordinator begins the Prepare phase for Durable 2PC participants. All participants registered for this protocol must respond Prepared or ReadOnly before a Commit notification is issued to a participant registered for either protocol.

Participants register for this protocol using the following protocol identifier:

```
http://schemas.xmlsoap.org/ws/2004/10/wsat/Durable2PC
```

3.3.3 2PC Diagram and Notifications

The diagram below illustrates the protocol abstractly:



The participant accepts:

Prepare

Upon receipt of this notification, the participant knows to enter phase 1 and vote on the outcome of the transaction. If the participant does not know of the

transaction, it must vote to abort. If the participant has already voted, it should resend the same vote.

Rollback
: Upon receipt of this notification, the participant knows to abort, and forget, the transaction. This notification can be sent in either phase 1 or phase 2. Once sent, the coordinator may forget all knowledge of this transaction.

Commit
: Upon receipt of this notification, the participant knows to commit the transaction. This notification can only be sent after phase 1 and if the participant voted to commit. If the participant does not know of the transaction, it must send a Committed notification to the coordinator.

The coordinator accepts:

Prepared
: Upon receipt of this notification, the coordinator knows the participant is prepared and votes to commit the transaction.

ReadOnly
: Upon receipt of this notification, the coordinator knows the participant votes to commit the transaction, and has forgotten the transaction. The participant does not wish to participate in phase 2.

Aborted
: Upon receipt of this notification, the coordinator knows the participant has aborted, and forgotten, the transaction.

Committed
: Upon receipt of this notification, the coordinator knows the participant has committed the transaction. That participant may be safely forgotten.

Replay
: Upon receipt of this notification, the coordinator may assume the participant has suffered a recoverable failure. It should resend the last appropriate protocol notification.

Conforming implementations MUST implement the 2PC protocol.

4. Policy Assertions

WS-Policy [WSPOLICY], WS-PolicyAttachment [WSPOLICYATTACH], and WS-PolicyAssertions [WSPOLICYASSERT] collectively define a framework, model, and grammar for expressing the capabilities, requirements, and general characteristics of entities in an XML Web services-based system. This specification leverages the WS-Policy family of specifications to enable participants and coordinators to describe and advertise their capabilities and/or requirements. The set of policy assertions for atomic transaction is defined below.

4.1. Spec Version

The protocol determines invariants maintained by the reliable messaging endpoints and the directives used to track and manage the delivery of messages. The assertion that will be used to identify the protocol (and version) either used or supported (depending on context) is the wsp:SpecVersion assertion that is defined in the WS-PolicyAssertions specification [WSPOLICYASSERT].

An example use of this assertion to indicate an endpoint's support for the atomic transaction protocol follows:

```
<wsp:SpecVersion
wsp:URI="http://schemas.xmlsoap.org/ws/2004/10/wsat"
wsp:Usage="wsp:Required"/>
```

4.2 Protocols

This section establishes well-known names for the protocols supported by atomic transactions.

The following pseudo schema defines these elements:

```
<wsat:Completion ... />
<wsat:DurableTwoPhaseCommit ... />
<wsat:VolatileTwoPhaseCommit ... />
```

The following describes the attributes and tags listed in the syntax above:

/wsat:Completion
: This element is a policy assertion as defined in WS-Policy Assertions. It indicates support for the Completion Protocol (Section 3.2)

/wsat:DurableTwoPhaseCommit
: This element is a policy assertion as defined in WS-Policy Assertions. It indicates support for the Two Phase Commit Protocol (Section 3.3.2)

/wsat:VolatileTwoPhaseCommit
: This element is a policy assertion as defined in WS-Policy Assertions. It indicates support for the Two Phase Commit Protocol (Section 3.3.1)

5. Transaction Faults

In addition to the faults defined in WS-Coordination, the following faults are available to Atomic Transaction implementations.

5.1 InconsistentInternalState

This fault is sent by a participant to indicate that it cannot fulfill its obligations. This indicates a global consistency failure and is an unrecoverable condition.

The qualified name of the fault code is:

```
wsat:InconsistentInternalState
```

6. Security Model

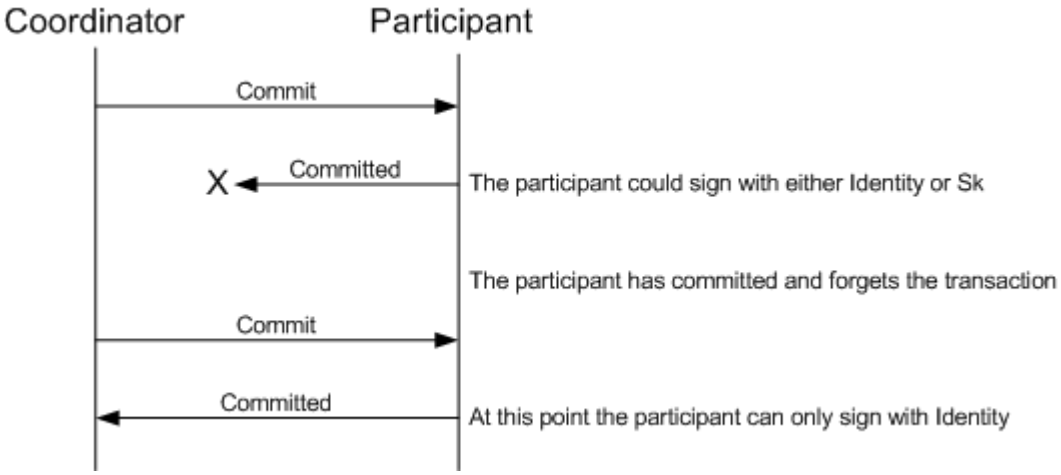
The security model for atomic transactions builds on the model defined in WS-Coordination [WSCOOR]. That is, services have policies specifying their requirements and requestors provide claims (either implicit or explicit) and the requisite proof of those claims. Coordination context creation establishes a base secret which can be delegated by the creator as appropriate.

Because atomic transactions represent a specific use case rather than the general nature of coordination contexts, additional aspects of the security model can be specified.

All access to atomic transaction protocol instances is on the basis of identity. The nature of transactions, specifically the uncertainty of systems means that the security context established to register for the protocol instance may not be available for the entire duration of the protocol.

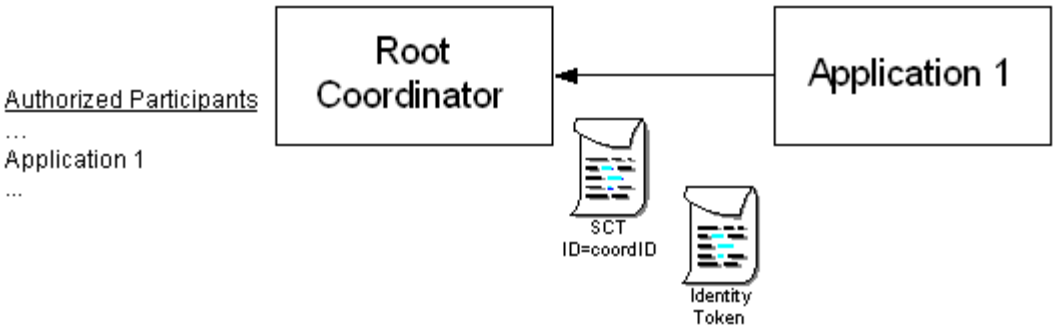
Consider for example the scenarios where a participant has committed its part of the transaction, but for some reason the coordinator never receives acknowledgement of the commit. The result is that when communication is re-established in the future, the coordinator will attempt to confirm the commit status of the participant, but the participant, having committed the transaction and forgotten all information associated with it, no longer has access to the special keys associated with the token.

The participant can only prove its identity to the coordinator when it indicates that the specified transaction is not in its log and assumed committed. This is illustrated in the figure below:



There are, of course, techniques to mitigate this situation but such options will not always be successful. Consequently, when dealing with atomic transactions, it is critical that identity claims always be proven to ensure that correct access control is maintained by coordinators.

There is still value in coordination context-specific tokens because they offer a bootstrap mechanism so that all participants need not be pre-authorized. As well, it provides additional security because only those instances of an identity with access to the token will be able to securely interact with the coordinator (limiting privileges strategy). This is illustrated in the figure below:



The "list" of authorized participants ensures that application messages having a coordination context are properly authorized since altering the coordination context ID will not provide additional access unless (1) the bootstrap key is provided, or (2) the requestor is on the authorized participant "list" of identities.

7. Security Considerations

It is strongly RECOMMENDED that the communication between services be secured using the mechanisms described in WS-Security [WSSec]. In order to properly secure messages, the body and all relevant headers need to be included in the signature. Specifically, the `<wscoor:CoordinationContext>` header needs to be signed with the body and other key message headers in order to "bind" the two together.

In the event that a participant communicates frequently with a coordinator, it is RECOMMENDED that a security context be established using the mechanisms described in WS-Trust [WSTrust] and WS-SecureConversation [WSSecConv] allowing for potentially more efficient means of authentication.

It is common for communication with coordinators to exchange multiple messages. As a result, the usage profile is such that it is susceptible to key attacks. For this reason it is strongly RECOMMENDED that the keys be changed frequently. This "re-keying" can be effected a number of ways. The following list outlines four common techniques:

- Attaching a nonce to each message and using it in a derived key function with the shared secret
- Using a derived key sequence and switch "generations"
- Closing and re-establishing a security context (not possible for delegated keys)
- Exchanging new secrets between the parties (not possible for delegated keys)

It should be noted that the mechanisms listed above are independent of the SCT and secret returned when the coordination context is created. That is, the keys used to secure the channel may be independent of the key used to prove the right to register with the activity.

The security context MAY be re-established using the mechanisms described in WS-Trust [WSTrust] and WS-SecureConversation [WSSecConv]. Similarly, secrets can be exchanged using the mechanisms described in WS-Trust. Note, however, that the current shared secret SHOULD NOT be used to encrypt the new shared secret. Derived keys, the preferred solution from this list, can be specified using the mechanisms described in WS-SecureConversation.

The following list summarizes common classes of attacks that apply to this protocol and identifies the mechanism to prevent/mitigate the attacks:

- **Message alteration** – Alteration is prevented by including signatures of the message information using WS-Security [WSSec].
- **Message disclosure** – Confidentiality is preserved by encrypting sensitive data using WS-Security.
- **Key integrity** – Key integrity is maintained by using the strongest algorithms possible (by comparing secured policies – see WS-Policy [WSPOLICY] and WS-SecurityPolicy [WSSecPolicy]).
- **Authentication** – Authentication is established using the mechanisms described in WS-Security and WS-Trust [WSTrust]. Each message is authenticated using the mechanisms described in WS-Security [WSSec].
- **Accountability** – Accountability is a function of the type of and string of the key and algorithms being used. In many cases, a strong symmetric key provides sufficient accountability. However, in some environments, strong PKI signatures are required.
- **Availability** – Many services are subject to a variety of availability attacks. Replay is a common attack and it is RECOMMENDED that this be addressed as described in the next bullet. Other attacks, such as network-level denial of service attacks are harder to avoid and are outside the scope of this specification. That said, care should be taken to ensure that minimal processing be performed prior to any authenticating sequences.
- **Replay** – Messages may be replayed for a variety of reasons. To detect and eliminate this attack, mechanisms should be used to identify replayed messages such as the timestamp/nonce outlined in WS-Security [WSSec]. Alternatively, and optionally, other technologies, such as sequencing, can also be used to prevent replay of application messages.

8. Use of WS-Addressing Headers

The messages defined in WS-Coordination and WS-AtomicTransaction can be classified into three types:

- Request messages: **CreateCoordinationContext** and **Register**.
- Reply messages: **CreateCoordinationContextResponse** and **RegisterResponse**.
- Notification messages: **Commit**, **Rollback**, **Committed**, **Aborted**, **Prepare**, **Prepared**, **ReadOnly** and **Replay**.

Request and reply messages follow the standard "Request Reply" pattern as defined in WS-Addressing. Notification messages follow the standard "one way" pattern as defined in WS-Addressing. There are two types of notification messages:

- A notification message is a terminal message when it indicates the end of a coordinator/participant relationship. **Committed**, **Aborted** and **ReadOnly** are terminal messages.
- A notification message is not a terminal message when it does not indicate the end of a coordinator/participant relationship. **Commit**, **Rollback**, **Prepare**, **Prepared** and **Replay** are not terminal messages.

The following statements define addressing interoperability requirements for the respective WS-Coordination and WS-AtomicTransaction message types:

Request messages

- MUST include a wsa:MessageID header.
- MUST include a wsa:ReplyTo header.

Reply messages

- MUST include a wsa:RelatesTo header, specifying the MessageID from the corresponding Request message.

Non-terminal notification messages

- MUST include a wsa:ReplyTo header

Terminal notification messages

- SHOULD NOT include a wsa:ReplyTo header

Notification messages are addressed by both coordinators and participants using the Endpoint References initially obtained during the Register-RegisterResponse exchange. If a wsa:ReplyTo header is present in a notification message it MAY be used by the recipient, for example in cases where a Coordinator or Participant has forgotten a transaction that is completed and needs to respond to a resent protocol message. Permanent loss of connectivity between a coordinator and a participant in an in-doubt state can result in data corruption.

All messages are delivered using connections initiated by the sender. Endpoint References MUST contain physical addresses and MUST NOT use well-known "anonymous" endpoint defined in WS-Addressing.

9. 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]
Web Services Coordination (WS-Coordination), Microsoft, IBM, and BEA Systems October 2004

[WSADDR]
Web Services Addressing (WS-Addressing), Microsoft, IBM, Sun, BEA Systems, SAP, Sun, August 2004

[WSPOLICY]
Web Services Policy Framework (WS-Policy), VeriSign, Microsoft, Sonic Software, IBM, BEA Systems, SAP, September 2004

[WSPOLICYASSERT]
Web Services Policy Assertions Language (WS-PolicyAssertions), Microsoft, IBM, BEA Systems, SAP, May 2003

[WSPOLICYATTACH]
Web Services Policy Attachment (WS-PolicyAttachment), VeriSign, Microsoft, Sonic Software, IBM, BEA Systems, SAP, September 2004

[WSDL]
Web Services Description Language (WSDL) 1.1
"http://www.w3.org/TR/2001/NOTE-wsdl-20010315"

[WSSec]
OASIS Standard 200401, March 2004, "Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)"

[WSSecPolicy]
Web Services Security Policy Language (WS-SecurityPolicy), Microsoft, VeriSign, IBM, RSA Security, 18 December 2002

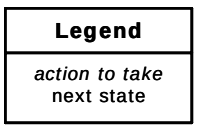
[WSSecConv]
Web Services Secure Conversation Language (WS-SecureConversation), OpenNetwork, Layer7, Netegrity, Microsoft, Reactivity, IBM, VeriSign, BEA Systems, Oblix, RSA Security, Ping Identity, Westbridge, Computer Associates, May 2004

[WSTrust]
Web Services Trust Language (WS-Trust), OpenNetwork, Layer7, Netegrity, Microsoft, Reactivity, VeriSign, IBM, BEA Systems, Oblix, RSA Security, Ping Identity, Westbridge, Computer Associates, May 2004

10. State Tables

The following state tables specify the behavior of coordinators and participants when presented with protocol messages or internal events. These tables present the view of a coordinator or participant with respect to a single partner. A coordinator with multiple participants can be understood as a collection of independent coordinator state machines.

Each cell in the tables uses the following convention:



Each state supports a number of possible events. Expected events are processed by taking the prescribed action and transitioning to the next state. Unexpected protocol messages will result in a fault message, with a standard fault code such as Invalid State or Inconsistent Internal State. Events that may not occur in a given state are labelled as N/A.

Atomic Transaction 2PC protocol (Coordinator View)							
Inbound Events	**States**						
	None	**Active**	**Preparing**	**Prepared**	**PreparedSuccess**	**Committing**	**Aborting**
Register	*Invalid State* None	*Send RegisterResponse* Active	*Durable: Invalid State* Aborting Volatile:*Send RegisterResponse* Active	*N/A*	*Invalid State* PreparedSuccess	*Invalid State* Committing	*Invalid State* Aborting
Prepared	*Durable: Send Rollback Volatile: Invalid State* None	*Invalid State* Aborting	*Record Vote* Preparing	*N/A*	*Ignore* PreparedSuccess	*Resend Commit* Committing	*Resend Rollback, and forget* Aborting
ReadOnly	*Ignore* None	*Forget* Active	*Forget* Preparing	*N/A*	*Invalid State* PreparedSuccess	*Invalid State* Committing	*Forget* Aborting
Aborted	*Ignore* None	*Forget* Aborting	*Forget* Aborting	*N/A*	*Invalid State* PreparedSuccess	*Invalid State* Committing	*Forget* Aborting
Committed	*Ignore* None	*Invalid State* Aborting	*Invalid State* Aborting	*N/A*	*Invalid State* PreparedSuccess	*Forget* Committing	*Invalid State* Aborting
Replay	*Durable: Send Rollback Volatile: Invalid State* None	*Send Rollback* Aborting	*Send Rollback* Aborting	*N/A*	*Ignore* PreparedSuccess	*Send Commit* Committing	*Send Rollback* Aborting
Internal Events							
User Commit	*Return Aborted* None	*Send Prepare* Preparing	*Ignore* Preparing	*N/A*	*Ignore* PreparedSuccess	*Return Committed* Committing	*Return Aborted* Aborting
User Rollback	*Return Aborted* None	*Send Rollback* Aborting	*Send Rollback* Aborting	*N/A*	*Invalid State* PreparedSuccess	*Invalid State* Committing	*Return Aborted* Aborting
Expires Times out	*N/A*	*Send Rollback* Aborting	*Send Rollback* Aborting	*N/A*	*Ignore* PreparedSuccess	*Ignore* Committing	*Ignore* Aborting
Comms Times out	*N/A*	*N/A*	*Resend Prepare* Preparing	*N/A*	*N/A*	*Resend Commit* Committing	*N/A*
Commit Decision	*N/A*	*N/A*	*Record Outcome* PreparedSuccess	*N/A*	*N/A*	*N/A*	*N/A*
Write Done	*N/A*	*N/A*	*N/A*	*N/A*	*Send Commit* Committing	*N/A*	*N/A*
Write Failed	*N/A*	*N/A*	*N/A*	*N/A*	*Send Rollback* Aborting	*N/A*	*N/A*
All Forgotten	*N/A*	Active	None	*N/A*	*N/A*	None	None

Notes:

1. Transitions with a “N/A” as their action are inexpressible. A TM should view these transitions as serious internal consistency issues, and probably fatal.
2. Internal events are those that are created either within a TM itself, or on its local system.
3. “Forget” implies that the subordinate’s is participation is removed from the coordinator (if necessary), and otherwise the message is ignored

Atomic Transaction 2PC protocol (Participant View)							
Inbound Events	**States**						
	None	**Active**	**Preparing**	**Prepared**	**PreparedSuccess**	**Committing**	**Aborting**
Register Response	*Register Subordinate* Active	*Invalid State* Active	*Invalid State* Aborting	*Invalid State* Prepared	*Invalid State* PreparedSuccess	*Invalid State* Committing	*Invalid State* Aborting
Prepare	*Send Aborted* None	*Gather Vote Decision* Preparing	*Ignore* Preparing	*Ignore* Prepared	*Resend Prepared* PreparedSuccess	*Ignore* Committing	*Resend Aborted, and forget* Aborting
Commit	*Send Committed* None	*Invalid State* Aborting	*Invalid State* Aborting	*Invalid State* Aborting	*Initiate commit decision* Committing	*Ignore* Committing	*InconsistentInternalState* Aborting
Rollback	*Send Aborted* None	*Initiate Rollback, Send Aborted, and Forget* Aborting	*Initiate Rollback, Send Aborted, and Forget* Aborting	*Initiate Rollback, Send Aborted, and Forget* Aborting	*Initiate Rollback, Send Aborted, and Forget* Aborting	*InconsistentInternalState* Committing	*Send Aborted, and Forget* Aborting
Internal Events							
Expires Times out	*N/A*	*Send Aborted* Aborting	*Send Aborted* Aborting	*Ignore* Prepared	*Ignore* PrepareSuccess	*Ignore* Committing	*Ignore* Aborting
Comms Times out	*N/A*	*N/A*	*N/A*	*N/A*	*Resend Prepared* PreparedSuccess	*N/A*	*N/A*
Commit Decision	*N/A*	*N/A*	*Record Commit* Prepared	*N/A*	*N/A*	*Send Committed and Forget* Committing	*N/A*
Rollback Decision	*N/A*	*N/A*	*Send Aborted* Aborting	*N/A*	*N/A*	*N/A*	*N/A*
Write Done	*N/A*	*N/A*	*N/A*	*Send Prepared* PreparedSuccess	*N/A*	*N/A*	*N/A*
Write Failed	*N/A*	*N/A*	*N/A*	*Initiate Rollback, Send Aborted, and Forget* Aborting	*N/A*	*N/A*	*N/A*
All Forgotten	None	*N/A*	*Send ReadOnly* None	*N/A*	*N/A*	None	None

Notes:

1. Transitions with a “N/A” as their action are inexpressible. A TM should view these transitions as serious internal consistency issues, and probably fatal.
2. Internal events are those that are created either within a TM itself, or on its local system.