Java™ 2 Enterprise Edition

XML API for Java™ Transactions

JSR 156

Specification Lead:
Mark Little
Hewlett-Packard

Technical comments:

Version 0.2
Expert Group Draft

Trademarks

Table of Contents

1. Introduction

This document describes the system design and interfaces for XML based transactions. JAXTX is the realization, within the J2EE programming model, of work that has been conducted in various commercial, academic, and standards organizations into enabling Web Services with the critical transactional capabilities that they need.

An increasingly large number of distributed applications are constructed by being composed from existing applications. The resulting applications can be very complex in structure, with complex relationships between their constituent applications. Furthermore, the execution of such an application may take a long time to complete, and may contain long periods of inactivity, often due to the constituent applications requiring user interactions. In a loosely coupled environment like the Web, it is inevitable that long running applications will require support for fault-tolerance, because machines may fail or services may be moved or withdrawn. A common technique for fault-tolerance is through the use of atomic transactions, which have the well know ACID properties[1], operating on persistent (long-lived) objects. Transactions ensure that only consistent state changes take place despite concurrent access and failures.

Traditional transaction processing systems are sufficient if an application function can be represented as a single top-level transaction. Frequently this is not the case. Top-level transactions are most suitably viewed as “short-lived” entities, performing stable state changes to the system; they are less well suited for structuring “long-lived” application functions (e.g., running for minutes, hours, days, …). Long-lived top-level transactions may reduce the concurrency in the system to an unacceptable level by holding on to resources (e.g., locks) for a long time; further, if such a transaction aborts, much valuable work already performed could be undone.

As has been shown elsewhere [1] there are a number of transaction models, each suited to different types of applications and services. The J2EE Activity Service work of JSR 95 is intended to support the development of these models and others. However, it does not concentrate on how these implementations can be exposed as Web Services and interacted with via SOAP and XML. Alternatively, some implementations of Web Services transactions already exist, e.g., BTP [2] and provide solutions for the specific problem domains. Other models will undoubtedly be developed to solve different application requirements. In addition, although ACID transactions may not be the solution for all transactional Web Service applications, it is likely that some applications will continue to require strict ACID semantics. Therefore, existing implementations of JTA/JTS may well require a Web Services persona.

[1] Atomic, Consistent, Isolated, Durable.

The purpose of JAXTX is to identify the common components within (traditional and extended) transaction systems and provide a standard set of interfaces to these components. Importantly, JAXTX interfaces do not imply a specific implementation for the underlying transaction system to which they talk. This is to allow a single set of interfaces to interact with a wide range of implementations. It is the purpose of this specification to enable existing XML-based transaction implementations to be integrated with JAXTX without requiring them to be modified. However, this specification also provides optional protocol definitions to allow new implementations to be defined entirely within the context of JAXTX.

Note, in the rest of this document we shall use the term *ACID transaction* to refer to traditional transaction functionality as presented by, for example, JTA. The terms *transaction* or *extended transaction*, shall refer to models that give different ref functionality to ACID transactions, for example by relaxing various aspects of the ACID properties.

1.1 Scope

This document and related javadoc describes the architecture, functionality and interfaces that must be provided by an implementation of JAXTX in order to support XML transactions.

Specific implementations of transaction services (e.g., JTA or BTP) will be able to use the interfaces defined by JAXTX to provide a uniform and standard API for their use.

1.2 Target Audience

The target audience of this specification includes:

- implementers of transaction services (e.g., JTA or BTP) who wish to make their components available as Web Services.
- implementers of application servers and EJB containers.
- implementers of Web Services that require some level of transactionality.

1.3 Organization

This document describes the architecture of JAXTX with its components and roles described.

Specific interfaces are described in general terms in this document and in more detail in the accompanying `javadoc` package.

1.4 Document Convention

A regular Times New Roman font is used for describing the architecture.

A regular `Courier` font is used when referencing Java interfaces and methods on those interfaces.

1.5 J2EE Activity Service Expert Group

The expert group consists of the following members:

Mark Little, mark_little@hp.com

Hewlett-Packard,
Arjuna Labs,
Central Square South,
Orchard Street,
Newcastle upon Tyne,
NE1 3AZ,
UK

Michael Abbot, mike@codemetamorphosis.com

15 Berenda Way Suite 300,
Portola Valley, CA 94028,
USA

Rahul Bhargava, rahul_technical@yahoo.com

Netscape Communications,
MS MV-015,
465 Ellis Street,
Mtn View, CA 94043,
USA

Pyounguk Cho, pyounguk.cho@iona.com

IONA Technologies,
2350 Mission College Blvd.,
Suite 650,
Santa Clara, CA 95054,
USA

Alan Davies, adavies@seebeyond.com

SeeBeyond Technology Corporation,
404 E Huntington Drive,

Monrovia,
CA 91016,
USA

Keith Evans, keith.b.evans@hp.com

Hewlett-Packard/Compaq Computer Corporation,
CAC16-01, Room 162100,
10100 North Tantau Avenue,
Cupertino, CA 95014,
USA

Peter Furniss, peter.furniss@choreology.com

Choreology Ltd,
13 Austin Friars,
London, EC2N 2JX,
UK

Christoph Liebig, christoph.liebig@sap.com

SAP AG
Neurottstr. 16
STERN1OG
Walldorf 69190
Germany

Kavantzas Nickolaos, nkavantz@us.oracle.com

Oracle,
500 Oracle Parkway,
MS 4op9,
Redwood Shores, 94065,
USA

Bill Pope, zpope@pobox.com

PO Box 6698,
Portsmouth NH 03801,
USA

Ian Robinson, ian_robinson@uk.ibm.com

International Business Machines,
IBM Hursley Lab,
Hursley,
Winchester,

Hant, SO21 2JN,
UK

Anne Thomas, atm@systinet.com

Idoox,
1 Broadway,
14th Floor,
Cambridge, MA 02142,
USA

Peter Walker, Peter.Walker@sun.com

Sun Microsystems, Inc.
901 San Antonio Road
MS USCA14-303
Palo Alto, CA 94303

1.6 Acknowledgements

2. Overview

Comment: Depending on your point of view, this section may be too long or not detailed enough. We're not talking about a field that has as many books, papers and other reference material as, say, traditional ACID transactions. So, there's an argument to be had that we need to make this document as self-contained as possible. However, at this point unless anyone strongly objects, I'd like to concentrate on the core of the specification and then come back to this.

Business-to-business (B2B) interactions may be complex, involving many parties, spanning many different organisations, and potentially lasting for hours or days. For example, the process of ordering and delivering parts for a computer which may involve different suppliers, and may only be considered to have completed once the parts are delivered to their final destination. Unfortunately, for a number of reasons (not least of which are commercial) parties involved in B2B simply cannot afford to lock (reserve) their resources exclusively on behalf of an individual indefinitely, thus ruling out the use of atomic transactions for all applications.

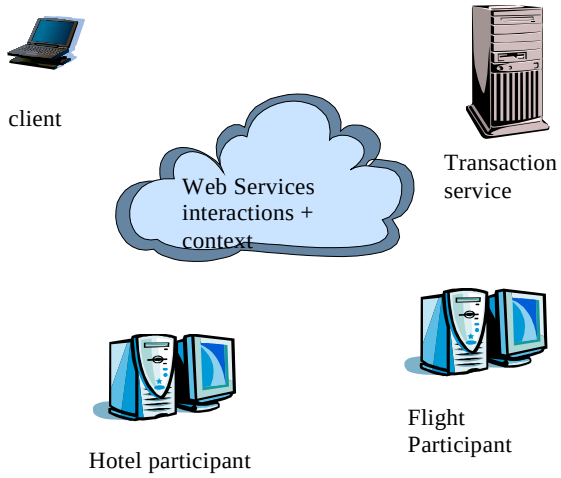

Figure 1: Web Services transaction actors.

In general a business transaction requires the capability for certain participants to be structured into a *consensus group*, such that all of the members in such a grouping agree to the same outcome. For example, consider the case shown in Figure 1, where a transaction is begun by the actions of a client to book a hotel room and a flight. Both of these actions are required or neither is, i.e., no partial outcome will be tolerated. The hotel and flight booking web services are therefore required to enrol respective participants with the transaction. When the client ends the transaction, the transaction service will ensure that both participants reach the same outcome such that *either* the hotel and flight are booked or neither operation occurs.

Importantly, different participants within the same business transaction may belong to different consensus groups. The business logic then controls how each group completes. In this way, a business transaction (driven, for example, by a workflow system) may

cause a subset of the groups it uses to perform the work it asks, while asking the other groups to undo the work.

2.1 When ACID is too strong

To ensure atomicity between multiple participants, a multi-phase (typically two) consensus mechanism is required, as illustrated in Figure 2: during the first (*preparation*) phase, an individual participant must make durable any state changes that occurred during the scope of the atomic transaction, such that these changes can either be rolled back (undone) or committed later once consensus to the transaction outcome has been determined amongst all participants, i.e., any original state must not be lost at this point as the atomic transaction could still roll back. Assuming no failures occurred during the first phase (in which case all participants will be forced to undo their changes), in the second (*commitment*) phase participants may make the new replace the original state with the new durable state.

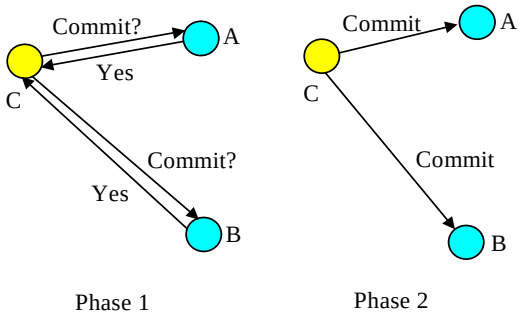

Figure 2: Two-phase commit protocol.

In order to guarantee consensus, two-phase commit is necessarily a blocking protocol: after returning the phase 1 response, each participant which returned a commit response *must* remain blocked until it has received the coordinator's phase 2 message telling it what to do. Until they receive this message, any resources used by the participant are unavailable for use by other atomic transactions, since to do so may result in non-ACID behaviour. If the coordinator fails before delivery of the second phase message these resources remain blocked until it recovers. In addition, if a participant fails after phase 1, but before the coordinator can deliver its final commit decision, the atomic transaction cannot be completed until the participant recovers: *all* participants must see *both* phases of the commit protocol in order to guarantee ACID semantics. There is no implied time limit between a coordinator sending the first phase message of the commit protocol and it sending the second, commit phase message; there could be seconds or hours between them.

2.2 Extended transactions

Structuring certain activities from long-running atomic transactions can reduce the amount of concurrency within an application or (in the event of failures) require work to be performed again. For example, there are certain classes of application where it is known that resources acquired within an atomic transaction can be released "early", rather than having to wait until it terminates; in the event of the atomic transaction rolling back, however, certain compensation activities may be necessary to restore the system to a consistent state. Such compensation activities (which may perform forward or backward recovery) will typically be application specific, may not be necessary at all, or may be more efficiently dealt with by the application.

For example long-running activities can be structured as many independent, short-duration top-level atomic transactions, to form a "logical" long-running transaction. This structuring allows an activity to acquire and use resources for only the required duration of this long-running activity. This is illustrated in Figure 3, where an application activity (shown by the dotted ellipse) has been split into many different, coordinated, short-duration top-level atomic transactions. Assume that the application activity is concerned with booking a taxi (*t1*), reserving a table at a restaurant (*t2*), reserving a seat at the theatre (*t3*), and then booking a room at a hotel (*t4*), and so on. If all of these operations were performed as a single atomic transaction then resources acquired during *t1* would not be released until the top-level atomic transaction has terminated. If subsequent activities *t2*, *t3* etc. do not require those resources, then they will be needlessly unavailable to other clients.

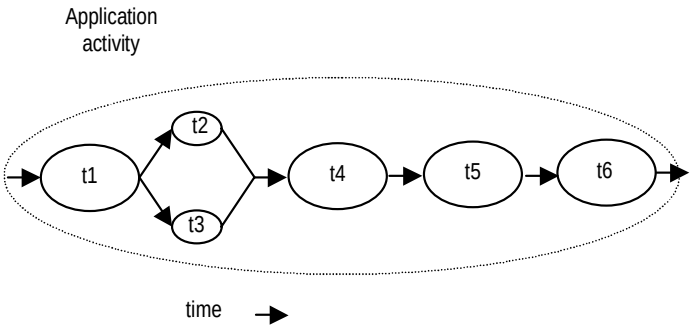

Figure 3: An example of a logical long-running "transaction", without failure.

However, if failures and concurrent access occur during the lifetime of these individual transactional activities then the behaviour of the entire "logical long-running transaction" may not possess ACID properties. Therefore, some form of (application specific) compensation may be required to attempt to return the state of the system to consistency. A transactional workflow system can be used to provide scripting facilities for expressing

the composition of these transactions with specific compensation activities where required. For example, let us assume that *t4* aborts. Further assume that the application can continue to make forward progress, but in order to do so must now undo some state changes made prior to the start of *t4* (by *t1*, *t2* or *t3*). Therefore, new activities are started; *tc1* which is a compensation activity that will attempt to undo state changes performed, by say *t2*, and *t3* which will continue the application once *tc1* has completed. *tc5'* and *tc6'* are new activities that continue after compensation, e.g., since it was not possible to reserve the theatre, restaurant and hotel, it is decided to book tickets at the cinema. Obviously other forms of composition are possible.

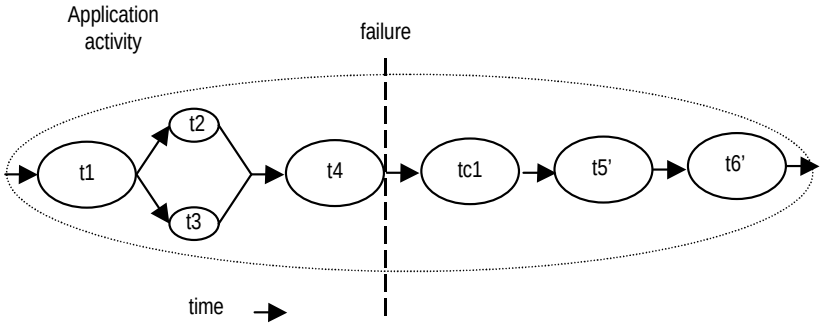

Figure 4: An example of a logical long-running "transaction", with failure.

Much research on structuring applications out of transactions has been influenced by the ideas of *spheres of control* [3]. There are several ways in which some or all of the application requirements outlined above could be met [4][5][6][7][8].

- *Independent top-level transactions*: with this mechanism it is possible to invoke a top-level transaction from within another transaction. If the invoking transaction rolls back, this does not lead to the automatic rollback of the invoked transaction, which can commit or rollback independently of its invoker, and hence release resources it acquires. Such transactions could be invoked either synchronously or asynchronously. In the event that the invoking transaction rolls back and compensation is required, compensating transactions may be invoked automatically by the transaction system or dealt with by the application.

- *Glued transactions*: long-running top-level transactions can be structured as many independent, short-duration top-level transactions, to form a “logical” long-running transaction; the duration between the end of one transaction and the beginning of another is not perceivable, and selective resources can be atomically passed from one transaction to the next. This structuring allows an activity to acquire and use resources for only the required duration of this long-running transactional activity. In the event of failures, to obtain transactional semantics for the entire long-running transaction may require compensation transactions which can perform forward or backward recovery.
- *Transactional workflows*: a transactional workflow system can be used to provide scripting facilities for expressing the composition of an activity (a business process) out of other activities (which could be transactional), with specific compensation activities.
- *Business transaction protocol*: the OASIS Business Transaction Protocol introduced the notion of an *open-top* two-phase termination protocol. Work is conducted within the scope of *atoms*, which are similar to atomic transactions and have the same all-or-nothing guarantees. However, the outcome of atoms are controlled by *cohesions* which allow their selective confirmation or cancellation by explicitly driving the two-phase protocol over a period of hours, days etc.

From a user’s (client/service) perspective, differences in how extended transaction protocols are coordinated and controlled are typically minimal: the user starts the transactional activity, communicates with services and propagates information about the transaction (the *context*), and terminates the transaction. Most differences occur between the transaction controller (the coordinator) and its enlisted participants. For example, does it use a two-phase or three-phase termination protocol?

2.3 Pluggable transactionality

What does it mean for a client to begin a BTP transaction, a JTA transaction, or a glued transaction, for example? In many cases the answer is very little, since much of the transactionality guarantees and semantic knowledge reside within and between the services and the coordinator with which the client interacts. Therefore, conceptually a single client implementation could interact with several different transaction services according to requirements placed on the application by: the services, the environment, the business needs, etc. In a Web Services environment, the services used by a client may be dynamically selected based upon the transaction model(s) they support (e.g., specified in UDDI) by a layer between the client and the actual service, explicitly by the business logic etc. At the level of the client/user all they are typically interested in is starting a transaction and performing work within its scope, leaving the actual transactional specific aspects to the services and the coordinator.

As a result, this specification aims to provide an interface to these different transaction protocols that allows different implementations to be plugged in. If necessary the user can introspect the underlying implementation to determine specific attributes and properties, but we believe that typically this will not be necessary. In addition, if specific models require enhanced interfaces in order to operate, these can be obtained in a structured and well defined manner.

JAXTX components allow the management in a Web services interaction of a number of *activities* or *tasks* related to an overall application. In particular JAXTX allows a user to:

- define demarcation points which specify the start and end points of transactional activities.
- register participants for the activities that are associated with the application.
- propagate transaction-specific information across the network.

The main components involved in using and defining JAXTX are illustrated in Figure 5:

1) A *Transaction Service*: Defines the behaviour for a specific transaction model. The Transaction Service provides a processing pattern (implemented by the *transaction coordinator*) that is used for outcome processing. For example, an ACID transaction service is one implementation that provides a two-phase protocol definition whose coordination sequence processing includes *prepare*, *commit* and *rollback*. Other examples of Transaction Service implementations include patterns such as Sagas, Collaborations, Nested or Real-Time transactions and non-transactional patterns such as Cohesions and Correlations. Multiple Transaction Service implementations may co-exist within the same application and processing domain. JAXTX does not specify how a Transaction Service is implemented: one service may be tied to a specific protocol, or the service could support multiple different protocols as in [1].
2) A *Transaction API*: Provides an interface for transaction demarcation and the registration of participants.
3) A *Participant*: The operation or operations that are performed as part of the transaction coordination sequence. During the lifetime of a transaction a coordinator may send many different XML messages to a participant, which may perform specific units of work depending upon how it interprets these XML messages.
4) The *Context*: this contains information necessary for services to use the transaction, e.g., enlist participants and scope work.

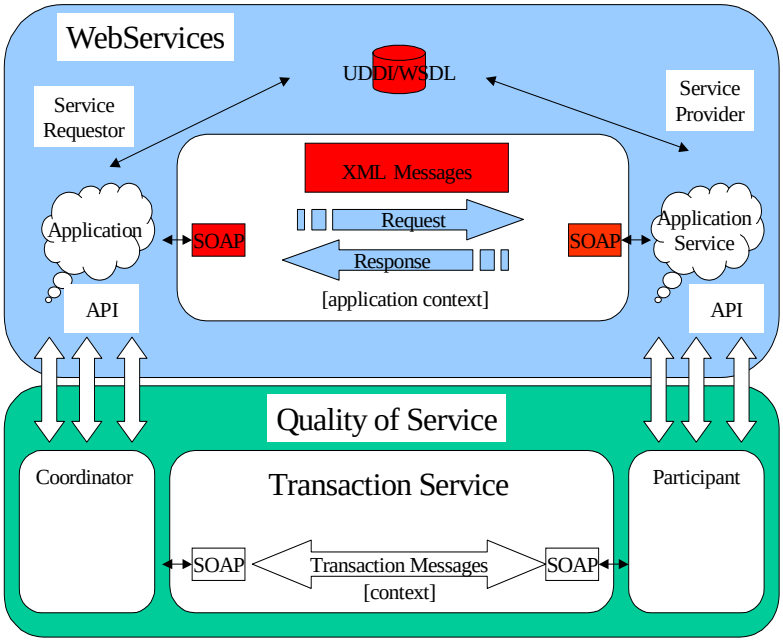

Figure 5: Web Services, transactions and contexts.

This specification does not provide a means whereby existing transaction service implementations can be provided to Web Services and their users via, say, SOAP and XML, since it is assumed that this has been done elsewhere in other specifications. Neither is it meant to place additional requirements on such implementations (e.g., the OASIS Business Transactions Protocol). What JAXTX does accomplish is to allow these implementations to be plugged into a common services architecture. Note, however, that JAXTX does specify protocols for new transaction service implementations to use that will allow for their straightforward integration with this architecture.

Later sections in this specification contain protocol specific information for Java Web Services transaction implementations that do not occur in other specifications. For example, a definition of how components of the JTA specification may be incorporated into Web Services. Although these mappings are optional, we recommend that providers of JAXTX who also support implementations of these additional transaction services use these mappings.

2.4 The invocation protocol

We do not assume that a single remote invocation mechanism (e.g., SOAP) will be the natural communication medium for all Web Services. How participants within and between activities appear to each other is not central to this discussion. They may be CORBA objects, communicating via IIOP, or they may be coarser grained Web Services objects, communicating via SOAP, for example. We assume that they will use the most appropriate invocation protocol for the application, e.g., it is unlikely that there will be much real-time video streaming over SOAP/HTTP. This does not preclude a given application from using multiple object models and communication protocols simultaneously.

Furthermore, we assume that protocol negotiation will occur on many levels (e.g., which transaction models are supported by a Web Service and in use by the invoker, which communications protocols are provided, business level issues, etc.) We do not prescribe when (or even if) such negotiation occurs.

3. JAXTX architecture

A *transaction* is a unit of (distributed) work, involving one or more parties (services, components, objects). A transaction is *created*, made to *run*, and then *completed*. An *outcome* is the result of a completed transaction and this can be used to determine subsequent flow of control to other transactions. Transactions may run over long periods of time (minutes, hours, days, …) and can thus be *suspended* and then *resumed* later.

A very high level view of the role of JAXTX within a Web Service architecture is shown in Figure 6. The transaction coordination service is responsible for managing transactions created by the client. Work done by the client may be executed under the scope of a transaction such that the coordinator will ultimately ensure that the work is performed in some manner consistent with the underlying transaction model (e.g., that it either happens or does not happen despite failures).

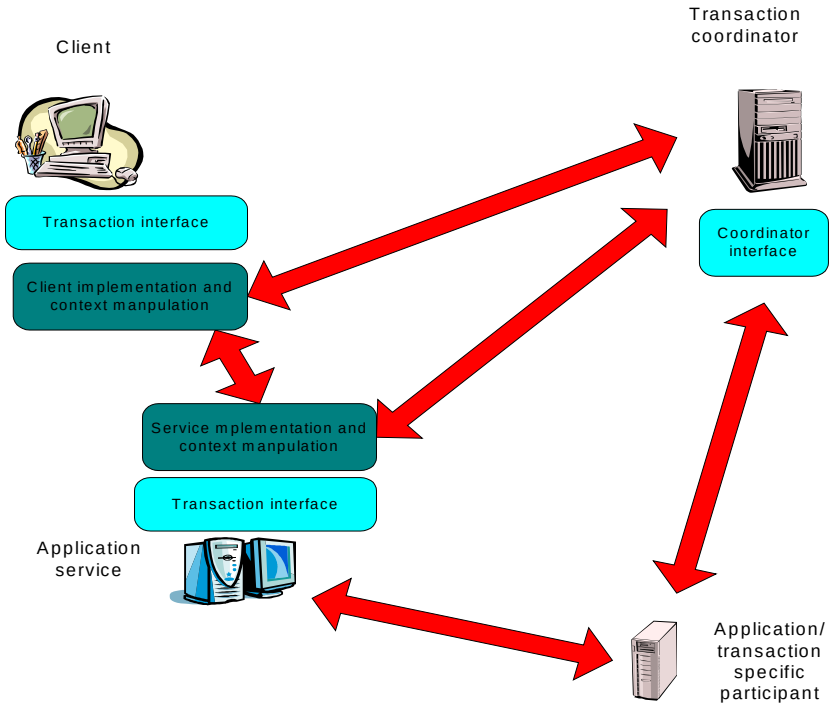

Figure 6: Applications and implementation components.

When a client performs an invocation on a service such that the work is required to be transactional, the client (or the infrastructure in which it is located) must flow

information about the transaction (the context) to the service. The service (or entities it interacts with, e.g., a JDBC driver) is then responsible for ensuring that the work is performed in such a way that any final decision on the outcome is controlled by the transaction coordinator. This normally occurs by enlisting a participant in the transaction that has control over the work, e.g., an `XAResource` that can issue *prepare*, *commit* or *rollback* on an unlderlying database when instructed to by the coordinator.

It is envisioned that in most situations the implementation of the client/service protocol (essentially the interactions shown) and context manipulation are provided by existing standards and systems. For example, the OASIS Business Transactions Protocol (BTP) defines precisely how the transaction context appears in transmitted XML documents, how coordinator specific messages are formatted, how participants behave under certain failure circumstances etc. However, the language interfaces to these components (e.g., the coordinator interface) may not be defined in these specifications. JAXTX provides a uniform set of interfaces for Java that are intended to indirect to the underlying implementations.

In other circumstances, it is possible that no appropriate transaction protocol currently exists in the Web services domain. As such, one of the goals of JAXTX is to support the development of these new protocols. Therefore, this specification provides optional protocols (including XML schemas) that designers can use in order to simplify their work and ease of integration within this framework.

3.1 The components and their roles

Before describing the interfaces which JAXTX provides, we shall first examine the components within a JAXTX architecture, as shown in Figure 7.

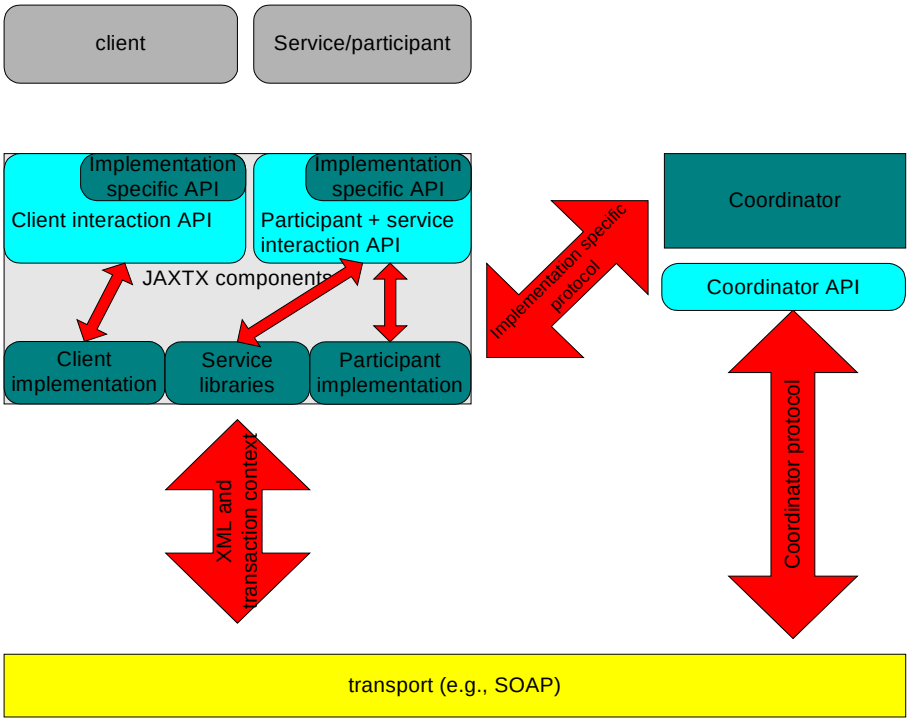

Figure 7: JAXTX architecture.

3.1.1 The XML context

In order for a transaction to span a distributed number of services/tasks, certain information has to flow between the sites/domains involved in the application. This is commonly referred to as the *context* and typically contains the following information:

The context typically includes the following information:

- A transaction identifier which guarantees global uniqueness for an individual activity (such an identifier can also be thought of as a “correlation” identifier or a value that is used to indicate that a task is part of the same work activity).
- The transaction coordinator location or endpoint address so participants can be enrolled.
- Transaction Service specific information, e.g., if the Service implements an ACID transaction model then this information may contain the transaction hierarchy that existed at the sending side in order that the importing domain may duplicate this hierarchy.

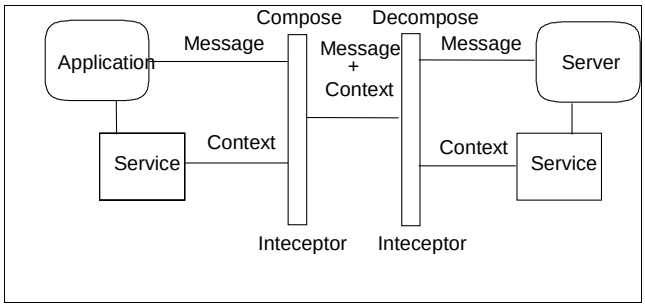

Figure 8: Services and context flow.

The context information is propagated to provide a flow of context information between distributed execution environments, for example using SOAP header information. This may occur transparently to the client and application services. The context may be propagated as part of normal message interchange within an application (e.g., as an additional part of the SOAP header) or may be sent as an explicit "out-of-band" context message, e.g., where the content of the message contains the context and a handle to link it to specific application messages which will be sent separately. Context propagation may also occur using different protocols than those used by the application. Therefore, JAXTX does not assume a specific means by which contexts are propagated, leaving this up to the specific coordination service implementation.

In those cases where existing XML-based transaction systems are being integrated into a JAXTX implementation, the context (and associated protocol) will already be defined. Rather than insist that such systems be changed to work with JAXTX, there is no mandatory context (or protocol) defined by this specification. However, for new transaction implementations that address problems not provided for by existing implementations, JAXTX does provide an optional context definition.

3.1.2 The Web Service

The *Web Service*, e.g., the taxi booking service. Whenever a user contacts a service whose work it wishes to be under the control of a transaction, components of the transaction system are responsible for flowing the *context* to that service. The service can then use this information to enlist a participant with the transaction. Note, in the rest of this document we shall use the term *container* when referring to the entity that is responsible for hosting a specific service, receiving requests for it, etc. This specification does not make any assumptions about the implementation of such a container and shall attempt to talk in terms of required functionality only. Note, a web service may also play the role of a participant.

3.1.3 The participant

The *participant* is the entity that actually does the real transaction work. The Web Service (e.g., a theatre booking system) contains some business logic for reserving a seat, enquiring availability etc, but it will need to be back-ended by something that maintains information in a durable manner. Typically this will be a database, but it could be a file system, NVRAM, etc. Now, although the service may talk to the back-end database directly, it cannot commit or roll back any changes it (the service) makes, since these are ultimately under the control of the transaction that scoped the work. In order for the transaction to be able to exercise this control, it must have some contact with the back-end resource (the database in our example) and this is accomplished by the participant.

Each participant supports a termination protocol specific to the transaction model implemented by the coordinator (e.g., two-phase commit). In addition, the work that a participant performs when instructed by the coordinator that its transaction is terminating is dependant on its implementation (e.g., commit the reservation of the theatre ticket). The participant will then return an indication of whether or not it succeeded. Unlike in ACID transactions, some extended transaction models may allow participant implementations that do not have to guarantee that they can remain in a *prepared state*; an implementation may indicate that it can only do so for a specified period of time, and also indicate what action it will take (confirm or undo) if it has not been told how to finish before this period elapses.

As with the transaction context, JAXTX provides an optional protocol definition for coordinator-participant interactions.

3.1.4 The coordinator

Associated with every transaction is a *coordinator*, which is responsible for governing the outcome of the transaction. The coordinator may be implemented as a separate service or may be co-located with the user for improved performance. It communicates with enlisted participants to inform them of the desired termination requirements, i.e., whether they should accept (e.g., *commit*) or reject (e.g., *roll back*) the work done within the scope of the given transaction. For example, whether to purchase the (provisionally reserved) flight tickets for the user or to release them. This communication will be an implementation specific protocol (e.g., two or three phase completion).

A transaction manager is typically responsible for managing coordinators for many transactions. The initiator of the transaction (e.g., the client) communicates with a transaction manager and asks it to start a new transaction and associate a coordinator

with the transaction. Once created the context can be propagated to Web Services in order for them to associate their work with the transaction.

3.2 The interfaces and protocols

3.2.1 Protocols

An application/client may wish to terminate a transaction in a number of different ways (e.g., commit or rollback). However, although the coordinator may attempt to terminate in a manner consistent with that desired by the client, it is ultimately the interactions between the coordinator and the participants that will determine the final outcome.

The context and coordinator-to-participant protocol are specific to the type of transaction model provided by the implementation. If an existing XML-based transaction service implementation is being integrated with JAXTX, then they will have a format that is specified in other protocol documents, e.g., BTP [2]. It is not the domain of this specification to impinge on these protocols and require changes to them in any way. As such, it is assumed that these will be specified elsewhere and relevant JAXTX component implementations will conform to them.

Although it is primarily expected that JAXTX will be used to provide a higher-level API for existing XML transaction systems, this specification does provide an optional XML defined protocol that may be used by designers when building new Web Service transaction implementations.

3.2.2 Interfaces

JAXTX defines both mandatory and optional interfaces. The mandatory interfaces can be categorised as follows:

- *Client*: these interfaces provide clients with a uniform means by which they may access and use client-specific functionality from different transaction implementations. These interfaces are designed to provide means whereby aspects of the underlying implementation may be accessed in a controlled manner. Typically these interfaces will be concerned with demarcating transactions.
- *Implementation*: the interfaces are for client-side extensions required for specific transaction models which may not be covered by the more generic client interfaces, e.g., when using the OASIS Business Transactions Protocol, it is often preferable to drive both phases of the two-phase completion protocol explicitly, whereas in a traditional transaction model the individual phases are hidden from users.

The optional interfaces are:

- *Service*: these interfaces provide services with a means by which they may access service-specific functionality from different transaction implementations. For example, a service provider may be able to support multiple different transaction protocols and use different participants for each; when the transaction implementation is imported by the service, it may determine the type of the transaction and act accordingly. Other service-side interfaces include participants and the means by which they are enlisted with transactions.

3.3 Combining implementations

Since we believe that there will be several different transaction implementations co-existing within the Web Services environment, it is logical to assume that some applications and services may require their use concurrently. As such, the JAXTX architecture does not place any limitations on the number of transaction implementations that may be utilised within a single application or JVM. Whether or not it makes sense for multiple implementations to co-exist, or for how they may inter-work (assuming inter-working is required) is beyond the scope of the JAXTX specification.

3.4 XML configuration

In order to select a specific transaction implementation for use, it is necessary to be able to define precisely the type of transaction service that is required. This must occur in an extensible manner that does not require changes to application code should the requirements or transaction definition change. As such, the use of XML is appropriate and JAXTS provides a *transaction-definition schema*.

Each implementer of a specific transaction service that can be accessed via JAXTX is required to define its characteristics using the JAXTX schema. These characteristics fall into two classes:

- *Mandatory*: this set of attributes is required to uniquely identify the type of transaction model supported by this implementation. Multiple implementations may support the same model and if differentiation is required based upon implementation specifics, then the optional fields may be used.
- *Optional*: the data maintained in these fields contains information that is not concerned with the specific model implemented by the service but which may still be relevant for users. For example, quality of service specifications (reliability requirements, performance guarantees, etc.), implementer details, etc.

3.4.1 Transaction configuration schema

The configuration portion of the transaction schema is shown below and described in the following section.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">

      <xsd:simpleType name="additional-information">
            <xsd:restriction base="xsd:string"/>
      </xsd:simpleType>

      <xsd:complexType name="transaction-type">
            <xsd:sequence>
                  <xsd:element name="transaction-name"
type="xsd:anyURI"/>
                  <xsd:element name="transaction-type-specific-
information" type="additional-information" minOccurs="0"
maxOccurs="unbounded"/>
            </xsd:sequence>
      </xsd:complexType>

      <xsd:element name="transaction-config">
            <xsd:complexType>
                  <xsd:sequence>
                        <xsd:element name="tx-info" type="transaction-
type" minOccurs="0" maxOccurs="1"/>
                        <xsd:element name="transaction-specific-
information" type="additional-information" minOccurs="0"/>
                  </xsd:sequence>
            </xsd:complexType>
      </xsd:element>
</xsd:schema>
```

The `transaction-config` node allows a user to specify the characteristics it requires of a transaction service implementation and conversely for a transaction service implementation to define its own characteristics. The elements within this node are:

- `tx-info`: is an instance of the `transaction-type` and allows the *unique* name of the transaction to be set (transaction-name) via a URI. Additional information about the model implemented may be specified in string form via the `transaction-type-specific-information` element. How this information is specified and later interpreted is up to the transaction implementer to determine. Where this information is published in order for JAXTX implementations to bind to such transaction implementations is beyond the scope of this specification.
- `transaction-specific-information`: additional information about the non-functional aspects of the transaction implementation may be specified in this element. For example, performance characteristics, reliability metrics, etc. Definers can specify which information must be taken into account when binding to the service and which is optional. How this information is specified and later interpreted is up to the transaction implementer to determine. Where this information is published in order for JAXTX implementations to bind to such transaction implementations is beyond the scope of this specification.

For example, if we assume we have a transaction service implementation based on traditional two-phase commit, then the associated XML configuration *might* be:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<jaxtx:transaction-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:jaxtx="http://someorg.org/schemas/JAXTX.xsd">
  <jaxtx:tx-info>
      <jaxtx:transaction-name>

      urn:sun:transaction_models:two_phase_commit_presumed_abort_no_heu
ristics
      </jaxtx:transaction-name>
      <jaxtx:transaction-type-specific-information>
              99.99% availability
      </jaxtx:transaction-type-specific-information>
      <jaxtx:transaction-type-specific-information>
              Implemented by John Doe of FooBar.com
      </jaxtx:transaction-type-specific-information>
  </jaxtx:tx-info>
</jaxtx:transaction-config>
```

Comment: Who maintains these and where?

In this configuration, the URI specifying the protocol indicates it is two-phase commit. However, there are various flavours of two-phase commit and the additional information within the URI specifies attributes which can be used to refine the search for a specific transaction model.[2]

[2] Who maintains the URNs for different transaction protocols?

4. JAXTX components

4.1 Features

The features provided by JAXTX to support the plugging of different transaction implementations within the architecture are described in this section and with reference to Figure 7. More information can be found in the associated `javadoc`.

4.1.1 Client interaction interface

Despite the variety of transaction implementations, the interfaces that most clients (and we include certain service-side interactions) use to interact with them are very similar. These typically involve: starting and ending a transaction (possibly terminating in multiple ways, e.g., commit or roll back), and associating/disassociating a transaction with/from the current thread of control. The `UserTransaction` is responsible for encapsulating this common functionality within a single interface. An implementation of the `UserTransaction` is responsible for mapping the JAXTX requirements to the underlying implementation. The `UserTransaction` then interacts with the coordinator service when instructed to by the client using whatever protocol (e.g., SOAP/XML) is required by the service implementation it encapsulates.

The client component of an application requires some means by which the transaction context is associated with any outgoing service invocation. As described earlier, the format of the context, where it occurs within an XML invocation document etc. will be specified by the appropriate implementation. The `UserTransaction` is required to encapsulate this context manipulation in a transparent manner to the client, such that as far as the client is concerned, the context flows implicitly with all relevant service invocations.

Note, in some transaction models, the context only flows to special transactional services (services which has identified themselves as being able to deal with transactional invocations). How these services are identified is an implementation decision and may be hidden by the `UserTransaction`.

4.1.2 Extended client interfaces

Some transaction implementations may require extensions to the general `UserTransaction` interface. For example, in the OASIS BTP there is scope for the client application to drive the two phase completion protocol explicitly, rather than leaving it to the coordinator, as typically happens in other models. Thus, there is a requirement for more than simply telling the transaction coordinator to end the transaction.

In order to determine the properties of the underlying implementation, the `UserTransaction` configuration and properties are available in the form of a `org.w3c.dom.Document`., which will return the structure defined in Section 3.4.

If the client requires access to protocol specific interfaces that the implementation supports, then it can either cast the underlying implementation to a well known interface or use reflection to obtain access to the extensions.

4.1.3 Coordinator interface

As far as a client is concerned, the coordinator associated with a transaction is rarely seen as it is typically hidden behind interfaces such as `UserTransaction`. However, the service, or some entity acting on its behalf (e.g., a JDBC driver) will eventually be required to enrol participants with the coordinator (transaction) in order that it can ultimately control the fate of work conducted by the service within the scope of the transaction.

As with the client, there are some similarities as to how a service or participant interacts with a transaction coordinator. However, this generality is on a much smaller scale, typically involving only registering and un-registering participants, which will themselves have implementation specific interfaces. Therefore, rather than provide a basic, common interface through which only some interactions with the coordinator can occur, JAXTX does not provide any mandated standard coordinator interface. Implementations are free to provide their own interfaces/classes for services to use in order to access the necessary coordinator functionality.

To assist in the development of new Web Services transactions protocols, in Section 4.3.5, there is a Coordinator WSDL definition as part of the optional JAXTX transaction protocol.

4.1.4 Participant interface

Section 3.1.3 described how a participant interface is typically specific to the transaction protocol in which it will be used. The coordinator-to-participant protocol is transaction model specific and leaves little room for generality: both the coordinator and participants interfaces are dependant upon the transaction model being used. As such, there is no mandated participant interface since its value would be minimal. Therefore, as with coordinator, the participant WSDL interface provided by JAXTX is part of the optional protocol.

4.1.5 Context management

Figure 6**Error! Reference source not found.** shows how both the client and the transactional Web Service will require context manipulation functionality. As illustrated in Figure 8, the context will typically be added to outgoing invocations at the client and

removed at the service (perhaps by the container in which the service resides); what is done with the context at the service-side is not specified by JAXTX, but it will typically include some mechanism for associating the work that is about to be performed by the invocation with the transaction specified in the received context.

Normally the client and service do not need to see the format (implementation) of the context: the implementation is only of concern to those entities that are responsible for manipulating and using it. However, clients and services may need to be able to identify specific transactions or entire contexts without knowing how they are represented within the implementation. For example, some transaction implementations require that only a single participant be registered for each transaction; in order to do this, a service must be able to identify the transaction currently being imported. As such, JAXTX provides interfaces to allow clients and services to manipulate contexts at a suitably higher level.

4.2 Core JAXTX packages

The JAXTX specification defines several new `javax` packages, described briefly in the following sections and in more detail in the associated `javadoc` which accompanies this specification. The `javax.jaxtx` package contains interfaces and classes to which all implementations of JAXTX must conform. The `javax.jaxtx.model.as`, `javax.jaxtx.model.jta` and `javax.jaxtx.model.btp` contain implementation specific enhancements for transaction models conforming to the Activity Service, JTA and BTP specifications respectively and will be described in Section 4.4. An implementation may support any combination of these models or may decide not to support any of them.

4.2.1 javax.jaxtx package

The classes and interfaces of the `javax.jaxtx`, `javax.jaxtx.exceptions`, `javax.jaxtx.completionstatus` and `javax.jaxtx.status` packages are required to abstract away from implementation specific details of the underlying transaction model**Error! Reference source not found.**. All classes provided by implementers will conform to one or more of these interfaces. These are summarized in this section and described in full in the accompanying `javadoc`.

4.2.1.1 CompletionStatus

The `javax.jaxtx.completionstatus.CompletionSatus` interface and associated classes define the basic modes in which a transaction can terminate.

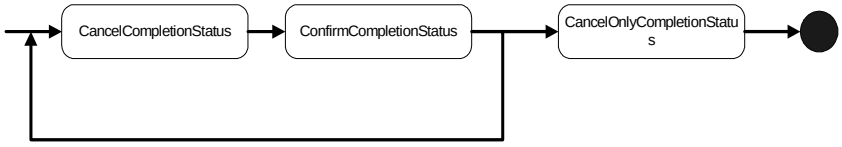

Figure 9: CompletionStatus state diagram.

As shown in Figure 9, at any time in its life a transaction will typically be in one of three *completion states*, such that if it is asked to complete (explicitly by the user or implicitly by virtue of failures, for example) it will do so in a manner prescribed by that state:

- `ConfirmCompletionStatus`: the transaction should complete in a successful state, i.e., all work performed within its scope should be accepted. Note, that if nesting of transactions is supported, this does not necessarily mean that any state changes are made permanent [9]. When in this state, the transaction may be transitioned into any other state. Note, if the transaction cannot complete in this state then it will transition to the `CancelCompletionStatus` state.
- `CancelCompletionStatus`: the transaction should complete in a failure state, i.e., all work performed within its scope should be undone in an implementation specific manner. When in this state, the transaction may be transitioned into any other state. This is the default completion state in which all transactions begin.
- `CancelOnlyCompletionStatus`: the transaction should complete in a failure state, i.e., all work performed within its scope should be undone in an implementation specific manner. Once in this state, the transaction state cannot transition any further.

Note, it is possible that a specific transaction model may have additional states in which is can complete, and these may need to be available to JAXTX users. Therefore, in order to ensure extensibility and that required state transitions do not conflict (e.g., state `foo` for one transaction model may mean something different for another), state types are specified as implementations of the `CompletionStatus` interface and can therefore be augmented by specific implementations

4.2.1.2 Status

The state of a transaction at any point in its life is determined by the `Status` class and its associated implementations, as illustrated in Figure 10.

- `ActiveStatus`: the transaction initially starts in this state. An implementation returns this value prior to the coordinator entering a termination protocol or being marked as `CompletionStausCancelOnly`.
- `MarkedCancelOnlyStatus`: the transaction has been forced into a state whereby its eventual completion must be to cancel, i.e., `CompletionStatusCancelOnly`.

- `CompletingConfirmStatus`: the transaction transitions to this state when told to complete and the completion status is `CompletionStatusConfirm`. The transaction will remain in this state until it has completed in one termination outcome or another. If it cannot complete in the `CompletionStatusConfirm` state then the transaction will transition to the `CompletionStatusCancel` state and complete. For example, in a two-phase completion protocol if the first phase (prepare) succeeds then the coordinator will commit (confirm) all state changes; if it fails then the coordinator will rollback (cancel) all state changes. If there are no failures, the transaction will typically remain in this state until it has received all responses to its termination protocol.
- `CompletedConfirmStatus`: the transaction has terminated successfully in the `CompletionStatusConfirm` completion status. If this status is returned from a call to `UserTransaction.status` then it is likely that errors specific to the protocol exist (e.g., heuristics); otherwise the transaction would have been destroyed and `NoTransactionStatus` returned.
- `CompletingCancelStatus`: the transaction transitions to this state when told to complete and the completion status is `CompletionStatusCancel`. The transaction will remain in this state until it has completed.
- `CompletedCancelStatus`: the transaction has terminated successfully in the `CompletionStatusCancel` completion status.
- `NoTransactionStatus`: there is no transaction associated with the invoking thread.

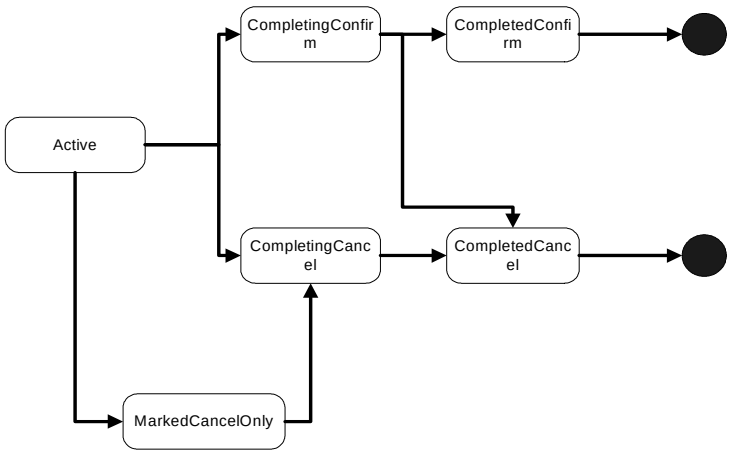

Figure 10: Transaction status transitions.

Note, it is possible that a specific transaction model may have additional states in which the transaction can be and these may need to be available to users. Therefore, in order to

ensure extensibility and that required state transitions do not conflict, state types are implementations of the `Status` interface and can be augmented by specific implementations.

4.2.1.3 Outcome

When a user informs a transaction to terminate, the act of termination will return information to the user about how the transaction completed. This information may be present both for successful and unsuccessful termination of a transaction. For example, in an implementation of the JTA protocol for Web Services, the `ConfirmCompletionStatus` state for a transaction would correspond to committing the transaction and a successful outcome to this would return no information, implicitly informing the user that the transaction committed. However, an unsuccessful termination, for example because of participant failures, could return a specific heuristic exception.

In order to support the different transaction model requirements for return types, `UserTransaction` is required to return an instance of classes derived from the `Outcome` interface. The types of `Outcome` and their internal encodings are required to be specified by the definer/implementer of the transaction implementation.

The type of the transaction `Outcome` is obtainable from the `name` method, whereas the actual state in which it terminated is available from the `completedStatus` method; a user may want the transaction to complete in a `ConfirmCompletionStatus` state but the eventual outcome is determined by the coordinator and its participants, such that failures or inabilities to complete the `ConfirmCompletionStatus` protocol may result in the transaction ending in a `CancelCompletionStatus` state.

Implementation specific data is made available in the form of an XML document (instance of `org.w3c.dom.Document`) through the `data` method. The user may obtain and parse this document to determine additional information on how the transaction terminated.

4.2.1.4 TxContext

Fundamental to the JAXTX architecture is the notion of a *transaction context*. As illustrated in **Error! Reference source not found.**, each thread is associated with a context via `UserTransaction`. This association may be *null*, indicating that the thread has no associated transaction, or it refers to a specific transaction. Contexts may be shared across multiple threads. In the presence of nested transactions a context remembers the stack of transactions started within the environment such that when the nested transaction ends the context of the thread can be restored to that in effect before the nested transaction was started.

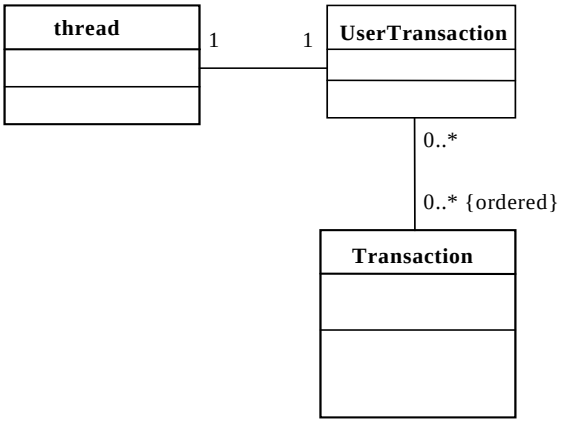

Figure 11: Thread and context relationship.

`TxContext` is used as a representation of a transaction (or stack of transactions) when it is suspended from a running thread. The implementation of the token can be as lightweight as required by the underlying implementation in order that it can uniquely represent all transaction instances. Since this is a client-facing class, it is unlikely that the application user will typically want to see the entire transaction context in order to simply suspend it from the thread.

4.2.1.5 UserTransaction

The `UserTransaction` is the interface that most users (e.g., clients and services) will see. This isolates them from underlying protocol specific aspects of the transaction implementation they are using. If required, an underlying implementation may provide additional methods to users, who may cast or reflect to the underlying implementation if necessary. Importantly, a `UserTransaction` does not represent a specific transaction, but rather is responsible for providing access to an implicit per-thread transaction context. Therefore, all of the `UserTransaction` methods implicitly act on the current thread of control.

A new transaction is begun and associated with the invoking thread by using the `start` method. A transaction that is already associated with the invoking thread will become the *parent* of this newly created transaction, i.e., the new *child* transaction is nested within the parent. If nesting is not supported or available (e.g., the implementation reaches a nesting limit imposed by the transaction model being used), then the `javax.jaxtx.exceptions.NestingNotAllowedException` is thrown. The state of the parent transaction may be such that it is invalid for a new transaction to be created within its scope (e.g., the parent is in the process of terminating) and in this situation the `javax.jaxtx.exceptions.InvalidStateException` is thrown.

The `setTimeout` method modifies a state variable associated with the `UserTransaction` that affects the time-out period in seconds associated with the transaction created by subsequent invocations of the `start` method. If parameter has a non-zero value and the transaction has not begun to terminate before this time period has elapsed, it will be completed in the `CancelCompletionStatus` mode. If the timeout period is invalid or not supported, then the `javax.jaxtx.exceptions.InvalidTimeoutException` is thrown. A value of zero means that no application specified time-out is established. This is the default value associated with all threads.

The `getTimeout` method returns the timeout value currently associated with the `UserTransaction`.

The state in which the current transaction should be completed is set using the `setCompletionStatus` method and passing in either one of the `CompletionStatus` defined types or one specific to the model being used. If the completion state is invalid given the state of the transaction (e.g., it has already begun to complete) then the `javax.jaxtx.exceptions.InvalidStateException` will be thrown. The completion status of the transaction currently associated with the invoking thread can be obtained from the `getCompletionStatus` method. As shown in Figure 9, if there has been no call to the `setCompletionStatus` then the default completion status is `CancelCompletionStatus`. If there is no transaction associated with the current thread then the `javax.jaxtx.exceptions.NoTransactionException` is thrown by both methods.

A transaction can be told to complete using the `end` method, which will also disassociate the thread from the transaction before it returns. The completion status the transaction implementation uses will either have been set by a previous call to `setCompletionStatus`, may be provided as a parameter to end, or will be the default of `CancelCompletionStatus`. If this is a top-level transaction, i.e., it has no parent, then upon completion the thread will have no transaction associated with it. Alternatively, it will be set to the parent of the just completed transaction. If no transaction is associated with the current thread then the `javax.jaxtx.exceptions.NoTransactionException` is thrown.

Some transaction models may restrict the ability for threads to end transactions, e.g., such that only the transaction creator may terminate the transaction (c.f. checked transactions in the JTS). If this is the case, the implementation must throw the `javax.jaxtx.exceptions.NoPermissionException`. If the specified completion state of the transaction is not compatible with the transaction's state (for example, its completion state is `CancelOnlyCompletionStatus` and the user has asked the transaction to terminate with a status of `ConfirmCompletionStatus`), or the state of the transaction does not allow it to be terminated, then the `javax.jaxtx.exceptions.InvalidStateException` will be thrown.

Obviously a transaction may not be able to complete in the state originally requested by the user. When the transaction terminates it may return an `Outcome` object to indicate how it did complete. It is valid for `end` to return a *null* `Outcome`. A transaction that completes and neither returns an `Outcome` nor throws an exception is assumed to have completed in the manner requested.

A thread of control may require periods of non-transactionality in order for it to perform work that is not associated with a specific transaction. In order to do this it is necessary to disassociate the thread from any transactions. The `suspend` method accomplishes this, returning a `TxContext` instance, which is a handle on the transaction context. If the current transaction is nested, then the `TxContext` will implicitly represent the stack of transactions up to and including the root transaction. The thread then becomes associated with no transaction.

The `resume` method can be used to (re-)associate a thread with a transaction(s) via its `TxContext`. Prior to association, the thread is disassociated from any transaction(s) with which it may be currently associated. If the `TxContext` is *null*, then the thread is associated with no transaction. The `javax.jaxtx.exceptionsInvalidTransactionException` is thrown if the transaction that the `TxContext` refers to is invalid in the scope of the invoking thread. It is up to the underlying transaction implementation to determine the validity of a specific context.

The `getTransactionContext` method returns the `TxContext` for the current transaction, or *null* if there is none associated with the invoking thread. Unlike `suspend`, this method does not disassociate the current thread from the transaction(s). This can be used to enable multiple threads to execute within the scope of the same transaction.

The current status of the transaction is returned by the `status` method. If there is no transaction associated with the thread then `NoTransactionStatus` is returned. It is possible that the exact determination of a transaction's status is not possible when `status` is invoked and in which case `UnknownStatus` is returned. This is a transient value and calling `status` again will eventually return a definitive answer.

Every transaction is required to have a unique identity, which can be obtained from the `globalIdentity` method. However, in order to ensure that transaction identifiers are unique across all transaction models, the value returned by `globalIdentity` is required to be prefixed with the package name of the transaction implementation. For example, "*javax.jaxtx.model.jta.1234*" or "*javax.jaxtx.model.as.1234*". If there is no transaction associated with the current thread then *null* will be returned.

In addition to `globalIdentity`, the `transactionName` method can be used to obtain a more user-friendly representation of the transaction. This may return the same as `globalIdentity` or more return other information more suitable for debugging purposes, for example. If there is no transaction associated with the current thread then *null* will be returned.

The parent of the current transaction is returned by `getParent`. If the current transaction has no parent, i.e., is a top-level transaction, then *null* is returned. If there is no transaction associated with the invoking thread then the `javax.jaxtx.exceptions.NoTransactionException` is thrown.

4.2.1.6 UserTransactionFactory

`UserTransactions` are obtained from a `UserTransactionFactory`, which is made available via JNDI lookup of **`java:comp/UserXMLTransactionFactory`**. The `UserTransaction` corresponding to the specific transaction type is returned from the factory's `getTransactionType` method, as defined in the associated XML document configuration. If no such transaction type is provided by the factory, then the `javax.jaxtx.exceptions.NoSuchServiceException` is thrown. Alternatively, all of the transaction types supported by the factory may be returned through the `getAllTransactionTypes` method. The factory may restrict the accessibility for clients to obtain all transaction types through this method and throw the `javax.jaxtx.exceptions.InvalidSecurityOptionException`.

4.3 Transaction framework protocol

Before describing the optional client and service-side interfaces, in this section we shall examine the optional XML protocol that JAXTX provides (*JAXTXP*), which is primarily intended for new transaction service implementations. Note, an implementation of JAXTX is free to choose which (if any) components of this protocol it wishes to support. Several of the extended transaction implementations to be described later in the specification use this protocol.

Section 2.2 described how a single (extended) transaction protocol is unlikely to be suitable for all problem domains: *one-size does not fit all*. Each extended transaction protocol is designed to target a specific problem area, e.g., relaxing atomicity or allowing controlled un-serialisable behaviour. Therefore, rather than specify an XML-based transaction protocol that is aimed at solving a single problem, the optional JAXTXP is deliberately meant to allow arbitrary transaction implementations to utilise it; in essence this is a framework for supporting many different transaction protocols. The goals of this protocol are therefore to:

- Provide a basic definition of a core transaction service consisting of a Coordinator Service. The protocol that this service uses will be hidden behind the interface (WSDL).
- Define the required infrastructure support, such as event mechanisms, etc.
- Define the roles and responsibilities of subcomponents.

The main components involved in using and defining the protocol are:

1) A *Coordination Service*: defines the behaviour of a specific transaction model and provides a processing pattern that is used for outcome determination. For example, a two-phase ACID transaction service is one implementation whose processing sequence includes *prepare*, *commit* and *rollback.* Other implementation examples include Sagas, Collaborations and Real-Time transactions.
2) A *Participant*: The operation or operations that are performed as part of coordination sequence processing when triggered by the Coordination Service. During the lifetime of a transaction, a coordinator may send many different XML messages to a participant, which may perform specific units of work depending upon how it interprets these XML messages. Note, an implementation of a participant could be a coordinator, forming a distributed tree structure, with root and sub coordinators.
3) The *context*: this contains information necessary to perform coordination as well as information specific to the Coordination Service.

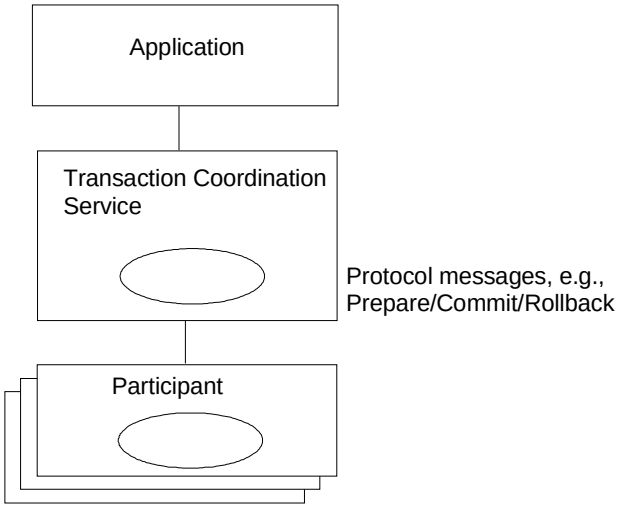

Figure 12: Coordination Service and action relationship.

4.3.1 Qualifiers

In order to be flexible and cope with a wide range of extended transaction models, it is necessary to support controlled extensibility of the JAXTXP XML message-set. The Qualifier concept is a means whereby implementation specific caveats or data may be applied to protocol messages. The format of the Qualifier is shown below:

```
<xsd:element name="qualifier">
      <xsd:annotation>
            <xsd:documentation>
                  qualifier represents a component of the sequence of
qualifiers.
```

```
            </xsd:documentation>
      </xsd:annotation>
      <xsd:complexType>
            <xsd:attribute name="name" use="required">
                  <xsd:annotation>
                        <xsd:documentation>
                              Attribute describes name element of
qualifier name-value pair.
                        </xsd:documentation>
                  </xsd:annotation>
            </xsd:attribute>
            <xsd:attribute name="value" use="required">
                  <xsd:annotation>
                        <xsd:documentation>
                              Attribute describes value element of
qualifier name-value pair.
                        </xsd:documentation>
                  </xsd:annotation>
            </xsd:attribute>
            <xsd:attribute name=”MustUnderstand” type=”xsd:boolean”
use=”optional”/>
      </xsd:complexType>
</xsd:element>

<xsd:element name="qualifiers">
      <xsd:annotation>
            <xsd:documentation>
                  qualifiers represents values for qualifiers name-
value pairs.
            </xsd:documentation>
      </xsd:annotation>
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element ref="qualifier"
maxOccurs="unbounded"/>
            </xsd:sequence>
      </xsd:complexType>
</xsd:element>
```

A *qualifier* is a name/value tuple-space, with *qualifiers* being an arbitrary sized list of these tuple-spaces. Each specific transaction implementation must define which qualifiers it accepts and enforce rules associated with them. A qualifier which has the associated MustUnderstand attribute set to true must be understood by the importing domain. If it is false or omitted, then the importing domain may ignore it.

4.3.2 Protocol messages

In Section 2 we saw the relationship between the transaction coordinator and participants. This relationship exists irrespective of the type of transaction protocol that is being used: at some point (or points) during the lifetime of the transaction, the coordinator will be required to communicate with the participants. This is a bi-directional

relationship, such that a service or participant may also be required to communicate with the coordinator.

For example, the termination of one activity may initiate the start/restart of other activities in a workflow-like environment. Messages can be used to infer a flow of control during the execution of an application. The information encoded within a message will depend upon the implementation of the extended transaction model.

A participant will use the message in a manner specific to the transaction implementation and return a result of it having done so. For example, upon receipt of a specific message, a participant could start another transaction running (e.g., a compensation transaction); another participant could commit any modifications to a database when it receives one type of message, or undo them if it receives another type.

JAXTXP does not require that communication between coordinator and participant be synchronous or based on any request/response pattern. Message exchanges may occur in an RPC approach if deemed necessary by the transaction implementation, or it may use more asynchronous, message-oriented approaches[3]. As a result, a participant may send responses to coordinator messages that have not yet been (and potentially may never be) produced, e.g., a two-phase participant may be able to autonomously prepare itself and signal this fact to the coordinator, even though the coordinator may never be asked to commit.

In addition, as we shall see in Section 4.3.4, unlike traditional ACID two-phase protocols, where a user's (direct or indirect) interaction with the coordinator signals the end of the transaction, this is not an assumption JAXTXP makes: coordinator interactions may occur at many times during the lifetime of an extended transaction; termination of the transaction upon completing a coordination interaction is an implementation decision.

```
<xsd:element name="protocol_message" substitutionGroup="message">
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element name="specific_data" type="xsd:anyType"
minOccurs="0" maxOccurs="unbounded"/>
                  <xsd:elememt ref=”qualifiers” minOccurs=”0”/>
            </xsd:sequence>
            <xsd:attribute name="identification" type="xsd:anyURI"
use="required"/>
            <xsd:attribute name="related_to" type="xsd:anyURI"
use="optional"/>
      </xsd:complexType>
</xsd:element>
```

[3] In order to allow the use of JAXM or JAXRPC do we need to say or do more?

- *identification*: this is the unique identification of this message and will be interpreted by the receiver in a protocol specific manner. For example, the two-phase commit transaction identified by the "urn:twophase" may define protocol messages "urn:twophase:prepare", "urn:twophase:commit" and "urn:twophase:rollback", which would be interpreted by receivers as either preparing, committing or rolling back the transaction accordingly. Their responses may be "urn:twophase:readonly", "urn:twophase:ok" and "urn:twophase:fail"[4].
- *specific_data*: additional protocol-specific information may be present in this field.
- *qualifiers*: any qualifiers necessary for the interpretation of this message are specified in this field.
- *related_to*: JAXTXP does not require that coordinator-to-participant interactions be synchronous and driven by the coordinator. As such, it is entirely possible that a participant may autonomously decide to send the coordinator a message to indicate its current state in the protocol. For example, a two-phase aware participant may be able to spontaneously prepare and send the result to the coordinator, on the assumption that the coordinator will eventually send it a prepare message. If the participant does, however, it is important that it also specify which message it assumed the coordinator would generate in order for it to send that response: if the coordinator does not eventually send that message then obviously the response from the participant is in error. As such, the *related_to* field is required to be present for all spontaneously generated participant responses and indicates the relevant coordinator protocol message. It is optional in all other message interactions.

4.3.3 More on context

The context information is propagated using SOAP messages, and is correlated to the application messages using unique identifiers in the SOAP header. The context may be propagated as part of normal message interchange within an application (e.g., as an additional part of the SOAP header) or may be sent as an explicit "out-of-band" context message, e.g., where the content of the message contains the context and a handle to link it to specific application messages which will be sent separately. Context propagation may also occur using different protocols than those used by the application. Therefore, JAXTXP does not assume a specific means by which contexts are propagated, leaving this up to the specific coordination service implementation.

[4] I think we need more defined structure for the URIs.

```
<xsd:element name="contextelement">
      <xsd:annotation>
            <xsd:documentation>
                  contextelement represents the information for an
entry in the hierarchy.
            </xsd:documentation>
      </xsd:annotation>
      <xsd:complexType>
            <xsd:attribute name="transaction-name" type="xsd:anyURI"
use="optional"/>
            <xsd:attribute name="coordinator" type="Coordinator"
use="required"/>
            <xsd:attribute name="ctxId" type="xsd:anyURI"
use="required"/>
            <xsd:attribute name="timeout" type="xsd:integer"
use="optional"/>
            <xsd:attribute name="implementation_specific_data"
type="xsd:anyType" use="optional"/>
            <xsd:sequence>
                  <xsd:element ref="qualifiers" minOccurs="0"/>
            </xsd:sequence>
      </xsd:complexType>
</xsd:element>

<xsd:element name="context" substitutionGroup="message">
      <xsd:annotation>
            <xsd:documentation>
                  context represents the context hierarchy.
            </xsd:documentation>
      </xsd:annotation>
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element ref="contextelement"/>
            </xsd:sequence>
            <xsd:attribute name="invocation_specific_data"
type="xsd:anyType" use="optional"/>
      </xsd:complexType>
</xsd:element>
```

In order to support nesting of transactions, the context is an arbitrary length sequence of `contextelement` data structures, where the first element in the list is the root transaction and the last is the current transaction. The data encoded within this structure is:

- `transaction-name`: an optional unique name for this transaction, this field may be used to identify the type of transaction being imported.
- `coordinator`: a reference to the coordinator responsible for this specific transaction. See Section 4.3.4 for more information about the coordinator.
- `ctxId`: a unique identifier for this transaction.

- `timeout`: some transaction implementations allow a timeout value (typically measured in seconds) to be associated with a transaction, such that, if the transaction has not been completed when this time elapses, it will be automatically terminated.
- `implementation_specific_data`: any additional information which may be required by the specific transaction implementation being used. How this is identified and interpreted is outside the scope of this specification.
- `qualifiers`: additional qualifiers that may be required to further interpret the information contained within the context.

4.3.4 Participant

To allow activities to be independent of the other activities, and also to allow the insertion of arbitrary coordination and control points, protocol messages are sent to *Participants*. A participant can then use the message in an application specific manner and return an indication of it having done so. Each participant supports the following methods:

- `process_message`: a protocol-specific message is sent to the participant and it should deal with it and possibly return a response.
- `identity`: the unique identity of this participant (i.e., a URI).

4.3.5 Coordinator

To drive the message exchanges a *coordinator* is associated with each transaction. Activities that require to be informed when coordinator sends a specific message can register a participant with that coordinator. When a message is sent by the coordinator (e.g., at termination time), it may result in a response being returned by the recipient (participant). Alternatively, depending upon the protocol in use, no such message may be forthcoming. As such, the coordinator-participant interaction is not required to be synchronous and follows a more asynchronous, message-oriented pattern. Obviously RPC-based (synchronous) invocations, may be layered on top of this.

In a traditional ACID transaction system, the two-phase protocol is only executed by the coordinator when the transaction terminates. Once the second phase has completed successfully, the transaction and associated coordinator are terminated. However, there are several extended transaction models that allow coordination to occur at times during the lifetime of the transaction (e.g., Split Transactions [10]). Therefore, the JAXTX coordinator can be driven at arbitrary points.

The implementation of the coordinator will obviously depend upon the type of extended transaction model being used. For example, if a Sagas type model is in use then a compensation message may be required to be sent to participants if a failure has

happened, whereas a coordinator for a strict transactional model may require to send a message informing participants to rollback.

The accompanying WSDL for the coordinator shows it to have the following methods:

- `complete_transaction`: this instructs the coordinator to terminate the transaction in the state specified.
- `coordinate`: the coordinator is required to execute the coordination protocol and return a response. Unless it does not make sense for the transaction to continue to execute, this operation does not terminate the transaction.
- `send_outcome`: a participant sends an outcome to the coordinator. Typically this is as a result of having received a coordination protocol message, but may occur autonomously, as described in Section 4.3.2.
- `get_outcome`: returns the outcome of the last execution of the coordination protocol.
- `add_participant`: registers the specified participant with the transaction.
- `remove_participant`: if allowed by both the transaction implementation and the state of the transaction, the specified participant is removed from the transaction.
- `set_response`: when a protocol message is sent to a participant, the response does not have to occur immediately. In addition, participants may autonomously send responses to messages that have not yet been generated. This method allows a participant to send a response and indicate the protocol message which (would) caused it.
- `get_parent_coordinator`: returns a reference to this transaction's parent. If the transaction is a root transaction, then it has no parent and a null reference will be returned.
- `get_transaction_id`: returns the unique identification for this transaction (URI).
- `get_status`: returns the current `javax.jaxtx.status.Status` for this transaction.
- `get_parent_status`: returns the current `javax.jaxtx.status.Status` for the target transaction's parent, or `StatusNoTransaction` is the current transaction is not nested.
- `is_same_transaction`: returns *true* if the transaction identified is the same as the target, *false* otherwise.

4.4 Model bindings

In the following sections we shall examine how specific transaction models may be mapped into a JAXTX environment. These models may define new client-side interfaces and optional service-side interfaces as well as make use of the JAXTXP.

4.4.1 The JTA

In this section we shall consider how to use the JTA transaction *model* within a Web Services environment. This involves specifying optional interfaces over and above those defined by core JAXTX. Although this implementation uses and enhances JAXTXP, it is not necessary for all such Web Services-based implementations of the JTA to do so[5].

Note, the interfaces described in this section occur within the `javax.jaxtx.model.jta.opt` package.

4.4.1.1 XML configuration

The configuration information required to be associated with an implementation of the JTA interfaces defined in this section is shown below:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<jaxtx:transaction-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:jaxtx="JAXTX.xsd">
<jaxtx:tx-info>
    <jaxtx:transaction-name>
      urn:com:sun:jaxtx:transaction_models:jta:1.0.1
    </jaxtx:transaction-name>
</jaxtx:tx-info>
</jaxtx:transaction-config>
```

4.4.1.2 Client-side interfaces

The standard client-side interface of `javax.jaxtx.UserTransaction` is sufficient to demarcate JTA transaction boundaries, with the following mappings:

JTA	JAXTX JTA equivalent
UserTransaction/TransactionManager.begin	UserTransaction.start
javax.transaction.NotSupportedException	javax.jaxtx.exceptions.NestingNotAllowedException
UserTransaction/TransactionManager/Transacti	CompletionStatusConfirm and

[5]Obviously the standard JTA interfaces of `UserTransaction`, `TransactionManager` etc. may be provided to users, with their internal implementations using the JAXTX interfaces and associated implementations to be described. This could then restrict clients to only working with this specific protocol, but that may be correct within that application domain.

on.commit	UserTransaction.end. Successful commitment returns an Outcome with the `name` *"javax.jaxtx.model.jta.commit"* and `completedStatus` of `CompletionStatusConfirm`.
javax.transaction.RollbackException	UserTransaction.end returns an Outcome with the `name` *"javax.jaxtx.model.jta.rolledback"* and `completedStatus` of `CompletionStatusCancel`
javax.transaction.HeuristicMixedException	javax.jaxtx.exceptions.HeuristicMixedException
javax.transaction.HeuristicHazardException	javax.jaxtx.exceptions.HeuristicHazardException
UserTransaction/TransactionManager/Transaction.rollback	CompletionStatusCancel or CompletionStatusCancelOnly and UserTransaction.end. Successful rollback returns an Outcome with the `name` *"javax.jaxtx.model.jta.rollback"* and `completedStatus` of `CompletionStatusCancel`.
UserTransaction/TransactionManager/Transaction.setRollbackOnly	UserTransaction.setCompletionStatus with CompletionStatusCancelOnly
javax.transaction.Status.STATUS_PREPARED	javax.jaxtx.model.jta.PreparedStatus

4.4.1.3 Service-side interfaces

The JTA defines two types of participant:

- `XAResource`: this is a Java mapping of the industry standard XA resource manager interface [11].
- `Synchronization`: the transaction manager provides a synchronization protocol that allows the interested party to be notified before and after the transaction completes.

As such, the Web Services equivalent also makes this distinction between types of transaction participant in the `javax.jaxtx.model.jta.opt.XAResource` and `javax.jaxtx.model.jta.opt.Synchronization` interfaces respectively.

Services can enrol participants (`XAResource` or `Synchronization` instances) through the `TransactionManager` interface, which returns a reference to the current transaction (instance of `javax.jaxtx.model.jta.opt.Transaction`) via the `getTransaction` method. The `Transaction` interface provides a means whereby participants of either type may be enrolled with a specific transaction; only `XAResources` may be removed from the transaction. Unlike `UserTransaction`, a `Transaction` does represent a specific transaction instance.

4.4.1.4 JAXTXP specifics

If JAXTXP is used to implement the JTA Web Services model, then the following specifics are required:

The `Synchronization` and `XAResource` interfaces are encapsulated by `Participants`[6].

The protocol message set is:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<jaxtx:jta_message_set
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xmlns:jaxtx="JAXTX.xsd">
    <jaxtx:protocol_message>
        <jaxtx:identification>
urn:com:sun:jaxtx:transaction_models:jta:1.0.1:before_completion
      </jaxtx:identification>
        <jaxtx:specific_data>
          <xsd:element name="tranID" type="xsd:anyURI" use="required"/>
      </jaxtx:specific_data>
    </jaxtx:protocol_message>

    <jaxtx:protocol_message>
        <jaxtx:identification>
urn:com:sun:jaxtx:transaction_models:jta:1.0.1:after_completion
      </jaxtx:identification>
        <jaxtx:specific_data>
          <xsd:element name="tranID" type="xsd:anyURI" use="required"/>
      </jaxtx:specific_data>
        <jaxtx:specific_data>
          <xsd:element name="status" type="jaxtx:status"
use="required"/>
      </jaxtx:specific_data>
    </jaxtx:protocol_message>

    <jaxtx:protocol_message>
        <jaxtx:identification>
          urn:com:sun:jaxtx:transaction_models:jta:1.0.1:commit
      </jaxtx:identification>
        <jaxtx:specific_data>
```

[6] Does this actually gain us anything? JAXTXP can be used but why not just keep with the “native” interfaces? Different degrees of optionality?

```
          <xsd:element name="tranID" type="xsd:anyURI" use="required"/>
      </jaxtx:specific_data>
        <jaxtx:specific_data>
          <xsd:element name="onePhase" type="xsd:boolean"
use="required"/>
      </jaxtx:specific_data>
    </jaxtx:protocol_message>

    <jaxtx:protocol_message>
        <jaxtx:identification>
          urn:com:sun:jaxtx:transaction_models:jta:1.0.1:start
      </jaxtx:identification>
        <jaxtx:specific_data>
          <xsd:element name="tranID" type="xsd:anyURI" use="required"/>
      </jaxtx:specific_data>
        <jaxtx:specific_data>
          <xsd:element name="flags" type="xsd:integer" use="required"/>
      </jaxtx:specific_data>
    </jaxtx:protocol_message>

    <jaxtx:protocol_message>
        <jaxtx:identification>
          urn:com:sun:jaxtx:transaction_models:jta:1.0.1:end
      </jaxtx:identification>
        <jaxtx:specific_data>
          <xsd:element name="tranID" type="xsd:anyURI" use="required"/>
      </jaxtx:specific_data>
        <jaxtx:specific_data>
          <xsd:element name="flags" type="xsd:integer" use="required"/>
      </jaxtx:specific_data>
    </jaxtx:protocol_message>

    <jaxtx:protocol_message>
        <jaxtx:identification>
          urn:com:sun:jaxtx:transaction_models:jta:1.0.1:forget
      </jaxtx:identification>
        <jaxtx:specific_data>
          <xsd:element name="tranID" type="xsd:anyURI" use="required"/>
      </jaxtx:specific_data>
    </jaxtx:protocol_message>

    <jaxtx:protocol_message>
        <jaxtx:identification>
          urn:com:sun:jaxtx:transaction_models:jta:1.0.1:isSameRM
      </jaxtx:identification>
        <jaxtx:specific_data>
          <xsd:element name="xares" type="xsd:anyURI" use="required"/>
      </jaxtx:specific_data>
    </jaxtx:protocol_message>

    <jaxtx:protocol_message>
        <jaxtx:identification>
urn:com:sun:jaxtx:transaction_models:jta:1.0.1:isSameRM_response
      </jaxtx:identification>
```

```
        <jaxtx:specific_data>
          <xsd:element name="isSameRM" type="xsd:boolean"
use="required"/>
      </jaxtx:specific_data>
    </jaxtx:protocol_message>

    <jaxtx:protocol_message>
        <jaxtx:identification>
          urn:com:sun:jaxtx:transaction_models:jta:1.0.1:prepare
      </jaxtx:identification>
        <jaxtx:specific_data>
          <xsd:element name="tranID" type="xsd:anyURI" use="required"/>
      </jaxtx:specific_data>
    </jaxtx:protocol_message>

    <jaxtx:protocol_message>
        <jaxtx:identification>
urn:com:sun:jaxtx:transaction_models:jta:1.0.1:prepare_response
      </jaxtx:identification>
        <jaxtx:specific_data>
          <xsd:element name="prepare" type="xsd:integer"
use="required"/>
      </jaxtx:specific_data>
    </jaxtx:protocol_message>

    <jaxtx:protocol_message>
        <jaxtx:identification>
          urn:com:sun:jaxtx:transaction_models:jta:1.0.1:recover
      </jaxtx:identification>
        <jaxtx:specific_data>
          <xsd:element name="flag" type="xsd:integer" use="required"/>
      </jaxtx:specific_data>
    </jaxtx:protocol_message>

    <jaxtx:protocol_message>
        <jaxtx:identification>
urn:com:sun:jaxtx:transaction_models:jta:1.0.1:recover_response
      </jaxtx:identification>
        <jaxtx:specific_data>
          <xsd:sequence>
              <xsd:element name="tranID" type="xsd:anyURI"
use="required"/>
          </xsd:sequence>
      </jaxtx:specific_data>
    </jaxtx:protocol_message>

    <jaxtx:protocol_message>
        <jaxtx:identification>
          urn:com:sun:jaxtx:transaction_models:jta:1.0.1:rollback
      </jaxtx:identification>
        <jaxtx:specific_data>
          <xsd:element name="tranID" type="xsd:anyURI" use="required"/>
      </jaxtx:specific_data>
    </jaxtx:protocol_message>
```

```
    <jaxtx:protocol_message>
        <jaxtx:identification>
urn:com:sun:jaxtx:transaction_models:jta:1.0.1:getTransactionTimeout
      </jaxtx:identification>
    </jaxtx:protocol_message>

    <jaxtx:protocol_message>
        <jaxtx:identification>
urn:com:sun:jaxtx:transaction_models:jta:1.0.1:getTransactionTimeout_re
sponse
      </jaxtx:identification>
        <jaxtx:specific_data>
          <xsd:element name="timeout" type="xsd:integer"
use="required"/>
      </jaxtx:specific_data>
    </jaxtx:protocol_message>

    <jaxtx:protocol_message>
        <jaxtx:identification>
urn:com:sun:jaxtx:transaction_models:jta:1.0.1:setTransactionTimeout
      </jaxtx:identification>
        <jaxtx:specific_data>
          <xsd:element name="seconds" type="xsd:integer"
use="required"/>
      </jaxtx:specific_data>
    </jaxtx:protocol_message>

    <jaxtx:protocol_message>
        <jaxtx:identification>
urn:com:sun:jaxtx:transaction_models:jta:1.0.1:setTransactionTimeout_re
sponse
      </jaxtx:identification>
        <jaxtx:specific_data>
          <xsd:element name="timeout_set" type="xsd:boolean"
use="required"/>
      </jaxtx:specific_data>
    </jaxtx:protocol_message>
</jaxtx:jta_message_set>
```

4.4.2 The Business Transactions Protocol

In this section we shall examine the extensions to the basic JAXTX interfaces to provide support for the OASIS Business Transactions Protocol (BTP).

An actor within the coordinating entity's system plays the role of *Superior* and an actor within the system of the party plays the role of an *Inferior*. Each Inferior has only one Superior. However, a single Superior may have multiple Inferiors within each or multiple parties. A tree of such relationships may be wide, deep, or both as shown in Figure 13.

An Inferior is associated with some set of application activities that create effects within the party. Usually, this will be a result of some operation invocations (on a "service

application element") from elsewhere (an "initiating application element"). The Inferior is responsible for reporting that it is "prepared" for the outcome to the Superior whether or not the associated operations' provisional effect can be confirmed or cancelled.

A Superior receives reports from its Inferiors as to whether they are "prepared" to give an outcome. It gathers these reports in order to determine which Inferiors should be cancelled and which confirmed. The Superior does this either by itself or with the cooperation of the application element responsible for its creation and control, depending upon whether the transaction is an atom or a cohesion as we shall see later.

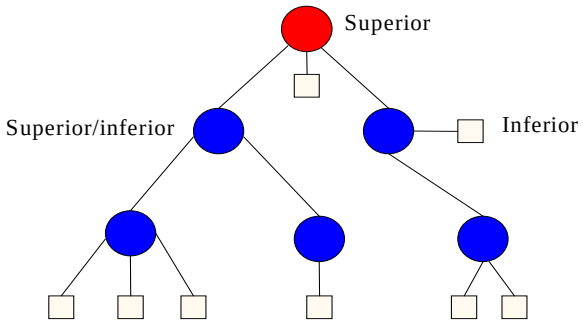

Figure 13: Transaction Relationships

4.4.2.1 XML configuration

The configuration information required to be associated with an implementation of the BTP interfaces defined in this section is shown below:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<jaxtx:transaction-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:jaxtx="JAXTX.xsd">
<jaxtx:tx-info>
    <jaxtx:transaction-name>
      urn:com:sun:jaxtx:transaction_models:btp:1.0
    </jaxtx:transaction-name>
</jaxtx:tx-info>
</jaxtx:transaction-config>
```

4.4.2.2 Client-side interfaces

The classes and interfaces described in this section are located within the `javax.jaxtx.model.btp` package. The OASIS BTP defines two types of extended transactions:

- *Atom*: an atom is the typical way in which "transactional" work performed on Web services is scoped. The outcome of an atom is guaranteed to be atomic, such that all enlisted participants (acting on behalf of their associated Web services) will see the same outcome, which will either be to accept (confirm) the work or reject (cancel) it.
- *Cohesion*: this type of transaction was introduced in order to relax atomicity and allow for the selection of work to be confirmed or cancelled based on higher level business rules. Atoms are the typical participants within a cohesion but, unlike an atom, a cohesion may give different outcomes to its participants such that some of them may confirm whilst the remainder cancel. In essence, the two-phase protocol for a cohesion is parameterised to allow a user to specify precisely which participants to prepare and which to cancel. The strategy underpinning cohesions is that they better model long-running business activities, where services enrol in atoms that represent specific units of work and as the business activity progresses, it may encounter conditions that allow it to cancel or prepare these units, with the caveat that it may be many hours or days before the cohesion arrives at its *confirm-set*: the set of participants that it requires to confirm in order for it to successfully terminate the business activity. Once the confirm-set has been determined, the cohesion collapses down to being an atom: all members of the confirm-set will see the same outcome.

In a traditional transaction system, the application has very few verbs with which to control transactions. Typically these are "begin", "commit" and "rollback". When an application asks for a transaction to commit, the coordinator will execute the entire two-phase protocol before returning an outcome (what BTP terms a *closed-top commit protocol*). The elapse time between the execution of the first phase and the second phase is typically milliseconds to seconds.

However, the two-phase algorithm does not impose any restrictions on the time between executing the first and second phases. Clearly the longer the period between first and second phases, the greater the chance for failures to occur and the longer resources remain locked. BTP allows the time between the two phases to be set by the application by expanding the range of verbs available to include explicit control over both phases, i.e., "prepare", "confirm" and "cancel"; what BTP terms an *open-top completion protocol*. The application has complete control over when transactions prepare, and using use whatever business logic is required later determine which transactions to confirm or cancel.

Therefore, although both atoms and cohesions can be terminated in a traditional "closed-top" manner, it is more typical for the two-phase completion protocol to be driven explicitly. When using the basic JAXTX client interfaces (i.e., closed-top termination scheme), the mappings are as follows:

BTP	JAXTX BTP equivalent
"One-shot" cancel termination.	CompletionStatusCancel or CompletionStatusCancelOnly and UserTransaction.end. Successful cancellation returns an Outcome with the `name` *"javax.jaxtx.model.btp.cancel"* and `completedStatus` of `CompletionStatusCancel`.
"One-shot" confirm termination.	CompletionStatusConfirm and UserTransaction.end. Successful confirmation returns an Outcome with the `name` *"javax.jaxtx.model.btp.confirm"* and `completedStatus` of `CompletionStatusCancel`. Unsuccessful has `CompletionStatusCancel` and *"javax.jaxtx.model.btp.cancelled*".
Hazard	javax.jaxtx.model.btp.HazardException
Mixed	javax.jaxtx.model.btp.MixedException
Resigned status	javax.jaxtx.model.btp.ResignedStatus
Resigning status	javax.jaxtx.model.btp.ResigningStatus
Prepared status	javax.jaxtx.model.btp.PreparedStatus
Preparing status	javax.jaxtx.model.btp.PreparingStatus

As such, the `javax.jaxtx.model.btp.Atom` and `javax.jaxtx.model.btp.Cohesion` exist to extend the default `UserTransaction` interface.

Instances of the `javax.jaxtx.model.btp.Vote` interface are returned from the Atom when it is instructed to prepare. An atom (and implicitly its participants) can return one of the following votes as a result:

- `VoteCancel`: the inferior votes that it has cancelled. The transaction service may inform the inferior of the final decision (hopefully to cancel as well), but it need not.
- `VoteConfirm`: the inferior votes to confirm. It will typically not have confirmed at this stage but will wait for the transaction outcome. Failure to do so may result in heuristics (contradictions).
- `VoteResign`: the inferior votes that all of its inferiors have resigned from the transaction. The transaction (atom) is terminated.

When a `javax.jaxtx.model.btp.Cohesion` is instructed to prepare, cancel or confirm its enlisted participants (inferiors), it may return a status value for each of them. It does this through the `javax.jaxtx.model.btp.StatusItem` interface. The status is a value from the `javax.jaxtx.model.btp.TwoPhaseOutcome` enumeration.

4.4.2.3 Service-side interfaces

The classes and interfaces described in this section are for the optional service-side interactions and are located in the `javax.jaxtx.model.btp.opt` package.

A BTP superior (e.g., instance on which participants may enrol) is represented by an instance of the `Superior` interface. Each participant within a BTP transaction is represented by the `Inferior` interface. This has methods for preparing, confirming and cancelling the participant as well as obtaining its current status. As shown previously, a superior may be an inferior in some distributed hierarchy. The root superior is known as the decider and is represented by the `Decider` interface.

Services can enrol participants through the `TransactionManager` interface, which returns a reference to the current transaction (`Superior`) via the `getTransaction` method. The `Transaction` interface provides a means whereby `Inferiors` may be enrolled with a specific transaction. Unlike `UserTransaction`, a `Transaction` does represent a specific transaction.

An `Inferior` is enrolled with a transaction using the `enrol` method; if the transaction is not in a state where enrolment is possible (e.g., it is terminating) then the `javax.jaxtx.exceptions.WrongStateException` will be thrown. If the `Inferior` has already been enrolled with this transaction then the `javax.jaxtx.model.btp.DuplicateInferiorException` is thrown and the `Inferior` will not be enrolled with the transaction.

BTP allows for an enrolled `Inferior` to leave a transaction. This may be accomplished through the `resign` method. If the coordinator is not in a state where leaving is valid then the `javax.jaxtx.exceptions.WrongStateException` is thrown. If the `Inferior` is invalid within the context of the transaction (e.g., it has not been enrolled) then the `javax.jaxtx.model.btp.opt.InvalidInferiorException` will be thrown.

4.4.2.4 JAXTXP specifics

At present there is no requirement for a mapping of BTP to JAXTXP.

4.4.3 The J2EE Activity Service

Comment: Do we want to add a mapping to WS-T.

[TBD based on the work on JSR95.] and using the JAXTXP.[7]

[7] The J2EE version of the CORBA Activity Service has not bee finalised, though it is close. Rather than base the JAXTX mapping on the CORBA version, which is subtly different, we may need to wait.

JAXTXP is heavily influenced by this work, so there may be additional feedback into other parts of the specification.

5. References

[1] Additional Structuring Mechanisms for the OTS Specification - OMG document orbos/2001-11-08 (http://www.omg.org/cgi-bin/doc?orbos/2001-11-08).

[2] “Business Transaction Protocol” Furniss, P. (ed). See: http://www.oasis-open.org/committees/business-transactions/#documents

[3] C. T. Davies, “Data processing spheres of control”, IBM Systems Journal, Vol. 17, No. 2, 1978, pp. 179-198.

[4] A. K. Elmagarmid (ed), “Transaction models for advanced database applications”, Morgan Kaufmann, 1992.

[5] H. Garcia-Molina and K. Salem, “Sagas”, Proceedings of the ACM SIGMOD International Conference on the Management of Data, 1987.

[6] S. K. Shrivastava and S. M. Wheater, “Implementing fault-tolerant distributed applications using objects and multi-coloured actions”, Proc. of 10th Intl. Conf. on Distributed Computing Systems, ICDCS-10, Paris, June 1990, pp. 203-210.

[7] G.Weikum, H.J.Schek, “Concepts and Applications of Multilevel Transactions and Open Nested Transactions”, in Database Transaction Models for Advanced Applications, ed. A.K. Elmagarmid, Morgan Kaufmann, 1992.

[8] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Gunthor and C. Mohan, “Advanced transaction models in workflow contexts”, Proc. of 12th Intl. Conf. on Data Engineering, New Orleans, March 1996.

[9] J. E. B. Moss, "Nested Transactions: an approach to reliable distributed computing”, Ph.D. Thesis 260, MIT, Cambridge, MA, April 1981.

[10] C. Pu, G. E. Kaiser and N. Hutchinson, “Split-Transactions for Open-Ended Activities”, Proc. of VLDB Conference, Los Angeles, CA, September 1998, pp. 26-37.

[11] X/Open CAE Specification – Distributed Transaction Processing: The XA Specification, X/Open Document No. XO/CAE/91/300 or ISBN 1 872630 24 3.