Java™ 2 Enterprise Edition

XML API for Java™ Transactions

JSR 156

Specification Lead:
Mark Little
Hewlett-Packard

Technical comments:

Version 0.1
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running applications will require support for fault-tolerance, because machines may fail or services may be moved or withdrawn.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 [ref] 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 [ref] 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.

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 [ref]. The terms *transaction* or *extended transaction*, shall refer to models that give different 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 behind JAXTX, and defines the function 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

Ian Robinson, ian_robinson@uk.ibm.com

International Business Machines,
IBM Hursley Lab,
Hursley,
Winchester,
Hant, SO21 2JN,
UK

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

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

Pyounguk Cho, pyounguk.cho@iona.com

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

Kavantzas Nickolaos, nkavantz@us.oracle.com

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

Peter Walker, Peter.Walker@sun.com

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

Rahul Bhargava, rahul_technical@yahoo.com

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

Peter Furniss, peter.furniss@choreology.com

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

Anne Thomas, atm@systinet.com

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

Alan Davies, adavies@seebeyond.com

SeeBeyond Technology Corporation,
404 E Huntington Drive,
Monrovia,
CA 91016,
USA

Bill Pope, zpope@pobox.com

PO Box 6698,
Portsmouth NH 03801,
USA

Michael Abbot, mike@codemetamorphosis.com

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

1.6 Acknowledgements

2. Overview

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.

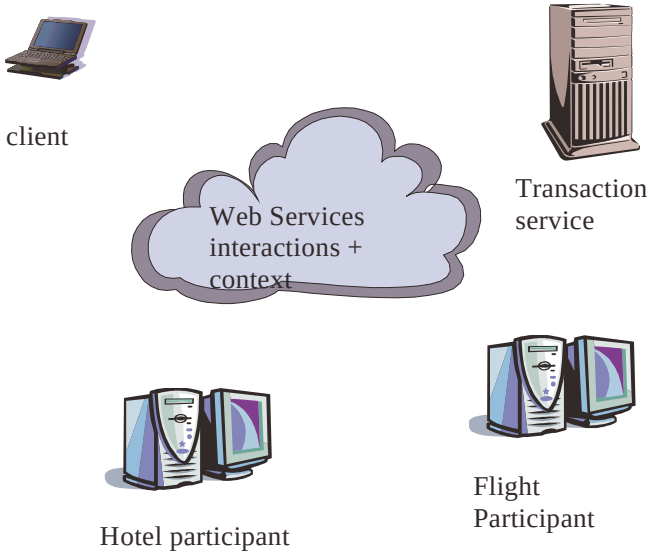

Figure 1: Web Services transaction actors.

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.

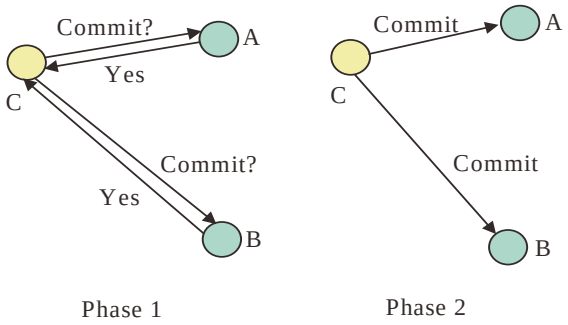

Figure 2: Two-phase commit protocol.

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.

Therefore, 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.

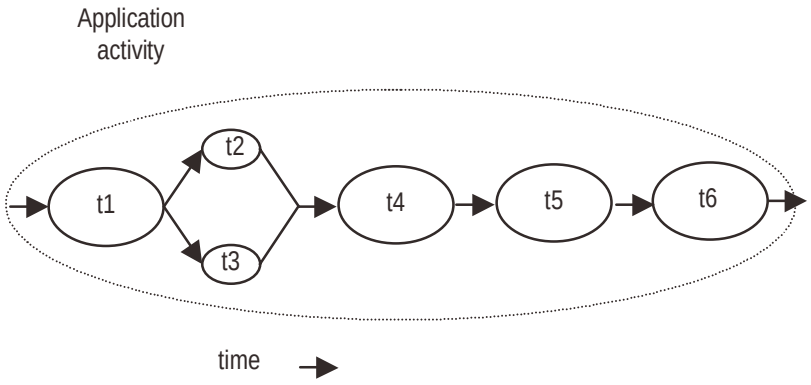

Figure 3: An example of a logical long-running "transaction", without failure.

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.

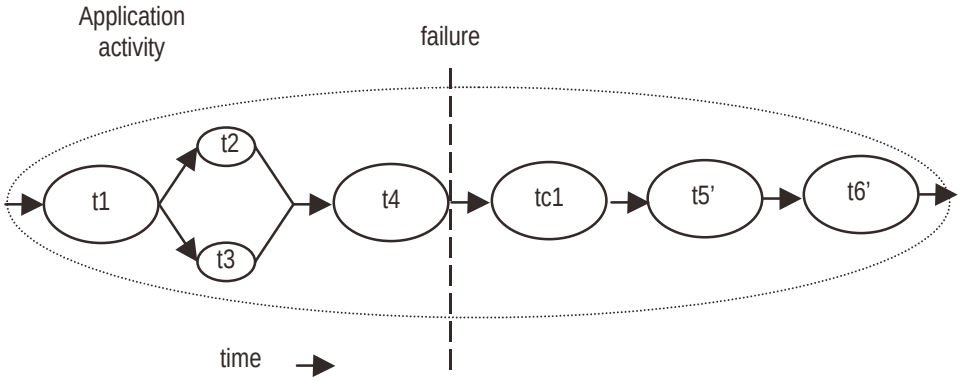

Figure 4: An example of a logical long-running "transaction", with failure.

Much research on structuring applications out of transactions has been influenced by the ideas of *spheres of control* [5]. There are several ways in which some or all of the application requirements outlined above could be met [6][7][8][9][10].

- *Independent top-level transactions*: with this mechanism it is possible to invoke a top-level transaction from within another transaction [6][11]. 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 [7]. 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 [8].
- *Business transaction protocol*: the OASIS Business Transaction Protocol [ref] 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?

2.2 Plugable transactionality

What does it mean to 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 of a Transaction Service 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 [14]. 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 [3].
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.

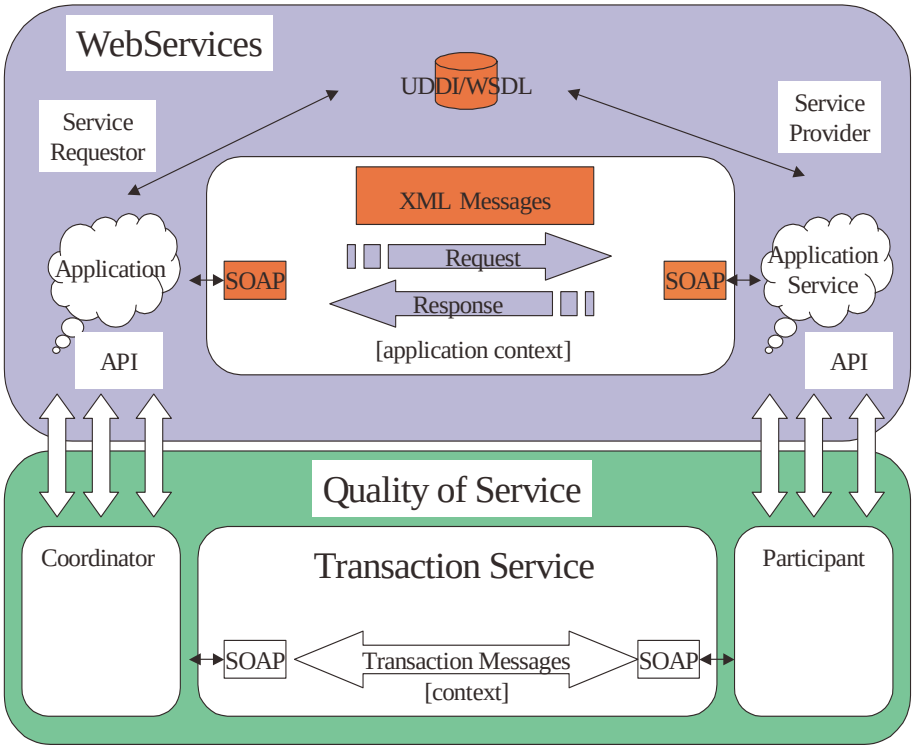

Figure 5: Web Services, transactions and contexts.

This specification does not provide a means whereby existing implementations can be provided to Web Services and their users via SOAP (for example) 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., BTP). What JAXTX does accomplish is to allow these implementations to be plugged into a common services architecture.

The appendices of 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/JTS specifications may be incorporated into Web Services, allowing the implementation of ACID Web transactions and services.

2.3 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 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 performed by the client may be executed under the scope of a transaction such that the coordinator will ensure that the work performed in some manner consistent with the underlying transaction model (e.g., that it either happens or does not despite failures).

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 is then responsible for ensuring that the work is performed in such a way that any final 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 a database when instructed to by the coordinator.

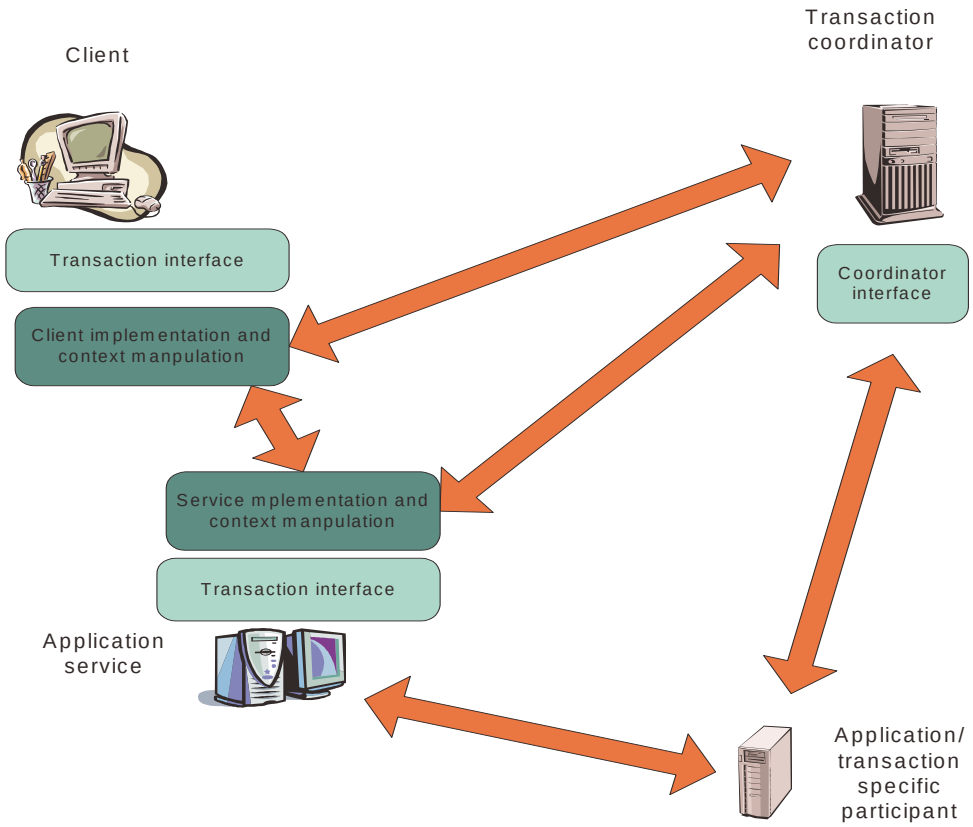

Figure 6: Applications and implementation components.

It is envisioned that in most situations the client/service protocol implementation (essentially the interactions shown) and context manipulation are already provided by existing standards and systems. For example, the OASIS Business Transactions Protocol 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 their specifications at all. JAXTX provides a uniform set of interfaces for Java that are intended to indirect to the underlying implementations.

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.

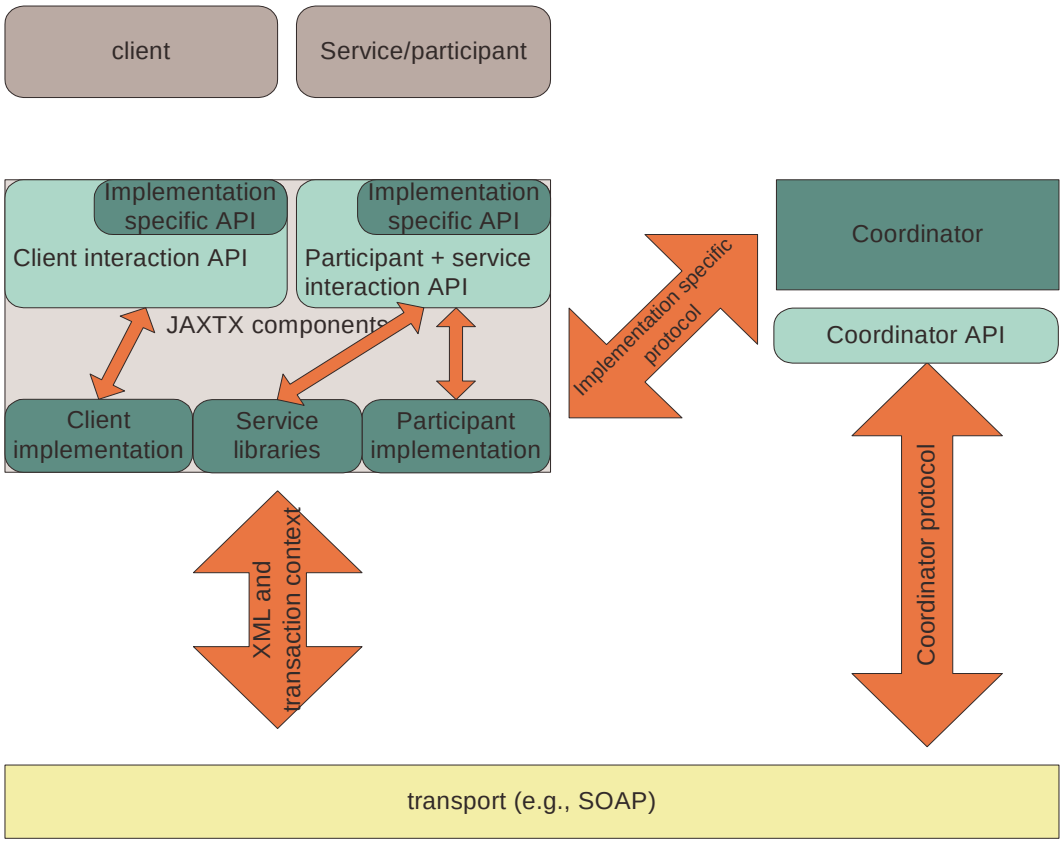

Figure 7: JAXTX architecture.

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 although not a JAXTX component, we describe it here in order to facilitate understanding of the other components.

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.

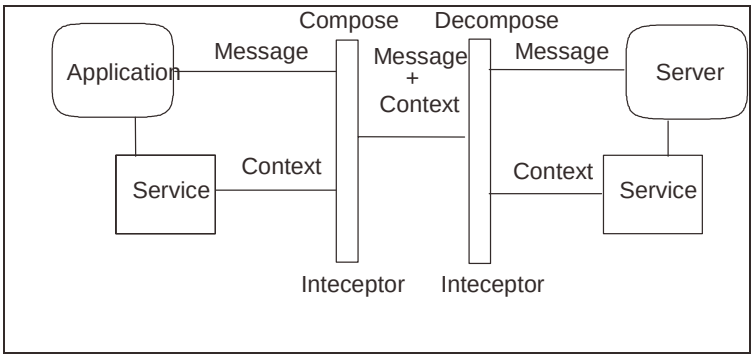

Figure 8: Services and context flow.

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.

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 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 an ACID transaction model, 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.

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.

3.1.5 Coordinator and participant protocol

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 actual final outcome.

The context and coordinator-to-participant protocol are implementation specific and have a format that is typically specified in other protocol documents, e.g., BTP [ref]. It is not the domain of this specification to impinge on these protocols and require changes to them. As such, it is assumed that these will be specified elsewhere and relevant JAXTX component implementations will conform to them. This specification does not attempt to provide a means whereby such protocol differences can be supported by a single implementation, since this is the domain of other specifications [jsr95][omgas]. This allows existing transaction systems to be plugged into a JAXTX implementation unmodified: all that is required is a standardisation of the specific coordinator-to-participant interface, which will already have been agreed upon by the designers of the transaction system in question, and for which JAXTX does provide some support.

3.2 The interfaces

JAXTX defines interfaces which may be categorised as follows:

- *Client*: these interfaces provide new and existing clients with a uniform means by which they may access and use client-specific functionality from different transaction implementations.
- *Service*: these interfaces provide services with a uniform means by which services 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.
- *Coordinator-participant*: these interfaces enable the creation of coordinators that can function within different transaction implementations and hence drive different participant implementations.

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

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, with reference to Figure 7.

4.1.1 Client interaction interface

Despite the variety of transaction implementations, the interfaces that most clients 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 SOAP/XML protocol 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 invocations.

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.

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`. 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 (e.g., the interfaces specified in Section XX) or use reflection to obtain access to the extensions.

4.1.3 Generic coordinator interface

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 smaller scale than for the front-end client, typically involving only registering and un-registering participants. These methods are provided by the `TransactionManager` interface, and as with the `UserTransaction`, services are expected to safely cast, or reflect over, the basic interface to a more protocol specific version if necessary. All configuration information is made available via an instance of the `org.w3c.dom.Document`.

As with the `UserTransaction`, `TransactionManager` implementations are expected to communicate with protocol specific counterparts. However, we do not require that a coordinator will only support a single type of participant. It is possible that a single coordinator may be able to drive multiple types of participant (e.g., two-phase or three-phase). However, a given service may try to enlist the wrong type of participant, and as such the coordinator is responsible for returning an indication of why the registration has failed. The type of participant(s) that a coordinator implementation supports may be made available via its XML configuration document.

4.1.4 Participant interface

The coordinator-to-participant protocol is extremely transaction model specific and leaves little room for generality. As such, the `Participant` interface is essentially a handle for the more specific participant implementation. The `TransactionManager` is expected to determine that the `Participant` is of a type that it can work with (through the available XML configuration information) and then safely narrow the `Participant` to the specific type. If the coordinator does not support the type of participant that is attempted to be enrolled with it, then the coordinator is required to fail the enrolment and indicate the reason why.

4.1.5 Service and context management interface

As shown in Figure 6, the transactional Web Service will require context manipulation functionality, typically provided by the container that the service resides within. As with the client, this context manipulation is expected to be transparent to the service.

4.1.6 XML configuration data

TBD

4.2 JAXTX packages

The JAXTX specification defines four 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 an implementer of JAXTX must conform. The `javax.jaxtx.as`, `javax.jaxtx.jta` and `javax.jaxtx.btp` contain implementation specific enhancements for transaction models conforming to the Activity Service [ref], JTA [ref] and BTP [ref] specifications respectively.

4.2.1 javax.jaxtx package

The classes and interfaces of the `javax.jaxtx` package are required to abstract away from implementation specific details of the underlying transaction model, as illustrated in Figure 6. 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 `CompletionSatus` interface and associated classes define the basic modes in which a transaction can terminate.

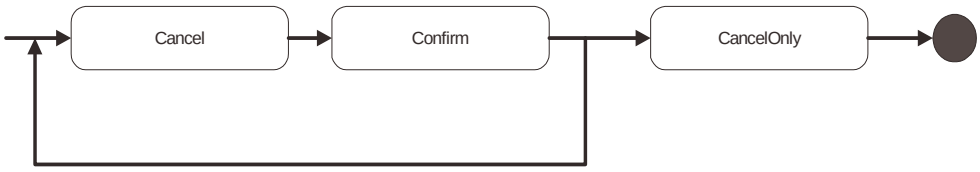

Figure 9: CompletionStatus state diagram.

As shown in Figure 9, at any time in its life a transaction can be in one of three *completion states*, to be described below, such that if it is asked to complete (explicitly by the user or implicitly by virtue of failures) 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 [moss]. 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 `CANCEL` 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 should 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 class implementations of the `CompletionStatus` interface.

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 `CancelOnly`.
- `MarkedCancelOnlyStatus`: the transaction has been forced into a state whereby its eventual completion must be to cancel, i.e., `CancelOnly`.
- `CompletingConfirmStatus`: the transaction transitions to this state when told to complete and the completion status is `Confirm`. The transaction will remain in this state until it has completed in one termination outcome or another. If it cannot complete in the `Confirm` state then the transaction will transition to the `Cancel` 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 `Confirm` 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 `Cancel`. The transaction will remain in this state until it has completed.
- `CompletedCancelStatus`: the transaction has terminated successfully in the `Cancel` completion status.

- `NoTransactionStatus`: there is no transaction associated with the invoking thread.

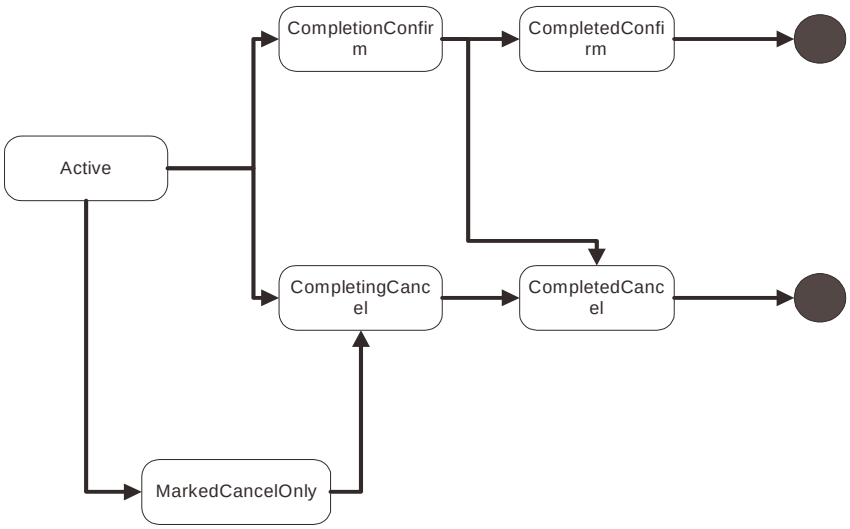

Figure 10: Transaction status transitions.

Note, it is possible that a specific transaction model may have additional states in which the transaction can be, and these should 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` class.

4.2.1.3 Outcome

When a user informs a transaction to terminate, the act of termination may return information to the user. 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. However, an unsuccessful termination, for example because of participant failures, could return a specific heuristic exception [jta].

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 encoding 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.

Additional 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 TxHandle

`TxHandle` 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; it is similar to the `UserTransaction` in the JTA specification [ref]. 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.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.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.InvalidTimeoutException` is thrown. A value of zero means that no application specified time-out is established. This is the default value associated with all using 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.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.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. 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. If no transaction is associated with the current thread then the `javax.jaxtx.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 OTS). If this is the case, the implementation will throw the `javax.jaxtx.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 `CompletionStatus.SUCCESS`), or the state of the transaction does not allow it to be terminated, then the `javax.jaxtx.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 does not return an `Outcome` or throw an exception is assumed to have completed successfully in the manner requested.

A thread of control may require periods of non-transactionality so that it may 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 `TxHandle` instance, which is a handle on the transaction. If the current transaction is nested, then the `TxHandle` 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 `TxHandle`. Prior to association, the thread is disassociated with any transaction(s) with which it may be currently associated. If the `TxHandle` is *null*, then the thread is associated with no transaction. The `javax.jaxtx.InvalidTransactionException` is thrown if the transaction that the `TxHandle` refers to is invalid in the scope of the invoking thread. Obviously the validity or not will be dependant on the underlying transaction implementation to determine.

The `getTransactionHandle` method returns the `TxHandle` for the current transaction, or *null* if there is none. 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 the transaction's status.

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.1234*" or "*javax.jaxtx.jta.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.NoTransactionException` is thrown.

Fundamental to the JAXTX architecture is the notion of a *transaction context*. As illustrated in Figure 11, 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.

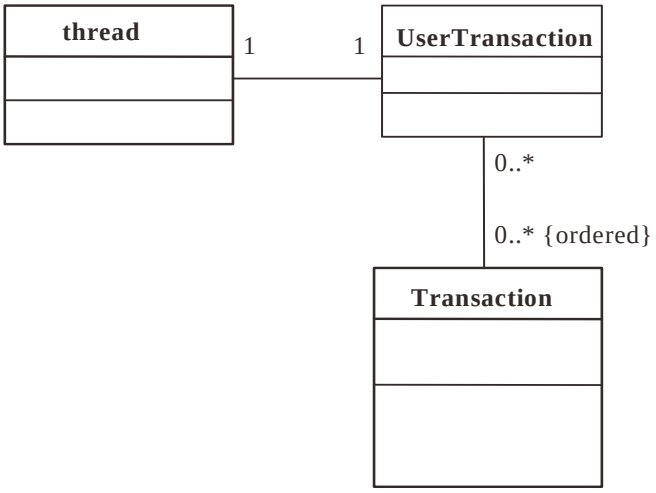

Figure 11: Thread and context relationship.

4.2.1.6 UserTransactionFactory

`UserTransactions` are obtained from a `UserTransactionFactory`, which is made available via JNDI lookup of **`java:comp/UserTransactionFactory`**. 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.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.InvalidSecurityOptionException`.

4.2.1.7 Participant

As shown in Figure 6, the protocol between transaction coordinator and participant will be extremely implementation specific. Therefore, rather than attempt to provide a generic and potentially overly complex interface to a participant, the `Participant` interface is provided so that implementations can identify themselves as participants to a coordinator. The coordinator implementation is expected to use associated XML configuration information (available via the `configuration` method) to determine whether or not the participant implementation has the right higher-level interface, and if so to use it.

4.2.1.8 Transaction

The `Transaction` interface provides a means whereby `Participants` may be enrolled with a specific transaction. Unlike `UserTransaction`, a `Transaction` does

represent a specific transaction. This isolates them from any underlying protocol specific aspects of the implementation they are using. If required, an implementation may enhance this interface and provide additional methods to users.

A `Participant` is enrolled with a transaction using the `enlist` method; if the coordinator is not in a state where enrolment is possible (e.g., it is terminating) then the `javax.jaxtx.InvalidStateException` will be thrown. Additional protocol specific information required to perform the enrolment is provided in the form of an XML document. As mentioned in Section 4.2.1.7, the coordinator implementation is expected to narrow the actual `Participant` interface to a protocol specific version with which it can work. If the `Participant` is not a valid type for the coordinator (i.e., it does not support the right syntactic methods *and* their semantic meaning) then the `javax.jaxtx.InvalidParticipantException` is thrown and the `Participant` will not be enrolled with the transaction.

Some transaction protocols allow for an enrolled `Participant` to leave a transaction. This may be accomplished through the `delist` method. If the coordinator is not in a state where leaving is valid, or the transaction protocol simply does not support `Participants` leaving, then the `javax.jaxtx.InvalidStateException` is thrown, and in this situation the `Participant` should continue to honour the protocol requirements for an enrolled transaction member. If the `Participant` is invalid within the context of the coordinator implementation then the `javax.jaxtx.InvalidParticipantException` will be thrown. A coordinator that does not know about the `Participant` must throw the `javax.jaxtx.UnknownParticipantException` in order to distinguish between the two failure conditions.

4.2.1.9 TransactionManager

The `TransactionManager` interface represents the service/container/participant's (*service-side users*) typical way in which to interact with the underlying transaction service implementation. It derives from the `UserTransaction` and so service-side users can create new transactions, suspend existing transactions etc.

As shown in Figure 12, it is the service container's responsibility to ensure that any imported transaction context is made available to the service via the `TransactionManager` implementation. In the figure, a client begins a new transaction via `UserTransaction.begin()` and then invokes a method (`foo`) on the remote service. The request, along with the transaction context, is transmitted between the client and the service using a protocol specific to the application environment (e.g., SOAP over HTTP).

In a typical transactional application implementation, the incoming service request is intercepted (e.g., by the service container or interceptor), the context is stripped off and a proxy for the imported transaction is created; this proxy is then associated with the thread of control that performs the actual work. Other protocol specific work may also be performed, such as registering a participant or performing interposition [ref].

In a JAXTX environment, all of the above work is obviously performed within the implementation specific layer. The JAXTX implementation of `TransactionManager` is required to utilise this layer in order to be able to present the service with a representation of the current transaction when, and if, required. Note, this may simply be a reference to the original (imported) transaction.

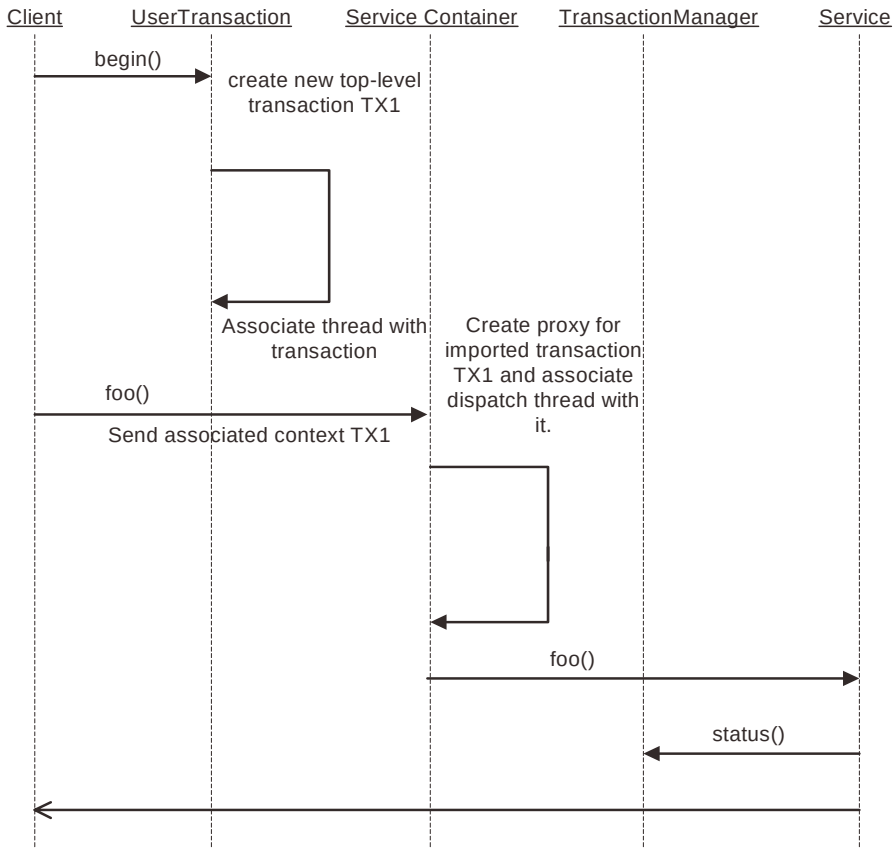

Figure 12: Containers and imported transactions.

In order to register participants with a transaction, the container or participant must use the relevant `Transaction` instance. This is obtained using the `getTransaction` method.

4.2.1.10 Exceptions

The following exceptions and their meanings are used by JAXTX:

- InvalidParentException
- InvalidParticipantException
- InvalidSecurityOptionException
- InvalidStateException
- InvalidTimeoutException

- InvalidTransactionException
- NestingNotAllowedException
- NoPermissionException
- NoSuchServiceException
- NoTransactionException
- SystemException
- UnknownParticipantException

5. Transaction model definitions

In the following sections we shall examine how specific transaction implementations may be mapped into a JAXTX environment.

5.1 The JTA

5.2 The Business Transactions Protocol

6. References