RFC 98 Transactions in OSGi

Draft

16 Pages

Abstract

An increasing number of service specifications in the OSGi Service Platform rely on some form of transactional behaviour. Other service specifications could improve if they had transactional behaviour. This RFC defines a transaction model and identifies Java transaction APIs for use in OSGi environments, including embedded and constrained environments.

0 Document Information

0.1 Table of Contents

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in [1].

`Source code is shown in this typeface.`

0.3 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	Aug 30 2004	Peter Kriens
rewriting	Oct 30 2007	Pavlin Dobrev
0.2	Oct 31 2007	Apply comments from Valentin Valchev
0.3	Nov 08 2007	Pavlin Dobrev – minor misspellings
0.4	Nov 22 2007	Peter Kriens – edit and added comments
0.5	Nov 29 2007	Pavlin Dobrev – answer on some comments
0.6	Dec 13 2007	Pavlin Dobrev – answer to Thomas Watson <tjwatson@us.ibm.com> comments
0.7	Jul 09 2008	Ian Robinson/Roman Roelofsen – rebase on JTA.
0.71	6 Aug 2008	Prepare for public draft.
0.8	21 Nov 2008	Address bugs : 891, 892, 893, 894, 895, 907, 908, 920, 924
0.9	12 Dec 2008	Address bugs : 950, 948, 959, 952, 949, 946, 953, 954, 947, 951,

1 Introduction

An increasing number of APIs in the OSGi are requiring transactional concepts. This is to be expected because transactions can simplify applications that have to run in a dynamic and distributed world. The OSGi expert groups had earlier discussions regarding transactions but at that time (1999) transactions were deemed too heavy and complex to add to the specifications. This RFC suffered the same fate in 2004 for the OSGi Release 4 due to lack of time. However, OSGi R5 seems to be the appropriate time to re-discuss this because of strong requirement for transactions in EEG.

This RFC introduces the transaction concepts and outlines the different trade-offs that need to be made in the API.

2 Application Domain

For the purposes of this specification, a *transaction* is a coordinated series of changes to one or more information stores . In almost any reasonable case, multiple closely related changes are required for a transaction; these changes can depend on external or internal values. This quickly introduces the problem of how to keep the system in a consistent state when there are unexpected failures and multiple parties that may change the same information. Transaction processing systems that address these problems have been at the heart of business computing systems since the early sixties. There are many different types of business transaction but this specification is concerned with those that have the following ACID properties (adapted from [2]):

1. *Atomic* – The transaction completes successfully (commits) or if it fails (aborts) all of its effects are undone (rolled back) The changes include database changes, messages and actions on actuators.
2. *Consistency* – A transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state. This requires that a transaction is a correct program.
3. *Isolation* – Intermediate states produced while a transaction is executing are not visible to other transactions. Furthermore transactions appear to execute serially, even if they are actually executed concurrently.
4. *Durability* – Once a transaction completes successfully (commits), its changes to the state survive failure.

Trying to achieve these properties in a program without proper assistance of the environment is difficult. It therefore became clear quickly to the pioneers in this area that support was needed. At first this was embedded in the database because the problems are most visible in the persistent storage. However, this turned out to be insufficient when multiple persistent stores were involved in the same transaction. For example, a money-transfer operation between two accounts is a transactional operation requiring both the credit and debit parts of the transfer to succeed.

A “local transaction” is one that involves a single information store (resource manager), A “global” or “distributed transaction” is one that may involve two or more resource managers. (The term “distributed” here does not necessarily imply that the transaction spans multiple execution processes – it is “distributed” from the perspective of the resource manager. Both terms are used interchangeably within the literature. This specification will refer to “global” transactions rather than “distributed” transactions).

A global transaction requires a transaction manager that is logically external from the resource managers to coordinate their joint outcome. The common model that evolved over the years is the “two phase commit” (2PC) model with resource managers being directed by the transaction manager. A concrete 2PC protocol implemented by all popular commercial resource managers and transactions is the XA protocol [3]. In the 2PC model, a transaction is started and the program will perform the steps to execute the transaction. Operations on transactional data occur in subsystems such as databases which are the transactional resource managers. Once a resource manager is accessed as part of a global transaction, it “*joins”* the *current* transaction so that the transaction manager is aware that the resource manager needs to participate in the outcome of the transaction. But what is the *current* transaction? One possibility for this would be to pass a *transaction* object in each call. However, this is error prone and cumbersome for the programmer. Since the days of multithreaded server environments, the usual model is that a transaction context is associated with the current *thread* of execution. All calls that are execution in a thread are then assumed to be part of the transaction. A data operation on a certain thread then implies a data operation in the context of a specific transaction and the resource manager for the data store joins that specific transaction for outcome coordination and manages visibility to resource updates in the context of that transaction. A purely thread-based transaction context is obviously not sufficient for distributed systems where the different systems should be part of the same transaction. Specifications such as the CORBA Object Transaction Service (OTS) [4] and WS-AtomicTransaction [5] defines mechanisms by which transaction contexts are implicitly propagated on remote requests over IIOP and SOAP/HTTP transports respectively, to be used by the transaction manager in the target system to associate the work of the local thread with the overall distributed transaction.

At the end of a successful transaction the application program must decide whether to initiate a commit or rollback request for all the changes made within the scope of the the transaction. The program requests that the transaction manager completes the transaction and the transaction manager then negotiates with the resource managers to reach a coordinated outcome. During the first (prepare) phase, an individual resource manager must make durable any state changes that occurred during the scope of the transaction, such that these changes can either be rolled back or committed later once the transaction outcome has been determined. Assuming no failures or vetoes to the successful outcome occurred during the first phase, in the second (commit) phase resource managers may “overwrite” the original state with the state made durable during the first phase. If just one of the resource managers vetoes during the prepare phase then the transaction rolls back.

In order to guarantee consensus, strict two-phase commit is necessarily a blocking protocol: after returning the first phase response, each participant who returned a commit response must remain blocked until it has received the coordinator’s phase 2 message. Until they receive this message, any resources used by the participant are unavailable for use by other transactions, since to do so may result in non-ACID behavior. If the coordinator fails before delivery of the second phase message these resources remain blocked until it recovers. To break this blocking nature, participants that have got past the prepare phase are allowed to make autonomous decisions as to whether they commit or rollback: such a participant *must* record this decision in case it is eventually contacted to complete the original transaction. If the coordinator eventually informs the participant of the transaction outcome and it is the same as the choice the participant made, then there’s no problem. However, if it is contrary, then a non-atomic outcome has obviously happened: a *heuristic outcome*. How this heuristic outcome is reported to the application and resolved is usually the domain of complex, manually driven system administration tools, since in order to attempt an automatic resolution requires semantic information about the nature of participants involved in the transactions.

2.1 Recovery

An important aspect of transaction atomicity is *recovery procesing*. Recovery processing may be required after a transaction manager or resource manager fails unexpectedly during a transaction. This can happen at any moment in time, including between the prepare and the commit phase. The XA 2PC protocol is, by definition, a *presumed-abort* protocol. This means that the transaction manager and resource managers all agree up-front that any failure that occurs before the prepare phase can be assumed to result in rollback. This gives resource managers the right to unilaterally rollback before they are prepared. It also means that a transaction manager does not need to persist any information about the transaction before it has completed the prepare phase.

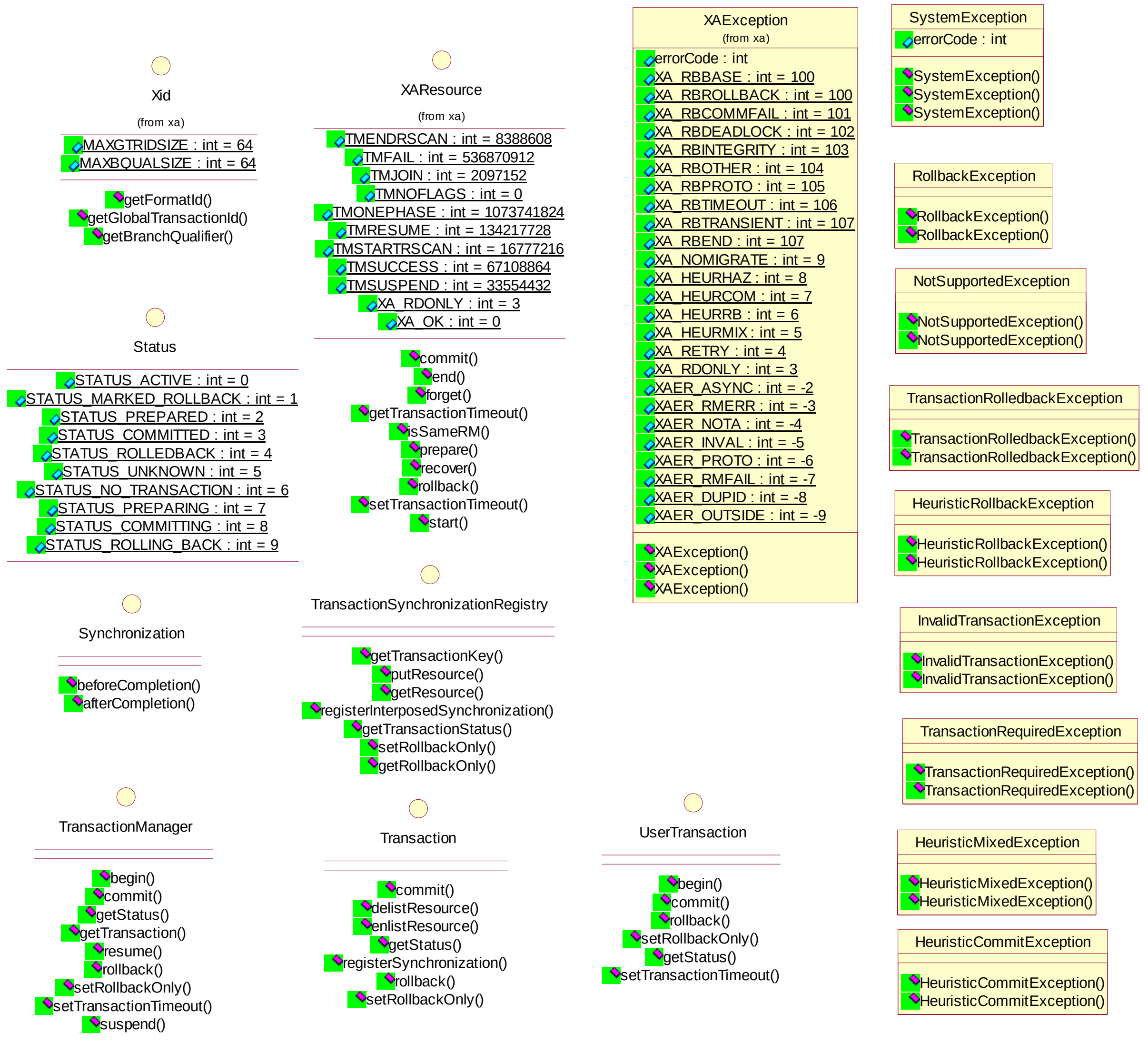
Recovery processing is required following a failure of the transaction manager to resolve any parts of a global transaction that were prepared but not completed at the point when the failure occurred. Resource managers with prepared work may be holding locks and need to be directed to commit or rollback their work. The XA 2PC protocol defines a recovery protocol between the TM and RMs to resolve such “in-doubt” work.

2.2 Java Transaction Architecture

As mentioned above, the most ubiquitous standard 2PC protocol between a transaction manager and a resource manager is the XA protocol [3]. The Java Transaction Architecture (JTA) [6] defines mappings of the XA protocol to local Java interfaces. It specifies the means for XA-compliant resource managers to be coordinated through a process-local Java “XA resource adapter” by a Java transaction manager.

A JTA transaction manager implements the interfaces of the javax.transaction package which contains the key interfaces for transaction management: TransactionManager, UserTransaction, and Transaction. A transactional resource manager implements the interfaces of the javax.transaction.xa packagethat contains the XAResource interface. The model in JTA is that resource managers (instances of XAResource) enlist with the TransactionManager after which they participate in the transaction.

JTA is a Java mapping of the XA interface. It supports all the well-known optimizations of two-phase commit and offers a mature and widely implemented means to support transactions in a Java runtime.



Class Diagram of javax.transaction API

2.3 Why Transactions

The concept of *atomic transactions* has played a cornerstone role in creating today's enterprise application environments by providing guaranteed consistent outcome in complex multiparty business operations and a separation of concerns in applications yielding well designed business process implementations. So just what is an atomic transaction (often abbreviated to just *transaction*)? Put simply, a transaction provides an "all-or-nothing" (atomic) property to work that is conducted within its scope, whilst at the same time ensuring that shared resources are isolated from concurrent users. Importantly application programmers typically only have to start and end a transaction; all of the complex work necessary to provide the transaction's properties is hidden by the transaction system, leaving the programmer free to concentrate on the more functional aspects of the application at hand.

3 Problem Description

Transaction semantics obviously provide a significant increase in robustness and simplicity. ACID properties can be guaranteed on both successful and failed execution.In current OSGi applications, this robust exception model is absent and it is left to the implementers of the different subsystem. In many cases it is virtually impossible to correctly clean up in the light of failures.

This problem is very visible, for example, in the mobile specifications that are being developed for R4. There is currently a proliferation of “transaction” like APIs in the mobile specifications. This proliferation causes:

- Duplication of effort and code on the platform.
- It is also likely that it creates quality problems because implementing transactions is a complex task without centralized support.
- Requires that that the transaction coordination is handled by the application programmer because application subsystems cannot find a current transaction to join.
- Increases the learning curve for application programmers due to the different semantics associated with the different subsystems
- There is no central overview of the transactional state of the system. This seriously hinders diagnosing and debugging systems as well as help desks.

There is therefore a need to centralize the management of transactions. This must minimize the code size, increase reliability, and enable tools.

Requirements

Provide a comprehensive model that allows components in an OSGi Service Platform to perform their actions in a transactional way.

1. Identify the Java APIs that must be provided by a Transaction Manager to delineate a transaction boundary and provide a means for resources to join a transaction. Identify the Java API that must be provided by a resource manager to support two-phase commit, including recovery.

2. The transaction API must be suitable both for enterprise runtimes and embedded systems

3. Reuse existing widely used Java transaction technology wherever possible and avoid repeating what is already specified elsewhere.

4. The specification should place no requirements on a transaction service implementation to be recoverable. It should be noted, however, that a transaction service implementation can only provide transaction atomicity if it supports recovery processing following a failure.

5. The transaction service specification must allow implementation of transactional applications without the need for external changes to the application's business interfaces and configuration.

4 Technical Solution

4.1 Using JTA

The existing JTA specification addresses the requirements stated above; in addition many providers of Java resource adapters already implement the JTA XAResource interface. The JTA specification [6] defines all the Java transaction interfaces and semantics for transaction and resource managers and this specification reuses those APIs as-is with no modification. The XA+ specification [3] defines the semantics of the underlying XA protocol. This specification defines only additional information related specifically to the OSGi architecture, such as how a transaction service reference can be discovered in the OSGi service registry.

4.2 Compliance

This specification defines an *OSGi transaction service implementation* as one or more OSGi bundles that collectively implement the classes and interfaces of the javax.transaction package defined in [6]. A *compliant* OSGi transaction service implementation must pass all the tests defined by the OSGi transaction service compatibility suite. A compliant OSGi transaction service implementation is not required to be certified as a compliant JTA implementation, although it may optionally be certified as compliant to the JTA specification when used as part of a Java EE profile.

4.3 Components of the Transaction Service

There are four basic roles for transaction support in the OSGi framework:

1. **Transaction originator**. This may be either an application role or a framework role and it is responsible for demarcating transactional units of work. Application components use the JTA UserTransaction interface to demarcate transaction contexts.

2. **Transaction manager**. The transaction manager provides the implementation of transaction capability. It responds to requests to demarcate transaction contexts, associates transaction contexts with the local thread of execution, potentially distributes/receives transaction contexts on remote requests, accepts the registration of transactional participants and coordinates participants to an atomic outcome. The transaction manager is the core of the OSGi transaction support and MUST implement the javax.transaction.UserTransaction, javax.transaction.TransactionManager, javax.transaction.Transaction and javax.transaction.SynchronizationRegistry interfaces as defined in [6]. The transaction manager is a logical component consisting of one or more OSGi services which are registered in and available through the OSGi service registry as described in 4.4.

3. **Volatile resources**. Some objects have an interest in the outcome of the transaction but do not participate in 2PC, for example persistent caches that need to be flushed at the end of the transaction immediately prior to 2PC. These objects implement the javax.transaction.Synchronization interface and are enlisted in the transaction via the TransactionSynchronizationRegistry.registerInterposedSynchronization(Synchronization).

4. **Transactional resources**. Resource managers that participate in 2PC provide an implementation of the javax.transaction.xa.XAResource interface. XAResources are enlisted in a transaction via the Transaction.enlistResource(XAResource) interface.

4.3.1 Transaction Originator

A transaction is requested to begin and end (commit or rollback) by the transaction originator and, once started, a transaction context is associated with the thread of its originator. Application components begin a global transaction using the begin() method of UserTransaction. Transactions may also be originated by framework components using the javax.transaction.TransactionManager interface. This is a richer interface than the UserTransaction interface and provides additional transaction context management operations such as suspend() and resume() that are not appropriate for application use.

Limitations:

- A global transaction may only be associated with a single thread at any point in time. The specific thread to which a transaction is associated may change over time. A thread may have no more than one global transaction concurrently associated with it.

4.3.2 Transaction Manager

The transaction manager is a logical component consisting of one or more OSGi services and provides transactional capabilities for the framework. It

- Provides the means for a transaction originator to demarcate transactional boundaries, through the javax.transaction.UserTransaction and javax.transaction.TransactionManager interfaces.

- Provides a means to represent an instance of a specific transaction using a javax.transaction.Transaction object, obtained by calling the getTransaction() method of a TransactionManager object.

- Maintains an association between a Transaction object and a thread of execution such that the Transaction is associated with at most one thread at a time.

- Accepts enlistment of volatile and transactional resources in a specific transaction.

- Notifies volatile resources of the outcome of the transaction.

- Coordinates transactional resources using the two-phase commit protocol at the end of the transaction.

- Drives the recovery interface of transactional resource following a failure of the transaction manager to ensure the atomic completion of transactional work.

4.3.3 Volatile Resources

Volatile resources are components that do not participate in 2PC but are called immediately prior to and after 2PC. If a request is made to commit the transaction then the volatile participants have the opportunity to perform some “beforeCompletion” processing such as flushing cached updates to persistent storage. Failures during beforeCompletion will cause the transaction to rollback. In both the commit and rollback cases the volatile resources are called after 2PC to perform “afterCompletion” processing (which cannot affect the outcome of the transaction).

4.3.4 Transaction Resources

Transaction resources are provided by transactional resource managers and MUST implement the javax.transaction.xa.XAResource interface described in [6]. An XAResource object can be enlisted with the transaction to ensure that work is performed within the scope of the transaction. The XAResource interface is driven by the transaction manager during the completion of the transaction and is used to direct the resource manager to commit or rollback any changes made under the transaction.

4.4 Locating OSGi transaction services

The Java EE specifications define standard JNDI names for the UserTransaction and TransactionSynchronizationRegistry interfaces in a Java EE server environment and deliberately do not define a standard means for acquiring an implementation of the TransactionManager interface. This is because the latter is considered to be a part of the internal implementation of a Java EE application server. An OSGi transaction service implementation MUST register service objects for the UserTansaction, TransactionSynchronizationRegistry and TransactionManager interfaces in the OSGi service registry, using the names “javax.transaction.UserTransaction”, “javax.transaction.TransactionSynchronizationRegistry” and “javax.transaction.TransactionManager” respectively. An OSGi transaction service MAY put restrictions on which bundles can use these service objects as described in 5.1.

An OSGi transaction service implementation MAY also bind references to UserTransaction and TransactionSynchronizationRegistry in a JNDI namespace.

4.5 Use Cases

4.5.1 Create Transaction

An application component uses the UserTransaction service interface to start a new transaction. If there is already active transaction in the context of the current tread, the transaction manager will indicate an error.

4.5.2 Join Transaction

When a transaction is started, the application performs some operations on the system. While modifying the current state, it invokes some methods or other services. These services are resource managers that can participate in the transaction. The resource managers each provide an XAResource object and join the transaction associated with the current thread via the enlistResource(XAResource) method of the Transaction object.

4.5.3 Commit Transaction

After performing the required operations, the transaction originator decides whether to initiate commit or rollback processing and requests it via the UserTransaction interface. During commit processing if at least one of the resources participating in the transaction fails to perform the required operations (vetoes the prepare phase), the transaction is rolled back.

4.5.4 Prepare Resource

The transactional manager uses the “two phase commit” protocol to ensure the consistent outcome across all resource managers. When the originator requests a commit of the transaction, the TM calls “prepare” on each participating resource. In this stage, the resource provisionally performs any updates and decides whether it can honour a commit outcome if that is what the external coordinator decides. It then waits to be told the final commit or rollback decision.

4.5.5 Commit Resource

If all participating resources have been successfully prepared, the TM calls the commit() method on each resource manager. The resource manager is responsible for making changes to the transactional resource persistent and visible outside the transaction.

4.5.6 Rollback Transaction

If the originator decides to request a rollback of the transaction, or if the transaction fails before a completion request is made, the original state of the system is restored to what existed before the transaction was started.

Rollback might be called either because commit failed or for some external reasons – like operation timeout as example.

4.5.7 Rollback Resource

During a transaction rollback, the TM calls “rollback” on each resource. This method discards any provisional updates within the transaction and so restores the original state of the resource.

4.5.8 Assign Transaction to Thread

A transaction MUST NOT be associated with more than one thread at a time but MAY be moved over time from one thread. While transaction-thread association is provided by the transaction manager, any movement of the transaction from one thread to another – via the suspend/resume methods of the TransactionManager interface - is driven by the framework hosting the OSGi transaction service and it is the responsibility of that framework and the transactional resource managers to understand which transaction context the transactional resources are running under.

4.6 Functionality

4.6.1 Scope of a global transaction

A transaction context is started and ended by requests from a transaction originator. In between,a transaction manager manages the transaction. Transactional resources may be enlisted in the transaction during its lifetime. Those transactional resources are coordinated to an atomic outcome by the transaction manager at the end of the transaction.

4.6.2 Correctness of the State

At the end of transaction, the application must decide whether the changes should be made persistent (committed) or rolled back. The application requests the transaction manager to perform commit or rollback processing. The collection of state changes to all transactional resources made under the transaction can have ACID properties when a global transaction is used. The transaction manager itself is responsible for providing the Atomic property of ACID by driving all resource managers to the same outcome. The resource managers are responsible for the Isolation and Durability properties of the changes. Theapplication and the resource managers provide the Consistency of the data changes.

4.6.3 End of Transaction

The transaction is disposed after it has been successfully committed or rolled back. The transaction manager automatically disassociates the transaction from the participating thread. This allows a new transaction to be started for that thread.

4.6.4 Performance

Global transaction processing can be expensive in terms of performance and resource utilization. Therefore, to optimize performance you may choose to execute a majority of the code without a transaction, and use transactions only when necessary. Using the credit card processing example, you may not use transactions to do *data loading, validation, verification, and posting*. However, at the point when you transfer money from the account holder to the holding bank you would then start a transaction. The XA and JTA specifications provide opportunities for implementation optimizations such as the well-known "one phase commit optimization of two-phase commit" which causes almost all of the cost of 2PC to be realized only when a transaction with more than one transactional resource begins commit processing.

4.6.5 Management of Transaction

The component that completes a transaction should be the same component that originated it. Therefore, only the business method that started the transaction should invoke the commit() and rollback() methods. Spreading transaction management throughout the application adds complexity and reduced maintainability of the application from a transaction management standpoint.

4.6.6 Heuristic Exceptions

Heuristic outcomes can result if a transactional resource does not keep the promise it made during the prepare phase, most typically as a result of a database administrator forcing a unilateral and administrative outcome for operational reasons. Under such circumstances, the administrator may need to take further action to maintain integrity across the global transaction as a whole.

4.6.7 Examples

4.6.7.1 Example 1 - Creating and using a Transaction

The following pseudo code illustrates the creation and use of a transaction involving a transactional resource "ConfigResource":

```
ServiceReference txRef =
    bundleContext.getServiceReference("javax.transaction.UserTransaction");
UserTransaction tx = (UserTransaction)bundleContext.getService(txRef);

// begin transaction
tx.begin();

// perform some operations in the context of the transaction
try {
  // Create a transactional resource
  ConfigResource config = ...;
  // Perform some transactional work
  Configuration x = config.createConfiguration("abc");
  tx.commit(); // make changes persistent
} catch (Throwable th) {
  th.printStackTrace();
  tx.rollback(); // rollback changes on fail
}
```

4.6.7.2 Example 1 - (Resource) Participating in Transaction

The following pseudo code illustrates the enlistment of an XAResource with a transaction; not all the methods of the XAResource interface are shown.

```
class ConfigResource implements XAResource {

  public Configuration createConfiguration(String pid) {
      // TransactionManager can be obtained from service registry
      ServiceReference tmRef =
         bundleContext.getServiceReference("javax.transaction.TransactionManager");
      TransactionManager tm = (TransactionManager)bundleContext.getService(txRef);
      // enlist the transactional resource
      tm.getTransaction().enlistResource(this);
      // Transactional operation
      addLog("createConfiguration", pid);
      ...
  }


  public void prepare(Xid xid) {
    // Make and persist a provisional “after copy” of data for this xid
    ...
  }

  public void commit(Xid xid, boolean onePhase) {
    // Replace the “before” copy with the “after” copy
    // Forget the transaction

  }

  public void rollback(Xid xid) {
    // Discard any provisional “after” copy
    // Forget the transaction
  }
  ...
}
```

5 Security Considerations

5.1 ServicePermissions

An OSGi service framework provider MAY chose to restrict which bundles are allowed to register OSGi transaction services. A framework provider optionally does this by requiring that any registering bundle has the ServicePermission.REGISTER permission for each of the service names listed in 4.4 that it wishes to register.

Similarly, an OSGi service framework provider MAY chose to restrict which bundles are allowed to get references to OSGi transaction service objects. A framework provider optionally does this by requiring that any bundle requesting a service reference has the ServicePermission.GET permission for the service names listed in 4.4 that it wishes to get.

6 Document Support

6.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Transaction Processing, Jim Gray and Andreas Reuter. Morgan Kaufmann Publishers, ISBN 1.55860-190-2

[3]. Distributed Transaction Processing: The XA+ Specification Version 2, The Open Group, ISBN: 1-85912-046-6

[4]. Object Transaction Service v1.4, OMG, http://www.omg.org/cgi-bin/doc?formal/2003-09-02

[5]. WS-AtomicTransaction v1.1, OASIS, http://docs.oasis-open.org/ws-tx/wsat/2006/06

[6]. JTA Specification v1.1, http://java.sun.com/products/jta/

6.2 Author's Address

Name	Peter Kriens
Company	aQute
Address	9C, Avenue St. Drézéry
Voice	+15123514821
e-mail	Peter.kriens@aQute.biz

Name	Pavlin Dobrev
Company	ProSyst Software GmbH
Address	Dürener Str. 405, 50858 Cologne, Germany
Voice	+49 221 6604-0
e-mail	p.dobrev@prosyst.com

Name	Roman Roelofsen
Company	ProSyst Software GmbH
Address	Duerener Str. 405, 50858 Cologne, Germany
Voice	+492216604406
e-mail	r.roelofsen@prosyst.com

Name	Ian Robinson
Company	IBM
Address	IBM Hursley Lab, Hursley Park, WINCHESTER, Hants, SO21 2JN, UK
Voice	+44-1962-818626
e-mail	ian_robinson@uk.ibm.com

6.3 Acronyms and Abbreviations

6.4 End of Document