**XA Transactions/JTA**

Some terminology

**ACID properties**: Atomicity, Consistency, Isolation, Durability

**Application Server and Webserver**:

Application server and web server in Java both are used to host Java web application. On Java J2EE perspective main difference between web server and application server is support of EJB. In order to run EJB or host enterprise Java application (.ear) file you need an application server like JBoss, WebLogic, WebSphere or Glassfish, while you can still run your servlet and JSP or java web application (.war) file inside any web server like Tomcat or Jetty. Application Server supports distributed transaction and EJB, while Web Server only supports Servlets and JSP. In terms of logical difference between web server and application server, web server is supposed to provide http protocol level service while application server provides support to web service and expose business level service e.g. EJB.

Application servers are heavier than web server in terms of resource utilization.

Application server example: Glassfish, JBoss, Wildfly

Webserver: Apache tomcat, Jetty

**Open Source JTA Implementations**: JBossTS, Atomikos and Bitronix. (for standalone apps)

# XA transactions using Spring

### JTA/XA transactions without the J2EE container.

Using J(2)EE application server has been a norm when high-end features like transactions, security, availability, and scalability are mandatory. There are very few options for java applications, which require only a subset of these enterprise features and, more often than not, organizations go for a full-blown J(2)EE server. This article focuses on distributed transactions using the JTA (Java Transaction API) and will elaborate on how distributed transactions (also called XA) can be used in a standalone java application, without a JEE server, using the widely popular Spring framework and the open source JTA implementations of JBossTS, Atomikos and Bitronix.

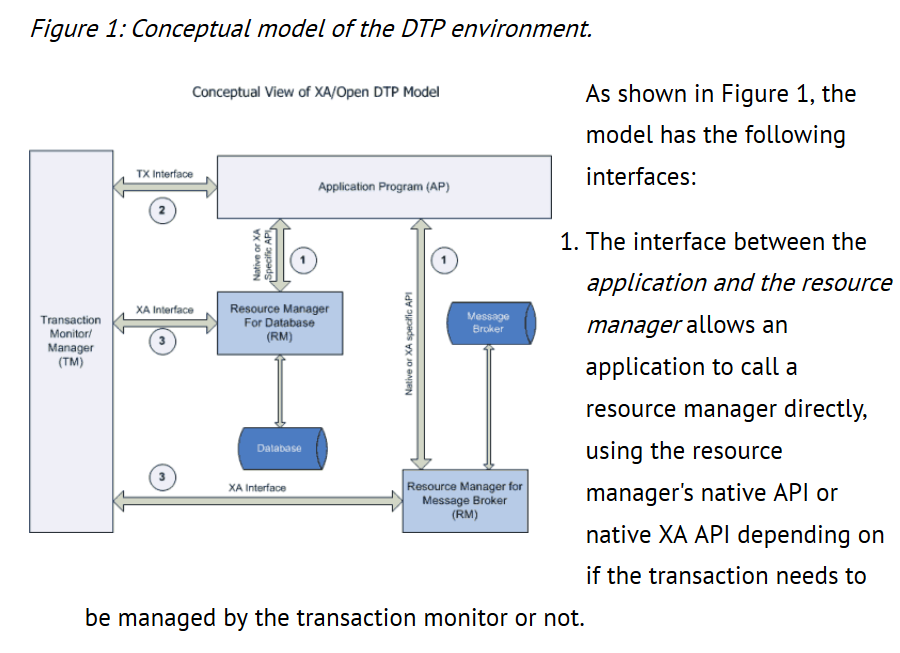
Distributed transaction processing systems are designed to facilitate transactions that span heterogeneous, transaction-aware resources in a distributed environment. Using distributed transactions, an application can accomplish tasks such as retrieving a message from a message queue and updating one or more databases in a single transactional unit adhering to the ACID (Atomicity, Consistency, Isolation and Durability) criteria. This article outlines some of the use cases where distributed transactions (XA) could be used and how an application can achieve transactional processing using JTA along with the best of the breed technologies. The main focus is on using Spring as a server framework and how one can integrate various JTA implementations seamlessly for enterprise level distributed transactions.

XA Transactions and the JTA API

Since the scope of the article is limited to using JTA implementations using the Spring framework, we will briefly touch upon the architectural concepts of distributed transaction processing.

XA Transactions

The X/Open Distributed Transaction Processing, designed by Open Group(a vendor consortium), defines a standard communication architecture that allows multiple applications to share resources provided by multiple resource managers, and allows their work to be coordinated into global transactions. The XA interfaces enable the resource managers to join transactions, to perform 2PC (two phase commit) and to recover in-doubt transactions following a failure.



1. The interface between the *application and the transaction monitor* (TX interface), lets the application call the transaction monitor for all transaction needs like starting a transaction, ending a transaction, rollback of a transaction etc.
2. The interface between the *transaction monitor and the resource manager* is the XA interface. This is the interface, which facilitates the two-phase commit protocol to achieve distributed transactions under one global transaction.

#### JTA API

The *JTA API*, defined by Sun Microsystems, is a high-level API which defines interfaces between a transaction manager and the parties involved in a distributed transaction system. The JTA primarily consists of three parts:

* A high-level application interface for an application to demarcate transaction boundaries. The UserTransaction interface encapsulates this.
* A Java mapping of the industry standard X/Open XA protocol (Item #3 in the X/Open interfaces listed above). This encompasses the interfaces defined in the javax.transaction.xa package, which consists of XAResource, Xid and XAException interfaces.
* A high-level transaction manager interface that allows an application server to manage transactions for a user application. The TransactionManager, Transaction, Status and Synchronization interfaces pretty much define how the application server manages transactions.

Now that we have a brief summary of what JTA and XA standards are, let us go through some use cases to demonstrate the integration of different JTA implementations using Spring, for our hypothetical java application.

## Our Use Cases

To demonstrate the integration of different JTA implementations with Spring, we are going to use the following use cases:

1. *Update two database resources in a global transaction*- We will use JBossTS as the JTA implementation. In the process, we will see how we can declaratively apply distributed transaction semantics to simple POJO's.
2. *Update a database and send a JMS message to a queue in a global transaction*- We will demonstrate integration with both Atomikos and Bitronix JTA implementations.
3. *Consume a JMS message and update a database in a global transaction*- We will use both Atomikos and Bitronix JTA implementations. In the process, we will see how we can emulate transactional MDP's (Message Driven POJO's).

We will be using MySQL for the databases and Apache ActiveMQ as our JMS messaging provider for our use cases. Before going through the use cases, let us briefly look at the technology stack we are going to use.

## Spring framework

Spring framework has established itself as one of the most useful and productive frameworks in the Java world. Among the many benefits it provides, it also provides the necessary plumbing for running an application with any JTA implementation. This makes it unique in the sense that an application doesn't need to run in a JEE container to get the benefits of JTA transactions. Please note that Spring doesn't provide any JTA implementation as such. The only task from the user perspective is to make sure that the JTA implementation is wired to use the Spring framework's JTA support. This is what we will be focusing on in the following sections.

#### Transactions in Spring

Spring provides both programmatic and declarative transaction management using a lightweight transaction framework. This makes it easy for standalone java applications to include transactions (JTA or non-JTA) either programmatically or declaratively. The programmatic transaction demarcation can be accomplished by using the API exposed by the PlatformTransactionManager interface and its sub-classes. On the other hand, the declarative transaction demarcation uses an AOP (Aspect Oriented Programming) based solution. For this article, we will explore the declarative transaction demarcation, since it is less intrusive and easy to understand, using the TransactionProxyFactoryBean class. The transaction management strategy, in our case, is to use the JtaTransactionManager, since we have multiple resources to deal with. If there is only a single resource, there are several choices depending on the underlying technology and all of them implement the PlatformTransactionManager interface. For example, for Hibernate, one can choose to use HibernateTransactionManager and for JDO based persistence, one can use the JdoTransactionManager. There is also a JmsTransactionManager, which is meant for local transactions only.

Spring's transaction framework also provides the necessary tools for applications to define the transaction propagation behavior, transaction isolation and so forth. For declarative transaction management, the TransactionDefinition interface specifies the propagation behavior, which is very much similar to EJB CMT attributes. The TransactionAttribute interface allows the application to specify which exceptions will cause a rollback and which ones will be committed. These are the two crucial interfaces, which make the declarative transaction management very easy to use and configure, and we will see as we go through our use cases.

#### Asynchronous Message Consumption using Spring

Spring has always supported sending messages using JMS API via its JMS abstraction layer. It employs a callback mechanism, which consists of a message creator and a JMS template that actually sends the message created by the message creator.

Since the release of Spring 2.0, asynchronous message consumption has been made possible using the JMS API. Though Spring provides different message listener containers, for consuming the messages, the one that is mostly suited to both JEE and J2SE environments is the DefaultMessageListenerContainer (DMLC). The DefaultMessageListenerContainer extends the AbstractPollingMessageListenerContainer class and provides full support for JMS 1.1 API. It primarily uses the JMS synchronous receive calls( MessageConsumer.receive()) inside a loop and allows for transactional reception of messages. For J(2)SE environment, the stand-alone JTA implementations can be wired to use the Spring's JtaTransactionManager, which will be demonstrated in the following sections.

## The JTA implementations

#### JBossTS

JBossTS, formerly known as Arjuna Transaction Service, comes with a very robust implementation, which supports both JTA and JTS API. JBossTS comes with a recovery service, which could be run as a separate process from your application processes. Unfortunately, it doesn't support out-of-the box integration with Spring, but it is easy to integrate as we will see in our exercise. Also there is no support for JMS resources, only database resources are supported.

#### Atomikos Transaction Essentials

Atomikos's JTA implementation has been open sourced very recently. The documentation and literature on the internet shows that it is a production quality implementation, which also supports recovery and some exotic features beyond the JTA API. Atomikos provides out of the box Spring integration along with some nice examples. Atomikos supports both database and JMS resources. It also provides support for pooled connections for both database and JMS resources.

#### Bitronix JTA

Bitronix's JTA implementation is fairly new and is still in beta. It also claims to support transaction recovery as good as or even better than some of the commercial products. Bitronix provides support for both database and JMS resources. Bitronix also provides connection pooling and session pooling out of the box.

## XA Resources

#### JMS Resources

The JMS API specification does not require that a provider supports distributed transactions, but if the provider does, it should be done via the JTA XAResource API. So the provider should expose its JTA support using the XAConnectionFactory, XAConnection and XASession interfaces. Fortunately Apache's ActiveMQ provides the necessary implementation for handling XA transactions. Our project (see Resources section) also includes configuration files for using TIBCO EMS (JMS server from TIBCO) and one can notice that the configuration files require minimal changes when the providers are switched.

#### Database Resources

MySQL database provides an XA implementation and works only for their InnoDB engines. It also provides a decent JDBC driver, which supports the JTA/XA protocol. Though there are some restrictions placed on the usage of some XA features, for the purposes of the article, it is good enough.

# XA Transactions (2 Phase Commit): A Simple Guide

In the early days of computing, there was no need for distributed transactions. As number of applications increased, synchronization of the data become an important issue. Companies paid a lot to maintain synchronized systems in terms of data flow. As a result, the 2 phase commit protocol referred to as XA(eXtended Architecture) arose. This protocol provides ACID-like properties for global transaction processing. Throughout this article, I will try to explain details of XA transactions and use of XA Transactions in Spring framework.

2 phase commit protocol is an atomic commitment protocol for distributed systems. This protocol as its name implies consists of two phases. The first one is commit-request phase in which transaction manager coordinates all of the transaction resources to commit or abort. In the commit-phase, transaction manager decides to finalize operation by committing or aborting according to the votes of the each transaction resource. We will next move on to implementation details of 2PC protocol.

XA transactions need a global transaction id and local transaction id(xid) for each XA resource. Each XA Resource is enlisted to XA Manager by start(xid) method. This method tells that XA Resource is being involved in the transaction(be ready for operations). After that, the first phase of the 2PC protocol is realized by calling prepare(xid) method. This method requests OK or ABORT vote from XA Resource. After receiving vote from each of XA Resource, XA Manager decides to execute a commit(xid) operation if all XA Resources send OK or decides to execute a rollback(xid) if an XA Resource sends ABORT. Finally, the end(xid) method is called for each of XA Resource telling that the transaction is completed. Look at the figure to understand better. As we build a background in XA transaction implementation, we will next go deeper and see types of failures and possible solutions.

Failures can occur at any time due to network loss, machine down and some administrator mistake. In XA transaction, we will categorize these failures according to the phases that they occur. The first failure phase is before protocol is started. This is a simple failure that system does need not to rollback or any kind of operation. We just do not do the operation for that particular moment. Second type of failure can occur at prepare(commit-request) phase which can be easily handled by rollbacks using timeout policies. Last but not the least is commit phase failures which can occur due to incomplete rollbacks and any problem in chain. In all of these above situation, transaction manager tries to recover the problem. We will next see how transaction manager tries to overcome failures.

For recovery, transaction manager calls recover method of each XA resource. XA Resources trace the logs and tries to rebuild its latest condition. Transaction Manager calls necessary rollback operations and mission is accomplished. This process can seem to be happy path but there are a lot of exceptional situations where logs are problematic like being corrupted. In these kinds of situations, transaction manager follows some heuristics to solve the problem. Moreover, the recovery process depends on the write-ahead logs where you write operation logs before applying. For performance issues these logs are written in their own format(not using any serialization) and system should better batch them if possible. We next go to fun part which is XA transaction support by Spring framework.

Spring framework provides extensive environment to develop web and stand alone applications. Like other utilities it provides, XA transactions are also supported by Spring. However, this support is not a native implementation and requires hibernate, web container or a framework that provides XA Transaction Management. Spring has JtaTransactionManager that provides transaction management utilities and hides the details. By this way, we can have transaction management for multiple DataSources which are updated simultaneously. When it comes to use XA Transaction Management, hibernate and web containers support for XA transactions are well documented, do not need to be mentioned. However, working with a framework that provides XA transactions can be confusing. Thus, I will continue this post by introducing Bitronix Transaction Manager.

Bitronix is easily configured while providing good support for transaction management. It is not commonly used in stand alone applications but I will try to give configuration for stand-alone application as follows.

<bean id="bitronixTMConfig" factory-method="getConfiguration" class="bitronix.tm.TransactionManagerServices">

<!--Disabling Jmx avoids registering JMX Beans to any container-->

<property name="disableJmx" value="true" />

</bean>

<bean id="bitronixTM" factory-method="getTransactionManager" class="bitronix.tm.TransactionManagerServices" depends-on="bitronixTMConfig" destroy-method="shutdown"/>

<bean id="transactionManager" class="org.springframework.transaction.jta.JtaTransactionManager">

<property name="transactionManager" ref="bitronixTM" />

<property name="userTransaction" ref="bitronixTM" />

<property name="allowCustomIsolationLevels" value="true" />

</bean>

We can now have multiple data sources which can be configured as follows. Each data source should have a uniqueName property which is unique. Below configuration is for Oracle, other databases can have different configurations. For any other detail, you can check Bitronix website.

bean id="xaDataSource" class="bitronix.tm.resource.jdbc.PoolingDataSource" init-method="init" destroy-method="close">

<property name="uniqueName" value="xaDataSource" />

<property name="minPoolSize" value="1" />

<property name="maxPoolSize" value="4" />

<property name="testQuery" value="SELECT 1 FROM dual" />

<property name="driverProperties">

<props>

<prop key="URL">jdbc:oracle:thin:@10.6.86.24:1521:test</prop>

<prop key="user">test</prop>

<prop key="password">test</prop>

</props>

</property>

<property name="className" value="oracle.jdbc.xa.client.OracleXADataSource" />

<property name="allowLocalTransactions" value="true" />

</bean>

To sum up, we have tried to explain what is XA Transactions, underlying protocols and Bitronix Transaction Management integration with Spring in a stand alone application. To extend, XA Transactions provides modifying different data sources at the same time. Furthermore, XA Transactions are supported by web containers or hibernate like frameworks. Nevertheless, we may need to integrate transaction management to a stand alone application in which we must configure transaction manager. In consequence, XA transaction provides consistent operations on multiple data sources and companies make use of them.