Da <http://xactor.sourceforge.net/guide/html/introduction.html>

.2. Distributed Transactions

A *distributed transaction* is transaction that comprises operations involving persistent resources on multiple nodes.   
  
The classical example of a distributed transaction is a transfer of funds between accounts kept in different database servers. Transfer of funds is a transaction that comprises two operations: a debit operation, which takes an amount of money from a given account, and a credit operation, whith credits that amount to another account.   
  
If the accounts involved live in different database servers, then the transfer of funds is a distributed transaction.

*Atomicity* is a crucial property for distributed transactions. When you do a transfer of funds, you should not be at risk of loosing money: under no circumstance the amount could be withdrawn from the source account but not credited the destination account. You need to be sure that this will not happen even if the database server that hosts the destination account crashes in the middle of the transaction.

1.3. Two-Phase Commit

*Two-phase commit* (2PC) is a completion protocol for distributed transactions.   
  
It ensures transaction atomicity in distributed scenarios.   
  
The 2PC protocol runs at transaction commit time, when the transactional application tells the transaction manager to commit a transaction, and consists of two-phases:   
  
a *voting phase* and   
a *termination phase*.

Both phases of the protocol are driven by a *coordinator*, which is usually the transaction manager running in the node where the transaction originated.

1. In the **voting phase,** the coordinator sends a *prepare* message to every resource involved in the transaction.   
   Each resource that keeps persistent data (such as a relational database) attempts to record in its own transaction log all the pending changes performed as part of the transaction, so that the transaction can be later committed even in the event of a resource crash.   
   If such a resource succeeds in persistently recording the pending changes in its transaction log, then it replies to the coordinator with a *vote commit* message. Otherwise the resource replies to the coordinator with a *vote rollback* message.

After collecting the votes from all participant resources, the coordinator defines the transaction outcome, which can be either commit or rollback.   
  
If all participants voted commit, the outcome is commit.   
  
Otherwise the transaction cannot be committed, and its outcome is rollback.   
  
At the end of the voting phase, the coordinator records the commit decision in its own transaction log, to ensure that the commit decision will be effective even in the event of a coordinator crash.

1. In the **termination phase**, the coordinatior sends *commit* or *rollback* messages to all participant resources, depending on the transaction outcome.   
     
   If the outcome is commit, each resource applies to itself the pending changes recorded in its transaction log.   
     
   If the outcome is rollback, each resource undoes any changes that may have been performed as part of the transaction.

1.4. Logging and Crash Recovery

What if a transaction coordinator crashes in the middle of the transaction? Or if one or more resources involved the transaction crash before the transaction reached its end? Crash recovery is the reason why the coordinator and the resources write the relevant 2PC information to a *transaction log* (also known as *recovery log*).

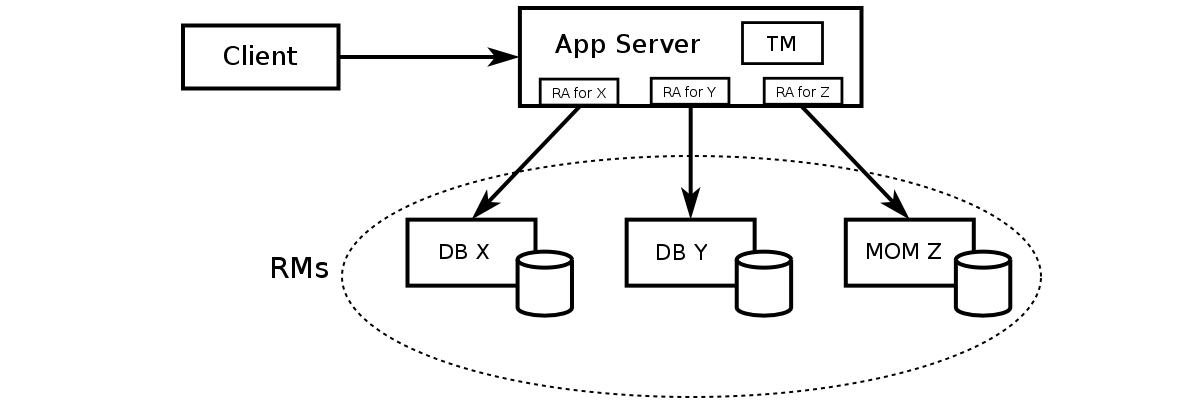
When a crashed server comes back up, its *recovery procedure* will be executed. The recovery procedure scans the transaction log of the crashed server and ensures that any pending transactions complete in an atomic way.

## 2.4. Single-TM and Multi-TM Transactions

A distributed transaction may involve a single transaction manager (TM) or it may involve multiple TMs. In a Java application server scenario, a single-TM transaction is a transaction that runs in a single application server. The application server's TM is the single TM involved in the transaction. A multi-TM transaction is a transaction that reaches multiple application servers. Each of these application servers has its own TM, which gets involved in the transaction.

### 2.4.1. Single-TM Transactions

[Figure 2.1, “Distributed transaction involving a single TM”](http://xactor.sourceforge.net/guide/html/concepts.html#single-tm-tx.fig) shows a single-TM transaction scenario. The application server starts the transaction on behalf of the client. The transaction uses three resource managers (RMs): it updates persistent data in two database systems (DBs X and Y) and performs a message send to a persistent queue in a message-oriented middleware system (MOM Z). It reaches these resource managers through resource adapters (RAs) specific to each resource manager: a JDBC driver for database X (RA for X), another JDBC driver for database Y (RA for Y), and a JMS provider for messaging system Z (RA for Z).



**Figure 2.1. Distributed transaction involving a single TM**

#### 2.4.1.1. TM-RM Interaction: the XA Standard

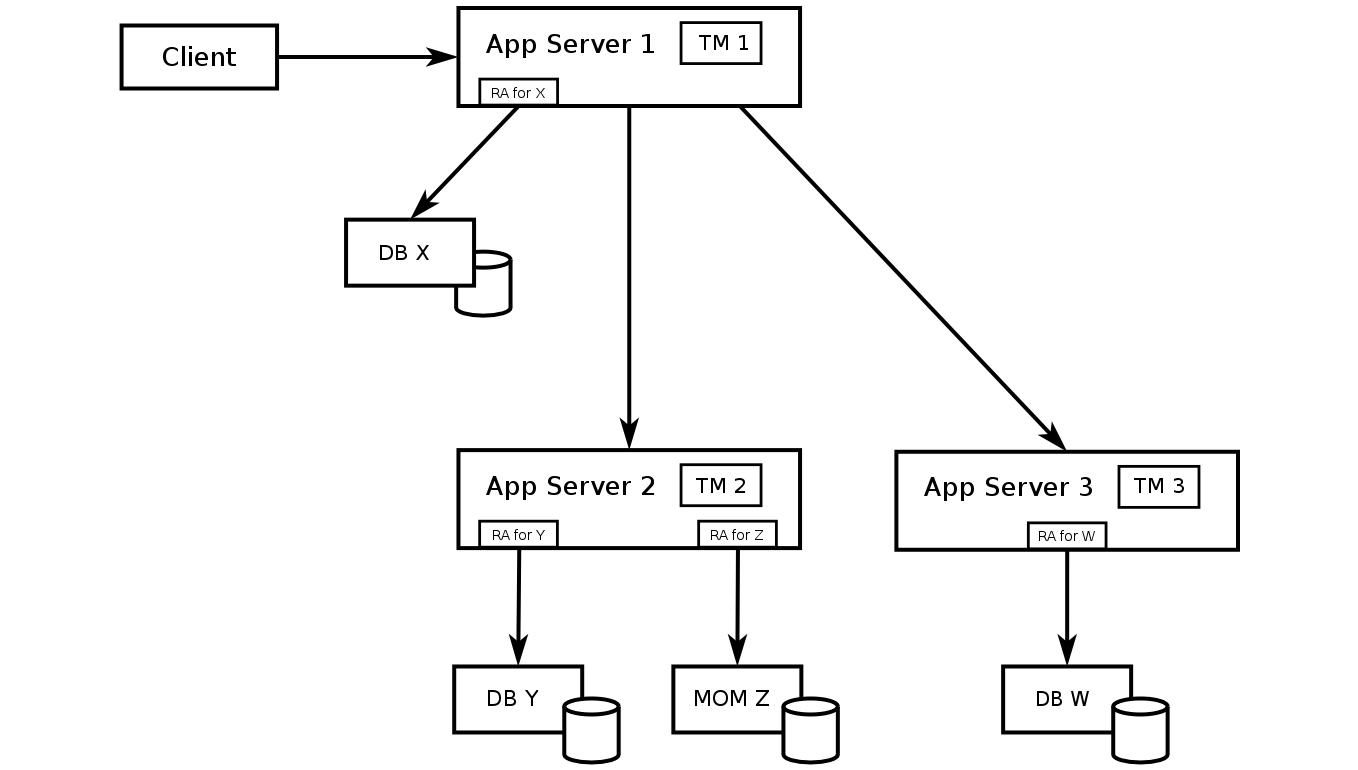
The X/Open XA specification and its Java mapping, the interface javax.transaction.XAResource (a part of the JTA specification), are the relevant standards for single-TM transactions. For this reason, single-TM transactions are also known as XA transactions.

To support distributed transactions, a resource adapter must implement the XAResource interface. At transaction commit time, the TM drives the two-phase commit protocol by calling XAResource methods (such as prepare and commit) on the RAs. On each such call, the RA takes care of sending the corresponding 2PC message to its RM, over an RM-specific protocol.

The bottom line is: for an RM to take part in distributed transactions, an RA with XA support is needed. In the case of a database system, this means an XA-enabled JDBC driver. In the case of a messaging system, it means an XA-enabled JMS provider. In the case of an enterprise information system (such as an ERP system), it means a JCA resource adapter with XA support.

### 2.4.2. Multi-TM Transactions

[Figure 2.2, “Distributed transaction involving multiple TMs”](http://xactor.sourceforge.net/guide/html/concepts.html#multi-tm-tx.fig) shows a multi-TM transaction scenario. Application server 1 starts the transaction on behalf of the client. The transaction uses services from components deployed in application servers 2 and 3. Each of the three application servers involved in the transaction has connections with one or more RMs and asks these RMs to perform work as part of the transaction.



**Figure 2.2. Distributed transaction involving multiple TMs**

Since the transaction originated in application server 1, its coordinator is TM 1, the transaction manager running in that application server. TM 2 and TM 2 act as sub-coordinators: while TM 1 views them as resources that partcipate in the transaction, the RMs to which they are connected view them as coordinators. To emphasize the difference between the coordinator role and sub-coordinator roles, the term root coordinator is also used for the coordinator of a multi-TM transaction.

At transaction commit time, the root coordinator (TM 1) drives the voting phase of 2PC by calling XAResource.prepare on the RA for X and by sending prepare requests to TMs 2 and 3. Each of these TMs then drives a "local voting phase" across its RMs, by callingXAResource.prepare on the corresponding RAs. If all RMs connected to a sub-coordinator (either TM 2 or TM 3) vote commit, the sub-coordinator sends a vote commit reply back to the root coordinator. Otherwise the sub-coordinator sends a vote rollback reply to the root coordinator.

After the voting phase is over, the root coordinator drives the completion phase of 2PC in a similar way. Each sub-coordinator gets a commit message (or a rollback message) from the root coordinator and drives a "local completion phase" across it RMs, by calling XAResource.commit (or XAResource.rollback) on the corresponding RAs.

Note that a plain XA transaction (a single-TM transaction) is a particular case of multi-TM transaction, in which the root coordinator is the only TM involved in the transaction. A transaction that reaches other TMs is the fully distributed case.

#### 2.4.2.1. Inter-TM Communication

XA and its Java mapping standardize the interaction between a TM, which acts as root coordinator or sub-coordinator of a given transaction, and the RMs involved in that transaction. What about the interaction between TMs, in a multi-TM transaction? Are there standards that define that interaction?

The most traditional standard for inter-TM communication is the Object Transaction Service (OTS), one of the CORBA services defined by the OMG. OTS, however, is applicable only if IIOP is used as the underlying remote method invocation protocol. Even though the J2EE and JEE specifications require all Java application servers to support IIOP for interoperability, many application servers typically support other protocols as well.

The JBoss Application Server allows users to choose between IIOP and JBossRemoting, a remote invocation framework that runs over various transport protocols, including a socket-based transport, an SSL socket-based transport, JRMP, and HTTP(S). Accordingly, JBoss JTS supports inter-TM communication either through OTS or through a JBoss-specific DTM service that is very similar to OTS. Instead of being based on CORBA/IIOP, the DTM service sits atop JBossRemoting. It can be used in homogeneoous scenarios comprised of multiple JBoss Application Server instances. OTS, on the other hand, supports heterogeneous scenarios that include JBoss Application Server instances and non-JBoss servers/TMs as well.

#### 2.4.2.2. Support to Heterogeneous Resources

...

#### 2.4.2.3. Transaction Context Propagation and Coordinator Interposition

...

## 2.5. Transaction Context Propagation

A remote method invocation involves sending a request to a particular object in a remote server. In more general terms, invoking any remote service involves sending a request message to the server that provides the service. If such an invocation is performed within a transaction, the service request carries additional information — the so-called transaction context.

Consider the following issues:

1. The client (possibly a multithreaded one) might have started multiple transactions and it might be invoking services that should performed (by the same server) as part of different transactions. The corresponding service requests arrive at the server over the same transport-level connection. How does the server distinguishes the service requests that should be performed as part of transaction A from the ones that should be performed as part of transaction B?
2. A server that performs "undoable work" for some transaction must register itself with the transaction coordinator. By doing so, the server tells the transaction coordinator to take it account, as a remote resource, when driving the 2PC protocol at transaction commit time. The server might not have received the service request directly from the node whose TM is acting as transaction coordinator. How does the server knows who the transaction coordinator is?

Propagating a transaction context along with transactional requests is a simple solution to these two issues. The transaction context includes the following fields:

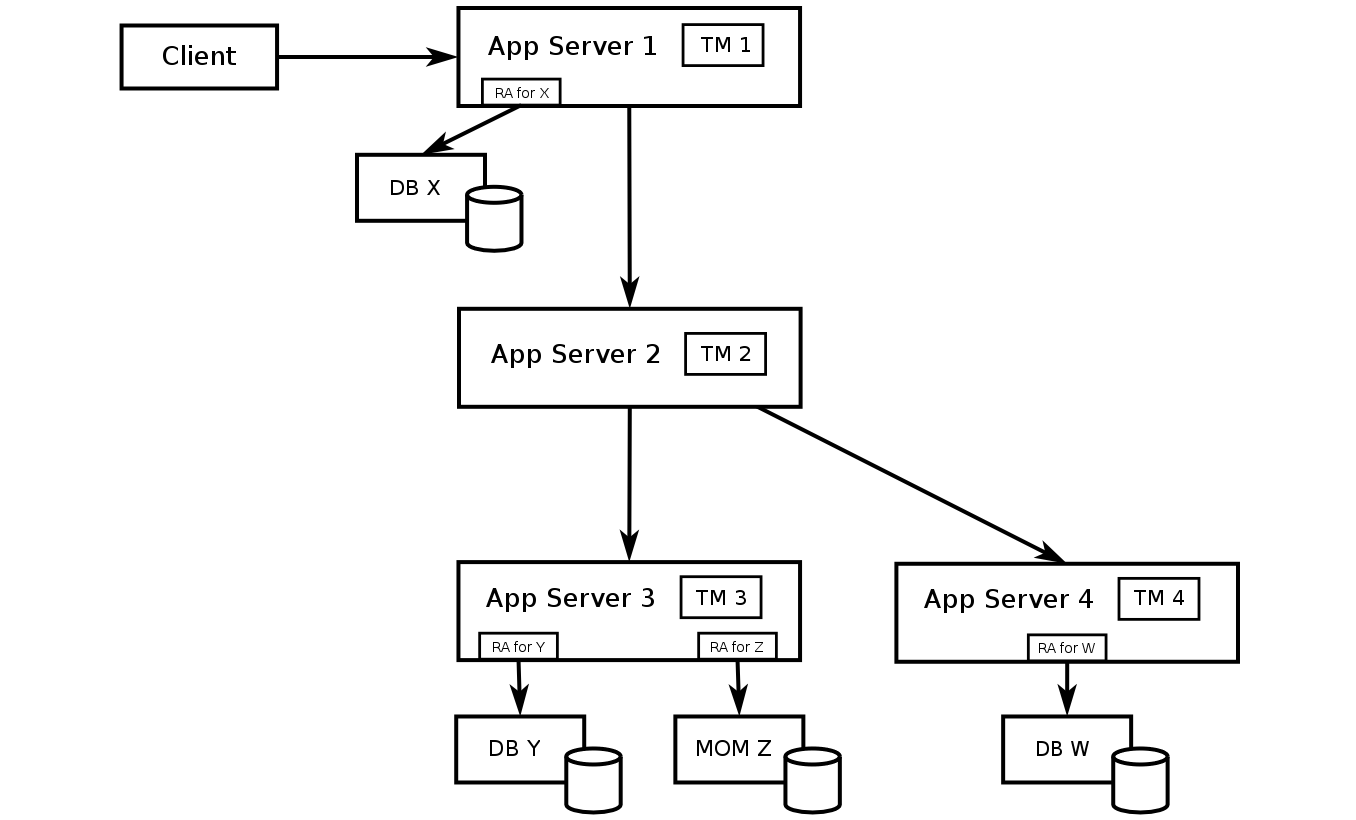
* + a globally unique transaction identifier, and
  + a reference to the transaction coordinator.

The global transaction identifier addresses issue 1 above. When a transaction starts, its orginator TM generates a transaction identifier in a way that ensures uniqueness across all transactions, including those that were born at other TMs.

The coordinator reference addresses issue 2. Its form depends on the protocol used for service invocations. If that protocol is IIOP, then the coordinator reference is a stringfied CORBA reference (IOR). If JBossRemoting is used instead, then the coordinator reference is a JBoss-specific URL. The coordinator reference field of a transaction context frequently points to the root coordinator (the TM where the transaction originated), but it may otherwise point to a sub-coordinator.

## 2.6. Coordinator Interposition

Consider the multi-TM transaction scenario in [Figure 2.3, “Distributed transaction touching an application server that does need to be involved in the 2PC protocol.”](http://xactor.sourceforge.net/guide/html/concepts.html#multi-tm-tx2.fig). Application server 1 starts the transaction on behalf of the client. The transaction uses a service from some component deployed in application server 2, which in turn uses services from components deployed in application servers 3 and 4. Each of the application servers 1, 3, and 4 has connections with one or more RMs and asks these RMs to perform work as part of the transaction. Application server 2, however, acts merely as a middleman. Neither it uses an RM nor does it perform "undoable work" in any way.



**Figure 2.3. Distributed transaction touching an application server that does need to be involved in the 2PC protocol.**

Due to its middleman role, application server 2 does not need to be involved in the 2PC protocol.