

ImplementationDetails

BTM Implementation details

BTM is made of multiple modules which map as close as possible the JTA API.

This page contains descriptions of those components and how they work internally in BTM 1.3 and 2.x.x as version 1.2 and earlier did not fully respect this modularization.

Contents

- [Bitronix implementation specifics](#)
 - [XID](#)
- [2PC engine](#)
 - [asynchronous2Pc](#)
- [XA connection pooling framework](#)
 - [allowLocalTransactions](#)
 - [twoPcOrderingPosition](#)
 - [deferConnectionRelease](#)
- [Disk journal](#)
- [Recovery engine](#)
 - [currentNodeOnlyRecovery](#)

Bitronix implementation specifics

XID

The JTA specification defines XID as an interface for which an implementation must be provided by the transaction manager. There are 3 important pieces composing a XID: the format ID, the Global TRansaction ID (GTRID) and the Branch QUALifier (BQUAL).

Format ID

The format ID identifies the transaction manager which generated the XID. It basically is an integer which must be unique across all implementations. The BTM format ID simply is the int-encoded version of the "Btnx" ASCII string: 0x42746e78.

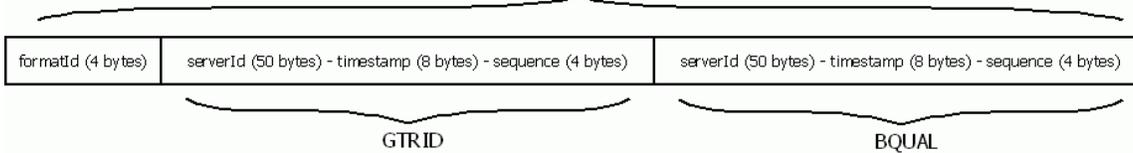
GTRID

The GTRID uniquely identifies the global transaction across all participating resources. The JTA spec says it must fit in a 64 bytes array and must be globally unique. The BTM implementation builds them by combining 3 data: the serverId, a timestamp and a sequence number.

BQUAL

The BQUAL uniquely identifies the local part of a global transaction inside a single participating resource. As for the GTRID the JTA spec says it must fit in a 64 bytes array and must be globally unique. The BTM BQUALs are built with the exact same format as the GTRID.

XID



- The **serverId** is the configured ASCII string, see [Configuration2x#TransactionengineSettings](#).
- The **timestamp** is the amount of milliseconds returned by calls to <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/System.html#currentTimeMillis%28%29>
- The **sequence** is a static counter that each JVM maintains independently. It is re-initialized to 0 each time the JVM running a BTM instance is restarted.

2PC engine

todo.

asynchronous2Pc

Async 2PC is a very complex optimization that isn't useful in most cases. The idea is to run the 2PC protocol in parallel (ie: sending the phase 1 then phase 2 commands in parallel to all participating resources). In theory this should give you a big performance boost but in practice it's negligible except if you have lots of participating resources. Performance improvement has been measured on average only when at least 4 resources are participating in transactions.

It is not recommended to enable this setting unless you measured your average 2PC execution time and identified a bottleneck.

XA connection pooling framework

XA transactions require connection pools with specific knowledge of transactions' state. For instance it is not possible to use a non-XA connection pool like C3P0 or Apache DBCP with the transaction manager.

An abstract XA connection pool lies at the heart of BTM with JDBC and JMS layers on top of it. Everything which can be pooled or cached actually is: JDBC and JMS connections are pooled, JDBC prepared statements can be cached as well as JMS sessions, producers and consumers.

Extra services and optimizations are also provided like automatic enlisting and delisting, deferral of connection closing, ordering of resources in the 2PC protocol, connection recycling, non-XA safeguard and Last Resource Commit.

allowLocalTransactions

todo.

twoPcOrderingPosition

todo.

deferConnectionRelease

todo.

Disk journal

The disk journal is a simple two-files (called *fragments*) rollover transaction store, this sometimes is called *append only* journal or *write ahead log*. It provides services to the transaction manager engine to record transaction states and reporting of transactions left in an unfinished state: transactions which reached the end of the 1st phase of the 2PC protocol (*prepare*) but suffered from a failure before the 2nd phase (*commit or rollback*) could finish up.

The disk journal implements all known optimizations:

- pre-allocated storage of variable size
- journal compacting
- disk force (or disk sync) batching with a flip-flop algorithm

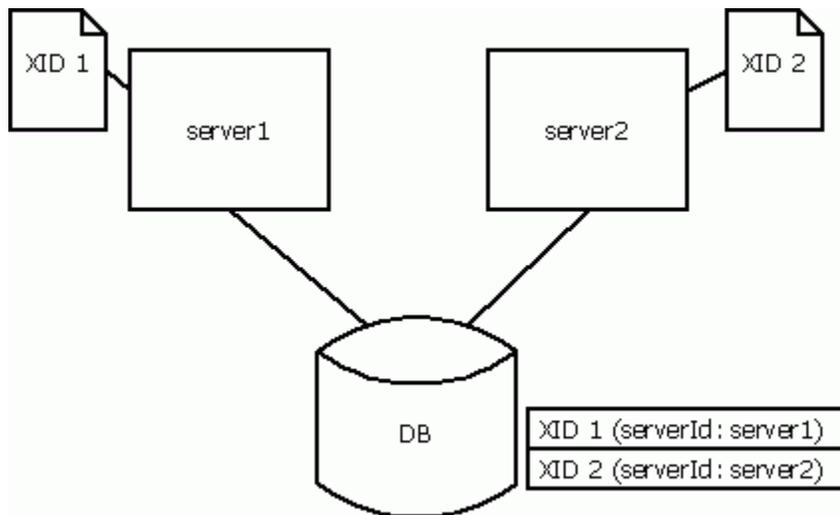
Recovery engine

todo.

currentNodeOnlyRecovery

This setting is very important when you're running a cluster of virtual machines all working on the same resource. In this case BTM will run multiple times in parallel, each node creating different transactions on the resource. Since the XIDs are guaranteed to be globally unique this isn't a problem for the transaction manager core.

The recovery engine can on the other hand get confused with this setup. Take for instance this scenario:



Two servers are accessing a single database. Server 1 started a transaction identified by XID 1 while at the same time server 2 started a transaction identified by XID 2. Now what would happen if recovery kicks in in the background on server 1 (or if server 1 gets restarted) while XID 2 is still in-flight on server 2? Server 1 would ask its journal for a list of unfinished transactions and compare that against the list of XIDs it got by querying the database. We can ignore XID 1 in this case as the TM has knowledge of it and we can assume it would handle it fine. But what is server 1 supposed to do with XID 2? It is reported by the database as an unfinished transaction and server 1 has no trace of it in its journal. As per the *presumed abort* optimization the recovery engine will assume that the transaction needs to be rolled back as it has no knowledge that it still is running on server 2!

To avoid this confusion, the recovery engine can be configured to peek at the XID to determine if it actually started it before taking any action. It can do that by extracting the `serverId` part of the XID stored in the database and comparing it to the current node's configured `serverId`. If they don't match then the XID can be ignored assuming

another node is taking/will take care of it.

This is what the `currentNodeOnlyRecovery` setting is about: it tells the recovery engine to ignore XIDs started by another node.