



DESIGN PROPOSAL

**Java Transaction API Integration with
Mulgara 1.1**

Andrae Muys

andrae@netymon.com

26th May 2007

Netymon Pty Ltd - ABN 56 113 088 145

Unit 3 / 13 Denham St Annerley 4103 Australia - +61 7 3392 1296 - mail@netymon.com

JTA Support Proposal

Requirement and History

Since early 2004 it has been an explicit long term goal of the Mulgara development team to provide support for J2EE interfaces where we could. As steps towards this goal Mulgara has made use of various standard interfaces internally, with the intent that as our understanding of these interfaces and their interaction with mulgara improved they could be exposed as external J2EE interfaces. Two of these in particular are the internal use of Managed-Beans, providing internal support for the Java Management eXtensions standard; and the internal use of the XAResource Interface and the Jboss TransactionManager providing internal support for the Java Transaction API. Other standards that are regularly revisited, but have proved more problematic to integrate include the JDBC and J2EE Connector API standards.

The Transaction API in particular is an important goal. JTA is the standardised mechanism for the correct handling of distributed transactions across heterogeneous applications. Mulgara's status as a metadata store suggests that many of our users will want to be able to query both metadata and concrete-data within the context of a single transaction. In the absence of such a capability, there is going to be increasing pressure to require Mulgara to store both, distracting us from further developing the core focus of semi-structured data management.

It is in recognition of this fact that JTA was originally used as the basis for transaction management in the Resolver SPI. It is intended that the design of the new version of the store-layer (XA2) should provide the transactional semantics required to finally provide full external JTA compliance. It has however been suggested that while a fully compliant JTA interface will require the development of XA2, the partial support possible with current store layer may be worth the effort to develop in the short-term pre-XA2.

Full JTA Support Requirements

In the language of the JTA specification (1.1) a Mulgara instance is referred to as a "Resource Manager"; a Session as a "Transaction Resource". From the view of a client, all transactional communication with the Resource Manager or the Transaction Resource is mediated via a "Transaction Resource Adaptor" - represented by the XAResource interface. The operations on this interface fall into four categories.

- Status methods (get|setTransactionTimeout/isSameRM)
- Transaction Lifecycle (start/suspend/resume/end)
- Transaction State (prepare/commit/rollback)
- Recovery (recover)

While the status methods are straight forward, integrating the other three categories with

Mulgara's existing transaction architecture each poses a particular challenge. In each case there are limitations as to the extent Mulgara can satisfy the full requirements of the standard.

Transaction Lifecycle

When participating in an external transaction there are two transaction lifecycles that need synchronising. First is the global transaction, represented by a global transaction-id. The second is the transaction internal to Mulgara that has no id that is independent of the individual resolvers participating in a query; rather it is represented by a Java Object that will need to be associated with the global id in some way. This association presents two problems.

The first, and by far the hardest to overcome, is the extensive use of implicit transactions in Mulgara. All read-only transactions are implicit, created in the process of executing a query, and with a lifetime bounded by the reference counts associated with the queries results.

The simplest approach is to constrain the use of external JTA support to read/write transactions — however it is extremely important to Mulgara's performance that these be restricted to operations that do actually perform writes. This leads to a non-orthogonality in the design that is awkward.

A more complex approach is to introduce the concept of a read-only transaction. The client obtaining the JTA adaptor from Mulgara would then be responsible for specifying if it was for read-only, or read-write use. Mulgara's internal transaction manager would need to be modified to allow a read-only transaction to be created explicitly, with its lifecycle determined by operations on the JTA adaptor. The JTA standard provides no guidance as to how a transaction resource adaptor is obtained from a transaction resource — it is therefore perfectly legitimate to require such a parameter to a `getXAResource()` method on the Mulgara Session interface. However this distinction between read-only and read/write transactions is one not recognised by the standard. Exactly how we might integrate this distinction into JTA's concept of a transaction is still unclear.

Combining the two (pretending to provide read-only transactions, but instead implicitly creating a new transaction for each query) is only slightly more complicated than restricting the adaptor to read/write transactions only. The resulting interface is also cleaner, and much preferable. On the other hand this compromise approach will entail violating transactional isolation. There would be no isolation between the JTA transaction and concurrent modifications to Mulgara — in effect providing only faux transactions except in the case of an explicit read/write transaction initiation.

It is worth noting that with the introduction of multiple-writer support with XA2 this problem will become moot. It is only while we differentiate between read-only and read/write transactions that this problem arises.

The second problem is the interaction between the JTA adaptor, and the current transaction

lifecycle methods on Mulgara's Session Interface. Alternative options include:

Mandating that the two approaches are incompatible and requiring clients to use only one transaction control interface for a given Session. It is unclear what work would be required to enforce this separation.

Redefining the existing methods to be equivalent to the relevant calls on the JTA adaptor. Possibly as a first step towards deprecating them completely in preference to the external JTA adaptor.

The core of this issue is the question of what to do with any existing transaction when the JTA adaptor receives a TRANSACTION_START event. Should the adaptor terminate the existing transaction or attempt to adopt the pre-existing mulgara transaction; allowing for the requirement that no two global-ids could share a single transaction?

Transaction State

At the session/transaction-management layer in Mulgara, the transaction interfaces are strictly 2-phase. 3-phase support exists in the lower layers; within the resolvers themselves. The internal use of JTA interfaces within Mulgara has exposed access to that 3-phase support, however at this stage that support is passed off to JOTM (the third-party transaction manager used by Mulgara). Prior to the recent Transaction refactoring the assumption that JOTM would handle the mapping from 2-phase to 3-phase was threaded throughout the codebase. Since the refactoring it has become feasible to consider abandoning that assumption. It will however require breaking a foundational assumption of the existing transaction-management code in Mulgara. While not prohibitive this does mean additional effort is required to track the impact of this assumption and to ensure that the full ramifications of the change are handled safely.

There is also a similar issue to resolve to that of the lifecycle; in that there is an overlap between the JTA adaptors state change notification methods and the transaction state control methods of the Session Interface. Similar solutions present themselves, except that the ramifications of concurrent use of the methods is more serious. Specifically the external transaction manager's understanding of Mulgara's internal transaction state will be immediately invalidated if one of the existing methods on Session is used. Worse this could not only result in Isolation violations, but Consistency and Durability violations in the case of an out-of-band call to commit or rollback respectively.

Recovery

The JTA standards requirements here cannot reasonably be met until post-XA2. The specific requirement is the need to be able to commit a prepared transaction across an intervening restart. On restart Mulgara performs a heuristic rollback of any prepared but uncommitted transaction. To provide this ability with XA1 would require retrofitting the ability to persist a global-id to transaction state to disk on a prepare, and for this to subsequently be detected

and reactivated on restart. The amount of effort required to retrofit this capability onto XA1 is almost certainly prohibitive.

The consequence of this is that in the case of a system failure; if the failure occurred after another resource in the distributed transaction had successfully performed a commit, but before Mulgara had been able to write the single 4k block to disk (which is what a commit consists of in Mulgara); Mulgara would rollback its share of the transaction leaving the distributed transaction partially committed, and therefore inconsistent. Any other scenario would result in either a full, consistent commit, or a full consistent rollback. There are a couple of scenarios where a system failure may result in a rollback where it might otherwise have successfully committed, but while pessimistic that would still be correct and consistent behaviour. Only the scenario described above could result in incorrect, non-JTA compliant behaviour.

Partial Support Alternatives

The simplest JTA support alternative is to provide:

- JTA adaptor support for read/write transactions only
- Redefine existing methods in terms of calls to the JTA adaptor
- Rollback any existing transaction on transaction start
- Ignore recovery

The next alternative is to provide:

- JTA adaptor support for read/write transactions and faux read-only transactions
- Add code to prevent concurrent use of Session and JTA transaction control interfaces
- Consider an existing r/w transaction to be an error and abort transaction start
- Ignore recovery

The fullest feasible alternative is to provide:

- JTA adaptor support for read/write **and** read-only transactions.
- Redefine existing methods in terms of calls to the JTA adaptor
but also
- Add code to prevent concurrent use of Session and JTA transaction control interfaces
- Allow an existing r/w transaction to be unaffected by any r/o transactions being initiated
- Ignore recovery

Resourcing Estimates

The **simple alternative** would require probably no more than **2 weeks, 3 at most**. Most of this time would be ensuring the invariant conditions on the internal transaction manager were maintained.

Adding faux read-only transactions is a trivial addition, and probably doesn't change the 2-3 week estimate. Adding the additional safety code to ensure proper isolation is likely to require an additional week. So **the intermediate alternative** will probably require **3-4 weeks** effort.

Read-only transactions have been discussed on multiple occasions as a possible feature to add to Mulgara. They would probably take 2-3 weeks without any JTA involvement. With JTA this is likely to be 3-4 weeks. A bigger problem is that there isn't complete agreement within the mulgara development team as to the desirability of this feature, but the disagreements are mostly to do with the treatment of blank-nodes which have previously been the motivation for the feature. It is unlikely that there would be any problem gaining agreement on the availability of read-only transactions in the context of JTA support. So **the fullest feasible alternative** is likely to require **6-7 weeks**, but may also require a couple of weeks duration (not effort) to reach agreement on the features acceptance into Mulgara.