The Transaction Model for BaBar Objectivity Federations

Simon J. Patton

Introduction

BaBar stores its production data in a number of Objectivity Federations. Access to data in such federations is managed through the concept of transactions to maintain data integrity. While these transaction are transparent to physicists using the Framework for data processing, subdetector programmers who design the classes which manage the transfer of objects between a Federation and the Framework need to understand the design of the interface to the BaBar Federations.

This document is to outlines the model used to develop that interface, and specifies the requirements which can be derived from this model.

Simple Transaction Model

The simplest Transaction Model would be to start a transaction at the beginning of any job accessing a Federation and to close the transaction at the end of the job. There are a number of drawbacks with this model.

Database locks are managed within a transaction and are only released at the commit or abort of a transaction. This means that in the simple model other users could not gain write access to any container of the federation which has been used by a running job (the container being the smallest lockable unit in an Objectivity Federation). While the BaBar federation design attempts to minimize such access by allocating, where possible, containers to a single job, there are some significant containers to which have to be accessed by all production jobs if they are to function correctly. Therefore the simple model would effectively serialize a significant portion of the production jobs.

Moreover when a job crashes it does not close, or more correctly abort, its transaction. This means that any container of the federation touched by a production job...
The Transaction Model for BaBar Objectivity Federations

using the simple model would remain locked until the unfinished transaction was cleaned up either automatically or by administrator intervention. It also means that all the results of the job will be lost when the transaction is cleaned up. As production jobs can last up to two hours this can mean a significant loss of data when such a job dies, especially late in the run.

The addition of the Objectivity feature which allows access to multiple federations within a single job breaks the simple model as well. In the present release of Objectivity each a transaction operates within the ooContext which was current when it was opened. Therefore if our access model requires more than one federation to be open at a time, which is certainly allowed by our present execution model, more than one transaction must be present in a job.

**Event Store Transaction Model**

To overcome the limitations of the "Simple Transaction Model" outlined above the transaction model used by the Event Store must allow for multiple starts, commits and aborts of any transaction associated with the Event Store. These actions are organized to minimize competition between multiple jobs for the same containers.

The execution model for the Event Store is the following.
1. Open Collection.
2. For each Event in the Collection:
   - process the event.
   - If output is required, create output Event (The output Collection is created the first time this step is executed.)
3. While there are more Collections, return to step 1.

During production each executing job is allocated its own set of containers into which it can write its output. This avoids different jobs competing for the same container. However, navigation to a particular Collection object, either input or output, is through a set of treeNode objects. These objects are organized in the form of a 'directory tree'. The result of this organization is that any job needs to access, and possibly modify, treeNode which may be required by competing jobs. (This could be a few as one treeNode, the 'root' node, or as many nodes as there are in the collection name.) This means that locks on those containers containing the common treeNode can hold up any other job wishing to modify these nodes. For this reason, immediately after a collection object has been located, the Event Store transaction should be committed so that any locks on containers required by other jobs are released.

Once any input collections have been located, the Event processing begins. If Events are being read for an Objectivity federation then a new transaction must be started and remain available while the 'current' event is being processed, as data could be retrieved from the Event Store at any time. In principle once processing of the Event has finished the transaction could be committed thus releasing any containers it has touched. However once a job is reading an Event it is not expected that any other job

---

1. Version 6.0
Non-Event Store Transactions

will require write access to the containers which hold that Events information\(^1\). Therefore there is not pressing reason to close the transaction after each Event has been read.

Similarly there is not expected to be competition for containers for the output Events as jobs are normally organized to be accessing different containers. There is, though, an issue with the *treeNodes* which must be shared between jobs to allow for the creation of the output Collection. Therefore once the output Collection has been created the transaction should be committed, as before, so that any locks on containers required by other jobs are released.

While other jobs may not be competing for a jobs containers there is a good reason to periodically commit the transaction associated with writing Events to a Federation, namely to avoid the lose of preceding Events processed by the jobs, should the jobs crash. Indeed, our production jobs set up in such a way that input Events are buffered until they have been committed to the output Federation. Therefore the output transaction should be committed periodically. The exact period should be chosen to balance the Event processing rate against the possible Event loss rate\(^2\).

With multiple federation now becoming available there is no reason for the input Events to exist in the same federation as the output Events. Indeed there are certain cases where it is preferable to use two different federation, e.g. creating a small subset for a detailed analysis. Therefore, as an *ooContext* can only operate with a single federation, there needs to be two *ooContexts*, with matching transactions, associated with the Event Store.

### Non-Event Store Transactions

The Event Store Transaction Model ignores some other activities which effect transactions. These other activities break down into two basic types: housekeeping, and non-Event Store Domains.

#### HOUSEKEEPING

The housekeeping category covers such tasks as database and container creation, and the access to various bookkeeping objects which are required for federation-wide cooperation. This category of activities can be summarized as being those which access objects common to more than single job, with the exception of *treeNode* access which has already been covered in “Event Store Transaction Model”.

Creation of both databases and containers require write locks on their containing entities, i.e. the federation and the database respectively. Clearly these locks will effect other jobs wishing to access these entities and therefore the transactions in which these tasks are done should be a short as possible. As this requirement is significantly different for the transaction requirements laid out about for the Event Store

---

1. In principle, with container reuse, this could occur in the production federations, but once that data has been swept in the analysis federations this can not happen as the databases are attached in read-only mode.
2. Consideration of how long it takes for a federation to be successfully inhibited should also be a consideration here. See “Transaction and *ooContext* Coordination” for more information about inhibiting federations.
it is preferable to carry out these activities in a separate, short, transaction. This con-
cept is known as a mini-transaction.

The requirements for a mini-transaction is that it should start a new transaction on
the ‘current’ federation, execute the set of tasks needed to complete the activity, and
commit the transaction before returning control to the rest of the job. This approach
means that the other parts of the job do not have to worry about transaction manage-
ment as is it the transaction state of the job is that same when the mini-transaction is
completed as it was when it was started.

The two main groups of bookkeeping objects which are required for federation-wide
cooperation are those which keep track of the usage of databases and containers, and
those which describe the authorization policies of the federation. These objects
clearly must be available to any job wishing to interrogate them or update them. This,
again, means that access to these objects require separate, short, transaction to be
efficient, and so this too is done by means of mini-transactions.

NON-EVENT STORE
DOMAINS

The BaBar federations are broken up into sets of self-contained objects, which are
called Domains. Each Domain is conceptually independent of the other Domains
and therefore administrative activities such as security, transaction management and
the API used to assess the Domain can be developed, at least in principle, without
regard to the other Domains. The Event Store is designed as a Domain as access to
its contents is independent of the other Domains (listed below). These other
Domains may be required to create meaningful information for the physicist, but
they are not necessary for simple access to the persistent objects stored in the Event
Store.

The non-Event Domains are:

• Conditions - contains calibration data, either specified, derived or the results of
  “rolling calibrations”.
• Configuration - contains the various configurations of the detector.
• Ambient - contains recorded data about the conditions during data taking.
• Spatial - contains the “rolling calibration” results for a single node processing
  part of a run.
• Temporal - contains the combined “rolling calibration” results for a complete
  run.

Given that the Event Store is one of many Domains in the BaBar software, why is its
Transaction Model dealt with separately, and ahead of, from the other Domains? The
reason is that the execution model of BaBar jobs is basically a straightforward copy
of the one used by the Event Store. The other Domains execution models do not
influence the overall model.

As each Domain is supposed to be able to operate independently of any other
Domain, this means that each Domain needs to be able to manage its own transac-
tions. As transactions cannot be nested or overlapped [page 2-2, Using Objectivity/
C++] within a ooContext, this means that each Domain must also be able to man-
ge any ooContexts it needs to complete its duties.

Furthermore, as all Domains are designed to be self-contained they can, in principle,
eexist in independent federations within a the context of accessing multiple federa-
tions in a single job. This, too, requires that each Domain has the ability to create any
necessary ooContexts as a ooContext object can only operate with a single federation.

Therefore all Domains should manage their own ooContexts and associated transactions to maintain their independence from the other Domains.

**Transaction and ooContext Coordination**

As each Domain is in charge of its own ooContexts and associated transactions this means that any job may have any number of transaction active at any point in time. One question which arises of this is “Whether there are any circumstances where these disparate transaction and ooContexts need to be coordinated?” The answer to this question is “Yes.” It turns out that there are a couple of circumstances that require coordination; when we wish to “inhibit” a federation, and managing an applications (or threads) current ooContext.

**INHIBITING FEDERATIONS**

“Inhibiting” a federation mean that we wish to stop all transactions against that federation so that we can modify the federation without worrying about competing for locks or causing jobs to use stale files. To achieve this each federation has a flag which specifies whether a transaction should be opened against that federation, or whether that transaction should be postponed until work on the federation is complete and the flag cleared.

The need for coordination arises from the use case where one or more transactions are active and another transaction, which is attempting to start, detects that its federation has been inhibited. In this case postponing the start of the transaction until its target federation is no longer inhibited would not be good as the other active transaction would not be able to commit (or abort!) in a timely manner. The effect of this action would be that those federations against which there were active transactions could not be inhibited in a timely manner because new transactions can not start (and detect that their federation is inhibited) until the existing ones had completed.

To be able to successfully inhibit one or more federations we therefore need some coordination between transactions. It has already been stated that the idea of inhibiting a federation is to postpone a transaction from begin started on that federation, but if one or more other transactions in the job are already active the requested start should not be postponed. The corollary of this is that the only time we can postpone the start of a transaction is when there are no other transactions active. This implies that we need to some coordination at the start and at the end of transactions.

The required behavior of a job is therefore that when any of the federations it may access is inhibited the job should pause as soon as practical and resume when none of its possible federations are inhibited.

The requirement that a job “should pause as soon as practical” paired with the fact that the pause can only happen when there are no active transactions, means that a job must have times where there are no active transactions. This clearly means that there must be a predetermined points during execution where all transactions are ended at the same time. The natural time to execute such a task is at the same time the Event Store commits its write transaction, or a similar time if there is no write transaction. As noted above this period is already based upon the Event processing
rate and the possible Event loss rate and now “to allow for timely inhibition of jobs” should be added to the criteria used to set this period.

As noted earlier in this section we can only postpone the start of a transaction, and thus pause the job, when there are no other transactions active. Therefore any attempt to start a transaction should first test if any other transaction are already active. If any other transaction is active then the attempt should be allowed to continue. If no other transactions are active then check whether any of the federations, that the job may access, are inhibited. If none are inhibited the attempt may continue, but if any is inhibited the attempt should be postponed and only allowed to commence when all possible federations are not inhibited.

A key part of the preceding outline is “check whether any of the federations, that the job may access, are inhibited”. This requires a job to be able to predict its entire future which would be hard, to say the least. We can make the problem tractable by modifying this requirement to be “check whether any of the federations, that the job may access, as given by the current state of any ooContexts managed by domains, are inhibited.” and adding a new requirement that “an ooContext can only switch the federation with which is associated when there are no active transactions”. The result of these two requirements is that there will always be a check for inhibited federations whenever there is change in the set of federations a job might access.

Therefore, to be able to successfully inhibit a job when it discovers an inhibited federation, our transaction coordination requires the following behaviors:

- An ability to end, either by commitment or aborting, all transactions at the same time.
- An ability to check to see if there are any other active transactions.
- An ability to check whether any of the federations, that the job may access, as given by the current state of any ooContexts managed by domains, are inhibited.
- An ooContext can only switch the federation with which is associated when there are no active transactions.

MANAGING THE CURRENT ooContext

An application (or more correctly any thread) can only have one ooContext active at any time, therefore if more than one Domain is present in a job, and thus more than one ooContext, there needs to be coordination between the Domains as to which Domain’s ooContext is the “current” one, i.e. the one against which federation access is performed. It is impractical to require that the correct context be set before each and every function which makes use of an ooContext. The solution chosen at BaBar was to build on an existing model which had been used when all Domains used the same main ooContext, and thus only has access to a single transaction.

In the original, single transaction, architecture access to non-Event Store Domains could not assume the state of the current transaction. Therefore any attempt at non-Event Store access is required to make certain that the transaction state when it fin-

---

1. The transactions discussed here do not include mini-transactions as they are designed to be completed within the routine they are started and thus should always be short.

2. Any Domain which manages more than one ooContext is responsible for deciding which is should be ‘current’.
ishes is the same as that when it started. This behavior was encapsulated in two member functions of BdbCondProxyBase, startTransaction and endTransaction. The functions could then be used by subclasses of BdbCondProxyBase to emulate the start and end of transactions while preserving the state of the transaction after access to the Conditions Domain.

This model works in the single transaction architecture because non-Event Store activity can be considered an “blocking request” with respect to the main Event Store activity and so the Event Store will not be using its transaction while there is non-Event Store activity. This behavior remains true even when each Domain manages its own transactions of ooContexts. Therefore it is possible to modify the original model so that non-Event Store access is required to make certain that the ooContext when it finishes is the same as that when it started.

This modification to the behavior of non-Event Store Domains, along with the fact that each Domain is in charge of its own transaction, means that the original requirement of preserving the transaction state is no longer absolutely necessary. However as non-Event Store access is less coordinated than Event Store access (for example all Event Store transaction management is done in ‘core’ code, not ‘client’ code) and to preserve the option of still being able to run using a single transaction it is still necessary for non-Event Store to conform to this requirement.

Therefore, to be able to successfully run a job which can only have one ooContext active at any time, and preserve the model of only having a single transaction for all Domains, we require the following behaviors:

- An ability to state when a Domain is about to make active use of its ooContext.
- An ability to state when a Domain is no longer going to use of its ooContext.
- Non-Event Store access is required to makes certain that the transaction state when it finishes is the same as that when it started.

Summary

The Transaction Model presented in this document requires that each Domain manage, i.e. start, commit and abort, its own transactions and associated ooContexts.

Housekeeping activities, which access objects common to more than one job, take place in mini-transaction which are of limited duration.

The need to be able to inhibit jobs when any of the federations they may access are undergoing certain modifications, and the fact that only one ooContext may be active at any time, means that there is a need for some coordination between the transactions and between the ooContexts of all Domains in a job.

The coordination between transactions includes being able to end all transaction and the same time and check whether there are any active transactions. While the coordination between ooContexts includes being all to test all associated federations to see if they are inhibited, to only allow an ooContext to change its associated federation when there are no active transactions, to be able to specify when a Domain is going to start using its ooContext and when it has finished.
To maintain compatibility with a single transaction model Non-Event Store access is required to make certain that the transaction state when it finishes is the same as that when it started.