WS-BA Coordinator/Participant Recovery Overview

Table of Contents

WS-BA Participant Side APIs

WS-BA has two generic participant interfaces

```
interface BusinessAgreementWithParticipantCompletionParticipant
{
    public void close () throws WrongStateException, SystemException;
    public void cancel () throws WrongStateException, SystemException;
    public void compensate ()
      throws FaultedException, WrongStateException, SystemException;
    public String status () throws SystemException;
    public void unknown () throws SystemException;
    public void error () throws SystemException;
}
```

and

```
interface BusinessAgreementWithCoordinatorCompletionParticipant
    extends BusinessAgreementWithParticipantCompletionParticipant
{
    public void complete () throws WrongStateException, SystemException;
}
```

Each of these two interfaces is employed by both the 1.0 and 1.1 implementations. They are both located in package `com.arjuna.wst`.

A web service which wishes to employ WS-BA to manage transactional resources must define classes implementing these interfaces. It can then register instances of these classes as participants using the enlistment APIs associated with the WS-BA business activity manager class. The enlistment method used should correspond to the implemented interface.

The two implementations differ only in how the participant moves from state active to state completed. `BusinessAgreementWithCoordinatorCompletionParticipant` is employed where the service is responsible for initiating this transition. For example, this would be appropriate in a situation where the service exposed an API to the client such as

```
@WebService class MyService
{
  @WebMethod String placeOrder(String productCode, int count);
  @WebMethod void cancelOrder(String orderId);
  @WebMethod void completeOrder();
}
```

The client might call `placeOrder` one or more times to make provisional orders and might possibly call `cancelOrder` to delete any existing orders. Once `completeOrder` is invoked the service would not be able to perform any more work within the scope of the business activity. At this point it can perform any work required to persist i) the changes it has made during the activity and ii) the data required in order to ensure that it can compensate the changes in the event that the activity is cancelled. It should then notify the coordinator that it has completed using the participant manager API (see below). If all orders had been cancelled when `completeOrder` was called it might instead decide to exit the business activity using the participant manager API.

If instead the web service API omitted method `completeOrder` then there would be no way for it to know that all the required orders had been placed and were not going to be cancelled subsequently. In this case the service would have to register using `CoordinatorCompletion`. The coordinator ensures that the `completed` method is called when the client notifies completion of the whole business activity. The participant should still do the same work as if it had initiated the completion process i.e. persist i) the changes it has made during the activity and ii) the data required in order to ensure that it can compensate the changes in the event that the activity is

cancelled.

The `close` method is called when a client `Close` request is sent to the coordinator and all the participants have completed successfully. The client responds to a Close request by deleting any compensation state associated with the business activity, effectively forgetting the activity. This method can only be called after the participant has persisted all changes associated with the business activity so it represents a successful outcome.

The `cancel` method is called when a client `Cancel` request is sent to the coordinator and the participant has not yet completed. In this case the client should respond by deleting any record of uncommitted changes associated with the business activity. This method will not be called once a `ParticipantCompletion` participant has called (or, rather, returned from calling) the `completed` method. It will not be called once a `CoordinatorCompletion` participant has had its `completed` method invoked. The `cancel` method may also be called if the coordinator crashes.

The `compensate` method is called when a client `Cancel` request is sent to the coordinator and the participant has already completed. In this case the client responds by deleting any record of uncommitted changes associated with the business activity. This method can only be called after the participant has persisted any changes associated with the business activity i.e. for a `ParticipantCompletion` participant once it has called (returned from calling?) its `completed` method and for a `CoordinatorCompletion` participant once it has had its `completed` method invoked. The `compensate` method may also be called if the coordinator crashes.

`status` is called during top-down recovery to determine what? whether the participant has completed or not? after all, there is a window at the active --> completed transition which we need to close.

`error` is called during bottom-up recovery to notify the participant of what? that the last state transition got through and it missed the reply?

WS-BA Coordinator Side Implementations

The two Participant interfaces described above are not just implemented by the web service application code. They are also implemented by the BA coordinator stub classes. Stub instances are created on the coordinator side when a BA participant is registered and installed in the coordinator's participant list. They are used to forward operations invoked by the coordinator across the wire to the registered participants.

```
BusinessAgreementWithCoordinatorCompletionStub implements
  BusinessAgreementWithCoordinatorCompletionParticipant,
PersistableParticipant
BusinessAgreementWithParticipantCompletionStub implements
  BusinessAgreementWithParticipantCompletionParticipant,
PersistableParticipant
```

The stub wraps a coordinator engine instance which actually dispatches the messages

```
CoordinatorCompletionCoordinatorEngine implements
  CoordinatorCompletionCoordinatorInboundEvents
ParticipantCompletionCoordinatorEngine implements
  ParticipantCompletionCoordinatorInboundEvents
```

The coordinator engine also implements an inbound events interface which allows it to serve as a focus for messages dispatched from the participant. By routing incoming and outgoing messages through this instance it is possible to ensure that all state changes in the participant are correctly synchronized and ordered. The inbound events interface ensures that the engine only exposes methods which correspond 1-1 with the Participant-Coordinator message protocol. Active

coordinator engines are stored in a lookup table keyed by participant id. This enables the Coordinator service to lookup the target engine for a message and dispatch the message.

WS-BA Participant Side Processing

Incoming messages received from the coordinator are handled by a participant engine which wraps the participant supplied by the web service in the enlist call. These messages are forwarded to the participant by invoking the methods defined in the relevant participant interface. The engine implementations come in two flavours

```
CoordinatorCompletionParticipantEngine implements
  CoordinatorCompletionParticipantInboundEvents
ParticipantCompletionParticipantEngine implements
  ParticipantCompletionParticipantInboundEvents
```

The participant inbound events interfaces ensure that the exposed operations correspond 1-1 with the messages defined in the BA Coordinator-Participant message protocol. Active participant engines are stored in a lookup table keyed by participant id. This enables the Participant service to lookup the target engine for a message and dispatch the message.

The engine instances also serve as a focus for outgoing requests from the web service to the coordinator invoked via the `BAParticipantManager` interface. By routing incoming and outgoing messages through the engine it is possible to ensure that all state changes in the participant are correctly synchronized and ordered.

When participant engines are created at enlist they are wrapped by an instance of a participant manager stub class which implements the participant manager interface

```
BACoordinatorCompletionParticipantManagerStub implements
BAParticipantManager
BAParticipantCompletionParticipantManagerStub implements
BAParticipantManager
```

`BAParticipantManager` provides methods allowing the participant to notify changes in its state to the coordinator

```
interface BAParticipantManager
{
  public void completed()
    throws WrongStateException, UnknownTransactionException,
SystemException;
  public void exit()
    throws WrongStateException, UnknownTransactionException,
SystemException;
  public void cannotComplete()
    throws WrongStateException, UnknownTransactionException,
SystemException;
  public void fail(final QName exceptionIdentifier) throws
SystemException;
}
```

n.b. `cannotComplete` is new with 1.1 and does not appear in 1.0. `fail` is called `fault` in 1.0.

If the web service invokes a method of the manager stub instance it is forwarded to the coordinator by invoking the corresponding method of the engine instance.

WS-BA Coordinator Side Processing

On the XTS coordinator side information about participants is retained using instances of the two

participant stub classes

`BusinessAgreementWithCoordinatorCompletionStub`
`BusinessAgreementWithParticipantCompletionStub`

The participant stubs are not directly referenced from the BA coordinator which drives the business activity. The coordinator class is `ACCoordinator` derived from classes `BasicAction` and `TwoPhaseCoordinator`. It wraps the stubs in `ParticipantRecord` instances which may be installed in the pending, prepared, failed and heuristic participant lists inherited from class `BasicAction`.

The BA `ACCoordinator` class maps most of the BA coordination process onto the standard 2PC operations implemented by classes `TwoPhaseCoordinator` and `BasicAction`. It does so in part by adding extra behaviour in class `ACCoordinator` and in part by mapping the methods of class `ParticipantRecord` invoked by `BasicAction` to behaviours appropriate to the BA protocol. However, this requires a certain amount of stretching of the semantics of classes.

Most of the operations of the coordinator are driven by requests dispatched from the client to the coordinator in accordance with the `BusinessActivityTerminator` message protocol. This is not a part of the BA spec but is critical to the operation of the BA protocol. The client application code does not see this protocol because it is wrapped up inside the implementation of the `UserBusinessActivity` class. The protocol dispatches `complete`, `close` and `cancel` requests to the coordinator in response to invocations of the corresponding methods of `UserBusinessActivitry` and notifies the client of any faults detected during processing of the request.

The `ACCoordinator` provides method `complete` which is used to implement coordinator driven completion in response to a Complete message from the `BusinessActivityTerminator`. It invokes the complete method of each `ParticipantRecord` which uses the participant stub to dispatch a complete message to each non-completed participant registered for coordinator completion. Any failures are recorded by marking the action as `ABORT_ONLY` and result in an exception being sent back to the client.

The `ACCoordinator` is not invoked directly when a `close` or `cancel` message is dispatched by the `BusinessActivityTerminator`. The `UserActivity` instance identified by the business activity id is terminated by calling `end` with either a success or failure status code supplied as argument. This eventually leads to the `end` or `cancel` methods of `TwoPhaseCoordinator` being invoked and these ultimately lead to invocations of `BasicAction.End` or `BasicAction.Abort`, depending upon the current state of the activity.

The prepare stage of the BA coordinator two phase commit is normally an empty step because the BA participant records in the `BasicAction` participant list always return a successful outcome in response to a call to `topLevelPrepare`. However, this step is not totally redundant. Firstly, invocation of the `prepare` operation changes the state of the `BasicAction` from `ACTIVE` to `PREPARING`, invalidating further attempts to add participants to the activity. Secondly, on completion of the prepare phase details of all completed (and non-exited) participants will be saved in the coordinator record written to the transaction log.

The BA participant records implement method `topLevelCommit` by dispatching a `close` message via the coordinator engine. Method `toplevelAbort` is implemented by sending a `cancel` message if the participant has not completed and a `compensate` message if it has completed. So, these implementations propagate the `close` or `cancel` request through to all participants by hijacking the implementations of `BasicAction.End` and `BasicAction.Abort`.

Note that a BA `ParticipantRecord` will return a `NOT_PREPARED` result from `topLevelCommit` if the resource has not exited and is not in state `COMPLETED`. This allows the `close` operation to continue. It results in a heuristic decision being recorded for the transaction. but this is thrown

away if no other type of error occurs.

Hmm, does this ensure that all outcomes are consistent, though? Should a `close` not check that all participants have completed? If they are not it could i) return an exception to the client ii) wait for the participant to complete (including, possibly, logging a heuristic TX and then closing it later) or iii) force a cancel i.e. rollback the activity.

Ok, let's vote for throwing an exception in the client. After all the participant's completed message is unacknowledged so it cannot be guaranteed to have arrived before the client attempts a close no matter when the participant dispatched it. This would allow the client to retry for as long as it wants or give up and cancel instead.

Note that at present the BA `ParticipantRecord` will return a `SystemError` if dispatch of a `close` or `cancel` message times out, resulting in a heuristic outcome for the transaction.

The BA `ACCoordinator` also implements methods `participantCompleted`, `participantFaulted` and `delistParticipant` which are invoked in response to messages dispatched via the participant manager methods `completed`, `fail` and `cannotComplete`/`exit`.

`participantCompleted` marks a participant record as completed, ensuring that it performs the correct action if it's `toplevelAbort` method is called. It should only ever be invoked with a participant id which identifies a participant registered for `ParticipantCompletion`.

`participantFaulted` and `delistParticipant` mark a participant record as exited, ensuring that it is not involved in any further completion processing.

WS-BA Coordinator Side Recovery

In order to be able to guarantee coordinator side recovery the WS-BA implementation could perform all of the following log operations:

1. write or update a log record of all participant stubs when a participant is enlisted
2. write or update a log record of all participant stubs when a participant notifies that it has completed
3. write or update a log record of all participant stubs when a client `Complete` message is received, rewriting it after each participant `Complete` message is sent and `Completed` message is returned (the participant states should change from `ACTIVE` to `COMPLETING` to `COMPLETED` before each write).
4. write or update the log record of all non-terminated participant stubs when a client `Close` or `Cancel` message is received, rewriting it after each `close`/`cancel`/`compensate` message is dispatched and is subsequently confirmed (the participant states should change from `COMPLETED` to `CLOSING`/`CANCELLING` before each write)
5. delete the log record of participant stubs after successfully handling a client `Close` or `Cancel` message
6. update the log record if any one of the participants returns a `Fail` message or times out responding to a `Close`, `Cancel` or `Compensate` message. in the former case the log record will contain heuristic participants, in the latter failed participants.
7. identify logged participant stub details during recovery processing and recreate and activate participant stubs from the saved details

It is possible to elide most of these steps if a 'presumed abort' protocol is adopted. In this situation it is assumed that participants resend messages to the coordinator in order to identify progress of the transaction.

Steps 1 and 2 can be elided into step 3. If the coordinator crashes before a client `Complete` message has been sent then after coordinator restart a client `Complete`, `Close` or `Cancel` message will be rejected with an `UnknownTransaction` exception thrown in the client.

Things are not quite so simple as far as participants are concened. A send or resend of `Completed` by a `ParticipantCompletion` participant will be ignored (that is as per both the spec and the current implementation). For `CoordinatorCompletion` participants there is no `Completed` message for them to resend so in either case the participant will not automatically detect that the business activity is no longer active. However, participants can still initiate departure from the transaction via the participant manager API, posting either a `CannotComplete`, an `Exit` or a `Fail` message (which one?). After a restart the coordinator will fail safe by acknowledging these messages. So, for presumed abort to work correctly the participant web service must implement some sort of timeout mechanism for presumed abort to work successfully.

The XTS participant side implementation cannot really drive this timeout process itself, even assuming that the BA context specified a timeout value. The correct behaviour for a timeout initiated from the participant side is to dispatch a `CannotComplete`, an `Exit` or a `Fail` message so that the coordinator is notified and can acknowledge the termination request. If the XTS implementation performed this action autonomously, say by posting a `Fail` message, then the registered participant would not be aware of the termination attempt nor its outcome. The XTS participant implementation could perhaps remedy this by calling the registered participant's `cancel` method before posting the termination request or after it has been acknowledged but this still leaves a window open between sending the message to the coordinator and calling cancel where the coordinator, XTS participant and registered partiicpant's view of the participant state are different.

Step 3 can be simplified to write the participant list when the client `Complete` message has been successfully handled. The participant list will identify all `COMPLETED` participants and indicate that the transaction has `COMPLETED` i.e. that it is still pending a client `Close` or `Compensate` message. If the coordinator crashes between handling `Complete` and a subsequent client `Close` or `Cancel` then it can reinstate the transaction, reactivate the participants and resend the participant `Complete` messages. This risks resending a message to some participants. The XTS implementation on the participant side must ensure that the registered participant's `complete` method is only called once.

Step 4 can be simplified merely to to rewrite the participant list once when the client `Close` or `Cancel` message is received before proceeding to handle it. The list will also indicate whether the transaction is `CLOSING` or `CANCELLING`, allowing the appropriate action to be taken after crash recovery. If the coordinator crashes before Step 5 or 6 then it can reinstate the transaction, reactivate the participants and resend the participant `Complete`, `Cancel` or `Compensate` messages. This risks resending a message to some participants. The XTS implementation on the participant side must ensure that the registered participant's `close`, `cancel` or `compensate` methods are only called once.

Steps 5 and 6 are both necessary and cannot be simplified. Step 6 is only appropriate where a failure occurs during handling of a client `Close` or `Cancel` request. In the case where `Fail` responses result in a heuristic outcome the log record will be rewritten with state `CLOSED` or `CANCELLED` and will list all heuristic participants. In the case where a communications failure occurs and the participant does not confirm a `Close`, `Cancel` or `Compensate` message the log record will be rewritten with state `CLOSING` or `CANCELLING` and will list all failed participants (and possibly some heuristic ones). In this case the periodic recovery thread will reinstate the

coordinator and reactivate the participants after a suitable delay and retry the `Close` or `Cancel` operation for all failed participants. Depending upon the outcome it will proceed to step 5 or 6.

Step 7 could employ calls to `getStatus` to establish the true state of participants. However, if the participants adopt the measures described above there should be no need to use this mechanism.

So the real requirement is that the implementation must:

1. write a log record of all completed participant stubs after performing a successful `complete` indicating that the transaction is `COMPLETING`

2. rewrite the log record of all completed participant stubs before performing a `close` or `cancel` indicating whether the transaction is `CLOSING` or `CANCELLING`

3. rewrite the log record with any of the participants which timeout before responding to a `Close` or `Compensate` message – they will be listed as failed participants in this case

4. rewrite the log record with any of the participants which respond to a `Close` or `Compensate` message with a `Fail` message – they will be listed as heuristic participants in this case

5. delete the log record of participant stubs if all participants successfully responded to a `Close` or `Cancel` message

6. identify logged participant stub details during crash recovery and resend `Close` or `Compensate` messages as appropriate

Note that it is possible that a client `Complete` or `Close` message may arrive before some participant completion participants have sent a `Completed` message. In the case of a client `Complete` message this could be regarded as an error. Alternatively, it could be ignored on the assumption that the participant will complete at a later date. In the case of a client `Close` message this is clearly an error. There is a choice of how to handle these cases. One option is to throw a system exception which will be communicated back to the client. Another is to cancel the activity, throwing a rollback exception from the client call. A third option is to delay the client `Complete` or `Close` request until all the participant `Completed` messages arrive. A final option is to perform the requested operation, sending `Complete` or `Close` requests. Note that in the case of a client `Close` request the last option would require the participant side implementation to break the BA specification. The `ParticipantCompletion` participant stub would have to be willing to accept a `Close` message and to call the enlisted participant's `close` method while in state `ACTIVE`.

The current XTS implementation will regard both these cases as errors and will handle them by rejecting the client message, throwing an exception from the client call. At a later date the implementation may extend the client (Termination) protocol to allow alternative behaviour. For example, the client may be provided with a method to check that all participants have completed or it may be able to send a variant of the `Complete` or `Close` messages which reverts to a `Cancel` if there are uncompleted participants.

WS-BA Coordinator Side Recovery State Processing

Saving, restoring and deletion of participant stub recovery state is managed by the WS-BA `ACCoordinator` class using the participant list management capabilities of class `TwoPhaseCoordinator`. Participant stubs are referenced from `ParticipantRecord` instances located in the coordinators participant list. At complete the details or each active stub are converted to a byte format and composed to construct the coordinator's saved state, each sub-

entry tagged with type `XTS_RECORD`, the type associated with the `ParticipantRecord` class. The combined transaction state is then saved to the TX object store in a location associated with the WS-BA `ACCoordinator` class.

During normal operation the XTS implementation maintains an active WS-BA participant stub map, an in-memory map from WS-BA participant ids to participant stubs. This is used to identify the primary target for incoming participant messages: `Fail`, `Exit`, `CannotComplete`, `Completed`, `Cancelled`, `Closed` and `Compensated`. The map is updated during enlist, complete and close/cancel processing to reflect the presence of active participant stubs and saved/deleted WS-BA Participant stub records in the TX object store.

During bootstrap the `ACCoordinatorRecoveryModule` recovery module recreates XTS WS-BA participant stub records as it scans `ACCoordinator` entries in the TX object store. Each WS-BA stub record is entered in the active participant map as it is recreated. So, once the first recovery scan is completed all active WS-BA participants are entered in the map. This point can be detected using a flag which defaults to false and is set to true after the first scan is completed.

Up to this point incoming participant messages may legitimately employ a participant id which is not found in the WS-BA active participant map. In such a case incoming messages which do not identify an entry in the map will be silently dropped to ensure that a valid response is provided when recovery is ready. After this point it is assumed that the message has been resent and processing follows the transition tables in the WS-BA spec.

WS-BA Participant Side Recovery

In order to be able to perform participant side recovery the XTS implementation must:

- save details of participants when they transition to state completed (before sending a `Completed` message if completion is participant driven, before sending a `Completed` response if this is coordinator driven)
- delete saved details of participants when they are either closed or compensated (after calling the enlisted web service participant's `close`/`compensate` method)
- delete saved details of participants when they fail during compensate (after confirmation of the `Fail` message by a `Failed` response -- n.b. this requires a `fail` call to pre-empt any compensate call which is trying to delete the record).
- identify saved participant details during crash recovery and recreate a participant from the saved details (this requires a helper to recreate the web service's participant -- does it also require identifying the participant's notion of what state it is in to close all the windows in the actions listed above?)

Saving, restoring and deletion of participant data, including the application-specific recovery state, is managed by the WS-BA implementation. However, recreation of saved participants during recovery involves creating and initialising instances of application-specific classes using the saved recovery state. Since this requires loading application-specific classes it must be done by application code. Hence, it will be necessary for the XTS implementation to provide a registration mechanism allowing applications to provide a recovery module to perform this step in the recovery process.

Recovery modules must be able to recognise that saved participant details belong to their associated application rather than some other application. This will be achieved by requiring participants to employ identifiers which are unique to their application (as well as unique within all participants created by that application). The id employed when the participant is registered will be used for this purpose.

A participant must support saving of its recovery state by implementing a method which encodes the recovery state as a byte array. At `complete`, this byte array will be obtained and written to disk by the WS-BA implementation, along with the participant identifier and the endpoint of the participant's coordinator. This participant recovery record will be deleted at the appropriate point during `close` or `compensate`. During recovery processing saved recovery records for outstanding transactions will be identified and reloaded. The byte array saved in the record will be used to reconstruct a web service participant and re-register it as an active participant with the participant service using the saved identifier and endpoint.

A recovery module must implement a method which discriminates amongst candidate identifiers. For those which are associated with its application, it will be required to construct a participant from the saved recovery state byte array. If the recovery module does not recognise a participant id then it will be assumed to belong to some other application.

Participant Recovery API

Participants are normally expected to implement the save state functionality by implementing interface `Serializable`. In such cases participants will be automatically serialized to a byte stream using Java serialization and the resulting byte data is stored in the recovery record. Specifically, for a participant completion participant this happens immediately after the call to the participant manager's completed method whereas for a coordinator completion participant this happens immediately upon return from the participant's completed method. The stream will be written by means of a single call to `ObjectOuputStream.writeObject` with the participant as argument. Alternatively, if the participant does not implement `Serializable` it can implement the following interface in package `com.arjuna.wst`:

```
public interface PersistableBAParticipant
{
    byte[] getRecoveryState() throws Exception;
}
```

In this case, `getRecoveryState` will be invoked at the points defined above and the returned byte sequence will be written by the BA implementation.

One or other of these interfaces must be implemented, otherwise the complete operation on the participant will fail and, ultimately, the participant will be cancelled.

Participant Recovery Module API

A recovery module must implement the following interface in package `org.jboss.xts.recovery`:

```
public interface XTSBARecoveryModule
{
    public BusinessAgreementWithParticipantCompletionParticipant
    deserializeParticipantCompletionParticipant(String id,
ObjectInputStream stream) throws Exception;

    public BusinessAgreementWithParticipantCompletionParticipant
    recreateParticipantCompletionParticipant(String id, byte[]
recoveryState) throws Exception;

    public BusinessAgreementWithCoordinatorCompletionParticipant
    deserializeCoordinatorCompletionParticipant(String id,
ObjectInputStream stream) throws Exception;

    public BusinessAgreementWithCoordinatorCompletionParticipant
    recreateCoordinatorCompletionParticipant(String id, byte[]
recoveryState) throws Exception;
}
```

If a participant was saved using serialization then one of the `deserialize` methods will be called to allow the module a chance to deserialize it. If a participant was saved by invoking the `getRecoveryState` method of `PersistableBAParticipant` then one of the `recreate` methods will be called to recreate it. The BA implementation records which type of completion protocol the participant was registered under so it knows which of the two possible `deserialize` or `recreate` methods is appropriate for the data saved in the record. Note that for any given recovery module all four of these methods may be called since the recovery module may be offered the opportunity to recreate participants which belong to other applications.

> `id` is the identifier under which the participant was registered. The recovery module should only attempt to reconstruct a participant if it recognizes the id.
>
> `stream` is a stream from which the application can read its participant by invoking method `readObject` of class `ObjectInputStream`.
>
> `recoveryState` is a byte array containing data returned from a call to `getRecoveryState`.

Whichever method is called, if it returns `null` then it is assumed that the recovery record does not belong to the recovery module's application. If an exception is thrown then it is assumed that the record does belong to the module's application and a warning is logged. Although this may indicate that the recovery data has been corrupted the participant recovery record is not deleted. The recovery thread will retry the reconstruction operation when it next runs in case the error is a transient one which can be recovered from. In such cases participant records must be deleted from the log manually by an administrator.

An application must register its recovery module during application deployment and unregister it during undeployment using the following API in package `org.jboss.xts.recovery.participant.BA`:

```
public class XTSBARecoveryManager
{
    ...
    public void registerRecoveryModule(XTSBARecoveryModule module);
    public void unregisterRecoveryModule(XTSBARecoveryModule module);
    ...
}
```

WS-BA Participant Recovery State Processing

An XTS WS-BA participant recovery record is modelled in-memory by class `BAParticipantRecord`. Each record will contain details of a single participant: a `String` identifier; an `enum` field indicating whether the participant is registered fro participant ompletion or coordinator completion; an XML format `String` representation of the associated coordinator endpoint reference (including any reference parameters); and a `byte[]` participant recovery state. No subordinate state is required so no persistence record types need be defined by the participant recovery system.

Since the endpoint reference format differs between the 1.0 and 1.1 implementations and since recovery processing will differ depending upon the protocol in use it is necessary to implement this class in two flavours, one for the WS-BA 1.0 protocol and another for the 1.1 protocol.

Class `BAParticipantRecord` implements the `PersistableParticipant` interface enabling the standard TX object state `read_committed` and `write_committed` operations to be used for creation, retrieval and deletion of the object store entry containing the record. Where appropriate delegation to methods on the 1.0 or 1.1 subclasses is used to generate the required disk data representation. The saved state includes details of the implementation class so that on recovery a valid in-memory version of the record can be reconstructed.

XTS WS-BA participant records are saved as top-level TX store entries, stored in a location in the TX object store defined by the XTS WS-BA participant record type (akin to the `ACCoordinator` type used on the coordinator side to store XTS WS-BA coordinator (transaction) state). WS-BA participant recovery records differ from other object store entries in that they are not derived from `AbstractRecord` nor from `StateManager`. The former class is only appropriate where the record is capable of being driven through prepare, commit and/or abort by a `BasicAction`, which is only appropriate on the coordinator side. The latter class is only appropriate as a superclass of classes which will be saved and restored when an `Action` is current in the saving/restoring thread context and that is not the case in the threads which handle Participant service incoming messages. Note, however, that this should present no problem in managing the object store entries in the absence of heuristic outcomes. A participant recovery record will only be written at complete. The BA implementation should ensure that the entry is deleted in response to `exit` or `fail` operations initiated by the participant. Recovery processing should always ensure that a `close` or `compensate` initiated by the coordinator eventually causes the entry to be deleted. This does present an issue for any tools provided to scan the object store as these need to be aware of the existence of BA recovery records and need to support manual deletion of records in the case that a crashed coordinator cannot proceed to recovery.

During normal operation the XTS implementation maintains an active WS-BA participant map, an in-memory map from WS-BA participant ids to WS-BA participant engines. An engine represents an active participant and is the primary target for incoming `Complete`, `Close` and `Cancel` messages. The map is updated in response to `enlistParticipant`, `exit`, `fail`, `cancel`, `compensate` or `close` operations to reflect the presence of: active participants; and saved or deleted WS-BA participant recovery records in the TX object store.

During bootstrap the XTS recovery module will scan for all XTS WS-BA participant recovery records. Each WS-BA record will be reconstructed in memory and be entered into a recovered

participant map, an in-memory map from WS-BA participant ids to in-memory participant recovery records. Once all records have been scanned and loaded into the recovery map each of the XTS WS-BA recovery modules registered with the XTS implementation will be used to try to restore an active participant from the recovery state located in each of the participant recovery records.

If a BA recovery module successfully converts an entry to a participant the recovery record will be atomically removed from the recovery map and a WS-BA participant engine entered into the active participant map. If no module successfully converts the entry to a participant a recovery warning will be generated and the entry will be left in the recovered participant map for conversion on a later recovery pass, possibly by a newly registered recovery module (registration may not be complete before the first recovery scan is completed). n.b. existing modules get another bite at the cherry i.e. the recovery records is passed to all modules registered at the time of the scan, not just to those which have been registered since the last scan. This covers the case where the failure to initially process the record was because the module needed to wait on other internal/external resources to start up.

An incoming `Close` or `Compensate` message may contain a participant id which is not found in the WS-BA active participant map. This may happen during bootstrap because the initial recovery scan is not complete. It may also happen after the initial scan because the recovery state has not yet been converted to a participant. The former case can be detected using a flag which defaults to false and is set to true after the first scan is completed. In the latter case there will be an entry for the participant in the recovered participant map.

In these two cases incoming `Close` or `Compensate` messages will be silently dropped to ensure that a valid response is provided when recovery is ready. If neither of these exemptions applies it is assumed that the message has been resent and processing follows the transition tables in the WS-BA spec: the response to a `Close` message will be to send a `Closed` response (i.e. to presume that the `Close` has been resent); the response to a `Compensate` request will be to send an `Compensated` response (i.e. to presume that the `Compensate` has been resent).