Transaction Bridging Guide

by Jonathan Red Hat Halliday

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts* [https://fedorahosted.org/liberation-fonts/] set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold`

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keycaps and key combinations. For example:

> To see the contents of the file `my_next_bestselling_novel` in your current working directory, enter the `cat my_next_bestselling_novel` command at the shell prompt and press **Enter** to execute the command.

The above includes a file name, a shell command and a keycap, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from keycaps by the hyphen connecting each part of a key combination. For example:

> Press **Enter** to execute the command.
>
> Press **Ctrl**+**Alt**+**F2** to switch to the first virtual terminal. Press **Ctrl**+**Alt**+**F1** to return to your X-Windows session.

The first paragraph highlights the particular keycap to press. The second highlights two key combinations (each a set of three keycaps with each set pressed simultaneously).

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in `mono-spaced bold`. For example:

> File-related classes include `filesystem` for file systems, `file` for files, and `dir` for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).

To insert a special character into a **gedit** file, choose **Applications** → **Accessories** → **Character Map** from the main menu bar. Next, choose **Search** → **Find…** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit** → **Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

Mono-spaced Bold Italic or ***Proportional Bold Italic***

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

To connect to a remote machine using ssh, type `ssh` *`username@domain.name`* at a shell prompt. If the remote machine is `example.com` and your username on that machine is john, type `ssh john@example.com`.

The `mount -o remount` *`file-system`* command remounts the named file system. For example, to remount the `/home` file system, the command is `mount -o remount /home`.

To see the version of a currently installed package, use the `rpm -q` *`package`* command. It will return a result as follows: *`package-version-release`*.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in `mono-spaced roman` and presented thus:

```
books        Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are also set in `mono-spaced roman` but add syntax highlighting as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
   public static void main(String args[])
       throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object         ref    = iniCtx.lookup("EchoBean");
      EchoHome       home   = (EchoHome) ref;
      Echo           echo   = home.create();

      System.out.println("Created Echo");

      System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
   }
}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.

Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

2. We Need Feedback!

You should over ride this by creating your own local Feedback.xml file.

About This Guide

The Transaction Bridging Guide contains information on how to use JBoss Transactions. This guide provides information on how to integrate JTA (XA) and XTS (WS-AT) transactions using the transaction bridge.

1.1. Audience

This guide is most relevant for application developers working in environments that integrate traditional JavaEE transactions usage and transactional Web Services.

1.2. Prerequisites

JBossTS uses the Java programming language and this manual assumes that you are familiar with programming in Java. In addition, a familiarity with the JTA and XTS components of JBossTS is assumed. You should read the relevant Programmer's Guides before tackling this document.

Introduction

2.1. Contextual Overview

Transactions provide a structuring mechanism for business logic. Use of transactions allows for grouping of data manipulations into constructs with certain properties. Traditional ACID transactions provide for properties of Atomicity, Consistency, Isolation and Durability.

In JavaEE applications, transaction support is provided via the Java Transaction API (JTA). The classes and interfaces in the javax.transaction and javax.transaction.xa packages provide a means by which the programmer may manage transaction demarcation (begin, commit, rollback) and, where necessary, interact with the transaction management system (e.g. enlistResource). In many JavaEE applications, further abstractions are provided on top of the JTA. For example, EJB3 @TransactionAttribute annotations may be used for transaction boundary demarcation in preference to explicit calls to the JTA's UserTransaction interface.

In distributed applications, the JTA implementation may provide propagation of transaction context and transaction control calls between containers (JVMs) using either a propriety transport or JTS, the Java mapping of the CORBA OTS standard on an RMI/IIOP transport. In JBossTS, both local and distributed (JTS) implementations of the JTA are available.

In Web Services applications, ACID transaction management and interoperable context propagation is provided for by the WS-AT standard. JBossTS XTS provides an implementation of both the 1.0 and 1.2 versions of this standard. Bridging is provided only on the more recent version. At the time of writing the standard covers only the web services API and protocol, not the Java API through which the protocol may be driven. Therefore, XTS provides a custom Java API to users, with characteristics broadly similar to the JTA.

For applications that combine traditional JavaEE transaction management and Web Service transaction management, it is often desirable to have some mechanism for linking these transaction types, such that a single transaction may span business logic written for either transaction type. Examples include exposing existing JavaEE transactional business logic (e.g. EJBs) as transactional Web Services, or allowing JavaEE transactional components to utilize transactional Web Services.

2.2. Transaction Bridging

We use the term Transaction Bridging to describe the process of linking the JavaEE and Web Services transaction domains. The transaction bridge component (txbridge) of JBossTS provides bi-directional linkage, such that either type of transaction may encompass business logic designed for use with the other type.

The technique used by the bridge is a combination of interposition and protocol mapping.

Interposition is used in transaction systems to allow a tree of transaction coordinators to be constructed, usually for performance reasons. Interposed coordinators function as transaction

managers for nodes below them in the tree, whilst appearing as resources (participants in WS-AT terminology) to the node above them.

Within a single transaction domain, interposition may be used to allow remote nodes to minimize the number of network calls necessary at transaction termination. The top level node is known as the root coordinator, whilst interposed coordinators are termed subordinate. This name indicates that they are not autonomously responsible for determining the transaction outcome, but rather are driven by their parent coordinator. Therefore, whilst a top level coordinator exposes only the commit and rollback methods for transaction termination and handles the 2PC internally, the subordinates additionally expose the prepare method to their parent, behaving much like resources during the termination protocol.

Figure 2.1. Transaction interposition in a distributed JTA environment

In the transaction bridge, an interposed coordinator is registered into the existing transaction and performs the additional task of protocol mapping. That is, it appears to its parent coordinator to be a resource of its native transaction type, whilst appearing to its children to be a coordinator of their native transaction type, even though these transaction types differ.

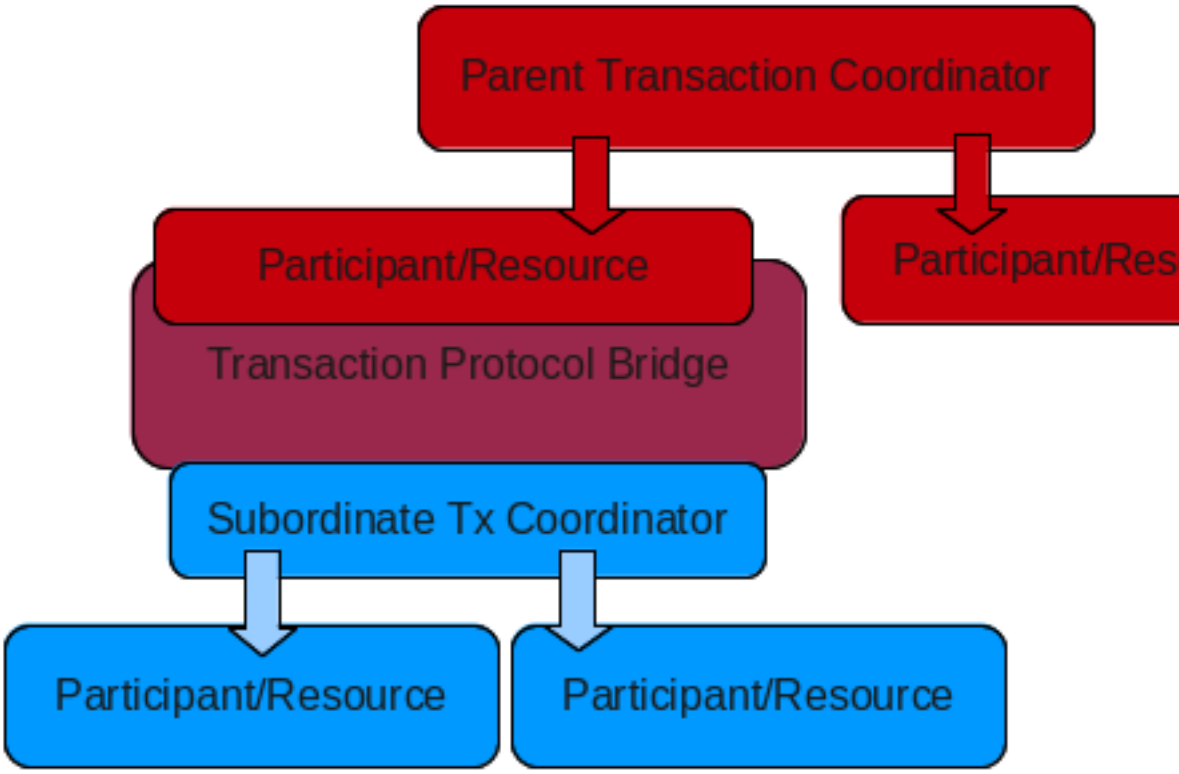

Figure 2.2. Transactional bridging interposition

The interposed coordinator is responsible for performing mapping between the transaction protocols. There is a strong correspondence between the API and protocol used by the JTA and WS-AT transaction types, which is unsurprising given their common heritage and shared problem domain. However, method signatures, exception types and such do differ. The bridge provides a abstraction layer to mask these distinctions as far as possible.

The net result of this is that existing business logic perceives its expected transaction environment, even though the transaction in which it is executing may be subordinate to one of a different type. No changes are necessary to existing transactional applications to allow them to operate in the scope of foreign transactions. This facilitates reuse of existing business logic components in new environments and increases the possibilities for new architectures and interoperability.

Transaction Bridge Architecture

3.1. Overview

The transaction bridge resides in the package org.jboss.jbossts.txbridge and its subpackages.. It consists of two distinct sets of classes, one for bridging in each direction.

The process of inflowing a WS-AT transaction context on a Web Service call into the container and converting it to a local JTA transaction context such that existing transactional JavaEE code (e.g. EJBs) may be called within its scope, is termed Inbound Transaction Bridging. When using inbound bridging, a parent WS-AT transaction coordinator has a subordinate JTA coordinator interposed into it via the transaction bridge.

The process of outflowing a WS-AT transaction context on a call to a transactional Web Service from a business logic method operating in a JavaEE transaction scope, is termed Outbound Transaction Bridging. When using outbound bridging, a parent JTA transaction coordinator has a subordinate WS-AT coordinator interposed into it via the transaction bridge.

For the purpose of understanding this naming convention, it is simplest to view the JTA as being local to the container in which it operates, whilst the Web Service protocol provides for transaction context propagation between servers. This is an accurate representation of the situation that exists where the local JTA version of JBossTS is being used alongside JBossTS XTS in an application server. However, it is an oversimplification of the situation where the JTS option is used. We will return to this case later.

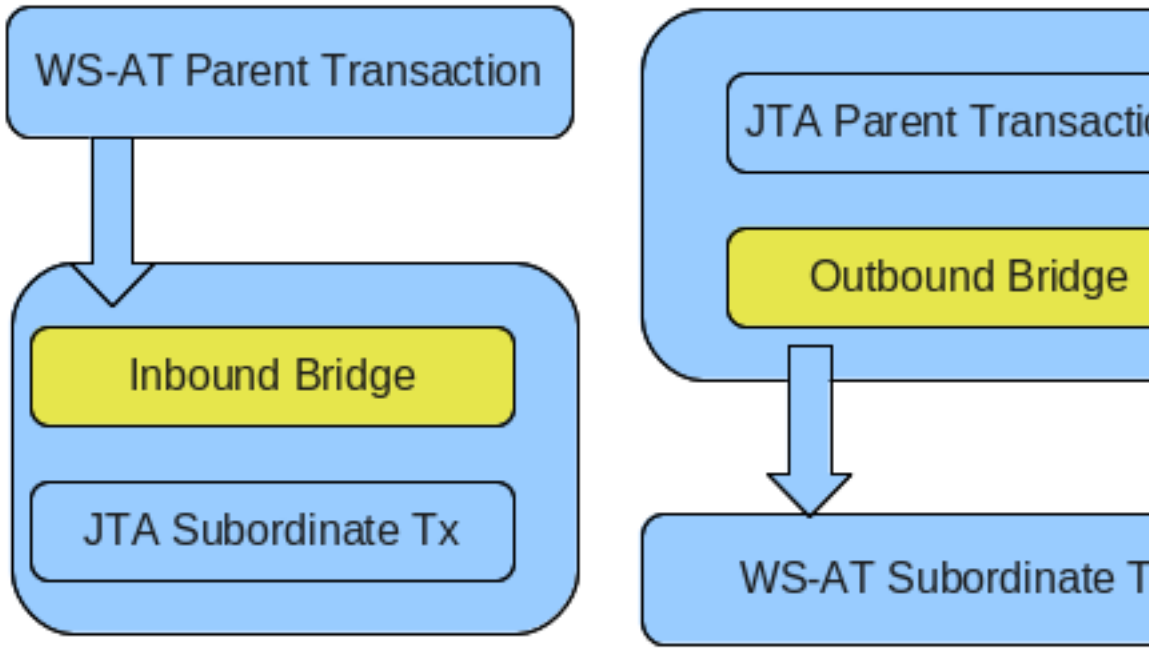

Figure 3.1. Simplified Bridge Architecture

3.2. Shared Design Elements

The design of the inbound and outbound bridges is conceptually very similar. Each provides the following:

- A *BridgeManager* , essentially a factory singleton, providing a means of managing Bridge and resource/participant instances. The chief role of the BridgeManager is to ensure a distinct mapping of a parent transaction context to a single Bridge and resource/participant instance.
- A *Bridge* , which provides Thread to transaction context association and disassociation functions for the subordinate transaction. The Bridge is usually called from the Handler, but may optionally be driven directly.
- A *Handler* , which is registered into the JAX-WS processing pipeline to provide minimally invasive management of Thread to transaction context bindings via the Bridge, an appropriate

instance of which it obtains from the BridgeManager. Whilst the bridge provides handlers only for JAX-WS, it's possible to use these as a model for the implementation of JAX-RPC versions if desired.

- A *VolatileParticipant* and *DurableParticipant* (in the case of the InboundBridge) or *Synchronization* and *XAResource* (in the case of the OutboundBridge) which are enlisted into the parent transaction and wrap the Subordinate transaction coordinator, providing mapping of the transaction termination protocol operations.
- A *RecoveryManager* , which is responsible for automatically restoring the state of crashed transactions and allowing them to complete correctly.

3.3. Inbound Bridging

The process flow when using the inbound bridge is as follows:

1. A remote client starts a WS-AT transaction and invokes a transactional Web Service in the scope of that transaction. The inbound WS invocation therefore has SOAP headers containing the WS-AT transaction context. The coordinator used for this transaction is the root coordinator. It may be remote from either or both of the client and the service it is invoking. The client needs access to a WS-AT implementation, but not a JTA or the transaction bridge deployed.
2. The call arrives at a web service container, which must have JBossTS JTA or JTS, XTS and the transaction bridge deployed. The JAX-WS handler chain for the web service should have both the XTS WS-AT transaction header processor and the inbound bridge handler registered, such that they are invoked in that order.
3. The transaction header processor takes the WS-AT transaction context from XML, creates a corresponding WS-AT TxContext and associates it to the Thread. The bridge handler calls the InboundBridgeManager to obtain an InboundBridge instance corresponding to the TxContext.
4. As the BridgeManager is seeing the TxContext for the first time, it creates a new Bridge instance. It also creates a new Bridge VolatileParticipant and DurableParticipant and registers them with the WS-AT transaction coordinator. These Participants wrap a subordinate JTA transaction.
5. The bridge header processor starts the bridge, which associates the JTA subordinate transaction context to the Thread. At this point the Thread has transaction contexts for both WS-AT and JTA.
6. The JAX-WS pipeline processing continues, eventually calling whatever business logic is exposed. This may be e.g. an EJB using JSR-181 annotations. The business logic may use the JTA transaction in the normal manner e.g. enlisting Synchronizations and XAResources or performing other transactional activity either directly or though the usual JavaEE abstractions.
7. On the return path, the bridge header processor disassociates the JTA transaction context from the Thread via the Bridge. The XTS context processor then does likewise for the WS-AT TxContext.

8. On subsequent web services calls to the same or other web services from the same client, the process is repeated. However, the BridgeManager will, upon seeing the same WS-AT transaction context again, return the existing Bridge instance and not register further Participant instances. This allows substantially better performance than registering one Participant per web service invocation.

9. Upon transaction termination by the client, the WS-AT transaction coordinator will drive the enlisted bridge Participants through the transaction termination protocol. The Participants maps these calls down to the JTA subtransaction coordinator, which in turn passes them on to any Synchronizations or XAResources enlisted in the transaction. This process is not visible to the business logic, except in so far as it may have registered its own Synchronizations, XAResources or Participants with the transaction.

3.4. Outbound Bridging

The process flow when using the outbound bridge is as follows:

1. A client starts a JTA transaction and invokes a remote transactional Web Service in the scope of that transaction. The client must have JBossTS JTA (or JTS) and XTS deployed, as well as the transaction bridge. The coordinator used for the JTA transaction is the root coordinator. The server hosting the target web service needs a WS-AT transaction implementation but not a JTA or the transaction bridge.

2. The outbound WS invocation flows though a handler chain that has the outbound transaction bridge handler and XTS header context processor registered, such that they are invoked in that order.

3. The bridge handler calls the outbound bridge manager to obtain an outbound bridge instance corresponding to the JTA transaction context. As the BridgeManager is seeing the context for the first time, it creates a new Bridge instance. It also creates a Synchronization and XAResource instance to wrap the subordinate WS-AT transaction and registers these with the JTA transaction.

4. The bridge handler starts the bridge, which associates the subordinate WS-AT transaction context to the Thread. The WS-AT header context processor then serializes this into XML in the headers of the outbound Web Services call.

5. The receiving Web Service sees a WS-AT context and can work with it in the normal manner, without knowing it is a subordinate context.

6. On the return path, the bridge handler disassociates the WS-AT TxContext from the Thread via the Bridge.

7. On subsequent calls to the same or other transactional Web Services in the scope of the same JTA transaction, the process is repeated. However, the BridgeManager will, upon seeing the same JTA transaction context again, return the existing Bridge and not register another

Synchronization or XAResource with the parent JTA transaction. This allows substantially better performance than registering once per web service invocation.

8. Upon transaction termination by the client, the JTA transaction coordinator will drive the enlisted bridge Synchronization and XAResource through the transaction termination protocol. The XAResource maps these calls down to the WS-AT subtransaction coordinator, which in turn passes them on to any Volatile or Durable Participants enlisted in the transaction. This process is not visible to the business logic, except in so far as it may have registered its own Participants, XAResources or Synchronizatons with the transaction.

3.5. Crash Recovery

The bridge includes independent crash recovery systems for the inbound and outbound sides. These are automatically installed and activated as part of the bridge deployment. They rely upon the recovery mechanisms in the JTA and XTS components, which are likewise deployed and activated by default as part of their respective components.

It is the responsibility of the application(s) to use suitable XAResources (inbound) or DurableParticipants (outbound). In general the former will be from XA datasources or messaging systems, whilst the latter will be custom implementations. In either case it is important to ensure recovery is correctly configured for the resource manager(s) before using them in production, via the bridge or otherwise. The JBossTS documentation set details crash recovery configuration, as does the application server administration guide. For resource manager specific information e.g. Oracle db permissions settings for recovery connections, please consult the vendor's documentation.

A bridged transaction will involve several distinct log writes, potentially on multiple hosts. Resolving the transaction may require more than one crash recovery cycle, due to ordering constrains on the events taking place during recovery. If a transaction fails to recover after all servers have been restored to service for more than two recovery cycles duration, the JBossTS objectstore browser and server logs may be useful for diagnosing the issue. Where a transaction involves multiple bridges the number of recovery cycles required to resolve it may further increase. For systems requiring maximum availability it is therefore not recommended to span a transaction through more than one bridge.

Note that the 1PC commit optimization should not be used with outbound bridged transactions in which the subordinate may contain more than one Participant. Even where only one Participant is used, crash recovery logs may not correctly reflect the actual transaction outcome. The 1PC optimization is on be default and may be disabled by setting <property name="commitOnePhase">false </property> on CoordinatorEnvironmentBean.

See the 'Design Notes' appendix for detailed information on potential crash recovery scenarios and how each is handled.

Using the Transaction Bridge

4.1. Introduction

This section describes how to use the transaction bridge in your applications. It is recommended you first read the preceding chapters for a theoretical background in the way the bridge functions.

4.2. Deployment

The txbridge.jar file should be placed in JBossAS server/lt;config>/deploy directory. The server must also be running JBossTS JTA (the default transaction manager) or JTS, and also JBossTS XTS. The versions of all these components must be consistent.

4.3. Inbound Bridging

To use the inbound bridge, register the JAX-WS handler into the handler chain of any Web Service as follows:

Example 4.1. Registering the `handler` for Inbound Bridging

```
<handler-chain>
  <protocol-bindings>##SOAP11_HTTP</protocol-bindings>
  <handler>
    <handler-name>TransactionBridgeHandler</handler-name>
                                                                    <handler-
class>org.jboss.jbossts.txbridge.inbound.JaxWSTxInboundBridgeHandler</handler-
class>
  </handler>

  <handler>
    <handler-name>WebServicesTxContextHandler</handler-name>
       <handler-class>com.arjuna.mw.wst11.service.JaxWSHeaderContextProcessor</
handler-class>
  </handler>
</handler-chain>
```

The web service may then operate as though running in the scope of a JTA transaction, as indeed it is. For example, it can call (or indeed simply be) an EJB3 business logic method annotated with @TansactionAttribute(TransactionAttributeType.MANDATORY).

Note that the handlers expect a WS-AT transaction context to be present on all inbound invocations. If you wish deploy your service in such a way as to make transactional invocation optional, you must expose it though two different endpoints, one transactional and one not, with the handlers registered only on the former. This limitation may be addressed in future versions.

4.4. Outbound Bridging

To use the outbound bridge, register the JAX-WS handler into the handler chain of any Web Service client application as follows:

Example 4.2. Registering the `handler` for Outbound Bridging

```
<handler-chain>
  <protocol-bindings>##SOAP11_HTTP</protocol-bindings>
  <handler>
    <handler-name>TransactionBridgeHandler</handler-name>
                                                                    <handler-
class>org.jboss.jbossts.txbridge.outbound.JaxWSTxOutboundBridgeHandler</
handler-class>
  </handler>

  <handler>
    <handler-name>WebServicesTxContextHandler</handler-name>
        <handler-class>com.arjuna.mw.wst11.client.JaxWSHeaderContextProcessor</
handler-class>
  </handler>
</handler-chain>
```

The web service client may then make calls to web service implementations that expect to be invoked in the scope of a WS-AT transaction.

Note that the handlers expect a JTA transaction context to be present on the client thread used to make the outbound web service invocation. If the context is not always present, different stubs must be used for the transactional and non-transactional cases and the handler chain registered only on the former. This limitation may be addressed in future versions.

4.5. Demonstration Application

A simple demonstration application is available to show usage of the bridge. It is modeled to some extent on the XTS 'Night Out' demonstrator application, with which readers are assumed to be familiar.

Since transactions mostly run without visible effect, the demo is useful mainly as an example of how to utilize the bridge. The bridge implementation does however contain trace level logging for most functions. Used in conjunction with verbose logging from XTS, the transaction manager, the Web Service stack and the EJB container, this can be used to gain a detailed understanding of the flow of events in the system. Alternatively, stepping though the demo using a source debugger can be instructive.

To deploy and run the demo application, edit demo/build.xml to ensure the jbossas.home and jbossas.server properties are set correctly, then execute 'ant dist' to build the application artifacts.

Start the application server, then deploy the service side of the demo using 'ant deploy-service' Once it has deployed, the client app can be similarly installed using 'ant deploy-client'. Depending on your server configuration, the client will then be accessible from e.g. *http://localhost:8080/txbridge-demo-client/*

4.5.1. Inbound Bridge

The demonstrator exposes a EJB3 SLSB as a transactional web service ('Bistro') via the inbound bridge. Note that the code implementing this service is standard EJB with JSR-181 annotations and has no compile time dependency on XTS or the txbridge. The only point of linkage is the usage of the @HandlerChain(file = "jaxws-handlers-server.xml") annotation to reference a xml file containing the XTS and txbridge handlers, as detailed above. Other than this the service side of the application uses only standard JavaEE elements and has no direct knowledge of WS-AT transactions.

A client starts a WS-AT transaction and makes an invocation on the web service. The client does not use JTA (XA) transactions. It uses @HandlerChain(file = "jaxws-handlers-client.xml") to register the XTS header context processor, but is otherwise similar to the XTS demo client.

In this demo, the inbound bridge converts the WS-AT context to a JTA one and invokes the EJB in that scope. By default the EJB is backed by the hsqldb embedded in JBossAS, for ease of deployment. This database does not support XA, so the resource registered for it uses LRCO. However, this point is not significant to the demo. Curious uses can readily use a true XA database by deploying it into JBossAS via the usual <xa-datasource> in a -ds.xml file, then alter the demo's dd/persistence.xml to reference it.

4.5.2. Outbound Bridge

The demonstrator client application can also be used to invoke the XTS Night Out demo Restaurant Service via the outbound bridge. Deploy the XTS demo application services, then select the 'JTA' transaction type in the client. In this scenario the client uses a JTA transaction only, whilst the service understands WS-AT type transactions only. Note that the client has its own copy of the service API, annotated with @HandlerChain(file = "jaxws-handlers-client.xml"), which is the only point of linkage with the transaction bridge. Once again neither the client nor server have any compile time dependency on the bridge.

4.6. Loops and Diamonds

In distributed environments that utilize transaction bridging, it is possible to construct arrangements of servers such that a transaction context passes though more than one interposition. These can give rise to some undesirable issues, including locking and performance problems.

A simple case would be a loop in which a JTA transaction context is bridged outbound to a WS-AT context, passed though one or more remote servers and inflowed back to the original server through an inbound bridge. This may result in a new subordinate JTA context, rather than reuse of the existing parent context in the original server.

This situation has two main observable effects. Firstly, the parent JTA transaction and indirectly subordinate JTA transaction are considered distinct and XAResources may not be shared between them. In most cases this will cause isolation between the transactions, such that they do not share locks or see eachother's changes. This may cause deadlocks in the application. Secondly, performance will be poor relative to reuse of the original context, particularly if the interposition chain becomes long.

A similar problem exists where a transaction context is propagated from a single source to a single destination server via two or more separate routes, the abstract paths forming a diamond shape. In such case the intermediate nodes operate independently and will bridge the original context to two separate interposed contexts. To the destination server these will appear unrelated, rather than as representations of the same transaction. Thus instead of recombining into a single shared transaction context at the destination, they will behave as different transactions, giving rise once again to potential deadlock and performance issues.

These problems may be partially addressed by having a shared context mapping service available on the network, which each bridge consults when working with a previously unseen transaction context for the first time. Using such a mechanism, bridge instances may identify transactions for which an established mapping already exists and reuse that relationship rather than creating a new one.

This shared service model does however cause some issues of its own with regard to performance and availability. It is not currently implemented. Therefore, users are urged to be cautious when constructing distributed applications. Whilst location abstraction is sometimes desirable, is is important to maintain a clear understanding of the deployment relationships between transactional components in the system.

4.7. Distributed JTA and the JTS

The JavaEE transaction engine in JBossTS comes in two varieties. These are the local only JTA, which does not support propagation of transaction context or transaction control calls between JVMs and the JTAX, which provides the JTA API implemented by a JTS engine that does support distributed usage.

JBossAS uses the local JTA implementation by default, but can be reconfigured to use the JTS via the JTA API, such that it supports distributed transactions without requiring any changes to business applications.

In environments requiring transaction propagation of JTA transactions, it is feasible to use either the JTS or an outbound and inbound bridge pair to achieve this. In the former case the transport is RMI/IIOP for the transaction control and RMI/IIOP or JRMP for the transactional business logic calls. In the latter case the transport is Web Services for both transaction control and business logic.

From a transaction management perspective the JTS solution is preferred, due to simplicity (no protocol mapping is needed), maturity (JBossTS JTS was the world's first JTS implementation

and has been extensively used and tested in production environments) and performance (binary vs. xml).

It is possible to use transactions that propagate context on some calls via JTS and on others via Web Services, such as a client invoking both EJBs via RMI/IIOP and Web services with WS-AT context. In such cases it's possible for a transaction to have multiple representations that the infrastructure cannot determine are related, even if they actually represent different contexts in the same interposition hierarchy. Care must therefore be taken to avoid the problems described previously in 'Loops and Diamonds'.

4.8. Logging

The transaction bridge uses the jboss-logging system. When running inside JBossAS 6, logging is configured via the server's deploy/jboss-logging.xml file. To enable full logging for the transaction bridge, which may be useful for debug purposes, the following should be used:

Example 4.3. Configuring Transaction Bridge Logging

```
<logger category="org.jboss.jbossts.txbridge">
  <level name="ALL" />
</logger>
```

Note that the transaction bridge is a thin layer on top of the XTS and JTA/JTS components of JBossTS, and that it also interacts with other parts of the application server. To gain a comprehensive understanding of the system's operation, it may be necessary to enable verbose logging for some of these other components also. The JBossTS logging system is discussed in detail in the accompanying documentation set, but for ease of reference the following may be used to enable verbose logging:

Example 4.4. Configuring verbose logging

```
<logger category="com.arjuna">
  <level name="ALL" />
</logger>
```

Note also that deployment ordering issues can result in JBossTS components, including the transaction bridge, becoming active before the logging system is fully configured. In such cases a default logging level may apply during startup, resulting in some more detailed debug messages being missed.

Known Limitations

The current transaction bridge release has the following limitations:

- The bridge operates only on WS-AT 1.2, not 1.0, although XTS includes implementations of both versions of WS-AT. Care must therefore be taken to deploy and configure the system correctly.
- The bridge provides JAX-WS handlers only, not JAX-RPC, although it is possible to create such if required.
- Long running activities that occur during the transaction termination process may cause timeouts in the transaction system, which can in turn cause inconsistent transaction outcomes or incomplete transaction termination. To minimize this problem, it is advised to manually flush data that would otherwise be flushed by Synchronizations during termination, such as hibernate session state.
- A transaction context must always be present on the Thread in order for the context processors to operate correctly, as detailed previously in 'Using the Transaction Bridge'.
- A subordinate transaction context will be created and registered into the parent transaction unconditionally, which can cause unnecessary overhead in situations where no transactional activity takes place in the scope of the subordinate. Care should be taken to register the bridge handlers only on methods that do require them. In future releases this may be addressed by the use of WS-Policy or lazy initialization techniques.
- Transaction mappings are local to BridgeManagers, which are singletons. This means mappings are classloader scoped and not shared across JVMs. This gives rise to issues where transactional resources are accessed indirectly though multiple bridges or transaction context transports, as described in 'Loops and Diamonds'.
- Crash recovery is subject to certain timing issues, due to the interaction between recovery of the JTA/XA and XTS sides of the transaction. It may take more than one crash recovery cycle for a bridged transaction to recover fully. Note that recovery of subordinate transactions is dependent on the recovery of their parent, so care must be taken to ensure the correct recovery of any external transaction manager used in that role. The transaction bridge does not currently provide dedicated tooling for the manual resolution of orphaned subordinates, instead relying on the general purpose objectstore maintenance tooling provided by JbossTS.
- Note that crash recovery will not behave correctly for outbound bridged transactions if 1PC commit optimization is used in the parent JTA transaction. This is not specific to the bridge, but rather is a generic issue with any transaction in which a single resource is an interposed subordinate coordinator. Inbound bridges transactions are unaffected as XTS (WS-AT) does not utilize a 1PC optimization.

Appendix A. Design Notes

A.1. General Points

This section records key design points relating to the bridge implementation. The target audience for this section is software engineers maintaining or extending the transaction bridge implementation. It is unlikely to contain material useful to users, except in so far as they wish to contribute to the project. An in-depth knowledge of JBossTS internals may be required to make sense of some parts of this appendix.

The txbridge is written as far as possible as a user application layered on top of the JTA and XTS implementations. It accesses these underlying components through standard or supported APIs as far as possible. For example, XAResource is favored over AbstractRecord, the JCA standard XATerminator is used for driving subordinates and so on. This facilitates modularity and portability.

It follows that functionality required by the bridge should first be evaluated for inclusion in one of the underlying modules, as experience has shown it is often also useful for other user applications. For example, improvements to allows subordinate termination code portability between JTA and JTS, and support for subordinate crash recovery have benefited from this approach. The txbridge remains a thin layer on top of this functionality, containing only purpose specific code.

The 'loops and diamonds' problem boils down to providing deterministic, bi-directional 1:1 mapping between an Xid (which is fixed length) and a WS-AT context (which is unbounded length in the spec, although bounded for instances created by the XTS). Consistent hashing techniques get you so far with independent operation, but the only 100% solution is to have a shared service on the network providing the mapping lookup. Naturally this then becomes a single point of failure as well as a scalability issue. For some scenarios it may be possible to use interceptors to propagate the Xid on the web services call as extra data, instead of trying to reproduce the mapping at the other end. Unfortunately XA does not provide for this kind of extensibility, although CORBA does, leading to the possibility of solving the issue without a centralized approach in mixed JTS+WS-AT environments.

Requiring a tx context on all calls is a bit limiting, but JBossWS native lacks a WS-Policy implementation. Things may change with the move to CXF. This is really a wider issue with XTS, not just the bridge.

A.2. Crash Recovery Considerations

As usual with transactions, it's the crash recovery that provides for the most complexity. Recovery for the inbound and outbound sides is handled independently. Because of event ordering between recovery modules (JTA, XTS), it requires two complete cycles to resolve some of these crash recovery situations.

A.2.1. Inbound Crash Recovery

An inbound transaction involves at least four log writes. Top down (i.e. in reverse order of log creation) these are: The WS-AT coordinator log (assumed here to be XTS, but may be 3rd party), the XTS Participant log in the receiving server, the JCA Subordinate transaction log and at least one XA Resource Manager log (which are 3rd party e.g. Oracle).

There is no separate log created by the txbridge. The XTS Participant log inlines the Serializable BridgeDurableParticipant via its writeObject method. Recorded state includes its identity (the Xid) and the identity of the separately logged JTA subordinate tx (a Uid).

XTS is responsible for the top level coordinator log. JBossTS is responsible for the JTA subordinate tx log and 3rd party RMs are each responsible for their own.

The following situations may exist at recovery time, according to the point in time at which the crash occurred:

RM log only: In this case, the InboundBridgeRecoveryManager's XAResourceOrphanFilter implementation will be invoked via JBossTS XARecoveryModule, will recognize the orphaned Xids by their formatId (which they inherit from the JCA subordinate, which the txbridge previously created with a specially constructed inflowed Xid) and will vote to have the XARecoveryModule roll them back as no corresponding JCA subordinate log exists, so presumed abort applies.

RM log and JTA subordinate tx log: The InboundBridgeRecoverytManager's scan of indoubt subordinate JTA transactions identifies the JTA subordinate as being orphaned and rolls it back, which in turn causes the rollback of the RM's XAResource.

RM log, JTA subordinate log and XTS Participant log: XTS is responsible for detecting that the Participant is orphaned (by re-sending Prepared to the Coordinator and receiving 'unknown tx' back) and initiating rollback under the presumed abort convention.

WS-AT coordinator log and all downstream logs: The coordinator re-sends Commit to the Participant and the transaction completes.

A.2.2. Outbound Crash Recovery

An outbound transaction involves log writes for the JTA parent transaction and the XTS BridgeWrapper coordinator. There is not a separate log created by the txbridge. The JTA tx log inlines the Serializable BridgeXAResource via its writeObject method. Recorded state includes the JTA tx id and bridgeWrapper id String. In addition a Web Service participating in the subordinate transaction will create a log. Assuming it's XTS, the participant side log will inline any Serializable Durable2PCParticipant, effectively forming the RM log.

The following situations may exist at recovery time, according to the point in time at which the crash occurred:

RM log (i.e. XTS Participant log, inlining Serializable Durable2PCParticipant) only: XTS is responsible for detecting that the Participant is orphaned (its direct parent, the subordinate

coordinator, is missing) and rolling it back. The bridge recovery code is not involved – XTS recovery deserializes and drives any app DurableParticipants directly.

RM log and XTS subordinate log: The DurableParticipant(s) (i.e. client side) and XTS subordinate coordinator / BridgeWrapper (i.e. server side) and reinstantiated by XTS. The BridgeWrapper, being subordinate to a missing parent, must be identified and explicitly rolledback by the bridge recovery code. The bridge recovery manager is itself a RecoveryModule, thus invoked periodically to perform this task. It identified its own BridgeWrapper instance from amongst all those awaiting recovery by means of an id prefix specific to the txbridge code. See JBTM-725 for further details.

RM log, XTS subordinate log and JTA parent log (with inlined BridgeXAResource): Top down recovery by the JTA recovery module drives tx to completion, taking the normal JTA parent->BridgeXAResource->XTS subordinate->DurableParticipant path. Note that if the bridge is the only XAResource in the parent, the JTA must have 1PC commit optimization disabled or it won't write a log for recovery.

A.3. Test framework

The test suite for the txbridge is split along two axis. Firstly, the inbound and outbound sides of the bridge have their own test suites in a parallel code package hierarchy. These are largely mirrors, containing tests which have matching intent but different implementation details. Secondly, the tests are split between those for normal execution and those for crash recovery.

The tests use a framework consisting of a basic servlet acting as client (the code pre-dates the availability of XTS lightweight client), a basic web service as server and a set of utility classes implementing the appropriate interfaces (Participant/Synchronization/XAResource). These classes contain the bare minimum of test logic. In order to make the tests as easy to understand and modify as possible, an attempt is made to capture the entirety of the test logic within the junit test function instead of splitting it over the framework classes. To facilitate this, extensive use is made of byteman and its associated dtest library, which provides basic distributed mock-like execution tracing and configuration. You probably need to take a detour and read the dtest docs before proceeding further.

The basic tests all follow the same pattern: make a call through the bridge, following different logic paths in each test, and verify that the test resources see the expected method calls. For example, in a test that runs a transaction successfully, expect to see commit called on enlisted resources and rollback not called. For a test that configures the prepare to fail, expect to see rollback called and commit not called. The tests verify behavior in the presence of 'expected' errors e.g. prepare failures, but generally don't cover unexpected failures e.g. exceptions thrown from commit.

Test normal execution targets in the tests/build.xml assume the server is started manually with byteman installed and has XTS, txbridge and the test artifacts deployed. Note that it also contains targets that may be called to achieve the last of these steps.

The crash rec tests start (and subsequently restart) the server automatically, but assume the that XTS, txbridge and the test artifacts are deployed. To manage the server they need to be provided with JBOSS_HOME and JAVA_HOME values in the build.xml.

Appendix B. Revision History

Revision History
Revision 1 Tue Apr 12 2010 TomJenkinson<tom.jenkinson@redhat.com>
Initial creation of book by publican