BEA WebLogic Server

WebLogic Tuxedo Connector ATMI Programmer's Guide

BEA WebLogic Tuxedo Connector Release 1.0
Document Date: June 29, 2001

Copyright

Restricted Rights Legend

Trademarks or Service Marks

WebLogic Tuxedo Connector ATMI Programmer's Guide

Part Number	Document Date	Software Version
N/A	June 29, 2001	BEA WebLogic Tuxedo Connector 1.0

Contents

3. Developing WebLogic Tuxedo Connector Service Applications

4. WebLogic Tuxedo Connector Transactions

5. Application Error Management

About This Document

This document introduces the BEA WebLogic Tuxedo Connector application development environment. It describes how to establish a development environment and how to package applications for deployment.

The document is organized as follows:

- Chapter 1, "Introduction to JATMI Programming," provides information about the development environment you will be using to write code for applications that interoperate between WebLogic Server and Tuxedo.
- Chapter 2, "Developing WebLogic Tuxedo Connector Client Applications," provides information on how to create client applications.
- Chapter 3, "Developing WebLogic Tuxedo Connector Service Applications," provides information on how to create service applications.
- Chapter 4, "WebLogic Tuxedo Connector Transactions," provides information on global transactions and how to define and manage them in your applications.
- Chapter 5, "Application Error Management," provide mechanisms to manage and interpret error conditions.

Audience

This document is written for system administrators and application developers who are interested in building distributed Java applications that interoperate between WebLogic Server and Tuxedo environments. It is assumed that readers are familiar with the WebLogic Server, Tuxedo, XML and Java programming.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation.

How to Print the Document

You can print a copy of this document from a Web browser, one main topic at a time, by using the File→Print option on your Web browser.

A PDF version of this document is available on the WebLogic Server documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Server documentation Home page, click Download Documentation, and select the document you want to print.

Adobe Acrobat Reader is available at no charge from the Adobe Web site at http://www.adobe.com.

Related Information

The BEA corporate Web site provides all documentation for WebLogic Server and Tuxedo.

For more information about Java applications, refer to the Sun Microsystems, Inc. Java site at http://java.sun.com/.

Contact Us!

Your feedback on BEA documentation is important to us. Send us e-mail at docsupport@bea.com if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the documentation.

In your e-mail message, please indicate the software name and version you are using, as well as the title and document date of your documentation. If you have any questions about this version of BEA WebLogic Server, or if you have problems installing and running BEA WebLogic Server, contact BEA Customer Support through BEA WebSupport at http://www.bea.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Usage
Ctrl+Tab	Keys you press simultaneously.
italics	Emphasis and book titles.

Convention	Usage

`monospace text`	Code samples, commands and their options, Java classes, data types, directories, and file names and their extensions. Monospace text also indicates text that you enter from the keyboard.
	Examples:
	`import java.util.Enumeration;`
	`chmod u+w *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

`monospace italic text`	Variables in code. *Example*: `String CustomerName;`
UPPERCASE TEXT	Device names, environment variables, and logical operators. *Examples*: LPT1 BEA_HOME OR
`{ }`	A set of choices in a syntax line.
`[ ]`	Optional items in a syntax line. *Example*: `java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]`
`|`	Separates mutually exclusive choices in a syntax line. *Example*: `java weblogic.deploy [list|deploy|undeploy|update] password {application} {source}`
`...`	Indicates one of the following in a command line: ■ An argument can be repeated several times in the command line. ■ The statement omits additional optional arguments. ■ You can enter additional parameters, values, or other information

Convention	Usage
. . .	Indicates the omission of items from a code example or from a syntax line.

CHAPTER

1 Introduction to JATMI Programming

The following sections provide information about the development environment you will be using to write code for applications that interoperate between WebLogic Server and Tuxedo:

- Developing WebLogic Tuxedo Connector Applications
- WebLogic Tuxedo Connector JATMI Primitives
- WebLogic Tuxedo Connector Typed Buffers

Developing WebLogic Tuxedo Connector Applications

In addition to the Java code that expresses the logic of your application, you will be using the Java Application -to-Transaction Monitor Interface (JATMI) to provide the interface between WebLogic Server and Tuxedo. This allows you to develop clients and servers without modifying existing Tuxedo services.

Note: For more information on the WebLogic Tuxedo Connector JATMI, view the WebLogic Tuxedo Connector Javadoc by opening the `index.html` file in the `doc` directory of your WebLogic Tuxedo Connector installation.

Developing WebLogic Tuxedo Connector Clients

A client process takes user input and sends a service request to a server process that offers the requested service. WebLogic Tuxedo Connector JATMI client classes are used to create clients that access services found in Tuxedo. These client classes are available to any service that is made available through the WebLogic Tuxedo Connector XML configuration file in the StartUpClass of your WebLogic Server.

Developing WebLogic Tuxedo Connector Servers

Servers are processes that provide one or more services. They continually check their message queue for service requests and dispatch them to the appropriate service subroutines. WebLogic Tuxedo Connector uses EJBs to implement services which Tuxedo clients invoke.

WebLogic Tuxedo Connector JATMI Primitives

The JATMI is a set of primitives used to begin and end transactions, allocate and free buffers, and provide the communication between clients and servers.

Table 1-1 JATMI Primitives

Name	Operation
`tpacall`	Use for asynchronous invocations of a Tuxedo service.
`tpcall`	Use for synchronous invocation of a Tuxedo service.
`tpdequeue`	Use for receiving messages from a Tuxedo /Q.
`tpenqueue`	Use for placing a message on a Tuxedo /Q.

Table 1-1 JATMI Primitives

Name	Operation
`tpgetreply`	Use for retrieving replies from a Tuxedo service.
`tpterm`	Use to close a connection to a Tuxedo object.

WebLogic Tuxedo Connector Typed Buffers

Note: UNICODE strings are not supported on the Tuxedo environment.

WebLogic Tuxedo Connector provides an interface called **`TypedBuffers`** that corresponds to Tuxedo typed buffers. Messages are passed to servers in typed buffers. The WebLogic Tuxedo Connector provides the following buffer types:.

Table 1-2 TypedBuffers

Buffer Type	Description
TypedString	Buffer type used when the data is an array of characters that terminates with the null character. Tuxedo equivalent: STRING.
TypedCArray	Buffer type used when the data is an undefined array of characters (byte array), any of which can be null. Tuxedo equivalent: CARRAY.
TypedFML	Buffer type used when the data is self-defined. Each data field carries its own identifier, an occurrence number, and possibly a length indicator. Tuxedo equivalent: FML.
TypedFML32	Buffer type similar to TypeFML but allows for larger character fields, more fields, and larger overall buffers. Tuxedo equivalent: FML32.
TypedXML	Buffer type used when data is an XML based message. Tuxedo equivalent: XML for Tuxedo Release 7.1 and higher.

2 Developing WebLogic Tuxedo Connector Client Applications

The following sections describe how to create client programs that take user input and send service requests to a server process that offers a requested service.

- Joining and Leaving Applications
- Basic Client Operation
- Example Client Application

WebLogic Tuxedo Connector JATMI client classes are used to create clients that access services found in Tuxedo.

Note: For more information on JATMI classes, view the WebLogic Tuxedo Connector Javadoc by opening the `index.html` file in the `doc` directory of your WebLogic Tuxedo Connector installation.

Joining and Leaving Applications

Tuxedo and WebLogic Tuxedo Connector have different approaches to connect to services.

Joining an Application

The following section compares how Tuxedo and WebLogic Tuxedo Connector join an application:

- Tuxedo uses **`tpinit()`** to join an application.
- WebLogic Tuxedo Connector uses the BDMCONFIG XML configuration file to provide information required to create a path to the Tuxedo service. It provides security and client authentication by configuring the T_DM_REMOTE_TDOMAIN and T_DM_IMPORT sections of the BDMCONFIG XML configuration file. This pathway is created when the WebLogic Server is started and the WebLogic Tuxedo Connector XML configuration file is loaded.
- WebLogic Tuxedo Connector uses **`TuxedoConnection`** to get a Tuxedo object and then uses **`getTuxedoConnection()`** to make a connection to the Tuxedo object.

Leaving an Application

The following section compares how Tuxedo and WebLogic Tuxedo Connector leave an application:

- Tuxedo uses **`tpterm()`** to leave an application.
- WebLogic Tuxedo Connector uses the JATMI primitive **`tpterm()`** to close a connection to a Tuxedo object.
- WebLogic Tuxedo Connector closes the pathway to a Tuxedo service when the WebLogic Server is shutdown.

Basic Client Operation

A client process uses Java and JATMI primitives to provide the following basic application tasks:

- Get a Tuxedo Object
- Perform Message Buffering
- Send and Receive Messages
- Closing a Connection

A client may send and receive any number of service requests before leaving the application.

Get a Tuxedo Object

Establish a connection to a remote domain by using the **`TuxedoConnectionFactory`** to lookup "`tuxedo.services.TuxedoConnection`" in the JNDI tree and get a **`TuxedoConnection`** object using **`getTuxedoConnection()`**.

Perform Message Buffering

Use the following buffer types when sending and receiving messages between your application and Tuxedo:

Table 2-1 TypedBuffers

Buffer Type	Description
TypedString	Buffer type used when the data is an array of characters that terminates with the null character. Tuxedo equivalent: STRING.
TypedCArray	Buffer type used when the data is an undefined array of characters (byte array), any of which can be null. Tuxedo equivalent: CARRAY.
TypedFML	Buffer type used when the data is self-defined. Each data field carries its own identifier, an occurrence number, and possibly a length indicator. Tuxedo equivalent: FML.

Table 2-1 TypedBuffers

Buffer Type	Description
TypedFML32	Buffer type similar to TypeFML but allows for larger character fields, more fields, and larger overall buffers. Tuxedo equivalent: FML32.
TypedXML	Buffer type used when data is an XML based message. Tuxedo equivalent: XML for Tuxedo Release 7.1 and higher.

Send and Receive Messages

Use the following JATMI primitives to send and receive messages between your application and Tuxedo:

Table 2-2 JATMI Primitives

Name	Operation
`tpacall`	Use for asynchronous invocations of a Tuxedo service.
`tpcall`	Use for synchronous invocation of a Tuxedo service.
`tpdequeue`	Use for receiving messages from a Tuxedo /Q.
`tpenqueue`	Use for placing a message on a Tuxedo /Q.
`tpgetreply`	Use for retrieving replies from a Tuxedo service.

Closing a Connection

Use **`tpterm()`** to close a connection to an object and prevent future operations on this object. This is the equivalent of the JCA **`close()`**.

Example Client Application

The following Java code provides an example of the ToupperBean client application which sends a string argument to a server and receives a reply string from the server.

Listing 2-1 Example Client Application

```
.
.
.
public String Toupper(String toConvert)
   throws TPException, TPReplyException
{
     Context ctx;
     TuxedoConnectionFactory tcf;
     TuxedoConnection myTux;
     TypedString myData;
     Reply myRtn;
     int status;

     log("toupper called, converting " + toConvert);

     try {
          ctx = new InitialContext();
          tcf = (TuxedoConnectionFactory) ctx.lookup(
                "tuxedo.services.TuxedoConnection");
     }
     catch (NamingException ne) {
          // Could not get the tuxedo object, throw TPENOENT
           throw new TPException(TPException.TPENOENT, "Could not get
TuxedoConnectionFactory : " + ne);
     }

     myTux = tcf.getTuxedoConnection();

     myData = new TypedString(toConvert);

     log("About to call tpcall");
     try {
          myRtn = myTux.tpcall("TOUPPER", myData, 0);
}
     catch (TPReplyException tre) {
          log("tpcall threw TPReplyExcption " + tre);
```

```
            throw tre;
        }
        catch (TPException te) {
            log("tpcall threw TPException " + te);
            throw te;
        }
        catch (Exception ee) {
            log("tpcall threw exception: " + ee);
            throw new TPException(TPException.TPESYSTEM, "Exception: " + ee);
        }
        log("tpcall successfull!");

        myData = (TypedString) myRtn.getReplyBuffer();

        myTux.tpterm();// Closing the association with Tuxedo

        return (myData.toString());
}
.
.
.
```

3 Developing WebLogic Tuxedo Connector Service Applications

The following sections provide information on how to create WebLogic Tuxedo Connector service applications:

- Basic Service Application Operation
- Example Service Application

Basic Service Application Operation

A service application uses Java and JATMI primitives to provide the following tasks:

- Receive Client Messages
- Buffer Messages
- Perform the Requested Service
- Return Client Messages

Receive Client Messages

Use the **`TPServiceInformation`** class **`getServiceData()`** method to receive client messages.

Note: For more detailed information on the **`TPServiceInformation`** class, view the WebLogic Tuxedo Connector Javadoc by opening the `index.html` file in the doc directory of your WebLogic Tuxedo Connector installation.

Buffer Messages

Use the following buffer types when sending and receiving messages between your application and Tuxedo:

Table 3-1 TypedBuffers

Buffer Type	Description
TypedString	Buffer type used when the data is an array of characters that terminates with the null character. Tuxedo equivalent: STRING.
TypedCArray	Buffer type used when the data is an undefined array of characters (byte array), any of which can be null. Tuxedo equivalent: CARRAY.
TypedFML	Buffer type used when the data is self-defined. Each data field carries its own identifier, an occurrence number, and possibly a length indicator. Tuxedo equivalent: FML.
TypedFML32	Buffer type similar to TypeFML but allows for larger character fields, more fields, and larger overall buffers. Tuxedo equivalent: FML32.
TypedXML	Buffer type used when data is an XML based message. Tuxedo equivalent: XML for Tuxedo Release 7.1 and higher.

Perform the Requested Service

Use Java code to express the logic required to provide your service.

Return Client Messages

Use the **`TuxedoReply`** class **`setReplyBuffer()`** method to respond to client requests.

Note: For more detailed information on the **`TuxedoReply`** class, view the WebLogic Tuxedo Connector Javadoc by opening the `index.html` file in the `doc` directory of your WebLogic Tuxedo Connector installation.

Example Service Application

The following provides an example of the TolowerBean service application which receives a string argument, converts the string to all lower case, and returns the converted string to the client.

Listing 3-1 Example Service Application

```
.
.
.
public Reply service(TPServiceInformation mydata) throws TPException {
     TypedString data;
     String lowered;
     TypedString return_data;

     log("service tolower called");

     data = (TypedString) mydata.getServiceData();
     lowered = data.toString().toLowerCase();
```

```
        return_data = new TypedString(lowered);

        mydata.setReplyBuffer(return_data);

        return (mydata);
}
.
..
.
.
```

4 WebLogic Tuxedo Connector Transactions

The following sections provide information on global transactions and how to define and manage them in your applications:

Global Transactions

A global transaction is a transaction that allows work involving more than one resource manager and spanning more than one physical site to be treated as one logical unit. A global transaction is always treated as a specific sequence of operations that is characterized by the following four properties:

- Atomicity: All portions either succeed or have no effect.
- Consistency: Operations are performed that correctly transform the resources from one consistent state to another.
- Isolation: Intermediate results are not accessible to other transactions, although other processes in the same transaction may access the data.

- Durability: All effects of a completed sequence cannot be altered by any kind of failure.

JTA Transaction API

The WebLogic Tuxedo Connector uses the Java Transaction API (JTA) to manage transactions.

Note: For more detailed information on the JTA API, go to http://java.sun.com/products/jta/index.html

Types of JTA Interfaces

JTA offers three types of transaction interfaces:

- Transaction
- TransactionManager
- UserTransaction

Transaction

The `Transaction` interface allows operations to be performed against a transaction in the target Transaction object. A Transaction object is created to correspond to each global transaction created. Use the `Transaction` interface to enlist resources, synchronize registration, and perform transaction completion and status query operations.

TransactionManager

The `TransactionManager` interface allows the application server to communicate to the Transaction Manager for transaction boundaries demarcation on behalf of the application. Use the `TransactionManager` interface to communicate to the transaction manager on behalf of container-managed EJB components.

UserTransaction

The `UserTransaction` interface is a subset of the `TransactionManager` interface. Use the `UserTransaction` interface when it is necessary to restrict access to Transaction object.

JTA Transaction Primitives

The following table maps the functionality of Tuxedo transaction primitives to equivalent JTA transaction primitives.

Table 4-1 Mapping Tuxedo Transaction Primitives to JTA Equivalents

Tuxedo	Tuxedo Functionality	JTA Equivalent
`tpabort`	Use to end a transaction.	**`setRollbackOnly`**
`tpcommit`	Use to complete a transaction.	**`commit`**
`tpgetlev`	Use to determine if a service routine is in transaction mode.	**`getStatus`**
`tpbegin`	Use to begin a transaction.	**`setTransactionTimeout`** **`begin`**

Defining A Transaction

Transactions can be defined in either client or server processes. A transaction has three parts: a starting point, the program statements that are in transaction mode, and a termination point.

To explicitly define a transaction, call the **`begin()`** method. The same process that makes the call, the initiator, must also be the one that terminates it by invoking a **`commit()`** or a **`setRollbackOnly()`**. Any service subroutines that are called between the transaction delimiter become part of the current transaction.

Starting a Transaction

A transaction is started by a call to **`begin()`**. To specify a time-out value, precede the **`begin()`** statement with a **`setTransactionTimeout(`**`int seconds`**`)`** statement.

Note: Setting **`setTransactionTimeout()`** to unrealistically large values delays system detection and reporting of errors. Use time-out values to ensure response to service requests occur within a reasonable time and to terminate transactions that have encountered problem, such as a network failure. For productions environments, adjust the time-out value to accommodate expected delays due to system load and database contention.

To propogate the transaction to Tuxedo, you must do the following:

- Look up a **`TuxedoConnectionFactory`** object in the JNDI.
- Get a **`TuxedoConnection`** object using **`getTuxedoConnection()`**.

Using TPNOTRAN

Service routines that are called within the transaction delimiter are part of the current transaction. However, if **`tpcall()`** or **`tpacall()`** have the flags parameter set to **`TPNOTRAN`**, the operations performed by the called service do not become part of that transaction. As a result, services performed by the called process are not affected by the outcome of the current transaction.

Terminating a Transaction

A transaction is terminated by a call to either **`commit()`** or a **`setRollbackOnly()`**. When **`commit()`** returns successfully, all changes to the resource as a result of the current transaction become permanent. **`setRollbackOnly()`** is used to indicate an abnormal condition and rolls back the any call descriptors to their original state.

In order for a **`commit()`** to succeed, the following two conditions must be met:

- The calling process must be the same one that initiated the transaction with a **`begin()`**
- The calling process must have no transaction replies outstanding

If either condition is not true, the call fails and an exception is thrown.

WebLogic Tuxedo Connector Transaction Rules

You must follow certain rules while in transaction mode to insure successful completion of a transaction.The basic rules of etiquette that must be observed while in a transaction mode follow:

- Processes that are participants in the same transaction must require replies for their requests.
- Requests requiring no reply can be made only if the flags parameter of **`tpacall()`** is set to **`TPNOREPLY`**.
- A service must retrieve all asynchronous transaction replies before calling **`commit()`**.
- The initiator must retrieve all asynchronous transaction replies before calling **`begin()`**.
- The asynchronous replies that must be retrieved include those that are expected from non-participants of the transaction, that is, replies expected for requests made with a **`tpacall()`** suppressing the transaction but not the reply.
- If a transaction has not timed out but is marked abort-only, further communication should be performed with the **`TPNOTRAN`** flag set so that the work done as a result of the communication has lasting effect after the transaction is rolled back.
- If a transaction has timed out:
  - the descriptor for the timed out call becomes stale and any further reference to it will return **`TPEBADDESC`**.
  - further calls to **`tpgetrply()`** or **`tprecv()`** for any outstanding descriptors will return the global state of transaction time-out by setting **`tperrono`** to **`TPETIME`**.

- asynchronous calls can be make with the *flags* parameter of **`tpacall()`** set to **`TPNOREPLY`** | **`TPNOBLOCK`** | **`TPNOTRAN`**.

- Once a transaction has been marked abort-only for reasons other than time-out, a call to **`tpgetrply()`** will return whatever represents the local state of the call, that is, it can either return success or an error code that represents the local condition.
- Once a descriptor is used with **`tpgetrply()`** to retrieve a reply or with **`tpsend()`** or **`tprecv()`** to report an error condition, it becomes invalid and any further reference to it will return **`TPEBADDESC`**.

Once a transaction is aborted, all outstanding transaction call descriptions (made without the **`TPNOTRAN`** flag) become stale, and any further reference to them will return **`TPEBADDESC`**.

Example Transaction Code

The following provides a code example for a transaction:

Listing 4-1 Example Transaction Code

```
public class TransactionSampleBean implements SessionBean {

.....

public int transaction_sample () {

     int ret = 0;
     try {
          javax.naming.Context myContext = new InitialContext();
          TransactionManager tm = (javax.transaction.TransactionManager)
          myContext.lookup("javax.transaction.TransactionManager");

// Begin Transaction
          tm.begin ();

          TuxedoConnectionFactory tuxConFactory = (TuxedoConnectionFactory)
          ctxt.lookup("tuxedo.services.TuxedoConnection");
```

```
// You could do a local JDBC/XA-database operation here
// which will be part of this transaction.
.....

// NOTE 1: Get the Tuxedo Connection only after
// you begin the transaction if you want the
// Tuxedo call to be part of the transaction!

// NOTE 2: If you get the Tuxedo Connection before
// the transaction was started, all calls made from
// that Tuxedo Connection are out of scope of the
// transaction.

          TuxedoConnection myTux = tuxConFactory.getTuxedoConnection();

// Do a tpcall. This tpcall is part of the transaction.
          TypedString depositData = new TypedString("somecharacters,5000.00");

          Reply depositReply = myTux.tpcall("DEPOSIT", depositData, 0);

// You could also do tpcalls which are not part of
// transaction (For example, Logging all attempted
// operations etc.) by setting the TPNOTRAN Flag!
          TypedString logData =
          new TypedString("DEPOSIT:somecharacters,5000.00");

          Reply logReply = myTux.tpcall("LOGTRAN", logData,
          ApplicationToMonitorInterface.TPNOTRAN);

// Done with the Tuxedo Connection. Do tpterm.
          myTux.tpterm ();

// Commit Transaction...
          tm.commit ();

// NOTE: The TuxedoConnection object which has been
// used in this transaction, can be used after the
// transaction only if TPNOTRAN flag is set.
}
          catch (NamingException ne) {
          System.out.println ("ERROR: Naming Exception looking up JNDI: " + ne);
          ret = -1;
}
          catch (RollbackException re) {
          System.out.println("ERROR: TRANSACTION ROLLED BACK: " + re);
          ret = 0;
}
          catch (TPException te) {
          System.out.println("ERROR: tpcall failed: TpException: " + te);
```

```
          ret = -1;
}
          catch (Exception e) {
          log ("ERROR: Exception: " + e);
          ret = -1;
}

          return ret;
}

.....
```

5 Application Error Management

The following sections provide mechanisms to manage and interpret error conditions in your applications:

- Testing for Application Errors
- WebLogic Tuxedo Connector Time-Out Conditions
- Application Event Log

Testing for Application Errors

Note: To view an example that demonstrates how to test for error conditions, see "Example Transaction Code" on page 4-6

Your application logic should test for error conditions after the calls that have return values and take suitable steps based on those conditions. In the event that a function returned a value, you may invoke a functions that tests for specific values and performs the appropriate application logic for each condition.

Exception Classes

The WebLogic Tuxedo Connector throws the following exception classes:

- Ferror: Thrown for errors occurring while manipulating FML.
- TPException: Thrown for errors occurring during a **`tpcall()`** or **`tpacall()`**.
- TPReplyException: Thrown for error occurring during a **`tpgetreply()`** or **`tpdequeue()`**.

Fatal Transaction Errors

In managing transactions, it is important to understand which errors prove fatal to transactions. When these errors are encountered, transactions should be explicitly aborted on the application level by having the initiator of the transaction call **`commit()`**. Transactions fail for the following reasons:

- The initiator or participant of the transaction caused it to be marked for rollback.
- The transaction timed out.
- A **`commit()`** was called by a participant rather than by the originator of a transaction.

WebLogic Tuxedo Connector Time-Out Conditions

There are two types of time-out which can occur when using the WebLogic Tuxedo Connector:

- Blocking time-out
- Transaction time-out.

Blocking vs. Transaction Time-out

Blocking time-out is exceeding the amount of time a call can wait for a blocking condition to clear up. Transaction time-out occurs when a transaction takes longer than the amount of timed defined for it in **`setTransactionTimeout()`**. By default, if a process is not in transaction mode, blocking time-outs are performed. When the *flags* parameter of a a communication call is set to **`TPNOTIME`**, it applies to blocking time-outs only. If a process is in transaction mode, blocking time-out and the **`TPNOTIME`** flag are not relevant. The process is sensitive to transaction time-out only as it has been defined for it when the transaction was started. The implications of the two different types of time-out follow:

- If a process is not in transaction mode and a blocking time-out occurs on an asynchronous call, the communication call that blocked will fail, but the call descriptor is still valid and may be used on a re-issue call. Further communication in general is unaffected.
- In the case of transaction time-out, the call descriptor to an asynchronous transaction reply (done without the **`TPNOTRAN`** flag) becomes stale and may no longer be referenced. The only further communication allowed is the one case described earlier of no reply, no blocking, and no transaction.

Effect on commit()

The state of a transaction if time-out occurs after the call to **`commit()`** is undetermined. If the transaction timed out and the system knows that it was aborted, **`setRollbackOnly()`** returns with an error.

If the state of the transaction is in doubt, you must query the resource to determine if any of the changes that were part of that transaction have been applied to it in order to discover whether the transaction committed or aborted.

Effect of TPNOTRAN

When a process is in transaction and makes a communications call with *flags* set to **`TPNOTRAN`**, it prohibits the called service from becoming a participant of that transaction. The success or failure of the service does not influence the outcome of that transaction.

Note: A transaction can time-out while waiting for a reply that is due from a service that is not part of that transaction.

Application Event Log

The event log is a file to which you can send messages from your clients and services to provide a record of events you consider worth recording.

How to Create an Event log

You can create an event log using **`System.out.println()`**. Create a **`log()`** method that takes a variable of type **char** and use the variable name as the argument to the call, or include the message as a literal within quotation marks as the argument to the call as shown in the example below.

Listing 5-1 Example Event Logging

```
log(“About to call tpcall”);
       try {
              myRtn = myTux.tpcall("TOUPPER", myData, 0);
       }
       catch (TPReplyException tre) {
              log("tpcall threw TPReplyExcption " + tre);
              throw tre;
       }
       catch (TPException te) {
              log("tpcall threw TPException " + te);
              throw te;
```

```
        }
        catch (Exception ee) {
                log("tpcall threw exception: " + ee);
                throw new TPException(TPException.TPESYSTEM,
"Exception: " + ee);
        }
        log("tpcall successfull!");
.

.

.

private static void

log(String s)

{       System.out.println(s);}

.

.

.
```

In this example, a series of log messages are used to track the progress of a **`tpcall()`**.