JAP transaction processing failure scenarios

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

MASTER THESIS

JAP TRANSACTION PROCESSING FAILURE SCENARIOS

Barbara Schwarz

Department of Computer Science
Swiss Federal Institute of Technology (ETH) Zurich
bskoko@ethz.ch

September 22nd 2008 – March 22nd 2009

Supervised by:

Prof. Dr. Gustavo Alonso, ETH Zürich
Dr. Peter Schnorf, Credit Suisse
Ulv Grönquist, Credit Suisse
Dr. Roger Weber, Credit Suisse
Dr. Andreas Geppert, Credit Suisse

Abstract

Current transaction processing in Credit Suisse is implemented as a centralized system. Now the possibility of implementing transaction processing as a distributed system on a Java application platform is being considered. In this new environment transactional applications communicate with each other through remote service calls. In order to prevent high transactional complexity, remote service calls are not executed from within the transaction. In case of a failure, atomicity is not guaranteed any more and needs to be reestablished. Reconciliation logic, however, must not be reinvented by each application over and over again, but provided once and for all by the application platform.

This master thesis provides an overview of possible failure scenarios and analysis their impact on transactional applications. The purpose of this document is to provide the system developers and administrators with information, which is essential for defining standard procedures and mechanism for failure handling.

Acknowledgements

First of all I would like to express my gratitude to Peter Schnorf and his team, for their support and mentoring during the last six months. It was a great honor working with them and I appreciate the time and effort they have invested in me and my master thesis. Special thanks go to Ulv Grönquist and Roger Weber, who always had time and patience for answering my questions.

I would also like to thank Prof. Gustavo Alonso for supervising and guiding me during the work and for giving me the opportunity to write my master thesis in Credit Suisse.

Zürich, March 22nd 2009

Barbara Schwarz

Table of Contents

1 Introduction

Credit Suisse's transaction processing is nowadays mostly done on the Mainframe TP/B (Transaction Processing / Batch) Application Platform. Data is being stored in DB2 and IMS (Information Management System) databases and IMS or CICS (Customer Information Control System) is being used as transaction manager. The Java Application Platform (JAP) is being used for implementing presentation logic and parts of the business logic. Mainframe and JAP communicate in a synchronous manner via CORBA (Common Object Request Broker Architecture) and in an asynchronous manner via WebSphere MQ (Message Queue).

Mainframe transaction applications are written in PL/1 (Programming Language One), which cannot be found in course catalogs of contemporary universities any more. Today, modern, object-oriented programming languages like C# and JAVA have taken its place and the new generations of computer scientists are not very enthusiastic about learning an old problem-oriented programming language, which came out of fashion. PL/1 experts have become a scarce resource. Consequently, Credit Suisse has difficulties finding enough manpower with skills, which are needed for the maintenance and development of the Mainframe Platform applications.

Since Credit Suisse already has a well-established Java application platform, the question has arisen whether transaction processing in Java, in particular on JAP would be feasible. In 2007/2008, a Java Transaction Processing Feasibility Study [1] has been carried out, where performance requirements of the test applications have been exceeded and availability tests have been passed. Motivated by these positive results, Credit Suisse has decided to invest in extensive research of transaction processing on JAP, part of which is also this master thesis. If JAP meets all demands, transaction processing might in the long run be migrated from Mainframe to JAP.

Moving from centralized to distributed transaction processing implies an increase in the number of possible failure scenarios, due to the use of remote procedure calls for communication between the transactional applications. Failure scenarios influencing transactional applications on JAP are being presented and analyzed in this work, for profound knowledge of these is essential for defining standard procedures and mechanism for failure handling.

This master thesis is organized as follows: Chapter 2 provides background information necessary for the understanding of the following work. Chapter 3 introduces Credit Suisse patterns for transactional applications. Chapter 4 presents the failure scenarios, which can occur in patterns and gives a detailed analysis of their development and impact on the application. This chapter also provides guidelines for failure handling. Chapter 5 concludes the work.

2 Background

2.1 JAP

An application platform is a set of integrated technical components, processes and guidelines for the development and operation of similar applications. Hundreds of applications are developed, maintained and operated within Credit Suisse and this has to be done in a cost-effective manner. Cost effectiveness cannot be achieved if applications are built in a full-custom fashion. Instead, common and shared functions of similar applications have to be addressed, designed and implemented once and then reused by many applications. This approach leads to higher productivity, as different developers do not have to search for a solution of the same problem over and over again. Another advantage of application platforms is increased application reliability. By following application platforms' standards and guidelines, developers can concentrate on the business logic and leave a big part of failure handling to the application platform. Detailed description of the JAP platform is provided in Credit Suisse internal documents [3], [4], [5].

The Java Application Platform (JAP) is the standard platform for client/server applications. The architecture and technology of JAP are based on the JEE standard [7], [8].

2.2 Architecture

One architecture unit in JAP consists of a cluster of two or more WLS (WebLogic Server) application servers, a Real Application Cluster (RAC) of two or more Oracle database servers and a shared file system, accessible via the SAN (storage area network). The shared file system contains Oracle database files, WLS, Oracle and MQ transaction logs. The underlying operating system is Sun Solaris. Synchronous communication with Mainframe is realized with CORBA and asynchronous communication with WebSphere MQ message queue.

Server clustering enables scalability and high-availability. The incoming jobs are being distributed equally to all instances (load balancing) and in case of a failure of one instance, another instance can take over the failed instance's jobs (failover). A requirement for load balancing and failover is that the needed information is stored in such a way, that it is accessible by all instances of the cluster. This requirement is met by keeping all data on the SAN.

In order to meet the failover, BCP (Business Continuity Planning) and SOX (Sarbanes-Oxley Act) requirements, JAP applications are deployed on two mirrored tracks residing in different data centers: primary site UH (Üetlihof) and disaster recovery site BGH (Betriebsgebäude Horgen).

The architecture described above is illustrated in Figure 1.

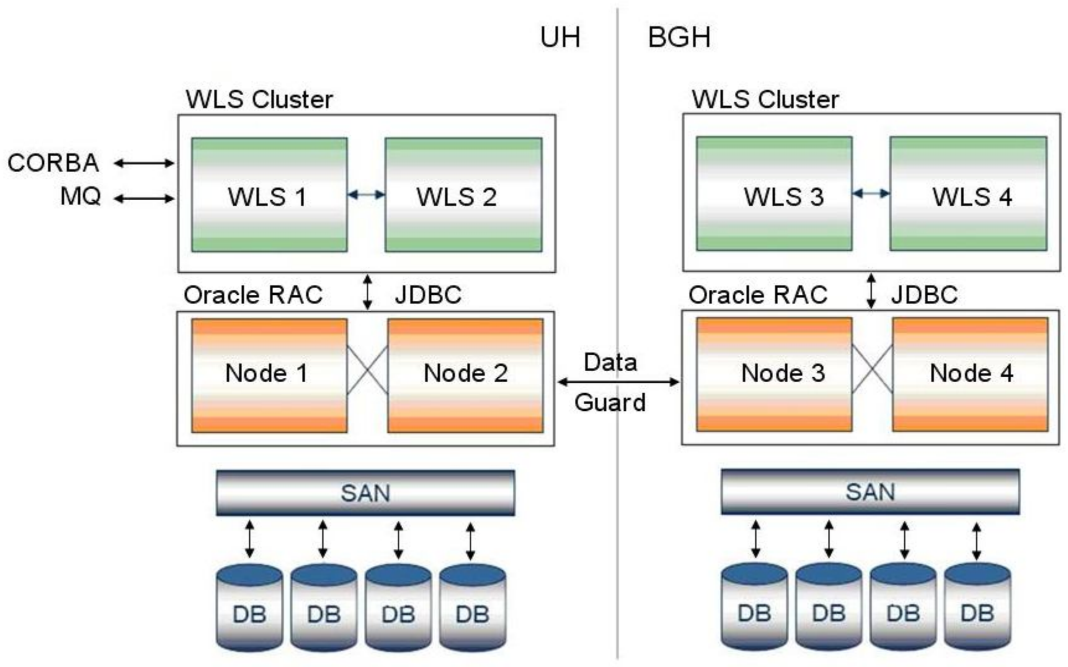


Figure 1: JAP architecture

Credit Suisse has a very complex application landscape, managing which is a difficult and demanding task. A proven instrument for managing complexity is partitioning. With respect to the application landscape, this means breaking up the application landscape down into application domains. Application domains are groupings of applications according to their business coherence, i.e. applications belonging to the same functional area are assigned to a domain. Domains are further divided into sub domains, which are the definition unit for the banking functionality: each sub domain has a defined and assigned set of banking functions (application objects, data, and actions).

Each WLS cluster is associated with one, and only one application sub domain. For capacity reasons, one application sub domain can span more than one WLS cluster. A WLS cluster together with its resources, builds a WLS domain. Each resource belongs to one, and only one WLS domain. An application running within a WLS domain cannot access the resources belonging to another WLS domain directly, but only through remote service calls, through well-defined interfaces.

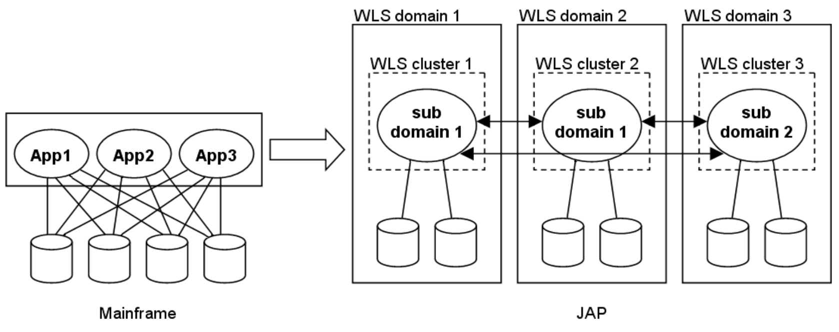


Figure 2: Mainframe and JAP architecture comparison

2.3 Distributed Transactions

A transaction implements a logical unit of work (LUW), a sequence of operations, preserving the ACID properties: Atomicity, Consistency, Isolation and Durability.

Atomicity requires the transaction to be executed following the "all or nothing" rule. If the execution of some parts of the transaction fails, the whole transaction fails.

Consistency requires the resources to be in a consistent state before and after the execution of the transaction, independent of the transaction outcome (success or failure). Consistent state means, that no integrity constraints are violated.

Isolation implies that no transaction sees the intermediate state caused by some other concurrent transaction. In other words, changes done by some active transaction are not visible to other, concurrent transactions until the active transaction completes.

Durability requires that after a successful transaction execution the new, consistent state is persisted and not undone, even in a case of a system failure.

The operations done by a transaction are executed on one or more resources. Most often the resource is a database, but the resource could be any other system managing persistent state. A typical JAP Transaction reads from or writes to a message queue and updates some information in a database.

A transaction is coordinated by a software component called transaction manager. A transaction manager does not interact with the resource directly, but through a component called resource manager. Many applications can request access to the same resource, using services provided by the resource manager.

Depending on the number of involved resource managers there are single-resource and distributed transactions. A single-resource transaction communicates only with one resource whereas distributed transaction communicates with more than one resource. In literature, distributed transactions are sometimes also called global transactions.

2.4 Two-Phase Commit Protocol

Preserving the ACID properties is the basis of reliable transaction processing and it has to be guaranteed even in cases of failure. In practice, consistency is enforced through integrity constraints, which act as a filter determining whether a transaction is acceptable or not. Isolation is enforced by concurrency control, which makes transaction believe there is no other transaction in the system and guarantees serializability.

Atomicity in single-resource transactions is ensured with the help of records in the log files. For guaranteeing atomicity in distributed transactions writing logs is not enough. A distributed transaction executes operations on different resources and it has to make sure that either all resources execute the changes or none of them. Atomicity of a distributed transaction is ensured by the two-phase commit protocol.

The transaction manager runs the two-phase commit protocol with the resource managers of the resources involved in the transaction. It has the role of a resource coordinator. In the first phase of the protocol, the transaction manager asks the participants to vote. Each participant sends its vote, which can be either commit or rollback. In the second phase, depending on the received votes, the transaction manager makes a decision on the outcome of the transaction and propagates it to the participants. If all participants have voted commit, the transaction manager will decide to commit. Otherwise, the transaction manager will decide to roll back the transaction. Each participant has to react accordingly: to commit changes to persistent storage or undo all changes.

2.5 JTA Specification

The JTA, or Java Transaction API, is a Java Enterprise API for managing distributed transactions. It defines a Java binding for the standard XA interface (defined by the X/Open Group): the bidirectional interface between a transaction manager and a resource manager [2]. TheJTA Specification [3] defines the local Java interfaces required for the transaction manager to support transaction management in the Java enterprise distributed computing environment. The transaction manager invokes methods of the JTA `XAResource` interface during the transaction execution. The main methods are `begin()`, `prepare()`, `commit()` and `rollback()`.

The WLS transaction manager implementation follows the JTA specification. The following sequence diagram of Figure 3 illustrates the flow of events in a distributed transaction coordinated by WLS, involving an Oracle database and a MQ message queue. A larger version of the diagram you will find at the end of the document. Reference elements in the sequence diagram are placeholders for the logic of transaction association, connection requests and transaction disassociation. Their content is not necessary for the understanding of the following work. More details, however, you will find in the JTA specification extract in Appendix, Chapter 6.1.

When a transactional Enterprise Java Bean (EJB) is invoked, WLS application server begins the transaction by calling the `begin()` method of the transaction manager interface. Then the EJB code is executed. Within the business logic, the EJB will request a new connection for each resource involved. Behind the scenes, WLS will associate involved resources with the new transaction, add the resources to the list in its transaction log (TLog) (not visible in the diagram) and return a

`Connection` object reference to the EJB. Having the Connection object, the application can now do the actual work on the resources (messages 5-17). In our example, the application reads from the database or updates the database alternatively and then puts a message into the message queue. Database updates are recorded in the log, but not yet visible to other users. Message is effectively inserted into the message queue, but locked to make it invisible towards other clients of the queue. After the work is done, the application closes the JDBC and message queue connections and gives the control to the application server. Application server disassociates the resources from the transaction and gives the control to the transaction manager in order to run the 2PC protocol.

In the first phase of the 2PC protocol the transaction manager asks each involved resource manager, by invoking the method `prepare()`, whether it can guarantee its ability to commit the transaction. If resource manager can commit its work, it records stably the information it needs to do so in its log file (messages 24, 31) and then replies affirmatively with the return value `XA_OK`. A negative reply (`XAException`) reports failure for any reason. After making a negative reply and rolling back its work, the resource manager can discard any knowledge of the transaction.

Depending on the answers from the resource managers, there are two alternatives for the transaction outcome (commit or rollback), which are illustrated in the alternative block. The second phase of 2PC begins as the transaction manager writes the votes of the resource managers and the final decision on the transaction outcome into the TLog. Only now, when the decision is persisted, the transaction manager can issue an actual request to resource managers to commit or roll back the transaction by invoking the method `commit()` or `rollback()` respectively. In case of a commit, the database updates are made persistent on the database and messages inserted in the queue are unlocked, in order to make them visible for other clients. In case of a rollback, the database updates are invalidated (they must not be persisted) and messages are removed from the message queue. Upon receiving the confirmation about the commit/rollback execution from all resource managers, the transaction manager will discard its knowledge of the transaction (delete the transaction from the TLog).

A resource manager can respond to the prepare request by asserting that it was not asked to update resources in this transaction. In this case transaction manager will receive the XA_RDONLY return value and will not record this RM stably in its list of participating resources. The phase 2 dialogue with this resource manager will then not take place.

If transaction manager knows that there is only one resource manager involved in the transaction, it can use the one-phase commit (1PC) protocol. In this optimisation the transaction manager makes its commit request without having made a prepare request. The first phase is herewith omitted. When transaction manager wishes to run the 1PC protocol, it will invoke the `commit()` method with parameter `onePhase` set to true.

The resources participating in a 2PC protocol must be configured as XA-enabled resources in the WLS console. 1PC protocol, on the other hand, can also be run with a non-XA resource. It is recommended, however, to configure each resource only as XA-enabled. With one configuration per resource, failures caused by wrong configuration choice are avoided and one can rely on the 1PC optimization provided by WLS.

One resource manager coordinates many transactions at the same time and each transaction can involve multiple resource managers. A resource manager always interacts with only one transaction manager, but during this interaction, it performs work for many transactions at the same time. Therefore, when transaction manager invokes a `XAResource` method, it needs to specify to which transaction this call refers to. This is realized by the parameter `xid`, which is a general transaction identifier.

One of the primary advantages of Enterprise JavaBeans in transaction processing is the ability to use declarative transaction management. Instead of writing the transactional code within the business logic, which reduces the clarity of the code and is error-prone, the transactional behavior of EJBs can be controlled by using `@javax.ejb.TransactionAtribute` annotation or by modifying the EJB deployment descriptor XML file. Both methods enable changing the transactional behavior of an EJB without changing the business logic. When declarative transaction management is used, the EJB container is responsible for starting, committing or rolling back the transactions based on the directives specified in the EJB code. The container then leverages JTA for transaction management in the background. Using exclusively container-managed transactions on JAP is a guideline already established by Credit Suisse [1]. User managed transactions are forbidden.

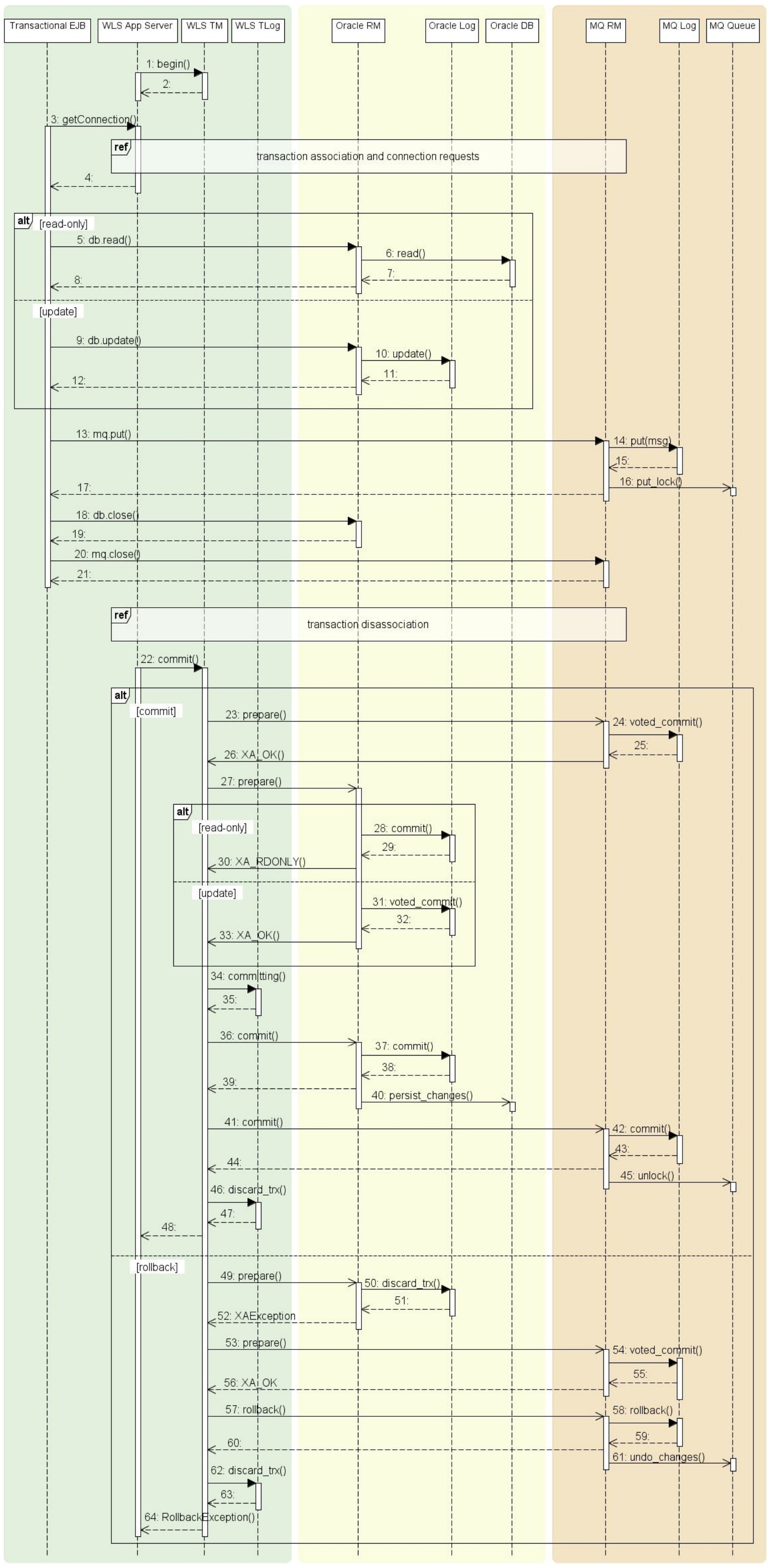


Figure 3: Distributed transaction sequence diagram

3 Patterns

Very often a transaction needs to synchronously invoke a remote service, for the rest of the work depends on its result. The remote service call can be a remote EJB method call (JAP - JAP) or a CORBA call (cross-platform communication, mostly with Mainframe). Remote procedure calls from within a transaction increase the transaction complexity: running a 2PC protocol with remote sub-coordinators is very expensive. Credit Suisse forbids therefore remote procedure calls from within a transaction.

Instead of propagating the transaction context to the remote transaction manager, the current transaction is being suspended before contacting the remote service. The remote transaction manager will start a new transaction, do the operations on its resources and complete its transaction. As soon as the remote call returnes, the primary transaction is resumed. Please note, that the outcome of the primary transaction is independent on the outcome of the remote transaction. Even if the remote transaction has been rolled back, the primary transaction can decide to commit. Another advantage of this model is decoupling of the two transactions. The resources involved in the remote transaction will be locked only during the lifetime of the remote transaction. If we would allow transaction context propagation to te remote system, the remote resources would also be involved in the 2PC protocol, run by the transaction manager of the primary transaction. This way, remote resources would stay locked all the way until the whole transaction completes. Note that such a transaction could span more than only one remote system and block a huge number of resources, making them unnecessarily unavailable for other transactions.

CORBA by default does not inherit the caller's transaction context from the application server and processes the call within a separate remote transaction. The CORBA OTS (Object Transaction Service) protocol, which is able to inherit a transactional context from the existing application server, is not implemented within Credit Suisse. EJB calls, on the other hand, can propagate transactional context, which must be prevented by setting the right transaction attributes. Methods of a remote EJB interface must be annotated with `REQUIRES_NEW` or `NOT_SUPPORTED` transaction attribute, depending on whether the remote service is transactional or not. Local interface methods of a transactional EJB should be annotated with the transaction attribute `REQUIRED`.

In this setting, where transactional applications invoke a remote service only from the outside of the transactional context, all the changes done on the remote system will not be committed by the 2PC protocol run by the transaction manager of the primary transaction. In other words, transaction manager cannot guarantee atomicity anymore. Assuring atomicity in this case, is the responsibility of the application logic. Even if the remote service does not change the remote state, but executes some read-only operations, the application has to be prepared for handling situations when the remote service is not reachable, or not available.

At the moment, most of the transactional applications on JAP make remote procedure calls, for 90% of productive data used by JAP applications is located on Mainframe.

There are three types of transactional applications on JAP: 1. applications that interact only with local resources (belonging to the same WLS domain); 2. applications that make read-only remote service calls; 3. applications that execute updates on remote resources. For each of these JAP application types, Credit Suisse has defined a pattern, which models the application type and provides a detailed description of its typical characteristics. Patterns also point out typical failure scenarios and provide solutions, even implementations for their handling. The main purpose of patterns is to give the application developers better understanding of the applications and to raise their awareness of possible failures.

3.1 Local Resources Pattern

The local resources pattern models a transactional application, which interacts only with local resources, the ones who belong to the same WLS domain. In other words, it models common single-resource and distributed transactions, which were described in Chapter 2.

3.2 Read-only Pattern

The read-only pattern describes how a read-only remote service should be invoked synchronously from within a transactional application and provides a framework for corresponding failure handling. Read-only pattern is an extension of a regular distributed transaction described in Figure 3. Figure 4 shows a sequence diagram describing the flow of events during the execution of a read-only pattern.

On first use the connections to the resources are established and resources are being associated with the transaction. At some time during the transaction, the example application in the diagram wants to invoke a remote, read-only EJB method. The calling method is annotated with the `(REQUIRES_NEW)` transaction attribute. This way, the transaction manager will automatically suspend the current transaction before making the remote service call (messages 5-8). As soon as the remote service call has returned, the application server will resume the transaction (messages 16-19) and the application will continue its work in the original transactional context. It is important to notice that messages 9-15 are not a part of the original transaction. Upon receiving the request, the remote service will start a new transaction, execute the service, deliver the result and close the transaction. In our example, the remote service transaction involves only one resource (Oracle database), so the remote transaction manager will run the 1PC protocol. The remote service will not be involved in the 2PC protocol run by the transaction manager of the primary transaction.

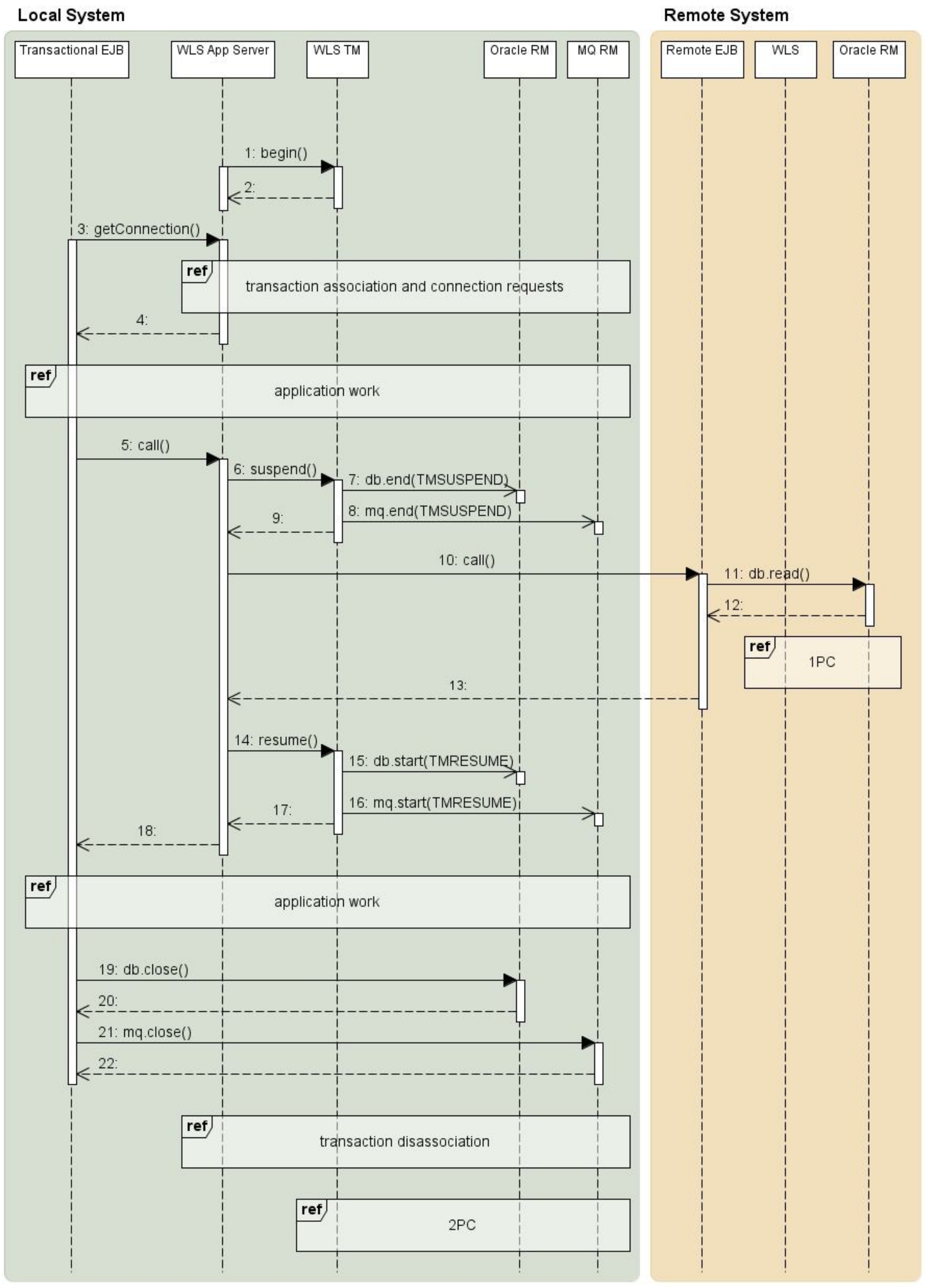


Figure 4: Read-only pattern sequence diagram

3.3 Reservation Pattern

The reservation pattern describes how a remote service changing the remote state should be invoked synchronously from within a transactional application and provides an implementation of pattern specific failure handling.

Imagine a transactional application calling a remote service, which updates a remote state. It could be a simple version of an ATM (Automated Teller Machine). As customer makes a withdrawal request, the ATM starts a new transaction. Customer's account data is stored on a remote system and ATM invokes a remote service to execute the booking. As soon as the procedure has returned without failures, the ATM dispenses the money to the customer. If for example ATM would crash suddenly, after the remote procedure has been executed, but before the ATM had a chance to dispense the money, the withdrawal amount would have been deducted from the customer's account, although the customer has never received the money.

In order to prevent such inconsistent states after failures, updating the remote state is done in two steps: synchronous reservation followed by an asynchronous confirmation. Only after receiving both of them, which happens only if the transaction commits, the remote service will execute the updates. The remote system needs to be able to associate each incoming confirmation with the corresponding reservation from its reservation database. Therefore, each reservation is uniquely identified by a freshly generated UUID (Universally Unique Identifier), which also needs to be contained in the corresponding confirmation.

With this model, in case of a failure on the client side, the transaction will be rolled back, hence the asynchronous confirmation will not be sent. The remote service will not receive a confirmation for the reservation and will therefore not make any permanent changes on the system.

ATM is a typical reservation application in Credit Suisse. The ATM application is running on JAP and all customers' account data is stored on Mainframe.

The exact flow of events in a reservation pattern is shown in the sequence diagram of Figure 5. This example application is a transactional EJB doing some local work on its local database and message queue, and invoking a remote EJB, which executes some updates on the remote system. Before forwarding the `reserve()` method call, the WLS application server suspends the transaction. As remote service is invoked, a new, remote transaction is started and the reservation is saved into the reservations database. In this example there is only one resource involved in the remote transaction, so remote transaction manager will run the 1PC protocol with the database. After closing the transaction, the remote server will return the result to the caller. Upon receiving the result, the local transaction will be resumed. Finally, the application will create a confirmation message and put it into the message queue. Before, after or between the synchronous reservation and asynchronous confirmation the application can optionally do some other work on its local resources. At the end, the transaction manager will run the 2PC protocol with the local resources and application server will close the transaction.

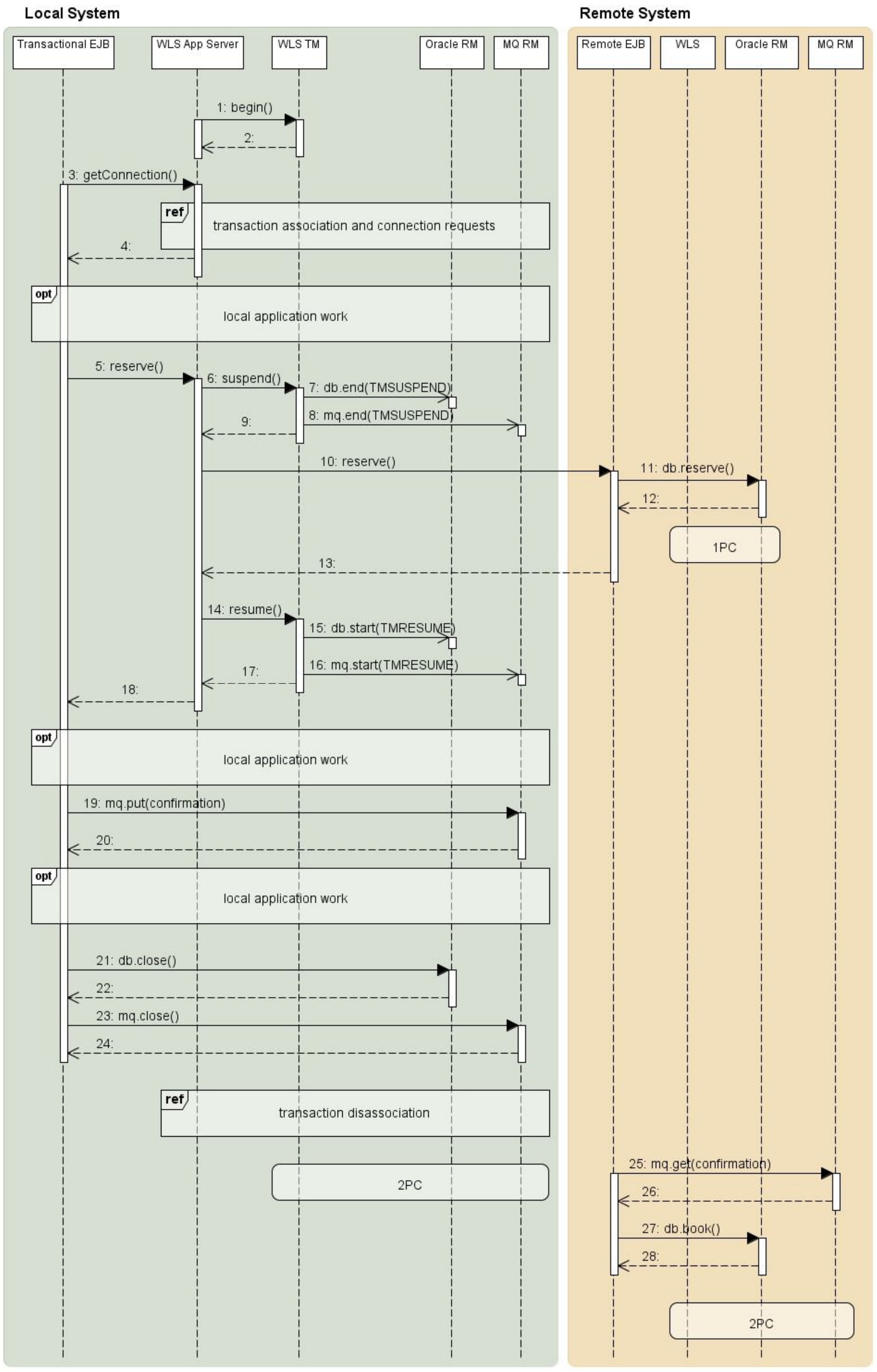


Figure 5: Reservation pattern sequence diagram

4 Failure Scenarios

4.1 Local Resources Pattern

The local resources pattern is logically divided in two phases. In the first phase, the application is executing the business logic. In the second phase, the transaction monitor is running the 2PC protocol. Failures that can occur in these two phases have a large impact on the outcome of the transaction, but are handled differently, so they will be discussed separately.

4.1.1 Timeouts

WLS, Oracle and MQ transactional timeouts shorten the time during which transactions are blocking the resources in case of a failure. They should be configured in a way that they do not abort healthy transactions during their execution, but only transactions, that are not processing their work properly and are blocking the resources. The timeouts setup should also guarantee that during the second phase of the 2PC protocol no automatic heuristic decisions are made. Here is the list of configurable timeouts in the application layer that influence transactions. There are also timeouts defined in underlying layers (for example TCP/IP timeouts), but they are not in the scope of this work.

TimeoutSeconds (default = 30 seconds): WLS
If the transaction has not reached the prepared state after this time, counting from the beginning of the transaction, it is automatically rolled back. It spans the application part of the transaction and the first phase of the 2PC protocol. On JAP this attribute is set to 60 seconds and can be configured on the Domain -> Configuration -> JTA tab of the WLS console (see Appendix, Chapter 6.3 for more details). This timeout can also be set for each EJB individually in the weblogic-ejb-jar.xml descriptor file.

```
<transaction-descriptor>
      <trans-timeout-seconds>20</trans-timeout-seconds>
</transaction-descriptor>
```

AbandonTimeoutSeconds (default = 86400 seconds = 24 hours): WLS
As soon as the transaction is in the prepared state, the second phase of the 2PC protocol starts. During this phase, the transaction manager will continue trying to complete the transaction until all resource managers indicate that the transaction is completed. This timeout defines the maximum period of time a transaction manager is allowed to persist in attempting to complete the transaction. Once this timeout is reached, the transaction will be abandoned. WLS will throw and log a `HeuristicMixedException`.

This timeout can be configured on the Domain -> Configuration -> JTA tab of the WLS console. See Appendix, Chapter 6.3 for more details.

KeepAliveInterval: MQ
This WebSphere MQ attribute defines how often the availability of the channel with WLS is checked. An unavailable channel denotes a failure of WLS. In such case all transactions that have not been prepared yet will be rolled back immediately. Prepared transactions will not be affected by the channel failure – they will stay in prepared state until WLS transaction manager contacts the MQ or manual transaction completion takes place. On JAP the value of the *KeepAliveInterval* attribute is set to 300 seconds, which applies to all existing channels. When not specified, the value of the underlying TCP/IP protocol is used. Note that, after a WLS crash, the messages involved in the transaction will stay locked and herewith not visible to other users for up to 300 seconds. The exact impact of such configuration on the overall system needs to be thoroughly analysed and taken into account. The result might be an adjustment of the standard value of this attribute.

distributed_lock_timeout (default = 60 seconds): Oracle
After this time period, counting from the beginning of the transaction, Oracle resource manager will roll back the transaction if it is not in a prepared state yet. This value can be configured for each resource manager separately and is then valid for all its transactions. On JAP *distributed_lock_timeout* is set to 100.

XAResource.setTransactionTimeout
WLS can propagate the *TimeoutSeconds* attribute to its resources by calling the method `boolean setTransactionTimeout(int seconds)` of the `XAResource` interface. Once set, this timeout value is effective until this method is invoked again with a different value. To reset the timeout value to the default value used by the resource manager, set the value to zero. If a resource manager does not support transaction timeout value to be set explicitly, this method returns false.

The effective implementation of the method defines how the value is used by a resource manager. WebSphere MQ, for example, does not support explicit setting of the transaction timeout. It ignores the propagated value and returns false. Oracle resource manager adapts the value of its *distributed_lock_timeout* attribute to the propagated value, if the propagated value is smaller than the current value of the attribute.

It is not clear yet, if this method should be used. The *TransactionTimeout* attribute can be set for each EJB individually, but when propagated to the Oracle database, the distributed_lock_timeout is set to a new value, which is then valid for all transactions. The consequences of such configuration must be thoroughly analysed.

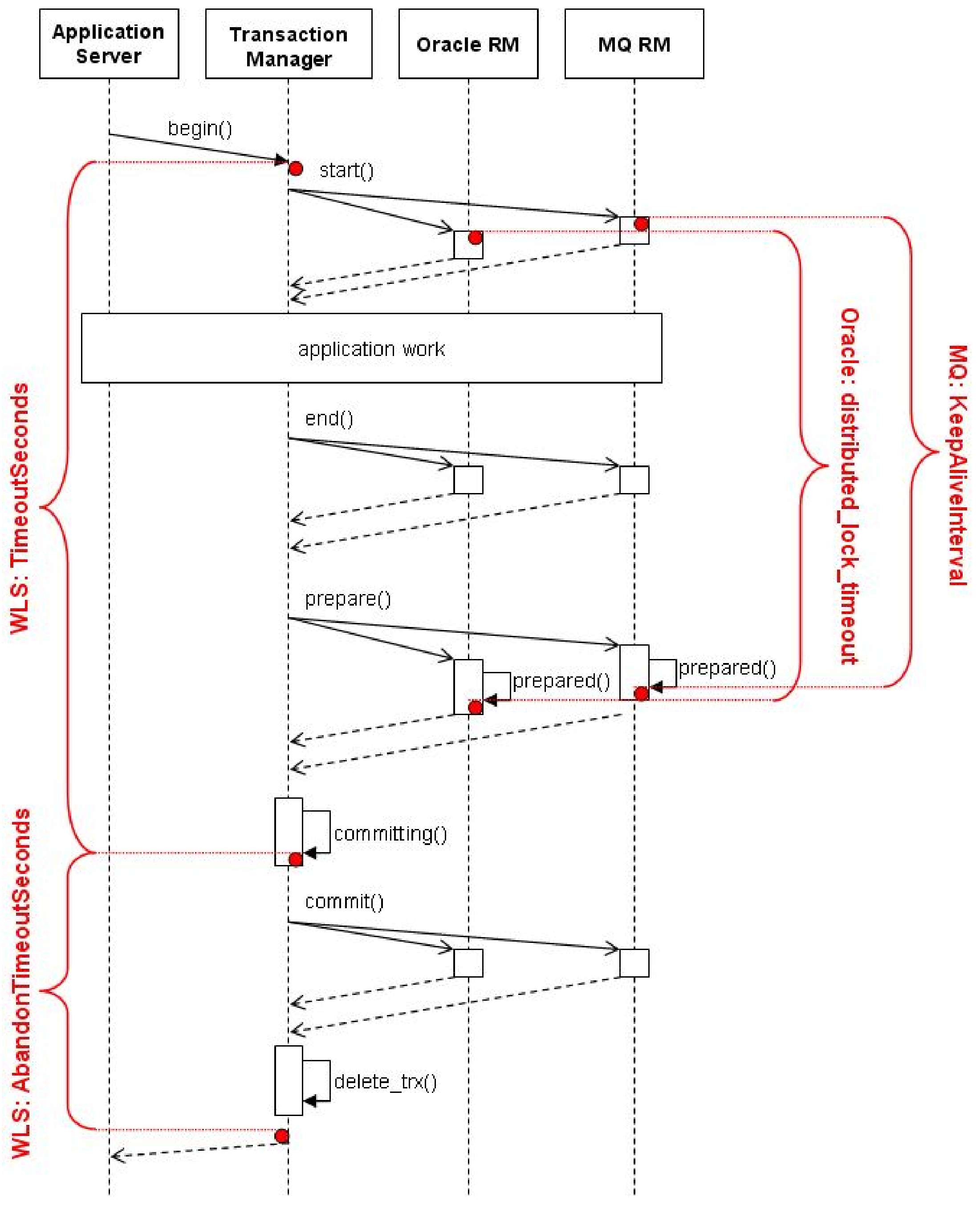


Figure 6: Timeouts overview

4.1.2 Exceptions during the application logic phase

Many things can go wrong during the execution of the business logic: a method call with invalid input parameters, a reference to a non existing object, an attempt to change a record in the database violating integrity constrains, etc. All these errors result in an exception.

Java exceptions are categorized in two groups: system exceptions and application exceptions. System exceptions include `java.lang.RuntimeException`, `java.rmi.RemoteException`, and all its subclasses. They can be thrown by the EJB container, when it detects an internal application server failure. The container handles system exceptions thrown from a bean method automatically: it rolls back the transaction, throws and logs an `EJBException` or `EJBTransactionRolledbackException` (depending on whether the transaction context has been propagated or not) and discards the EJB instance. `EJB*Exception` is a subtype of `RuntimeException`, so it is considered a system exception. In contrast to other system exceptions, `EJBException` cannot be turned into an application exception using the `@ApplicationException` annotation.

All other exceptions, that are not system exceptions, are application exceptions. Application exceptions can, but do not have to roll back the business process. If not stated explicitly with `@ApplicationException(rollback=true)` annotation, an application exception will not cause the transaction to roll back.

It is a Credit Suisse guideline that each transaction throwing an application exception must be rolled back by calling the `setRollbackOnly()` method. Further on, after calling the `setRollbackOnly()` method, an exception has to be thrown or a state has to be returned, informing the caller about the negative outcome of the transaction.

There are three types of application exceptions in JAP: technical, business and other application exceptions. A technical exception should be thrown, when a technical failure prevented the system from executing the work. If the failure was of a business nature, the business exception should be thrown. This distinction between the two types provides some additional information to the caller. Throwing a technical exception tells the caller that the procedure could not be executed because of some internal failure. Throwing a business exception on the other hand, puts the blame of failing on the caller. Other application exceptions are thrown by subsystems (for example `SQLException` thrown by JDBC).

Very generally, an application can catch or not catch an exception. Once the exception is caught, the application can handle it or throw a new exception and pass the problem to its caller. An exception can be handled in three ways. The first option is ignoring the exception, which is a good choice when the result of the failed method is not crucial. If the result is necessary, but there are alternative ways of getting to the information, the application will compensate the failed method call by calling another method. Finally, if the result is necessary and cannot be obtained by any other means, the application might retry the method call. The following decision making tree in Figure 7 illustrates the options described above.

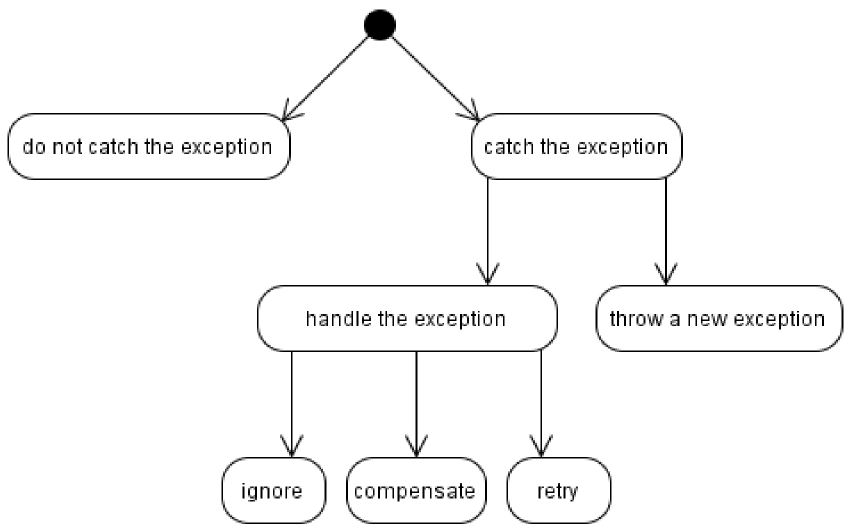


Figure 7: Exception handling decision tree

The following table gives an overview of exception handling recommendations for JAP transactions. Each exception type has been considered separately as well as the options of propagating or not propagating the transaction context. The transaction attributes denote the value, which annotates the EJB throwing the exception. The calling EJB is always transactional. As opposed to `EJBException` and application exceptions, system exceptions caught by an EJB do not originate from another EJB, but from a method of its own.

	REQUIRES_NEW	REQUIRED
system exception	A: Do not catch the exception - let the container rethrow it as `EJBException` and rollback the transaction automatically	A: Do not catch the exception - let the container rethrow it as `EJBTransactionRolledback-Exception` and rollback the transaction automatically
	B: Catch the exception and handle it – the transaction will not be rolled back	B: Catch the exception and handle it – the transaction will not be rolled back
	C: Catch the exception, call `setRollbackOnly()` method and throw an application exception – the transaction will be rolled back	C: Catch the exception, call `setRollbackOnly()` method and rethrow it as an application exception – the transaction will be rolled back
EJBException / application exception	A: Catch the exception and handle it – the transaction of EJB1 will not be rolled back	Catch the exception and rethrow it as an application exception - the transaction of EJB1 will also be rolled back
	B: Catch the exception, call `setRollbackOnly()` method and throw an application exception - the transaction of EJB1 will be rolled back	

Table 1: Exception handling

Figure 8, Figure 9, Figure 10 and Figure 11 visualize the content of Table 1:

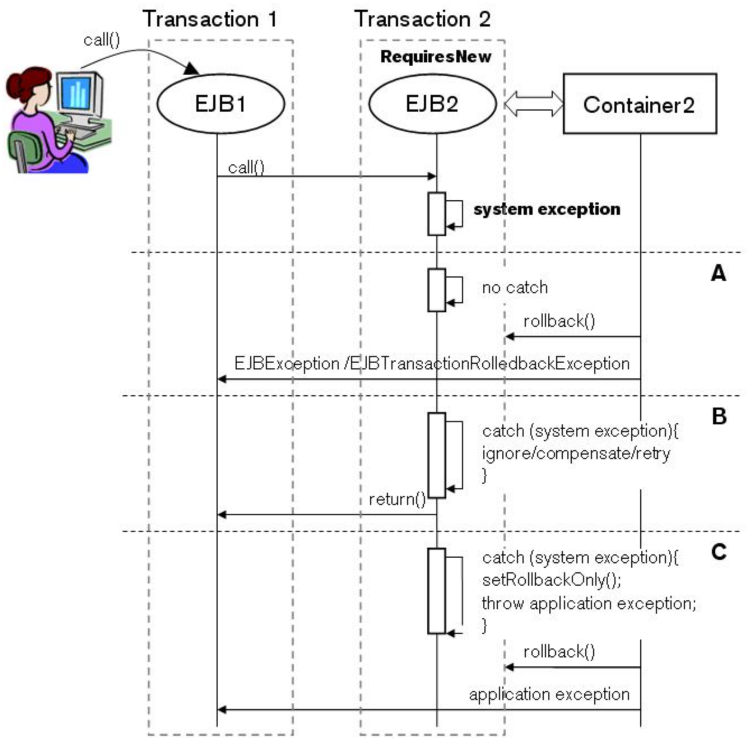


Figure 8: System exception handling - transaction attribute "RequiresNew"

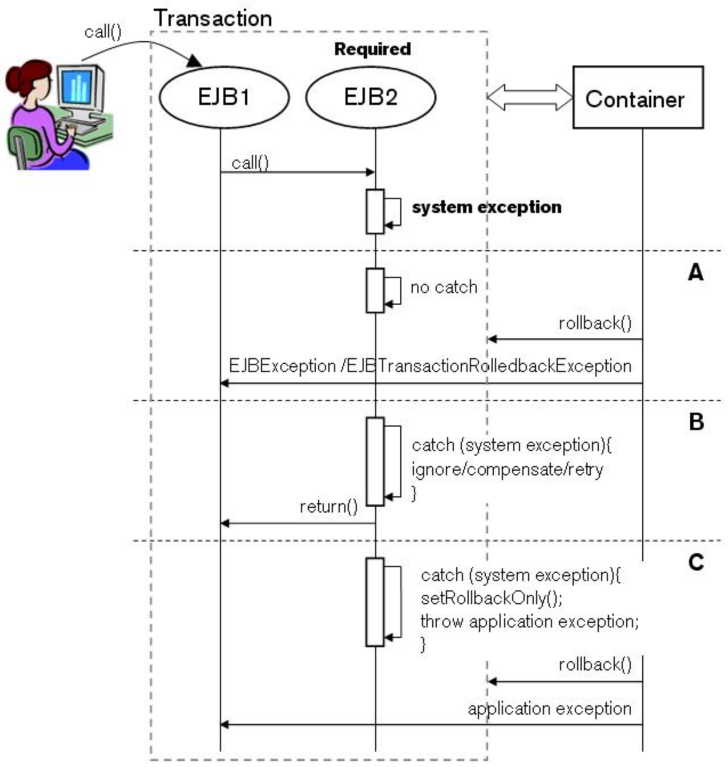


Figure 9: System exception handling – transaction attribute "Required"

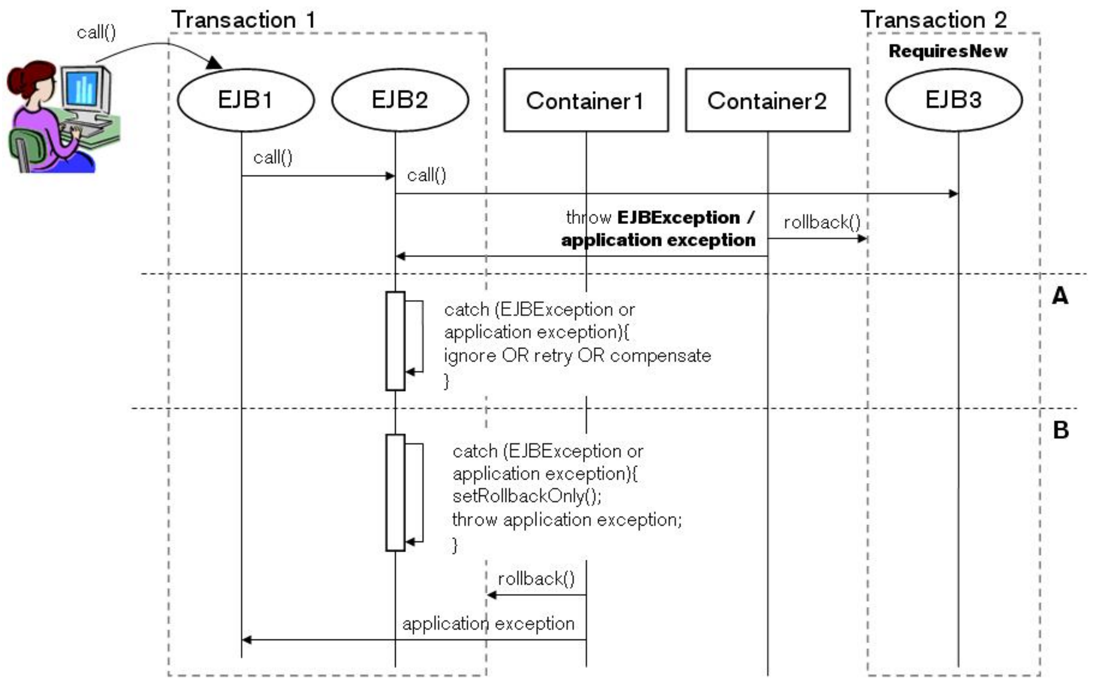

Transaction 1
Transaction 2
RequiresNew
call()
EJB1
EJB2
Container1
Container2
EJB3
call()
call()
throw EJBException / application exception
rollback()
A
catch (EJBException or application exception){ ignore OR retry OR compensate }
B
catch (EJBException or application exception){ setRollbackOnly(); throw application exception; }
rollback()
application exception

Figure 10: EJBException and application exception handling – transaction attribute "RequiresNew"

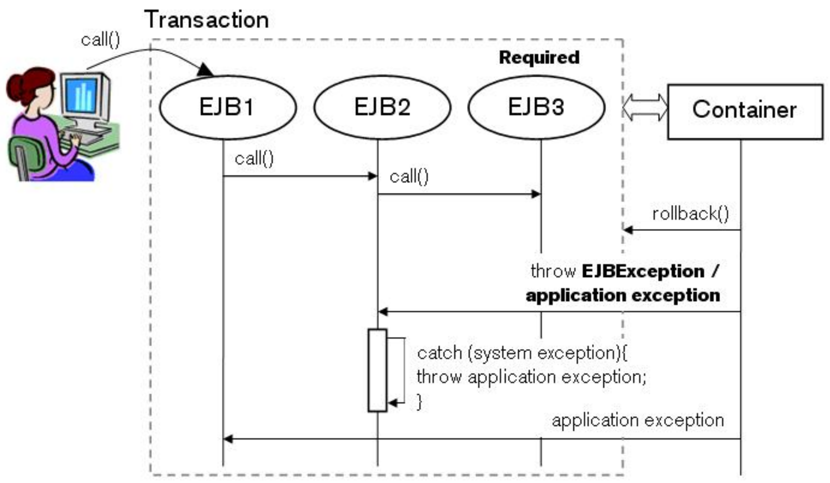

Transaction
call()
Required
EJB1
EJB2
EJB3
Container
call()
call()
rollback()
throw EJBException / application exception
catch (system exception){ throw application exception; }
application exception

Figure 11: EJBException and application exception handling – transaction attribute "Required"

Although it is technically possible, `EJBExceptions` and application exceptions should not be left unhandled or rethrown as `EJBExceptions`. Rethrowing these exceptions as technical or business exceptions gives the caller more information about the context of the failure that occurred, in terms that the caller can understand. In addition, in case of transaction propagation with the `Required` attribute, `EJBExceptions` and application exceptions should not be ignored within a catch block. The transaction of the calling EJB is at that moment already marked for rollback and any changes done after ignoring the exception, will be a part of a transaction which is not valid any more.

As Table 1 shows, exception are handled differently when transactional context is propagated or not, but the difference is very slight. Handling a system exception is in both cases the same. Rethrowing a `EJBException` or an application exception in case of propagated transaction context does not require a `setRollbackOnly()` method call, for the transaction of the calling EJB has already been marked for rollback. An explicit call, however, would do no harm. The developer still has to check if it makes sence to continue computation on behalf of the current transaction or not. This check should be done by calling the `getRollbackOnly()` method of the `EJBContext` interface. Therefore, in order to make the exception handling guidelines clear and brief, we summarize them as follows:

system exception	A: Do not catch the exception - let the container rethrow it as `EJBException` or `EJBTransactionRolledbackException` and roll back the transaction automatically
	B: Catch the exception and ignore it – the transaction will not be rolled back
	C: Catch the exception, call `setRollbackOnly()` method and rethrow it as an application exception – the transaction will be rolled back
`EJBException` / application exception	A: Catch the exception and call the `getRollbackOnly()` method. If the transaction is not marked for rollback, handle the exception
	B: Catch the exception, call the `setRollbackOnly()` method and throw an application exception

Table 2: Exception handling guidelines

4.1.3 Failures during the 2PC phase

Timeouts described in Chapter 4.1.1 are one example of failures that can happen during the execution of the 2PC protocol. Another one is heuristic transaction completion. This happens when a resource manager's work is manually committed or rolled back, independently of the transaction manager. After heuristically completing the transaction, the resource manager is not allowed to forget the transaction, in order to be able to inform the transaction manager about the situation as soon as it is contacted again. If the manual decision does not match with the decision the transaction manager has made, the resource will be left in an inconsistent state and the ACID properties will not be preserved.

Problems can also occur, when one of the resource managers fails. The damage done depends very much on the exact moment during the 2PC, where the failure happened. In some cases, the transaction manager can roll back the transaction and lead it to the consistent state, in some cases not. If the transaction ends with an inconsistent state, manual intervention is needed for restoring the consistence.

A failure of the transaction manager is also a big challenge for atomic commitment. Also here, the impact of the failure on the outcome of the transaction is very dependent on the exact moment when the failure happened.

The goal of this work is not to deliver an extensive list of all possible failures, but to give an overview of the faulty states the transaction can end in. The set of possible failures is huge, but many of them are handled in the same way and they cause the transaction to end in the same state. In order to define the end states and understand how they are reached, I have analysed the interaction of

TransactionManager and XAResource JTA interfaces. WLS transaction manager implements the TransactionManager interface, while WebSphere MQ and Oracle resource managers implement the XAResource interface.

Each failure in the system that cannot be repaired by a single component results in an exception. If the transaction manager and its resource managers cannot repair the failure through their interaction, the transaction will be affected. The question is: Which exceptions are there, and which of them are hazardous for the transaction and under which circumstances.

XAException is an exception thrown by a resource manager to inform the transaction manager of an error encountered during transaction processing. The errCode parameter gives a description of the error cause. You can find a complete list of all possible error codes according to the JTA specification in Appendix, Chapter 6.2. Here are the ones relevant for this work:

XA_RB*
The resource manager has rolled back the transaction work and has released all held resources. WLS transaction manager treats all XA_RB* error codes equally, it does not distinguish between them.

XAER_RMERR, XAER_RMFAIL
These error codes denote an error on the resource manager, which makes it unavailable. The action requested by the transaction manager may or may not have been executed. The WLS transaction manager treats these two error codes equally and therefore they are summarized as XAER_RM* in this work. XAER_RM* will be returned in cases when all RAC nodes have failed and there is not a single healthy RAC node, which can serve the transaction requests.

XAER_NOTA
The resource manager returns this error code if the transaction manager has made a request for a transaction with an xid, which the resource manager does not recognize.

XAER_HEUR* = XAER_HEURCOM, XAER_HEURRB, XAER_HEURMIX, XAER_HEURHAZ
The resource manager returns the XAER_HEUR* error codes after completing the transaction work heuristically: XAER_HEURCOM if the heuristic decision was commit, XAER_HEURRB if the heuristic decision was rollback, XAER_HEURMIX if some parts of the transaction work have been committed and some rolled back and XAER_HEURHAZ if the transaction work might have resulted in a heuristic mix, but resource manager cannot confirm the mixed outcome.

XAER_INVAL, XAER_PROTO
These two error codes are never treated separately, but are handled in the last catch block with all other XAExceptions.

Following are the most important methods of the XAResource interface, which can be invoked by the transaction manager during the 2PC protocol:

int prepare(Xid xid) throws XAException

Transaction manager calls this method to request a resource manager to prepare for committing any work done on behalf of the transaction specified by xid (global transaction identifier). The return value indicates the resource manager's vote on the outcome of the transaction: XA_RDONLY or XA_OK. After returning XA_RDONLY the resource manager may release all resources and forget about the transaction. If all resource managers vote with XA_RDONLY, the transaction manager will not run the second phase of 2PC, but commit the transaction immediately. If any resource manager returns a XAException, no matter of which error code it contains, the transaction manager will roll back the transaction.

Possible error codes: XA_RB*, XAER_RM*, XAER_NOTA, XAER_INVAL, XAER_PROTO

Error code	**Best outcome**	**Worst outcome**
any	RollbackException	HeuristicMixedException

void commit(Xid xid, Boolean onePhase) throws XAException

Transaction manager calls this method to request the resource manager to commit the work done on behalf of the transaction specified by xid. Any changes made to resources on behalf of this transaction are made permanent and resource locks are released.

Possible error codes: XA_HER*, XAER_RM*, XA_RB*, XAER_NOTA, XAER_INVAL, XAER_PROTO

XA_RB* error codes can be returned only if the transaction manager is running the 1PC protocol (onePhase parameter is set to true).

If a resource manager returns the XAER_HEURCOM error code, the heuristic decision of the resource manager matches the decision of the transaction manager, so the transaction manager will call the forget() method on this resource manager and lead the transaction to normal commit. The transaction will end in a consistent state.

If some resource manager returns the XAER_HEURRB error code, the transaction manager cannot lead the transaction to a consistent state any more and has to throw a HeuristicMixedException. It will not call the forget() method on this resource manager, for the transaction context needs to be saved for the subsequent manual intervention.

If some resource manager returns a XAER_RM* error code, the transaction manager cannot lead the transaction to a consistent state until this resource manager is available again. Note that other resource managers will have committed their work already and have forgotten the transaction. The transaction manager will try calling the commit() method three times in a row on the failed resource manager. If the method call still does not succeed, transaction manager will throw a SystemException, which will abort the application. WLS transaction manager, however, will keep trying calling the `commit()` method every 60 seconds until the *AbandonTimeoutSeconds* timeout is reached. Then the `forget()` method is called and `HeuristicMixedException` thrown and logged.

Any other `XAException` will make the transaction manager throw a `HeuristicMixedException` and leave the transaction in an inconsistent state. Manual intervention for restoring the consistent state is then necessary.

Error code	**Best outcome**	**Worst outcome**
`XAER_RM*`	commit	HeuristicMixedException
`XAER_NOTA` `XAER_HEURCOM`	commit	commit
other	HeuristicMixedException	HeuristicMixedException

`void rollback(Xid xid) throws XAException`

Transaction manager calls this method to request the resource manager to roll back the work done on behalf of the transaction specified by xid. Any resources held by the resource manager for this transaction are released and all outstanding updates are invalidated.

Possible error codes: `XA_HER*, XAER_RM*, XAER_NOTA, XAER_INVAL, XAER_PROTO`

If a resource manager returns the `XAER_HEURRB` error code, the heuristic decision of the resource manager matches the decision of the transaction manager, so the transaction manager will call the `forget()` method on this resource manager and lead the transaction to normal roll back. The transaction will end in a consistent state.

If some resource manager returns the XAER_HEURCOM error code, the transaction manager cannot lead the transaction to a consistent state any more and has to throw a `HeuristicMixedException`. It will not call the `forget()` method on this resource manager, for the transaction context needs to be saved for the subsequent manual intervention.

If some resource manager returns a XAER_RM* error code, the transaction manager cannot lead the transaction to a consistent state until this resource manager is available again. Note that other resource managers will have rolled back their work already and have forgotten the transaction. The transaction manager will try calling the `rollback()` method three times in a row on the failed resource manager. If the method call still does not succeed, transaction manager will throw a `SystemException`, which will abort the application. Transaction manager, however, will keep trying calling the `rollback()` method every 60 seconds, until the *AbandonTimeoutSeconds* timeout is reached. Then the `forget()` method is called and `HeuristicMixedException` thrown and logged.

Any other `XAException` will make the transaction manager throw a `HeuristicMixedException` and leave the transaction in an inconsistent state. Manual intervention for restoring the consistent state is then necessary.

Error code	Best outcome	Worst outcome
XAER_RM*	RollbackException	HeuristicMixedException
XAER_NOTA XAER_HEURRB	commit	commit
other	HeuristicMixedException	HeuristicMixedException

Xid[] recover(int flag) throws XAException

Transaction manager calls this method to obtain a list of `xids` for which the resource manager is in prepared or in heuristically completed state. WLS TM will invoke this method only in a situation, when it does not know in which status the resources are. This happens when the transaction manager crashes before writing its decision into the TLog. After the restart, the transaction manager reads the TLog in order to detect the state in which the transaction was before the crash. It sees then the list of all resource managers involved in the transaction and notices that the decision about the outcome of the transaction has not been made yet. At this moment, transaction manager decides to roll back the transaction. It calls the `recover()` method on every resource manager involved in the transaction and then sends a rollback request to each resource manager, which returned the `xid` of the current transaction.

If WLS crashes after writing its decision into the TLog, it will read the decision from the TLog upon restart and continue leading the transaction towards this outcome.

The return value is a list of zero or more xids. It is the transaction manager's responsibility to ignore the xids that do not belong to it.

Possible error codes: `XAER_RM*, XAER_INVAL, XAER_PROTO`

It has not been ascertained yet, how WLS transaction manager reacts to XAExceptions after this method call. The likelihood for such a failure scenario, where a resource manager fails very shortly after the WLS has crashed, is very slight. However, it is still necessary to analyse this case and define the impact it can have on the transaction outcome.

void forget(Xid xid) throws XAException

A resource manager that heuristically completes work done on behalf of the transaction must keep track of the transaction along with the heuristic decision until told otherwise (by transaction manager or by human, manual intervention). Transaction manager then calls `forget()` method to permit the resource manager to erase its knowledge of the `xid` transaction. Any effort to contact the resource manager concerning the transaction specified with `xid` upon successful return of the `forget()` method will result with resource manager returning the `XAER_NOTA` error code.

If, after a heuristic decision, the transaction ends in an inconsistent state, consistency needs to be restored manually. The information about the transaction held by the resource manager is crucial for successful manual recovery. In such case, transaction manager is not allowed to call the `forget()` method. This can and must be prevented by setting the *ForgetHeuristics* attribute on the JTA panel of the WebLogic Console to false (see Appendix, Chapter 6.3).

Possible error codes: `XAER_RM*, XAER_NOTA, XAER_INVAL, XAER_PROTO`

Transaction manager will ignore all `XAExceptions,` no matter of the error code. If `XAException` has been thrown, the resource manager might have or might have not forgotten the transaction. If the transaction has been completed heuristically, the person who made the heuristic decision has to make sure that the transaction is not visible any more.

The logic of the WLS transaction manager during the 2PC protocol described above is depicted in the state diagram of Figure 12. A larger print of the diagram you will find at the end of the document. This model of the XA protocol is a very useful tool for getting an overview of the possible failure scenarios and their impacts on the transaction outcome. The most interesting part of the diagram is the red area on the right, containing failure scenarios, which end in an inconsistent state. In such a case, manual intervention is necessary.

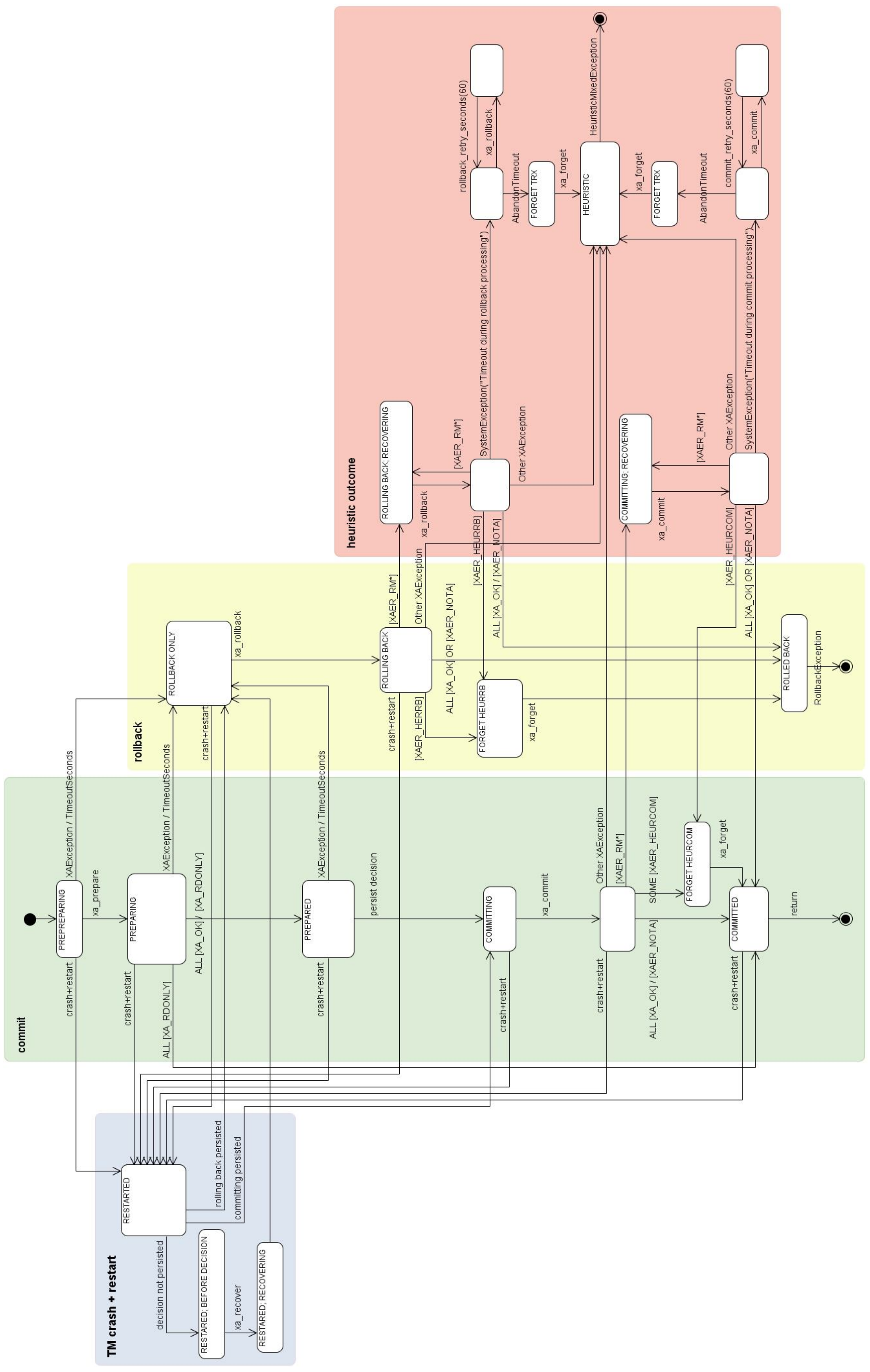


Figure 12: WLS transaction manager state diagram (2PC)

4.2 Read-Only Pattern

Read-only pattern is an extension of a distributed transaction described in Chapter 2. All failure scenarios occurring during the 2PC protocol described in Chapter 4.1 also apply to the read-only pattern, but the additional remote service call entails some new failure scenarios.

The failures from the remote service call can be categorized in two groups:

1) Calls that result with an exception
2) Calls that return an error code
3) Calls that do not return within a predefined time

When a remote service call returns an exception or does not return at all within a specified time window, the application can decide to execute the remote call once again and hope for a better outcome. Since the read-only service does not change the remote state, executing the service multiple times will also not change it. However, there is one side effect that developers have to be aware of. Each retry call will be executed within a separate remote transaction. Hence the results of multiple consecutive calls might differ, for some other remote transaction might have changed the remote state between the calls. Note that the isolation property is not preserved.

Retrying the remote service call however is not always the right solution. After receiving a deadlock exception, retrying after some reasonable period is a very sinful reaction. If the remote service call returns a fatal error on the other hand, the retry call will not make much sense, for the service will most probably not have recovered yet. Even in case of a timeout, consecutive retries might not be the best solution. Timeouts happen mostly when the service is overloaded. Imagine all applications requesting the same, overloaded service running in a time out and retrying the calls consecutively. The situation would get even worse: the service would get even more overloaded and the server would get overloaded too. One suggestion for improvement could be exponential growing retry intervals and even more important, only a finite number of retries. One could also observe the timeout occurrence frequency and after a certain threshold decide to declare the service unavailable and prevent future calls.

4.3 Reservation Pattern

As described in Chapter 3.3, the reservation pattern application executes an update on a remote system in two steps: synchronous reservation followed by an asynchronous confirmation. Updates executed on the local and remote system in the reservation pattern are not a part of the same transaction.

4.3.1 Crash or rollback after remote service call

When a remote service call fails, if the application knows that the service has not been executed, the application can try to execute the remote service again and commit, or it can decide to roll back the transaction by calling the `setRollbackOnly()` method. In the opposite case, when the local transaction decides to rollback or the application server crashes after the remote service call was successfully executed, the application will be in an inconsistent state. In both cases (application server crash or rollback), the local updates will be lost, but the reservation will be persisted. If such a failure happens before the remote service call, the consistency is not endangered.

In case of an error-free execution of the reservation pattern, the application sends the confirmation message and commits the transaction. At this moment, the remote service, however, has not processed the confirmation message yet and hence has not committed the updates. Anyway, new service requests during this period, will see the new, consistent state, which the service combines from the current (inconsistent) state and the changes announced in the reservations database. This transition period will last until the remote service reads the confirmation message from its message queue, matches the confirmation with the corresponding reservation and executes the final action.

In case of a successful commit, this is not a problem. But what happens, when the application decides to roll back the transaction or application server crashes?

The remote service now has the reservation in its database, but the confirmation is not there yet. When a new request arrives, the service will simulate the "new" state, which it assumes to be valid, any time the confirmation arrives. The service cannot know that the transaction has been rolled back and that the confirmation will not arrive at all. The local and remote state are inconsistent.

At the first moment, it seems like we have not solved the problem, which the simple implementation of the ATM had. But this is not the case. The inconsistency after such a failure cannot be avoided, but the duration of the inconsistency period can be shortened. This is done by introducing a time to live attribute for reservations. If a reservation does not get confirmed within this period, it will be deleted from the database and the updates will never take place. The time to live attribute can be configured for each service individually, so that each service can fulfill its own business requirements. Too short time-to-live attribute configuration might lead to a "false negatives" situation: the reservation was made, the confirmation has been sent, but the time-to-live is so short, that the confirmation message does not always reach the remote service during the reservation's life time. Handling of such a failure is application specific and depends heavily on the business requirements. If, on the other hand, the time-to-live attribute is too long, the inconsistent state after a failure will last longer than necessary and affect the customer's convenience.

In the real Credit Suisse ATM application, the reservations can live up to a couple of days. Imagine a client with 150 CHF on his account making a withdrawal of 150 CHF. During the transaction processing the ATM crashes after the reservation has been processed. The client sees that this machine is not working any more and goes to another ATM. He tries again. This time however, the ATM will decline the transaction, because the balance is too low, namely 0 CHF. This client will not be able to withdraw his money, until the reservation time to live has elapsed or a bank operator manually cleans up.

Another example is a unmanned gas station. Before being able to use the gas pump, the client must insert his bank card into the machine, which then makes a reservation for 150 CHF on the client's account. After refueling, the confirmation, including the effective amount the client has spent, is sent. The client's balance, however, is lower for 150 CHF. Reservations and confirmations are matched once a day, on business days. If the transaction has taken place on a Saturday, the balance will be inconsistent until next Monday.

However, ATM withdrawals are typically relative small amounts and this kind of failures is very rare.

4.3.2 Failures during 2PC

Failures that can occur during the 2PC protocol execution have been thoroughly analyzed in Chapter 4.1.3. Also their impact on the transaction outcome of the local resources pattern and read-only pattern has been discussed. Such failures within a reservation pattern cause much more trouble.

The application executes its work and decides if it wants to commit or roll back the transaction. Then the transaction manager takes over and tries to lead the transaction to the final state, as requested by the application. If failures occur, the transaction manager might not be able to achieve its goal and the transaction might end in one of the following states: rolled back or mixed outcome. At this point in time, the transaction context does not exist any more. The application server will inform the caller of the application about the failure by throwing an appropriate exception: `EJBTransactionRolledbackException` if the transaction has been rolled back or `SystemException` in case of a mixed outcome. In such case, the caller does not know in which state the remote system really is: Have both, reservation and confirmation, been executed? Or just the reservation? Or none of them?

4.3.3 Reservation timeout

As in read-only pattern, the remote service call in reservation pattern can result in an exception or the call may not return at all. The remote service call of the reservation pattern, in contrast to read-only pattern, updates the remote state and is therefore not idempotent. If the application cannot decide on the success of the remote service call, it will mark the transaction for rollback and inform the caller about the failure.

4.3.4 Retry

In the last three chapters, different failure scenarios, typical for the reservation pattern, have been presented. The question now is: how should the caller react to these failures? Can the pattern be executed once again? For most of the applications, the inconsistency after a failure is acceptable, but the impact of the inconsistency grows with number of unsuccessful executions. How many retries can business logic tolerate? Answers to these questions are very application specific and need to be seriously considered by developers.

4.3.5 Cancel function

For most of the existing applications in Credit Suisse, which implement the reservation pattern, the transient inconsistent state after a failure is not an issue. For the other applications, the reservation pattern framework offers an additional feature, which can shorten the inconsistency period – the cancel function.

The cancel function can be explicitly enabled by calling the right method on the reservation pattern framework interface. The framework will then automatically generate a wrapper, an intermediate element between the transactional EJB, which implements the reservation pattern, and its caller. In case of a failure, it is the wrapper's responsibility to catch all exceptions thrown by the EJB and its container, cancel the reservation and then forward the exceptions to the caller. The reservation is canceled in a separate transaction, asynchronously, by sending a cancelation message through the message queue. The cancel function is executed only once. In case of failures, no further attempts will be made.

In order to spare application developers from generating and managing UUIDS, the reservation pattern framework takes over that work. This is also necessary for the cancelation execution. The wrapper needs to have access to the used UUIDS, for the cancelation message needs to contain the same UUID as its corresponding reservation.

Please note, that the remote service can never receive a confirmation and a cancelation for a certain reservation. It either receives a confirmation, a cancelation or none of them. The cancelation will be triggered only if an exception has been thrown and an exception implies that the transaction has been rolled back. Transaction rollback further implies that the confirmation message could not have been sent. Also note that the failure, which aborted the confirmation, might also cause the cancelation to fail. Hence, when the caller receives an exception, it cannot assume that the reservation has been canceled, although the framework was told to do so. Application developers therefore need to keep in mind, that the cancel function is a best-effort service.

5 Conclusion

There are different teams within Credit Suisse working on JAP transaction processing. Although they work on the same project, they all have different perspectives. The lack of documentation and communication, in addition, results with each team having its own definition of the interfaces and protocols between the components. And sometimes, these definitions do not match.

In order to be able to define the failure scenarios, we had to understand the implementation of the 2PC protocol first. The documents on the implementation of the XA protocol in WLS, Oracle and MQ are internal, non-disclosure documents, not available to Credit Suisse employees. Through numerous interviews with external WLS, Oracle and MQ specialists in Credit Suisse, we have collected, merged, filtered and summarized the information, which has now been published in this master thesis. Summarizing this information and making it available was the first and necessary step towards the understanding of transaction processing on JAP.

The WLS transaction state diagram, the model of the 2PC protocol implementation is a good visual tool for analyzing the 2PC failure scenarios. Looking at the diagram, one can easily define failure scenarios, which lead to an inconsistent transaction state and require manual handling. For such cases, standardized procedures and mechanisms for failure handling need to be designed.

Next step towards the goal of building an operational model is answering the following questions: What information is necessary for manual recovery and must therefore be provided to the operator? Where does this information reside and how is it accessible? Which log record patterns indicate an occurrence of a specific failure? How can the log information be accessed by the monitoring system? Which logs need to be observed? In case of a failure detection, who and how should be informed?

The knowledge summarized in this master thesis might be a helpful information source for implementing automatic failure handling system as a part of the JAP platform.

6 Appendix

6.1 JTA: Transaction Association and Connection Requests

This session provides a brief walkthrough of how an application server may handle a connection request from the application. The figure that follows illustrates the usage of JTA. The steps shown are for illustrative purposes, they are not prescriptive:

1. Assuming a client invokes an EJB bean with a TX_REQUIRED transaction attribute and the client is not associated with a global transaction, the EJB container starts a global transaction by invoking the `TransactionManager.begin` method.

2. After the transaction starts, the container invokes the bean method. As part of the business logic, the bean requests for a connection-based resource using the API provided by the resource adapter of interest.

3. The application server obtains a resource from the resource adapter via some *`ResourceFactory.getTransactionalResource`* method.

4. The resource adapter creates the *`TransactionalResource`* object and the associated `XAResource` and `Connection` objects.

5. The application server invokes the `getXAResource` method.

6. The application server enlists the resource to the transaction manager.

7. The transaction manager invokes `XAResource.start` to associate the current transaction to the resource.

8. The application server invokes the `getConnection` method.

9. The application server returns the `Connection` object reference to the application.

10. The application performs one or more operations on the connection.

11. The application closes the connection.

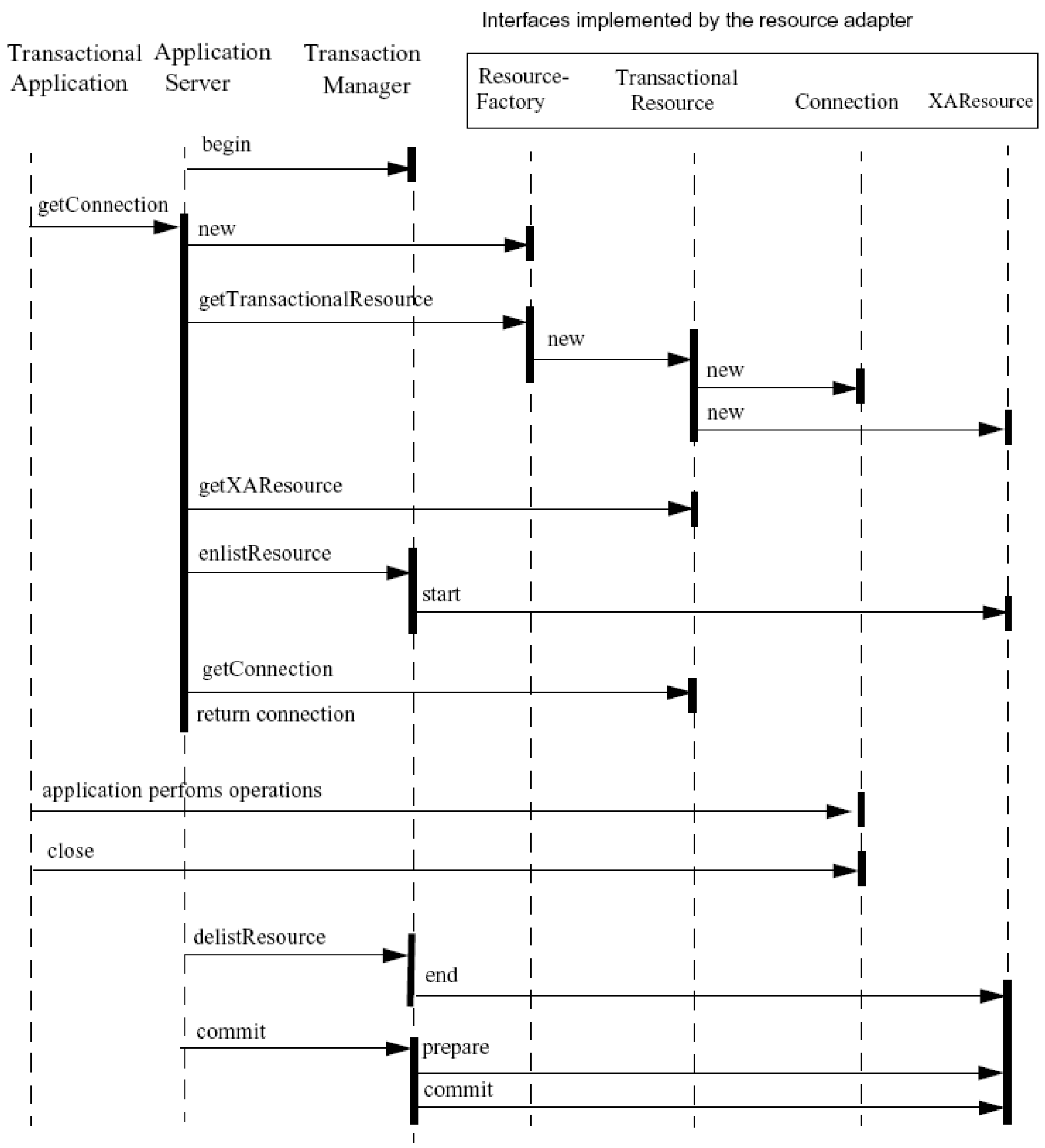
12. The application server delists the resource when notified by the resource adapter about the connection close.

13. The transaction manager invokes `XAResource.end` to disassociate the transaction from the `XAResource`.

14. The application server asks the transaction manager to commit the transaction.

15. The transaction manager invokes `XAResource.prepare` to inform the resource manager to prepare the transaction work for commit.

16. The transaction manager invokes `XAResource.commit` to commit the transaction. This example illustrates the application server's usage of the `TransactionManager` and `XAResource` interfaces as part of the application connection request handling.



6.2 XAException error codes

```
public class javax.transaction.xa.XAException extends
java.lang.Exception
{
public XAException();
public XAException(String s);
public XAException(int errCode);
}
```

XA_RBBASE
`public final static int XA_RBBASE = 100`
The inclusive lower bound of the rollback code.

- **XA_RBROLLBACK**
`public final static int XA_RBROLLBACK = XA_RBBASE`
The rollback was caused by an unspecified reason.

- **XA_RBCOMMFAIL**
`public final static int XA_RBCOMMFAIL = XA_RBBASE + 1`
The rollback was caused by a communication failure.
Java Transaction API

- **XA_RBDEADLOCK**
`public final static int XA_RBDEADLOCK = XA_RBBASE + 2`
A deadlock was detected.

- **XA_RBINTEGRITY**
`public final static int XA_RBINTEGRITY = XA_RBBASE + 3`
A condition that violates the integrity of the resources was detected.

- **XA_RBOTHER**
`public final static int XA_RBOTHER = XA_RBBASE + 4`
The resource manager rolled back the transaction branch for a reason not on this list.

- **XA_RBPROTO**
`public final static int XA_RBPROTO = XA_RBBASE + 5`
A protocol error occurred in the resource manager.

- **XA_RBTIMEOUT**
`public final static int XA_RBRBTIMEOUT = XA_RBBASE + 6`
A transaction branch took too long.

- **XA_RBTRANSIENT**
`public final static int XA_RBTRANSIENT = XA_RBBASE + 7`
May retry the transaction branch

- **XA_RBEND**
`public final static int XA_RBEND = XA_RBTRANSIENT`
The inclusive upper bound of the rollback codes.

- **XA_NOMIGRATE**
`public final static int XA_NOMIGRATE = 9`
Resumption must occur where suspension occurred.

- **XA_HEURHAZ**
`public final static int XA_HEURHAZ = 8`
The transaction branch may have been heuristically completed.

- **XA_HEURCOM**
`public final static int XA_HEURCOM = 7`
The transaction branch has been heuristically committed.

- **XA_HEURRB**
`public final static int XA_HEURRB = 6`
The transaction branch has been heuristically rolled back.
Java Transaction API

- **XA_HEURMIX**
`public final static int XA_HEURMIX = 5`
The transaction branch has been heuristically committed and rolled back.

- **XA_RDONLY**
`public final static int XA_RDONLY = 3`
The transaction branch was read-only and has been committed.

- **XAER_RMERR**
`public final static int XAER_RMERR = -3`
A resource manager error occurred in the transaction branch

- **XAER_NOTA**
`public final static int XAER_NOTA = -4`
The XID is not valid.

- **XAER_INVAL**
`public final static int XAER_INVAL = -5`
Invalid arguments were given.

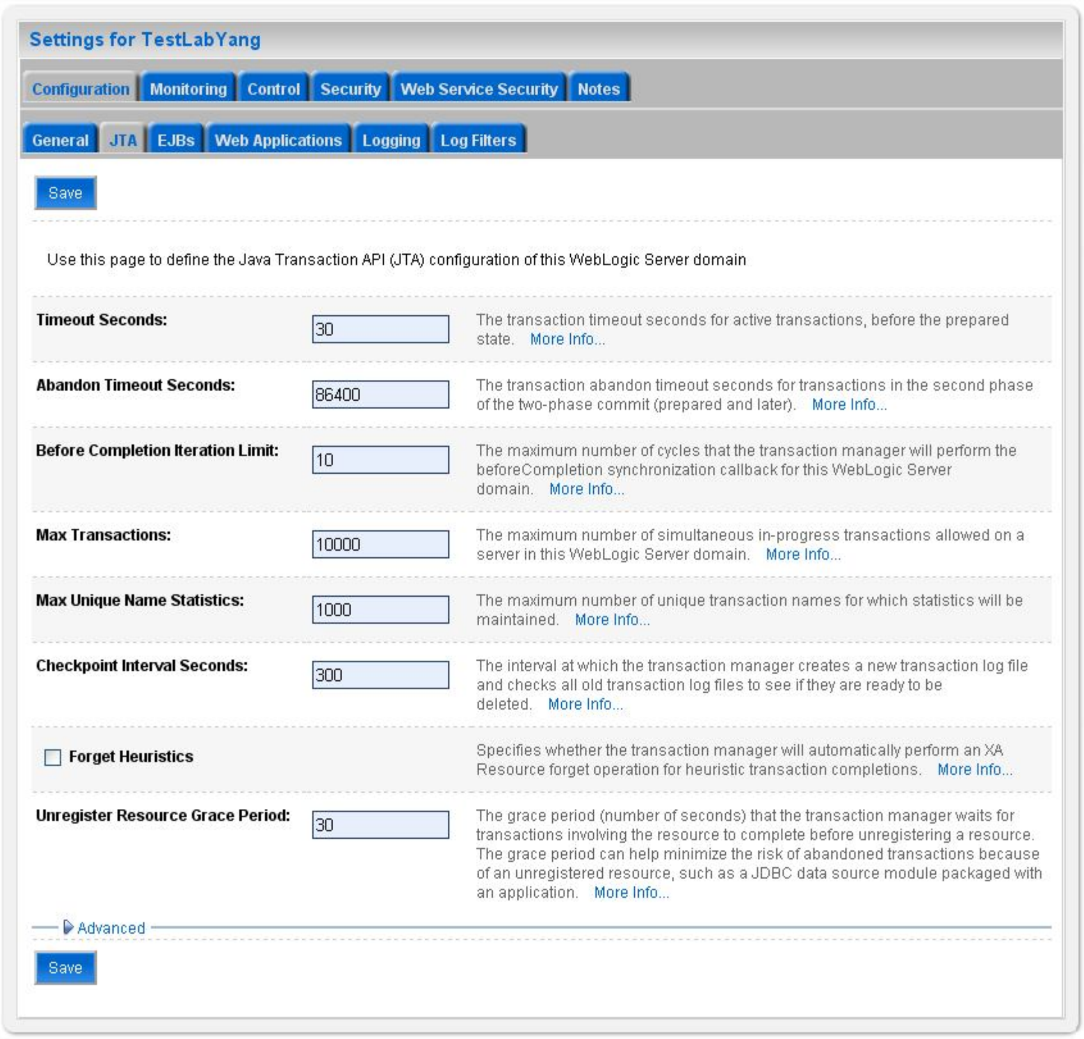
- **XAER_PROTO**
`public final static int XAER_PROTO = -6`
Routine invoked in an improper context.

- **XAER_RMFAIL**
`public final static int XAER_RMFAIL = -7`
Resource manager unavailable.

- **XAER_DUPID**
`public final static int XAER_DUPID = -8`
The XID already exists.

- **XAER_OUTSIDE**
`public final static int XAER_OUTSIDE = -9`
Resource manager doing work outside global transaction.

6.3 JTA configuration in WLS console



Use this page to define the Java Transaction API (JTA) configuration of this WebLogic Server domain

Timeout Seconds:	30	The transaction timeout seconds for active transactions, before the prepared state. More Info...
Abandon Timeout Seconds:	86400	The transaction abandon timeout seconds for transactions in the second phase of the two-phase commit (prepared and later). More Info...
Before Completion Iteration Limit:	10	The maximum number of cycles that the transaction manager will perform the beforeCompletion synchronization callback for this WebLogic Server domain. More Info...
Max Transactions:	10000	The maximum number of simultaneous in-progress transactions allowed on a server in this WebLogic Server domain. More Info...
Max Unique Name Statistics:	1000	The maximum number of unique transaction names for which statistics will be maintained. More Info...
Checkpoint Interval Seconds:	300	The interval at which the transaction manager creates a new transaction log file and checks all old transaction log files to see if they are ready to be deleted. More Info...
☐ **Forget Heuristics**		Specifies whether the transaction manager will automatically perform an XA Resource forget operation for heuristic transaction completions. More Info...
Unregister Resource Grace Period:	30	The grace period (number of seconds) that the transaction manager waits for transactions involving the resource to complete before unregistering a resource. The grace period can help minimize the risk of abandoned transactions because of an unregistered resource, such as a JDBC data source module packaged with an application. More Info...

7 References

[1] Java Transaction Processing - Feasibility Study
A. Geppert - Credit-Suisse

[2] Distributed Transaction Processing: The XA Specification
X/Open Company Ltd.

[3] Java Transaction API (JTA) Specification, Version 1.1 2002
Susan Cheung, Vlada Matena - Sun Microsystems Inc.

[4] Application Platform Architecture
A. Geppert – Credit Suisse

[5] Java Application Platform (JAP) Overview
R. Weber – Credit Suisse

[6] JAP AR5 Transaction Study
U. Tonazzi, H. Leiser, A. Meyer – Credit Suisse

[7] Java Enterprise in a Nutshell, Third Edition
J. Farley, W. Crawford

[8] Enterprise JavaBeans 3.0, Fifth Edition
B. Burke, R. Monson-Haefel

[9] Transaction Patterns: Design & Java-Framework
Marco Stöckli - ETH Zürich