VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

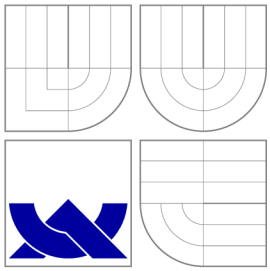
IMPLEMENTATION OF DISTRIBUTED TRANSACTIONS IN BPEL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

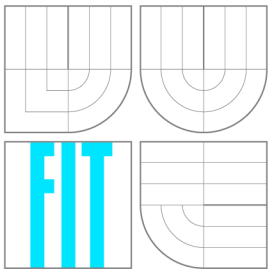
AUTOR PRÁCE
AUTHOR

IVO BEK

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

IMPLEMENTACE PODPORY DISTRIBUOVANÝCH TRANSAKCÍ V BPEL

IMPLEMENTATION OF DISTRIBUTED TRANSACTIONS IN BPEL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

IVO BEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ZDENĚK LETKO

BRNO 2012

Abstrakt

Cílem této bakalářské práce je implementovat podporu distribuovaných transakcí do projektu RiftSaw tak, aby webové služby mohly být volány v rámci distribuovaných transakcí podnikovými procesy. A to pouze v tom případě, že operace webové služby vyžaduje být provedena v rámci distribuované transakce. Oproti již funkčním implementacím přináší podporu specifikace WS-BusinessActivity a jiný způsob kontroly, zda má podnikový proces použít distribuované transakce u volaných webových služeb.

Abstract

The goal of this work is to implement a support of distributed transactions into the project RiftSaw so that web services can be invoked within distributed transactions by business processes. And only if a web service operation requires to be performed within a distributed transaction. Comparing to already working implementations, the presented sulution brings support of WS-BusinessActivity specification and a different way of checking that a business process use distributed transactions for invoked web services.

Klíčová slova

řízení podnikových procesů, distribuované transakce, webové služby, BPEL, RiftSaw, Switchyard, JBoss Transactions

Keywords

business process management, distributed transactions, web services, BPEL, RiftSaw, Switchyard, JBoss Transactions

Citace

Ivo Bek: Implementation of Distributed Transactions in BPEL, bakalářská práce, Brno, FIT VUT v Brně, 2012

Implementation of Distributed Transactions in BPEL

Declaration

I hereby declare that this thesis is my original authorial work which I have worked out by my own under supervision of Zdeněk Letko and Jiří Pechanec. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in the bibliography.

.....
Ivo Bek
May 16, 2012

Acknowledgements

Special thanks to my colleagues, Gary Brown, Jeff Yu and Marek Baluch for consulting and technical help. Also thanks must be given to Zdeněk Letko for reading and correcting the thesis.

Contents

1 Introduction **3**

2 Transaction Processing **4**
2.1 Distributed Transactions 5
2.2 Two-Phase Commit Protocol 6

3 Web Services **7**
3.1 Using and Implementing Web Services 8
3.2 WS-Coordination 9
3.3 WS-AtomicTransaction 10
3.4 WS-BusinessActivity 10

4 JBoss Transactions **11**

5 Business Process Management **12**
5.1 Business Process Execution Language 12
5.2 RiftSaw 13
5.3 SwitchYard for Business Processes 15

6 Problem Analysis **16**
6.1 Criteria of a Satisfactory Solution 16
6.2 Related Works 17
6.3 Sub-problems 18
6.4 Possibilities of a Solution 20

7 Design of Integration **21**
7.1 Intended Cooperation 21

8 Implementation in the RiftSaw **23**
8.1 Check Policy 23
8.2 Transaction Context in the SOAP Header 24
8.3 Subordinate Transaction in a Business Process 24
8.4 Business Activity Support 25

9 Testing and Results **26**
9.1 Demo Application 26
9.2 Basic Testing Scenarios 28
9.3 Review 30

10 Conclusions **31**

A JBoss AS7 XTS Configuration **35**

B Compact Disk **36**

Chapter 1

Introduction

Information systems today continue to become more complex and try to raise reliability and availability. Because the number of clients increases, the parts of information systems are separated on different machines. To follow the trend even in the area of information system reliability, we use distributed transactions. Interfaces of information systems are often available through web services which are managed by business processes. Because we have to ensure that some of web service operations will be performed within a distributed transaction, it is necessary to have the support of distributed transactions in the business processes.

This thesis is about implementing the support of distributed transactions in the project RiftSaw so that web services can be invoked within distributed transactions by business processes running in the RiftSaw. To decide which web service operation needs to be invoked within a distributed transaction, we have to choose a suitable way that we use because there are at least two existing possibilities.

Every theoretical part of this thesis is very large so it is briefly introduced and describes the most important things for the thesis. In the following Chapter 2, we describe the transactions, especially the distributed transactions. Then we focus on web services in Chapter 3 and business processes in Chapter 5 because we will use them to test the implementation of distributed transactions in business processes. We come to the practical part of thesis and do an analysis of the problem in Chapter 6. We discover that the implementations of distributed transaction in business processes already exist in Chapter 6.2 but because they are commercial and we need own support in JBoss community projects, we have to deal with it. In Chapter 7, we design the integration of JBoss Transactions in the RiftSaw and describe an intended cooperation between a business process, web service and transaction manager. Then we look at the interesting parts of implementation like subordinate transactions or WS-BusinessActivity support in Chapter 8. Finally, we evaluate the solution in Chapter 9.3 with the criteria defined in the analysis of the problem and with required functionality that was tested by basic testing suite in Chapter 9.2.

Chapter 2

Transaction Processing

In this chapter we will focus on a transaction processing. Its meaning, possibilities and usability. Because the thesis targets implementation of distributed transactions, it leads to the transaction processing on several machines.

Transaction is a series of operations which have to be performed all or none [25]. To decide what should be done, there are two operations that we can use, the *commit* and *rollback*. The *commit* operation confirms a transaction and the *rollback* operation aborts a transaction. Transactions are required in some specific situations when we need to be absolutely sure that operations was successfully accomplished. An example application is a bank transaction. The bank transaction withdraws money from one account and deposits to another. These operations have to be performed all or none because if not, the account credits would be inconsistent and the money could be lost.

Place where we store data and do transaction processing is usually called a *transaction resource*. For a resource we can use databases, message queues or file systems. Transaction resource which is registered in a transaction is called *transaction participant*. This term occurs in the technical articles or books if it describes more than one resource used in a transaction [25]. Transactions usually follow properties which are collectively referred as ACID:

- ***A****tomicity* - In order to ensure that if a transaction is successful, then all the operations happen, and if unsuccessful, then none of the operations happen.
- ***C****onsistency* - The application performs valid state transitions at completion.
- ***I****solation* - The effects of the operations are not shared outside the transaction until it completes successfully.
- ***D****urability* - Once a transaction successfully completes, the changes are final and can be recovered after a failure.

Mutually agreed outcome labeled as *atomicity* is not the only thing to be achieved. We need to be sure that transaction resources will do changes from one valid state to another valid state labeled as *consistency*. These changes have to be stored on a durable place because they have to survive possible failures. This property is usually labeled as *durability*. What about if we have more opened transactions using the same resources? For this, we need to specify how the changes will be visible outside the transaction which is labeled as *isolation*.

2.1 Distributed Transactions

Distributed transaction is a transaction that runs in multiple processes, usually on several systems, and involves actions against two or more transaction resources [25]. Distributed systems present risk in reliability. Decentralization can cause that some parts of the system could fail whereas another parts could work. It could tend to abnormal behavior in an application. Thus, for using resources in a distributed system and increase the reliability, the distributed transactions are used.

Transaction resource is managed by resource manager that is responsible for communication with transaction manager via 2PC protocol. We need the resource manager because the transaction manager using the resource can be on another machine. Transaction manager is the component that manages and coordinates transactions across a distributed system. Each of the machine from distributed system has its own transaction manager. Client application communicates with transaction manager to create a distributed transaction and commit or abort the transaction.

Because we can have lot of transactions we need to recognize each of them. For this purpose we have an identifier to recognize the particular transaction. The identifier is a part of transaction context. Transaction context is described by coordination context and contains additional information about transaction. To determine that the context is transactional, the coordination type have to be set properly. The transaction context is received from transaction manager after a request to create a distributed transaction. Then every request or response contains the transaction context in its message header. Coordination contexts and coordination type values are described in Chapter 3.2.

There exist two way how to communicate across network between transaction participants and coordinator. We presume to have transaction participants on another machine than the coordinator. The basic way is that coordinator communicates with transaction participant like they are on the same machine as coordinator. When we increase number of transactions or participants, the network traffic rapidly growing. To moderate the network traffic growing we can use interposition. Interposition uses hierarchy of coordinators. Each coordinator can be placed on machine where is the resource manager. It is even particularly useful for web service transactions (discussed in Chapter 3) to increase performance.

Distributed systems can be absolutely different and written in different programming languages. If we run a remote service it should not be important in which programming language it is implemented. For example, there are some implementations which can run a remote service only with information about name of the service and Inter-operable Object Reference (IOR), the address of service. Common Object Request Broker Architecture (CORBA) is a standard which was developed as a common interconnection bus for distributed objects [20]. Communication of distributed systems is based on Internet Inter-ORB Protocol (IIOP). "The Object Request Broker (ORB) is the basic mechanism by which objects transparently make requests to and receive responses from each other on the same machine or across a network." [20]

2.2 Two-Phase Commit Protocol

To guarantee consensus between transaction participants we use Two-phase commit (2PC) protocol. Transactions can be coordinated by this protocol to be able to use more transaction resources.

Both phases depicted in Figure 2.1 consists from coordinator and transaction participants. Coordinator is responsible for governing the outcome of the transaction. During the first preparation phase, the transaction coordinator attempts to communicate with all of the participants to determine whether they will commit or abort. An abort reply, or no reply, from any participant acts as a veto, causing the entire action to abort. If the coordinator decide to commit, this decision is recorded on a durable storage. In the second commit or recover phase, the coordinator forces the participants to carry out the decision. Any participant should not change the decision in the second phase because it could cause heuristic outcome which is mentioned further.

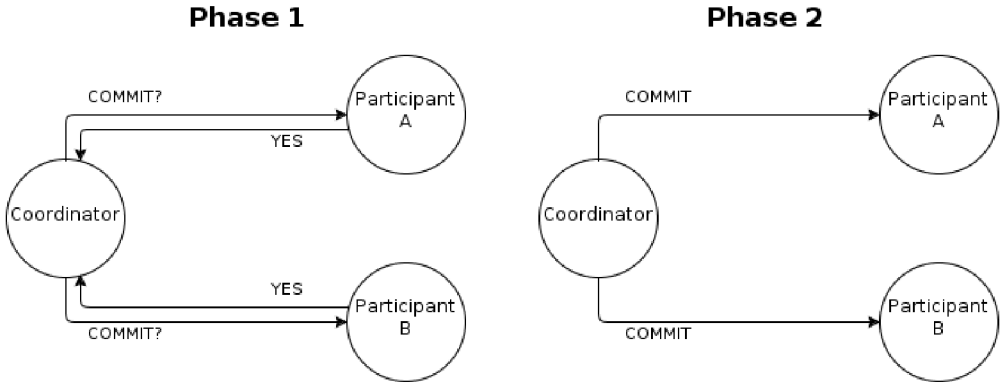


Figure 2.1: Two-phase commit protocol

The transaction resources are blocked and unavailable for use by other actions until a commit/rollback response message from the coordinator is received. Imagine that we have simultaneous transactions using same transaction resources. We presume to have a well-formed design which does not cause deadlock. If one of the coordinators would fail before delivery the commit/rollback message, the transaction resources would remain blocked for the second transaction for an indefinite period of time. Some applications (flight systems, stock exchange systems etc.) and participants cannot tolerate this blocking. To break this blocking, participants which get past the prepare phase are allowed to make autonomous decision. They can commit or rollback transaction on their decision from the prepare phase. There is no problem when the transaction outcome from coordinator is same as choice the participant made. However, if the coordinator detect after failure recovery that decisions are contrary, then a possibly non-atomic (may heuristic) outcome has happened. Because the participant has logs about the changes, one of possible solution is that coordinator will call functions to take back changes in the transaction participant. After that transaction manager will raise heuristic rollback exception to the client which requested commit. It might happen that one transaction participant would make a commit and another transaction participant would make a rollback. This is the worst scenario which could happen and the transaction manager would raise a heuristic mixed exception. The ways how to figure out are complex and we do not need to go so deep. Some solutions can be found in the book Java Transaction Processing [25].

Chapter 3

Web Services

Web services are web based applications that use open standards based on Extensible Markup Language (XML) and transport protocols to exchange data with clients [28]. They are designed to support inter-operable machine-to-machine interaction over a network [16]. Of the protocols in existence today, Hypertext Transfer Protocol (HTTP) is the one specific protocol that all platforms tend to agree on. Web service developer can use any language he wish and a web service consumer can use standard HTTP to invoke web service operations.

Web services can be represented as a modular, reusable software components that are created by exposing business functionality through a web service interface. The foundation of most Service Oriented Architecture (SOA) applications are web services for their interoperability.

The following basic specifications originally define the web services space.

- *Simple Object Access Protocol* (SOAP) is a lightweight protocol for exchange of information in a decentralized, distributed environment [13].
- *Web Service Definition Language* (WSDL) is an XML format for describing network services as a set of endpoints operating on messages [16].
- *Universal Description, Discovery and Integration* (UDDI) provides an XML/SOAP standards based framework for describing, discovering and managing web services [15].

Web services can be composed in two ways, orchestration and choreography. This paradigms describe the ways how to deal with complex operation which is separated to several tasks. Each task is implemented as a web service. In the orchestration paradigm which is usually used in business processes, a central process takes control of the tasks and coordinates the execution of different operations on the web services [21]. Only the central coordinator of the orchestration is aware of operation goal. Choreography, in contrast, does not rely on a central coordinator and each web service involved knows exactly when to execute operations and with whom to interact. Orchestration is more flexible paradigm, but choreography is better aimed on goal of operation.

3.1 Using and Implementing Web Services

This chapter presumes better understanding in JavaEE because the chapter describes some of details how to use and implement a web service. For testing purposes we will need to create web services so there are information of proceeding.

First, we focus on using already implemented web services. To invoke a web service operation, we need to know some details. These details are provided by a WSDL interface [16]. In the WSDL interface there are placed data type definitions, abstract definitions of messages, port type as a set of abstract operations, binding and service definition used to aggregate a set of related ports. To invoke the operation we need to know a web service endpoint from the service definition and operation described in binding, especially input and output message types which define how to create a SOAP message.

If we know only an endpoint of a web service, there is a rule to obtain the WSDL interface. To get the WSDL interface we add to the end of the endpoint "*?wsdl*." On this address we will find the WSDL interface.

When we know how to invoke an operation, we can create a SOAP message to do it. We will also receive response in SOAP message, if an operation is in request-response type.

Figure 3.1 describes a SOAP message structure. We have an envelope which is composed of two parts, the `header` and `body`. The header is optional and it can contain transactional context or security related information. To do an operation request we set the body part. The response details are received in the body part.

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
<!-- can contains transactional context or security related information -->
  </env:Header>
  <env:Body>
<!-- message content which can contains an operation request or response -->
  </env:Body>
</env:Envelope>
```

Figure 3.1: SOAP message

For creating web services in Java, The Java API for XML Web Services (JAX-WS) [14] is used. It is not the only API to define web service, the alternative could be for instance an Apache Axis2 [2].

Following paragraphs are about implementation details how to create a web service in the JAX-WS.

We need to mark where we can find the web service which methods are operations, define their parameters and results. To do it, we are using annotations. A java class annotated with a `@javax.jws.WebService` annotation will be used as a web service. Class package is mapped to a *wsdl:definitions* element and an associated *targetNamespace* attribute by following rules.

- The package name is tokenized using the dot character as a delimiter.
- The order of the tokens is reversed.

- The value of the *targetNamespace* attribute is obtained by concatenating "*http://*" to the list of tokens separated by "*http://*" and '/'.

The java package "*org.jboss.example*" e.g. would be mapped to the target namespace "*http://example.jboss.org*".

Web service operations are all public methods which are not static or final. By annotation `@javax.jws.WebMethod` with attribute `exclude`, we can specify operation to exclude from web service interface. Also we can specify in detail the web method parameters with annotation `@javax.jws.WebParam` or an operation result by annotation `@javax.jws.WebResult`.

We have two ways how to deploy web Services on application server. As a servlet in web archive *.war or as an EJB in java archive *.jar. If we choose EJB, the web service have to be annotated with `@Stateless` annotation.

3.2 WS-Coordination

WS-Coordination is a specification describing an extensible framework for providing protocols that coordinate the actions of distributed applications [27]. The framework introduces a generic service based on the coordinator service model. It is a controller service known as a coordinator or coordination service used for coordinating participants. The coordinator controls following three services:

- *Activation service* is responsible for the creation of a new context and for associating this context to a particular activity.
- *Registration service* allows participating services to use context information received from the activation service to register for a supported context protocol.
- *Protocol-specific services* represent the protocols supported by the coordinator's coordination type.

A service that wants to take part in an activity managed by WS-Coordination must request the coordination context from the activation service. It can then use this context information to register for one or more coordination protocols. A service that has received a context and has completed registration is considered a participant in the coordinated activity [17].

The completion process begins with client service request to the coordination service. Then the coordinator sends completion request messages to all coordination participants. Each participant service responds with a completion acknowledgment message [17].

Each coordinator is based on a coordination type. The most commonly associated coordination types with WS-Coordination are WS-AtomicTransaction and WS-BusinessActivity. These extensions provide a set of coordination protocols. A protocol is a set of rules that are imposed on activities and the rules must be followed by all registered participants [17].

Coordination context contains information about coordination type, identifier and registration service. Example of coordination context supporting a transaction service in SOAP message is described in the WS-Coordination specification [27].

3.3 WS-AtomicTransaction

WS-AtomicTransaction specification provides the definition of the atomic transaction coordination type that is to be used with the extensible coordination framework described in the WS-Coordination [24]. It enables shared resources to be protected from concurrent applications.

The specification defines three protocols for atomic transactions [24]. They are often used to enable a two-phase commit.

1. *Completion Protocol* initiates commit processing and is based on each protocol of registered participants, the coordinator begins with Volatile 2PC and then proceeds through Durable 2PC. The final result is signaled to the client application.

2. *Volatile 2PC Protocol* is used for the participants managing volatile resources such as a cache register.

3. *Durable 2PC Protocol* is used for the participants managing durable resources such as a database register.

If we want to use an atomic transaction for participants in a web service, the SOAP message with a request has to contain coordination context with coordination type set to the value `http://docs.oasis-open.org/ws-tx/wsat/2006/06`.

3.4 WS-BusinessActivity

Business activities consist of long-running, complex transactions involving numerous services [18]. Period of a business activity can be hours, days or even weeks.

The important different against to WS-AtomicTransaction is that WS-BusinessActivity does not offer rollback capability. Instead, business activities provide an optional compensation process that, much like a "plan B," can be invoked when exception conditions are encountered [18].

WS-BusinessActivity specification provides the definition of two coordination types that are to be used with the extensible coordination framework described in the WS-Coordination specification [26]. The first AtomicOutcome[1] coordinator type determines that coordinator must direct all participants either to close or to compensate. Coordinator for the second MixedOutcome[2] coordination type must direct all participants to an outcome but may direct each individual participant to close or compensate.

WS-BusinessActivity specification also defines two specific Business Activity agreement coordination protocols [18].

1. *Business Agreement With Participant Completion Protocol* allows a participant to determine when it has completed its part in the business activity.

2. *Business Agreement With Coordinator Completion Protocol* requires that a participant rely on the business activity coordinator to notify it that it has no further processing responsibilities.

[1] http://docs.oasis-open.org/ws-tx/wsba/2006/06/AtomicOutcome
[2] http://docs.oasis-open.org/ws-tx/wsba/2006/06/MixedOutcome

Chapter 4

JBoss Transactions

JBoss is a trademark and a division of Red Hat, Inc that specializes in java enterprise middleware. JBoss Transactions is a java transaction service that supports transaction processing, compliant with Java Transaction API (JTA) [1], Java Transaction Service (JTS) [3] and web service standards. What does it mean for us? It enables managing and creating local transactions with JTA, distributed transactions with JTS or web service transactions with XML Transaction Service (XTS) [23]. The most important part of the JBoss Transactions for this thesis is the XTS component.

JTA specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system, the resource manager, the application server, and the transactional applications [1]. Transaction manager can be obtained from the Java Naming and Directory Interface (JNDI) location *java:/TransactionManager*. It is not recommended to use it because transaction manager associates transactions with threads therefore the operations will be associated with the calling thread. We should use *UserTransaction* instead which allows an application to explicitly manage transactions.

JTS is Java binding to the CORBA Object Transaction Service (OTS) [3]. This service is important for providing an interoperability between the transaction managers to support the distributed transactions. Local transactions are faster than distributed transactions because the distributed transactions require ORB to communicate.

XTS provides support for general coordination framework WS-Coordination and web service transactions WS-AtomicTransaction and WS-BusinessActivity. JTA or JTS transactions are similar but not identical to web service transactions. JBoss Transactions enables bi-directional interoperability between JTA, JTS and WS-T component. Web service transactions are not adequate by themselves because they require integration with back-end solutions (JTS).

The client is free to use any appropriate API to do invocations on transactional Web Services. But there are two requirements imposed on the client side. The header of the SOAP message for invoke must contain details of the current transaction. The second requirement is that client must process any responses in the transaction context of the correct transaction. The XTS includes handlers which put or read the context automatically.

JBoss Transactions supports interposition for implicit and explicit transaction context propagation. An explicit transaction context propagation means that transaction context is passed as parameter whereas implicit transaction context propagation is passed from client to object implicitly and all operations are assumed transactional.

Chapter 5

Business Process Management

Business process management is a systematic approach to making an organization's workflow more effective. Approach of business process management is composed from six activities: vision, design, modeling, execution, monitoring and optimization. We focus on the modeling and execution of business processes because we only need to model and execute business processes to test the implementation of distributed transactions.

A business process is a set of logically related tasks performed to achieve a well-defined business outcome [28]. It is typically associated with operational objectives and business relationships described in Business Process Model and Notation (BPMN) model. BPMN is a graphical notation that depicts the steps in a business process.

In the following sections we see that a business process can be represented by specific language. That the language is managed by an engine. We will be interested in RiftSaw engine because it is the part which will be modified to support distributed transactions. And finally, we look at SwitchYard project where the RiftSaw is part of it. For that reason we need to know how a business process must be configured to be run in SwitchYard.

5.1 Business Process Execution Language

Business Process Execution Language (BPEL) is an imperative markup language used for definition and execution of business processes using web services. It is a dialect of XML supporting specifications like web services support, the SOAP, WSDL and UDDI. BPEL specification defines a model and a grammar for describing the behavior of business process based on interactions between the process and its partners [12].

The aim of BPEL is to provide system-to-system interaction. BPEL provides a way to compose several web services into composite service which presents business process [21]. With the aid of orchestration paradigm, described in Chapter 3, we can invoke web services in specific order and share data between them. To model relationships between BPEL process and partner services we define partner links. Figure 5.1 shows a BPEL process which can be presented in two roles. For `receive` activity, the current process has role defined by `myRole` attribute. When we want to invoke the BPEL process from another process, it has role defined by `partnerRole` attribute.

BPEL is used by programmers creating business systems in Service Oriented Architecture (SOA) and by analysts searching ways to make business processes more effective. SOA can be realized through composition, orchestration and coordination of web services. BPEL does not provide a standard visual notation for representing business processes. However, there is BPMN to BPEL mapping.

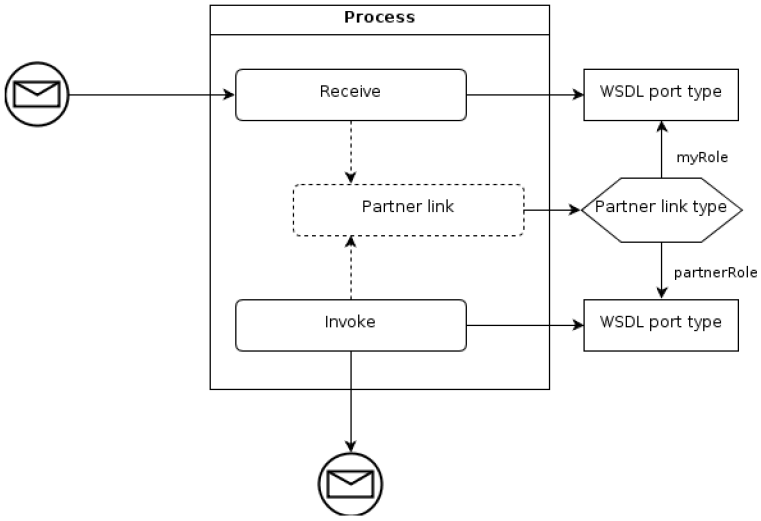


Figure 5.1: BPEL partner link

BPEL engine is a process manager which interprets a process definition defined by BPEL specification. The engine creates process instances from process definitions and manages their performing. Activities of process instance are performed by BPEL engine. BPEL engine also responds for handling exceptions and security. Many implementations of BPEL engine exist but we will focus only on RiftSaw in next Section 5.2 because it is the engine where the distributed transactions will be implemented.

Business processes are important part of service oriented applications which manage services therefore they have to be run always in any time and situation. Although, we would need to update BPEL process, a communication could be interrupted or machine could be broken, the BPEL processes should not be stopped. The importance depends on every business process. Versioning of process has influence on support secure updating of process. BPEL engine which supports versioning can decide in dependency of configuration what way a running process can continue after deploy a new version of BPEL process. The running process which is updated by BPEL engine can be terminated, restarted or can continue running in old or new version of BPEL process. When communication is interrupted while a process is invoking service, the process instance is paused and BPEL engine will save the process state to database. After a specific time, the process instance is restored by BPEL engine and try to invoke service again.

In the BPEL language we can construct control structures with conditional statements, loops, declare variables, copy and assign values, define fault handlers and more. By combining all these constructs we can define any business processes. BPEL allows the modeling of highly concurrent activities with native support of flows and advance synchronization of these flows.

5.2 RiftSaw

Project RiftSaw is a BPEL 2.0 engine that is optimized for the JBoss Application Server container [19]. RiftSaw provides a JBoss AS integration for the Apache Orchestration Director Engine (ODE) [4] that implements BPEL specification.

Because we will need to test the implementation of distributed transactions in the

RiftSaw we look at how business processes can be created in the RiftSaw. To provide a business process with RiftSaw it is necessary to:

- create a BPEL process,
- define a WSDL interface for the BPEL process,
- add WSDL interfaces for web service invocations and configure them by setting partner link roles,
- define a deployment descriptor which declares the BPEL process and its partner links.

BPEL process can be created manually writing XML BPEL process or with RiftSaw BPEL Designer which is part of JBoss Tools [5]. We can use JBoss Developer Studio which contains all the tools.

WSDL interface of BPEL process has a service definition which must tell us where the process is located. The WSDL also contains interface of BPEL operations and their messages. Different against to common WSDL of web service is that we have to specify partner link roles.

Figure 5.2 shows an example of deployment descriptor for the *BusinessTravel* BPEL process. Deployment descriptor contains nonzero process nodes. Each `process` node defines their `partnerLink` and service which provides. The process definition tells RiftSaw about processes to be deployed. When we have a web service to be invoked, we add `invoke` node under the `provide` node because the RiftSaw must know about web service invoke details.

```
<deploy ... >
  <process name="examples:BusinessTravel">
    <provide partnerLink="client">
      <service name="examples:BusinessTravelService"
               port="BusinessTravelPort"/>
    </provide>
    <invoke partnerLink="airport">
      <service name="airport:AirportService"
               port="AirportServicePort"/>
    </invoke>
  </process>
</deploy>
```

Figure 5.2: Deployment descriptor for the BPEL process

RiftSaw supports clustering and we can use it for two reasons, to increase performance and raise reliability. To increase performance, RiftSaw distributes every incoming SOAP request by load balancer. The second reason for clustering is to raise reliability. When some of clustered machine with running processes fail, another one restore the processes from a last known state and continue from this state.

5.3 SwitchYard for Business Processes

This section introduces using the SwitchYard for creating and using BPEL processes. SwitchYard is a lightweight service delivery framework providing full life-cycle support for developing, deploying, and managing service-oriented applications [6]. We will use the SwitchYard to show integration of distributed transaction in RiftSaw.

SwitchYard uses RiftSaw as a BPEL component to execute business processes. When there is a SOAP message for business service, it is propagated from SwitchYard into RiftSaw which executes the process. `Invoke` operation causes that request message leaves the process and is delivered internally into SwitchYard with `invoke` details. The SwitchYard transforms the message to needed format and invokes the service operation which was requested by business process. Then a result is returned back to the business process.

Creating BPEL process for SwitchYard is different against to older versions of RiftSaw. Final step of creating BPEL process is about specifying details about configuration of BPEL service in *switchyard.xml* which is located in the folder *META-INF*.

The SwitchYard configuration of BPEL service in Figure 5.3 contains a component and service definitions. The component consists of an implementation node to identify the BPEL process and a service WSDL interface which is for access BPEL process. It can also contain references to services which can be invoked from the process. The service defines BPEL process name *BusinessTravelService* and binding details where are specified a socket address *:18001* and context path of the service *wstdemo*. Then full path of the service is "*http://127.0.0.1:18001/wstdemo/BusinessTravelService*."

```
<switchyard xmlns="urn:switchyard-config:switchyard:1.0" ...
    targetNamespace="urn:switchyard-quickstart:wstdemo:0.1.0"
    name="wstdemo">
  <sca:composite name="wstdemo" targetNamespace="urn:bpel:test:1.0">
    <sca:service name="BTravelService" promote="BTravelService">
      <soap:binding.soap>
        <soap:wsdl>BusinessTravelArtifacts.wsdl</soap:wsdl>
        <soap:socketAddr>:18001</soap:socketAddr>
        <soap:contextPath>wstdemo</soap:contextPath>
      </soap:binding.soap>
    </sca:service>
    <sca:reference name="AirportService"> ... </sca:reference>
    <sca:component name="BusinessTravelService">
      <bpel:implementation.bpel xmlns:sh=".../bpel/examples"
        process="sh:BusinessTravel" />
      ...
```

Figure 5.3: BPEL service in SwitchYard configuration

RiftSaw has become a SwitchYard BPEL component since version 3. RiftSaw 3 itself is not responsible for invoking web services. This job is performed by SwitchYard. RiftSaw only sends internal messages with a specific details to invoke service. The great advantage of RiftSaw is that it is not limited only on web services but it can invoke every service available through WSDL interface which SwitchYard provides.

Chapter 6

Problem Analysis

The goal of this work is to ensure that web service operations will be performed within a distributed transaction in the business processes. In the next section, we specify criteria of satisfactory solution and the goals. Because this thesis should be innovative, we look at existing implementations of related works to discover absences. Then we analyze sub-problems which must be considered and try to find suitable solution which fits the defined criteria. And finally, we specify the possibilities of solution how to check a web service operation if it needs to use a distributed transaction. Depending on the advantages of possible solutions, we choose one solution to be implemented.

6.1 Criteria of a Satisfactory Solution

The finally solution should allow to determine which web service operation will be used within a distributed transaction. To review the accomplishment of thesis and decide which possibility will be used, we specify the criteria. Some of them are described more detailed below the summary.

(a) Distributed transaction processing should be autonomous to make creating and using BPEL process comfortable.

(b) Adding support of distributed transactions should require least count of changes in existing service oriented applications.

(c) Process operation which has WS-AtomicTransaction or WS-BusinessActivity policy requirement should create subordinate transaction if invoked service operation requires distributed transaction.

(d) With the previous criterion it should be possible to use own distributed transaction which would be propagated through BPEL process into services.

(e) We must be sure that if some failure occurs in BPEL process, the distributed transaction will be rolled back.

(f) When distributed transaction is not required in any of service used by BPEL process, XTS does not have to be run.

Automatic creating and completing distributed transaction. This is a more detailed description of criterion *(a)*. RiftSaw will automatically create distributed transaction for an operations with policy assertion requiring WS-AtomicTransaction or WS-BusinessActivity. In the end of business process the transaction will be automatically completed. Automatic creating and completing distributed transaction makes it transparent and the user application of business process does not have to care about it.

Using own distributed transaction. This is a more detailed description of criterion *(c)* and *(d)*. BPEL process operation which has WS-AtomicTransaction or WS-BusinessActivity policy requirement will create subordinate transaction if invoked service operation requires distributed transaction. Then a requester, the client of BPEL process, should be able to create a distributed transaction, provide the context to the BPEL process and commit or rollback transaction itself.

XTS requirement. This is a more detailed description of criterion *(f)*. JBoss Transactions XTS is not in the default profile configuration on application server JBoss AS7 with SwitchYard. Because of that it should be required only if some of BPEL processes have a WS-AtomicTransaction or WS-BusinessActivity policy assertion requirement. Then if the user is conscious that he requires XTS, he should set the application server properly. How to set the application server to use the XTS is described in Appendix A.

6.2 Related Works

There exist some already working implementations which integrate distributed transactions into BPEL engine. The BPEL engines with support of distributed transactions are commercial so the information are only from the related documents.

Oracle BPEL Process Manager [8] is a component of Oracle SOA Suite [10] supporting distributed transactions. The Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite 11g [9] describes how a BPEL process must be set to support WS-AtomicTransaction. There is composite.xml file which resembles deploy.xml and contains BPEL component and web service references. To add support of atomic transaction into BPEL process it is necessary to add transaction properties into binding. BPEL process which is invoked by another BPEL process can be enlisted in the transaction if we set *bpel.config.transaction* property to *required* [22, 9]. In Figure 6.1 notice these properties (tagged as `property`) how they are set.

There are some web service optimizations which do not allow to use atomic transactions so we should be aware when we can use them. In documentations of Oracle Fusion Middleware Search there were not any information about support WS-BusinessActivity.

Second BPEL engine also supporting distributed transactions is IBM Business Process Manager [7]. To enable WS-AtomicTransaction, the deployment descriptors for web and EJB modules must be configured [29]. Rational Application Developer or WebSphere Integration Developer set the configurations by default. Every module can be specified to propagate the transaction or require transaction context to be run in.

We did not notice about any support of WS-BusinessActivity in the related works. But our solution will support the WS-BusinessActivity specification. In Chapter 6.4 we analyze the solution of related works and describe own solution.

```
<component name="WSATBPELClient">
  <implementation.bpel src="WSATBPELClient.bpel"/>
  <property name="bpel.config.transaction"
       many="false" type="xs:string">required</property>
</component>
<reference name="BankAccountService" ui:wsdlLocation="http://...?WSDL">
  <interface.wsdl interface="..."/>
  <binding.ws port="..." location="http://...?WSDL" soapVersion="1.1">
    <property name="weblogic.wsee.wsat.transaction.flowOption"
      type="xs:string" many="false">SUPPORTS</property>
    <property name="weblogic.wsee.wsat.transaction.version"
      type="xs:string" many="false">WSAT11</property>
  </binding.ws>
</reference>
```

Figure 6.1: Example of composite.xml

6.3 Sub-problems

The goal of this work is to ensure that web service operations will be performed within a distributed transaction in the business processes. Below the summary of sub-problems, we analyze the sub-problems which must be considered.

1. distributed transaction requirement
2. propagation of transaction context
3. distributed transaction completing
4. simultaneous processes
5. clustering
6. distributed transaction recovery

Distributed transaction requirement. First, we need to determine on what suggestion the transaction will be created. Every business process does not have to create distributed transaction because the transaction is needed only when the process uses web services requiring distributed transaction. We focus on this problem closely in Section 6.4.

Propagation of transaction context. When we have the transaction created, the second problem we discover is how we propagate transaction context into web services. We can propagate the transaction automatically or manually. Automatic propagation is performed by context handlers which are set in binding of the service. With this binding instance we call operations as methods. But because we use SwitchYard to invoke services, we must choose manually putting transaction context into message header. The reason is that the message in this state is not a SOAP message yet but only an internal message.

Distributed transaction completing. Well, the final part of basic functionality is completing distributed transaction. Because the process does not have any ways how to return information about result of the transaction, we have to figure out some alternative ways. The part of problem are one-way operations where the process cannot respond with fail message.

When a user application needs to know if everything or "nothing" has been done, it has own ways how to check. For example we can have a participant which sends Java Messaging Service (JMS) or SOAP message when the participant does commit or rollback. The message would invoke another business process to do some activities or user application would have JMS listener which would inform about transaction completion.

Simultaneous processes. For a one thread we can have only one distributed transaction. Every business process instance is run in own thread and composes from jobs (e.g., actions from `receive` activity to `wait` activity). The problem is that we need to put the distributed transaction from previous job into following (e.g., actions from `wait` activity to `reply` activity). We should have a map of distributed transactions which would keep information for the process instances about the related distributed transactions. Then the distributed transaction of process instance would be set at the beginning of job.

Clustering. Very interesting area is clustering (two or more running application servers). In the current version of RiftSaw 3 it is not possible to use clustering but in the near future it will be supported. After that there will be critical problems (e.g., fail over). It would be nice to support fail over including distributed transaction.

Distributed transaction recovery. A key requirement of a transaction service is to be resilient to a system crash by a host running a participant, as well as the host running the transaction coordination services [11]. To guarantee reliability, there should be implemented support for transaction recovery of coordinator and its participants. Activities of business process like `invoke` or `wait` are safe-points. The last safe-point of business process is saved in database so whether application server go down, the processes are recovered from last safe-point after the application server is restarted. If the recovered process uses distributed transaction, the transaction must be recovered too. Approach of transaction recovery is different for before the prepare phase and after the prepare phase of 2PC protocol. The phases are described below.

before the prepared phase After the restart of application server which crashed, the coordinator sends *rollback* to all participants. Because business process can continue from the middle of process, it is the problem. The transaction was rolled back and next invoke requiring web service transaction could create another one. Then the process is done in half way.

prepared phase and later After the restart of an application server which crashed, the coordinator sends *commit* to all participants. We do not have to be worried about re-delivery because the XTS participant implementation is resilient to re-delivery of the commit messages. If the participant implements recovery functions, the coordinator will send commit messages even if coordinator and the participant fail together.

Unfortunately, the recovery of process does not work in RiftSaw 3 either because the clustering is not supported yet. The fail over processing should also recover processes after

the crash but until it will be implemented we do not have to do anything for the transaction recovery before the prepared phase.

Thus the result of these thoughts is that the transaction recovery should work partially for some cases. For example when the participants fail on the other machine than the coordinator does, they can be recovered.

6.4 Possibilities of a Solution

From the user point of view, we need to describe that BPEL process has to use distributed transaction to interact with web services. First thoughts were about providing the description to deployment descriptor, the deploy.xml file as Oracle BPEL Process Manager has in the composite.xml. The deployment descriptor contains information about process and its partner links. We would only add attribute to the partner links which web service extension should be used, WS-AtomicTransaction or WS-BusinessActivity. There are unfortunately some fundamental negatives of this thought. We would not be able to describe which operation should use distributed transaction and which should not. And it would need to change deployment descriptor schema. Then, the deployment descriptor schema would be transferred into java classes by Apache XMLBeans.

The second idea was about use policy assertion in WSDL interface. The WSDL describes web service and with the policy assertion we would be able to mark each operation with a policy requiring WS-AtomicTransaction or WS-BusinessActivity. RiftSaw, the BPEL engine, would create distributed transaction for the first operation with this policy. Then it would propagate transaction context for each operation even for that which would not require WS-AtomicTransaction or WS-BusinessActivity. Propagation transaction context to the web service operation which does not require web service transaction should not cause any issues. if we check every time the WSDL operation, we could decide whether we will propagate the transaction context or will not.

As we defined the criteria we choose the idea which requires least changes in existing service oriented applications. The changes for both possibilities are almost same but for the second idea it is more clear why the web service transaction policy is there. On the other hand an user must know or find how to define the policy for WSDL interface and the properties and attributes are more easier to remember or can be easily integrated into BPEL Designer plugin. Finally, the second idea had been chosen because it does not require any changes in the deployment descriptor and it is innovative comparing to related works.

Chapter 7

Design of Integration

This chapter describes the design of integration to implement the distributed transactions into the RiftSaw. To run a BPEL business process we use the RiftSaw 3 so the most important changes will be made there. JBoss Transactions XTS component will provide the support of WS-AtomicTransaction and WS-BusinessActivity implementations in the business processes. Because RiftSaw 3 is used as BPEL component of SwitchYard, some changes will be made in the SwitchYard as well.

The Figure 7.1 shows integration between the projects. RiftSaw 3 is already integrated in SwitchYard through the BPEL component. Our work will be the part between RiftSaw and JBoss Transactions XTS. But the Riftsaw 3 should not absolutely depend on the JBoss Transactions XTS. When the XTS is not required in any of business processes, the JBoss Transactions XTS does not have to be run in the application server.

The RiftSaw communicates with the SwitchYard and the SwitchYard communicates with the RiftSaw. JBoss Transactions XTS will be used only by RiftSaw.

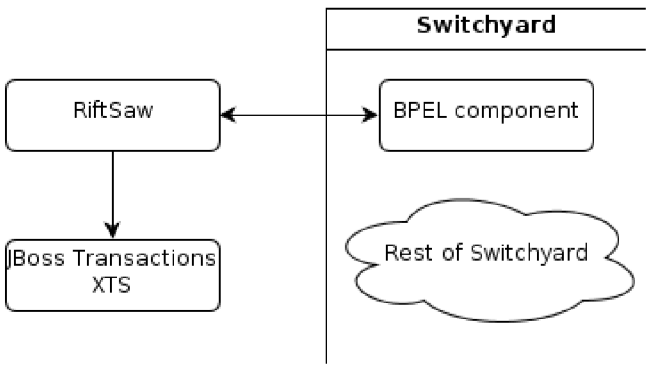


Figure 7.1: Design of integration

7.1 Intended Cooperation

Intended cooperation of a business process, web service, transaction manager and its transaction coordinator is described on Figure 7.2. In the collaboration diagram of Figure 7.2, we have the business process with `invoke` activity requiring web service transaction. The transaction manager and the coordinator are the parts which are not implemented in the business process workflow.

The workflow of Figure 7.2 is described by an enumeration. Every request or order is marked with a number which fits the enumeration below.

1. The idea is that before the web service invocation is performed, the business process creates a distributed transaction via transaction manager.
2. Then the transaction manager requests activation service to create a coordination context for a web service transaction.
3. When we have a transaction context in the process, we can begin with invoking web service. To invoke web service with support of web service transactions we need to propagate the transaction context.
4. Web service creates transaction participants which are enlisted in the transaction.
5. The participants are registered via registration service.
6. After all performed tasks in the process it is decided to commit or rollback and the request is sent into transaction manager.

7 - 8. The transaction manager commands coordinator protocol service to do transaction completion with 2PC protocol. If the commit or rollback is successfully done, the business process properly ends.

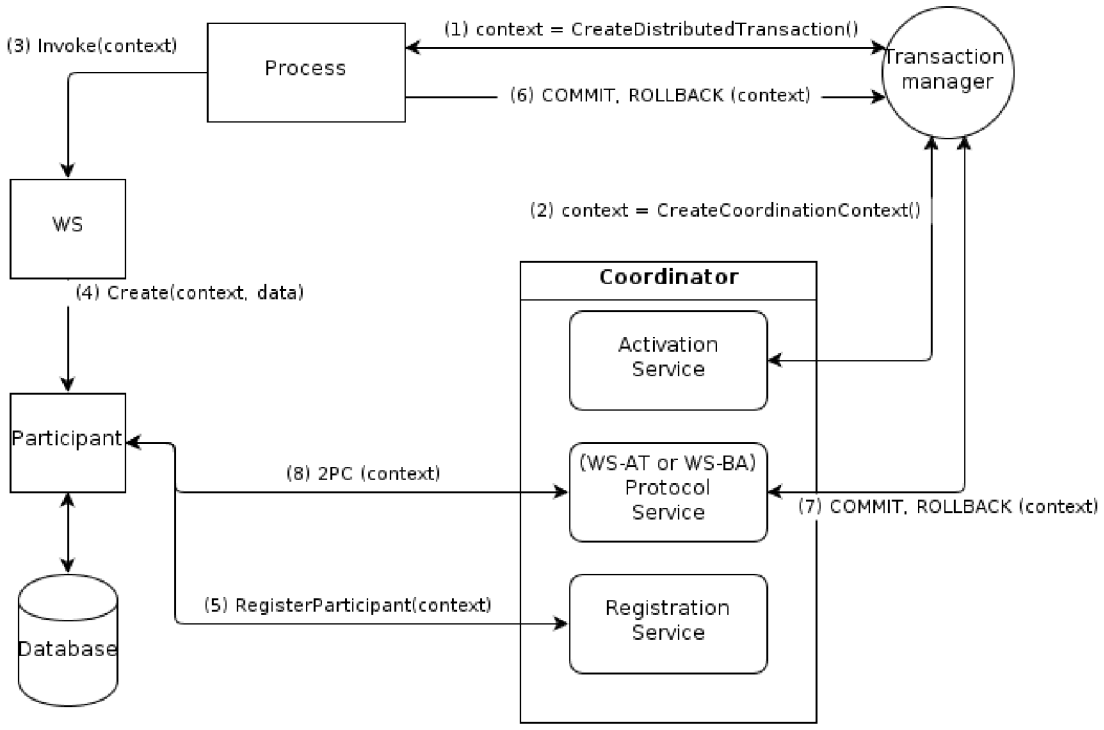


Figure 7.2: Collaboration diagram

Chapter 8

Implementation in the RiftSaw

This chapter provides a description how the distributed transactions were implemented in the RiftSaw. The implementation was split into the parts which could be completed separately. General aim was to provide a simple functional implementation.

At first it is necessary to check a policy in WSDL as a transaction requirement when a distributed transaction has to be created and propagated. The next step is transaction context propagation which had to be implemented manually for the reason described in Chapter 6.

To support the use of own distributed transaction, we use the subordinate transactions. The main difference against to the already existing solutions is WS-BusinessActivity support. Because the implementation of business activity transactions is more complicated than implementation of atomic transactions, we will focus on them.

8.1 Check Policy

As we determined in the Chapter 6, we have to specify when a distributed transaction will be used. Thus, we decided to check the policies. The Figure 8.1 shows how the WS-AtomicTransaction policy can be set. Checking policy is based on searching the policy reference in the invoked operation which is checked to referring on the WS-AtomicTransaction or WS-BusinessActivity policy.

```
...
<wsp:Policy wsu:Id="TransactedPolicy" >
  <wsat:ATAssertion wsp:optional="true" />
</wsp:Policy>

<wsdl:binding name="AirportServiceSoapBinding" type="tns:AirportService">
  <wsdl:operation name="order">
    <wsp:PolicyReference URI="#TransactedPolicy" wsdl:required="true" />
...
```

Figure 8.1: Setting the WS-AtomicTransaction policy in WSDL.

The searching is operated through all binding operations to find the operation which is invoked. When we find the operation, we check that the policy reference is present. Every policy reference has a Uniform Resource Identifier (URI) property. Then we will go through

all policies defined in WSDL definition and compare reference URI with a policy URI. If the matched policy contains WS-AtomicTransaction or WS-BusinessActivity requirement, the distributed transaction is created and propagated.

8.2 Transaction Context in the SOAP Header

In Chapter 6, we decided to put the transaction context manually into SOAP header. This section contains the approach of manually putting the transaction context into a SOAP header as an element.

Transaction context can be taken from the `com.arjuna.mw.wst11.TransactionManager` via `currentTransaction` method. There is a class to serialize and deserialize the transaction context into or from the SOAP header. It is the `com.arjuna.mw.wst11.common.CoordinationContextHelper` and the parameters for the serialization are header element and coordination context type. Before we use the serialization function, we have to prepare the header and get the coordination context type.

The coordination context type can be obtained from the transaction context via `context` method and `getCoordinationContext` then. To prepare the header we add an element representing coordination context which has to have a name of the element set to *CoordinationContext*[1] in *http://docs.oasis-open.org/ws-tx/wscoor/2006/06* namespace[2].

After that we are prepared to call the static method `serialise` in the `com.arjuna.mw.wst11.common.CoordinationContextHelper`. The `serialise` method will add all necessary information about the distributed transaction into SOAP header. What information are added is described in Chapter 3.2.

8.3 Subordinate Transaction in a Business Process

One of criteria is the support of own distributed transactions. A user application can propagate its own distributed transaction through the transaction context in the SOAP header. When the invocation of web service requires the distributed transaction, the RiftSaw creates a new subordinate transaction from the propagated transaction. The Figure 8.2 shows how to create the subordinate transaction from the propagated transaction and the following paragraphs describe all necessary steps to create the subordinate transaction.

At first we have to resume the distributed transaction from the transaction context. In the previous section we had the `com.arjuna.mw.wst11.common.CoordinationContextHelper` where we used the `serialise` method. The class also provides `deserialise` method which can be used exactly for our needs, to get the coordination context type from the SOAP header element. Then we create own `TxContext` instance via `com.arjuna.mwlabs.wst11.at.context.TxContextImple` for atomic transaction and `com.arjuna.mwlabs.wst11.ba.context.TxContextImple` for business activity transaction. The constructor needs only one parameter, the coordination context type that we obtained in the previous step. And we resume the propagated transaction with `resume(txcontext)` method from the transaction manager.

We obtain the propagated transaction via `UserTransaction.getUserTransaction()` or `UserBusinessActivity.getUserBusinessActivity()`. But that is not all because we

[1] recommended using CoordinationConstants.WSCOOR_ELEMENT_COORDINATION_CONTEXT
[2] recommended using CoordinationConstants.WSCOOR_NAMESPACE

```
boolean subordinate = false;
CoordinationContextType cct = CoordinationContextHelper.deserialise(header);
if (cct != null) {
  TxContext ctx = new TxContextImple(cct);
  TransactionManager.getTransactionManager().resume(ctx);
  subordinate = true;
}
  // get the (propagated)? transaction
UserTransaction tx = UserTransaction.getUserTransaction();
if (subordinate) {
    // get the subordinate transaction
  tx = UserTransaction.getUserTransaction().getUserSubordinateTransaction();
}
tx.begin();
```

Figure 8.2: Implementation of subordinate transaction from the propagated transaction.

need to create subordinate transaction which can be self completed when the business process ends. To create subordinate transaction we call `getUserSubordinateTransaction()` method from the propagated transaction instance.

8.4 Business Activity Support

The WS-BusinessActivity specification is designed for long running transactions. Because some of business processes are run long time and we do not want to block transaction resources for indefinite period of time, it is obvious that we use the business activity transaction.

Beginning of the business activity transaction is performed by `begin` method as the atomic transaction does. The distributed transaction is partially completed every continuous block of activities by `complete` method which informs all the participants enlisted for business agreement with coordinator completion. The another option is that participant completes itself and enlists for business agreement with participant completion. In the end of the business process, the distributed transaction is closed or canceled. When the distributed transaction closes and some of participants are completed, the participants are informed to compensate which should rollback the changes or do any other necessary actions.

There are two options how to define policy requirements for business activity, the atomic outcome and the mixed outcome requirements. But, they are not considered so it is not important if the policy is set to `BAAtomicOutcomeAssertion` or `BAMixedOutcomeAssertion`.

Chapter 9

Testing and Results

This chapter describes the testing used to check the implementation of distributed transactions in BPEL. For the testing purposes we created a demo application which should be able to test the main approaches of atomic transaction and business activity transaction.

9.1 Demo Application

The demo application presents Business Travel System. All the components (business process, airport and hotel services) used in the demo application are depicted in Figure 9.1. Business travel system simulates ordering fly tickets and hotel rooms by using external services in a distributed transaction.

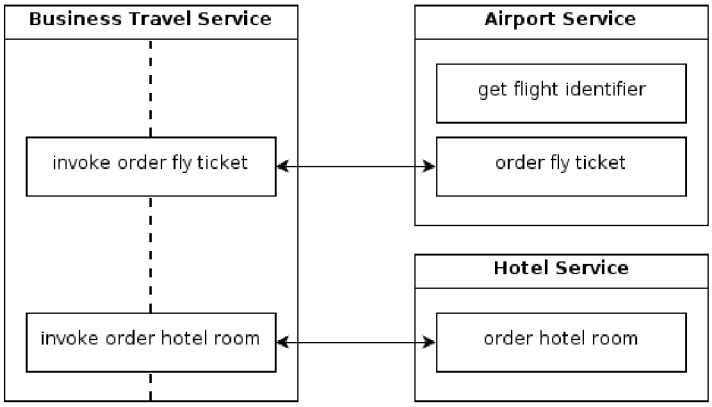


Figure 9.1: Demo application

General part of demo application is a BPEL process which plan a business travel. The business travel process designed in BPEL editor is depicted in Figure 9.2. By requesting the process we can order fly tickets and hotel rooms. The orders are accomplished in the end of process by `complete` operation. The complete operation can also simulate exception if we set the `simulateException` parameter to `true`.

`Order` and `complete` operations in the BPEL process are correlational. It means that if we send a request with a specific id and a second request has the same id, the process will continue from previous process state. Imagine that we would have lot of running processes. Some of them would be in the same state and waiting for a specific message. Correlation provide the way how to say which process will receive the message by correlation identifier.

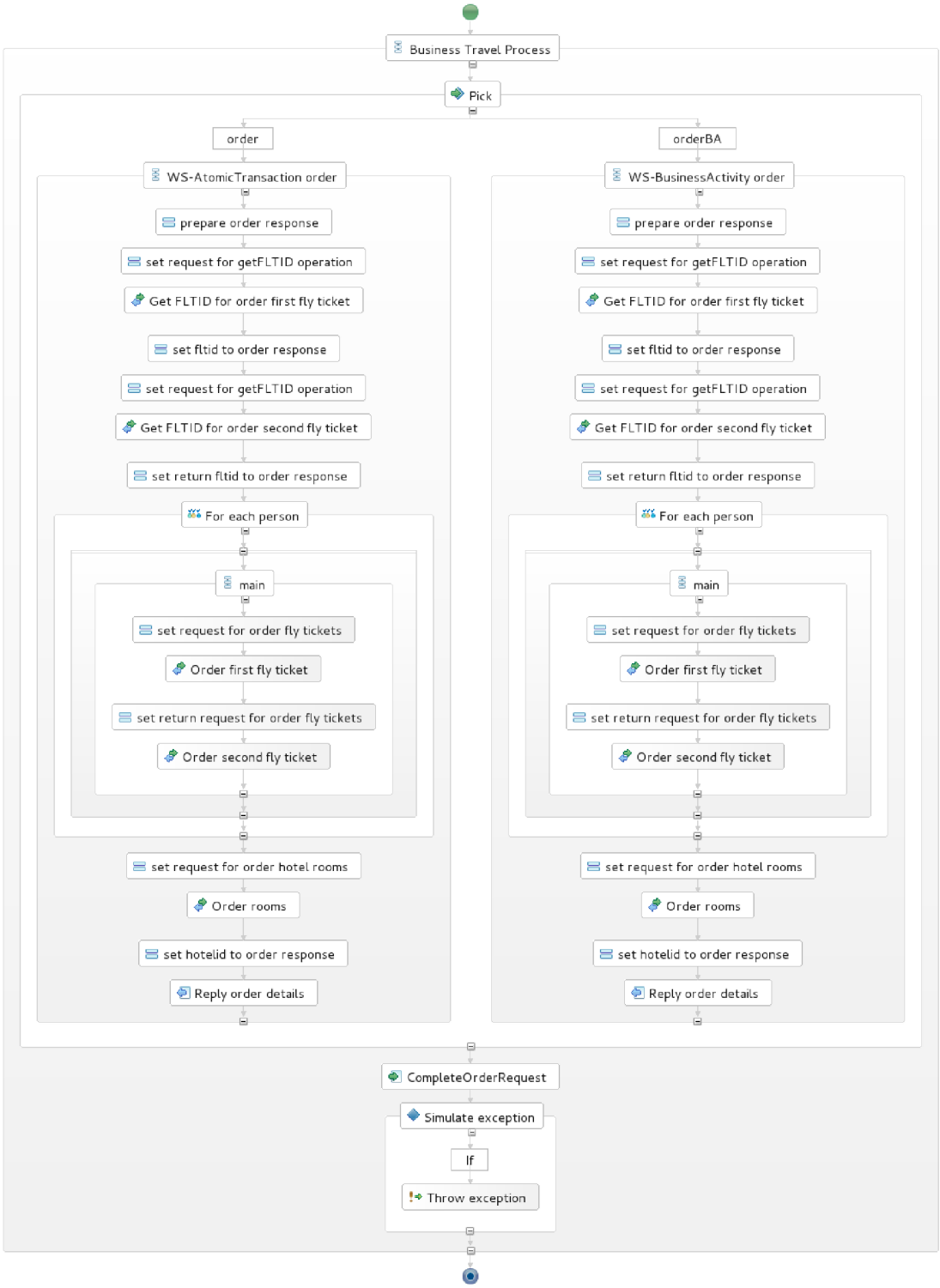


Figure 9.2: Business travel process in BPEL designer.

9.2 Basic Testing Scenarios

Atomic Transaction Commit Test

To reach the commit of a demo process it is enough to end the process properly. Thus, we send order request to the demo process and wait on the registration of all participants in the invoked web services. All information about invocation of web services and the registration should be visible in the server log. Then the demo process waits on complete request. After the complete request is received, the demo process ends and the distributed transaction is completed through 2PC protocol. Information that the participants were prepared and successfully committed should be in the server log. The expected and given output is:

1. Two participants from the airport service and one participant from the hotel service enlist into the distributed transaction.
2. All the participants are prepared.
3. All the participants are committed.

Atomic Transaction Participant Rollback Test

2PC protocol gives to participant an opportunity to rollback a distributed transaction because in the first phase it asks if participants want to commit or rollback. This test is not practical because for some bad parameters it waits for the prepare phase to inform coordinator about abort instead of throw an exception. When we know it, it is enough to send a message with bad parameters and the distributed transaction rollback when we want to commit. The expected and given output is:

1. Two participants from the airport service and one participant from the hotel service enlist into the distributed transaction.
2. One of the participants want to abort in the prepared phase of 2PC protocol.
3. All the registered participants are rolled back.

Atomic Transaction Process Rollback Test

The another way how to rollback distributed transaction is with exception or termination of process. The rollback is performed also if running of some process causes unexpected exception in the RiftSaw. This test uses exception which is thrown by demo process. The expected and given output is:

1. Two participants from the airport service and one participant from the hotel service enlist into the distributed transaction.
2. After the `complete` operation is sent, the demo process throws exception which causes that the distributed transaction rollback from the demo process.
3. All the participants are rolled back without any preparation.

Business Activity Close Test

This test is almost equivalent to the atomic transaction commit test, especially in the progress. We still only need to reach the end of demo process to close the distributed transaction. The difference is between behavior of the WS-BusinessActivity specification because the participants are completed before the demo process ends. The expected and given output is:

1. Two participants from the airport service and one participant from the hotel service enlist into the distributed transaction.
2. When the order operation responds in client (the test), the distributed transaction completes the participants which were enlisted for business agreement with coordinator completion.
3. All the participants are closed after the `complete` request is sent.

Business Activity Compensate Test

In the previous test we noticed that the participants are completed before the demo process ends. This scenario tests that all the completed participants are informed to compensate when the demo process does not end properly. The distributed transaction is canceled if the demo process throws any exception or it is terminated. The expected and given output is:

1. Two participants from the airport service and one participant from the hotel service enlist into the distributed transaction.
2. When the order operation responds in client (the test), the distributed transaction completes the participants which were enlisted for business agreement with coordinator completion.
3. Because the `complete` operation throws exception, the distributed transaction is closed and all the completed participants are informed to compensate.

Own Atomic Transaction Test

During the testing an own atomic transaction we found out that we do not have any possible way how to set handler chain for BPEL process yet. The JAX-WS specification defines handler chain through that is the request sent but when `mustUnderstand` attribute stays after the request leaves the handler chain, the client is informed that the server does not support the required part. For the web services we use `@javax.jws.HandlerChain` annotation so there is not any problem. The handler which must be set is `com.arjuna.mw.wst11.client.JaxWSHeaderContextProcessor`. It confirms that the server side supports web service transaction and the `mustUnderstand` attribute in the coordination context is cleared then. The issue could be resolved when we would be able to set the handler chain in the SwitchYard configuration file.

9.3 Review

To look back and review the solution, we specified the criteria in Chapter 6.1. In this chapter, we describe which of the criteria has been accomplished.

Current solution in the RiftSaw enables the distributed transaction processing to be autonomous. Thus, a user application call a business process without any creating of distributed transaction. The only change, which must be done if we want to use the distributed transaction, is putting the transaction policy to a required operation in the WSDL.

Criteria (c) and (d) are not fully accomplished because we are not able to set the handler chain for a business process now. But, the creating of subordinate transactions and using an own distributed transaction are implemented. The most important limitation lies in the fact that the SwitchYard is still developed[1].

If an exception occurs in a running process, the criterion (e) defines that the business process should rollback the distributed transaction. The performed tests verified that the distributed transactions are rolled back if an exception is thrown.

The XTS is required only when any of business processes needs to use the distributed transactions. But then the application server must be configured properly, like the Appendix A describes the JBoss AS7 XTS configuration.

[1] Current version of SwitchYard is 0.4 Final.

Chapter 10

Conclusions

In the beginning of this thesis we described that we need to guarantee reliability for the distributed systems. Because the parts of distributed system usually communicate via web service interfaces which are managed by business processes, we implemented the support of distributed transactions in the business processes.

The implementation enables the distributed transaction processing to be autonomous in invoking web service operations. The only requirement is that the web service operation has to specify transaction policy in the WSDL. We support WS-AtomicTransaction and WS-BusinessActivity specifications which can be set as transaction policy. Integration of JBoss Transactions XTS in the RiftSaw is not too tight so the RiftSaw can be run without XTS on the application server if we do not use the distributed transactions in the business processes.

This thesis could be extended by support of full recovery processing which would recover a business process with a related distributed transaction. It would need an intervention into database tables to keep details about the related distributed transaction for a business process instance. The another useful proceeding could be a practical application which would test the implementation and the results would tell us what changes to improve are needed.

To be more competitive, the companies start using the business process management because they can easily optimize their business processes and faster react on business changes. I personally think that the distributed transactions will be soon an important part of business processes in the near future.

Bibliography

[1] JSR-000907 Java(tm) Transaction API (JTA) Specification. http://download.oracle.com/otn-pub/jcp/jta-1.1-spec-oth-JSpec/jta-1_1-spec.pdf, 2007-02-14 [cit. 2012-01-14].

[2] Apache Axis2/Java - Next Generation Web Services [online]. http://axis.apache.org/axis2/java/core/, 2011-09-09 [cit. 2012-03-26].

[3] Java Transaction Service (JTS) [online]. http://www.oracle.com/technetwork/java/javaee/tech/jts-140022.html, [cit. 2012-01-14].

[4] Apache Orchestration Director Engine [online]. http://ode.apache.org/, [cit. 2012-03-25].

[5] JBoss Tools [online]. http://www.jboss.org/tools, [cit. 2012-03-25].

[6] Switchyard [online]. http://www.jboss.org/switchyard, [cit. 2012-03-31].

[7] IBM Business Process Manager [online]. http://www-01.ibm.com/software/integration/business-process-manager/, [cit. 2012-04-24].

[8] Oracle BPEL Process Manager [online]. http://www.oracle.com/technetwork/middleware/bpel/overview/index.html, [cit. 2012-04-24].

[9] Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite 11g Release 1 [online]. http://docs.oracle.com/cd/E21764_01/integration.1111/e10224/sca_bindingcomps.htm, [cit. 2012-04-24].

[10] Oracle SOA Suite [online]. http://www.oracle.com/technetwork/middleware/soasuite/overview/index.html, [cit. 2012-04-24].

[11] Andrew Dinn, Kevin Connor, and Mark Little. Transactions XTS Administration And Development Guide [online]. http://docs.jboss.org/jbosstm/5.0.0.M1/guides/xts-administration_and_development_guide/, [cit. 2012-04-15].

[12] Andrews, Tony. Web Services Business Execution Language (WS-BPEL) 2.0 [online]. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, 2007-04-11 [cit. 2011-10-29].

[13] Box, Don and Ehnebuske, David and Kakivaya, Gopal and Layman, Andrew and Mendelsohn, Noah and Nielsen, Henrik Frystyk and Thatte, Satish and Winer, Dave. Simple Object Access Protocol (SOAP) 1.1 [online]. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/, 2000-05-08 [cit. 2011-11-09].

[14] Chinnici R., Hadley M., Mordani R. JSR 224: JavaTM API for XML-Based Web Services (JAX-WS) 2.0 [online]. http://www.jcp.org/en/jsr/detail?id=224, 2006.

[15] Clement, Luc and Hately, Andrew and Von Riegen, Claus and Rogers, Tony. UDDI Version 3.0.2 [online]. http://www.uddi.org/pubs/uddi_v3.htm, 2004-10-19.

[16] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1 [online]. http://www.w3.org/TR/wsdl, 2007-03-15 [cit. 2011-11-05].

[17] Erl, Thomas. The WS-Coordination Context Management Framework [online]. http://www.soaspecs.com/ws-coordination.php, [cit. 2011-12-05].

[18] Erl, Thomas. Long-Running Transactions with WS-BusinessActivity [online]. http://www.whatissoa.com/soaspecs/ws-businessactivity.php, [cit. 2012-03-31].

[19] Gary Brown, Kurt Stam, Heiko Braun, Jeff Yu. Riftsaw 2.3.0.Final [online]. http://docs.jboss.org/riftsaw/releases/2.3.0.Final/userguide/html/, 2011-07-12.

[20] Interoperability, Corba. Common Object Request Broker Architecture (CORBA) Part 1 : CORBA Interfaces [online]. http://www.omg.org/spec/CORBA/3.2/Interfaces/PDF/, 2011-11-01.

[21] Juric, Matjaz. A Hands-on Introduction to BPEL [online]. http://www.oracle.com/technology/pub/articles/mataz_bpel1.html, [cit. 2011-10-29].

[22] Koser, Stefan. Web Service Transactions Part 2: WS-AtomicTransaction with SOA Composite calling EJB-Web Service [online]. http://stefankoser.blogspot.com/2010/08/web-service-transactions-with-ws_13.html, 2010-08-13 [cit. 2012-04-24].

[23] Little, Mark. What is XTS? [online]. http://www.jboss.org/dms/jbosstm/resources/whitepapers/WhatIsXTS.pdf, 2006-01-24 [cit. 2012-01-14].

[24] Little, Mark and Wilkinson, Andrew. Web Services Atomic Transaction (WS-AtomicTransaction) Version 1.2 [online]. http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec-os/wstx-wsat-1.2-spec-os.html, 2009-02-02.

[25] M. Little, J. Maron, G. Pavlik. *Java Transaction Processing: Design and Implementation*. Prentice Hall, 2004. ISBN 0-13-035290-X.

[26] Newcomer, Eric and Robinson, Ian. Web Services Business Activity (WS-BusinessActivity) Version 1.2 [online]. http://docs.oasis-open.org/ws-tx/wstx-wsba-1.2-spec-os/wstx-wsba-1.2-spec-os.html, 2009-02-02.

[27] Newcomer, Eric and Robinson, Ian. Web Services Coordination (WS-Coordination) Version 1.2 [online]. http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec.html, 2009-02-02.

[28] Papazoglou, M. *Web services: principles and technology.* Pearson Prentice Hall, 2008. ISBN 978-0-321-15555-9.

[29] Xu, Peter. Transactionally integrate Web services with BPEL processes in WebSphere Process Server [online]. http://www.ibm.com/developerworks/websphere/library/techarticles/0703_xu/0703_xu.html, 2007-03-07 [cit. 2012-04-04].

Appendix A

JBoss AS7 XTS Configuration

This appendix describes how to configure JBoss AS7 to use XTS and how to change modified SwitchYard. Appendix B contains pre-configured JBoss AS7 with modified SwitchYard but we will describe how it was done.

The JBoss AS7 and SwitchYard can be downloaded from the http://www.jboss.org. SwitchYard project offer the JBoss AS7 with integrated SwitchYard so it can be used instead installing SwitchYard into JBoss AS7. The way how to install SwitchYard into JBoss AS7 is described in the guide[1].

When we have the JBoss AS7 with SwitchYard, we prepare XTS configuration. Prepared XTS configuration is in *jbossas7/docs/examples/configs/standalone-xts.xml*. Copy this configuration into *standalone/configuration* folder but we have not finished yet. Example configuration does not contain SwitchYard module. We copy all SwitchYard parts from *standalone.xml* configuration, the extension module and SwitchYard subsystem.

To start the JBoss AS7 we run the script "*./standalone.sh -c standalone-xts.xml*." It seams to work but if we try to run BPEL process requiring XTS it throws ClassNotFoundException on some XTS class. It can be thrown because of missing dependency in BPEL module. Shutdown the application server and add this dependency

```
<module name="org.jboss.xts"/>
```

into *jbossas7/modules/org/switchyard/component/bpel/main/module.xml*. Then the BPEL process requiring XTS should work after restarting the JBoss AS7.

[1] https://docs.jboss.org/author/display/SWITCHYARD/Getting+Started

Appendix B

Compact Disk

Attached CD contains:

- source code of RiftSaw and SwitchYard components in the *impl* directory
- source code of this thesis in LaTeX format in the *doc* directory
- this thesis in PDF format
- source code of demo application in the *test/wstdemo.zip*
- pre-configured JBoss AS7 with modified SwitchYard and deployed demo application in the *test/jboss-as-wstx.zip*
- readme that describes RiftSaw installation from the sources and testing with the demo application