Diploma Thesis Nr. 3173

SOAP over JMS Support for the Stuttgarter Workflow Machine

Tobias Rohm

Study Course:	**Informatics**
Examiner:	**Prof. Dr. Frank Leymann**
Supervisor:	**Dipl. Phys. Dieter H. Roller**
Commenced:	**18 April 2011**
Completed:	**18 October 2011**
CR-Classification:	**C.2.4, C4, D.2.11, D.3.4, H.3.5, D.4.4, H.4.1, H.5.3**

UNIVERSITY STUTTGART

INSTITUTE OF ARCHITECTURE OF APPLICATION SYSTEMS

Content

List of Figures

List of Examples

1. Introduction

Modern business today relies more and more on complex computational information processing. The evolution of hard- and software provides possibilities and challenges formally not known. The last milestone in this process was the expansion of computer networks. It is possible today to exchange digital information over the whole world. Computers became mobile and even applications so called software agents can migrate through the World Wide Web.

In the past the term application was associated with a computer program running on a designated machine. With the possibilities of networking today modern applications stepped over all formally known boarders for example company intern networks. The electronic data processing changed from huge mainframes serving the demand of multiple clients to distributed solutions that provided a more efficient use of shared resources of all kinds.

New approaches resulted in so called Service Component Architectures (SCA). A component based architecture benefits amongst others from the reusability of programs implementing algorithms that solve well known recurring problems with comprehensive researched solutions. These components can then be combined to applications meeting the demands of more and more complex challenges.

To achieve a maximum of component combination possibilities even in heterogeneous networks one software design and architecture focus concentrated on the decoupling of the systems working together collaboratively. To handle compounds of arbitrary complexity the whole system has to be broken down in meaningful layers with distinguishable assignments. The OSI (Open System Interconnection) layered Specification is an example for a protocol stack enabling the developers to concentrate on the specific challenges at each layer.

Application servers are another example. The environment an application server offers cares for important aspects of software components like security or transactional execution. Therefore the developer can concentrate on the individual solutions often referred as business logic of the code.

Depending on the point of view today a web-based application can be a compound of computational software utilizing all kinds of resources. Such an application can consist of software components spread all over the world running on thousands of computers with different operating systems and hardware platforms. Furthermore the interconnected components can change dynamically during the runtime.

Even medium and small enterprises utilize these new possibilities. But the evolution process did not stop at the level of information processing and management.

Modern Companies are using workflow management systems to perform their business. Today a workflow system is able not only to serve for information exchange but can steer whole executions of business processes. Furthermore the systems evolved in the last years from centralized solutions to distributed systems. The capabilities where enhanced even for cross company boarder collaborative work.

The challenges for workflow management systems is eased by international standards specifying the demands, rules and interchange formats for business to business relations and communication. An emerging standard to exchange documents or business data is the Extensible Mark-up Language (XML). XML can also serve as a description language for models of business processes and other information used by distributed computer systems to work together.

Whereas the networking and computing possibilities increased some challenges remained fixed. First, interacting programs have to agree on a common data format and exchange protocol. Second to get interconnected they have to find each other and establish and keep a connection.

To allow a loose coupling of software components today the actual implementation details are hidden and the components expose themselves only over interfaces. A standard to provide component implementations are web services. Web services are accessed by a standardized and specified description in the Web Service Description language (WSDL). WSDL describes the necessary information to interact. For example how the XML messages exchanged are structured and the operations a service offers.

From WSDL definitions client code to use the service as well as code skeletons for the actual implementation can be generated automatically. Alternatively from existing implementations a WSDL description can be generated to offer the functionality as a web service.

For business demands another language in XML is the de facto standard to determine how a set of web services can cooperate in a workflow. This is the Web Service Business Process Execution Language (WS-BPEL). In combination of these description files a workflow machine implementing the standards can navigate through a business process and invoke the appropriate activities as web services.

The combination of abstract descriptions for the business process in WS-BPEL and the interfaces between distinguished working steps leads to a two level programming model. The so called programming in the large part refers an abstract description of a business process model. The individual steps of a business process are subdivided into activities in WS-BPEL.

The actual implementation of the activities can be of every kind depending on the business demands. This so called programming in the small part deals with the real implementation for example by applications that human users can use to perform their work. The actual computational realization is hidden. Such an approach allows easy replacement, reuse and flexible combination of task implementations.

Nevertheless in the end messages have to be exchanged between the web services. This diploma thesis presents the integration of the Java Messaging Service (JMS) to handle the task of reliable, possibly synchronous, especially asynchronous message transfer in an existing WS-BPEL aware workflow system. This is the Stuttgarter Workflow Machine (SWoM), which is developed and enhanced as high performance workflow management system in between a research project of the University Stuttgart.

1.1. Motivation

A modern workflow system is designed to support as much business scenarios as possible. The Business Execution Language WS-BPEL is amongst others specified in an abstract manner to respect all kinds of demands a business process may encompass. These are for example long term business processes. The complete execution of a business process may encompass working steps that can take hours or even days to be accomplished. Hence some steps can only be done after other activities have all in all completed, the workflow machine is forced to wait for the completion indicated at leas for example by the reception of a acknowledge message.

In general there are two ways a web service can be invoked. The first one is synchronous which means that the invoking web service expects an instant reply for his request. The other way is an asynchronous invocation. This interaction pattern does not expect a reply message at all.

Nevertheless a request response interaction pattern can be realized if the reply can be correlated with the request message. With appropriate correlation data included the reply message can be sent as separate message even at a later point in time. Nevertheless there are no guarantees how long it will take until the reply message arrives or that the reply message will arrive at all.

For sensitive business data it might not be acceptable that the communication is unreliable. Therefore for both the synchronous and asynchronous communication there are a number of possible problems that have to be considered.

Even under the assumption of a high reliable network connection there is still a small probability that messages are lost. Furthermore it is possible that no connection can be established between communication partners at a specific point in time for various reasons. For example servers might crash or have been shut down for maintenance.

The WSDL specification does not stipulate a specific transport protocol. There is one wide spread approach to use the Simple Object Access Protocol to encapsulate the business data messages. SOAP provides a standard for self describing messages with arbitrary content. For the purposes of the Stuttgarter Workflow Machine these are XML text documents but other formats for example application specific encoded files like Computer Aided Design (CAD) or multimedia files are possible.

Nevertheless SOAP needs an underlying transport protocol to transmit the SOAP messages. A common approach is to use the Hyper Text Transfer Protocol HTTP. One reason is that nearly any system connected to the World Wide Web understands HTTP. Another is that HTTP is able to pass most firewalls. Therefore it is a suitable protocol for the communication between web services.

HTTP gains some reliability from the underlying network protocol the Transmission Control Protocol (TCP). TCP is a connection oriented protocol that is able to ensure that lost data segments are acknowledged or will be repeated and the whole message will be reassembled at the receiver side correctly.

Therefore TCP performs well as long as a proper connection can be established and maintained until the successful transmission of for example a SOAP message. The actual

SOAP messages will be part of HTTP messages as result of a one way HTTP GET or POST operation with response.

HTTP is in the first line a stateless protocol. If for some reason a TCP connection fails there is a problem. The only possibility is to retry the connection build up. In that case the message to send has to be preserved. In extreme situations such an approach might not scale. Even worse if the reason for a connection failure is for example a server overload or network congestion retries might exacerbate the situation.

For an asynchronous request that expects a response the situation is even more complicated. The requesting business partner is forced to expose an HTTP connection endpoint permanently for an indefinite amount of time. In Java™ that would be for example a servlet waiting for a connection request for the answer.

For that and other reasons there is another approach for middleware systems connecting heterogeneous applications. A so called Message Oriented Middleware (MOM) system provides a separate infrastructure to deal with reliable message transmission. Especially for asynchronous communication message oriented architecture provides another level to decouple applications.

A specified common standard with various implementations a message oriented middleware can build on is the Java™ Messaging Service (JMS). A sophisticated JMS implementation manages with all aspects of the message transport and delivery. Once a message is handed over the JMS provider can take care for permanent storage and reliable delivery even in temporary absence of a message receiver.

JMS supports arbitrary message formats and therefore SOAP messages too. Hence the Stuttgarter Workflow Machine was developed with IBM WebSphere as application server framework and WebSphere already provides a JMS provider implementation it is obvious to integrate JMS as transport extension into the workflow machine.

This thesis presents the considerations and implementation approaches to enable the Stuttgarter Workflow Machine to support SOAP over JMS as reliable and robust message transport alternative between the participating web services of business processes.

1.2. Structure of the Document

This thesis is subdivided into eight main chapters. After the introduction and a discussion of the reasons to integrate the Java™ Message Service (JMS) as transport alternative into an existing high performance workflow machine some of the details of JMS are introduced in the second chapter. In particular aspects and the handling of administered JMS objects are presented.

The third chapter introduces and gives an overview about the Stuttgarter Workflow Machine (SWoM) project architecture and actual implementation as starting point and target for the work of this thesis. The chapter takes a closer look on the participating components and their importance for this work.

The following chapter discusses the use of JMS to transport SOAP messages in more detail. Furthermore some basic design decisions are presented as origin for further refinements.

The fifth chapter encompasses these refinements and a comprehensive discussion of the related message transmission patterns. Furthermore the details of the actual architecture, design and implementation for the SOAP over JMS support is presented. The structure and interrelationships between developed components are discussed too.

The realization of the SOAP over JMS support encompasses the generation of code for some of the new components and affects the deployment of the description for the process models running on the Stuttgarter Workflow Machine. The sixth chapter discusses the necessary steps from the import of a process model description to the actual deployment on WebSphere as application server.

Followed by the references for the literature used for the work chapter seven summarizes the considerations and results. Furthermore some proposals and suggestions for further work are suggested.

2. Java™ Message Service

This chapter will start with a short overview about the model and architecture of the Java™ Message Service (JMS). The following sections describe the necessary recourses that have to be provided and configured to enable the use of the Java™ Message Service. Hence the implementation for this thesis was done with the Rational Application Developer version 8.0.2 as programming environment and WebSphere version 7.0 as application server the default messaging provider was used as Java™ Message Service implementation. This messaging provider is an implementation of the Java™ Message Service specification. The related IBM Redbook says: WebSphere Application Server V7 is fully compliant with the Java EE 5 specification. These requirements are documented in section 6.6, “Java Message Service (JMS) 1.1 Requirements,” of the Java EE 5 Specification” [IBM2009][2], page 2. For more information on the Java™ Message Service specification see [JMS2002][2].

A possible alternative would have been the use of WebSphere MQ as a sophisticated vendor specific message provider but for the purposes of this thesis the requirements are satisfied by the default messaging provider implementation. For detailed information about the concepts, configuration and administration of the default messaging provider see [IBM2009][2], chapter 2 and 3. Further Information about WebSphere MQ can be found there too.

Further Information and implementation examples for the Java™ Message Service application programming interface (API) can be found in [JMS2002][1]. The book covers basic Java™ messaging and programming for the J2EE™ platform.

The messaging provider implements the interfaces defined by the specification. It is the basis to use the JMS API and provides administrative and control features. Therefore the following subsections provide a brief discussion about the used JMS resources, their creation and administration relevant for this thesis and the SOAP over JMS messaging extension for the Stuttgarter Workflow Machine.

2.1. JMS API Model and Architecture

The architecture of JMS includes a number of parts with different issues. Figure 1 shows the basic architecture of the JMS application programming interface on the left side. Including for example the message provider discussed in the introduction to this chapter. All participating applications are signified as JMS clients whether they will send, receive messages or both. These applications include the code using the JMS application programming interface a message provider implements.

The complete architecture incorporates special objects that the message provider will use to perform its objective the reliable transport of JMS messages from senders to receivers. These are at a high level view connection factories and destinations. These so called administered JMS objects are created and configured in advance. The creation and configuration is done via some kind of administrative interface either by hand from an administrator or via an automated administration process.

Once the JMS resources are created and configured JMS client applications can utilize them.

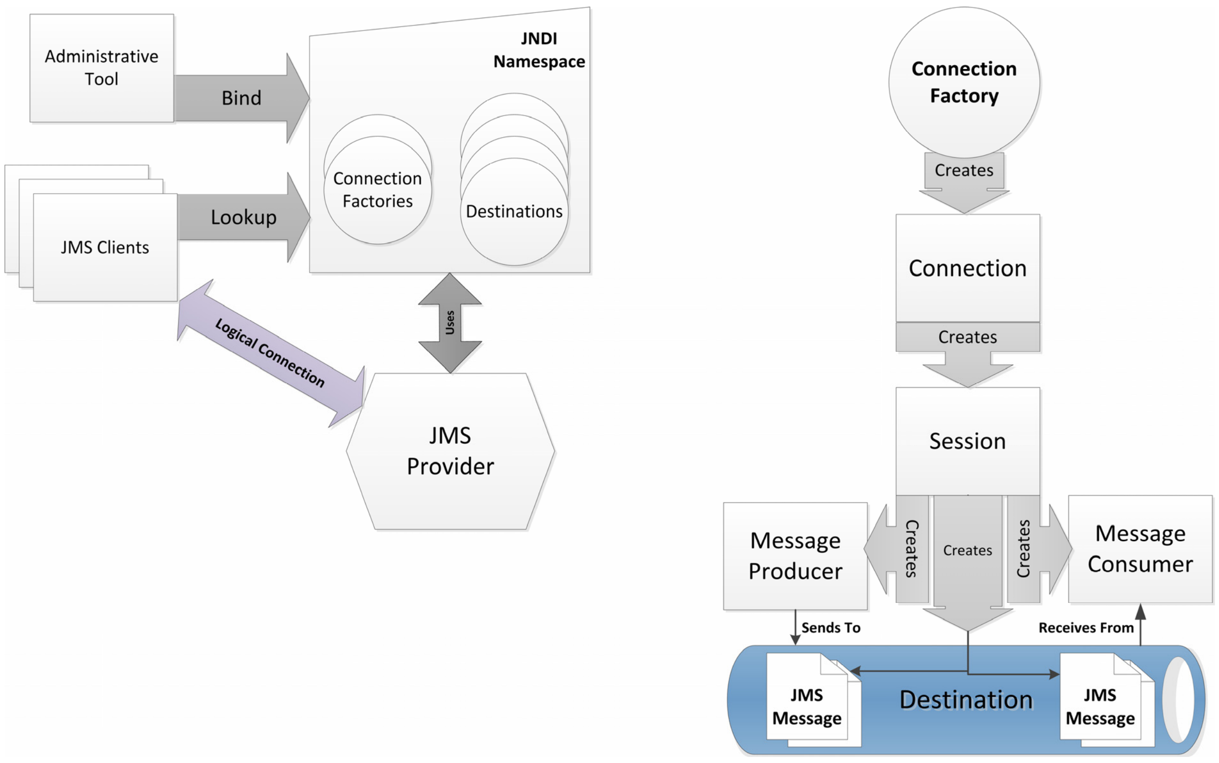

Figure 1. JMS Architecture and Programming Model

An overview of the general implementation of JMS clients is shown at the right side of the picture. A JMS client application will perform for example a JNDI lookup for an appropriate connection factory. With help of the factory the client can create a connection object. From the connection session objects can be obtained. In between a JMS session the client can act as message producer, message consumer or both. That includes the creation, sending and receiving of JMS messages in between a transactional context.

The following section will provide a brief introduction to the JMS objects used for the purposes of this thesis.

2.2. Java™ Message Service Runtime Resources

The Stuttgarter Workflow Machine will act as a JMS client, in fact as several clients and will receive and produce JMS messages the most objects related to the Java™ Message Service API are used. Except for the confinement that only objects related to so called point to point (PTP) message exchange are used. For the purpose of this implementation the publish\subscribe domain is not needed.

The basic JMS objects used can be subdivided into two parts, basic objects and administered objects. They differ insofar, that they: “are best maintained administratively rather than programmatically. The technology underlying these objects is likely to be very different from one implementation of the JMS API to another. Therefore, the management of these objects belongs with other administrative tasks that vary from provider to provider" [JMS2002][1], page 16.

Following this distinction the administered objects are discussed separately in the next subsection. The process for creation and administration is described in subsection 3.2. They have to be created and configured on WebSphere before they can be used by the implementation. After the creation and administration of administered objects a reference for them can be obtained for example by a JNDI (Java™ Naming and Directory Interface) lookup. Alternatively message driven beans can use dependency injection since EJB 3.0. More information about can be found in [EJB2006][1] and [EJB2006][2].

Following the time dimension the next subsections cover the JMS resources used for the Stuttgarter Workflow Machine in the sequence of their creation.

2.2.1. Administered Objects

In the first stages of development these resources were configured via the administrative console for a dedicated WebSphere server. Finally the required resources are created automatically within the deployment process.

Connection factories are used by applications to connect to JMS providers. The connection factory object encapsulates a set of connection specific configuration parameters. These configuration parameters are via by administrative tasks. The configuration determines amongst others the JINDI name to bind it into the WebSphere namespace. Furthermore all information needed to connect to the messaging provider. For this project using the default messaging provider this is at least the bus name used by the connection factory. For more detailed information related to JMS see [JMS2002][1] or [JMS2002][2]. More information about the configuration of a connection factory for the WebSphere application server and the default messaging provider can be found at [IBM2009][2], section 1.4.1.

JMS destinations are a representation for a physical queue of the messaging system. Or in other words: "JMS Clients use JMS destination objects to specify the target of messages that they produce and the source of messages that they consume. A JMS destination provides a specific endpoint for messages" [IBM2009][2], page 12. For the point to point-messaging domain used in this implementation destinations are also called queues. These queues are used by JMS-Clients for example message-driven beans to specify the targets for produced messages and the source for consumed messages. For more detailed information related to JMS see for example [JMS2002][1] or [JMS2002][2]. More Information about the configuration of JMS destinations on the WebSphere application server can be found at [IBM2009][2], section 2.1.5 and 3.6.

Activation specifications in WebSphere and for the default messaging provider are correspondent to message listener ports. An activation specification is needed for message-driven beans to retrieve messages from specific queues. A message-driven Bean will be invoked when a message arrives at a defined destination on the specified bus. The message-driven bean associated with an activation specification is bound to the specification via a

XML file in the project. Per default the file has the name ibm-ejb-jar-bnd.xml and is located in the /ejbModule/META-INF folder. More detailed information about the configuration and properties for an activation specification on the WebSphere Application Server and the default messaging provider can be found at [IBM2009][2], section 1.5.3.

A short overview to the configuration and administration of JMS administered objects with the administrative console of WebSphere provides for example the section 5.4.2 in [IBM2009][3].

2.2.2. Basic JMS Objects

Basic JMS objects are created programmatically with the corresponding create methods. For the purposes of this thesis they are used in the message-driven beans to create outgoing messages for example to send a reply message back to a synchronously requesting client. Hence there is a lot of literature covering the properties and implementation they will be discussed only very shortly for completeness. Please refer to the given references for more information.

JMS connections objects encapsulate a virtual connection to a JMS provider, for example a TCP/IP socket network connection. As the name implies connections are created by calling the correspondent create method of the previously discussed administered connection factory object. Connections must be started before the application can consume messages and could be stopped to interrupt the message delivery. They should be closed to release the resources on the JMS provider. For more information see for example [JMS2002][2], section 5.6 and 9.1.3.

JMS sessions are providing methods to create message producers and consumers. Furthermore a JMS session is a single-threaded context for producing and consuming JMS messages in between a transaction. That means that multiple sends and receives of messages are serialized in an atomic execution sequence. Such a transaction is either committed or aborted. Messages will only be delivered if the session is committed.

As long as for the purposes of this implementation sessions are used in message-driven beans the arguments of the session creation have no effect. “When you create a session in an enterprise bean, the container ignores the arguments you specify because it manages all transactional properties for enterprise beans” [JMS2002][1], page 73. For more information about JMS sessions see for example [JMS2002][2], section 4.3.6. For a detailed discussion of distributed transactions with enterprise beans see [JMS2002][1], section 6.3.

JMS message producers are objects created via a session. Such objects are used to produce and send messages. For the purposes of this implementation a producer is used to create the reply messages for a synchronous request of a client in the message-driven beans. The destination for the produced message can be specified within the send method of the producer or within the creation method of the corresponding session object. For this implementation the latter approach was used. For more information about the creation of JMS message producers and consumers see for example [JMS2002][1], .section 4.4.2

JMS message consumers are objects created via a session too. They are used to receive messages sent to a destination. For this implementation the message consumers are message-driven beans implementing the MessageListener interface. When a message arrives at the destination the listener is observing the onMessage(Message message) method of the message-driven bean is called. Implement a listener is from a strict point of view asynchronous. It is possible to create synchronous message consumers too, but this implementation uses the first approach. Hence the asynchronous internal behaviour is transparent to the client. For more information to JMS message consumers see for example [JMS2002][1], page 309.

JMS messages are the exchanged entities. Hence the structure is essential for the design and implementation for the SOAP over JMS support for the Stuttgarter Workflow Machine the structure of the used JMS messages will be discussed in detail in section 4.1.2 of this thesis. For detailed information about the message interface see [JMS2002][1], chapter 25. There are a number of different JMS message types for the body part or payload of a JMS message. For developing, implementation and test of the Java™ code for this thesis a JMS bytes message was used because it is the most abstract possible payload format. Nevertheless other types like a text message may be more efficient for specific scenarios. For more information about the JMS bytes message see for example [JMS2002][1], page 274.

2.2.3. Service Integration Bus

The WebSphere default messaging provider is associated with an underlying so called service integration bus. The related IBM Redbook says: "The service integration bus is the underlying messaging provider for the default messaging provider" [IBM2009][2], page 66. The architecture allows for advanced treatment of the messaging traffic within multiple WebSphere application servers working collaboratively together. This enables multiple application servers or clusters of application server to manage specific failover and high workload scenarios and can provide more robustness, reliability and scalability.

In [IBM2009][2], page 66, the concepts and architecture of the service integration bus are described as: "The service integration bus (or just bus) provides a managed communication framework that supports a variety of message distribution models, reliability options and network topologies. It provides support for traditional messaging applications as well as enabling the implementation of service-oriented architectures (SOA) within the WebSphere application server environment."

To discuss the possible network topologies is out of the scope of this thesis. For example is for each server, or a cluster of server that is added as bus member at least one separate messaging engine created. But for a server cluster as member of a bus it is also possible to create additional messaging engines and to configure them to run on specific cluster members. Therefore the related Redbook says: "Such a configuration enables a bus to be scaled to meet the needs of applications that generate high message volumes" [IBM2009][2], page 72.

Important is that advanced server combinations are possible. Nevertheless hence the Stuttgarter Workflow Machine is developed as a high performance workflow management system such advanced concepts might be of interest for future versions. For more information about the capabilities of the service integration bus related to messaging please refer to [IBM2009][2], chapter 2, or for security considerations [IBM2009][2], chapter 4.

A concept like the service integration bus needs additional resources and abstraction layers. Therefore some of the JMS resources can be created at different scopes. The most comprehensive scope for the purposes of this thesis is the cell scope, followed by the node and server scope. At the current stage of the project a single WebSphere application server is used and all scoped messaging resources are created at cell scope.

Furthermore servers or clusters can be members of more than one service integration bus, but the current implementation uses a single bus for the Stuttgarter Workflow Machine. That applies to the SOAP over JMS messaging support too. For further information about the configuration and administration of the WebSphere application server V7 see [IBM2009][1].

Java™ Enterprise Edition applications access the service integration bus through the JMS API to send and receive messages. The JMS destinations for this implementation have to be associated with service integration bus destinations. The interrelationship is shown in figure 2.

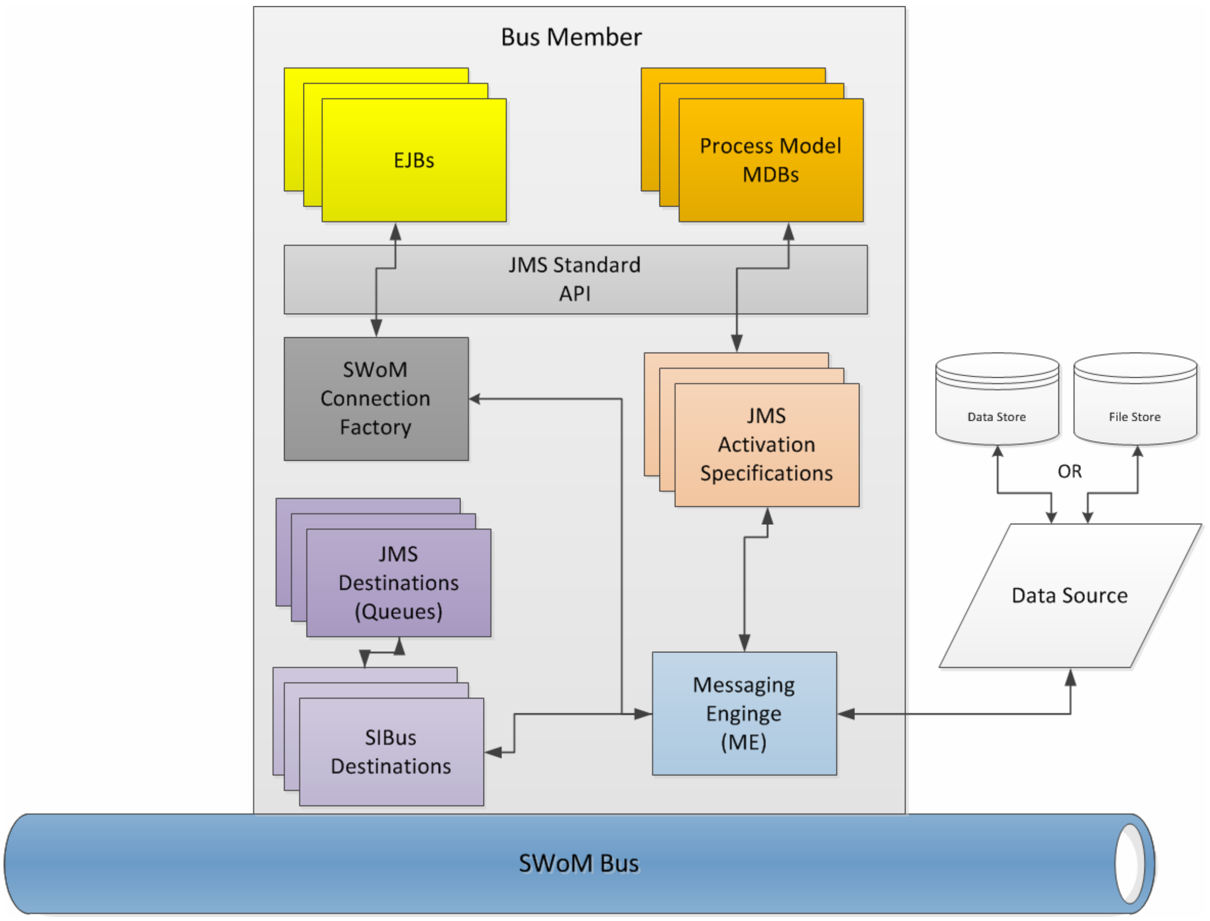

Figure 2. Service Integration Bus, JMS, and the WebSphere Default Messaging Provider

Except of the programmatically maintained JMS objects the administered JMS resources have to be created and configured by administrative actions. The next section describes the automated generation and configuration for needed resources.

2.3. Administrative Tasks

With a message driven bean-associated with each process model that will be imported and deployed the necessary administered object resources for JMS messaging have to be created and configured. This must be done before the actual deployment of the enterprise application.

For single scenarios and at the beginning of the development for the message-driven bean code the administered JMS resources were managed with the administrative console for WebSphere servers. But a sophisticated workflow management system must be prepared to deal with an unpredictable number of deployments and un-deployments of process models the JMS resources creation and configuration has to be automated.

The related Redbook says: “The administrative console is sufficient for tasks that are non-repetitive, have a minimal number of administrative steps, and are relatively simple. For administration that requires many steps, which can be repetitive, and time consuming to configure, wsadmin combined with scripts is an ideal tool”. [IBM2009][1], page 439. Following this recommendation an extensible script was developed encompassing all necessary administration and configuration tasks for the JMS administered resources. The following subsections are covering the major aspects.

2.3.1. Administration with scripting

The scripting interface wsadmin is based on an open source project called Bean Scripting Framework (BSF). The interface works between Java™ applications and scripting languages. For the demands of this implementation Jython as successor for Jacl was used as scripting language, which is state of the art with newer WebSphere versions like V7.0. Utilizing the architecture of BSF wsadmin can make various Java™ objects available to the Jython script. For a more information about wsadmin see [IBM2009][1]. An overview and figure to wsadmin scripting can be found at page 440.

The implemented Jython script uses AdminTask, and AdminConfig objects. The first object is used to query the existent resources and the second to create and delete required resources. During to the fact that the method calls especially to create resources are quite comprehensive they were encapsulated in Jython objects for a simplified handling. That means that all the configuration properties not needed for the development have default values. Nevertheless they are in implemented in the script and can easily be completed step by step to more complex resource configuration.

The script implements all necessary administrative and configuration tasks that are needed for JMS with the Java™ EE platform and EJB 3.0. For example are methods defined that can create a service integration bus or add a server as bus member. The choice of functions exposed to the Stuttgarter Workflow Machine was restricted to a minimum for clarity.

For example for all generated message-driven beans related to this implementation the connection factory of the Stuttgarter Workflow Machine is used. If this shall lead to scalability problems the implementation could be extended to create a connection factory for the replies of each process model. For even more sophisticated deployment scenarios the remaining functions of the script could be provided as additional command line arguments to the deployment code.

During the deployment process the script is run as external process for each process model. Vice versa for the un-deployment calling the correspondent delete functions. For this purpose the wsadmin executable is called with the script as argument and additional arguments that are handed over to the script specifying the configuration parameters for the administered objects. The next section discusses the details.

2.3.2. Running the Script as Java™ Process

It is important to call wsadmin either in the root directory of the target profile or with the profile as parameter. This implementation uses the first approach. For the simplified usage, that is the profile location where the Stuttgarter Workflow Machine was installed.

To pass the arguments for the configuration properties for the administered JMS resources through the Jython script run as command line tool a very simple syntax was used.

Example 2-1 Administrative Script Argument Syntax

```
[ -createDestination:<BusName>,<DestinationName>]
[ -createQueue:<BusName>,<QueueName>,<QueueJNDIName>]
[ -createActivationSpecification:<BusName>,<ActivationSpecName>,<ActivationSpecJNDIname>]

[ -deleteActivationSpecification:<ActivationSpecName>]
[ -deleteQueue:<QueueName>]
[ -deleteDestination:<BusName>,<DestinationName>]
```

The command names are self describing. After the command name separated by a colon the necessary parameters for the creation or the delete of resources are specified. If more than one parameter has to be specified they are separated by commas.

With the current script implementation only the absolute necessary parameters are passed. For additional parameters the script uses internally the default values adopted from the administrative console of the WebSphere server.

3. Stuttgarter Workflow Machine

The Stuttgarter Workflow Machine is designed to meet the demands of a high performance workflow management system. The architecture strives to benefit from all state of the art information technologies. Primarily the opportunities of the Java™ Enterprise platform are utilized.

Sophisticated application server frameworks like WebSphere provide a runtime environment for enterprise applications that can be configured scalable, robust, and secure. Multiple application servers can be combined to clusters and interconnected over a network providing shared distributed resources workload balancing and more.

The following section provides a short overview about the basic challenges for a modern workflow management system. For more detailed information about workflow management see [LR2000].

From an information technology perspective a sophisticated workflow management system must meet nearly all thinkable demands for a state of the art distributed application. It should be scalable, robust, secure, effective and efficient. Furthermore it should be loose coupled to support the integration of various other systems or already existing applications. Last but not least it should be able to support as most as possible business scenarios.

To meet all this demands is not an easy task. For example for the integration of heterogeneous hardware resources and applications additional layers of abstraction have to be added. A state of the art approach is to use a two level programming model. The actual tasks and their sequence constituting a workflow are modelled in an extra language. The most workflow systems implement the Business Process Execution Language (WS-BPEL) as state of the art description language.

WS-BPEL is standardized and specified. Furthermore the graphical aided design of WS-BPEL business process models is supported. The Business Process Model Notation is another specified standard defining a graphical representation for WS-BPEL models. It is possible to generate a WS-BPEL model from, a so called Business Process Diagrams (BPD) with BPMN.

As the name implies WS-BPEL concentrates on the execution of a workflow. To support as most as possible scenarios modelling languages like WS-BPEL are to some extend abstract. The sequence and interrelationships between units of work called activities are modelled. The actual implementation of single tasks is not part of a WS-BPEL business model description. Since WS-BPEL deals with the relationships between activities, which can represent various implementations of a task, the modelling process with WS-BPEL is often referred as programming in the large.

WS-BPEL defines the flow of a business process execution. There are a number of basic activities that determine the sequence and the data flow between the activities. Activities related to applications working with the provided business data are invoked as web services. Therefore a business process model description is often referred as orchestration. Like in a real orchestra the WS-BPEL process model is the score the workflow management system as conductor uses to carry out a workflow.

Nevertheless it is necessary to specify, which implementations are related to the activities and how they can be activated. This so called programming in the small part provides the necessary implementations and the interfaces to use them. The actual implementation can vary depending on the companies needs. To provide a maximum of flexibility for application and other active resources another specified language standard evolved, the Web Service Description Language (WSDL).

WSDL is a abstract description of web services. As the name implies web services can be computational services of all kind provided over a network, for example the internet. The real implementation realizing a web service is not determined by the WSDL description of a web service. WSDL defines and arranges sets of operations the web service supports. Furthermore a WSDL description can specify the exchanged data format for example by means of XML schema. Last but not least a WSDL description can provide real network endpoints. These can be for example a URL with a protocol like HTTP where the service operations can be invoked.

Web services allow not only to separate the actual implementation from the invocation but also to utilize external web services located anywhere in a network. As a result multiple instances of business processes can work collaboratively together. A business process can exchange data with other business processes or invoke new instances that perform needed processing. Insofar the architecture supports even collaborative work between different enterprises.

The real work behind a web service can be of any kind and complexity as long as the outcome is as expected. How the achievements of the service are realized does not matter for the service requestor as long the task is fulfilled. All details about web services and the correspondent WS-BPEL specifications and aspects are very complex and comprehensive. As an entry point to the subject please refer to [BPEL2007].

Important for this thesis is, that WS-BPEL and WSDL form a well specified and standard framework to deal with workflow management challenges. The abstract specification of both languages supports the modelling of nearly any kind of automated business processes, even in distributed and heterogeneous environments.

The next sections provide a brief overview about the architecture and the processing of workflows by the Stuttgarter Workflow Machine. Furthermore additional details about the deployment of process models and the messages exchanged because these parts have a notably relevance for this diploma thesis.

3.1. Stuttgarter Workflow Machine Architecture

The figures and discussion in this section are mainly based on information and presentation slides provided by Dipl. Phys. Dieter H. Roller.

The architecture of the Stuttgarter Workflow Machine can be seen as an advanced three tier architecture. An elaborated assignment of tasks enhances efficiency. Furthermore the utilizing of sophisticated caching increases the performance.

The Stuttgarter Workflow Machine combines several databases with processing components. The processing components are realized as Java™ enterprise applications. Therefore they can profit from the benefits a container managed execution provides. An example is the so called hot pooling of Java™ enterprise beans. The application server container can keep several instances of stateless Java™ enterprise beans ready for use. If such a bean is provided as web service the instance can be activated with an incoming request at once.

Figure 3 shows a picture of the basic architecture and the major components of the Stuttgarter Workflow Machine.

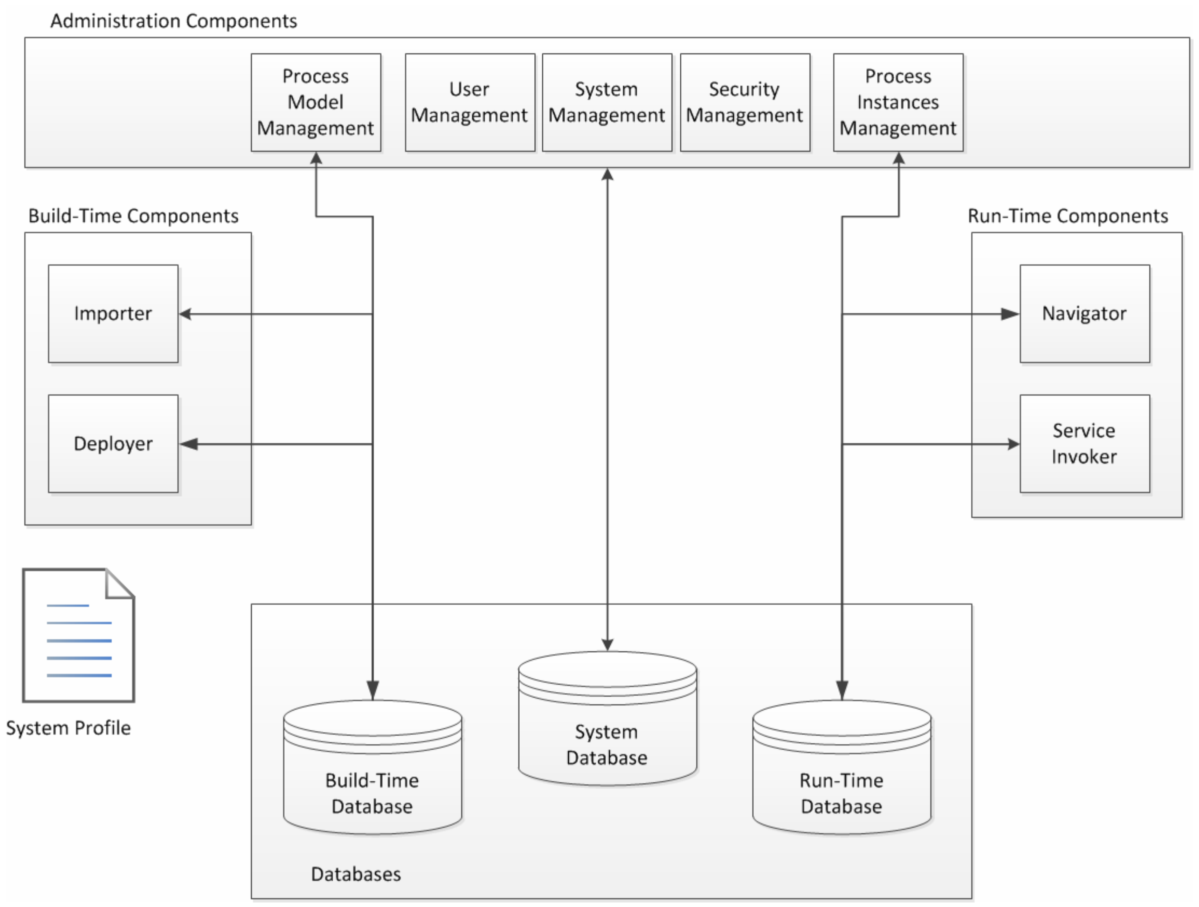

Figure 3. Basic Architecture of the Stuttgarter Workflow Machine

The components and databases shown in figure 3 constitute the backbone of the workflow system. For the purposes of this diploma thesis the build-time and run-time components will be especially important. The following will give a brief overview about the databases and components, even though the implementation of the SOAP over JMS support for the Stuttgarter Workflow Machine will not use direct SQL calls on the databases. Nevertheless the implementation and at least the design will be affected indirectly by all system parts.

As the figure implies the major components split up in three groups. There are administrative components, build-time components and run-time components.

As the arrows indicate state information is eventually stored in databases. For this thesis and the development of the Stuttgarter Workflow Machine DB2 V9.7 databases from the vendor IBM where used. Even though the challenges of modern database design, management and efficiency are well researched SQL calls are still relative costly.

Therefore the components interact as most as possible not directly with the databases but rather via caches. Optimally the caches provide all information that is needed at a specific point in time directly. The possibilities and challenges of sophisticated caching are very complex. Caching strategies range from simple assumptions that the same data set touched the last time will be needed once more over probabilistic prediction to the analysis of recent access scenarios. Additional consistency considerations have to be bearded in mind. The content of a cache especially in a multi threaded environment can become invalid. But if the rate of so called cache hits which means that the actual needed information is already in the cache can increase the performance of information processing notably.

The aspects of caching are out of the scope of this thesis. The design and development of the implementation is done under the precondition that the needed information provided by the information database access components is correct. This thesis concentrates on the directly related parts of the Stuttgarter Workflow Machine. These are rather the processing components than details about the databases access.

Therefore the following will give a short description about the responsibilities of the management and processing component groups.

The administration components are controlling and supervising the workflow management systems state. For example the management of user accounts and the corresponding security profiles. Security constraints have to be defined and enforced, for example the monitoring and control of user actions. These tasks belong to the security management components.

Furthermore a complex system like the Stuttgarter Workflow Management system has to be able to react on internal failures and deal with states and messages that cannot be further processed. Otherwise the system could be polluted with unusable resources. To support debugging, re-factoring, and further development the system activities have to be monitored. For example logging of failures and collecting useful information about the processing. Not only for further statistical and optimization analysis but also judicial issues a so called audit trail can be very useful. These aims can be subsumed as system management.

The model of a business process is deployed before it can be instantiated. This is done by the process model management components. Because these components are directly related to the purposes of this thesis they will be revisited and discussed in more detail in the next sections.

The process instance management components care for the actual instances of imported and deployed process models. For example the instances can be suspended or resumed. Another example would be an administrator with appropriate authorisation enforcing the termination and deletion of a process model instance.

The management components constitute the controller for the graphical user front-ends. They are realized as Java™ Server Pages (JSP) in conjunction with Java™ Server Faces (JSF).

The build-time components provide the functionality to import and deploy business process models. Because they are essential to the implementation for the SOAP over JMS support they will be discussed more elaborately in section 3.3.1.

The run-time components interact directly with the SOAP over JMS implementation as well. These components perform the internal processing of the deployed process model instances. Furthermore they can receive and produce requests and replies from and for other web services. Therefore these components produce prepare the data message content and call the corresponding operations to send them. These operations are implemented in special in part generated SOAP over HTTP and SOAP over JMS components. These components will be discussed in more detail too in section 3.3.2.

The business model description is the basis for the execution of business processes. Because these descriptions are the point of origin for further considerations, design decisions and the implementation of the SOAP over JMS support they will be discussed in more detail in the next section.

3.2. Business Process Model Description

The description for each business process model of the Stuttgarter Workflow Machine consists of three files that are packed into a common ZIP-encoded archive.

A WS-BPEL file contains the description for the workflow execution. Furthermore the archive includes one or more WSDL files with the description for the XML content of the exchanged messages and the abstract operation sets used by the participating web services.

The third file type is the so called Stuttgarter Workflow Machine Process Deployment Descriptor (SPDD) file with meta-information about the execution options for the process. The entries in the file determine amongst others the run time behaviour of the Stuttgarter Workflow Machine for deployment and processing of the correspondent business process model and instances.

Additional the Deployment Descriptor file provides the information about the end-point binding for the sets of operations that are exposed as web services. This part is especially interesting for the purposes of this thesis. Because the interrelationships are important to understand the next sections will take a closer look at each of the description files.

Figure 4 shows a sample flow described by a WS-BPEL file used for developing and testing of the Stuttgarter Workflow Machine.

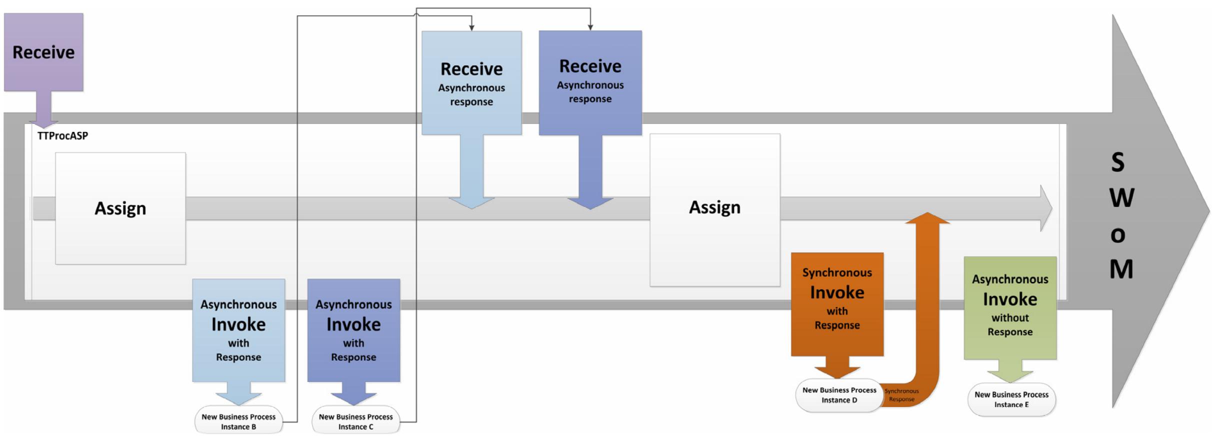
Receive
TTProcASP
Assign
Asynchronous Invoke with Response
Asynchronous Invoke with Response
New Business Process Instance B
New Business Process Instance C
Receive Asynchronous response
Receive Asynchronous response
Assign
Synchronous Invoke with Response
New Business Process Instance D
Synchronous Response
Asynchronous Invoke without Response
New Business Process Instance E
S W o M

Figure 4. Example Business Process Execution Flow

The execution of an instance of the sample process named TTProcASP has a lifecycle represented by the grey arrow in the middle. The great grey arrow represents the processing by the Stuttgarter Workflow Machine. There can be more than one instance of the corresponding business process model.

During his lifetime the process performs a number of activities. Three of them are so called receive activities, which means the process expects a message from a web service. These web services can be located anywhere in a network or running on the installation host of the Stuttgarter Workflow Machine. The receive activities are placed at the top of the processing arrow.

The first of the receive activities coloured in violet is a special one. First it is a so called solicit receive, that means it will not lead to a reply message. Second it is a start activity which means with the reception of a appropriate start activity message a new process instance is created. With that activity a business partner or client is able to initiate the execution of an instance of TTProcASP serving his demands. Example 3-1 shows the WS-BPEL part describing the initial receive activity.

Example 3-1 Receive Activity WS-BPEL Description

```
<receive createInstance="yes"
        name="A"
        operation="start"
        partnerLink="TTProcASPPL"
        portType="tns:TTProcASPPT"
        variable="InRequest">

        <sources>
                <source linkName="LinkAB"/>
        </sources>
</receive>
```

The receive activity has a name and is linked with a start operation. Furthermore there is a relationship with a partner link and port type and variable. The operation, port type and variable are related to specific entries in the corresponding WSDL definition. For example the variable represents the initial business data the initiating web service client provides. The details will be discussed in detail following the WS-BPEL discussion.

After the initial receive activity the process performs a data transition. A WS-BPEL process has an internal data flow. This data flow is realized by assign activities. The business data included in the messages are associated with WS-BPEL variables. The business data can be stored in variables and used for further selective processing. The processing results and the original business data can be used to constitute new messages. Example 3-2 shows the WS-BPEL description for an assign activity.

Example 3-2 Assign Activity WS-BPEL Description

```
<assign name="B">
        <targets>
                <target linkName="LinkAB"/>
        </targets>
        <sources>
                <source linkName="LinkBC"/>
                <source linkName="LinkBD"/>
        </sources>

        <copy>
                <from variable="InRequest" part="message1">
                        <query>//Field1</query>
                </from>

                <to variable="OutRequest1" part="message">
                        <query>//Field</query>
                </to>
        </copy>

        <copy>
                <from variable="InRequest" part="message1">
                        <query>//Field2</query>
                </from>

                <to variable="OutRequest2" part="message">
                        <query>//Field</query>
                </to>
        </copy>
</assign>
```

The example shows two copy operations from parts of the initial business data of the InRequest variable to parts of the OutRequest1 and OutRequest2 which will be used by following activities.

The next two receive activities belong to corresponding invoke activities with the same colour, blue and dark-blue.

Invoke activities realize the counterpart to receive activities. A business process can not only receive request messages but also send requests to other web services. Since every in- and outgoing communication is performed by web services an invoke activity means to act as a web service consumer.

The necessary business data is send as part of the message invoking an external web service. Precisely, external means for figure 4 that the invoked web service can be outside the actual process displayed here. The partner web service can run on the same workflow management system or anywhere else as long the correspondent message can be send over a network connection.

Figure 4 shows four outgoing invoke activities. The first two look similar at the first look but there is a difference. Both activities are asynchronous, which means the further processing of TTProcASP will not stop execution and wait for a reply message. But the messages send are targeted to different partner services. Example 3-3 shows the part of the WS-BPEL file describing an asynchronous invocation of an external web service.

Example 3-3 Invocation of a Web Service

```
<invoke name="C"
        operation="startTTProcBSP"
        partnerLink="TTProcBSPPL"
        portType="tns:TTProcBSPPT"
        inputVariable="OutRequest1">

        <targets>
                <target linkName="LinkBC"/>
        </targets>
        <sources>
                <source linkName="LinkCE"/>
        </sources>

        <correlations>
                <correlation initiate ="yes" set="correlation1"/>
        </correlations>
</invoke>
```

Since often it is not predicable how long the processing of a partner web service which can be a full blown business process of any complexity himself will take the invoke activities described here will not block the calling process. As the name of the used operation implies a new instance of the partner process TTProcBSP will be created. After successful processing the reply message is obtained by a corresponding receive activity. At least a fault message should be received when the partner service was not able to fulfil the request.

Since there can be more than one instance of TTProcBSP running even at the same destination a correlation mechanism is used to mark the outgoing request. With that information the workflow management system is able to correlate incoming reply message with a corresponding receive activity of the appropriate business process instance.

The asynchronous message flow is represented by the black arrows. The associated invoke and receive activities have the same colour.

The next two invoke activities colored orange and green differ insofar that they do not have a correlated receive activity. Furthermore the orange one is synchronous. Example 3-4 shows the WS-BPEL description for the synchronous invocation activity.

Example 3-4 Invoke Activity WS-BPEL Description

```
<invoke name="H"
        operation="startTTProcDSP"
        partnerLink="TTProcDSPPL"
        portType="tns:TTProcDSPPT"

        inputVariable="InInvoke"
        outputVariable="OutInvoke">

        <targets>
                <target linkName="LinkGH"/>
        </targets>
        <sources>
                <source linkName="LinkHI"/>
        </sources>
</invoke>
```

This invoke activity has an in- and output variable. The invoke activity will start a new process instance of a business process named TTProcDSP.

But after the invoke TTProcASP will stop the execution this time and wait for an instantaneous reply from the invoked partner process. Again at least a meaningful fault message should be sent if the partner process could not complete the request successfully.

In contrast the last invoke activity does not expect an answer at all. The request by the invoke activity coloured in green is asynchronous. It will cause also the start of a new business process instance but this time no reply is required from the invoked partner process as long as the work is done.

Until now not all activities, aspects and details for the WS-BPEL descriptions where discussed. For more information see for example the specification at [BPEL2007].

The last part took a closer look at the business process execution description in WS-BPEL. The process described is provided as web service and uses other web services, which can be business processes themselves. For the actual message exchange further information has to be provided. This is particularly the description of the related web services and the message structure they use. The next part will take a closer look at the WSDL definitions used by TTProcASP.

The description of a web service can be subdivided into two parts. One part defines the abstract functionality and the used data types of the web service. The other part can contain concrete information about how and where a web service can be activated. The second part is closely related to the actual implementation of the web service. The sometimes called physical part can include a binding to a network transport protocol and actual network addresses that can be used to invoke the service. Both parts are enclosed in the <definitions> element in a WSDL description.

The following describes the logical abstract part. After a closer look to the general elements of a WSDL file the concrete part will be discussed. The abstract part can be subdivided into a number of elements.

The Types definitions bracketed in the <types> element of the WSDL description defines the data types used in the <message> elements. XML Schema Definitions (XSD) is the most widely data typing method and also used for example 3-5.

Example 3-5 Types Element in WSDL

```
<wsdl:types>
        <xsd:schema targetNamespace="http://iaas.perfTest.org/datatypes"
        xmlns:dt="http://iaas.perfTest.org/datatypes">

        <xsd:element name="message1">
                <xsd:complexType>
                        <xsd:all>
                                <xsd:element name="Field1" type="xsd:string"/>
                                <xsd:element name="Field2" type="xsd:string"/>
                        </xsd:all>
                </xsd:complexType>
        </xsd:element>
</wsdl:types>
```

The example shows a complex type that consists of two string element. Such definitions are used to determine the types a message can contain.

The messages elements define the actual format of the messages exchanged by web services.

Example 3-6 Message Element in WSDL

```
<wsdl:message name="message1">
        <wsdl:part element="dt:message1" name="message1"/>
</wsdl:message>
```

Example 3-6 references the used data types by name. These message elements can be used within the <input> and <output> elements of operations defining as the name implies the exchanged business data.

The operations a web service supports can be grouped to subsets of operations. The subsets are defined in port type elements.

Example 3-7 PortType Element in WSDL

```
<wsdl:portType name="TTProcASPPT">
        <wsdl:operation name="start">
                <wsdl:input message="tns:message1"/>
        </wsdl:operation>
</wsdl:portType>
```

The example shows a port type that has only one operation that will lead to the start of a new process instance of TTProcASP and is related to the initial receive activity coloured in violet in figure 4.

The set of operations defined in a port type element are bound to specific concrete protocols. “A <binding> element takes the abstract definition of the operations and their input/output messages and maps them to the concrete protocol that the web service uses” [CT2002] Example 3-8 shows a binding for one of the participating web services of TTProcASP.

Example 3-8 SOAP over HTTP Binding Example

```
<wsdl:binding name="TTProcASPPTBinding" type="tns:TTProcASPPT">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
                <wsdl:operation name="start">
                        <soap:operation soapAction="http://iaas.perfTest.org/wsdl/TTProcASP/start"
                                style="document"/>
                                <wsdl:input>
                                        <soap:body use="literal"/>
                                </wsdl:input>
                </wsdl:operation>
</wsdl:binding>
```

The type attribute of the binding element defines the port type that will be mapped to the binding. The child element <soap:binding> indicates that the SOAP binding extension is used to map the operations and defines the default binding style as document. The transport of the SOAP messages will be HTTP. The subsequent operations can define their own binding style overriding the default.

The mapped port type has only one operation named start. The soapAction attribute with an URI as value represents the action that shall be occur as result of a message arrival at the destination. For the purposes of the Stuttgarter Workflow Machine it refers an activity of the process model. The SOAP action will appear as the value for the SOAPAction header in the SOAP message.

Furthermore the encoding of the message is defined for the operation as literal. It is possible to define arbitrary encoding formats as long as the communication partners are able to process the encoding. TTProcASP uses XML documents and therefore a literal standard encoding. The location where the business data for the operation appears in the message defined too. For this example it is the body of the SOAP message.

So far the WSDL description determines how a web service requestor shall constitute the message and which transport protocol should be used. Missing is the actual network endpoint a web service consumer should sent the message.

Example 3-9 Service Definition

```
<wsdl:service name="TTProcASP">
        <wsdl:port name="TTProcASPPTBindingPort" binding="tns:TTProcASPPTBinding">
                <soap:address location="localhost/TTProcASP/TTProcASPTTProcASPPTBindingPort"/>
        </wsdl:port>
</wsdl:service>
```

Example 3-9 shows a sample WSDL service definition defining the network address appropriate messages can be targeted for. A service describes the actual endpoint of a web service. That can be for example an URI that is an URL. The port element defines the binding used which is the SOAP over HTTP binding presented previously. "The <port> element has a <soap:address> sub-element---an element defined as part of the SOAP binding extension. The <soap:address> element identifies the URL of the web service. If this service used a different binding extension, this element would be different as well" [CT2002].

Therefore a binding for SOAP over JMS will look similar but different for some parts as discussed in section 6.6.1 that covers the generation of SOAP over JMS bindings. The Stuttgarter Workflow Machine generates its own proprietary bindings during the deployment process, which will be discussed in more detail in the next section.

3.2.1. Build-Time and Process Deployment

Before an actual instance of a business process model can be created the description information has to be imported. Based on the information discussed in the last section the necessary data must be stored persistent in the corresponding databases. By means of this information an actual implementation has to be provided to enable the workflow management system to run the process instances and make the participating web services unambiguously reachable.

For a scalable and dynamic design the actual processing is hidden behind stateless façade beans. These beans are exposed to the network as web-services and care for the message un-marshalling and preparation before the navigator components executing the process instances are activated. The internal distributed processing will be discussed in the next section.

Figure 5 shows an overview about the steps for the import and deployment of business process models by the Stuttgarter Workflow Machine. The final deployment includes steps of code generation, which will be discussed in more detail.

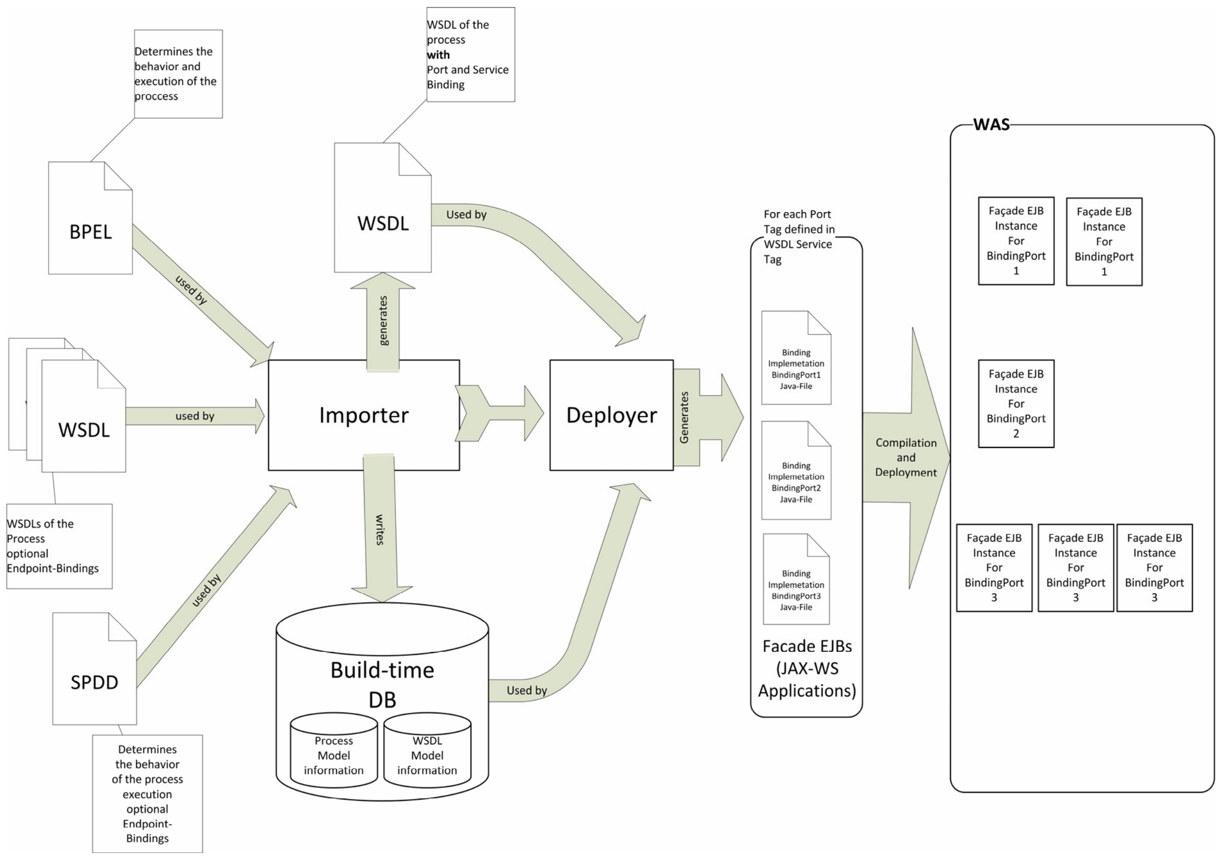
Determines the behavior and execution of the proccess
BPEL
used by
WSDL of the process with Port and Service Binding
WSDL
Used by
generates
WSDL
used by
Importer
Deployer
Generates
writes
WSDLs of the Process optional Endpoint-Bindings
SPDD
used by
Determines the behavior of the process execution optional Endpoint-Bindings
Build-time DB
Process Model information
WSDL Model information
Used by
For each Port Tag defined in WSDL Service Tag
Binding Implemetation BindingPort1 Java-File
Binding Implemetation BindingPort2 Java-File
Binding Implemetation BindingPort3 Java-File
Facade EJBs (JAX-WS Applications)
Compilation and Deployment
WAS
Façade EJB Instance For BindingPort 1
Façade EJB Instance For BindingPort 1
Façade EJB Instance For BindingPort 2
Façade EJB Instance For BindingPort 3
Façade EJB Instance For BindingPort 3
Façade EJB Instance For BindingPort 3

Figure 5. Stuttgarter Workflow Machine Process Model Deployment

The importer components read and parse the related files from the SPAR file describing the business process model. A data basis for the included activities is created. Furthermore WSDL definitions for the concrete endpoint bindings are generated.

For receive activities concrete endpoint implementations are generated. Either a façade bean for each receiving activity is generated or alternatively one façade bean including all receive activities. Figure 5 shows the first approach. After the deployment as enterprise web applications the façade beans can be invoked as web services. The related endpoint addresses and the description for the used transport protocols as well as the used message formats are included in the generated WSDL description.

The description here belongs to the deployment of applications related to the SOAP over HTTP message transport. Because the generation of corresponding code for SOAP over JMS support will follow a similar approach the following introduces briefly the steps performed for the generation and deployment of the façade beans handling the messaging with SOAP over HTTP.

The invocation of partner web services is handled by the invocation components and will therefore be discussed separately. The relationship and complete message flow will be introduced and discussed in the following subsections. The next part will take a closer look at the creation of components that deal with incoming requests.

The deployment components perform four steps based upon the imported process model data. First Java™ code is generated for the façade beans representing the receive activities. To obtain Java™ byte code that can be run by a Java™ virtual machine the previously generated code is compiled into class files. In next step the class files are prepared for the execution on an application server. The last step is the deployment on the application server as enterprise application. The next sections discuss the single steps more elaborate.

The Java™ code is created with the help of a number of code templates. These templates provide the code common to all generated façade beans. That is for example the code to lookup the needed enterprise resources like the navigator component. Furthermore there are templates for the necessary meta-information files the application server needs to deploy the façade beans as enterprise application. The variable parts are represented by placeholders in the template.

These variable parts and the combination of templates determine the relationship to the process model for example the associated receive activities. Amongst other the generated façade beans contain methods to retrieve the operation correlated with received messages. Furthermore the business data of received messages is extracted and added to an internal message that the navigator component can process. Figure 6 shows an overview to the structure of the generated code for a synchronous invocation of a façade bean.

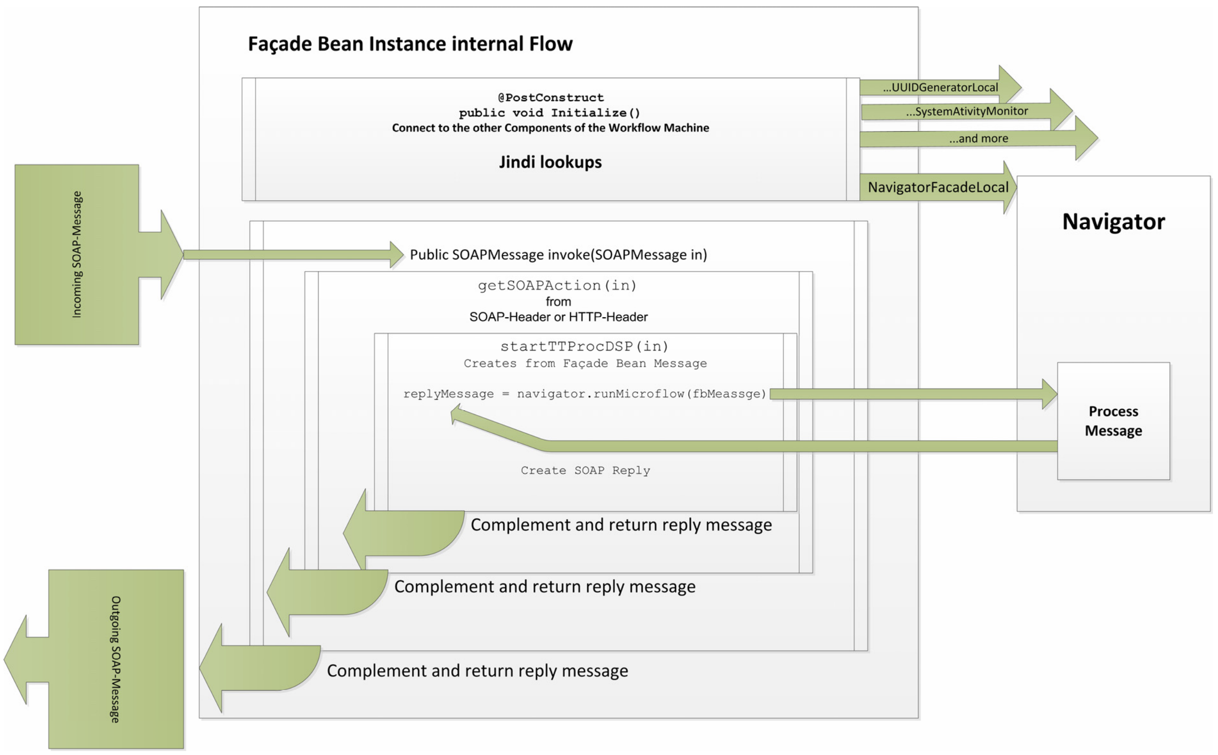
Façade Bean Instance internal Flow
@PostConstruct
public void Initialize()
Connect to the other Components of the Workflow Machine
Jindi lookups
...UUIDGeneratorLocal
...SystemAtivityMonitor
...and more
NavigatorFacadeLocal
Navigator
Incoming SOAP-Message
Public SOAPMessage invoke(SOAPMessage in)
getSOAPAction(in)
from
SOAP-Header or HTTP-Header
startTTProcDSP(in)
Creates from Façade Bean Message
replyMessage = navigator.runMicroflow(fbMeassge)
Create SOAP Reply
Process Message
Complement and return reply message
Complement and return reply message
Complement and return reply message
Outgoing SOAP-Message

Figure 6. Façade Bean Structure

For the internal correlation the generated Java™ implementation sets the Process Model Identifier (PMID) and the Activity Identifiers (AID) for the target actions in between the correspondent process. These values act as unique keys upon which the navigator components can query the corresponding database entries.

During the code generation the templates are combined to actual Java™ code. The Java™ code files are aggregated and complemented with necessary shared packages of the Stuttgarter Workflow Machine for compilation.

The Compilation of the Java™ Code is done as external process. The Java™ compiler (javac) is called as command line tool with appropriate arguments. That includes amongst other possible arguments the class-path to related Java™ archives (JAR) including the shared packages. These are for example the shared class files for the internal message declaration. The class files in Java™ byte code as result of the compilation are stored in a temporary system folder.

For the Creation of an enterprise application an appropriate temporary sub-folder structure is created. Afterwards the structure is completed with the necessary meta-information files for example deployment descriptors and archived into an Enterprise Application file (EAR).

The deployment of the EAR file is the last step in the sequence to make the process model available as enterprise application. The actual deployment is similar for the generated implementations of SOAP over HTTP and over JMS and will be discussed in more detail in chapter 6.

Once the process model is imported and deployed successfully the container can create instances of the deployed façade beans, which react on web services request. The generated and deployed façade beans are the same for each process model instance. The correlation to the actual business process target instance is performed internally by the navigator components.

The next section takes a closer look how the internal processing is done. Furthermore the relationships between the participating components will be discussed.

3.2.2. Run-Time and Process Execution

The actual processing of the business process instances is carried out as transaction. This transaction encompasses a set of navigator and service invoker transactions using stratification. The sub transactions are chained by internal messages. A set of navigator transactions is carried out as a transaction flow. If there is only one of such transactions it is called a micro flow. The approach ensures that every sub transaction in the chain can be rolled back.

Furthermore the execution is stateless. That means that the individual component instances maintain only temporary state information in between a single transaction. The business process state is stored in the runtime database. As mentioned before the use of sophisticated caching strategies can minimize costly database accesses.

The navigator and invoker components are realized as stateless or message-driven Java™ enterprise beans. That enables the application server to create as much instances as needed and even hot pooling. Furthermore the application server container can manage the transactional execution. Overall the approach allows for a quasi parallel execution of multiple process instances derived from different process models.

The internal messages chaining single transactions are exchanged via queues. A navigator component instance gets a message from the navigator queue and performs a navigation step. A resulting message from the processing of the navigation step is placed in the queue in turn. After this put operation the message could be fetched by another navigator instance performing the next navigation step and so on.

Another important aspect of the architecture is that a business process instance can wait if further processing depends on the execution of other web services without blocking passive processing components. The other web services a process is waiting for can be business processes themselves and therefore it is often unpredictable how long the processing will take. But a new bean instance will only be created or a pooled one activated on the arrival of a corresponding message. As a result no resources like volatile memory are wasted providing higher scalability. Instead the state is stored stable in the database.

By the time a business process needs to invoke a partner web service a message is placed in the queue for the invocation components. An instance of the invoke components will get it from the queue and handle the extern for example SOAP over HTTP call to the other process. The answer can be received either synchronous and be placed after processing and preparation by the invoking component again in the navigator queue for further processing or asynchronous.

If the asynchronous invoke demands an answer, it must be received. The reception of an asynchronous reply resulting from a invoke activities of the business processes are handled by instances of generated façade beans.

Figure 7 shows an extract of some examples of transaction chains that can be carried out.

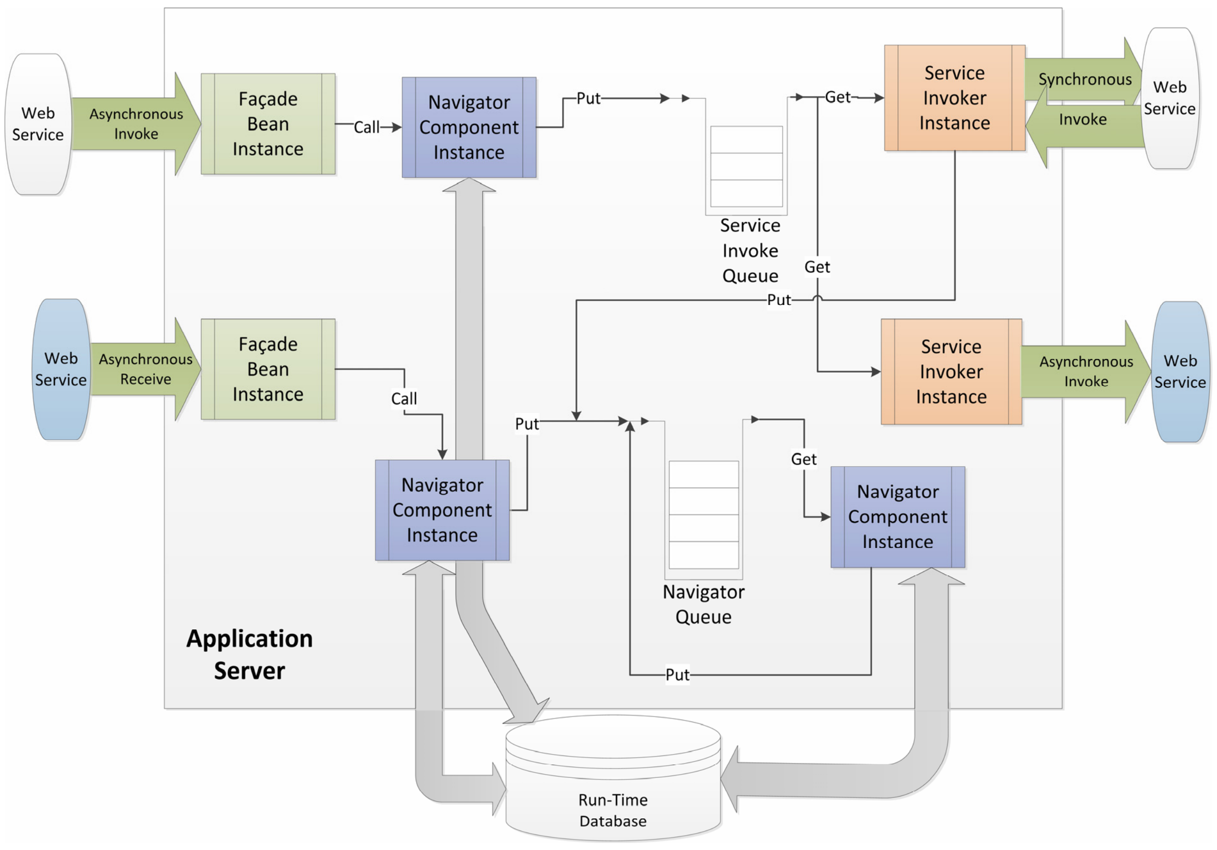

Figure 7. Stateless Execution of the Stuttgarter Workflow Machine

The grey arrows indicate the retrieval and storing of state information. The figure should only give an impression of the interrelationships. For example indicated by the colour the web service in the middle of the left hand side of the figure could be the same than the ones on the right side. That would mean that the invocation of the blue web service must be done before a reception on the left side can happen. So the internal arrows follow only the message flow not a timeline. Furthermore the web services are displayed as outside the application server. Of course they could be also processed on the same application server.

The figure shows only some of the message exchange pattern with other web services. Because the internal message flow is already implemented and will be used indirectly it is of no special interest for this thesis.

More important are the incoming and outgoing messages. Therefore the next section will take a closer look to the details. Consequently the internal execution of the business processes will be denoted only when needed or treated as black box.

3.3. Messaging Details

The business processes defined by WS-BPEL can act in concert with other web services. The interfaces for the message exchange are defined abstract in WSDL. During his lifetime a process can act in both roles, as a web service provider and a web service requestor. That means that a business process can be a message sender and a message receiver often multiple times. The next section describes the possible interaction patterns for abstract messages.

3.3.1. Message Flow

The WSDL specification defines the following message exchange patterns for transmission. The patterns are named Request-response, Solicit-response, One-way and Notification. The first two patterns involve a reply message the third and last define only a message in one direction.

The transmission primitives belong to WSDL operations. These operations can have an <input> and an <output> element. Their occurrence and sequence determine the transmission pattern.

For the Request-response pattern the <input> element precedes the <output> element. That means that the web service and therefore the correspondent process first receives a message performs his internal processing and sends the reply back. The message exchange can also be described as synchronous receive-reply. That means that the initiating web service will block or for stateless execution at least wait until the answer arrives.

The Solicit-response is to some extend the opposite. But the roles are changed. Staying at the process from the last section it will this time initiate the message exchange and block or wait for the reply. Therefore in the WSDL description the <output> element precedes the <input> element. The pattern is synchronous too.

The One-way and Notification pattern belong to no direct related message exchange. In both cases the sending web service does not expect a direct reply. Nevertheless they can be used to initiate an asynchronous request and reply message exchange.

That demands that a web service can determine that a message received is the reply to a message that the web service sent before. Such a correlation mechanism needs additional information. That can be either the endpoint where the messages are arriving or information in the message content self. In fact as later described a combination is used.

The WSDL description for a one-way message has only an <input> element. Analogous the description for the notification has only an <output> element.

Figure 8 shows the possible communication patterns.

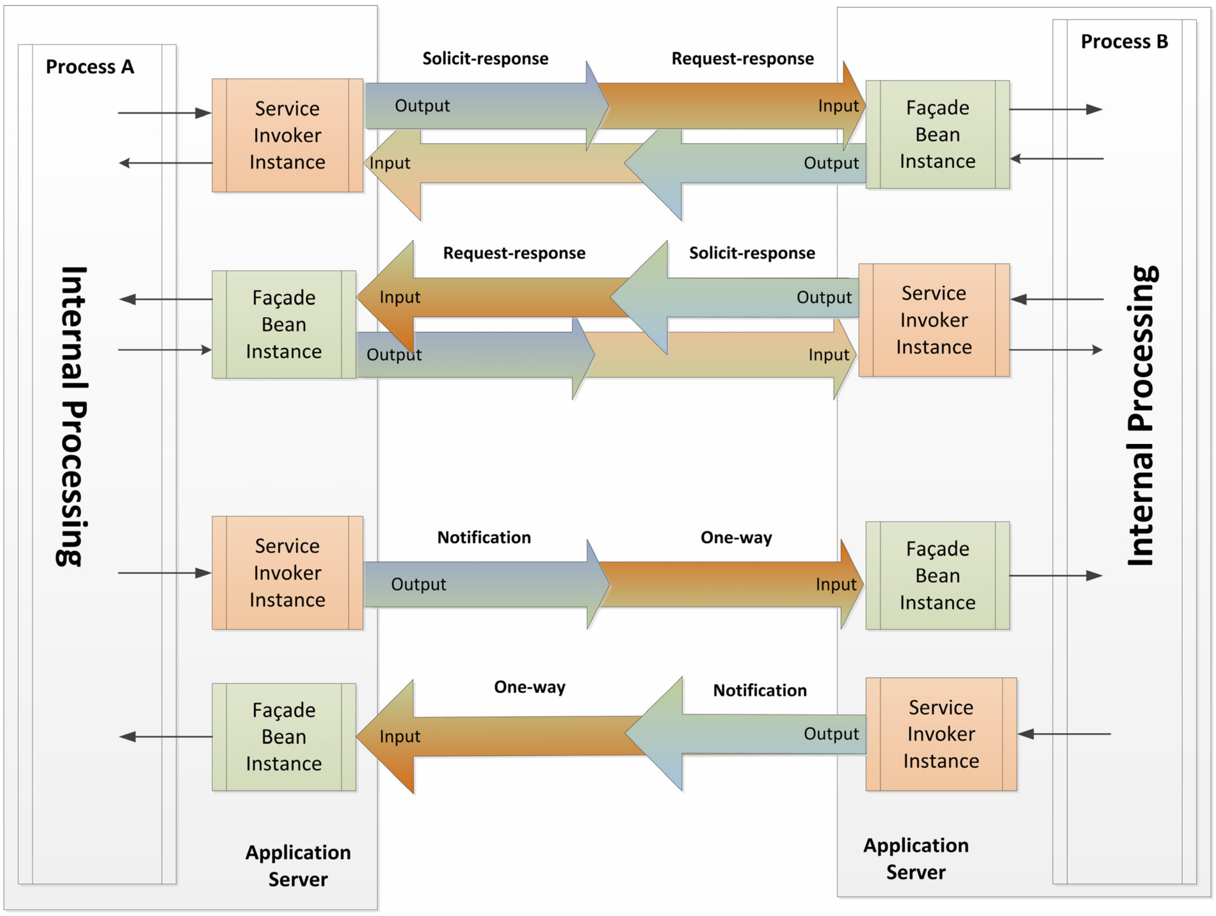

Figure 8. WSDL Transmission Patterns

From an abstract point of view the communicating web services are on an equal footing with each other. As the figure implies the transmission patterns are to some extend symmetric. That means that the participating web services act as web service provider or consumer or both depending on the exchange pattern and the point of view.

For the implementation of the SOAP over JMS support this means that the processes will act as message producers, consumers or both. If the first web service of two partners, of course there can be more than one, acts as producer the other one will consume the messages and vice versa.

So far the exchanged messages are described as abstract input and output for the operations a web service supports. Furthermore there is no transport defined. The original implementation uses SOAP messages embedded in HTTP requests and replies. The next chapter will discuss considerations, details and the resulting design decisions for SOAP over JMS messaging.

4. SOAP over JMS

The main different between SOAP over JMS and SOAP over HTTP is the transport protocol. As a matter of fact the SOAP part of the Stuttgarter Workflow Machine can and will remain nearly complete unchanged. It is irrelevant for the SOAP protocol on which network or protocol the messages are transmitted as long as they are delivered.

Strictly spoken the Java™ Message Service is not a network protocol. It is an application programming interface (API) that uses other underlying network protocols for its own purposes. But JMS can achieve the same as a network protocol like HTTP and more. The major advantage is the possibility for persistent messaging. HTTP for example needs a stable TCP connection. Or in other words at least an end to end connection from the client to the server is a precondition. For JMS all participants are clients and it is even not necessary that they are online at the same time.

The advantages of JMS for the message transport have their price. Additional resources like file stores or databases for the message storing must be maintained to achieve persistence. Therefore the next sections discuss the first basic considerations about the integration of JMS as messaging service into the Stuttgarter Workflow Machine.

4.1. Basic Design Decisions

The Stuttgarter Workflow Machine is a complex system. As a high performance workflow management system it cannot be seen as a single application. In realistic scenarios the system must deal with many process models which in turn can have multiple instances. All of the instances may wish to send and receive JMS messages multiple times during their execution.

There are two major challenges that must be considered. The first is the restriction that a single JMS queue cannot manage an infinite number of messages. The second is that every single business process instance must receive all and only the messages that are designated for this individual business process instance exactly once. Furthermore an individual process instance can have multiple send and receive activities exposed as web services.

Therefore the next section will discuss in detail major considerations about the queue relations. Furthermore about the message structure that will affect the dispatching problem between individual business process instances.

4.1.1. Message Queuing

The following section describes considerations about the design to connect the Stuttgarter Workflow Machine to for example business partners who want to federate in a business process. The considerations belong to point to point communication with JMS queues.

Assuming that a message producer is able to place a suitable message for the workflow machine into a queue there are some design decisions about the number of queues and their relationships to message driven beans, which act as message consumers for process model instances.

The communication between web service partners is to some extend symmetric. Therefore the next section will not consider a separation between request- and reply queues. The following considerations apply for both. Basically there are three approaches thinkable.

A single queue is used for the entire external message exchange of the whole workflow machine.

Multiple queues act as intermediate endpoints for each business process model.

Each process instance has its own queues.

For completeness shall be mentioned that mixed approaches would be also possible. Nevertheless such approaches are tending to incorporate the advantages and disadvantages of both designs, with additional programmatically overhead. A mixture would make only sense for some special scenarios.

The single Queue Approach seems to be at the first glance straightforward and easy to implement. At a closer look the approach proves as a solution with serious additional implementation effort and exposes some serious problems.

The advantages of a single queue approach are discussed in the following. The complexity of the generated WSDL file for each process model would be only enriched by a single port (WSDL 1.0) or endpoint (WSDL 2.0) for the binding. The same endpoint definition could be used for all process models and instances. Obviously a single endpoint can be a bottleneck and a single point of failure.

A single queue may ease the audit and control of messages. Furthermore the approach requires only a small and limited number of resources at server side, for example on the Service Integration Bus (SIB) of WebSphere. Thesee resources have to be configured only once for example together with the installation of the workflow machine.

On the other hand there are a number of disadvantages. Because the clients are connecting to a single endpoint, additional information has to be included in each message, to identify the process model the message was intended for. That means additional message overhead. Although the web service specification includes mechanisms for additional meta-information to enable correlation mechanisms at both sides the contract between client and service provider will be more complex.

Some components behind the queue, for example message-driven beans or the navigator components have to perform the dispatching of the messages to the correct process model and instance. That means more complex components and additional implementation effort.

To provide techniques to improve the effectiveness of the workflow machine like load balancing or to improve the quality of service with for example message priorities must be implemented in the workflow machine directly.

Furthermore critical situations are possible when the queue may not able to hold all messages from all clients. Therefore additional countermeasures against lost messages and as a follow up against duplicate messages have to be implemented. Last but not least, when the queue fails all business processes will be unreachable for that transport mechanism.

Even considering that the container may be able to create or even pool in advance multiple instances of message-driven beans to consume messages the approach eventually will not scale. It would be suitable for only one combination of message occurrence. This means that not more messages arrive together at one point in time than the queue can handle. Furthermore all messages for all process instances have to arrive at uniformly distributed points in time.

Because workflows and hence a workflow machine should be designed to be able to operate in a loose coupled and highly distributed system cooperative with all kinds of other business related systems exposed as web services it would be incautious to rely on assumptions about the message traffic.

Due to the fact that the JMS transport will used to support asynchronous data exchange and long running processes predictions about the message exchange over time are extreme difficult and without synchronization mechanisms impossible.

Overall considering that an application server environment like WebSphere provides a number of in-built features to deal with many resources including JMS administered objects the approach would require a considerable overhead of additional implementation with all subsequent disadvantages only to save already sophisticated implemented resources.

Furthermore the approach is eventually not scalable, which is not acceptable for a sophisticated, reliable and high available workflow management system.

From a high point of architectural view the approach requires a tighter coupling of clients and the server in means of necessary meta-information exchanged by client and server. In the end with a single queue as endpoint the whole workflow machine is exposed as single web service with many sub-services. Strictly spoken the approach violates the whole model of business processes as web services, with no more benefit than a more complex and less scalable system.

The following sections discuss the advantages and disadvantages of the approach with queues for each process model. The approach is similar to the design the Stuttgarter Workflow Machine realizes the reception of SOAP over HTTP messages for good reasons. Each business process model is associated with distinguished queues and the necessary runtime resources. Instead the façade beans for message receiving the messages of each queue can be consumed by multiple instances the number is determined by the application server container of message driven beans.

These message driven beans de-marshal the incoming messages and create a subsequent message with the necessary information for the navigator component to associate it with the correct process model and activity without further dispatching implementation. Figure 9 shows simplified how that part of message processing would look like.

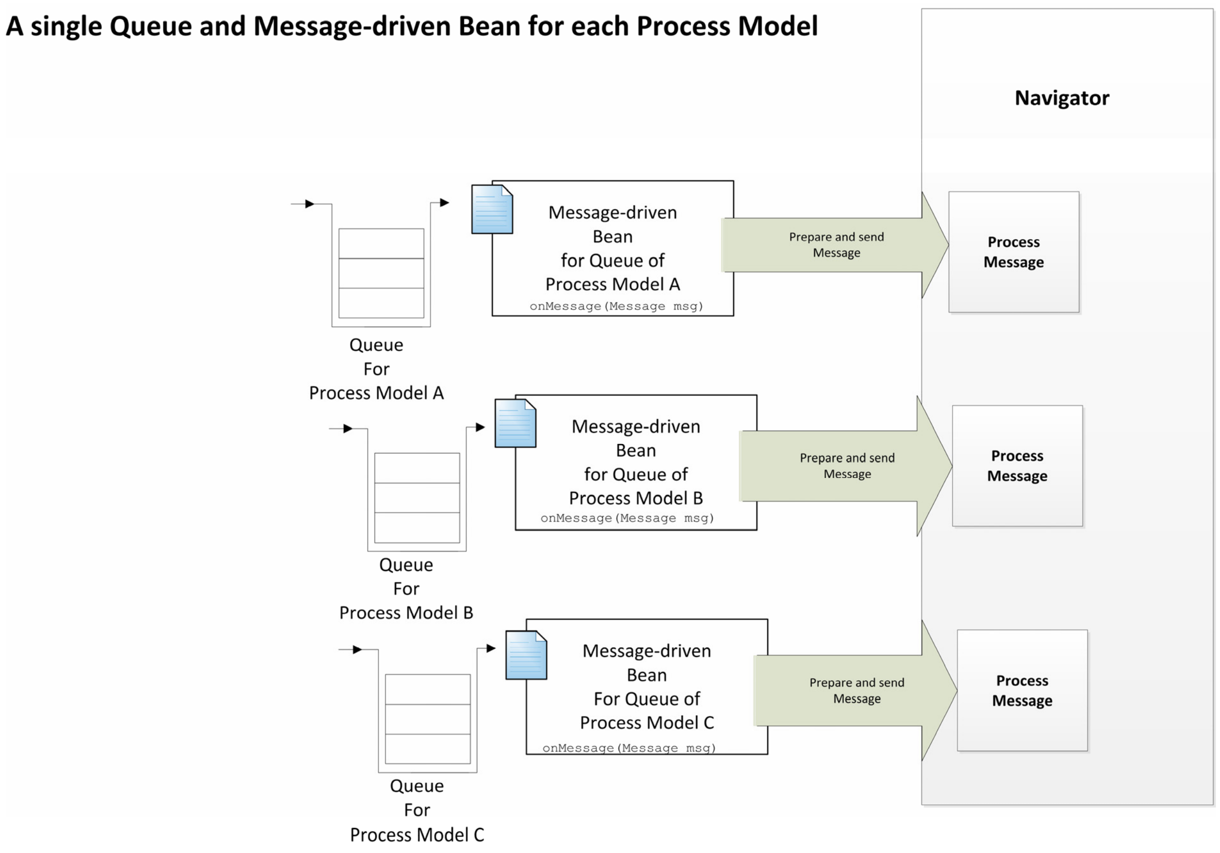
A single Queue and Message-driven Bean for each Process Model
Navigator
Message-driven Bean for Queue of Process Model A
onMessage(Message msg)
Prepare and send Message
Process Message
Queue For Process Model A
Message-driven Bean for Queue of Process Model B
onMessage(Message msg)
Prepare and send Message
Process Message
Queue For Process Model B
Message-driven Bean For Queue of Process Model C
onMessage(Message msg)
Prepare and send Message
Process Message
Queue For Process Model C

Figure 9. Queues and Message Driven Bean for each Process Model

A process model is correlated with an endpoint and therefore the incoming client messages without further meta-information needed in the message to determine the target process model.

The application server environment for example the WebSphere Service Integration Bus can perform optimization mechanisms for the resources. These would be for example load balancing on JMS-provider resources like messaging engines.

The inbuilt functions of messaging resources can be utilized, for example the priority of queues on the Service Integration Bus. Security handling could be configured for each process model separately. Furthermore the probability for a queue overflow is independent from the number of deployed process models.

Debugging and analysis could be easily focused on an individual process model by monitoring the messages of specific queues. Hence every resource creation is part of the process model deployment the administration components have full control and knowledge about the number of created resources.

Besides of the opportunity for clients to use another transport mechanism than for example HTTP the architecture of the workflow machine has not to be modified.

On the other hand the approach requires a set of resources for each deployed process model. To ensure correct dispatching the endpoints have to be unique for each process model. The creation and configuration requires additional steps for the deployment of the process models.

The application server environment has to process a number of resources that is eventually defined in advance. This may, besides the additional processing overhead for the resources, require dynamic administration at runtime.

Nevertheless the approach provides the opportunity to utilize the opportunities of a variety of properties from already implemented JMS components. Furthermore it fits seamless into the actual architecture for the price of the creation, configuration and administration of resources.A proper configured server environment with enough hardware and software resources, for example a cluster of federated application servers, would be able to deal even with high message traffic.

The third approach would be to associate queues with each process model. In theory possible the approach is besides the waste of resources not applicable. Hence the process instances are created at runtime as a result of the reception of a special message it would be necessary to create JMS resources dynamically. It is possible to work with temporary queues. But the specification of web services stipulates that the consumers can rely on the endpoints defined in a WSDL description. The question remains how this should be realized efficient and dynamically.

In the end every communication partner would be forced to determine the current endpoint respectively the queue dynamically for every message exchange. There are various solutions thinkable to solve the arising problems but all of them need additional overhead for example an extra message exchange in advance to negotiate upon a temporary queue. The only advantage would be that a queue overflow is nearly impossible. But for a price that is not acceptable.

Comparing the three approaches the decision is for the second approach with multiple queues. The approach demands the use of a number of resources or in this special case administered JMS Objects, but on the other hand these resources were designed, implemented and improved over years by experts. So it is not advisable to deal with already solved problems again. Professional application server environments are designed to deal with challenges of scalability, security and more.

The primary goals of the JMS support for the Stuttgarter Workflow Machine are that all correct messages shall be processed by the correct process model instance exactly once and as fast as possible.

The second approach with distinguished queues for each process model is the one that is able to meet all three challenges with well-balanced effort. Therefore in the following the details of an architecture and design with individual queues for each process model will be presented.

The following subsection will complement the discussion for basic details about the messages that will be used and a closer look to the main challenge the dispatching of incoming messages.

4.1.2. Message Structure

The implementation of SOAP over JMS for this thesis uses JMS messages that include SOAP messages as payload. This SOAP messages in turn contains the actual business data as payload. Both, JMS and SOAP message have a specified structure. Figure 10 shows the parts the combination of SOAP and JMS messages can contain and their nesting.

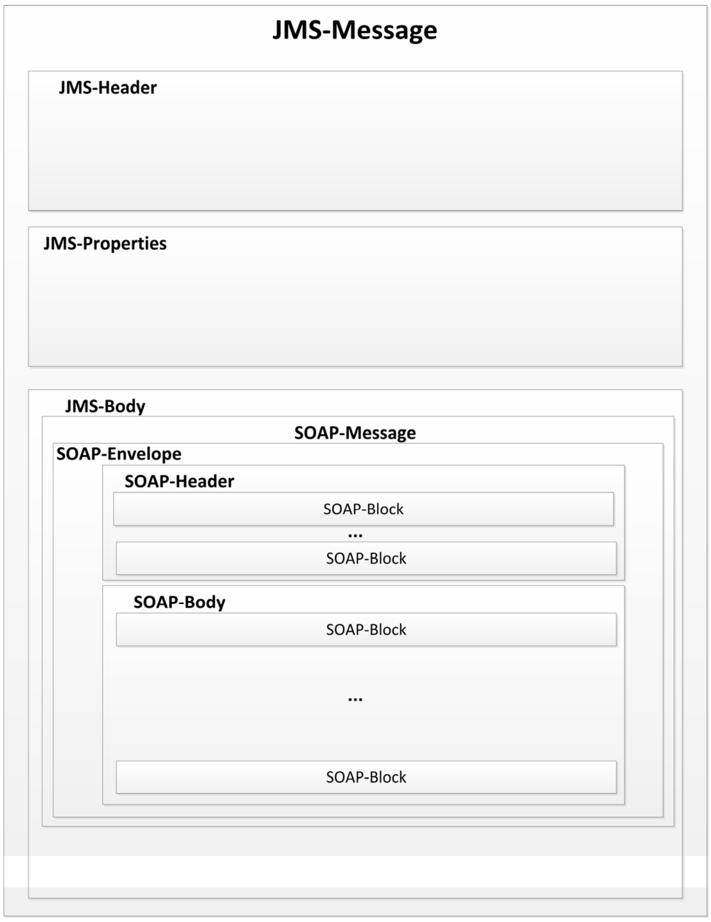

Figure 10. Nested structure of a SOAP over JMS message.

The parts are discussed starting with the innermost parts constituting the SOAP message to the outermost JMS parts. The encapsulated SOAP message in a JMS message is not much different from a SOAP message marshalled in a HTTP message. SOAP messages are used because they follow a common and specified standard for web services.

The optional SOAP header and mandatory SOAP body elements can have multiple child elements presented in figure 10 as blocks. The blocks embraced in the body element are used for the actual structured business data for example in XML. The SOAP body data is intended for the ultimate receiver. For the Stuttgarter Workflow Machine that is a receiving activity of an individual business process instance.

The header element content is intended for directives for the SOAP message processor. Both elements are wrapped into a SOAP envelope which delineates the SOAP document. The actual constitution of the SOAP message does not affect the SOAP over JMS implementation for the Stuttgarter Workflow Machine. For more information about the SOAP message standard please refer for example to [SOAP2007] or [CT2002], chapter 4.

Actual the JMS specification does not stipulate which kind of message format must be used for the payload. The SOAP over JMS working draft says “JMS is strictly an API and does not define a message format” [SOAP2010], section 2.1. But the document determines that “The contents of the JMS Message body MUST be the SOAP payload as a JMS bytes message or text message” [SOAP2010], section 2.4

Therefore the implementation for this thesis will use JMS bytes messages to be prepared for further enhancements for example SOAP messages that wrap non text data parts. For example SOAP supports the use of arbitrary encoded attachments separated and defined by Multipurpose Internet Mail Extensions (MIME) headers. For more information about so called multipart related SOAP messages can be found at [SOAP 2000]or [CT2002] Independent from the content for this implementation eventually the SOAP message is simply written via a byte stream as payload in the JMS message body.

More important for the purposes of this thesis is the JMS header and the optional JMS properties. The specification says: “Header - All messages support the same set of header fields. Header fields contain values used by both clients and providers to identify and route messages “ [JMS2002][2], page 30.

Additional to the default JMS header fields the JMS specification defines that “messages provide a build-in facility for adding optional header fields to a message” [JMS2002][2], page 30. As discussed later in more detail the possibility for self defined message header properties will be used to determine associations between asynchronous requests and the related reply. Furthermore JMS properties can allows internal dispatching or routing of a workflow management system. The next section will take a closer look to the basic aspects of the possible dispatching alternatives.

4.1.3. Message Dispatching

As mentioned before one challenge for the Stuttgarter Workflow Machine is to realize a unique specific point to point message transmission from a specific single web service to another specific individual web service. To some extend the problem commemorates to the routing in networks. As a matter of fact there are many similarities.

Together with the analysis of the JMS message structure there are two main possibilities to determine the dispatching of the JMS messages. One dispatching decision is given with the JMS destination. All messages that are sent to a specific destination will be only visible for the JMS message consumers that are listening on that destination.

As discussed before that dispatching mechanism will be used to determine the process model the message is intended for. After a successful connection and session establishment the message producer is created with the target queue as parameter. The JMS header properties for messages of that producer related to the target queue will be set automatically.

The second possibility is to encode dispatching information explicitly somewhere into the JMS message. In principle there are a number of options which part of the JMS message shall contain the dispatching information. The innermost location would be the payload or body of the SOAP message. In other words the location would be somewhere in the encapsulated business data.

The next location in the nesting would be the SOAP header. Implicit for both locations the dispatching information would be in the JMS body.

The outermost possible location is the JMS Header. Like for the routing in computer networks it is meaningful to provide the routing or dispatching information as fast as possible. That would be the outermost part of nested messages. The deeper the required information is encoded in a nested message structure the higher are the costs for the processing to find the exact position because the message must be unwrapped at least to that position.

The conclusion is to use JMS header properties whenever possible. As discussed later, optional JMS properties will be used to correlate the request and response for synchronous communication and to determine the target activity. These optional properties must be set by the web service requestor and are will determine the message driven bean that will consume the message.

The asynchronous dispatching to the correct process instance is different. To use header information for such a routing or dispatching information would mean that both sides have to remember the properties. In other words both communication partners would be forced to maintain state. It is well known that such approaches tend to further problems if the state gets lost. Furthermore approaches maintaining state do not scale in highly distributed systems with an unpredictable number of communication partners. As a result another correlation mechanism is needed and WS-BPEL specifies correlation sets for that purpose.

Because of the stateless execution the navigator components are responsible to determine the target process instance an asynchronous message is designated for. The message driven beans will take care for the dispatching only for the process model and activity respectively the target web service.

5. Design, Architecture and Implementation

There are many possibilities to connect process models over JMS queues. As founded in chapter 4.1.1 each process model shall be equipped with correspondent queues. The communication among web services encompasses single directed asynchronous message flows. Furthermore there are synchronous message exchange flows possible where one partner service sends a message, which is processed by his partner. The result of the processing is send back as reply, which the initiating process receives.

Therefore it is obvious to design the queue mapping with two distinct queues. One for reception of messages the other to handle the replies. That leads to the question to which one of the partners the queues should belong. For the message exchange itself the assignment of the queues to a process model is ridiculous at runtime.

As a matter of fact all messages could be handled by a single queue since the latest specification of the Enterprise Java ™ Beans like defines no constraints on the sequence the messages are delivered like First-In-First-Out (FIFO). For example the message consumption of message driven beans is controlled by the container and can occur at any point in time. But besides scalability problems such an approach would lead to a unpredictable sequences of message processing. At least for debugging and re-factoring reasons there should be a traceable structure for the queues.

Therefore in the following each process model is associated with per default or dependent on the used WS-BPEL activities at most two queues one for the receive of messages and the other for the produced replies.

The following sections will discuss the relationships and message flows between the queues and components. The next section is starting with the asynchronous communication between two partner process instances.

5.1. Asynchronous Message Exchange

The main property of asynchronous communication is that no reply is expected. Therefore a reply queue is not used or needed. Nevertheless there are distinguishable patterns of processing hence the communicating web services are part of a business process instance. The differences have an impact to the implementation and generation of the participating components.

Figure 11 shows a possible asynchronous interaction pattern between business process partner instances. The figure is derived from the approach the Stuttgarter Workflow Machine uses to handle the send and received JMS messages. This is not a must for external processes. As a matter of fact an extern partner that is invoked or receives a message asynchronously must not be a process at all. As long it is a web service that can receive and send messages from and to the queues it can interact with processes from the Stuttgarter Workflow Machine. Of course each partner must be able to consume and create appropriate JMS messages.

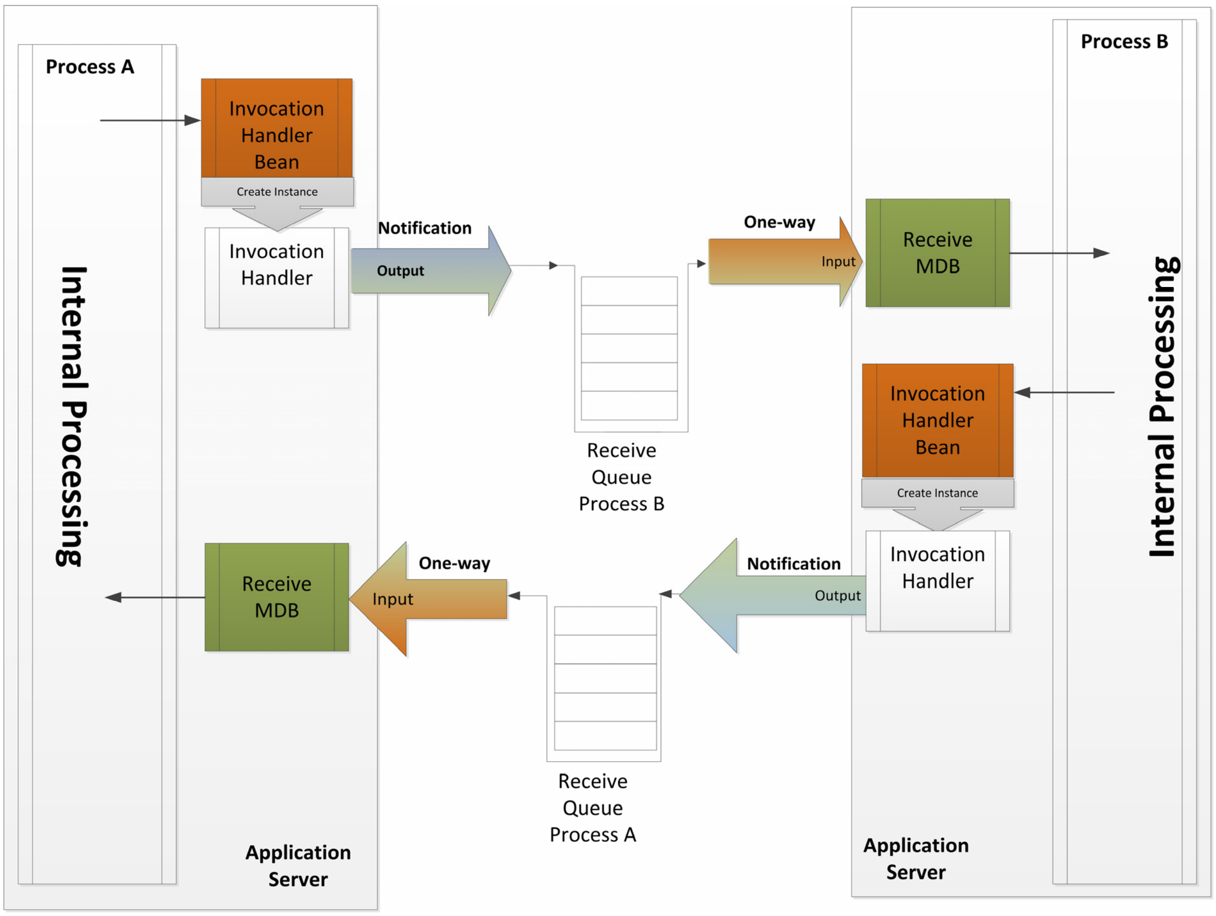

Figure 11. Asynchronous Invocation

The steps between the internal processing and the components handling the messaging are not shown in detail. For example the invocation handler bean can be a stateless session bean which is used as resource by the navigator components. They could be accessed by dependency injection and method calls from the navigator.

Another alternative are stateless message driven beans. If an appropriate message arrives at the internal invocation queue the application server creates an instance of the message driven bean. In the post-construct annotated initialize method an instance of the invocation handler component is created, which is dealing with the transport dependent creation and sending of the messages targeted to a from a the view of the process external destination.

The communication is initiated by a web service of the process on the left hand side. The invocation handler prepares and creates an appropriate JMS message and places it in the receive queue related to the web service it wants to utilize. In WS-BPEL this is related to an asynchronous invoke activity. Because the invoking process does not expect an answer at this point in time the activity is performed as a notification operation in WSDL terms.

For this example the invoked web service is part of a partner process. In WS-BPEL the related activity would be an asynchronous receive. The correspondent WSDL operation is a one-way operation because the process is not supposed to deliver an instant reply.

If the partner process follows the approach of this thesis an instance of a message driven bean is created. The message driven bean was generated and deployed before during the process model deployment for process B.

After the internal processing to some extend symmetric steps are performed for the invocation of a web service from the initiating process. Then the initiating process can perform a correspondent receive activity. In this manner the partners perform a request response communication pattern.

Often the communication partner of a web service is referred as web service client. Hence for the relationships shown in figure 11 both partners are web services this is somehow difficult. From the perspective of each web service the relative partner on the other side is the client and vice versa. In more restrictive WSDL terms the figure shows four web services. First process A acts as a web service consumer and performs an invocation of a web service provided by process B. Later on the next interaction transmits the reply with reverted operations.

The interaction shown in figure 11 is not a must. Each interaction in both directions can stand alone. Furthermore both partners are interchangeable, which means that process B could initiate the communication too. Nevertheless the interaction shown here is very useful. Hence all intermediate communication steps are asynchronous none of the partners is forced to block or wait and can continue with its internal processing until the reply is indispensable for further work.

The internal processing before, after and between the invocations and receives can be of arbitrary complexity. The possibilities range from simple calculations of a single web-service to full blown business process executions. These executions can involve the interaction with other web services not shown in the figure. These web services can be part of other business processes in turn that can invoke and receive messages from additional web services and so on. Although difficult to illustrate the variety of possibilities to arrange interactions constitutes the mightiness of the architecture.

Because the partners providing the web services can be full blown business processes there could be more than one instance of each process model. With the design discussed here the process model is determined by the used queues. A message that is placed in a specific receive- or reply-queue is targeted to the corresponding process model. But the message must be dispatched to the correct process instance too.

For asynchronous communication WS-BPEL provides a mechanism for the correlation of message with a specific business process instance. The so called correlation sets can include a set of properties providing meta-information that can be used to identify the correct business process target instance. The correlation does not affect the SOAP over JMS part of the implementation directly. It is performed by the navigation components to determine the target process instance. Nevertheless it must be part of the transmitted message. Hence meta-information will be placed in the payload of the used JMS messages. Therefore it is not directly related with the transport of the messages.

For the correct message routing or message dispatching another mapping must be done. If the partners on both sides are full blown business processes they can include more than one receive or invoke activity. Therefore the message driven beans on both sides will be generated with a specific message selector. That means that all requests and reply messages will be handled by the same receive or reply queue for the process model. But only the message driven bean with the correspondent activity as selection criteria will consume the message. The implementation developed for this thesis uses the SOAP action as property in the JMS header. The details will be discussed in section 5.4.1 about the message driven bean structure.

The asynchronous request response pattern illustrated in figure 11 resembles the indirect return of processed data in programming languages by the call of a procedure or method of the requesting program part to deliver the response for a previous call. Therefore the message flow from the right to the left process in the second half of the figure is sometimes referred as callback.

The other possibility to deliver response data is direct or synchronous like the return object of a non void Java™ method call. There are similar transmission patterns for the communication of web services too. Therefore the next section will discuss these synchronous interactions in detail.

5.2. Synchronous Message Exchange

As well WS-BPEL and thus WSDL define descriptions of synchronous request response interactions. The main difference to the asynchronous communication is that this time a reply is expected. Furthermore the reply is expected immediately or at least in between specific time bounds. To start such an interaction the initiating web service solicits a response from the web service he wants to use. The first part of the invocation is similar as for an asynchronous invocation.

As a result from a message putted into the invocation queue or a direct call by the navigator components the instance of a bean is created. The bean handling the message will create an instance of the invocation handler too. But this time the invocation handler will not only deal with the preparation and sending of the request message. Instead it will wait and listen on the reply queue of the invoked web service for a specified amount of time. For the approach used for this thesis this is for example the reply queue of the invoked partner process.

Figure 12 shows the participating components and the message flow for a single request response interaction.

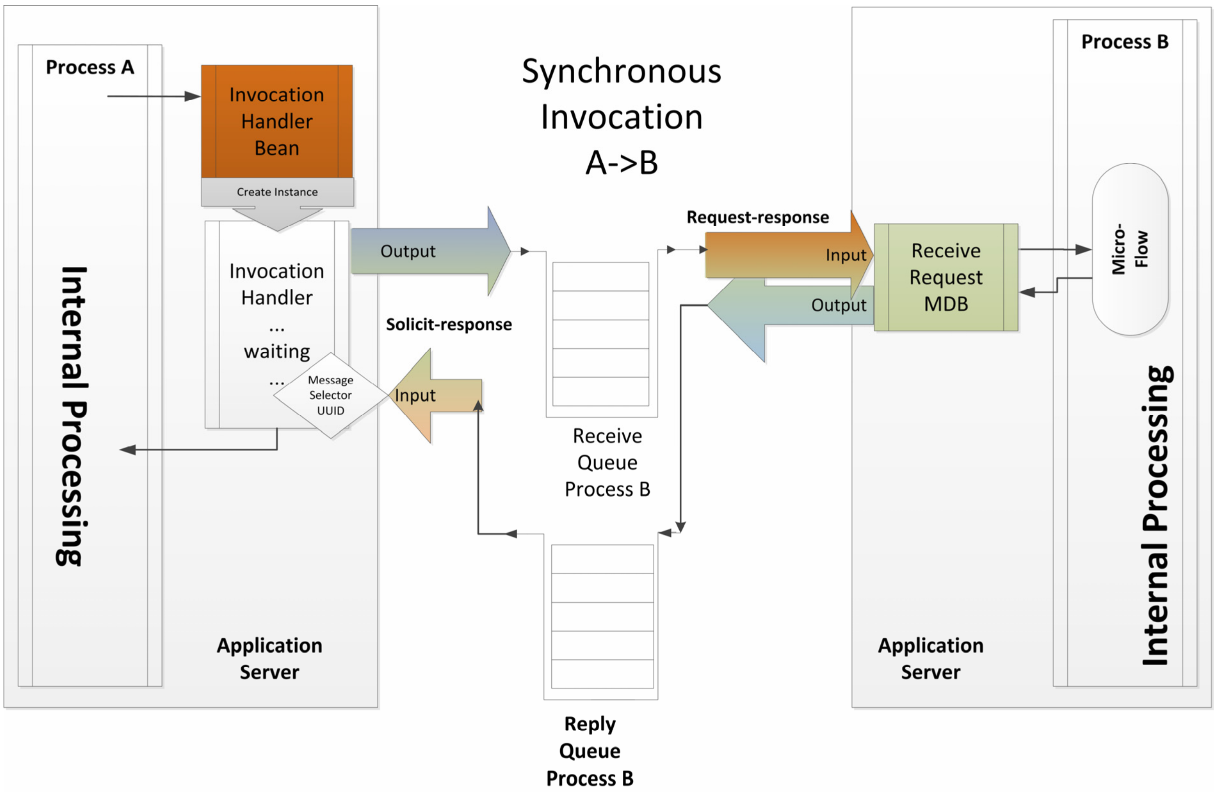

Figure 12. Synchronous Invocation

The illustration shows that both activities, the invocation and receive have an input and output. The message driven bean at the right side will not only consume the message from the receive queue but also act as message producer. Not shown here is that the message driven bean will use a message selector to fetch only messages that are input for the correspondent receive activity. The message is de-marshalled and prepared for the internal processing and handed over to the navigator components.

The invocation handler instance will create an output message and send it to the receive queue of the target web service. Hence the message producer is created from a session object, which means transacted serialisation of all produced and consumed messages the invocation handler has to call the commit method of the session otherwise the message will not be sent.

Since the request shall be handled synchronous the invocation handler expects an instant reply for his request. Therefore it will create a connection to the reply queue and listen for the arriving of an appropriate response message. The message selector is used to correlate the incoming responses.

There can be other response messages in the reply queue for example replies to the requests of other web services or the same web service but from another process instance the right one has to be identified. Therefore an unambiguously mapping from the sent request message to the correspondent reply message has to be provided. Actual the UDDI generator is used to generate a unique key value, which is provided as JMS Property in the request message. The replying web service has to repeat the unique key value in the response message. Since the invocation handler message selector has this unique key value as filter criteria it will fetch the correct reply message.

Figure 13 shows another detail, which is only exemplary. The partner process shown on the right side executes a so called micro flow. That is a process execution that can be performed in between a single transaction. These are processes which include no transaction boundaries for example a wait or receive activity. The navigator performs the process execution and returns the reply immediately. The message driven bean receives the navigator message from a direct method call. The content is marshalled into the JMS reply message and sent to the reply queue.

As for the asynchronous invocation the internal execution of the partner process can be a simple web service up to a process execution with more than one transaction. Figure 13 shows the same scenario but this time the message driven bean will not act as a sender. Instead the process finishes the whole internal execution. At the end of the process instance execution a navigator component prepares and sends a message to the reply queue of the requestor.

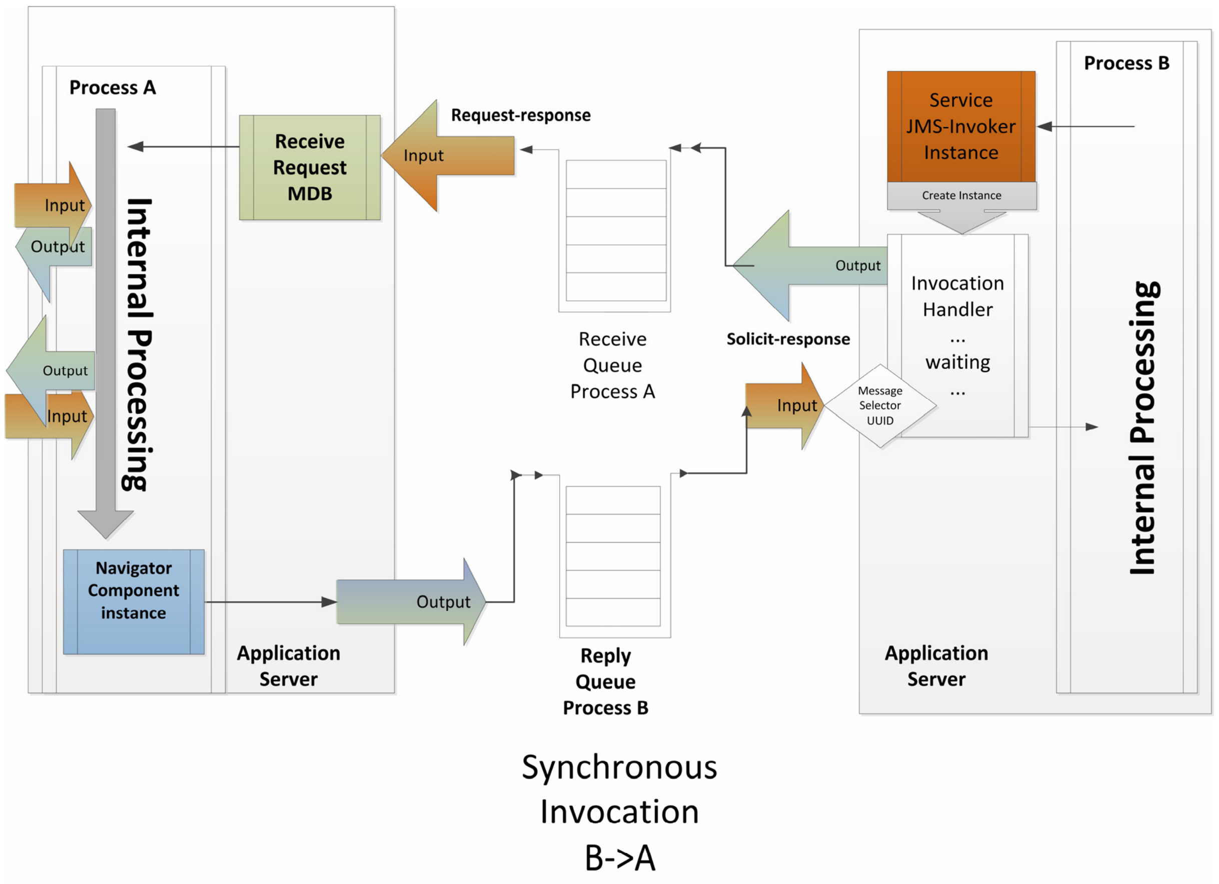

Figure 13. Detailed Synchronous Invocation

This time the initiating process is shown at the right side of the figure. As indicated by the arrows on the left border of the figure the internal execution of the process processing the request can be of any complexity. That encompasses all possible WS-BPEL activities. For example it can include other synchronous or asynchronous invoke and receive activities. Of course this applies also to the partner process on the right side but is not shown explicitly.

The next section will discuss an example. In particular a generated external client invoking a process synchronously executed by the Stuttgarter Workflow Machine.

5.2.1. JAX-WS SOAP over JMS Client

This section describes a small sample business process model presented to the outside as a web service by the Stuttgarter Workflow Machine. A WSDL description with concrete binding was used to generate a proprietary client to test the invocation of a synchronous call to the workflow machine starting a business process instance and receive the reply.

Due to the fact that the real implementation of a web service is transparent to the client it was possible to generate a fake web service and an appropriate JMS as transport service for the SOAP messages. If a generated client follows the specifications for web services it should be possible to fulfil requests independent, if a generated web service with a rudimentary implementation or a much more complex system like a workflow machine processes the request.

As expected it was only necessary to remove the generated web service the WebSphere application server to communicate with the deployed process model instead of the generated one.

The next sections describe briefly the properties and requirements of the generated client using JMS for the SOAP message transport. The decision for a generated client has several reasons.

First of all there is the save of time for the client implementation. There was already a sophisticated infrastructure for code generation implemented by experts with a lot of experience. Therefore it was obvious to utilize the capabilities of Rational Application Developer wizards and tools.

Furthermore relying on a professional developing platform ensures a certain compatibility with other professional code generation tools.

The generated client deals with all details of the SOAP Message generation and underlying XML representation of the business data in a sophisticated way.

Last but not least Rational Application Developer generates comprehensive client code including even Java Server Pages (JSP) as graphical user interfaces, which eases the call and testing of the web service. For the purposes here, this was the creation and start of a business process instance on the Stuttgarter Workflow machine up to the reception of the computational result from the sample workflow.

Nevertheless it shall not be omitted, that the generated code from Rational Application Developer is not WSI compliant. The check for WSI compliance has to be skipped hence the SOAP over JMS transport is already not included in the WSI compliance specification. Nevertheless is can be expected, that future specifications will follow similar approaches.

Maybe leave this subsection out, will be probably a repetition

The generation of both, the client and the web service rely on a number of JMS resources that have to be created and started on the WebSphere application server in advance.

Because details about the needed JMS resources were discussed in section 2.2 the following just provides a brief enumeration for the JMS resources used for the generated test client in conjunction with a synchronous process model invocation on the Stuttgarter Workflow Machine. These resources are sufficient for the demands of one business process model. As described in chapter 6 the resources have to be created for each deployed process model before the business process can be invoked as web service with SOAP over JMS messages.

Figure 14 shows a simplified view on the message flow and the participating JMS resources and components.

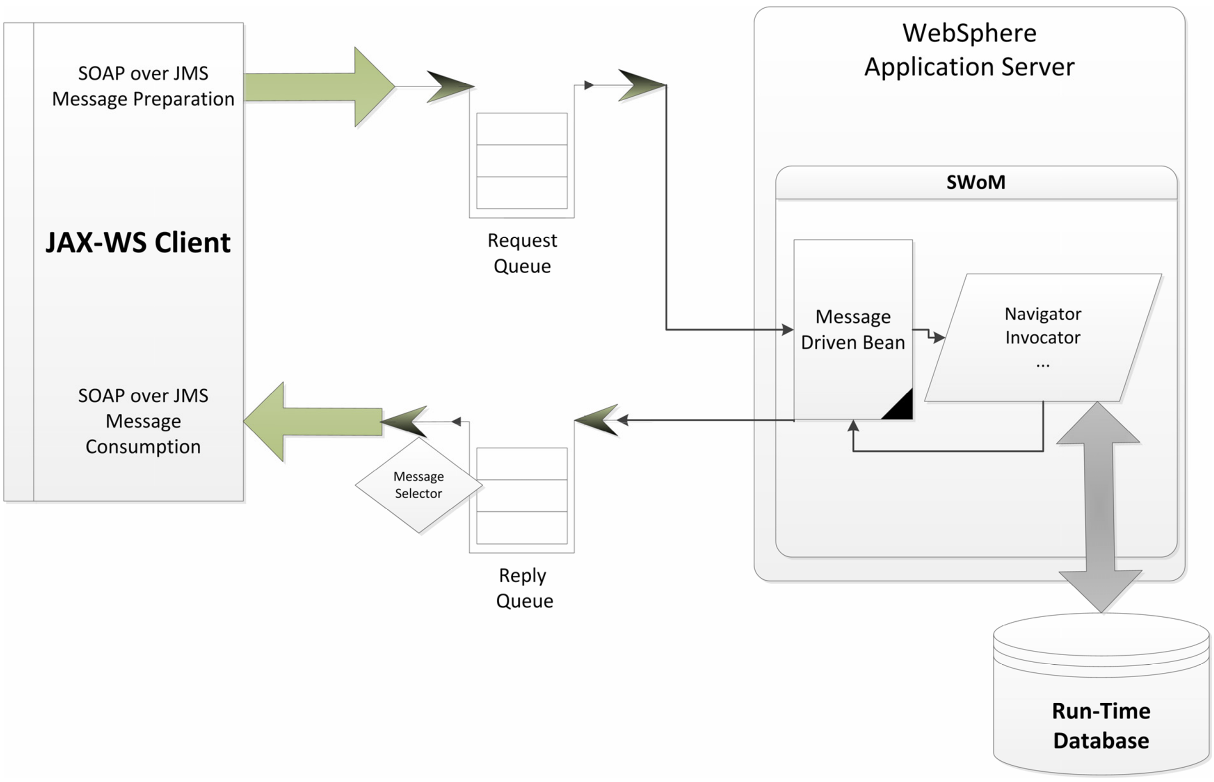

Figure 14. Interaction with a generated JAX-WS Client

The generated client and the workflow machine are connected primary by two intermediate queues. One queue is dealing with request messages and the other manages the reply messages. The client creates an appropriate JMS bytes message encompassing a SOAP message with the business data as XML document included in the SOAP body. The message is placed into the request queue.

The message-driven-bean from the workflow machine implements a message listener for the request queue. At arrival of a message an instance of the message-driven-bean is created or activated if there was a already a pooled instance. With the arrival of an appropriate message the onMessage(Message message) method of the bean is called.

First the request message is de-marshalled. For that purpose the bytes message payload is extracted. Then the SOAP envelope is isolated as String and a new SOAPMessage object created from the string object.

Furthermore in the onMessage(Message message) method of the message-driven-bean the target activity of the request message is retrieved from the SOAPJMS_soapAction string property set by the client for the JMS message. Depending on the SOAP action a method named after the activity with the new created SOAPMessage object as parameter is called processing the further workflow internal actions and calls.

For a synchronous invocation an internal equivalent for a FromFacadeBeanMessage is created. The necessary properties for this message are set. These are in particular the PMID (Process Model Identifier) and the AID (Activity Identifier) needed for the further internal processing by the workflow machine.

Both properties are generated from a generator implementation for unique identifiers and determined at deployment time. The PMID is effectual for every instance of the message-driven-bean serving for an individual process model. The AID differs from method to method representing the requested activity.

A synchronous call leads to a call of the navigator for the sample process used here initiating a micro flow, the method call blocks until the reply message is received from the navigator as return value. The response message is returned to the onMessage(Message message) method. Hence the sample process is called synchronously and will send a reply message an appropriate JMS message must be created.

For an asynchronous request the navigator call returns immediately without a return object for the onMessage(Message message) method and no further programmatically actions are needed. At this point in time the container could decide to deactivate or destroy the bean instance.

This section only applies to synchronous invocations of the message-driven bean. Only for synchronous requests from the client a response message is created. Then the call to the navigator initiating a micro flow returns a ReplyMessage object, which is a Stuttgarter Workflow Machine specific message format. The reply message is converted in a correspondent SOAPMessage, either a fault message or a well formed SOAP response message for requesting client is created.

In the end the OnMessage(Message message) method receives a SOAPMessage as reply for the synchronous invocation. To send the reply with JMS as transport, the SOAP part is extracted and written to the body of a JMS message. The JMS message including the SOAP response message is created as usual for JMS. First a QueueConnection to the reply queue is created then a session by the connection object. The session allows for the creation of a new bytes message. The SOAP content is written as string into the bytes message.

The required resources to create and connect to the reply queue can be retrieved by JNDI lookups in the @PostConstruct annotated initialize() method of the message driven bean, which means the resource lookups are performed after an instance creation of the message driven bean, but before any other methods, especially the onMessage(Message message) methods are called.

The generated JAX-WS Client requires some proprieties at the reply JMS message header. Furthermore the original generated client code or the URL calling the entry JSP must be changed. Without changes the generated code uses a temporarily created reply queue. To use a designated queue for the responses of a specific process model there are three ways to modify the behaviour of the client.

The first possibility is to provide the permanent reply queue as the optional JMS replyToName property in the URI of the JMS endpoint for the web service. The second is to set the JNDI name in the client implementation. The third variant would be to set the reply queue as Java™ virtual machine (JVM) system property. For tests with the generated JAX-WS SOAP over JMS client the first and second variant were used.

5.3. Message Flow

With SOAP over JMS as transport binding for the messages the challenge is to ensure that the correct messages are delivered to the appropriate process model instance exactly once. Or in other words the message routing or dispatching at each step in the message flow has to be ensured. The following will give an abstract or general summary of the dispatching steps and the resulting options for components and their execution.

Figure 15 shows the different possibilities and the resulting processing for the messages.

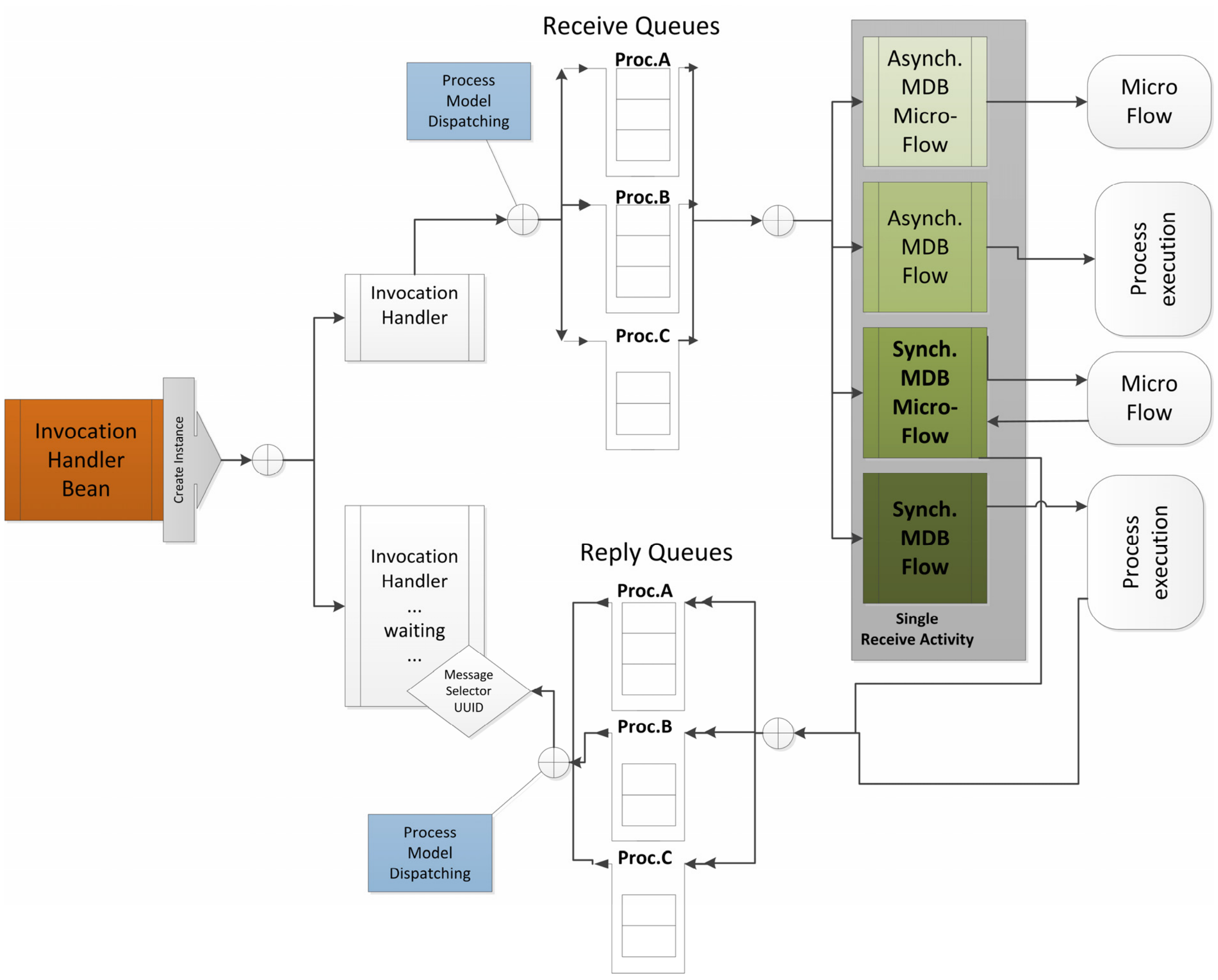

Figure 15. Message Dispatching

As a result of a invoke activity an invocation handler instance is created which will either only deal with the message sending or wait for a reply if one is expected. The dispatching for the correct target process model will be done by selecting the receive queue for the process model. For each process model message driven beans are created. Figure 15 illustrates the possible types of message driven beans. The message driven beans have in common that they all receive a message related to a receive activity of the process that they belong to.

Not shown explicitly is that there is another dispatching mechanism for the activity. There are two approaches. First for each activity a message driven bean is created which deals with the preparation of the internal message to the navigator. The second approach is that one message driven beans implements all receive activities of the process model. Such an implementation would include all four types of message driven bean implementations.

Figure 15 shows the second approach and the alternatives for the message-driven bean implementations. Depending on the type of receive activity the message-driven bean will call the navigator components with different methods. The call is either asynchronous or synchronous. The synchronous calls will lead to an instant reply message as result of the navigator processing.

Furthermore which navigator method is called will cause a different stateless execution from the navigator components. Either a micro flow or a full blown message execution could be performed. For the invocation of a synchronous micro flow execution the message driven bean will receive the reply message directly from the navigator method call as return value. This is the case were the message driven bean will act as message producer too. Otherwise the navigator must initiate the reply sending himself after the invoked business process instance has finished his execution. Hence the asynchronous receives will not produce a reply at all there is no additional implementation needed.

The target process model will be determined by the reply queue that is used. The decision string of the message selector will care for the dispatching to the appropriate invocation handler component. As mentioned before for that purpose the JMS message provides a unique property to identify the correct reply for a request.

Depending on the flow the message will take the implementation of the message driven beans differs. Therefore the next section will take a closer look at the implementation of the message-driven beans.

5.4. Implementation

The implementation of the related component is determined by the communication pattern and the constructed messages. The next section will discuss the implementation structure of the generated message driven beans handling the receive activities.

5.4.1. Message Driven Bean Structure

The structure of the generated message driven beans can be subdivided in distinguishable parts with different objectives. Figure 16 shows the related parts. The parts in violet are related with the necessary resources, instance variables, configuration properties and imports. All message-driven bean implementations will use a number of resources. For example references to navigator Java™ enterprise bean components. Furthermore of course references for the used JMS connection factory or queues. In principle there are three different ways to obtain references to Java™ enterprise resources since specification 3.0.

The first is a JNDI lookup using for example an initial context object. Such JNDI lookups can be performed in the initialize() method of the message-driven bean. With the @PostConstruct annotation the lookups will happen as first actions after the creation of a bean instance by the container.

The next one is to use for example the @Resource annotation at the instance or local variable instantiation. Without further information such bindings have to be defined explicit in a deployment descriptor for IBM WebSphere this is the ibm-ejb-bnd.xml file. Somehow special are the activation configuration properties, which determine for example the destination on which the message driven bean is listening for messages. These properties can be a set of @ActivationConfigProperty annotated properties with a name and value. The set himself is annotated with the @MessageDriven annotation.

If the message driven bean uses a message selector it can be defined as such an activation configuration property. The message selector is a string. The JMS specification says: "A message selector is String whose syntax is based on a subset of the SQL92 conditional expression syntax" [JMS2002][2], page 42. More detailed information about the syntax can be found there too.

Example 5-1 shows an example of an activation configuration.

Example 5-1 Annotated Activation Configuration Properties

```
@MessageDriven(
activationConfig = {
        @ActivationConfigProperty(
                propertyName = "destinationType", propertyValue = "javax.jms.Queue"),

        @ActivationConfigProperty(propertyName = "destination", propertyValue =
                                                "jms/SWoM_TTProcDSP_Request_Q"),

        @ActivationConfigProperty(propertyName = "messageSelector", propertyValue = "")})
```

The third option is to provide the information in the deployment descriptor(s) of the message driven bean. If annotations and deployment descriptors are used in combination the deployment descriptor definitions, bindings and configurations will override the annotations. The simplified EJB 3.0 API specification says: "Although it is not anticipated as a typical use case, it is possible for the application developer to combine the use of metadata annotations and deployment descriptors in the design of an application. When such a combination is used, the rules for the use of deployment descriptors as an overriding mechanism apply" [EJB2006][1], page 14.

If no annotations or JNDI lookups are used they will not be part of the code templates of the message driven bean. Therefore the following will concentrate on the implementation of the SOAP over JMS message handling part.

There are implementation parts that are common to all generated message driven beans. These are colored in green in figure 16. The other parts except the deployment, declaration or initialization parts are only needed if the message driven bean is intended to produce and send a reply message. As discussed before that would be a message driven bean implementing one or more synchronous receives.

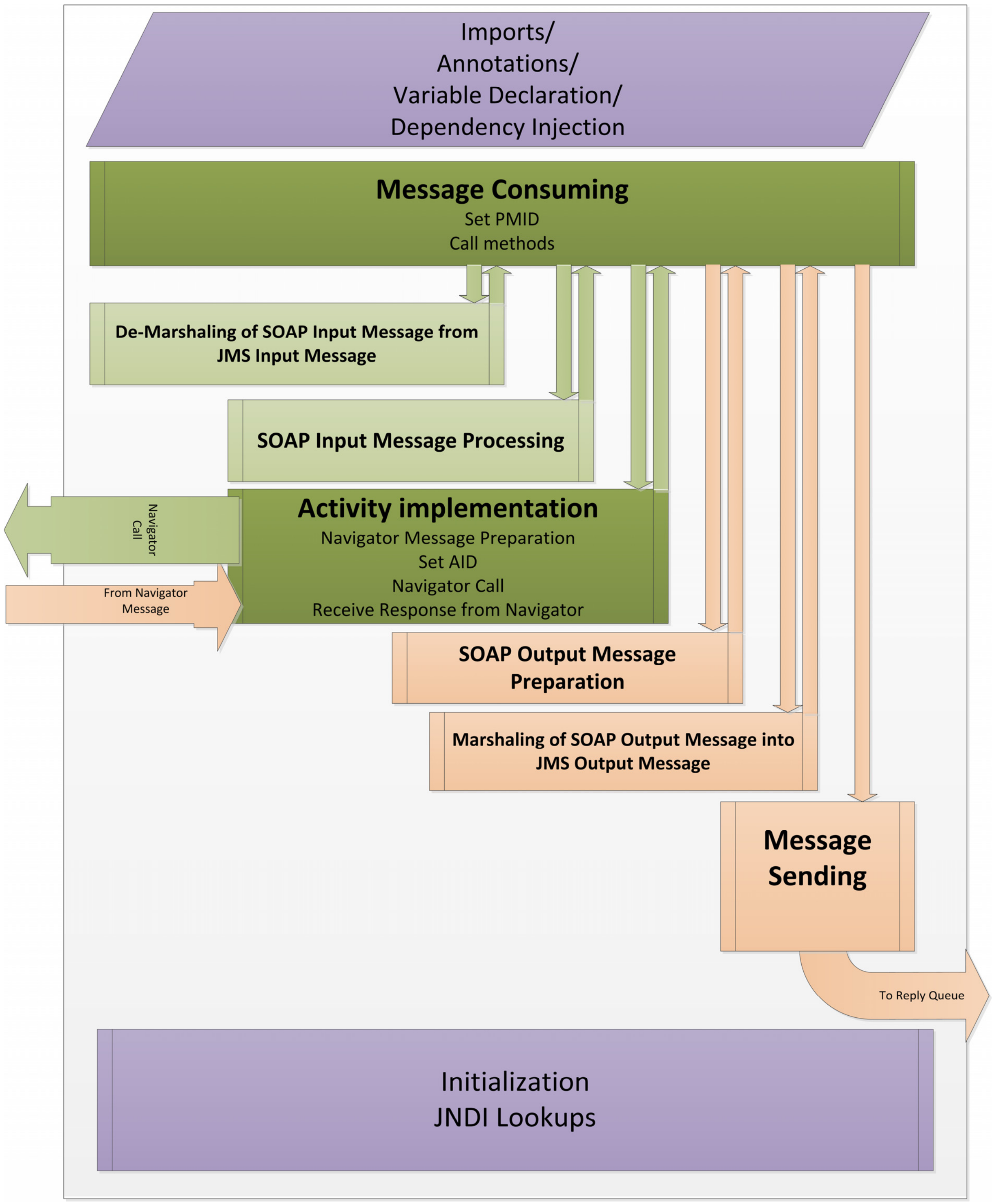
Imports/
Annotations/
Variable Declaration/
Dependency Injection
Message Consuming
Set PMID
Call methods
De-Marshaling of SOAP Input Message from JMS Input Message
SOAP Input Message Processing
Navigator Call
Activity implementation
Navigator Message Preparation
Set AID
Navigator Call
Receive Response from Navigator
From Navigator Message
SOAP Output Message Preparation
Marshaling of SOAP Output Message into JMS Output Message
Message Sending
To Reply Queue
Initialization
JNDI Lookups

Figure 16. Structure of the generated Message Driven Beans

Depending on the activation configuration properties the onMessage(Message message) method of the message driven bean will be called if a appropriate JMS message arrives. For the purposes of this implementation the message driven beans will listen on the receive queue of the process model they were generated for.

At this point the process model is already definitely determined. Therefore the current code templates for the message driven bean include the initiating of the variable for the unique process model identifier (PMID) in this part of the implementation.

If separated message driven beans are used to implement each receive activity the beans will use a message selector to filter for the SOAP action property of the JMS messages corresponding to the activity they implement. Only activities the message driven bean implements will cause the specific bean implementation to consume the message.

In figure 16 the message consuming is the starting point for further processing of the incoming message. In the onMessage(Message message) method the methods for the processing steps will be called and the result passed to the next part as indicated by the arrows. The following will discuss this call sequence from top to down shown in figure 16.

The SOAP message for the activity is encoded in the body of the incoming JMS message provided as input parameter of the onMessage() method. For further processing this message has to be de-marshaled from the JMS message.

The Stuttgarter Workflow Machine uses a special message format for the internal stateless execution. Those internal messages encompass the business data that was wrapped by the SOAP messages and mapping data determining the process model and activity. Furthermore additional parameters can be set for example indicating that the message is related to a start activity causing a new process instance. Therefore first the SOAP input message has to be processed and transformed into the format the navigator components need. The message processing parts are colored in light green.

The mapping data for the process model was set before. The unique activity identifier is set in a method that is associated with the corresponding activity. Therefore the method name is the same as the activity for better human readability.

The method is then responsible to create an appropriate message and will pass it to an appropriate navigator façade component. The following processing will be workflow machine internal

As mentioned before the following parts in figure 16 are optional. If the request belongs to a synchronous receive activity and starts a micro flow execution the navigator call will return a reply message immediately.

Hence the reply message will be sent to the requesting web service with SOAP over JMS an appropriate SOAP message must be created including the business data returned from the navigator. Afterwards the created SOAP message is wrapped into a JMS message and sent to the appropriate reply queue.

In figure 16 the step is called marshaling of the JMS output message. The step also includes the setting of the JMS properties for the reply message. For example a unique value the message selector of the message listener on the requestor side can be used to correlate the JMS reply message with his request. This unique value was provided as JMS property of the incoming input message and is stored for example in an instance variable of the message driven bean.

The last part shown in figure 16 is responsible for the sending of the created JMS message. That involves the usual steps a JMS sender performs. For the details please refer to section 2.1.

The message driven beans are the components for each receive activity. The counterpart on the other side can be a business process executed on the Stuttgarter Workflow Machine. Such a process can also call an external web service. The invoked web service can be part of an external business process. Nevertheless for SOAP over JMS an appropriate invoke or request message has to created and placed into the receive queue of the partner process. That is the responsibility of the invocation components. The next section will take a closer look to the realization and how the invocations are handled.

5.4.2. Invocation Components

An invocation is an activity that will be initialized by the navigator components. An internal JMS message with the appropriate business data is put into the invocation queue. The container will create or activate a message driven bean instances to retrieve these internal messages. The invocation message is extracted from the internal JMS message and handed over to a new instance of the invocation handler. Figure 17 shows these initial steps and the further processing by the invocation handler.

From the internal invocation message provided by the navigator a new SOAP message is created. In particular the XML business data is marshaled into an SOAP envelope. Hence the SOAP message will be marshaled into the body of a JMS message no explicit SOAP header properties are needed.

Before a JMS message can be created a reference to an appropriate connection factory and the reply queue of the partner service is needed. That can be done for example with dynamic JNIDI lookups. For this lookups the JNDI names are needed. This information is obtained by the SOAP over JMS binding description from the process that shall be invoked. All information about the binding was stored previously in the build time database during the import of the process model files.

Before sending the message the JMS header properties must be set. Some of the properties like the destination queue will be set automatically if they are not set explicitly. Other properties are obtained from the information that was stored in the build time database previously. Notably these are optional JMS properties that are needed if the invocation is synchronously.

Similar to the message-driven bean described in the last section there are some steps that are optional. These steps belong to the further computation that is needed if the invocation handler shall receive a response from the target web service. For that purpose additional dispatching information is needed to map the request message to a specific reply message.

As mentioned before an additional optional JMS property is set if a reply should follow the request message. For example a SHA-2 hash over the whole request message could be used as JMS byte property. Another approach illustrated in figure 17 is to produce and a unique string for the property. The current implementation uses a special generator for unique key values the Stuttgarter Workflow Machine implementation includes already.

If the partner web service will send a reply message back the property is set identically in the request and reply message. With the help of this value that must be unique as long as the message was not consumed successful the invocation handler can create a message selector and listen for messages with that JMS property and value.

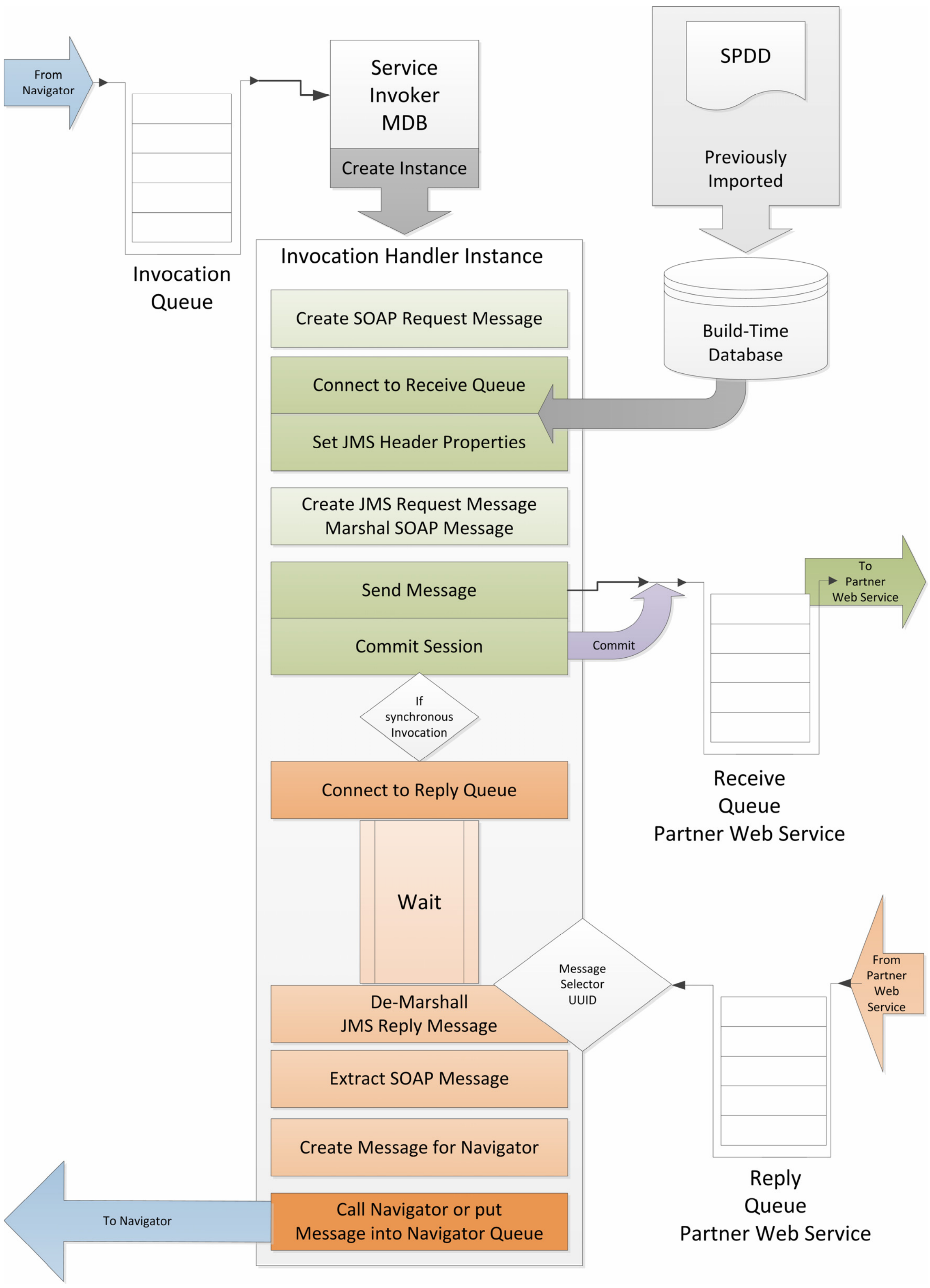
From Navigator
Invocation Queue
Service Invoker MDB
Create Instance
SPDD
Previously Imported
Build-Time Database
Invocation Handler Instance
Create SOAP Request Message
Connect to Receive Queue
Set JMS Header Properties
Create JMS Request Message Marshal SOAP Message
Send Message
Commit Session
Commit
To Partner Web Service
Receive Queue Partner Web Service
If synchronous Invocation
Connect to Reply Queue
Wait
Message Selector UUID
From Partner Web Service
De-Marshall JMS Reply Message
Extract SOAP Message
Create Message for Navigator
Reply Queue Partner Web Service
To Navigator
Call Navigator or put Message into Navigator Queue

Figure 17. Invocation Components

After the preparation of the JMS request message it will be send to the receive queue of the web service that shall be invoked. Regardless if the invocation is synchronous or asynchronous the invocation handler must commit the JMS session. Although there will be only one message send the JMS session is executed in a transactional context. Hence the implementation of the invocation handler as discussed here is not an enterprise bean the session must be committed explicitly.

An asynchronous invocation can finish after the sending. The last invocation handler method called by the invocation message driven bean would return without a return value. If the application server container decides to remove the invocation message-driven bean instance the reference to the invocation handler instance is not needed any more and the Java™ garbage collector can remove it. Otherwise the bean will wait for the next invocation message from the internal queue and the next invocation can be processed.

If the invocation is synchronous the invocation handler has to stand by for the arrival of the reply message. Actual the invocation handler will wait and listen for a specific amount of time. If no failures occur the message will arrive at the reply queue of the partner web service. The invocation handler instance with the corresponding message selector for the unique JMS property value discussed before will consume the message.

The further processing is straightforward. The SOAP message will be extracted from the JMS message body and transformed into an appropriate message for the navigator. The transformation could be implemented in the invocation handler. Alternatively the SOAP message can be the return value for the invocation message driven bean for the remaining preparation. The process model and activity identifier are already known from the previous invocation processing.

6. Process Model Deployment

The complete deployment of a process model on the Stuttgarter Workflow Machine can be subdivided into 7 steps. Figure 18 shows these steps and the execution sequence.

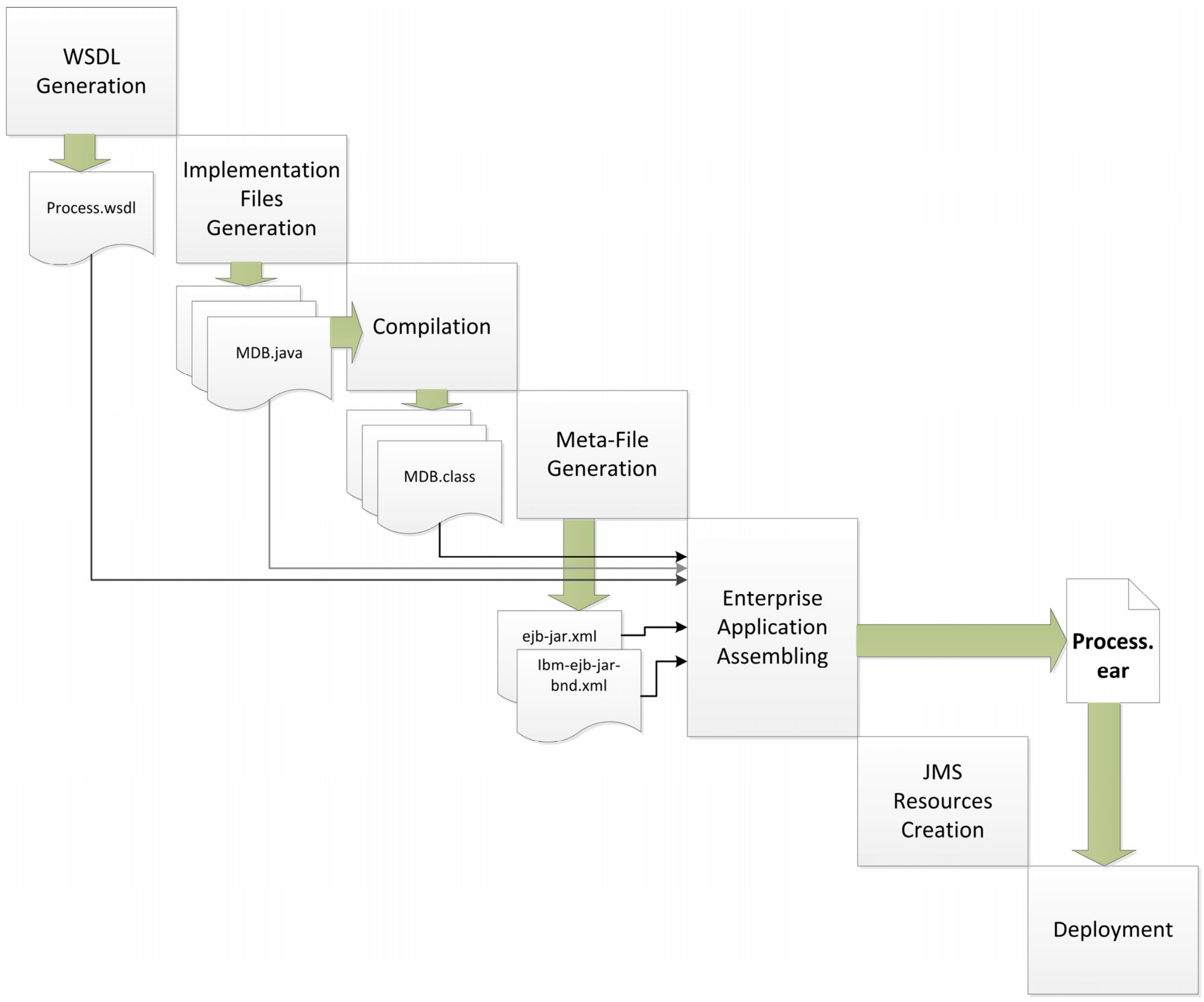

Figure 18. Business Process Model Deployment

The next sections are describing the actions for a process model deployment in more detail. The next section discusses the details about the generation of a WSDL description including the bindings for SOAP over JMS. The actions for the Java™ code generation, compilation and generation of the deployment descriptors illustrated in figure 18 are subsumed in the following section.

6.1.1. WSDL File Generation

The deployment of a business process model on the Stuttgarter Workflow Machine starts with at least three files. There is the WS-BPEL file describing the process execution and WSDL files describing the used data formats and sets of operations. Furthermore there is the Stuttgarter Workflow Machine process deployment descriptor (SPDD) that contains settings how business process instances will be executed by the workflow machine and information about the bindings that should be used.

At the start of the deployment these files will be processed by the importer components. The incorporated information is stored in the build-time database. From this database the responsible components retrieve the necessary data for the further deployment process.

Figure 19 shows the sequence of actions leading to the first step in figure 18. In particular this is the generation of a concrete WSDL file for the business process model.

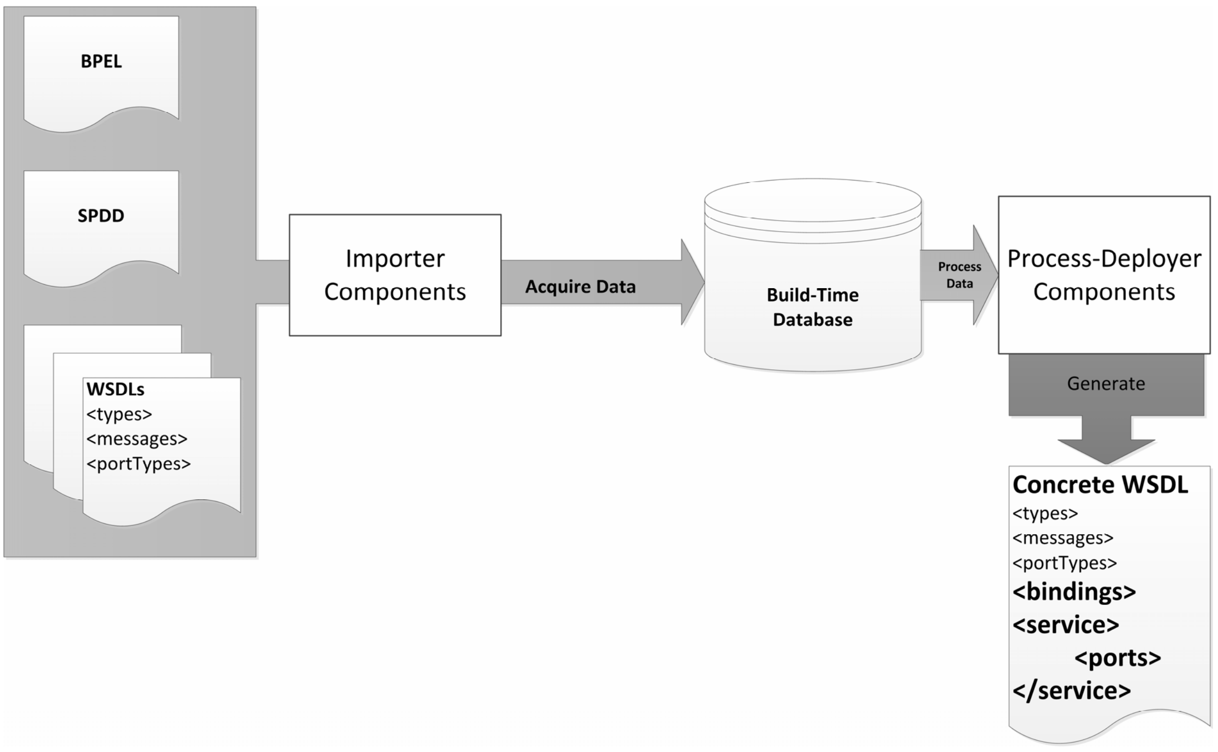

Figure 19. Import of the Process Model Definition Files

The settings in the process deployment descriptor include information about the virtual or actual network communication too. Or in other words specific bindings to the network and data exchange protocols used. The former implementation used bindings to SOAP as message wrapping protocol for the business data in XML. As the name implies the SOAP action determined the activity that will be performed as a result of an incoming message. The SOAP messages can use for example HTTP as transport protocol. If bindings for these protocols are specified they will be inserted in the generated WSDL file.

The same applies to the SOAP over JMS bindings. If the process deployment descriptor states that SOAP over JMS bindings should be used the process deployment components will care for the further steps. Optional both bindings can be generated. The following is an example of a WSDL SOAP over JMS binding description for a synchronous invocation of a sampl e business process model named TTProcDSP. As a result of an incoming message a business process instance of TTProcDSP will be created and started. The operation and the related binding have an output. Therefore an output message for the synchronous reply will be generated.

Example 6-1 SOAP over JMS Binding Example

```
<wsdl:binding name="TTProcDSPPTSoapJMSBinding" type="tns:TTProcDSPPT">
        <soap:binding style="document" transport="http://www.w3.org/2010/soapjms/"/>
        <wsdl:operation name="startTTProcDSP">
        <soap:operation
        soapAction="http://iaas.perfTest.org/wsdl/TTProcDSP/startTTProcDSP"
                        style="document"/>
                <wsdl:input>
                        <soap:body use="literal"/>
                </wsdl:input>
                <wsdl:output>
                        <soap:body use="literal"/>
                </wsdl:output>
        </wsdl:operation>
        </wsdl:binding>
```

If the original WSDL files processed by the importer components already specify bindings they will be ignored. Even with the concept of XML namespaces there is a small probability that names will not be unique. Therefore the Stuttgarter Workflow Machine uses own patterns to define unique internal binding definitions. With such specific internal naming conventions the internal processing can be eased too. For example identifiers can be derived from the process model name without extra WSDL processing or database accesses.

The former example specifies only the SOAP binding and the used transport as JMS. The binding style is document and uses a literal encoding since the business data will be encoded in XML. For both, the input and output elements of the startTTProcDSP operation will be placed in the body of the SOAP message. Until now the only difference to the SOAP over HTTP binding is the transport value that indicates that the messages will be transmitted over JMS. The actual endpoints for the web services using SOAP over JMS as transport will be JMS destinations. These are in particular JMS queues, which are specified in the next part of the generated WSDL document the service element. The next example shows an endpoint specification for SOAP over JMS providing a connection target to the web service implementing the start activity for TTProcDSP.

Example 6-2 SOAP over JMS Service Definition with Parameters and Endpoint

```
<wsdl:service name="TTProcDSP">
<wsdl:port binding="tns:TTProcDSPPTSoapJMSBinding" name="TTProcDSPPTSoapJMSBindingPort">

        <soapjms:jndiConnectionFactoryName>
                jms/swom/ConnectionFactory
        </soapjms:jndiConnectionFactoryName>

        <soapjms:replyToName>
                jms/SWoMTTProcDSPReplyQ
        </soapjms:replyToName>

        <soapjms:priority>4</soapjms:priority>



        <soap:address location="jms:jndi:SWoM_TTProcDSPReceiveQ?targetService=TTProcDSPPT;"
        />
</wsdl:port>
</wsdl:service>
```

A web service description can define multiple endpoints at different locations where the service can be invoked. Therefore a service element can include multiple ports. An individual endpoint is specified in the port element. A port element is related to a binding for example SOAP over JMS.

The example shown here is based on the last call working draft of the W3C by the SOAP-JMS Binding Working Group [SOAP2010]. It is not a standard specification yet, but the authors consider it as stable. In this example the location of the web service related to the start activity of TTProcDSP is encoded in an URI the location string value of the soap address element. The URI of type any URI defines the process specific receive queue and the target service.

Hence for JMS various properties can be defined the draft includes a schema how the properties can be defined in the concrete part of a WSDL document. For this example some of the properties are defined in elements qualified with the leading string soapjms. An alternative location for some of the properties in the WSDL document would be the URI. The example shows the replyToName, priority and connection factory property defined in soapjms qualified elements.

For example the priority property is restricted to a range of 0 to 9 and a simple type of integer by the schema. Some of the XML descriptions of the property map to the predefined properties a JMS message can include in the header.

For detailed information please refer to the working draft document [SOAP2010]. The XML schema for example can be found in appendix B and SOAP over JMS underlying protocol binding examples in appendix D.

6.1.2. Enterprise Application Implementation Generation

The actual architecture for the SOAP over JMS support for the Stuttgarter Workflow machine stipulates the generation of process model dependent Java™ enterprise component implementations. As discussed before these are message-driven beans for web services with SOAP over JMS bindings. To some extend the generated message-driven beans can be seen as the counterpart to the generated façade beans for the SOAP over HTTP bindings.

Therefore the code generation follows the same approach as the generation of the façade beans implementing receive and optional reply activities. As for the façade bean generation the resulting Java™ code is derived from code template files too. As a matter of fact parts of the generated code for both the message-driven and the façade beans are the same and were reused. Particularly these are the code parts for the SOAP message processing, SOAP reply message preparation and the constitution of the internal messages for the navigator components.

The code templates are text files. These text files are simply read into string objects. The variable parts are represented by place holders. The place holders have unique names and are marked with surrounding special tag symbols. The specific bean code dependent on the process model is added by applying string replace methods. The variable parts that are on the level of Java ™ identifiers or variable values do not need an extra code template and are added in a last step during the code generation process.

Similarly all generated message-driven beans for all process models have code sections in common. Therefore there are multiple possibilities how the code templates can be modularized and combined. Some basic approaches of partitioning the code parts into separated files can be identified. One approach would be to identify all different kinds of message-driven beans that may be generated. That leads to a straightforward approach with a code template for each kind of bean. As a result a lot of common code would reappear in all code templates.

Another approach would be to separate the common code from all variable parts and combine them depending on the deployment information in a nested manner. The approach would lead to a number of code templates dealing with specific parts of the Java implementation for example single methods, or declaration sections. As a result less code would be repeated but for example an iterative implementation to combine the code template would be more complex.

Also intermediate solutions are thinkable. The aims and meaning of the different parts the message-driven beans can encompass were discussed already in section 5.4.1. As long as the outcome is a syntactical and semantic correct Java™ implementation the actual shape of the code templates is of minor interest for the purposes of this thesis. Furthermore the generation process will be performed only once at deployment time so efficiency is not a crucial factor.

Therefore the remaining part of this section will set the focus on the generation of the meta-information files that are needed for the deployment as enterprise application on WebSphere as application server.

In contradiction to the web services provided with HTTP as transport protocol the message-driven beans do not expose a network front end. There is no Java™ servlet waiting on incoming HTTP GET or POST requests including SOAP messages that will be processed like for the façade beans. Instead there are message-driven bean instances listening on the process specific receive queue for incoming messages. As a result the message-driven beans must not be deployed as web application. A deployment as EJB module is sufficient.

Nevertheless at least two or strictly spoken three meta-information files must be provided. For WebSphere these are two XML documents. First there is the deployment descriptor with the name ejb-jar.xml. The file mainly defines namespaces and the display names of the message-driven beans. Furthermore for example the message selector of the message driven bean could be defined here instead of annotations in the generated code.

The second is a XML file with the name ibm-ejb-jar-bnd.xml. As the file name abbreviation implies the file contains bindings to application server resources. Example 6-3 shows the binding XML description for the sample process TTProcDSP.

Example 6-3 Resource Bindings for a Message-Driven Bean

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar-bnd
        xmlns="http://websphere.ibm.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://websphere.ibm.com/xml/ns/javaee
                http://websphere.ibm.com/xml/ns/javaee/ibm-ejb-jar-bnd_1_0.xsd"
version="1.0">

        <message-driven name="TTProcDSPMDB">
                <jca-adapter
                        activation-spec-binding-name="eis/SWoM_TTProcDSPRequestActSpec"
                destination-binding-name="jms/SWoMTTProcDSP_ReceiveQ"/>
                <resource-ref name="iaas.swom.mdb.TestMDB/replyCF"
                        binding-name="jms/swom/ConnectionFactory" />
                <message-destination-ref name="iaas.swom.mdb.TTProcDSPReplyQueue"
                        binding-name="jms/SWoMTTProcDSPReplyQ" />
        </message-driven>
</ejb-jar-bnd>
```

The example shows bindings for the JCA adapter with the JNDI name for the activation specification and destination queue the message-driven bean is using. Furthermore the JNDI name of the used connection factory. The last reference defines the queue that reply messages as result of synchronous requests will have as destination.

The bindings are embraced by the message-driven bean they belong to. The approach used for this thesis is to generate a message-driven bean implementation for every receive activity that the corresponding process model defines. As a result more than one binding definition could appear in the XML file associated on the basis of the message-driven bean class name.

Comparing example 6-2 and 6-3 the relationship is obvious. Therefore for example the values for the JNDI names could be obtained from the concrete part of the generated WSDL as discussed in the last subsection. Alternatively the values could be generated implicit using naming conventions that apply for all deployed process model of the Stuttgarter Workflow

Machine. If the process name is provided and unique the values can be derived by simple concatenation of relative identifiers like “jms” or “eis” with the process model name and the role indicating that this is a JNDI name.

The third file type is a nearly empty MANIFEST.MF file, which has in fact no special impact for this realization.

The generated meta-information files are text documents and do not need further processing. But the generated Java™ code files must be compiled to Java™ byte code. The class-files are created using the same implementation as for the SOAP over HTTP façade bean generation described in section 3.3.1. The Java™ compiler is called as command line tool equipped with necessary shared classes for example the internal messages used by the Stuttgarter Workflow Machine as JAR archives.

To prevent objectionable side effects and remaining artifacts all generated files are managed in a special folder structure in separated temporary system directories for each process model. The files for different transport bindings are stored and retrieved from separate directories for HTTP and JMS too if both bindings are generated in between the same deployment process.

The next subsection discusses the steps performed to constitute a deployable enterprise bean component. The result is a deployable enterprise archive stored in an EAR file.

6.1.3. Enterprise Application Assembling

A deployable enterprise archive must have a proper structure. The archive is ZIP compressed. ZIP is a compression format that is able not only to compress lossless but also to preserve the original file and directory structure starting from a root directory. For example the meta-information files will be located in a directory named META-INF. The class files for the generated message driven beans will be stored in subdirectories with a path relative to the module directory that reflects the bean package.

For the packaging three temporary subfolders relative to the temporary process deployment directory for message-driven beans are used. In one directory copies of the JAR archives are created that are needed for the compilation of the java code. The Java™ compiler will be started with this directory as class path command line argument.

That applies to the second temporary subfolder too. The directory aggregates all implementation files that constitute the enterprise archive. Therefore a special directory for the compiler output class files is provided as argument. Depending on the package of the beans an appropriate subfolder structure is created in advance and the generated java code files are placed there. After the compilation the final subfolders in the directory hierarchy contain the java and class files for each message driven bean. Furthermore a META-INF directory is created. The deployment components that are responsible for the meta-information file generation will store these files there.

The third temporary work folder assembles the files partly compressed in a final directory structure. For that purpose the content of the work folder containing the implementation files is compressed into a JAR archive and stored in the modules subfolder. Furthermore a META-INF directory with a MANIFEST.MF file is created.

In a last step the content of this folder is compressed into the final EAR archive. Afterwards the generated enterprise archive for the process model is nearly ready for deployment on the application server.

Prior to the final deployment on WebSphere the referenced JMS resources must be created otherwise the runtime bindings discussed before cannot be resolved. The next section will provide an example of the administrative tasks for the creation.

6.1.4. JMS Resources Creation

The WebSphere related actions to create JMS resources in particular the necessary administrative tasks were already discussed in section 2.3. As described there actual creation is done by running an administrative script in an appropriate profile. What kind of resources should be created and especially their names is encoded in the command line parameters for the administrative script.

The necessary information to construct the command line string is available after the binding generation for the process model. Like for the binding meta-information files the necessary names and JNDI names can be parsed out of the generated WSDL file or created implicitly.

Example 6-4 shows the Java™ code for the execution of the administrative script with parameters for the sample process model named TTProcDSP.

Example 6-4 Administrative Script for JMS Resources Execution

```
RunCommandLineTool createJMS_Resources_runner = new RunCommandLineTool();

createJMS_Resources_runner.executeCommandLineTool("C:\\IBMSDP\\runtimes\\base_v7\\profiles\\w
as70profile1\\bin\\wsadmin.bat", "-lang jython -f
C:\\SWoM_Workspace3\\SOAPoJMS_SWoM_JythonAdminScripts\\Create_SOAPoJMS_Resources.py "

        -createDestination:swombus,SWoMTTProcDSPReceiveQ " +
        -createDestination:swombus,SWoMTTProcDSPReplyQ " +

        -createQueue:swombus,SWoMTTProcDSPReceive_Q,jms/SWoMTTProcDSPRequestQ " +
        -createQueue:swombus,SWoMTTProcDSPReplyQ,jms/SWoMTTProcDSPReplyQ" +
                        "                                                       -
        createActivationSpecification:swombus,SWoMTTProcDSPActivationSpec,
                eis/SWoMTTProcDSPRActivation,jms/SWoMTTProcDSPReceiveQ");
```

The administrative script is executed with help of the wsadmin binary executable that is located in the directory of the target profile. First the script creates the service integration bus destinations. Next the corresponding physical queues are created and the JNDI names are assigned. The last step performs the creation of the activation specification that will be bound to the generated message-driven bean.

Now, after a successful generation of the enterprise application and the JMS resources, the enterprise application realizing SOAP over JMS can be deployed on the application server.

To avoid any pollution of the system all temporary files will be deleted after a successful deployment. Such a cleanup should be performed always if any of the deployment steps fails or is throwing a critical exception. As well if the deployment process has already created JMS resources they should be erased from the WebSphere application server in case of failures.

6.1.5. Deployment on the WebSphere Application Server

Once the necessary resources and the EAR file were created the generated message-driven beans are deployed as EJB module. For the actual deployment on the WebSphere application server cell no new implementation was needed. The already existing implementation creates an AppDeploymentController object via a static factory method with the EAR file and a newly created hash table as parameters.

The deployment controller performs a preparation phase in which the Web Sphere specific information is collected. This information is collectively called bindings. These bindings are used to tie the module deployment descriptors to the WebSphere runtime.

From the local EAR file a sequence of AppDeploymentTask objects are created. These objects collect the information to prepare the application for deployment. Afterwards the controller object is saved, which will write some of the collected task data in the EAR file and the rest in the hash table.

After a successful preparation a hash table with the deployment options can be obtained from the controller and passed together with the EAR file to an AppManagementProxy object, which provides a method to start the application on the WebSphere application server.

Finally after a successful deployment on WebSphere the generated message-driven beans are ready to receive messages from the created queue for the process model.

7. Summary and Outlook

As a result it can be said that it is possible to provide SOAP over JMS as an alternative transport mechanism for web services. In comparison to SOAP over HTTP the message transport self will be more reliable without additional effort. As long as the JMS messaging provider operates without failures every message will be delivered with a high probability. Because the challenges of reliable message queuing are elaborately researched a sophisticated JMS messaging provider implementation will improve the reliability of the message transmission. That applies for the communication between web services and therefore web service based business process executions too.

Integration into an advanced and sophisticated server environment like WebSphere allows for backup mechanism where one messaging engine can take over if another fails. Since the messages are stored in stable storage the messaging provider can overcome crashes of message consumers until they are replaced or recover. Therefore with additional security measurements like encryption of the messages and authentication against the provider JMS as transport mechanism can improve the robustness of the whole system.

Nevertheless it should not be omitted that JMS for message transport does not solve all problems that might occur. The responsibility and therefore the advantages of the JMS message transport ends with the message clients. A high reliable end to end message transmission would need further research. For example the actions that are necessary if the reply message to a synchronous invocation does not arrive in time were not discussed in detail.

In special cases the reliability of JMS could be seen as a disadvantage. Assuming the invocation handler ends with a timeout but the partner service is able to recover successful there might be a reply message that has no recipient any more. This is only one example for situations were messages could pollute queues because of problems at the endpoints. Solutions like dead letter queues are out of the scope of this paper but further considerations and research would surely be useful.

The W3C draft provides a specification standard recommendation that meet all demands a high performance workflow management system like the Stuttgarter Workflow Machine needs to operate. The details of the bindings can be integrated into the abstract definition languages like WSDL seamlessly. The thesis has shown that an implementation with SOAP over JMS message transport is possible. The additional opportunities JMS provides for example to define the priorities of queues are an auxiliary bonus.

Hence JMS has no constraints for the message payload there will be no problems to deal with arbitrary messages. This thesis outlined SOAP as underlying message format because it is a common standard. Nevertheless other arbitrary formats would be possible.

Furthermore the architecture and design presented in this thesis ensures scalability to some degree. The approach to equip every deployed process model with its own queues will scale even for a huge number of deployed process models as long as the hardware resources are sufficient. There are various other architectures, optimizations and designs thinkable. Due to the complexity of high performance it is not possible to cover all aspects even for SOAP over JMS in this thesis.

Nevertheless some remaining interesting questions shall be mentioned here. With the knowledge that SOAP over JMS can replace SOAP over HTTP a performance comparison between both transport mechanisms would be interesting. Benchmark tests possibly with various scenarios would complement the picture.

Furthermore the thesis does not discuss the integration of other JMS providers than the default messaging provider of IBM WebSphere. For development purposes that may be enough but for real business to business scenario challenges the aspects of communication with other application server systems than WebSphere should be evaluated. For example the use of WebSphere-MQ formally known as MQ-Series as widely used sophisticated message provider could be an approach to further enhancement of the Stuttgarter Workflow Machine.

8. References

[AM2009] Adams, Phil; Mostafia, Zina:
Develop a SOAP/JMS JAX-WS Web services application with WebSphere Application Server V7 and Rational Application Developper V7.5:
IBM delevoperWorks, 2009
http://www.ibm.com/developerworks/websphere/library/tutorials/0903_adams/0903_adams-pdf.pdf
Verified at: 09.03.2011

[BPEL2007] OASIS: *Web Services Business Process Execution Language Version 2.0*:
OASIS Standard: 11 April 2007
http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf
Verified at: 14.10.2011

[CT2002] Chappell, David A.; Jewell, Tyler:
Java WebServices
1005 Gravenstein Highway North, Sebastopol, CA: O´Reilly & Associates, Inc, 2002

[EJB2006][1] DeMichiel, Linda; Keith Michael: EJB 3.0 Expert Group:
EJB 3.0 Simplified API:
JSR220: Enterprise JavaBeans™: Final Release:
Sun Microsystems, 2006
http://www.cs.bilkent.edu.tr/~ccelik/cs412/ejb-3_0-fr-spec-simplified.pdf
Verified at: 14.10.2011

[EJB2006][2] DeMichiel, Linda; Keith Michael: EJB 3.0 Expert Group
EJB Core Contracts and Requirements:
JSR220: Enterprise JavaBeans™: Final Release:
Sun Microsystems, 2006
http://www.cs.bilkent.edu.tr/~ccelik/cs412/ejb-3_0-fr-spec-ejbcore.pdf
Verified at: 14.10.2011

[IBM2009][1] Albertoni, Fabio; Blunt Leonard; Conolly, Michael; Kwiatkowski, Stefan; Sadtler, Carla; Shanmugaratnam, Thayaparan; Sjostrand, Hendrik; Tanikawa, Saori; Ticknor, Margaret; Veser; Joerg-Ulrich:
WebSphere Application Server V7 Administration and Configuration Guide:
IBM International Technical Support Organization, March 2010
http://www.redbooks.ibm.com/abstracts/sg247615.html
Verified at: 28.08.2011

[IBM2009][2] Sadtler, Carla; Blunt, Leonard; Suram M Neela:
WebSphere Application Server V7 Messaging Administration Guide:
IBM International Technical Support Organization, July 2009
http://www.redbooks.ibm.com/abstracts/sg247770.html
. Verified at: 27.08.2011

[IBM2009][3] Cui, Henry; Lara, Raymond Josef Ecvard A.; Makar, Rosaline; Moelholm, Nicky; Rodrigues, Felipe Bitela:
IBM WebSphere Application Server V7.0 Web Services Guide:
IBM International Technical Support Organization, July 2009
Verified at: 27.08.2011

[JMS2002][1] Hapner, Mark; Burridge, Rhich; Sharma, Rahul; Fialli, Joseph; Haase, Kim
JAVA™ Message Sevice:
API Tutorial and Reference: Messaging for the J2EE™ Platform:
Sun Microsystems, Inc, 2002

[JMS2002][2] Hapner Mark; Burridge, Rhich; Sharma, Rahul; Fialli, Joseph; Stout, Kate:
Java Messaging Service:
SUN Microsystems, Version 1.1 April 12, 2002
http://www.oracle.com/technetwork/java/docs-136352.html
Verified at: 27.08.2011

[LR2000] Leymann, Frank; Roller, H. Dieter:
Production Workflow:
Concepts and Techniques
Jersey: Prentice Hall 2000

[SOAP 2000] Barton, John J.; Thatte, Satish; Nielsen, Henrik Frystyk; W3C:
SOAP Messages with Attachments:
W3C Note 11 September 2000
http://www.w3.org/TR/SOAP-attachments
Verified at: 08.10.2011

[SOAP2007] Mitra, Nilo; Lafon, Ericsson Yves; W3C:
SOAP Version 1.2 Part 0: Primer (Second Edition):
W3C Recommendation 27 April 2007
http://www.w3.org/TR/soap12-part0
Verified at: 08.10.2011

[SOAP2010] Phil, Adams; Easton, Peter; Johnson, Eric; Merric, Roland; Phillips, Mark; W3C:
SOAP over Java Message Service 1.0:
W3C Working Draft 26 October 2010
http://www.w3.org/TR/2010/WD-soapjms-20101026
Verified at: 18.04.2011

[WSA2006] Gudgin, Martin; Hadley,Marc; Rogers, Toni:
Web Services Addressing 1.0 –Core:
W3C Recommendation 9 May 2006
http://www.w3.org/TR/ws-addr-core/
Verified at 12.10.2011

Declaration:

This thesis is the result of my own work. Material from the published or unpublished work of others, which is referred to in the thesis, is explicit referenced and credited to the authors.

The thesis has not been submitted for the award of any other degree or diploma in any other tertiary institution.

(Place, Date) (Tobias Rohm)