The JMS message processing using the Java Connector Architecture 1.5

BEOM-SU SEO[0], SEUNG-WOOG JUNG, SUNG-HOON KIM, JOONG-BAE KIM
Internet Computing Department
Electronics and Telecommunications Research Institute
161 Gajeong-dong, Yuseong-gu, Deajeon, 305-350, SOUTH KOREA

Abstract: - The JCA(Java Connector Architecture) 1.5 is the specification for integrating a J2EE application server with legacy systems such as DBMS, ERP, JMS, etc. Speaking of a message system, the EJB 2.0 recommending JCA 1.0 supported only the JMS system, but in the EJB 2.1 with JCA 1.5 many kinds of message systems are able to be integrated by virtue of the work management contract and the message inflow contract of the JCA 1.5. An application server performs special works on behalf of a resource adapter like communicating a network endpoint or monitoring a socket port to intercept an incoming message. In this paper, I briefly introduce what the JCA 1.5 is and discuss how to process JMS messages using the JCA 1.5 with the implementation of the work management contracts and the message inflow contract

Key-Words: - JCA, EJB, JMS, J2EE

1 Introduction

The JCA(Java Connector Architecture)[4,5] provides the formal method to integrate J2EE[1] compatible application servers with EIS(Enterprise Information System)s such as DBMS, ERP, message service, etc. The JCA is a direct method using API. As a result, it is not for inter-enterprise integration but for intra-legacy integration which should be merged in an enterprise for instance.

Basically, the JCA 1.0[4] provides the API sets for a J2EE application server to communicate with legacy systems using three kinds of contracts. First, it provides the APIs for the connection management contract to determine how to get the connection from data source. Secondly, it defines the security management contract to access a connection in secure manner along with the connection management contract. Finally it presents the transaction management contract to guarantee the ACID properties of a transaction using JTA[6].

The next version, JCA 1.5[5], had been proposed as final draft so that it was released in January, 2004. In the JCA 1.5, many new aspects are added such as the resource adapter(the object that takes the responsibility for legacy systems) life cycle management contract, the work management contract with which an application server can pool threads and assign a thread for executing specific works a resource adapter submits. And the message inflow contract is proposed to process incoming messages from external message provider. The EJB 2.0[2] limited a message provider to the JMS(Java Message Service)[7] for a message driven bean to be a consumer only for the JMS. With this contract, the EJB 2.1[3] provides message provider pluggability that means any types of message providers(mail, socket, etc.) can be plugged in an application server in a same manner.

The JCA 1.5 categorizes these contracts into outbound and inbound communication contracts. The outbound communication contracts include the connection, the security and the transaction management contract. On the other hand, if a message provider sends a message to a specific JMS queue or topic, a listener accepts the message and passes it to a message bean container located in an application server. The inbound communication contracts treat this process and define the required environments. The resource manager of an application server creates a resource adapter and initiates the inbound/outbound communication facilities with the BootstrapConext object which includes the WorkManager reference for the work management contract. An application server manages the resource adapter's lifecycle calling start(BootstarpContext) and stop() methods.

A resource adapter can try to monitor a specific network port or control an application's resources using a thread. But it could be very dangerous for an application server to abdicate a thread control to a resource adapter actually provided by external resource adapter vendors. And it could not be an efficient and effective way to manage system resources. So, if a resource adapter wants to do a job

using a thread, it should create a work to access the system resources and submit the work to the WorkManager in an application server. Then the application server chooses a thread for executing the work to let the resource adapter avoid creating new thread. This is the main purpose of the work management contract.

Using this work mechanism, the message inflow contract can be implemented. The resource adapter creates the work for eventually listening specific a topic or queue described in the resource adapter deployment descriptor. The application server chooses a thread to process the work. If a message client sends a message, the work receives the message and delegates it to the proper message container which selects a message bean instance and calls the bean's method with the received message.

There is another important contract I've not mentioned; the transaction inflow contract. According to this contract, a resource adapter starts a transaction and propagates it with a message into an application server. The application server acts like one of participant of 2PC protocol. The resource adapter has to provide a XAResource and be able to convert a transaction context into that of the application server. But in this paper, I don't talk about the transaction inflow contract in this paper.

I discuss the implementation of the work management contract and the message inflow contract to process a JMS message in the following sections.

2. The lifecycle management contract

This contract defines the interface to create and remove a resource adapter. Using this contract, an application server can manage the lifecycle of a resource adapter. An application server creates an resource adapter instance and calls its start(BootstrapContext) and stop() method to control it. Calling start(BootstrapContext) method, it delivers an BootstrapContext object for the resource adapter to access a WorkManager to submit specific jobs to an application server. An resource adapter may initialize its own local objects or variables in start(BootstrapContext) method and finalize these objects in stop() method

.

3. The implementation of the work management contract

A resource adapter can use threads to wait data incoming through a specific network endpoint or communicate other network peer. The JCA 1.5 provides the work management contract with which an application server can allocate threads for a resource adapter and manage them.

There are three types of methods related with a work submission depending on whether a resource adapter waits or not.

(1) doWork() method : A resource adapter has to wait until an application server completes the work.
(2) startWork() method : A resource adapter waits just until an application starts the work. The start point means that the application server selects a thread and assigns the work to it but doesn't call the work's run() method. In a run() method, the work can do its own job.
(3) scheduleWork() method : A resource adapter waits just unit an application server accepts the work. The accept point means that an application server selects a thread but doesn't assign the work to the thread. An application server just selects the work from the work queue for processing.

Fig 1 shows the detail process about the creation of the work manager in our implementation. The work manager creates thread pools to store the Work-Threads. It also creates the WorkDispatcher and the WorkQueue to keep a submitted work. The WorkDispatcher dispatches a work in the WorkQueue monitoring the queue.

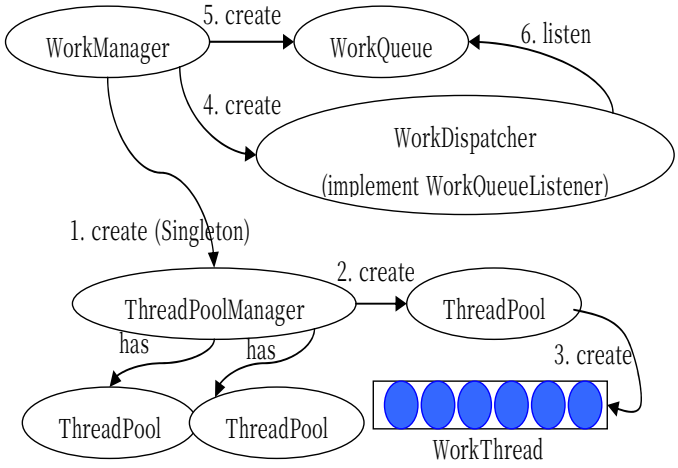

Fig 1. The initialization of the WorkManager

After the resource adapter submits a work using WorkManager's methods, the work manager wraps it with an InternalWork and then puts the InternalWork into the WorkQueue. In each object, local variables of

an InternalWork are used to keep the status and the time for submission, rejection, acceptance and completion. The WorkThread changes the status and the WorkManager uses the changed status to suspend or resume the main thread. To execute the work,

(1) A resource adapter submits a Work to the WorkManager in a BootstrapContext object.
(2) The WorkManager wraps the Work with an InternalWork.
(3) The WorkManager puts the InternalWork into the WorkQueue.
(4) The WorkQueue sets the acceptance time of the InternalWork and informs the acceptance to the resource adapter using WorkListener provided with a submission method parameter.
(5) The WorkQueue notifies the arrival of the new work to the WorkDispatcher which monitors the WorkQueue.
(6) The WorkDispatcher gets a new WorkThread from a ThreadPool.
(7) The WorkDispatcher gets the submitted InternalWork and assign it to the WorkThread. Fig 2 describes this process

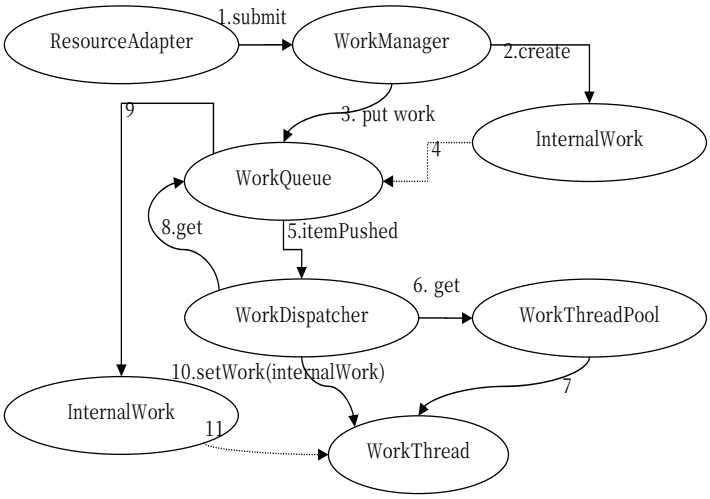

Fig 2. Work submission and thread allocation

(8) Along with three types of methods, the WorkManager is blocked and waits for other thread's notify() method on the IntrenalWork. If an InternalWork isn't allocated, WorkThread is blocked.
(9) After the InternalWork is allocated, the run() method of the WorkThread is started to execute.
(10) In the run() method, the WorkThread may send a start notification using WorkListener to the resource adapter. And it calculates the submission time and start time to determine whether the work started within a given time the resource adapter specified as a submission method parameter or not.
(11) The WorkThread wakes up the WorkManager blocked in the submission methods on the InternalWork.wait() method to resume the main thread calling the InternalWork.notify() method.
(12) The WorkThread calls the run() method of the original Work to execute a work the resource adapter wants to do.
(13) After the work completion, the WorkThread may send the completion notification using WorkListener to the resource adapter with a WorkEvent object.
(14) The WorkThread nullifies the internalWork and pushes itself into the ThreadPool.
(15) The ThreadPool waits another request.

4. The JMS message processing

The inbound resource adapter is responsible for receiving a message, checking its type and doing other jobs like transaction process or delegating it to the container. A message resource adapter submits the work which monitors the specific message destination(JMS topic or queue, socket port) and message clients send a message to the destination. Then the work receives it and processes it using a special object; so called MessageEndpoint. Although a message driven bean finally processes a message, a MessageEndpoint intercepts a message to do transaction related jobs or user defined jobs. A MessageEndpoint object is an application server side proxy equivalent to a message driven bean existed in a message driven container. A resource adapter has to provide an environment object; i.e. ActivationSpec. An application server creates a message driven container and activates a message endpoint using an ActivationSpec to verify configured information provided in the deployment descriptor of a message driven container. After each container is created and applications are deployed, the resource manager located in an application server creates a MessageEndpointFactoty which is responsible for creating a message endpoint(i.e. a proxy) object and an ActivationSpec object. The resource manager demands the resource adapter to activate an endpoint. Fig 3 depicts this process. In this paper, even though I explain the implementation of the message inflow contract in case of the JMS message, but similar structure could be adapted in other message types. A

message container doesn't have any specific information but only does take from an Activation-Spec and a deployment descriptor including a message type itself. Because even a message type is decided dynamically in the execution time, any types of message could be possible in EJB 2.1.

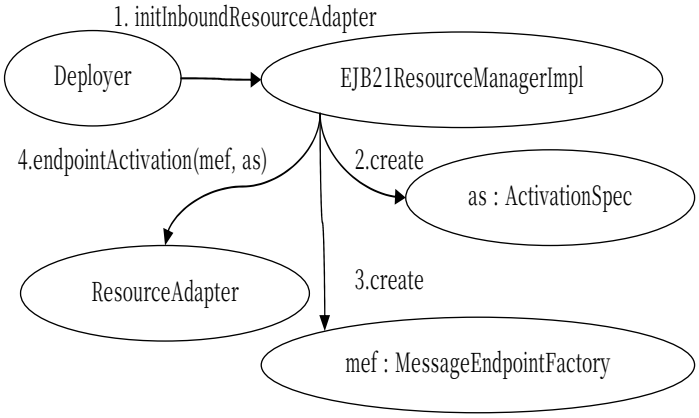

Fig 3. Initialization of the Inbound Resource Adapter

4.1 JMS MessageEndpointActivationWork submission

After receiving an activation request in fig 3, a resource adapter creates the work to listen a message destination and submits it to the WorkManager of an application server. In our implementation, the JMSEndpointActivationWork takes charge of receiving a message. If developers try to adapt other message type, they have to write their own work to handle other message type.

4.2 JMS message reception

The JMSEndpointActivationWork internally creates javax.jms.MessageListener implementation object and registers it to a TopicSubscriber or QueueReceiver object monitoring a topic or queue. At this point, an ActivationSpec object and the deployment descriptor xml provide lots of information for a MessageListener object to receive a message such as JMS destination name, message selector, acknowledge mode, durability and transacted value, etc. Fig 4 describes the creation of a message listener that a TopicSubscriber or QueueReceiver has. If TopicSubscriber or QueueReceiver receives a message, it calls the listener's onMessage() method to delegate the message

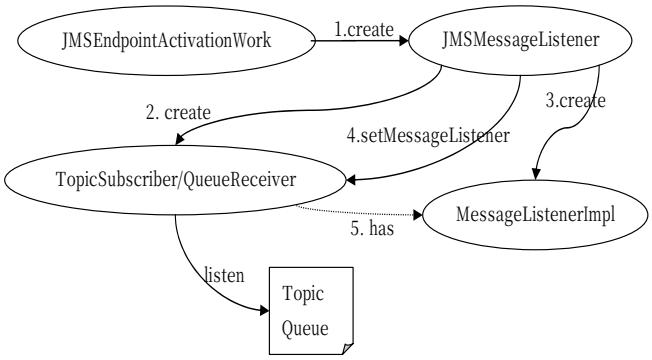

Fig 4. JMS message listening

4.3 JMS message processing

If the MessageListenerImpl object in fig 4 receives a message, it calls the JMSEndpointActivation-Work.invok() method with the message. Then, the JMSEndpointActivationWork requests a message endpoint object calling the createEndpoint() method of MessageEndpointFactory set by the resource adapter in the initialization step. The message end-point object is a proxy object which could be cast as a message listener type that the resource adapter has to support (javax.jms.MessageListener in this implementation) and javax.resource.spi.endpoint.MessageEnd-point interface. In case of a container managed transaction, a message could be processed within a transaction to execute the whole processing as a part of the transaction. To participate in the transaction, the resource adapter should provide a XAResource object that the message endpoint proxy would enlist within the transaction. In a transaction commit or rollback phase, the XAResource would be used to commit or rollback the transaction using 2PC protocol. Speaking of intervention of a message delivery, the JCA 1.5 suggests two options shown fig 5 and fig 6.

In option A, an application server takes control over a transaction so that there is no room for a resource adapter to intervene in the middle of message delegation. In option B, on the other hand, a resource adapter can intercept a message before delegation to do some jobs calling beforeDelivery() method and finalize the delivery calling afterDelivery() method of the MessageEndpoint interface. It means that a resource adapter can control the transaction scope, which gives more flexibility to a resource adapter and an application server. The proxy must have listener's methods(in case of JMS, the listener method is onMessage() defined in javax.jms.MessageListener

interface) for option A, and MessageEndpoint methods (beforeDelivery() and afterDelivery() methods) for option B. In fact, because a message listener type written in the deploy descriptor xml is determined in an execution time, to support option A and B, an application server should provide a static code implementing a special listener and MessageEndpoint interface in compile time or create a dynamic code(i.e. dynamic proxy) in execution time.

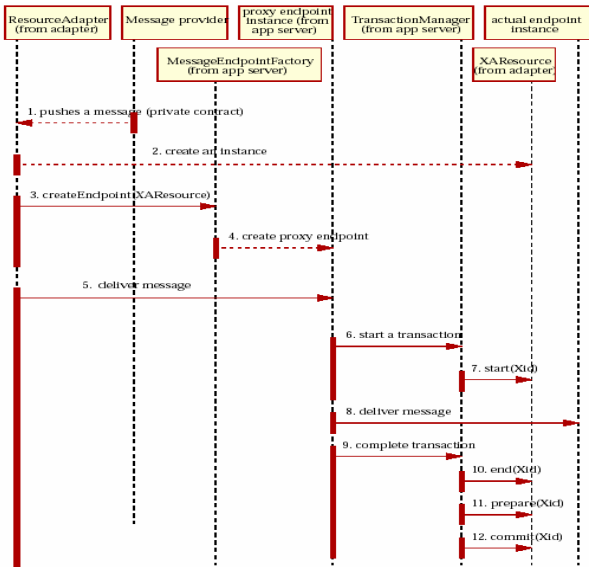

Fig 5. Transacted message delivery : Option A

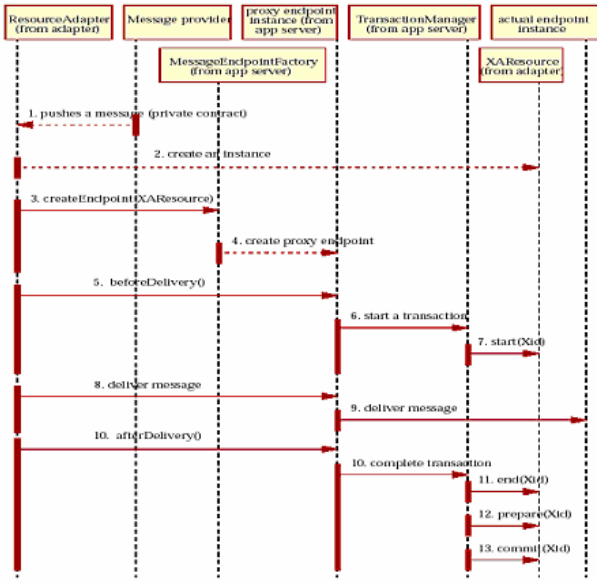

Fig 6. Transacted message delivery : Option B

For the purpose of a dynamic proxy, Java provides some reflection methods like java.lang.reflect.Proxy and java.lang.reflect.InvocatonHandler. A proxy object is created using Proxy.newInstance(ClassLoader, Class[], InvocationHandler). The Class[] array contains MessageEndpoint.class and javax.jms.MessageListener.class in this implementation. The latter may be changed depending on a message provider type. Every method call of MessageEndpoint and MessageListener interface over the created proxy is delivered to InvocationHandler.invoke(proxy, method, args) method. In option B, beforeDelivery() and afterDelivery() of a MessageEndpoint could be called compared with the method parameter. In case of MessageListener method, the invocation handler delegates the call to the JMSInvocationHandler that delivers a message to a message driven container. The JMSEndpointActivationWork requests a proxy object to the MessageEndpointFactory and casts it with MessageEndpoint or MessageListener depending on the proper option. Fig 7 shows a part of the JMSEndpointActivationWork how to cast and call methods depending on the options

```
public invoke(Method method, Object[] args) {
…
MessageListener ep = (MessageListener)
msgEpFactory.
            createEndpoint(xaResource);
if (optionA)
      ep.onMessage( (Message) args[0]);
else { // option B
  ((MessageEndpoint)ep).beforeDelivery(method);
   ep.onMessage( (Message) args[0]);
   ((MessageEndpoint)ep).afterDelivery();
…
}
```

Fig 7. JMSEndpointAtcivationWork

Fig 8 describes the message delivery. The MessageEndpointInvoker is the implementation object of InvocationHandler interface. The MessageEndpointFactory's createEndpoint() method creates a proxy instance with a message listener class and a MessageEndpoint class.

(1) A client sends a message to a topic or queue.
(2) A message listener receives the message
(3) The listener delivers it to a JMSEndpointActivationWork calling invoke() method.
(4) The JMSEndpointActivationWork requests a message endpoint proxy to a MessageEndpointFactory.
(5) The MessageEndpointFactory creates a MessageEndpoint proxy with MessageLis-

tener.class and MessageEndpoint.class as parameters of Proxy's newInstance() method. An ActivationSpec from a xml descriptor provides a listener interface for the proxy so that a dynamic proxy creation would be possible.

(6) Actually, the MesssageEndpointFactory creates a MessageEndpointInvoker as a invocation handler to intercept every method of the listener and the MessageEndpoint interface on its invoke() method.

(7) The JMSEndpointActivationWork calls beforeDeliver()(afterDelivery()) or listener's method(onMessage() in JMS) depending on the option.

(8) The MessageEndpointInvoker, an actual invocation handler, delivers the message to a JMSInvocationHandler which has a reference about a message driven container. In fact, in the implementation of the EJB 2.0, the JMSInvocationHandler receives directly a message from a message listener and delegates it to a message driven container.

(9) The JMSInvocationHandler calls a message driven container and the message driven container selects a message bean from a pool to process the incoming message.

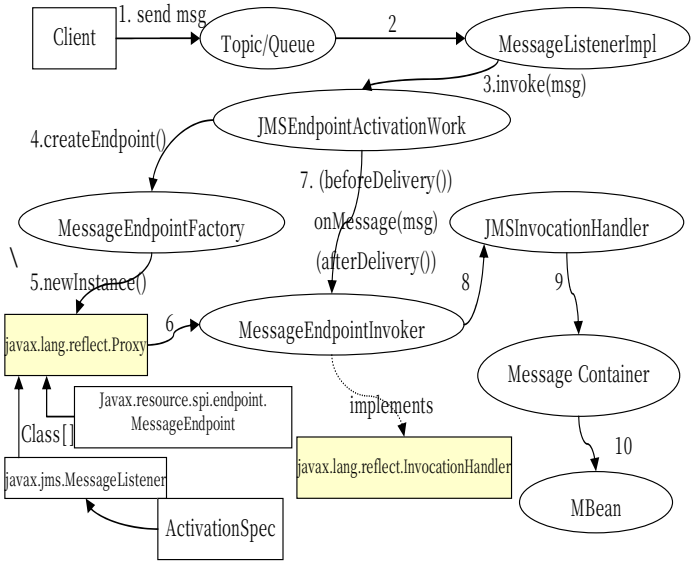

Fig 8. Message Delivery

5. Conclusion

The JCA1.5 is resource integration contracts within the EJB 2.1 proposed by SUN, which is supported by many major companies such as IBM, Oracle, BEA, HP, Fujitsu, SAP and so on. It re-categorizes the three contracts in JCA 1.0(connection, security, transaction management contract) as outbound communication contracts and defines inbound communication contracts with message inflow contract, transaction inflow contracts. To give a thread execution to a resource adapter and control a resource adapter, it provides the work management and the lifecycle contract.

If a resource adapter tries to wait a message from a destination, JCA 1.5 enables an application server to allocate a thread for a work on behalf of a resource adapter and execute the work to receive a message. In this paper, I introduced the implementation of the work management contract and the message inflow contract focused on JMS. I didn't discuss the transaction inflow contract which will be implemented in the near future.

References :

[1] JavaTM 2 Platform Enterprise Edition Specification, v1.4 Public Draft, SUN Microsystems, July 15, 2002

[2] Linda G. Demichiel, "Enterprise Javabeans Specification, Version 2.0," Sun Microsystems, 2001

[3] Linda G. Demichiel, "Enterprise Javabeans Specification, Version 2.1," Proposed Final Draft, Sun Microsystems, 2002

[4] J2EE Connector Architecture, Final Release version 1.0, Sun Microsystems, 2001

[5] J2EE Connector Architecture, version 1.5 Proposed Final Draft 2, Sun Microsystems, 2002

[6] Susan Cheung, "Java Transaction API (JTA) Version 1.0.1," Sun Microsystems, 1999

[7] Java Message Service Specification, version 1.1, April 12, 2002