Spring Java Message Service

**JAVA MESSAGE SERVICE – 2ND EDITION**

Chapter4. Point-to-Point Messaging

**Overview**

Point to point Messaging is used when you need to send a message to only one message consumer. Even though multiple consumers may be listening on the queue for the same message, only one of those consumer threads will receive the message. This is different from the publish –and-subscribe model in Chapter5, where a message is broadcast to (and consumed by) multiple consumers.

**When to use Point-to-point**

Point-to-point is used when you want one receive to process any given message once and only once. P2P guarantees that only one consumer will process a given message. P2P model also provides a QueueBrowser that allows the JMS client to take a snapshot of the queue to see messages waiting to be consumed. Pub/sub does not include a browsing feature.

Another case is when you need synchronous communication between components, but those components are written in different programming languages or implement in different technology platforms(E.G., J2EE and .NET).

To allow of high level through put

**Message Correlation**

In general, whenever using the request/reply model, you must make sure the response you are receiving is associated with the original message you sent. Message correlation is the technique used to ensure that you receive the right message. The most popular method for correlating messages is leveraging the JMSCorrelationID message header property in conjunction with the JMSMessageID header property. The JMSCorrelationID property contains a unique String value that is known by both the sender and receiver. The JMSMessageID is typically used, since it is unique and is available to the sender and receiver.

**Load Balancing Using Multiple Receivers**

When multiple receivers are attached to a queue, each message in the queue is delivered to one receiver. The absolute order of messages cannot be guaranteed, since one receiver may process messages faster than another. From the receiver’s perspective, the messages it consumes should be in relative order; messages delivered to the queue earlier are consumed first. However, if a message needs to be redelivered due to an acknowledgement failure, it is possible that it could be delivered to another receiver. The other receiver may have already processed more recently delivered messages, which would place the redelivered message out of the original order.

**Examing a Queue Using Queue Browser**

**Chapter 7 Guaranteed Messaging and Transactions**

JMSProvieder – The JMS server

JMSCLIENT (CLIENT RUNTIME) – the client that publishes or consumes messages from the provider.

**Message Autonomy**

Message is self-contained autonomous, which can be sent and resent multiple times

**Store-and-Forward Messaging:** When messages are marked persistent, it is the responsibility of the JMS provider to utilize a store-and-forward mechanism to fulfill its contract with the sender. The storage mechanism is used for persisting messages to disk (or some other reliable medium) in order to ensure that the message can be recovered in the event of a provider failure or a failure of the consuming client. Message storage may be centralized or Decentralized or both.

**Message Acknowledgement and Failure Conditions: -** JMS specifies a number of acknowledgement modes. These acknowledgements are a key part of guaranteed messaging. A message acknowledgement is part of the protocol that is established between the client runtime portion of the JMS provider and the server. Servers acknowledge the receipt of messages from JMS producers, and JMS consumers acknowledge the receipt of messages from servers. The acknowledgement protocol allows the JMS provider to monitor the progress of a message so that it knows whether the message was successfully produced and consumes. With this information, the JMS provider can manage the distribution of messages and guarantee their delivery.

Message acknowledgements: The acknowledgement mode is set on a JMS provider when a Session is created; Session session = Connect.createQueuSession(false, Session.ClIENT\_ACKNOWLEDGE);

Auto\_acknowlege: - The acknowledgement is always the last thing to happen implicitly after the onMessage() handler returns.

* From the perspective of the Message Producer: - under the covers, the TopicPublisher().publish() or QueueSender.send() methods are synchronous. These methods are responsible for sending the message and blocking until an acknowledgement is received from the message server. Once an acknowledgement has been received, the thread of execution resumes and the method returns; processing continues as normal. The underlying acknowledgement is not visible to the client programming model. If a failure condition occurs during this operation, an excecption is thrown and the message is considered underlivered.
* From the perspective of the Message Server: - The acknowledgement sent to the producer (sender) from the server means that the server has received the message and has accepted responsibility for delivering it. From the JMS server’s perspective, the acknowledgement sent to the producer is not tied directly to the delivery of the message. They are logically two separate steps. For persistent messages, the server writes the message out to disk (the store part of store-and –forward), then acknowledges to the producer that the message was received. For the nonpersistent messages, this means the server may acknowledge the sender as soon as it has received the message and has the message in momory. If there are no subscribers for the message’s topic, the message may be discarded depending on the vendor.
  + In a publish-and –subscribe model, the message server delivers a copy of a message to each of the subscribers. For durable subscribers, the message server does not consider a message fully delivered until it receives an acknowledgement from all of the message’s intended receipeients. It knows on a per-consumer basis which clients have received each message and which have not.
  + Once the message server has delivered the message to all of its known subscribers and has received acknowledgements from each of them, the message is removed from its persistent store.
  + If the subscriptions are durable and the subscribers are not currently connected, then the message will be held by the message server until either the subscriber becomes available or the message expires. This is true even for nonpersistent messages. If a nonpersistent message is intended for a disconnected durable subscriber, the message server saves the message to disk as though it were a persistent message. In this case, the difference between persistent and nonpersistent messages is subtle, but very important. For nonpersistent messages, there may be a window of time after the message server has acknowledged the message to the sender and before it has had a chance to write the message out to disk on behalf of the disconnected durable subscribers. If the JMS provider fails during this window of time, the message may be lost
  + With persistent messages, a provider may fail and recover gracefully, as illustrated in Figures [**Figure 7-4**](javascript:moveTo('ch07-93264');) and [**Figure 7-5**](javascript:moveTo('ch07-73872');). Since the messages are held in persistent storage, they are not lost, and will be delivered to consumers when the provider starts up again. If the messages are sent using a p2p queue, they are guaranteed to be delivered. If the messages were sent via publish-and-subscribe, they are guaranteed to be delivered only if the consumers’ subscriptions are durable. The delivery behavior for nondurable subscribers may vary from vendor to vendor.
* The Message Consumer’s perspective: - If the session is in AUTO\_ACKNOWLEDGE mode, the JMS provider’s client runtime must automatically send an acknowledgment to the server as each consumer gets the message. If the server doesn’t receive this acknowledgement, it considers the message undelivered and may attempt redelivery.
* For P2P queues, messages are marked by the producer as either persistent or non-persistent . Persistent messages are written to disk and subject to the same acknowledgement rules, failure conditions, and recovery as persistent messages in the PnS model.
  + From the receiver’s perspective, the rules are somewhat simpler, since only one consumer can receive a particular instance of a message. A message stays in a queue until it is either delivered to a consumer or it expires. This is analogous to a durable subscriber in that a receiver can be disconnected while the message is being produced without losing the message**. If the messages are non-persistent, they are not guaranteed to survive a provider failure.**

**Message redelivery**

The message may be lost if the provider fails while delivering a message to a consumer with a nondurable subscription. If a durable subscriber receives a message, and a failure occurs before the acknowledgement is returned to the provider, then the JMS provider considers the message undelivered and will attempt to redeliver it. In this case, the ocne-and-only once requirement is in doubt. The consumer may receive the message again, because when delivery is guaranteed, it’s better to risk delivering a message twice than to risk losing the message entirely, A redelivered message will have the JMSRedelivered flag set. A client application can check this flag by calling the getJMSRedelivered method on the Mesasage object. Only the most recent message received is subject to this ambiguity.

To guard against duplicate messages while in AUTO\_ACKNOWLEDGE mode, an application must check whether a redelivered message was already processed. One common technique for checking is to use a database table that is keyed on the JMSMessageId header. A JMSMessageID is unique for all messages and is intended for historical monitoring of messages in a repository. The JMSMEssageID is therefore guaranteed to retain its uniqueness across provider failures. An alternative approach would be to use the CLIENT\_ACKNOWLEGE mode or to use a transacted message.

**DUPS\_OK\_ACKNOWLEDGE**

Specifies that it’s okay to send duplicate messages.

**CLIENT\_ACKNOWLEDGE**

The client receiving the messages can get finer-grained control over the delivery of guaranteed messages by specifying the CLIENT\_ACKNOWLEDGE mode on the consuming session.

* The key to this is the acknowledge () method on the Message object.
* The acknowledge method informs the JMS provider that the message has been successfully received by the consumer. This method throws an exception to the client if a provider failure occurs during the acknowledgement process. The provider failure results in the message being retained by the the JMS server for redelivery. Therefore, the exception handling code should undo the results of any partially processed business logic in preparation for receiving the message again, or it should log the message as processed so that the redelivered message can be ignored. The acknowledge() method should only be used with the LCIENT\_ACKNOWLEDGE mode.

**Transacted Messages**

JMS transactions follow the convention of separating the send operations from the receive operations. Figure 7-8 shows a transactional send, in which a group of messages are guaranteed to get to the message server, or none of them will. From the sender’s perspective, the messages are cached by the JMS provider until a commit () is issued. If a failure occurs, or a rollback () is issued, the messages are discarded. Messages delivered to the message server in a transaction are not forwarded to the consumers until the producer commits the transaction.

The JMS provider will not start delivery of the messages to its consumers until the producer has issued a commit () on the session, even though it has received all of the messages from the sender. The scope of a JMS transaction can include any number of messages. Although similar in concept, the session commit () is not the same as a Java Transaction API (JTA) transaction commit (). The session transaction is managed by the JMS provider, not JTA.

It should be no surprise that JMS also supports transactional receives, in which a group of transacted messages are received by the consumer on an all-or-nothing basis. From the transacted receiver’s perspective, the messages are delivered to it as expeditiously as possible, yet they are held by the JMS provider until the receiver issues a commit () on the session object. If a failure occurs or a rollback () is issued, then the provider will attempt to redeliver the messages, in which case the messages will have the redelivered flag set.

**Creating and Using a JMS Transaction**

// Create a transacted TopicSession

TopicSession session =

connect.createTopicSession(true, Session.AUTO\_ACKNOWLEDGE);

// Create a transacted QueueSession

QueueSession =

connect.createQueueSession(true, Session.AUTO\_ACKNOWLEDGE);

When a session is transacted, all messages sent or received using that session is automatically grouped in a transaction. The transaction remains open until either a session. Rollback () or a session. Commit() happens, at which point a new transaction is started. An additional Session method, getTransacted(), returns true or false indicating whether or not the current session is transactional. – This is called “transaction chaining’, which means the end of one transaction automatically starts another.

**Lost Connections**

**The ExceptionListener**

**SPRING AND JMS – 1ST Edition of Java Messaging Services Book**

* Shields the user from the differences between JMS 1.0.2 and 1.1 APIs
* The JmsTemplate class is used for message production and synchronous message reception. For asynchronous reception similar to J2EE’s message-driven bean style, Spring provides a number of message listener containers that are used to create Message-Driven POJOs(MDPs).
* The package org.springframework.jms.core provides the core functionality for using JMS.
* The package org.springframework.jms.connection provides an implementation of the connectionFactory suitable for use in standalone applications. It also contains an implementation of spring’s PlatformTransactionManager for JMS called JmsTransactionManager. This allows for seamless integration of JMS as a transactional resource into spring’s transaction management mechanisms.

**Message Listener Containers**

One of the most common uses of JMS messages in the EJB world is to drive message-driven beans (BDMs). Spring offers a solution to create message-driven POJOs (BDPs) in way that does not tie a user to an EJB container. There are 3 standard JMS message listener containers packaged with Spring, each with its specialized feature set.

1. SimpleMessageListenerContainer
2. DefaultMessageListenerContainer
3. ServerSessionMessageListenerContainer

**Receiving a Message**

1. **Synchronous Reception:** - While JMS is typically associated with asynchronous processing, it is possible to consume message synchronously. The overloaded receive (..) methods provide this functionality. During a synchronous receive, the calling thread blocks until a message becomes available. This can be a dangerous operation since the calling thread can potentially be blocked indefinitely. The property receiveTimeout specifies how long the receiver waits before giving up waiting for a message.
2. **Asynchronous Reception** : – Message-Driven POJOs: - In a fashion similar to a Message-Driven Bean(MDB) in the EJB world, the Message-Driven POJO (MDP) acts as a receiver for JMS messages. The one restriction on an MDP is that it must implement the **javax.jms.MessageListener** interface. Be aware that in the case where your POJO will be receiving messages on multiple threads, it is important to ensure that your implementation is thread-safe.
3. **The SessionAwareMessageListener interface:** - a Spring-specific interface that provides a similar contract the JMS MessageListener interface, but also provides the message handling method with access to the JMS Session from which the Message was received.
   1. **Package org.**springframework.jms.listener;
   2. **Public interface SessionAwareMessageListener {**
      1. **Void onMessage(Message message, Session session) throws JMSException;**
   3. **}**

You can choose to have your MDPs implement this interface (in preference to the standard JMS MessageListener interface) if you want your MDPs to be able to respond to any received messages(using the Session supplied in the onMessage(Message, Session) method. All o fth emessage listener container implementations that ship with Spring have support for MDPs that implement either the Messagelistener or SessionAwareMessagelistener interface. Classes that implement the SessionAwareMessageListener come with the caveat that they are then tied to Spring through the interface. The choice of whetehr or not to use it is left entirely up to you as an applicaton developer or architect.

Note that the ‘onMessagte(..)’ method of the SessionAwareMessageListener interface throws JMSException. In contrast to the standard JMS MessageListener interface, when using the SessionAwareMessageListener interface, it is the responsibility of the client code to handle any exceptions thrown.

**Processing messages within Transactions**

Messages delivered to the message server in a transaction are not forward to the consumers until the producer commits the transaction.