Middleware-Konzepte

Java Message Service (JMS)

Dr. Gero Mühl

Kommunikations- und Betriebssysteme
Fakultät für Elektrotechnik und Informatik
Technische Universität Berlin
Agenda

> Introduction
> JMS Programming Model
  > Point-to-Point Model
  > Publish/Subscribe Model
> Messages
> Transacted Sessions
> Distributed Transactions
> Message-driven Beans
> Open Issues
Java Message Service (JMS)

- Java Message Service (current version 1.1)
- Enables reliable asynchronous messaging
- Supports two styles of messaging (aka. messaging domains)
  - **Point-to-Point**
    - Message delivered to a single queue
  - **Publish/Subscribe**
    - Message delivered to potential many subscribers
- Vendor-independent standard
- Part of **J2EE (Java 2 Platform, Enterprise Edition)**
Products

> Specification defines *interfaces* only
  ⇒ Many distinct (incompatible) implementations

> JMS licensed by a lot of vendors
  > BEA          WebLogic
  > IBM          WebSphere MQ
  > Sun          Sun ONE Message Queue
  > TIBCO        TIBCO Enterprise for JMS
  > Oracle       Oracle Java Message Service (OJMS)
  > Fiorano      FioranoMQ

> Open source implementations available (e.g., OpenJMS)
JMS Programming Model

- **Connection Factory**: Object used to create a connection.
- **Connection**: Single-threaded context used to send and receive messages.
- **Message Producer**: Creates a Session.
- **Session**: Creates a Message.
- **Message Consumer**: Receives from a Destination.
- **Destination**: Administered JNDI object encapsulating the identity of a message destination.
- **Message**: Creates a Destination.
- **Lookup**: Used to find objects in the JNDI hierarchy.
A Simple JMS Application

> Use JNDI (Java Naming and Directory Interface) to lookup a ConnectionFactory and one or more Destinations
> Use the ConnectionFactory to create a Connection with message delivery inhibited
> Use the Connection to create one or more Sessions
> Use a Session and a Destination to create the MessageProducers and MessageConsumers needed
> Tell the Connection to start delivery of messages
Point-to-Point (PTP) Model

> Messages are addressed to specific destination queues
  ⇒ Each message has only one consumer

> Clients
  > Extract messages from queues
  > Acknowledge successful processing

> Queues retain messages until successfully processed or expired
  ⇒ Producers and consumers have no timing dependency

> PTP model should be used if every message must be processed successfully by exactly one consumer

> JMS defines no interfaces for managing queues
  > Queues are static objects managed (e.g., created) by the administrator
Point-to-Point (PTP) Model

- QueueConnectionFactory
  - QueueConnection
    - QueueSession
      - QueueSender
      - QueueSession
        - QueueReceiver
        - Message
      - QueueReceiver
        - Queue
          - lookup
          - lookup
          - sends to
          - creates
          - creates
          - receives from
          - lookup
Synchronous Receive

1. Sends

QueueSender

Message

Queue

2. Retrieves

Queue

Message

QueueReceiver

3. Acknowledges
Asynchronous Receive

1. Sends
QueueSender → Queue

2. Delivers
Queue → QueueReceiver

3. Acknowledges
PTP Model (contd.)

<table>
<thead>
<tr>
<th>PTP-Specific Subclass</th>
<th>Unspecific Superclass</th>
</tr>
</thead>
<tbody>
<tr>
<td>QueueConnectionFactory</td>
<td>ConnectionFactory</td>
</tr>
<tr>
<td>QueueConnection</td>
<td>Connection</td>
</tr>
<tr>
<td>QueueSession</td>
<td>Session</td>
</tr>
<tr>
<td>Queue</td>
<td>Destination</td>
</tr>
<tr>
<td>QueueSender</td>
<td>MessageProducer</td>
</tr>
<tr>
<td>QueueReceiver, QueueBrowser</td>
<td>MessageReceiver</td>
</tr>
</tbody>
</table>
PTP Initialization

// Lookup QueueConnectionFactory
queueConnectionFactory = (QueueConnectionFactory)
    jndiContext.lookup("QueueConnectionFactory");

// Create QueueConnection
queueConnection =
    queueConnectionFactory.createQueueConnection();

// Create QueueSession
queueSession =
    queueConnection.createQueueSession(false,
        Session.AUTO_ACKNOWLEDGE);
Simple PTP Sender

// Lookup Queue
queue = (Queue) jndiContext.lookup(queueName);

// Create QueueSender
queueSender = queueSession.createSender(queue);

// Create a TextMessage
message = queueSession.createTextMessage();
message.setText("This is a message!");

// Send message
queueSender.send(message);
Simple PTP Receiver (Synchronous Receive)

// Create QueueReceiver
queueReceiver = queueSession.createReceiver(queue);

// Tell QueueConnection to start message delivery
queueConnection.start();

// Receive message
Message m = queueReceiver.receive();
Publish/Subscribe Model

> Messages are addressed to unique *topics*
> Consumers subscribe to topics
> Messages are delivered to consumers with matching topic
  ⇒ Each message may have any number of receivers
> Further filtering possible with *message selectors*
> Topics retain messages only until received and acknowledged by the *current* set of subscribers
  > Consumers only get messages published “after” their subscription
  > Subscriptions need some time to become effective
  ⇒ Producers and consumers have a timing dependency
Publish/Subscribe Model

- TopicConnection Factory
  - creates
  - TopicConnection
  - lookup

- TopicPublisher
  - creates
  - creates
  - sends To

- TopicSession
  - creates

- TopicSubscriber
  - receives from

- Message

- Topic
  - lookup
  - lookup

- Topic

Pub/Sub Model (contd.)

1. Publishes

2. Delivers

3. Acknowledges

TopicPublisher

Message

Topic

Message

TopicSubscriber 1

... ... ...

Message

TopicSubscriber n

... ... ...

2. Delivers

3. Acknowledges
Pub/Sub Model (contd.)

> Two types of subscriptions
  > **Non-Durable**
  > Messages are missed if subscriber is disconnected
  > **Durable**
  > Messages are retained while subscriber is disconnected until they have been delivered or expired

> Details of topics not defined
  > Use of wildcards, hierarchy of topics etc.

> JMS defines no interfaces for managing topics
  > Topics are static objects managed (e.g., created) by the administrator
## Pub/Sub Model (contd.)

<table>
<thead>
<tr>
<th>Pub/Sub-Specific Subclass</th>
<th>Superclass</th>
</tr>
</thead>
<tbody>
<tr>
<td>TopicConnectionFactory</td>
<td>ConnectionFactory</td>
</tr>
<tr>
<td>TopicConnection</td>
<td>Connection</td>
</tr>
<tr>
<td>TopicSession</td>
<td>Session</td>
</tr>
<tr>
<td>Topic</td>
<td>Destination</td>
</tr>
<tr>
<td>TopicPublisher</td>
<td>MessageProducer</td>
</tr>
<tr>
<td>TopicSubscriber</td>
<td>MessageReceiver</td>
</tr>
</tbody>
</table>
Pub/Sub Initialization

// Lookup TopicConnectionFactory
topicConnectionFactory = (TopicConnectionFactory) jndiContext.lookup("TopicConnectionFactory");

// Create TopicConnection
topicConnection =
    topicConnectionFactory.createTopicConnection();

// Create TopicSession
topicSession =
    topicConnection.createTopicSession(false,
        Session.AUTO_ACKNOWLEDGE);
Simple Pub/Sub Publisher

// Lookup Topic
topic = (Topic) jndiContext.lookup(topicName);

// Create TopicPublisher
topicPublisher = topicSession.createPublisher(topic);

// Create a TextMessage
message = topicSession.createTextMessage();
message.setText("This is a message!");

// Send message
topicPublisher.publish(message);
Simple Pub/Sub Subscriber (Asynchronous Receive)

    // Create TopicSubscriber
    topicSubscriber = topicSession.createSubscriber(topic);

    // Create MessageListener
    topicListener = new TextListener() {
        public void onMessage(Message message) {
            ...
        }
    }

    // Set MessageListener
    topicSubscriber.setMessageListener(topicListener);

    // Tell TopicConnection to start message delivery
    topicConnection.start();
Messages

- A JMS message consists of three parts
  - **Header**: consists of predefined fields
  - **Properties** (optional): additional fields
  - **Body** (optional): actual message content
Message Header

> Consists of predefined *fields* (name/value pairs)

<table>
<thead>
<tr>
<th>Header Field</th>
<th>Description</th>
<th>Set By</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMSDestination</td>
<td>Destination to which the message is being sent</td>
<td>send() or publish()</td>
</tr>
<tr>
<td>JMSDeliveryMode</td>
<td>PERSISTENT/NON_PERSISTENT</td>
<td>send() or publish()</td>
</tr>
<tr>
<td>JMSExpiration</td>
<td>Message expiration time</td>
<td>send() or publish()</td>
</tr>
<tr>
<td>JMSPriority</td>
<td>Message priority</td>
<td>send() or publish()</td>
</tr>
<tr>
<td>JMSMessageID</td>
<td>Unique message identifier, prefix “ID:”</td>
<td>send() or publish()</td>
</tr>
<tr>
<td>JMSTimestamp</td>
<td>Time the message was sent</td>
<td>send() or publish()</td>
</tr>
<tr>
<td>JMSCorrelationID</td>
<td>Identifier used to link messages to each other</td>
<td>Client</td>
</tr>
<tr>
<td>JMSReplyTo</td>
<td>Destination to which a reply should be sent</td>
<td>Client</td>
</tr>
<tr>
<td>JMSType</td>
<td>Identifier indicating the type of the message</td>
<td>Client</td>
</tr>
<tr>
<td>JMSRedelivered</td>
<td>Indicates that a message could be a duplicate</td>
<td>JMS Provider</td>
</tr>
</tbody>
</table>
Message Properties

> Consists of optional fields
> Are set by the client before the message is sent
> Some properties are predefined (prefix JMSX)
  > E.g. JMSXUserId specifies the user sending the message
> Can be evaluated by message selectors
> Values can be of type
  > boolean, byte, short, int, float, double, and String
## Message Types

> Refer to the content of the message body

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Message Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message</td>
<td>Empty; only header and properties</td>
</tr>
<tr>
<td>StreamMessage</td>
<td>Stream of Java primitive values</td>
</tr>
<tr>
<td>MapMessage</td>
<td>Set of name/value pairs</td>
</tr>
<tr>
<td>TextMessage</td>
<td>String (e.g., a XML document)</td>
</tr>
<tr>
<td>ObjectMessage</td>
<td>Serialized Java Object</td>
</tr>
<tr>
<td>BytesMessage</td>
<td>Stream of uninterpreted bytes</td>
</tr>
</tbody>
</table>
Message Reliability

> Property of the message

> *Non-persistent* messages
  > Delivered *at-most once*
  > Messages may get lost if provider fails
  > Low overhead

> *Persistent* messages
  > Delivered *exactly-once*
  > Guaranteed to survive provider failure
    (messages are logged to stable storage)
  > High overhead
Message Acknowledgement

> Clients acknowledge messages to indicate successful receipt or processing of a message
> Mode bound to session
> **Acknowledgement modes**
  > **DUPS_OK_ACKNOWLEDGE**
  > duplicates may be delivered if provider fails
  > **AUTO_ACKNOWLEDGE**
  > messages are automatically acknowledged
  > **CLIENT_ACKNOWLEDGE**
  > client acknowledges each message separately
Message Consumption

> Messages can be consumed either synchronously or asynchronously

> **Asynchronous consumption**
> > Client implements `MessageListener` Interface
> > Messages are delivered to `onMessage(Message m)` method

> **Synchronous consumption**
> > Client calls `receive()` method on destination
> > Can be `blocking` (infinitely or with timeout) or `non-blocking` (polling)
Message Selectors

> Specified as *String*

> Used by clients to select messages by evaluating conditions on header fields and properties

> Body values cannot be evaluated!

> Selector syntax is subset of *SQL92 conditional expressions*

> E.g., “`JMSType = ‘StockQuote’ and Symbol = ‘DT’`”
Message Selectors (contd.)

> Semantics in the PTP domain
  > Client extracts matching messages from queue (synchronous consumption)
  > Only matching messages are delivered to the client (asynchronous consumption)
  > Non-matching messages remain in the queue

> Semantics in the pub/sub domain
  > Only matching messages are delivered to the client
  > Non-matching messages are not retained
Using Temporary Destinations for Request/Reply

> Two types of temporary destinations
  > TemporaryQueue PTP model
  > TemporaryTopic Pub/Sub model
> Only consumers belonging to the same connection that created the temporary destination can receive messages
> Are created programmatically instead of administratively
  > queueSession.createTemporaryQueue() PTP model
  > topicSession.createTemporaryTopic() Pub/Sub model
> Can be used for request/reply interaction
  > Temporary destination is given in JMSReplyTo
  > JMSCorrelationID of reply set to JMSMessageID of request
Connections

> Connections connect the application to a message server
> Two types of connections
  > QueueConnection PTP model
  > TopicConnection Pub/Sub model
Destinations

> Are JNDI administered objects
> Client uses JNDI to lookup destinations
> Used by the client to specify the target of produced messages and the source of consumed messages
> Two types of destinations
  > Queue PTP model
  > Topic Pub/Sub model
Sessions

> Single-threaded context for producing and consuming messages
> Also determine the mode of acknowledgement
> Can be *transactional* or *non-transactional*
> Created by a connection
> Multiple sessions can be created within a connection
> Two types of sessions
  > *QueueSession* PTP model
  > *TopicSession* Pub/Sub model
Message Producers

> Created by a session
> Used by the client to send messages to a destination
> Two types of Message Producers
  > QueueSender PTP model
  > TopicPublisher Pub/Sub model
Message Consumers

> Created by a session
> Used by the client to receive messages sent to a destination

> Three types of MessageConsumers
  > QueueReceiver PTP model
  > QueueBrowser PTP model
  > TopicSubscriber Pub/Sub model

> MessageListeners used for asynchronous receive
JMS Interfaces Summary

- Connection
  - QueueConnection
  - TopicConnection

- Destination
  - Queue
  - Topic

- Session
  - QueueSession
  - TopicSession

- MessageProducer
  - QueueSender
  - TopicPublisher

- MessageConsumer
  - QueueReceiver
  - QueueBrowser
  - TopicPublisher
Transacted Sessions

> Sessions may be specified as *transacted*

> Transacted sessions

  > Realize *local messaging transactions*

  > Group a set of consumed and produced messages into an atomic unit of work

  > A transaction either commits (succeeds) or aborts (fails)

  > Messages are actually received/sent if transaction succeeds

> Transactions in a transacted session are chained

  ⇒ Transacted sessions always involved in a current transaction
Transacted Sessions (contd.)

> Producer Side
  > Produced messages are retained until commit
  > If transaction aborts messages are discarded

> Consumer Side
  > All consumed messages are kept until commit and are automatically acknowledged on commit
  > If transaction aborts messages are redelivered
Transacted Sessions Implications

> Transacted sessions cannot be used to implement request/reply interaction
> Production and consumption of the same message cannot be part of the same transaction
> Pub/Sub- and PTP-operations cannot be combined in a single transaction
Achieving Reliability

> For maximum reliability use
  > persistent messages
  > transacted sessions to send and receive messages
  > non-temporary queues (PTP) or durable subscriptions (Pub/Sub)

> Additional Mechanisms
  > Set message priority level 0 (low) - 9 (high)
    > JMS tries to deliver higher priority messages first
  > Allow messages to expire to get rid of obsolete messages
Distributed Transactions

> JMS supports the **Java Transaction API (JTA)** which is based on the **X/Open XA Interface**
> XA uses a **two-phase commit (2PC) protocol** and is supported by many transaction monitors
> Session acts as a transacted resource (⇒ 2PC)
> Still send and receive of a message cannot be part of the same transaction
> JMS provides **XA-compliant classes** with corresponding point-to-point and pub/sub-specific subclasses
  > XAConnectionFactory, XAConnection, XASession
> If J2EE is used, normally the details of transactions are transparently handled by the EJB container
Distributed Transactions (contd.)

// Lookup the transaction monitor
TransactionMonitor txm = jndiContext.lookup("TransactionMonitor");

// Start global transaction
txm.begin();

// Get transaction object
Transaction transaction = txm.getTransaction();

// Get XA-compliant factory, connection, and Session
XATopicConnectionFactory factory = (XATopicConnectionFactory)
    jndiContext.lookup("XATopicConnectionFactory");
XATopicConnection con = factory.createXATopicConnection();
XASession session = con.createXATopicSession();
Distributed Transactions (contd.)

// Get XAResource from session
XAResource xaResource = session.getXAResource();

// Enlist XAResource
transaction.enlist(xaResource);

// Publish and consume some JMS messages
...

// Commit transaction
transaction.commit();
Message-Driven Beans (MDBs) in J2EE

> In J2EE there are three types of beans
  > Entity beans, session beans, and message-driven beans (MDBs)
> Only MDBs can *asynchronously* receive JMS messages
> Are stateless but can call session or entity beans
> Have no remote or home interface
  ⇒ Cannot be “called”
> Are managed by the container
> All instances of a specific MDB are equivalent
  > A message is delivered to any instance
  > Concurrent processing
Message Server Bridges

> JMS implementations are not compatible to each other and to other message-driven middleware products (e.g., TIBCO Rendezvous)
> **Bridges** connect message servers by filtering, transforming, and forwarding messages
JMS Open Issues

> Important issues are not addressed by the JMS specification
  > Exception Handling
  > Load Balancing
  > Fault-Tolerance
  > End-to-end Security
  > Administration
  > Message Type Repository

> Many of these points are addressed and implemented differently by individual vendors
⇒ Products are incompatible!
Bibliography