ISEngineering

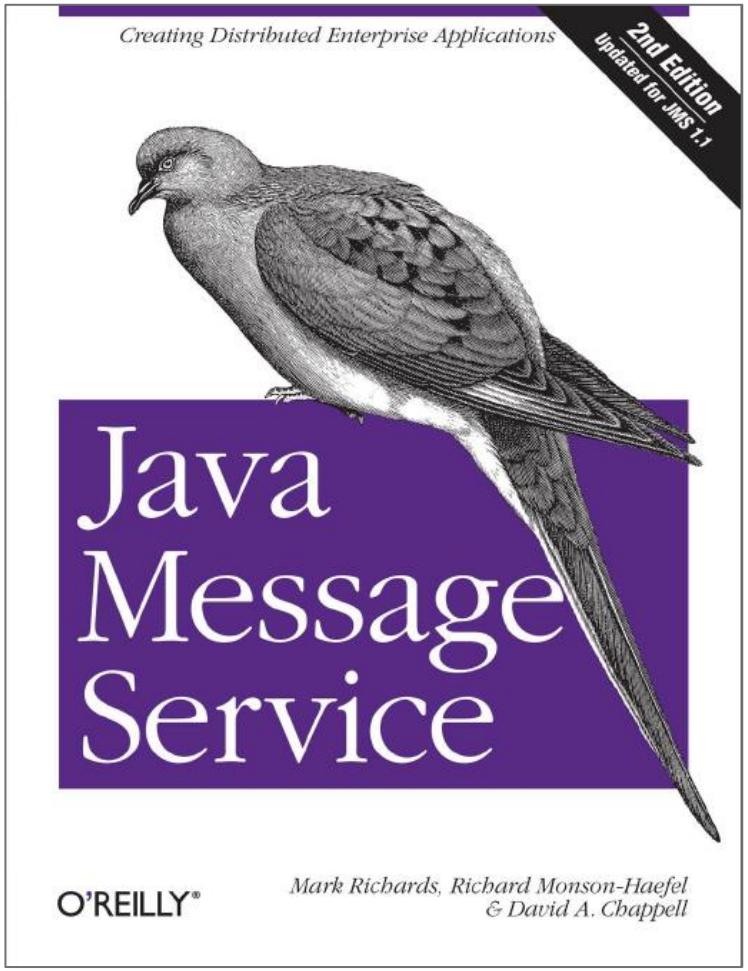
Technische Universität Berlin

Enterprise Computing: Exercise 1 - JMS

Markus Klems, Stefan Tai

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Java Message Service



Technische Universität Berlin
ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Point-to-Point

- The point-to-point messaging model allows JMS clients to send and receive messages both synchronously and asynchronously via virtual channels known as **queues**.
- In the point-to-point model, message producers are called **senders** and message consumers are called **receivers**.
- The point-to-point messaging model has traditionally been a pull-based or **polling-based model**, where messages are requested from the queue instead of being pushed to the client automatically.
- One of the distinguishing characteristics of point-to-point messaging is that messages sent to a queue are **received by one and only one receiver**, even though there may be many receivers listening on a queue for the same message.

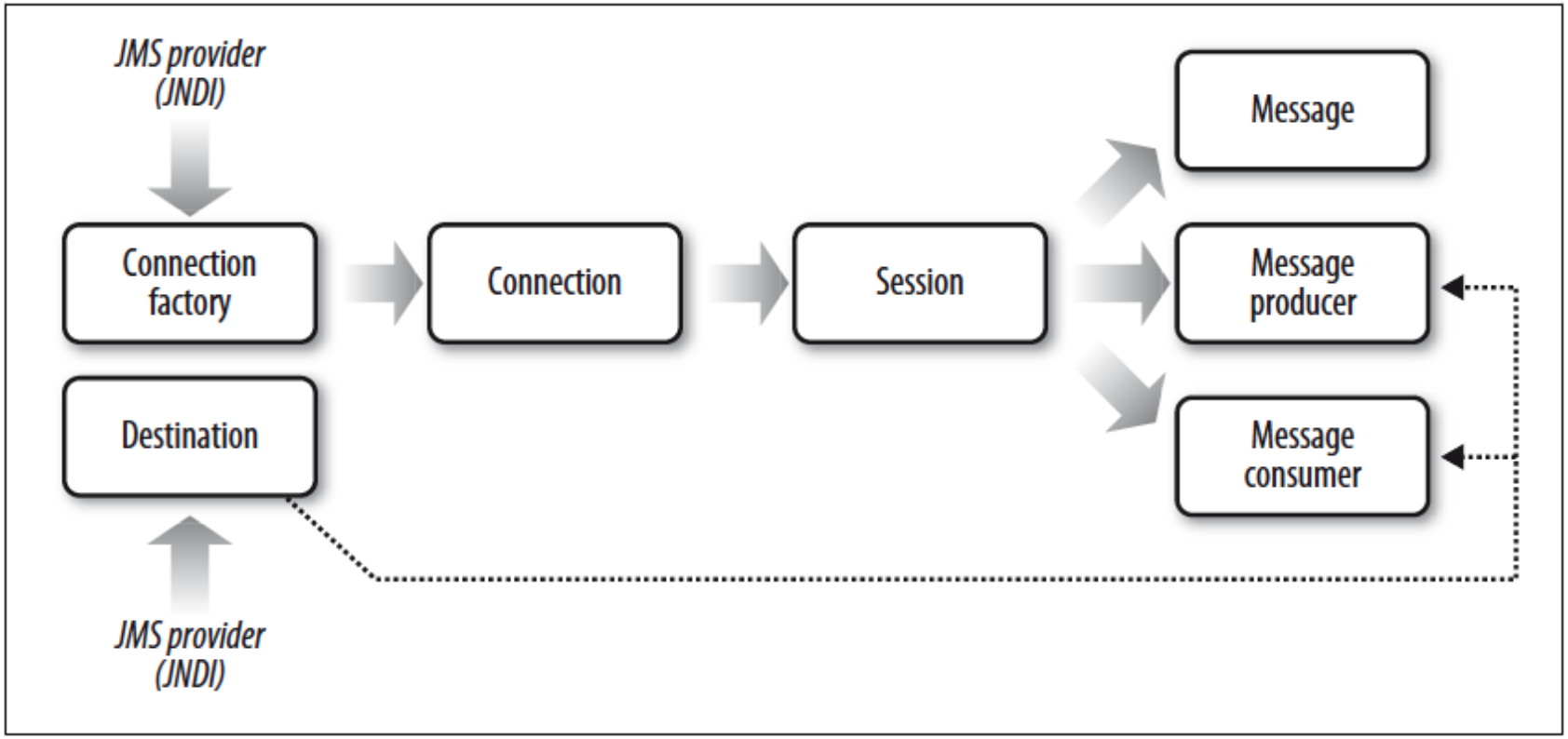
Source: "Java Message Service", O'Reilly, 2nd Ed.

Publish-and-Subscribe

- In the publish-and-subscribe model, messages are published to a virtual channel called a **topic**.
- Message producers are called **publishers**, whereas message consumers are called **subscribers**.
- Unlike the point-to-point model, messages published to a topic using the publish-and-subscribe model can be received by multiple subscribers. This technique is sometimes referred to as **broadcasting** a message.
- Every subscriber receives a copy of each message. The publish-and-subscribe messaging model is by and large a **push-based model**, where messages are automatically broadcast to consumers without them having to request or poll the topic for new messages.

Source: "Java Message Service", O'Reilly, 2nd Ed.

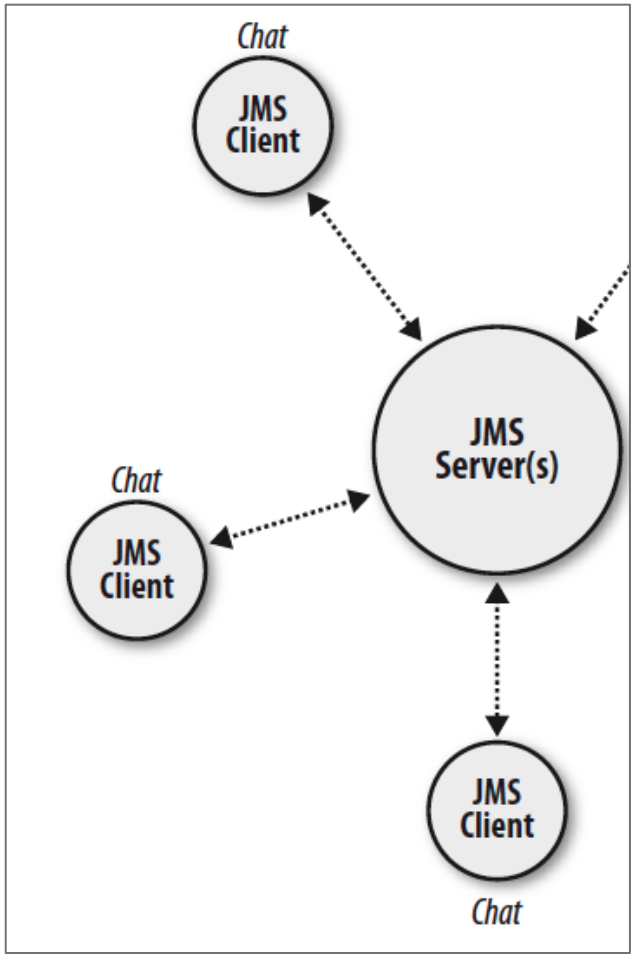
JMS general API core interfaces



Source: "Java Message Service", O'Reilly, 2nd Ed.

JMS Chat Example

- The chat client creates a JMS publisher and subscriber for a specific topic. The topic represents the chat room.
- The JMS server registers all the JMS clients that want to publish or subscribe to a specific topic.
- When text is entered at the command line of one of the chat clients, it is published to the messaging server.
- The messaging server identifies the topic associated with the publisher and delivers the message to all the JMS clients that have subscribed to that topic.



Source: "Java Message Service", O'Reilly, 2nd Ed.

JMS Chat Example: JNDI

- The chat client starts by obtaining a JNDI connection to the JMS messaging server. JNDI is an implementation-independent API for directory and naming systems.
- A directory service provides JMS clients with access to `ConnectionFactory` and `Destination` (topics and queues) objects.
  - `ConnectionFactory` and `Destination` objects are the only things in JMS that cannot be obtained using the JMS API—unlike connections, sessions, producers, consumers, and messages, which are manufactured using the factory pattern within the JMS API.
  - The **`ConnectionFactory`** is used to create JMS connections, which can then be used for sending and receiving messages.
  - **`Destination`** objects, which represent virtual channels (topics and queues) in JMS, are also obtained via JNDI and are used by the JMS client.

Source: "Java Message Service", O'Reilly, 2nd Ed.

JMS Chat Example: Initial Context

- Creating a connection to a JNDI naming service requires that a `javax.naming.InitialContext` object be created.
- An `InitialContext` is the starting point for any JNDI lookup—it's similar in concept to the root of a filesystem.
- The `InitialContext` provides a network connection to the directory service that acts as a root for accessing JMS administered objects.
- The properties used to create an `InitialContext` depend on which JMS directory service you are using.
- You could configure the initial context properties using the `Properties` Object directly in your source code, or preferably using an external *jndi.properties* file located in the classpath of the application.

Source: "Java Message Service", O'Reilly, 2nd Ed.

JMS Chat Example: TopicConnection

- The `TopicConnection` is created by the `TopicConnectionFactory`:

```
// Look up a JMS connection factory and create the connection
TopicConnectionFactory conFactory =
(TopicConnectionFactory)ctx.lookup(topicFactory);
TopicConnection connection = conFactory.createTopicConnection();
```

- The `TopicConnection` represents a connection to the message server. Each `TopicConnection` that is created from a `TopicConnectionFactory` is a unique connection to the server.

Source: "Java Message Service", O'Reilly, 2nd Ed.

JMS Chat Example: TopicSession

- After the `TopicConnection` is obtained, it's used to create `TopicSession` objects:

```
// Create two JMS session objects
TopicSession pubSession = connection.createTopicSession(
  false, Session.AUTO_ACKNOWLEDGE);
TopicSession subSession = connection.createTopicSession(
  false, Session.AUTO_ACKNOWLEDGE);
```

- The `TopicSession` is also used to create the `Message` objects that are delivered to the topic. The `pubSession` is used to create `Message` objects in the `writeMessage()` method.

Source: "Java Message Service", O'Reilly, 2nd Ed.

JMS Chat Example: MessageListener

- The pub/sub messaging model in JMS includes an in-process Java event model for handling incoming messages. An object simply implements the listener interface, in this case the `MessageListener`, and then is registered with the `TopicSubscriber`. A `TopicSubscriber` may have only one `MessageListener` object. Here is the definition of the `MessageListener` interface used in JMS:

```
package javax.jms;
public interface MessageListener {
  public void onMessage(Message message);
}
```

- When the `TopicSubscriber` receives a message from its topic, it invokes the `onMessage()` method of its `MessageListener` objects.

Source: "Java Message Service", O'Reilly, 2nd Ed.

QBorrower and QLender Example

To illustrate how point-to-point messaging works, we will use a simple decoupled request/reply example where a `QBorrower` class makes a simple mortgage loan request to a `QLender` class using point-to-point messaging. The `QBorrower` class sends the loan request to the `QLender` class using a `LoanRequest` queue, and based on certain business rules, the `QLender` class sends a response back to the `QBorrower` class using a `LoanResponseQ` queue indicating whether the loan request was approved or denied. Since the `QBorrower` is interested in finding out right away whether the loan was approved or not, once the loan request is sent, the `QBorrower` class will block and wait for a response from the QLender class before proceeding. This simple example models a typical messaging request/reply scenario.

Source: "Java Message Service", O'Reilly, 2nd Ed.

QBorrower

- The `QBorrower` class is responsible for sending a loan request message to a queue containing a salary and loan amount.
- The class is fairly straightforward: the constructor establishes a connection to the JMS provider, creates a `QueueSession`, and gets the request and response queues using a JNDI lookup.
- The main method instantiates the `QBorrower` class and, upon receiving a salary and loan amount from standard input, invokes the `sendLoanRequest` method to send the message to the queue.

Source: "Java Message Service", O'Reilly, 2nd Ed.

QLender

- The role of the `QLender` class is to listen for loan requests on the loan request queue, determine if the salary meets the necessary business requirements, and finally send the results back to the borrower.
- The `QLender` class is what is referred to as an asynchronous message listener, meaning that unlike the prior `QBorrower` class it will not block when waiting for messages. This is evident from the fact that the `QLender` class implements the `MessageListener` interface and overrides the `onMessage` method.
- The `onMessage` method first casts the message to a `MapMessage` (the message type we are expecting to receive from the borrower). It then extracts the salary and loan amount requested from the message payload, checks the salary to loan amount ratio, then determines whether to accept or decline the loan request.
- Once the loan request has been analyzed and the results determined, the `QLender` class sends the response back to the borrower.

Source: "Java Message Service", O'Reilly, 2nd Ed.

Message Correlation

- Once the message has been sent, the `QBorrower` class will block and wait for a response from the `QLender` on whether the loan was approved or denied.
- The first step in this process is to set up a message selector so that we can correlate the response message with the one we sent. This is necessary because there may be many other loan requests being sent to and from the loan request queues while we are making our loan request.
- To make sure we get the proper response back, we would use a technique called **message correlation**. Message correlation is required when using the request/reply model of point-to-point messaging where the queue is being shared by multiple producers and consumers:

```
String filter = "JMSCorrelationID = '" +
   msg.getJMSMessageID() + "'";
QueueReceiver qReceiver =
   qSession.createReceiver(responseQ, filter);
```

Source: "Java Message Service", O'Reilly, 2nd Ed.