Distributed Systems/Middleware

JMS
Introduction to MOM

• RPC/RMI foster a synchronous model
  – Natural programming abstraction, but:
    • Supports only point-to-point interaction
    • Synchronous communication is expensive
    • Intrinsically tight coupling between caller and callee, leads to “rigid” architectures

• Asynchronous communication models:
  – Often centered around the notion of message/event
    • Or persistent communication
  – Often supporting multi-point interaction
  – Brings more decoupling among components
The most straightforward form of asynchronous communication is message passing
- Typically directly mapped on/provided by the underlying network OS functionality (e.g., socket, datagrams)

In the case of MOM, the messaging infrastructure is often provided at the application level, by several “communication servers”
- Through what is nowadays called an overlay network
MOM: Types of communication

- Synchronous vs. asynchronous
  - Synchronous: the sender is blocked until the recipient has stored (or received, or processed) the message
  - Asynchronous: the sender continues immediately after sending the message
- Transient vs. persistent
  - Transient: sender and receiver must both be running for the message to be delivered
  - Persistent: the message is stored in the communication system until it can be delivered
- Several alternatives (and combinations) are provided in practice
- Many are used also for implementing synchronous models (e.g., RPC)
Different MOM flavors

- Message oriented communication comes in two flavors: Message queuing and publish-subscribe systems
- Several common characteristics
  - Both are “message oriented”
  - Both offer a strong decoupling among components
  - Both are often based on a set of “servers” to route messages from sender to recipients and/or to support persistency
- But also a lot of differences...
Message queuing

• Point-to-point persistent asynchronous communication
  – Typically guarantee only eventual insertion of the message in the recipient queue (no guarantee about the recipient’s behavior)
  – Communication is decoupled in time and space
  – Can be regarded as a generalization of the e-mail

• Intrinsically peer-to-peer architecture

• Each component holds an input queue and an output queue

• Many commercial systems:
  – IBM MQSeries (now WebSphere MQ), DECmessageQ, Microsoft Message Queues (MSMQ), Tivoli, Java Message Service (JMS), …
Queuing: Communication primitives

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put</td>
<td>Append a message to a specified queue</td>
</tr>
<tr>
<td>Get</td>
<td>Block until the specified queue is nonempty, and remove the first message</td>
</tr>
<tr>
<td>Poll</td>
<td>Check a specified queue for messages, and remove the first. Never block</td>
</tr>
<tr>
<td>Notify</td>
<td>Install a handler to be called when a message is put into the specified queue</td>
</tr>
</tbody>
</table>

program A
begin
...attach_queues();
...loop
  put(msg);
  ...
...get(msg);
end

program B
begin
...attach_queues();
...loop
  get(msg);
  ...
...put(msg);
end

Distributed Systems/Middleware: JMS
**Client-Server with queues**

- Clients send requests to the server’s queue
- The server asynchronously fetches requests, processes them, and returns results in the clients’ queues
  - Thanks to persistency and asynchronicity, clients need not remain connected
  - Queue sharing simplifies load balancing
Message queuing: Architectural Issues

- Queues are identified by symbolic names
  - Need for a lookup service, possibly distributed
  - Often pre-deployed static topology

- Queues are manipulated by queue managers
  - Local and/or remote, acting as relays (a.k.a. applicative routers)

- Relays often organized in an overlay network
  - Messages are routed by using application-level criteria, and by relying on a partial knowledge of the network
  - Improves fault tolerance
  - Provides applications with multi-point without IP-level multicast

- Message brokers provide application-level gateways supporting message conversion
  - Useful when integrating sub-systems
Publish-subscribe

- Application components can *publish* asynchronous *event notifications*, and/or declare their interest in event classes by issuing a *subscription*
  - extremely simple API: only two primitives (*publish*, *subscribe*)
  - event notifications are simply messages
- Subscriptions are collected by an *event dispatcher* component, responsible for routing events to all matching subscribers
  - Can be centralized or distributed
- Communication is transiently asynchronous, implicit, multipoint
- High degree of decoupling among components
  - Easy to add and remove components
  - Appropriate for dynamic environments
Subscription Language

- The expressiveness of the subscription language allows one to distinguish between:
  - **Subject-based** (or topic-based)
    - The set of subjects is determined a priori
    - Analogous to multicast
    - e.g., subscribe to all events about “Distributed Systems”
  - **Content-based**
    - Subscriptions contain expressions (event filters) that allow clients to filter events based on their content
    - The set of filters is determined by client subscriptions
    - A single event may match multiple subscriptions
    - e.g., subscribe to all events about a “Distributed System” class with date greater than 01.01.2015 and held in classroom EG2
- The two can be combined
JMS: An introduction

- The Java Message Service (JMS) defines an API to allow for creation, sending, receiving and reading messages belonging to an enterprise system
  - Strives for portability among different platforms
  - Tries to integrate seamlessly with the Java language
- First specification appeared in 1998
- JMS core concepts:
  - **JMS provider**: The entity that implements the JMS API for a messaging product
  - **JMS client**: The applicative software that uses the services of a JMS provider through the JMS API
  - **JMS domains**: Either point-to-point (queue based) or publish-subscribe (topic based)
**JMS domains**

- **The point-to-point domain:**
  - Each message is addressed to a specific *queue*
  - Client extract messages from the queue(s) established to hold their messages

- **The publish-subscribe domain is centered on a specific content-hierarchy (*topics*)**
  - The system takes care of distributing the messages belonging to a specific part of the content hierarchy to all clients interested in that content

- **JMS provides a set of domain-independent interfaces allowing communication in both domains under a common API**
  - Usually called “common interfaces”
• Each message has only one consumer.

• A sender and a receiver of a message have no timing dependencies. The receiver can fetch the message whether or not it was running when the client sent the message.

• The receiver acknowledges the successful processing of a message.
Each message can have multiple consumers.

Publishers and subscribers have a timing dependency. A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.

The JMS API relaxes this timing dependency to some extent by allowing subscribers to create **durable subscriptions**, which receive messages sent while the subscribers are not active. Durable subscriptions provide the flexibility and reliability of queues but still allow clients to send messages to many recipients.
JMS Architecture

- **JMS Clients**: Java programs sending and/or receiving messages
- **Messages**: Data containers defined by the application to exchange information
- **JMS Provider**: The messaging product implementing the JMS API together with (possible) administrative and quality-control functionality
- **Administered Objects**: Preconfigured JMS objects created by an administrator and used by the clients
• **Destinations** and **connection factories** 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

• JMS clients access these objects through interfaces that are portable, so a client application can run with little or no change on more than one implementation of the JMS API.

• Ordinarily, an administrator configures administered objects in a JNDI namespace, and JMS clients then access them by using **resource injection**.
**Administered Objects**

- A **connection factory** is the object a client uses to create a connection to a provider.
  - A connection factory encapsulates a set of connection configuration parameters that has been defined by an administrator.
  - Each connection factory is an instance of the `ConnectionFactory`, `QueueConnectionFactory`, or `TopicConnectionFactory` interface.

- A **destination** is the object a client uses to specify the target of messages it produces and the source of messages it consumes.
  - In the PTP messaging domain, destinations are called **queues**.
  - In the pub/sub messaging domain, destinations are called **topics**.
JMS Setup

- Administered objects are placed in a JNDI namespace under well-known identifiers
- To establish a connection to a JMS provider, clients retrieves a `ConnectionFactory` from the JNDI namespace
JMS Object Relationships
JMS Connections


- A **connection** encapsulates a virtual connection with a JMS provider. For example, a connection could represent an open TCP/IP socket between a client and a provider service daemon.

- Connections implement the Connection interface. When you have a ConnectionFactory object, you can use it to create a Connection:

```java
Connection connection = connectionFactory.createConnection();
```

- Before an application completes, you must close any connections you have created
  - `connection.close();`

- Before your application can consume messages, you must call the connection’s start method.
  - If you want to stop message delivery temporarily without closing the connection, you call the stop method.
A session is a single-threaded context for producing and consuming messages. You use sessions to create the following:

- Message producers
- Message consumers
- Messages
- Queue browsers
- Temporary queues and topics
• Sessions serialize the execution of message listeners.
• A session provides a transactional context with which to group a set of sends and receives into an atomic unit of work.

• Sessions implement the Session interface.
• After you create a Connection object, you use it to create a Session:

```java
Session session = connection.createSession(false,
                                           Session.AUTO_ACKNOWLEDGE);
```

• The first argument means the session is not transacted; the second means the session automatically acknowledges messages when they have been received successfully.
**Message Producers**

- **Message producer** is an object that is created by a session and used for sending messages to a destination. It implements the MessageProducer interface.

- You use a Session to create a MessageProducer for a destination. The following examples show that you can create a producer for a Destination object, a Queue object, or a Topic object.
  
  ```java
  MessageProducer producer = session.createProducer(dest);
  MessageProducer producer = session.createProducer(queue);
  MessageProducer producer = session.createProducer(topic);
  ```

- You can create an unidentified producer by specifying null as the argument to `createProducer`. With an unidentified producer, you do not specify a destination until you send a message.

- After you have created a message producer, you can use it to send messages by using the send method:
  ```java
  producer.send(message);
  ```
• Messages can be consumed in either of two ways:

• **Synchronously:** A subscriber or a receiver explicitly fetches the message from the destination by calling the receive method. The receive method can block until a message arrives or can time out if a message does not arrive within a specified time limit.

• **Asynchronously:** A client can register a message listener with a consumer. A message listener is similar to an event listener. Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener’s onMessage method, which acts on the contents of the message.
Message Consumers

• A **message consumer** is an object that is created by a session and used for receiving messages sent to a destination.

• A message consumer allows a JMS client to register interest in a destination with a JMS provider:
  
  ```java
  MessageConsumer consumer = session.createConsumer(dest);
  MessageConsumer consumer = session.createConsumer(queue);
  MessageConsumer consumer = session.createConsumer(topic);
  ```

• After you have created a message consumer it becomes active, and you can use it to receive messages. You can use the close method for a `MessageConsumer` to make the message consumer inactive.

• Message delivery does not begin until you start the connection you created by calling its start method.
You use the receive method to consume a message synchronously. You can use this method at any time after you call the start method:

```java
connection.start();
Message m = consumer.receive();
Message m = consumer.receive(1000); // time out after a second
```
Asynchronous Consumption

- A message listener is an object that acts as an asynchronous event handler for messages. This object implements the MessageListener interface, which contains one method, `onMessage`.

- You register the message listener with a specific MessageConsumer by using the `setMessageListener` method:
  ```java
  Listener myListener = new Listener();
  consumer.setMessageListener(myListener);
  ```

- After you register the message listener, you call the start method on the Connection to begin message delivery.

- When message delivery begins, the JMS provider automatically calls the message listener’s `onMessage` method whenever a message is delivered. The `onMessage` method takes one argument of type `Message`, which your implementation of the method can cast to any of the other message types.
Developing JMS applications

- A JMS client will typically:
  - Use JNDI to find a ConnectionFactory object
  - Use JNDI to find one or more Destination object(s)
    - Predefined identifiers can also be used to instantiate new Destinations
  - Use the ConnectionFactory to establish a Connection with the JMS Provider
  - Use the Connection to create one or more Session(s)
  - Combine Session(s) and Destination(s) to create the needed MessageConsumer and MessageProducer
  - Tell the Connection to start delivery of messages
**Example: The point-to-point domain**

```java
class Sender {
    static Context ictx = null;
    public static void main(String[] args) throws Exception {
        ictx = new InitialContext();
        Queue queue = (Queue) ictx.lookup("queue");
        QueueConnectionFactory qcf = (QueueConnectionFactory) ictx.lookup("qcf");
        ictx.close();

        QueueConnection qc = qcf.createQueueConnection();
        QueueSession qs = qc.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
        QueueSender qsend = qs.createSender(queue);
        TextMessage msg = qs.createTextMessage();

        int i;
        for (i = 0; i < 10; i++) {
            msg.setText("Test number " + i);
            qsend.send(msg);
        }
        qc.close();
    }
}

class Receiver {
    static Context ictx = null;
    public static void main(String[] args) throws Exception {
        ictx = new InitialContext();
        Queue queue = (Queue) ictx.lookup("queue");
        QueueConnectionFactory qcf = (QueueConnectionFactory) ictx.lookup("qcf");
        ictx.close();

        QueueConnection qc = qcf.createQueueConnection();
        QueueSession qs = qc.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
        QueueReceiver qrec = qs.createReceiver(queue);
        TextMessage msg;

        qc.start();

        int i;
        for (i = 0; i < 10; i++) {
            msg = (TextMessage) qrec.receive();
            System.out.println("Msg received: " + msg.getText());
        }
        qc.close();
    }
}
```
Example: The pub-sub domain

```java
import javax.jms.*;
import javax.naming.*;
import java.util.*;

public class Publisher {
    static Context ictx = null;
    public static void main(String[] args) throws Exception {
        ictx = new InitialContext();
        Topic topic = (Topic) ictx.lookup("topic");
        TopicConnectionFactory tcf = (TopicConnectionFactory) ictx.lookup("tcf");
        ictx.close();
        TopicConnection tc = tcf.createTopicConnection();
        TopicSession ts = tc.createTopicSession(true,
            Session.AUTO_ACKNOWLEDGE);
        TopicPublisher tpub = ts.createPublisher(topic);
        TextMessage msg = ts.createTextMessage();
        int i;
        for (i = 0; i < 10; i++) {
            msg.setText("Test number " + i);
            tpub.publish(msg);
        }
        ts.commit();
        tc.close();
    }
}

class MsgListener implements MessageListener {
    String id;
    public MsgListener() {id = "";}
    public MsgListener(String id) {this.id = id;}
    public void onMessage(Message msg) {
        TextMessage tmsg = (TextMessage) msg;
        try {
            System.out.println(id + ":: " + tmsg.getText());
        } catch (JMSException jE) {
            jE.printStackTrace();
        }
    }
}

class Subscriber {
    static Context ictx = null;
    public static void main(String[] args) throws Exception {
        ictx = new InitialContext();
        Topic topic = (Topic) ictx.lookup("topic");
        TopicConnectionFactory tcf = (TopicConnectionFactory) ictx.lookup("tcf");
        ictx.close();
        TopicConnection tc = tcf.createTopicConnection();
        TopicSession ts = tc.createTopicSession(true,
            Session.AUTO_ACKNOWLEDGE);
        TopicSubscriber tsub = ts.createSubscriber(topic);
        tsub.setMessageListener(new MsgListener());
        tc.start();
        System.in.read();
        tc.close();
    }
}
```

Distributed Systems/Middleware: JMS
Example: The common interfaces

```java
import javax.jms.*; import javax.naming.*;

public class Producer {
    static Context ictx = null;
    public static void main(String[] args) throws Exception {
        ictx = new InitialContext();
        Queue queue = (Queue) ictx.lookup("queue");
        Topic topic = (Topic) ictx.lookup("topic");
       ConnectionFactory cf =
            (ConnectionFactory) ictx.lookup("cf");
        ictx.close();

        Connection cnx = cf.createConnection();
        Session sess = sess.createSession(false,
            Session.AUTO_ACKNOWLEDGE);
        MessageProducer prod = sess.createProducer(null);
        TextMessage msg = sess.createTextMessage();

        int i;
        for (i = 0; i < 10; i++) {
            msg.setText("Test number " + i);
            prod.send(queue, msg);
            prod.send(topic, msg);
        }

        cnx.close();
    }
}

import javax.jms.*; import javax.naming.*;

public class Subscriber {
    static Context ictx = null;
    public static void main(String[] args) throws Exception {
        ictx = new InitialContext();
        Queue queue = (Queue) ictx.lookup("queue");
        Topic topic = (Topic) ictx.lookup("topic");
        Connection cnx = cf.createConnection();
        Session sess = sess.createSession(false,
            Session.AUTO_ACKNOWLEDGE);
        MessageProducer prod = sess.createProducer(null);
        TextMessage msg = sess.createTextMessage();

        int i;
        for (i = 0; i < 10; i++) {
            msg.setText("Test number " + i);
            prod.send(queue, msg);
            prod.send(topic, msg);
        }

        cnx.close();
    }
}
```
JMS 2.0
A Simplified API

• The classic API is not deprecated and will remain part of JMS indefinitely.

• The simplified API consists of three new interfaces
  – **JMSContext** replaces the separate Connection and Session objects in the classic API with a single object.
  – **JMSProducer** is a lightweight replacement for the MessageProducer object in the classic API. It allows message delivery options, headers, and properties to be configured using *method chaining* (sometimes known as a *builder pattern*).
  – **JMSCConsumer** replaces the MessageConsumer object in the classic API and is used in a similar way.
Sending Comparison

```java
public void sendMessageJMS11(ConnectionFactory connectionFactory, Queue queue, String text) {
    try {
        Connection connection = connectionFactory.createConnection();
        try {
            Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            MessageProducer messageProducer = session.createProducer(queue);
            TextMessage textMessage = session.createTextMessage(text);
            messageProducer.send(textMessage);
        } finally {
            connection.close();
        }
    } catch (JMSException ex) {
        // handle exception (details omitted)
    }
}

public void sendMessageJMS20(ConnectionFactory connectionFactory, Queue queue, String text) {
    try (JMSCContext context = connectionFactory.createContext()) {
        context.createProducer().send(queue, text);
    } catch (JMSRuntimeException ex) {
        // handle exception (details omitted)
    }
}
```

Distributed Systems/Middleware: JMS
What's different?

- The JMS 1.1 version used a finally block to call close on the Connection after use.
  - In JMS 2.0, the JMSContext object also has a close method that needs to be called after use.
  - There's no need to explicitly call close from your code. JMSContext implements the Java SE 7 java.lang.AutoCloseable interface. This means that if we create the JMSContext in a try-with-resources block, the close method will be called automatically at the end of the block without the need to explicitly add it to your code.

- Non-transacted sessions in which any received messages are acknowledged automatically is the default in JMS 2.0

- There's no need to create a TextMessage object and set its body to be the specified string.
  - Done automatically

- Runtime Exception instead of checked exceptions
Receiving Comparison

public String receiveMessageJMS11(ConnectionFactory connectionFactory, Queue queue) {
    String body = null;
    try {
        Connection connection = connectionFactory.createConnection();
        try {
            Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            MessageConsumer messageConsumer = session.createConsumer(queue);
            connection.start();
            TextMessage textMessage = TextMessage)messageConsumer.receive();
            body = textMessage.getText();
        } finally {
            connection.close();
        }
    } catch (JMSException ex) {
        // handle exception (details omitted)
    }
    return body;
}

public String receiveMessageJMS20(
    ConnectionFactory connectionFactory; Queue queue) {
    String body = null;
    try (JMSContext context = connectionFactory.createContext();)
    {
        JMSConsumer consumer = session.createConsumer(queue);
        body = consumer.receiveBody(String.class);
    } catch (JMSRuntimeException ex) {
        // handle exception (details omitted)
    }
    return body;
}

Distributed Systems/Middleware: JMS
What's different?

- Whereas in JMS 1.1 we need to call `connection.start()` to start delivery of messages to the consumer, in the JMS 2.0 simplified API we don't: the connection is automatically started.

- There's no need to receive a `Message` object, cast it to a `TextMessage`, and then call `getText` to extract the message body. Instead, we call `receiveBody`, which returns the message body directly.
Injection

• If you're writing a Java EE Web or EJB application, it is possible to "inject" the JMSCContext into your code and leave it to the application server to work out when to create it and when to close it.

• The following code is a fragment from a Java EE 7 session bean or servlet that injects a JMSCContext and uses it to send a message:

```java
@Inject @JMSConnectionFactory("jms/connectionFactory")
private JMSCContext context;

@Resource(lookup = "jms/dataQueue") private Queue dataQueue;

public void sendMessage(String body) {
    context.send(dataQueue, body);
}
```
Message Body Extraction

- In JMS 1.1, the message body is obtained using methods specific to the message type, such as the `getText` method on `TextMessage`.
  - However, when a message is received by an application, the JMS API always provides the message as a `javax.jms.Message` object, which needs to be cast to the appropriate subtype before the body can be obtained.

  ```java
  Message message = consumer.receive(1000); // returns a TextMessage
  String body = ((TextMessage) message).getText();
  ```

- JMS 2.0 adds the `getBody` method on `Message`. This method takes the expected body type as a parameter and does not require you to perform a cast on either the message or the body.

  ```java
  Message message = consumer.receive(1000); // returns a TextMessage
  String body = message.getBody(String.class);
  ```
void onMessage(Message message) { // delivers a BytesMessage
    int bodyLength = ((BytesMessage)message).getBodyLength();
    byte[] bytes = new byte[bodyLength];
    int bytesCopied = ((BytesMessage)message).readBytes(bytes);
    ...
}

void onMessage(Message message) { // delivers a BytesMessage
    byte[] bytes = message.getBody(byte[].class);
    ...
}

JMSCConsumer consumer = ...
Message message = consumer.receive(1000); // returns a TextMessage
String body = ((TextMessage) message).getText();

JMSCConsumer consumer = ...
Message message = consumer.receive(1000); // returns a TextMessage
String body = message.getBody(String.class);

JMSCConsumer consumer = ...
String body = consumer.receiveBody(String.class,1000);
Easier Resource Configuration

- Java EE 7 introduces a platform default JMS connection factory. This is a built-in connection factory that connects to the application server's built-in JMS provider.
  - Applications can obtain this connection factory by performing a JNDI lookup using the name java:comp:DefaultJMSConnectionFactory without the need to previously create the connection factory using administrative tools:

  ```java
  @Resource(lookup="java:comp/DefaultJMSConnectionFactory") ConnectionFactory cf
  ```

- This connection factory is intended for use by the many applications that use the built-in JMS provider and don't need to add any application-specific configuration.
JMS 2.0 MESSAGING
Multiple Consumers on Single Topic

- In JMS 1.1, a subscription on a topic was not permitted to have more than one consumer at a time.
  - This meant that the work of processing messages on a topic subscription could not be shared among multiple threads, connections, or Java Virtual Machines (JVMs), thereby limiting the scalability of the application.

- JMS 2.0 introduces the shared subscription

- JMS 1.1 Example

  ```java
  private void createUnsharedConsumer(ConnectionFactory connectionFactory, Topic topic)
  throws JMSException {
      Connection connection = connectionFactory.createConnection();
      Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      MessageConsumer messageConsumer = session.createConsumer(topic);
      connection.start();
      Message message = messageConsumer.receive(10000);
      while (message != null) {
          System.out.println("Message received: " + ((TextMessage) message).getText());
          message = messageConsumer.receive(10000);
      }
      connection.close();
  }
  ```

How do we make the application more scalable by sharing the work of processing these messages between, say, two JVMs, with one JVM processing some of the messages and the other JVM processing the remaining messages?
Multiple Consumers on Single Topic

- If you use createConsumer to create a second consumer in a separate JVM (or a separate thread on the same JVM), each consumer will use a separate subscription, and so it will receive a copy of every message received by the topic.
  - That's not what we want.
  - If you think of a "subscription" as a logical entity that receives a copy of every message sent to the topic, then we want the two consumers to use the same subscription.

```java
private void createSharedConsumer(ConnectionFactory connectionFactory, Topic topic) throws JMSException {
    Connection connection = connectionFactory.createConnection();
    Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    MessageConsumer messageConsumer = session.createSharedConsumer(topic, "mySubscription");
    connection.start();
    Message message = messageConsumer.receive(10000);
    while (message != null) {
        System.out.println("Message received: " + ((TextMessage) message).getText());
        message = messageConsumer.receive(10000);
    }
    connection.close();
}
```
JMS 2.0 Topic Subscriptions

- **Unshared nondurable subscriptions.** These are available in both JMS 1.1 and JMS 2.0 and are created using createConsumer. They can have only a single consumer. Setting the client identifier is optional.

- **Unshared durable subscriptions.** These are available in both JMS 1.1 and JMS 2.0 and are created using createDurableSubscriber or (in JMS 2.0 only) createDurableConsumer. They can have only a single consumer. Setting the client identifier is compulsory, and the subscription is identified by the combination of the subscription name and client identifier.

- **Shared nondurable subscriptions.** These are available in JMS 2.0 only and are created using createSharedConsumer. They can have any number of consumers. Setting the client identifier is optional. The subscription is identified by the combination of the subscription name and the client identifier, if it is set.

- **Shared durable subscriptions.** These are available in JMS 2.0 only and are created using createSharedDurableConsumer. They can have any number of consumers. Setting the client identifier is optional. The subscription is identified by the combination of the subscription name and the client identifier, if it is set.
Delivery Delay

- You can now specify a delivery delay on a message. The JMS provider will not deliver the message until after the specified delivery delay has elapsed.

```java
private void sendWithDeliveryDelayClassic(ConnectionFactory connectionFactory, Queue queue) throws JMSException {
    // send a message with a delivery delay of 20 seconds
    try (Connection connection = connectionFactory.createConnection();)
    {
        Session session = connection.createSession();
        MessageProducer messageProducer = session.createProducer(queue);
        messageProducer.setDeliveryDelay(20000);
        TextMessage textMessage = session.createTextMessage("Hello world");
        messageProducer.send(textMessage);
    }
}

private void sendWithDeliveryDelaySimplified(ConnectionFactory connectionFactory, Queue queue) throws JMSException {
    // send a message with a delivery delay of 20 seconds
    try (JMSContext context = connectionFactory.createContext();)
    {
        context.createProducer().setDeliveryDelay(20000).send(queue, "Hello world");
    }
}
```

Asynchronous Message Sending

- Normally, when a persistent message is sent, the send method does not return until the JMS client has sent the message to the server and received a reply to notify the client that the message has been safely received and persisted. We call this a **synchronous send**.

- When a message is sent **asynchronously**, the send method sends the message to the server and then returns control to the application without waiting for a reply from the server.
  - When a reply is received back from the server to indicate that the message has been received by the server and persisted, the JMS provider notifies the application by invoking the callback method **onCompletion** on an application-specified **CompletionListener** object.
MORE JMS CONCEPTS
More JMS concepts

Concurrency

- JMS restricts concurrent access to Destination, ConnectionFactory and Connection
- Session, MessageProducer and MessageConsumer are not designed for concurrent usage by multiple threads
  - Session supports transactions, for which it is often difficult to write multithreading support
    - If concurrent send and receive of messages is required, it is better to use multiple sessions
  - Asynchronous message consumers running concurrently would require complicated concurrency support
    - Also, either a Session uses synchronous receive, or asynchronous ones
      - It is erroneous to use both
More JMS concepts
Message ordering & acknowledgment

- FIFO between a given sender and a given receiver modulo priority
- No guarantee across destinations or across a destination’s messages sent by different producers
- If a `Session` is transacted, message acks are automatically handled by `commit`
- If a `Session` is not transacted, you can either set
  - `DUPS_OK_ACKNOWLEDGE`: instructs the `Session` to lazily acknowledge messages; this minimizes overhead while possibly generating duplicates messages if the Provider fails
  - `AUTO_ACKNOWLEDGE`: the `Session` automatically acknowledges messages while avoiding duplicates at the cost of higher overhead
  - `CLIENT_ACKNOWLEDGE`: the JMS Client explicitly acknowledges messages by calling a message’s `acknowledge()` method
More JMS concepts
Message delivery mode

- It can either be:
  - NON_PERSISTENT: does not require logging to stable storage while exposing minimum overhead, a JMS provider’s failure can cause a message to be lost
  - PERSISTENT: instructs the JMS provider to take extra care to guarantee message delivery
- A JMS provider must deliver a NON_PERSISTENT message at most once
- On the contrary, a JMS provider must deliver a PERSISTENT message once and only once
More JMS concepts

JMS message model

• The header part contains information used by both clients and providers to identify and route messages
  – All messages support the same set of header fields
  – Several fields are present, among them:
    • JMSMessageID contains a value that uniquely identify a message
    • JMSCorrelationID contains an identifier that links this message to another one (it can be used to link a query to a reply)
    • JMSReplyTo contains a Destination supplied by a client where a reply to the message must be sent
    • JMSTimestamp contains the time a message has been handed off to a JMS provider to be sent
    • JMSPriority contains the message’s priority, ten levels of priority can be defined, from 0 (lowest) to 9 (highest)
• Additional information can be possibly added using the property part
  – It can contain application-specific properties, standard properties defined by JMS (act as optional header fields), or provider-specific properties
  – Properties are <string, value> pairs, where value can be one of Java built-in types or String
  – When a message is received, its properties are in read-only mode
  – The method getPropertyNames() returns an Enumeration of all the property names
  – The main use of properties is to support customized message selection
• The body part specifies the actual message content
More JMS concepts

JMS message selectors

- Allow a JMS Client to customize the messages actually delivered, based on the application’s interests
- A message selector is a predicate (a `String`) on a message’s header and property values, whose syntax is derived from SQL’s conditional expression syntax
  - The selector is passed as a parameter when the `Consumer` is created
  - The message is delivered when the selector evaluates to true
  - Type checking must be taken into account
- Example:

```java
myMsg.setIntProperty("Property1", 2);
myMsg.setStringProperty("Property2", "Hi");
myMsg.setFloatProperty("Property3", 3.4821);
myMsg.setStringProperty("Property4", "2");
```

<table>
<thead>
<tr>
<th>Message Selector</th>
<th>Matches?</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Property1 = 2 AND Property3 &lt; 4”</td>
<td>Yes</td>
</tr>
<tr>
<td>“Property3 NOT BETWEEN 2 AND 5”</td>
<td>No</td>
</tr>
<tr>
<td>“Property2 IN (’Ciao’, ’Hallo’, ’Hi’)”</td>
<td>Yes</td>
</tr>
<tr>
<td>“Property5 IS NULL”</td>
<td>Yes</td>
</tr>
<tr>
<td>“Property4 &gt; 1”</td>
<td>No</td>
</tr>
</tbody>
</table>
AWS - SQS

QUEUES IN THE CLOUD
Simple Queue Service - SQS

- Allows building highly scalable processing pipelines using loosely coupled parts (queues)
- Queues allow for flexibility, asynchrony, and fault tolerance
- Each step in the pipeline retrieves work units from an instance of the queue service, processes them as appropriate, and then writes completed work into another queue for further processing
- Queues work well when the requirements—be it time, CPU, or I/O speed—for each processing step for a particular work unit vary widely
- SQS usage is charged based on the amount of data transferred and the number of requests made to SQS
Asynchronous Messaging Patterns
Cloud Example

• http://animoto.com/
• A web application that produces videos from photos, music and video clips
• About 4 million users
• Launched in 2007
• Facebook application launched in April 2008
  – Mid-April about 750,000 people signed up in three days
• iPhone app
• Big need to scale according to workload
• Operations connected through queues (SQS)
  – One queue to fetch images from URLs
  – One queue with the list of render jobs
  – Decouples various parts of the site
  – Scaling is decided looking at queues states
• Photos, videos, music stored in S3
• Producing video takes 8-9 minutes in average
• The system is managed using the Rightscale solution
• Up to 4000 EC2 instances used simultaneously
• Focus on CPU scalability
• Take advantage on new GPU-based instances
  – Better quality videos