argonavis
tecnologia e arte

JMS
java message service

Helder da Rocha

JAVA EE 7

Capítulo 9: Java Message Service

1 Introdução

A mensageria é uma solução de integração de sistemas que permite a construção de aplicações distribuídas cujas partes não interagem entre si diretamente, mas trocam mensagens de forma assíncrona, através de um mediador central, com baixíssimo acoplamento.

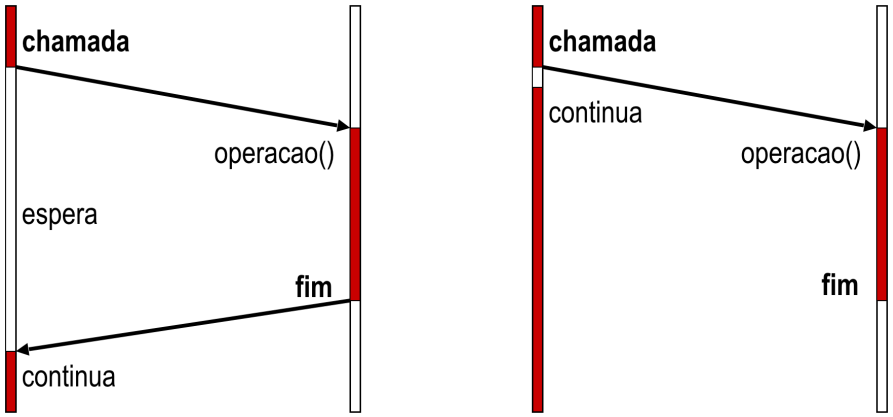
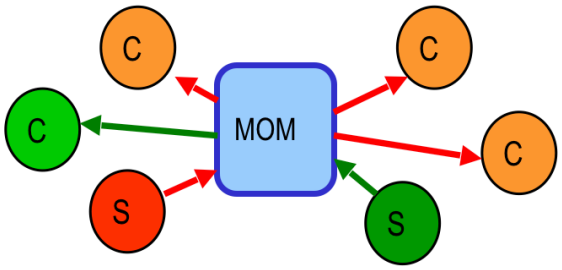


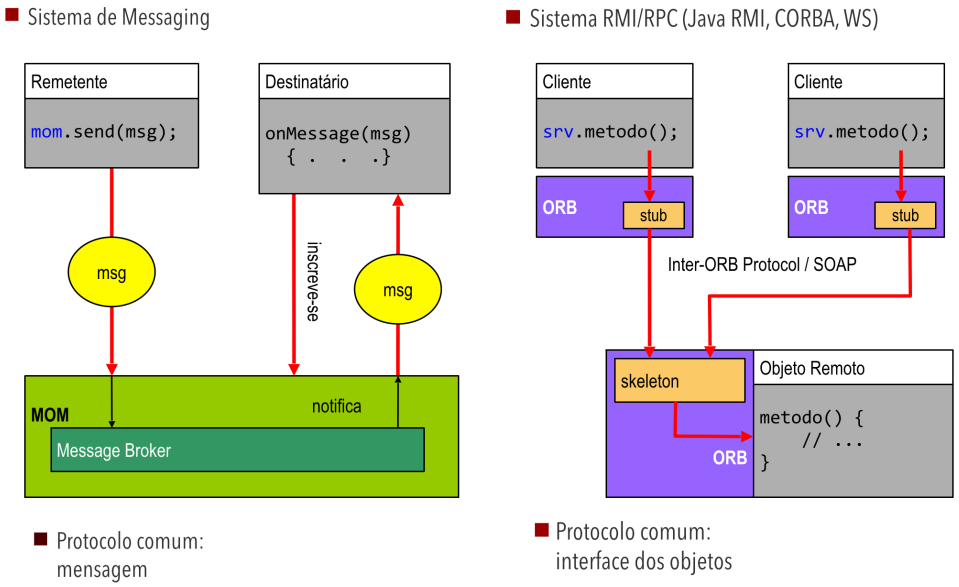
Figura 1.1 Comunicação síncrona (RMI/EJB) e assíncrona (Mensageria/JMS).

O mediador central geralmente é um serviço fornecido por um servidor de aplicação (standalone ou cluster). É o provedor de serviços de mensageria e também é chamado de Message Oriented Middleware – MOM, Message Broker ou Message Queue – MQ. Toda a comunicação entre aplicações de mensageria é realizada através desse provedor.



Um MOM oferece serviços de mediação e enfileiramento de mensagens, fábrica para criação de conexões e canais para troca de mensagens. A maior parte dos provedores de mensageria também oferece serviços adicionais como transações, segurança, configurações de QoS, etc.

Aplicações que se comunicam via mensageria distinguem-se das comunicações que se comunicam via RPC não apenas pela natureza assíncrona da comunicação, mas também pelo protocolo de comunicação. Mensageria utiliza mensagens que encapsulam todos os detalhes da aplicação, enquanto que RPC requer a exposição da interface dos serviços.



A mensageria é disponibilizada em servidores de aplicação Java EE através da API Java Message Service (JMS).

2 Java Message Service

JMS é a API Java padrão para mensageria e que é implementada em todos os servidores que suportam Java EE (JBoss/WildFly, GlassFish, TomEE, etc.). JMS também pode ser usada em servidores que não oferecem outros serviços além da mensageria (como ActiveMQ). JMS consiste de uma coleção de interfaces uniformes para serviços de mensageria e permite escrever aplicações que irão rodar em diferentes provedores.

A versão de JMS correspondente ao Java EE 7 é JMS 2.0, mas neste tutorial iremos mostrar também JMS 1.1, já que serviços populares como ActiveMQ ainda não suportam JMS 2.0.

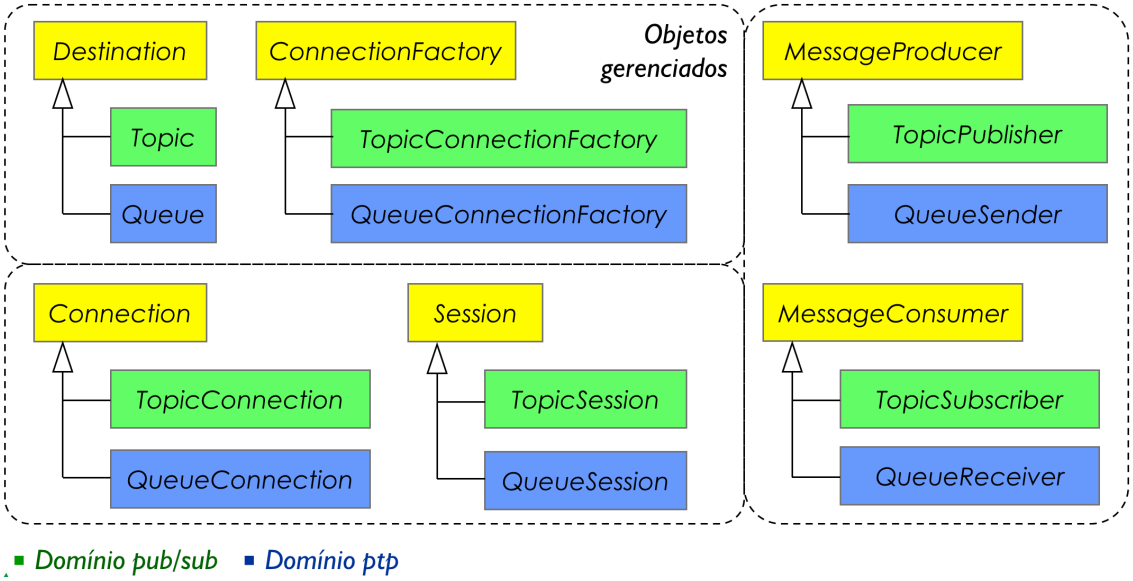
As interfaces JMS permitem que aplicações tenham acesso ao serviço de mensageria. Aplicações JMS são sempre clientes de mensageria. O serviço funciona como um mediador que viabiliza a comunicação entre clientes produtores e consumidores. O ponto de comunicação é chamado de destino (Destination) e representa um canal de mensageria. Geralmente é implementado como uma fila de mensagens. Produtores e consumidores comunicam-se enviando e retirando mensagens do canal.

A comunicação usando JMS suporta recebimento síncrono (usando um blocking method) ou assíncrono (usando um listener de eventos). As mensagens podem conter qualquer tipo de objeto. Aplicações JMS podem usar a infraestrutura de mensageria para transferir dados, texto, tipos primitivos, objetos serializados, XML, JSON, etc.

A API consiste de diversas interfaces implementados por objetos criados por uma Abstract Factory inicializada a partir de objetos gerenciados pelo serviço. Os objetos são conexões e destinos. Eles são obtidos através da injeção de recursos ou JNDI. A partir de uma conexão (que é multithreaded) é possível obter sessões (único thread), produtores, consumidores e mensagens.

2.1 Arquitetura JMS

JMS suporta dois estilos (ou domínios) de mensageria: ponto-a-ponto (PTP) e difusão (Publish-Subscribe). Esses domínios estão refletidos nas interfaces da API:



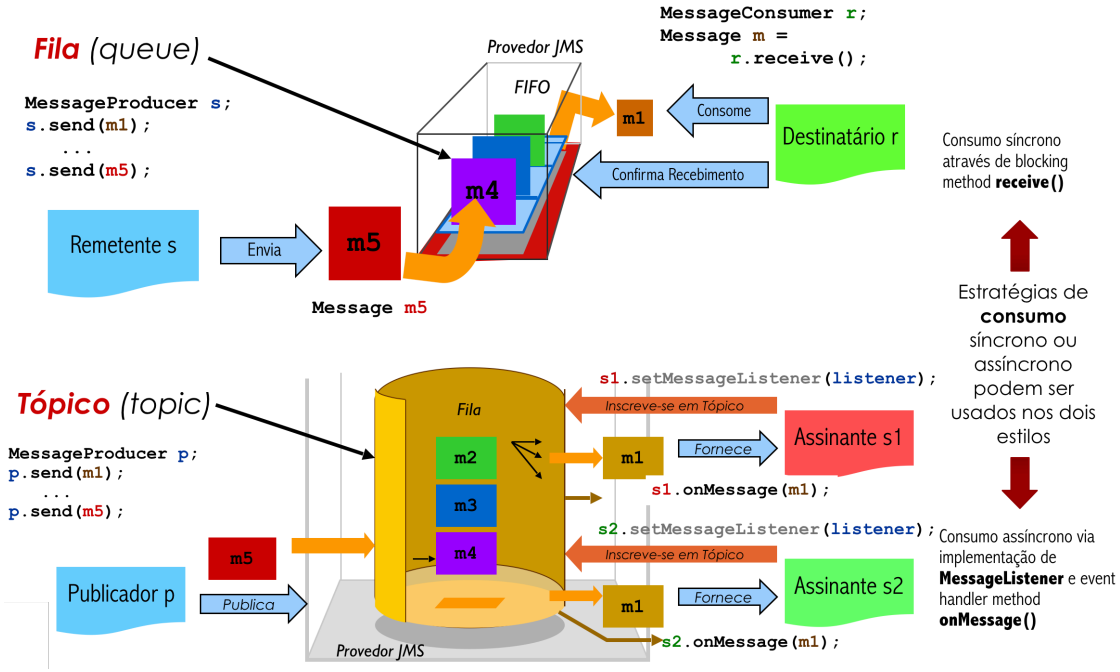
No domínio PTP, cada mensagem possui apenas um consumidor. Um ou mais produtores podem enviar mensagens para uma fila, e quando um consumidor consumir a mensagem, ela é retirada da fila. O envio e recebimento são desacoplados. Um produtor pode enviar uma mensagem para um consumidor que ainda não está disponível esperando mensagens. Quando o consumidor conectar-se no serviço, poderá consumir a mensagem.

Mensagens enviadas para canais configurados no domínio Publish-Subscribe podem ser consumidas por diversos consumidores. Os consumidores mantém assinaturas para o canal de difusão (chamado de topic) e quando um produtor envia uma mensagem ela é consumida por todos os assinantes.

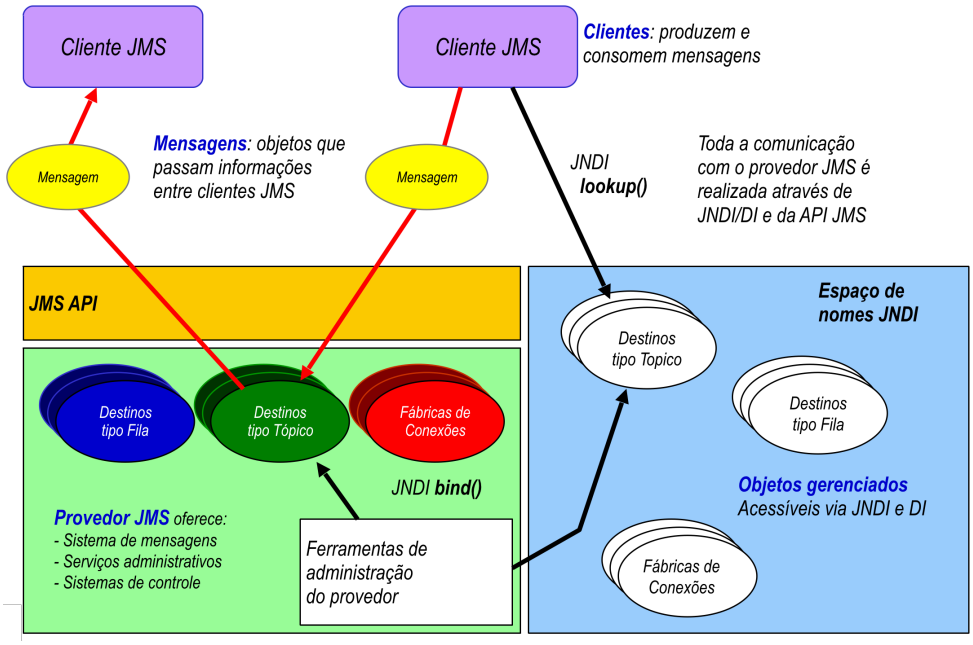
Clientes podem consumir mensagens depois que a assinatura foi iniciada, mas eles precisam estar ativos quando a mensagem for enviada ou elas serão perdidas. É possível configurar assinantes como duráveis, para que suas mensagens sejam guardadas até que conectem-se novamente.

Desde JMS 1.1 pode-se usar o mesmo código para criar aplicações que usam qualquer um dos estilos e recomenda-se sempre utilizar as classes genéricas (e não as implementações de cada domínio) para construir aplicações (ex: use MessageProducer e não TopicPublisher ou QueueSender).

O diagrama abaixo ilustra os dois domínios do JMS (ponto a ponto e difusão), e duas formas diferentes de consumir mensagens (consumo síncrono ou baseado em eventos).



Um provedor de serviços de mensageria que implementa JMS oferece canais de comunicação e conexões através de objetos gerenciados, obtidos via JNDI ou injeção de dependências.



2.2 Destinos (via JNDI)

2.2.1 Filas (Queue)

O canal utilizado como destino no domínio ponto-a-ponto é representado pela interface Queue. Um Queue retém mensagens até que sejam retiradas da fila ou expirem. Apenas um cliente pode retirar cada mensagem enviada.

```
Queue fila = (Queue) ctx.lookup("jms/nome-jndi");
```

Normalmente utiliza-se um Queue através da interface Destination, a menos que se precise usar operações específicas de Queues (como um QueueBrowser, que analisa mensagens sem consumi-las.)

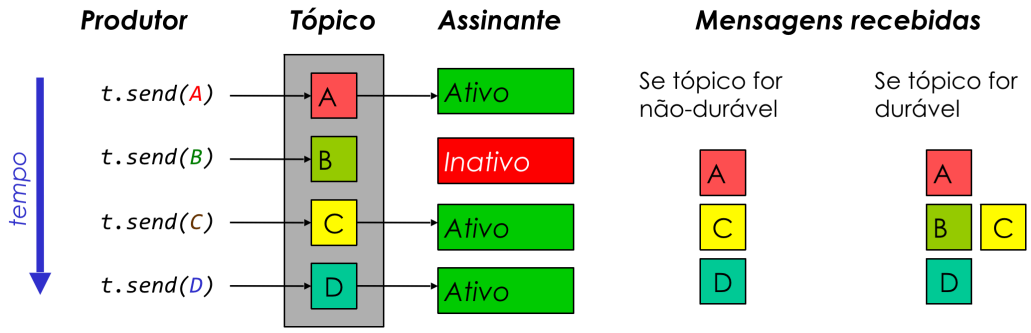
```
Destination fila = (Destination) ctx.lookup("jms/nome-jndi");
```

2.2.2 Tópicos (Topic)

O canal utilizado como destino no domínio de difusão é representado pela interface Topic. Cada Topic pode ter diversos clientes assinantes, cada um dos recebe uma cópia das mensagens enviadas. Os clientes precisam ser assinantes antes que o envio ocorra.

```
Topic topico = (Topic) ctx.lookup("jms/nome-jndi");
Destination topico = (Destination) ctx.lookup("jms/nome-jndi");
```

Em tópicos "duráveis", os assinantes não precisam estar ativos no momento do envio. Suas mensagens serão preservadas (mas podem expirar, se tiverem um timeout).



2.3 Conexões

Conexões suportam múltiplas sessões paralelas. Uma conexão é representada pela interface Connection. Os métodos start() e stop() respectivamente iniciam e interrompem (temporariamente) o envio de mensagens. O método close() encerra a conexão.

```
con.start();
```

Para clientes que exigem autenticação, há um método que permite criar uma conexão informando nome e senha.

```
Connection con = factory.createConnection(userid, senha);
```

2.4 Sessões

Uma sessão é obtida a partir de uma conexão:

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

Cada sessão utiliza um único thread. Os métodos usados para a criação de sessões permitem configurar modos de comunicação com o serviço de mensageria quanto à confirmação de recebimento/envio:

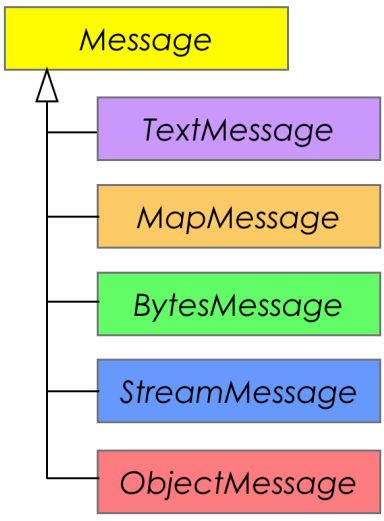
```
AUTO_ACKNOWLEDGE
CLIENT_ACKNOWLEDGE
DUPS_OK_ACKNOWLEDGE
```

2.5 Mensagens

Mensagens possuem duas partes: um cabeçalho que contém cabeçalhos padrão JMS e possíveis propriedades definidas pelo usuário, e um corpo, que é opcional e contém dados de qualquer tipo anexados à mensagem.

Há seis tipos de mensagens em JMS. Elas são criadas a partir de uma sessão. A interface Message é superinterface dos outros cinco tipos:

- Message: contém apenas cabeçalho (não tem corpo). É suficiente para enviar propriedades (strings, números, etc.) Todos os outros tipos de mensagem são descendentes de Message.
- TextMessage: pode conter corpo de texto (ex: XML).
- MapMessage: contem Map como corpo.
- BytesMessage: corpo de stream de bytes (ex: imagens, dados comprimidos).
- StreamMessage: corpo de tipos primitivos (interface DataInput/DataOutput).
- ObjectMessage: corpo contém objeto serializado.



2.6 Criação e configuração de mensagens

Objetos Session possuem métodos para criar cada tipo de mensagem. Os métodos tem a forma create*Tipo*():

```
TextMessage tm = session.createTextMessage();
Message m = session.createMessage();
```

A interface Message possui métodos para guardar e recuperar tipos primitivos e strings, como propriedades no cabeçalho da mensagem:

```
m.setStringProperty("Codigo", "123");
m.setIntProperty("Quantidade", 900);
String codigo = m.getStringProperty("Codigo");
```

Cada subclasse de message possui métodos próprios para anexar e recuperar dados anexados no corpo. Por exemplo, mensagens de texto possuem o método get/setText() que permite ler ou gravar texto no corpo da mensagem.

```
tm.setText("<resultado><codigo>200</codigo></resultado>");
String anexo = tm.getText();
```

2.7 Cabeçalhos e propriedades

As propriedades no cabeçalho da mensagem que iniciam em "JMS" são reservadas e usadas para propriedades geradas pelo sistema. Alguns exemplos são: JMSMessageID, JMSDestination, JMSExpiration, JMSPriority.

Para a configuração de propriedades definidas pelo programador é preciso usar os métodos get/set*Tipo*Property() da classe Message:

```
message.setStringProperty("Formato", "Imagem JPEG");
```

2.8 Seletores

Seletores são expressões usadas para filtrar as mensagens que serão consumidas. Elas são expressões SQL *where* que operam sobre propriedades e cabeçalhos da mensagem. Por exemplo:

```
String seletor = "Formato LIKE '%Imagem%'AND " +
                 "JMSExpiration > 10   AND " +
                 "Size IS NOT NULL";
```

Seletores são usados para configurar consumidores. A expressão abaixo cria um consumidor de mensagens usando um método de Session, e passa o seletor como segundo argumento. Apenas mensagens compatíveis com o seletor serão consumidas (as que não forem, permanecerão no canal):

```
MessageConsumer consumer = session.createConsumer (topico, seletor);
```

2.9 Produtores

A interface MessageProducer representa produtores de mensagens. Um produtor é criado através de uma sessão junto com o canal onde produz mensagens.

```
MessageProducer producer =
                session.createProducer(fila);
```

Uma vez criado, o produtor pode ser usado para enviar mensagens. Seu principal método é send(mensagem), que têm versões que além da mensagem a ser enviada, recebe parâmetros para configurara a qualidade do serviço: send(msg, modo, prioridade, ttl), onde:

- *modo* indica se a mensagem será armazenada em meio persistente ou não. Os valores podem ser DeliveryMode.NON_PERSISTENT (default) ou DeliveryMode.PERSISTENT.
- *prioridade* é um número de 0 – 9 , representando a prioridade da mensagem. O valor default é 4 = Message.DEFAULT_PRIORITY; mensagens com prioridade maior podem chegar antes.
- *ttl* e o tempo de vida (time-to-live) da mensagem em milissegundos. O default é zero e significa que a mensagem não expira nunca; esta propriedade também pode ser configurada através da propriedade JMSExpiration.

Diversos aspectos da QoS podem ser configurados via métodos do MessageProducer, por exemplo get/setDeliveryMode(), get/setPriority() e get/setTimeToLive().

2.10 Consumidores síncronos

A interface MessageConsumer é usada para criar consumidores:

```
MessageConsumer consumer =
                session.createConsumer(fila);
```

É preciso que a conexão seja iniciada antes de começar a consumir (isto não é necessário antes de produzir porque a produção e envio automaticamente inicia a conexão):

```
con.start();
```

O consumo de forma síncrona é realizado através de receive(), receive(timeout) ou receiveNoWait(). Os dois primeiros métodos bloqueiam o thread até que a mensagem esteja disponível ou que expire o timeout. O terceiro método consome a mensagem apenas se ela já estiver esperando na fila.

```
Message message = consumer.receive();        // blocking
                  consumer.receive(20000);   // 20 segundos
                  consumer.receiveNoWait(); // no blocking
```

Se um seletor for definido no consumidor, apenas as mensagens que passarem pelo seletor serão consumidas:

```
session.createConsumer (fila, "Codigo-produto = 'L940");
```

2.11 Consumidores assíncronos

O consumo assíncrono requer a criação de um event handler: a implementação da interface MessageListener que possui método onMessage(javax.jms.Message):

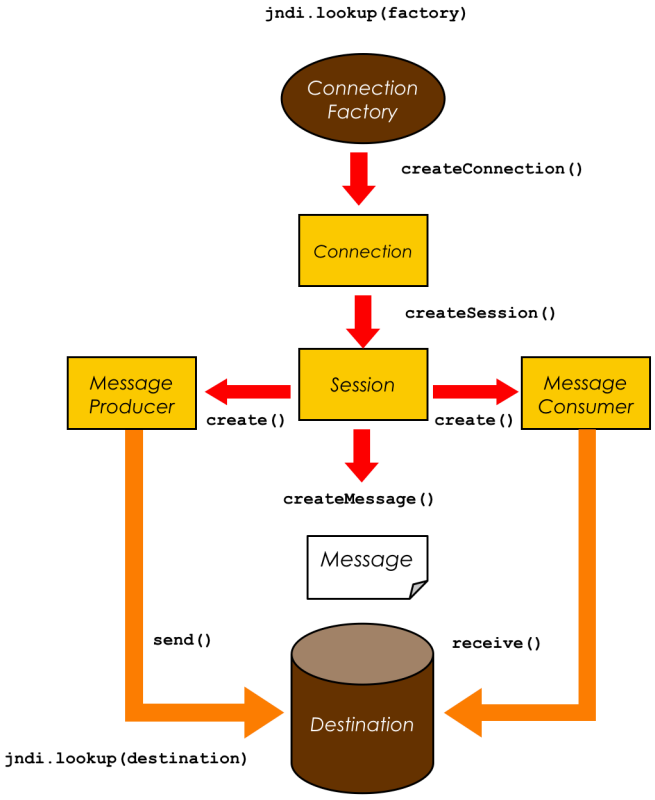
```
public class MyListener implements MessageListener {
   public void onMessage(Message msg) {
      TextMessage txtMsg = (TextMessage) msg;
      System.out.println( "Mensagem recebida: " +
                          txtMsg.getText() )
   }
}
```

O método onMessage() não pode deixar escapar exceções (JMS 1.x), portanto o código acima precisa estar em um bloco try-catch.

Para que objeto seja notificado, é preciso registrá-lo em um MessageConsumer:

```
consumer.setMessageListener( new MyListener() );
con.start();  // iniciar a conexão
```

O diagrama abaixo ilustra a criação de consumidores, produtores e mensagens em JMS 1.1



3 Message Driven Beans

Um MessageDrivenBean (MDB) é um consumidor JMS assíncrono. Ele implementa a interface MessageListener e o método onMessage(). É também um objeto gerenciado pelo servidor de aplicações (Enterprise JavaBean).

3.1 Configuração

Um MDB deve ser anotado com @MessageDriven. Serviços Java EE disponíveis no servidor de aplicações podem ser configurados de diversas formas. O contexto pode ser injetado usando

```
@Resource MessageDrivenContext
```

E vários serviços do JMS podem ser configurados de forma declarativa (seletores JMS, filas, tópicos, etc.)

```
@MessageDriven(activationConfig={
   @ActivationConfigProperty(propertyName="destinationType",
                             propertyValue="javax.jms.Queue"),
   @ActivationConfigProperty(propertyName="destination",
                             propertyValue="queue/ProdutoQueue")
})
public class ProdutoMDB implements MessageListener {
    @Resource
    private MessageDrivenContext mdc;
    public void onMessage(Message message) { … }
}
```

3.2 Produtores JMS para MDB

Um MDB é apenas um listener para um canal. É um consumidor assíncrono de mensagens. Produtores podem ser quaisquer clientes que enviam mensagens para o mesmo destino, sejam locais ou remotos. Podem ser produtores externos e clientes standalone (como os mostrados anteriormente) ou outros EJBs e componentes dentro do servidor de aplicações. Produtores que estiverem dentro do mesmo container podem injetar filas e conexões:

```
@Stateless
public class ClienteEJB {
    public void metodo() throws Exception {

        @Resource(mappedName="java:/JmsXA") ConnectionFactory connectionFactory;

        Connection con =  factory.createConnection();
        Session session =  con.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

```
        @Resource(mappedName="topic/testTopic") Topic topic;

        Producer publisher = session.createProducer(topic);
        TextMessage msg = session.createTextMessage();
        msg.setText("Teste");
        publisher.send(msg);
    }
}
```

4 JMS 2.0 (Java EE 7)

JMS 2.0 simplificou bastante a interface JMS combinando a conexão e sessão JMS em um único objeto: JMSContext. Com um JMSContext é possível criar produtores, consumidores, mensagens, destinos temporários e queue browsers. Servidores Java EE 7 precisam suportar JMS 2.0, mas muitos servidores standalone ainda não suportam (ex: ActiveMQ) e podem usar apenas JMS 1.1.

4.1 JMSContext

Um JMSContext pode ser criado a partir de um ConnectionFactory ou injetado:

```
JMSContext ctx = connectionFactory.createContext();
@Inject @JMSConnectionFactory("jms/MyConnectionFactory")   private JMSContext ctx2;
```

Ideal é criar dentro de um bloco try-with-resources, que fecha o contexto automaticamente ao final:

```
try(JMSContext ctx = connectionFactory.createContext();) { … }
```

4.2 JMSProducer e JMSConsumer

A partir de um contexto pode-se obter produtores e consumidores diretamente, que são implementações das interfaces JMSProducer e JMSConsumer:

```
JMSProducer producer = jmsCtx.createProducer();
```

Os exemplos a seguir comparam o uso de JMS 2.0 e 1.1 para enviar e receber mensagens.

4.2.1 Envio de mensagens JMS 1.1

```
public void sendJMS11(ConnectionFactory conFactory, Queue queue, String text) {
   try {
      Connection con = conFactory.createConnection();
      try {
         Session session = con.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
      }
   }
```

4.2.2 Envio de mensagens JMS 2.0

```
   public void sendJMS2(ConnectionFactory conFactory, Queue queue, String text) {
      try (JMSContext context = conFactory.createContext();){
         context.createProducer().send(queue, text);
      } catch (JMSRuntimeException ex) {
         // handle exception
      }
   }
```

4.2.3 Recebimento síncrono JMS 1.1

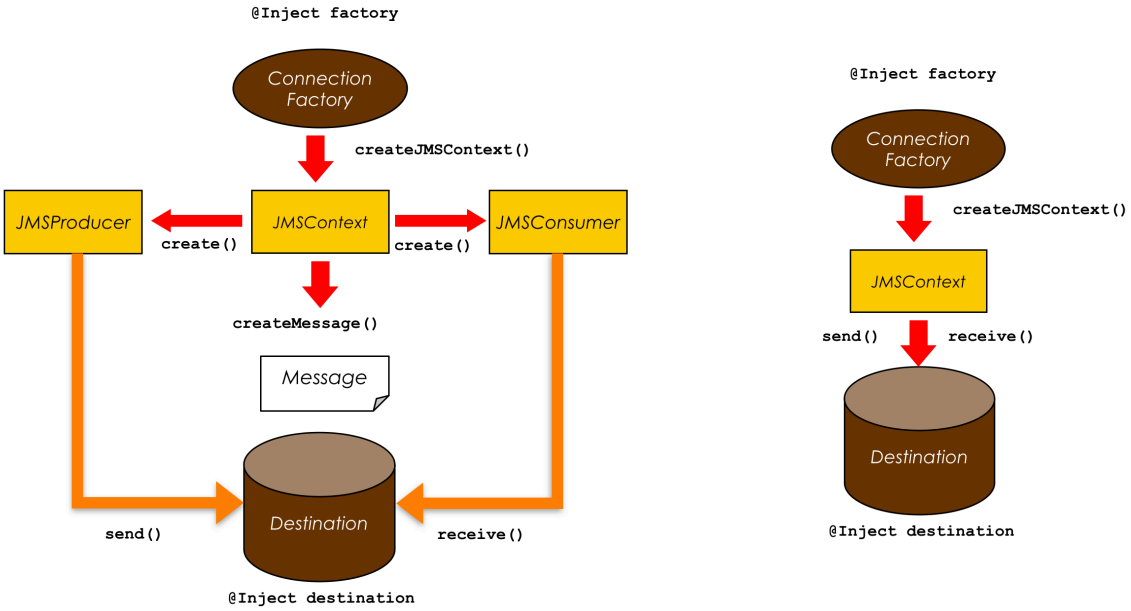
```
   public void sendJMS11(ConnectionFactory conFactory, Queue queue, String text) {
      try {
         Connection con = conFactory.createConnection();
         try {
            Session session = con.createSession(false,Session.AUTO_ACKNOWLEDGE);
            MessageConsumer messageConsumer = session.createConsumer(queue);
            con.start();
            TextMessage textMessage = (TextMessage)messageConsumer.receive();
            this.messageContents = textMessage.getText();
         } finally {
            connection.close();
         }
      } catch (JMSException ex) {
         // handle exception
      }
   }
```

4.2.4 Recebimento síncrono JMS 2.0

```
   public void sendJMS2(ConnectionFactory conFactory, Queue queue, String text) {
      try (JMSContext context = conFactory.createContext();){
         JMSConsumer consumer = context.createConsumer(queue);
         this.messageContents = consumer.receiveBody(String.class);
      } catch (JMSRuntimeException ex) {
         // handle exception
   }
```

```
}
```

O diagrama abaixo ilustra o uso desses recursos. Compare com o diagrama anterior que ilustra os mesmos processos em JMS 1.1:



4.3 Outras diferenças

Em JMS 1.1, um canal pode ter múltiplas assinaturas (cada assinatura recebe uma cópia da mensagem), mas uma assinatura tem sempre apenas um consumidor. Em JMS 2.0 é possível criar assinaturas compartilhadas, ou seja, dois consumidores ou mais em uma assinatura. A mensagem pode então ser recebida por um ou ambos. Isto permite maior escalabilidade.

JMS 2.0 também introduziu destinos temporários, que são canais que duram o tempo da conexão. Isto evita a necessidade de criar canais novos no servidor para tarefas simples.

É mais fácil extrair o conteúdo de uma mensagem em JMS 2.0 usando Message.getBody(tipo.class) em vez de getText(), getBytes(), etc. de cada tipo de mensagem:

```
void onMessage(Message message) { // BytesMessage JMS 1.1
     int len = ((BytesMessage)message).getBodyLength();
     byte[] bytes = new byte[len];
     int bytes = ((BytesMessage)message).readBytes(bytes);
  …
void onMessage(Message message){ // BytesMessage JMS 2.0
     byte[] bytes = message.getBody(byte[].class);
   …
```

Existe também um método para fazer o oposto. JMSConsumer.receiveBody(tipo.class) é usado em recebimento síncrono e devolve diretamente o conteúdo da mensagem.

5 Referências

5.1 Especificações Java EE

[1] EJB e MDB http://jcp.org/aboutJava/communityprocess/final/jsr345/

[2] JMS https://java.net/projects/jms-spec/pages/Home

[3] Java EE https://java.net/projects/javaee-spec/pages/Home

5.2 Tutorial Java EE

[4] Java EE 6 http://docs.oracle.com/javaee/6/tutorial/doc/

[5] Java EE 7 http://docs.oracle.com/javaee/7/tutorial/doc/

5.3 JMS 2.0 (Java EE 7)

[6] Nigel Deakin. "What's New in JMS 2.0". Oracle. 2013.

[7] Part I http://www.oracle.com/technetwork/articles/java/jms20-1947669.html

[8] Part II http://www.oracle.com/technetwork/articles/java/jms2messaging-1954190.html

5.4 Configuração de JNDI externo

[9] Glassfish https://glassfish.java.net/javaee5/ejb/EJB_FAQ.html#StandaloneRemoteEJB

[10] JBoss http://stackoverflow.com/questions/14336478/jboss-7-jndi-lookup

[11] ActiveMQ http://activemq.apache.org/jndi-support.html