Universidad de los Andes

Mensajería
Java Message Service (JMS) 1.1
Message Driven Beans (MDBs)

Por: Rafael Gustavo Meneses M.Sc.

Departamento de Ingeniería de Sistemas y Computación
Especialización en Construcción de Software
Bogotá, COLOMBIA

Universidad de los Andes

Agenda

- Introducción
- Message-Oriented Middleware (MOM)
- Modelos de Mensajería
  - Point to Point (P2P)
  - Publish-Subscribe (pub-sub)
- Java Message Service (JMS)
- Message-Driven Beans (MDBs)
- Referencias

Universidad de los Andes

Agenda

- **Introducción**
- Message-Oriented Middleware (MOM)
- Modelos de Mensajería
  - Point to Point (P2P)
  - Publish-Subscribe (pub-sub)
- Java Message Service (JMS)
- Message-Driven Beans (MDBs)
- Referencias

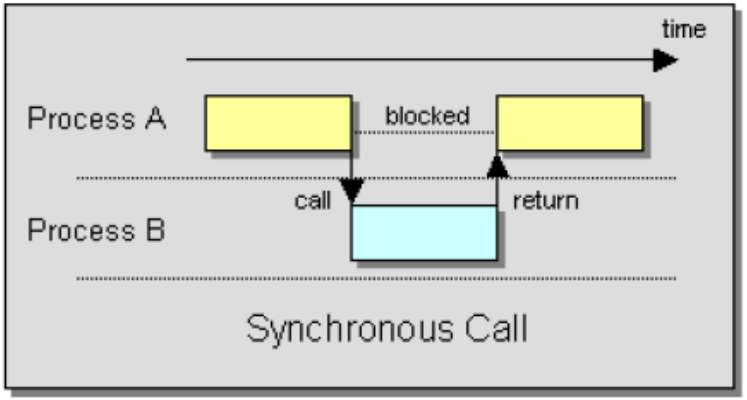
Mensajería

- En JEE el término mensajería se refiere a la comunicación con bajo acoplamiento entre un emisor y un receptor, de forma asíncrona

- Su comportamiento se puede asimilar a un buzón de voz, donde un emisor (*producer*) deja un mensaje en un destino específico, para que un receptor (*consumer*) lo recoja cuando pueda

Comunicación Síncrona

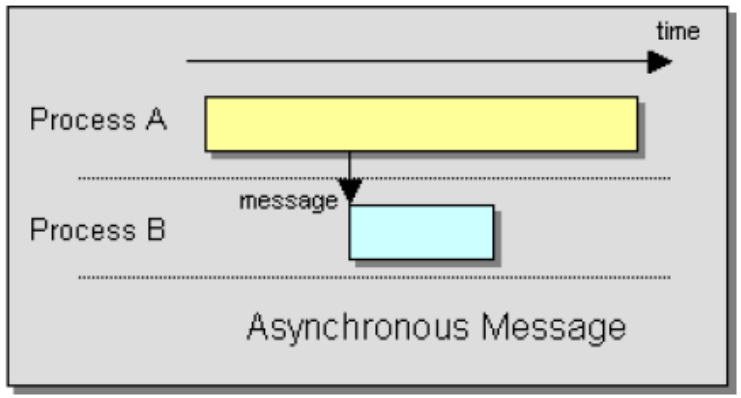
- Invocación de métodos → Java RMI
- *Quien invoca* y *quien es invocado* deben estar presentes para que la comunicación se realice
- El emisor debe esperar respuesta del receptor para realizar otra petición
- Ejemplo: Llamada telefónica



Tomado de [1]

Comunicación Asíncrona

- No se necesita la presencia del emisor y el receptor para realizar la comunicación
- El receptor responde en el momento que pueda
- Se requiere de un sistema de mensajería → *Message-Oriented Middleware - MOM*
- Ejemplo: Buzón de voz



Tomado de [1]

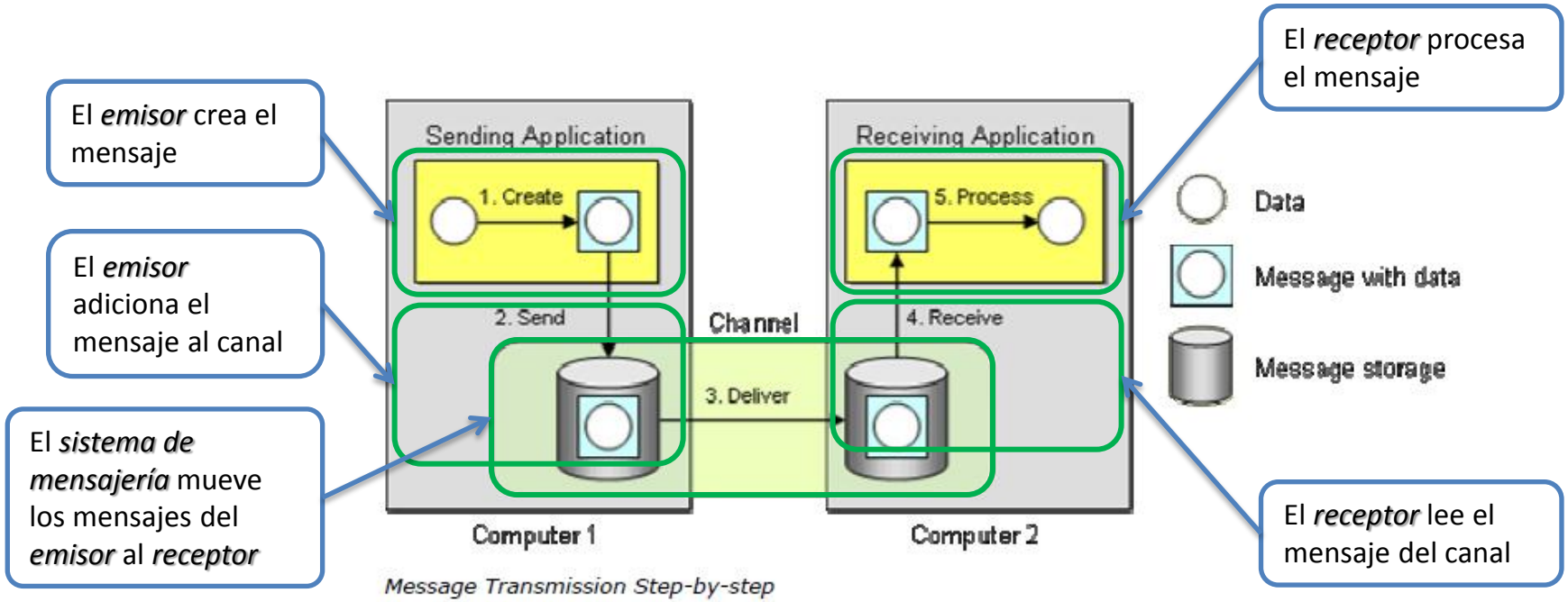
Universidad de los Andes

Agenda

- Introducción
- **Message-Oriented Middleware (MOM)**
- Modelos de Mensajería
  - Point to Point (P2P)
  - Publish-Subscribe (pub-sub)
- Java Message Service (JMS)
- Message-Driven Beans (MDBs)
- Referencias

Message-Oriented Middleware (MOM)

- Sistema de mensajería que coordina y administra el envío y recepción de mensajes.
- Su principal propósito es mover los mensajes del componentes emisor al componente receptor.

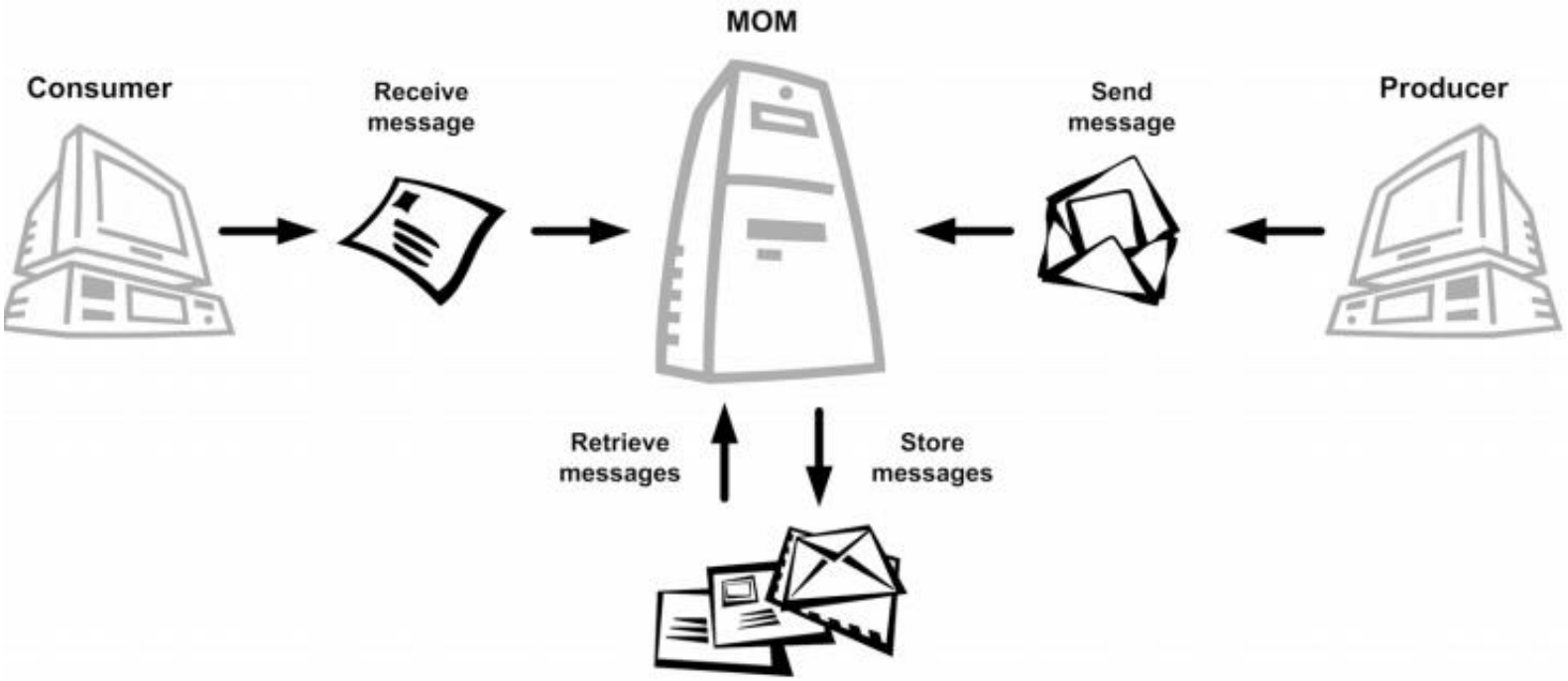


Tomado de [1]

Message-Oriented Middleware (MOM)

- Software que permite la comunicación asíncrona entre componentes
- Cuando un mensaje es enviado, el MOM lo almacena en una ubicación especificada por el emisor y reconoce lo recibido
- El MOM permite implementar soluciones en las que se requiere un *bajo acoplamiento* y *garantizar confiabilidad* en la entrega de información
- Existen diferentes productos libres y comerciales:
  - *IBM WebSphere MQ*
  - *TIBCO Rendezvous*
  - *SonicMQ*
  - *Apache ActiveMQ*
  - *OpenMQ*

Message-Oriented Middleware (MOM)



Tomado de [4]

Universidad de los Andes

Agenda

- Introducción
- Message-Oriented Middleware (MOM)
- **Modelos de Mensajería**
  - Point to Point (P2P)
  - Publish-Subscribe (pub-sub)
- Java Message Service (JMS)
- Message-Driven Beans (MDBs)
- Referencias

Universidad de los Andes

Agenda

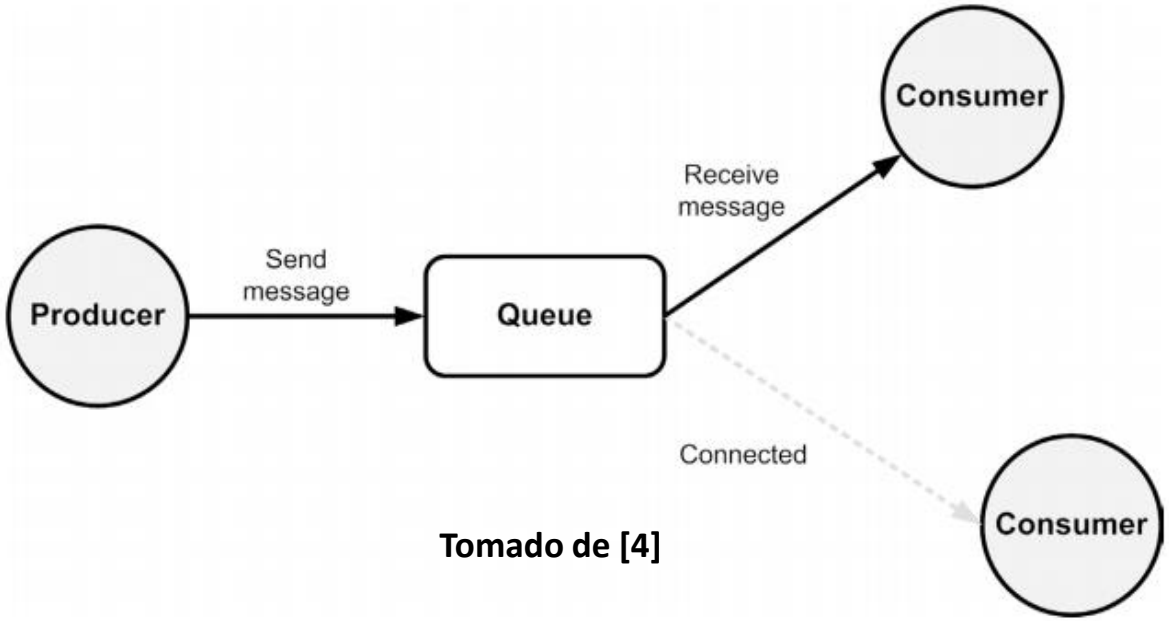
- Introducción
- Message-Oriented Middleware (MOM)
- **Modelos de Mensajería**
  - **Point to Point (P2P)**
  - Publish-Subscribe (pub-sub)
- Java Message Service (JMS)
- Message-Driven Beans (MDBs)
- Referencias

Point to Point (P2P)

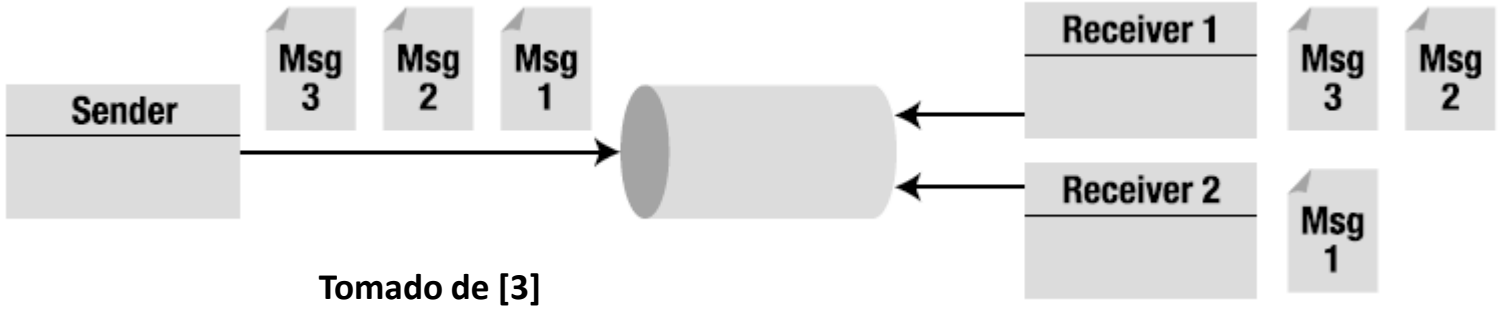
- El mensaje va de un único *emisor A* a un único *receptor B*
- El emisor y el receptor ***no*** tienen dependencias temporales:
  - El emisor *puede* producir y enviar mensajes en *cualquier momento*
  - El receptor *puede* consumirlos también en *cualquier momento*
- El mensaje queda almacenado en una cola
- La cola *no* garantiza el orden de entrega de los mensajes
- Las colas conservan todos los mensajes, hasta que son consumidos o expiran
- Si existe más de un receptor potencial se elige uno de forma aleatoria
- Cada mensaje es entregado a uno y sólo un receptor
- Una vez entregado el mensaje, desaparece de la cola

Modelos de Mensajería

Point to Point (P2P)



Tomado de [4]



Tomado de [3]

Universidad de los Andes

Agenda

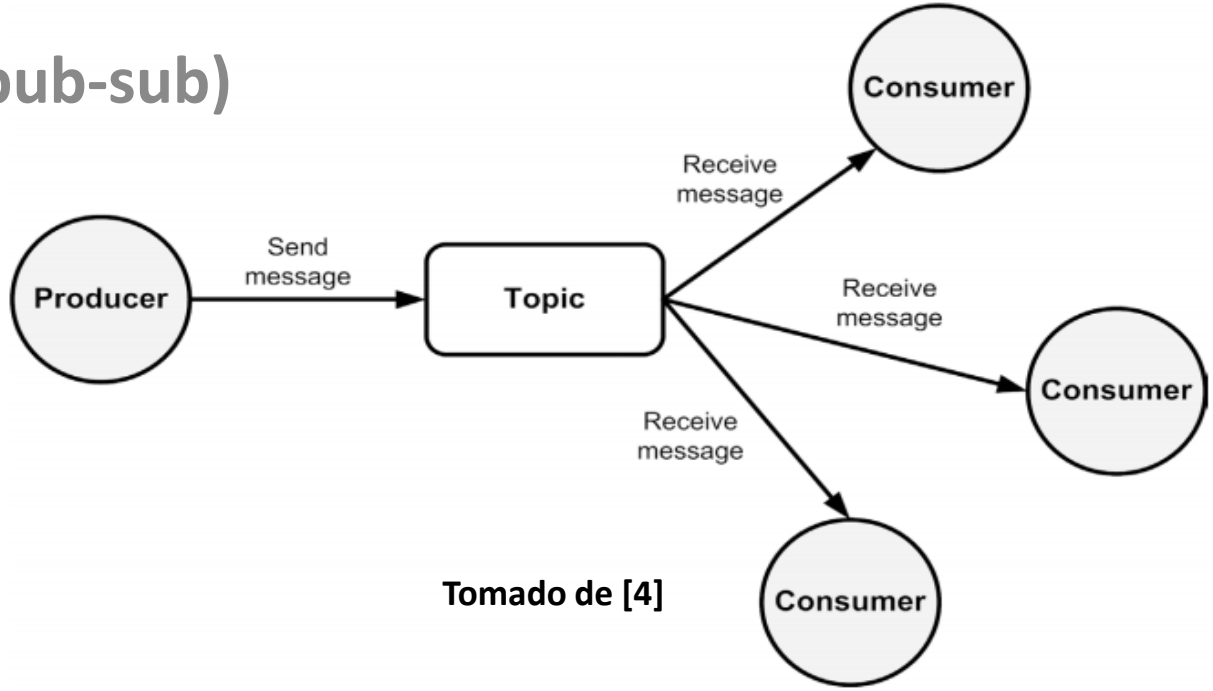
- Introducción
- Message-Oriented Middleware (MOM)
- **Modelos de Mensajería**
  - Point to Point (P2P)
  - **Publish-Subscribe (pub-sub)**
- Java Message Service (JMS)
- Message-Driven Beans (MDBs)
- Referencias

Publish-Suscribe (pub-sub)

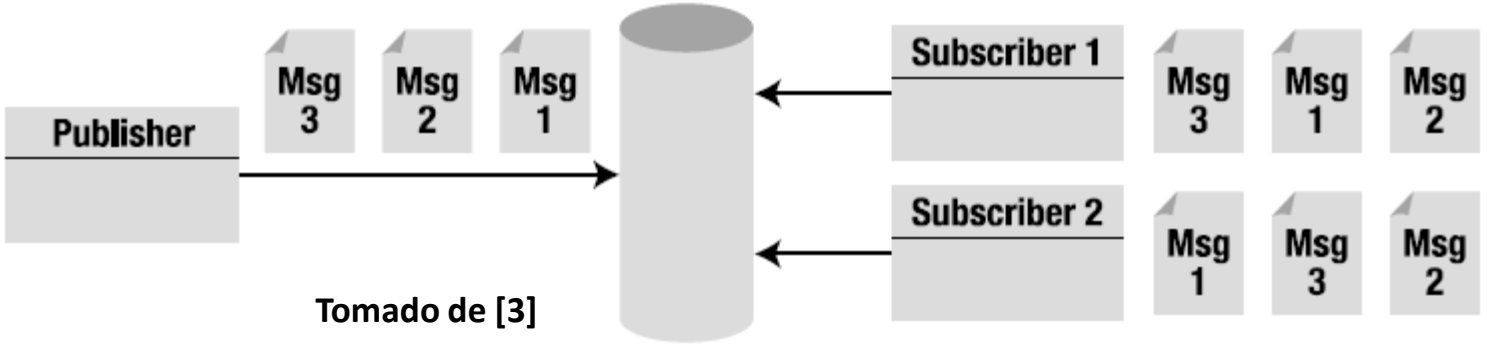
- El mensaje enviado es recibido por todos los consumidores suscritos
- Una vez enviado el mensaje no se guarda copia
- El productor y el(los) suscriptor(es) tienen dependencias temporales:
  - Los suscriptores *no* reciben los mensajes enviados previamente a su suscripción
  - Si un suscriptor está inactivo por un periodo de tiempo, *no* recibirá mensajes enviados hasta que vuelva a estar activo
- Los mensajes se tratan como un *Topic*, que funciona como una cola pero puede tener múltiples consumidores
- Utilizado en sistemas de broadcast

Modelos de Mensajería

Publish-Suscribe (pub-sub)



Tomado de [4]



Tomado de [3]

Universidad de los Andes

Agenda

- Introducción
- Message-Oriented Middleware (MOM)
- Modelos de Mensajería
  - Point to Point (P2P)
  - Publish-Subscribe (pub-sub)
- **Java Message Service (JMS)**
- Message-Driven Beans (MDBs)
- Referencias

Generalidades

- API de Java que permite comunicación distribuida con bajo acoplamiento, confiable y asíncrona
- Define un conjunto de interfaces y clases que permite a programas comunicarse con proveedores de mensajes
- Provee un API para manipular mensajes
  - *Crear*
  - *Enviar*
  - *Recibir*
  - *Leer*
- Provee un modelo estándar en Java para acceder a MOM
- Permite crear aplicaciones de mensajería portables
- JMS es análogo a JDBC. Los clientes JMS pueden usar el API JDBC

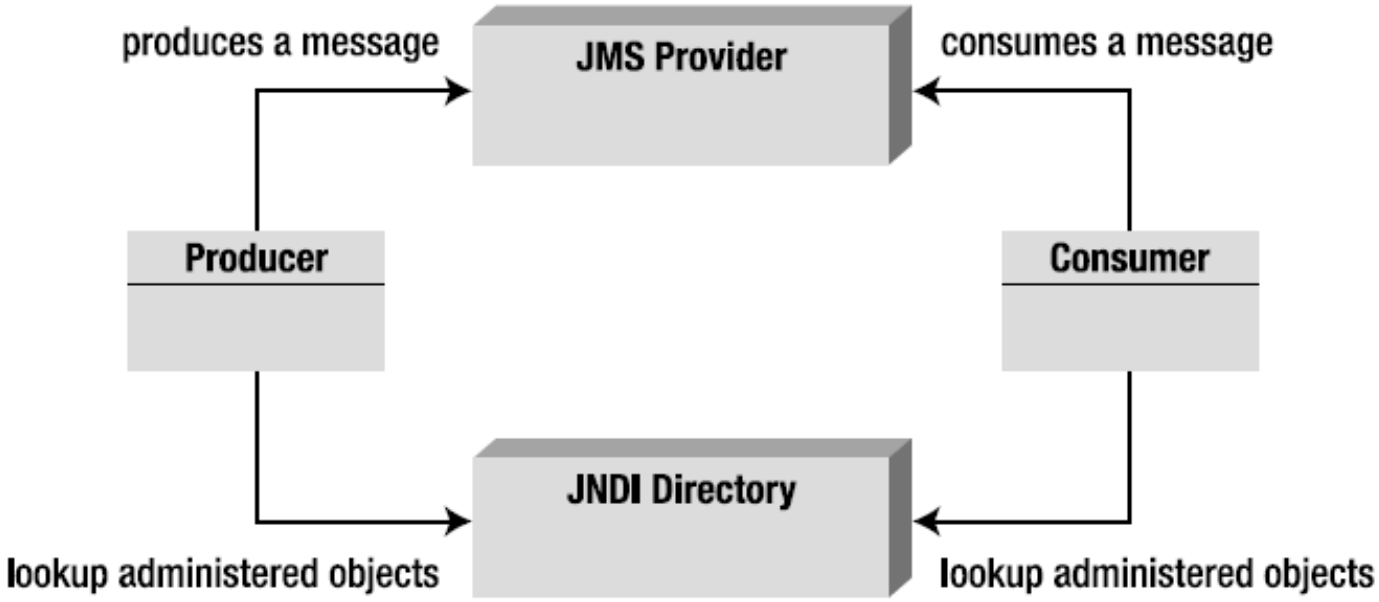
Generalidades (2)

- JMS no incluye:
  - Balanceo de carga y tolerancia a fallos
  - Administración → JMS no especifica un API para administración de mensajería
  - Seguridad → JMS no especifica un API para controlar la confidencialidad e integridad de los mensajes

Arquitectura de JMS

- **JMS Clients:** Programas en Java que envían y reciben mensajes JMS. Cliente es el término genérico para: *Productor*, *Emisor*, *Publicador*, *Consumidor*, *Receptor* o *Suscriptor*
- **Messages:** Conjunto de mensajes definidos en cada aplicación para comunicar información entre sus clientes
- **JMS Provider:** Sistema de mensajería que administra la recepción y entrega de mensajes → *Message Broker*
- **Administered Objects:** Objetos JMS pre-configurados, creados por un proveedor JMS para el uso de sus clientes

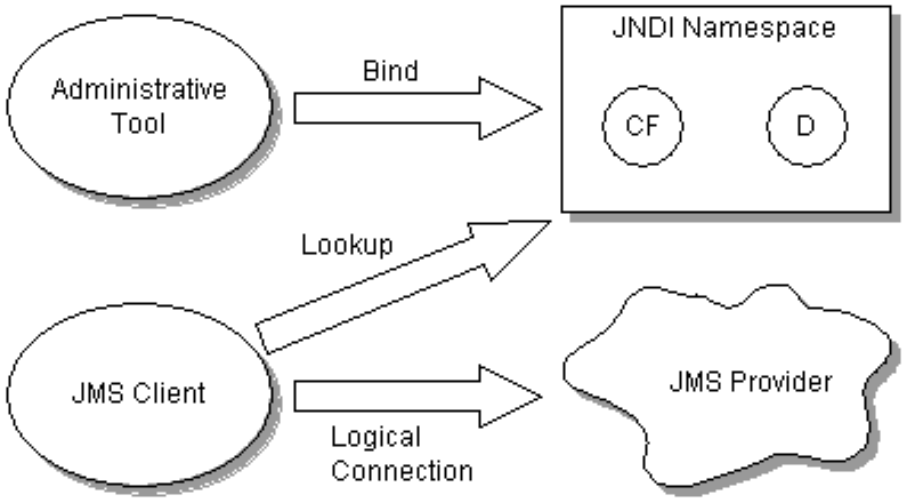
Arquitectura de JMS (2)



Tomado de [3]

Objetos Administrados

- JMS tiene dos tipos de objetos administrados:
  - **ConnectionFactory:** Objeto que usa un cliente para crear una conexión con un proveedor → `javax.jms.ConnectionFactory`
  - **Destination:** Objeto que un cliente usa para especificar el destino y el origen de los mensajes → `javax.jms.Destination`
- Estos objetos son guardados en un *namespace* de JNDI

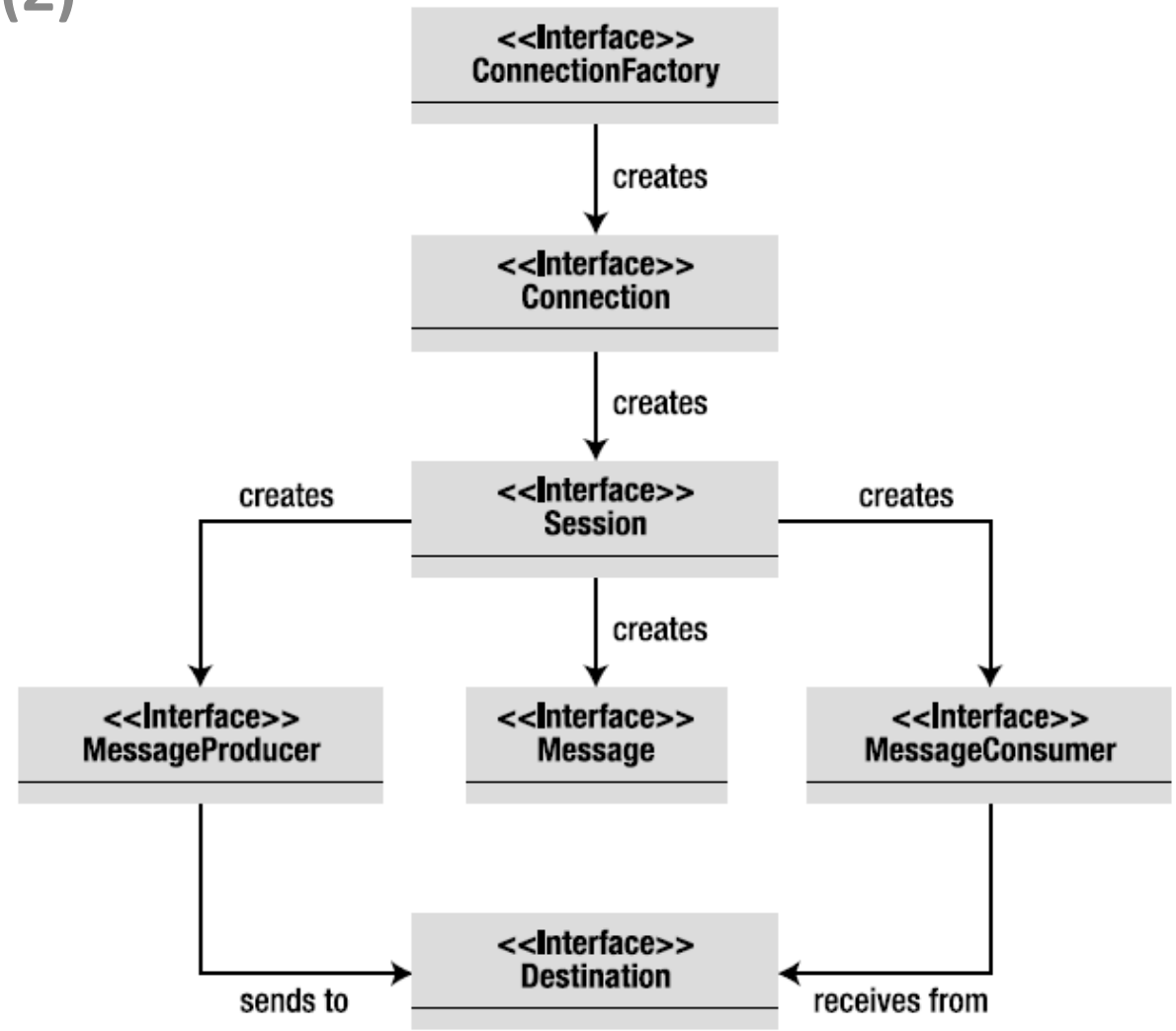


Tomado de [7]

API JMS

- Clases e interfaces del paquete **`javax.jms`**:
  - **`ConnectionFactory`** → Objeto *administrado* usado por un cliente para crear una conexión a un proveedor JMS
  - **`Connection`** → Conexión activa a un proveedor JMS
  - **`Session`** → Contexto transaccional único para enviar y recibir mensajes
  - **`Message`** → Objeto que encapsula la información a ser enviada a un destino
  - **`Destination`** → Objeto *administrado* que encapsula la identidad del destino del mensaje
  - **`MessageProducer`** → Objeto creado por un *Session* que es usado para enviar mensajes a un destino
  - **`MessageConsumer`** → Objeto creado por un *Session* que es usado para recibir mensajes de un destino

API JMS (2)



Tomado de [3]

Interfaces API JMS

- Interfaces del API a ser utilizadas dependiendo del modelo de mensajería (tipo de destino)

Generica	Point-to-Point	Publish-Suscribe
`Destination`	`Queue`	`Topic`
`ConnectionFactory`	`QueueConnectionFactory`	`TopicConnectionFactory`
`Connection`	`QueueConnection`	`TopicConnection`
`Session`	`QueueSession`	`TopicSession`
`MessageConsumer`	`QueueReceiver`	`TopicSuscriber`
`MessageProducer`	`QueueSender`	`TopicPublisher`

¿Cómo enviar y recibir mensajes?

- Un cliente JMS ejecuta las siguientes actividades:
  - Usa JNDI para localizar un objeto *ConnectionFactory*
  - Usa JNDI para encontrar uno o más objetos *Destination*
  - Usa el *ConnectionFactory* para crear un objeto *Connection*
  - Usa el *Connection* para crear una o más *Sessions*
  - Usa un *Session* y los *Destinations* para crear los objetos *MessageProducer* y *MessageConsumer* necesarios
  - Usa el *Connection* para iniciar la entrega de mensajes

Ejemplo uso API JMS – Envío de mensajes

```
public class Sender {

    public static void main(String[] args) {

        // Gets the JNDI context
        Context jndiContext = new InitialContext();

        // Looks up the administered objects
        ConnectionFactory connectionFactory = (ConnectionFactory)↪
                    jndiContext.lookup("jms/javaee6/ConnectionFactory");
        Queue queue = (Queue) jndiContext.lookup("jms/javaee6/Queue");

        // Creates the needed artifacts to connect to the queue
        Connection connection = connectionFactory.createConnection();
        Session session = connection.createSession(false, ↪
                                            Session.AUTO_ACKNOWLEDGE);
        MessageProducer producer = session.createProducer(queue);

        // Sends a text message to the queue
        TextMessage message = session.createTextMessage();
        message.setText("This is a text message sent at " + new Date());
        producer.send(message);

        connection.close();
    }
}
```

Tomado de [3]

Ejemplo uso API JMS – Recepción de mensajes

```
public class Receiver {

    public static void main(String[] args) {

        // Gets the JNDI context
        Context jndiContext = new InitialContext();

        // Looks up the administered objects
        ConnectionFactory connectionFactory = (ConnectionFactory) ↪
              jndiContext.lookup("jms/javaee6/ConnectionFactory");
        Queue queue = (Queue) jndiContext.lookup("jms/javaee6/Queue");

        // Creates the needed artifacts to connect to the queue
        Connection connection = connectionFactory.createConnection();
        Session session = connection.createSession(false, ↪
                                          Session.AUTO_ACKNOWLEDGE);
        MessageConsumer consumer = session.createConsumer(queue);
        connection.start();

        // Infinite loop to receive the messages
        while (true) {
            TextMessage message = (TextMessage) consumer.receive();
            System.out.println("Message received: " + message.getText());
        }
    }
}
```

Tomado de [3]

Ejemplo API JMS – Recepción de mensajes con inyección

```
public class Receiver {

    @Resource(lookup = "jms/javaee6/ConnectionFactory")
    private static ConnectionFactory connectionFactory;
    @Resource(lookup = "jms/javaee6/Queue")
    private static Queue queue;

    public static void main(String[] args) {

        // Creates the needed artifacts to connect to the queue
        Connection connection = connectionFactory.createConnection();
        Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        MessageConsumer consumer = session.createConsumer(queue);
        connection.start();

        // Loops to receive the messages
        while (true) {
            TextMessage message = (TextMessage) consumer.receive();
            System.out.println("Message received: " + message.getText());
        }
    }
}
```

Tomado de [3]

Ejemplo API JMS – Consumidor Asíncrono

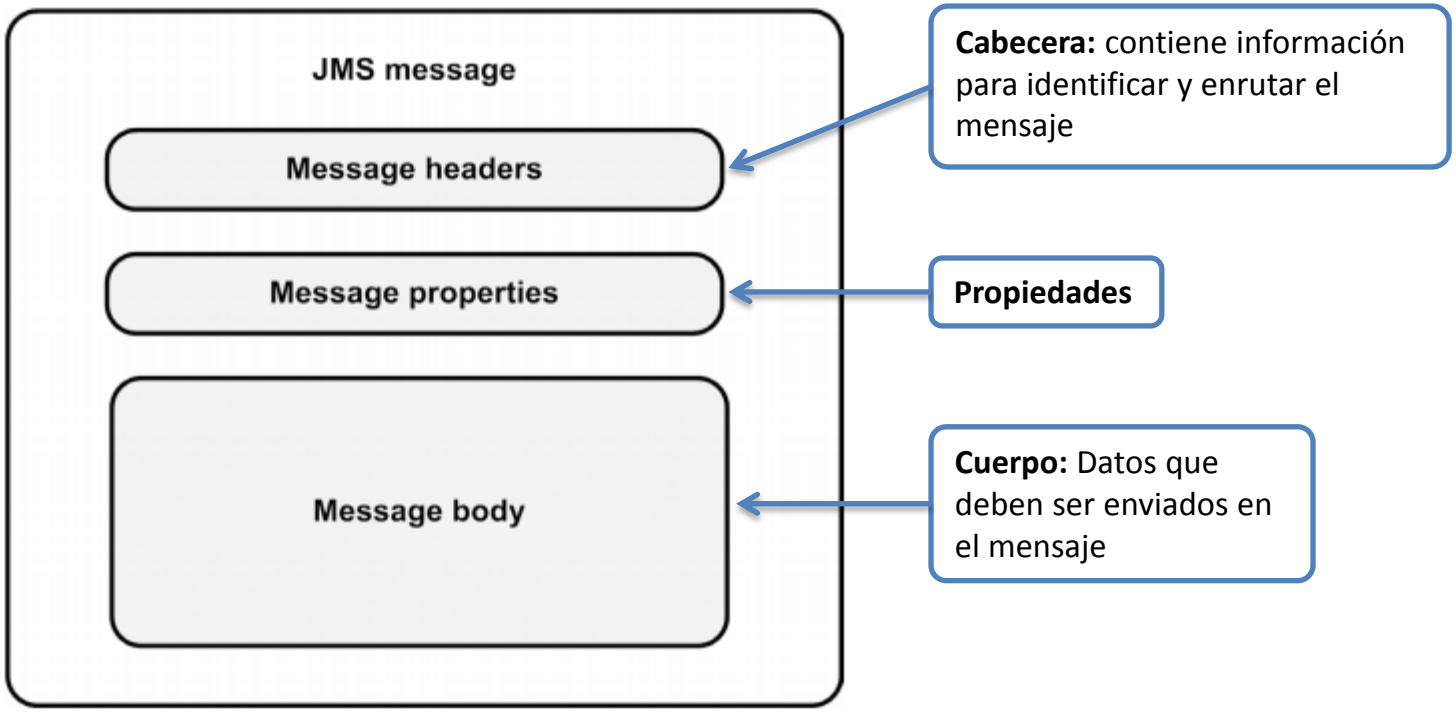
- El cliente debe crear un message listener que implementa la interfaz *MessageListener*
- El *MessageListener* es notificado asíncronamente cuando un mensaje es publicado en la cola
- En ese momento se ejecuta el método ***onMessage*** de la interfaz *MessageListener*

Ejemplo API JMS – Consumidor Asíncrono

```
public class Listener implements MessageListener {

    @Resource(lookup = "jms/javaee6/ConnectionFactory")
    private static ConnectionFactory connectionFactory;
    @Resource(lookup = "jms/javaee6/Topic")
    private static Topic topic;

    public static void main(String[] args) {

        // Creates the needed artifacts to connect to the queue
        Connection connection = connectionFactory.createConnection();
        Session session = connection.createSession(false, ➥
                                                Session.AUTO_ACKNOWLEDGE);
        MessageConsumer consumer = session.createConsumer(topic);
        consumer.setMessageListener(new Listener());
        connection.start();
    }

    public void onMessage(Message message) {
        System.out.println("Message received: " + ➥
                           ((TextMessage) message).getText());
    }
}
```

Tomado de [3]

Mensajes JMS



Mensajes JMS (2)

- **Header →** Todos los mensajes soportan el conjunto de campos de cabecera. Los campos de cabecera contienen valores usados por el cliente y el proveedor
- **Properties →** Se utiliza para información adicional a los campos de la cabecera
  - Application-specific properties
  - Standard properties
  - Provider-specific properties
- **Body →** Define varios tipos de mensajes

Mensajes JMS (3) - Header

Header Fields	Set by
`JMSDestination`	Send Method
`JMSDeliveryMode`	Send Method
`JMSExpiration`	Send Method
`JMSPriority`	Send Method
`JMSMessageID`	Send Method
`JMSTimestamp`	Send Method
`JMSCorrelationID`	Client
`JMSReplyTo`	Client
`JMSType`	Client
`JMSRedelivered`	Provider

Para más información consultar [7]

Mensajes JMS (4) - Properties

- Son campos similares a los de cabecera, pero definidos por la aplicación
- JMS ya tiene algunas propiedades por defecto* que pueden ser configuradas por la aplicación o por el proveedor → *JMSXUserID*
- Pueden ser valores de tipo **`boolean`**, **`byte`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**, **`String`**
- Ejemplo:

```
message.setFloatProperty("orderAmount", 1245.5f);
message.getFloatProperty("orderAmount");
```

*** Consultar en [7] las propiedades predefinidas en JMS**

Mensajes JMS (5) - Cuerpo

- JMS provee cinco (5) formas para el cuerpo de un mensaje, cada una definida por una interfaz de mensaje:

Forma de Mensaje	Descripción
`StreamMessage`	Contiene un conjunto de valores primitivos de Java. Se llena y lee secuencialmente
`MapMessage`	Contiene un conjunto de pares nombre-valor, los nombres son String y los valores tipos primitivos Java. Las entradas pueden ser accedidos secuencialmente o aleatoriamente por el nombre
`TextMessage`	Contiene un String
`ObjectMessage`	Contiene un objeto serializable Java
`BytesMessage`	Contiene un flujo de bytes sin interpretar

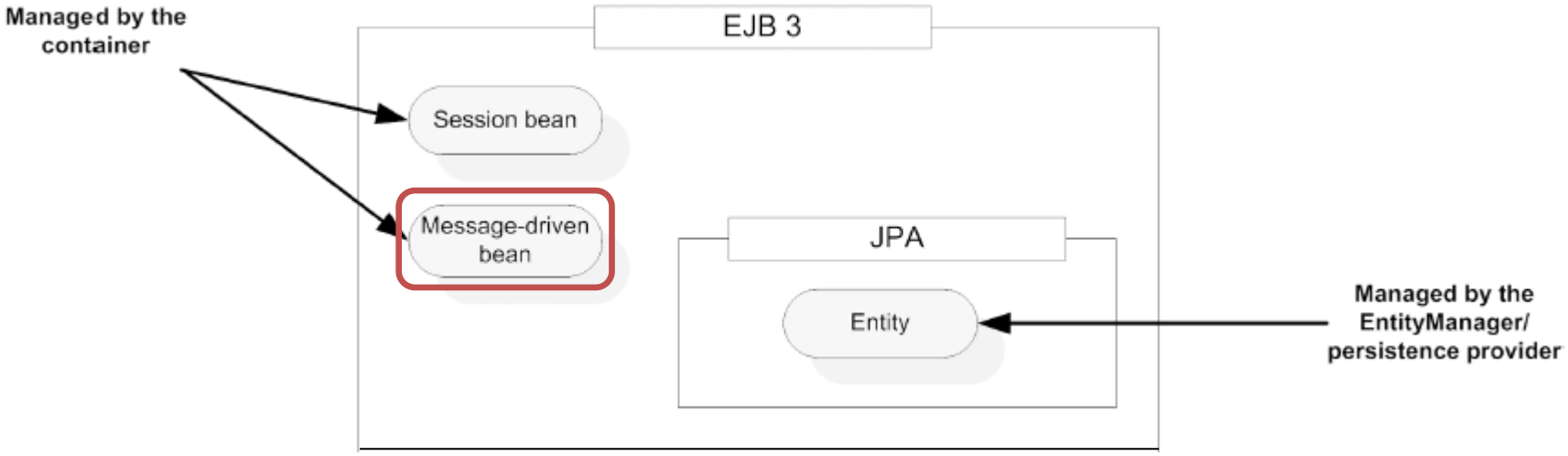
Agenda

- Introducción
- Message-Oriented Middleware (MOM)
- Modelos de Mensajería
  - Point to Point (P2P)
  - Publish-Subscribe (pub-sub)
- Java Message Service (JMS)
- **Message-Driven Beans (MDBs)**
- Referencias

Message-Driven Beans (MDBs)

Generalidades

- Es un componentes EJB que permite a las aplicaciones JEE procesar mensajes de forma asíncrona



Tomado de [4]

Características

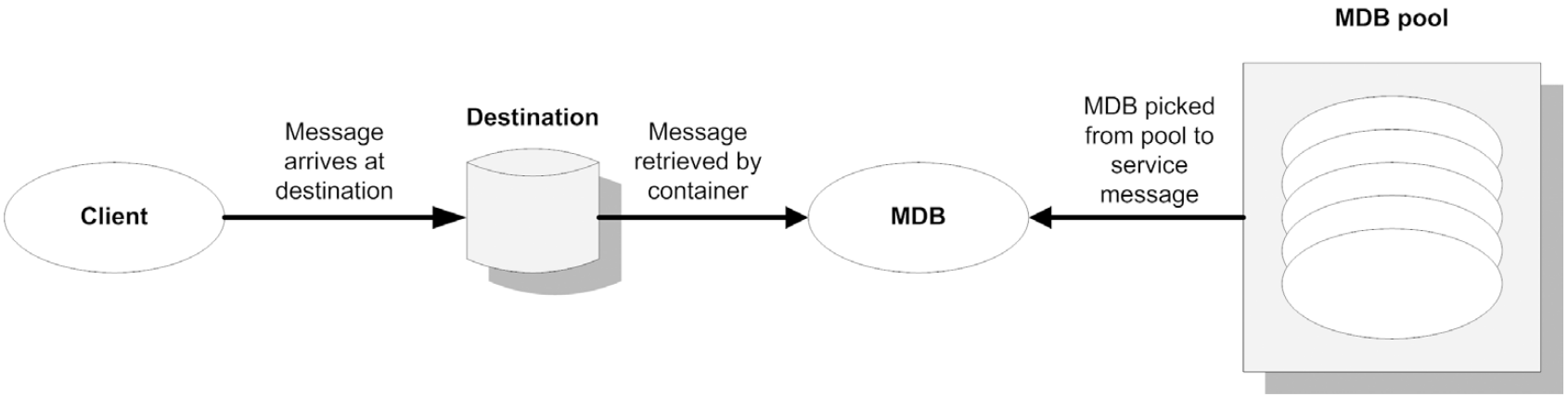
- Actúa como un *listener* de mensajes JMS
- Los mensajes pueden ser enviados por:
  - Cualquier componente JEE (aplicación cliente, otro EJB)
  - Una aplicación JMS
  - Un sistema que no utilice la tecnología JEE
- Son implementados con tecnología JMS
- Las conversaciones no mantienen el estado conversacional de las instancias de un cliente específico

Características (2)

- Todas las instancias de un MDB son equivalentes
- El contenedor EJB permite asignar un mensaje entrante a una instancia de un MBD
- El contenedor mantiene un pool de instancias y permite el flujo de mensajes para ser procesados concurrentemente
- Los MBDs son anónimos, ***no*** son identificados por el cliente
- Los componentes cliente ***no*** localizan los MDBs, ***ni*** invocan directamente los métodos del mismo
- Son invocados de forma asíncrona
- Un MDB puede procesar mensajes de múltiples clientes, pero solo de uno en un momento determinado

Proceso MDB

- Un MDB es invocado por el contenedor cuando un mensaje llega al *endpoint* que es expuesto por el MDB
- Un MDB es definido de acuerdo con la interfaz *MessageListener* utilizada
- Para un cliente, un MDB es simplemente un consumidor de mensajes que maneja el procesamiento de mensajes



Tomado de [4]

Beneficios de MDBs

¿Por qué utilizar MDBs cuando se pueden utilizar clientes JMS stand-alone?

Rta./ *Porque el contenedor, que administra multithreading, seguridad y transaccionalidad, simplifica en gran medida el código del consumidor JMS*

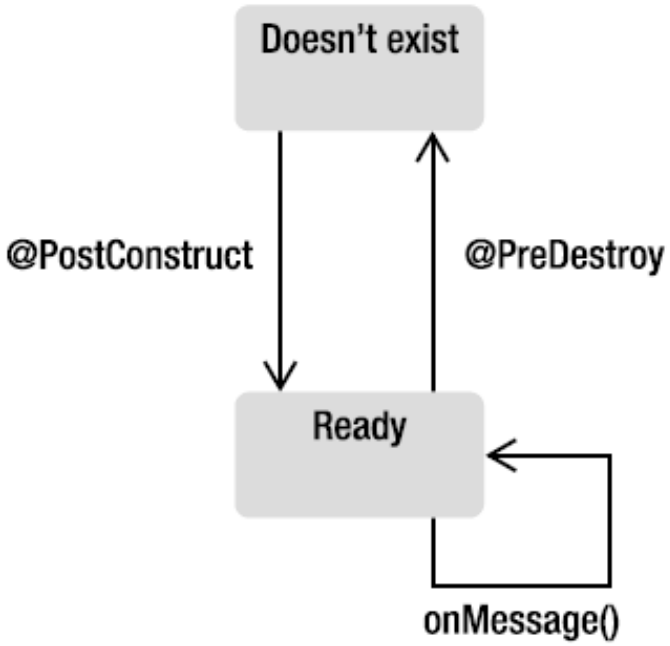
Beneficios de MDBs (2)

- Son manejados por un sistema de *pooling* que facilita el proceso de mensajes en paralelo al usar múltiples instancias de MDBs
- Gestionan el proceso de lectura de mensajes
- Automatizan el proceso de conectarse al proveedor de mensajes y leer los mensajes pendientes alojados allí

Ciclo de Vida

- El contenedor EJB mantiene un pool de instancias de MDBs
- Para cada instancia el contenedor ejecuta las siguientes tareas:
  - Si el MDB usa inyección de dependencias, el contenedor inyecta esta(s) referencia(s) antes de instanciarlo
  - El contenedor invoca el método anotado `@PostConstruct`, si lo hay
  - El MDB *nunca* es pasado a estado *passivated*. Solamente tiene dos estados: *nonexistent* and *ready* para recibir mensajes
- Al final del ciclo de vida, el contenedor invoca el método anotado `@PreDestroy`, si lo hay
- La instancia del EJB es procesada por el *Garbage Collector*

Ciclo de Vida (2)



Tomado de [3]

Message-Driven Beans (MDBs)

¿Cómo escribir un MDB?

```
@MessageDriven(mappedName = "jms/javaee6/Topic")
public class BillingMDB implements MessageListener {

    public void onMessage(Message message) {
        TextMessage msg = (TextMessage)message;
        System.out.println("Message received: " + msg.getText());
    }
}
```

Tomado de [3]

Modelo de Implementación de MDBs

- Los requerimientos para desarrollar un MDB son:
  - La clase desde estar anotada con **`@MessageDriven`** o su equivalente en un descriptor XML
  - La clase debe implementar, directa o indirectamente, la interfaz **`MessageListener`**
  - La clase debe estar definida como pública y no debe ser final o abstracta.
  - La clase debe tener un constructor sin argumentos
  - La clase no debe definir el método **`finalize()`**

Modelo de Implementación de MDBs - Anotaciones

- Un MDB implementa una interfaz *MessageListener* de JMS
- El contenedor usa la interfaz listener para registrar el MDB con el proveedor de mensajes y transmitir los mensajes invocando los métodos implementados
- Usando *MessageListener* como parámetro de la anotación **`@MessageDriven`**:

```
@MessageDriven(
   name = "ShippingRequestJMSProcessor",
   messageListenerInterface = "javax.jms.MessageListener")
public class ShippingRequestProcessorMDB {
```

- Realizando la implementación directamente:

```
public class ShippingRequestProcessorMDB implements MessageListener {
```

Modelo de Implementación de MDBs - Anotaciones

- **`@MessageDriven`**

```
@Target(TYPE) @Retention(RUNTIME)
public @interface MessageDriven {
    String name() default "";
    Class messageListenerInterface default Object.class;
    ActivationConfigProperty[] activationConfig() default {};
    String mappedName();
    String description();
}
```

Modelo de Implementación de MDBs - Anotaciones

- **`@ActivationConfigProperty`**

```
@Target({}) @Retention(RUNTIME)
public @interface ActivationConfigProperty {
    String propertyName();
    String propertyValue();
}
```

Activation Configuration Properties

Propiedad	Descripción
`destinationType`	El tipo de destino, **TOPIC** o **QUEUE**
`destinationName`	El nombre del destino
`messageSelector`	El selector del mensaje utilizado por el MDB
`acknowledgeMode`	El modo de reconocimiento, por defecto **AUTO_ACKNOWLEDGE**
`subscriptionDurability`	La durabilidad de la suscripción **TOPIC**, por defecto **NON_DURABLE**
`connectionFactoryJndiName`	Namespace JNDI en donde están alojados los Objetos Administrados

Activation Configuration Properties (2)

```
@MessageDriven(mappedName = "jms/javaee6/Topic", activationConfig = {
    @ActivationConfigProperty(propertyName = "acknowledgeMode", ➥
                              propertyValue = "Auto-acknowledge"),
    @ActivationConfigProperty(propertyName = "messageSelector", ➥
                              propertyValue = "orderAmount < 3000")
})
public class BillingMDB implements MessageListener {

    public void onMessage(Message message) {
        TextMessage msg = (TextMessage)message;
        System.out.println("Message received: " + msg.getText());
    }
}
```

Tomado de [3]

Activation Configuration Properties (3)

- Uso de **`@ActivationConfigProperty`** → *P*rovee información para la configuración de un sistema de mensajería a través de un arreglo de instancias de *ActivationConfigProperty*

```
@MessageDriven(
    name="ShippingRequestProcessor",
    activationConfig = {
        @ActivationConfigProperty(
            propertyName="destinationType",
            propertyValue="javax.jms.Queue"),
        @ActivationConfigProperty(
            propertyName="connectionFactoryJndiName",
            propertyValue="jms/QueueConnectionFactory"
        ),
        @ActivationConfigProperty(
            propertyName="destinationName",
            propertyValue="jms/ShippingRequestQueue")
    }
)
```

Comunica al contenedor que este MDB está escuchando por una cola. Si escucha un Topic el valor es **`javax.jms.Topic`**

Especifica el nombre JNDI de la fábrica de conexión que debe ser usada para crear las conexiones JMS para el MDB

Estamos escuchando los mensajes que llegan al destino con el nombre JNDI *jms / ShippingRequestQueue*

Activation Configuration Properties (4)

- **`acknowledgeMode`**
  - Los mensajes no son removidos de la cola hasta que el consumidos los *reconoce*
  - El modo de reconocimiento por defecto en una sesión JMS es **`AUTO_ACKNOWLEDGE`** → El mensaje se reconoce tan pronto es recibido
  - **`DUPS_OK_ACKNOWLEDGE`** → El contenedor EJB envía la confirmación en cualquier momento luego de la recepción del mensaje
  - Con CMT:
    - Si la transacción es *exitosa*, el contenedor EJB enviará un mensaje de reconocimiento
    - Si la transacción *falla*, el mensaje no es reconocido, es reubicado en la cola y posteriormente reenviado

Activation Configuration Properties (5)

- **`suscriptionDurability`**
  - Sólo aplica para el modelo *pub-sub*
  - Se puede definir si la suscripción a un Topic es ***durable*** o no
  - Si un suscriptor ***no*** esta conectado y registrado a un Topic en el momento en que un mensaje es enviado, ***no*** podrá leer dicho mensaje → ***non-durable***
  - Si falla el contenedor EJB una vez desplegado el MDB, éste pierde la conexión con el proveedor JMS y sólo podrá leer los mensajes entrantes una vez el contenedor se reinicie → ***non-durable***

```
@ActivationConfigProperty(propertyName =
                                    "subscriptionDurability",
                          propertyValue = "Durable")
```

Manejo de Transacciones

- MDBs pueden usar *CMTs* o *BMTs*
- MDBs pueden marcar una transacción para hacer rollback por medio del método `MessageDrivenContext.setRollbackOnly()`
- Una transacción en un MDB *no* puede ser ejecutada en el contexto transaccional del cliente emisor de un mensaje
- En CMT, los atributos de transacción permitidos son `REQUIRED` y `NOT_SUPPORTED`
- En CMT, por defecto el contenedor:
  - Inicia una transacción antes de invocar el método `onMessage(Message msg)`
  - Hace *commit* de la transacción cuando el método termina su ejecución, a menos que la transacción sea marcada para hacer *rollback*

Message-Driven Beans (MDBs)

Inyección de Dependencias

```
@PersistenceContext
private EntityManager em;
@EJB
private InvoiceBean invoice;
@Resource(lookup = "jms/javaee6/ConnectionFactory")
private ConnectionFactory connectionFactory;
```

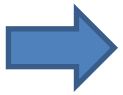
- El contexto MDB tambien puede ser inyectado utilizando la anotación **`@Resource`**

```
@Resource private MessageDrivenContext context;
```

Message-Driven Beans (MDBs)

Ejemplo – MDB como Productor (1)

```
@MessageDriven(mappedName = "jms/javaee6/Topic", activationConfig = {
        @ActivationConfigProperty(propertyName = "acknowledgeMode", ➥
                                  propertyValue = "Auto-acknowledge"),
        @ActivationConfigProperty(propertyName = "messageSelector", ➥
                                  propertyValue = "orderAmount < 3000")
})
public class BillingMDB implements MessageListener {

    @Resource(lookup = "jms/javaee6/Queue")
    private Destination printingQueue;
    @Resource(lookup = "jms/javaee6/ConnectionFactory")
    private ConnectionFactory connectionFactory;
    private Connection connection;

    @PostConstruct
    private void initConnection() {
        connection = connectionFactory.createConnection();
    }
```



Tomado de [3]

Ejemplo – MDB como Productor (2)

```
    @PreDestroy
    private void closeConnection() {
        connection.close();
    }

    public void onMessage(Message message) {
        TextMessage msg = (TextMessage)message;
        System.out.println("Message received: " + msg.getText());
        sendPrintingMessage();
    }

    private void sendPrintingMessage() throws JMSException {
        Session session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
        MessageProducer producer = session.createProducer(printingQueue);
        TextMessage message = session.createTextMessage();
        message.setText("This message has been received and sent again");
        producer.send(message);
        session.close();
    }
}
```

Tomado de [3]

Buenas prácticas

- Escoja su modelo de mensajería cuidadosamente: P2P o pub-sub
- Recuerde modularizar → Es necesario modularizar y desacoplar teniendo presente el *concern* de mensajería
- Haga buen uso de los filtros de mensajes
- Escoja los tipos de mensajes cuidadosamente
- Configure el tamaño del pool de MDB

Universidad de los Andes

Agenda

- Introducción
- Message-Oriented Middleware (MOM)
- Modelos de Mensajería
  - Point to Point (P2P)
  - Publish-Subscribe (pub-sub)
- Java Message Service (JMS)
- Message-Driven Beans (MDBs)
- **Referencias**

Referencias

1. **Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.** Gregor Hohpe, Bobby Woolf. 2004.
2. **The Java™ EE 6 Tutorial.** Eric Jendrock. Oracle Corporation. 2011.
3. **Beginning Java™ EE 6 Platform with GlassFish™ 3**, Second Edition. Antonio Goncalves. 2010.
4. **EJB 3 in Action.** Panda Debu, Rahman Reza, Lane Derek. Manning. 2007.
5. **EJB 3 Developer Guide.** Michael Sikora. 2008.
6. **Mastering Enterprise JavaBeans™ 3,0**. Rima Patel Sriganesh, Gerald Brose, Micah Silverman. Wiley Publising, Inc. 2006.
7. **Java™ Message Service Specification, Versión 1.1.** April 12, 2002.

Universidad de los Andes

Gracias !!!

Rafael Meneses

rg.meneses81@uniandes.edu.co