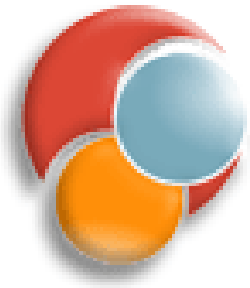


Servicios de Mensajes con JMS

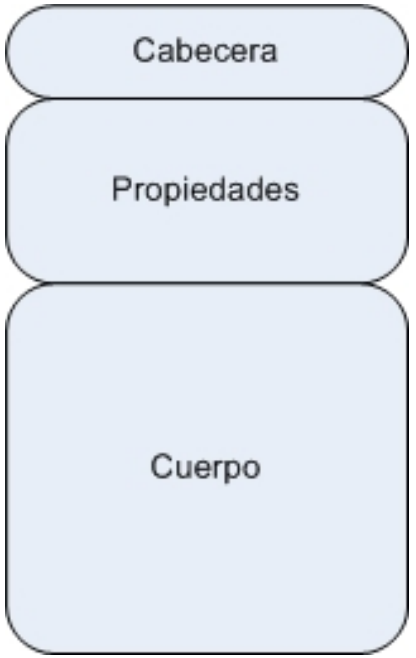
Sesión 2: Mensajes.
Robustez en JMS
Subscripciones Duraderas

Puntos a tratar

- Mensaje
  - Cabecera
  - Propiedades
  - Cuerpo / Tipos de Mensaje
- Browser de Mensajes
- JMS Robusto
- Llamadas Síncronas con Request/Reply
- Subscripciones Duraderas

Mensaje

- Concepto más importante de JMS
- Al haber múltiples MOMs, existen múltiples formatos de mensaje
- JMS define un modelo de mensajes unificado y abstracto
- Los mensajes implementan el interfaz `javax.jms.Message`
  - Divide al mensaje en tres partes



Cabecera (I)

- Todos los mensajes JMS soportan la misma lista estándar de cabeceras

- Cabeceras rellenadas automáticamente por el proveedor:
  - **JMSDestination**: destino al que se envía el mensaje
  - **JMSDeliveryMode:** persistente (1) / no-persistente (0,1)
  - **JMSExpiration**: previene la entrega de mensajes expirados
  - **JMSMessageID:** identificador unívoco del mensaje
  - **JMSPriority**: nivel de importancia (0 poca – 9 mucha)
  - **JMSTimestamp:** instante de envío

Cabecera (II)

- Cabeceras asignadas de forma opcional por el cliente:
  - **JMSCorrelationID:**- asocia el mensaje actual con el anterior.
  - **JMSReplyTo:** destino al cual debería responder el cliente
  - **JMSType** - identifica semánticamente al mensaje (tipo del cuerpo)

    Esta cabecera la utilizan muy pocos proveedores.

- Cabeceras asignadas de forma opcional por el proveedor:
  - **JMSRedelivered:** indica que un mensaje ha sido reenviado.

    Esto puede suceder si un consumidor falla en el acuse de recibo o si el proveedor JMS no ha sido notificado del envío.

Propiedades

- Son cabeceras adicionales que pueden especificarse en un mensaje.
- Al contrario que las cabeceras, son opcionales.
- Son pares de `{nombre, valor}`, y JMS ofrece métodos para trabajar con propiedades cuyo tipo sea `boolean, byte, short, int, long, float, double, o String.`
- Las propiedades se inicializan cuando se envía un mensaje, y al llegar al destino pasan a ser de sólo lectura.
- Existen tres tipos de propiedades

1. **Propiedades de Aplicación**
   - Son arbitrarias y las define una aplicación JMS.
   - Los desarrolladores de aplicaciones JMS puede definir libremente cualquier propiedad que necesiten

     Ej: `message.setIntProperty("Anyo", 2008);`

Propiedades (II)

1. **Propiedades Definidas por JMS**
   - JMS reserva el prefijo 'JMSX' para las propiedades definidas por JMS.
   - El soporte de estas propiedades es opcional:
     JMSXAppID - Identifica la aplicación que envía el mensaje.
     JMSXConsumerTXID - Id de la transacción en la que se ha consumido el msj.
     JMSXDeliveryCount - Número de intentos de entrega del mensaje.
     JMSXGroupID - Grupo del mensaje.
     JMSXGroupSeq - Número de secuencia dentro del grupo.
     JMSXProducerTXID – Id de la transacción en la que se ha producido el msj.
     JMSXRcvTimestamp - Instante de entrega del mensaje por el proveedor.
     JMSXState - Define un estado específico del proveedor.
     JMSXUserID - Identifica al usuario que envía el mensaje.
   - JMS recomienda que las propiedades *JMSXGroupID* y *JMSXGroupSeq* deberían utilizarse por los clientes cuando se agrupen mensajes.

2. **Propiedades Específicas del Proveedor**
   - JMS reserva el prefijo 'JMS_vendor-name' para éstas propiedades.
   - Cada proveedor define sus propios valores para sus propiedades.
     En su mayoría las utilizan los clientes no-JMS, atados al proveedor
     No deberían utilizarse entre mensajerías JMS.

Cuerpo y Tipos

- Permite enviar y recibir datos e información con diferentes formatos, ofreciendo compatibilidad con los formatos de mensaje existentes.
- También se conoce como **carga** (*payload*).
- JMS define seis tipos de cuerpo:
  - **Message** - Mensaje base. Es un mensaje sin cuerpo, solo cabeceras y propiedades. Uso: notificación simple de eventos.
  - **MapMessage** - Compuesto de un conjunto de pares `{nombre,valor}`. El tipo de los nombres es `String`, y los valores tipos primitivos Java.
  - **BytesMessage** - Contiene un array de bytes. Uso: hacer coincidir el cuerpo con un formato de mensaje existente (*legacy*).
  - **StreamMessage** - El cuerpo es un flujo de tipos primitivos Java, cuya lectura y escritura se realiza de modo secuencial.
  - **TextMessage** – Se trata de un `String`. Uso: para enviar texto simple y datos XML.
  - **ObjectMessage** - La carga es un objeto Java `Serializable`. Uso: para trabajar con objetos Java complejos.

Creando y Recibiendo Mensajes

- Para crear un mensaje, a partir de una `Session`, mediante los métodos `createXXXMessage`, siendo XXX el tipo de mensaje.

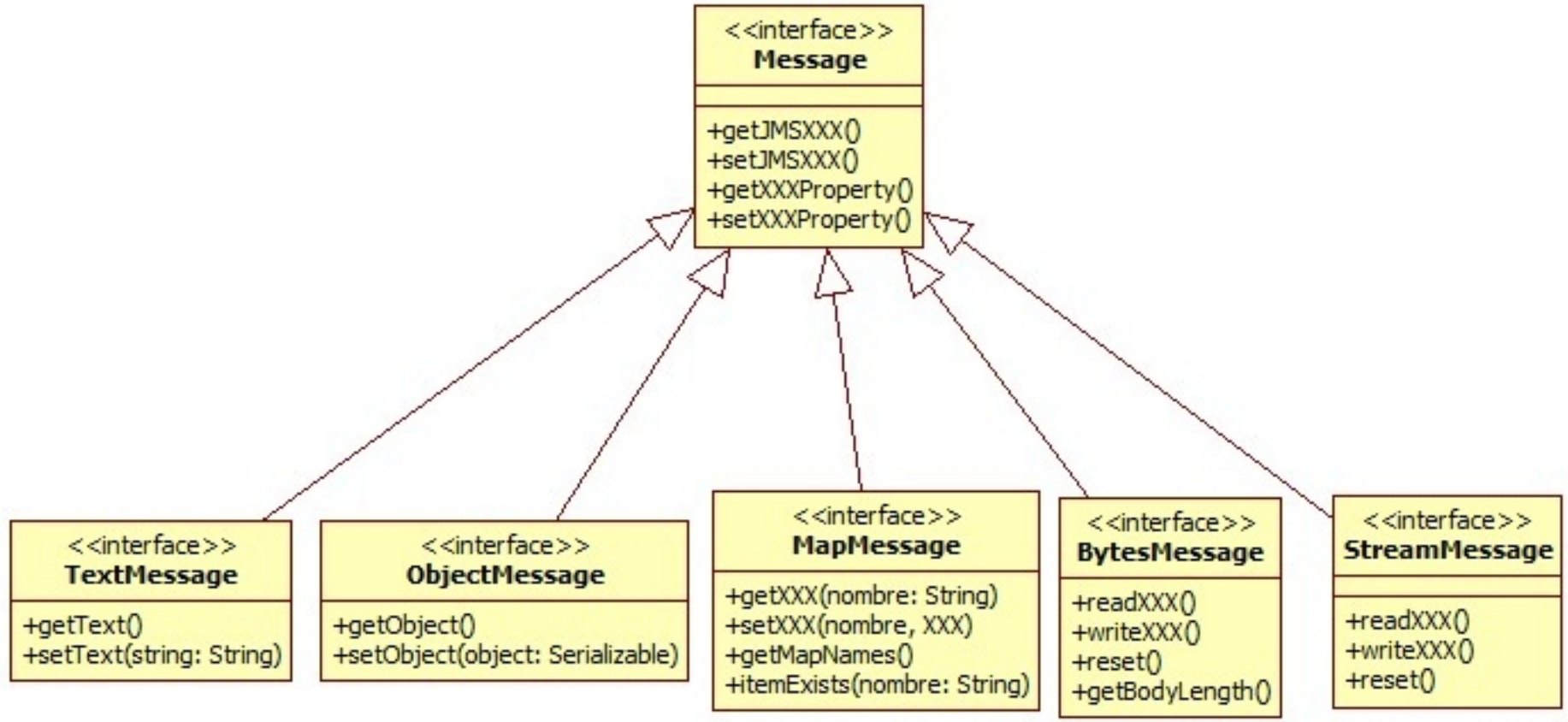
```
TextMessage mensaje = session.createTextMessage();
mensaje.setText(miTexto);
producer.send(mensaje);
```

```
ObjectMessage mensaje = session.createObjectMessage();
mensaje.setObject(miLibroEntity);
// el objeto miLibroEntity debe ser Serializable !!!!
```

- En el destino, el mensaje llega como un `Message` genérico, y debemos hacer el *cast* al tipo de mensaje apropiado.
- Cada tipo de mensaje ofrece métodos para extraer el contenido.

```
Message m = consumer.receive();
if (m instanceof TextMessage) {
   TextMessage mensaje = (TextMessage) m;
   System.out.println("Mensaje: " + mensaje.getText());
} else { // Manejar el error }
```

Tipos y Métodos de Mensaje



Selector de Mensajes

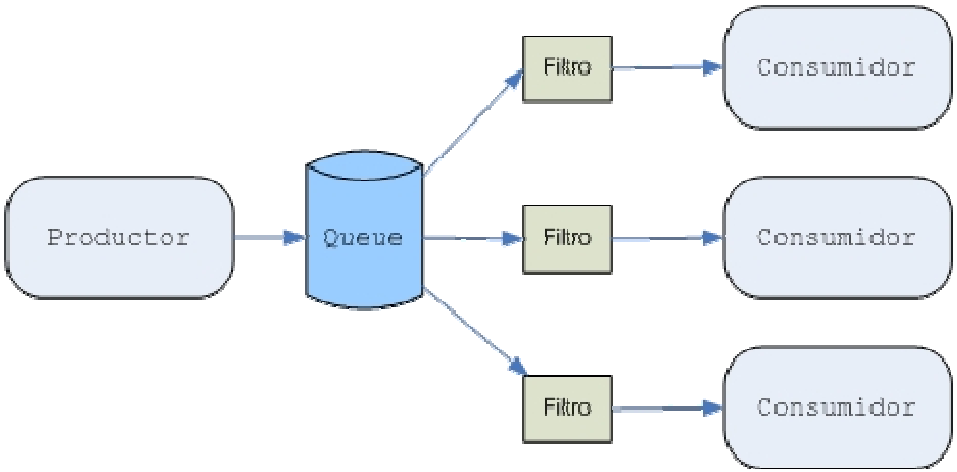
- Filtra los mensajes que se reciben a través del uso de las cabeceras y propiedades
- Los selectores **no pueden referenciar al cuerpo** del mensaje
- Expresión subconjunto de las condicionales de SQL92
  - Los operandos son las cabeceras/propiedades.
  - Operadores adecuados: operador lógico, `between`, `like`, `in(...)`, `is null`, ...

    `Anyo = 2008, Mes = 'Diciembre', Mes LIKE '%BRE', Anyo BETWEEN 2000 AND 2008`
- Más información: java.sun.com/javaee/5/docs/api/javax/jms/Message.html
- Se utiliza al crear el consumidor de mensajes mediante `Session.createConsumer(Destination dest, String selectorMensaje)`
  - Si el selector no es correcto: `javax.jms.InvalidSelectorException`.
  - Podemos consultar el selector de un consumidor con `getMessageSelector()`

```
MessageConsumer consumer = session.createConsumer(queue, "Anyo = 2008");
String selectorAnyo2008 = consumer.getMessageSelector();
```

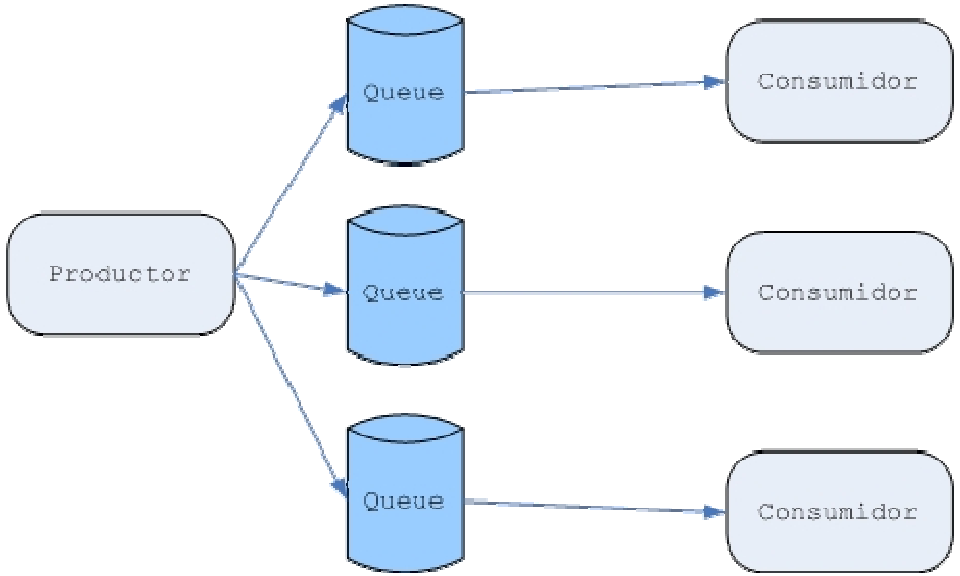
- Los mensajes que no cumplen el selector no se entregan.
- Una vez se establece el selector de mensajes, no puede cambiarse.
  - Tenemos que cerrar el consumidor y crear un nuevo consumidor con su nuevo selector de mensajes.

Selectores vs Múltiples Destinos



- Enfoque "**Filtrado de Mensajes**"
- El control lo tiene el consumidor

- Enfoque "**Múltiples Destinos**"
- El control lo tiene el productor
- El filtrado se realiza antes de enviar el mensaje (en Java)



Browser de Mensajes

- Permite consultar el contenido de la **cola** (sólo colas) sin *consumir los mensajes*
- Utilizaremos el método `createBrowser` de la `Session`, indicándole la cola a inspeccionar:

```
QueueBrowser browser = session.createBrowser(queue);
```

- B*rowser* que filtra los mensajes con selector:

```
QueueBrowser browser = session.createBrowser(queue,
                                "provincia = 'Alicante'");
```

Uso del *Browser*

- Métodos de la interfaz `QueueBrowser`:
  - `getEnumeration()`: obtiene los mensajes
  - `getMessageSelector()`: String que actúa como filtro de mensajes (si hay alguno para ese consumidor)
  - `getQueue()`: cola asociada al browser.

```
Enumeration iter = browser.getEnumeration();
if (!iter.hasMoreElements()) {
        System.out.println("No hay mensajes en la cola");
} else {
        while (iter.hasMoreElements()) {
                Message tempMsg = (Message) iter.nextElement();
                System.out.println("Mensaje: " + tempMsg);
        }
}
```

JMS Robusto

- JMS garantiza la robustez
  - y (si se puede) la eficiencia de las aplicaciones distribuidas.
  - No permite mensajes que no llegan o bien llegan duplicados

JMS garantiza que:
  - El mensaje será recibido
  - Y solamente será recibido una vez
- Esto implica que el servidor de aplicaciones dote a JMS con una capa de persistencia.

- La forma más fiable de **producir** un mensaje es enviar un mensaje **PERSISTENT** dentro de una transacción.
- La forma más robusta de **consumir** un mensaje es dentro de una **transacción** y a partir de una cola o de un *Durable Subscriber* en un tópico.

Control del Acuse de Recibo

- Hasta que no se recibe un acuse de recibo de un mensaje, éste no se considera consumido exitosamente.
- La consumición exitosa de un mensaje se lleva a cargo en tres fases:
  - El cliente recibe el mensaje
  - El cliente procesa mensaje
  - Se acusa el recibo del mensaje. Este acuse se inicia por parte del proveedor JMS o por el cliente, dependiendo del modo de acuse de la sesión.

- En las sesiones **transaccionales** (mediante las transacciones locales), el acuse ocurre automáticamente cuando se hace el *commit* de la transacción.
- Si se produce un *rollback* de la transacción, todo los mensajes consumidos se vuelven a re-entregar.

Control del Acuse de Recibo (II)

- En las sesiones **no transaccionales**, hay 3 posibilidades:
- **Session.AUTO_ACKNOWLEDGE**: Acuse automático si el cliente ha retornado exitosamente tanto de `MessageConsumer.receive()` como del listener.
  - Una recepción síncrona en una sesión AUTO_ACKNOWLEDGE es la excepción a la consumición en tres fases, ya que la recepción y el acuse tienen lugar en un único paso, seguido por el procesamiento del mensaje.
- **Session.CLIENT_ACKNOWLEDGE**: El cliente envía el acuse mediante la llamada al método `acknowledge()` del mensaje.
  - El acuse se realiza a nivel de sesión (el acuse de un mensaje implica el acuse de la recepción de **todos** los mensajes que se han consumido por dicha sesión).
    Si un consumidor consume 10 mensajes y acusa el 5º mensaje entregado, los 10 mensajes son acusados.
- **Session.DUPS_OK_ACKNOWLEDGE**: El acuse de recibo se realiza de un modo tardío (***lazy***).
  - Supone la entrega de duplicados en el caso de fallo del proveedor
  - Solo debería utilizarse cuando se puedan tolerar mensajes duplicados
  - Esta opción reduce la sobrecarga de la sesión minimizando la cantidad de trabajo que ésta realiza para prevenir duplicados.

Control del Acuse de Recibo (III)

- Especificaremos el tipo de acuse al crear la sesión:

```
TopicSession session =
   topicConnection.createTopicSeccion(false, Session.CLIENT_ACKNOWLEDGE);
```

- Si se recibe un mensaje en una cola pero no se ha realizado el acuse al terminar la sesión, el proveedor JMS lo retiene y lo re-entrega cuando el consumidor vuelve a acceder a la cola.
  - También retiene aquellos mensajes que no han sido acusados para aquellas sesiones realizadas por un `TopicSubscriber`.

- `Session.recover()` detiene una sesión no transaccional y la reinicia con su 1er mensaje sin acuse de recibo realizado.
  - Las sesiones de los mensajes enviados se resetean hasta el punto posterior al último mensaje acusado.
  - El mensaje que ahora se envía puede ser diferente de aquellos que originalmente fueron enviados (si los mensajes han expirado o si han llegado mensajes con una prioridad mayor)
  - Para un `TopicSubscriber` no durable, el proveedor puede perder mensajes sin acusar cuando se recupera su sesión.

Persistencia de los Mensajes

- JMS soporta 2 modos de entrega de mensajes, definidos como campos del interfaz `DeliveryMode`:
  - **PERSISTENT** (por defecto): indica al proveedor JMS que se asegure que no se pierde ningún mensaje en el caso de que fallase el proveedor JMS.
    Al enviar un mensaje PERSISTENT se almacena es un almacenamiento estable.
  - **NON_PERSISTENT** no obliga al proveedor a almacenar el mensaje
    No garantiza que no se pierda el mensaje en el caso de que falle el proveedor.
- Podemos especificar el modo de entrega de 2 maneras:
  - `MessageProducer.setDeliveryMode()` para indicar el modo de entrega para todos los mensajes enviados por dicho productor.

```
producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
```

  - Modo largo del método `send()` o `publish()` para cada mensaje de forma individual. El segundo parámetro indica el modo de entrega.

```
producer.send(message, DeliveryMode.NON_PERSISTENT, 3, 10000);
```

- Con `DeliveryMode.NON_PERSISTENT` podemos mejorar el rendimiento y reducir la sobrecarga por almacenamiento, pero en aquellas aplicaciones que puedan permitirse la pérdida de mensajes.

Nivel de Prioridad

- Podemos utilizar diferentes niveles de prioridad para que el proveedor JMS envíe primero los mensajes más urgentes.
- 2 modos:
  - Mediante el método `MessageProducer.setPriority()` para indicar el nivel de prioridad de todos los mensajes enviados por dicho productor.

```
producer.setPriority(7);
```

  - Mediante el modo largo del método `send()` o `publish()` para cada mensaje de forma individual, con el 3er parámetro.

```
producer.send(message, DeliveryMode.NON_PERSISTENT, 3, 10000);
```

- Los 10 niveles van desde el 0 (el más bajo) al 9 (el más alto).
- Si no se especifica ningún nivel, el nivel por defecto es 4.
- El proveedor JMS debería entregar los mensajes de mayor prioridad antes que los de menor prioridad, pero esto no implica que lo tenga que entregar en el orden exacto de prioridad.

Expiración de Mensajes

- Por defecto, un mensaje nunca expira.
- Si queremos que un mensaje se vuelva obsoleto tras pasar un cierto periodo de tiempo, podemos indicar un periodo de expiración.
- 2 modos:
  - `MessageProducer.setTimeToLive()` para indicar un periodo de expiración para todos los mensajes enviados por dicho productor.

```
producer.setTimeToLive(60000); // 1 min
```

  - Con el modo largo del método `send()` o `publish()` para cada mensaje de forma individual, con el 4º parámetro (`timeToLive`) en milisegundos.

```
producer.send(message, DeliveryMode.NON_PERSISTENT, 3, 10000);
```

- Si el `timeToLive` es 0, el mensaje nunca expira.
- Al enviar el mensaje, el periodo especificado como `timeToLive` se añade al tiempo actual para calcular el periodo de expiración.
  - Cualquier mensaje que no se ha enviado antes del periodo de expiración especificado será destruido.
    - La destrucción de los mensajes obsoletos preserva el almacenamiento y los recursos computaciones.

Crear Destinos Temporales

- Por norma general, los destinos JMS (colas y tópicos) se crean de modo administrativo más que programativo.
  - Todo proveedor JMS incluye una herramienta de administración.
  - Estos destinos, una vez creados, no suelen borrarse.
- JMS también permite crear destinos temporales (`TemporaryQueue` y `TemporaryTopic`) que duran lo que dura la conexión que los creó.
- Crearemos estos destinos de forma dinámica mediante los métodos `Session.createTemporaryQueue()` y `Session.createTemporaryTopic()`.

- Los únicos consumidores de mensajes que pueden consumir de un destino temporal son aquellos creados por la misma conexión que creó los destinos.
- Cualquier productor de mensajes puede enviar a un destino temporal.
- Si cerramos la conexión a la que pertenece un destino temporal, el destino se cierra y se pierden sus contenidos.

Llamadas Síncronas con Request/Reply

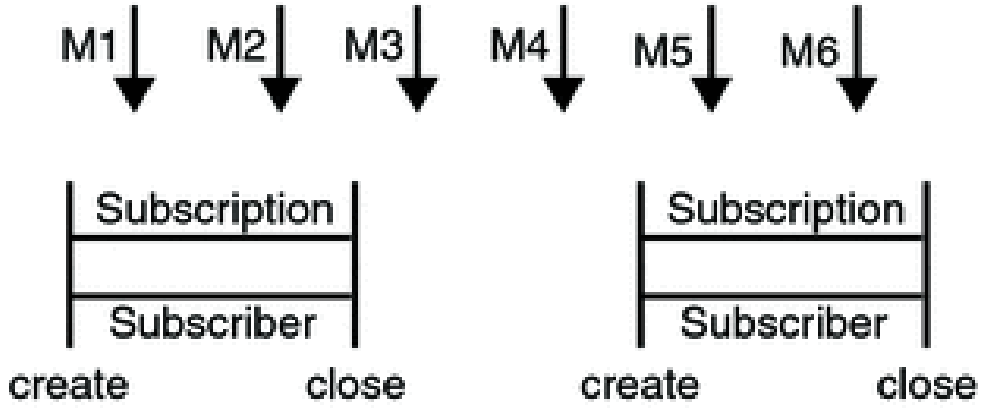
- En ocasiones es necesario implementar un mecanismo de *petición y respuesta* de tal forma que quedemos a la espera de que la petición se satisfaga.
- Mediante este mecanismo simulamos una comunicación síncrona mediante un sistema asíncrono.
- Los pasos que necesita un cliente para simular una comunicación síncrona con un dominio punto a punto son:
  1. Crear una cola temporal mediante el método `createTemporaryQueue()` de la sesión de la cola
  2. Asignar a la cabecera `JMSReplyTo` esta cola temporal (de este modo, el consumidor del mensaje utilizará este cabecera para conocer el destino al que enviar la respuesta).
  3. Enviar el mensaje.
  4. Ejecutar un `receive` con bloqueo en la cola temporal, mediante una llamada al método `receive` en la cola del receptor para esta cola temporal
- El consumidor también puede referenciar a la petición original, asignando a la cabecera `JMSCorrelationID` del mensaje de respuesta, el valor de la cabecera `JMSMessageID` de la petición.

Subscripciones Duraderas

- Para que una aplicación Pub/Sub reciba todos los mensajes publicados:
  - Publicadores: modo de envío `PERSISTENT`
  - Subscriptores: D*urable Subscriber.*
- Una subscripción duradera permanece activa a pesar de que el subscriptor cierre su conexión
  - Se recibirán los mensajes que se publicaron cuando el subscriptor no estaba activo.
- Ofrece la fiabilidad de las colas dentro del dominio Pub/Sub.

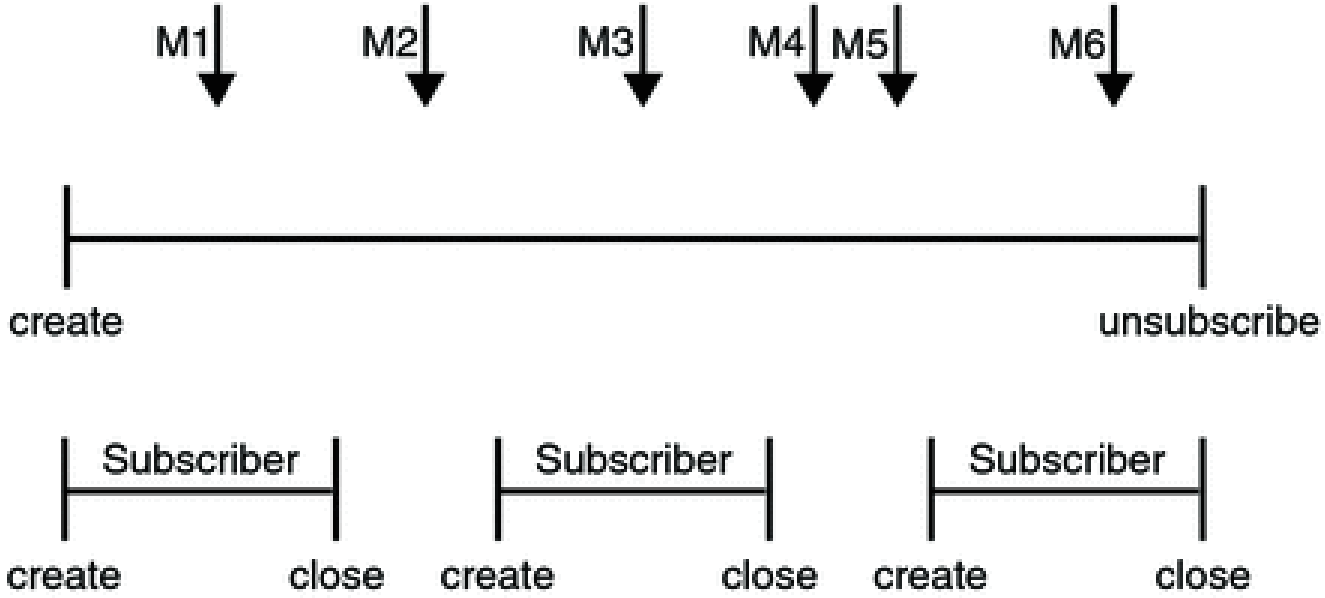
Durable vs Non Durable (I)

- ***Non Durable***: tanto el subscriptor como la subscripción comienzan y finalizan al mismo tiempo
  - Su vida es idéntica
  - Al cerrar un subscriptor, la subscripción finaliza.



Durable vs Non Durable (II)

- ***Durable***: el subscriptor puede cerrarse y recrearse, pero la subscripción sigue existiendo.
- Mantiene los mensajes hasta que la aplicación realiza una llamada al método *`unsubscribe`*

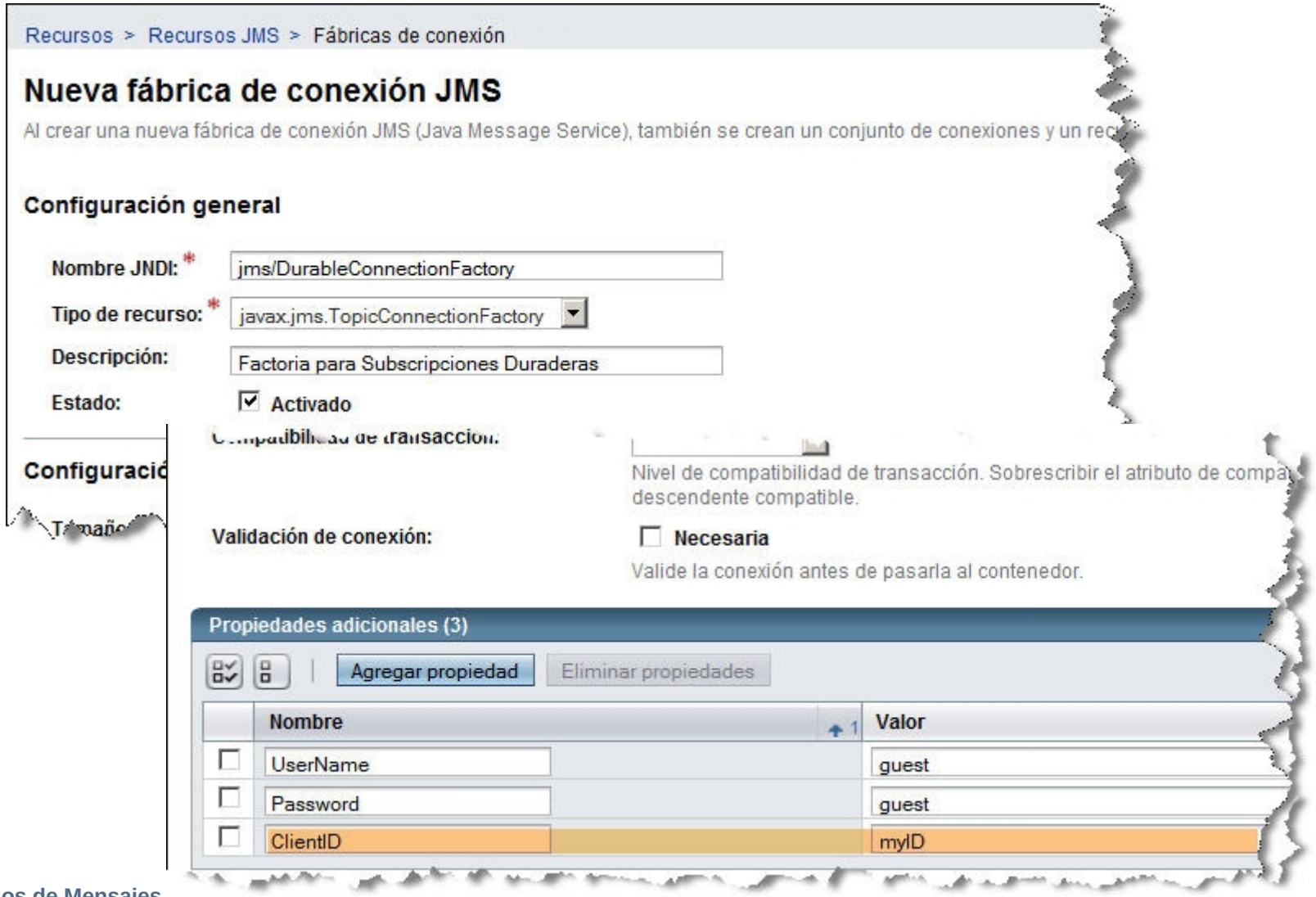


Creando un Subscripción Duradera

- `Session.createDurableSubscriber` crea un subscriptor durable
  - Solo puede tener un subscriptor activo a la vez.
- Hay que definir una **identidad única**
  - ID de cliente para la conexión
  - Nombre de tópico y de subscripción para el subscriptor

- Los siguientes subscriptores que tengan la misma identidad continúan con la subscripción en el mismo estado en el que quedó la subscripción una vez finalizó el subscritor previo.
  - Si una subscripción duradera no tiene ningún subscriptor activo, el proveedor JMS retiene los mensajes hasta que alguien los consume o que expiran.

ID Única en *Glassfish*



Manejo de una Subscripción Duradera

- Para crear la subscripción:

```
String nombreSub = "MiSub";
MessageConsumer durSubs = session.createDurableSubscriber(topic, nombreSub);
```

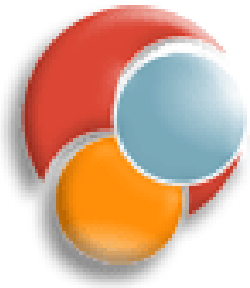
  - El subscriptor se activa tras iniciar la conexión.

- Si queremos cerrar el subscriptor, llamaremos a:

```
durSubs.close();
```

- Para eliminar una subscripción duradera
  1. Cerrar el subscriptor,
  2. Eliminar la subscripción y el estado que mantiene el proveedor para el subscriptor, mediante el método `unsuscribe( )` (pasándole el nombre de la subscripción) :

```
durSubs.close();
Session.unsubscribe(nombreSub);
```



¿Preguntas...?