Enterprise Application Integration

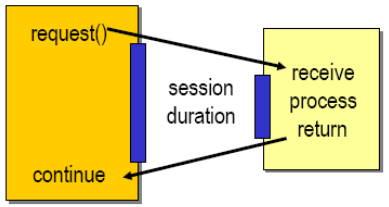
3. Message-Oriented Middleware
3.1 Introduction

2014/2015

Filipe Araújo
Informatics Engineering Department
University of Coimbra
filipius@uc.pt
(slides adapted by Nuno Laranjeiro)

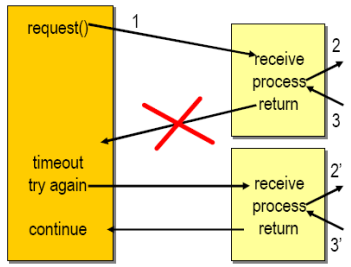
Problems associated to the Client/Server model

- Synchronous interaction which forces both parties to be present at the same time
- Normally blocking: who calls is forced to wait for the server processing to be over
- Maintaining sessions is "expensive" in terms of resources, limiting the number of simultaneous clients
  - Use of *connection-pools*
  - Use of *thread-pools*
  - Use of Asynchronous I/O



Faults affecting the synchronous model (Client/Server)

- If a fault occurs, the session context is lost
  - Re-synchronizing systems can be EXTREMELY complicated!
  - Especially so when nested calls are performed.
- Some examples of faults:
  - If a fault occurs before (1), "having has actually happen".
  - If the fault takes place between (1) and (2) – server fault – the request is lost
  - If the fault takes place between (2) and (3) – subsystems fault – data inconsistencies may occur
  - If the fault happens after (3), corresponding to a lost response, the client will retry the operation



Possible Solution: 2PC

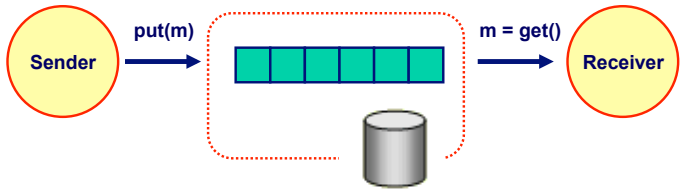
But it's so SLOW

Faults affecting the synchronous model (Client/Server)

- It's possible to use an "exactly-once" semantic
  - But for that it's necessary to use a Two-Phase Commit
- There are some other problems:
  - Speed and scalability
  - User-in-the-loop problem
  - A coordinator must be present
- The probability of failure increases with the number of elements involved in the interaction
- The more the elements involved in an interaction the harder it is to design the system:
  - In terms of software design and implementation
  - In terms of enterprise integration (how to know, from the start, if it is possible to request and use all the needed systems?)
  - In terms of system management (upgrades, restarts, connections, etc.)

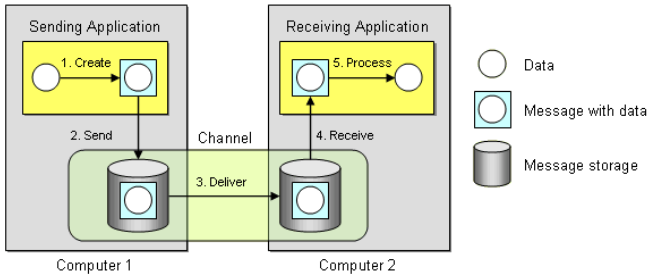
MOM = Message Oriented Middleware

- Asynchronous Interactions!
- A sender puts a message in a queue, typically a "persistent queue", and continues to execute.
- A receiver eventually retrieves the message from the queue.
  - When the message is put on the queue it's possible to notify the receiver.
  - When possible, the receiver collects the message



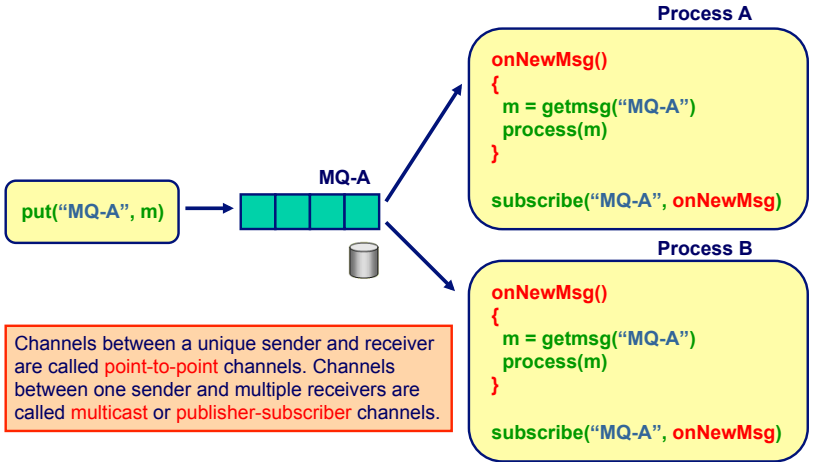
MOM = Message Oriented Middleware (2)

- Who sends the message is not blocked!
  - SEND-AND-FORGET
- Who receives the message doesn't have to be permanently available/online!
  - STORE-AND-FORWARD
- Message queues can be transactional!!



Publish/Subscribe Model

- Message queues also support the *publisher-subscriber* model
  - Allows multi-casting of messages (e.g. 1-to-N model)
  - Allows to receive messages asynchronously



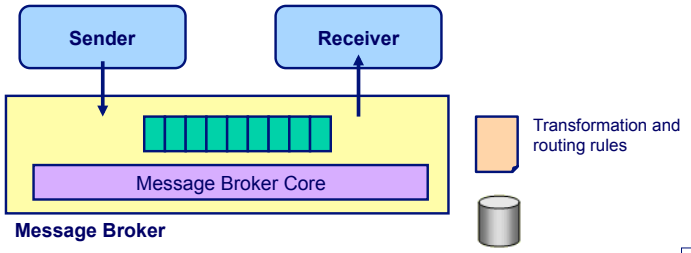
Channels between a unique sender and receiver are called point-to-point channels. Channels between one sender and multiple receivers are called multicast or publisher-subscriber channels.

Advantages / Disadvantages

- Advantages
  - Space, time and flow decoupling
    - Adequate for modular development and enterprise integration
  - Simple to program:
    - In terms of the programmer
    - On how to handle failures (*crashes*, network, exceptions in the business model rules)
    - Good for complex, autonomous and independent systems
  - Naturally support transactions
    - Although with relaxed "isolation"
  - Scalability
- Disadvantages
  - Slower than RPCs
    - Intermediate loops (latency)
  - Less familiar programming model to developers
    - It's necessary to have some care while developing code (e.g. message correlation)
  - No data typing/IDL:
    - In general, BLOBs are sent and received. Sender and receiver must understand the meaning.

Advanced Functionality

- Many MOM systems support advanced functionality:
  - Filtering, Transforming, Priority Establishing, Comparison, etc.
- Message-Broker
  - *Middleware* capable of performing complex operation over the messages it stores and transmits.
  - Warning! Many of these operations are not visible at a "higher level". To use a process orchestrator is normally a better option.



Publish/Subscribe Systems

- Some applications need
  - Asynchronous notifications
  - Information based on contents instead of source

  - Publish/Subscribe Applications
- Pub/sub systems can also be seen as a way of managing multicast groups
- **Typical Applications**
  - Stock quotes, auctions
  - Newsgroups
  - Broadcast of sports results (also through SMS)
  - Systems management and control
    - Alarms and exceptions

Characteristics of MOM

- Parts are loosely coupled:
  - In Space
  - Flow
  - And Time
- Note that
  - Pub/sub is event-based communication
  - But, event-based communication is not necessarily pub/sub
- Provides fault tolerance, load balancing, scalability, transactional support, priorities, synchronous/asynchronous delivery, several delivery semantics (e.g., exactly-once), time-to-live, etc.
- **Vendors implement their own networking mechanism, but provide the same API for message manipulation**
  - JMS providers may interact
  - STOMP is exactly the opposite!

MOM vs. RPCs

Item	MOM	RPC
Metaphor	Post office	Telephone
Client/Server communication	Asynchronous	Synchronous
Client/Server sequence	None	Server must be up before client
Paradigm	Queued	Call-return
Partner needs to be available	No	Yes
Load balancing	Single queue can be used to implement FIFO or priority-based policy	Requires separate TP monitor
Transactional support	Yes, in some products	No. Requires transactional RPC
Message Filtering	Yes	No
Latency	High because of queue	Low
Throughput	Potentially high	Potentially low

MOM Availability

- There are many available platforms!
- Its embedded in many operating systems and application servers:
  - Windows: Microsoft Message Queuing (MSMQ)
  - Oracle AP
  - Java Messaging Service (JMS) is part of J2EE
    - IBM WebSphere
    - BEA WebLogic (now Oracle)
    - JBoss (JBossMQ → HornetQ)
    - Apache Geronimo (Apache ActiveMQ)
    - Glassfish (Open MQ)
    - Sun Java System Application Server
    - Progress Software JMS
    - ...
  - EAI
    - IBM MQ Series
    - TIBCO
    - Vitria
    - WebMethods
    - ...

Enterprise Application Integration

3. Message-Oriented Middleware

3.2 Java™ Message Service

2014/2015

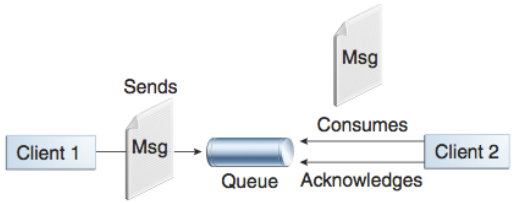
Filipe Araújo
Informatics Engineering Department
University of Coimbra
filipius@uc.pt
(slides adapted by Nuno Laranjeiro)

JMS – Introduction

- JMS is an API to access Enterprise Messaging Systems from Java programs
  - Enterprise Messaging Systems = Message-Oriented Middlewares (MOM)
- The JMS interface can, therefore, be offered by different (asynchronous) message-oriented suppliers
  - The should offer persistent messages
  - Transactional support
  - Authentication
  - ...

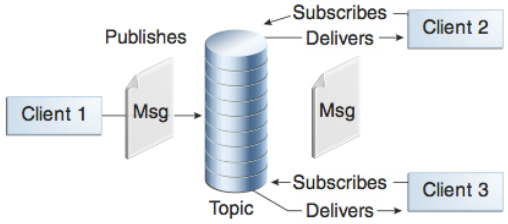
Communication Paradigms – JMS Domains I

- Point-to-Point Paradigm
  - Built over message queues
  - Messages are sent to queues
  - Clients read messages from these queues



Communication Paradigms – JMS Domains II

- Publisher-subscriber
  - Messages sent to a specific node in a hierarchy of contents
  - Publisher and subscriber are usually anonymous
  - Publisher and subscriber may dinamically register
  - The system must send publisher-generated contents to the node subscribers



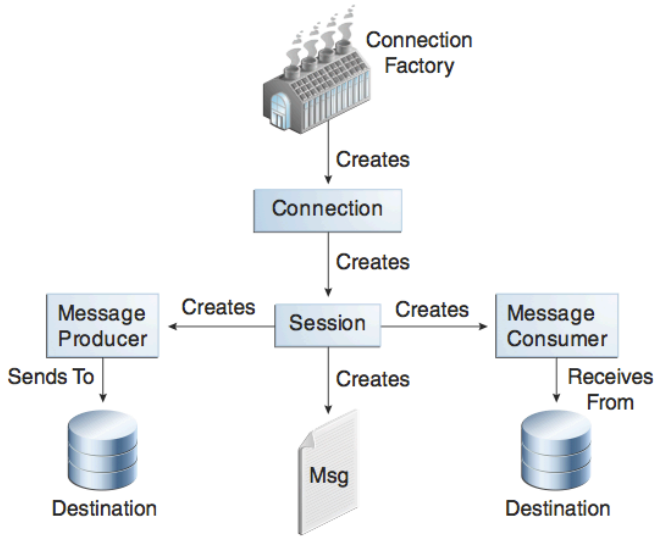
Architecture - Outline

- JMS Architecture Components
- JMS Architecture Scheme
- JMS Client
- JMS Interfaces
- Administered JMS Objects
- JNDI - Java Naming and Directory Interface
- Multi-threading support

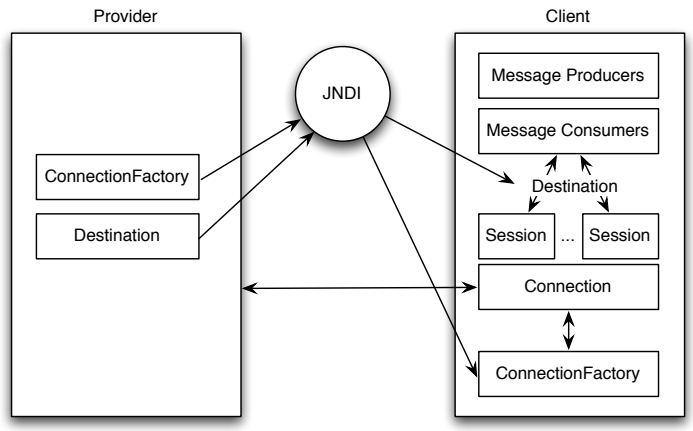
JMS Architecture Components

- Clients
- Messages
  - Each application defines a set of messages used for communication among clients
- JMS Provider
- Administered Objects
  - JMS objects previously created and configured by the administrator for client's use

The JMS API Programming Model



Overview of JMS Interfaces



JMS Interfaces

JMS Common	PTP Domain	Pub/Sub Domain	Thread-Safe?
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory	Yes
Connection	QueueConnection	TopicConnection	Yes
Destination	Queue	Topic	Yes
Session	QueueSession	TopicSession	No
MessageProducer	QueueSender	TopicPublisher	No
MessageConsumer	QueueReceiver	TopicSubscriber	No

Common Facilities (PTP/PubSub) - Overview

- JMS Administered Objects
- Connection
- Connection – client identification
- Connection – preparation
- ...
- Message Delivery and Acknowledgment Mode

JMS Administered Objects

- Two types of JMS administered objects
  - ConnectionFactory
  - Destination

Administered Objects (I)

- One might expect that different providers administer their systems differently
  - Clients should not see such differences
- JMS administered objects contain configuration information created by the administrators and later seen by the clients
- Advantages of separating JMS and administration:
  - Hides specific provider configuration details
  - Abstracts administered information, possibly in a common console, from JMS objects
  - Permite abstrair informação administrativa do JMS em objectos Java organizados e administrados a partir duma consola comum
  - These objects may be accessed from any location using JNDI

Administered Objects (II)

- These objects are created in the provider and then placed in a JNDI namespace by an administrator
  - `Destination`
    - JMS does not define a standard addressing syntax
    - The `Destination` object encapsulates provider specific addresses
    - It may also contain other provider-specific configuration information
    - JMS also supports the utilization of provider-specific addresses from the clients
  - `ConnectionFactory`
    - Encapsulates configuration parameters
    - Clients uses a `ConnectionFactory` to create a connection to the JMS provider.

Conexão (`Connection`)

- Uma conexão JMS é uma ligação activa do cliente ao fornecedor JMS
  - Precisa de recursos fora da máquina virtual
- Fins a que se destina uma conexão:
  - Encapsular uma conexão aberta ao fornecedor JMS
    - Tipicamente representa um *socket* TCP/IP entre o cliente e o *daemon* do serviço
  - Fazer autenticação na fase de estabelecimento
  - Especificar uma identificação única para o cliente
  - Criar sessões
    - Que usam a conexão para produzir e consumir mensagens
  - Fornecer `ConnectionMetaData`
  - Suporta um `ExceptionListener` opcional

Conexão – preparação

- Uma conexão é criada no modo parado (*stopped*)
  - Na fase de preparação o cliente cria sessões, produtores e consumidores de mensagens
  - Nenhuma mensagem pode ser entregue através duma conexão parada
  - Quando a preparação está concluída o cliente invoca o método `start()` da conexão
- Também é possível começar (`start`) a conexão antes de fazer a preparação
  - Mas o cliente pode receber mensagens assincronamente, enquanto faz a preparação

Conexão – encerramento (I)

- A norma do JMS aconselha a fazer o encerramento imediato de conexões desnecessárias, i.e., não esperar pelo *garbage collector*
- Um *close* termina todas as recepções pendentes nas sessões dos consumidores
  - Provavelmente obterão excepções – não podem assumir que a mensagem será uma referência nula (null)
  - Recepções e envios pendentes devem terminar ordeiramente, primeiro

Conexão – encerramento (II)

- As capacidades de processamento de mensagens previamente recebidas devem continuar disponíveis
- Fechar uma ligação faz *rollback* de todas as transacções em progresso de sessões transaccionadas
- Utilização duma conexão fechada lança uma `IllegalStateException`
- Fechar uma conexão fechada não lança uma excepção

Suspender Recepção de Mensagens

- Método `stop()` – suspender temporariamente a recepção de mensagens
  - *MessageConsumers* inibidos e *MessageListeners* não recebem
- Método `start()` – reiniciar a recepção
- Parar uma conexão parada ou reiniciar uma conexão (re)iniciada não tem qualquer efeito

ExceptionListener

- Numa conexão pode, opcionalmente, ser registado um ExceptionListener
- Caso aconteça algum problema, o cliente pode ser alertado assincronamente
- O ExceptionListener não deve receber excepções geradas por uma chamada JMS
  - Estas deverão ser entregues a quem executa a chamada

Sessão

- Contexto mono-tarefa para produzir e consumir mensagens
- Objecto ligeiro do JMS
  - Embora possa reservar recursos fora da JVM
- Serve para:
  - É o suporte dos *MessageProducers* e *MessageConsumers*
  - É o suporte para destinos temporários
  - Permite criar objectos `Destination` para clientes que precisam de dinamicamente manipular nomes de destinos específicos ao fornecedor
  - Oferece suporte para mensagens optimizado pelo fornecedor
  - Suporte de transacções
  - Define uma ordem para as mensagens que produz e consome
  - Uma sessão retém as mensagens que consome até que estas sejam confirmadas
  - Serializa execução de *MessageListeners* registados

MessageProducer, *MessageConsumer* e *MessageListener*

- Todos os que dizem respeito à mesma sessão são executados por um único fluxo de execução
- *MessageProducer*
  - pode escrever para uma fila ou para um tópico
- *MessageConsumer*
  - idem, mas para leitura
- *MessageListener*
  - é um consumidor assíncrono de mensagens registado junto da sessão
- Tipicamente uma tarefa bloqueia-se num consumidor (síncrono ou assíncrono) e depois utiliza um ou mais produtores
- Uma sessão não pode ter receptores síncronos e assíncronos em simultâneo
- Uma sessão pode ser usada para vários destinos em simultâneo (tópicos ou filas)

Fechar uma Sessão

- Uma sessão deve ser encerrada logo que deixe de ser necessária
  - *Garbage collector* pode não ser suficientemente eficaz
- Termina todo o processamento de mensagens da sessão
- Fecho só deve terminar quando todo o processamento de mensagens tiver sido terminado ordeiramente
- Fechar uma sessão transaccionada deve fazer o *rollback* da transacção em progresso
- Usar os consumidores ou produtores depois de fechada a sessão gera uma `IllegalStateException`

Sessões Transaccionadas

- Opcionalmente a sessão pode ser transaccionada
- Só é suportada uma série de transacções de cada vez
- Cada transacção agrupa um conjunto de mensagens produzidas e consumidas numa unidade atómica de trabalho
  - Produtores
    - Commit: o *message broker* envia todas as mensagens
    - Rollback: o m.b. liberta todas as mensagens
  - Consumidores
    - Commit: o m.b. liberta todas as mensagens
    - Rollback: o m.b. reenvia todas as mensagens
- Uma nova transacção começa logo que termine a anterior com o `commit()` ou com o `rollback()`
- Numa sessão transaccionada o `commit()` faz a confirmação das mensagens recebidas

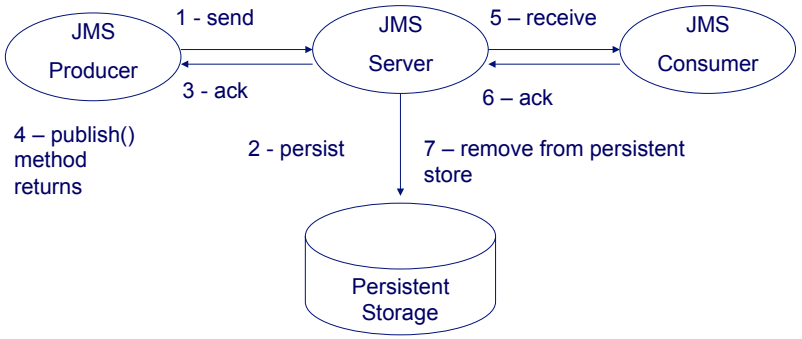
Delivery Mode & Message Confirmation

- NON_PERSISTENT (lower overhead)
  - at-most-once semantics
- PERSISTENT
  - once-and-only-once semantics
- Confirmation Modes
  - AUTO_ACKNOWLEDGE
  - CLIENT_ACKNOWLEDGE
  - DUPS_OK_ACKNOWLEDGE

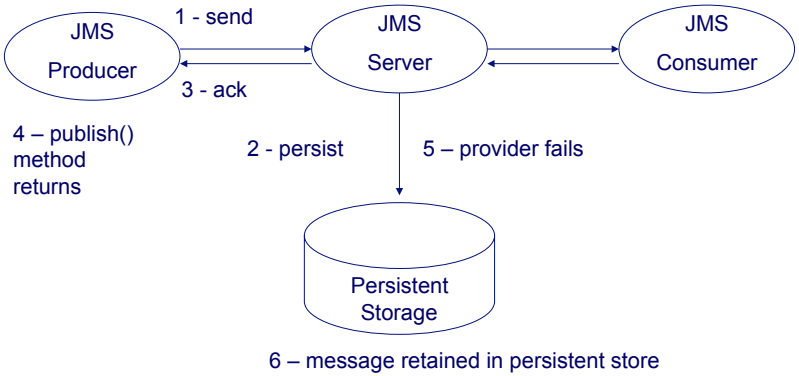
However, semantics depend on the confirmation mode

Only for non-transacted sessions

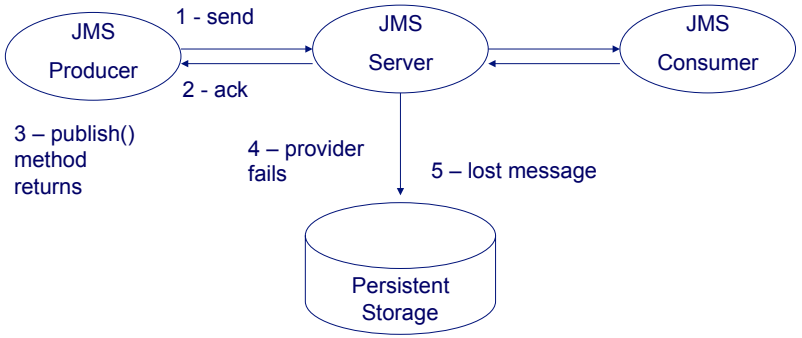
Persistent Messages (Normal Case)



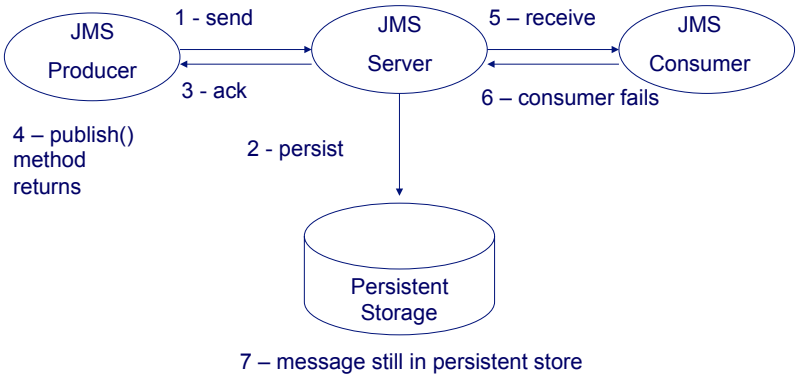
Persistent Messages (Provider Failure)



Non-Persistent Messages (Provider Failure)



Durable Subscriptions (Consumer Failure)



Ordenação no Envio de Mensagens (I)

- Paradigma pub/sub
  - Um cliente pode não receber mensagens NON_PERSISTENT
  - A ordem só é garantida dentro de cada modo de envio (PERSISTENT/NON_PERSISTENT)
  - Um cliente pode usar sessões transaccionadas para agrupar mensagens em unidades atómicas. A ordem das mensagens na transacção é importante

Ordenação no Envio de Mensagens (II)

- Paradigma ponto-a-ponto
  - Mensagens são entregues pela ordem de chegada
  - Subscrições são sempre duráveis
  - Mensagens podem ser ou não persistentes
  - Uma mensagem persistente é sempre entregue, a menos que expire
  - Uma mensagem não persistente pode perder-se

Ordenação no Envio de Mensagens (III)

- Prioridades
  - Duas categorias:
    - Normal: 0-4
    - Rápida: 5-9
  - Servidores JMS podem (não é obrigatório) alterar a ordem de entrega das mensagens com base na prioridade

MessageConsumer

- Dois modos de recepção
  - Recepção síncrona
    - Usa um dos métodos `receive()` da interface *MessageConsumer*
  - Recepção assíncrona
    - Usa um *MessageConsumer* que implementa a interface *MessageListener*. Regista este objecto e o fornecedor do serviço JMS chama o método `onMessage()`.
- É possível filtrar as mensagens que se deseja receber

MessageProducer

- Usado para enviar mensagens para um destino
  - **`session.createMessageProducer(Destination)`**
- Alguns produtores não recebem o destino no momento da criação
  - permitem enviar respostas para destinos diferentes (campo `JMSReplyTo` das mensagens)
- Um cliente pode especificar o modo de entrega, prioridade e TTL para mensagens dum produtor, mas também pode indicar essa informação por mensagem

Mensagens - Objectivos

- Fornecer uma API única
- Fornecer uma API para criar mensagens que se adequem ao formato das aplicações não-JMS existentes
- Suportar aplicações que funcionem em múltiplos S.O., arquitecturas e linguagens
- Suportar mensagens que contenham objectos Java
- Suportar mensagens que tenham XML

Tipos de Mensagens

- BytesMessage
  - Uma *stream* de *bytes* não interpretados
- MapMessage
  - Um conjunto de pares nome-valor, em que os nomes são strings e os valores são tipos primitivos do Java. As entradas podem ser acedidas sequencial ou aleatoriamente
- TextMessage:
  - Mensagem que contém uma String.
- StreamMessage:
  - Uma *stream* de tipos primitivos do Java, preenchidos e lidos sequencialmente
- ObjectMessage:
  - Mensagem que contém um objecto Java serializável.

Formato das Mensagens

- Cabeçalho
  - Informação usada por ambos, clientes e fornecedores para identificar e encaminhar mensagens
- Propriedades
  - Campos opcionais adicionais ao cabeçalho. Há três tipos de propriedades:
    - específicas à aplicação
    - standard
    - específicas ao fornecedor
- Corpo

Campos do Cabeçalho

Campos do cabeçalho	Marcados por
JMSDestination	Método Send
JMSDeliveryMode	Método Send
JMSExpiration	Método Send
JMSPriority	Método Send
JMSMessageID	Método Send
JMSTimestamp	Método Send
JMSCorrelationID	Cliente
JMSReplyTo	Cliente
JMSType	Cliente
JMSRedelivered	Fornecedor

Propriedades

- Uma propriedade é identificada por um nome
- Ao nome da propriedade corresponde um valor do tipo booleano, byte, short, int, long, float, double ou String
- As propriedades são iniciadas antes do envio da mensagem
- Depois de o cliente receber a mensagem deve receber a excepção `MessageNotWritableException` se tentar alterar propriedades

Interface *Message*

- Interface base para todas as mensagens JMS
- Define os cabeçalhos JMS das mensagens, os mecanismos de construção de propriedades e o método acknowledge usado por todas as mensagens

Selecção de Mensagens

- O JMS fornece aos clientes a capacidade de delegarem no fornecedor capacidades de filtragem da mensagem
  - Especialmente útil quando as mensagens são recebidas por mútliplos clientes
  - Clientes adicionam critérios de selecção usando as propriedades às mensagens
  - Quando inicia um MessageConsumer um cliente especifica também um selector de mensagens
    - selector de mensagens usa um subconjunto da sintaxe SQL92

Subscriber

```
String redSelector = "color='red'";
MessageConsumer mc = ts.createConsumer(t,
redSelector);
```

Publisher

```
tm1.setStringProperty("color", "red");
```


JMSReplyTo, JMSCorrelationID & TemporaryQueues

- JMSReplyTo
  - Allows the sender to specify where should the receiver reply to
    - Usually a temporary queue
- TemporaryQueues
  - Created dynamically by an endpoint
- JMSCorrelationID
  - Allows the correlation of different JMS messages
    - E.g., inside a single business process

Let's see a small demo…

- **Jboss (HornetQ)**

IMPORTANT NOTICE