

Java Message Service

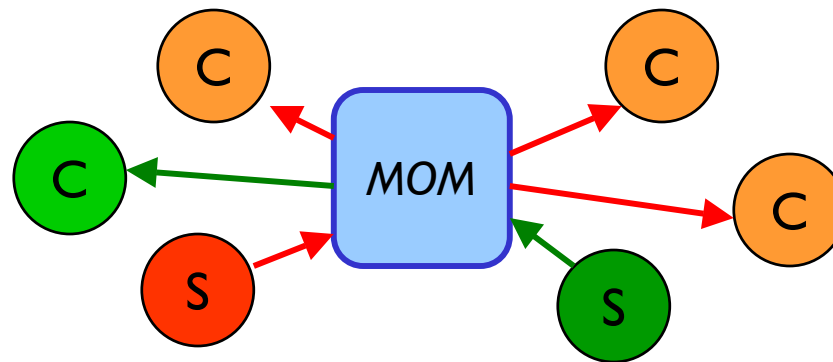
Helder da Rocha (helder@acm.org)
www.argonavis.com.br

- **Introdução**
 - *Messaging e Message-Oriented Middlewares (MOMs)*
 - *Quando usar MOMs? Vantagens e desvantagens*
 - *O que é o Java Message Service*
- **Conceitos fundamentais**
 - *Arquitetura*
 - *Domínios: ponto-a-ponto e pub/sub*
 - *Produção e consumo de mensagens*
- **Plataforma de desenvolvimento JMS**
 - *Objetos gerenciados*
 - *Conexões, sessões, produtores, consumidores, mensagens, exceções*
- **Exemplos de aplicações**
 - *Configuração do ambiente (J2EE RI e JBossMQ)*
 - *Exemplo de aplicação ponto-a-ponto*
 - *Exemplo de aplicação pub/sub*

- *O objetivo deste minicurso é apresentar uma introdução ao modelo de comunicações baseado em mensagens (messaging) e como implementá-lo em aplicações Java usando o **Java Message Service (JMS)***
 - *Conceitos fundamentais de messaging e MOMs*
 - *Exemplos de aplicações (estilo "Hello World") para os dois paradigmas de messaging (PTP e pub/sub)*
 - *Como usar o J2EE RI e JBossMQ como provedores JMS*

O que é Messaging

- **Método de comunicação entre componentes ou aplicações**
 - Arquitetura *peer-to-peer* com **serviço centralizado** para repasse de mensagens recebidas e enviadas
 - Clientes e servidores enviam e recebem mensagens para canais administrados por serviço central de mensagens (MOM)




- **Viabiliza comunicação distribuída com acoplamento fraco**
 - Interface genérica: MOM ignora conteúdo e repassa qualquer coisa. Formato do conteúdo deve ser conhecido pelas partes
 - Assíncrona: Comunicação pode ocorrer mesmo que o cliente e servidor não estejam disponíveis ao mesmo tempo

Message-oriented Middleware (MOM)

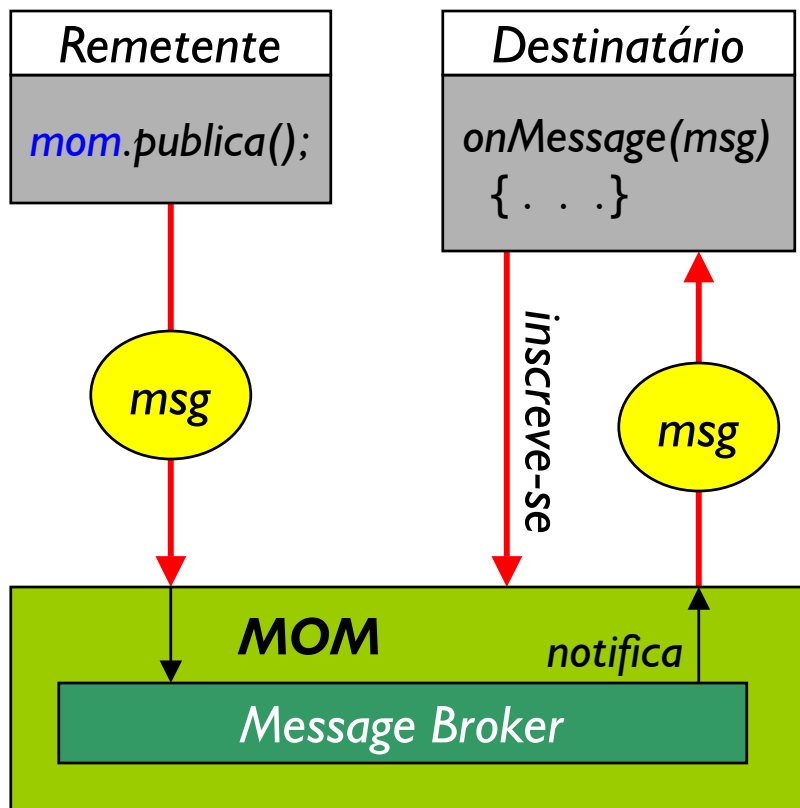
- *Sistemas de messaging são freqüentemente chamados de **Message-Oriented Middleware (MOM)***
 - *Conceito não é novo: já existiam há algum tempo em implementações proprietárias (e incompatíveis)*
 - *JMS API: solução independente de fabricante para acessar serviços MOM a partir de clientes Java*
- *Alguns produtos MOM compatíveis com JMS:*
 - *Open source: JBossMQ, OpenJMS, JORAM*
 - *IBM MQSeries, IPlanet MQ, Bea WebLogic, HP-MS, Progress SoniqMQ*
 - *Mais em: java.sun.com/products/jms/*

Messaging vs. RMI/RPC vs. E-mail

- **Messaging**
 - Mensagens são representadas como **eventos** (que causam número limitado de ações por parte do MOM)
 - **Interface genérica** (pode ser reutilizada para aplicações diferentes)
 - Arquitetura **centralizada** (tudo passa pelo MOM)
 - Serviços de diretórios localizam **canais de comunicação** (destinos)
- **RMI/RPC (Corba, Java RMI, etc.)**
 - Mensagens são representadas como **chamadas** para métodos remotos (número ilimitado de ações)
 - Cada aplicação se comunica através de uma **interface definida**
 - Pode ser **descentralizado** (rede de ORBs ligados por IIOP)
 - Serviços de diretórios localizam **objetos**
- **E-mail**
 - Uma ou ambas as partes podem ser usuários humanos
 -  Messaging é sempre comunicação **100% B2B**

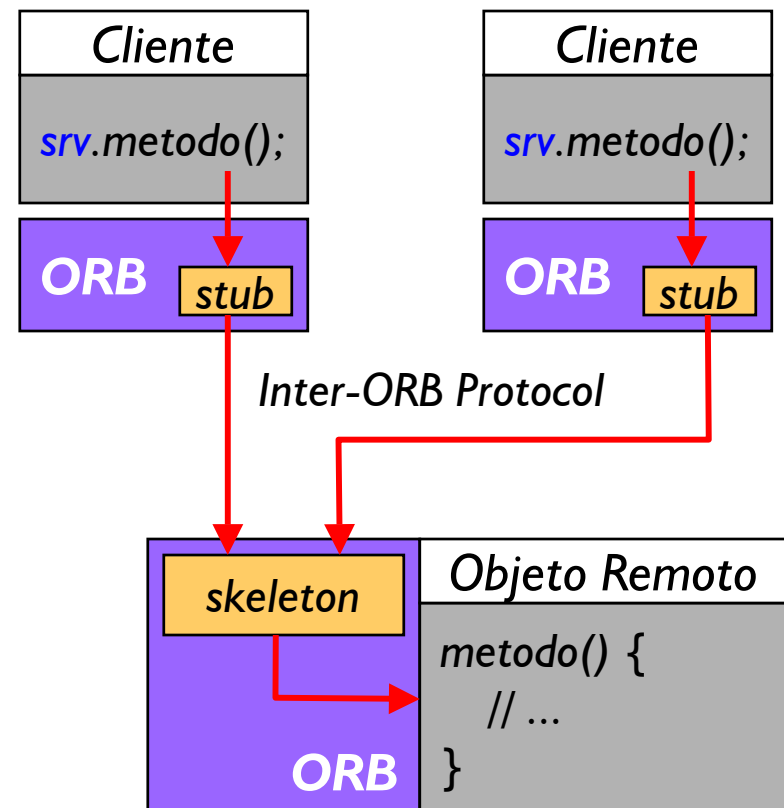
Messaging vs. RMI/RPC

- Sistema de Messaging com paradigma pub/sub



Protocolo comum:
mensagem

- Sistema RMI/RPC (Java RMI ou CORBA)



Protocolo comum:
interface dos objetos

Desvantagens dos MOMs

■ Desvantagens

- Camada adicional para repasse de mensagens
- Centralização em único ponto introduz risco de falha de todo o sistema caso o serviço de mensagens falhe

➡ Solução: replicação, clustering

■ Desvantagens relativas

- **Muito genérica**: aplicações precisam decifrar as mensagens para que possam operar; esconde a interface de programação remota dentro das mensagens
- **Comunicação assíncrona** (geralmente): dificulta a criação de aplicações que necessitam de comunicação síncrona.
- Não faz tratamento de representação de dados (**data marshalling**) - MOM é apenas meio de transporte

Vantagens dos MOMs (I)

■ Escalabilidade

- *Para aumentar a capacidade servidora, basta acrescentar mais servidores (não é preciso mexer nos componentes)*
- *Novos clientes podem se conectar para usar mensagens em outras aplicações*
- *Infraestrutura é reutilizada para novas aplicações*

■ Comunicação assíncrona

- *Componentes podem realizar outras tarefas enquanto não estão ocupados lidando com requisições*
- *Podem sondar o servidor em busca de novas mensagens quando estiverem livres (PTP)*
- *Podem se cadastrar para, quando houver mensagens novas, receber notificação (pub/sub)*

Vantagens dos MOMs (2)

- **Desacoplamento**
 - *Maior modularidade, maior reuso (substituibilidade), maior simplicidade, maior robustez (falhas localizadas)*
 - *Papéis bem definidos simplificam o desenvolvimento: produtor, consumidor e serviço tem única interface, independente da aplicação*
 - *Servidor de messaging é responsável pela qualidade do serviço (não é preocupação dos componentes)*
- **Flexibilidade**
 - *API definida pelo tipo das mensagens (e não por interfaces)*
 - *Meio comum é a **mensagem**: se componentes a entendem, o resto (linguagens, plataformas, etc.) não importa!*

Quando usar um MOM em vez de RMI/RPC?

- ... ou, quando decidir por acoplamento mais fraco?
 - Quando a comunicação se baseia mais no formato de mensagens que em interfaces rígidas (componentes **não dependem da interface** de outros componentes)
 - Quando a disponibilidade dos componentes é imprevisível, mas sua aplicação precisa rodar mesmo que componentes **não estejam todos acessíveis**
 - Quando for preciso suportar comunicação assíncrona: componente pode enviar informações para outro e **continuar a operar mesmo sem receber resposta imediata**
- ➡ Cenário comum em muitas aplicações B2B!**

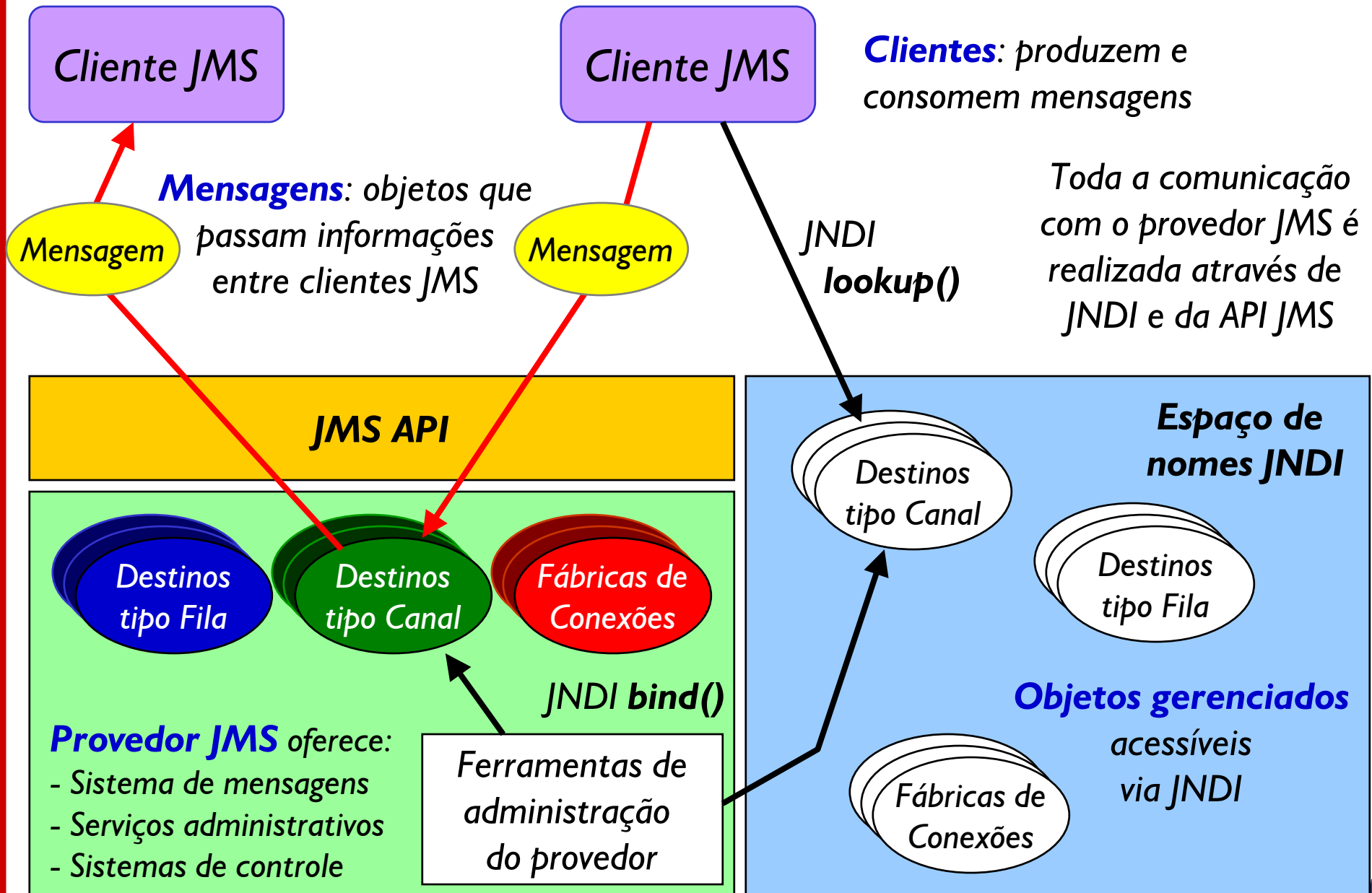
Java Message Service

- Interface Java única para unir as MOMs incompatíveis
- **API** que permite que aplicações criem, enviem, recebam e leiam mensagens através de um MOM
 - API consiste principalmente de **interfaces** (implementadas pelo fabricante do MOM)
 - Parte integral da **plataforma J2EE** (acrescenta possibilidade de comunicação assíncrona a EJBs)
- **Resumo das metas (segundo a especificação JMS)**
 - Oferecer uma API simples, unificada e compatível com aplicações existentes (não-JMS)
 - Suportar aplicações heterogêneas em diferentes SOs, plataformas, arquiteturas e linguagens
 - Suportar mensagens contendo objetos serializados Java e páginas XML

Java Message Service (2)

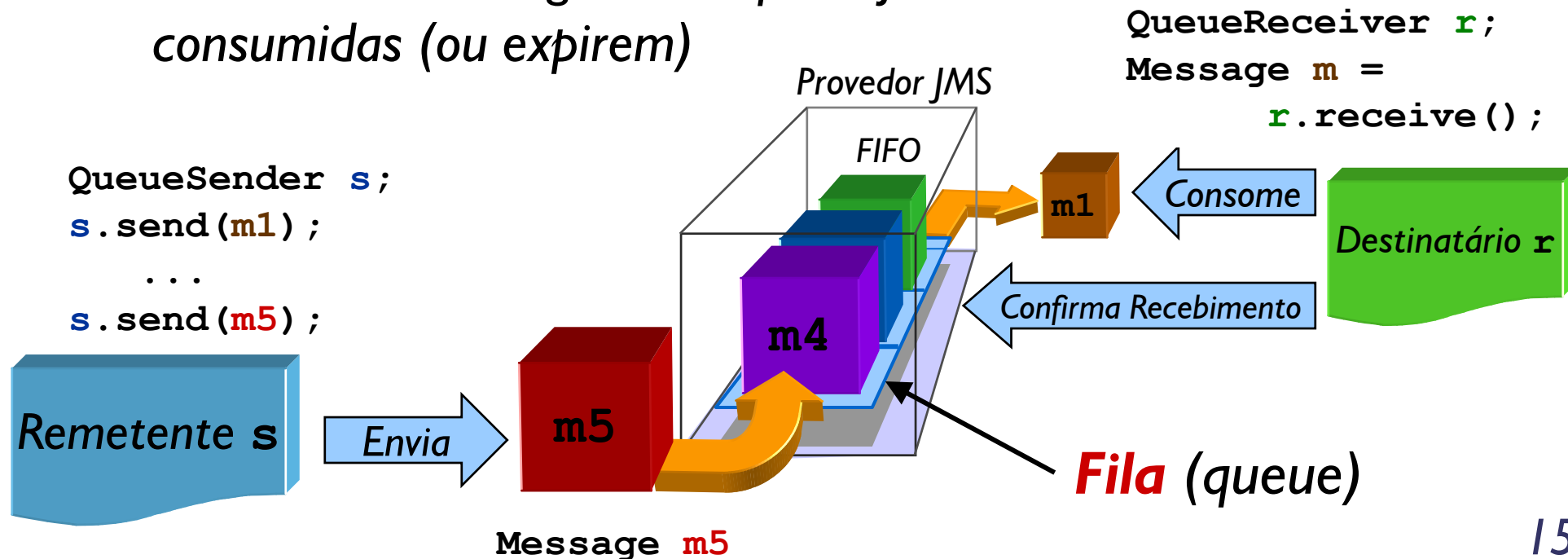
- *Principais características*
 - Modelo flexível de desenvolvimento baseado em **dois domínios**: ponto-a-ponto e publish/subscribe
 - Controle de **persistência**, **tempo de vida**, **prioridades** e **durabilidade** associados a serviços e mensagens
 - Suporte à comunicação **síncrona** e **assíncrona**
 - Suporte a **transações** no envio e recebimento de mensagens
 - **Roteamento seletivo** através da filtragem de propriedades das mensagens (suporta subconjunto do SQL-92 usado para fazer queries em mensagens)
 - Suportado por **todos** os servidores de aplicação J2EE (implementam os dois domínios: PTP e pub/sub)

Arquitetura JMS



Domínio ptp (ponto-a-ponto)

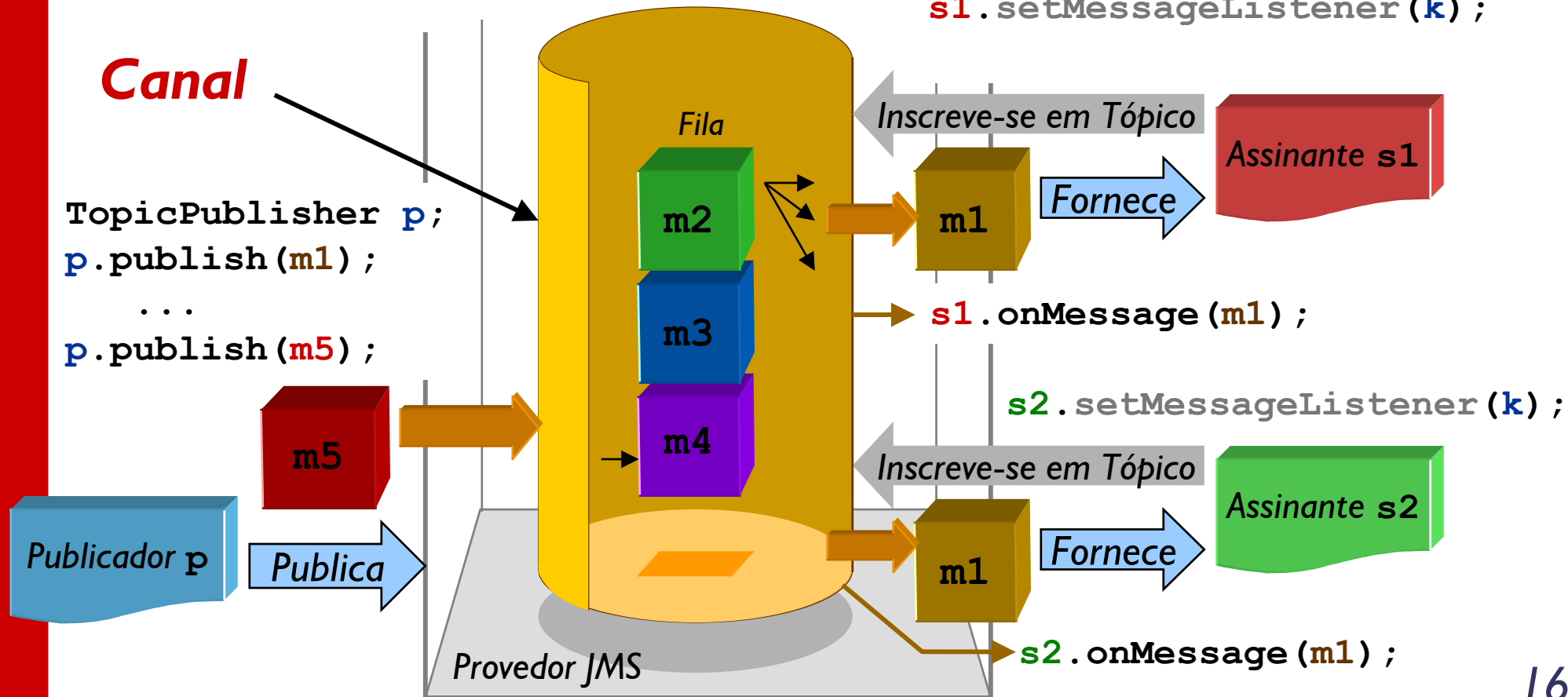
- Baseado no conceito de filas, remetentes e destinatários
 - Um para um: cada mensagem é enviada para uma fila específica e é **consumida** por um destinatário (que pode ou não estar disponível no momento)
 - Destinatário confirma que a mensagem foi recebida e processada corretamente (**acknowledgement**)
 - Filas retêm mensagens até que sejam consumidas (ou expirem)



Domínio pub/sub (publica/inscreve)

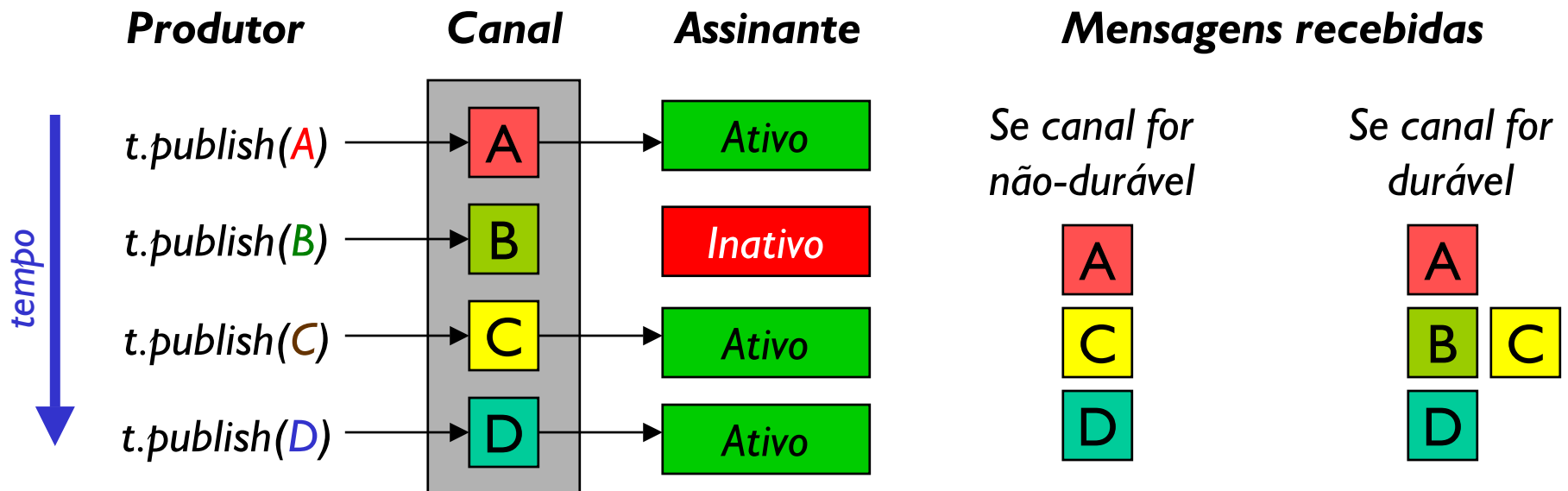
- **Baseado em canais (tópicos)**
 - *Muitos para muitos: mensagens são enviadas a um canal onde todos os assinantes do canal podem retirá-la*
 - Assinantes recebem **notificação**

```
TopicSubscriber s1;  
MessageListener k =  
    new MessageListener() {  
        public void  
            onMessage(Message m) {...}  
    };  
s1.setMessageListener(k);
```



Canais duráveis e não-duráveis

- Cada mensagem enviada a um canal pode ter múltiplos consumidores
 - Mensagem permanece disponível até que todos os assinantes a tenham retirado
 - Em canais **não-duráveis** assinante perde as mensagens enviadas nos seus períodos de inatividade

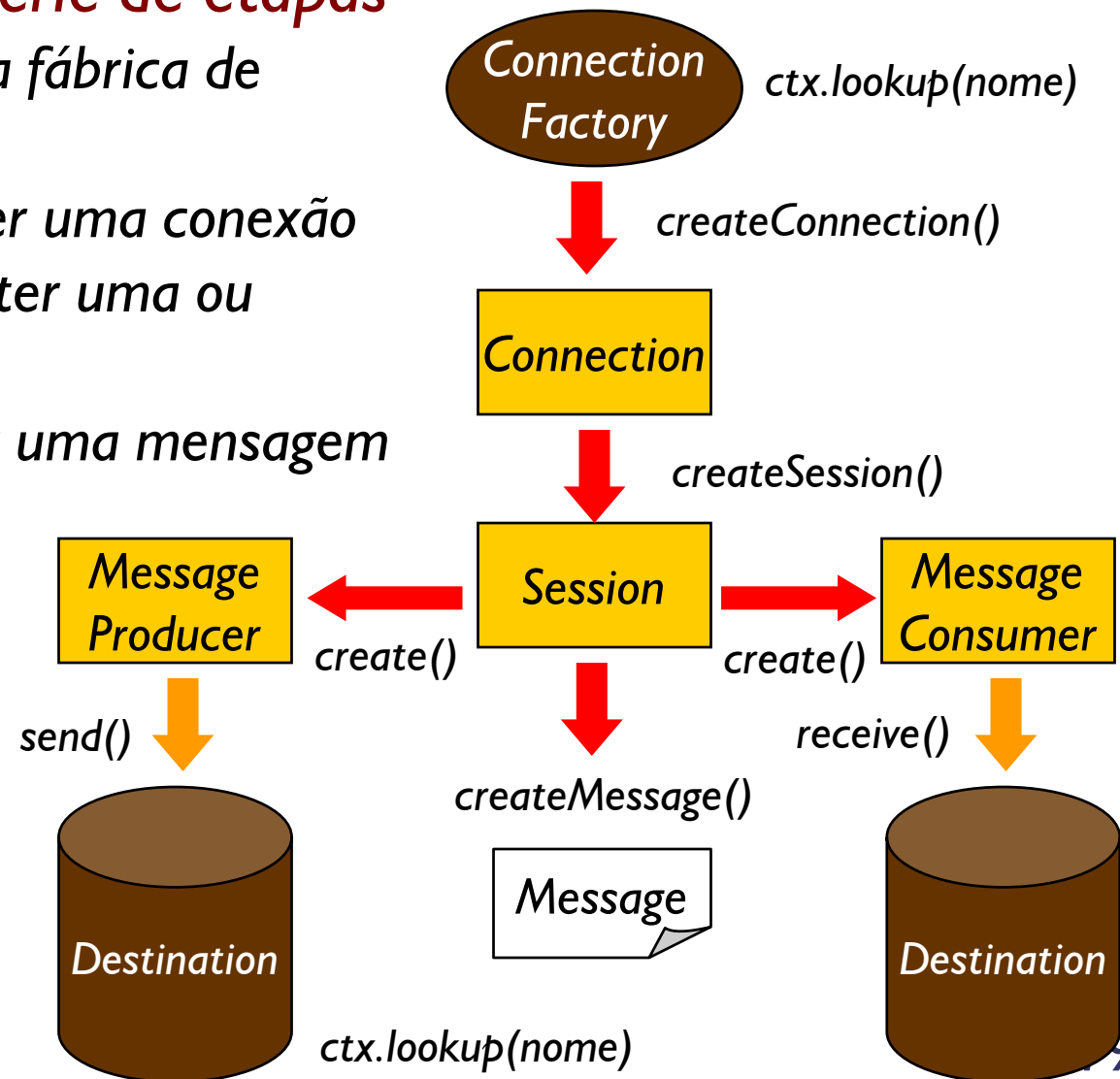


Consumo de mensagens

- *Sistemas de messaging são sempre assíncronos no sentido de que não há dependência quanto ao tempo de envio e recebimento das mensagens*
- *JMS porém permite um tipo de sincronismo*
 - *Pode-se bloquear as operações em um destinatário até que uma determinada mensagem chegue*
- *A especificação JMS, portanto, define que mensagens podem ser consumidas de duas formas*
 - **Síncrona**: *quando o destinatário envia uma chamada receive() e fica a esperar pelo recebimento de mensagens*
 - **Assíncrona**: *o cliente registra-se como ouvinte de mensagens e é notificado quando elas chegam*

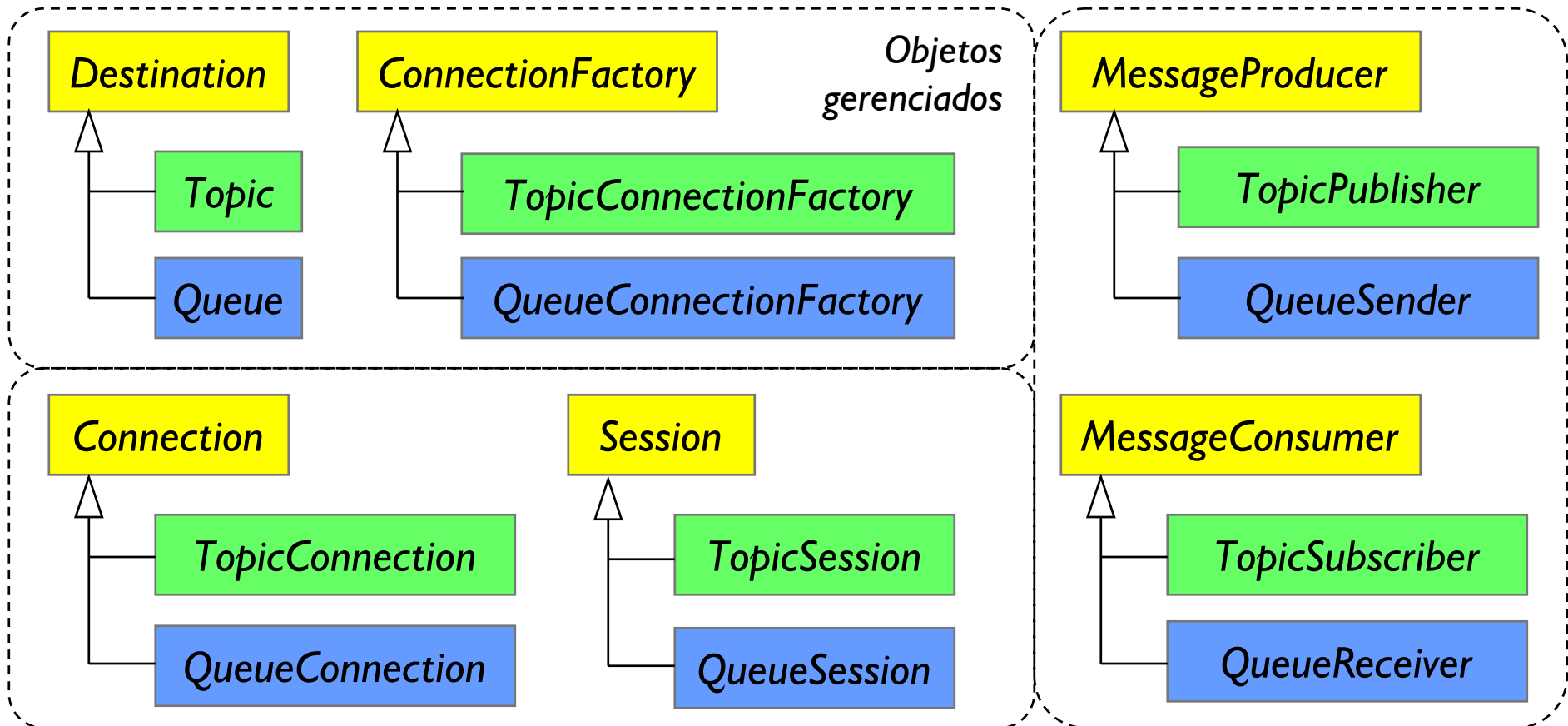
Desenvolvimento de aplicações JMS

- Para escrever aplicações que enviam ou recebem mensagens é preciso realizar uma série de etapas
 - Obter um destino e uma fábrica de conexões via JNDI
 - Usar a fábrica para obter uma conexão
 - Usar a conexão para obter uma ou mais sessões
 - Usar a sessão para criar uma mensagem
 - Iniciar a sessão
- Depois, pode-se
 - Enviar mensagens
 - Receber mensagens
 - Cadastrar ouvintes para receber mensagens automaticamente



Interfaces: dois domínios

- **É preciso escolher um domínio**
 - Para cada domínio há um destino, fábrica de conexões, conexão, sessão, produtor e consumidor
 - Interfaces são semelhantes



- Domínio pub/sub
- Domínio ptp

Objetos gerenciados

- **Destinos** (filas e tópicos) e **fábricas de conexões** não podem ser criados através da API
 - São criados pelo provedor JMS e ligados a JNDI através de suas ferramentas administrativas. No J2EE-RI:
 - > `j2eeadmin -addJmsDestination queue/MinhaFila queue`
- **Para usar um objeto gerenciado**
 - O cliente precisa obtê-lo através de uma consulta (lookup) no serviço de nomes JNDI
 - Não há padronização do espaço de nomes (nomes default variam entre fabricantes)
- **Exemplo (obtenção de um destino do tipo Topic):**

```
String nomeJRI = "jms/Topic"; //Default J2EE RI
String nomeJBoss = "topic/testTopic"; // Default JBossMQ
```

```
Context ctx = new InitialContext();
Topic canal = (Topic) ctx.lookup(nomeJBoss);
```

Há dois tipos de destinos JMS

▪ **Filas (Queue)**

- *Retêm todas as mensagens que recebem até que sejam retiradas ou expirem*
- *Para cada mensagem enviada, apenas um cliente pode retirá-la*

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

▪ **Canais (Topic)**

- *Cada canal pode ter vários clientes assinantes*
- *Cada assinante recebe uma cópia das mensagens enviadas*
- *Para receber uma mensagem publicada em um canal, clientes precisam já ser assinantes dele antes do envio.*
- *Em canais "duráveis", assinantes não precisam estar ativos no momento do envio. Retornando, receberão mensagens não lidas.*

```
Topic canal = (Topic) ctx.lookup("jms/Topic");
```

Fábricas de conexões

- *Antes que se possa*
 - **enviar** uma mensagem para uma fila,
 - **publicar** uma mensagem em um canal,
 - **consumir** uma mensagem de uma fila ou
 - **fazer uma assinatura** de um canal
- é preciso obter uma **conexão** ao provedor JMS
- Isto é feito através de uma **fábrica de conexões**. Há duas:
 - **TopicConnectionFactory** - para conexões no domínio Topic
 - **QueueConnectionFactory** - para conexões no domínio Queue
- É preciso conhecer o **nome JNDI**

```
String nomeJRI = "TopicConnectionFactory"; //default J2EE-RI
String nomeJBoss = "ConnectionFactory"; // JBossMQ
```

```
Context ctx = new InitialContext();
TopicConnectionFactory factory =
    (TopicConnectionFactory) ctx.lookup(nomeJBoss);
```

Precisa ser definido (geralmente arquivo `jndi.properties` no CPATH)

- *Encapsulam uma conexão virtual com o provedor JMS*
 - *Suportam múltiplas sessões (threads)*
- *Uma vez obtida uma fábrica de conexões, pode-se obter uma conexão*

```
QueueConnection queueCon =  
    queueConnectionFactory.createQueueConnection();
```

```
TopicConnection topicCon =  
    topicConnectionFactory.createTopicConnection();
```

- *Quando a conexão terminar, é preciso fechá-la*
 - *O fechamento fecha todas as seções, produtores e consumidores associados à conexão*

```
queueCon.close();  
topicCon.close();
```


- *Contexto onde se produz e se consome mensagens*
 - *Criam produtores, consumidores e mensagens*
 - *Processam a execução de ouvintes*
 - *Single-threaded*
- *Podem ser configuradas para definir*
 - *forma de acknowledgement*
 - *uso ou não de transações (tratar uma série de envios/recebimentos como unidade atômica e controlá-la via commit e rollback)*

Exemplos

```
TopicSession topicSession =  
    topicCon.createTopicSession(false,  
        Session.AUTO_ACKNOWLEDGE);  
QueueSession queueSession =  
    queueCon.createQueueSession(true, 0);
```

Sem transações

Confirmação automática após recebimento correto

Sessão controlada por transações

Tratamento de confirmações não especificado

Produtores de mensagens

- Objeto criado pela sessão e usado para enviar mensagens para um destino

- *QueueSender*: domínio ponto-a-ponto
- *TopicPublisher*: domínio pub/sub

```
QueueSender sender =  
    queueSession.createSender(fila);  
TopicPublisher publisher =  
    topicSession.createPublisher(canal);
```

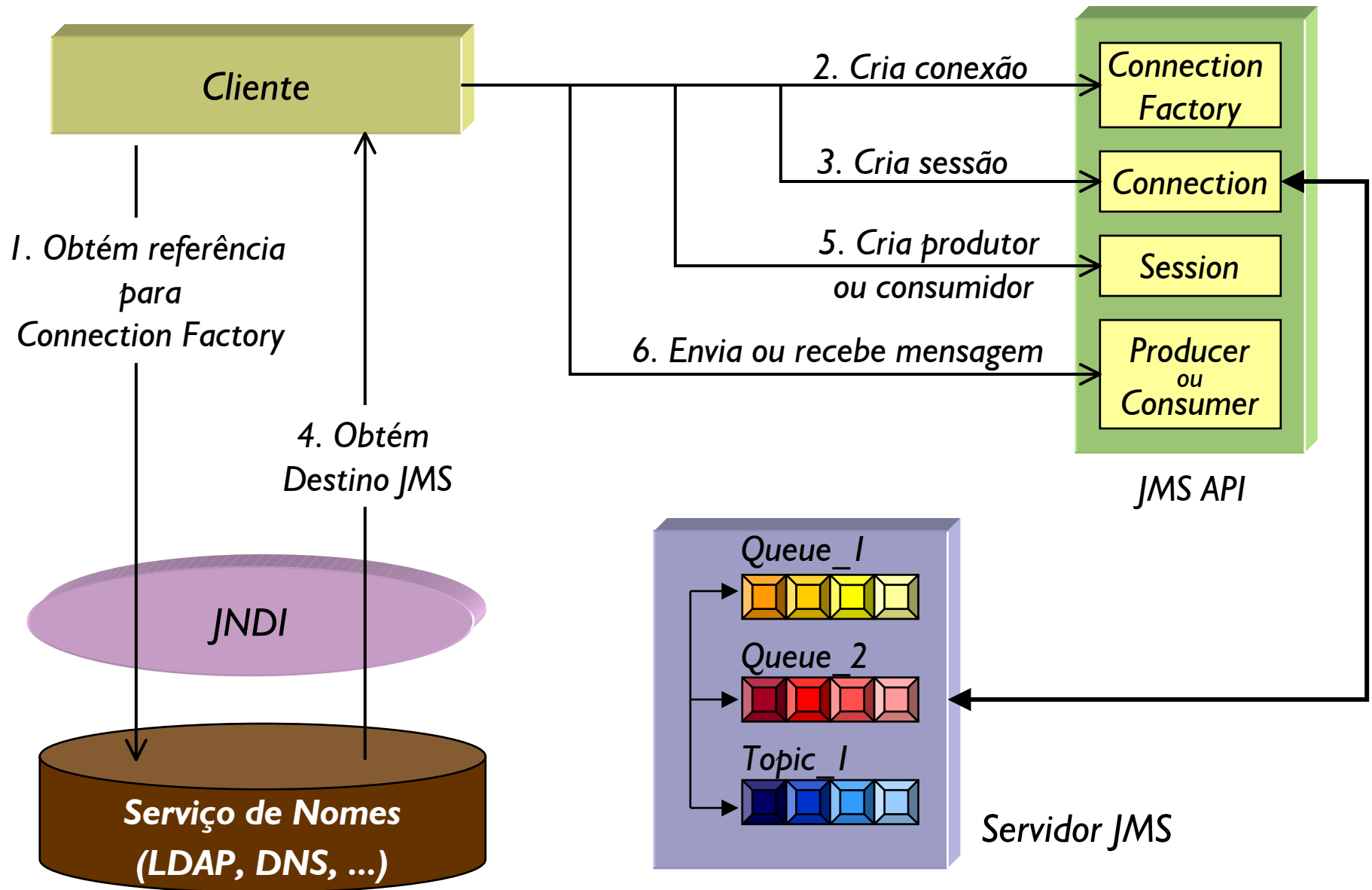
- Uma vez criado o produtor, ele pode ser usado para enviar mensagens

- *send()* no domínio ponto-a-ponto
- *publish()* no domínio pub/sub

```
sender.send(message);  
publisher.publish(message);
```

➔ Durante o envio cliente pode definir **qualidade do serviço** como modo (persistente ou não), prioridade (sobrepõe comportamento FIFO) e tempo de vida (TTL)

Envio e recebimento de mensagens



Qualidade do Serviço

- **Persistência: *Guaranteed Message Delivery***
 - Mensagens persistem em banco de dados, arquivo, etc. até que seja enviada a um consumidor e ele confirme o correto recebimento
- **Variações de GMD (depende do fabricante)**
 - ***Certified Message Delivery***: não só garante a entrega como gera um recibo de consumo enviado de volta ao criador da mensagem
 - ***Store and Forward***: permite que um produtor de mensagem envie uma mensagem com sucesso a um sistema MOM inativo
- **Prioridades: 0 a 9**
 - Uma fila FIFO para cada prioridade (mensagens de prioridade maior tomam a frente das de menor prioridade)
- **Tempo de Vida (*Time To Live*)**
 - No envio, pode-se definir o tempo (ms) de vida de uma mensagem
 - Ela irá expirar quando o tempo chegar ao fim: 0 = sem expiração

Consumidores de mensagens

- Objeto criado pela sessão e usado para receber mensagens
 - *QueueReceiver* e *QueueBrowser*: domínio ponto-a-ponto
 - *TopicSubscriber*: domínio pub/sub

```
QueueReceiver receiver =  
    queueSession.createReceiver(fila);  
TopicSubscriber subscriber =  
    topicSession.createSubscriber(canal);
```

- É preciso iniciar a conexão antes de começar a consumir:
`topicCon.start();`
- Depois, pode consumir mensagens de forma *síncrona* (método é o mesmo para domínios PTP e pub/sub)

```
Message queueMsg = receiver.receive();  
Message topicMsg = subscriber.receive(1000);
```
- Para consumir mensagens de forma *assíncrona* é preciso criar um *MessageListener*

MessageListener

- *Event handler que detecta o recebimento de mensagens*
 - *Para usar, implemente **MessageListener** e seu método `onMessage()`:*

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

- *Método `onMessage()` não deve deixar escapar exceções*
 - *Código acima deve estar em um bloco try-catch*
- *Para que objeto seja notificado, é preciso registrá-lo em um **QueueReceiver** ou **TopicSubscriber***

```
subscriber.setMessageListener( new MyListener() );  
topicCon.start(); // iniciar a conexão
```

- *Mensagens são compostas de três partes*
 - **Propriedades** (opcionais): pares nome/valor (nomes e valores arbitrários definidos pela aplicação); contém tipos primitivos Java (int, boolean, etc.) ou String.
 - **Cabeçalhos**: propriedades com nomes e tipos de valores padrão definidos na especificação JMS
 - **Corpo**: conteúdo que não pode ser representado através de propriedades
- *Os tipos de mensagem correspondem a formatos de dados armazenados no corpo de mensagem*
 - Texto, objetos serializados, bytes, valores primitivos, etc.
- *Mensagens são criadas a partir de uma Session*

Cabeçalhos e propriedades

- **Cabeçalhos: conjunto de propriedades (chave: valor) definidas na especificação JMS e usadas pelo sistema para identificar e rotear mensagens**
 - Chaves começam com "JMS". Ex: JMSMessageID, JMSDestination, JMSExpiration, JMSPriority, JMSType
 - A maioria são criadas automaticamente durante a chamada do método de envio (dependem dos parâmetros do método)
- **Propriedades definidas pelo programador**
 - Pode guardar informações de conteúdo textual em mensagens simples sem corpo (onde a mensagem consiste das propriedades)
 - Usadas para qualificar a mensagem e permitir sua filtragem
 - Úteis para preservar a compatibilidade com outros sistemas de messaging

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

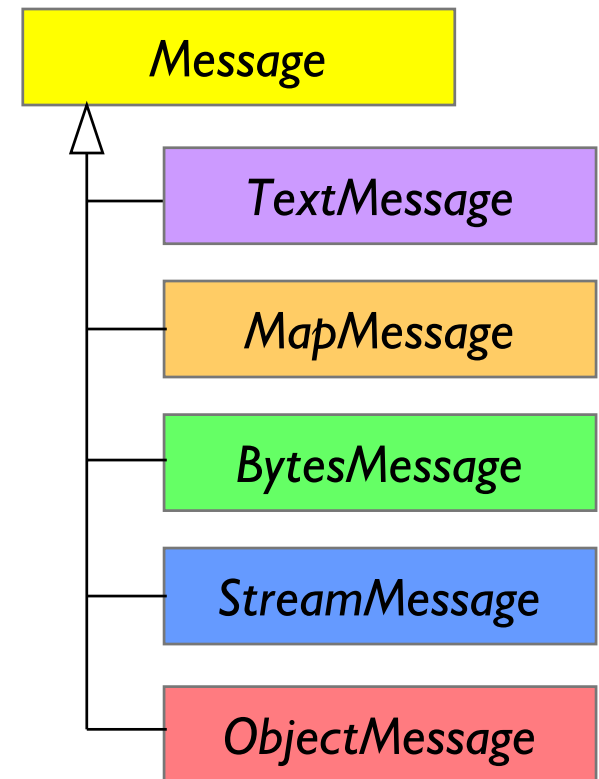

Filtros (seletores) de mensagens

- *Consumidor pode especificar quais as mensagens que lhe interessam através de expressões SQL*
 - *Expressão deve retornar valor booleano*
 - *Consiste de String passado como argumento de métodos `createReceiver()` e `createSubscriber()`*
 - *Expressão é tipicamente usada para comparar valores de propriedades de mensagens*
 - *Consumidor só consome as mensagens em que propriedades, aplicadas à expressão resultem verdadeiro*
- *Exemplos*

```
String seletor = "Formato LIKE '%Imagem%' AND " +  
                "JMSExpiration > 0 AND " +  
                "Valor IS NOT NULL";  
topicSession.createSubscriber (canal, seletor)
```

Seis tipos de mensagens

- **Message**
 - Mensagem genérica sem corpo (contendo apenas cabeçalho e possíveis propriedades)
- **TextMessage**
 - Objeto do tipo *String* (ex: conteúdo XML)
- **MapMessage**
 - Conjunto de pares nome/valor onde nomes são *Strings* e valores são tipos primitivos
- **BytesMessage**
 - Stream de bytes não interpretados
- **StreamMessage**
 - Seqüência de tipos primitivos Java
- **ObjectMessage**
 - Objeto Java serializado



Criação de mensagens

- *Para cada tipo de mensagem, Session fornece método create()*
 - `createMessage()`, `createTextMessage()`, `createBytesMessage()`, `createObjectMessage()`, `createMapMessage()`, `createStreamMessage()`

```
TextMessage message =  
    queueSession.createTextMessage();  
message.setText(msg_text); // msg_text é String  
sender.send(message);
```

- *Após receber uma mensagem, via receive() ou onMessage(), é preciso fazer o cast para ter acesso aos métodos específicos de cada tipo de mensagem*

```
Message m = receiver.receive();  
if (m instanceof TextMessage) {  
    TextMessage message = (TextMessage) m;  
    System.out.println("Recebido: " + message.getText());  
}
```

- *A maior parte das operações JMS potencialmente causam `javax.jms.JMSException`*
- *Há várias sub-classes de `JMSException` para tratar situações específicas*
 - *Veja documentação da API ou JMS Tutorial (3.7) para mais detalhes*

Exemplos de aplicações

- *Os exemplos estão no capítulo 4 do JMS Tutorial*
 - *Use-o como referência para configurar o ambiente e executar as aplicações no J2EE RI*
- *As versões em **cap04/src/** estão levemente alteradas*
 - *Todas estão nos pacotes `jmstut.topic` ou `jmstut.queue`*
 - *Use 'ant' para compilar*
 - *Use 'ant run' ou 'ant runjboss' para executar*
- *Aplicação ponto-a-ponto (no J2EE RI)*
 - > **ant run -Dclass=jmstut.queue.SimpleQueueSender**
 -Ddestino=jms/Queue
 -Dqte=5
 - > **ant run -Dclass=jmstut.queue.SimpleQueueReceiver**
 -Ddestino=jms/Queue
- *Aplicação pub/sub*
 - *Mesma coisa! Mude apenas a classe e o destino*

Configuração do ambiente

- *Sun J2EE Reference Implementation*
 - *Consulte o JMS Tutorial, capítulo 4.*
- *JBossMQ*
 - *Inicie o servidor*
 - *Rode o ant (slide anterior) usando a target 'runjboss'*
- *Nomes JNDI para destinos default*
 - *queue/testQueue*
 - *topic/testTopic*
- *Nomes JNDI para fábricas default*
 - *ConnectionFactory* (usado para os dois domínios)
- *Veja o build.xml e build.properties para detalhes sobre configuração*
- *Quer criar novos tópicos, fábricas?*
 - *http://localhost:8082*

- [1] Kim Haase. *The JMS Tutorial*. Sun Microsystems, 2002
<http://java.sun.com/products/jms/tutorial/> Este capítulo é baseado no JMS Tutorial.
- [2] Sun Microsystems. *The Java Message Service Specification*
<http://www.javasoft.com/products/jms/docs.html>
- [3] Peter Antman. *JBoss and JMS*. JBoss User's Manual, Chapter 8.
<http://www.jboss.org/online-manual/HTML/ch08.html> Contém breve tutorial sobre JMS
- [4] Todd SundSted. *Messaging helps move Java into the Enterprise*. JavaWorld, Jan 1999.
<http://www.javaworld.com/jw-01-1999/jw-01-jms.html>
Introdução aos conceitos fundamentais de sistemas de mensagens.
- [5] Todd SundSted. *Messaging makes its move, Parts I & II*. JavaWorld, Fev e Mar 1999.
http://www.javaworld.com/javaworld/jw-02-1999/jw-02-howto_p.html
Série de dois artigos mostrando como usar JMS.
- [6] Gordon Van Huizen. *JMS: An infrastructure for XML-based B2B communication*
<http://www.javaworld.com/javaworld/jw-02-2000/jw-02-jmsxml.html> JavaWorld, 2000.
Introdução a JMS motivada pelo seu potencial para soluções B2B com XML
- [7] Michael Schoffner. *Write your own MOM*. JavaWorld, 1998.
<http://www.javaworld.com/javaworld/jw-05-1998/jw-05-step.html>
- [8] Ed Roman et al. *Mastering EJB 2, Chapter 8: Message Drive Beans*
<http://www.theserverside.com/books/masteringEJB/index.jsp>
Contém um breve e objetivo tutorial sobre JMS no início do capítulo

helder@acm.org

argonavis.com.br

© 2001, 2002 Helder da Rocha

J533 - Minicurso de Java Message Service

Revisão 1.1

Setembro 2002