

Como Implementar um Middleware?

Parte IV

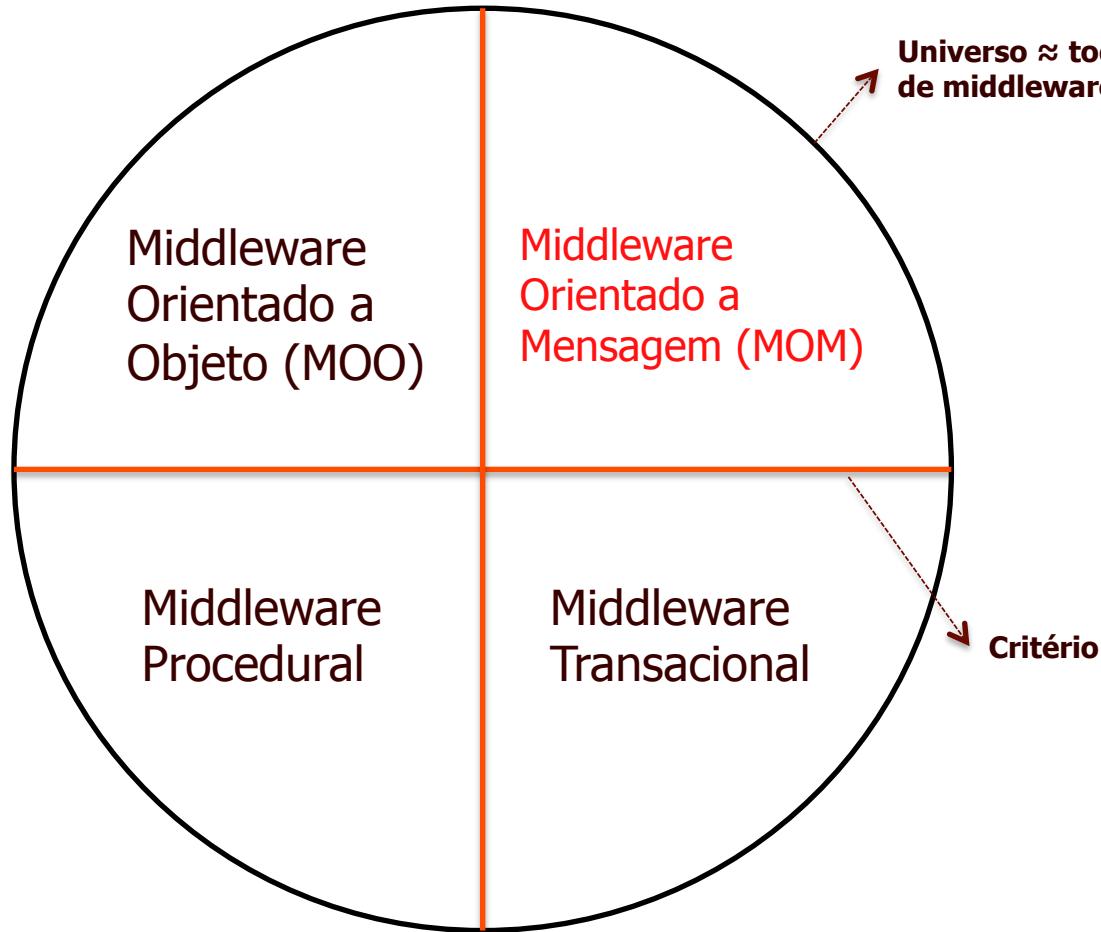
Nelson Rosa – nsr@cin.ufpe.br

Objetivos

- **Introduzir conceitos básicos de Middleware Orientado a Mensagem**
- **Apresentar uma estratégia de implementação de MOM**

A estória até agora...

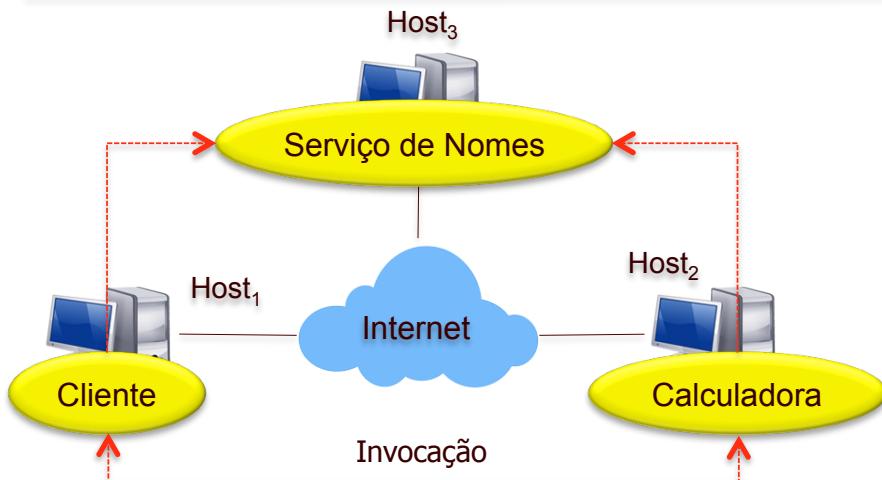
Critério: tipo de primitiva de comunicação fornecida pelo middleware para o desenvolvimento de aplicações distribuídas.



IMPORTANTE:

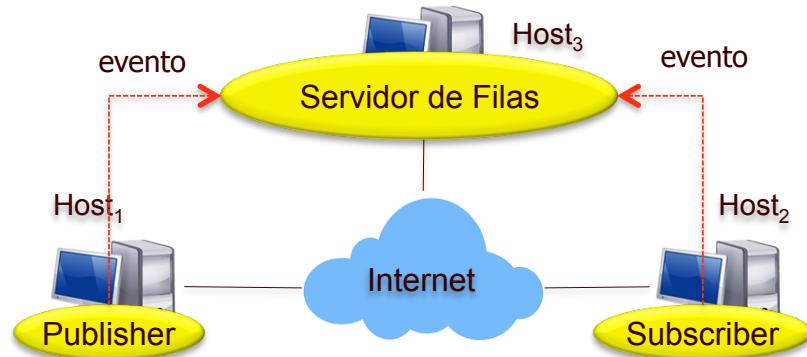
- Os termos **Publish/Subscibe** e **MOM** são geralmente usados sem muita distinção para se referir aos modelos de middleware orientados a mensagem.
- MOMs usam o modelo de interação Publish/Subscribe.

A estória a partir de agora...



Middleware Orientado a Objetos

- ✓ Comunicação síncrona [tipicamente]
- ✓ Comunicação 1-to-1
- ✓ Comunicação direta
- ✓ Aplicações “processing-centric” [tipicamente]
- ✓ Mensagem ≈ Invocação
- ✓ Forte acoplamento
- ✓ e.g., RMI



Middleware Orientado a Mensagem

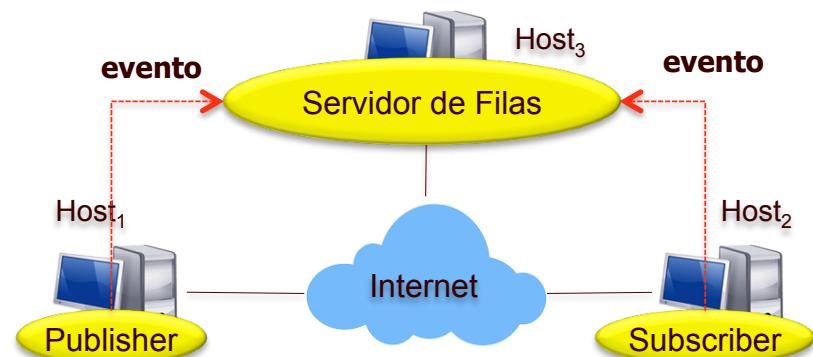
- ✓ Comunicação assíncrona [tipicamente]
- ✓ Comunicação 1-to-1 e 1-to-N
- ✓ Comunicação através de fila
- ✓ Comunicação anônima
- ✓ Aplicações “data-centric” [tipicamente]
- ✓ Mensagem ≈ evento
- ✓ Fraco acoplamento
- ✓ e.g., JMS (Java Message Service)

IMPORTANTE: Os termos “Servidor de Eventos” e “Servidor de Mensageria” também são utilizados para se referir ao “Servidor de Filas”.

MOM:: Elementos

■ Evento

- Um evento é uma **informação**
- As **interações** entre componentes da aplicação distribuída são **baseadas** em **eventos**
- Eventos são gerados por **publishers** (ou produtores) e propagados assincronamente a todos os **subscribers** (ou consumidores) que expressaram o interesse pelo evento
- Publishers/subscribers são totalmente desacoplados no **tempo, espaço e sincronização**



MOM:: Elementos

■ Subscribers

- Expressam o interesse em **eventos ou padrões de eventos**
- São **notificados** sobre a ocorrência dos eventos
- **Consumem** eventos

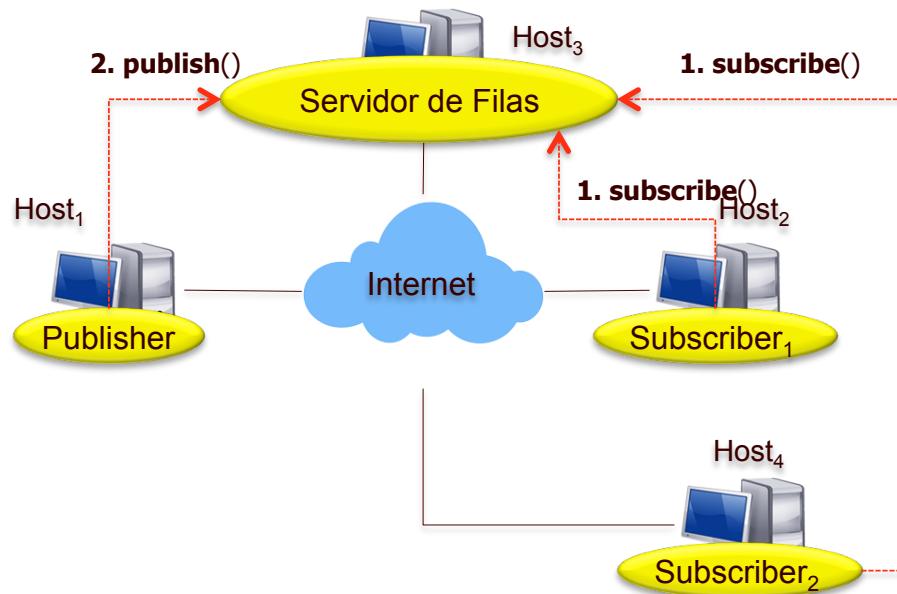
■ Publishers

- **Produzem** eventos

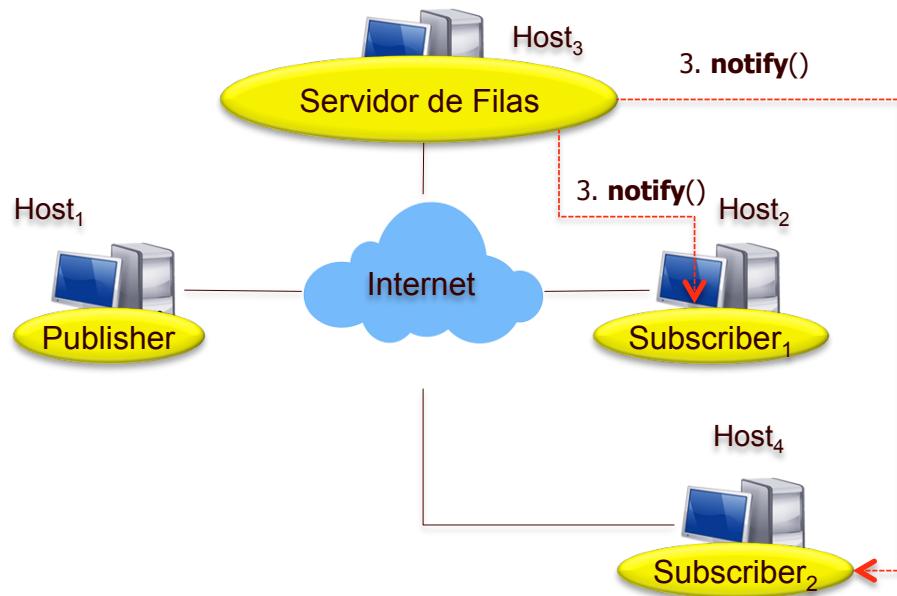
■ Serviço de Eventos/Notificação/Servidor de Filas

- Armazena e gerencia subscrições
- Envia eventos

MOM:: Interações



MOM:: Interações



MOM:: Desacoplamento

■ **Espacial**

- Componentes da aplicação distribuída **não se conhecem**
- Publishers (Subscribers) **não mantém referência** sobre Subscribers (Publishers)

■ **Temporal**

- Componentes da aplicação distribuída não precisam participarativamente da interação ao **mesmo tempo**
- Publisher pode publicar um evento enquanto o **Subscriber** está **desconectado**
- Subscriber pode ser notificado de um evento enquanto o **Publisher** está **desconectado**

■ **Assincronia**

- Publishers **não** são bloqueados enquanto produzem eventos
- Subscribers são notificados de forma **assíncrona** enquanto executam alguma tarefa

■ **Consequências do desacoplamento**

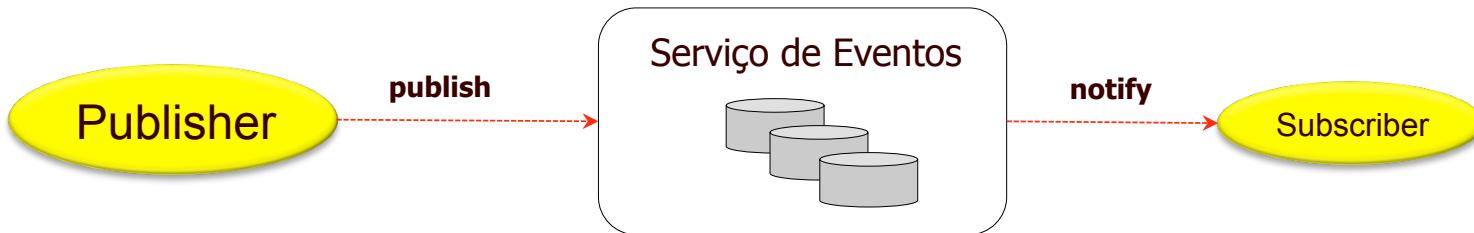
- Aumento da escalabilidade
- Modelo de interação bom para **ambientes móveis** e para **integração de sistemas entre empresas diferentes**

MOM:: Desacoplamento

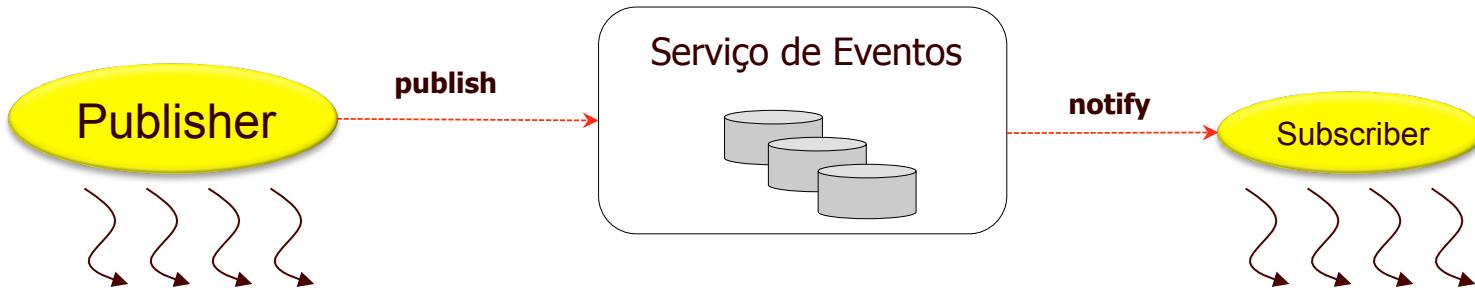
Desacoplamento Espacial



Desacoplamento Temporal



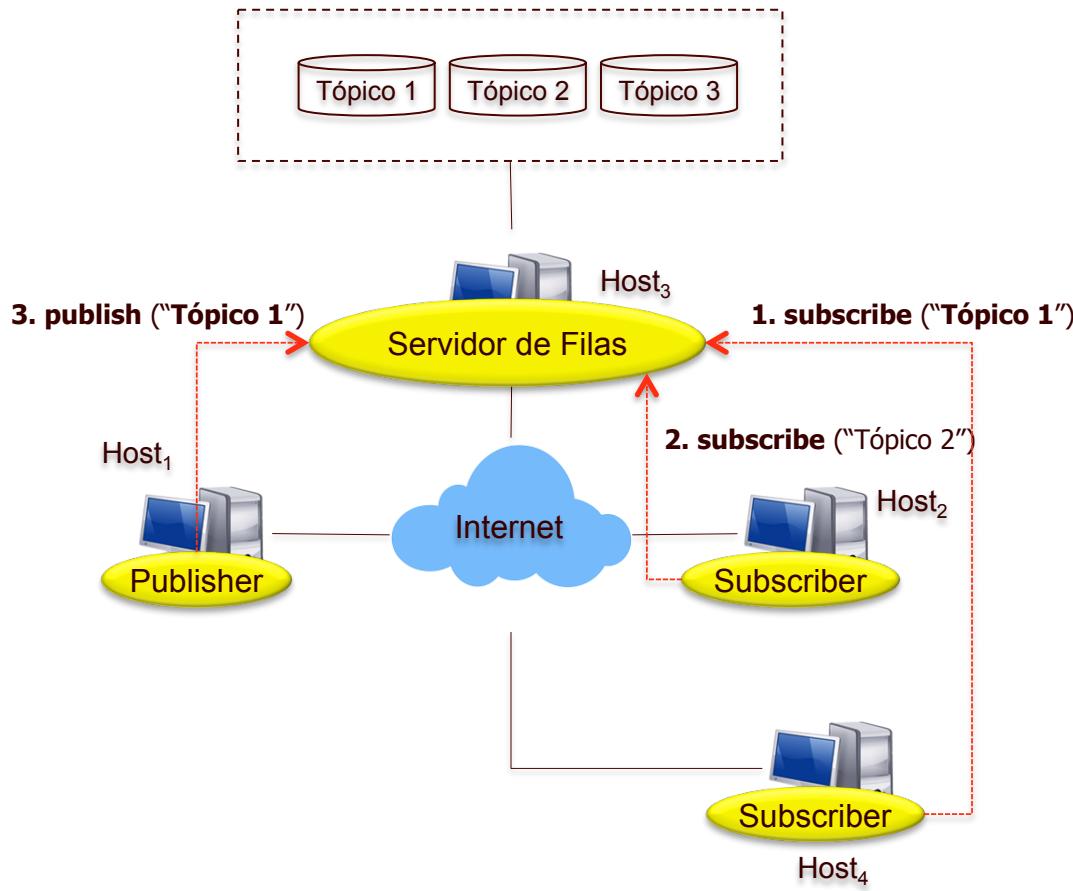
Assincronia



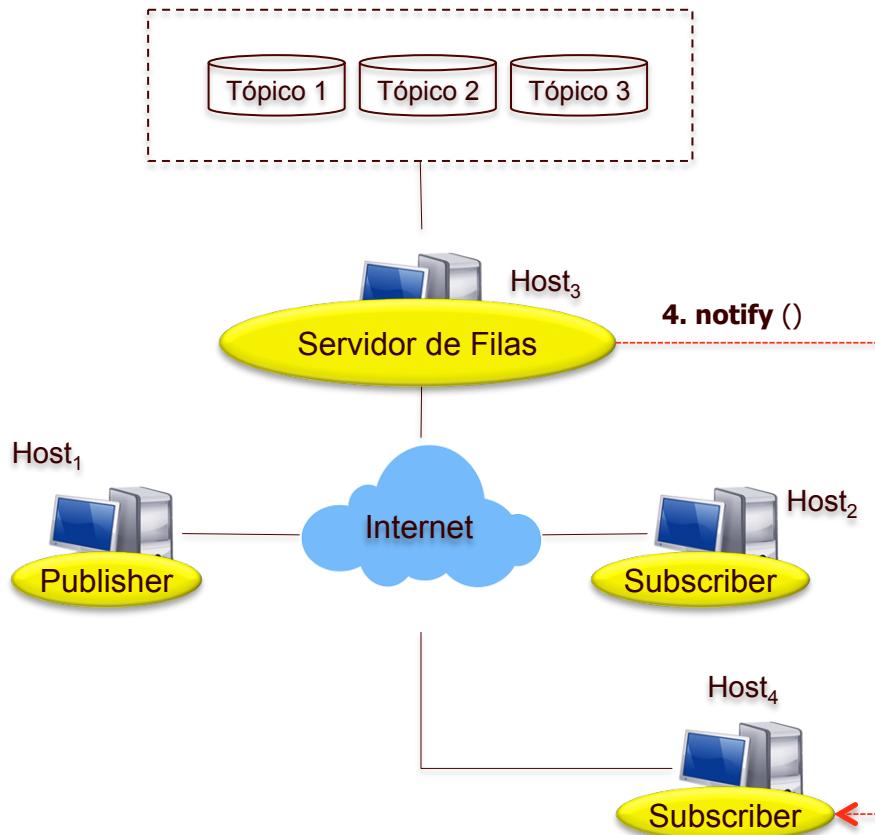
MOM:: Variações:: Tópico

- “tópico” ≈ assunto
- Baseado na caracterização do conteúdo (assunto) das mensagens
 - “esta mensagem é de um determinado assunto”
 - Publishers geram mensagens de um determinado tópico
 - Subscribers indicam o interesse em receber mensagens de um determinado tópico
- Pode ser criada uma hierarquia de tópicos

MOM:: Variações:: Tópico



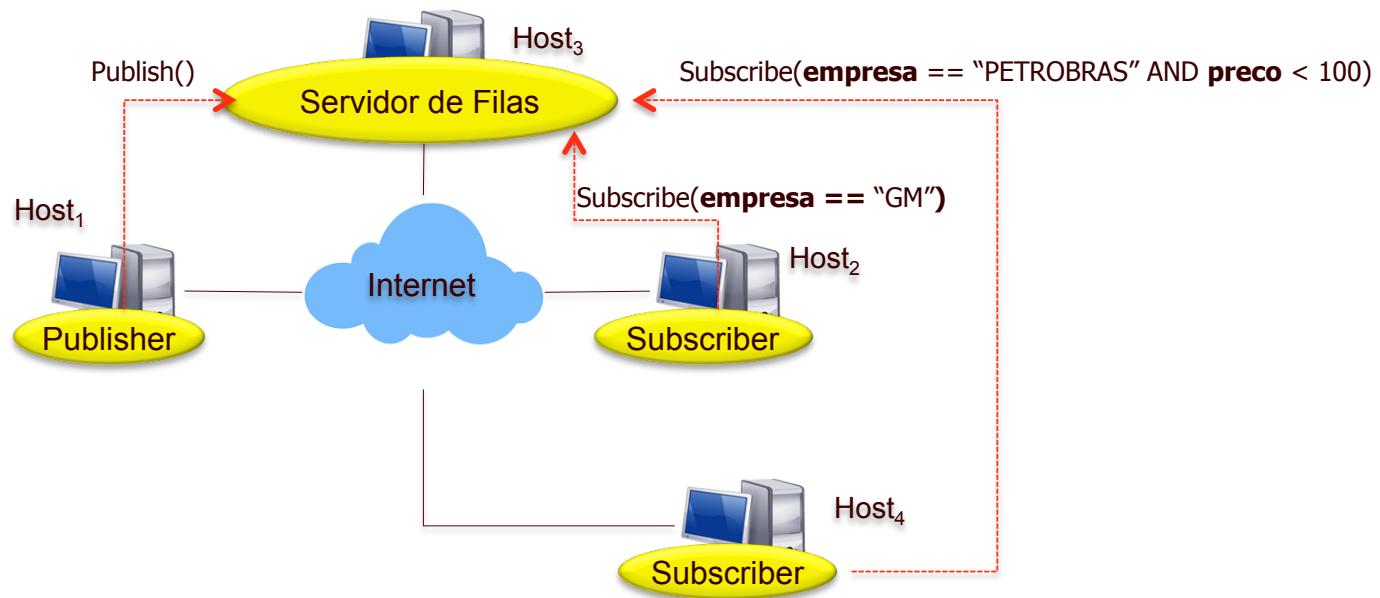
MOM:: Variações:: Tópico



MOM:: Variações:: Conteúdo

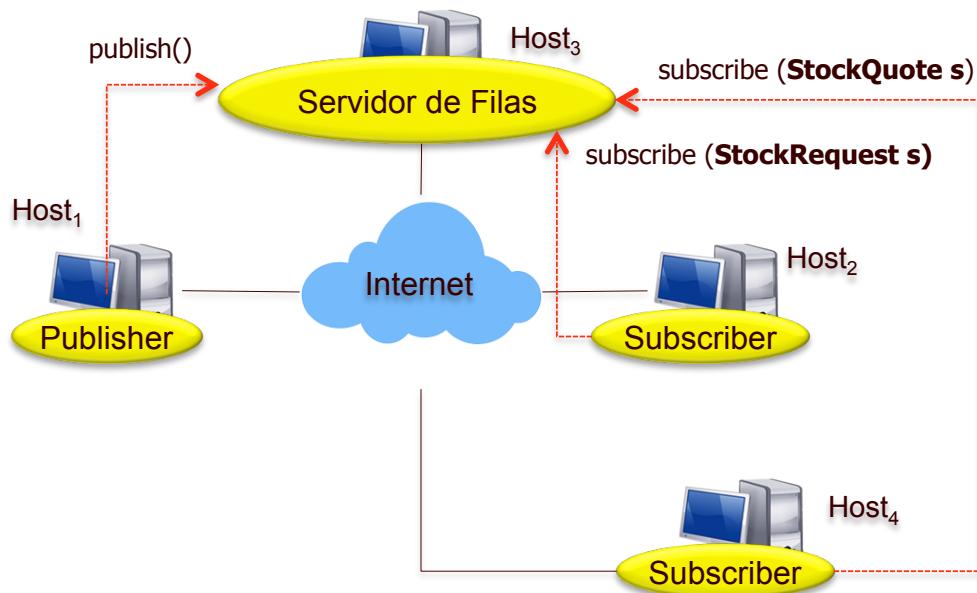
- Melhoria do publish/subscribe baseado em tópico
- Subscribers definem filtros aplicados ao conteúdo da mensagem
- Mensagens são caracterizadas pelas suas próprias propriedades
- Subscribers podem indicar um interesse numa “combinação de mensagens” (padrão de mensagens)
- Como representar os padrões de mensagem?
 - Strings: empresa == “PETROBRAS” AND preco < 100
 - Templates: mensagens devem “casar” com os atributos de um objeto
 - Código executável: uso de objeto capaz de filtrar eventos em tempo de execução

MOM:: Variações:: Conteúdo



MOM:: Variações:: Tipo

- Assume que mensagens tem similaridades não apenas no conteúdo, mas também na estrutura
- Mensagens são filtradas pelo tipo



MOM:: Implementação:: Eventos

■ Tipos de eventos

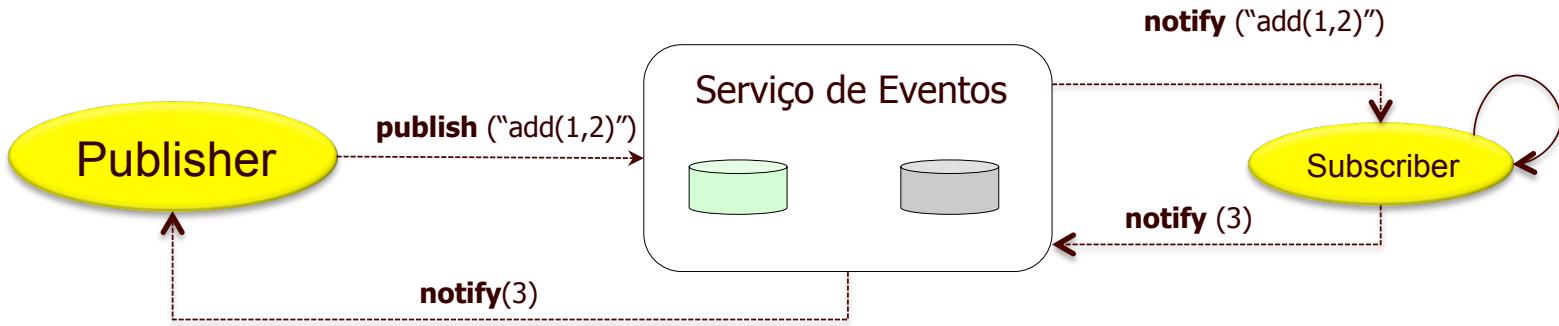
- Mensagens
 - Eventos são enviados ao subscriber através de uma notificação (~**não tem retorno**)
 - Eventos são criados pela aplicação
 - Formato da mensagem: **Header** (e.g., identificador, prioridade, tempo de expiração, transmissor) + **Payload** (informação da mensagem)
- Invocações
 - Eventos “disparam” a **execução de uma operação** no Subscriber (~”MOM orientado a objetos”)
 - Subscriber tem uma interface com a operação invocada

MOM:: Implementação:: Eventos

Evento (Mensagem)



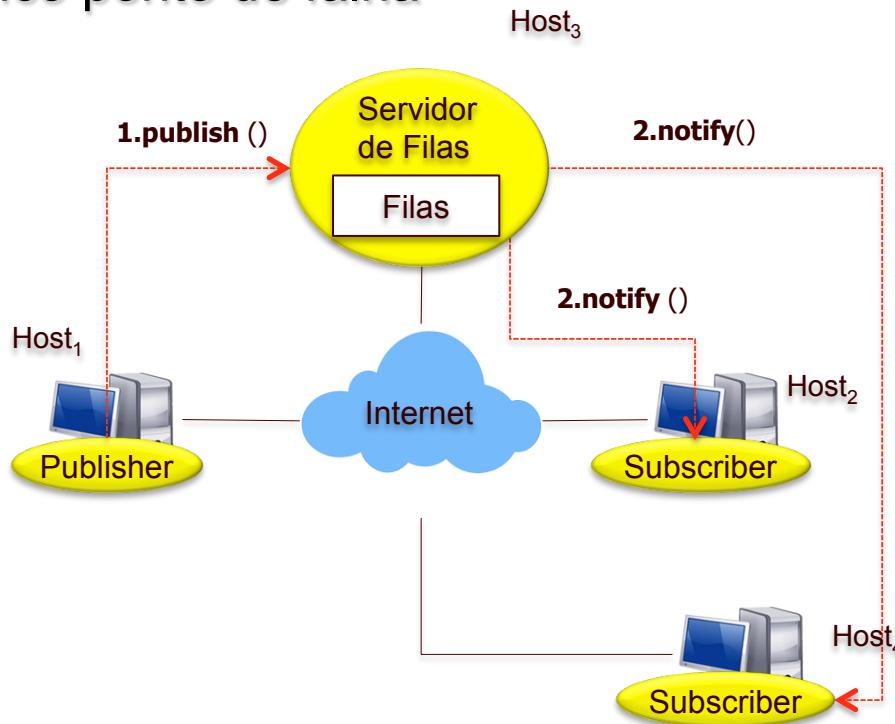
Evento (Invocação)



MOM:: Meio de Transporte (servidor de fila)

■ Centralizado

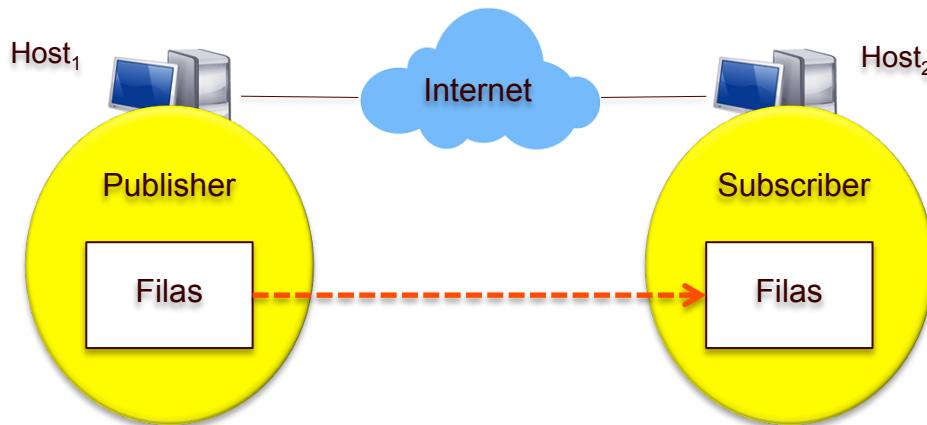
- Publishers enviam mensagens para uma “entidade” que armazena e encaminha as mensagens aos Subscribers
- Único ponto de falha



MOM:: Meio de Transporte

■ Distribuído

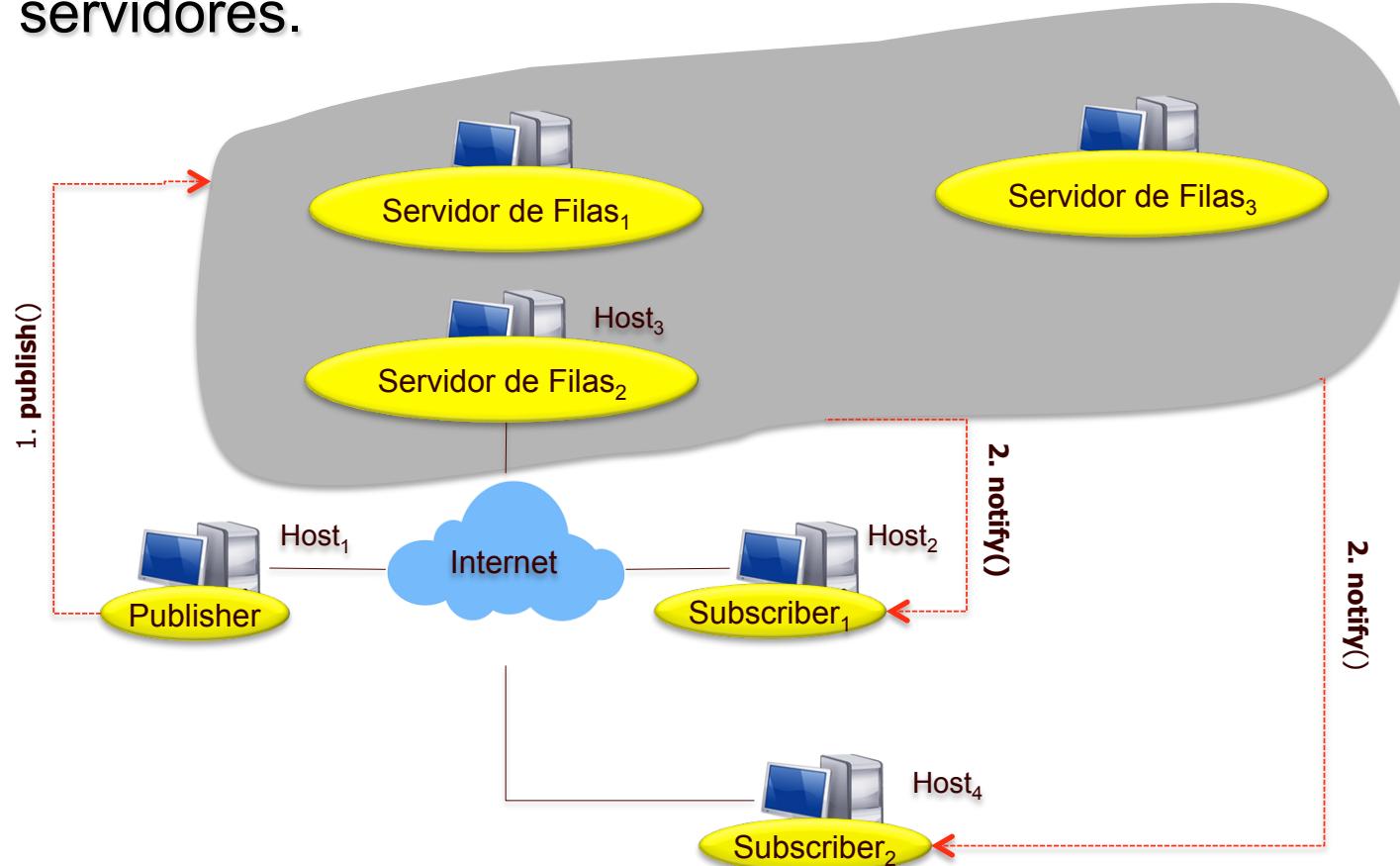
- Transporte implementado no Publisher e no Subscriber



MOM:: Fatos Básicos:: Meio de Transporte

■ Híbrido

- Implementar o serviço de filas através de uma rede de servidores.



MOM:: Qualidade de Serviço

■ Persistência

- **MOO**: invocações tem um tempo de vida curto
- **MOM**: mensagens podem ser transmitidas (sem reply) e serem processadas horas depois do envio
- **MOM** precisa garantir a **confiabilidade** e a **durabilidade** das mensagens
- **MOMs** podem usar **SGBDs** para persistir as mensagens
- Reenvio de mensagens (se necessário)

■ Prioridades

- Mensagens podem ser **ranqueadas**, e.g., uma mensagem de notificação de falha pode precisar ter prioridade sobre outras mensagens

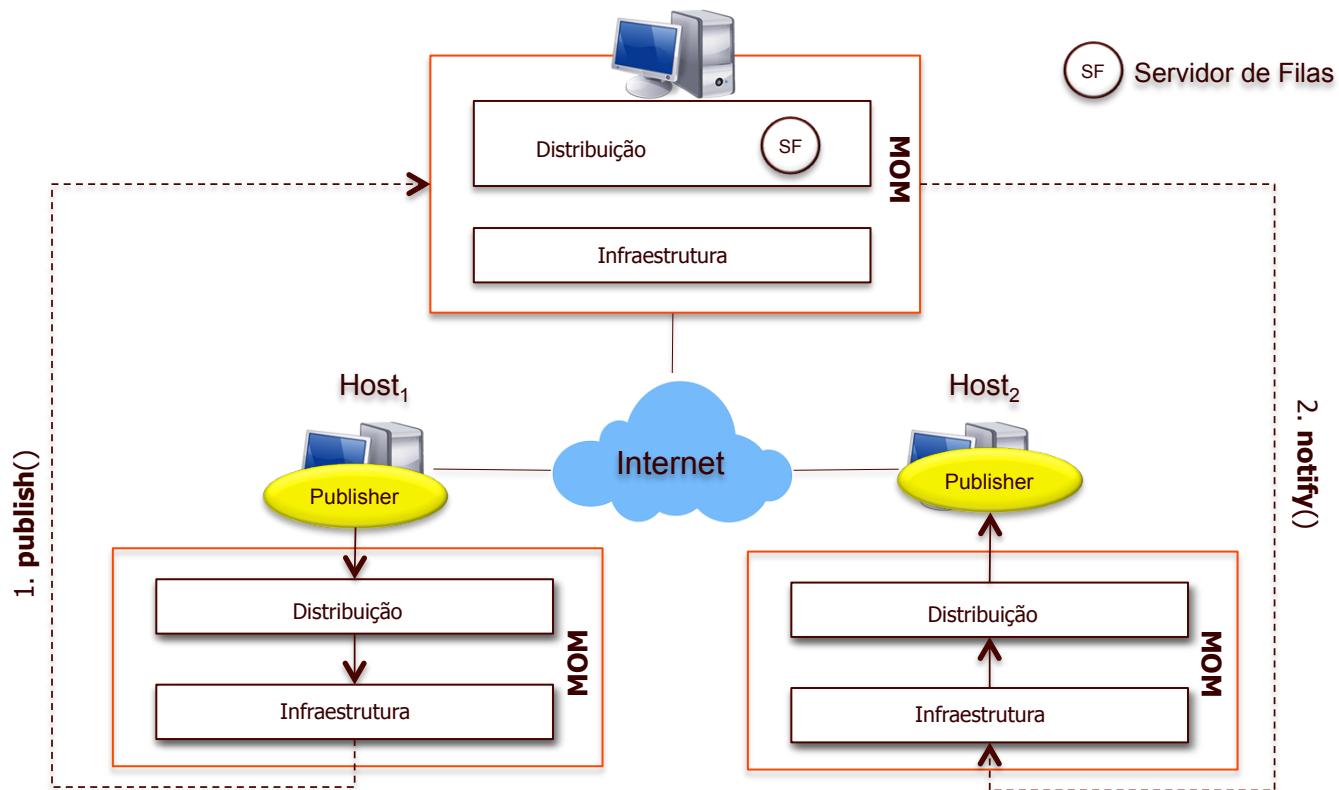
■ Transação

- Mensagens podem ser **agrupadas** em unidades atômicas
 - e.g., Mensagens semanticamente relacionadas precisam ser consumidas por completo

■ Confiabilidade

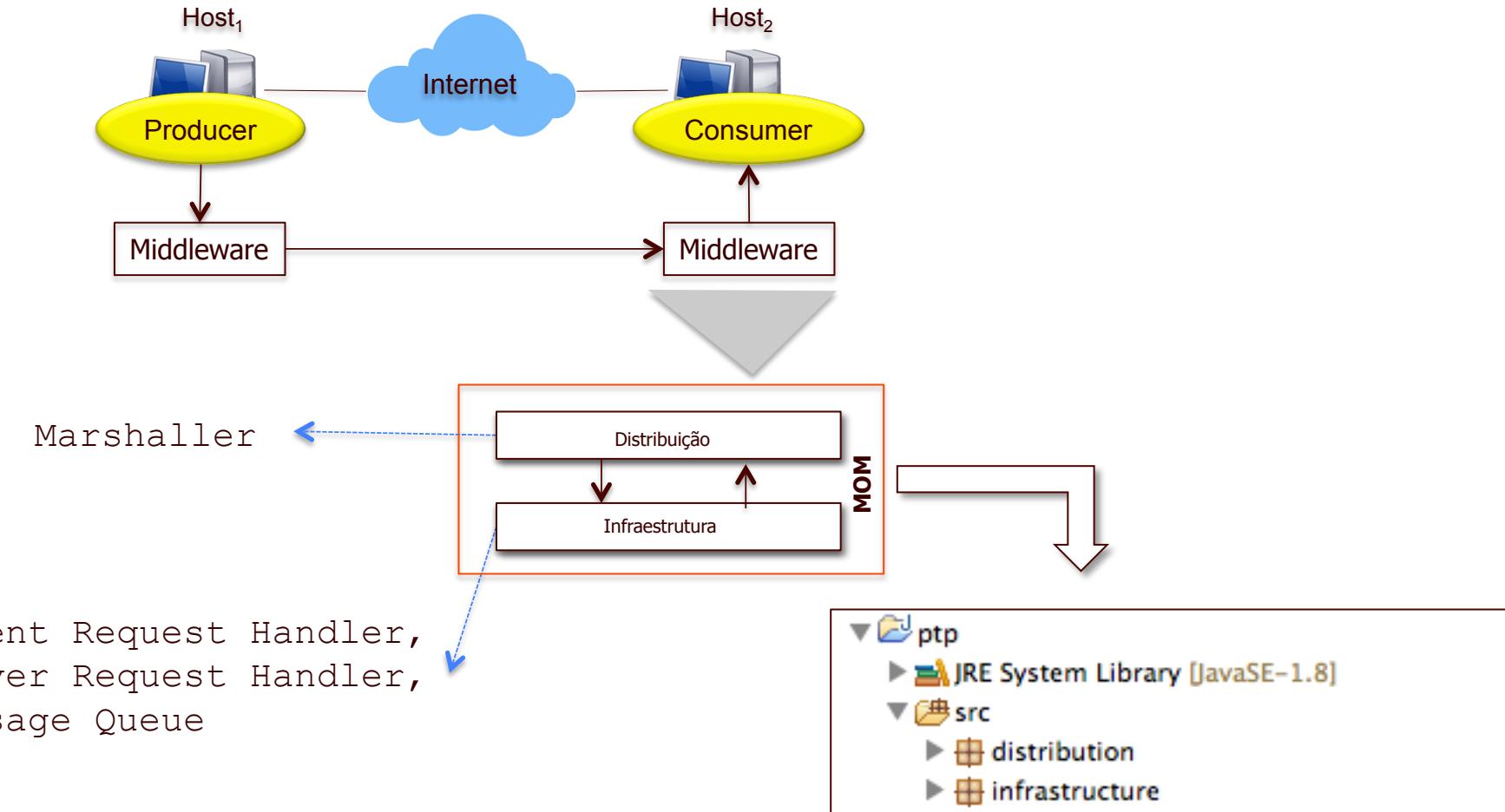
- MOM precisa **garantir** o envio das mensagens ao destinatário
- Uso de canais de **comunicação ponto-a-ponto confiáveis**
- **Manter** cópia das mensagens em um local de **armazenamento** estável

MOM:: [possível] Arquitetura

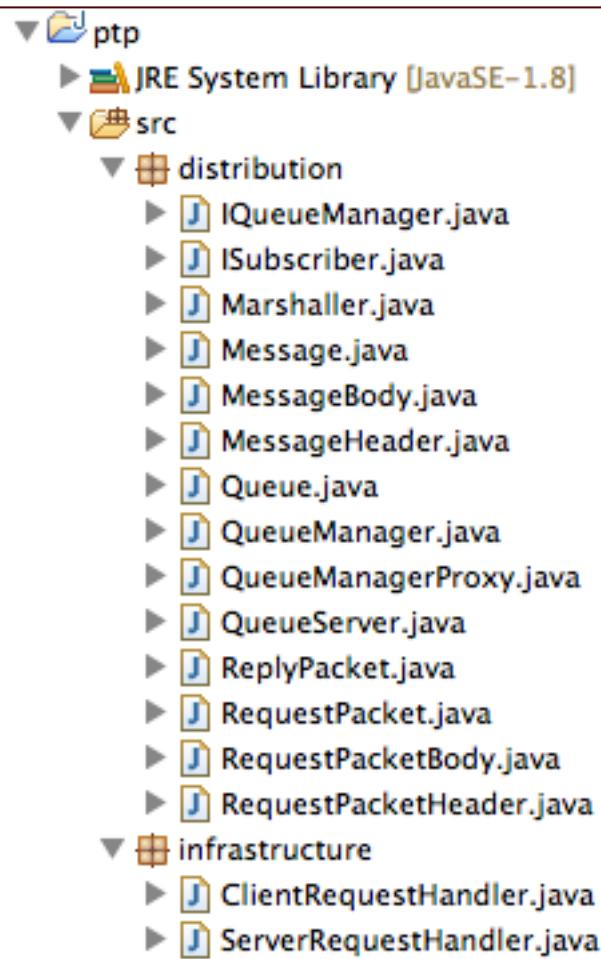


Implementação

MOM:: Arquitetura

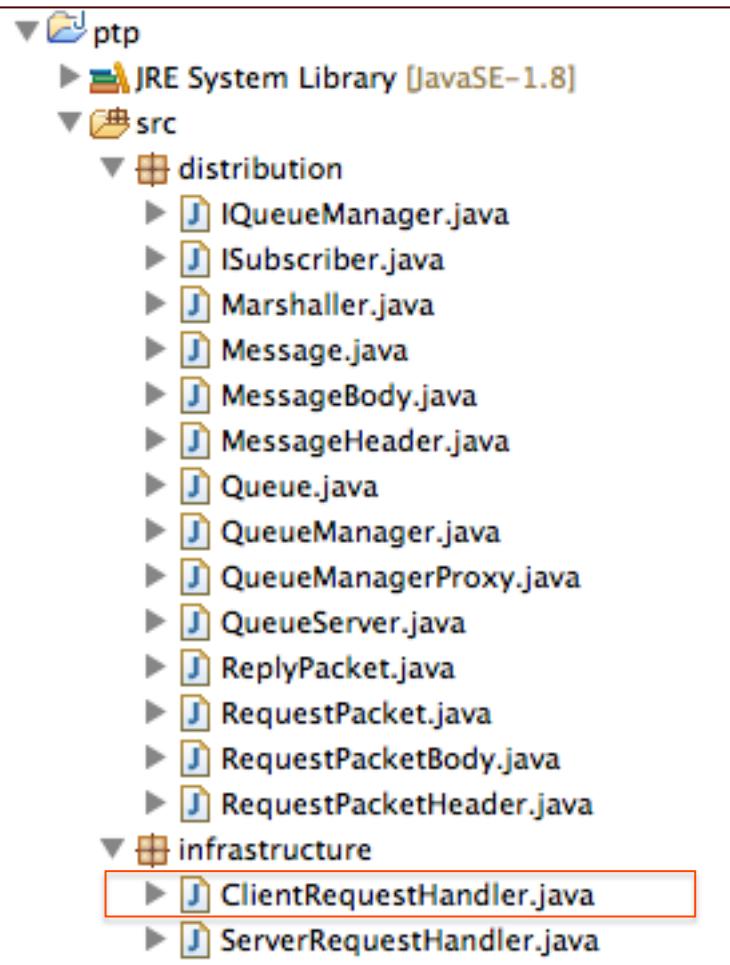


MOM:: Implementação



- ✓ **Reuso** da implementação MOO dos Remoting Patterns da Camada de Infraestrutura
- ✓ Reuso do padrão **Marshaller** para serialização
- ✓ Comunicação **1:1** (produtor -> consumidor)
- ✓ Comunicação **assíncrona**
- ✓ Servidor de filas **centralizado**
- ✓ Formato da **mensagem** simplificado
- ✓ Uso de um **proxy** para acesso ao servidor de filas.
- ✓ **IMPORTANTE:** Esta é apenas uma implementação entre as várias possíveis.

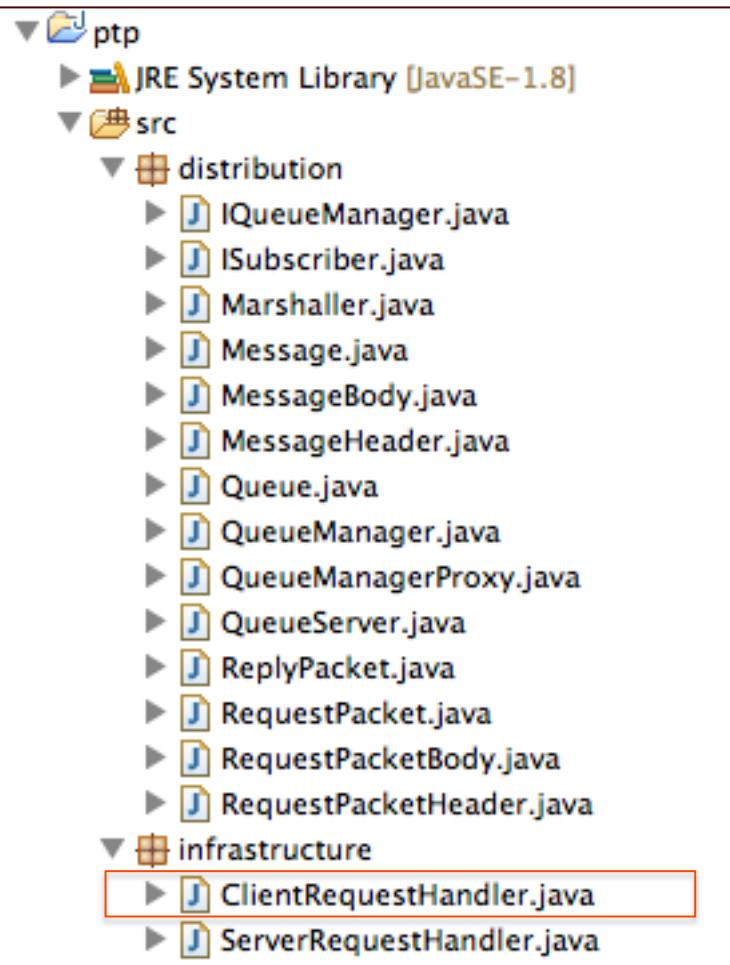
MOM:: Implementação:: ClientRequestHandler



```
8 public class ClientRequestHandler {  
9     private String host;  
10    private int port;  
11    private boolean expectedReply;  
12  
13    int sentMessageSize;  
14    int receiveMessageSize;  
15  
16    Socket clientSocket = null;  
17    DataOutputStream outToServer = null;  
18    DataInputStream inFromServer = null;  
19  
20+   public ClientRequestHandler(String host, int port, boolean expectedReply) {}  
25  
26+   public void send(byte[] msg) throws IOException, InterruptedException {}  
46  
47+   public byte[] receive() throws IOException, InterruptedException, {}  
63  
64+   public boolean isExpectedReply() {}  
...  
...
```

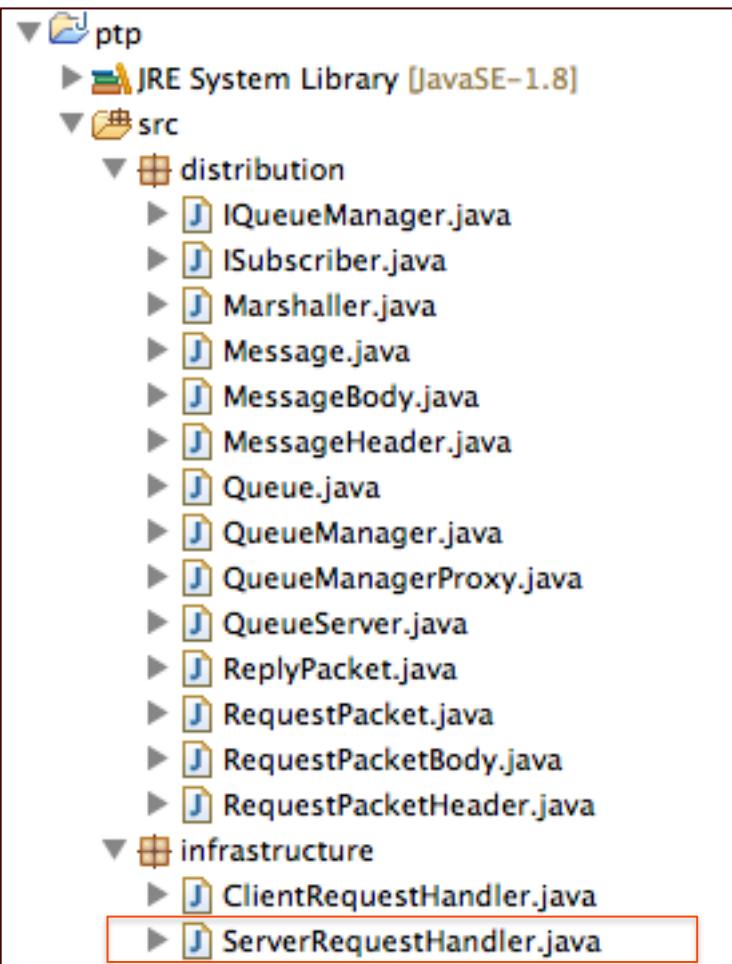
- ✓ A implementação MOO do ClientRequestHandler foi estendida com o método “**isExpectedReply**” para permitir “send” **sem** que haja “**reply**”.
- ✓ O método “send” do MOO sempre tem reply.

MOM:: Implementação:: ClientRequestHandler



```
26     public void send(byte[] msg) throws IOException, InterruptedException {  
27  
28         clientSocket = new Socket(this.host, this.port);  
29         outToServer = new DataOutputStream(clientSocket.getOutputStream());  
30         inFromServer = new DataInputStream(clientSocket.getInputStream());  
31  
32         sentMessageSize = msg.length;  
33         outToServer.writeInt(sentMessageSize);  
34         outToServer.write(msg, 0, sentMessageSize);  
35         outToServer.flush();  
36  
37         // close socket if no reply is expected  
38         if (!this.expectedReply) {  
39             clientSocket.close();  
40             outToServer.close();  
41             inFromServer.close();  
42         }  
43  
44         return;  
45     }
```

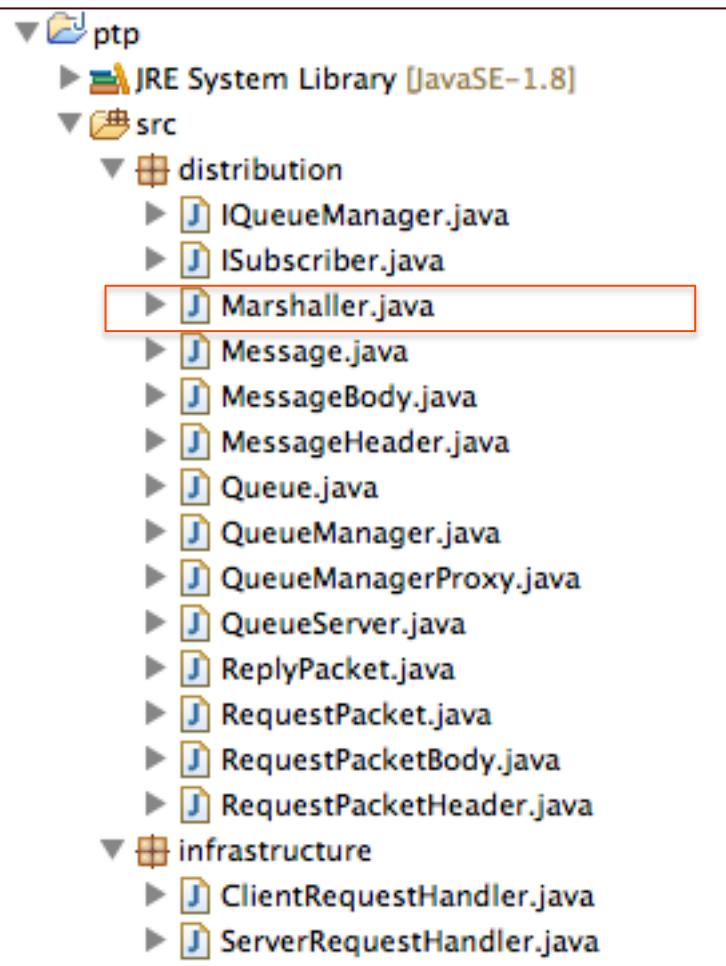
MOM:: Implementação:: ServerRequestHandler



```
9  public class ServerRequestHandler {  
10     private int port;  
11     private ServerSocket welcomeSocket = null;  
12  
13     Socket connectionSocket = null;  
14  
15     int sentMessageSize;  
16     int receivedMessageSize;  
17     DataOutputStream outToClient = null;  
18     DataInputStream inFromClient = null;  
19  
20+    public ServerRequestHandler(int port) throws IOException {}  
24  
25+    public byte [] receive() throws IOException, Throwable {}  
41  
42+    public void send(byte [] msg) throws IOException, InterruptedException {}  
55  
56 }
```

- ✓ A implementação MOO do ServerRequestHandler foi **reusada integralmente**.

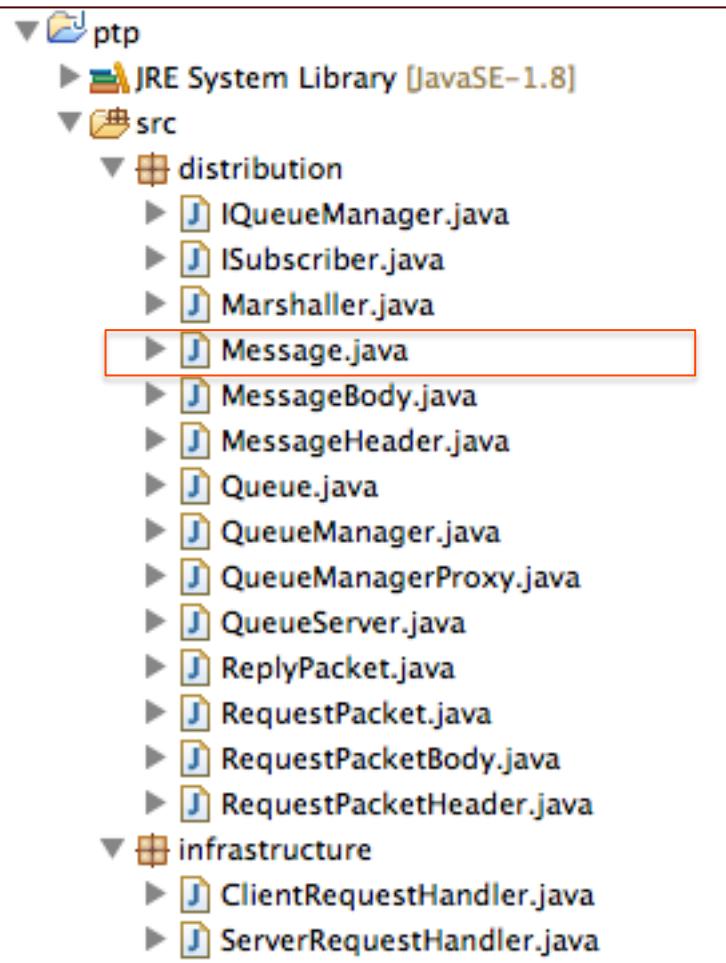
MOM:: Implementação:: Marshaller



```
9  public class Marshaller {  
10  
11+     public byte[] marshall(Object msgToBeMarshalled) throws IOException, E..  
20  
21+     public Object unmarshall(byte[] msgToBeUnmarshalled) { ..  
41 }
```

- ✓ A implementação MOO do Marshaller foi **reusada integralmente**.

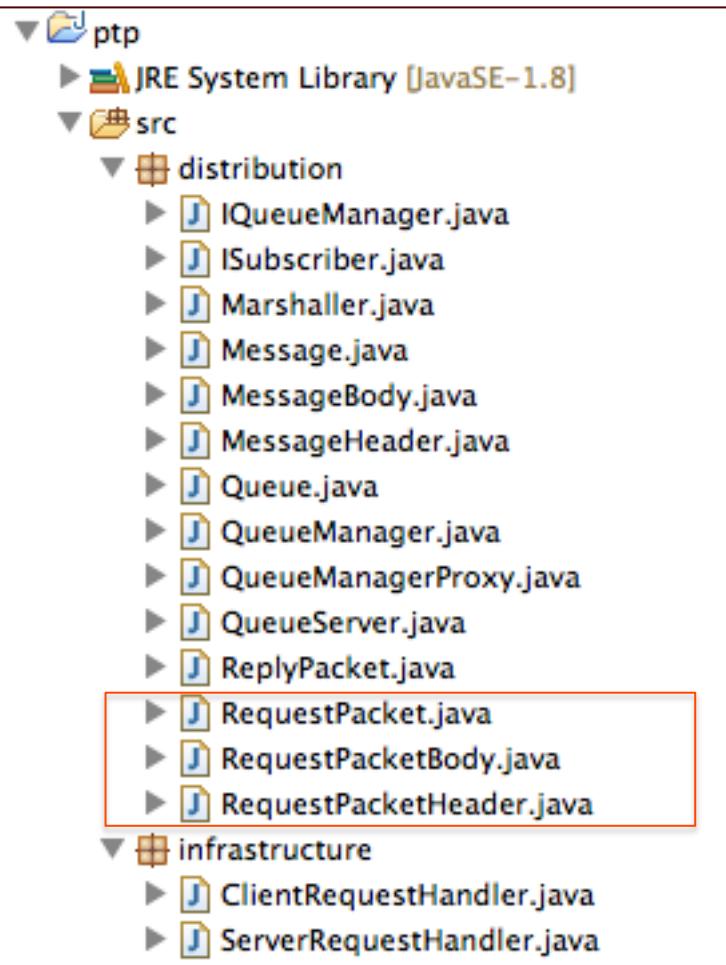
MOM:: Implementação:: Message



```
5 public class Message implements Serializable {  
6     private MessageHeader header;  
7     private MessageBody body;  
  
5 public class MessageHeader implements Serializable {  
6     private String destination;  
  
5 public class MessageBody implements Serializable {  
6     private String body;
```

- ✓ O nome da fila onde a mensagem é armazenada é o **“destination”**
- ✓ Apenas um tipo de mensagem é possível: **String**.
- ✓ A mensagem é colocada dentro de um **“packet”**.

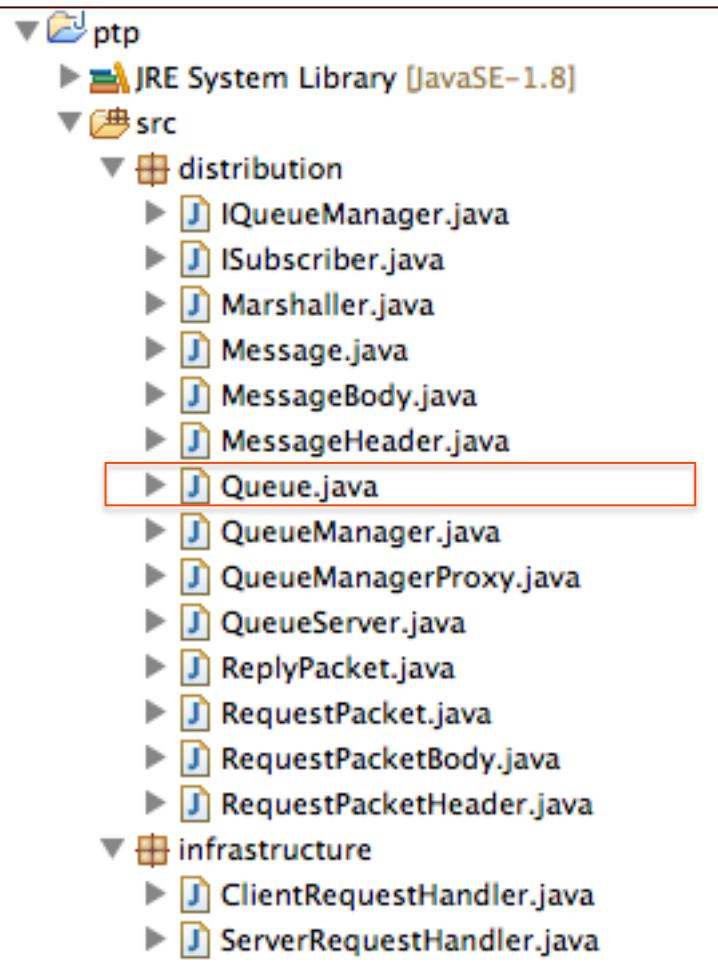
MOM:: Implementação:: Packet



```
5 public class RequestPacket implements Serializable {  
6     private RequestPacketHeader packetHeader;  
7     private RequestPacketBody packetBody;  
8  
5 public class RequestPacketHeader implements Serializable {  
6     private String operation;  
7  
6 public class RequestPacketBody implements Serializable {  
7     private ArrayList<Object> parameters = new ArrayList<Object>();  
8     private Message message;
```

- ✓ Enquanto **message** é usada para comunicação entre as aplicações (produtor/consumidor), **packet** é usado para comunicação cliente/servidor dos consumidores e produtores (clientes) com o servidor de fila (servidor).

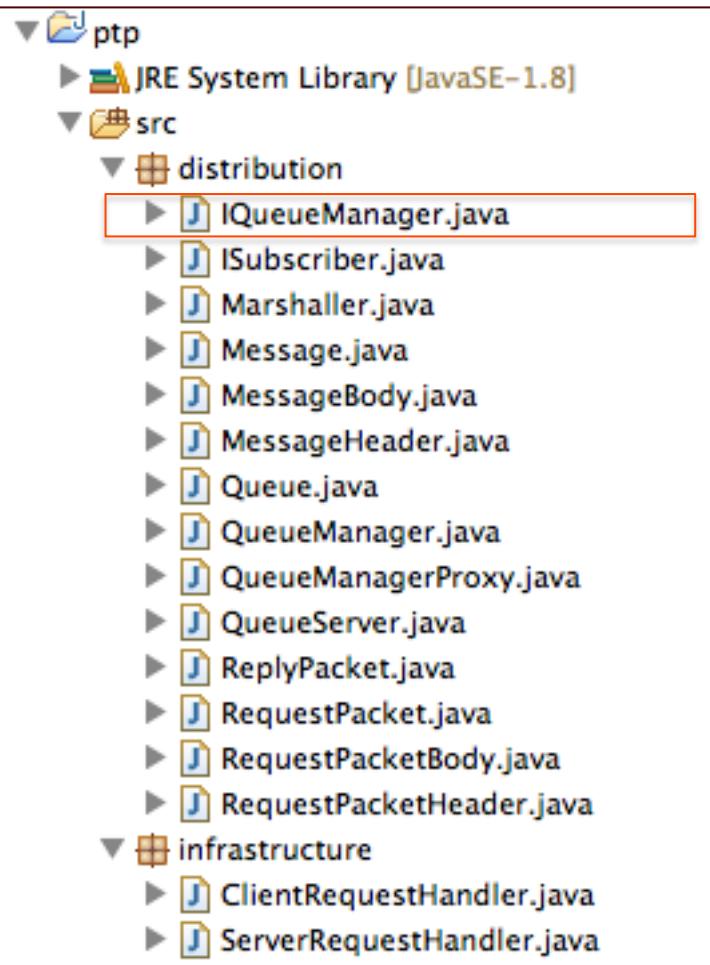
MOM:: Implementação:: Queue



```
5 public class Queue {  
6     private ArrayList<Message> queue = new ArrayList<Message>();  
7  
8     public Queue(){};  
9  
10+    public void enqueue(Message msg){}  
13  
14+    public Message dequeue(){}  
22  
23+    public int queueSize(){}  
26 }
```

- ✓ **Queue** é mantida em memória apenas.
- ✓ **Queue** é uma fila de mensagens (**Message**).
- ✓ A fila é gerenciada pelo **QueueManager**.

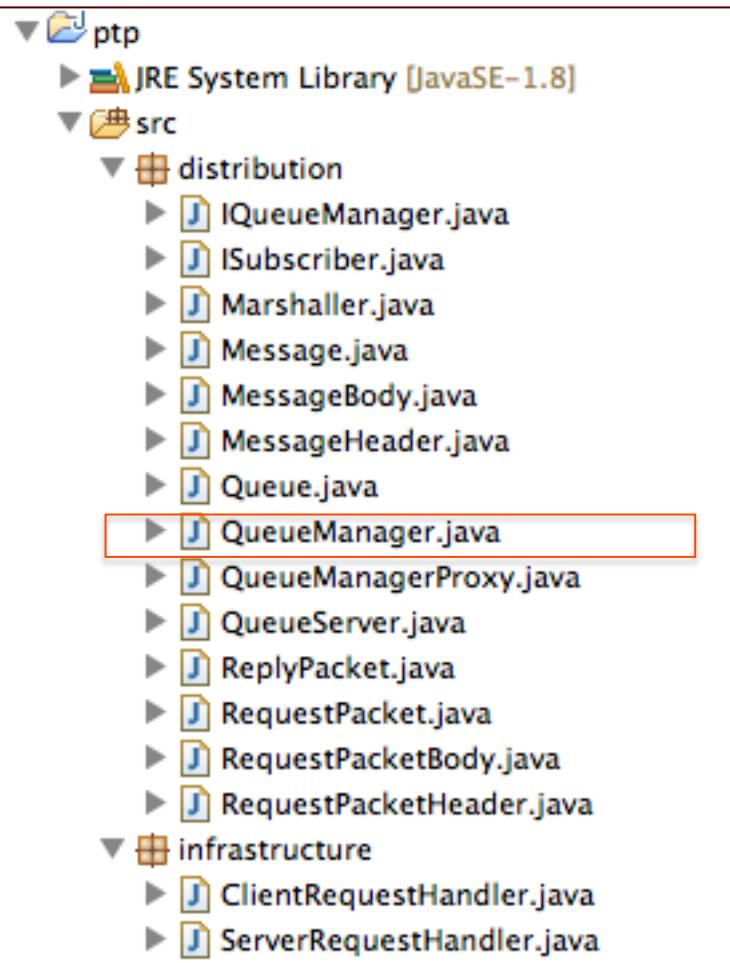
MOM:: Implementação:: QueueManager



```
5  public interface IQueueManager {  
6  
7      public void send(String msg) throws IOException, InterruptedException;  
8  
9      public String receive() throws IOException, InterruptedException,  
10         ClassNotFoundException;  
11  }  
12 }
```

- ✓ Armazena (**send**) e remove (**receive**) mensagens da fila.
- ✓ **send** é invocada pelo **produtor** de mensagem para envio de mensagens à fila.
- ✓ **receive** é invocada pelo **consumidor** para receber mensagens da fila.

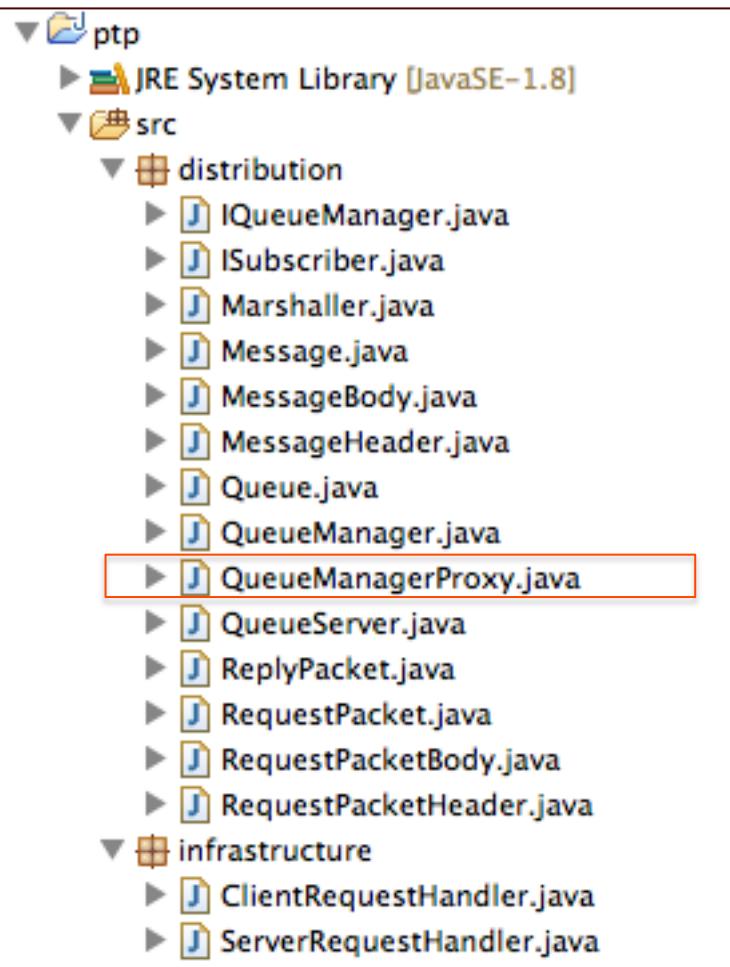
MOM:: Implementação:: QueueManager



```
11 public class QueueManager {  
12     private String host;  
13     private int port;  
14     Map<String, Queue> queues = new HashMap<String, Queue>();  
15 }
```

- ✓ O **QueueManager** armazena várias filas (**Queue**).
- ✓ Cada fila possui um nome.
- ✓ O **QueueManager** é acessado por produtores e consumidores através de um **proxy** (**QueueManagerProxy**)

MOM:: Implementação:: QueueManagerProxy



```
8  public class QueueManagerProxy implements IQueueManager {  
9      private String queueName = null;  
10  
11+     public QueueManagerProxy(String queueName) {}  
12  
13+     public void send(String m) throws IOException, InterruptedException {}  
14  
15+     public String receive() throws IOException, InterruptedException, {}  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44+
```

- ✓ **QueueManagerProxy** é usado por produtores e consumidores de forma similar ao **ClientProxy** do MOO.
- ✓ **QueueManagerProxy** usa o **ClientRequestHandler** para acessar o **QueueManager**.

MOM:: Implementação:: QueueManagerProxy

```
11+ public QueueManagerProxy(String queueName) {..  
14  
15-     @Override  
16     public void send(String m) throws IOException, InterruptedException {  
17         // configure  
18         ClientRequestHandler crh = new ClientRequestHandler("localhost", 1313,  
19             false);  
20         Marshaller marshaller = new Marshaller();  
21         RequestPacket packet = new RequestPacket();  
22         Message message = new Message();  
23  
24         // configure message  
25         message.setHeader(new MessageHeader(this.queueName));  
26         message.setBody(new MessageBody(m));  
27  
28         // configure packet  
29         RequestPacketBody packetBody = new RequestPacketBody();  
30         ArrayList<Object> parameters = new ArrayList<Object>();  
31  
32         packetBody.setParameters(parameters);  
33         packetBody.setMessage(message);  
34         packet.setPacketHeader(new RequestPacketHeader("send"));  
35         packet.setPacketBody(packetBody);  
36  
37         // send request  
38         crh.send(marshaller.marshall((Object) packet));
```

MOM:: Implementação:: Produtor

```
9  public class Producer {  
10  
11     public static void main(String[] args) throws UnknownHostException,  
12                     IOException, Throwable {  
13  
14         QueueManagerProxy queue01Proxy = new QueueManagerProxy("queue01");  
15  
16         QueueManagerProxy queue02Proxy = new QueueManagerProxy("queue02");  
17         Random generator = new Random();  
18  
19         while (true) {  
20             queue01Proxy.send("queue01: " + generator.nextInt(100));  
21             queue02Proxy.send("queue02: " + generator.nextInt(1000));  
22         }  
23     }  
24 }
```

MOM:: Implementação:: Consumidor

```
8 public class Consumer {  
9  
10    public static void main(String[] args) throws UnknownHostException,  
11                  IOException, Throwable {  
12        QueueManagerProxy queue01Proxy = new QueueManagerProxy("queue01");  
13        QueueManagerProxy queue02Proxy = new QueueManagerProxy("queue02");  
14  
15        while (true) {  
16            System.out.println(queue01Proxy.receive());  
17            System.out.println(queue02Proxy.receive());  
18        }  
19    }  
20 }
```

Fim dos Slides