Instituto Superior de Engenharia de Lisboa

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Mestrado em Engenharia Informática e de Computadores (MEIC)
Mestrado em Engenharia Informática e Multimédia (MEIM)

Comunicação por eventos/mensagens

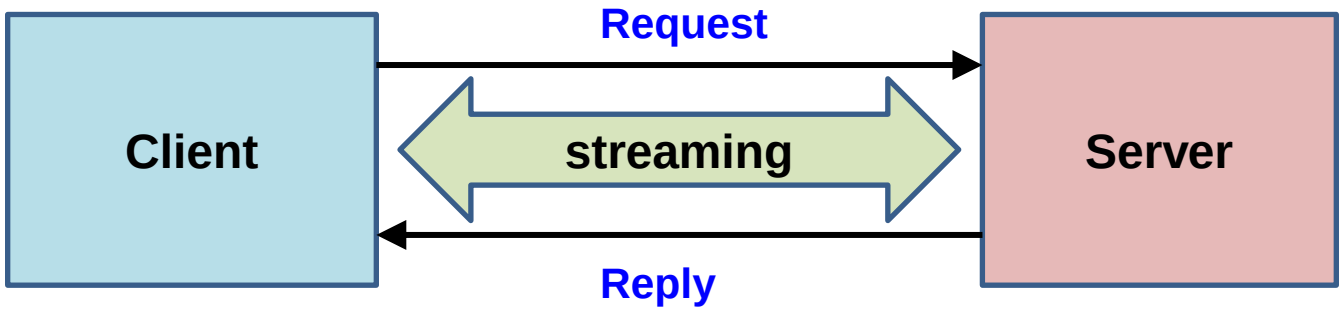
- *Message Oriented Middleware (MOM)*
- *Event-driven architecture*
- Modelo *Publish/Subscribe*

- **Exemplos com *RabbitMQ***

Luís Assunção (lass@isel.ipl.pt ; luis.assuncao@isel.pt)
José Simão (jsimao@cc.isel.ipl.pt ; jose.simao@isel.pt)

Modelo *Request/Reply*

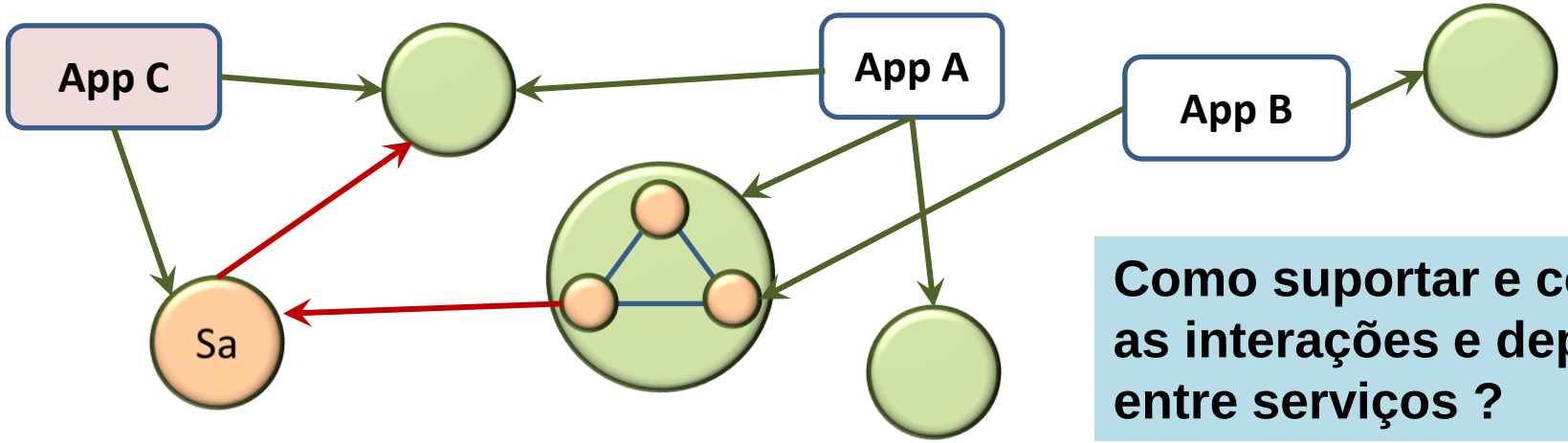
- O modelo *Request/Reply* é normalmente usado nas interações entre os participantes das arquiteturas *Client/Server*
- No modelo GRPC vimos que existia também a possibilidade do servidor tomar a iniciativa de comunicar com o cliente (*stream* de servidor)
- No entanto, as interações são sempre limitadas a dois participantes



E se o Server aceder a outros servers/serviços para poder responder ?

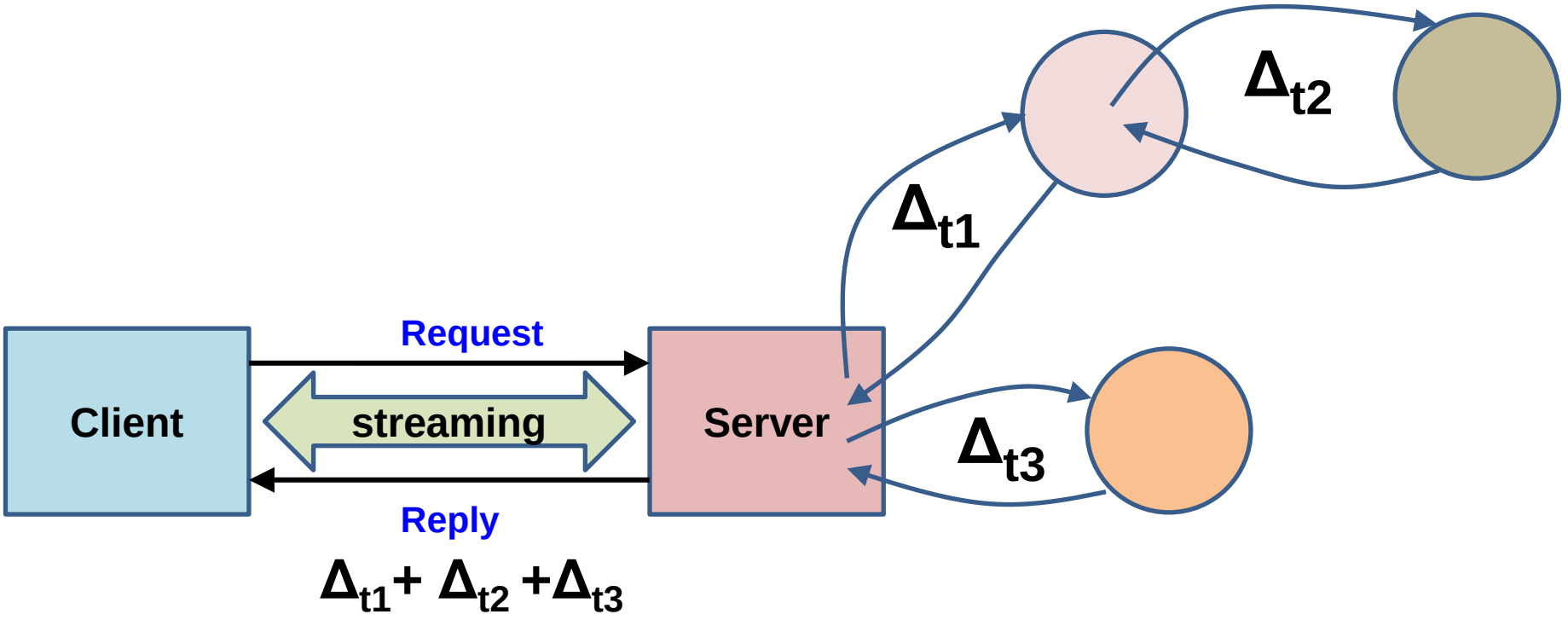
Arquitetura Orientada aos Serviços (*SOA - Service Oriented Architecture*)

- **Arquiteturas distribuídas baseadas no conceito de serviço ou *microservice*;**
- **Os serviços são as unidades básicas (*building blocks)* das *aplicações* e disponibilizam funcionalidades de negócio bem definidas (ex: "*check credit*", "*translate text*", "add numbers", etc.);**
- **Um serviço tem uma fronteira bem definida (contrato), escondendo os detalhes internos de implementação. O alojamento (*deployment*) deve ser autónomo e suportar um mecanismo de comunicação consistente e fracamente acoplado (*loosely coupled*);**
- **Um serviço pode ser a composição de outros serviços;**



Como suportar e coordenar as interações e dependências entre serviços ?

Múltiplos *Servers*/serviços



- No modelo *Request/Reply* o cliente pode não conhecer o que se passa no *backend* do servidor a quem fez o pedido;
- A solução é criar desacoplamento e assincronismo entre o *Request* e uma notificação posterior com o *Reply*

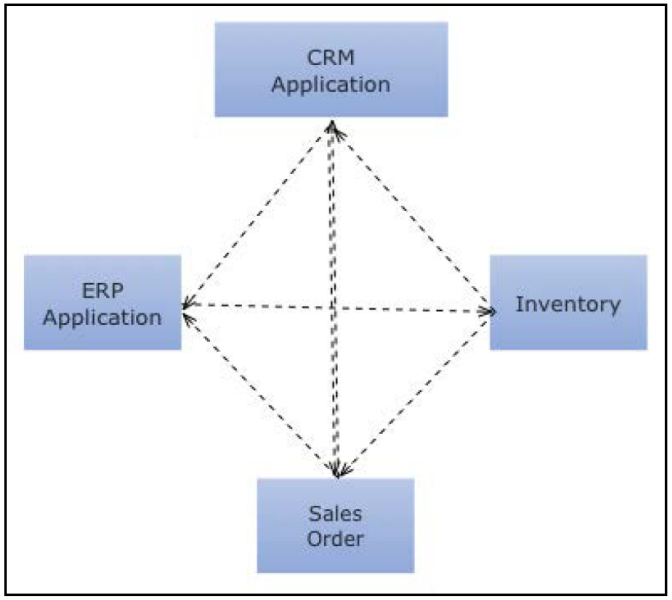
Processamento *Event-driven*

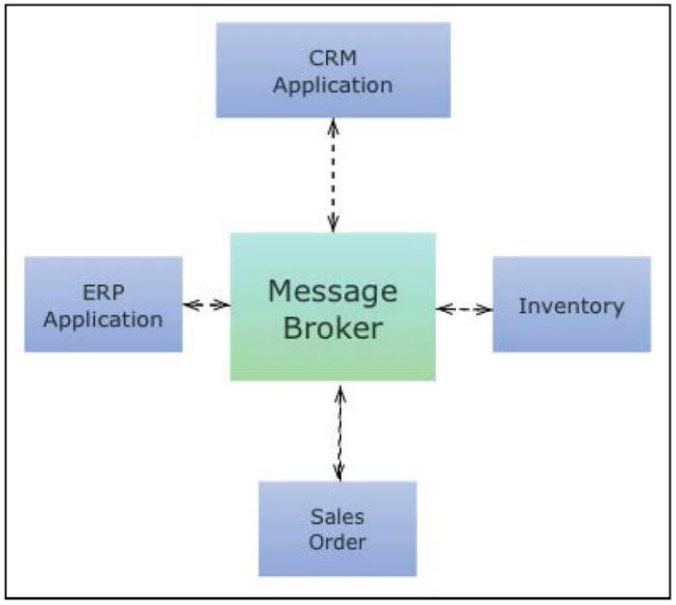
- As características dos sistemas distribuídos, tais como, interações assíncronas, heterogeneidade e um adequado acoplamento fraco (*loosely coupled*) requerem mecanismos de notificação baseados em eventos/mensagens;
- A notificação por eventos, permite comunicação *Many-to-Many* onde os múltiplos participantes podem ter um, ou mesmo os dois, dos seguintes papéis:
  - Produtor de eventos/mensagens (*Producer*/*Publisher*)
  - Consumidor de eventos/mensagens (*Subscriber*/*Consumer*)
- Para suportar a interação assíncrona e os diferentes ritmos de produção e consumo dos eventos/mensagens é necessário a existência de uma entidade intermediária, normalmente designada de *Broker e/ou Router*
- Um *Broker*/*Router* tem mecanismos baseados em múltiplas filas de retenção de mensagens, suportando assim;
  - Desacoplamento entre produtores e consumidores
  - Interações assíncronas
  - Diferentes ritmos de produção e consumo dos eventos/mensagens
  - Realizar encaminhamentos (*routing*) de mensagens para múltiplas filas segundo padrões flexíveis

O *Broker* como intermediário e integrador

***Tightly coupled*:**

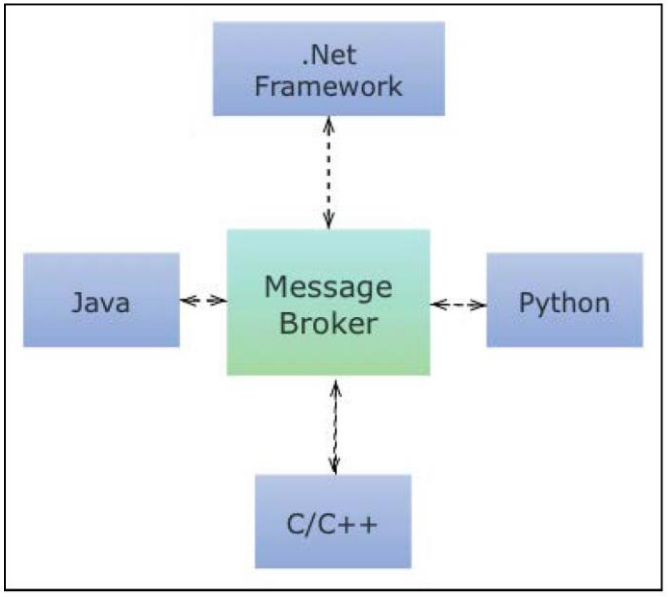
- **Interação de todos com todos**



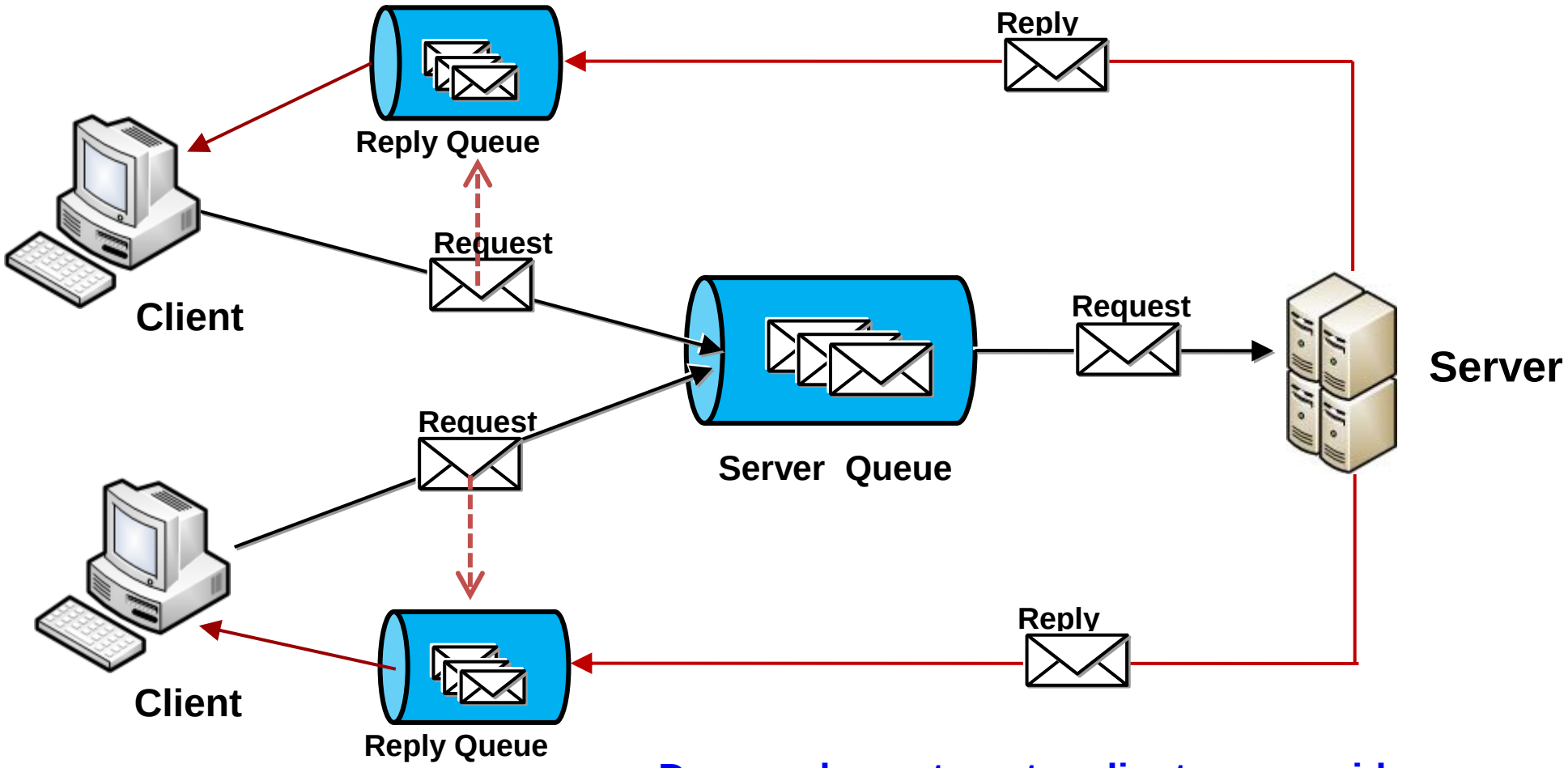


***Loosely coupled*:**

- **Interação de todos com todos através de *Message Broker*;**
- **Integração com heterogeneidade**



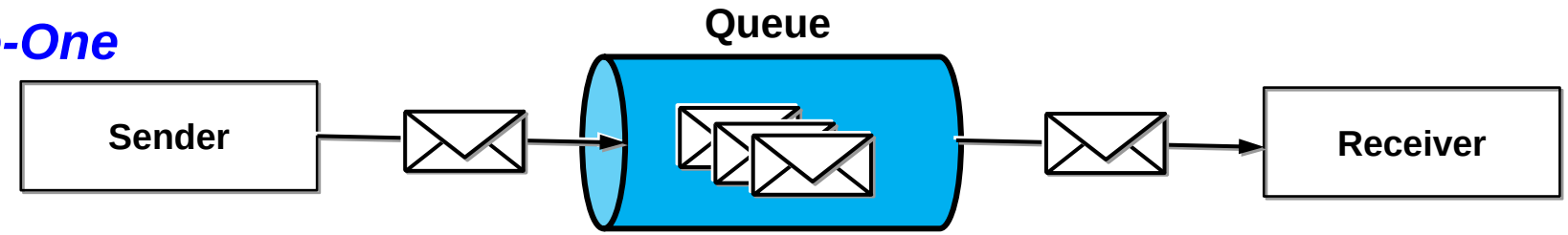
Arquitetura *Client/Server* com Filas (*Queues*) de mensagens



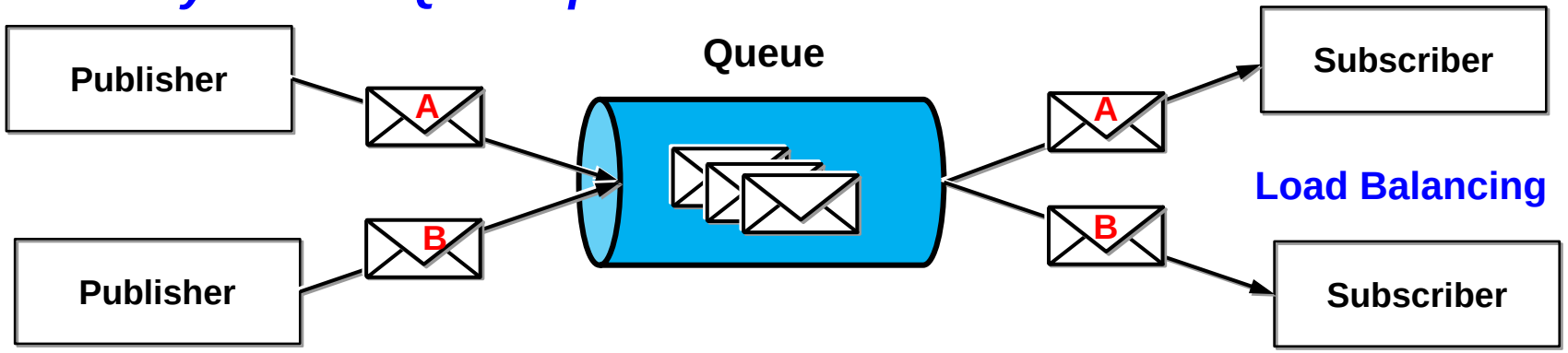
- **Desacoplamento entre clientes e servidor**
- **Assincronismo entre *Requests* e *Replies***
- **Tolerância a falhas e balanceamento de carga**

Padrões baseados em Filas (*Queues*)

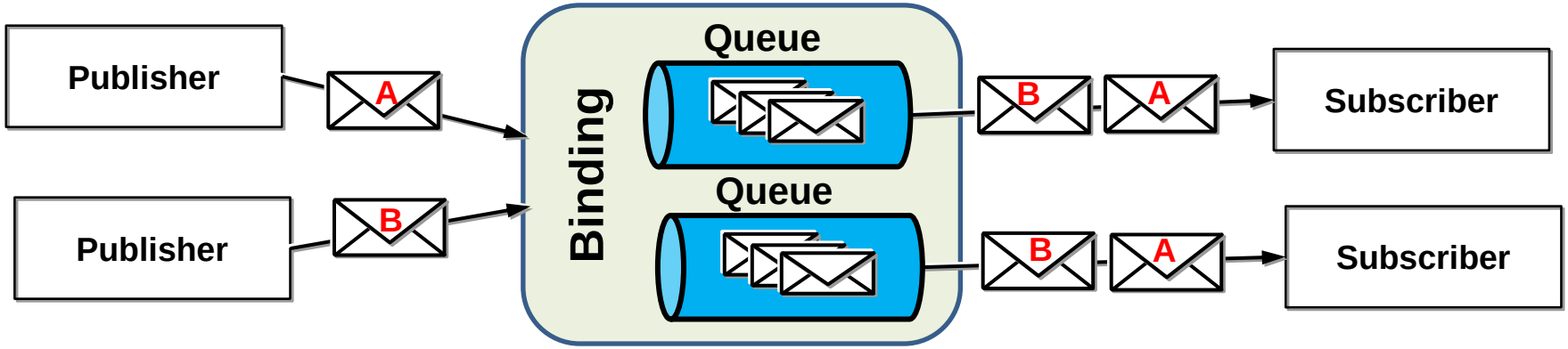
One-to-One



Many-to-Many – Work-Queue pattern

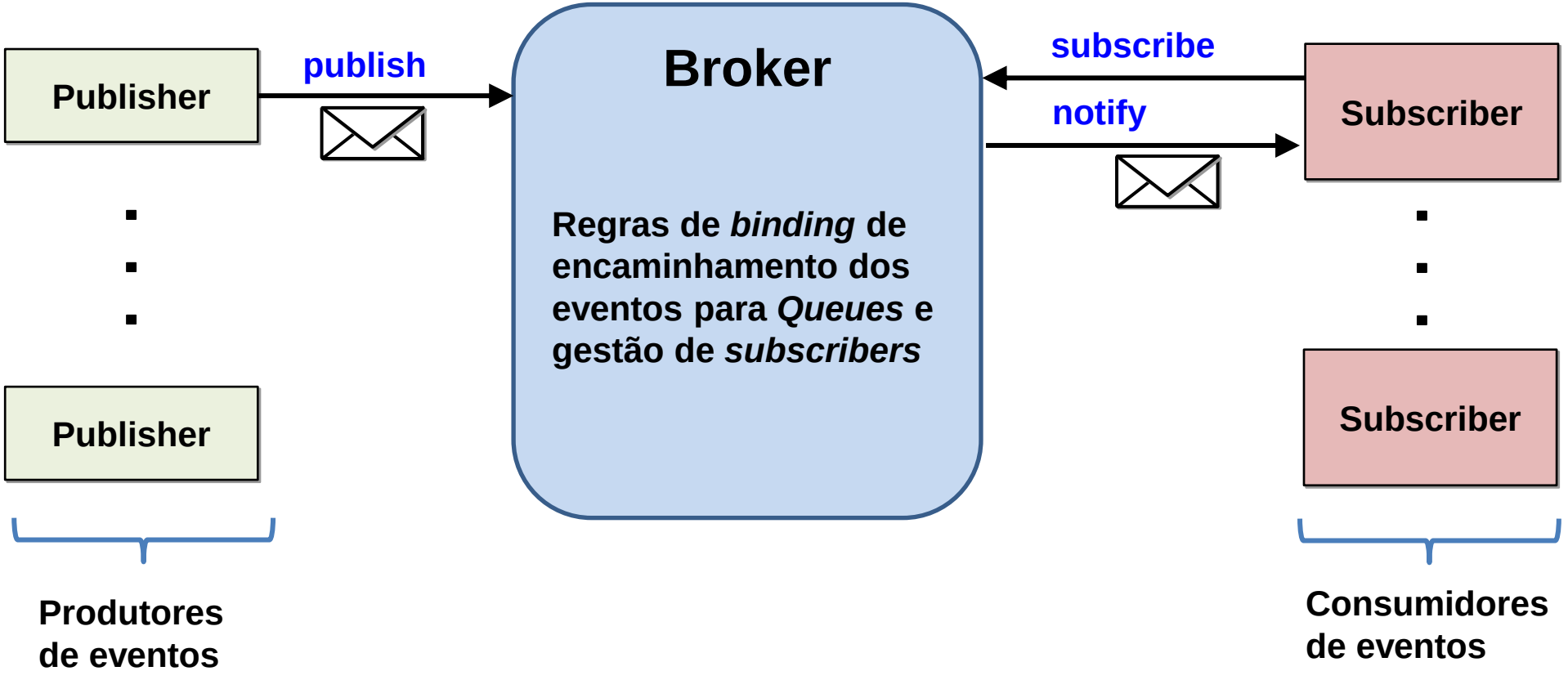


Many-to-Many – Fanout pattern



Modelo *Publish/Subscribe*

- ***Publishers***: Produzem informação na forma de eventos
- ***Subscribers***: Subscrevem (declaram interesses) os eventos publicados
- ***Broker***: de acordo com ***bindings*** armazena os eventos em ***queues*** e notifica os ***subscribers*** que declararam interesse



História de *Message-Oriented-Middleware* (MOM)

- (1993) IBM MQSeries, renamed *WebSphere MQ* (2002) e renamed IBM MQ (2014)
- (final 90s) TIBCO Rendezvous na plataforma *Enterprise Service Bus*
- (final de 90s) COM+ (Windows Server e .NET), renamed Microsoft MSMQ
- (1998) JMS (Java Message Service); JBoss Messaging (2006); Oracle OpenMQ (2012, 2014). Atualmente é designado por Jakarta Messaging
- (2003-2011) AMQP (*Advanced Message Queuing Protocol*) como iniciativa de múltiplas empresas para definir um protocolo standard para um *message-oriented middleware* (*Fedora AMQP; Apache Qpid; RabbitMQ*; etc.);
- (2007) ZeroMQ foi implementado com sockets TCP/IP para cenários de mensagens pequenas e requisitos de baixa latência.
- (2011) Apache Kafka é um sistema de streaming de eventos que suporta múltiplos brokers em cluster, usando na sua coordenação o serviço *ZooKeeper*

❖ Alguns destas tecnologias têm dependências com plataformas proprietárias, com instalação/configuração complexas e com limitações de escalabilidade

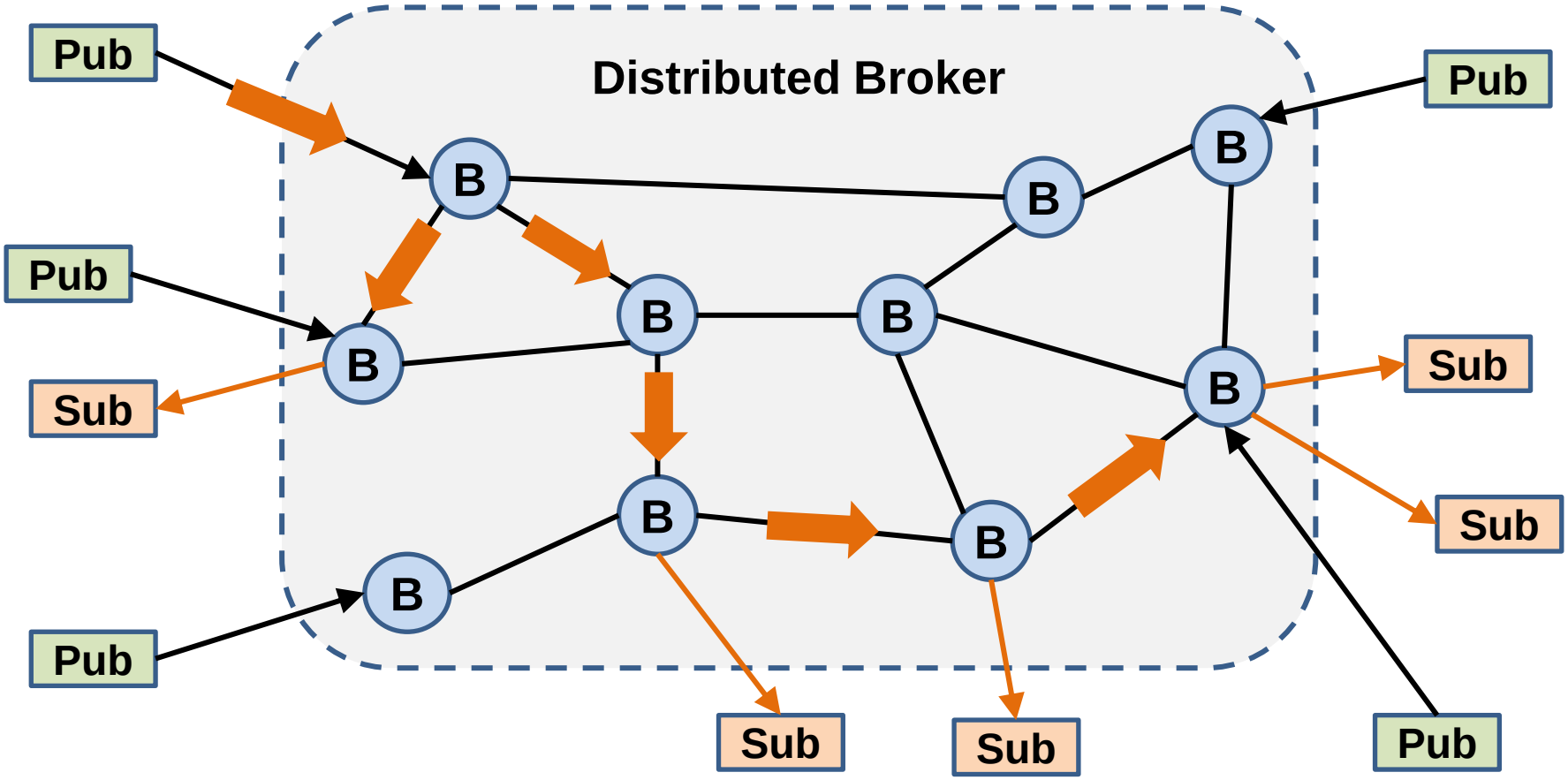
❖ Atualmente na Cloud (sistemas de eventos/mensagens em larga escala):

✓ *Cloud Amazon SQS (Simple Queue Service); Azure Queue Storage; Google Pub/Sub*

Brokers centralizados e/ou distribuídos

- **As soluções tecnológicas de implementação de Brokers inicialmente eram centralizadas num único servidor;**
- **Atualmente as implementações suportam redes de brokers distribuídos interligados com serviço de coordenação e encaminhamento (*routing*) de eventos**
- **Os encaminhamentos (*routing)* podem ser realizados por:**
  - **Palavras chave de encaminhamento**
  - **Filtragem baseados em padrões de chaves de encaminhamento**
  - **Conteúdo de atributos nos eventos**
  - **Etc.**

Distributed Broker como rede de encaminhamento de eventos



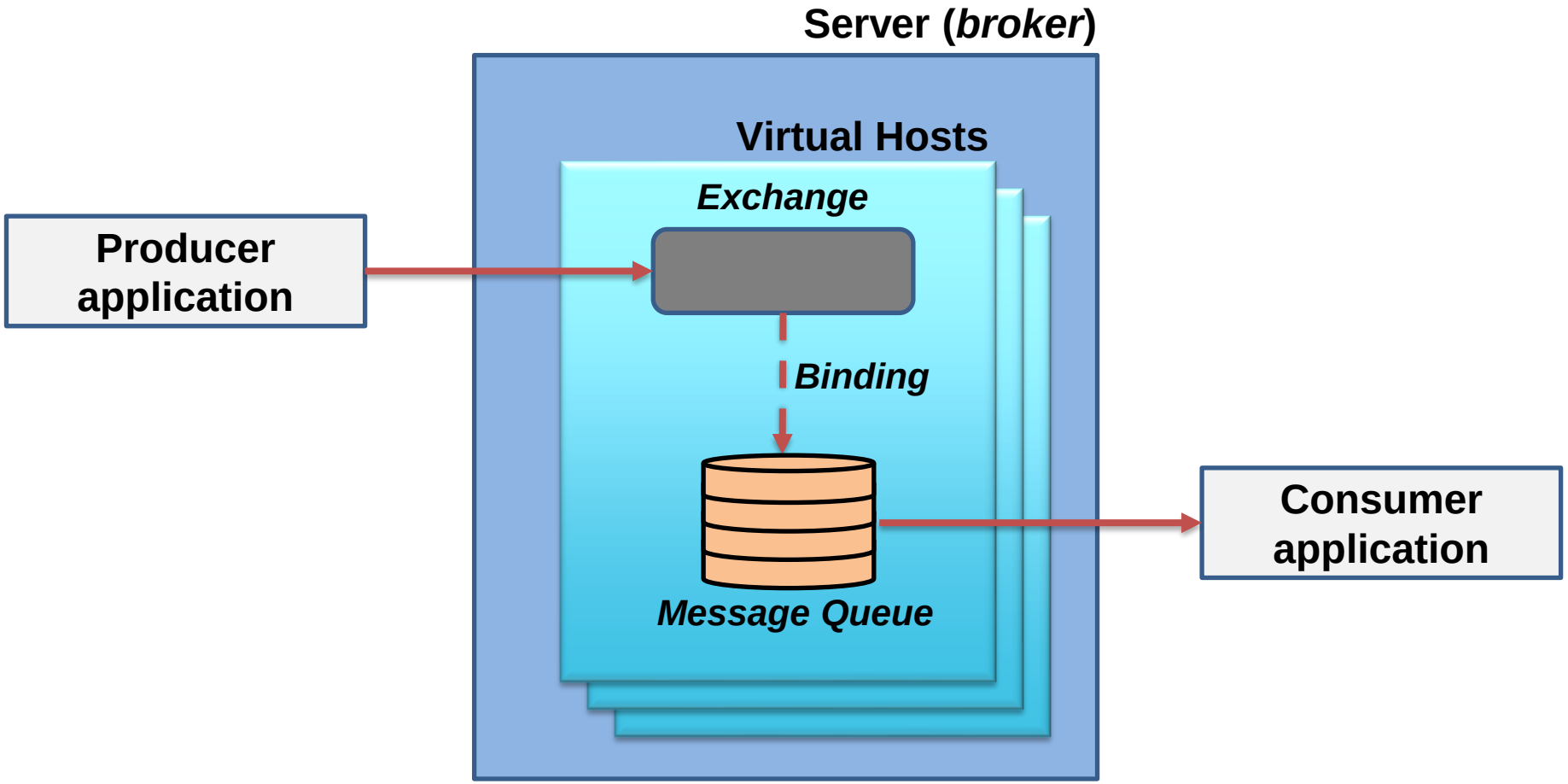
- ✓ **O desafio é coordenar os mecanismos de *binding* entre *brokers* e de encaminhamento eficiente dos eventos sem inundar a rede com todos os eventos até atingir todos os *subscribers* (*flooding*)**

RabbitMQ

Suporte do protocolo AMQP (*Advanced Message Queuing Protocol*)

Advanced Message Queuing Protocol (AMQP)

- **O AMQP 0-9-1 (*Advanced Message Queuing Protocol*) é um protocolo com especificação aberta para encaminhamento de mensagens através de *broker***



https://www.amqp.org/specification/0-9-1/amqp-org-download

RabbitMQ – Implementação do AMQP

- ***Virtual Host***: **partição isolada dentro do servidor com o seu conjunto de *exchanges* e *message queues*. O servidor pode ter 1 ou mais *virtual hosts*, o que corresponde a uma abordagem *multi-tenant*.**

- ***Exchange***: **entidade responsável pelo encaminhamento de mensagens de aplicações publicadoras para filas, segundo uma das estratégias pré-definidas.**

- ***Message Queue***: **Uma fila é uma estrutura com estratégia FIFO que guarda mensagens oriundas de um *Exchange*, as quais serão lidas por aplicações consumidoras**

- ***Bindings***: **Associações entre *Exchanges* e *Message Queues***

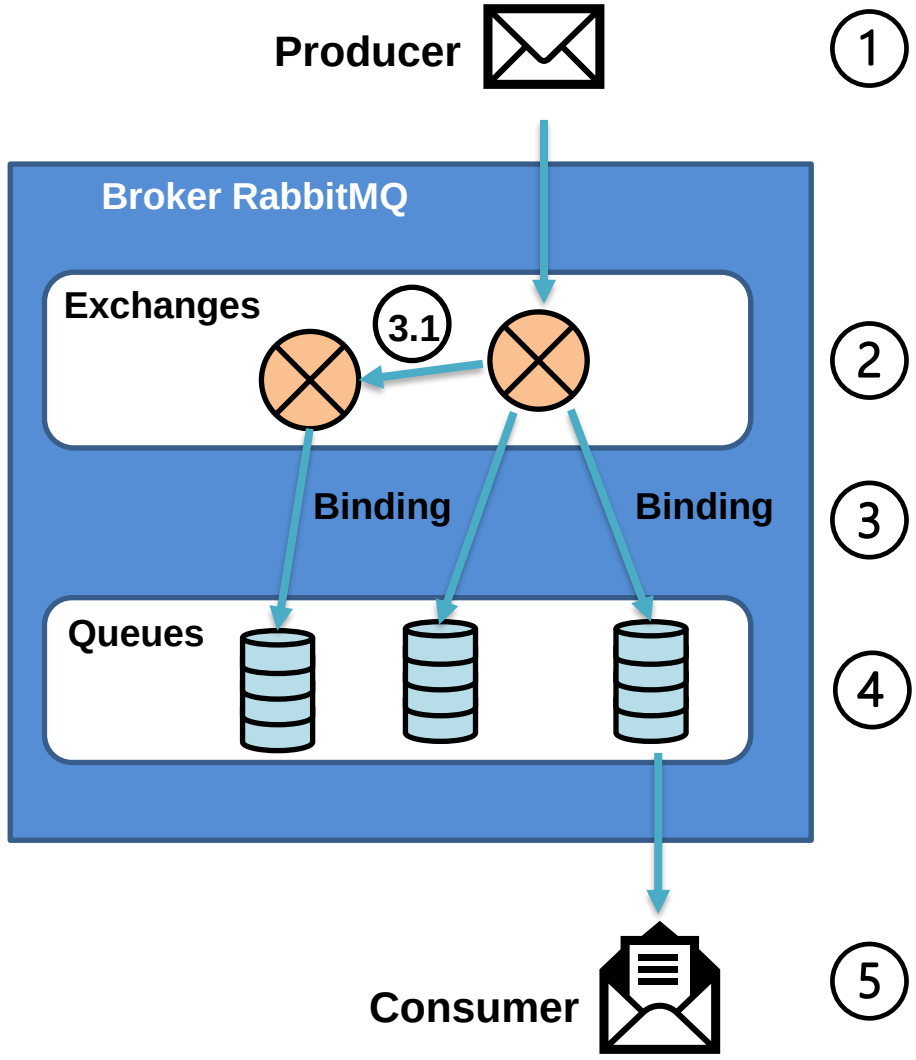
- ***Routing key***: **Palavra (*string)* usada no processo de encaminhamento de mensagens entre *Exchanges* e *Message queues* de acordo com os *Bindings***

Tipos de *Exchange*

- Existem quatro tipos de *exchange* que determinam a forma de encaminhamento (*routing*) das mensagens
  - ***Fanout:*** Encaminha mensagens para todas as *message queues* que estão ligadas (*binding*) ao *exchange*.
  - ***Direct:*** Encaminha mensagens para *messages queues* cuja *binding key* corresponde à *routing key* existente na mensagem
  - ***Topic:*** Semelhante ao *Direct* mas onde a *routing key* é comparada com um padrão de encaminhamento (palavras separadas por ponto, eventualmente com os *wildcards* * e #) especificado no *binding*
  - ***Headers:*** Encaminha mensagens para *messages queues* de acordo com *headers*, pares (key, value), especificados no *binding* da *message queue* e de acordo com os *headers* enviados na mensagem. No *binding* com a *message queue* é possível indicar um dos pares:
    - `("x-match", "all")` só encaminha se a mensagem contiver todos os *headers* definidos no *binding*
    - `("x-match", "any")` encaminha com a presença de qualquer um dos *headers* definidos no *binding.*

Ciclo de vida de uma mensagem

1. O produtor publica uma mensagem num *exchange*
2. O *exchange* recebe a mensagem e é responsável pelo seu encaminhamento para uma *queue*
3. A mensagem é encaminhada em função do tipo de *exchange* e correspondente *binding* com as *queues*, tendo em conta atributos de *routing* presentes na mensagem
   3.1. *Binding* com outro *Exchange*
4. A mensagem fica na *queue* até ser consumida
5. A aplicação consumidora remove a mensagem da *queue*, processa o seu conteúdo e dá ou não *acknowledge* ao consumo da mensagem



Execução de Broker RabbitMQ

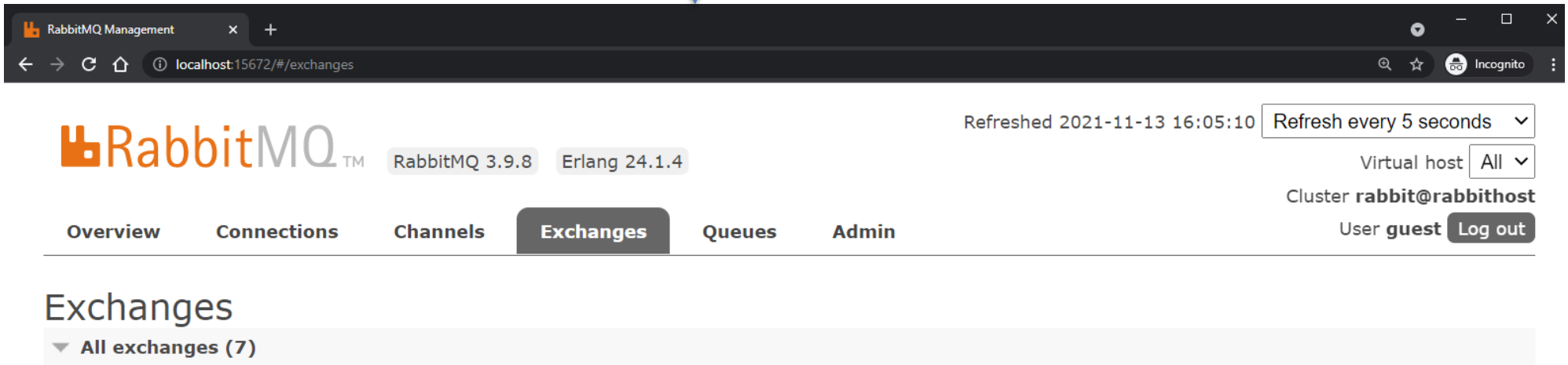
```
$ docker run -d --hostname rabbithost --name rabbitmg -p 5672:5672 -p
15672:15672 rabbitmq:management
```

Porto da aplicação web para gerir o servidor RabbitMQ

Porto do servidor RabbitMQ

HTTP Management Application

RabbitMQ Broker



Resumo da API de cliente

- Classe `ConnectionFactory`: fábrica de ligações para o servidor com possibilidade de configurar parâmetros (ex: endereço IP e porto do servidor *broker*)
- Classe `Connection`: Ligação TCP ao servidor RabbitMQ
- Classe `Channel`: Representa uma canal dentro de uma *Connection*, servindo de base para a chamada de operações de criação, configuração de *exchanges e message queues* (*bindings*), de envio e receção de mensagens, …
  - A classe `Channel` promove a multiplexagem da mesma ligação TCP, permitndo que aplicações multi-threaded possam assim abrir vários canais independentes, sobre a mesma ligação TCP
  - Métodos de configuração: `exchangeDeclare`, `queueDeclare`, `queueBind`, `exchangeBind`
  - Métodos de publicação e consumo de mensagens: `basicPublish`, `basicConsume`

```
<dependency>
    <groupId>com.rabbitmq</groupId>
    <artifactId>amqp-client</artifactId>
    <version>5.13.1</version>
</dependency>
```

Resumo da API de cliente (Classe Channel): Exchange, Queue

https://github.com/rabbitmq/rabbitmq-java-client/blob/main/src/main/java/com/rabbitmq/client/Channel.java

```
 /**
  * Declare an exchange
  * @param exchange the name of the exchange
  * @param type the exchange type
  * @param durable true if we are declaring a durable exchange (the exchange will survive a server restart)
  */
exchangeDeclare(String exchange, String type, boolean durable)
```

```
/**
 * Declare a queue
 * @param queue the name of the queue
 * @param durable true if we are declaring a durable queue (the queue will survive a server restart)
 * @param exclusive true if we are declaring an exclusive queue (restricted to this connection)
 * @param autoDelete true if we are declaring an autodelete queue (server will delete it when no longer in use)
 * @param arguments other properties (construction arguments) for the queue
 * @return a declaration-confirm method to indicate the queue was successfully declared
 */
queueDeclare(String queue, boolean durable, boolean exclusive,
                           boolean autoDelete, Map<String, Object> arguments)
```

Resumo da API de cliente (Classe Channel): Bindings de queues

```
/**
 * Bind a queue to an exchange
 * @param queue the name of the queue
 * @param exchange the name of the exchange
 * @param routingKey the routing key to use for the binding
 * @return a binding-confirm method if the binding was successfully created
 */
queueBind(String queue, String exchange, String routingKey);
```

```
/**
 * Bind a queue to an exchange, with extra arguments.
 * @param queue the name of the queue
 * @param exchange the name of the exchange
 * @param routingKey the routing key to use for the binding
 * @param arguments other properties (binding parameters, e.g. binding headers)
 * @return a binding-confirm method if the binding was successfully created
 */
queueBind(String queue, String exchange, String routingKey,
                                         Map<String, Object> arguments);
```

Resumo da API de cliente (Classe Channel): Bindings de exchanges

- É possível ter um *Exchange* a encaminhar mensagens para outro *Exchange*

```
/**
 * Bind an exchange to an exchange.
 * @param destination the name of the exchange to which messages flow across the binding
 * @param source the name of the exchange from which messages flow across the binding
 * @param routingKey the routing key to use for the binding
 * @return a binding-confirm method if the binding was successfully created
 */
exchangeBind(String destination, String source, String routingKey);
```

```
/**
 * Bind an exchange to an exchange, with extra arguments.
 * @param destination the name of the exchange to which messages flow across the binding
 * @param source the name of the exchange from which messages flow across the binding
 * @param routingKey the routing key to use for the binding
 * @param arguments other properties (binding parameters)
 * @return a binding-confirm method if the binding was successfully created
 */
exchangeBind(String destination, String source, String routingKey,
                                                Map<String, Object> arguments)
```

Resumo da API de cliente (Classe Channel): Publicar mensagem

```
/**
 * Publish a message.
 * Publishing to a non-existent exchange will result in a channel-level
 * protocol exception, which closes the channel.
 *
 * @param exchange the exchange to publish the message to
 * @param routingKey the routing key
 * @param props other properties for the message - routing headers, etc
 * @param body the message body
 */
basicPublish(String exchange, String routingKey, BasicProperties props, byte[] body)
```

Resumo da API de cliente (Classe Channel): Consumir mensagem

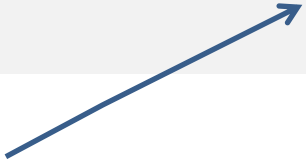
```
/**
 * Start a non-nolocal, non-exclusive consumer, with a server-generated consumerTag.
 * @param queue the name of the queue
 * @param autoAck true if the server should consider messages acknowledged once delivered;
 *        false if the server should expect explicit acknowledgements
 * @param deliverCallback callback when a message is delivered
 * @param cancelCallback callback when the consumer is cancelled
 * @return the consumerTag generated by the server
 */
basicConsume(String queue, boolean autoAck, DeliverCallback deliverCallback,
                                            CancelCallback cancelCallback)
```

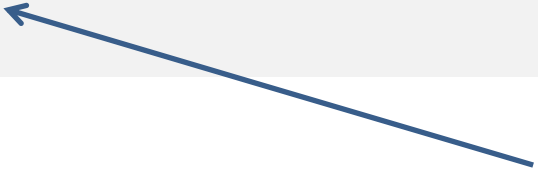
```
/**
 * Acknowledge one or several received messages.
 * @param deliveryTag the tag from the received message
 * @param multiple true to acknowledge all messages up to and including the supplied delivery tag;
 * false to acknowledge just the supplied delivery tag.
*/
void basicAck(long deliveryTag, boolean multiple)
```

```
/**
 * Reject one or several received messages.
 * @param deliveryTag the tag from the received message
 * @param multiple true to reject all messages up to and including the supplied delivery tag;
 * false to reject just the supplied delivery tag.
 * @param requeue true if the rejected message(s) should be requeued rather than discarded/dead-lettered
 */
void basicNack(long deliveryTag, boolean multiple, boolean requeue)
```

Callbacks de entrega de mensagens

```java
/**
 * Callback interface to be notified when a message is delivered.
 * Prefer it over {@link Consumer} for a lambda-oriented syntax,
 * if you don't need to implement all the application callbacks.
 */
@FunctionalInterface
public interface DeliverCallback {
    /**
     * Called when a deliver is received for this consumer.
     * @param consumerTag the <i>consumer tag</i> associated with the consumer
     * @param message the delivered message
     */
    void handle(String consumerTag, Delivery message) throws IOException;
}
```





Valor retornado pela chamada ao basicConsume(…) onde o *callback* foi registado

Conteúdo da mensagem (*body*) e propriedades (ex: *headers*, *routing key*, *delivery tag*)

Callbacks de cancelamento de consumo

```
/**
 * Callback interface to be notified of the cancellation of a consumer.
 * Prefer it over {@link Consumer} for a lambda-oriented syntax,
 * if you don't need to implement all the application callbacks.
 */
@FunctionalInterface
public interface CancelCallback {
    /**
     * Called when the consumer is cancelled for reasons other than by a call to
     * @param consumerTag the consumer tag associated with the consumer
     */
    void handle(String consumerTag) throws IOException;
}
```

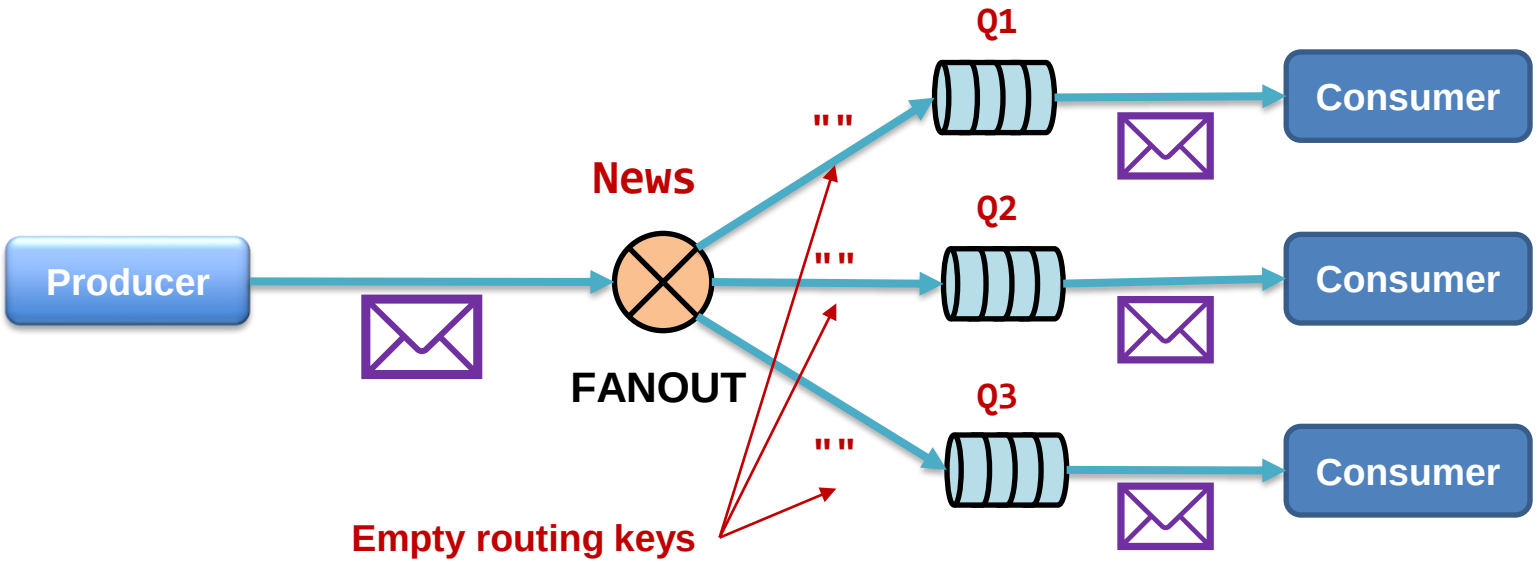
Exchange do tipo *FANOUT* – Configuração e produção de mensagens

```
channel.exchangeDeclare("News", BuiltinExchangeType.FANOUT);
channel.queueDeclare("Q1", true, false, false, null);
channel.queueDeclare("Q2", true, false, false, null);
channel.queueDeclare("Q3", true, false, false, null);
channel.queueBind("Q1", "News", "");
channel.queueBind("Q2", "News", "");
channel.queueBind("Q3", "News", "");
```

Criação do *Exchange*, *Queues* e respetivos *bindings*

```
String message = "Message body";
channel.basicPublish("News", "", null, message.getBytes())
```

Producer



Exchange do tipo *FANOUT* – Consumo de mensagens

Criação de *callbacks* e registo para consumo (Q1) com confirmação automática

```
// Consumer handler to receive messages
DeliverCallback deliverCallback = (consumerTag, delivery) -> {
    String recMessage = new String(delivery.getBody(), "UTF-8");
    String routingKey = delivery.getEnvelope().getRoutingKey();
    System.out.println("Message Received: " + routingKey + " : " + recMessage);
};

// Consumer handler to receive cancel receiving messages
CancelCallback cancelCallback = (consumerTag) -> {
    System.out.println("CANCEL Received! " + consumerTag);
};

// Consumes with auto acknowledge
String consumerTag = channel.basicConsume("Q1", true,
                                          deliverCallback, cancelCallback);
```

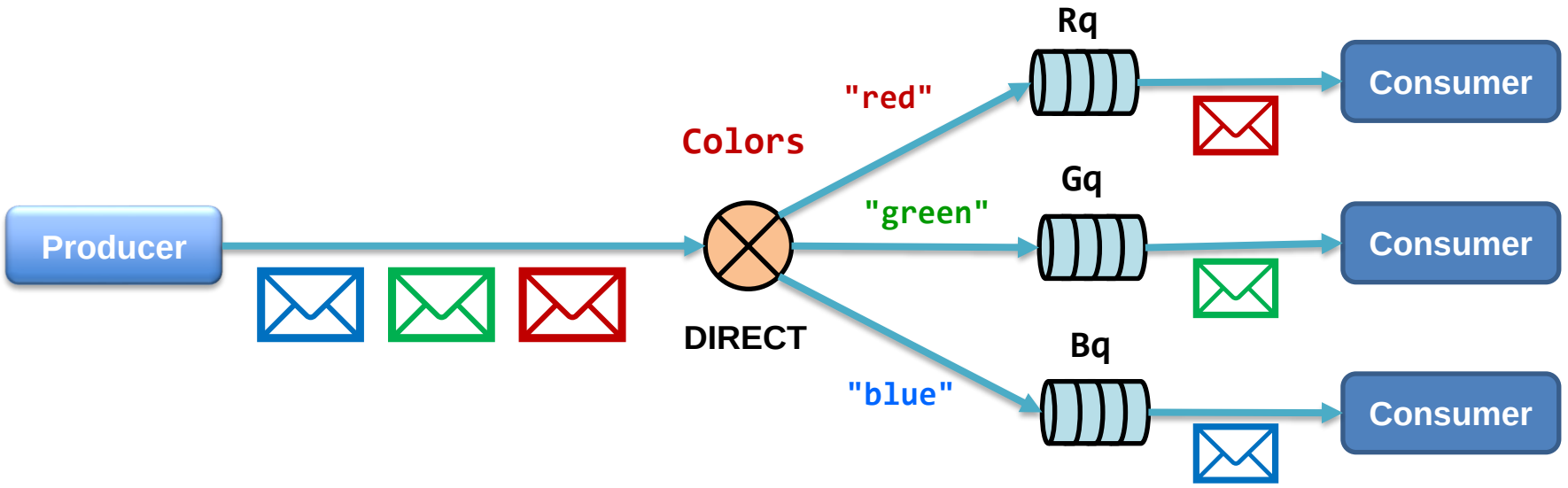
Exchange do tipo *DIRECT*

```
channel.exchangeDeclare("Colors", BuiltinExchangeType.DIRECT);
channel.queueDeclare("Rq", true, false, false, null);
channel.queueDeclare("Gq", true, false, false, null);
channel.queueDeclare("Bq", true, false, false, null);
channel.queueBind("Rq", "Colors", "red");
channel.queueBind("Gq", "Colors", "green");
channel.queueBind("Bq", "Colors", "blue");
```

Criação do *Exchange, Queues* e respetivos *bindings*

Producer

```
String message = "Message body";
channel.basicPublish("Colors", "red", null, message.getBytes())
channel.basicPublish("Colors", "green", null, message.getBytes())
channel.basicPublish("Colors", "blue", null, message.getBytes())
```



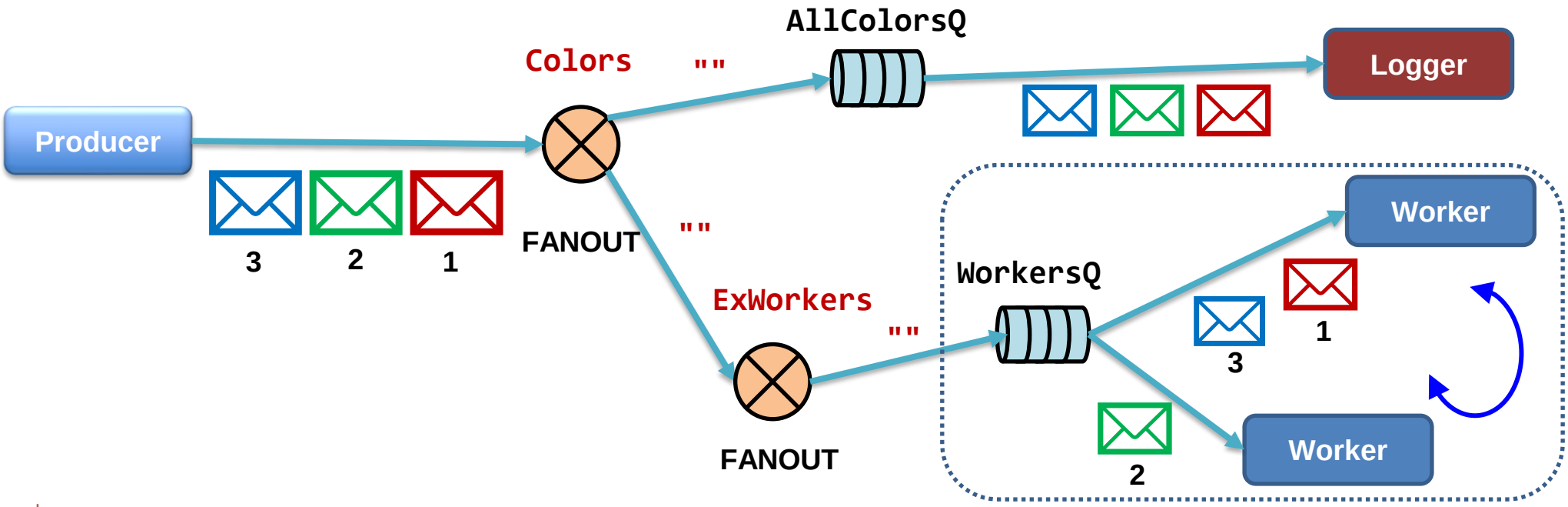
Cenário com padrão Work-Queue

```
channel.exchangeDeclare("Colors", BuiltinExchangeType.FANOUT);
channel.exchangeDeclare("ExWorkers", BuiltinExchangeType.FANOUT);
channel.queueDeclare("AllColorsQ", true, false, false, null);
channel.queueDeclare("WorkersQ", true, false, false, null);
Channel.exchangeBind("ExWorkers","Colors", "");
channel.queueBind("AllColorsQ", "Colors", ""); channel.queueBind("WorkersQ", "ExWorkers", "");
```

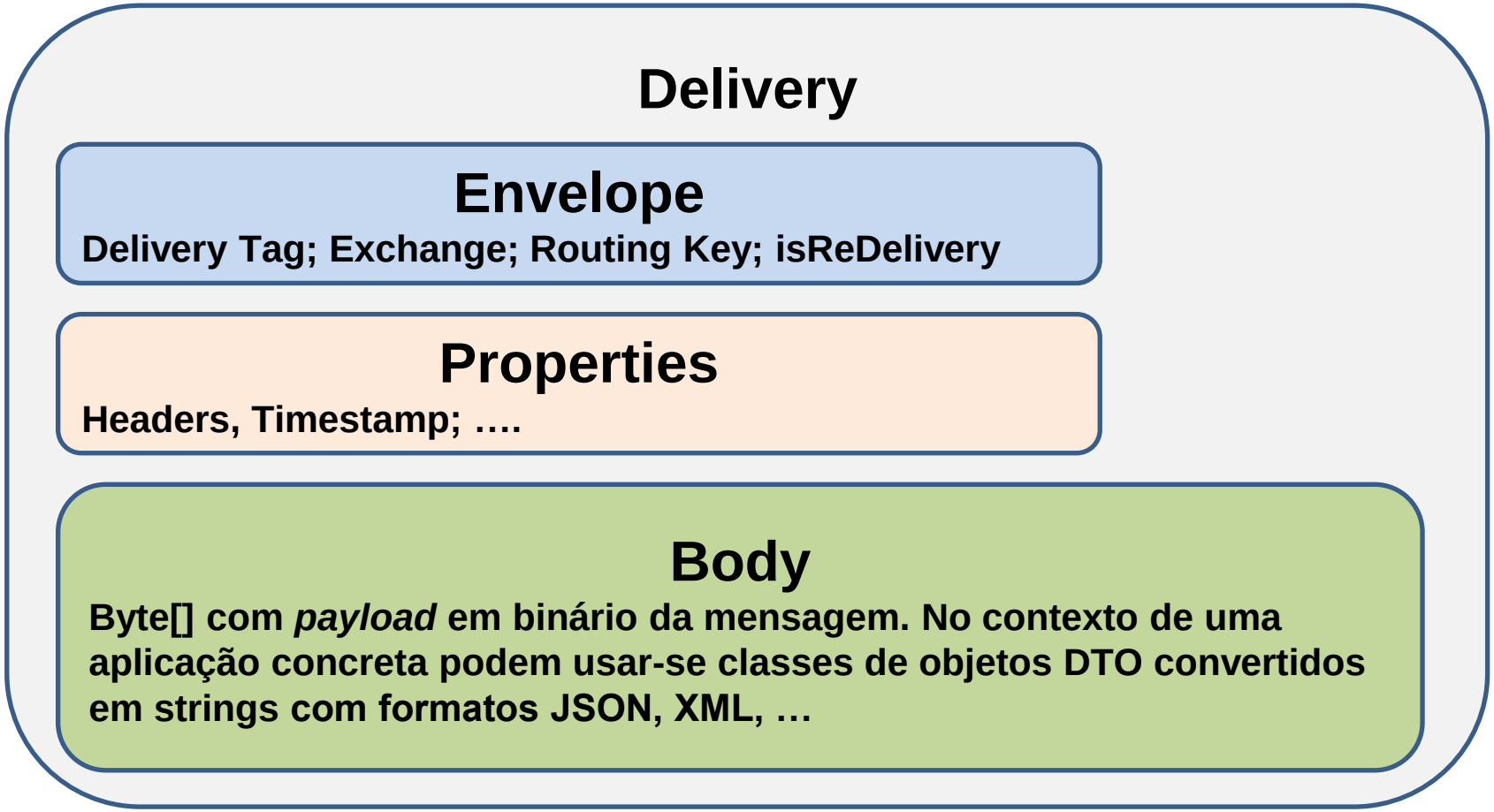
Criação de Exchanges, Queues e *bindings*

```
while (. . .) {
  String message = "Some color";
  channel.basicPublish("Colors","", null, message.getBytes())
}
```

Producer



Estrutura de uma mensagem recebida (*Delivery*)



Confirmação de processamento de mensagens

Acknowledge automático?

```
channel.basicConsume("SomeQueue", false, deliverCallback, cancelCallback);
```

- É possível não definir *acknowledge* automático (*false* no parâmetro) podendo na receção de uma mensagem dar ou não o *acknowledge* para todas as mensagens anteriores, incluindo a mensagem recebida.
  - Cada mensagem entregue por um canal a um consumidor transporta uma *delivery tag.* O uso de uma *delivery tag* no canal errado resultará numa exceção
  - No caso de *acknowledge* negativo é possível ainda indicar se a mensagem é reposta ou não na *queue* para ser posteriormente reenviada

```
DeliverCallback deliverCallback = (consumerTag, delivery) -> {
   String message = new String(delivery.getBody(), "UTF-8");
   ...
   if (<message was processed>)
        channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false );
   else
        channel.basicNack(delivery.getEnvelope().getDeliveryTag(), false , true);
};
```

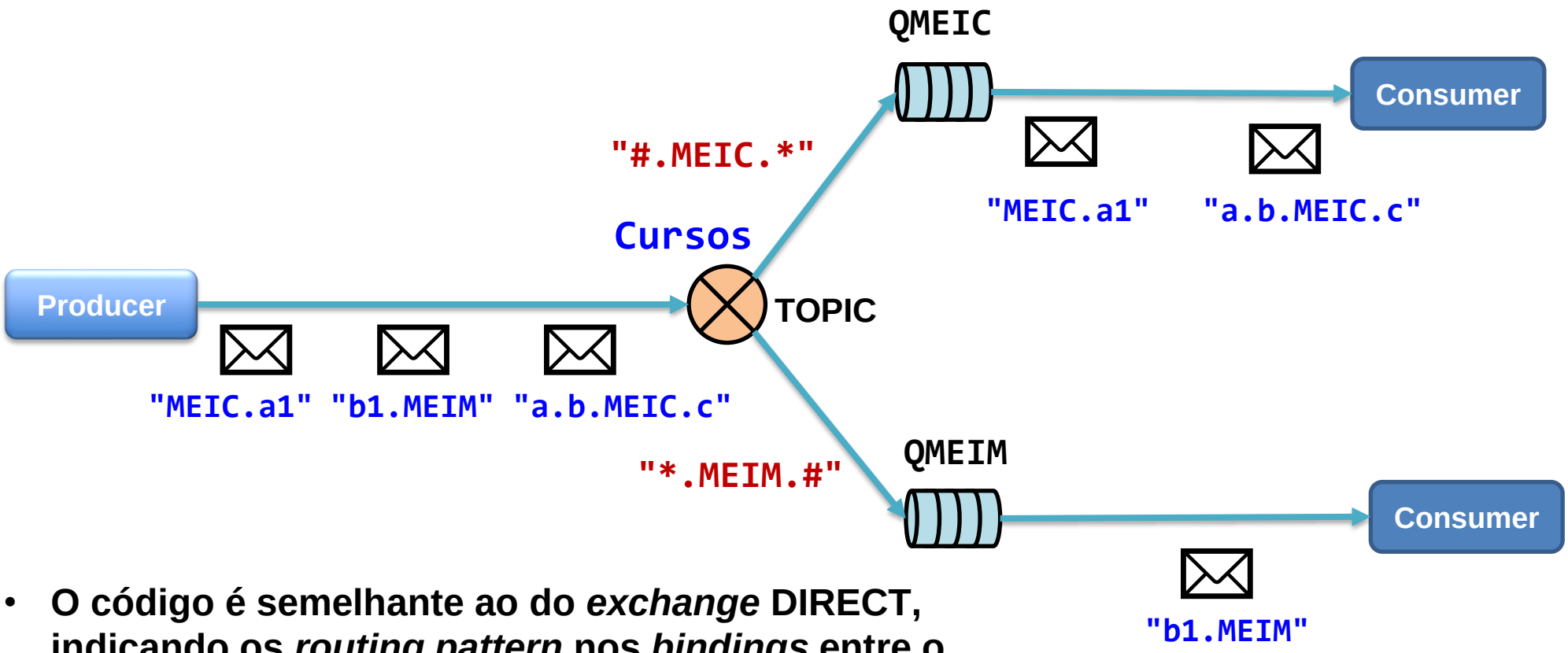
Só mensagem corrente

Repor mensagem na queue

Exchange do tipo *TOPIC*

- A *routing key* nos *exchange* do tipo TOPIC são designados como *routing pattern*
- O *routing pattern* é um conjunto de palavras com os *wildcard* (. * e #) em que (.) é o separador de palavras `(X.Y.Z)`
- (*) significa que é permitida uma única palavra no seu lugar
- (#) significa zero ou mais palavras no seu lugar
- Por exemplo, se o *routing pattern* for `aluno.*` significa que uma mensagem com *routing key* `aluno.curso` é encaminhada mas `curso.aluno` não é encaminhada;
- Um *routing key* `#.MEIC.*` significa que são encaminhadas as mensagens com as *routing key* (`MEIC.aviso; aluno.MEIC.info; mapa.matriculas.MEIC.info`)

Exemplo de Exchange TOPIC

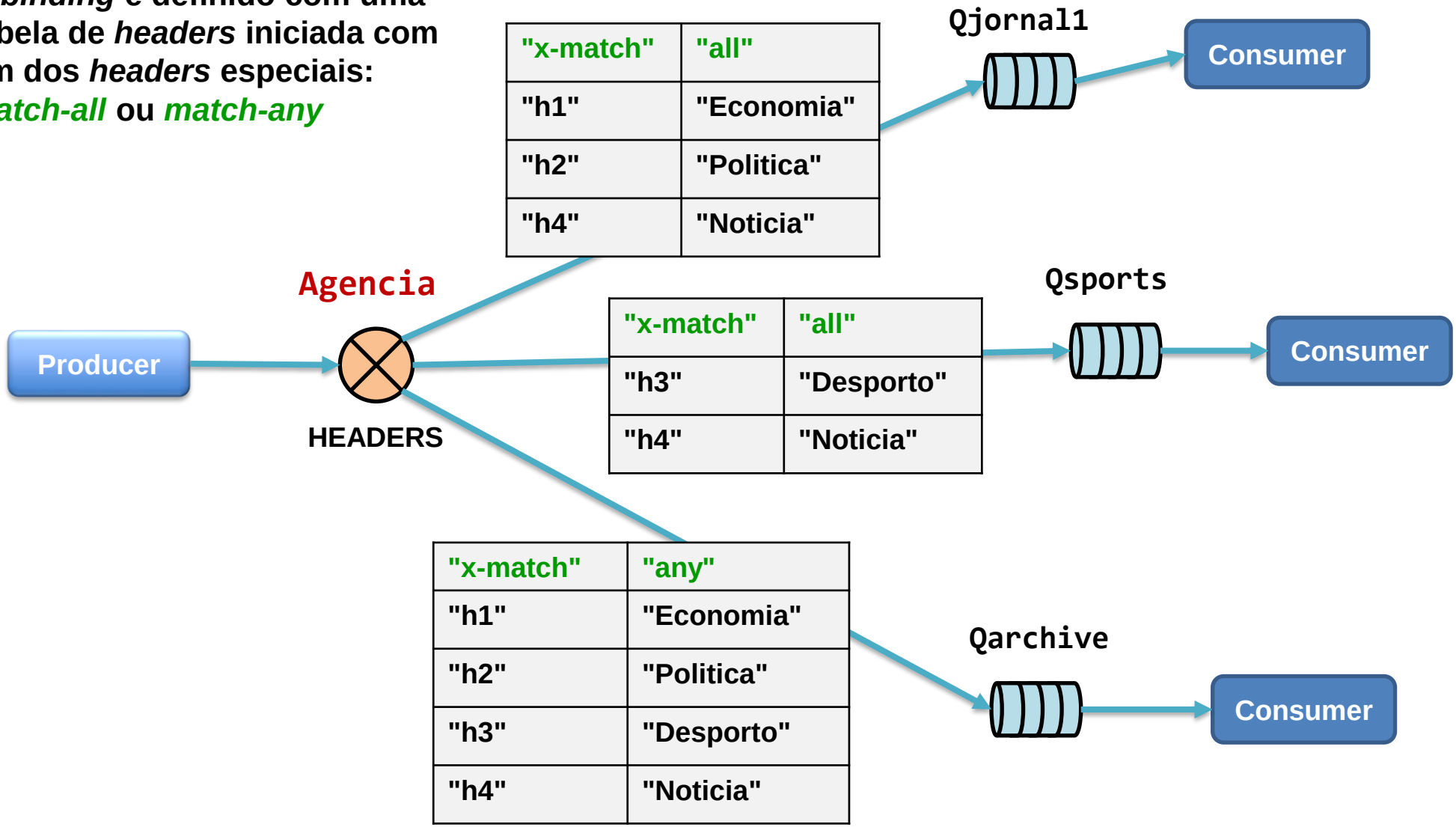


- **O código é semelhante ao do *exchange* DIRECT, indicando os *routing pattern* nos *bindings* entre o *exchange* Cursos e as *queues* QMEIC e QMEIM**

```
channel.exchangeDeclare("Cursos", BuiltinExchangeType.TOPIC);
channel.queueDeclare("QMEIC", true, false, false, null);
channel.queueDeclare("QMEIM", true, false, false, null);
channel.queueBind("QMEIC", "Cursos", "#.MEIC.*");
channel.queueBind("QMEIM", "Cursos", "*.MEIM.#");
```

Exchange do tipo *HEADERS*

O *binding* é definido com uma tabela de *headers* iniciada com um dos *headers* especiais: *match-all* ou *match-any*



Exemplo de Binding da Queue <Qarchive>

```
channel.exchangeDeclare("Agencia", BuiltinExchangeType.HEADERS);
// criar filas
// . . .
Map<String, Object> arqHeaders = new HashMap<>();
arqHeaders.put("x-match", "any"); //Match any of the headers
arqHeaders.put("h1", "Economia");
arqHeaders.put("h2", "Politica");
arqHeaders.put("h3", "Desporto");
arqHeaders.put("h4", "Noticia");
channel.queueBind("Qarchive", "Agencia", "", arqHeaders);
```

Criação de *Exchange HEADERS* e respetivo *binding para a queue Qarchive*

Envio de mensagem que será recebida nas *queues* Qsports e Qarchive

```
String messageBody = "Uma noticia de ultima hora";
Map<String, Object> headers = new HashMap<>();
headers.put("h3", "Desporto"); headers.put("h4", "Noticia");
AMQP.BasicProperties properties = new AMQP.BasicProperties()
                                        .builder().headers(headers).build();

channel.basicPublish("Agencia", "", properties, messageBody.getBytes());
```

Conclusões

❖ Vantagens

- ✓ Desacoplamento (*loosely coupled*) entre produtores e consumidores
- ✓ Diferentes ritmos de produção e consumo
- ✓ Contrariamente ao modelo *Request/Reply* existe assincronismo entre enviar e receber uma mensagem com a existência de armazenamento das mensagens em filas (*queues*) com possibilidades de encaminhamento flexíveis

❖ Desvantagens

- ✓ Não existe uma garantia forte para o *Producer* de que os *Consumers* receberam a mensagem ou mesmo se alguma não foi encaminhada ou processada.
- ✓ Perante o aumento de *Producers* e *Consumers* o *Broker* pode atingir pontos de sobrecarga (*bottleneck*).
- ✓ A Confidencialidade (encriptação) das mensagens é dificultada pelo facto do *Broker* ter de interpretar o contexto das mensagens para efeitos de encaminhamento ou filtragem. O *Broker* pode também amplificar ataques de *denial of service* ao enviar as mensagens para todos os *Consumers*