UNIVERSIDADE FERNANDO PESSOA

Java RabbitMQ Publish/Subscribe

Sistemas Distribuídos (PL)

3º ano da Licenciatura em Engenharia Informática

Professor: Ivo Pereira (ivopereira@ufp.edu.pt)

Java RabbitMQ Publish/Subscribe

- https://www.rabbitmq.com/tutorials/tutorial-three-java.html

- Enviar uma mensagem para múltiplos consumidores ao mesmo tempo
  - Padrão Publish/Subscribe

- Exemplo: Sistema de logging
  - Produtor emite logs
  - Consumidor recebe e guardar/imprime
    - Por exemplo: um consumidor pode receber os logs e guardar num ficheiro; o outro consumidor pode mostrar ao utilizador na consola

  - As mensagens publicadas serão enviadas para todos os consumidores

RabbitMQ Architecture Components

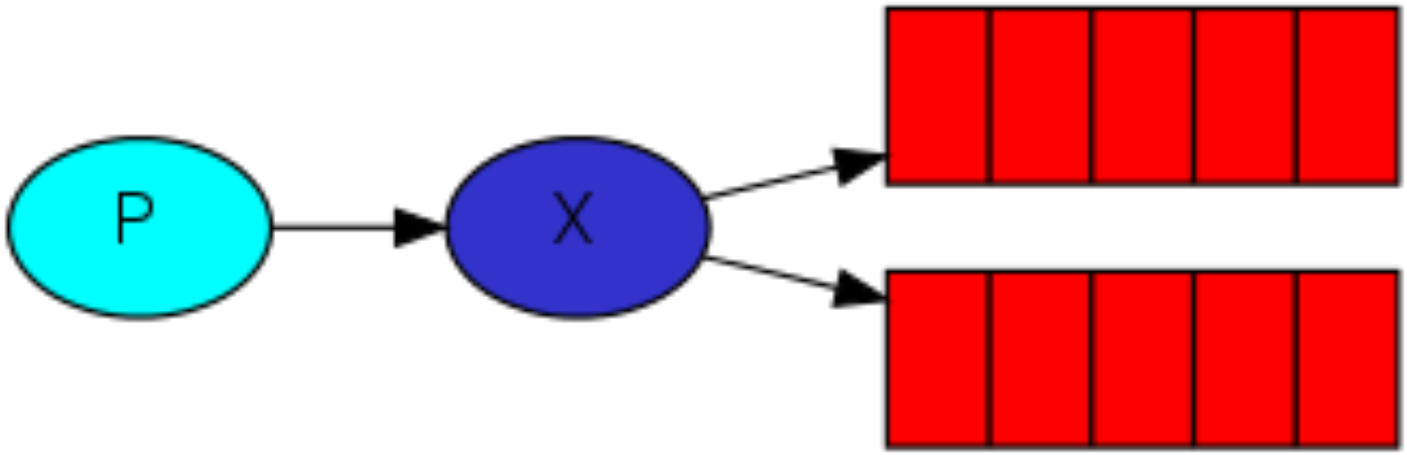
- **Exchange:**
  - component that receives messages from producers and then routes them to queues
- **Queue:**
  - data structure on disk/memory that stores messages
- **Binding:**
  - connection between an exchange and a queue
  - defines what messages should be delivered to what queues

Exchange Types

- **Header:** exchange allows routing messages based on header values instead of routing keys

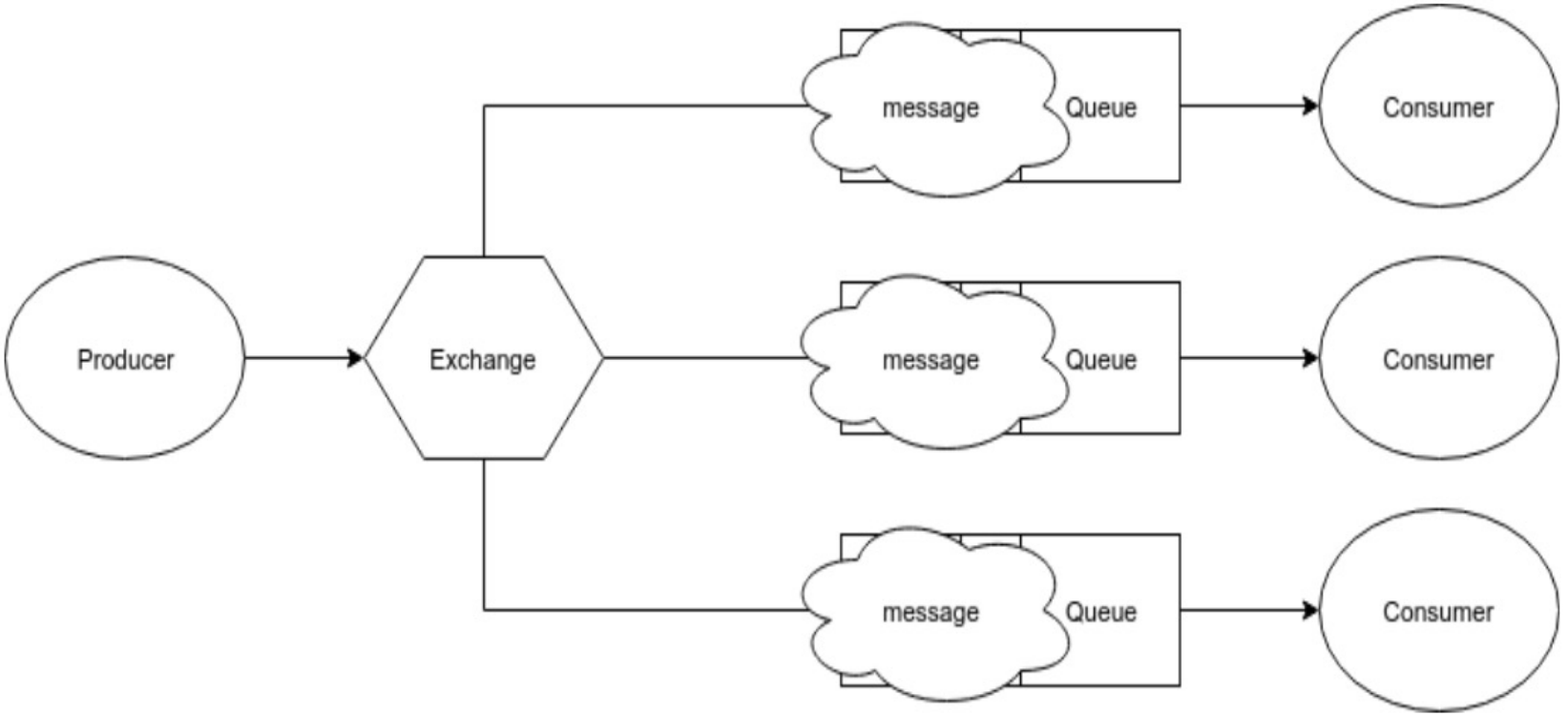
- Default **nameless** exchange: compares routing key to queue name (instead of binding key)
  - publishing msg with routing key = "order", will route it to queue with queue name = "order"

Java RabbitMQ Publish/Subscribe



Exchange Types: **Fanout**

- **Fanout**: exchange sends msg to all queues it knows



Java RabbitMQ Publish/Subscribe

- **Exercício 1:**
  - Copiar package *_01_hello* para *_03_pubsub*
  - Configurar no setenv as variáveis:

```
PACKAGE=_03_pubsub
QUEUE_NAME_PREFIX=pubsub
EXCHANGE_NAME_PREFIX=logs
PRODUCER_CLASS_PREFIX=EmitLog
CONSUMER_CLASS_PREFIX=ReceiveLogs
```

  - Alterar o nome do consumer para *ReceiveLogs* e o nome do producer para *EmitLog*
  - Alterar os nossos *runproducer* e *runconsumer* para usarem a variável BROKER_EXCHANGE em vez da BROKER_QUEUE

Java RabbitMQ Publish/Subscribe

- **Exercício 1:**
  - No consumer ReceiveLogs:

```
//Read args passed via shell command
String host=args[0];
int port=Integer.parseInt(args[1]);
//String queueName=args[2];
String exchangeName=args[2];

// Open a connection and a channel to rabbitmq broker/server
Connection connection=RabbitUtils.newConnection2Server(host, port, username: "guest", passwd: "guest");
Channel channel=RabbitUtils.createChannel2Server(connection);

//Declare queue from which to consume (declare it also here, because consumer may start before publisher)
//channel.queueDeclare(queueName, false, false, false, null);
//channel.queueDeclare(Send.QUEUE_NAME, true, false, false, null);

/* Use the Exchange FANOUT type: broadcasts all messages to all queues */
channel.exchangeDeclare(exchangeName, BuiltinExchangeType.FANOUT);

/* Create a non-durable, exclusive, autodelete queue with a generated name.
The string queueName will contains a random queue name (e.g. amq.gen-JzTY20BRgKO-HjmUJj0wLg) */
String queueName=channel.queueDeclare().getQueue();
```

Java RabbitMQ Publish/Subscribe

- **Exercício 1:**
  - No consumer ReceiveLogs:

```
/* Create binding: tell exchange to send messages to a queue;
The fanout exchange ignores last parameter (routing/binding key) */
String routingKey="";
channel.queueBind(queueName, exchangeName, routingKey);

Logger.getAnonymousLogger().log(Level.INFO, msg: Thread.currentThread().getName()
        +": Will create Deliver Callback...");
System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

//DeliverCallback is an handler callback (lambda method) to consume messages pushed by the sender.
//Create an handler callback to receive messages from queue
DeliverCallback deliverCallback=(consumerTag, delivery) -> {
    String message=new String(delivery.getBody(), charsetName: "UTF-8");
    System.out.println(" [x] Consumer Tag [" + consumerTag + "] - Received '" + message + "'");
};
CancelCallback cancelCallback=(consumerTag) -> {
    System.out.println(" [x] Consumer Tag [" + consumerTag + "] - Cancel Callback invoked!");
};
channel.basicConsume(queueName, b: true, deliverCallback, cancelCallback);

//DO NOT close connection neither channel otherwise it will terminate consumer
```

Java RabbitMQ Publish/Subscribe

UNIVERSIDADE FERNANDO PESSOA

- **Exercício 1:**
  - No producer EmitLogs:

```
//Read args passed via shell command
String host=args[0];
int port=Integer.parseInt(args[1]);
//String queueName=args[2];
String exchangeName=args[2];

/* try-with-resources will close resources automatically in reverse order... avoids finally */
try (Connection connection=RabbitUtils.newConnection2Server(host, port, username: "guest", passwd: "guest");
     Channel channel=RabbitUtils.createChannel2Server(connection)) {

    // Declare a queue where to send msg (idempotent, i.e., it will only be created if it doesn't exist);
    //channel.queueDeclare(queueName, false, false, false, null);
    //channel.queueDeclare(QUEUE_NAME, true, false, false, null);

    System.out.println(" [x] Declare exchange: '" + exchangeName + "' of type "
            + BuiltinExchangeType.FANOUT.toString());
    /* Set the Exchange type MAIL_TO_ADDR FANOUT (multicast to all queues). */
    channel.exchangeDeclare(exchangeName, BuiltinExchangeType.FANOUT);
```

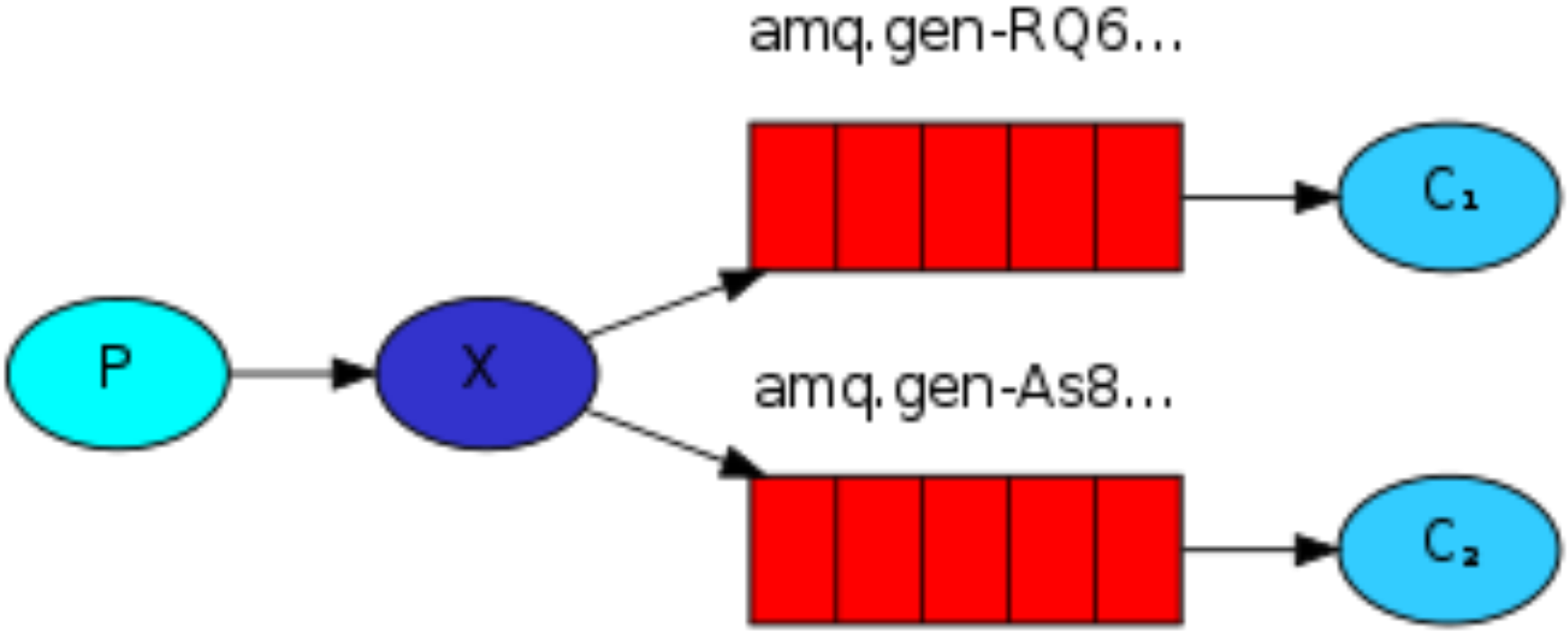
Java RabbitMQ Publish/Subscribe

- **Exercício 1:**
  - No producer EmitLogs:

```
// Gets the message
String message=RabbitUtils.getMessage(args, messageIndex: 3);

/* Publish messages to the logs_exchange instead of the nameless one.
   Fanout exchanges will ignore routingKey (hence set routingKey="").
   Messages will be lost if no queue is bound to the exchange yet. */
String routingKey="";
channel.basicPublish(exchangeName, routingKey, basicProperties: null,
        message.getBytes( charsetName: "UTF-8"));
System.out.println(" [x] Sent: '" + message + "'");
```

Java RabbitMQ Publish/Subscribe



Java RabbitMQ Publish/Subscribe

UNIVERSIDADE FERNANDO PESSOA

- **Exercício 2:**
  - Criar package *_03_pubsub.chatgui* e copiar as classes ObserverGuiClient e Observer disponibilizadas no ficheiro *pubsub.zip*
  - Acrescentar no setenv a variável OBSERVER_CLASS_PREFIX=ObserverGuiClient
  - Criar um script novo *runobserver* que receba um argumento (o username) para correr o ObserverGuiClient
  - Completar o código na classe Observer
    - Comentários **// TODO:**
  - Executar dois observers e interagir um com o outro. Acrescentar outro observer e analisar os comportamentos de troca de mensagens.