

EP2

UC: **Sistemas Distribuídos**

Alunos: **Thauany Moedano**

RA: **92486**

Exercício 1. Prosseguir com o tutorial de RabbitMQ.

Instalação

Para instalar o RabbitMQ é bem simples. Basta baixar o instalador disponível no site. O server ficará rodando automaticamente na máquina e é possível dar start ou stop a partir do menu iniciar.

Como o estudo foi feito utilizando linguagem java, foi necessário baixar os jars com as dependências das bibliotecas de RabbitMQ.

Hello World

O primeiro passo do tutorial é um programa ' Hello World ' que executa os passos mais simples para uma execução utilizando o RabbitMQ. Nesse estudo foi utilizado linguagem java.

A aplicação é dividida em duas classes: um publisher, que conectará ao RabbitMQ e depositará sua mensagem e uma aplicação consumidora que conectará ao RabbitMQ para receber as mensagens.

O exemplo básico é criar um publisher e um consumidor que apenas recebe as mensagens.

A classe Send.java envia as mensagens. Primeiramente é necessário estabelecer uma conexão com o RabbitMQ e criar um canal. Isso tudo é feito através do uso das classes disponíveis na biblioteca.

```
1
2 ConnectionFactory factory = new ConnectionFactory();
3     factory.setHost("localhost");
4     Connection connection = factory.newConnection();
5     Channel channel = connection.createChannel();
6
7     channel.queueDeclare(QUEUE_NAME, false, false, false, null);
8     String message = "Hello World!";
9     channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
10    System.out.println(" [x] Sent '" + message + "'");
```

A classe Recv.java recebe as mensagens. Ela também faz uma conexão com o server e declara a fila que receberá as mensagens. Essa fila deve ser a mesma que a classe Send.java utiliza. Nessa mesma classe, tem a classe Consumer aninhada que é responsável por fazer a tratativa da mensagem quando a mesma é recebida.

```
1
2 Consumer consumer = new DefaultConsumer(channel) {
3     @Override
4     public void handleDelivery(String consumerTag, Envelope envelope,
5         AMQP.BasicProperties properties, byte[] body)
6         throws IOException {
7         String message = new String(body, "UTF-8");
8         System.out.println(" [x] Received '" + message + "'");
9     }
10 };
```

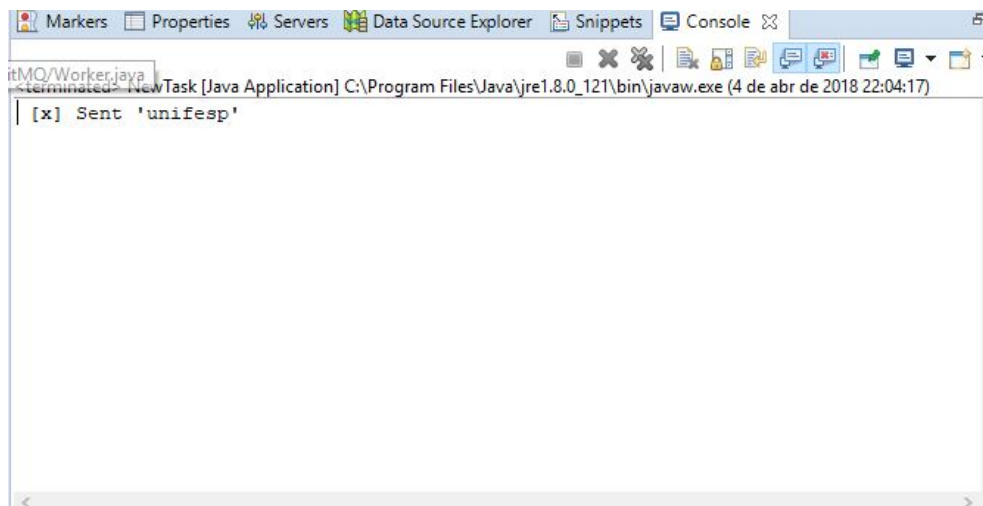


Figura 1: Mensagem enviada.

Work Queues

No segundo tutorial, surgem as classes NewTask.java e Worker.java. Essas são modificações das classes Send.java e Recv.java, respectivamente.

A NewTask.java introduz mensagens arbitrárias, sendo possível enviar qualquer tipo de mensagem além de Hello World.

A classe Worker.java tem como objetivo fazer algum tipo de trabalho com a mensagem recebida. No caso desse tutorial, o trabalho é simulado, apenas disparando um Thread.sleep(). A diferença da classe Worker.java para a Recv.java é que a classe Worker.java tem como objetivo distribuir as tarefas entre vários Workers.

A principal modificação, portanto, acontece na definição do Consumer:

```
1 final Consumer consumer = new DefaultConsumer(channel) {
2     @Override
```

```

3      public void handleDelivery(String consumerTag, Envelope envelope,
4          AMQP.BasicProperties properties, byte[] body) throws IOException {
5          String message = new String(body, "UTF-8");
6
7          System.out.println(" [x] Received '" + message + "'");
8          try {
9              doWork(message);
10             } finally {
11                 System.out.println(" [x] Done");
12                 channel.basicAck(envelope.getDeliveryTag(), false);
13             }
14     };

```

Portanto, com essas duas classes é possível enviar uma mensagem, distribuí-la para um Worker e esperar algum processamento.

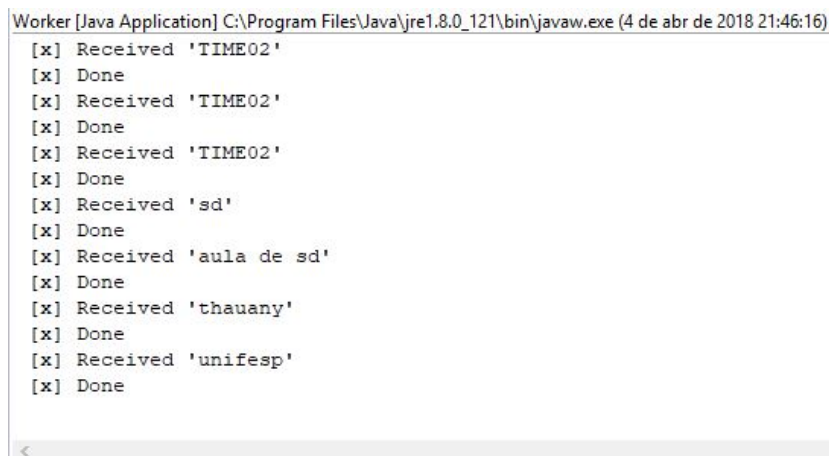


Figura 2: Worker recebendo as mensagens.

Publish-Subscribe

No tutorial anterior, uma tarefa era enviada para apenas um Worker. A ideia nesse tutorial é que uma mensagem possa ser enviada para múltiplos consumidores. E isso é o padrão publish-subscribe.

A ideia desse tutorial é fazer um sistema simples de Logs, onde um sender pode enviar um log e vários consumidores podem recebê-las.

Para fazer esse tutorial, um intermediador é introduzido: o *exchange*. O exchange é responsável por definir se uma mensagem irá para mais de um consumidor, para um consumidor específico ou se deve ser descartada.

Além disso, as filas são todas gerenciadas diretamente pelo servidor do RabbitMQ agora, não sendo necessário declarar nomes para as filas. O servidor cria um nome para a fila e internamente faz a associação da fila com o canal.

A principal modificação de NewTask.java para EmitLog.java:

```

1      channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.FANOUT);

```

```

2
3 String message = getMessage(argv);
4
5 channel.basicPublish(EXCHANGE_NAME, "", null, message.getBytes("UTF-8"));
6 System.out.println(" [x] Sent " + message + " ");

```

E da Worker.java para ReceiveLogs.java:

```

1 channel.exchangeDeclare(EXCHANGE_NAME, BuiltInExchangeType.FANOUT);
2 String queueName = channel.queueDeclare().getQueue();
3 channel.queueBind(queueName, EXCHANGE_NAME, "");
4
5 System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

```

Com o auxílio das ferramentas do RabbitMQ na linha de comando, é possível ver as filas rodando:

```

C:\Program Files\RabbitMQ Server\rabbitmq_server-3.7.4\sbin>rabbitmqctl list_bindings
Listing bindings for vhost /...
exchange      amq.gen-1xDkelybVJXcOdZIL45j0w queue amq.gen-1xDkelybVJXcOdZIL45j0w []
exchange      amq.gen-J--zU9c3qEeKa1eM7inuog queue amq.gen-J--zU9c3qEeKa1eM7inuog []
exchange      hello queue hello []
exchange      task_queue queue task_queue []
logs exchange  amq.gen-1xDkelybVJXcOdZIL45j0w queue []
logs exchange  amq.gen-J--zU9c3qEeKa1eM7inuog queue []
C:\Program Files\RabbitMQ Server\rabbitmq_server-3.7.4\sbin>

```

Figura 3: Filas após rodar o commando rabbitmqctl list-bindings.

Routing

No tutorial anterior, uma mensagem era enviada para todos os consumidores. COM RabbitMQ é possível definir um conjunto de consumidores específicos para receber uma mensagem através de roteamento.

Para isso é necessário utilizar um outro tipo de exchange que vai fazer um *match* entre a mensagem e um consumidor.

No tutorial temos três tipos de chave: info, warning e error. As mensagens do tipo warning e error são enviadas aos dois consumidores mas a chaves do tipo info são enviadas a apenas um.

Para isso, temos que mudar o EmitLog.java para fazer bind com uma fila do tipo de chave que queremos.

```

1 channel.exchangeDeclare(EXCHANGE_NAME, BuiltInExchangeType.DIRECT);
2
3 String severity = getSeverity(argv);
4 String message = getMessage(argv);
5
6 channel.basicPublish(EXCHANGE_NAME, severity, null,
    message.getBytes("UTF-8"));

```

E no ReceiveLogs precisamos definir quais chaves iremos aceitar:

```

1

```

```

2 channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.DIRECT);
3 String queueName = channel.queueDeclare().getQueue();
4
5 if (argv.length < 1){
6     System.err.println("Usage: ReceiveLogsDirect [info] [warning] [error]");
7     System.exit(1);
8 }
9
10 for(String severity : argv){
11     channel.queueBind(queueName, EXCHANGE_NAME, severity);
12 }

```

Dessa forma, quando criamos os dois consumidores como mencionado acima, a mensagem do tipo info só é recebida por um deles:

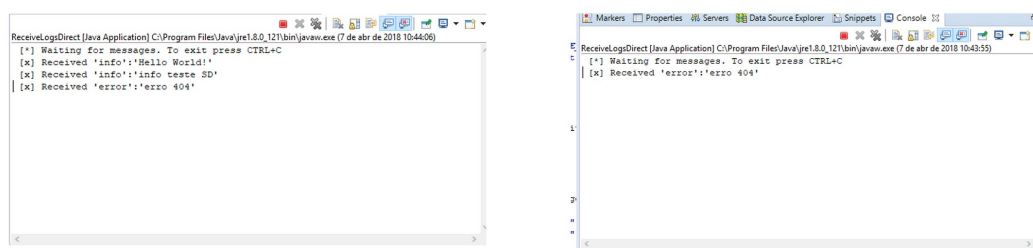


Figura 4: Mensagem do tipo info só é recebida por um consumidor.

Topics

O próximo tutorial trata de melhorar o sistema de roteamento. É possível utilizar mais de um critério para rotear uma mensagem.

Ao invés de somente ter o nível de severidade do log (info, warning, error), o sistema também filtrará pela parte do sistema que vem esse log.

A grande diferença é que agora o tipo de exchange é topic e devemos declarar nossa chave com as palavras separadas por ponto.



Figura 5: Consumidor configurado para receber todos os tipos de log.

RPC

O último tutorial mostra como implantar uma simples arquitetura RPC usando RabbitMQ.

O cliente faz um request que é processado no servidor e tem uma resposta. Nesse tutorial, o request será para calcular a sequência de fibonacci.

O que muda aqui é a necessidade de declarar a função de call que irá fazer a requisição no sistema:

```
1 public String call(String message) throws IOException, InterruptedException {
2     final String corrId = UUID.randomUUID().toString();
3
4     AMQP.BasicProperties props = new AMQP.BasicProperties
5         .Builder()
6         .correlationId(corrId)
7         .replyTo(replyQueueName)
8         .build();
9
10    channel.basicPublish("", requestQueueName, props,
11        message.getBytes("UTF-8"));
12
13    final BlockingQueue<String> response = new ArrayBlockingQueue<String>(1);
14
15    channel.basicConsume(replyQueueName, true, new DefaultConsumer(channel) {
16        @Override
17        public void handleDelivery(String consumerTag, Envelope envelope,
18            AMQP.BasicProperties properties, byte[] body) throws IOException {
19            if (properties.getCorrelationId().equals(corrId)) {
20                response.offer(new String(body, "UTF-8"));
21            }
22        }
23    });
```

```

22
23     return response.take();
24 }

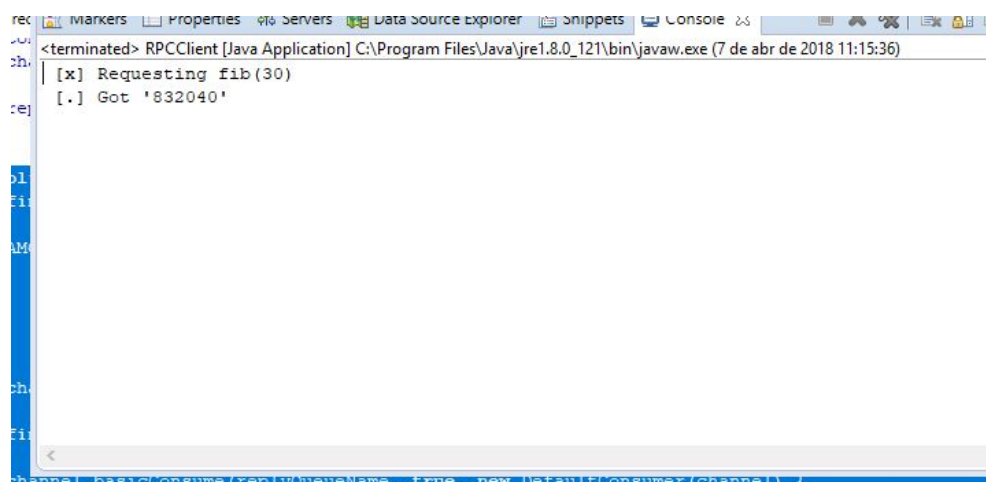
```

O servidor necessita da função `handleDelivery` para fazer a tratativa da requisição:

```

1  public void handleDelivery(String consumerTag, Envelope envelope,
    AMQP.BasicProperties properties, byte[] body) throws IOException {
2      AMQP.BasicProperties replyProps = new AMQP.BasicProperties
3          .Builder()
4          .correlationId(properties.getCorrelationId())
5          .build();
6
7      String response = "";
8
9      try {
10         String message = new String(body, "UTF-8");
11         int n = Integer.parseInt(message);
12
13         System.out.println(" [.] fib(" + message + ")");
14         response += fib(n);
15     }
16     catch (RuntimeException e){
17         System.out.println(" [.] " + e.toString());
18     }
19     finally {
20         channel.basicPublish( "", properties.getReplyTo(), replyProps,
21             response.getBytes("UTF-8"));
22         channel.basicAck(envelope.getDeliveryTag(), false);
23         // RabbitMq consumer worker thread notifies the RPC server owner
24         // thread
25         synchronized(this) {
26             this.notify();
27         }
28     }
29 }

```



```
rec | Markers | Properties | Servers | Data Source Explorer | Snippets | Console 25
ch. <terminated> RPCClient [Java Application] C:\Program Files\Java\jre1.8.0_121\bin\javaw.exe (7 de abr de 2018 11:15:36)
re: | [x] Requesting fib(30)
    | [.] Got '832040'
```

Figura 6: Requisição do cálculo de fibbonacci e a resposta.