UNIVERSIDADE FEDERAL DE ITAJUBÁ
INSTITUTO DE MATEMÁTICA E COMPUTAÇÃO

COM242 - Sistemas Distribuídos

Prof. Rafael Frinhani

TUTORIAL

AMQP (Advanced Message Queuing Protocol)

Grupo:

Arthur Reznik - 2016001460
Caio Noriyuki Sasaki - 2016009842
Guilherme Delgado de Carvalho da Costa Braga - 2019003806
Ivan Leoni Vilas Boas - 2018009073
Karen de Souza Pompeu - 2016001610
Rodrigo Filippo Dias - 2016001479

Dia 26 de Agosto, 2020

Sumário

---

1 Introdução

Este trabalho foi desenvolvido como requisito parcial para a disciplina sistemas distribuídos e tem como objetivo o desenvolvimento de um minicurso sobre AMQP (*Advanced Message Queuing Protocol*). AMQP é um protocolo de padrão aberto em camada de aplicação para *middleware* orientado a mensagens e permite roteamento, confiabilidade e segurança [O'Hara, 2007]. O seu foco é que as esse *middleware* orientado a mensagens sirva para facilitar o desenvolvimento de aplicações nas redes empresariais interoperáveis com base em troca de mensagens [Russo e Carmo, 2012].

2 Arquitetura AMQP

3 Tutorial de execução

3.1 Requisitos

Realizar o download e instalação dos seguintes componentes:

- O RabbitMQ é o software de mensageria, responsável pela troca de mensagens entre o ambiente do cliente e a nuvem. `https://www.rabbitmq.com/download.html`
- Erlang – (O Erlang é a linguagem de programação utilizada pelo RabbitMQ e o OTP é o conjunto de bibliotecas e frameworks responsáveis pela execução do Erlang). Veja as compatibilidades aqui: `https://www.rabbitmq.com/which-erlang.html`
- JavaJDK - Biblioteca para desenvolvimento com Java. Disponivel em: `https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html`
- Python - Interpretador para a linguagem Python. Disponivel em: `https://www.python.org/downloads/`

3.2 Configuração

Não há uma ordem estrita para a instalação dos requisitos listados acima, com exceção do RabbitMQ que deve ser instalado após o Erlang.
Após realizar a instalação de todos os items listados vamos preparar o ambiente com as bibliotecas necessárias para nos comunicar com a interface do RabbitMQ:

3.2.1 Java

Após instalar o JKD é necessário também baixar 3 bibliotecas java que são as responsáveis pela comunicação com a interface do RabbitMQ, você pode encontra-las em:

- Cliente RabbitMQ
- SLF4J API
- SL4F Simple

Crie uma pasta chamada lib no mesmo diretorio onde irá criar o seu programa java e coloque estes 3 arquivos dentro da pasta que acaba de criar.
Caso você use uma IDE para compilar programas java pode ir para a próxima seção, do contrario neste tutorial compilaremos o programa diretamente do terminal e para isso talvez seja necessário acrescentar o diretório no qual você instalou o JDK na variável Path no seu sistema. Para isso procure pelas variaveis do sistema e acrescente o caminho para a pasta *bin* do local de instalação do JDK (Por padrão costuma ser C:/Arquivos de Programa/Java/) Após isso você consegue testar utilizando o comando javac no terminal. Se o comando for reconhecido (mesmo que de erro) quer dizer que está tudo certo.

3.2.2 Python

No seu terminal, de preferência, utilize o seguinte comando para instalar a biblioteca *Pika* para poder trabalhar com o RabbitMQ:

```
python -m pip install pika --upgrade
```

3.3 Implementação

Dividiremos a implementação em 2 etapas: A 1ª será o servidor (em python) que será responsável por receber os pedidos, realizar os procedimentos que iremos definir e retornar com o resultado para os clientes. E o 2º será o cliente (em java) que enviará uma mensagem com a informação que deverá ser processada.

3.3.1 Servidor

Listing 1: servidor.py

```
import pika

connection = pika.BlockingConnection(
    pika.ConnectionParameters(host='localhost'))

channel = connection.channel()
#channel.queue_declare(queue='fila_rpc')
```

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)

def on_request(ch, method, props, body):
    n = int(body)

    print(" [.] fib(%s)" % n)
    response = fib(n)

    ch.basic_publish(exchange='',
                     routing_key=props.reply_to,
                     properties=pika.BasicProperties( correlation_id
                        = props.correlation_id),
                     body=str(response))
    ch.basic_ack(\\delivery_tag=method.delivery_tag)

channel.basic_qos(prefetch_count=1)
channel.basic_consume(queue='fila_rpc', on_message_callback=
   on_request)

print(" [x] Awaiting RPC requests")
channel.start_consuming()
```

3.3.2 Cliente

Listing 2: cliente.java

```
//bibliotecas proprietarias do rabbitMQ
import com.rabbitmq.client.AMQP;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

import java.io.IOException;
import java.util.UUID;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.TimeoutException;


public class ProducerA implements AutoCloseable {

    private Connection connection;
    private Channel channel;
    private String requestQueueName = "fila_rpc";

    //esse metodo ira definir os componentes basicos da classe que
       realizara a conexao com o broker
```

```
public ProducerA() throws IOException, TimeoutException {
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("localhost");

    connection = factory.newConnection();
    channel = connection.createChannel();
}

public static void main(String[] argv) {
    //nosso programa realizara 10 requisicoes ao servidor,
       recebendo o valor da sequencia de fibonacci para os
       numeros [0;9]
    try (ProducerA fibonacciRpc = new ProducerA()) {
        for (int i = 0; i < 10; i++) {
            String i_str = Integer.toString(i);
            System.out.println(" [x] Requesting fib(" + i_str + "
               )");
            String response = fibonacciRpc.call(i_str);
            System.out.println(" [.] Got '" + response + "'");
        }
    } catch (IOException | TimeoutException |
       InterruptedException e) {
        e.printStackTrace();
    }
}
//metodo responsavel por fazer a requisicao ao servidor
public String call(String message) throws IOException,
   InterruptedException {
    final String corrId = UUID.randomUUID().toString();

    String replyQueueName = channel.queueDeclare().getQueue();
    AMQP.BasicProperties props = new AMQP.BasicProperties
            .Builder()
            .correlationId(corrId)
            .replyTo(replyQueueName)
            .build();

    channel.basicPublish("", requestQueueName, props, message.
       getBytes("UTF-8"));

    final BlockingQueue<String> response = new ArrayBlockingQueue
       <>(1);

    //definimos a estrutura basica da mensagem e as propriedades
       necessarias da mesma para que a comunicacao seja possivel

    String ctag = channel.basicConsume(replyQueueName, true, (
       consumerTag, delivery) -> {
        if (delivery.getProperties().getCorrelationId().equals(
           corrId)) {
            response.offer(new String(delivery.getBody(), "UTF-8"
```

```
                    ));
            }
        }, consumerTag -> {
        });

        String result = response.take();
        channel.basicCancel(ctag);
        return result;
    }

    public void close() throws IOException {
        connection.close();
    }
}
```

3.4 Teste

Certifique-se que o RabbitMQ está ativo, você pode conferir isso olhando a lista de processos do seu sistema, caso não esteja inice-o. Agora utilizaremos os seguintes comandos no terminal para executar os programas: O Servidor:

```
py servidor.py
```

O Cliente:

```
javac -cp .;lib/* cliente.java && java -cp .;lib/* cliente
```

Referências

[O'Hara, 2007] O'Hara, J. (2007). Toward a commodity enterprise middleware: Can amqp enable a new era in messaging middleware? a look inside standards-based messaging with amqp. *Queue*, 5(4):48–55.

[Russo e Carmo, 2012] Russo e Carmo, T. (2012). Uso do padrão amqp para transporte de mensagens entre atores remotos. Master's thesis, Universidade de São Paulo.