Aluno: João Pedro Moura Oliveira - 190030879

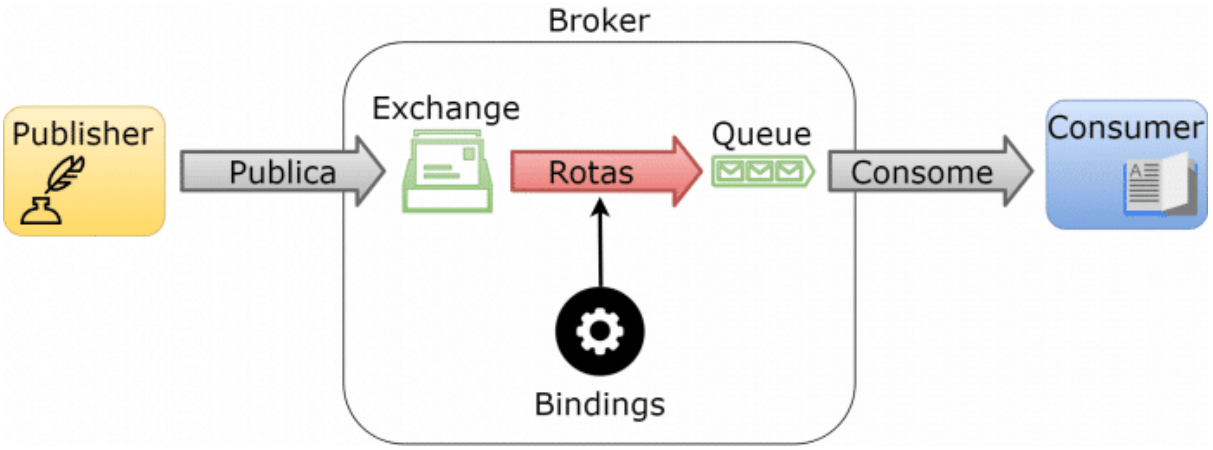
Lab03 –Construindo aplicações distribuídas usando o paradigma *publish-subscriber*
PSPD – Programação para Sistemas Paralelos e Distribuídos
Turma A (Prof. Fernando W Cruz)

1) Introdução

Prosseguindo os estudos sobre programação para sistemas paralelos e distribuídos, parte-se então para a utilização de sistemas de mensageria através do protocolo AMQP provido pelo *broker* RabbitMQ. Esse protocolo, é um sistema de correios assíncrono entre processos, que é independente de *hardware*, sistema operacional e linguagem de programação, constituindo uma solução muito utilizada no mercado de software.

Esse protocolo funciona de maneira bi-direcional, além de promover aos desenvolvedores uma alta capacidade customização dos objetos presentes no AMQP. Esses objetos são: o(s) *publisher(s),* o *exchange*, as *bindings*, as filas (*queues*) e por fim os *consumer(s)*.

FIGURA 1: Protocolo AMQP e seus objetos.



Fonte: Medicci T. S. 2018.

Entendido, portanto, o funcionamento do protocolo AMQP parte-se para a compreensão sobre o RabbitMQ e o problema apresentado. O RabbitMQ nada mais é do que o *broker*, representado na figura 1. Através dele, é possível configurar o formato como as mensagens enviadas pelo *publisher* e que estão armazenadas no *exchange*, serão distribuídas para as filas (*queues*) pelos *bindings* programados pelo desenvolvedor.

O problema, por sua vez, não tem diferença em relação aos anteriores sobre RPC e *sockets*. É necessário criar uma aplicação distribuída que encontre os maiores e menores valores do vetor gerado pela seguinte fórmula:

$$v[i] = \sqrt{(i - \frac{tamanho\_do\_vetor}{2})^2}$$

2) Solução para um vetor de 100 posições

Para essa primeira solução, construiu-se o código que será também responsável por atender vetores de posições maiores (mais de 100). Portanto, para a arquitetura (figura 2) escolhida optou-se por utilizar uma fila principal assíncrona apenas para *requests* do cliente para o servidor, e diversas filas geradas automaticamente pelo RabbitMQ para as *responses* de comunicação dos *workers* com os *clients*.

FIGURA 2: Arquitetura de filas da solução

```
# Cria um canal exclusivo para a comunicação (request) -> assíncrono
self.canal.queue_declare(queue='fila_rpc')

# Cria um canal exclusivo para a comunicação (response)
# Worker -> Cliente
resultado = self.canal.queue_declare(queue='', exclusive=True)
self.nome_fila = resultado.method.queue

self.canal.basic_consume(
    queue=self.nome_fila,
    on_message_callback=self.on_response,
    auto_ack=True
)
```

Fonte: Autor.

Também é importante ressaltar que para a transmissão das mensagens, assim como os *sockets* que não possuem um arquivo de definição de interfaces, é necessário realizar a conversão desses valores para *strings* para então serem transmitidas do lado cliente para o servidor e vise e versa. Portanto, a solução encontrada para suprimir esse problema, foi a utilização da biblioteca json (figura 3 e 4) que converte dicionários em strings no formato JSON. Dessa forma, é possível serializar e desserilizar as informações passadas.

FIGURA 3: Serialização do lado do *client*

```
self.canal.basic_publish(
    exchange='',
    routing_key='fila_rpc',
    properties=pika.BasicProperties(
        reply_to=self.nome_fila,
        correlation_id=self.corr_id,
    ),
    # Serialização das informações em JSON
    body=json.dumps({'len': n, 'values': vec})
)
```

Fonte: Autor.

FIGURA 4: Serialização do lado do *worker*

```
vec = json.loads(corpo)
print(f'Estrutura recebida de tamanho {vec["len"]}')
response = find_values(vec['values'])
print(f'Menor valor encontrado: {response[0]}, maior

# Publica mensagem no canal exclusivo do cliente
# através propriedade reply_to settada no Client
canal.basic_publish(
    exchange='', routing_key=propriedades.reply_to,
    properties=pika.BasicProperties(
        correlation_id=propriedades.correlation_id),
    # Desserialização das informações do JSON
    body=json.dumps({
        'menor': response[0],
        'maior': response[1]
    })
)
canal.basic_ack(delivery_tag=metodo.delivery_tag)
```

Fonte: Autor.

Por fim, é interessante citar que não foram encontrados problemas ou limitações maiores para a experimentação de um vetor com apenas 100 posições. Entretanto, vale ressaltar que a solução utilizada para esse primeiro problema já é a final, ou seja, os empecilhos encontrados fazem referência a vetores com mais elementos.

3) Solução para um vetor de mais posições

Como já citado anteriormente, a solução utilizada para esse problema é a mesma que a anterior. Por esse motivo, serão explicadas aqui apenas as partes da implementação que foram feitas em especial para a resolução de vetores com muitas posições (acima de 10.000).

A principal alteração para suportar esses vetores maiores, foi a utilização de múltiplos *workers* com o auxílio de *threads* para uma execução mais paralelizada. A figura 5 e 6 demonstra como foi feita essa criação e gerência das *threads* para encontrar o menor e o maior valor dentre todas as respostas.

FIGURA 5: Criação de *threads* e envio de informações

```
for worker in range(qtd_workers):
    inicio = worker * tam_por_worker
    final = inicio + tam_por_worker
    if worker + 1 >= qtd_workers:
        final = inicio + tam_por_worker + (n % qtd_workers)

    resps = MinEMaxValor()
    thread = threading.Thread(target=thread_func, args=(inicio, final, vec, resps))
    threads.append((thread, resps))
    thread.start()
```

Fonte: Autor.

FIGURA 6: Gerência de *threads* e resolução do problema

```
maior = float('-inf')
menor = float('inf')
for thread, resps in threads:
    thread.join()
    if resps.menor < menor:
        menor = resps.menor
    if resps.maior > maior:
        maior = resps.maior
```

Fonte: Autor.

Apesar da utilização de *threads*, não foi possível realizar o envio do objetivo principal de 1.000.000.000 elementos. Isso se deve a dois principais problemas encontrados ao longo do desenvolvimento. O primeiro deles, está relacionado com o poder computacional, mais especificamente a memória RAM do computador, pois o python, por ser tipado dinamicamente, não possui uma gerência eficiente de memória. Dessa forma, o programa fica refém da quantidade de memória livre do sistema para prosseguir sua execução.

Outro ponto, que prejudicou o alcance da meta foi o limite máximo de tamanho de uma mensagem dentro de uma fila, definido pelo RabbitMQ. Apesar da identificação desse problema, é possível contorná-lo aumentando o tamanho máximo da instância do RabbitMQ, ou utilizar mais processos que criam suas filas próprias diminuindo, dessa forma, o tamanho da mensagem passada para cada *worker*.

4) Opinião geral e nota

Finalizado o laboratório, parte-se então para a opinião geral e nota. No meu ponto de vista esse laboratório foi um dos mais tranquilos pois a implementação do RPC básico já havia sido realizada na apresentação da palestra 2 sobre RabbitMQ. Entretanto, esse laboratório serviu como uma revisão e aprendizagem de alguns limites que não haviam sido explorados anteriormente. Como autoavaliação para esse laboratório, acredito que seria 10 visto que foi necessário um esforço para a criação da aplicação, apesar de já estar um pouco adiantada.

5) Referências

[1] AMQP – Protocolo de Comunicação para IoT. Medicci T. S. 2018. Disponível em: <https://www.embarcados.com.br/amqp-protocolo-de-comunicacao-para-iot/>. Acesso em: 03 de mar. de 2022.