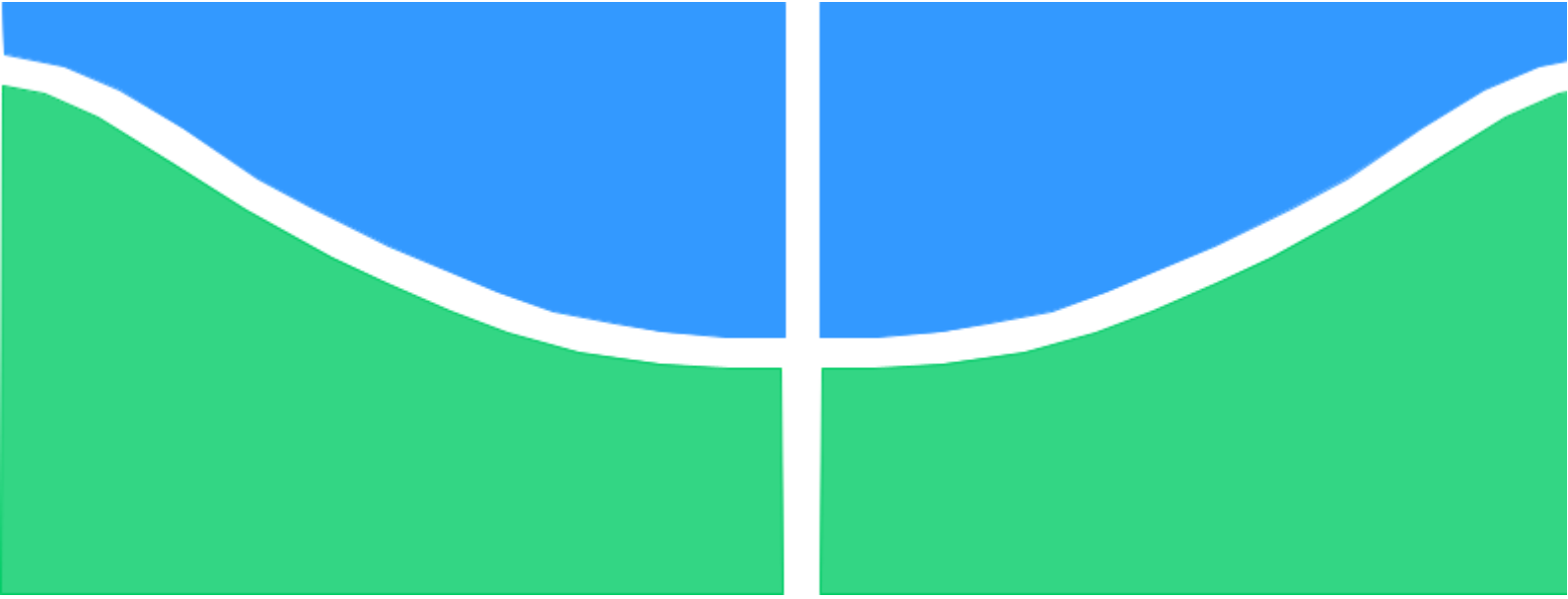

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Curso de Engenharia de Software

Lab04_distribuídos - Compreendendo funcionalidades do broker RabbitMQ

Fundamentos de Redes de Computadores - Turma A

Professor: Fernando William Cruz

Autores: Bruno Carmo Nunes - 18/0117548
Marcos Vinícius R. da Conceição - 17/0150747

Brasília, DF
2021

1. INTRODUÇÃO

O Advanced Message Queuing Protocol (AMQP) é um protocolo que roda numa camada superior do protocolo TCP, pertencendo então a camada de transporte. Ele é um protocolo que permite o envio e recebimento de mensagens de forma assíncrona independente do hardware, sistema operacional e linguagem de programação. De forma genérica, o AMQP pode ser enxergado como o equivalente assíncrono do HTTP, sendo ele um cliente capaz de comunicar-se com um broker “no meio do caminho”. [1]

O broker é uma entidade que recebe mensagens de publishers - clientes que produzem mensagens - e encaminha mensagens a consumers, que são clientes que recebem mensagens. Sendo assim, o AMQP é um protocolo bi-direcional onde um cliente pode enviar e receber mensagens de outros clientes através do broker. [1]

O RabbitMQ é um broker leve e fácil de implementar nas premissas e na nuvem. Ele suporta múltiplos protocolos de mensagens. Ele pode ser também implementado em configurações distribuidoras e federadas para encontrarem um alta escalabilidade e alta disponibilidade dos requisitos. Sendo ele também executado por vários sistema operacionais e também em ambientes cloud. Como recursos temos a seguir:

- Mensagens assíncronas
- Experiencia de desenvolvedor
- Desenvolvimento distribuído
- Ferramentas e Plugins
- Gerenciamento e Monitoramento

2. DESENVOLVIMENTO

2.1. Configuração produtor/consumidor

De uma forma resumida, o funcionamento do AMQP pode ser comparado a um sistema de correios conforme a figura 1: o cliente Publisher escreve uma mensagem e a encaminha para um exchange, que se assemelha a uma caixa de correio. A seguir, já no broker - que pode ser entendido como os Correios, por exemplo - esta mensagem é encaminhada de acordo com regras específicas (bindings) através de rotas a uma queue, que pode ser entendida como uma caixa postal, por exemplo. Finalmente, um cliente Consumer monitora a queue (caixa postal) para, então , ler a mensagem. [1]

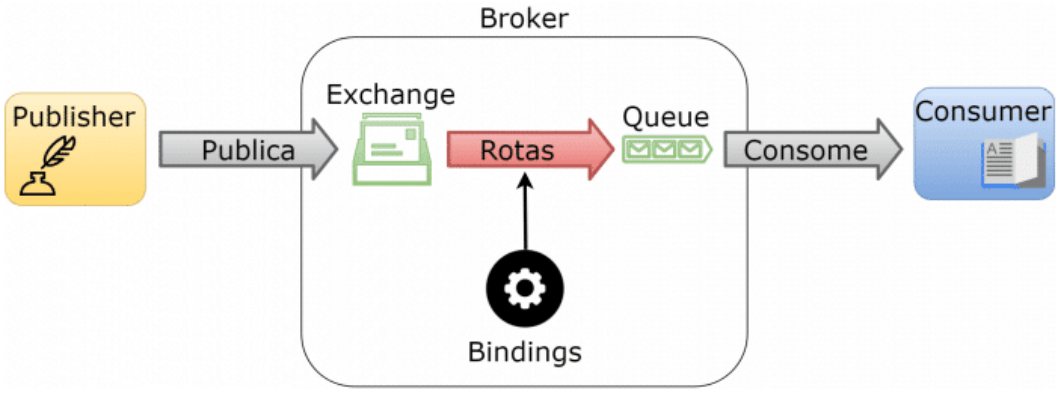

Figura 1- Diagrama do Funcionamento do Exchange tipo Direto no AMQP.

2.2. Descrição do contexto

I. Hello World!

Nesse exemplo, é somente implementado dois pequenos programas em Python; um produtor manda uma mensagem única e um consumidor recebe mensagens e exibe elas. No diagrama abaixo P é o produtor e C o nosso consumidor. A caixa vermelha no meio é o queue - buffer de mensagem que o RabbitMQ guarda pelo consumidor. O produtor manda mensagens pelo queue “hello”, e o consumidor recebe mensagens dele

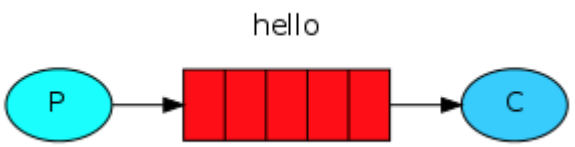

Figura 2 - Exemplo Hello World!

II. Work queues

O primeiro tutorial é programas que enviam e recebem mensagens de uma queue nomeada. Nesse é criada uma Work Queue que vai ser usada para distribuir tarefas de tempo consumidos sobre múltiplos workers. A ideia por trás do Work Queues é evitar fazer tarefas de recursos intensivos imediatamente e ter que esperar ela para completar. Invés de agendar uma tarefa para ser feita posteriormente. Tem-se o encapsulamento de uma tarefa como mensagem e é enviada para uma queue. O processo do worker rodando por trás vai disparar as tarefas e eventualmente executar o processo. Quando você executa muitos trabalhadores as tarefas serão compartilhadas entre eles.

O conceito é especialmente útil para aplicações web onde é impossível se manusear uma tarefa complexa durante uma pequena abertura de pedido HTTP.

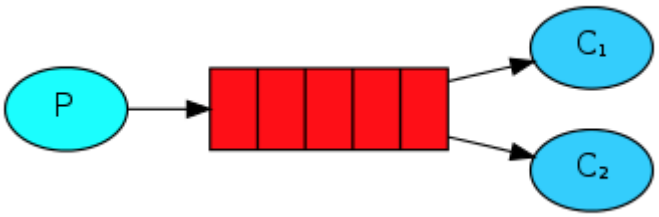

Figura 3 - Exemplo Work Queues

III. Publish/Subscribe

No tutorial passado foi criado o Work Queues, a suposição por trás do work queue é que cada tarefa é enviada exatamente para um worker. Nessa parte é feito algo completamente diferente, é entregue a mensagem para múltiplos consumidores. Esse padrão é conhecido como publish/subscribe. Para ilustrar o exemplo é construindo um simples sistema de login que consiste em dois programas, o primeiro vai emitir um log de mensagens e o segundo vai receber e imprimir ele. No sistema de login toda cópia do programa executado vai receber as mensagens enviadas. Desse jeito é possível executar um receptor e logs diretos no disco e ao mesmo

tempo é possível executar outro receptor para ver os logs na tela. Essencialmente, os logs de mensagens serão transmitidas para todos os receptores.

A ideia principal do modelo de mensagens do RabbitMQ é que o produtor nunca manda nenhuma mensagem diretamente ao queue. Na verdade, frequentemente o produtor nem sabe se a mensagem é realmente entregue ao queue. Em vez disso, o produtor só pode mandar mensagens em um exchange. Um exchange (intercâmbio) é algo bem simples, de um lado recebe mensagens de um produtor e de outro manda para as queues. O exchange tem que saber exatamente o que fazer com a mensagem que recebe. Ele deve anexar para uma queue partícula? Ele deve anexar para várias queues? Ou deve ser descartada? Essas regras são definidas pelos tipos de exchanges, sendo elas direct, topic, headers e fanout. Focando no fanout exchange, ele transmite para todos as mensagens que recebe do produtor para todas as queues, e isso é exatamente o que é necessário para o nosso logger.

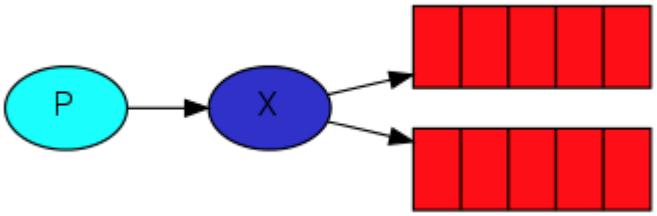

Figura 4 - Exemplo Publish/Subscribe

IV. Routing

No tutorial passado foi construído um simples sistema de login. Onde foi possível transmitir um log de mensagens para todos os receptores. Nesse tutorial iremos adicionar um novo recurso a ele - faremos então possível subscrever apenas para alguns subconjuntos das mensagens. Por exemplo, iŕa ser possível direcionar somente mensagens de erro crítico para o arquivo de log, ainda sendo possível mostrar todas os logs de mensagens no console.

No exemplo anterior foi criado um biding, que é o relacionamento entre o exchange e a queue, sendo resumidamente escrito como: a queue está interessada nas mensagens do exchange. Os bindings podem receber parametros extras como “routing_key”, para evitar a confusão com o parametro “basic_publish” que é

chamado de “binding key”. O significado da “binding key” depende do tipo de exchange, o fanout exchage que usamos previamente simplesmente ignora esse valor.

Falando sobre o Direct exchange, o nosso sistema de login transmite todas as mensagens para todos os consumidores. Queremos então estender isso permitindo a filtragem de mensagens baseadas na severidade delas. Por exemplo, queremos então um modo que o script dependa da escrita do log de mensagens para o disco que só recebe erros críticos e não desperdiça disco de espaço nesses avisos ou mensagens de logs de informações.

Quando usamos o “fanout” exchange - o que não dá muita flexibilidade - é somente capaz de transmitir mensagens estúpidas. Sendo então optado pela utilização do “direct” exchange. O algoritmo por trás do “direct” exchange é simples, a mensagem vai para a queues que a binding key combina com a routing key da mensagem.

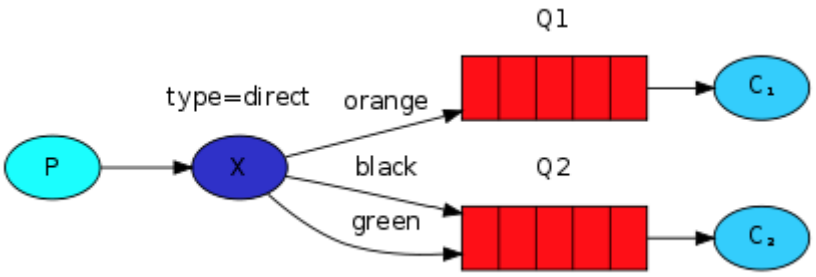

Figura 5 - Exemplo direct exchange

Nesse setup, podemos ver que o direct exchange está no X com duas queues vinculadas a ele. A primeira queue está vinculada com a binding key “orange”, e a segunda tem dois vínculos, é vinculado com a binding key “black” e outro com o “green’. Sendo assim, a mensagem publicada para o exchange com a routing key “orange” vai ser roteada para a queue Q1. Mensagens com a routing key “black” ou “green” vão para a Q2. Todas as outras mensagens irão ser descartadas.

Para os múltiplos bindings, podemos ter vários binding keys em múltiplas queues. No nosso exemplo adicionamos uma binding entre X e Q1 com a binding key “black”. Nesse caso, o direct exchange vai se comportar como um fanout que vai transmitir a

mensagem para todos os queues combinantes. A mensagem do routing key black vai ser entregue tanto para Q1 quanto para Q2.

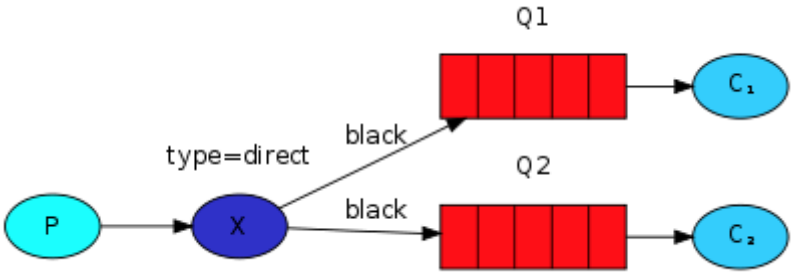

Figura 6 - Exemplo Bindings Múltiplos

Para a emissão de logs invés de fanout é mandando as mensagens pelo direct exchange. Iremos então fornecer a severidade de log como routing key.
Para o subscribing, receber as mensagens serão feitas que nem no tutorial passado, porém com uma exceção, iremos criar uma nova binding para cada severidade que estaremos interessados.

Juntando tudo teremos então diferentes mensagens enviadas por um produtor e analisadas pelo exchange, para direcionar a queues diferentes que vão ser redirecionadas a consumidores diferentes, como mostra a figura a seguir.

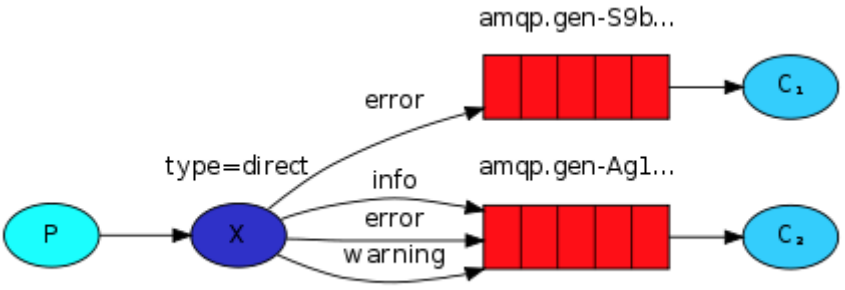

Figura 7 - Exemplo Routing

V. Topics

As mensagens enviadas para uma troca de tópico não podem ter um routing_key arbitrário - deve ser uma lista de palavras, delimitada por pontos. As palavras podem ser qualquer coisa, mas geralmente especificam alguns recursos conectados à mensagem. Alguns exemplos de chave de roteamento válidos: "stock.usd.nyse", "nyse.vmw", "quick.orange.rabbit". Pode haver quantas palavras você quiser na chave de roteamento, até o limite de 255 bytes.

A chave de ligação também deve estar no mesmo formato. A lógica por trás da troca de tópicos é semelhante a uma direta - uma mensagem enviada com uma chave de roteamento específica será entregue a todas as filas vinculadas a uma chave de ligação correspondente. No entanto, existem dois casos especiais importantes para chaves de ligação:

* (estrela) pode substituir exatamente uma palavra.
# (hash) pode substituir zero ou mais palavras.

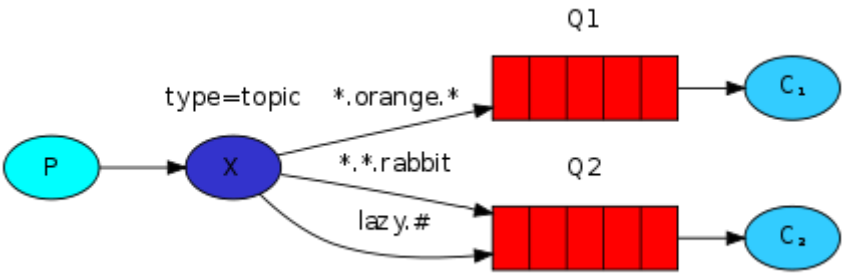

Figura 8 - Criação de tópicos no rabbitMQ

A troca de tópicos é poderosa e pode se comportar como outras trocas.

Quando uma fila é vinculada com a chave de ligação "#" (hash) - ela receberá todas as mensagens, independentemente da chave de roteamento - como na troca de fanout.

Quando os caracteres especiais "*" (estrela) e "#" (hash) não são usados nas ligações, a troca de tópico se comporta de maneira direta.

VI. RPC

Uma chamada de procedimento remoto é iniciada pelo cliente enviando uma mensagem para um servidor remoto para executar um procedimento específico. Uma resposta é retornada ao cliente. Uma diferença importante entre chamadas de procedimento remotas e chamadas de procedimento locais é que, no primeiro caso, a chamada pode falhar por problemas da rede. Nesse caso, não há nem mesmo garantia de que o procedimento foi invocado.

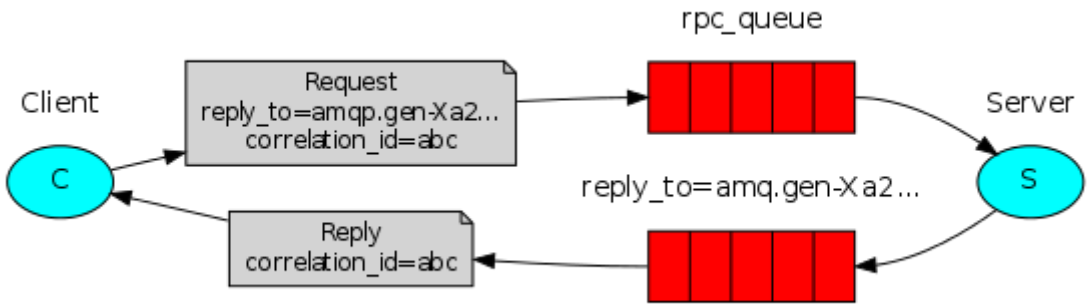

Figura 9 - Representação funcionamento RPC no rabbitMQ

Quando o Cliente é inicializado, ele cria uma fila de retorno de chamada exclusiva anônima.

Para uma solicitação RPC, o Cliente envia uma mensagem com duas propriedades: reply_to, que é definido para a fila de retorno de chamada e Correlation_id, que é definido com um valor único para cada solicitação.

A solicitação é enviada para uma fila rpc_queue.

O worker RPC (também conhecido como: servidor) fica esperando por solicitações nessa fila.
Quando uma solicitação aparece, ele faz o trabalho e envia uma mensagem com o resultado de volta para o Cliente, usando a fila do campo reply_to.
O cliente espera por dados na fila de retorno de chamada. Quando uma mensagem aparece, ele verifica a propriedade Correlation_id.
Se corresponder ao valor da solicitação, ele retornará a resposta ao aplicativo.

VII. Publisher Confirms

O Publish Confirms é uma extensão do RabbitMQ para implementar uma publicação confiável. Quando as confirmações do editor são habilitadas em um canal, as mensagens que o cliente publica são confirmadas de forma assíncrona pelo corretor, o que significa que foram atendidas no lado do servidor.

2.3. Descrição da solução

A solução do problema não foi complicada pois os tutoriais explicavam e publicaram de maneira direta os scripts de cada exemplo. Sendo assim todos foram rodados como servidor e cliente na mesma máquina proporcionando a solução de cada exemplo proposto. O problema maior foi na execução do RabbitMQ, onde no começo não foi feita a execução dele pelo docker pois ocasionava erros na instalação do mesmo. Foi feita então a instalação diretamente no sistema operacional Ubuntu onde teve que ser feita mudanças nos artefatos dos scripts propostos pelos tutoriais.
Tivemos problemas na execução do último tutorial “Publish Confirms” pois foi feito em Java o que acarretou vários problemas na interpretação do código, onde fizemos inteiramente os tutoriais em python. Apesar de termos então estudado o funcionamento para a interpretação do problema, não conseguimos testar pela IDE o código proposto na solução porque ocasionou erros nas bibliotecas do Java.

3. CONCLUSÃO

Este laboratório nos deu a oportunidade de entender uma parte das inúmeras capacidades que um broker como o RabbitMQ é capaz de oferecer. Fundamentamos sobre a finalidade e importância de um Broker como sendo o responsável por permitir estruturar aplicações distribuídas através de componentes desacoplados que interagem através de invocações remotas, coordenando todas as comunicações realizando o encaminhamento de requisições, resultados, exceções e caso necessário possui a capacidade de guardar e documentar toda a movimentação do sistema através dos seus logs. Oferecendo ao sistema um local comum para o envio e recebimento de mensagens além de um local seguro com garantia que a mensagem será entregue ou recebida caso ocorra algum problema no cliente ou servidor.

A utilização do broker neste laboratório se mostrou bem mais simples que a implementação de maneira direta dos sockets e mesmo a utilização do RPC na linguagem C, em parte pela facilidades que o python oferece, fora que com a utilização do broker podemos nos preocupar mais na implementação da solução de um problema e menos em como irá ocorrer “por debaixo dos panos” a comunicação entre o cliente e servidor.

4. REFERÊNCIAS

[1] **AMQP - Protocolo de comunicação para IoT**, acessado dia 2 de setembro de 2021.

[2] RabbitMQ - **https://www.rabbitmq.com/getstarted.html** , acessado dia 2 de setembro de 2021.

[3] What is remote desktop connection broker? , acessado dia 2 de setembro de 2021.