Continuando a série sobre mensageria, hoje vamos descer na toca do coelho.

O que é

O RabbitMQ é um servidor de mensageria feito para suportar AMQP. Foi escrito em Erlang e segundo seu site, é robusto, fácil de usar, roda nos principais sistemas operacionais, suporta enorme número de plataformas de desenvolvimento, é Open Source, sua licença é Mozzila 1.1 e pode ter suporte comercial.

Onde aprender sobre o RabbitMQ

Site: https://www.rabbitmq.com/

Diversas apresentações em (algumas são excelentes): http//www.rabbitmq.com/how.html

Livro: RabbitMQ in Action (de onde tirei muito do que está escrito abaixo)

Um link muito interessante principalmente para os rubistas está no Runy Inside em Why Rubyists Should Care About Messaging (A High Level Intro)

InfoQ, September, 2009: Getting started with AMQP and RabbitMQ

Link antigo mas vem de quem fez o RabbitMQ: RabbitMQ Tech Talk at Google London, September 2008

Blog em português: RabbitMQ, AMQP e Spring-AMQP

Inclui alguns links sobre AMQP no primeiro artigo desta série em Introdução à mensageria

Download e instalação do RabbitMQ

Instalação no OSX usando Homebrew

Antes de começar verifique se existe o diretório /usr/local/etc e caso exista, veja a quem pertence. No meu caso ele não existia e primeiro fiz mkdir /usr/local/etc.

A seguir verifique a quem pertence o diretório /usr/local/sbin Eu precisei fazer sudo chown -R lucabastos /usr/local/sbin

E eu já inclui logo no meu ~/.bash_profile a linha export PATH=${PATH}:/usr/local/sbin

Depois basta seguir as instruções do link de download e instalação com Homebrew . Atenção que demora um bocado.

Como o PATH já aponta certo, para inicializar o servidor basta escrever: rabbitmq-server Use a opção -detached para inicializar em background como daemon

Se optar por rodar no terminal, aparecerá o logo do RabbitMQ em formato ASCII e na última linha broker running.

Para confirmar, abra um outro terminal e escreva: rabbitmqctl status Este comando rabbitmqctl tem um monte de opções, é o canivete suiço do RabbitMQ.

Para finalizar o servidor em outro terminal que não aquele onde o RabbitMQ está rodando: rabbitmqctl stop

Um pouco de administração do RabbitMQ

Se precisar mudar algo na configuração,
veja https://www.rabbitmq.com/configure.html#configuration-file

Se ocorrer algum erro e precisar ver os logs, na inicialização do RabbitMQ sem ser como daemon, aparece no terminal onde ficam os logs. Na minha instalação estão em /usr/local/var/log/rabbitmq/rabbit@nomeminhamáq-meuusuário.log

Para finalizar o servidor rodando em outro nó, passe a opção -n rabbit@[hostname] (nó é um conceito do Erlang que é mais do que dizer que nó é uma instância do RabbitMQ. É um conceito que não cabe explicar aqui)

Dentro do conceito de nó que não vou explicar, é possível parar somente a aplicação RabbitMQ sem parar as outras aplicações Erlang que rodam no mesmo nó. Para isso o comando é: rabbitmqctl stop_app

Para reiniciliazar o RabbitMQ, é preciso usar o comando com rabbitmqctl stop e a seguir usar rabbitmq-server

E o RabbitMQ também tem um console web acessado como https://localhost:55672. Mas primeiro é preciso habilitar o plugin. Como coloquei o sbin no PATH, para habilitar o plugin fiz rabbitmq-plugins enable rabbitmq_management e depois reinicializei o RabbitMQ. Ao entrar no site, é pedido usuário e senha. Usei guest/guest e entrei. Daí para frente a coisa complica e é preciso conhecer um pouco o RabbitMQ para fuçar com sucesso.

Além da interface web, o RabbitMQ ainda fornece uma interface de linha de comando CLI e uma ferramenta chamada de Command Line Tool, que na verdade é um script Python de nome rabbitmqadmin (que para funcionar precisa ser instalada e ser dada permissão para execução)

Mensagem no RabbitMQ

Uma mensagem tem duas partes:

1. payload – corpo com os dados transmitidos. Pode ser qualquer coisa desde um array JSon até mesmo um filminho MPEG

2. label – cabeçalho que descreve o payload e também como o RabbitMQ sabe quem deve receber a mensagem.

Os consumidores se conectam ao broker e subscrevem uma “fila” como se fosse uma caixa de entrada de emails. Quando chega uma nova mensagem, o RabbitMQ envia para um dos consumidores que subscreveram ou estão “escutando” a fila. O consumidor recebe apenas o payload.

O RabbitMQ não informa quem mandou a mensagem. Caso esta informação seja importante, cabe ao produtor incluí-la dentro do payload.

Conexão TCP e o conceito de canais

Antes de consumir ou publicar uma mensagem, é preciso se conectar, isto é, criar uma conexão TCP entre sua aplicação e o broker. Uma vez conectado e autenticado no broker, sua aplicação cria um canal AMQP que nada mais é do que uma conexão virtual dentro da conexão TCP real. Cada canal tem um ID único que quem trata é a API.

Abrir uma conexão TCP real é uma operação custosa e o próprio sistema operacional limita o número de quantas podem estar abertas. Mas não há limite de quantos canais uma conexão TCP real pode ter. Este conceito de conexão virtual, permite repetir a criação de canais centenas ou milhares de vezes, sem mexer com o sistema operacional. Quando uma thread começa, um novo canal é criado sem onerar a pilha TCP. Assim muitas threads compartilham uma conexão real TCP.

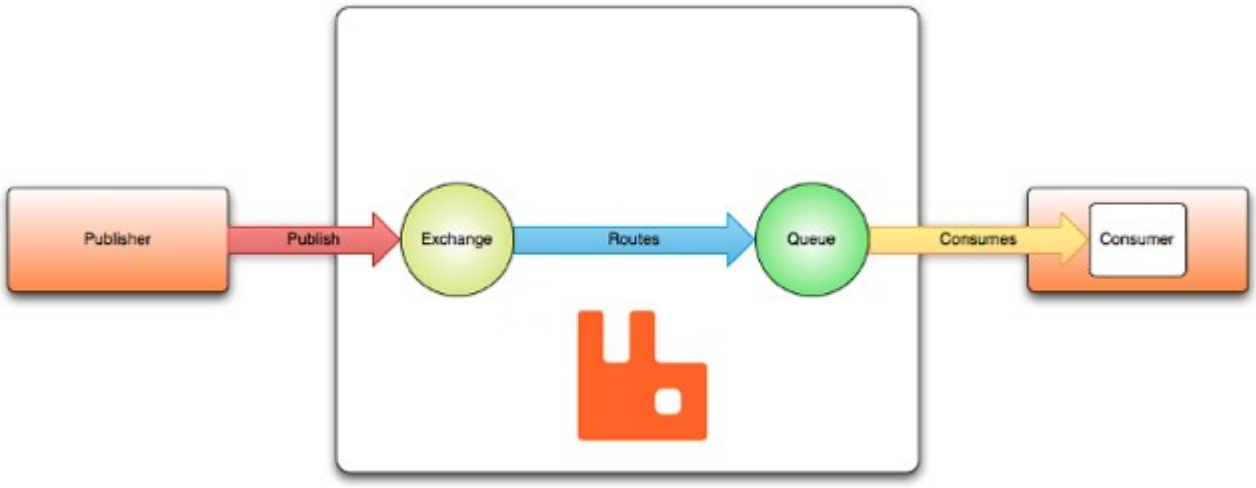
Filas

Conceitualmente são três os componentes em um roteamento com sucesso de uma mensagem AMQP:

– exchanges – são onde os produtores publicam suas mensagens

– queues – onde as mensagens ficam e são recebidas pelos consumidores

– bindings – modo como as mensagens são roteadas do exchange para uma determinada queue



Os consumidores podem pegar mensagens na fila usando dois comandos AMQP:

- – basic.consume – coloca o canal em modo de receber mensagens da fila até que se termine a subscrição. O consumidor recebe uma mensagem disponível na fila assim que consumir ou rejeitar a mensagem anterior. Modo que deve ser usado para processar muitas mensagens e/ou receber automaticamente.

- – basic.get – consume só a próxima mensagem. Obs.: nunca deve ser usado em loop como alternativa ao comando basic.consume porque onera o processamento do RabbitMQ

As filas e os consumidores

Se existem um ou mais subscritores em uma fila, as mensagens são roteadas imediatamente a eles. Caso não exista nenhum consumidor subscrito, a mensagem espera na fila. Tão logo apareça algum subscritor, a mensagem é enviada àquele subscritor.

Quando vários consumidores subscrevem na mesma fila, as mensagens recebidas são servidas à moda round-robin. Cada mensagem é enviada a apenas um consumidor registrado.

Exemplo: Seja uma queue chamada Q que os consumidores Bob e Alice subscreveram. As mensagens chegando em Q são distribuídas do seguinte modo:

1. Mensagem M1 chega em Q
2. RabbitMQ envia M1 para Bob
3. Bob envia uma mensagem do tipo ACK confirmando que recebeu corretamente (usando comando “basic.ack” ou com o parâmetro “auto_ack” em True quando subscreveu a fila Q)
4. RabbitMQ retira mensagem M1 da fila
5. Mensagem M2 chega em Q
6. RabbitMQ envia M2 para Alice
7. Alice envia uma mensagem ACK confirmando que recebeu corretamente
8. RabbitMQ retira mensagem M2 da fila

Se um consumidor X por algum motivo não envia o ACK, o RabbitMQ considera que a mensagem não foi enviada corretamente e a reenvia para o próximo subscritor da fila. O tal consumidor X não recebe nova mensagem enquanto não enviar o ACK ou reiniciar sua conexão ou ainda enviar um comando “basic.reject”. Com o parâmetro “requeue” como True no comando “basic.reject”, o RabbitMQ envia a tal mensagem para o próximo. Porém se “requeue” for False, o RabbitMQ remove a mensagem da fila imediatamente.

Tanto consumidores como produtores podem criar filas com o comando AMQP “queue.declare”. Entretanto os consumidores não podem fazer isto quando já estiverem com subscrição em outra fila do mesmo canal. Este comando pode configurar várias propriedades da fila como por exemplo “exclusive” para ter uma fila privada e “auto-delete” mandando apagar a fila automaticamente tão logo o último consumidor desista da subscrição.

Se sua aplicação não admite perder mensagens, tanto o produtor como o consumidor devem tentar criar as filas que precisam. É importante que ambos tentem porque se um produtor tentar enviar uma mensagem para uma fila que ainda não existe, a mensagem será descartada pelo RabbitMQ.

A queue é o bloco fundamental na troca de mensagens AMQP

– é onde a sua mensagem espera para ser consumida

– facilita o load balancing. Basta juntar um grupo de subscritores na fila e deixar o mecanismo de round robin do RabbitMQ distribuir as mensagens igualmenta aos consumidores

– é o ponto final de qualquer mensagem no RabbitMQ (a menos daquelas sem fila que são descartadas)

Exchanges e bindings

Uma fila é dita ligada (bound) a uma exchange por uma routing key. Não é obrigatorio o uso desta routing key mas usar abre um leque de cenários interessantes de trocas de mensagens que seriam impossíveis ou muito difíceis de conseguir se o broker apenas permitisse publicar mensagens diretamente na filas.

Bem, o broker roteia mensagens do exchange para a queue baseado na routing key. Mas como enviar mensagens para múltiplas filas? Então para atender a este caso e outros, o protocolo AMQP prevê 4 tipos de exchanges, cada um implementando um algoritmo diferente de roteamento:

– **direct** – simples, se a routing key da mensagem bate com a da fila, é para esta que vai. Se vem em branco e existe uma fila sem routing key, então vai para esta.

– **fanout** – neste caso o exchange fará o multicast da mensagem para todas as filas a ele ligadas

– **topic** – este tipo de exchange permite que mensagens oriundas de diferentes fontes cheguem a fila

– **headers** – permite definir uma espécie de routing key para ser usada no lugar da outra. Fora isto, funciona igual ao normal porém com muito pior desempenho. O benefício é pouco para tanta perda e quase nunca é usado.

Esta arquitetura de mensageria é bastante poderosa. Para deixar isto mais claro precisaria me aprofundar muito mais do que meu objetivo mas penso que já deu para perceber o potencial.

Virtual hosts

Dentro de cada servidor RabbitMQ podemos criar message brokers virtuais chamados de virtual hosts (vhosts). Cada um é como se fosse um mini RabbitMQ com seus próprios exchanges, bindings, queues e principalmente suas próprias permissões.

Funcionando como se fosse uma VM, permite usar com total segurança o mesmo servidor RabbitMQ para aplicações diferentes sem medo de uma alterar uma fila da outra. Também serve para separar múltiplos clientes evitando colisões de nomes de filas e exchanges. Não fosse isso, seria necessário rodar múltiplas instâncias do RabbitMQ e provavelmente com mais trabalho de administrar.

O que ocorre com os exchanges e as queues quando o RabbitMQ precisa ser reinicializado?

Há uma propriedade em cada queue e em cada exchange chamada “durable”, que por padrão é False. Ela diz ao RabbitMQ se ele precisa recriar as queues e os exchanges ao iniciar. Mas o pior é que não basta colocar “durable” como True. Uma mensagem que sobrevive a um crash do broker AMQP é chamada de “persistent”. Para isto precisa que seja colocado o valor 2 na opção “delivery

mode” (o seu programa deve usar alguma constante legível para seres humanos). Em resumo, para uma mensagem sobreviver a um crash, precisa:

– ter sua opção “delivery mode” configurada como 2 (persistent)

– ser publicada em um exchange com durable = True

– chegar em uma fila igualmente com durable = True

– que o crash não tenha sido justamente no dispositivo de armazenamento.

É claro que agindo assim ocorre uma penalização no desempenho (falam em até 10x). Uma outra consideração é que este mecanismo “persistent/durable” não funciona bem em cluster, que é uma das coisas fáceis de fazer com o RabbitMQ por ele ter sido escrito em Erlang.

Como decidir? Bem, caso tenha muitas mensagens para processar, só pense no modo “persistent/durable” se tiver um dispositivo de armazenamento muitíssimo rápido. No caso de cluster, talvez seja interessante separar um cluster para as mensagens que não precisam de “persistent/durable” e um outro par de máquinas não clusterizadas com o RabbitMQ no esquema “persistent/durable”, sendo uma ativa e outra hot-standby com balanceamento de carga.