

# Sistema de Mensageria

Narciso de Sousa Rodrigues, Paulo Henrique de C. Silva, Saul Sousa da Rocha

**Abstract**—Com o avanço das novas tecnologias e a necessidade constante de novas funcionalidades para as aplicações complexas, temos a necessidade de uma descentralização de componentes a fim de melhorar o desempenho dessas novas aplicações. Microserviços traz exatamente esta ideia, onde toda aplicação complexa pode ser dividida entre componentes com funções bem definidas.

A maior forma de uso de microserviços se dá por meio de tecnologias de comunicação WEB como SOAP e REST. O grande problema é que o uso tem um alto poder de gerar sobrecargas na rede o que pode ocasionar perdas de mensagens. Estas perdas de mensagens acabam no futuro gerando um mau funcionamento nas aplicações.

Para resolver este problema temos sistemas que aumentam a disponibilidade e a partição a falhas que são conhecidos como “sistemas de mensagerias” onde em vez da comunicação direta entre os componentes temos um sistema simples que exerce um papel de broker na entrega de mensagens entre os componentes na rede.

**Index Terms**—Microserviços, Comunicação, Sistema de Mensageria.

## I. INTRODUÇÃO

Dentro do cenário atual da tecnologia e das técnicas de desenvolvimento temos a necessidade de uma maneira de abstrair a complexidade entre componentes de uma aplicação, a fim de tornar estas aplicações o mais otimizadas possível. Neste cenário deste desafio surgiu a ideia dos microserviços.

Microserviços tem como base que toda aplicação complexa pode ser dividida entre componentes com funções bem definidas. Com a arquitetura de microserviços espera-se que os componentes tenham baixo acoplamento, consequentemente, a substituição deles ocorrerá de forma mais rápida dando ao sistema mais flexibilidade.

Uma vez que os componentes estão em unidades de execução espalhadas localmente ou globalmente, a comunicação entre elas se torna um ponto crítico no ambiente real de execução. As tecnologias de comunicação baseadas na WEB, como SOAP e REST, são bem desenvolvidas, e vêm sendo aplicadas por várias aplicações, especialmente as que exigem uma comunicação planetária, como por exemplo: Netflix, Youtube, Google Drive e etc.

Apesar das tecnologias de comunicação WEB se encontrarem bem desenvolvidas, seu uso nativamente por microserviços tem um alto poder de gerar sobrecargas nas redes e, consequentemente, perdas de mensagens. Isso se deve ao fato de que os microserviços geralmente lidam com grandes volumes de dados em tempo real, o que pode sobrecarregar as redes.

Uma técnica para aumentar a disponibilidade e a partição da falha é utilizar um sistema de mensageria em vez da comunicação direta entre os componentes, reduzindo sobrecarga de comunicações cruzadas, além de possíveis rejeições de mensagens devido a sobrecarga na rede.

## II. REFERENCIAL TEÓRICO

A tabela abaixo demonstra os principais softwares de mensageria, logo após uma breve descrição. Isto com o intuito de demonstrar as soluções já existentes e para que seja possível uma análise do software proposto neste documento.

TABLE I  
REFERENCIAL TEÓRICO

| Software         | Funcionalidade   | Aprendizagem               |
|------------------|--|----------------------------|
| [1] Apache Kafka | Alto grau de funcionalidades além do sistema de mensageria                                 | Alto grau de aprendizagem  |
| [2] RabbitMQ     | Ideal para casos onde apenas é necessário o uso da mensageria                              | Baixo grau de aprendizagem |
| [3] Amazon SQS   | Ideal para quando o sistema já está integrado a AWS  | Médio grau de aprendizagem |
| [4] ActiveMQ     | Ideal para sistemas que estejam em desenvolvimento ou já tenham sido desenvolvidos em Java | Médio grau de aprendizagem |
| Este Software    | Ideal para representação de como um sistema de mensageria funciona                         | Baixo grau de aprendizagem |

### III. DESENVOLVIMENTO

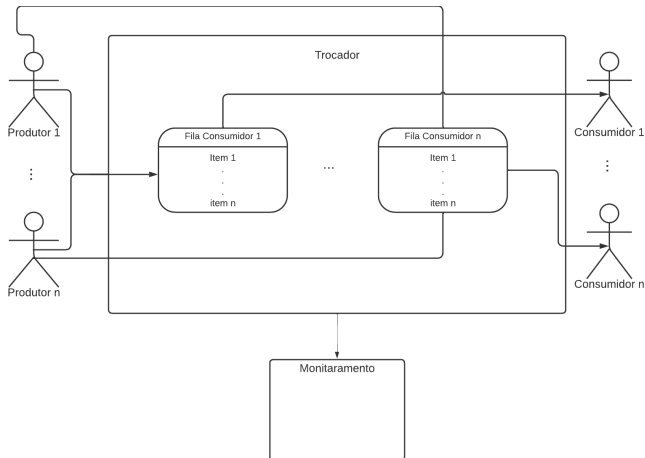


Fig. 1. Diagrama do sistema de mensageria.

O sistema se divide em varias partes, sendo elas o Trocador, que é o processo que funcionará como *Middleware* e deve gerenciar o uso de cada fila. O Produtor, que é responsável por produzir os conteúdos que serão encaminhados através do Trocador para seus respectivos tópicos. O Consumidor que será responsável por receber os conteúdos produzidos pelos Produtores. E também a parte do monitoramento do sistema, que pode ser visto em tempo real via *WEB*.

A comunicação entre esses processos são através de *RPC* (*Remote Procedure Call*) auxiliada pelo pacote python *Pyro4*, uma biblioteca que permite criar aplicativos nos quais os objetos podem se comunicar pela rede, com o mínimo de esforço de programação. O mais interessante disso é que ocultamos boa parte da comunicação do sistema.

A figura 1 mostra o funcionamento do sistema de mensageria desenvolvido. Basicamente o sistema terá um servidor remoto na qual o Trocador disponibilizará seus métodos, em que Produtores e Consumidores tem acesso. O sistema poderá ter N produtores e N Consumidores (fica a critério do usuário), na qual os Produtores irão produzir os conteúdos e os Consumidores irão capturar os conteúdos em suas respectivas filas. Vale resaltar que Os produtores podem produzir conteúdo para qualquer Consumidor.

Os Produtores e Consumidores funcionam por *Threads*, que são criadas assim que são instaciadas pelo usuário. E ao criar um Consumidor, também será criada sua fila de consumo, na qual o mesmo irá retirar os conteúdos remetidos a ele. Quanto ao *Fanout*, no sistema ele é considerado um tópico diferente dos demais, pois ele encaminhará os conteúdos para todas as filas de consumidores.

### IV. EXECUÇÃO E RESULTADOS

Segue o passo a passo de como é feita a execução de cada uma das imagens em seus respectivos containers:

Primeiro, faça o download de todos os containers ([www.encurtador.com.br/ftDIJ](http://www.encurtador.com.br/ftDIJ)) e os armazene em uma única pasta para que fique mais fácil a execução, como visto na figura 2 abaixo.

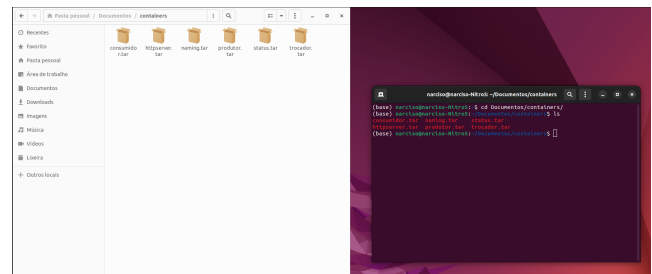


Fig. 2. Demonstração pasta de arquivos com os containers em .tar.

Em sequencia é feito o carregamento dos arquivos .tar em imagens docker, para que possam ser executados. Isso é executado pelo comando *docker load*, vale lembrar que no print está sendo executado no linux, no windows pode haver a necessidade de substituir o ;(menor que) por *-input*, de modo que o código passaria de *docker load ; naming.tar* para *docker load -input naming.tar*, segue o exemplo de execução na figura 3.

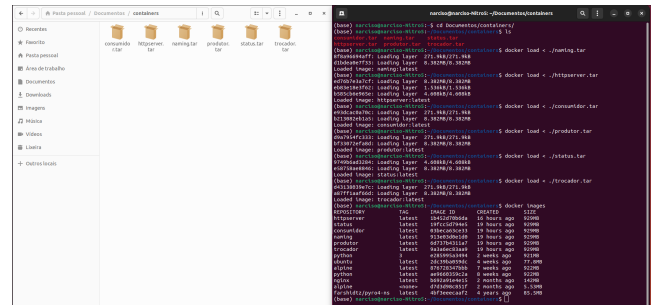


Fig. 3. Carregamento das imagens para o Docker.

A partir do carregamento logo podemos verificar que todas as imagens estão no docker, por sua vez prontas para execução.

Diante das imagens prontas para execução, vamos a execução de cada container. Preferencialmente siga a sequencia, pelo menos dos containers *naming* (*docker -name naming -it naming*) e *trocador* (*docker run -name trocador -it trocador*), pois fazem necessário para execução de demais processos. Segue o passo a passo de execução dos dois containers principais do sistema de mensageria na figura 4.



Fig. 4. Execução do naming server.

Assim que o naming é executado, basta que execute o trocador.

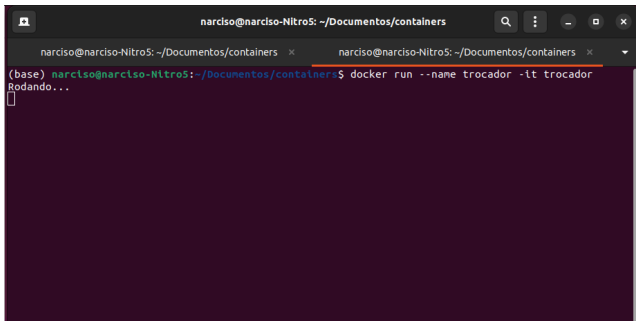


Fig. 5. Execução do container trocador.

Diante a execução dos dois primeiros processos principais, fica a seu critério a ordem de execução dos demais, abaixo vamos executar o Produtor(*docker run --name produtor -it produtor*) para que o sistema comece a possuir informações e do Consumidor para que haja consumo das informações captadas pelo sistema, como mostra a figura 6 e figura 7.

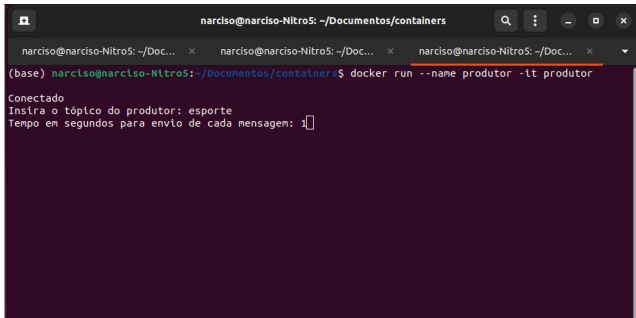


Fig. 6. Execução do container produtor.

Agora para execução do consumidor, basta que você execute o código *docker run --name consumidor -it consumidor* como mostra a figura 7 abaixo.

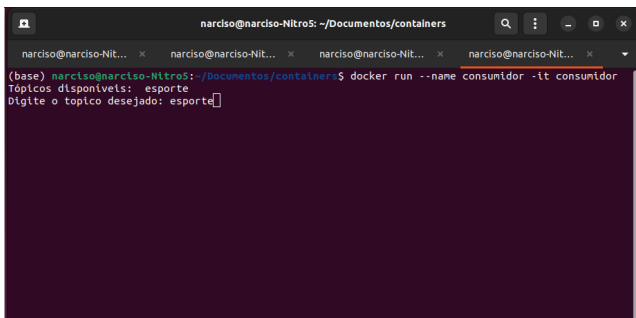


Fig. 7. Execução do container consumidor.

Para visualização de características mais específicas você deve executar o container status, este a execução é um pouco diferente, pois usaremos o bash do container para execução do mesmo, abaixo na figura 8 temos o exemplo de execução do container e do status para exibição dos consumidores conectados. Este se dá de maneira diferente para que não exista a necessidade de executar um container toda vez que for verificar o status. Primeiramente você inicializa o container o acessando via terminal com o comando: *docker run --name*

*status -it status bash*, após está inicializado o container o bash ficará aberto e você pode executar a verificação de status com auxílio das tags c(consumidor), p(produtor), f(filas), F(também fila onde passa o nome da fila e deve ser usado conjuntamente com as tags t(tamanho) ou s(estatística)), deste modo você pode executar por exemplo: *python status.py -c* para verificar os consumidores ativos, *python status.py -F esporte -t* para verificar a quantidade de mensagens do tópico esporte.

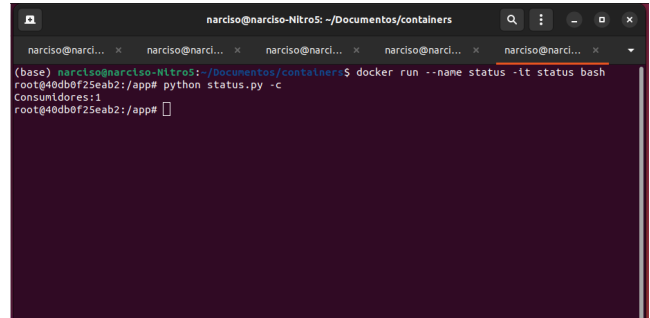


Fig. 8. Execução do container status.

Dando sequencia, para melhor visualização das informações vamos executar o servidor http para que possa ser feita visualização os dados diretamente no navegador, desta maneira facilitando o acesso. Primeiramente executa-se o código do container do mesmo, que é parecido com o do status: *docker run --name httpserver -it httpserver bash*, para que seja aberto o prompt de comando do docker httpserver, na sequência se executa o comando padrão python para criação de servidor http: *python -m http.server*, vale lembrar que se for executar no windows é necessária a designação das portas do container, de modo que o código de inicialização ficaria assim: *docker run --name httpserver -p 8000:8000 -it httpserver bash* e como foi designada a porta 8000 é necessário que o http server seja inicializado nessa porta, utilizando o código *python -m http.server 8000*. segue o passo a passo na figura 9:

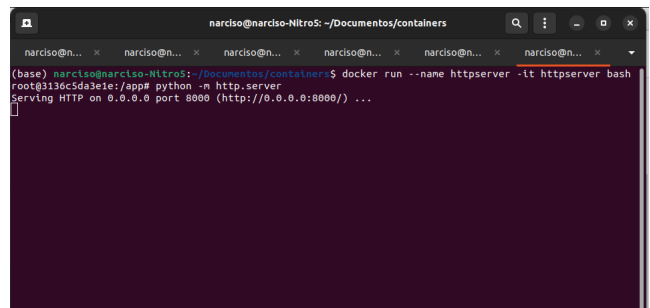
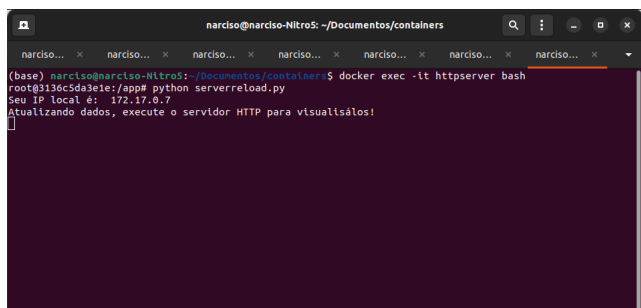


Fig. 9. Execução do container httpserver e inicialização do httpserver python.

Acima vemos que o servidor HTTP está em execução na porta 8000, você pode escolher essa porta incrementando ao fim da linha de execução do servidor o número da porta: *python -m http.server (número da porta)*. Agora falta somente a execução do código que atualizará as informações do servidor HTTP bem como nos exibirá o IP para que possamos executar no nosso navegador, caso você use o servidor windows vale lembrar que o acesso será feito pelo *localhost:8000*

com a porta designada, neste caso a 8000. Feita a execução do servidor faz-se necessária a execução do atualizador das informações que estão sendo processadas, usando o código: `docker exec -it httpserver bash` para acessar o container httpserver que está em execução e `python serverreload.py`



```
narciso@narciso-Nitro5: ~/Documentos/containers
narciso... x narciso... x narciso... x narciso... x narciso... x narciso... x narciso... x
(base) narciso@narciso-Nitro5:~/Documentos/containers$ docker exec -it httpserver bash
root@3136c5da3e1e:/app# python serverreload.py
Seu IP local é: 172.17.0.7
Atualizando dados, execute o servidor HTTP para visualizá-los!
```

Fig. 10. Execução do container httpserver para atualização dos dados.

Na figura 10 acima vemos o IP. A porta foi exibida no *print* anterior a este representado pela figura 9, caso acesse utilizando as características do windows essas informações são irrelevantes pois você já setou de maneira manual o localhost e a porta para acesso. Com as informações em mãos pode-se acessar os dados pelo navegador como demonstrado abaixo na figura 11.



Fig. 11. Demonstração servidor http funcionando e retornando os dados atualizados automaticamente.

## REFERENCES

- [1] N. Garg, *Apache kafka*. Packt Publishing Birmingham, UK, 2013.
- [2] D. Dossot, *RabbitMQ essentials*. Packt Publishing Ltd, 2014.
- [3] J. Varia, S. Mathew *et al.*, “Overview of amazon web services,” *Amazon Web Services*, vol. 105, 2014.
- [4] B. L. Gessner and M. M. Mattos, “Protótipo de sistema de troca de mensagens em delphi baseado em apache activemq.”