Gustavo José Neves da Silva

Marlon Henry Schweigert

Análise de disponibilidade utilizando Docker Swarm

Joinville

2018

UNIVERSIDADE DO ESTADO DE SANTA CATARINA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Gustavo José Neves da Silva
Marlon Henry Schweigert

ANÁLISE DE DISPONIBILIDADE UTILIZANDO DOCKER SWARM

Trabalho de conclusão de curso submetido à Universidade do Estado de Santa Catarina como parte dos requisitos para a obtenção do grau de Bacharel em Ciência da Computação

Charles Christian Miers
Orientador

Joinville, Junho de 2018

ANÁLISE DE DISPONIBILIDADE UTILIZANDO DOCKER SWARM

Gustavo José Neves da Silva

Marlon Henry Schweigert

Este Trabalho de Conclusão de Curso foi julgado adequado para a obtenção do título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo Curso de Ciência da Computação Integral do CCT/UDESC.

Banca Examinadora

Charles Christian Miers - Doutor (orientador)

-

-

Agradecimentos

AGRADECIMENTOS

Resumo

A crescente popularização de jogos massivos demanda por novas abordagens tecnológicas a fim de suprir as necessidades dos usuários com menor custo de recursos computacionais. Projetar essas arquiteturas, do ponto de vista da rede, é algo pertinente e impactante para o sucesso desses jogos. O objetivo deste trabalho é propor uma análise voltada a identificar abordagens para otimização dos recursos computacionais consumidos pelas arquiteturas identificadas. Esse objetivo será atingido após realizar uma pesquisa referenciada, seguida de uma análise das principais arquiteturas e, preferencialmente, a execução de simulações usando uma nuvem computacional para auxiliar na identificação de gargalos de recursos. Os resultados obtidos auxiliarão provedores de serviços *Massively Multiplayer Online Role-Playing Game* (MMORPG) a reduzir gastos de manutenção e melhorar a qualidade de tais serviços.

Palavras-chaves: Arquitetura de microsserviços, Docker, Docker Swarm

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

API *Application Programming Interface*

HTTP *Hypertext Transfer Protocol*

JSON *JavaScript Object Notation*

MMORPG *Massively Multiplayer Online Role-Playing Game*

PaaS *Platform as a Service*

REST *Representational State Transfer*

1 Introdução

1.1 Fundamentação Teórica

A fim de orientar o análise encontrada neste trabalho, se faz necessário a descrição e pesquisa por fundamentação teórica dos conceitos básicos relevantes no contexto de microsserviços e implantação dessas arquiteturas. Por esse motivo, esta seção irá introduzir os conceitos de microsserviços, containers, docker e implantação utilizando a técnica de docker swarm.

1.1.1 Microserviços

Entende-se por microsserviço, aplicações que executam operações menores de um macrosserviço, da melhor forma possível (WILLSON, 2017; NEWMAN, 2015). O objetivo de uma arquitetura de microsserviços é funcionar separadamente de forma autônoma, contendo baixo acoplamento (NEWMAN, 2015). Seu funcionamento deve ser desenhado para permitir alinhamentos de alta coesão e baixo acoplamento entre os demais microsserviços existentes em um macrosserviço (ACEVEDO; JORGE; PATIñO, 2017).

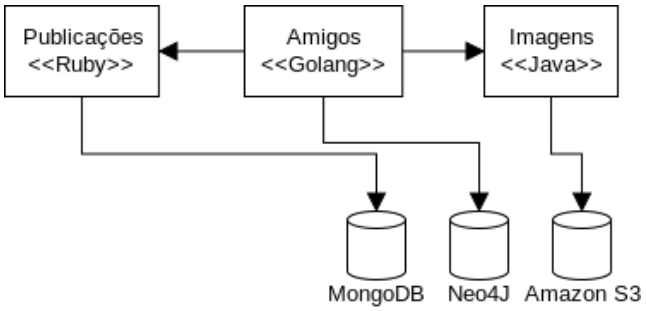
Arquiteturas de microsserviços iniciam uma nova linha de desenvolvimento de aplicações preparadas para executar sobre nuvens computacionais, promovendo maior flexibilidade, escalabilidade, gerenciamento e desempenho, sendo a principal escolha de arquitetura de grandes empresas como Amazon, Netflix e LinkedIn (KHAZAEI et al., 2016; VILLAMIZAR et al., 2016). Um microsserviço é definido pelas seguintes características (ACEVEDO; JORGE; PATIñO, 2017):

- Deve possibilitar a implementação como uma peça individual do macrosserviço.
- Deve funcionar individualmente.
- Cada serviço deve ter uma interface. Essa interface deve ser o suficiente para utilizar o microsserviço.

- A interface deve estar disponível na rede para chamada de processamento remoto ou consulta de dados.
- O serviço pode ser utilizado por qualquer linguagem de programação e/ou plataforma.
- O serviço deve executar com as dependências mínimas.
- Ao agregar vários microsserviços, o macrosserviço resultante poderá prover funcionalidades complexas.

O microsserviço deverá ser uma entidade separada. A entidade deve ser implantada como um sistema independente em um *Platform as a Service* (PaaS). Toda a comunicação entre os microsserviços de um macrosserviço será executada sobre a rede, a fim de reforçar a separação entre cada serviço. As chamadas pela rede com o cliente ou entre os microsserviços será executada através de uma *Application Programming Interface* (API), permitindo a liberdade de tecnologia em que cada um será implementado (NEWMAN, 2015). Isso permite que o sistema contenha tecnologias distintas que melhor resolvam os problemas relacionados ao contexto deste microsserviço. Isso pode ser visualizado na Figura 1.1.

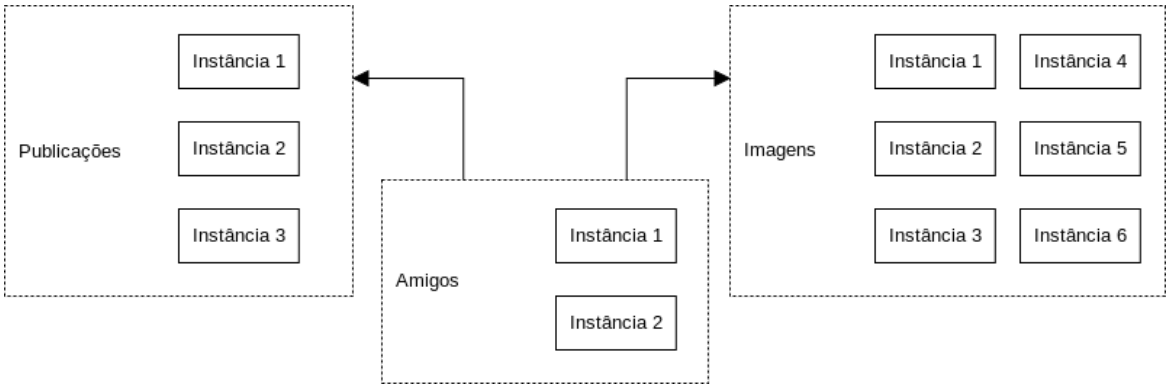
Figura 1.1: Microsserviços podem ter diferentes tecnologias



Adaptado de: (NEWMAN, 2015)

Uma arquitetura de microsserviços é escalável, como visível na Figura 1.2. Ela permite o aumento do número de microsserviços sob demanda para suprir a necessidade de escalabilidade. Este modelo computacional obtém maior desempenho, principalmente se executar sobre plataformas de computação elástica, na qual o orquestrador do macrosserviço pode aumentar o número de instâncias conforme a necessidade de requisições (NADAREISHVILI et al., 2016).

Figura 1.2: Microsserviços são escaláveis



Adaptado de: (NEWMAN, 2015)

Microsserviços desenvolvidos para web utilizam arquitetura *Representational State Transfer* (REST) baseado sobre o protocolo *Hypertext Transfer Protocol* (HTTP). É uma boa prática utilizar o corpo com conteúdo da requisição e resposta no formato *JavaScript Object Notation* (JSON) nas chamadas a uma API de microsserviço web (NADAREISHVILI et al., 2016).

Entretanto, necessita-se de um método para garantir o seu isolamento, a qual, em sistemas Linux, podem ser garantidos utilizando Containers. Por este motivo, se faz necessário descreve-lo.

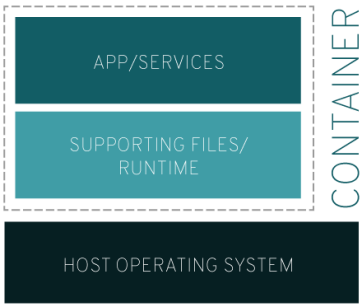
1.1.2 Containers

Os container permitem ao desenvolvedor ”empacotar”sua aplicação e todas as suas dependências(ex. bibliotecas), dessa forma, lhe é garantido que a aplicação terá o mesmo comportamento independentemente do hospedeiro Linux.

Um container Linux é um conjunto de processos que executam de forma isolada do restante do sistema. Esses processos utilizam arquivos providos de uma imagem, a qual lhe garante a compatibilidade a fim de evitar problemas de versionamento e conflitos de processos. Essa separação pode ser visualizada na Figura 1.3.

Entretanto, se faz necessário uma ferramenta para gerenciamento de imagens e containers. Entra em cena a partir de 2008 o *Docker*, uma ferramenta que permite, de forma prática, a publicação de imagens no formato Dockerfile, além de contar com um gerenciador e repositório de imagens.

Figura 1.3: Containers sobre sistema linux



Fonte: (RedHat, 2018)

1.1.3 Docker

Docker é uma plataforma que nos permite ”construir, emabarcar e rodar uma aplicação em qualquer lugar”. Ele percorreu um longo caminho em um período de tempo incrivelmente curto e atualmente é considerado uma solução padrão para um dos apectos mais custosos do software: a implantação (citar Docker in Pratice)

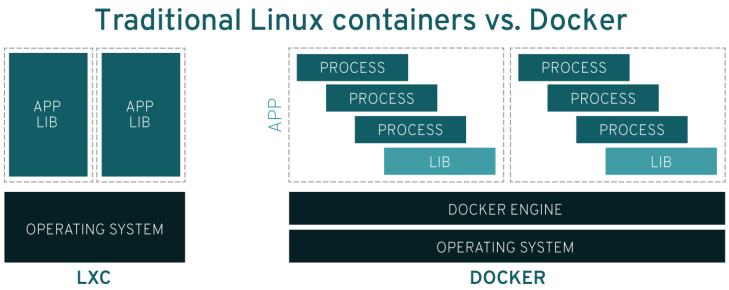
Docker é uma ferramenta desenvolvida para facilitar o processo de criação, implantação e execução de aplicações por meio do uso de containers. Pode ser pensado como um tipo de maquina virtual, porém diferentemente desta que necessita a criação de todo um sistema operacional virtual, o Docker, permite que as aplicações compartilhem o mesmo kernel Linux que o hospedeiro, reduzindo assim o tamanho da aplicação e obtendo um ganho de desempenho.

A tecnologia Docker usa o *kernel* Linux, abstraindo sistemas como *Cgroups* e *namespaces* para segregar processos a fim que eles possam ser executados de forma independente. O objetivo dos containers criados pelo Docker continua da mesma forma que os containers Linux, conhecida como *LXC*. A diferença entre Docker e *LXC* é relevante a escalabilidade, visto que containers Docker permitem multiplos processos executando juntamente a uma biblioteca, já o padrão *LXC* permite somente um processo junto a sua biblioteca, consumindo mais recursos da máquina. Essa comparação pode ser visualizada na Figura 1.4.

1.1.4 Docker Swarm

É uma ferramenta nativa do Docker que permite criar clusters de containers, chamado swarms, o que possibilitam a escalabilidade de recursos de acordo com a demanda(carga)

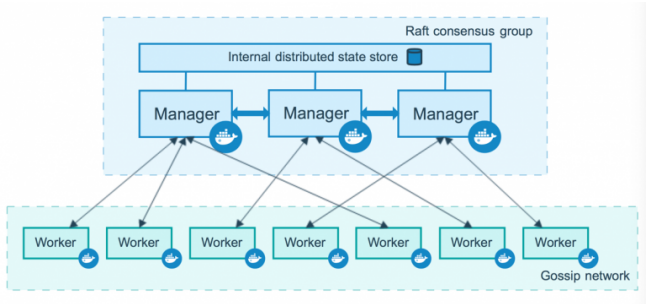
Figura 1.4: Tecnologia Docker em comparação aos containers Linux (LXC).



Fonte: (RedHat, 2018)

(citar The Docker Book) O que possibilita que diversos hospedeiros de Docker estejam inseridos no mesmo pool de recursos, facilitando assim a implantação de containers, uma vez que o Swarm disponibiliza uma API de integração que abstrai grande parte das atividades necessárias a administração dos conteiners e promove um tipo de tolerancia a falhas

Figura 1.5: Rede de Docker Swarm.



Fonte: (RedHat, 2018)

O seu principal objetivo é resolver problemas de gerência de microsserviços, a qual antes eram resolvidos somente com *Kubernets*, uma ferramenta criada pela Google em 2015. Com Docker Swarm, pode-se ter um nó líder que gerenciará a rede, e nós trabalhadores. Um exemplo de rede Docker Swarm pode ser visualizado na Figura 1.5.

1.2 Histórico

1.2.1 Arquitetura Monolítica

(Pensar) Arquitetura Monolítica, é uma arquitetura de desenvolvimento, na qual típicamente , apesar da complexidade dos sistemas ser quebrada ao se utilizar módulos, esses são projetados para a criação de um único executável, o qual possui todos os seus módulos

executados em um mesma máquina. Com o passar do tempo, o sistema cresce e tende a tornar-se cada vez mais complexo, o que gera diversos problemas em sua manutenção, por exemplo temos, a escalabilidade do sistema, que exige que o mesmo seja replicado inteiramente, mesmo que apenas uma parte desse seja necessária na nova instância, aumentando assim os custos.

Com a necessidade de escalabilidade, as arquiteturas de microsserviços obteram sucesso em grandes projetos comerciais como LinkedIn, Google e Youtube.

1.3 Funcionamento

1.4 Boas práticas

- Não armazenar dados em containers, uma vez que esses podem ser parados, destruidos ou mesmo substituidos, se ncessário armazenar dados deve ser feito em um volume, com o cuidado de evitar que dois ou mais containers escrevam dados em um mesmo volume, o que poderia causar o corrompimento dos dados.
- Não criar imagens grandes, já que essas possuirão uma distribuição complexa, uma imagens de possuir apenas as bibliotecas e arquivos necessários para a execução da aplicação ou processo.
- Não executar mais de um processo/aplicação em um único container, pois tal comportamento acarretará em aumento da complexidade do gerenciamento e no número de logs.
- Não dos endereços IP dos containers, o endereço IP do container pode se alterado quando o mesmo é iniciado e parado. Caso seja necessária a comunicação entre a aplicação ou microsserviço e um outro container, é recomendado o uso de variáveis de ambiente para transmitir o hostname e porta corretos de um container para outro.

1.5 Principais aplicações

1.5.1 Aplicações Web

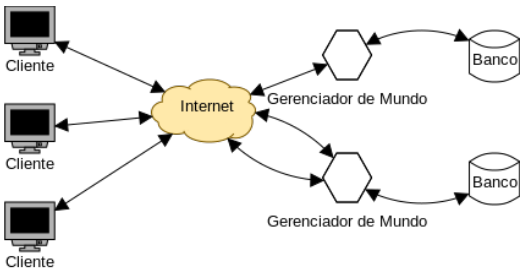
Microsserviços desenvolvidos para web utilizam arquitetura REST baseado sobre o protocolo HTTP. É uma boa prática utilizar o corpo com conteúdo da requisição e resposta no formato JSON nas chamadas a uma API de microsserviço web (NADAREISHVILI et al., 2016).

1.5.2 Streaming

1.5.3 Jogos

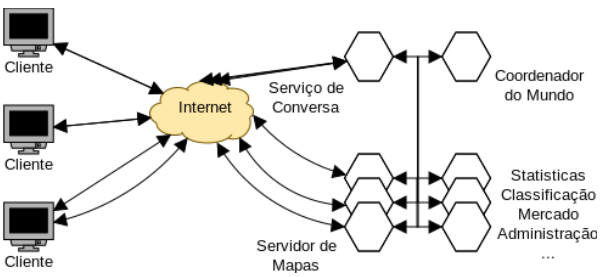
Alguns exemplos de arquitetura de microsserviços para jogos MMORPG são as arquiteturas apresentadas por Rudy (Figura 1.6), Salz (Figura 1.7) e a arquitetura escrita por Knowles (Figura 1.8).

Figura 1.6: Arquitetura Rudy, utilizada no jogo Tibia.



Adaptado de: (RUDDY, 2011)

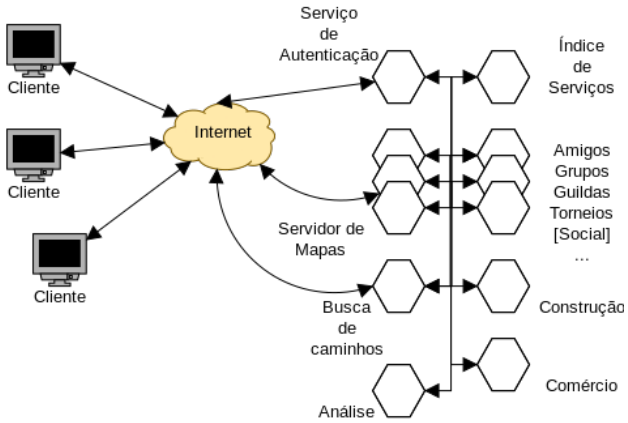
Figura 1.7: Arquitetura Salz, utilizada no jogo Albion.



Adaptado de: (SALZ, 2016)

A arquitetura Rudy (Figura 1.6) é formada por um sistema cliente-servidor monolítico, na qual cada microsserviço individual gerencia um mundo mútuo dos demais

Figura 1.8: Arquitetura Knowles, utilizada no jogo Guild Wars 2.



Adaptado de: (WILLSON, 2017)

gerenciadores de mundo (RUDDY, 2011). Essa arquitetura dificulta a escalabilidade, modificações e manutenção (ACEVEDO; JORGE; PATIñO, 2017), além de segregar a comunidade de jogadores em servidores menores (RUDDY, 2011). Inicialmente essa arquitetura foi pensada para ser um sistema Cliente-Servidor monolítico. A arquitetura Rudy é uma arquitetura de microsserviços adaptada de um serviço cliente-servidor (RUDDY, 2011). O jogo Tibia[1], operante sobre essa arquitetura, possui 68 mundos oficiais (Sendo 2 servidores de teste) (RUDDY, 2011), com capacidade para 1.050 clientes em cada servidor, na qual encontra-se restringido pelo gerenciador de mundo.

A arquitetura Salz (Figura 1.7) é formada por diversos microsserviços (SALZ, 2016). O principal objetivo dessa arquitetura é modularizar o serviço visando melhorar a escalabilidade. Ela é atualmente utilizada no jogo Albion Online[2]. A arquitetura é planejada para funcionar conforme a seguinte especificação(SALZ, 2016):

- O mundo é distribuído sobre os vários servidores de mapas. Cada microsserviço gerencia uma região do mundo, denominado *chunk*.
- Jogadores mudam a conexão com os microsserviços quando estão posicionados na borda de um *chunk*.
- A autorização de acesso aos microsserviços é obtido pelo banco de dados.
- O coordenador do mundo é responsável por tudo que seja de escopo global (*e. g.*, Grupos, chat global, guildas, *etc.*).

[1]Tibia: **http://www.tibia.com**
[2]Albion Online: **https://albiononline.com**

A arquitetura Knowles (Figura 1.8) é distribuída em diversos microsserviços, assim como a arquitetura Salz (Figura 1.7). A diferença em comparação a arquitetura Salz (Figura 1.7) está na decomposição da arquitetura para outros microsserviços e a conexão direta entre esses microsserviços e o cliente. O principal objetivo dessa arquitetura é facilitar a manutenção e desempenho de reinicialização do macrosserviço (WILLSON, 2017). Guild Wars 2[3] é um jogo que executa sobre a arquitetura Knowles. Ele é popularmente conhecido por ter seus serviços sempre ativos, visto que a arquitetura possibilita desativar pequenos pedaços do serviço para manutenções básicas e a sua reinicialização é rápida para manutenções críticas.

[3]Guild Wars 2: **https://www.guildwars2.com**

2 Casos Comentados

2.1 Walmart

2.2 Spotify

2.3 Amazon

2.4 Guild Wars 2

3 Análise

3.1 Método de deploy

3.2 Arquitetura obtida

3.3 Análise sobre a arquitetura obtida

4 Conclusão

Conclusão

Referências

ACEVEDO, C. A. J.; JORGE, J. P. G. y; PATIñO, I. R. Methodology to transform a monolithic software into a microservice architecture. In: *2017 6th International Conference on Software Process Improvement (CIMPS)*. Zacatecas, Mexico: IEEE, 2017. p. 1–6.

KHAZAEI, H. et al. Efficiency analysis of provisioning microservices. In: *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. Luxembourg, Austria: IEEE, 2016. p. 261–268.

NADAREISHVILI, I. et al. *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly Media, 2016. ISBN 978-149195625-0. Disponível em: <https://www-.amazon.com/Microservice-Architecture-Aligning-Principles-Practices/dp/1491956259>.

NEWMAN, S. *Building Microservices*. O'Reilly Media, 2015. ISBN 978-149195035-7. Disponível em: <https://www.amazon.com.br/Building-Microservices-Sam-Newman-/dp/1491950358>.

RedHat. *O que é um container Linux?* 2018. [Online; accessed 24. May 2018]. Disponível em: <https://www.redhat.com/pt-br/topics/containers/whats-a-linux-container>.

RUDDY, M. *Inside Tibia, The Technical Infrastructure of an MMORPG*. 2011. Disponível em: <http://twvideo01.ubm-us.net/o1/vault/gdceurope2011/slides-/Matthias_Rudy_ProgrammingTrack_InsideTibiaArchitecture.pdf>.

SALZ, D. *Albion Online - A Cross-Platform MMO (Unite Europe 2016, Amsterdam)*. 2016. Disponível em: <https://www.slideshare.net/davidsalz54/albion-online-a-crossplatform-mmo-unite-europe-2016-amsterdam>.

VILLAMIZAR, M. et al. Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In: *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. Cartagena, Colombia: IEEE, 2016. p. 179–182.

WILLSON, S. C. *Guild Wars Microservices and 24/7 Uptime*. 2017. Disponível em: <http://twvideo01.ubm-us.net/o1/vault/gdc2017/Presentations/Clarke-Willson_Guild Wars 2 microservices.pdf>.