Análise de desempenho dos orquestradores: Kubernetes e Docker Swarm

Nikolas Jensen[1], **Charles C. Miers**[1]

[1] Departamento de Ciência da Computação (DCC)
Universidade do Estado de Santa Catarina (UDESC)

nikolas.j@edu.udesc.br, charles.miers@udesc.br

Resumo. *Os contêineres visam uma melhor eficiência no uso dos recursos computacionais ao mesmo tempo que fornecem um ambiente virtualizado para aplicações, mas crescem as preocupações sobre a segurança destes. Para utilizar o melhor desta tecnologia surgiram os orquestradores, os quais realizam a manipulação dos contêineres de uma aplicação distribuída. Este trabalho tem por objetivo apresentar os resultados iniciais de uma análise de desempenho de dois orquestradores comumente empregados: Docker Swarm e Kubernetes.*

1. Introdução

Os orquestradores oferecem diversas possibilidades aos utilizadores, podem tanto realizar o serviço de *front-end* quanto de *back-end* [Marathe et al., 2019], balanceando o volume de trabalho entre diversos nós. É possível oferecer serviços de diversas maneiras, e.g., *Infrastructure as a Service* (IaaS), *Load Balance as a Service* (LBaaS), *Software as a Service* (SaaS), etc. A orquestração conjuntamente aos contêineres oferece uma considerável gama de serviços a serem ofertados. O Kubernetes é um orquestrador mais robusto, com mais funcionalidades do que o Docker Swarm, as quais são o resultado de mais de 15 anos de desenvolvimento pela Google [Pan et al., 2019]. Contudo, de outra maneira o Kubernetes apresenta considerável complexidade na hora de configurá-lo [Pereira Ferreira and Sinnott, 2019], sendo assim, uma consequência da ampla gama de funcionalidades. Porém, existem diversos outros critérios a serem abordados que devem ser levados em consideração na escolha do orquestrador para uma aplicação. Neste trabalho são abordados alguns critérios de análise de desempenho, é utilizada a ferramenta *Sysbench* para realizar a comparação do desempenho dos orquestradores estudados. Também é testada a resistência do orquestrador quando submetido a ataques *Denial of Service* (DoS). Os *benchmarks* realizados visam comparar os orquestradores em relação a utilização da CPU.

2. Orquestração de contêineres

Os contêineres podem ser utilizados em um servidor ao número de milhares, então, foi necessária a criação de orquestradores, uma vez que se torna uma tarefa complexa e desnecessária para humanos realizarem. A orquestração de contêineres ocorre de diversas maneiras, e.g., controle do volume de trabalho por contêiner, este possui um controlador de recursos do nó e um agendador de tarefas para ditar o que cada um deve fazer [Zhou et al., 2020]. Utilizar a orquestração de contêineres garante uma melhor utilização dos recursos computacionais, assim reduzindo custos de operação de uma organização [Rodriguez and Buyya, 2018]. A orquestração de contêineres na nuvem e no meio empresarial, atualmente, deixou de ser um opcional, e passou a ser necessária. Os dois principais orquestradores são:

- Docker Swarm: são diversos contêineres Docker reunidos [Marathe et al., 2019], de maneira que trabalhem de forma conjunta para uma mesma aplicação. O Docker Swarm é baseado na arquitetura mestre-trabalhador [Dhumal and Janakiram, 2020], possuindo um nó mestre que define o que cada nó trabalhador deve fazer. Segundo [Swarm, 2020], o Docker Swarm oferece comunicação entre contêineres e nós via *overlay*, a qual é baseada numa rede virtual que é gerada para a comunicação dos serviços.
- Kubernetes: traz mais dificuldades na hora executar um simples *cluster*, mas é possível utilizar diversos tipos de contêineres, i.e., Docker, LXC, LXD, etc. Por mais que o Kubernetes proporcione mais eficiência na orquestração, sua configuração é mais complexa, tornando sua curva de aprendizado íngreme [Pereira Ferreira and Sinnott, 2019], ou seja, é uma tarefa mais desafiadora para dominar a ferramenta.

3. Definição de problema e cenário

Segundo [Chelladhurai et al., 2016], os ataques DoS vêm sendo um problema crítico, podendo tornar um sistema mais lento ou até mesmo inutilizável. Um ataque DoS é utilizado por um atacante para um sistema computacional não funcionar como o desejado, ou então, comprometer a disponibilidade de recursos [Masdari and Jalali, 2016]. Os ataques DoS são simples de serem realizados, e.g., um laço infinito, com alocação de memória sendo realizada a cada iteração causando um esgotamento da memória do sistema. No trabalho [Buyya et al., 2018] são utilizados de 16 a 22 contêineres para realizar os testes, nestes testes foram utilizados 30 contêineres. Além disso, para a comparação dos orquestradores é utilizado o mecanismo de xadrez *Stockfish* [Costalba, 2021] como aplicação e um código de DoS junto a ser executado em paralelo para verificar utilização de CPU com o *Sysbench* no decorrer do tempo. A ferramenta Sysbench oferece diversos testes para verificar o desempenho da máquina, neste trabalho foram utilizados os testes de CPU. Segundo [Manpages, 2020] no teste de CPU são realizados experimentos para verificar se determinados números são primos ou não, o maior número do intervalo fica a critério do usuário.

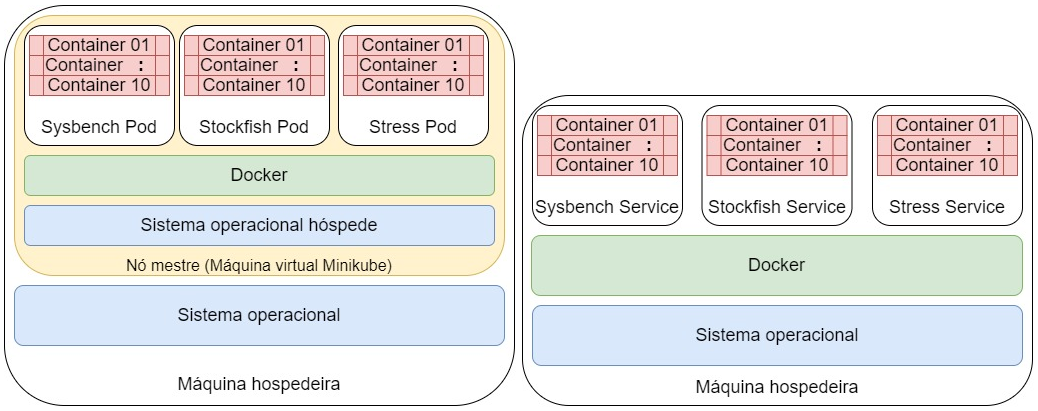
4. Critérios

O critério utilizado é a verificação da utilização da CPU dos contêineres. Além disso, foi verificado o desempenho do *cluster* quando submetido a um ataque do tipo DoS básico. Para realizar a simulação de um ataque DoS, utilizou-se uma ferramenta de *Stress*, a qual é disponibilizada no Docker Hub [polinux, 2021]. Como parâmetros da ferramenta de *Stress*, foi utilizado 200 como número de trabalhadores para utilizar a CPU. Para avaliar o desempenho da CPU são utilizados diversos valores no parâmetro "–cpu-max-prime", os quais, quanto mais alto mais lento será o processamento. No trabalho de [Pereira Ferreira and Sinnott, 2019] foi utilizado 30.000 como parâmetro para o máximo de números primos. Neste trabalho foi utilizado 100.000 como parâmetro, para exaurir ainda mais a CPU. No mecanismo de xadrez, utilizou-se o comando "go infinite", o qual calcula infinitas jogadas num jogo aleatório.

5. Experimentos

Como parte do *cluster* são utilizados contêineres que contém o *Sysbench*, o *Stockfish* e um *Stress* para esgotar de recursos da CPU. São gerados 10 contêineres por imagem utilizada,

i.e., 30 contêineres no total. O Kubernetes é executado na máquina local, utilizando a máquina virtual ”Minikube”o qual configura um *cluster* (Figura 1 (a)). Cada um será executado em um *Pod*, que é a menor parte de um *cluster* Kubernetes. No Docker Swarm são utilizados os contêineres Docker reunidos num Swarm (Figura 1 (b)).



(a) Cenário no Kubernetes. (b) Cenário no Docker Swarm.

Figura 1. Cenários dos experimentos.

Os resultados iniciais dos testes realizados (Tabela 1), revelam uma pequena diferença nos tempos de execução dos dois orquestradores. O Kubernetes apresentou melhores resultados em todos os testes realizados, um número maior de eventos por segundo, sendo assim, gerando um tempo de execução e tempo total menor. Nos testes realizados, foram empregados 30 contêineres dentro de um nó, mas em um *cluster* com mais nós e contêineres a diferença no desempenho pode ser considerável.

Tabela 1. Resultados preliminares.

	Eventos por segundo, desvio padrão e mediana	Tempo total (segundos), desvio padrão e mediana	Tempo de execução (segundos), desvio padrão e mediana
Kubernetes	$3.598 \pm 1.257 \pm 3.070$	$10.176 \pm 0.196 \pm 10.099$	$10.162 \pm 0.196 \pm 10.096$
Docker Swarm	$3.506 \pm 0.478 \pm 3.480$	$10.198 \pm 0.126 \pm 10.198$	$10.192 \pm 0.127 \pm 10.181$

Nota-se que o desvio-padrão do Kubernetes foi maior que o do Docker Swarm, mostrando que houve mais variação de valores, principalmente nos eventos por segundo. O que gerou um desvio-padrão maior foi um único valor que estava acima do normal entre os outros. Diferentemente do desvio-padrão o Kubernetes apresentou valores menores que o Docker Swarm. Em contrapartida, o tempo de configuração do Kubernetes, para realizar os testes, foi de aproximadamente duas horas. Para realizar a configuração do Docker Swarm, foram necessário em torno de 10 minutos.

6. Considerações e Trabalhos futuros

Mesmo que o Kubernetes forneça um desempenho melhor, a complexidade de configuração dele pode ser um empecilho para a escolha de seu uso numa aplicação que não necessite de muitos recursos ou do melhor desempenho possível. Os resultados deste trabalho podem ser levados em consideração quando for necessário realizar a escolha de um orquestrador. Os resultados obtidos ainda são dados preliminares, e para trabalhos futuros pretende-se realizar mais testes, em diferentes situações, e.g., mais ambientes, ataques, maior número

de contêineres e nós, etc. Os ataques futuros abrangerão mais recursos computacionais para testá-los. Os testes realizados neste trabalho serão validados com mais experimentos para gerar maior confiabilidade.

Agradecimentos: Os autores agradecem o apoio do LabP2D/UDESC e da FAPESC.

Referências

Buyya, R., Rodriguez, M. A., Toosi, A. N., and Park, J. (2018). Cost-efficient orchestration of containers in clouds: A vision, architectural elements, and future directions. *CoRR*, abs/1807.03578.

Chelladhurai, J., Chelliah, P. R., and Kumar, S. (2016). Securing docker containers from denial of service (dos) attacks. pages 856–859.

Costalba, M. (2021). Stockfish. https://github.com/mcostalba/Stockfish.

Dhumal, A. and Janakiram, D. (2020). C-balancer: A system for container profiling and scheduling.

Manpages, D. (2020). Sysbench(1). https://manpages.debian.org/testing/sysbench/sysbench.1.en.html.

Marathe, N., Gandhi, A., and Shah, J. M. (2019). Docker swarm and kubernetes in cloud computing environment. In *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*, pages 179–184.

Masdari, M. and Jalali, M. (2016). A survey and taxonomy of dos attacks in cloud computing. *Security and Communication Networks*, 9(16):3724–3751.

Pan, Y., Chen, I., Brasileiro, F., Jayaputera, G., and Sinnott, R. (2019). A performance comparison of cloud-based container orchestration tools. pages 191–198.

Pereira Ferreira, A. and Sinnott, R. (2019). A performance evaluation of containers running on managed kubernetes services. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 199–208.

polinux (2021). Stress in a docker (alpine). https://hub.docker.com/r/polinux/stress.

Rodriguez, M. A. and Buyya, R. (2018). Containers orchestration with cost-efficient autoscaling in cloud computing environments.

Swarm, D. (2020). Swarm mode key concepts. https://docs.docker.com/engine/swarm/key-concepts/.

Zhou, N., Georgiou, Y., Zhong, L., Zhou, H., and Pospieszny, M. (2020). Container orchestration on hpc systems.