Universidade Federal de Alagoas
Instituto de Computação

Trabalho de Conclusão de Curso

Provisionamento automático de infra-estrutura para processamento de atividades em segundo plano

Paulo Victor Vieira de Melo
paulovictorv@gmail.com

Orientador:

Prof. Dr. Leandro Melo Sales

Maceió, 05 Fevereiro de 2018

Paulo Victor Vieira de Melo

Provisionamento automático de infra-estrutura para processamento de atividades em segundo plano

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciências da Computação do Instituto de Computação da Universidade Federal de Alagoas.

Orientador:

Prof. Dr. Leandro Melo Sales

Maceió, 05 Fevereiro de 2018

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciências da Computação do Instituto de Computação da Universidade Federal de Alagoas, aprovada pela comissão examinadora que abaixo assina.

Prof. Dr. Leandro Melo Sales - Orientador
Instituto de Computação
Universidade Federal de Alagoas

Prof. Dr. André Lage - Examinador
Instituto de Computação
Universidade Federal de Alagoas

Prof. Dr. Rafael Lemos - Examinador
Instituto de Computação
Universidade Federal de Alagoas

Maceió, 05 Fevereiro de 2018

Agradecimentos

Agradeço em primeiro lugar a meus pais, pois sem o apoio incondicional nenhum objetivo teria sido alcançado. A minha irmã, por estar sempre presente nos momentos que precisei, e também nos momentos que não precisei. Agradeço a minha noiva e futura esposa, Rafaele Oliveira, por me ajudar a alinhar todos os pontos deste trabalho. Por fim, agradeço a Carlos Azevedo, ex-aluno da UFAL e grande amigo. Se não fosse a sua paciência em explicar-me o que cada parte de um computador faz, 21 anos atrás, talvez este trabalho não estivesse sendo apresentado.

Resumo

Com o avanço da tecnologia e a introdução da computação em nuvem em larga escala, as operações de usuários com um sistema não são mais triviais. Ao atingir um número elevado de requisições, faz-se necessário expandir a capacidade computacional. Nos dias de hoje, é comum utilizar provedores de infra-estrutura como serviço (*Infrastructure as a Service - IaaS*), os quais são capazes de fornecer capacidade computacional sob demanda, através de máquinas virtuais. Entretanto, a decisão de criar uma máquina virtual envolve um custo operacional por ser um processo em sua grande parte manual, cabendo ao ser humano tomar a decisão de qual configuração de máquina virtual criar. Apesar de existirem técnicas tanto na literatura quanto na indústria para automatizar esse processo de decisão, nenhum deles possui como objetivo a facilidade de uso por parte do desenvolvedor. Neste trabalho é proposta uma ferramenta capaz de automatizar o provisionamento de máquinas virtuais ao utilizar-se uma arquitetura publicador/consumidor baseada em filas, totalmente integrada com tecnologias do ecossistema Docker. Utiliza-se uma média móvel exponencial para definir quando uma nova máquina deve ser criada, dada medição de utilização de memória. Utilizaram-se dados de um cenário real para processamento de arquivos em PDF para mostrar a eficácia da proposta. Com a aplicação da proposta utilizando a média móvel exponencial, 90% das operações que ocasionariam em máquinas criadas desnecessariamente e a redução da utilização de memória da aplicação publicadora.

Palavras-chave: Provisionamento Automático. Otimização. Cloud. Docker. média móvel exponencial.

Abstract

With the introduction of large scale cloud computing, user interactions with a software are not trivial. Upon reaching a high number of request, it is necessary to expand computing capacity. It is an industry standard to use infrastructure as a service providers (*IaaS*), which are capable of providing on-demand computing capacity, through virtual machines. However, the decision making process of provisioning an virtual machine incurs in increased operational costs, due to the fact of being a mostly manual process, leaving the user with the hard task of deciding which machine configuration to provision, and when to do it. In the current state of art, there are techniques to automate this process, despite that, none of them focuses on user experience, meaning that it's still hard for the user to make decisions. In this paper, it is proposed a tool to automate the provisioning of virtual machines applied to publisher/subscriber architectures based on message queues, integrated with tools from the Docker ecosystem. A moving exponential average is used to decide when a new machine is supposed to be provisioned, using the average memory available to the cluster. A real scenario is used to show the solution's efficacy. With the proposed solution, together with the application of the moving exponential average, it was shown a net gain of 90% when avoiding to create new machines unnecessarily.

Keywords: automatic provisioning, optimization, cloud, aws, exponential moving average

Lista de Figuras

3.1 Descrição da classe Job 19
3.2 Descrição da classe Channel 19
3.3 Descrição da classe Service 20

4.1 Sequência para geração de uma *thumbnail* na arquitetura anterior 24
4.2 Sequência para geração de uma thumbnail na arquitetura atual 24
4.3 Sequência para geração de uma thumbnail na aplicação consumidora 25
4.4 Relação entre número de requisições para ligar uma máquina consideradas versus número de requisições ignoradas 26

Sumário

1 Introdução **12**
1.1 **Problemática** . . . 12
1.2 **Objetivos da Proposta** . . . 13
1.3 **Metodologia** . . . 13
1.4 **Organização do Trabalho** . . . 13

2 Fundamentação Teórica **14**
2.1 Cloud Computing . . . 14
2.2 Infraestrutura . . . 15
2.3 Execução . . . 15

3 Proposta **18**
3.1 Aspectos Operacionais . . . 18
3.2 Provisionamento Automático de Máquinas Virtuais . . . 18
3.3 Monitor de Trabalhos (*JobMonitor*) . . . 19
3.3.1 Canal de Mensagens (*Channel*) . . . 19
3.3.2 Serviço (*Service*) . . . 20
3.3.3 Fator de Escala (*ScaleFactor*) . . . 20
3.4 Gerenciador de Trabalhos (*JobManager*) . . . 21
3.5 Monitor de Máquinas (*MachineMonitor*) . . . 21
3.5.1 Captura do estado do cluster . . . 21
3.6 Gerenciador de Máquinas (*MachineManager*) . . . 21
3.6.1 Adicionar máquinas ao cluster . . . 21
3.6.2 Remover máquinas do cluster . . . 22

4 Experimento **23**
4.1 Descrição do Experimento . . . 23
4.1.1 Aplicação Publicadora . . . 23
4.1.2 Aplicação Consumidora . . . 24
4.2 Utilizando a solução proposta no problema . . . 24
4.3 Principais Parâmetros . . . 25
4.4 Resultados . . . 25

5 Trabalhos Relacionados **27**
5.1 Parâmetros de comparação . . . 27
5.2 IronWorker[1] . . . 27

[1] https://www.iron.io/worker

5.3 AWS Batch[2] . . . 28
5.4 Azure Batch[3] . . . 28
5.5 Microscaling[4] . . . 28
5.6 Comparativo final . . . 29

6 Conclusão e Trabalhos Futuros **30**
6.1 Trabalhos Futuros . . . 30
6.1.1 Aplicação da MME para eventos de CLUSTER_DOWN . . . 30
6.1.2 Avaliação de eventos de atualização de *Jobs* . . . 30
6.1.3 Monitoramento e análise de uso de memória de aplicações consumidoras . . . 31
6.1.4 Problema de Agendamento de Trabalhos (*Job Shop Scheduling Problem*) . . . 31

Referências Bibliográficas **32**

[2]https://aws.amazon.com/batch/
[3]https://azure.microsoft.com/en-us/services/batch/
[4]https://microscaling.com/

1

Introdução

Em aplicações *web*, podemos ter requisições que não precisam ser imediatamente respondidas ao usuário. Neste cenário, é comum a implementação de uma arquitetura publicador/consumidor, geralmente utilizando, mas não restrito a, uma fila como canal de mensagens [Eugster et al. 2003]. Em linhas gerais, a implementação segue a descrição abaixo:

- Uma aplicação publicadora, que publica mensagens em um canal de mensagens
- Um canal de mensagens
- Uma aplicação consumidora, que consome mensagens do canal acima mencionado

1.1 Problemática

Para o desenvolvedor de uma aplicação *web*, ao se deparar com um tipo de requisição de usuário que pode demandar mais recursos computacionais do que o esperado, a solução comum seria delegar o trabalho para uma *thread* em segundo plano. Esta abordagem funciona, em especial com *frameworks*[1] que possibilitam programação reativa [Bainomugisha et al. 2013]. Porém existe um limite: simplesmente a requisição do usuário demanda muito mais recurso do que apenas um servidor é capaz de prover. Se nesta aplicação ocorrer diversas requisições simultâneas, muito provavelmente um só servidor não será suficiente para processá-las.

Diante deste cenário, o desenvolvedor é obrigado a implementar uma estrutura similar com a descrita na Seção 1 deste trabalho. O problema surge no momento em que faz-se necessário interagir diretamente com o provedor de infraestrutura (*Infrastructure as a Service - IaaS*), devido a este processo ser em sua grande parte manual e complexo. Além disto, a decisão de quando ligar ou desligar uma máquina não é trivial. Em [Jiang et al. 2013] é possível perceber que a decisão de aumentar ou diminuir a capacidade computacional envolve variáveis como tempo mínimo de resposta e orçamento. Já em [Netto et al. 2014], que trabalha também com tempo mínimo de resposta e orçamento, procura-se encontrar um índice para determinar em quantas máquinas deve-se aumentar ou diminuir a capacidade computacional do conjunto, ou seja, quantas máquinas deve-se ligar ou desligar.

[1] http://reactivex.io/

No mercado existem soluções que resolvem este problema[2,3,4], porém nenhuma das soluções existentes resolve este problema da forma proposta neste trabalho.

1.2 Objetivos da Proposta

O objetivo deste trabalho é desenvolver uma ferramenta capaz de decidir o melhor momento de ligar ou desligar uma máquina virtual ao valer-se do uso da média móvel exponencial na tomada de decisão. Através de integração com Docker, a solução proposta também consegue controlar o número de instâncias de uma aplicação consumidora a dado instante. Estes dois fatores aliados garantem uma operação automática das máquinas virtuais, obtendo um ganho em tempo por evitar a intervenção manual.

1.3 Metodologia

A fim de colher dados e mensurar se a solução proposta é eficaz em sua implementação, aproximadamente um milhão de documentos PDF foram publicados em uma fila de mensagens, e consumidos por uma aplicação consumidora. Foram obtidos o número de vezes as quais uma máquina poderia ter sido ligada e onde a proposta evitou que isso acontecesse. Demonstra-se também que ao introduzir-se uma arquitetura publicador/consumidor no cenário exposto, houve diminuição no uso de memória da aplicação consumidora.

1.4 Organização do Trabalho

Abaixo está descrita a organização do trabalho para guiar o leitor.

- **Capítulo 2**: apresenta-se a fundamentação teórica que guiou a escrita do trablho.
- **Capítulo 3**: apresenta-se a solução para o problema aqui descrito
- **Capítulo 4**: aplica-se a solução proposta no Capítulo 3 em um cenário real
- **Capítulo 5**: relata-se em mais detalhes o cenário de soluções similares à apresentada neste trabalho
- **Capítulo 6**: apresentam-se as conclusões gerais da aplicação da solução no problema real e trabalhos futuros são propostos

[2] https://www.iron.io/worker
[3] https://aws.amazon.com/batch/details/
[4] https://azure.microsoft.com/en-us/services/batch/

2

Fundamentação Teórica

Neste capítulo são abordados os principais conceitos e tecnologias utilizados para o desenvolvimento deste trabalho, com o objetivo de facilitar a compreensão e interpretação do tema proposto.

2.1 Cloud Computing

Computação em nuvem (*Cloud Computing*) é o termo que define um conjunto de técnicas que permite o acesso a recursos de computação praticamente ilimitados[1]. No mercado existem três formas principais de *Cloud Computing*: infraestrutura como serviço (*Infrastructure as a Service - IaaS*), plataforma como serviço (*Platform as a Service - PaaS*) e software como serviço (*Software as a Service - SaaS*).

IaaS fornece mecanismos para provisionar infraestruturas básicas de computação: máquinas virtuais, discos virtuais e definição de redes. O maior fornecedor de *IaaS* atualmente é a *Amazon Web Services* (*AWS*)[2].

Ferramentas de *PaaS* fornecem serviços de mais alto nível, onde o usuário não se preocupa necessariamente com a infraestrutura, somente com o código de aplicação que deve executar. Geralmente têm-se uma menor flexibilidade com estas ferramentas, pois elas são customizadas para suportar um tipo específico de aplicação. Como exemplo, temos o Heroku[3], que sugere um conjunto de 12 regras[4] para guiar o desenvolvedor ao criar uma aplicação com a intenção de executar no Heroku.

Em um nível ainda mais alto, *SaaS* fornece serviços diretamente ao consumidor final, onde o mesmo não se preocupa com a infraestrutura e as técnicas de implantação de um software. Geralmente, *SaaS* está disponível através de um plano de assinatura mensal, onde o usuário realiza ou não um pagamento mensal e tem acesso a um software assim como estivesse contratando um serviço. Como exemplo, ao contratar um plano mensal na plataforma de vídeos Netflix[5], o usuário está contratando um serviço análogo a uma assinatura de TV á cabo, porém tendo acesso através de um software.

---

[1] https://aws.amazon.com/what-is-cloud-computing/
[2] https://aws.amazon.com/
[3] https://www.heroku.com/
[4] https://12factor.net/
[5] https://netflix.com

2.2 Infraestrutura

Infra-estrutura refere-se ao computador que executa uma instância da aplicação consumidora. Para o contexto deste trabalho, trataremos computadores como sendo máquinas virtuais hospedadas em data-centers de um provedor de *IaaS*.

IaaS e máquinas virtuais

Provedores de IaaS fornecem uma gama de serviços como virtualização sob demanda, banco de dados gerenciados, filas distribuídas, entre outros. Somente a virtualização sob demanda é relevante para esse trabalho. *Provisionar* uma máquina virtual significa criar a mesma no provedor escolhido, e *de-provisionar* significa remover o registro daquela máquina virtual do provedor de IaaS. Ao provisionar uma máquina virtual, o usuário tem a liberdade de escolher quanta memória e quanto poder de processamento ele deseja. Quanto mais poderosa a máquina, maior a razão centavo/hora.

Os provedores de IaaS geralmente fornecem um painel de controle e/ou ferramentas baseadas em linha de comando para provisionar, desligar, ligar e de-provisionar máquinas virtuais. Entretanto, qualquer *software* que precise ser instalado na máquina virtual precisa ser feito manualmente pelo usuário, através de conexão remota via Secure Shell (SSH) para máquinas Linux e Remote Desktop Protocol (RDP) para máquinas Windows. Para evitar a intervenção manual, o usuário pode escolher utilizar uma ferramenta de provisionamento tal como *Chef*[6], *Puppet*[7] ou *Ansible*[8].

2.3 Execução

A execução refere-se ao *como* dada aplicação consumidora será de fato executada na máquina.

Contêineres de Aplicação

A forma mais simples e direta de explicar o que é um contêiner de aplicação é compará-lo com uma máquina virtual. Uma máquina virtual possui um ambiente isolado do hospedeiro, e essa garantia é mantida pelo hypervisor, o qual é responsável por administrar o acesso aos recursos do hospedeiro pela máquina virtual. A máquina virtual pode executar qualquer sistema operacional, independente do hypervisor e do sistema operacional do sistema hospedeiro.

Um contêiner de aplicação também possui isolamento do hospedeiro. Porém, contêiner e hospedeiro compartilham o mesmo *kernel* do sistema operacional do hospedeiro. Logo, não é possível ter um contêiner *Windows* em um hospedeiro *Linux*, já que o primeiro não compartilha o mesmo *kernel* do segundo.

Por outro lado, contêineres oferecem um ambiente de execução com quase nenhum *overhead* [Xavier et al. 2013], não existe um *hypervisor* coordenando o acesso a recursos, uma vez que o acesso é feito de forma direta, coordenado pelo sistema operacional da mesma forma que um processo seria coordenado.

[6]https://www.chef.io/chef/
[7]https://puppet.com/
[8]https://www.ansible.com/

Apesar de muito similares a máquinas virtuais, contêineres oferecem uma melhor experiência de uso uma vez que é mais rápido instanciar um contêiner do que uma máquina virtual [Kominos, Seyvet e Vandikas 2017, IV Evaluation F. Boot-up time].

Docker Engine

A *Docker Engine*[9] é a peça principal do ecossistema *Docker*. Ela é responsável por criar contêineres de aplicação e gerenciar seu ciclo de vida, além de introduzir conceitos interessantes como o de *imagem de contêiner*. Uma imagem é uma definição de um contêiner: uma especificação a qual a *Docker Engine* é capaz de interpretar e a partir dela criar novas instâncias de contêineres.

Docker Engine é composta por dois componentes: o cliente e o servidor (também chamado de *daemon*). O cliente utiliza um *socket linux* para se comunicar com o *daemon*, instruindo qual ação deve ser aplicada a qual contêiner. Também é possível de comunicar-se com a *Docker Engine* através de uma API HTTP [10].

Dockerfile

O *Dockerfile*[11] é um arquivo de texto onde estão contidas as instruções de como criar um contêiner de aplicação: qual distribuição Linux usar, quais aplicações devem estar instaladas e quais bibliotecas devem estar disponíveis e finalmente, qual executável utilizar para de fato disponibilizar a aplicação. No Trecho de Código 2.1 está um exemplo de um *Dockerfile* onde é descrita a instalação do *MongoDB*.

```
1  FROM ubuntu:latest
2
3  RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
4
5  RUN echo "deb http://repo.mongodb.org/apt/ubuntu "$(lsb_release -sc)"/
     mongodb-org/3.0 multiverse" | tee /etc/apt/sources.list.d/mongodb-org
     -3.0.list
6
7  RUN apt-get update && apt-get install -y mongodb-org
8
9  RUN mkdir -p /data/db
10
11 EXPOSE 27017
12
13 ENTRYPOINT ["/usr/bin/mongod"]
```

Trecho de Código 2.1: Dockerfile descrevendo como instalar o banco de dados MongoDB

Docker Swarm

Docker Swarm provê soluções de *clustering* nativas ao *Docker*. Essencialmente, o Swarm é capaz de abstrair um conjunto de máquinas que estejam rodando Docker Engine como somente uma.

Como explicado na Seção 2.3, temos que a *Docker Engine* é dividida em dois módulos: cliente e servidor. No contexto de uma *Swarm*, o cliente enxerga a *Swarm* inteira como se

[9] https://docs.docker.com/engine/docker-overview/
[10] https://docs.docker.com/engine/api/v1.32/
[11] https://docs.docker.com/engine/reference/builder/

fosse apenas um servidor. Dessa forma, o cliente pode interagir com o todas as máquinas pertencentes ao *cluster* como se fosse apenas uma.

Além de prover acesso transparente a um cluster, o *Swarm* provê meios de orquestrar contêineres de acordo com suas características internas e possui capacidades de reparação automática (*self-healing*). No evento de uma máquina deixar o cluster por qualquer motivo, o *Swarm* é capaz de re-orquestrar os contêineres órfãos para outros membros do cluster. Para atingir tal nível de automação, Docker Swarm implementa um algoritmo de consenso chamado *Raft*[Howard 2014], o qual implementa mecanismos para eleição de líder e replicação de informações entre membros de um cluster, além de garantir que essa comunicação entre membros do cluster seja segura.

3

Proposta

Neste capítulo está descrito em detalhes como é implementada a proposta. Estão descritos os aspectos operacionais relativos a solução, destacando a problemática e como a solução foi dividida em módulos de software.

3.1 Aspectos Operacionais

O principal problema que esta proposta visa solucionar é o processo manual de operar máquinas virtuais em dado provedor de infraestrutura como serviço (*Infrastructure as a Service - IaaS*) ao implementar-se uma arquitetura publicador/consumidor com um canal de mensagens baseado em fila. A dado instante, este canal pode estar com um número elevado de mensagens disponíveis, necessitando de mais capacidade computacional para processá-las, e em outro instante o mesmo pode estar com menos mensagens e até mesmo vazio. Dado um número x de mensagens, temos que necessitamos de um número y de instâncias de uma aplicação consumidora para processá-las. Por outro lado, quando temos diversas instâncias de uma aplicação consumidora, potencialmente necessita-se de mais recursos computacionais, ou seja, mais máquinas virtuais pertencentes ao cluster.

Entretanto, o provisionamento manual destas máquinas é complexo, e exige que o usuário conheça a fundo como operar as especificidades do *IaaS* escolhido, e além disso, é responsabilidade do usuário definir qual a configuração da máquina que deseja criar e em qual momento ela deve ser desligada e/ou ligada.

3.2 Provisionamento Automático de Máquinas Virtuais

Este trabalho visa trazer uma solução com menos ou nenhum *overhead* de configuração e operação, seguindo a *filosofia Docker* [@solomonstre 2014]. A intenção é que o usuário somente tenha que implementar a sua aplicação consumidora e não deva preocupar-se com os aspectos operacionais citados na Seção 3.1. Os parâmetros abaixo foram definidos para garantir que a solução proposta resolva os problemas levantados.

- Código aberto
- Não acoplado a nenhum canal de mensagens específico
- Não acoplado a nenhum *IaaS* em específico

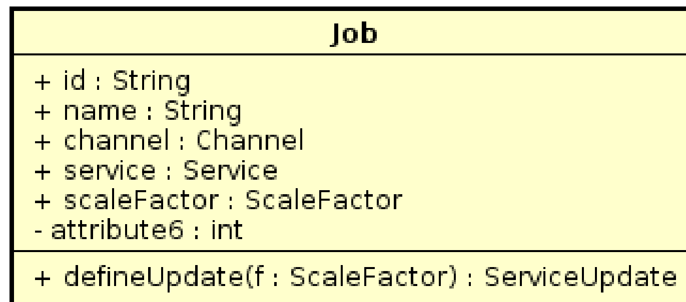

Figura 3.1: Descrição da classe Job

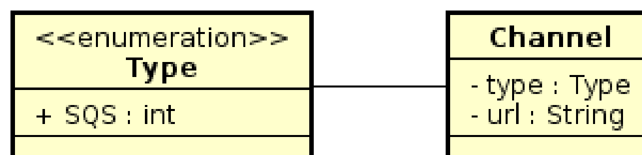

Figura 3.2: Descrição da classe Channel

- Distribuído em uma ou mais imagens Docker, com instruções simples de implantação
- Interface web para controle e monitoramento das atividades em execução

Para atingir tais objetivos, a ferramenta então é dividida em quatro módulos: Monitor de Trabalhos (*JobMonitor*), descrito na Seção 3.3; Gerenciador de Trabalhos (*JobManager*), descrito na Seção 3.4; Monitor de Máquinas, descrito na Seção 3.5 e por fim o Gerenciador de Máquinas (*MachineManager*), descrito na Seção 3.6

3.3 Monitor de Trabalhos (*JobMonitor*)

Um Trabalho (*Job*) une a aplicação consumidora com o canal o qual a mesma consome, e o fator que determina o número ideal de instâncias desta aplicação para um dado número de mensagens.

Através do atributo *scaleFactor* (descrito na Seção 3.3.3), o *JobManager* é capaz de decidir quando criar ou destruir uma instância da aplicação consumidora, esta descrita no atributo *service* (Seção 3.3.2). Por fim, o atributo *channel* (Seção 3.3.1) define qual implementação de canal de mensagens será monitorada.

O *JobMonitor* tem acesso a uma base de dados onde estão armazenados todos os *Jobs* que devem ser monitorados.

3.3.1 Canal de Mensagens (*Channel*)

A classe *Channel* descreve qual implementação da interface *ChannelMonitor* devemos utilizar através de seu atributo *type*, como descrito na Figura 3.3.1. O padrão de projeto *Factory Method* foi aplicado neste cenário, pois o mesmo possibilita um menor acoplamento do código de domínio do código necessário para interagir com um dado canal [Schmidt 1995].

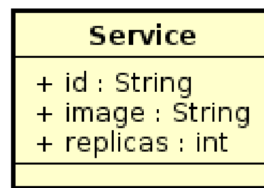

Figura 3.3: Descrição da classe Service

3.3.2 Serviço (*Service*)

Um *Service* é a descrição de como executar uma instância da aplicação consumidora em si. A Figura 3.3.2 ilustra como está modelada a classe que o representa. O atributo *image* indica qual imagem *Docker* será utilizada para criar o contêiner, e o atributo *replicas* indica quantos contêineres da aplicação consumidora estão em execução neste instante.

3.3.3 Fator de Escala (*ScaleFactor*)

O *ScaleFactor* define a relação entre número de mensagens disponíveis na fila e número de contêineres disponíveis para processá-las. Este fator é calculado da seguinte forma:

$$S_f = M_q/C \tag{3.1}$$

Onde M_q é o número de mensagens na fila, e C é o número de contêineres necessário para processar dito número de mensagens.

Para funcionar da maneira intencionada, o *JobMonitor* calcula o *fator de escala real*. O cálculo segue a mesma fórmula descrita acima, com a ressalva de que neste contexto a aplicação utiliza dados reais. Em outras palavras, o *JobMonitor* utiliza o número de contêineres em execução e o número de mensagens disponíveis no momento.

Ao comparar os dois fatores, o *JobMonitor* pode decidir qual tipo de ação tomar:

- fator real < fator desejado: notifica o *JobManager* para criar contêineres, indicando também a quantidade de contêineres a serem criados (evento de **scale up**)
- fator real > fator desejado: notifica o *JobManager* para destruir contêineres, indicando também a quantidade de contêineres a serem destruíds (evento de **scale down**)

Definição do Número C

Em primeiro momento, este número será fornecido pelo usuário no momento do registro do seu *Job*. Uma vez definido este valor, a solução pode definir um novo C, atualizando o mesmo de tal maneira que:

1. Reduza o tempo médio de permanência da mensagem na fila
2. Mantenha o custo com máquinas dentro do limite adequado

Vale ressaltar que esta estratégia depende do provedor do canal de mensagens, e exige configuração específica para cada cenário.

3.4 Gerenciador de Trabalhos (*JobManager*)

Este módulo expõe uma simples API RESTful [Meng, Mei e Yan 2009] para gerenciar *Jobs*, permitindo criar, atualizar, remover ou listá-los. Esta API foi criada para possibilitar com que, no futuro, seja criada uma interface Web que se comunique com a mesma no intuito de administrar *Jobs*.

Além de expor dita API, este módulo é responsável por comunicar-se com a *Docker Swarm* e de fato atualizar o número de contêineres de um *Service* associado a um *Job*.

3.5 Monitor de Máquinas (*MachineMonitor*)

Como o *JobManager* constantemente muda a escala dos contêineres em execução, podemos ter o cenário onde a máquina não possui nenhum contêiner ativo, ou, de forma análoga, o cluster pode estar sobrecarregado. Desta forma, podemos redimensionar o cluster para a carga de trabalho disponível nesse momento.

Este módulo da solução é responsável por monitorar o estado do cluster, comunicando-se com o *MachineManager* quando for necessário criar mais uma máquina para alocar contêineres, ou desligar uma máquina ociosa.

3.5.1 Captura do estado do cluster

Para definir quando aumentar ou diminuir o tamanho do cluster, o *MachineMonitor* monitora o percentual de memória livre no cluster. Para isso, temos um componente chamado *monitor de memória*, sendo executado em cada máquina presente no cluster. Abaixo está descrita a sequência de passos necessárias:

- A cada 1s, o *MachineMonitor* dará início a coleta do estado de memória do cluster
- É mandada uma mensagem para cada máquina do cluster, de maneira paralela
- Cada máquina responde com seu percentual de memória livre
- O *MachineMonitor* então calcula uma média simples destes valores, obtendo o percentual de memória livre do cluster

Com estes dados em mãos segue de forma direta como será esquematizado o processo de aumentar ou diminuir o cluster:

- memória disponível $<$ 10% do total: adicionar nova máquina ao cluster, emite evento de CLUSTER_UP para o *MachineManager*.
- memória disponível $>=$ 10% do total: remover máquina do cluster que possui mais memória livre, emite evento de CLUSTER_DOWN para o *MachineManager*.

3.6 Gerenciador de Máquinas (*MachineManager*)

3.6.1 Adicionar máquinas ao cluster

Para adicionar uma nova máquina ao cluster, primeiro é feita uma análise baseada no cálculo da *média móvel exponencial* (MME) [Hunter et al. 1986] do intervalo de tempo

entre dois eventos de CLUSTER_UP e CLUSTER_DOWN. Esta análise tem o propósito de evitar que uma nova máquina seja ativada desnecessariamente.

Se denominarmos tal intervalo como ΔT, um evento de CLUSTER_UP só é consumado se o ΔT_n é menor que ou igual à MME atual. Em outras palavras, o *MachineManager* utiliza esta comparação para inferir que provavelmente em breve um evento de CLUSTER_DOWN será emitido, o que anula o evento de CLUSTER_UP.

O algoritmo escrito em Java para tomar essa decisão encontra-se no Trecho de Código 3.1.

```
1     if (Duration.between(lastScaleDown, Instant.now()).getSeconds() <=
          movingAvg) {
2         lastScaleUp = Instant.now();
3         logger.info("CLUSTER_UP request ignored");
4     } else {
5         lastScaleUp = Instant.now();
6         scaleUp(swarm)
7         logger.info("SCALE_UP");
8     }
```

Trecho de Código 3.1: Algoritmo para adicionar ou não máquinas ao cluster

3.6.2 Remover máquinas do cluster

O ato de remover uma máquina é simplesmente solicitar ao provedor de infra-estrutura desligar a máquina. O cluster perceberá que um dos seus nós foi desativado e efetuará automaticamente o rebalanceamento dos contêineres que por ventura estivessem em execução neste nó.

4

Experimento

Neste capítulo está descrito o experimento executado, destacando o problema de negócio resolvido pela proposta. É mostrado também os resultados obtidos com a aplicação da solução.

4.1 Descrição do Experimento

Como aplicação publicadora, temos o Clip[1], que é um Gestor Eletrônico de Documentos, descrito em mais detalhes na Seção 4.1.1. O canal de mensagens escolhido é o *Simple Queue Service*[2] (SQS), serviço fornecido pela *Amazon Web Services* (AWS), e por fim como consumidora uma aplicação escrita em Java, descrita na Seção 4.1.2.

4.1.1 Aplicação Publicadora

O Clip é um sistema Gerenciador Eletrônico de Documentos. Sua principal função é armazenar documentos e permitir uma pesquisa rápida e eficiente destes documentos. Para cada documento, é gerada uma imagem de pré-visualização da primeira página, também conhecida como *thumbnail*.

Na primeira versão da solução deste problema, a própria aplicação era responsável por gerar a *thumbnail* no momento do envio do documento pelo usuário, o que significa que era necessário fazer o *download* do arquivo pdf por inteiro, armazená-lo em memória, gerar a imagem *thumbnail* e fazer upload da mesma para o servidor de arquivos. O Diagrama de Sequência que descreve tal interação pode ser visto na Figura 4.1.1.

No entanto, a partir do momento em que o número de requisições aumentou, juntamente com o tamanho dos arquivos, a memória disponível no servidor passou a não ser mais suficiente. Além disso, muitas vezes o código, por falta de memória disponível, falhava ao tentar gerar a thumbnail, o que acarretou em diversos documentos não possuindo uma *thumbnail* correspondente.

Desta forma, uma arquitetura como a exposta na Seção 1 foi implementada. A Figura 4.1.1 ilustra como se comporta a sequência de chamadas para a arquitetura atual. Vale ressaltar que esta requisição acontece de forma assíncrona: a aplicação publicadora não espera uma resposta do código que gera as *thumbnails*, mesmo porque não há mecanismo implementado para tal interação.

---

[1] http://www.goclip.com.br/

[2] https://aws.amazon.com/sqs/

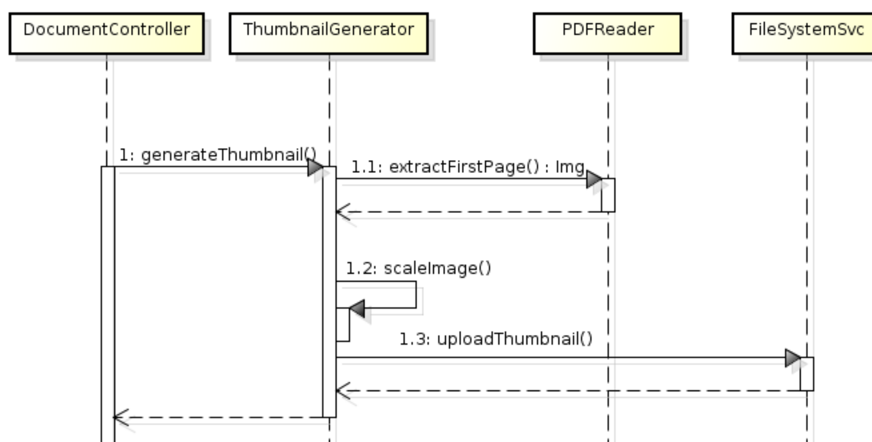

Figura 4.1: Sequência para geração de uma *thumbnail* na arquitetura anterior

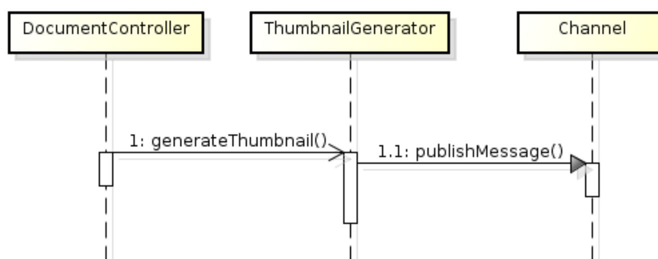

Figura 4.2: Sequência para geração de uma thumbnail na arquitetura atual

4.1.2 Aplicação Consumidora

A aplicação consumidora executa em um *loop* infinito, checando a cada cinco segundos se existem novas mensagens na fila para serem processadas. Caso haja, o mesmo procedimento ilustrado na Figura 4.1.1, referente somente à geração da *thumbnail* é executado. A Figura 4.1.2 ilustra a sequência resultante após o recebimento de uma mensagem.

4.2 Utilizando a solução proposta no problema

Em primeiro lugar, a aplicação consumidora deve estar contida em uma imagem *Docker*. Para isso, foi definido o *Dockerfile* no Trecho de Código 4.1. Neste Trecho de Código, estamos instruindo a *Docker Engine* para utilizar como base a distribuição Linux *Alpine*[3]. Logo em seguida, através da instrução *ADD*, o *jar* da aplicação é inserido dentro da imagem. Por fim, o comando *CMD* indica que ao criar um contêiner a partir desta imagem, este comando seja executado. Temos então a descrição completa tanto do ambiente onde a aplicação consumidora vai executar, quanto como ela vai ser executada.

```
1 FROM anapsix/alpine-java:8_server-jre
2
3 ADD target/thumbnailer.jar .
4
5 CMD ["java", "-jar", "thumbnailer.jar"]
```

Trecho de Código 4.1: Dockerfile descrevendo como criar uma imagem para a aplicação consumidora

[3]https://alpinelinux.org/

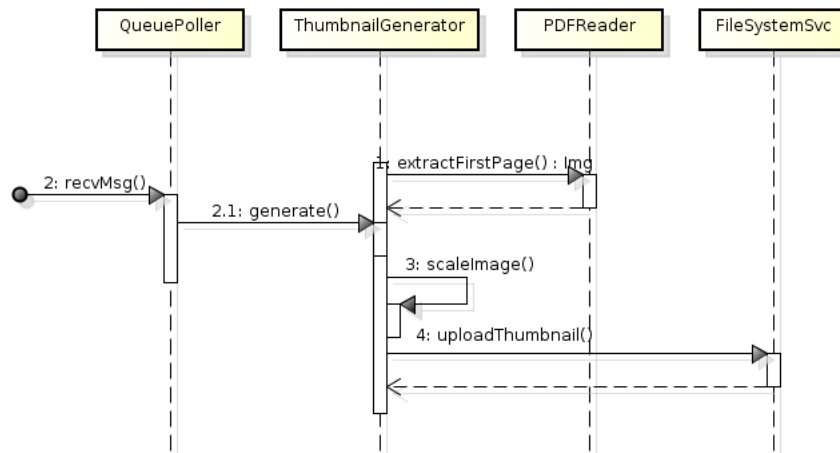

Figura 4.3: Sequência para geração de uma thumbnail na aplicação consumidora

Após criada, esta imagem é então publicada para o *Docker Hub*, e seu nome utilizado para criar o *Job*.

4.3 Principais Parâmetros

Para que o efeito mencionado na Seção 3.6.1 possa ser visualizado, foram definidos parâmetros para melhor evidenciar a eficácia da solução proposta. Como parâmetro da média móvel exponencial, definiu-se $\alpha = 0.8$. Esta decisão decorre da intenção de desconsiderar medições antigas mais rapidamente, e focar nas medições mais recentes, propriedade inerente da média móvel exponencial. explicar sobre escolha Na tabela 4.3 estão descritos os demais parâmetros, visando simular um cenário real onde um fluxo contínuo de mensagens é entregue ao canal.

Variável	Valor
Número de Documentos	942230 documentos
Taxa de Envio	1000 documentos por segundo

Tabela 4.1: Descrição dos parâmetros da avaliação

4.4 Resultados

Ao aplicar a solução com os parâmetros mostrados na Seção 4.3, foi possível extrair dados para construir o gráfico mostrado na Figura 4.4. Podemos observar que 90% do número de requisições para ligar uma máquina foram ignorados, reduzindo potenciais custos com virtualização. Além do exposto, ao refatorar a arquitetura da aplicação publicadora, foi possível reduzir o uso de memória drasticamente, mantendo o uso constante durante a operação normal da aplicação.

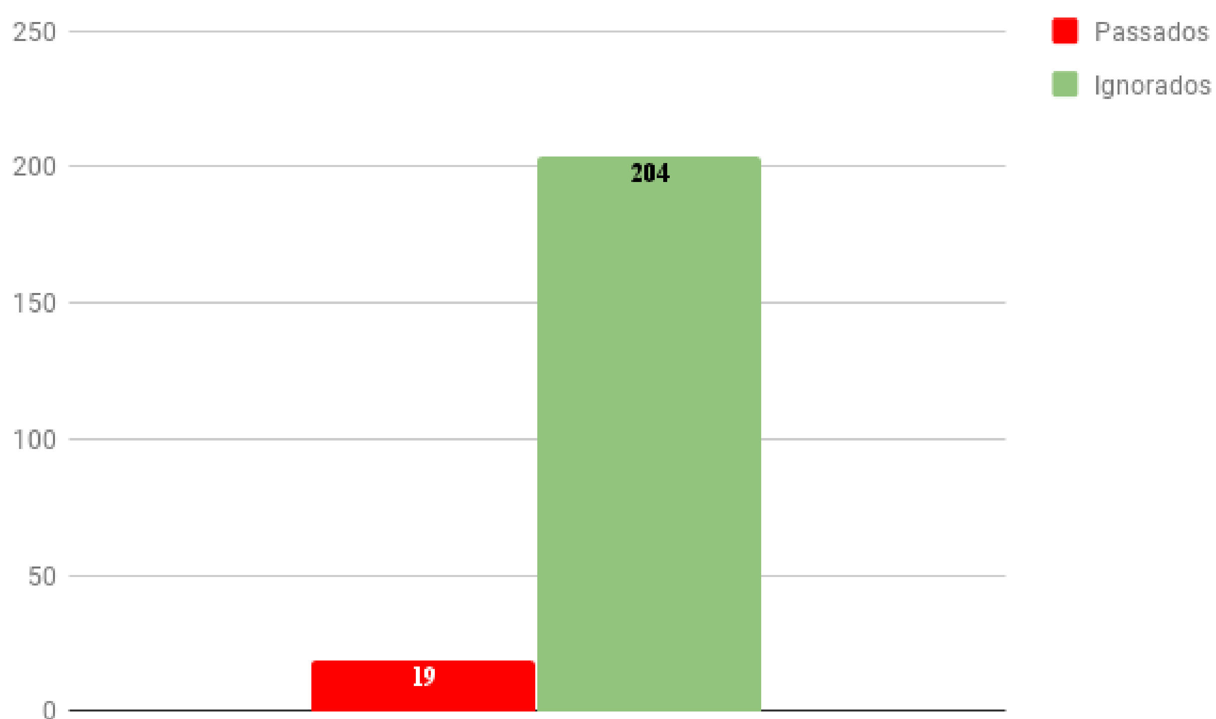

Figura 4.4: Relação entre número de requisições para ligar uma máquina consideradas versus número de requisições ignoradas

5

Trabalhos Relacionados

Neste capítulo são definidos os parâmetros para comparar a solução proposta no Capítulo 3 com os principais competidores do mercado.

5.1 Parâmetros de comparação

As características que definem e diferenciam a solução aqui proposta são essas:

- Código aberto
- Não acoplado a nenhum canal de mensagens específico
- Não acoplado a nenhum *IaaS* em específico
- Distribuído em uma ou mais imagens Docker, com instruções simples de implantação
- Interface web para controle e monitoramento das atividades em execução

5.2 IronWorker[1]

- Código aberto: não
- Não acoplado a nenhum canal de mensagens específico: não, utiliza canal de mensagens proprietário
- Não acoplado a nenhum *IaaS* em específico: não, utiliza datacenter não informado
- Distribuído em uma ou mais imagens Docker, com instruções simples de implantação: por ser uma ferramente web Software como Serviço (*Software as a Service*, ou *SaaS*[2]), a mesma não é distribuída para implantação em infra-estrutura própria
- Interface web para controle e monitoramento das atividades em execução: sim

[1] https://www.iron.io/worker

[2] https://en.wikipedia.org/wiki/Software$_{a}s_{a}s ervice$

5.3 AWS Batch[3]

- Código aberto: não
- Não acoplado a nenhum canal de mensagens específico: não, utiliza canal de mensagens proprietário *Simple Queue Service*
- Não acoplado a nenhum *IaaS* em específico: não, utiliza datacenter da *AWS*
- Distribuído em uma ou mais imagens Docker, com instruções simples de implantação: por ser uma ferramente web Software como Serviço (*Software as a Service*, ou *SaaS*[4]), a mesma não é distribuída para implantação em infra-estrutura própria
- Interface web para controle e monitoramento das atividades em execução: sim

5.4 Azure Batch[5]

- Código aberto: não
- Não acoplado a nenhum canal de mensagens específico: não, utiliza canal de mensagens proprietário Azure Storage Queue
- Não acoplado a nenhum *IaaS* em específico: não, utiliza datacenter da *Azure*
- Distribuído em uma ou mais imagens Docker, com instruções simples de implantação: por ser uma ferramente web Software como Serviço (*Software as a Service*, ou *SaaS*[6]), a mesma não é distribuída para implantação em infra-estrutura própria
- Interface web para controle e monitoramento das atividades em execução: sim

5.5 Microscaling[7]

- Código aberto: sim
- Não acoplado a nenhum canal de mensagens específico: sim, possui mecanismo flexível para monitorar diversos canais de mensagem
- Não acoplado a nenhum *IaaS* em específico: não se aplica, pois *Microscaling* não provisiona, liga ou desliga novas máquinas
- Distribuído em uma ou mais imagens Docker, com instruções simples de implantação: sim, documentação disponível na página do repositório
- Interface web para controle e monitoramento das atividades em execução: sim

[3]https://aws.amazon.com/batch/
[4]https://en.wikipedia.org/wiki/Software$_as_aservice$
[5]https://azure.microsoft.com/en-us/services/batch/
[6]https://en.wikipedia.org/wiki/Software$_as_aservice$
[7]https://microscaling.com/

5.6 Comparativo final

Na Tabela 5.1 podemos observar uma comparação ponto a ponto da solução proposta com os demais competidores.

Competidores	Código aberto	Não acoplado a nenhum canal de mensagens específico	Distribuído em uma ou mais imagens Docker
IronWorker			
AWS Batch			
Azure Batch			
Microscaling	X	X	X

Tabela 5.1: Comparativo entre a solução e os trabalhos relacionados

6

Conclusão e Trabalhos Futuros

A solução implementada neste trabalho possibilita para o usuário uma maneira rápida de implementar paradigmas Publicadores/Consumidores. Como descrito nos capítulos anteriores, a operação da infra-estrutura necessária para Aplicações Consumidoras existirem é totalmente transparente e automática. Isto foi possível devido a uma extensa pesquisa e avaliação das *API's* da AWS e da *API* da Docker Engine.

Entretanto na situação atual que encontra-se a solução proposta, a relevância da mesma é restrita, pois provou-se somente eficaz, até agora, em arquiteturas publicador/-consumido com canais baseados em fila. Um estudo maior deve ser conduzido para definir a aplicação e eficácia da solução proposta em outros tipos de cenários, tais como aplicações em tempo real [Deng e Liu 1997].

6.1 Trabalhos Futuros

6.1.1 Aplicação da MME para eventos de CLUSTER_DOWN

Como mostrado na Seção 4.4, é possível enxergar que a aplicação da MME reduziu o número de vezes que máquinas são ligadas desnecessariamente. Porém, faz-se necessário avaliar antes de se desligar uma máquina, evitando que a mesma seja desligada e logo em seguida ligada novamente.

6.1.2 Avaliação de eventos de atualização de *Jobs*

A Seção 3.3 mostra que através do monitoramento de um *Channel*, podemos aumentar ou diminuir o número de instâncias de uma Aplicação Consumidora. Entretanto, no canal de mensagens escolhido para o estudo de caso, o SQS, existe um detalhe de implementação: o número de mensagens medido a dado instante é aproximado. Por exemplo, uma medição da fila em T_n pode resultar 400, e uma nova medição em T_{n+1} pode resultar 0. Dado este número, o *JobMonitor* vai agir como especificado e reduzir o número de instâncias para 0, para possivelmente em uma nova medição aumentar o número de instâncias para outro valor. A utilização de técnicas de suavização de dados pode reduzir o impacto de tais medições imprecisas [Velleman 1980].

6.1.3 Monitoramento e análise de uso de memória de aplicações consumidoras

Para que instâncias de Aplicações Consumidoras sejam executadas com eficiência, é necessário que no cluster hajam máquinas com memória disponível. A partir desta informação, a solução pode calcular a quantidade de memória ideal da máquina a ser provisionada, no objetivo de minimizar operações de atualização do cluster.

6.1.4 Problema de Agendamento de Trabalhos (*Job Shop Scheduling Problem*)

Job Shop Scheduling Problem (JSP) é um problema conhecido no estado da arte, com diversas soluções tanto matemáticas quanto computacionais, utilizando aprendizagem computacional [Błażewicz, Domschke e Pesch 1996]. Existem semelhanças entre o problema exposto neste trabalho e o JSP se considerarmos que uma máquina virtual como uma *Machine*, e um *Job* como um trabalho a ser agendado. Em posse disso, soluções para o JSP potencialmente podem ser consideradas para o futuro deste trabalho, de forma a otimizar a alocação de uma instância da Aplicação Consumidora em uma máquina adequada.

Referências Bibliográficas

BAINOMUGISHA, E. et al. A survey on reactive programming. *ACM Computing Surveys (CSUR)*, ACM, v. 45, n. 4, p. 52, 2013.

BŁAŻEWICZ, J.; DOMSCHKE, W.; PESCH, E. The job shop scheduling problem: Conventional and new solution techniques. *European journal of operational research*, Elsevier, v. 93, n. 1, p. 1–33, 1996.

DENG, Z.; LIU, J.-S. Scheduling real-time applications in an open environment. In: IEEE. *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*. [S.l.], 1997. p. 308–319.

EUGSTER, P. T. et al. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, ACM, v. 35, n. 2, p. 114–131, 2003.

HOWARD, H. *ARC: Analysis of Raft Consensus*. [S.l.], 2014. Disponível em: <http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-857.pdf>.

HUNTER, J. S. et al. The exponentially weighted moving average. *J. Quality Technol.*, v. 18, n. 4, p. 203–210, 1986.

JIANG, J. et al. Optimal cloud resource auto-scaling for web applications. In: IEEE. *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. [S.l.], 2013. p. 58–65.

KOMINOS, C. G.; SEYVET, N.; VANDIKAS, K. Bare-metal, virtual machines and containers in openstack. In: IEEE. *Innovations in Clouds, Internet and Networks (ICIN), 2017 20th Conference on*. [S.l.], 2017. p. 36–43.

MENG, J.; MEI, S.; YAN, Z. Restful web services: A solution for distributed data integration. In: IEEE. *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*. [S.l.], 2009. p. 1–4.

NETTO, M. A. et al. Evaluating auto-scaling strategies for cloud computing environments. In: IEEE. *Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on*. [S.l.], 2014. p. 187–196.

SCHMIDT, D. C. Using design patterns to develop reusable object-oriented communication software. *Communications of the ACM*, ACM, v. 38, n. 10, p. 65–74, 1995.

@SOLOMONSTRE. *Filosofia Docker*. 2014. Disponível em: <https://twitter.com/solomonstre/status/530537767122784256>.

VELLEMAN, P. F. Definition and comparison of robust nonlinear data smoothing algorithms. *Journal of the American Statistical Association*, Taylor & Francis Group, v. 75, n. 371, p. 609–615, 1980.

XAVIER, M. G. et al. Performance evaluation of container-based virtualization for high performance computing environments. In: IEEE. *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*. [S.l.], 2013. p. 233–240.