

**Universidade Paulista - UNIP**

**Bruno Zutim**

**AUTOMATIZANDO O PROCESSO DE ESCALONAMENTO VISANDO GANHAR  
ELASTICIDADE REATIVA NO ORQUESTRADOR DOCKER SWARM COM UM  
SHELL SCRIPT**

**Limeira  
2018**

**Universidade Paulista - UNIP**

**Bruno Zutim**

**AUTOMATIZANDO O PROCESSO DE ESCALONAMENTO VISANDO GANHAR  
ELASTICIDADE REATIVA NO ORQUESTADOR DOCKER SWARM COM UM  
SHELL SCRIPT**

Trabalho de conclusão de curso apresentado à banca examinadora da Faculdade UNIP, como requisito parcial à obtenção do Bacharelado em ciência da computação sob a orientação dos professores Me. Antonio Mateus Locci e Me. Marcos Vinícius Gialdi.

**Limeira  
2018**

**Bruno Zutim**

**AUTOMATIZANDO O PROCESSO DE ESCALONAMENTO VISANDO GANHAR  
ELASTICIDADE REATIVA NO ORQUESTRADOR DOCKER SWARM COM UM  
SHELL SCRIPT**

Trabalho de conclusão de curso apresentado à banca examinadora da Faculdade UNIP, como requisito parcial à obtenção do Bacharelado em ciência da Computação sob a orientação dos professores Me. Antonio Mateus Locci e Me. Marcos Vinícius Gialdi.

Aprovada em 14 de Novembro de 2018.

**BANCA EXAMINADORA**

---

---

---

## RESUMO

**Introdução:** O orquestrador Docker Swarm, o gerenciador de *cluster* nativo do Docker, provê escalabilidade as aplicações através da replicação e balanceamento de carga deles nos nós daquele *cluster*. **Justificativa:** Porém, esse processo não é automático, necessitando que algum usuário execute um comando que aumentará ou diminuirá o número de réplicas do serviço. **Objetivo:** Automatizar esse processo através de um *shell script*, monitorando, periodicamente, os serviços definidos pelo usuário, e utilizando do conceito de elasticidade em computação, aumentar ou diminuir o número de réplicas automaticamente. O *script* monitora o uso de processamento e memória de todas as réplicas do serviço, tira a média aritmética e caso o uso médio fique acima de um limite definido ele criará uma nova réplica, caso fique abaixo, matará uma réplica, caso fique entre os limites, manterá o número atual. **Metodologia:** Através de pesquisa bibliográfica e testes laboratoriais, estudar sobre o orquestrador *docker swarm*, elasticidade em computação e *bash script* para criar uma ferramenta que dará ao primeiro o segundo utilizando o terceiro, além de simular seu funcionamento em um *cluster* composto de três Raspberry Pi 3. **Resultados:** Monitorando um container contendo o servidor HTTP Apache provendo uma página web e executando o ApacheBench, uma ferramenta para performar testes fazendo várias requisições http a um endereço web, simulando um pico de acesso. Ao iniciar o teste, nota-se que o uso do processador sobe imediatamente, alguns segundos depois, o *script* entra em ação e cria uma nova réplica, dividindo a carga de acesso entre mais containers, quando o teste acaba as réplicas são mortas uma por uma, voltando ao estado inicial. **Conclusão:** Ao final conclui-se que utilizando da alta capacidade de escalonamento nativa do Docker Swarm é possível dar a ele elasticidade no uso dos recursos do *cluster* automatizando esse processo baseado na carga de trabalho das réplicas do serviço. Porém, algumas questões de segurança devem ser levadas em consideração.

**Palavras Chave:** Auto escalonamento, Cluster, Gerenciamento de Recursos, Sistemas distribuídos, Trabalho de conclusão de curso.

## ABSTRACT

**Introduction:** The Orchestrator Docker Swarm, Docker's native cluster manager, provides applications scalability by replicating and load-balancing them on the nodes of that cluster. **Justification:** However, this process is not automatic, requiring some user to execute a command that will increase or decrease the number of replicas of the service. **Objective:** To automate this process through a shell script, periodically monitoring user defined services, and using the concept of elasticity in computing to increase or decrease the number of replicas automatically. The script monitors the use of processing and memory of all replicas of the service, takes the arithmetic mean of them and if the average use stays above a defined limit it will create a new replica, if it is below, it will kill a replica, if it is among them, keep the current number. **Methodology:** Through bibliographic research and laboratory tests, study on the orchestrator docker swarm, elasticity in computing and bash script to create a tool that will give the first one the second using the third, in addition to simulating its operation in a cluster composed of three Raspberry Pi 3. **Results:** Monitoring a container containing the Apache HTTP server providing a web page and running ApacheBench, a tool for performing tests by making multiple HTTP requests to a web address, simulating a peak access. When starting the test, it is noticed that the use of the processor goes up immediately, a few seconds later, the script goes into action and creates a new replica, dividing the access load between more containers, when the test ends the replicas are killed one by one, returning to the initial state. **Conclusion:** It is concluded that using Docker Swarm's native scalability capability, it is possible to give it elasticity in the use of cluster resources, automating this process based on the workload of the replicas of the service. However, some safety issues must be taken into account.

**Key Words:** Auto-scaling, Cluster, Resource Management, Distributed systems, Thesis.

## LISTA DE FIGURAS

Figura 01 - Comparação entre Máquina Virtual e Contêiner .....	12
Figura 02 - Esquema do nó balanceador dentro de uma rede .....	14
Figura 03 - Diferenças de modelos: (a) sem elasticidade; (b) com elasticidade ..	15
Figura 04 - Esquema exemplo de um <i>cluster</i> Docker Swarm .....	18
Figura 05 - Funcionamento do protocolo SSH .....	20
Figura 06 - Raspberry Pi 3 Model B .....	22
Figura 07 - Trecho do código que filtra os IPs e faz a conexão SSH .....	24
Figura 08 - Esquema de funcionamento do <i>script</i> .....	25
Figura 09 - Gráfico de exemplo para tomada de decisão .....	25
Figura 10 - Foto do <i>cluster</i> com três RPi e o switch de rede .....	27
Figura 11.1 - Swarm Visualizer em execução, com duas réplicas .....	31
Figura 11.2 - “ <i>docker stats</i> ” em execução nos nós <i>Worker1</i> e <i>Worker2</i> , com duas réplicas .....	32
Figura 12.1 - Swarm Visualizer em execução, com três réplicas .....	32
Figura 12.2 - “ <i>docker stats</i> ” em execução nos nós <i>worker1</i> e <i>worker2</i> , com três réplicas .....	33

## LISTA DE ABREVIATURAS

CPU	Central Processing Unit
HPA	Horizontal Pod Autoscaler
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
PaaS	Plataform as a Service
PID	Process Identifier
RAM	Random Access Memory
RPi	Raspberry Pi
RPi3	Raspberry Pi 3
SCP	Secure Copy
SSH	Secure Shell
VM	Virtual Machine
YAML	YAML Ain't Markup Language

## SUMÁRIO

### CAPÍTULO I

<b>1.1 INTRODUÇÃO</b>	<b>10</b>
1.1.1 Objetivo .....	10
1.1.2 Justificativa .....	11
1.1.3 Metodologia .....	11

### CAPÍTULO II

<b>2.1 CONTAINER DE SOFTWARE</b>	<b>12</b>
2.1.1 Diferença entre Máquina Virtual e Contêiner .....	12
2.1.2 Orquestrador de Container .....	13
<b>2.2 CLUSTER</b>	<b>13</b>
2.2.1 balanceamento de carga .....	14
<b>2.3 COMPUTAÇÃO ELÁSTICA</b>	<b>15</b>
2.3.1 Comparação entre escalabilidade e elasticidade .....	16
<b>2.4 DOCKER</b>	<b>16</b>
2.4.1 Dockerfile .....	17
2.4.2 Orquestrador Docker Swarm .....	17
2.4.2.1 Compose file .....	18
2.4.2.1.1 YAML .....	19
<b>2.5 SHELL SCRIPT</b>	<b>19</b>
2.5.1 Shell .....	19
2.5.2 Bash .....	20
<b>2.6 SECURE SHELL</b>	<b>20</b>
2.6.1 SSH-KeyGen .....	21
2.6.1.1 Autenticação por chave pública e privada .....	21
<b>2.7 SERVIDOR HTTP APACHE</b>	<b>21</b>
2.7.1 ApacheBench .....	22
<b>2.8 RASPBERRY PI</b>	<b>22</b>

### CAPÍTULO III

<b>3.1 DESENVOLVIMENTO</b>	<b>23</b>
3.1.1 Configuração do cluster .....	27
3.1.2 Testes .....	31
<b>CONCLUSÃO E TRABALHOS FUTUROS</b>	<b>34</b>
<b>REFERÊNCIAS BIBLIOGRÁFICAS</b>	<b>35</b>

<b>APÊNDICES</b>	<b>37</b>
<b>A</b> Dockerfile do container “DKScale” .....	37
<b>B</b> Compose file do serviço Apache .....	38
<b>C</b> Código fonte do script “dkrun.sh” .....	39
<b>D</b> Código fonte do script “dkscale” .....	40
<b>E</b> Código fonte do script “dkset.sh” .....	41

## CAPÍTULO I

Nesse capítulo consta a introdução ao tema, a justificativa, objetivo e a metodologia usada durante a elaboração do trabalho.

### 1.1 INTRODUÇÃO

A Tecnologia da informação vem evoluindo de forma rápida e criando ferramentas incríveis para as necessidades do mundo digital em que vivemos, com o aumento do acesso à internet surgiu a necessidade de novas ferramentas para gerenciar toda a infraestrutura necessária para suprir essa demanda, uma dessas ferramentas foram os containers de software e os orquestradores, um software que centraliza a gerência dos serviços e recursos computacionais, dando ao administrador de sistema um nível de abstração muito maior sobre o controle da infraestrutura.

Nesse trabalho mostrarei os conceitos básicos de *containers* de software e do orquestrador *Docker Swarm*, assim como propor um *shell script* que controlará o processo de escalonamento conforme a utilização dos recursos computacionais a ele reservados aplicando o conceito de elasticidade em computação.

#### 1.1.1 Objetivo

Apesar de ser simples escalonar um serviço com o *Docker Swarm* ele não possui uma ferramenta de auto escalonamento, necessitando que uma pessoa de o comando manualmente. O objetivo é criar um *shell script* utilizando o *Bash* que monitorará, periodicamente, os serviços definidos pelo usuário que são necessários terem elasticidade, aumentando ou diminuindo o número de réplicas. O *script* monitorará o uso da memória e processamento de todas as réplicas do serviço, tira a média aritmética delas e caso o uso médio da memória ou processamento entre todas as réplicas for maior ou igual a 90% da reservada para aquele serviço, executará um comando que adicionará mais uma réplica ao *cluster*, se o uso for menor ou igual a 10% de memória e processamento ele destruirá uma das réplicas, qualquer valor entre esses limites será mantido o número atual de réplicas, esses, porém, são os valores padrão do *script*, para utilizar outros valores o usuário poderá passar seus próprios parâmetros na chamada do *script* a fim de usar suas próprias métricas, assim como poderá definir máximo de réplicas que aquele serviço pode ter.

### 1.1.2 Justificativa

A auto replicação dos micro serviços pode ser muito útil em um ambiente clusterizado servindo vários clientes diferentes, pois permite a rápida locação e desalocação dos recursos computacionais daquele *cluster* baseado na carga de trabalho dos containers. Além disso, o *Kubernetes*, outro orquestrador, mais antigo e mais utilizado em ambiente produção, possui nativamente o HPA (*Horizontal Pod Autoscaler* - Auto escalonador horizontal de Pod, em tradução livre) que monitora o uso da CPU dos containers e ajusta a quantidade de réplicas conforme essa métrica.

### 1.1.3 Metodologia

- **Etapa 1:** Estudo sobre *cluster* de balanceamento de carga e escalonamento horizontal com *Docker Swarm*.
- **Etapa 2:** Estudo sobre *bash script* para definir a melhor abordagem a ser utilizada no desenvolvimento do *script*.
- **Etapa 3:** Desenvolvimento do *script* de auto escalonamento.
- **Etapa 4:** Configuração de um ambiente de teste com três *Raspberry PI 3* para demonstração.
- **Etapa 5:** Testes do *script* funcionando no *cluster*.
- **Etapa 6:** Discussão dos resultados do teste.

## CAPÍTULO II

Esse capítulo contém um pequeno descritivo das ferramentas e conceitos utilizados nesse trabalho.

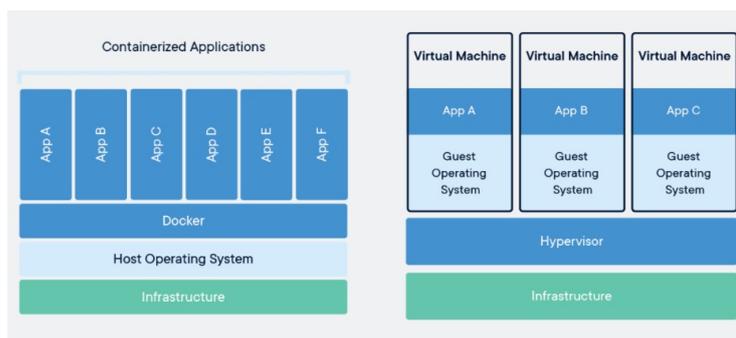
### 2.1 CONTAINER DE SOFTWARE

Um container de software é uma aplicação empacotada de forma padronizada, portátil e com todas as suas dependências, de modo a garantir que a aplicação rode igualmente independente do ambiente. Ele veio para resolver problemas entre os processos de desenvolvimento e produção onde os softwares de desenvolvimento ou as bibliotecas utilizadas pelo desenvolvedor (em seu notebook, por exemplo) podiam estar em versões diferentes das rodadas no ambiente de produção, causando todo tipo de comportamento inesperado por parte do software, ou simplesmente não funcionando. Agora o desenvolvedor pode gerar um container daquele software, garantindo que nenhum problema de incompatibilidade de versões ou falta de bibliotecas aconteça. [VITALINO, Jeferson Fernando Noronha; CASTRO, Marcus André Nunes. 2016]

#### 2.1.1 Diferença entre Máquina Virtual e Contêiner

Servidores fazem um grande uso de virtualização pois permite isolar as aplicações das outras além de garantir o controle sobre a utilização dos recursos computacionais, como uso do processador e memória. Contudo, em ambientes que requerem muitas virtualizações, esse tipo de solução se torna muito custosa em termos de hardware já que ela requer que um ou vários novos sistemas operacionais rodem paralelamente, o que faz boa parte dos recursos computacionais ficarem alocados somente para o funcionamento desses sistemas.

Figura 01 – Comparação entre Máquina Virtual e Contêiner.



Fonte: captura de tela do site oficial *docker.com*.

Como pode ser notado na imagem acima, os containers de software apareceram para mudar esse cenário, ao contrario de virtualizar um ou vários sistemas operacionais, os containers herdam o *kernel* (núcleo) do sistema hospedeiro em modo “apenas leitura” para rodar as aplicações, o que faz os containers serem muito mais leves que máquinas virtuais, permitindo que se execute mais aplicações com a mesma infraestrutura que antes eram usadas com VMs (*Virtual Machine*, em português: Máquina Virtual). Para acessá-los usamos uma porta na máquina host que será redirecionada para uma porta dentro do contêiner, e utilizando dos módulos do *kernel* do sistema garante o isolamento e controle de recursos sem a necessidade da instalação de um novo sistema operacional completo. [What is a Container]

### 2.1.2 Orquestrador de Container

Um orquestrador de container é responsável por organizar todos os container de vários serviços de forma que possam trabalhar juntos independente do nó de processamento em que se encontram, aplicativos containerizados são dispostos a trabalharem no modelo de micro serviços, conectados por um rede virtual entre eles, tudo isso é controlado pelo orquestrador de container, além disso o orquestrador dá ao administradores de sistema um nível de abstração maior de modo que possa controlar as configurações dos micros serviços mais facilmente. Outras características encontradas nos orquestradores são as opções de *update* e *rollback* dos serviços (processos de atualizar e reverter para o estado anterior, respectivamente), gerenciamento de acesso a recursos, e a escalabilidade dos serviços.

O nome “orquestrador” vem do contexto de uma orquestra sinfônica, onde os containers seriam a orquestra e o orquestrador o maestro, que é responsável por controlar o fluxo de trabalho entre todos os envolvidos de forma harmoniosa.

## 2.2 CLUSTER

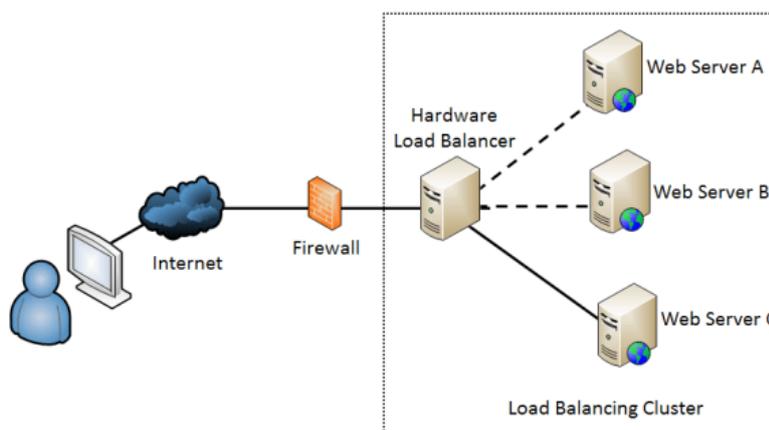
Um *cluster* (em português: aglomerado) são dois ou mais computadores trabalhando como se fossem uma única máquina, cada computador dessa rede é denominado um *node* (em português: nó), dividindo seus recursos para processar uma quantidade de dados que uma única maquina não conseguiria. Esse processo

é totalmente transparente para o cliente, de modo que ele tenha a sensação de estar acessando um único servidor. Existem vários tipos de *cluster* como por exemplo o de alta disponibilidade, que garante que se um nó do *cluster* parar de funcionar outro tomará seu lugar, sem que o cliente perceba que ocorreu alguma falha. Outros tipos muito utilizado é o de *load-balance* (em português: balanceamento de carga), que divide o acesso entre os nós, o de escalabilidade horizontal, que permite que mais recursos (nós) sejam adicionados aquele *cluster* a qualquer momento, sem interferir no funcionamento dos outros nós. Os tipos de cluster podem ser combinados para atenderem as demandas da regra de negócios. [BARROS, Andersown Becher Paes]

### 2.2.1 balanceamento de carga

O balanceamento de carga é muito utilizado na computação, ele permite dividir a carga de trabalho de um serviço entre mais servidores, parte do conceito de que os recursos são limitados e é necessário garantir que o limite não seja atingido, o que comprometeria o funcionamento do serviço. Esse processo geralmente é feito por um nó balanceador que redireciona cada requisição, com base em algum algoritmo de escalonamento, tentando manter uniforme o número de acessos a cada nó, ajudando a maximizar o uso dos recursos, minimizar o tempo de resposta e diminuir a carga de trabalho sobre um único recurso. Esse processo é totalmente transparente para o usuário, que o enxerga como um único sistema, além de garantir um nível de tolerância a falha, já que se algum nó vier a falhar os outros ainda continuarão respondendo as requisições.

Figura 02 - Esquema do nó balanceador dentro de uma rede.

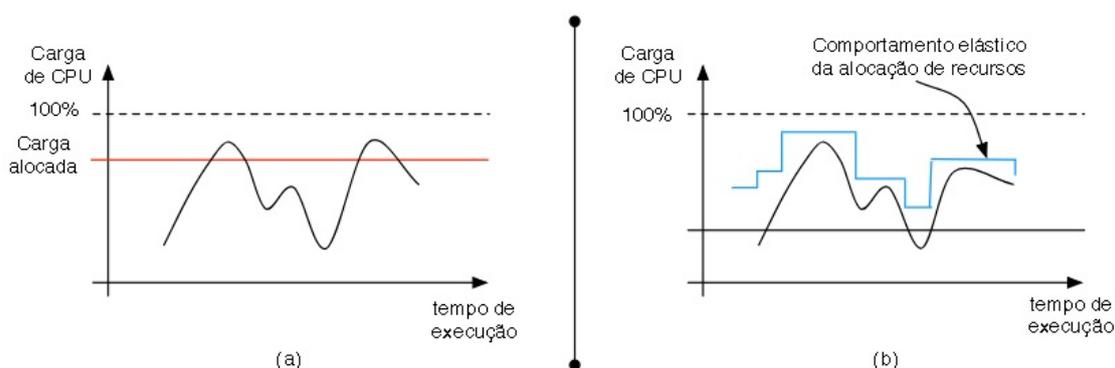


Como se pode ver na imagem acima, varias pessoas acessam o nó balanceador ao mesmo tempo, e cada uma é redirecionada a um nó diferente, dividindo a carga de acesso entre eles. [MELLO, Vanessa De Oliveira. 2018]

### 2.3 COMPUTAÇÃO ELÁSTICA

Em computação, elasticidade significa que o sistema se adaptará as mudanças da carga de trabalho de um serviço, realocando a CPU, RAM, Rede e Disco de forma automatizada, de modo que os recursos alocados serão o mais proximo possivel da demanda atual da aplicação, aumentando-os ou diminuindo-os conforme a necessidade, conforme exemplificado na imagem abaixo.

Figura 03 – Diferenças de modelos: (a) sem elasticidade; (b) com elasticidade.



Fonte: RIGHI, Rodrigo da Rosa. 2013. p. 3.

O ato de alocar mais ou menos recursos para uma única instancia da aplicação é chamado de elasticidade vertical, a elasticidade horizontal se dá quando se replica o número de instancias do serviço, geralmente para outros nós de processamento. Existe também as modalidades de elasticidade reativa e proativa. A reativa, a mais comum, reage à carga de trabalho atual do serviço, ao passar um limiar (*threshold*) de recursos, o controle reativo faz uma ação de escalonamento. O proativo, além disso, identifica padrões, tendências, para fazer uma ação de escalonamento momentos antes do limiar ser ultrapassado. Por exemplo, todo dia as quinze horas um determinado serviço sofre uma sobrecarga de acessos, o sistema reativo começaria o processo de escalonamento apenas quando o serviço já estivesse sob estresse, o proativo começaria minutos antes das quinze horas,

antevendo, com base nos dias anteriores, a sobrecarga de acessos. [RIGHI, Rodrigo da Rosa. 2013]

### 2.3.1 Comparação entre escalabilidade e elasticidade

O termo elasticidade está diretamente relacionado a alocar uma quantidade de recursos iguais a necessidade da aplicação a qualquer momento automaticamente. Já a escalabilidade está relacionada com a facilidade com que recursos são estaticamente adicionados ou removidos. Do ponto de vista do software, um ambiente elástico é por definição altamente escalável, porém um ambiente escalável não necessariamente é elástico. A elasticidade estaria mais relacionado com como a escalabilidade acontece, nesse caso visando maximizar o uso dos recursos usados pelas aplicações aumentando e diminuindo o acesso a eles **automaticamente**, reduzindo os custos da infraestrutura em modelos de pague-o-quanto-usar.

Não necessariamente um ambiente elástico sempre terá vantagem sobre um não elástico, por exemplo, uma aplicação onde não se tenha mudanças bruscas de uso de recursos ou em que as mudanças são muito previsíveis, usá-lo em um ambiente elástico não trará nenhum benefício, podendo até custar mais caro. Já aplicações que podem sofrer variações no uso dos recursos a qualquer momento tendem a se beneficiarem de ambientes elásticos, garantindo um melhor desempenho da aplicação em si a qualquer momento e reduzindo os custos para o cliente. [SCHOEB, Leah. 2017]

## 2.4 DOCKER

O Docker nasceu em 2013 quando a dotCloud, uma empresa que oferecia ambientes containerizados como PaaS (*Plataform as a service*, em português: Plataforma como serviço), decidiu tornar seu sistema *open-source* (código aberto), disponibilizando-o no GitHub, um repositório público de código-fonte onde todos podem contribuir. Logo fez muito sucesso e em seis meses já tinha por volta de 170 pessoas ao redor do mundo contribuindo com o projeto. Por volta de um ano depois o Docker estava chegando a sua versão “1.0” e considerada pronta para produção. Fez sucesso rápido por sua facilidade de uso e grandes empresas como Google e Amazon começaram a suporta-lo em suas *clouds*.

O Docker faz uso de vários módulos do *kernel* Linux para seu funcionamento, como o *namespaces*, que permite que cada container tenha seu próprio ambiente isolado; o *cgroups*, responsável por permitir que se limite o uso dos recursos computacionais usados pelos containers; e o *netfilter*, que permite aos containers se comunicarem entre si, criando regras de roteamento através do *iptables*. [VITALINO, Jeferson Fernando Noronha; CASTRO, Marcus André Nunes. 2016]

#### 2.4.1 Dockerfile

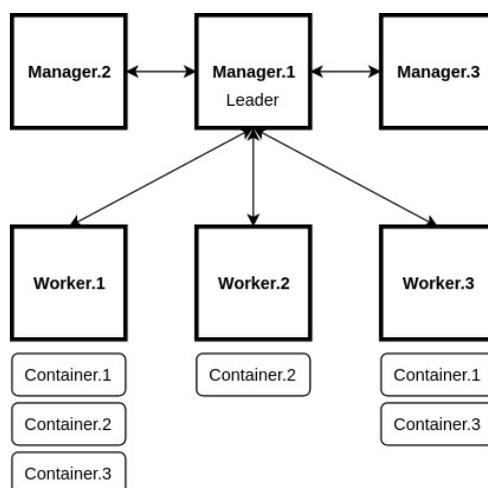
O Docker consegue contruir imagens de containers de forma automática lendo um arquivo chamado “*dockerfile*”. Um *dockerfile* é um arquivo que contenha todos os comandos que um usuário conseguiria utilizar na linha de comando durante a montagem de uma imagem de container. Usando o comando “*docker build*” o usuário pode construir um container de forma automatizada, executando uma cadeia de comandos em sucessão. [DOCKERFILE reference. Docker Documentation]

#### 2.4.2 Orquestrador Docker Swarm

O Docker Swarm é o gerenciador de *cluster* nativo do Docker, possui uma CLI declarativa muito simples de se usar, permitindo que nós sejam adicionados ao cluster muito rapidamente, bastando os computadores terem o Docker engine instalado, estarem na mesma rede, dividirem a mesma arquitetura (exemplo: x86\_64) e o tipo de sistema (alguma distribuição Linux). Definimos uma ou mais máquinas como máquinas gerente (*manager*) e uma ou mais máquinas como operárias (*worker*), todas as máquinas inseridas nesse ‘*swarm*’ (em português: enxame) são nós desse cluster, formando um cluster de balanceamento de carga e escalonamento horizontal, ou seja, podemos criar réplicas do mesmo serviço (um contêiner contendo uma aplicação) e dividir a carga de trabalho entre os *workers*, com o balanceamento de carga dos acessos a esses serviços sendo feito pelo *manager*. Por exemplo, ao executar o comando que adicionará um novo serviço, a máquina gerente irá automaticamente distribuir as réplicas do serviço entre os operários ativos, se colocarmos o servidor HTTP Apache como um serviço e especificarmos que ele terá três réplicas, isso significa que três containers do Apache serão instanciados, o gerente do *swarm* então fará o balanceamento de

carga entre esses três contêiner pelo método *round-robin*, ou seja, cada requisição HTTP será redirecionado para um contêiner Apache diferente por vez, voltando ao primeiro quando não houver outra réplica além das que já foram utilizadas.

Figura 04 – Esquema exemplo de um *cluster* Docker Swarm.



Fonte: próprio autor, elaborado no site *draw.io*.

Essa quantidade de réplicas de containers pode ser alterada, para mais ou para menos, a qualquer momento no *manager*, o que garante um certo nível de escalabilidade ao entregar o serviço, se o serviço Apache começar a ter um pico de acesso momentâneo, o número de réplicas dele pode ser rapidamente aumentado, dividindo a carga de acesso entre mais *workers*, quando o pico de acesso diminuir, o número de réplicas pode ser diminuído.

Ele garante também tolerância a falhas, já que os nós gerentes estão constantemente verificando a situação dos outros nós e containers rodando neles, caso um container não esteja respondendo, ele é finalizado e uma nova cópia é criada, se o nó não estiver respondendo todos os containers que estavam rodando naquele nó são replicados em outros nós. [SWARM mode overview]

#### 2.4.2.1 Compose file

Um compose file é um arquivo do tipo YAML que define um *stack* de containers, ou seja, varias aplicações containerizadas trabalhando juntas através de uma rede virtual no modelo de microserviços. Nela você define a imagem dos containers que serão usadas, as portas que serão redirecionadas, a rede virtual que usarão entre sí, definições de uso dos recursos, update e roll-back, entre varias

outras opções, então com apenas um comando você pode criar e executar varios containers com as configurações especificadas para cada um, que juntos entregarão um serviço. [DOCKERFILE reference]

#### 2.4.2.1.1 YAML

“YAML *Ain't Markup Language*” (em português, “YAML não é linguagem de marcação”) é uma linguagem de serialização legível por humanos comumente usada em arquivos de configurações. Varias linguagens de programação oferecem suporte para leitura e escrita YAML, além de várias IDEs oferecem suporte para apontar estruturas aninhadas e erros de sintaxe. YAML é inspirado por linguagens como o XML e Python porém mais minimalista.

### 2.5 SHELL SCRIPT

Um shell script é um programa de computador feito para ser rodado por um Unix Shell, um interpretador de comandos, a fim de realizar tarefas utilizando comandos específicos da linguagem do shell, pode conter funções, variáveis, *alias* (apelidos) e comandos do sistema operacional. São geralmente usados para automatizar tarefas de rotinas no S.O., economizando tempo dos administradores de sistema. Uma das principais vantagens do shell script é que os comandos e sintaxe utilizadas na programação são as mesmas usadas na linha de comando (do inglês: *command-line interface*, CLI) do sistema operacional, diferentemente de outras linguagens de programação ou de linguagens compiladas.

#### 2.5.1 Shell

Um shell é uma interface de usuário que permite a manipulação dos serviços do S.O., como gerenciamento de arquivos e diretórios, gerenciamento de processos, monitoramento e configuração do sistema. O shell pode ser usado localmente ou remotamente, através de alguma ferramenta de acesso remoto para a administração em sistemas multiusuários, que permitem mais de um usuário ao mesmo tempo. Um shell pode ser por linha de comando, ou gráfico (*graphic user interface*, GUI), que geralmente é integrado ao ambiente gráfico do sistema. Exemplos de shells CLI: Bash, o shell da maioria dos sistemas Unix como o Linux, MacOS/OS X e Solaris, no Windows temos o Prompt de comando e o PowerShell.

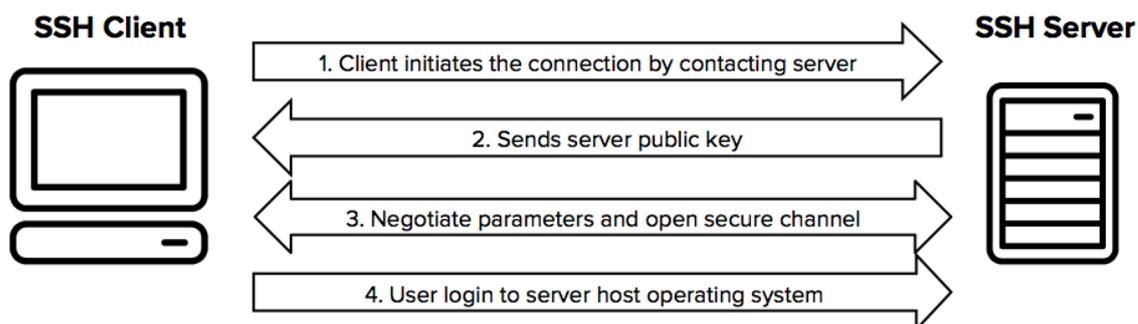
### 2.5.2 Bash

Bash (*Bourne-again Shell*) é um interpretador de comandos e linguagem de comando do Unix Shell por CLI, é o shell padrão do MacOS, Solaris 11 e da maioria das distribuições Linux. Ele é uma ferramenta muito poderosa pois permite fazer operações complicadas com apenas alguns comandos simples. Surgiu para substituir o Bourne Shell em 1989, faz várias operações que o Bourne Shell não faz como por exemplo o cálculo de inteiros sem a necessidade de processos externos. Sua sintaxe simplifica o redirecionamento de I/O (Input/Output), utilizando operadores para manipular a entrada e saída como o “|” (*pipe*), que passa a saída de um programa como a entrada de outro; o “>” (*write*) que redireciona a saída de um programa para outro local (um arquivo, por exemplo); e o “>>” (*append*) que anexa a saída de um programa em um local (um arquivo, por exemplo). A diferença do *write* para o *append* é que o primeiro apaga todo o conteúdo do destino para então escrever seus dados, o segundo acrescenta as informações às já existentes.

### 2.6 SECURE SHELL

O Secure Shell (SSH) é um protocolo de segurança utilizado por programas como o OpenSSH, PuTTY e CyberDuck que permite a administração remota de sistemas através da rede de modo seguro utilizando uma conexão criptografada, de modo que todas as autenticações de usuários, comandos, entrada/saída de dados e transferência de arquivos sejam protegidos contra a leitura por pessoas não autorizadas.

Figura 05 – Funcionamento do protocolo SSH.



Fonte: captura de tela do site oficial [ssh.com](http://ssh.com).

Como se pode ver na figura acima ele fornece uma conexão segura entre um cliente e um servidor através da troca de chaves publicas entre eles, garantindo que as informações trocadas pela rede estejam protegidos contra pessoas mal-intencionadas que as interceptem. Na etapa 1. o cliente inicia uma conexão contatando o servidor. Na etapa 2. o servidor envia a chave publica. Na etapa 3. ambos negociam os parâmetros da conexão e abrem um canal seguro. Na etapa 4. o cliente faz o login no sistema operacional do servidor. [SSH (SECURE SHELL)]

### **2.6.1 SSH-KeyGen**

O ssh-keygen é uma ferramenta usada para criar um par de chaves publica/privada para o SSH, suporta varios tipos de algoritmos criptograficos como o rsa, dsa, ecdsa e ed25519. Chaves publicas e o SSH podem ser usadas para automatizar o acesso a servidores, permitindo o login automático em um unico passo, sem a necessidade de digitar a senha a cada conexão, são usualmente utilizadas em scripts, sistemas de backup e ferramentas de configuração por desenvolvedores e administradores de sistema.

#### **2.6.1.1 Autenticação por chave publica**

A criptografia por chave publica ou criptografia assimétrica, é um sistema criptografico que usa um par de chaves, uma publica que pode ser distribuida livremente e uma privada que precisa ser secreta e estar na posse apenas de seu dono, ambas diferentes porém matematicamente ligadas, onde a chave publica será usada para criptografar as mensagens enviadas que só poderão ser decriptadas pelo detentor da chave privada, garantindo que pessoas não autorizadas não acessem aqueles dados.

## **2.7 SERVIDOR HTTP APACHE**

Um servidor *web* é um programa capaz de processar solicitações HTTP (*Hyper-Text Transfer Protocol*, em português: Protocolo de transferencia de Hiper Texto), que é o protocolo de comunicação padrão da *World Wide Web* (WWW), o Servidor Apache é um dos mais utilizados para essa finalidade, entre suas principais características estão o baixo consumo de recursos, suporte ao protocolo de transferencia de arquivos FTP (*File Transfer Protocol*), sua modularidade e o fato de ser *open-source* (código livre), ou seja, qualquer pessoa pode ter acesso ao seu código fonte e alterá-lo para seus propósitos específicos.

### 2.7.1 ApacheBench

O ApacheBench é uma ferramenta de *benchmark* (teste de mesa) feito pela Apache Foundation, a mesma do servidor web Apache, para testar o tempo de resposta de seu servidor HTTP, apesar disso, é genérico o suficiente para fazer testes em qualquer servidor que suporte o protocolo HTTP. Seu funcionamento basicamente é bombardear um determinado endereço *web* com requisições HTTP, a frequência com que ele bombardeia o endereço pode ser configurada durante a inicialização da ferramenta, por exemplo, para simular um pico de acesso buscando um cenário perto do real, ou, bombardear com uma quantidade ridícula de requisições para descobrir o pico máximo que aquele servidor pode processar.

### 2.8 RASPBERRY PI

O Raspberry Pi é um hardware do tamanho de um cartão de crédito com todas as características de um computador, tem um processador do tipo ARM, o mesmo utilizado em *smartphones*, oferece entradas USB (*Universal Serial Bus*) para conexão de mouse, teclado e outros dispositivos, porta HDMI (*High-Definition Multimedia Interface*) para um monitor, assim como uma porta ethernet para conexão de rede por cabo e um módulo *wireless* embutido para conexão sem fio. É muito usado para testes e ensino por seu baixo custo, pequeno tamanho e alta modularidade.

Figura 06 - Raspberry Pi 3 Model B.



Fonte: captura de tela do site oficial [raspberrypi.org](http://raspberrypi.org).

A imagem acima representa o modelo RPi3 *Model B*, possui as dimensões de 5,6cm x 8,5cm.

## CAPÍTULO III

Esse capítulo contém detalhes do desenvolvimento do *shell script*, do *cluster* de teste e dos resultados obtidos.

### 3.1 DESENVOLVIMENTO

O *script* desenvolvido em *bash*, consiste de alguns menus simples para sua fácil utilização pelo usuário, ao digitar '*dkscale --help*' o *script* retorna as opções básicas de uso, como, por exemplo, mostrar a versão: '*dkscale --version*' e o comando de auto escalonamento: '*dkscale auto*'. Com esse comando podemos iniciar um processo que irá monitorar o uso da memória e/ou processador das réplicas de um serviço,

Ao iniciar um processo de monitoramento o nome do serviço e o PID (*process identifier*) do processo são armazenados em um *log*, com o comando '*dkscale stop <nomeserviço>*' o *script* busca o número do PID armazenado no *log* e efetua um comando *kill <pid>*, matando o processo, parando o monitoramento. O mesmo *log* também serve para o comando '*dkscale show*' que mostra todos os serviços ao qual o *script* está monitorando.

Toda vez que o *script* faz um processo de replicação ou destruição de um contêiner essa ação é salva em um *log* chamado "*scale.log*", ele salva a data atual no formato "ano-mês-dia", a hora no formato "hora:minuto:segundo" e a mensagem que pode variar de *Scale UP* (aumentou) ou *Scale DOWN* (diminuiu), seguido do numero de replicas total após a operação, exemplo:

```
2018-08-07 at 13:00:15 Scale UP service apache_webserver to 2
```

```
2018-08-07 at 13:04:41 Scale DOWN service apache_webserver to 1
```

Caso ocorra algum erro durante alguma etapa do *script* a mensagem será salva no *err.log* redirecionando o *stderr* (2>>).

O código do *script* é dividido nas funções:

1. *dkhelp* – mostra um menu de orientação de como deve ser passado os parâmetros;
2. *show* – mostra os serviços que o *script* está monitorando;
3. *stop* – passando o nome do serviço como parâmetro para parar o monitoramento;
4. *auto* – função que vai verificar todos os parâmetros passados antes de iniciar o monitoramento ;

5. `scale` – função que vai fazer o monitoramento em sí, essa função é chamada em background (&) dentro da função 4 (auto) para que o terminal não seja bloqueado, já que ela roda em `loop`;

A função `auto` verifica todos os parâmetros passados pelo usuário para verificar se são válidos, por exemplo, verifica o nome do serviço para ver se ele existe ou já está sendo monitorado, passar um parâmetro com letras quando se espera um numérico, nesses casos uma mensagem exibe o erro e o `script` é encerrado.

Na função `scale` o `script` utiliza de comandos do Linux, como `grep`, `awk` e `sed`, que filtram um texto procurando por um padrão específico, por exemplo, para gravar o `hostname` e o IP dos nós em que o serviço está rodando. Após isso ele entra em um loop (`while`) que vai executar uma conexão SSH, a cada laço, em todos os nós em que o serviço tenha réplicas para obter as informações e salvá-las em um arquivo, como pode ser visto no trecho do código abaixo:

Figura 07 – Trecho do código que filtra os IPs e faz a conexão SSH.

```
# Verifica e filtra o uso da CPU e Memória das replicas do servico que estao rodando no manager e salva no log
docker stats --no-stream --format "table {{.Name}}\t{{.CPUPerc}}\t{{.MemPerc}}" | grep -w $SERVICE > $RESFILE

# Retorna o nome de todos os nodes que o servico esta rodando exceto este
docker service ps $SERVICE | sed -n '1!p' | grep 'Running' | grep -w $SERVICE | awk '{print $4}' | grep -v $(hostname) > $NODESFILE

if [ -s $NODESFILE ];then
# Pega o endereço ip de todos os nodes que o servico esta rodando
while read -r LINE
do
    NODEIP=$(docker node inspect $LINE --format '{{.Status.Addr }}')
    sed -i "s/$LINE/$NODEIP/g" $NODESFILE
done < "$NODESFILE"

# Remove linhas duplicadas, caso tenha mais de uma replica em um mesmo nó
awk '!a[$0]++' $NODESFILE > $IPFILE

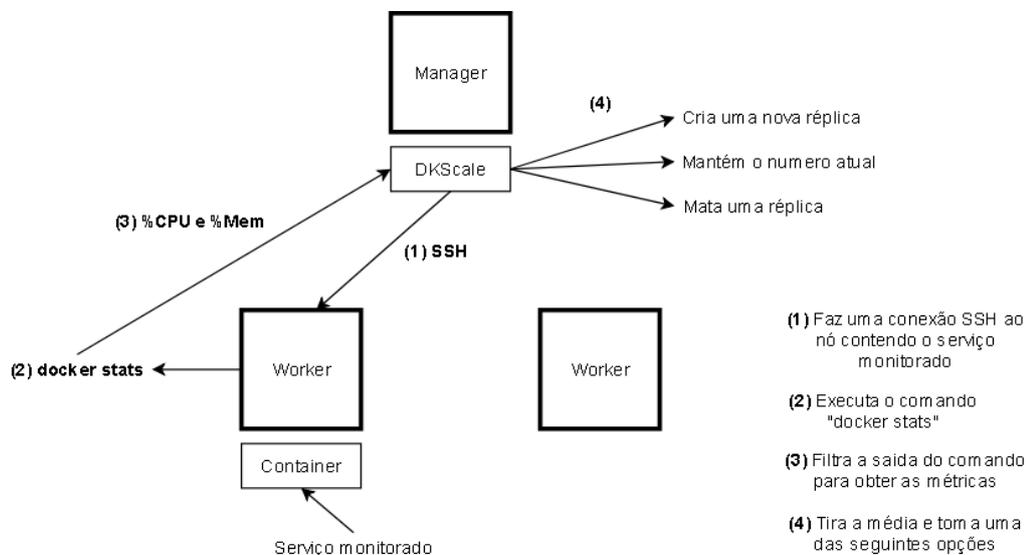
# Conecta nos nodes para pegar os dados do servico
while read -r LINE
do
    CON=$USER@$LINE
    ssh -n $CON 'docker stats --no-stream --format "table {{.Name}}\t{{.CPUPerc}}\t{{.MemPerc}}"' | grep -w $SERVICE >> $RESFILE
done < "$IPFILE"
fi
```

Fonte: próprio autor, captura de tela.

Na imagem acima pode-se ver toda a rotina de obtenção dos IPs dos nós em que o serviço monitorado está rodando, em seguida é feita uma conexão SSH para cada um deles executando o comando `ssh -n $CON docker stats --no-stream --format "table {{.Name}}\t{{.CPUPerc}}\t{{.MemPerc}}"`, que retorna os nomes de todos os containers rodando naquele nó junto com a porcentagem de utilização da CPU e da memória RAM, após isso é feita uma filtragem utilizando o `grep`, buscando pelas linhas que contenha o nome do serviço monitorado e salvando-os em um arquivo. Com os dados salvos, ele tira a média aritmética do uso dos

recursos e entra em uma série de *if* e *elif* que baseado nos parâmetros decidirá se criará uma nova réplica, destruirá uma ou manterá o número atual.

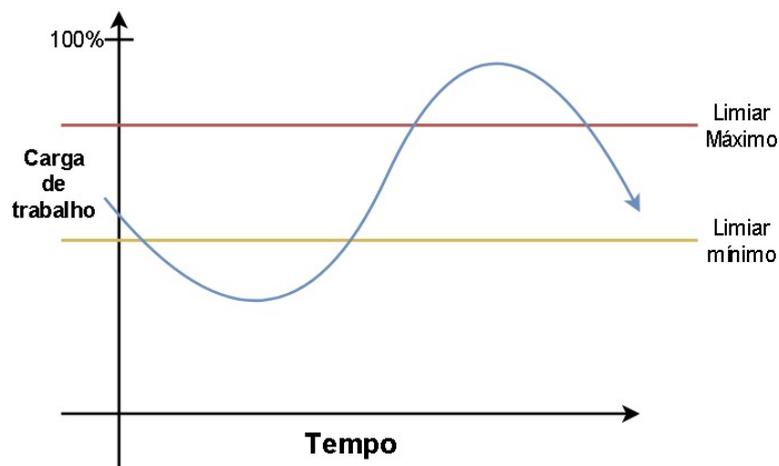
Figura 08 - Esquema de funcionamento do *script*.



Fonte: próprio autor, elaborada no site *draw.io*.

Na imagem acima pode-se ver o fluxo de funcionamento do *script*, na etapa (1) ele faz uma conexão SSH para o nó contendo o serviço monitorado; na etapa (2) ele executa o comando que obterá as métricas dos containers rodando naquele nó; na etapa (3) ele filtra pelo nome do serviço monitorado e a porcentagem de uso da CPU e RAM; na etapa (4) ele faz uma ação baseada nos limites mínimo e máximo passados, como pode se ver na imagem abaixo:

Figura 09 - Gráfico de exemplo para tomada de decisão.



Fonte: próprio autor, elaborado no site *draw.io*.

Na figura acima dado a carga de trabalho da aplicação pelo tempo de execução definimos um limiar mínimo e um máximo, que seria a porcentagem de uso da CPU e/ou RAM, caso a média entre todas as réplicas fique abaixo do limiar mínimo entende-se que o serviço está subutilizando os recursos alocados a ele, ou seja, ele tem mais recursos alocado do que precisa, nesse caso o *script* mata uma réplica. Caso a média fique acima do limiar máximo, entende-se que o serviço está quase atingindo o máximo de recursos alocados a ele, nesse caso cria-se uma nova réplica, oferecendo aquele serviço mais recursos para que continue em execução sem problemas. O objetivo é manter a média de utilização entre os dois limiares, afim de sempre oferecer o serviço com uma boa performance ao usuário.

Com o *script* funcionando criei o *dockerfile* para containeriza-lo, ele copia o *script* para o diretório `"/usr/local/sbin"` dentro do container que está no PATH do sistema, facilitando na hora de executá-lo. Em caso de falha, criei um segundo script chamado `"dkrun.sh"` que inicia junto do container, esse *script* é responsável por reiniciar o processo de monitoramento caso o container ou o sistema operacional sejam reiniciados, ele lê periodicamente o *log* de serviços que estão sendo monitorados e verifica nos processos ativos do container se eles ainda existem, caso existam não faz nada, caso não exista ele reinicia o monitoramento executando o *script* `dkscale` passando os mesmos parâmetros definidos anteriormente antes da reinicialização. Para evitar repetir o código criei também um *script* chamado `"dkset.sh"` que contem todas as funções e variáveis que os outros dois *scripts* (`dkscale` e `dkrun.sh`) usam, carregando-os através do comando `"source"`.

Outra necessidade pensada foi a de tolerância a falhas por parte do *script*, é possível atingir esse objetivo utilizando um sistema de arquivos distribuídos, permitindo que os mesmos *logs* do *script* sejam compartilhados entre os nós gerentes, o *script* verifica se o *host* atual é o `"Leader"` (líder, em tradução livre), esse gerente líder é definido automaticamente pela *docker engine*, é ele que toma as decisões dentro do *swarm*, ou outros nós gerentes são identificados como `"reachable"` (alcançável, em tradução livre), o que significa que são possíveis substitutos do líder. Se o nó líder parar de funcionar, por qualquer motivo, algum deles automaticamente se tornará o novo líder e continuará a executar as tarefas do *cluster*, e o container contendo o *script* continua o monitoramento de onde o outro parou.

### 3.1.1 Configuração do cluster

Para fim de demonstração configurei um pequeno *cluster* com três Raspberry pi 3 model B, montados juntos na mesma caixa junto com um switch de rede de cinco portas. Instalei o Ubuntu Server 18.04.1 LTS como sistema operacional. Atualizei o sistema e configurei seus *hostnames*, respectivamente, para *manager1*, *worker1* e *worker2* assim como um IP fixo para os três, respectivamente, 192.168.1.101, 192.168.1.102 e 192.168.1.103 .

Após isso iniciei o *daemon* do *ssh* com o comando `'systemctl start ssh'` e habilitei-o para iniciar com o sistema operacional com o comando `'systemctl enable ssh'`, isso é requerido porque o *script*, rodando no *manager1*, fará várias conexões *SSH* nos *workers* para coletar as informações dos recursos computacionais usados. Foi necessário também alterar o arquivo `/boot/firmware/cmdline.txt` para que ele carregue no *boot* do sistema os módulos necessários para que o *docker* consiga determinar a quantidade de memória RAM usada pelos containers, bastando adicionar no final da linha desse arquivo os seguintes parametros: `cgroup_enable=memory cgroup_memory=1 swapaccount=1`. Isso é necessário apenas nessa versão de RPi3, na versão padrão X86\_64 não é necessário. Após isso reiniciei o sistema para fazer valer as alterações.

Figura 10 - Foto do *cluster* com três RPi e o switch de rede.



Fonte: próprio autor, foto.

Utilizando o `ssh-keygen` gerei um par de chaves pública/privada do tipo ECDSA de 521 bits para habilitar a conexões em um único passo sem a necessidade de senha, a chave privada ficará na pasta `/home/ubuntu/.ssh` do *manager1* que nesse caso é o cliente, para configurar os servidores que nesse caso são os *workers* basta copiar com o comando `scp` (Secure Copy) a chave pública para a pasta `.ssh` no diretório *home* do usuário ao qual a conexão será feita (e que pertença ao grupo `docker`) e depois copiar o conteúdo da chave pública para o arquivo `authorized_keys` com o comando `cat id_ecdsa.pub >> authorized_keys`.

Prosseguir para a instalação do `docker`, utilizei o *script* da página oficial deles em <https://get.docker.com> com o comando `curl -fsSL get.docker.com -o get-docker.sh`, alterei com o editor de texto CLI *nano* o valor da variável `DEFAULT_CHANNEL_VALUE="edge"` para `"stable"`, a fim de instalar a versão estável, após isso bastou executar o script para fazer a instalação automaticamente, instalando o `Docker` versão 18.06.1-CE. Finalmente com o `docker` instalado só basta iniciar o *swarm mode* com o comando `docker swarm init` no *manager1*, que já começara como um *'manager'* (gerente) gerando uma *join-token*, um comando com uma chave para ser usado nos outros nós do *cluster* que entrarão como *workers* (trabalhadores). Exemplo de uma *'join-token'*:

```
docker swarm join --token SWMTKN-1-
4ixulw6bb5ewt4qmzkclj7hr2javze22gydjzg06x0wa2y6bho-
6bxp4rrt46rb6d84l733108dc 192.168.1.111:2377
```

Basta colar esse comando no shell dos outros nós para adicioná-los no `cluster Swarm`.

Como usarei o servidor HTTP `apache` como exemplo, preciso criar uma pasta compartilhada entre os nós para garantir que o `index.html` esteja acessível para todos, para isso utilizei o servidor de arquivos distribuídos `GlusterFS` no *manager1* instalando o software `glusterfs-server` e criando um volume para acesso no diretório `\compartilhado`. Nos *workers* instalei o software `glusterfs-client` e editei o `\etc/fstab` para montar o volume através IP do *manager1*.

Para finalizar criei um diretório chamado `apache_webserver` dentro do diretório `\compartilhado` onde criei o arquivo `index.html` que será usado pelo `apache`, dentro do arquivo chamei uma função básica do `php` para mostrar suas informações `<?php phpinfo(); ?>`.

Para o *script* funcionar duas coisas precisam estar configuradas: um usuário que pertença ao grupo ‘*docker*’ nos nós do cluster, de modo que ele possa executar comandos privilegiados na *docker engine*, e a criptografia de chave pública para permitir conexão em um único passo sem a necessidade de senha através do *ssh*. Após isso basta iniciar o container com os seguinte comando:

```
docker run -tid --privileged --name dkcontainer --cpus=0.3 -m 32M --network=host --
volume /var/run/docker.sock:/var/run/docker.sock --volume ~/.ssh:/root/.ssh/ --
volume /var/log/dkscale:/var/log/dkscale/ --restart=always bruzt/dkscale:latest
```

**docker run -tid:** Comando que inicia um container em modo interativo e em segundo plano;

**--privileged:** flag que dá ao container mais privilégios dentro do sistema;

**--name dkcontainer:** Dá ao container o nome “*dkcontainer*”, permitindo acessá-lo através desse nome;

**--cpus=0.3:** Limita o tempo de CPU disponível ao container a no máximo 30% de uso;

**-m 32M:** Limita em 32 Megabytes a memória RAM disponível ao container;

**--network=host:** Coloca o container na mesma rede do sistema *host*, permitindo-o acessar os componentes daquela rede, necessário para acessar os outros nós do *Swarm*;

**--volume /var/run/docker.sock:/var/run/docker.sock:** Cria um volume do *socket* do *docker* da máquina *host* com o *docker* dentro do container, permitindo que um controle o outro;

**--volume ~/.ssh:/root/.ssh/:** Cria um volume do diretório *~/.ssh* do *host* com o diretório */root/.ssh* do container, dando acesso ao container a todas as chaves públicas e privadas do *host*;

**--volume /var/log/dkscale:/var/log/dkscale/:** Cria um volume do diretório usado pelo *script dkscale* para guardar os logs de seu funcionamento, necessário para a persistência dos dados;

**--restart=always:** Garante que se o container parar de funcionar ou o sistema operacional for reiniciado esse container será automaticamente reiniciado;

**bruzt/dkscale:latest:** Nome da imagem e versão utilizada do container, nesse caso utilizando o repositório do Docker Hub e a última versão.

Com o container iniciado é preciso chamar o *script* passando o nome do serviço que você deseja que tenha elasticidade e o *username* que pertence ao grupo *docker* nos nós, permitindo que o *script* colete as informações através de uma conexão SSH de forma automatizada utilizando o comando “*docker stats*”, esse comando retorna uma lista com todos os container sendo executados naquele sistema junto com a utilização da CPU e memória RAM de cada um e é com essas informações que o *script* toma as decisões de escalonamento. Abaixo um exemplo do comando necessário para iniciar o *script*:

```
docker exec dkcontainer dkscale auto --name apache_webserver --user ubuntu --mincpu 30 --maxcpu 70 --time 10 --cpuonly
```

**docker exec:** Comando que permite executar um comando dentro de um container;  
**dkscale auto:** Executa o *script* com a opção *auto*, responsável por verificar as informações passadas pelo usuário e iniciar o processo que fará o escalonamento;  
**--name apache\_webserver:** Passa o nome do serviço que queremos tornar elástico, nesse caso “*apache\_webserver*”;  
**--user ubuntu:** Passa o nome do usuário que pertence ao grupo *docker* em todos os nós, nesse caso o usuário “*ubuntu*”;  
**--mincpu 30:** Define que o limiar de CPU para matar uma réplica é 30% ou menos do uso;  
**--maxcpu 70:** Define que o limiar de CPU para criar outra réplica é 70% ou mais do uso;  
**--time 10:** Define que o processo de escalonamento só deve acontecer no mínimo a cada dez segundos;  
**--cpuonly:** Define que o processo só levará em conta o uso da CPU, o uso da memória será ignorado.

O algoritmo leva em consideração a quantidade de CPU disponível para o serviço, por exemplo, se o container tem acesso a no máximo 10% do tempo de CPU, ele define esse valor como 100%, então, nesse caso, os 30% mínimos que foi definido na chamada do *script* representam 3% do uso da CPU, o mesmo vale para os 70%, que nesse caso representam 7%.

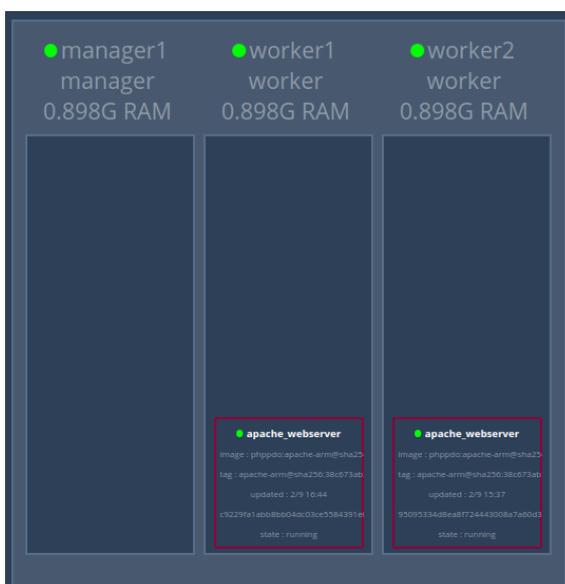
### 3.1.2 Testes

Configurei o serviço do Apache para ter acesso a somente 64MB de memória do sistema e no máximo 10% do tempo da CPU, essa configuração é bem escassa nos recursos, levando em consideração que está sendo executado em um RPi3, de modo que o limiar seja atravessado o mais rápido possível para demonstrar o funcionamento do script.

Para acompanhar o processo, utilizei o *Swarm Visualizer*, um container disponível no docker hub que permite ter uma representação gráfica em tempo real dos nós e serviços do *Swarm* e também pelo comando “*docker stats*” para verificar os recursos utilizados em cada nó do cluster.

Utilizando o programa ApacheBench para simular o uso intensivo do serviço, defini para o programa fazer uma série de requisições por segundo durante dois minutos através do comando “*ab -t 120 http://192.168.1.101:8080/index.php*”, nesse momento existe apenas uma réplica do apache no *cluster*, já que ele está ocioso. Assim que o teste começa o uso do processamento atinge os 10% reservados para ele, que representa 100% do valor limitado a ele, cerca de quinze segundos depois, o script aumenta o número de réplicas para dois, como pode se ver na imagem abaixo:

Figura 11.1 - Swarm Visualizer em execução, com duas réplicas.



Fonte: Próprio autor, captura de tela.

A figura acima representa cada nó (cada RPi) como uma coluna e cada quadrado dentro das colunas como uma réplica de um serviço sendo executado naquele nó. Para acompanhar a quantidade de recursos utilizados por cada réplica utilizei o comando “`docker stats`” com a opção “`--format 'table {{.Name}}\t{{.CPUPerc}}\t{{.MemPerc}}'`”, que só mostrará o nome de cada container e suas porcentagens de utilização da CPU e memória RAM, como pode ser visto na imagem abaixo:

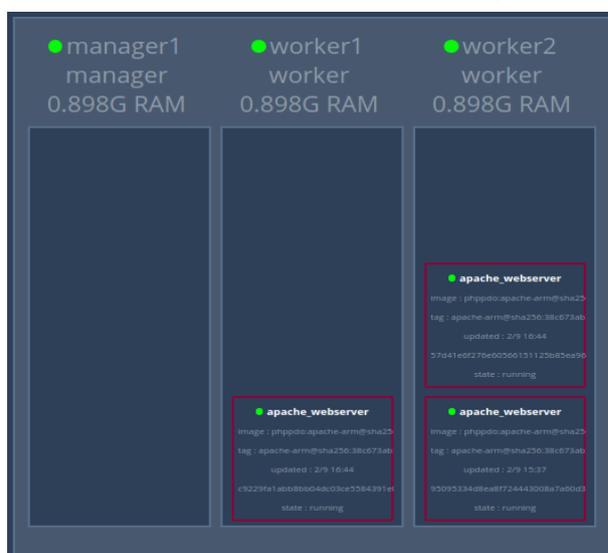
Figura 11.2 - “`docker stats`” em execução nos nós *Worker1* e *Worker2*, com duas réplicas.

NAME	CPU %	MEM %
apache_webserver.1.zqzs8tm03erv2ccjdzub8rmce	8.40%	15.12%
NAME	CPU %	MEM %
apache_webserver.2.wrgdtcse7ru8da44pr3irhse	8.69%	11.04%

Fonte: Próprio autor, captura de tela.

A imagem acima mostra as réplicas no *worker1* e *worker2* e o quanto da CPU e memória RAM reservados para eles cada um está usando, note que limitei o uso total de processamento para esse serviço em 10%, então os ~8% utilizados pelos dois representam 80% do total reservado para eles, acima do limiar de 70% que defini anteriormente. Cerca de vinte segundos depois uma terceira réplica é criada, como pode se ver na imagem abaixo do *Swarm Visualizer*:

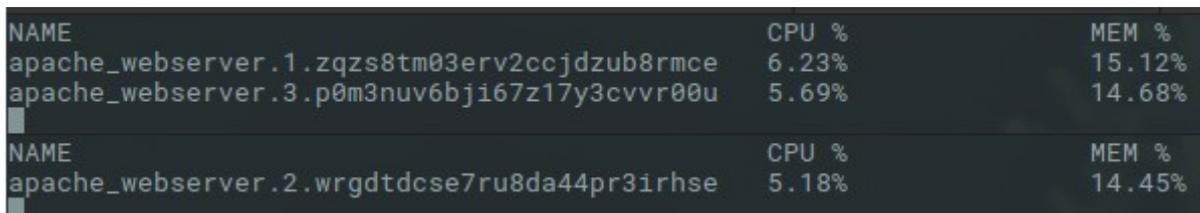
Figura 12.1 - *Swarm Visualizer* em execução, com três réplicas.



Fonte: Próprio autor, captura de tela.

Abaixo podemos ver como ficou o “docker stats” após a terceira réplica:

Figura 12.2 - “docker stats” em execução nos nós *worker1* e *worker2*, com três réplicas.



```
NAME                                CPU %    MEM %
apache_webserver.1.zqzs8tm03erv2ccjdzub8rmce 6.23%    15.12%
apache_webserver.3.p0m3nuv6bj167z17y3cvvr00u 5.69%    14.68%
NAME                                CPU %    MEM %
apache_webserver.2.wrgdtdcse7ru8da44pr3irhse 5.18%    14.45%
```

Fonte: Próprio autor, captura de tela.

Na imagem acima podemos ver que ao criar a terceira réplica o uso médio da CPU fica por volta de 5% (50% do total disponível), dentro dos limites que definimos quando chamei o *script* (30% mínimo e 70% máximo), com isso, não importa o tempo em que se deixe o ApacheBench rodando, o serviço tende a permanecer com três réplicas.

Após o apache bench ser finalizado o uso da CPU nos três cai para 0%, e cerca de dez segundos depois uma das réplicas é eliminada, as duas remanescentes também ficam com 0% de uso, e cerca de quinze segundos depois outra réplica é eliminada, ficando com apenas uma, como no início.

## CONCLUSÃO E TRABALHOS FUTUROS

Ao final conclui que utilizando da alta capacidade de escalonamento nativa do *Docker Swarm* é possível dar a ele elasticidade no uso dos recursos do *cluster*, automatizando-o através de um *bash script* e conexões SSH para monitorando o uso de processamento e memória de cada container sendo executado nos nós. Porém algumas questões de segurança devem ser levadas em consideração, por exemplo, ao usuário pertencente ao grupo *docker* utilizado para executar o comando de obtenção dos recursos, como em tal ambiente geralmente se trabalha apenas com aplicações containerizadas esse usuário teria acesso para manipular todas elas, como um usuário administrador teria, nesse caso as políticas aplicadas a usuários administradores também devem ser aplicadas a esse usuário, outro ponto seria em relação as chaves primarias que dão acesso a esse usuário sem a necessidade de senha, que também devem ser devidamente protegidas.

Para o futuro estou pensando em fazer uma interface gráfica pois manipular o *script* por linha de comando pode não ser tão fácil para algumas pessoas, a interface seria acessível pelo navegador, uma página PHP onde você define os parâmetros e ao clicar no botão para executá-la ela faria uma chamada ao sistema, executando o *script* com os parâmetros desejados, faria também um sistema de autenticação de usuário, para que só pessoas autorizadas tenham acesso a essa página.

## REFERÊNCIAS BIBLIOGRÁFICAS

VITALINO, Jeferson Fernando Noronha; CASTRO, Marcus André Nunes. **Descomplicando o Docker**. Rio de Janeiro: Brasport, 2016.

MORENO, Daniel. **Certificação Linux LPIC-1**. São Paulo: Novatec, 2016.

RIGHI, Rodrigo da Rosa. Elasticidade em cloud computing: conceito, estado da arte e novos desafios. **Revista Brasileira de Computação Aplicada**, Passo Fundo, v. 5, n. 2, p. 2-17, out. 2013. Disponível em:  
<<http://seer.upf.br/index.php/rbca/article/view/3084/2370>> Acessado em 02 set. 2018.

SCHOEB, Leah. **Cloud Elasticity Vs Cloud Scalability**. Turbonomic, 2017. Disponível em: <<https://turbonomic.com/blog/on-technology/cloud-elasticity-vs-cloud-scalability/>> Acessado em 15 Out. 2018.

FELTER, Wes; *et al.* **An Updated Performance Comparison of Virtual Machines and Linux Containers**. Austin: IBM Research, 2014. Disponível em:  
<[https://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/\\$File/rc25482.pdf](https://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/$File/rc25482.pdf)>. Acessado em 15 nov. 2017.

GOMES, Rafael; SOUZA, Rodrigo. **Docker - Infraestrutura como código, com autonomia e replicabilidade**. Ondina: Superintendência de Tecnologia da Informação, Universidade Federal da Bahia (UFBA). Disponível em:  
<<http://www.ixwticifes.ufba.br/modulos/submissao/Upload-275/66257.pdf>>. Acessado em: 21 ago. 2018.

BARROS, Andersown Becher Paes. **Computação em Cluster**. Cuiabá: Instituto Cuiabano de Educação. Disponível em:  
<[http://www.ice.edu.br/TNX/encontrocomputacao/artigos-internos/prof\\_andersown\\_computacao\\_em\\_cluster.pdf](http://www.ice.edu.br/TNX/encontrocomputacao/artigos-internos/prof_andersown_computacao_em_cluster.pdf)>. Acessado em: 12 abr. 2018.

SVIATOSLAV, A.; SERGEY, C. **Advantages of Using Docker for Microservices.**

Disponível em: <<https://rubygarage.org/blog/advantages-of-using-docker-for-microservices>>. Acessado em 24 ago. 2017.

WHAT is a Container. **Docker.** Disponível em:

<<https://www.docker.com/resources/what-container>> Acessado em 22 ago. 2018.

DOCKERFILE reference. **Docker Documentation.** Disponível em:

<<https://docs.docker.com/engine/reference/builder/>>. Acessado em 22 ago. 2017.

SWARM mode overview. **Docker Documentation.** Disponível em:

<<https://docs.docker.com/engine/swarm/>> Acessado em 23 ago. 2017.

COMPOSE file version 3 reference. **Docker Documentation.** Disponível em:

<<https://docs.docker.com/compose/compose-file/>>. Acessado em 23 ago. 2017.

RIBEIRO, Fernando. **Conexões SSH sem senha fácil e descomplicado.** Viva o

Linux. Disponível em: <<https://www.vivaolinux.com.br/artigo/Conexoes-SSH-sem-senha-facil-e-descomplicado>>. Acessado em 17 mar. 2018.

RUBENS, Paul. **What are containers and why do you need them?**. 2017.

Disponível em: <<https://www.cio.com/article/2924995/software/what-are-containers-and-why-do-you-need-them.html>>. Acessado em 13 set. 2018.

SSH (SECURE SHELL). **SSH Communications Security.** Disponível em: <<https://www.ssh.com/ssh/>>. Acessado em 27/07/2018.

WHAT is a Raspberry Pi?. **Raspberry Pi.** Disponível em:

<<https://www.raspberrypi.org/help/what-%20is-a-raspberry-pi/>> Acessado em 26 out. 2018.

MELLO, Vanessa De Oliveira. **Load Balance: o que é e como funciona.** KingHost.

2018. Disponível em: <<https://king.host/blog/2018/07/load-balance/>> Acessado em 26 out. 2018.

## APÊNDICES

### A - Dockerfile do container “DKScale”

```
1 FROM docker:stable
2
3 RUN apk update && apk upgrade && apk --no-cache add bash procps openssh
4 RUN mkdir /usr/local/sbin
5
6 COPY ./dkscale /usr/local/sbin/dkscale
7 COPY ./dkset.sh /usr/local/sbin/dkset.sh
8 COPY ./dkrun.sh /usr/local/sbin/dkrun.sh
9
10 RUN chmod +x /usr/local/sbin/dkscale
11 RUN chmod +x /usr/local/sbin/dkset.sh
12 RUN chmod +x /usr/local/sbin/dkrun.sh
13
14 RUN sed -i 's/exec "$@"/' /usr/local/bin/docker-entrypoint.sh
15
16 RUN echo "exec dkrun.sh &" >> /usr/local/bin/docker-entrypoint.sh
17 RUN echo 'exec "$@"' >> /usr/local/bin/docker-entrypoint.sh
18
19 EXPOSE 22
```

## B - Compose file do serviço Apache

```
1  version: '3.5'
2
3  services:
4  webserver:
5  image: bruzt/phppdo:apache-arm
6  networks:
7  - network
8  ports:
9  - 8080:80
10 volumes:
11 - /mnt/gfs/apache_webserver:/var/www/html
12 healthcheck:
13 start_period: 15s
14 deploy:
15 replicas: 1
16 resources:
17 limits:
18   cpus: '0.1'
19   memory: 64M
20
21 networks:
22 network:
23 driver: overlay
```

### C - Código fonte do script “dkrun.sh”

```

1  #!/bin/bash
2
3  source /usr/local/sbin/dkset.sh
4
5  while true
6  do
7      if [ -s $SERVPIDLOG ] # -s retorna true se o arquivo existe e tem o tamanho maior
      que 0 (tem alguma coisa escrita nele)
8          then
9              while read -r LINE
10             do
11                 SERVNAME=$(echo $LINE | awk '{print $1}')
12                 SERVPID=$(echo $LINE | awk '{print $2}')
13
14                 TESTPID=$(ps -eo pid,cmd | grep -w $SERVNAME | grep -w
$SERVPID | grep -v "grep" | awk '{print $1}')
15                 if [ -z $TESTPID ]
16                 then
17                     COMMAND=$(cat $PARAMETERSLOG | grep -w
$SERVNAME)
18                     cat $SERVPIDLOG | grep -v $SERVNAME > $SERVPIDLOG
19                     cat $PARAMETERSLOG | grep -v $SERVNAME >
$PARAMETERSLOG
20
21                     dkscale auto $COMMAND
22                 fi
23             done < "$SERVPIDLOG"
24         fi
25     sleep 10
26
27 done

```

## D - Código fonte do script “dkscale”

```

1  #!/bin/bash
2
3  source /usr/local/sbin/dkset.sh
4
5  #####
6  ##### MAIN #####
7  #####
8
9  while test $# -gt -1
10 do
11     case "$1" in
12         -h)
13             dkshelp
14             exit 0
15         ;;
16         --help)
17             dkshelp
18             exit 0
19         ;;
20
21         -v)
22             echo $VERSION
23             exit 0
24         ;;
25         --version)
26             echo $VERSION
27             exit 0
28         ;;
29
30         auto)
31             auto "$@"
32             exit 0
33         ;;
34
35         show)
36             show
37             exit 0
38         ;;
39
40         stop)
41             stop "$@"
42             exit 0
43         ;;
44
45         *)
46             dkshelp
47             exit 0
48         ;;
49     esac
50
51     shift
52 done

```

## E - Código fonte do script "dkset.sh"

```

1  #!/bin/bash
2
3  # Versao do script
4  VERSION="DkScale 1.2"
5
6  # Directorio dos logs
7  LOGDIR='/var/log/dkscale'
8
9  # Verifica se o directorio dos logs existe, caso nao ele o cria
10 if [ ! -d "$LOGDIR" ]; then
11     mkdir $LOGDIR
12 fi
13
14 # Coloca o endereco dos logs em variaveis
15 SCALELOG=$LOGDIR/scale.log           # Log do escalonamento
16 ERRLOG=$LOGDIR/err.log              # Log de erro
17 SERVPIDLOG=$LOGDIR/servpid.log      # Log onde fica os nomes dos
    serviços monitorados e PIDs desses processos
18 PARAMETERSLOG=$LOGDIR/parameters.log # Log onde fica os parametros dos
    monitoramentos
19
20 # Logs de escalonamento e erro
21 touch $SCALELOG
22 touch $ERRLOG
23 touch $SERVPIDLOG
24 touch $PARAMETERSLOG
25
26 # Variaveis com os valores padrao
27 #####
28 USER=root
29
30 MINCPU=10
31 MINMEMORY=10
32
33 MAXCPU=90
34 MAXMEMORY=90
35
36 MINREPLICAS=1
37 MAXREPLICAS=10
38
39 TIME=60
40
41 CPUONLY=1
42 MEMORYONLY=1
43 #####
44
45 NUMBERS='^[0-9]+$' # Para verificar se nao e um numero
46
47 #####
48 ##### HELP #####
49 # Funcao para exibir um menu de ajuda, mostrando as opcoes iniciais
50 function dkshelp {
51     echo "
52     dkscale auto [option]
53
54     auto          auto-scale option
55     -h, --help    Help menu
56     show          Show the services in monitoring

```

```

57         stop          Stop monitoring a service
58         -v, --version  Show dkscale version"
59     }
60
61     #####
62     ##### SHOW #####
63     function show {
64         cat $SERVPIDLOG | awk '{print $1}' 2>>$ERRLOG    # Printa o nome dos servicos
que estao sendo monitorados
65     }
66 }
67
68 #####
69 ##### STOP #####
70 function stop {
71     shift
72     SERVICE=$1
73     if [ -z $SERVICE ];then          # Se nao ouver nada em $service
74         echo "
75             dkscale stop [service]
76             "
77         exit 1
78     fi
79
80     PIDK=$(cat $SERVPIDLOG | grep -w $SERVICE | awk '{print $2}' 2>>$ERRLOG)
81     if [ -z "$PIDK" ];then          # -z retorna true se a variavel estiver vazia
82         echo "Service not found"
83         exit 1          # Finaliza o scrip caso o servico nao seja encontrado
84     fi
85
86     kill $PIDK          # Mata o processo
87     cat $SERVPIDLOG | grep -v $SERVICE 1> $SERVPIDLOG 2>>$ERRLOG    #
Remove do log o processo finalizado
88     cat $PARAMETERSLOG | grep -v $SERVICE > $PARAMETERSLOG
89     rm -rf $LOGDIR/$SERVICE          # Remove o diretorio
do servico
90     printf "Stopping monitoring $SERVICE\n"          # printa mensagem
91 }
92
93 #####
94 ##### AUTO #####
95 # funcao que verificara as condicoes do servico
96 function auto {
97
98     if [ "$(echo "$@" | awk '{print $1}')" == "auto" ]
99     then
100         shift
101     fi
102
103     echo "$@" 1>> $PARAMETERSLOG 2>>$ERRLOG
104
105     while test $# -gt 0
106     do
107
108         case "$1" in
109
110             --help)
111                 echo "
112                 dkscale auto --name [service] [Options]
113
114                 --help          Show Menu.

```

```

115         --mincpu           Minimum amount in % of CPU usage to Scale DOWN.
        Default: 10
116         --minmemory       Minimum amount in % of RAM usage to Scale DOWN.
        Default: 10
117         --maxcpu          Maximum amount in % of CPU usage to Scale UP. Default:
        90
118         --maxmemory       Maximum amount in % of RAM usage to Scale UP. Default:
        90
119         --minreplicas      Minimum number of replicas. Default: 1
120         --maxreplicas      Maximum number of replicas. Default: 10
121         --user             The username with admin privilege over docker engine.
        Default: root
122         --time            Time, in seconds, to wait, after make a Scale action. Default:
        60
123         --cpuonly         Scale based on CPU usage only.
124         --memoryonly      Scale based on Memory usage only."
125         # cat $PARAMETERSLOG | grep -wv "$@" >
        $PARAMETERSLOG
126         sed -i '$ d' $PARAMETERSLOG
127         exit 1
128         ;;
129
130         --name)
131             shift
132             SERVICE=$1
133             ;;
134
135         --user)
136             shift
137             USER=$1
138             ;;
139
140         --time)
141             shift
142             if [[ $1 =~ $NUMBERS ]];then
143                 TIME=$1
144             else
145                 echo "--time invalid!"
146                 # cat $PARAMETERSLOG | grep -wv "$@" >
        $PARAMETERSLOG
147                 sed -i '$ d' $PARAMETERSLOG
148                 exit 1
149             fi
150             ;;
151
152         --mincpu)
153             shift
154             if [[ $1 =~ $NUMBERS ]];then
155                 MINCPU=$1
156             else
157                 echo "mincpu invalid!"
158                 # cat $PARAMETERSLOG | grep -wv "$@" >
        $PARAMETERSLOG
159                 sed -i '$ d' $PARAMETERSLOG
160                 exit 1
161             fi
162             ;;
163
164         --maxcpu)
165             shift
166             if [[ $1 =~ $NUMBERS ]];then

```

```

167             MAXCPU=$1
168         else
169             echo "maxcpu invalid!"
170             # cat $PARAMETERSLOG | grep -wv "$@" >
171             $PARAMETERSLOG
172             sed -i '$ d' $PARAMETERSLOG
173             exit 1
174         fi
175     ;;
176     --minmemory)
177         shift
178         if [[ $1 =~ $NUMBERS ]];then
179             MINMEMORY=$1
180         else
181             echo "time invalid!"
182             # cat $PARAMETERSLOG | grep -wv "$@" >
183             $PARAMETERSLOG
184             sed -i '$ d' $PARAMETERSLOG
185             exit 1
186         fi
187     ;;
188     --maxmemory)
189         shift
190         if [[ $1 =~ $NUMBERS ]];then
191             MAXMEMORY=$1
192         else
193             echo "maxmemmory invalid!"
194             # cat $PARAMETERSLOG | grep -wv "$@" >
195             $PARAMETERSLOG
196             sed -i '$ d' $PARAMETERSLOG
197             exit 1
198         fi
199     ;;
200     --minreplicas)
201         shift
202         if [[ $1 =~ $NUMBERS ]];then
203             MINREPLICAS=$1
204         else
205             echo "minreplicas invalid!"
206             # cat $PARAMETERSLOG | grep -wv "$@" >
207             $PARAMETERSLOG
208             sed -i '$ d' $PARAMETERSLOG
209             exit 1
210         fi
211     ;;
212     --maxreplicas)
213         shift
214         if [[ $1 =~ $NUMBERS ]];then
215             MAXREPLICAS=$1
216         else
217             echo "maxreplicas invalid!"
218             # cat $PARAMETERSLOG | grep -wv "$@" >
219             $PARAMETERSLOG
220             sed -i '$ d' $PARAMETERSLOG
221             exit 1
222         fi

```

```

223             ;;
224
225             --cpuonly)
226                 MEMORYONLY=0
227             ;;
228
229             --memoryonly)
230                 CPUONLY=0
231             ;;
232
233             *)
234                 echo "Invalid argument $1"
235                 # cat $PARAMETERSLOG | grep -wv "$@" >
$PARAMETERSLOG
236                 sed -i '$ d' $PARAMETERSLOG
237                 exit 1
238             ;;
239
240         esac
241         shift
242
243     done
244
245     if [ -z $SERVICE ];then
246         echo "Service not found"
247         # cat $PARAMETERSLOG | grep -wv "$@" > $PARAMETERSLOG
248         sed -i '$ d' $PARAMETERSLOG
249         exit 1          # Finaliza o scrip caso o servico nao seja encontrado
250     fi
251
252     # Verifica se o servico ja esta sendo monitorado
253     SERVTAM=$(cat $SERVPIDLOG | grep -w $SERVICE | wc -l)
254     if [ "$SERVTAM" -ne "0" ];then
255         echo "Service already monitored"
256         # cat $PARAMETERSLOG | grep -wv "$@" > $PARAMETERSLOG
257         sed -i '$ d' $PARAMETERSLOG
258         exit 1
259     fi
260
261     if [ "$MINCPU" -ge "$MAXCPU" ];then          # -ge = maior ou igual
262         echo "mincpu needs to be less than maxcpu"
263         # cat $PARAMETERSLOG | grep -wv "$@" > $PARAMETERSLOG
264         sed -i '$ d' $PARAMETERSLOG
265         exit 1
266     fi
267
268     if [ "$MINMEMORY" -ge "$MAXMEMORY" ];then
269         echo "minmemory needs to be less than maxmemory"
270         # cat $PARAMETERSLOG | grep -wv "$@" > $PARAMETERSLOG
271         sed -i '$ d' $PARAMETERSLOG
272         exit 1
273     fi
274
275     if [ "$MINREPLICAS" -ge "$MAXREPLICAS" ];then
276         echo "minreplicas needs to be less than maxreplicas"
277         # cat $PARAMETERSLOG | grep -wv "$@" > $PARAMETERSLOG
278         sed -i '$ d' $PARAMETERSLOG
279         exit 1
280     fi
281
282     if [ "$CPUONLY" -eq "0" ] && [ "$MEMORYONLY" -eq "0" ];then

```

```

283         echo "--cpuonly and --memoryonly cannot be used together"
284         # cat $PARAMETERSLOG | grep -wv "$@" > $PARAMETERSLOG
285         sed -i '$ d' $PARAMETERSLOG
286         exit 1
287     fi
288
289     # Procura nos servicos pelo servico desejado
290     SERVTAM=$(docker service ls | awk '{print $2}' | grep -w $SERVICE | wc -l)    # wc
-l conta as linhas
291
292     # So deve haver 1 ocorrencia do servico, qualquer numero diferente de 1 ou o
servico nao existe ou o nome esta incorreto
293     if [ "$SERVTAM" -ne "1" ];then          # -ne = not equal
294
295         echo "Service not found"
296         # cat $PARAMETERSLOG | grep -wv "$@" > $PARAMETERSLOG
297         sed -i '$ d' $PARAMETERSLOG
298         exit 1          # Finaliza o scrip caso o servico nao seja encontrado
299     fi
300
301     printf "Monitoring $SERVICE\n"        # printa mensagem de monitoramento
302     # Executa o escalonamento no background
303     scale $SERVICE &          # Roda a funcao em background
304 }
305
306 #####
307 ##### SCALE #####
308 function scale {
309     SERVICE=$1
310
311     # Cria o diretorio do servico caso nao exista
312     if [ ! -d "$LOGDIR/$SERVICE" ]; then
313         mkdir $LOGDIR/$SERVICE
314     fi
315
316     # Coloca o endereco dos logs do servico em variaveis
317     RESFILE=$LOGDIR/$SERVICE/resources.log
318     PROCFILE=$LOGDIR/$SERVICE/proc.log
319     MEMFILE=$LOGDIR/$SERVICE/mem.log
320     NODESFILE=$LOGDIR/$SERVICE/nodes.log
321     IPSFILE=$LOGDIR/$SERVICE/ips.log
322
323     # Cria os arquivos de log do servico
324     touch $RESFILE
325     touch $PROCFILE
326     touch $MEMFILE
327     touch $NODESFILE
328     touch $IPSFILE
329
330     # Contador que controlara a quantidade de replicas, seu valor inicial e igual a
quantidade atual de replicas
331     COUNT="$(docker service ls | grep -w $SERVICE | awk '{print $4}' 2>>$ERRLOG)"
332     COUNT="${COUNT:2:3}"          # Seleciona da variavel count, apos o segundo
caractere os tres caracteres subsequentes (Seleciona o numero total de replicas -/X)
333     PID="$(ps -eo uid,pid,cmd | grep -w auto | grep -w dkscale | grep -w $SERVICE | awk
'{print $2}' | sed '1!d')"        # pega o PID do processo
334     echo "$SERVICE    $PID    $COUNT" 1>> $SERVPIDLOG 2>>$ERRLOG    #
Salva o nome do servico e o PID desse processo
335
336     DID=0 # Variavel que verifica se foi feito algum Scale

```

```

337     while true
338     do
339
340         # Verifica se o servico continua existindo
341         SERVAM=$(docker service ls | awk '{print $2}' | grep -w $SERVICE | wc -l)
# wc -l conta as linhas
342         if [ "$SERVAM" -eq "0" ];then           # -eq = equal
343             cat $SERVPIDLOG | grep -v $SERVICE > $SERVPIDLOG      #
Remove do log o processo finalizado
344             rm -rf $LOGDIR/$SERVICE                               #
Remove o diretorio do servico
345             exit 1
346         fi
347
348         MANAGER=$(docker node ls | grep -w $(hostname) | awk '{print $6}')
349         if [ "$MANAGER" == "Leader" ] # faz o processo se for o manager Leader
350         then
351
352             COUNT=$(awk '{print $3}' $SERVPIDLOG)
353
354             # Verifica e filtra o uso da CPU e Memoria das replicas do servico
que estao rodando no manager e salva no log
355             docker stats --no-stream --format "table {{.Name}}\t{{.CPUPerc}}\
\t{{.MemPerc}}" | grep -w $SERVICE 1> $RESFILE 2>>$ERRLOG
356
357             # Retorna o nome de todos os nodes que o servico esta rodando
exceto este
358             docker service ps $SERVICE --format "table {{.Name}}\t{{.Node}}\
\t{{.DesiredState}}" | sed 's/\\_//g' | sed -n '1!p' | grep 'Running' | grep -w $SERVICE | awk
'{print $2}' | grep -v $(hostname) 1> $NODESFILE 2>>$ERRLOG
359
360             if [ -s $NODESFILE ];then
361                 # Pega o endereco ip de todos os nodes que o servico esta
rodando
362                 while read -r LINE
363                 do
364                     NODEIP=$(docker node inspect $LINE --format '{{
.Status.Addr }}' 2>>$ERRLOG)
365                     sed -i "s/$LINE/$NODEIP/g" $NODESFILE
366                     done < "$NODESFILE"
367
368                     # Remove linhas duplicadas, caso tenha mais de uma replica
em um mesmo nó
369                     awk '!a[$0]++' $NODESFILE > $IPFILE
370
371                     # Conecta nos nodes para pegar os dados do servico
372                     while read -r LINE
373                     do
374                         CON=$USER@$LINE
375                         ssh -n $CON 'docker stats --no-stream --format "table
{{.Name}}\t{{.CPUPerc}}\t{{.MemPerc}}"' | grep -w $SERVICE 1>> $RESFILE 2>>$ERRLOG
376                         done < "$IPFILE"
377                     fi
378
379                     awk '{print $2}' $RESFILE > $PROCFILE          # Grava a segunda
coluna do resfile em procfle
380                     sed -i 's/%/' $PROCFILE                       # Remove o simbolo '%'
381                     awk '{print $3}' $RESFILE > $MEMFILE          # Grava a terceira coluna do
resfile em memfile
382                     sed -i 's/%/' $MEMFILE                         # Remove o simbolo '%'
383

```

```

384          LINES=$(wc -l < $PROCFILE) # Conta o numero de linhas no
arquivo, que sao igual ao numero de replicas
385
386          # Tira a media aritimetica do uso de CPU de todas as replicas do
servico
387          PROC=0.0
388          while read -r LINE
389          do
390              PROC=$(awk "BEGIN {print $PROC+$LINE; exit}")
391              done < "$PROCFILE"
392          PROC=$(awk "BEGIN {print int($PROC/$LINES); exit}") #
Variavel contendo a media da CPU
393
394          # Defina a porcentagem da media de uso da CPU
395          CPU=$(docker service inspect $SERVICE | grep "NanoCPUs" | tr -d
""NanoCPUs:" | tr -d ',' | head -n1)
396          if [ -z $CPU ];then
397              CPU=100
398          else
399              CPU=$(awk "BEGIN {print $CPU/10000000; exit}")
400          fi
401          CPU=$(awk "BEGIN {print int(($PROC/$CPU)*100); exit}")
402
403          # Tira a media aritimetica do uso de Memoria de todas as replicas do
servico
404          MEM=0.0
405          while read -r LINE
406          do
407              MEM=$(awk "BEGIN {print $MEM+$LINE; exit}")
408              done < "$MEMFILE"
409          MEM=$(awk "BEGIN {print int($MEM/$LINES); exit}") # Variavel
contendo a media da Memoria
410
411          DID=0 # Variavel que verifica se foi feito algum Scale
412          if [ "$CPUONLY" -eq "1" ] && [ "$MEMORYONLY" -eq "1" ];then
413              # Faz o escalonamento baseado na CPU e Memoria
414              # Se o servico esta usando menos de $MINCPU do total de
processamento E menos de $MINMEMORY do total de memoria ele mata uma replica
415          if [ "$CPU" -le "$MINCPU" ] && [ "$MEM" -le
"$MINMEMORY" ] && [ "$COUNT" -gt "$MINREPLICAS" ];then # -le = menor ou igual, -gt =
maior que
416              COUNT=$(awk "BEGIN {print $COUNT-1; exit}")
# Decrescenta o contador
417              SERVSCALE=$SERVICE'=$COUNT
418              docker service scale $SERVSCALE 1> /dev/null 2>>
$ERRLOG
419          echo "$(date +"%Y-%m-%d") at $(date +"%T") Scale
DOWN service $SERVICE to $COUNT" >> $SCALELOG # Salva no log
420          DID=1
421          sleep $TIME
422
423
424          # Se o servico esta usando mais de $MAXCPU do total de
processamento OU mais de $MAXMEMORY do total de memoria ele cria uma nova replica
425          elif ([ "$CPU" -ge "$MAXCPU" ] && [ "$COUNT" -lt
"$MAXREPLICAS" ]) || ([ "$MEM" -ge "$MAXMEMORY" ] && [ "$COUNT" -lt
"$MAXREPLICAS" ]);then # -ge = maior ou igual, -lt = menor que
426              COUNT=$(awk "BEGIN {print $COUNT+1; exit}")
# Incrementa o contador
427              SERVSCALE=$SERVICE'=$COUNT

```

```

428                                     docker service scale $SERVSCALE 1> /dev/null 2>>
$ERRLOG
429                                     echo "$(date +%Y-%m-%d)" at $(date +%T) Scale
UP service $SERVICE to $COUNT" >> $SCALELOG # Salva no log
430                                     DID=1
431                                     sleep $TIME
432                                     fi
433
434                                     elif [ "$CPUONLY" -eq "1" ] && [ "$MEMORYONLY" -eq "0" ];then
435                                     # Faz o escalonamento baseado apenas na CPU
436                                     # Se o servico esta usando menos de $MINCPU do total de
processamento ele mata uma replica
437                                     if [ "$CPU" -le "$MINCPU" ] && [ "$COUNT" -gt
"$MINREPLICAS" ];then # -le = menor ou igual, -gt = maior que
438                                     COUNT=$(awk "BEGIN {print $COUNT-1; exit}")
# Decrescenta o contador
439                                     SERVSCALE=$SERVICE'=$COUNT
440                                     docker service scale $SERVSCALE 1> /dev/null 2>>
$ERRLOG
441                                     echo "$(date +%Y-%m-%d)" at $(date +%T) Scale
DOWN service $SERVICE to $COUNT" >> $SCALELOG # Salva no log
442                                     DID=1
443                                     sleep $TIME
444
445                                     # Se o servico esta usando mais de $MAXCPU do total de
processamento ele cria uma nova replica
446                                     elif [ "$CPU" -ge "$MAXCPU" ] && [ "$COUNT" -lt
"$MAXREPLICAS" ];then # -ge = maior ou igual, -lt = menor que
447                                     COUNT=$(awk "BEGIN {print $COUNT+1; exit}")
# Incrementa o contador
448                                     SERVSCALE=$SERVICE'=$COUNT
449                                     docker service scale $SERVSCALE 1> /dev/null 2>>
$ERRLOG
450                                     echo "$(date +%Y-%m-%d)" at $(date +%T) Scale
UP service $SERVICE to $COUNT" >> $SCALELOG # Salva no log
451                                     DID=1
452                                     sleep $TIME
453                                     fi
454
455                                     elif [ "$CPUONLY" -eq "0" ] && [ "$MEMORYONLY" -eq "1" ];then
456                                     # Faz o escalonamento baseado apenas na Memoria
457                                     # Se o servico esta usando menos de $MINMEMORY do
total de memoria ele mata uma replica
458                                     if [ "$MEM" -le "$MINMEMORY" ] && [ "$COUNT" -gt
"$MINREPLICAS" ];then # -le = menor ou igual, -gt = maior que
459                                     COUNT=$(awk "BEGIN {print $COUNT-1; exit}")
# Decrescenta o contador
460                                     SERVSCALE=$SERVICE'=$COUNT
461                                     docker service scale $SERVSCALE 1> /dev/null 2>>
$ERRLOG
462                                     echo "$(date +%Y-%m-%d)" at $(date +%T) Scale
DOWN service $SERVICE to $COUNT" >> $SCALELOG # Salva no log
463                                     DID=1
464                                     sleep $TIME
465
466                                     # Se o servico esta usando mais de $MAXMEMORY do total
de memoria ele cria uma nova replica
467                                     elif [ "$MEM" -ge "$MAXMEMORY" ] && [ "$COUNT" -lt
"$MAXREPLICAS" ];then # -ge = maior ou igual, -lt = menor que

```

```

469                                     COUNT=$(awk "BEGIN {print $COUNT+1; exit}")
# Incrementa o contador
470                                     SERVSCALE=$SERVICE'=$COUNT
471                                     docker service scale $SERVSCALE 1> /dev/null 2>>
$ERRLOG
472                                     echo "$(date +"%Y-%m-%d") at $(date +"%T") Scale
UP service $SERVICE to $COUNT" >> $SCALELOG # Salva no log
473                                     DID=1
474                                     sleep $TIME
475                                     fi
476                                     fi
477                                     fi
478
479                                     # Se não fez nenhum escalonamento
480                                     if [ "$DID" -eq "0" ];then
481                                         sleep 5
482                                     else
483                                         # Atualiza o valor do contador no arquivo
484                                         sed -i "s/$SERVICE $PID .*/$SERVICE $PID $COUNT/g"
$SERVPIDLOG
485                                     fi
486
487                                     done
488 }

```