UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Carla de Oliveira Barden

CONFIGURAÇÃO DE UM AMBIENTE ESCALÁVEL USANDO DOCKER SWARM PARA APLICAÇÕES DE BIG DATA

Santa Maria, RS
2021

Carla de Oliveira Barden

CONFIGURAÇÃO DE UM AMBIENTE ESCALÁVEL USANDO DOCKER SWARM PARA APLICAÇÕES DE BIG DATA

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação, da Universidade Federal de Santa Maria (UFSM) - Campus Santa Maria, como requisito parcial para obtenção do grau de **Bacharel em Ciência da Computação**.

ORIENTADOR: Prof. João Vicente Ferreira Lima

474

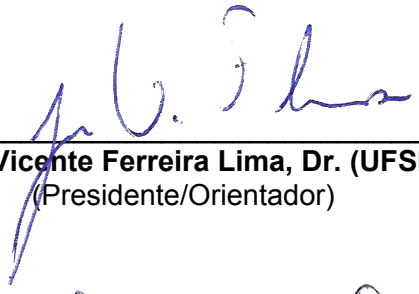
Santa Maria, RS
2021

CARLA DE OLIVEIRA BARDEN

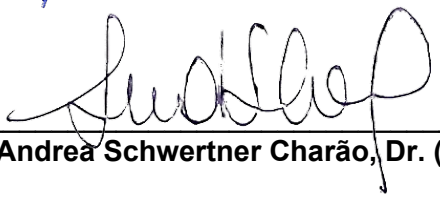
CONFIGURAÇÃO DE UM AMBIENTE ESCALÁVEL USANDO DOCKER SWARM PARA APLICAÇÕES DE BIG DATA

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Bacharel em Ciência da Computação**.

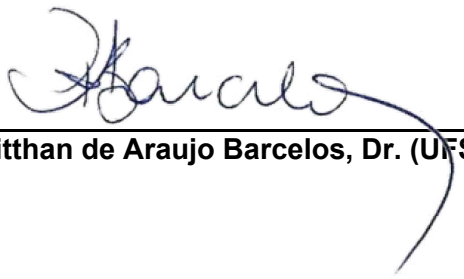
Aprovado em 10 de Fevereiro de 2021:



João Vicente Ferreira Lima, Dr. (UFSM)
(Presidente/Orientador)



Andrea Schwertner Charão, Dr. (UFSM)



Patrícia Pitthan de Araujo Barcelos, Dr. (UFSM)

Santa Maria, RS
2021

RESUMO

CONFIGURAÇÃO DE UM AMBIENTE ESCALÁVEL USANDO DOCKER SWARM PARA APLICAÇÕES DE BIG DATA

AUTORA: Carla de Oliveira Barden
ORIENTADOR: João Vicente Ferreira Lima

A proposta deste trabalho foi apresentar um ambiente de nuvem privada, virtualizada a nível de sistema operacional, para provisionar um *framework* de *Big Data*. Para isso, foi utilizado o Docker em conjunto com o Docker Swarm e o Spark. Foram executadas três aplicações de teste, sendo elas: *word count*, *sort* e cálculo do número PI, medindo-se o tempo de execução enquanto manteve-se o tamanho da entrada e aumentou-se o número de contêineres executando o Spark. Apesar de o ambiente apresentar elasticidade e ser facilmente reproduzível, para um mesmo tamanho de entrada, os experimentos demonstraram o aumento do tempo de execução das aplicações conforme o incremento do número de contêineres. Ainda assim, demonstrou-se a viabilidade da configuração proposta, que pode apresentar vantagens dependendo do contexto.

Palavras-chave: Virtualização Leve, *Big Data*, Computação em Nuvem

ABSTRACT

A SCALABLE ENVIRONMENT WITH DOCKER SWARM FOR BIG DATA APPLICATIONS

AUTHOR: Carla de Oliveira Barden
ADVISOR: João Vicente Ferreira Lima

This work presents a private cloud environment virtualized at the operating system level for a big data framework. We use Docker with Docker Swarm and Spark for running three application tests: word count, sort, and the PI calculus. Our measures considered the execution time with the same input size and increasing the number of containers using Spark. Our results show the impact of increasing the number of Docker containers on the application's performance for the same input file size.

Keywords: Lightweight Virtualization, Big Data, Cloud Computing

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CaaS	Container as a Service
CPU	Central Process Unit
FIFO	First In First Out
HDFS	Hadoop Distributed File System
IaaS	Infrastructure as a Service
KVM	Kernel-based Virtual Machine
LXC	Linux Containers
MPI	Message Passing Interface
OpenMP	Open Multi-Processing
PaaS	Plataform as a Service
RDD	Resilient Distributed Datasets
REST	Representational State Transfer
SaaS	Software as a Service
SQL	Standard Query Language
VM	Virtual Machine
XML	Extensible Markup Language
YARN	Yet Another Resource Negotiator
YML	Yaml Ain’t Markup Language

SUMÁRIO

1 INTRODUÇÃO

A aplicação massiva da computação na resolução de problemas, automatização de tarefas, e na criação de novos produtos e serviços nas mais diversas áreas teve como uma de suas consequências o crescimento em escala exponencial da geração de dados. O processamento e a análise desses grandes conjuntos de dados podem fornecer informações e descobertas relevantes em muitos contextos, sejam eles comerciais, científicos, militares ou cotidianos (KUNE et al., 2016). Há diversas ferramentas e tecnologias disponíveis para realizar operações com dados em larga escala, porém, elas geralmente demandam esforços consideráveis para serem configuradas corretamente em ambiente de produção e este ambiente não é facilmente replicado.

Além disso, o armazenamento, o processamento e a análise desses dados requerem uma robusta infraestrutura computacional distribuída, de alto custo de configuração e manutenção, conforme aumentam-se a quantidade de dados usados e a complexidade das operações a serem efetuadas. Idealmente, essa infraestrutura deve ser altamente escalável, a fim de responder adequadamente ao rápido aumento de sua demanda. Outro atributo desejável é a baixa necessidade de manutenção, de forma a ter toda a configuração do ambiente da maneira menos complexa possível, já que quanto mais complexo um sistema for, maior é a probabilidade de problemas ocorrerem (MARZ; WARREN, 2015).

Assim, o uso de computação em nuvem apresenta-se como uma boa alternativa para prover esse tipo de infraestrutura, já que fornece recursos sob demanda com custos proporcionais ao uso real (elasticidade) e permite que a infraestrutura seja ampliada rapidamente (escalabilidade), adaptando o sistema à demanda atual (ASSUNÇÃO et al., 2015).

Tendo em vista esses critérios, a proposta deste trabalho foi apresentar um ambiente de nuvem privada, com elasticidade e de fácil manutenção, para provisionar um *framework* de *Big Data*. Para isso, foi utilizado Docker/Docker Swarm para o gerenciamento de *cluster* e o *framework* Spark para o processamento de *Big Data*. A fim de verificar a viabilidade dessa proposta, foram executadas as aplicações *word count*, *sort* e cálculo do número PI, medindo-se o tempo gasto em cada uma delas conforme altera-se a quantidade de contêineres na execução de uma mesma carga de trabalho, e averiguando se o aumento no número de contêineres reflete positivamente no tempo de execução.

O restante deste trabalho está organizado em capítulos. No capítulo 2, é feita a revisão do estado da arte e o embasamento teórico da proposta. No capítulo 3, é apresentada a arquitetura implementada, as ferramentas utilizadas, uma discussão acerca do que é proposto, a maneira com que foi realizada a execução dos testes e os resultados obtidos. Por fim, no capítulo 4, são realizadas algumas considerações finais.

2 REFERENCIAL TEÓRICO

Com o intuito de embasar os conceitos que serão utilizados ao longo deste trabalho, foram feitas pesquisas sobre Virtualização, Computação em Nuvem e uma revisão do estado da arte do processamento de dados em larga escala (*Big Data*). Neste capítulo, são descritos os principais tipos de Virtualização e as classificações mais relevantes da Computação em Nuvem. Também é abordada uma das definições de *Big Data* e o modelo de programação mais usado para este fim - o *MapReduce*. Ao final, também são apresentados alguns trabalhos relacionados.

2.1 VIRTUALIZAÇÃO

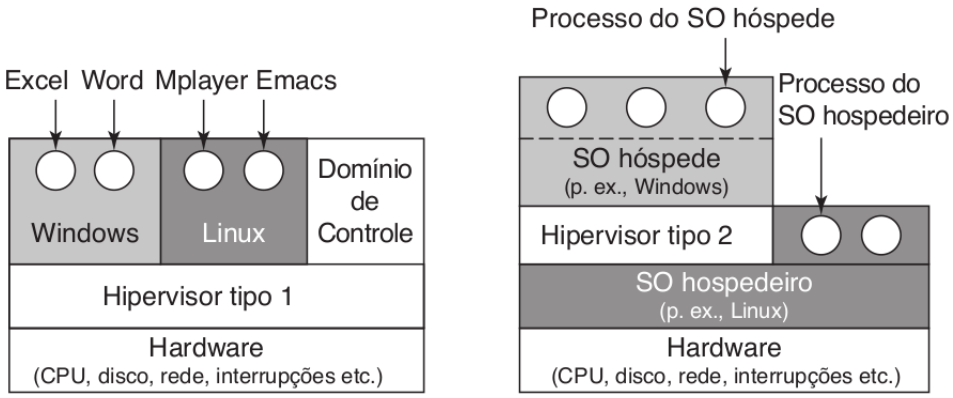
A virtualização, segundo Tanenbaum e Bos (2015), permite que um único computador seja o hospedeiro de múltiplas máquinas virtuais (*Virtual Machines* - VMs). Geralmente, com a máquina virtual, aplicações e sistemas operacionais convidados são executados em um ambiente que, para eles, parece ser hardware nativo e que se comporta em relação a eles como hardware nativo, mas que, além disso, os protege, gerencia e limita (SILBERSCHATZ; GAGNE, 2015). No entanto, nem toda tecnologia de virtualização tenta fazer com que o convidado acredite que possui o sistema todo. Algumas vezes, o objetivo é permitir que um processo execute mesmo que ele tenha sido escrito para um sistema operacional e/ou arquitetura diferentes (TANENBAUM; BOS, 2015).

A virtualização tradicional é baseada em **hipervisores**, responsáveis por criar, gerenciar e executar máquinas virtuais. Existem dois tipos de hipervisores: **Tipo 1**, conforme exibe a Figura 2.2(a), e **Tipo 2**, conforme exibe a Figura 2.2(b). O hipervisor de tipo 1, tecnicamente, é como um sistema operacional, já que ele executa diretamente sobre o hardware (TANENBAUM; BOS, 2015). O hipervisor de tipo 2, por sua vez, executa como um processo regular em um sistema operacional hospedeiro (SILBERSCHATZ; GAGNE, 2015). Exemplos de hipervisores tipo 1 são o KVM, o Microsoft Hyper-V e o VMware vSphere, enquanto, de tipo 2, pode-se citar o VMware Workstation e o Oracle VirtualBox.

Já a virtualização leve (ou virtualização a nível de sistema operacional) fornece uma maneira de isolar processos e possibilita uma forma fácil de iniciar, interromper, mover e gerenciar aplicações (SILBERSCHATZ; GAGNE, 2015). Nessa situação, há apenas um *kernel* e é criada uma camada virtual (contêiner) entre ele e as aplicações. Conforme o objetivo e a implementação, pode-se classificar softwares de contêiner em duas categorias: contêineres de **Aplicação** e contêineres de **Sistema**.

Os contêineres de sistema são feitos para fornecer um sistema operacional completamente funcional, assim como máquinas virtuais tradicionais, porém, implementados de maneira mais leve e de forma a prover infraestrutura em camadas. Já os contêineres de aplicação são pro-

Figura 2.1 – Hipervisores de Tipo 1 e de Tipo 2



(a) Tipo 1 (b) Tipo 2

Fonte: Tanenbaum e Bos (2015)

jetados para encapsular uma única tarefa (microsserviço) em uma imagem, a fim de distribuir aplicativos de forma eficiente. A execução desse tipo de contêiner é interrompida imediatamente após a conclusão de sua tarefa e a sua imagem usa um sistema de arquivos em camadas, permitindo o reuso destas camadas por outras imagens (RUAN et al., 2016). Exemplos de contêineres de sistema são o LXC e o OpenVZ; já, exemplos de contêineres de aplicação são o Rocket e o Docker.

2.2 COMPUTAÇÃO EM NUVEM

A computação em nuvem é um serviço que distribui recursos computacionais e até mesmo aplicativos via rede, usando como base a virtualização. Assim, o usuário efetua o pagamento desses serviços sob demanda (SILBERSCHATZ; GAGNE, 2015). Conforme a privacidade, os ambientes de computação em nuvem podem ser classificados em:

- **Nuvem pública**, disponível a quaisquer pessoas/empresas dispostas a pagar pelo serviço;
- **Nuvem privada**, operada por uma organização para uso interno;
- **Nuvem híbrida**, que possui atributos de nuvem pública e de nuvem privada.

Considerando o tipo de serviço oferecido, as principais classificações são:

- **IaaS** - Infraestrutura como Serviço, fornecendo servidores ou espaço de armazenamento via internet;
- **CaaS** - Contêiner como Serviço, fornecendo contêineres via internet (RUAN et al., 2016);

- **PaaS** - Plataforma como Serviço, apresentando uma pilha de software pronta para uso via internet;
- **SaaS** - Software como Serviço, oferecendo aplicativos para uso via internet.

Atualmente, diversas empresas oferecem serviços em nuvem. Alguns exemplos conhecidos são a Amazon, Digital Ocean, Google, HP, IBM e Microsoft. Todas provisionam infraestrutura como serviço, enquanto Google e Microsoft também oferecem softwares como serviço. Também vale mencionar a Cloudera, que oferece uma plataforma de software para engenharia, armazenamento e análise de dados, e aprendizado de máquina, sendo possível usá-la localmente ou em nuvem.

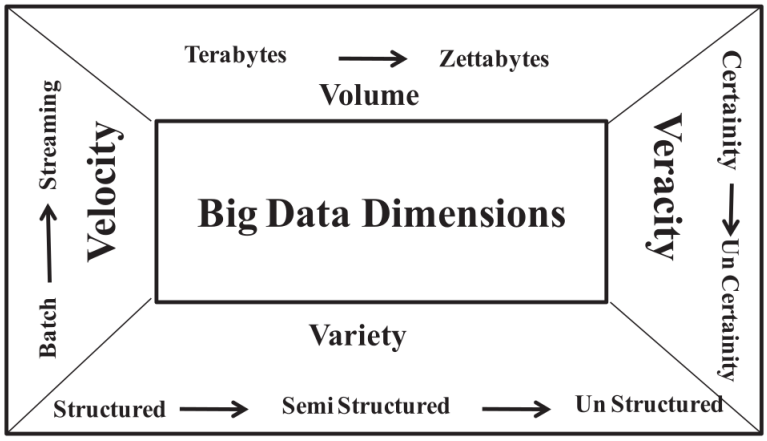
2.3 *BIG DATA*

Big Data, usualmente, é compreendido como grandes conjuntos de dados, estruturados ou não, provenientes de diversas fontes. Geralmente, é descrito pelo *Four V's Model* (KUNE et al., 2016), que define quatro dimensões para caracterizar *Big Data*, sendo elas:

- **Variedade**: refere-se à diversidade de formatos dos dados. Eles podem ser classificados em Estruturados, Não Estruturados, Semi-Estruturados e Mistos.
- **Velocidade**: os dados são gerados, armazenados e analisados em uma velocidade cada vez maior. Os dados podem ser processados em Lotes, em Fluxo, em Tempo Real ou em Semi Tempo Real.
- **Volume**: relacionado à quantia cada vez maior de dados que é gerada;
- **Veracidade**: relacionada à autenticidade e à confiabilidade dos dados;

Além dessas, a dimensão de **Valor** também pode ser utilizada para caracterizar a qualidade dos dados (KUNE et al., 2016), ou seja, o quanto é possível extrair destes dados informações úteis ao seu contexto. Cada uma destas dimensões, expostas na Figura 2.2, apresenta suas características próprias, sendo que essa caracterização do *Big Data* auxilia na tomada de decisões de projeto e de ferramentas a serem utilizadas.

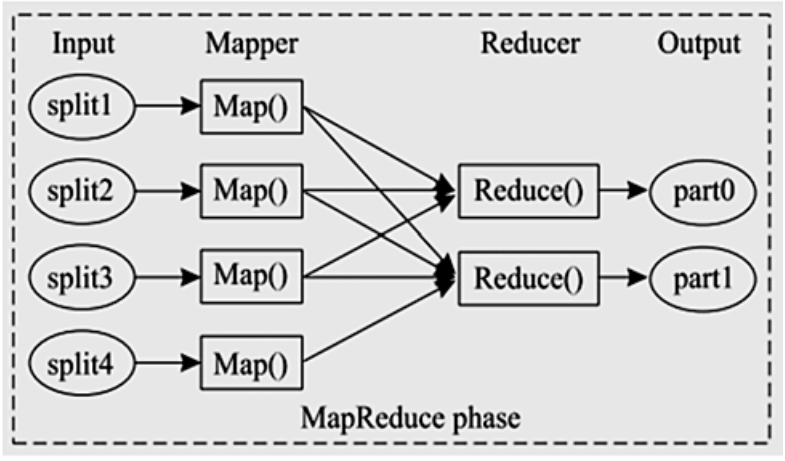
Figura 2.2 – As quatro dimensões do *Big Data*



Fonte: Adaptado de Kune et al. (2016)

Levando em conta essas dimensões, um dos modelos de programação mais populares para o processamento de *Big Data* é o *MapReduce*. O *MapReduce* consiste em um modelo de programação projetado para operar com grandes quantidades de dados, assim como demonstra a Figura 2.3. Baseado nas operações de mapear (filtrar e classificar) e reduzir (agregar os dados filtrados anteriormente) presentes no paradigma de programação funcional, opera de modo paralelo e recursivo em vários conjuntos de dados distribuídos (KUNE et al., 2016).

Figura 2.3 – A arquitetura *MapReduce*



Fonte: Adaptado de Salam et al. (2019)

O *MapReduce* faz uso da arquitetura mestre-trabalhador, sendo que o mestre é responsável por enviar as tarefas aos trabalhadores e por armazenar os seus estados. Também, após a execução das tarefas do tipo *map*, o mestre armazena a localização dos dados intermediários a fim de transmití-la para as tarefas do tipo *reduce*. Já os trabalhadores são responsáveis pela execução das tarefas em si e pelo armazenamento do seu resultado em disco.

Kune et al. (2016) definem várias áreas onde é possível a aplicação de processamento de *Big Data*, já que uma gama gigantesca de dados de sensores, de dispositivos de segurança, de sites comerciais e, ainda, dados de redes sociais, são gerados todos os dias. Dentre essas áreas, estão a exploração científica, a área de saúde, genética, questões urbanas (como gestão de trânsito e segurança), análises financeiras, análise web e internet das coisas.

2.4 TRABALHOS RELACIONADOS

Em (ENES et al., 2018), é proposta uma plataforma como serviço (PaaS) para a execução dos *frameworks* Hadoop e Spark baseada em contêineres Docker em um *cluster* gerenciado pelo Mesos. O desempenho dessa configuração foi comparado com os mesmos *frameworks* implantados em VMs no OpenStack, por meio da ferramenta Big Data Evaluator. Aqui, há a comparação entre dois tipos de virtualização (tradicional - OpenStack e virtualização a nível de sistema operacional), com o *cluster* (e os microsserviços) sendo gerenciado pelo Mesos - diferentemente do que é proposto neste trabalho, com o *cluster* gerenciado pelo próprio Docker Swarm. Já em (Ceesay; Barker; Varghese, 2017), é apresentado um ambiente baseado em Docker para automatizar a configuração e execução do Hibench, partindo do pressuposto de que já há um *cluster* com o Hadoop e Spark configurados em *bare metal*. Também foram executadas e calculados os custos de diversas cargas de trabalho do Hibench, para avaliar o desempenho da arquitetura adotada. Neste caso, também não é usado o Docker Swarm, e apenas o Hibench está sendo executado em contêineres, como microsserviço, sendo que as configurações necessárias para que ele funcione adequadamente no *cluster* preexistente foram colocadas no *dockerfile* que gera a sua imagem.

Em (Bhimani et al., 2017), compara-se o desempenho de aplicações típicas do Spark usando virtualização tradicional e contêineres. Foram usados um nó mestre e quatro nós trabalhadores em ambas as situações, e, para testes, executaram-se *benchmarks* de aprendizado de máquina, computação em grafos e consultas SQL, a fim de comparar diversas métricas, dentre elas tempo de execução, uso de CPU, disco, e memória principal. Neste trabalho, assim como em (ENES et al., 2018), também é feita a comparação entre a virtualização tradicional e a virtualização em nível de sistema operacional. No entanto, apesar de o Spark estar configurado como microsserviço, o número de nós trabalhadores não é escalável em tempo de execução porque os contêineres trabalhadores foram configurados estaticamente em um arquivo YML (indicando que a pilha de microsserviços foi implementada usando-se o Docker Compose). Ao usar o Docker Swarm, como é proposto neste trabalho, é possível replicar um microsserviço mesmo que a sua pilha já esteja em execução, de forma dinâmica.

Lei et al. (2016), Blair, Olmsted e Anderson (2017) e Zeng et al. (2017) utilizam o Docker em conjunto com o Spark, sendo que em Lei et al. (2016) é proposta uma ferramenta para gerenciamento em *cluster* do Spark em contêineres Docker (novamente, não foi usado o

Docker Swarm, como é proposto aqui). Em Blair, Olmsted e Anderson (2017), o desempenho do Spark virtualizado utilizando-se Docker e KVM é comparado. Já em Zeng et al. (2017), o Spark é utilizado como microsserviço implementado no Docker, com o *cluster* gerenciado pelo Docker Swarm. No entanto, essa pilha foi construída para uma aplicação de análise de roteamento de rede, e não para averiguar tempo de execução das tarefas do Spark.

Ahmed et al. (2016) e Reyes-Ortiz, Oneto e Anguita (2015) usam o Spark em ambiente de nuvem. O primeiro, no OpenStack, usando o YARN como gerenciador de cluster, HDFS para armazenamento distribuído e usando o *benchmark* Hibench para testar diversas métricas. O último, no Google Cloud Plataform, comparando o Spark sobre a pilha do Hadoop (HDFS e YARN) e o MPI/OpenMP no Beowulf. Apesar de ambos os trabalhos utilizarem o Spark em ambiente de nuvem, nenhum deles usa Docker para a virtualização. Por fim, Pahl et al. (2016) e Nguyen e Bein (2017) usam o Docker em *cluster*. Pahl et al. (2016) usam o Kubernetes como gerenciador de *cluster*, para demonstrar a possibilidade de configurar uma infraestrutura PaaS em um *cluster* formado por Raspberries PI, com foco em Internet das Coisas. Enquanto isso, Nguyen e Bein (2017) usam o Docker Swarm para implementar um *cluster* MPI. Ambas os trabalhos utilizam Docker em *cluster*, porém, nenhuma delas usa o Spark para processamento. As ferramentas mais relevantes utilizadas pelos trabalhos apresentados são comparadas no Quadro 2.1.

Quadro 2.1 – Comparação das ferramentas utilizadas nos Trabalhos Relacionados

Autores	**Plataformas**	**Configuração Adotada**
Enes et al. (2018)	Hadoop e Spark	Contêineres Docker em *cluster* Mesos *versus* máquinas virtuais no OpenStack
Ceesay, Barker e Varghese (2017)	Hadoop e Spark	Plataformas em *bare metal* Hibench em contêineres Docker
Bhimani et al. (2017)	Spark	VMware Workstation *versus* contêineres Docker
Lei et al. (2016)	Spark	Docker e gerenciamento de *cluster* proposto no trabalho (DCSpark)
Blair, Olmsted e Anderson (2017)	Spark	KVM *versus* contêineres Docker
Zeng et al. (2017)	Spark	Docker e Docker Swarm
Ahmed et al. (2016)	Spark	Configuração no OpenStack, com YARN e HDFS
Reyes-Ortiz, Oneto e Anguita (2015)	Spark, MPI e OpenMP	Google Cloud Plataform, com YARN e HDFS, *versus* MPI/OpenMP no Beowulf
Pahl et al. (2016)	-	Contêineres Docker em *cluster* Kubernetes
Nguyen e Bein (2017)	MPI	MPI em contêineres Docker em *cluster* Docker Swarm
Trabalho Proposto	Spark	Spark como microsserviço no Docker em *cluster* Docker Swarm

3 METODOLOGIA

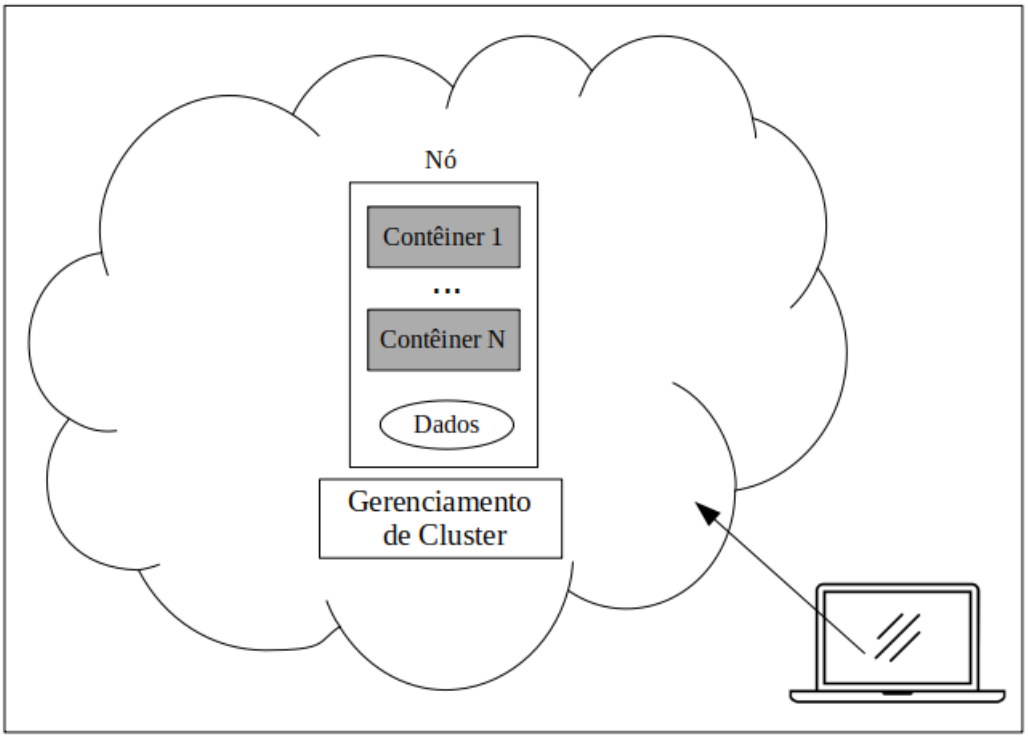
Neste capítulo, apresenta-se a arquitetura implementada e descreve-se as ferramentas usadas na elaboração do ambiente proposto. As aplicações utilizadas para testar a implementação são descritas e, também, especifica-se o critério de avaliação escolhido (tempo de execução), assim como a forma de obtê-lo.

3.1 ARQUITETURA PROPOSTA

Como dito anteriormente, o processamento de grandes quantias de dados exige uma grande e complexa infraestrutura. Esta, preferencialmente, deve ser escalável tanto verticalmente (respondendo proporcionalmente à melhoria de recursos como processamento e memória) quanto horizontalmente (tornando fácil a agregação e gerenciamento de nós de trabalho). Considerando um contexto de computação em nuvem provido por virtualização em nível de sistema operacional, outro atributo importante é a elasticidade, ou seja, a capacidade que a infraestrutura possui de adaptar-se à carga de trabalho de forma dinâmica. Pela própria natureza do *Big Data* - dados provenientes de diversas fontes e de formatos variados - as ferramentas e tecnologias usadas para processar e armazenar esses dados não são facilmente configuradas e/ou replicadas em ambiente de produção.

Assim, este trabalho implementa uma configuração de plataforma para o processamento de dados, em ambiente de nuvem privada, usando virtualização leve para prover elasticidade. Conforme a Figura 3.1, há um cliente que acessa aos recursos dispostos na nuvem privada. No caso deste trabalho, esta rede é local. Se necessário, é possível aumentar ou diminuir o número de contêineres trabalhadores, devido ao uso do Docker Swarm para o gerenciamento de *cluster*.

Figura 3.1 – A arquitetura proposta.



3.2 FERRAMENTAS UTILIZADAS

Para o provisionamento da estrutura da nuvem e gerenciamento do *cluster* optou-se pela plataforma **Docker** e o seu gerenciador de *cluster* nativo, Docker Swarm. O Docker[1] é uma plataforma de código aberto que provisiona a criação e gerenciamento de contêineres de aplicação. Ele é desenvolvido com base na arquitetura cliente-servidor, sendo composto por um processo servidor *dockerd*, que cria objetos Docker, como imagens, contêineres, redes e volumes; uma API REST, que define interfaces que instruem o servidor sobre as ações a serem tomadas; e um cliente em linha de comando, que usa a API REST para controlar o servidor.

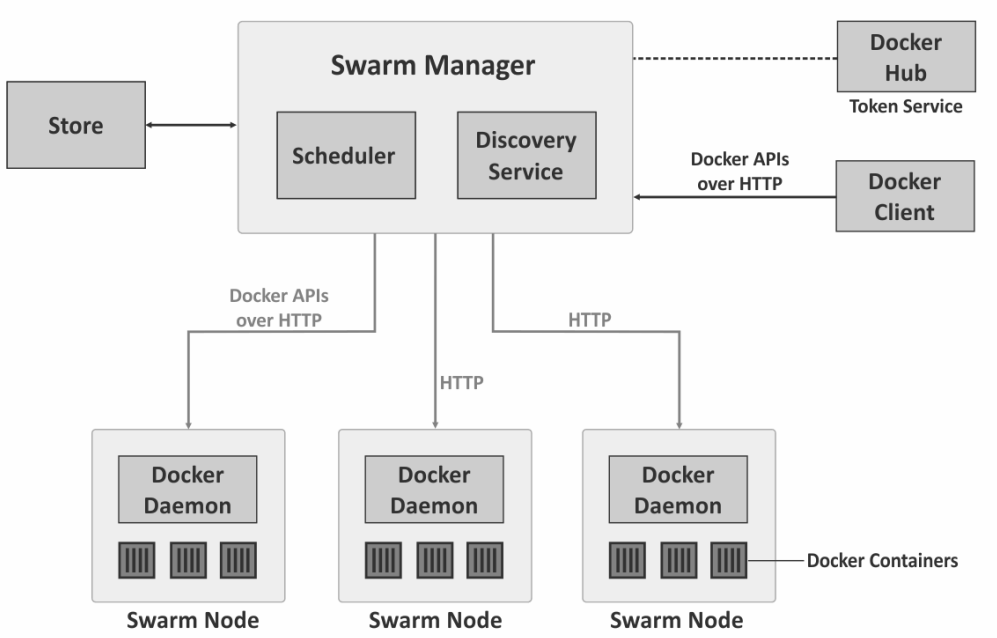
Internamente, o Docker faz uso de diversos recursos do *kernel* do Linux para fornecer sua funcionalidade, como os *namespaces*, para fornecer isolamento; o *cgroups*, que gerencia e pode limitar o acesso a recursos de hardware pelos contêineres; e o *Union File Systems* e seus variantes para o fornecimento de um sistema de arquivos em camadas leve e rápido.

O Docker também possui outras ferramentas, como o Docker Compose e o Docker Swarm. Pensando em termos de microsserviços (cada contêiner executando a menor tarefa possível), o Docker Compose permite que múltiplos contêineres sejam configurados para trabalhar em conjunto e executar aplicações ou serviços, localmente. Já o Docker Swarm, cuja arquitetura

[1] https://docs.docker.com/

é exibida na Figura 3.2, é o gerenciador de *cluster* nativo do Docker e permite que os contêineres executem serviços independente do nó onde estejam, de maneira transparente. O Docker Swarm fornece balanceamento de carga automático e possibilita a replicação dos serviços para aumentar a sua disponibilidade.

Figura 3.2 – O funcionamento do Docker Swarm



Fonte: Adaptado de (YU, 2020).

O *framework* de processamento de dados em larga escala escolhido para este trabalho foi o Apache Spark[2], devido a seus inúmeros casos de uso, vasta documentação e facilidade de uso e implementação. Esta ferramenta é capaz de processar grandes conjuntos de dados, em lotes ou em fluxo, de forma paralela e distribuída, utilizando o modelo de programação *MapReduce* (assim como o Hadoop) e estendendo a sua funcionalidade através de operações, como *filter, collect,* entre diversas outras. Além disso, apresenta melhor desempenho que o *MapReduce* do Hadoop, por armazenar os dados intermediários majoritariamente na memória principal, por meio de RDDs (*Resilient Distributed Datasets*), que são abstrações de objetos distribuídos no *cluster* e onde as operações são executadas.

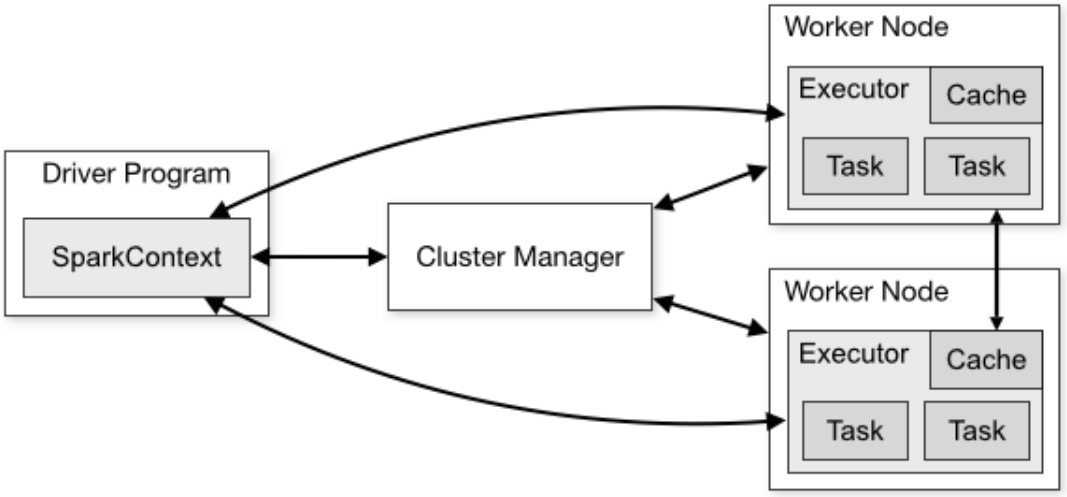
É relativamente comum encontrar o Spark configurado em um *cluster* de forma conjunta com o Hadoop, já que ambos podem compartilhar da mesma infraestrutura de gerenciamento de *cluster* e de armazenamento distribuído (HDFS). Contudo, apesar de parecidos, ambos possuem funcionalidades distintas: basicamente, o Hadoop é composto pelo HDFS, YARN e por uma implementação do *MapReduce*, que armazena os dados intermediários em disco. Já o Spark é um *framework* que carece de armazenamento distribuído. Contudo, a sua versão do *MapReduce* usa o conceito de RDDs, além de possuir diversas outras bibliotecas e funções para

[2]https://spark.apache.org/

o processamento de dados. O Spark suporta o desenvolvimento de programas nas linguagens Scala, Java, Python e R, e fornece APIs para as três primeiras. Ele também dispõe de bibliotecas para processamento de dados de fluxo em tempo real (Spark Streaming), para consultas SQL em dados estruturados (Spark SQL), para aprendizado de máquina (Spark MLib) e para o processamento e análise de grafos (Graph X).

A arquitetura de uma aplicação Spark (Figura 3.3) é composta pelo *driver program*, pelo gerenciador de *cluster* e pelos nós trabalhadores. O *driver program* é o software feito pelo desenvolvedor, onde o *spark context* é invocado. O gerenciador de *cluster* pode ser tanto um gerenciador externo, como o Mesos ou o YARN, quanto o gerenciador de *cluster* do próprio Spark (modo autônomo). Uma vez que o *spark context* é invocado, ele aloca executores em nós trabalhadores para que as tarefas enviadas pelo *driver program* sejam realizadas.

Figura 3.3 – A arquitetura de uma aplicação Spark



Fonte: Adaptado da documentação do Spark (APACHE, 2020)

Em um *cluster* configurado no modo autônomo, há duas formas de enviar aplicações para execução: modo *cluster* e modo cliente. No modo cliente, o *driver program* é iniciado no mesmo processo em que o cliente envia o aplicativo, e isso se aplica quando o computador que invoca o *spark context* está na mesma rede que o *cluster*. Já no modo *cluster*, o *driver program* é iniciado a partir de um dos processos trabalhadores dentro do *cluster*, e o processo que efetua o envio do *driver program* termina assim que este envio é concluído. Este modo deve ser utilizado quando o computador onde está o *driver program* é parte de outra rede, para minimizar a latência da comunicação entre o *driver program* e os executores. É importante ressaltar que o envio de aplicações para execução em um *cluster* Spark é independente do gerenciador de *cluster* usado. Considerando um *cluster* configurado de maneira autônoma, o gerenciador de *cluster* do Spark efetua o escalonamento das tarefas usando FIFO por padrão.

3.3 DISCUSSÃO

Para a configuração de um *cluster* autônomo, basta ter o arquivo binário do Spark em cada nó, executar o *script* do mestre em um deles e o *script* do trabalhador nos demais, indicando o endereço de rede do nó mestre. Esta será a opção adotada neste trabalho, já que facilita a configuração do *cluster* em contêineres por possibilitar que a mesma imagem seja usada por nós mestres e trabalhadores, sendo necessário apenas definir os serviços de cada um. Dessa forma, o Docker Swarm gerencia apenas em que nó cada contêiner executa (balanceamento de carga), reiniciando ou realocando o serviço em outro nó ou contêiner em caso de falha.

No entanto, essa configuração carece de armazenamento distribuído. Por conta disso, é necessário haver uma cópia dos dados usados pelo Spark em cada nó do *cluster*, em locais de caminho equivalente. De certa maneira, isso pode ser considerado positivo, visto que a tarefa será finalizada mesmo que algum nó trabalhador falhe. Assim, o único ponto de falha é o nó mestre, por ele ser único. Entretanto, o aproveitamento dos recursos de armazenamento do *cluster* é péssimo devido à redundância excessiva gerada pelo alto número de réplicas dos dados.

A alternativa mais comum para solucionar esse problema seria utilizar o HDFS (sistema de arquivos distribuído do Hadoop) como ferramenta de armazenamento do *cluster*. Dessa forma, o Spark teria acesso aos dados necessários de forma transparente, independente de onde eles estivessem localizados. Contudo, não foi encontrada uma forma de configurar o HDFS sem o uso do YARN, que, assim como o Docker Swarm, é um gerenciador de *cluster*. Apesar do YARN suportar contêineres para o isolamento de aplicações, não foi identificada uma maneira satisfatória para configurar o próprio Spark em contêineres. Logo, o uso do YARN (e do HDFS) acarretaria no desvio da proposta principal deste trabalho.

Assim, optou-se, neste trabalho, por testar a viabilidade da execução do Spark como serviço em um ambiente Docker Swarm, replicando-se, em caso de necessidade, os dados em cada nó por meio de volumes do Docker. Isso posto, verificou-se o tempo de execução das aplicações *word count*, *sort* e cálculo do número PI, a fim de investigar de que maneira esse tempo varia conforme contêineres trabalhadores do Spark são acrescentados na pilha do Docker Swarm e se esse crescimento no número de trabalhadores implica em uma diminuição do tempo de execução da aplicação.

3.4 EXECUÇÃO

Para configurar o ambiente do *cluster* utilizado neste trabalho [3] e os microsserviços necessários, foi criado um *dockerfile* que gera uma imagem contendo o Spark. Essa imagem serve tanto para o microsserviço de mestre quanto para o de trabalhador. Os microsserviços (mestre

[3] https://github.com/carlabarden/tcc

e trabalhador) e as variáveis de ambiente necessárias, assim como a rede a ser utilizada foram definidos em um *compose file*. Através desse *compose file*, é possível iniciar a pilha de micros-serviços, assim como, pelo uso do Docker Swarm, é possível iniciar, de forma dinâmica, cada microsserviço em um contêiner, provendo elasticidade à infraestrutura. Também, graças ao Swarm, é possível replicar um ou vários determinados microsserviços, de forma transparente, gerando escalabilidade e fornecendo balanceamento de carga. A versão usada do Docker é a 19.03.2, do Docker Compose, 3.5 e a do Spark, 2.4.

A fim de testar a configuração proposta, foram usadas as aplicações de *word count* [4], *sort*[5] e de cálculo do número PI[6], todas implementadas em Python usando o *framework* Spark. A aplicação de *word count* recebe como argumento um arquivo de texto e, a cada linha forma pares *chave-valor*, sendo a chave uma palavra e o valor, 1. Ao aplicar a operação de *reduce by key* somando os valores, conta-se a quantidade de vezes que cada palavra aparece no arquivo. A aplicação de *sort*, por sua vez, ordena as palavras do texto passado como argumento. Por fim, o cálculo do PI é realizado de uma forma computacionalmente intensiva, já que é feita gerando-se pontos pseudo-aleatórios no quadrado unitário Q (x, y), tal que $0 <= x <= 1$ e $0 <= y <= 1$. Após, verifica-se quais desses pontos pertencem ao círculo unitário, e aplica-se o conceito da série de Gregory-Leibniz para obter-se o valor de PI.

O hardware usado para a execução dos testes é um servidor Dell Power Edge R730 160GB RAM e dois processadores Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz, com 10 núcleos cada, e com cada núcleo contendo duas *threads* - portanto, sendo visto pelo sistema como tendo 40 *cores*. Os softwares *word count* e *sort* recebem como argumento um arquivo de texto[7], que, neste caso, possui aproximadamente 95 *megabytes* (sem as marcações XML). Já o software de cálculo do PI necessita como argumento o número de partições (um bloco atômico de dados, armazenado em um nó, cujo um conjunto destes blocos forma um RDD) que o Spark tem de usar. Em todos os testes, o valor de partições fornecido foi 400. É importante ressaltar que o servidor utilizado para os testes é compartilhado, portanto, pode ter havido interferência de outros processos nos resultados obtidos.

Por fim, o critério de avaliação adotado foi o tempo computado pelo próprio Spark, desde o início da leitura da entrada até a geração dos resultados de saída. Neste trabalho, esse tempo é referido como "tempo de execução". Foram realizadas 30 execuções de cada aplicação, considerando 10, 20, 30 e 40 contêineres trabalhadores do Spark, e, a partir disso, foi calculado o tempo médio, em segundos, de cada uma dessas execuções. Cada contêiner foi limitado a usar um *core* do servidor e a entrada de texto necessária para duas das três aplicações estava acessível aos contêineres trabalhadores através de um volume do Docker compartilhado entre todos eles.

[4]https://github.com/apache/spark/blob/master/examples/src/main/python/wordcount.py
[5]https://github.com/apache/spark/blob/master/examples/src/main/python/sort.py
[6]https://github.com/apache/spark/blob/master/examples/src/main/python/pi.py
[7]http://cs.fit.edu/ mmahoney/compression/enwik8.zip

3.5 RESULTADOS

Considerando o cenário descrito em 3.4, a arquitetura proposta em 3.1 pode ser considerada viável, já que a execução das aplicações apresentadas em 3.2 realizou-se conforme o esperado mesmo com a variação do número de contêineres trabalhadores do Spark. Ainda conforme o cenário de execução descrito em 3.4, pôde-se calcular o tempo médio de execução das aplicações descritas em 3.2, à medida que aumentou-se o número de contêineres trabalhadores do Spark, com os resultados apresentados na tabela 3.1.

Tabela 3.1 – Tempo médio e desvio padrão, em segundos, das aplicações executadas

Aplicação	Trabalhadores	Tempo Médio (s)	Desvio Padrão (s)
PI	10	12,67	0,37
	20	11,39	0,42
	30	12,56	0,34
	40	14,54	0,48
Word Count	10	25,01	0,38
	20	26,95	0,24
	30	28,51	0,32
	40	29,90	0,45
Sort	10	62,97	3,50
	20	84,43	0,78
	30	87,48	0,96
	40	90,66	1,00

Observando-se a Tabela 3.1 e os gráficos presentes nas Figuras 3.4, 3.5 e 3.6, pode-se notar uma tendência de aumento no tempo médio de execução das aplicações conforme acrescentam-se mais contêineres, ainda que o tamanho da entrada seja mantido. Isso pode ser confirmado ao observar que o desvio padrão do tempo médio, em segundos, apresentado na Tabela 3.1, é menor que um segundo na maior parte das execuções. Essa tendência pode ser explicada pelo tamanho das entradas, já que elas podem não ter sido grandes o suficiente para que fosse vantajoso o custo computacional extra do particionamento, agrupamento, comunicação e mapeamento das tarefas da aplicação em cada contêiner. Apesar de não ter existido melhora no tempo médio de execução, o ambiente implementado apresenta elasticidade e é facilmente reproduzível.

Figura 3.4 – Contêineres trabalhadores do Spark *versus* tempo médio(s) de execução - "*word count*"

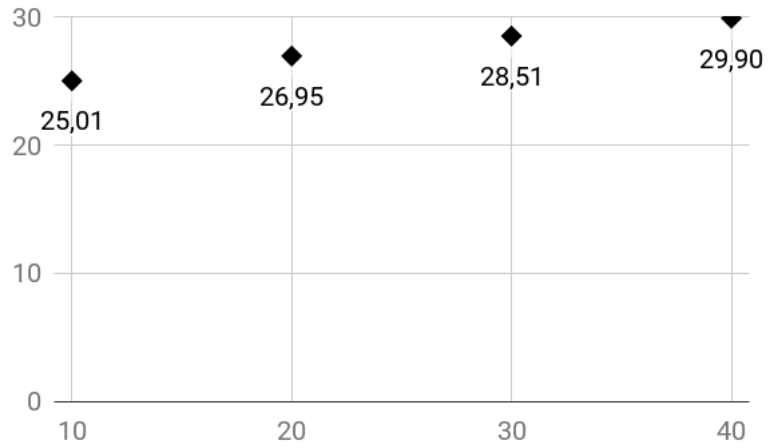


Figura 3.5 – Contêineres trabalhadores do Spark *versus* tempo médio(s) de execução - "*sort*"

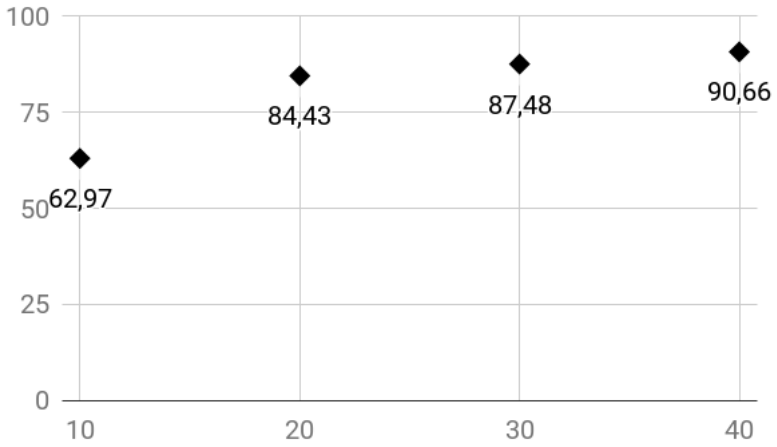
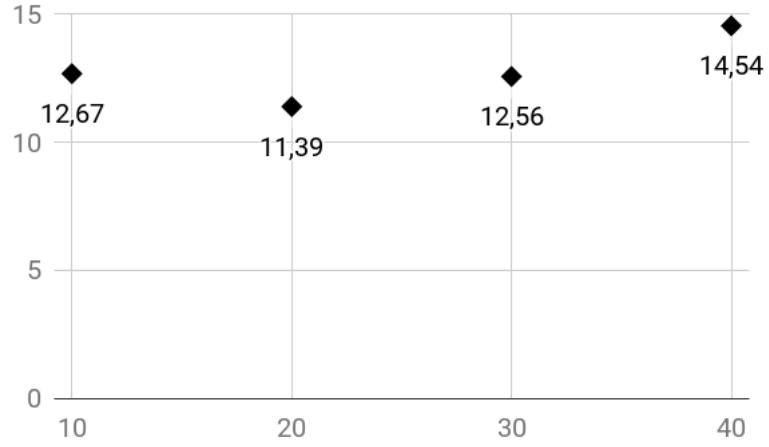


Figura 3.6 – Contêineres trabalhadores do Spark *versus* tempo médio(s) de execução - "cálculo PI"



4 CONSIDERAÇÕES FINAIS

Neste trabalho, foi apresentado um ambiente em nuvem privada, baseado em Docker e Docker Swarm, para o uso do Spark como microsserviço, demonstrando-se a viabilidade da proposta por meio de testes com as aplicações de *word count*, *sort* e cálculo do número PI. Os três principais tópicos abordados (virtualização leve, computação em nuvem e *Big Data*) encontram-se, atualmente, em destaque, tanto no meio acadêmico quanto no meio comercial, por conta da sua aplicabilidade em diferentes contextos.

O ambiente implementado possui elasticidade, sendo possível agregar e retirar contêineres trabalhadores do Spark de forma completamente dinâmica. Apesar de a implementação realizada não ter apresentado ganho no tempo de execução, ele é facilmente reproduzível em qualquer computador que possua o Docker e o Docker Swarm em funcionamento, podendo servir para propósitos estudantis e para a execução de outros testes.

Em trabalhos futuros, pode-se utilizar a vazão como critério de avaliação do ambiente apresentado, variando-se o tamanho da entrada de cada aplicação de forma proporcional ao aumento do número de contêineres trabalhadores do Spark. Também pode-se averiguar se há algum sistema de arquivos distribuído que funcione adequadamente em conjunto com o Docker Swarm e com o Spark, de forma a eliminar a redundância excessiva de dados que há na proposta deste trabalho. Pode-se, também, utilizar-se de outras aplicações e/ou *benchmarks* para avaliar o ambiente proposto em diversos outros critérios, tais quais uso de processamento, de memória principal, rede, disco, entre outros.

REFERÊNCIAS BIBLIOGRÁFICAS

AHMED, Hameeza; ISMAIL, Muhammad Ali; HYDER, Muhammad Faraz; SHERAZ, Syed Muhammad; FOUQ, Nida. Performance comparison of spark clusters configured conventionally and a cloud service. **Procedia Computer Science**, Elsevier, v. 82, p. 99–106, 2016.

APACHE. **Cluster Mode Overview**. 2020. Disponível em: <https://spark.apache.org/docs/latest/cluster-overview.html>.

ASSUNÇÃO, Marcos D; CALHEIROS, Rodrigo N; BIANCHI, Silvia; NETTO, Marco AS; BUYYA, Rajkumar. Big data computing and clouds: Trends and future directions. **Journal of Parallel and Distributed Computing**, Elsevier, v. 79, p. 3–15, 2015.

Bhimani, J.; Yang, Z.; Leeser, M.; Mi, N. Accelerating big data applications using lightweight virtualization framework on enterprise cloud. In: **2017 IEEE High Performance Extreme Computing Conference (HPEC)**. [S.l.: s.n.], 2017. p. 1–7.

BLAIR, Walter; OLMSTED, Aspen; ANDERSON, Paul. Docker vs. kvm: Apache spark application performance and ease of use. In: IEEE. **2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)**. [S.l.], 2017. p. 199–201.

Ceesay, S.; Barker, A.; Varghese, B. Plug and play bench: Simplifying big data benchmarking using containers. In: **2017 IEEE International Conference on Big Data (Big Data)**. [S.l.: s.n.], 2017. p. 2821–2828.

ENES, Jonatan; CACHEIRO, Javier López; EXPÓSITO, Roberto R.; TOURIÑO, Juan. Big data-oriented PaaS architecture with disk-as-a-resource capability and container-based virtualization. **Journal of Grid Computing**, Springer Science and Business Media LLC, v. 16, n. 4, p. 587–605, ago. 2018. Disponível em: <https://doi.org/10.1007/s10723-018-9460-4>.

KUNE, Raghavendra; KONUGURTHI, Pramod Kumar; AGARWAL, Arun; CHILLARIGE, Raghavendra Rao; BUYYA, Rajkumar. The anatomy of big data computing. **Software: Practice and Experience**, Wiley Online Library, v. 46, n. 1, p. 79–105, 2016.

LEI, Zhou; DU, Hongguang; CHEN, Shengbo; ZHU, Caixin; LIU, Xianyang. Dcspark: Virtualizing spark using docker containers. In: IEEE. **2016 International Conference on Audio, Language and Image Processing (ICALIP)**. [S.l.], 2016. p. 13–18.

MARZ, Nathan; WARREN, James. **Big Data: Principles and best practices of scalable realtime data systems**. [S.l.]: New York; Manning Publications Co., 2015.

NGUYEN, Nikyle; BEIN, Doina. Distributed mpi cluster with docker swarm mode. In: IEEE. **2017 ieee 7th annual computing and communication workshop and conference (ccwc)**. [S.l.], 2017. p. 1–7.

PAHL, Claus; HELMER, Sven; MIORI, Lorenzo; SANIN, Julian; LEE, Brian. A container-based edge cloud paas architecture based on raspberry pi clusters. In: IEEE. **2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)**. [S.l.], 2016. p. 117–124.

REYES-ORTIZ, Jorge Luis; ONETO, Luca; ANGUITA, Davide. Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf. In: **INNS Conference on Big Data**. [S.l.: s.n.], 2015. v. 8, p. 121.

RUAN, Bowen; HUANG, Hang; WU, Song; JIN, Hai. A performance study of containers in cloud environment. In: SPRINGER. **Asia-Pacific Services Computing Conference**. [S.l.], 2016. p. 343–356.

SALAM, Mahmoud A; BAHGAT, Waleed M; EL-DAYDAMONY, Eman; ATWAN, Ahmed. A novel framework for web service composition. **International Journal of Simulation–Systems, Science & Technology**, v. 20, n. 3, 2019.

SILBERSCHATZ, Abraham; GAGNE, Greg. **Fundamentos de sistemas operacionais**. [S.l.]: Ltc, 2015.

TANENBAUM, A.S.; BOS, H. **Sistemas Operacionais Modernos**. [S.l.]: PEARSON BRASIL, 2015.

YU, Kevin. **Swarm Architecture**. 2020. Disponível em: <https://hikariai.net/cloud/docker-swarm/>.

ZENG, Hao; WANG, Baosheng; DENG, Wenping; TANG, Junxing. A prototype for analyzing the internet routing system based on spark and docker. In: IEEE. **2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2)**. [S.l.], 2017. p. 267–270.

APÊNDICE A – CONFIGURAÇÃO DO AMBIENTE

Considerando que já há o Docker em funcionamento no dispositivo e que o modo Swarm foi inicializado, primeiramente, é necessário gerar a imagem contendo o Spark a ser utilizada pelo Docker para a criação dos contêineres. Essa imagem pode ser obtida por meio de um arquivo com o seguinte conteúdo:

Figura A.1 – *Dockerfile* utilizado para gerar a imagem contendo o Spark *spark.df*

```
FROM alpine:latest

ENV SPARK_VERSION=2.4.0
ENV HADOOP_VERSION=2.7

RUN apk update && apk upgrade
RUN apk add --no-cache curl bash openjdk8-jre python3 py-pip \
    nss \
        && wget https://archive.apache.org/dist/spark/spark-${SPARK_VERSION}/spark-${SPARK_VERSION}-bin-hadoop${HADOOP_VERSION}.tgz \
        && tar -xvzf spark-${SPARK_VERSION}-bin-hadoop${HADOOP_VERSION}.tgz \
        && mv spark-${SPARK_VERSION}-bin-hadoop${HADOOP_VERSION} spark \
        && rm spark-${SPARK_VERSION}-bin-hadoop${HADOOP_VERSION}.tgz \
        && cd /

ENV PYTHONHASHSEED 1
```

E o comando para gerá-la:

Figura A.2 – Comando para gerar a imagem contendo o Spark

```
docker build -f spark.df -t spark --force-rm .
```

A configuração dos microsserviços mestre e trabalhador é apresentada em um arquivo YML (*compose file*), que, neste caso, foi chamado de *cluster.yml*. Como os serviços usam a rede para a comunicação entre eles e o *driver* de rede do Docker Swarm é o Overlay, necessitou-se definir uma rede personalizada, conforme segue:

Figura A.3 – Serviços definidos para a execução do Spark - arquivo *cluster.yml* - serviço mestre

```
version :    '3.5 '

services :
      spark - master :
            image :  spark
            command :/ spark / bin / spark - class    org . apache . spark .
    deploy . master . Master   -h   spark - master
            hostname :  spark - master
            environment :
               SPARK_MASTER :spark :// spark - master :7077
               SPARK_PUBLIC_DNS : 127.0.0.1
            ports :
               -   7077:7077
               -   8080:8080
               -   6066:6066
            volumes :
               -   ./ data :/ data /
            networks :
                  spark_nw :
                        ipv4_address :    10.5.0.5

```

Figura A.4 – Serviços definidos para a execução do Spark - arquivo *cluster.yml* - serviço trabalhador e rede

```
21      spark - worker :
22            image :  spark
23            depends_on :
24               -   spark - master
25            command :/ spark / bin / spark - class    org . apache . spark .
    deploy . worker . Worker  spark :// spark - master :7077
26            hostname :  spark - worker
27            environment :
28               SPARK_MASTER : spark :// spark - master :7077
29               SPARK_WORKER_CORES 1
30               SPARK_WORKER_MEMORY 1:g
31               SPARK_WORKER_PORT 8881
32               SPARK_WORKER_WEBUI_PORT 8081
33               SPARK_PUBLIC_DNS : 127.0.0.1
34            ports :
35               -   8081:8081
36            volumes :
37               -   ./ data :/ data /
38            networks :
39               -   spark_nw
40
41      networks :
42      spark_nw :
43            name : spark_nw
44            driver :    overlay
45            attachable :    true
46            ipam :
47                  driver :    default
48                  config :
49                     -   subnet :   10.5.0.0/24
```

Para iniciar a pilha de serviços demonstrada acima, usa-se o comando:

Figura A.5 – Comando para iniciar a pilha de serviços no Docker Swarm

```
1 docker   stack   deploy   -- compose - file    cluster . yml    spark
```

Para aumentar ou diminuir o número de contêineres executando um serviço (a exemplo, o serviço trabalhador do Spark), executa-se:

Figura A.6 – Comando para definir a quantidade de contêineres executando um mesmo serviço

```
1 docker service scale spark_spark - worker = N
```

Onde N é a quantidade de contêineres executando o serviço em questão. A fim de parar os serviços sendo executados na pilha demonstrada, utiliza-se:

Figura A.7 – Comando para remover a pilha de serviços

```
1 docker stack rm spark
```

Um exemplo de aplicação (*word count*) executada sobre a pilha de serviços demonstrada neste apêndice pode ser encontrado nesse repositório[1].

[1] https://github.com/carlabarden/tcc