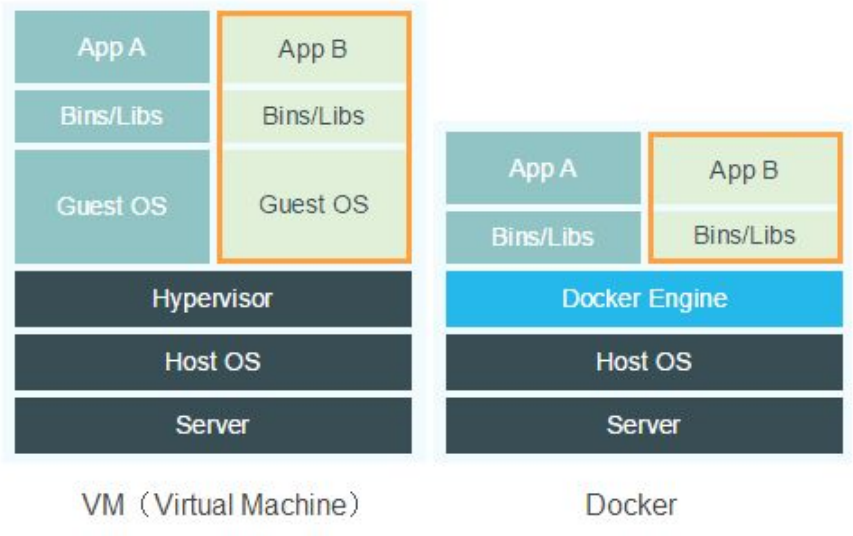
Seleção de Números Primos em um Intervalo Utilizando Docker – Servidor Cliente

Gustavo Lucas Moreira[1], Philipe Lemos Parreira[1]
Departamento de Computação, Universidade Federal de Ouro Preto -
campus Morro do Cruzeiro 35400-000, Ouro Preto - MG, Brasil
{gustavo.lucas, philipe.parreira}@aluno.ufop.edu.br

Resumo: *Este artigo apresenta a implementação de um algoritmo para selecionar e contar os números primos em um determinado intervalo. O objetivo deste artigo é desenvolver e entender o funcionamento do Docker e suas características e diferença com Kubernetes.*

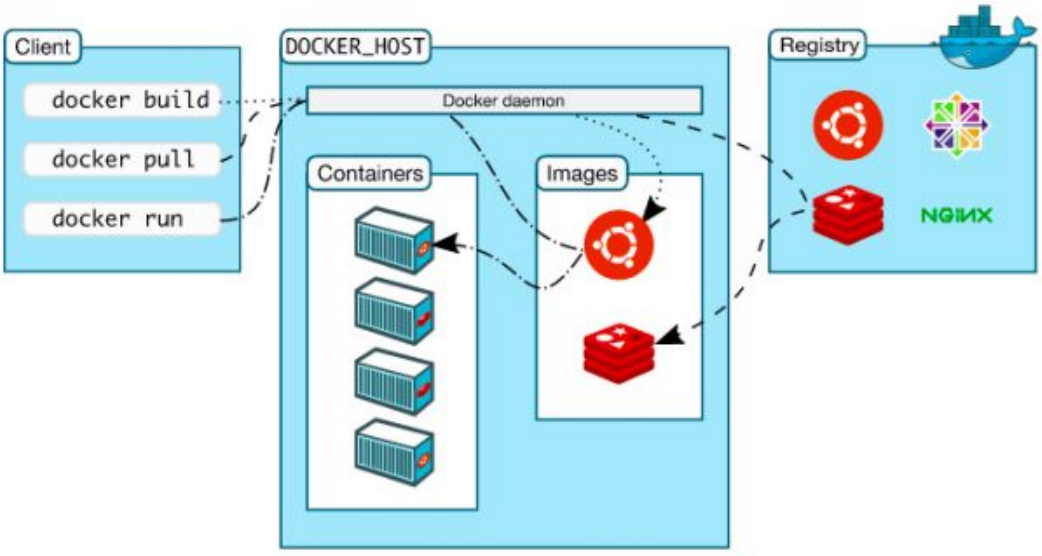
I. Containers Docker

O Docker [1] é uma plataforma que automatiza a implantação de aplicações dentro de ambientes isolados denominados *containers*. O principal objetivo dessa tecnologia é proporcionar o empacotamento de aplicações com todas as suas dependências (bibliotecas do sistema operacional, ferramentas, etc.) em uma unidade padronizada para desenvolvimento de software. O ambiente Docker é bem semelhante às estruturas de ambiente virtualizados tradicionais como *hypervisor*. Entretanto, difere no que diz respeito à necessidade de uma camada intermediária de sistema operacional entre o hospedeiro e as aplicações hospedadas. Tal camada intermediária é dispensável pela plataforma Docker, uma vez que as soluções baseadas nesta tecnologia utilizam o mesmo kernel para todas as aplicações, criando ambientes isolados a nível de disco, memória e processamento. Portanto, o Docker é mais "leve" que um virtualizador (*hypervisor*).



II. Arquitetura do Docker

O Docker é baseado em uma arquitetura cliente-servidor conforme apresentado na figura abaixo. O servidor Docker, representado na Figura por DOCKER_HOST, é responsável pela gerência e execução de containers. Cada container é visto como uma máquina virtual completa, com seu próprio sistema operacional, sistema de arquivos e interfaces de rede. O servidor Docker oferece uma API REST para as operações de gerenciamento de containers. Existem diversos clientes Docker (tais como linha de comando, interface Web, etc) que podem se comunicar com um servidor Docker através de sua API REST.



Arquitetura do Docker

Cada *container* corresponde a uma instância de uma imagem Docker. As imagens são templates com instruções de como criar um container. As imagens são semelhantes a imagens de máquinas virtuais e endereçam o problema da dependência provendo um arquivo binário no qual todos os softwares são instalados, configurados e testados. Imagens podem ser adquiridas de componentes chamados Docker Registry, que consiste em uma biblioteca de imagens Docker. Os Docker Registries podem ser públicos (ex.: hub.docker.com) ou privados, hospedados na própria instituição e permitindo que as imagens fiquem disponíveis apenas para as pessoas autorizadas. Com

isso, é possível também a criação de imagens privadas, customizadas para um determinado ambiente.

III. Instalaçao e Uso

O Docker está disponível em diversas plataformas, sendo mais comumente utilizado sistema operacional Linux. Durante os testes foi utilizado o sistema operacional Linux Ubuntu 18.04 x64 LTS, e a seguir serão apresentados os procedimentos para configuração dos repositórios e a instalação do Docker neste sistema operacional.

Inicialmente, são instaladas as dependências e chaves criptográficas necessárias para utilização do repositório.

```
$ uname -r
$ sudo apt-get update
$ sudo apt install apt-transport-https ca-certificates curl
software-properties-common
$ curl -fsSL https://get.docker.com/ | sudo apt-key add -
$ sudo add-pt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu
bionic stable"
$ sudo apt update
$ sudo usermod -aG docker ${USER}
```

Procedimento 01: Instalando dependencias do Docker, e inserindo os repositorios

Para testar o ambiente será criado um contêiner baseado na imagem padrão do GCC como exemplo (Procedimento 02):

```
SERVER:
       FROM gcc:4.9
       MAINTAINER Gustavo Lucas
       COPY . /usr/src/server
       WORKDIR /usr/src/server
       RUN g++ -o server *.cpp -lpthread -std=c++11
       CMD ["./server"]

CLIENTE:
       FROM gcc:4.9
       MAINTAINER Gustavo Lucas
       COPY . /usr/src/cliente
       WORKDIR /usr/src/cliente
```

```
RUN g++ -o cliente *.cpp -lpthread -std=c++11
CMD ["./cliente"]
```

Procedimento 02: Exemplo de Dockerfile para GCC

O comando *FROM* especifica qual será a imagem e tag(versão) utilizada como base para criação do contêiner, e o *COPY* é utilizado para copiar o conteudo do diretorio raiz que se encontra na máquina hospedeira para o diretório */usr/src/cliente* que se encontra no container. Ao comando *RUN* temos os termos necessários para compilação da aplicação assim como suas dependências e o comando *CMD* realiza a diretiva de compilação em sistemas Linux.

Após salvar o arquivo com o nome *Dockerfile* em cada repositório, cliente e server, via shell efetue o build do container (o build baixe a imagem a partir do hub do Docker)

```
$ docker build -t server .
$ docker build -t cliente .
```

Procedimento 02: Efetuando build no container do GCC

O parâmetro -t define qual nome será utilizado para a imagem, e para esse exemplo utilizaremos "server" e "cliente" . Por fim é informado o diretório onde se encontra o Dockerfile, ou pode ser utilizado "." para considerar o diretório atual.

O próximo passo é criar um container a partir da imagem (apacheimg) gerada no passo anterior. Para criar o container execute o seguinte comando: (Procedimento 03)

```
$ docker run -it --rm --name my-running-server server bash
$ docker run -it --rm --name my-running-cliente cliente bash
```

Procedimento 03: Executando o container do GCC

O parâmetro -it é utilizado para que o contêiner seja interativo, em seguida foi utilizado o parâmetro --rm para que seja removido toda e qualquer dado gerado com essa aplicação, ao parâmetro --name para especificar o nome do container (nesse exemplo foi utilizado my-running-server e my-running cliente), e o nome da imagem gerada no Procedimento 02, ou seja, server e cliente, e assim por fim o comando bash gera uma interface de comando com a aplicação.

IV. Descrição do problema

Um número é classificado como primo se ele é maior do que 1 e é divisível apenas por 1 e por si mesmo. Assim, deseja-se obter um número de números primos em um determinado intervalo.

O algoritmo implementado foi executado em uma máquina virtual Ubuntu com 4 GB de memória RAM e 100 GB ROM, ao uso do Docker adotamos a utilização mais básica do programa para execução dos processos.

~~1~~	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	~~49~~	~~50~~
~~51~~	~~52~~	53	~~54~~	~~55~~	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	~~65~~	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	~~77~~	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	~~85~~	~~86~~	~~87~~	~~88~~	89	~~90~~
~~91~~	~~92~~	~~93~~	~~94~~	~~95~~	~~96~~	97	~~98~~	~~99~~	~~100~~

Figura 1.0: Números primos

V. Descrição da Resolução do Problema

A resolução deste problema é feita de maneira simples. para cada número do intervalo de 0 ao valor recebido do cliente, executa-se um algoritmo que implementa o Teorema que diz: “Se n é um número composto, então não possui um divisor primo menor ou igual a n1⁄2.”, ou seja, um inteiro é primo se ele não é divisível por nenhum primo menor ou igual a sua raiz quadrada.

A solução implementada contém dois loops para realizar todas as verificações, o que torna o algoritmo demorado para intervalos muito grandes. Dessa forma também apresentamos duas resoluções que utilizam o mesmo algoritmo, porém utilizando o auxílio de threads e fork para comparar o desempenho do programa.

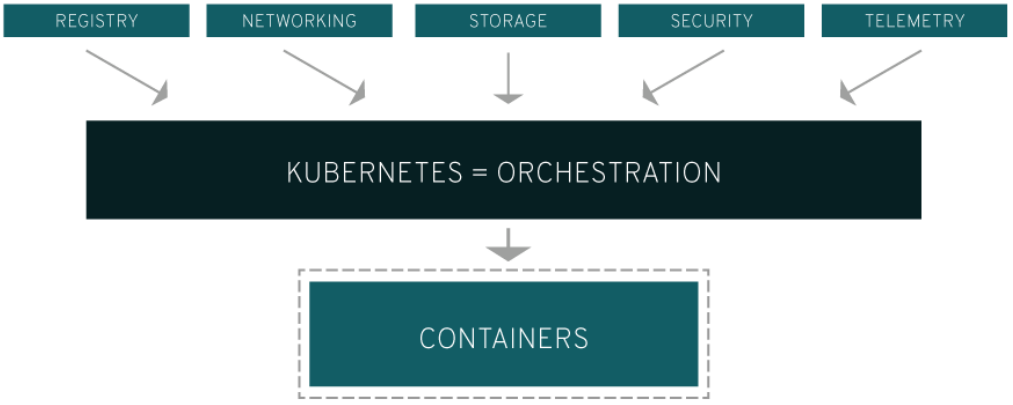
VI. Implementação utilizando Server-Cliente

A função de enviar e receber mensagens de cliente é dado pela utilização do modelo UDP de envio de dados assim, os dados são enviado, recebido, calculado e enviado novamente ao cliente com a quantidade de números primos presente no intervalo. Assim como o código foi de reuso para outra aplicação, este foi realizando pequenas alterações para o uso neste trabalho.

VII. Kubernetes

O Kubernetes, k8s(k + 8 caracteres + s) é um produto Open Source utilizado para automatizar a implantação, o dimensionamento e o gerenciamento de aplicativos em contêiner. Ele funciona agrupando contêineres que compõem uma aplicação em unidades lógicas para facilitar o gerenciamento e a descoberta de serviço. Concebido com base nos mesmos princípios que permitem ao Google executar milhares de contêineres por semana, o Kubernetes pode ser redimensionado sem aumentar sua equipe de operações. Originalmente, o Kubernetes foi criado e desenvolvido pelos engenheiros do Google.

Aplicações de produção abrangem múltiplos contêineres. Eles devem ser implantados em vários host do servidor. A segurança dos containers tem várias camadas e pode ser complexa. É aí que o Kubernetes entra em cena. Ele oferece recursos de orquestração e gerenciamento necessários para implantar containers m escala para essas cargas de trabalho.



VIII. Arquitetura Kubernetes

Kubernetes [2] é composto por uma série de componentes, cada um com um propósito diferente. Para garantir que exista uma separação de responsabilidades e que o sistema seja resiliente, o **K8s** utiliza um cluster de máquinas para ser executado.

As máquinas de um cluster são separadas em três tipos:

Node

O primeiro tipo é chamado de *Node*. O papel de um *Node* é executar os contêineres que encapsulam as aplicações sendo gerenciadas pelo K8s.

Quando você faz o *deploy* de uma aplicação em um cluster K8s, essa aplicação vai ser executada em um dos *Nodes* do cluster. O conjunto de *Nodes* forma o que chamamos de *Workers*.

etcd

O segundo tipo de nó é o *etcd*. O *etcd* é, na verdade, o nome da **base de dados distribuída** que é utilizada para armazenar tudo o que está acontecendo dentro do cluster, incluindo o estado da aplicação.

Em ambientes de produção, um bom gerenciamento desses nós é essencial para garantir que o *cluster* esteja sempre disponível.

Master

Finalmente, o último tipo de nó é o que chamamos de *Master*. É nesse tipo de nó que os principais componentes do Kubernetes são executados, como o *Scheduler*, o qual tem a responsabilidade de controlar a alocação de recursos no cluster.

O conjunto de nós *Master* forma o que pode ser considerado o cérebro de um cluster Kubernetes: o *Control Plane*.

Control Plane

O *Control Plane* do Kubernetes pode ser considerado o cérebro de um cluster. Ele é responsável por gerenciar os principais componentes do sistema e garantir que tudo está funcionando de acordo com o estado desejado da aplicação.

Para facilitar a representação desse estado, o K8s trabalha com uma abstração chamada de *Object*. Um *Object* representa parte do estado da aplicação e quando o seu estado atual se difere do estado desejado, mudanças são aplicadas para que os dois estados se igualem novamente.

Existem diversos tipos de *Objects* em um ambiente Kubernetes, mas alguns deles são essenciais para entendermos como um cluster funciona.

O primeiro deles é a *Pod*.

Na seção sobre aplicações *cloud-native*, uma das principais características ressaltadas é que essas aplicações utilizam contêineres para encapsular seus microsserviços. No entanto, quando falamos sobre aplicações sendo executadas em um cluster Kubernetes, não falamos sobre contêineres diretamente, mas sim sobre *Pods*.

Pods são a unidade básica de um um cluster **K8s**. Elas encapsulam um ou mais contêineres de uma aplicação e representam um processo dentro do cluster. Quando fazemos o *deploy* de uma aplicação no K8s, estamos criando uma ou mais *Pods*.

No entanto, *Pods* são efêmeras, ou seja, elas são criadas e destruídas de acordo com as necessidades do cluster.

Para garantir que o acesso a um microsserviço esteja sempre disponível, existe um *Object* chamado *Service* que encapsula uma ou mais *Pods* e é capaz de encontrá-las dinamicamente em qualquer *Node* do cluster.

Outro elemento importante é o *Deployment*. Esse tipo de *Object* oferece uma série de funcionalidades que automatizam todos aqueles passos que descrevemos de um cenário típico de desenvolvimento de software, com *deploys* manuais ou semi-automatizados de uma aplicação.

Utilizando *Deployments*, nós podemos descrever qual o estado desejado da nossa aplicação e um *Deployment controller* vai se encarregar de transformar o estado atual no estado desejado, caso eles sejam diferentes.

IX. Considerações Finais

Este documento apresentou um relato das atividades realizadas durante a realização do trabalho prático 2, assim como particularizar os conceitos exigidos e discutidos neste trabalho.

Dentre as dificuldades encontradas, podemos salientar problemas apresentados pela implementação e entendimento dos conceitos e aplicações do Docker, assim como a comunicação entre os dois contêineres. Outro problema encontrado também foi o desenvolvimento da aplicação servidor-cliente, o que acarretou na excessiva perda de dados entre os mesmos.

Ao analisarmos o tempo decorrido na execução dos cálculos dos primos no intervalo desejado temos uma execução coerente com a aplicação dos contêineres. Ao lixo no caso do resposta ao cliente deve-se pelo uso do protocolo UTP. Assim, este trabalho nos proporcionou um conhecimento e prática dos conceitos do Docker na aplicação de gerenciamento de contêineres e o poder dessa ferramenta nos dias atuais.

X. Referências

1. Documentação Docker, Disponível em http://docs.docker.com/, acessado em 29/11/2019.
2. Kubernets Arquitetura, Disponivel em https://blog.geekhunter.com.br/kubernetes-a-arquitetura-de-um-cluster/, acessado em 29/11/2019.
3. Repositório do Trabalho, Disponivel em: https://github.com/gustahlucas/SOTP.git , acessado em 01/11/2019.