Sessão 8: Isolamento de processos e conteinerização

Nesta sessão, iremos configurar o sistema de conteinerização Docker, testando suas funcionalidades de criação rápida de containers, escalabilidade e orquestração via Docker Swarm. Compararemos a facilidade e rapidez de configuração com um sistema web mais "tradicional", como o que fizemos na sessão anterior.

1) Topologia desta sessão

Criaremos duas novas máquinas nesta sessão, a saber:

- `docker1`, nodo-mestre de configuração do Docker Compose/Swarm. Endereço IP 10.0.42.9/24.
- `docker2`, nodo-escravo (ou *worker*) do Docker Compose/Swarm. Endereço IP 10.0.42.10/24.

1. Como de costume, vamos à criação dos registros DNS. Acesse a máquina `ns1` como o usuário `root`:

   ```
   # hostname ; whoami
   ns1
   root
   ```

   Edite o arquivo de zonas `/etc/nsd/zones/intnet.zone`, inserindo entradas A para a máquinas indicadas no começo desta atividade. **Não se esqueça** de incrementar o valor do serial no topo do arquivo!

   ```
   # nano /etc/nsd/zones/intnet.zone
   (...)
   ```

   ```
   # grep docker /etc/nsd/zones/intnet.zone
   docker1 IN    A                10.0.42.9
   docker2 IN    A                10.0.42.10
   ```

   Faça o mesmo para o arquivo de zona reversa:

   ```
   # nano /etc/nsd/zones/10.0.42.zone
   ```

   ```
   # grep docker /etc/nsd/zones/10.0.42.zone
   9       IN   PTR               docker1.intnet.
   10      IN   PTR               docker2.intnet.
   ```

   Assine o arquivo de zonas usando o *script* criado anteriormente:

```
# bash /root/scripts/signzone-intnet.sh
reconfig start, read /etc/nsd/nsd.conf
ok
ok
ok
ok removed 6 rrsets, 2 messages and 0 key entries
```

Verifique a criação das entradas usando o comando `dig`:

```
# for host in docker1 docker2; do echo -n "$host: "; dig ${host}.intnet +short;
done
docker1: 10.0.42.9
docker2: 10.0.42.10
```

```
# for ip in 9 10; do echo -n "10.0.42.${ip}: "; dig -x 10.0.42.${ip} +short; done
10.0.42.9: docker1.intnet.
10.0.42.10: docker2.intnet.
```

2) Criação da VM docker1 e instalação

1. Primeiro, vamos criar a VM `docker1` e instalar o software Docker nela. Clone a máquina `debian-template` para uma de nome `docker1`, com uma única interface de rede conectada à DMZ. O IP da máquina será 10.0.42.9/24.

   Concluída a clonagem, ligue a máquina e logue como `root`. Depois, use o script `/root/scripts/changehost.sh` para fazer a configuração automática, como de costume.

   ```
   # hostname ; whoami
   debian-template
   root
   ```

   ```
   # bash ~/scripts/changehost.sh -h docker1 -i 10.0.42.9 -g 10.0.42.1
   Signing ssh_host_ecdsa_key.pub key...
   Signing ssh_host_ed25519_key.pub key...
   Signing ssh_host_rsa_key.pub key...
   Configuring host key trust...
   Configuring user key trust...
   All done!
   ```

2. Aplique o *baseline* de segurança à máquina `docker1`, repetindo o que fizemos no passo (2), atividade (2) da sessão 7:

```
$ hostname ; whoami
client
ansible
```

```
$ sed -i '/\[srv\]/a docker1' ~/ansible/hosts
```

```
$ ansible-playbook -i ~/ansible/hosts -l docker1 -Ke ansible_become_method=su
~/ansible/srv.yml ; ansible-playbook -i ~/ansible/hosts -l docker1
~/ansible/srv.yml
SUDO password:

(...)

PLAY RECAP
*********************************************************************************
***********************

docker1                    : ok=10   changed=8    unreachable=0    failed=0
```

3. Agora, acesse a máquina `docker1` como o usuário `root`:

```
# hostname ; whoami
docker1
root
```

4. Vamos proceder com os passos de instalação seguindo o manual oficial do Docker, disponível https://docs.docker.com/install/linux/docker-ce/debian/#install-docker-ce . Primeiramente, vamos habilitar a instalação de pacotes APT via HTTPS, instalando os pacotes a seguir:

```
# apt-get install -y \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg2 \
    software-properties-common
```

5. Agora, adicione a chave GPG do repositório do Docker com o comando:

```
# curl -fsSL https://download.docker.com/linux/debian/gpg | sudo apt-key add -
OK
```

Verifique que a chave foi recebida corretamente:

```
# apt-key fingerprint 0EBFCD88
pub   rsa4096 2017-02-22 [SCEA]
      9DC8 5822 9FC7 DD38 854A  E2D8 8D81 803C 0EBF CD88
uid           [ unknown] Docker Release (CE deb) <docker@docker.com>
sub   rsa4096 2017-02-22 [S]
```

6. Adicione o repositório do Docker à lista de repositórios disponíveis para instalação de pacotes:

```
# echo "deb [arch=amd64] https://download.docker.com/linux/debian \
    $(lsb_release -cs) \
    stable" > \
    /etc/apt/sources.list.d/docker.list
```

   Atualize a lista de pacotes disponíveis, e instale o `docker-ce`:

```
# apt-get update ; apt-get install -y docker-ce
```

7. Cheque qual versão do Docker foi instalada:

```
# docker --version
Docker version 18.09.0, build 4d60db4
```

   Verifique a correta instalação do Docker rodando a imagem `hello-world`, uma imagem de teste:

```
# docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
d1725b59e92d: Pull complete
Digest: sha256:0add3ace90ecb4adbf7777e9aacf18357296e799f81cabc9fde470971e499788
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

O Docker detecta que a imagem `hello-world` não está disponível localmente, faz o download da mesma a partir do *registry* global do Docker Hub, e a executa. Falaremos mais sobre o *registry* global em atividades posteriores.

8. Agora, desligue a VM `docker1`. Para não ter que repetir os passos de instalação na VM `docker2`, vamos cloná-la a partir da `docker1` e aproveitar o trabalho que já realizamos até aqui.

```
# halt -p
```

3) Criação da VM docker2

1. Com a VM `docker1` desligada, clone-a para uma máquina de nome `docker2`. O IP dessa máquina será 10.0.42.10/24.

   Concluída a clonagem, ligue **apenas** a máquina `docker2` e logue como `root`. Não ligue a máquina `docker1` ainda, ou haverá um conflito de IP na rede. Observe: como a máquina `docker1` já estava configurada para operar com o `sudo` distribuído via Ansible, será necessário escalar privilégio a partir de um usuário autorizado, como `aluno`.

```
$ hostname ; whoami
docker1
aluno
```

```
$ sudo -i
[sudo] senha para aluno:
root@docker1:~#
```

Depois, use o script `/root/scripts/changehost.sh` para fazer a configuração automática, como de costume.

```
# bash ~/scripts/changehost.sh -h docker2 -i 10.0.42.10 -g 10.0.42.1
Signing ssh_host_ecdsa_key.pub key...
Signing ssh_host_ed25519_key.pub key...
Signing ssh_host_rsa_key.pub key...
Configuring host key trust...
Configuring user key trust...
All done!
```

2. Para manter a organização em nosso ambiente, adicione a máquina `docker2` ao inventário do Ansible. Como o usuário `ansible` na máquina `cliente`, execute:

```
$ hostname ; whoami
client
ansible
```

```
$ sed -i '/\[srv\]/a docker2' ~/ansible/hosts
```

Note que não é necessário re-executar o *playbook* para a máquina `docker2`, já que todos os controles de segurança foram aplicados anteriormente à máquina `docker1`, a partir da qual fizemos a clonagem.

3. Feito isso, ligue também a máquina `docker1`, e prossiga com as atividades desta sessão.

4) Trabalhando com containers

1. Acesse a máquina `docker1` como o usuário `root`:

```
# hostname ; whoami
docker1
root
```

Agora, crie um diretório vazio `/root/docker`, e entre nele.

```
# mkdir ~/docker ; cd ~/docker
```

2. Vamos criar um *Dockerfile* — um arquivo que define o que será instalado e configurado dentro do seu container. Nesse arquivo são mapeados acessos a recursos como interfaces de rede e volumes de disco virtualizados, em um ambiente isolado do restante do sistema operacional.

   Crie o arquivo novo `/root/docker/Dockerfile` com o seguinte conteúdo:

```
 1 # Usar uma imagem oficial do runtime Python como imagem-pai
 2 FROM python:2.7-slim
 3
 4 # Configurar o diretorio de trabalho como /app
 5 WORKDIR /app
 6
 7 # Copiar o conteudo do diretorio corrent para dentro do container em /app
 8 COPY . /app
 9
10 # Instalar quaisquer dependencias do Python especificadas no arquivo
requirements.txt
11 RUN pip install --trusted-host pypi.python.org -r requirements.txt
12
13 # Expor a porta 80 para o mundo externo, fora do container
14 EXPOSE 80
15
16 # Definir uma variavel de ambiente $World
17 ENV NAME World
18
19 # Rodar a aplicacao app.py ao lancar o container
20 CMD ["python", "app.py"]
```

   Esse *Dockerfile* faz referência a dois arquivos que ainda não criamos — `app.py` e `requirements.txt`. Vamos criá-los.

3. Primeiro, crie o arquivo novo `/root/docker/requirements.txt` com o seguinte conteúdo:

```
1 Flask
2 Redis
```

   O comando `pip install -r requirements.txt`, invocado no *Dockerfile*, irá portanto instalar as bibliotecas Flask e Redis para o ambiente Python do container.

4. Agora, vamos à aplicação em si. Crie o arquivo novo `/root/docker/app.py` com o seguinte conteúdo:

```
from flask import Flask
from redis import Redis, RedisError
import os
import socket

# Connect to Redis
redis = Redis(host="redis", db=0, socket_connect_timeout=2, socket_timeout=2)

app = Flask(__name__)

@app.route("/")
def hello():
    try:
        visits = redis.incr("counter")
    except RedisError:
        visits = "<i>cannot connect to Redis, counter disabled</i>"

    html = "<h3>Hello {name}!</h3>" \
           "<b>Hostname:</b> {hostname}<br/>" \
           "<b>Visits:</b> {visits}"
    return html.format(name=os.getenv("NAME", "world"), hostname=socket
.gethostname(), visits=visits)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)
```

A aplicação acima, bastante simples, irá exibir a *string* `Hello World!`, uma página web que mostra o *hostname* da máquina local (no caso, o identificador do container) e um contador do número de visitas realizadas ao site. Esse contador é mantido em um volume uniforme, acessível por todos os containers da aplicação, com a biblioteca Redis.

5. Liste o conteúdo do diretório `/root/docker`. Você deve ter os arquivos abaixo:

```
# ls -1 ~/docker/
app.py
Dockerfile
requirements.txt
```

Para fazer o *build* do container, basta rodar o comando `docker build`:

```
# cd ~/docker ; docker build -t pyhello .
Sending build context to Docker daemon   5.12kB
Step 1/7 : FROM python:2.7-slim
2.7-slim: Pulling from library/python
a5a6f2f73cd8: Pull complete
8da2a74f37b1: Pull complete
09b6f498cfd0: Pull complete
f0afb4f0a079: Pull complete
Digest: sha256:f82db224fbc9ff3309b7b62496e19d673738a568891604a12312e237e01ef147
Status: Downloaded newer image for python:2.7-slim
 ---> 0dc3d8d47241
Step 2/7 : WORKDIR /app
(...)
Step 3/7 : COPY . /app
(...)
Step 4/7 : RUN pip install --trusted-host pypi.python.org -r requirements.txt
(...)
Step 5/7 : EXPOSE 80
(...)
Step 6/7 : ENV NAME World
(...)
Step 7/7 : CMD ["python", "app.py"]
(...)
Successfully built d2923f9142e3
Successfully tagged pyhello:latest
```

O Docker irá executar os comandos do *Dockerfile*, em ordem:

1. Ao detectar que a imagem `python:2.7-slim` não existe na máquina local, ela será baixada do *registry* global do Docker Hub, como feito anteriormente com a imagem `hello-world.`
2. Deriva-se uma nova imagem a partir de `python:2.7-slim`, e o diretório `/app` é criado na raiz do container.
3. Os arquivos da pasta local são copiadas para `/app`.
4. O comando `pip install -r requirements.txt` instala as bibliotecas necessárias ao funcionamento da aplicação, Flask e Redis, bem como suas dependências.
5. A porta 80/TCP do container é exposta para o mundo externo.
6. Cria-se uma nova variável de ambiente, `$World`.
7. Roda-se o comando `python app.py`, executando a aplicação. Como este comando objetiva apenas a criação da imagem do container, a aplicação é encerrada logo em seguida, e a imagem do container é finalizada sob a *tag* `pyhello`.

Para listar a imagem recém-criada, use o comando `docker image ls`:

```
# docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
pyhello             latest              d2923f9142e3        6 minutes ago
131MB
python              2.7-slim            0dc3d8d47241        36 hours ago
120MB
hello-world         latest              4ab4c602aa5e        2 months ago
1.84kB
```

6. Para rodar o container, basta executar `docker run`:

```
# docker run -p 7080:80 pyhello
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
```

   O comando acima irá iniciar o container escutando na porta 80/TCP, e mapeando-a para a porta 7080/TCP da máquina virtual `docker1`.

7. Para conseguir acessar o container a partir do IP público do firewall (interface `enps0s3` da máquina `ns1`), precisamos adicionar algumas regras novas. Acesse a máquina `ns1` como `root`:

```
# hostname ; whoami
ns1
root
```

   Para que consigamos atingir a máquina `docker1` será necessário criar uma regra de DNAT na tabela `nat`, *chain* PREROUTING, além de uma regra na tabela `filter`, *chain* FORWARD, correspondente. Mapearemos a porta externa 7080/TCP para a porta interna 7080/TCP, sem alterações.

```
# iptables -t nat -A PREROUTING -i enp0s3 -p tcp -m tcp --dport 7080 -j DNAT --to
-destination 10.0.42.9
```

```
# iptables -A FORWARD -i enp0s3 -d 10.0.42.9/32 -p tcp -m tcp --dport 7080 -j
ACCEPT
```

8. Em sua máquina física, abra o navegador e aponte-o para o IP público do firewall (interface `enps0s3` da máquina `ns1`), na porta 7080/TCP:

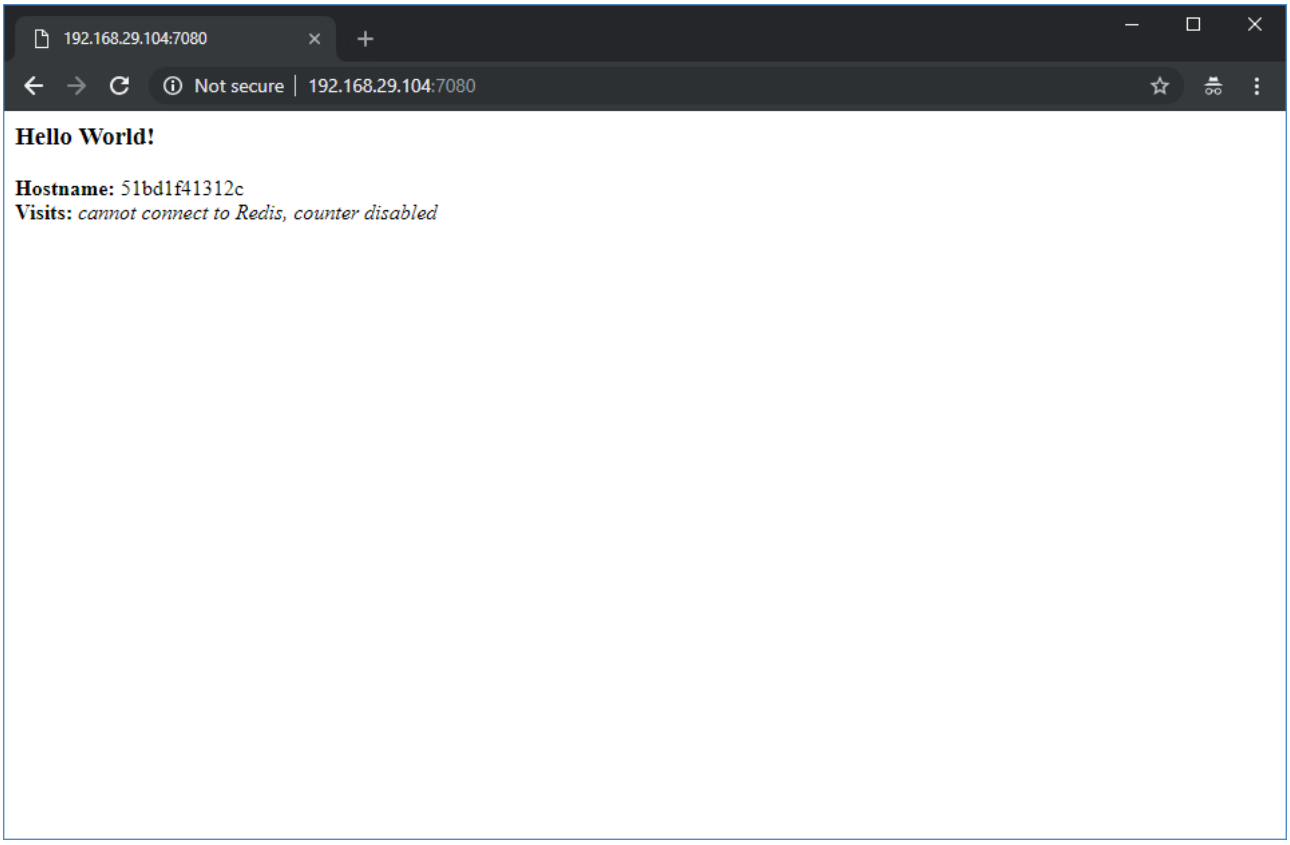

Figura 1. Container operacional no Docker

Tudo certo!

9. De volta à máquina `docker1` como `root`, note que o acesso que fizemos foi registrado na console, com as seguintes mensagens:

```
192.168.29.106 - - [17/Nov/2018 20:10:53] "GET / HTTP/1.1" 200 -
192.168.29.106 - - [17/Nov/2018 20:10:53] "GET /favicon.ico HTTP/1.1" 404 -
```

Para encerrar o container, digite `CTRL + C`. Vamos reexecutá-lo em *background* com a opção `-d` (*detached*):

```
# docker run -d -p 7080:80 pyhello
2d93cc85f066dc6afa9a5c9ea5d0fd08112f52cec99aad8d1d862608d67ddc81
```

O ID do container é mostrado, e retomamos controle do terminal. Para visualizar quais containers estão em operação neste momento, use o comando `docker container ls`:

```
# docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS
      NAMES
2d93cc85f066        pyhello             "python app.py"     42 seconds ago      Up
41 seconds       0.0.0.0:7080->80/tcp   modest_stallman
```

Tente acessar novamente o container no navegador em sua máquina física — ele está funcionando normalmente.

Para parar um container rodando em *background*, use `docker container stop` e passe como parâmetro o ID do container, assim:

```
# docker container stop 2d93cc85f066
2d93cc85f066
```

5) Distribuindo containers para um *registry* externo

1. Vamos distribuir o container que criamos no passo anterior para o *registry* global do Docker Hub. Para isso, o primeiro passo é criar uma conta em https://hub.docker.com/ . Acesse essa página através do navegador em sua máquina física e preencha os campos em *New to Docker?*; **importante**: use um endereço de e-mail real em seu cadastro.

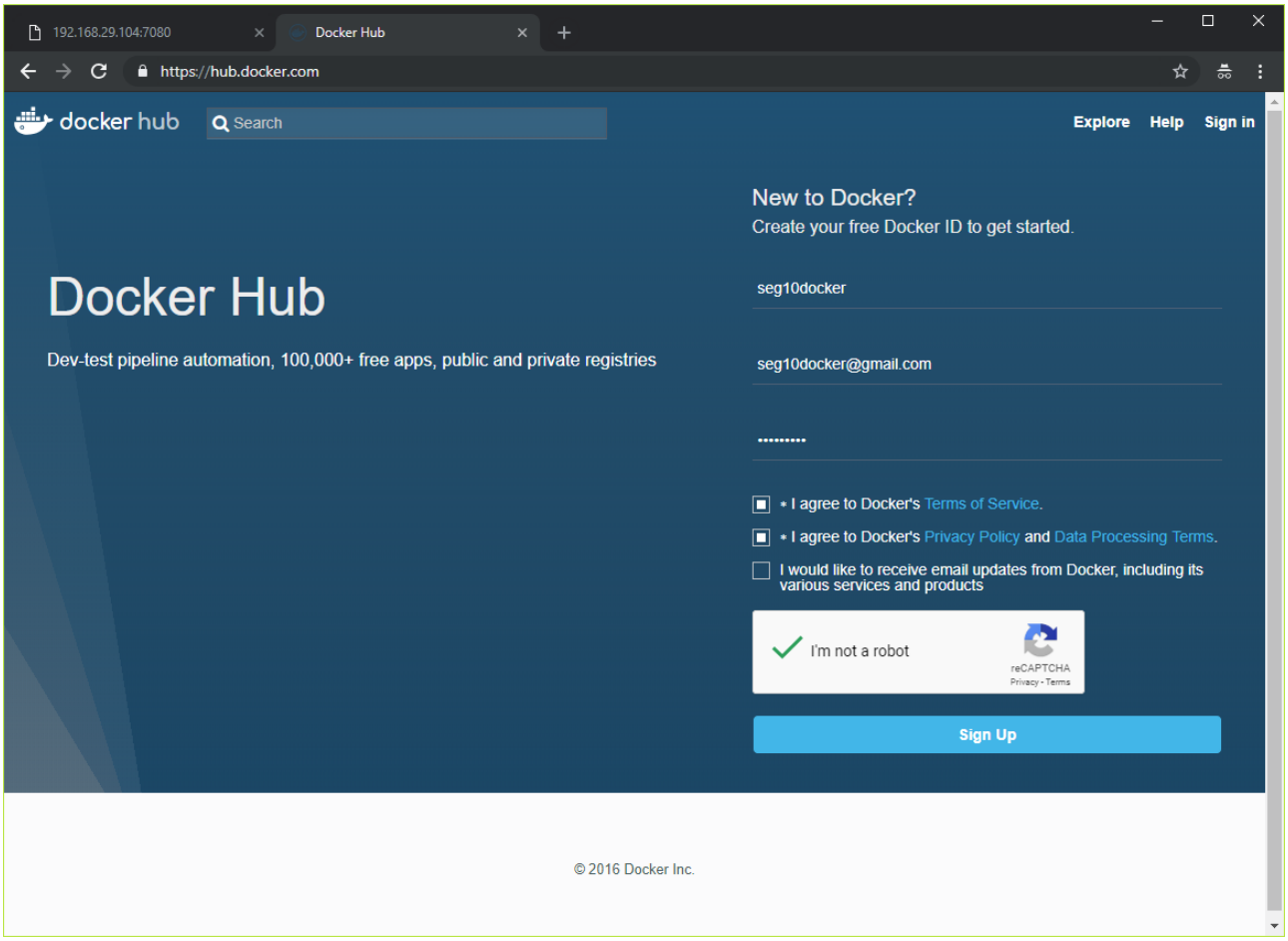

Figura 2. Cadastro no Docker Hub

Após seu cadastro, o Docker Hub irá enviar um e-mail de confirmação. Acesse a conta de e-mail informada e clique no botão para completar o cadastro. Finalmente, faça login no Docker Hub usando sua conta:

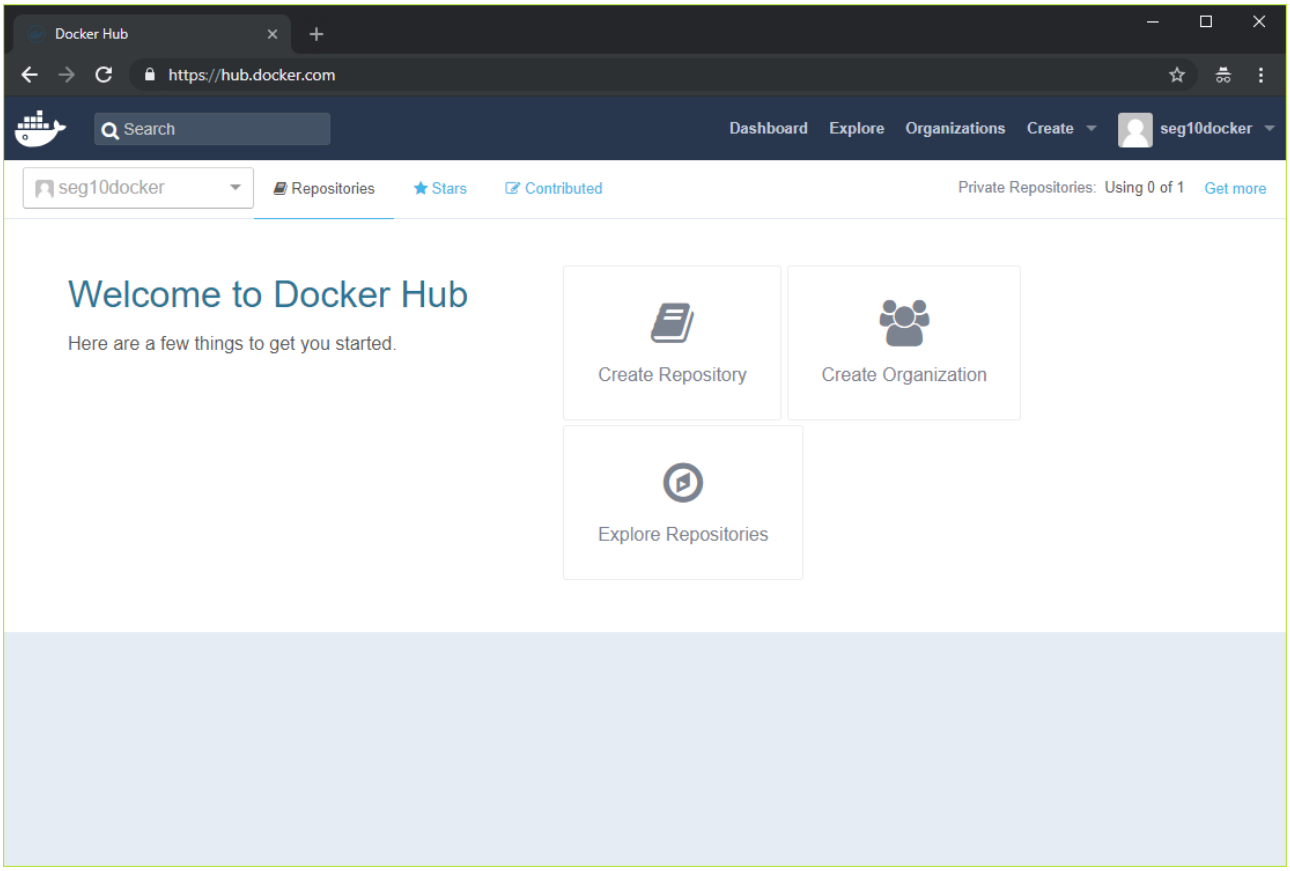

Figura 3. Interface do Docker Hub

Na tela acima, clique em *Create Repository*. Na nova tela, digite `seg10` como o nome do repositório; em *Description*, informe `Repositório-teste para o curso SEG10`; mantenha *Visibility* como `Public`.

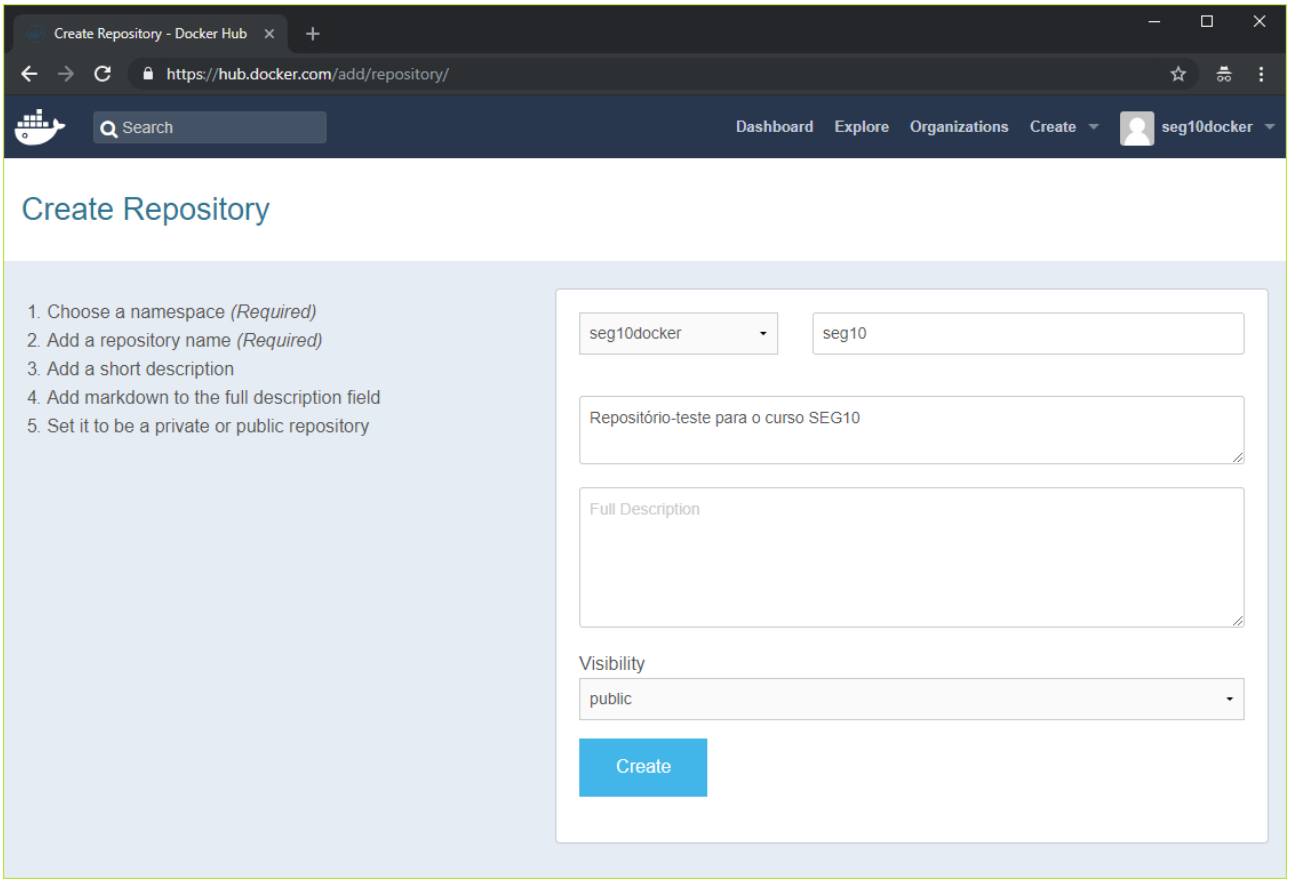

Figura 4. Criação de novo repositório no Docker Hub

Clique em *Create*. Concluído o processo, você verá seu novo repositório como na tela a seguir:

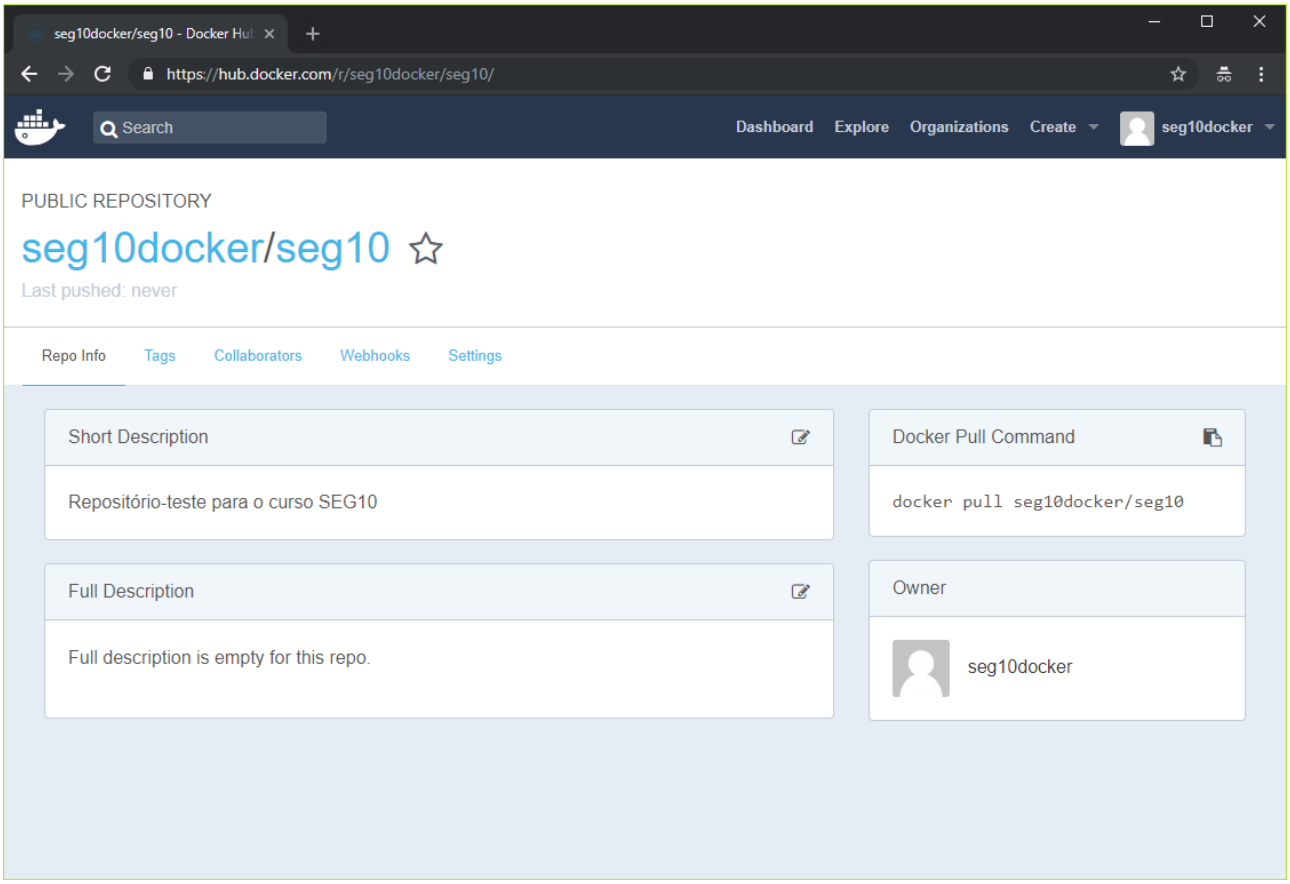

Figura 5. Repositório seg10 no Docker Hub

2. Agora, volte à máquina `docker1` como o usuário `root`.

```
# hostname ; whoami
docker1
root
```

Para fazer login no Docker Hub via linha de comando, use `docker login`. Use a mesma combinação de usuário e senha que você criou no passo (1) desta atividade.

```
# docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't
have a Docker ID, head over to https://hub.docker.com to create one.
Username: seg10docker
Password:
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

3. O próximo passo é criar uma *tag* para a imagem de container que criamos na atividade anterior. Uma *tag* é descrita por uma combinação `usuário/repositório:tag`, e serve para identificar e versionar imagens de containers no Docker.

Para criar a *tag*, use o comando `docker tag image` — substitua no comando abaixo o nome de usuário `seg10docker` pelo usuário que você criou no Docker Hub no passo (1) desta atividade:

```
# docker tag pyhello seg10docker/seg10:pyhello-v1
```

Para ver a nova imagem criada com a *tag*, use `docker image ls`:

```
# docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
seg10docker/seg10   pyhello-v1          d2923f9142e3        37 minutes ago
131MB
pyhello             latest              d2923f9142e3        37 minutes ago
131MB
python              2.7-slim            0dc3d8d47241        37 hours ago
120MB
hello-world         latest              4ab4c602aa5e        2 months ago
1.84kB
```

4. Para publicar a imagem à qual aplicamos a *tag* para o *registry* global do Docker Hub, use o comando `docker push`:

```
# docker push seg10docker/seg10:pyhello-v1
The push refers to repository [docker.io/seg10docker/seg10]
1efa6f0c4ef3: Pushed
2134161361ab: Pushed
1acdc2a51c84: Pushed
6cffeea81e5d: Mounted from library/python
614a79865f6d: Mounted from library/python
612d27bb923f: Mounted from library/python
ef68f6734aa4: Mounted from library/python
pyhello-v1: digest:
sha256:2879351d8aa37d80e5cebeb676f70af2a93de107d0b801bb51e47d937f99f3ff size: 1787
```

Concluído este processo, a imagem estará publicada e disponível no Docker Hub. De volta ao navegador em sua máquina física, acesse a aba *Tags* em seu repositório para visualizar a imagem que foi enviada:

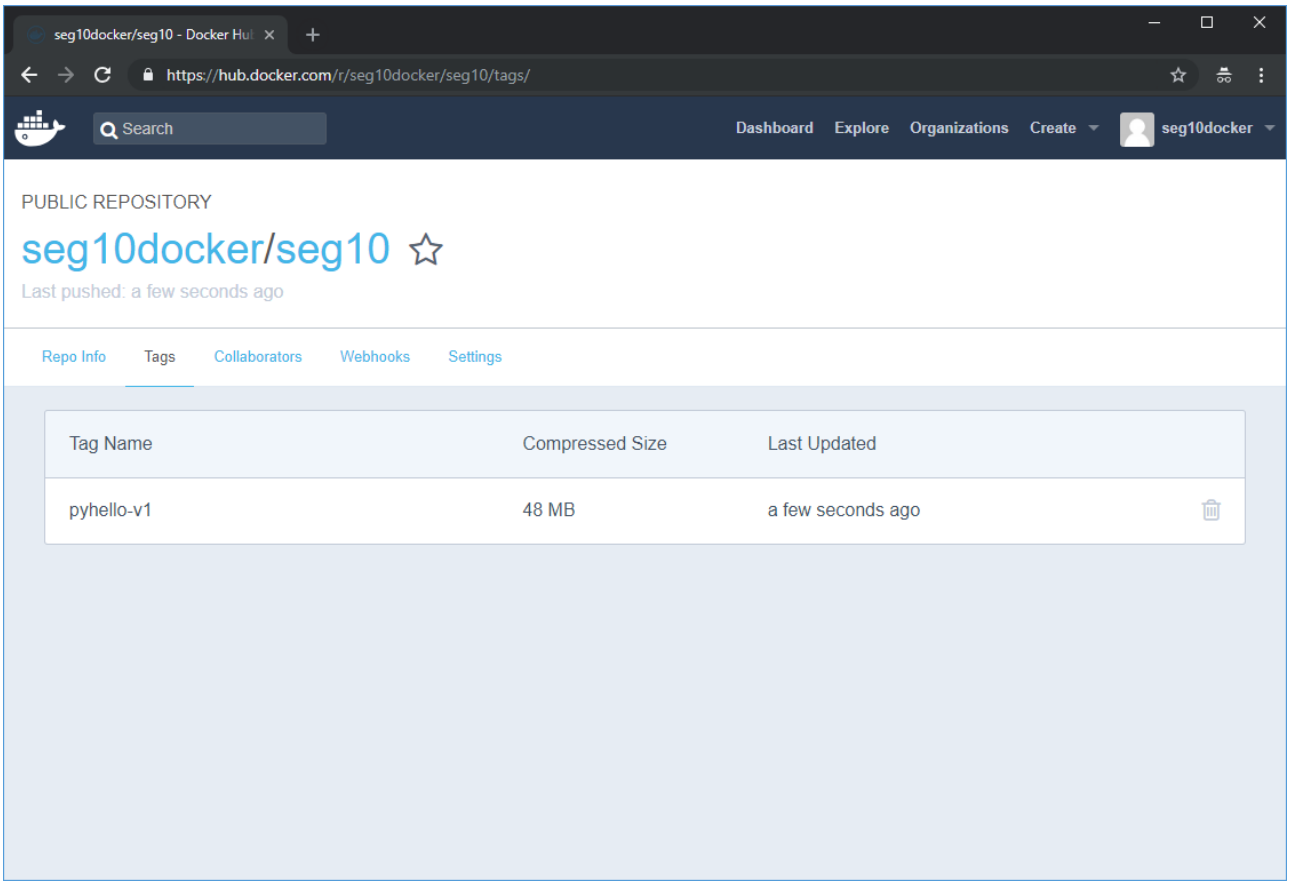

Figura 6. Imagem publicada no Docker Hub

5. A partir deste momento, é possível executar a imagem do container que publicamos para o Docker Hub a partir de qualquer máquina que possua o Docker instalado, diretamente. Por exemplo, acesse a máquina `docker2` como o usuário `root`:

```
# hostname ; whoami
docker2
root
```

Agora, execute a imagem com o comando:

```
# docker run -p 7080:80 seg10docker/seg10:pyhello-v1
Unable to find image 'seg10docker/seg10:pyhello-v1' locally
pyhello-v1: Pulling from seg10docker/seg10
a5a6f2f73cd8: Pull complete
8da2a74f37b1: Pull complete
09b6f498cfd0: Pull complete
f0afb4f0a079: Pull complete
b0ce05758094: Pull complete
dde7e744bb50: Pull complete
0662477f0e17: Pull complete
Digest: sha256:2879351d8aa37d80e5cebeb676f70af2a93de107d0b801bb51e47d937f99f3ff
Status: Downloaded newer image for seg10docker/seg10:pyhello-v1
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
```

Observe que o Docker detecta que a imagem não está disponível localmente, e então faz o download da mesma, instala as dependências do Python via `requirements.txt` e executa a aplicação. Tudo é gerenciado de forma transparente, de forma que apenas tivemos que invocar o container que preparamos e enviamos para o *registry* anteriormente.

Antes de prosseguir, encerre o container `CTRL + C`.

6) Construindo serviços com o Docker

1. Agora, vamos escalar a execução do nosso container para um ambiente simulado de produção. Acesse a máquina `docker1` como o usuário `root`:

```
# hostname ; whoami
docker1
root
```

2. Crie o arquivo novo `/root/docker/docker-compose.yml` com o conteúdo abaixo:

```
 1 version: "3"
 2 services:
 3   web:
 4     # substitua username/repo:tag com suas informacoes de nome de usuario,
repositorio e imagem
 5     image: username/repo:tag
 6     deploy:
 7       replicas: 5
 8       resources:
 9         limits:
10           cpus: "0.1"
11           memory: 50M
12       restart_policy:
13         condition: on-failure
14     ports:
15       - "7080:80"
16     networks:
17       - webnet
18 networks:
19   webnet:
```

Como mencionado no comentário acima, não se esqueça de substituir a *string* `username/repo:tag`, que identifica a imagem a ser executada, pelas informações de usuário, repositório e imagem que foram criadas por você na atividade anterior. Por exemplo, no caso do usuário `seg10docker` que foi ilustrado até aqui, a linha ficaria assim:

```
# grep 'image:' ~/docker/docker-compose.yml
    image: seg10docker/seg10:pyhello-v1
```

O arquivo acima irá:

- Baixar a imagem que enviamos para o *registry* global do Docker Hub, se necessário.
- Rodar 5 instâncias de um serviço denominado `web`, com cada instância ocupando no máximo 10% de CPU e 50 MB de memória RAM.
- Reiniciar imediatamente quaisquer containers que venham a falhar.
- Mapear a porta 7080 do *host* Docker para a porta 80 do container.
- Instruir os containers do serviço `web` a compartilhar a porta 80 através de uma rede com balanceador de carga denominada `webnet`.
- Definir a rede `webnet` com opções padrão (no caso, uma rede com balanceador de carga em *overlay*).

3. Vamos testar? Primeiro, temos que iniciar o *swarm*, que consiste em um conjunto de máquinas rodando o Docker que operam em *cluster*. Como, neste momento, apenas a máquina `docker1` integrará esse *cluster*, ela atuará como o administrador do *swarm*.

```
# docker swarm init
Swarm initialized: current node (1iys4vxt3k1pkcasdufb4m5vc) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-
3jpdh6q1i5a9fay1y3nwavb18enoqhujwxk9bgin2k4as1p38z-6ft56u5yq0zxpc7wcsmvgzsux
10.0.42.9:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow
the instructions.
```

Agora sim, vamos iniciar a *stack* do serviço. Iremos nomeá-la `pyhello-stack`:

```
# cd ~/docker ; docker stack deploy -c docker-compose.yml pyhello-stack
Creating network pyhello-stack_webnet
Creating service pyhello-stack_web
```

A partir desse momento, a *stack* do serviço está rodando 5 instâncias de containers da imagem `pyhello-v1`.

4. Verifique se, de fato, todos esses containers estão rodando com `docker service ls`:

```
# docker service ls
ID                  NAME                MODE                REPLICAS
IMAGE                          PORTS
nrglkuxxdyer        pyhello-stack_web   replicated          5/5
seg10docker/seg10:pyhello-v1   *:7080->80/tcp
```

O serviço reporta que há 5 réplicas operando. Para visualizá-las individualmente, use `docker service ps`:

```
# docker service ps pyhello-stack_web
ID                  NAME                  IMAGE                          NODE
DESIRED STATE       CURRENT STATE           ERROR                PORTS
x69lu01rscpt        pyhello-stack_web.1   seg10docker/seg10:pyhello-v1   docker1
Running             Running 2 minutes ago
i0id1ygbu7ba        pyhello-stack_web.2   seg10docker/seg10:pyhello-v1   docker1
Running             Running 2 minutes ago
uiu2pvojpzvw        pyhello-stack_web.3   seg10docker/seg10:pyhello-v1   docker1
Running             Running 2 minutes ago
9tpopx8y3prr        pyhello-stack_web.4   seg10docker/seg10:pyhello-v1   docker1
Running             Running 2 minutes ago
wp539pcus74u        pyhello-stack_web.5   seg10docker/seg10:pyhello-v1   docker1
Running             Running 2 minutes ago
```

Também é possível listar todos os containers operando via `docker container ls`:

```
# docker container ls -q
f12b9b2db505
81c893056bfa
4b0c462de8f5
70b9a762aed2
0e4742201217
```

No navegador em sua máquina física, acessando o IP da interface `enp0s3` da máquina `ns1`, porta 7080, solicite o recarregamento da página diversas vezes com o atalho `F5`. Note como, a cada *refresh*, o ID do container muda no campo *Hostname*.

5. Suponhamos que temos um pico de acessos, e é necessário aumentar de 5 para 8 o número de réplicas de container ativas. O Docker permite que façamos isso sem ter que reiniciar o serviço, veja: primeiro, edite o arquivo `/root/docker/docker-compose.yml`.

```
# sed -i 's/^\([[:space:]]*replicas:\).*/\1 8/' ~/docker/docker-compose.yml
```

```
# grep replicas ~/docker/docker-compose.yml
      replicas: 8
```

Agora, faça o *redeploy* do serviço:

```
# cd ~/docker ; docker stack deploy -c docker-compose.yml pyhello-stack
Updating service pyhello-stack_web (id: nrglkuxxdyer9andac5feb51t)
```

Note que o número de réplicas de containers aumenta imediatamente:

```
# docker service ls
ID                  NAME                MODE                REPLICAS
IMAGE                          PORTS
nrglkuxxdyer        pyhello-stack_web   replicated          8/8
seg10docker/seg10:pyhello-v1   *:7080->80/tcp
```

```
# docker container ls -q
fc5d90deda22
afeb8dc18eea
bb599b864df5
f12b9b2db505
81c893056bfa
4b0c462de8f5
70b9a762aed2
0e4742201217
```

6. Para interromper o serviço, remove a *stack* com o comando `docker stack rm`:

```
# docker stack rm pyhello-stack
Removing service pyhello-stack_web
Removing network pyhello-stack_webnet
```

Depois, abandone o *swarm* usando `docker swarm leave`. Como a máquina `docker1` é um administrador do *swarm*, temos que usar o parâmetro `--force`:

```
# docker swarm leave --force
Node left the swarm.
```

Observe que todos os containers foram encerrados, como esperado.

```
# docker container ls -q | wc -l
0
```

7) Operando com múltiplos membros no cluster

1. Operar com múltiplas máquinas no *cluster* (ao invés de apenas uma, como fizemos com a máquina `docker1` na atividade anterior) é bastante fácil. Primeiro, acesse a máquina `docker1` como `root`:

```
# hostname ; whoami
docker1
root
```

2. Inicie o *swarm* como fizemos anteriormente, com o comando `docker swarm init`:

```
# docker swarm init
Swarm initialized: current node (k0nruds0qup7nscmf0j43jb30) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-
3mw379ioa403z9kpu2rfbnjdzct1o4p33xh83zngsyey0rw9lp-afg4svcx1rjprgvug53vm21v4
10.0.42.9:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow
the instructions.
```

Observe o que fala a segunda linha do *output* acima: para adicionar novas máquinas ao *swarm*, execute o comando abaixo na máquina-alvo. Vamos fazer exatamente isso.

3. Acesse a máquina `docker2` como `root`.

```
# hostname ; whoami
docker2
root
```

Copie o comando `docker swarm join` mostrado no *output* do passo (2), acima, e execute-o na máquina `docker2`:

```
# docker swarm join --token SWMTKN-1-
3mw379ioa403z9kpu2rfbnjdzct1o4p33xh83zngsyey0rw9lp-afg4svcx1rjprgvug53vm21v4
10.0.42.9:2377
This node joined a swarm as a worker.
```

Pronto! As máquinas `docker1` e `docker2` estão agora juntas no *cluster*, sendo a máquina `docker1` o *manager* e a `docker2` o *worker* nesse cenário.

4. Volte à máquina `docker1`, como `root`, e faça o *deploy* do serviço `pyhello-stack`:

```
# hostname ; whoami
docker1
root
```

```
# cd ~/docker ; docker stack deploy -c docker-compose.yml pyhello-stack
Creating network pyhello-stack_webnet
Creating service pyhello-stack_web
```

Verifique quantos containers estão executando na máquina `docker1`:

```
# docker container ls -q | wc -l
4
```

Ué, apenas 4 containers? Onde estão os outros 4? O comando `docker service ps` nos dá uma visão mais ampla da situação:

```
# docker service ps pyhello-stack_web
ID                  NAME                  IMAGE                          NODE
DESIRED STATE       CURRENT STATE            ERROR                PORTS
pmshw6028oin        pyhello-stack_web.1   seg10docker/seg10:pyhello-v1   docker2
Running             Running 42 seconds ago
ilcy0qubdj7v        pyhello-stack_web.2   seg10docker/seg10:pyhello-v1   docker1
Running             Running 44 seconds ago
qm8frg324qjj        pyhello-stack_web.3   seg10docker/seg10:pyhello-v1   docker2
Running             Running 42 seconds ago
vdijie0369g8        pyhello-stack_web.4   seg10docker/seg10:pyhello-v1   docker1
Running             Running 43 seconds ago
nqt5l1sm678e        pyhello-stack_web.5   seg10docker/seg10:pyhello-v1   docker2
Running             Running 43 seconds ago
8czx47nhep2n        pyhello-stack_web.6   seg10docker/seg10:pyhello-v1   docker1
Running             Running 44 seconds ago
ryrf4ovcq7d6        pyhello-stack_web.7   seg10docker/seg10:pyhello-v1   docker2
Running             Running 42 seconds ago
4fr3zlwdgqjg        pyhello-stack_web.8   seg10docker/seg10:pyhello-v1   docker1
Running             Running 44 seconds ago
```

Veja que temos 4 containers rodando na máquina `docker1`, e outros 4 rodando na máquina `docker2`.

5. Agora, pode surgir uma questão em sua mente: "Ora, se as configurações que fizemos no firewall instruem o repasse de pacotes na porta 7080/TCP diretamente para a máquina `docker1`, então é evidente que os 4 containers rodando na máquina `docker2` estão inacessíveis, pelo menos até que corrijamos as regras de firewall, certo?"

   Será mesmo? Na máquina `docker2`, como `root`, liste os containers em operação:

```
# hostname ; whoami
docker2
root
```

```
# docker container ls
CONTAINER ID        IMAGE                          COMMAND             CREATED
STATUS              PORTS
              NAMES
37bbca732ec7        seg10docker/seg10:pyhello-v1   "python app.py"     6 minutes
ago       Up 6 minutes        80/tcp
              pyhello-stack_web.7.ryrf4ovcq7d6cvrxyoyh372kk
ab9171ebcf69        seg10docker/seg10:pyhello-v1   "python app.py"     6 minutes
ago       Up 6 minutes        80/tcp
              pyhello-stack_web.5.nqt5l1sm678e96ctsv2kwuk9f
f0d973c6c635        seg10docker/seg10:pyhello-v1   "python app.py"     6 minutes
ago       Up 6 minutes        80/tcp
              pyhello-stack_web.1.pmshw6028oinz61bh509suza3
c420565a43f1        seg10docker/seg10:pyhello-v1   "python app.py"     6 minutes
ago       Up 6 minutes        80/tcp
              pyhello-stack_web.3.qm8frg324qjj5roggzfwwznbn
```

Agora, em seu navegador na máquina física, recarregue a página algumas vezes e observe os IDs de container que são mostrados:

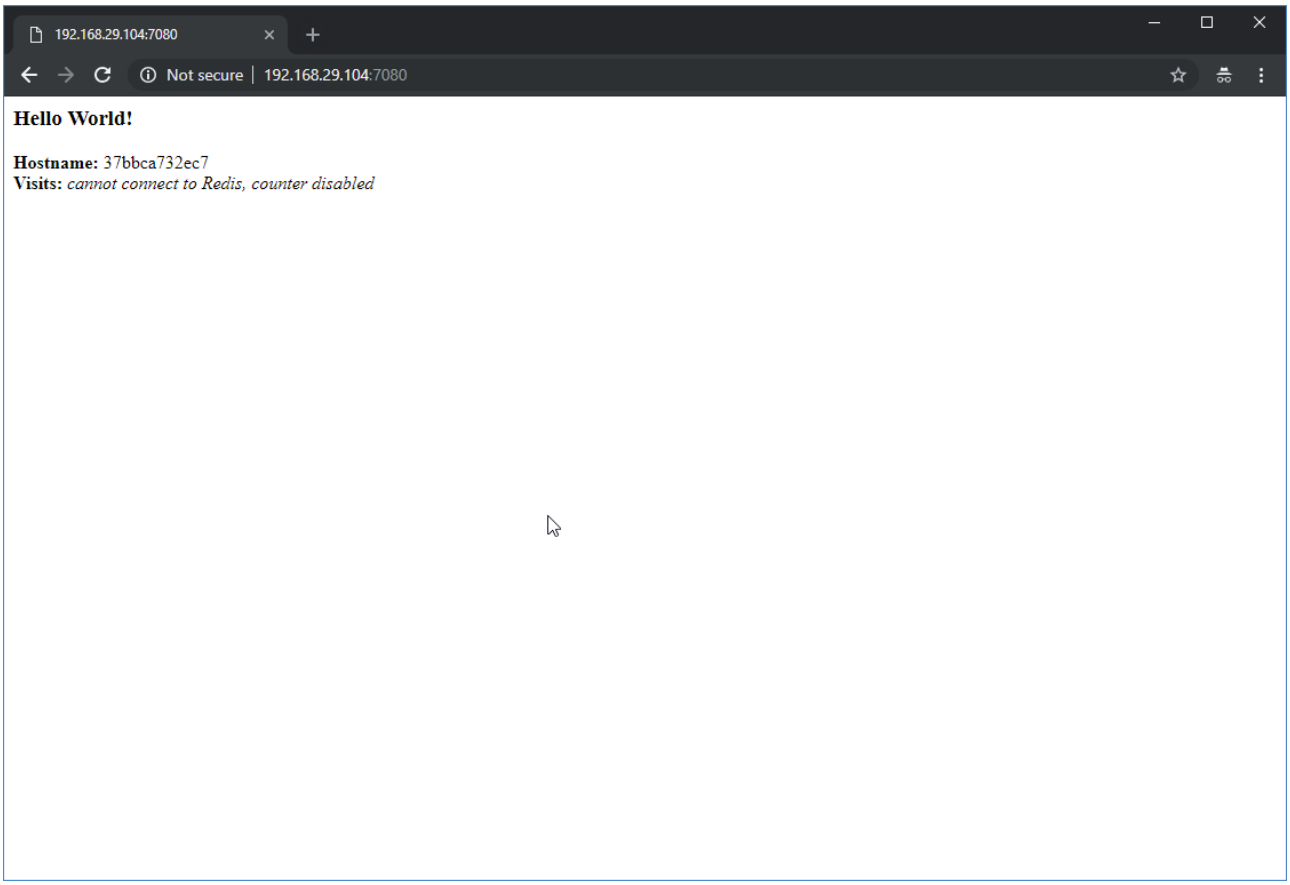

Figura 7. Container da máquina docker2 sendo acessado

Note que o ID de container mostrado acima está executando na máquina `docker2`, mas claramente é impossível que tenhamos acessado essa máquina, já que as regras de firewall criadas anteriormente redirecionam pacotes **apenas** para a máquina `docker1`. Como isso está acontecendo?

A razão pela qual as duas máquinas estão acessíveis é porque um nodo do *swarm* Docker participa de um roteamento ingresso do tipo *mesh*, como ilustrado pela figura a seguir. Esse roteamento garante que um serviço alocado a uma porta em seu *swarm* sempre terá essa porta reservada para si, independente de qual nodo esteja rodando o container. Note, no exemplo, que é perfeitamente possível que uma requisição chegue à máquina `docker1` mas seja atendida por um container em `docker2`, ou vice-versa.

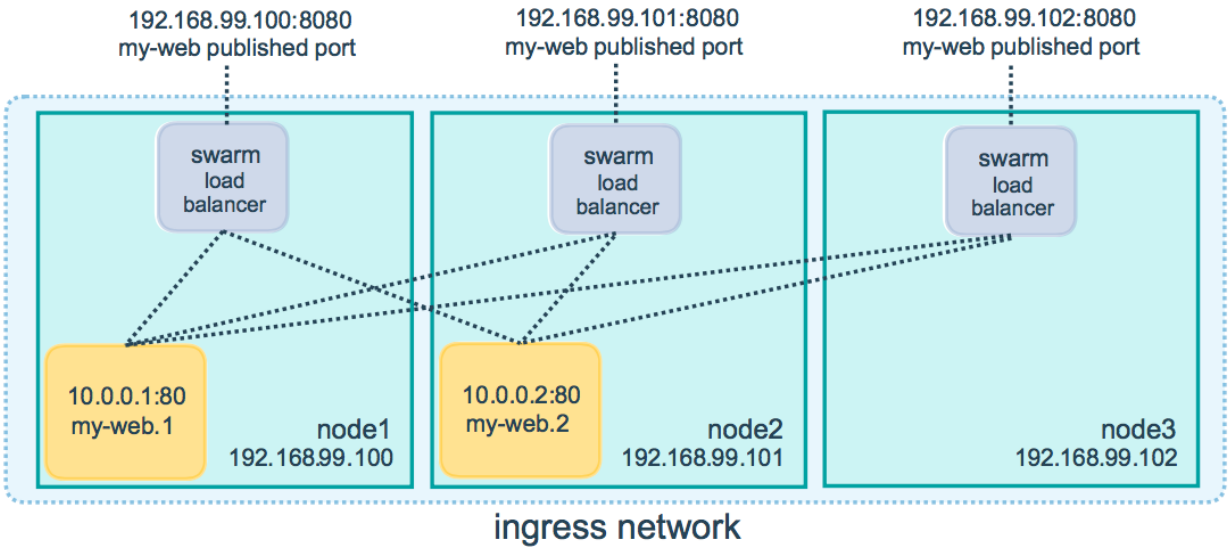

Figura 8. Roteamento do tipo mesh no Docker

6. Mesmo com a funcionalidade acima, é interessante que o firewall envie requisições para ambas as máquinas `docker1` e `docker2`, indistintamente. Volte à máquina `ns1`, como `root`.

```
# hostname ; whoami
ns1
root
```

Apague as regras que havíamos criado anteriormente.

```
# iptables -t nat -D PREROUTING -i enp0s3 -p tcp -m tcp --dport 7080 -j DNAT --to
-destination 10.0.42.9
```

```
# iptables -D FORWARD -i enp0s3 -d 10.0.42.9/32 -p tcp -m tcp --dport 7080 -j
ACCEPT
```

Em seu lugar, insira regras que redirecionem o tráfego para ambas as máquinas, em modalidade *round robin*:

```
# iptables -t nat -A PREROUTING -i enp0s3 -p tcp -m tcp --dport 7080 -j DNAT --to
-destination 10.0.42.9-10.0.42.10
```

```
# iptables -A FORWARD -i enp0s3 -d 10.0.42.9/32,10.0.42.10/32 -p tcp -m tcp --dport
7080 -j ACCEPT
```

Em seu navegador na máquina física, verifique que o serviço continua ativo, recarregando a página algumas vezes.

8) Adicionando novos serviços ao cluster

1. É bastante fácil adicionar novos serviços a um *cluster* Docker, mesmo quando em operação. Acesse a máquina `docker1` como `root`:

```
# hostname ; whoami
docker1
root
```

2. Vamos adicionar um serviço *visualizer*, um container pré-pronto do Docker que permite que observemos como está a alocação de containers em nosso *cluster*. Edite o arquivo de configuração do *swarm*, `/root/docker/docker-compose.yml`, com o comando `sed` a seguir:

```
# sed -i '/^networks:$/i\
  visualizer:\
    image: dockersamples/visualizer:stable\
    ports:\
      - "8080:8080"\
    volumes:\
      - "/var/run/docker.sock:/var/run/docker.sock"\
    deploy:\
      placement:\
        constraints: [node.role == manager]\
    networks:\
      - webnet' ~/docker/docker-compose.yml
```

O comando acima irá:

- Adicionar um novo serviço, `visualizer`, usando a imagem de container `dockersamples/visualizer:stable` baixada do *registry* global do Docker Hub.
- Mapear a porta externa 8080 para a porta interna 8080, dentro do contexto do container.
- Criar um mapeamento de volume do *socket* `/var/run/docker.sock` na máquina *host* para o mesmo caminho dentro do container. Vale observar que este *socket* está disponível exclusivamente no nodo *manager* do *cluster*.
- Por esse motivo, cria-se uma restrição de alocação desse container, que deve rodar exclusivamente em nodos que possuam a *role* de *manager* no *cluster*.
- Finalmente, conecta-se o serviço à mesma rede `webnet` que havíamos criado antes.

3. Para lançar o novo serviço, basta executar um *redeploy* da *stack*:

```
# cd ~/docker ; docker stack deploy -c docker-compose.yml pyhello-stack
Creating service pyhello-stack_visualizer
Updating service pyhello-stack_web (id: fld46go1hn34kirew3xn3019v)
```

   O Docker detecta que a imagem `dockersamples/visualizer:stable` está indisponível localmente, e faz o download da mesma do *registry* global, lançando-a em seguida.

4. Presumindo que o serviço está ativo, note que estamos fazendo um novo mapeamento de portas: da 8080 (externa) para a 8080 (interna). Temos, naturalmente, que fazer ajustes em nosso firewall de rede. Acesse a máquina `ns1` como `root`:

```
# hostname ; whoami
ns1
root
```

   Apague as regras recém-criadas:

```
# iptables -t nat -D PREROUTING -i enp0s3 -p tcp -m tcp --dport 7080 -j DNAT --to
-destination 10.0.42.9-10.0.42.10
```

```
# iptables -D FORWARD -i enp0s3 -d 10.0.42.9/32,10.0.42.10/32 -p tcp -m tcp --dport
7080 -j ACCEPT
```

   Torne-as mais abrangentes, incluindo também a porta 8080:

```
# iptables -t nat -A PREROUTING -i enp0s3 -p tcp -m multiport --dports 7080,8080 -j
DNAT --to-destination 10.0.42.9-10.0.42.10
```

```
# iptables -A FORWARD -i enp0s3 -d 10.0.42.9/32,10.0.42.10/32 -p tcp -m multiport
--dports 7080,8080 -j ACCEPT
```

   Agora sim, sendo estas regras definitivas, grave-as na configuração do firewall local:

```
# /etc/init.d/netfilter-persistent save
[....] Saving netfilter rules...run-parts: executing /usr/share/netfilter-
persistent/plugins.d/15-ip4tables save
run-parts: executing /usr/share/netfilter-persistent/plugins.d/25-ip6tables save
done.
```

5. Em sua máquina física, aponte agora o navegador para o IP público do *datacenter*, o endereço

do interface `enp0s3` da máquina `ns1` na porta 8080:

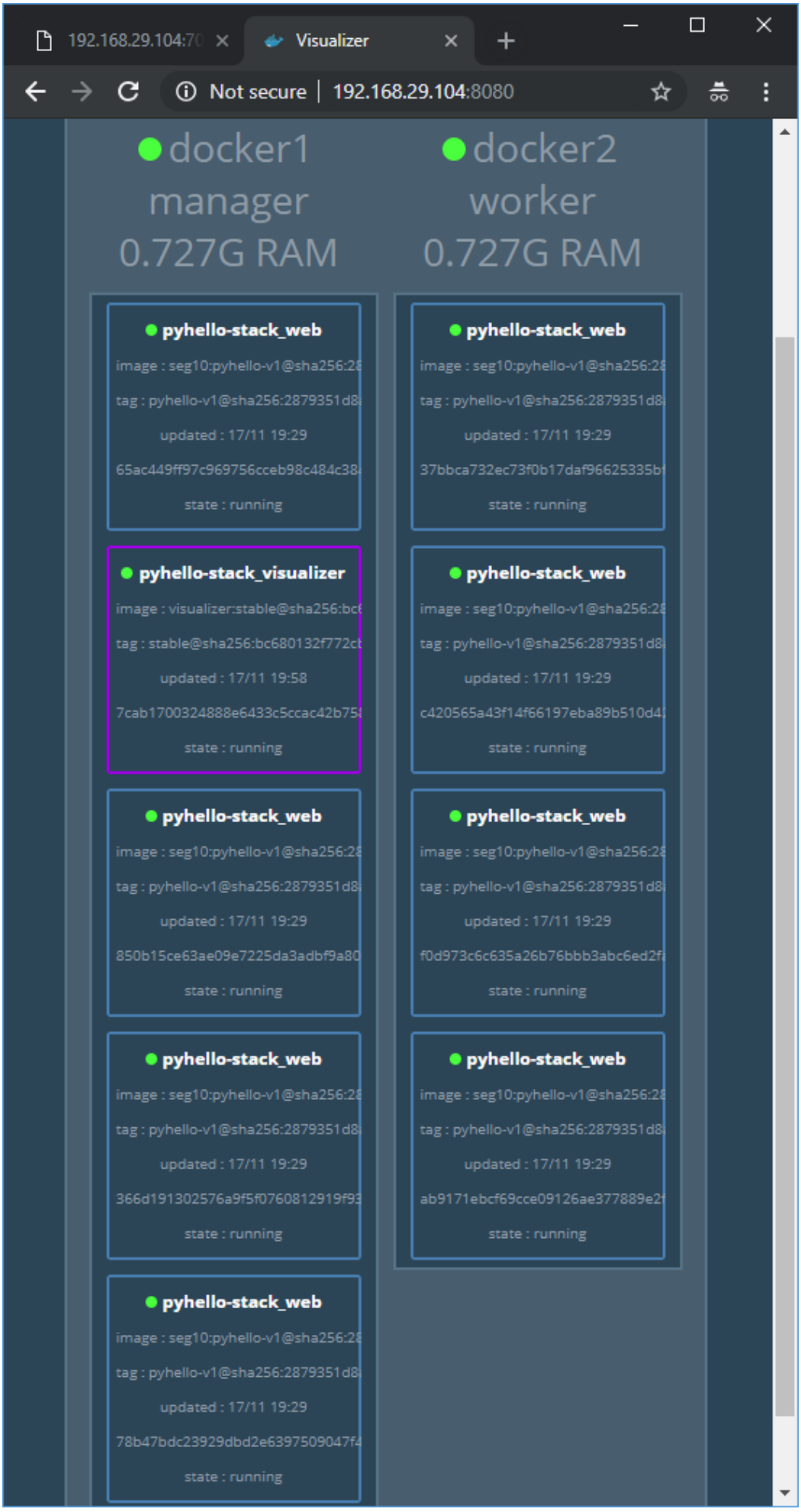

Figura 9. Visualizando a operação do cluster Docker

O container `visualizer` nos mostra, de forma fácil, como está o estado do *cluster* Docker. Note como o container `visualizer` está operando apenas na máquina `docker1`, já que ela é o *manager* do *swarm*. Os containers `web`, por outro lado, podem operar em qualquer nodo, estando distribuídos 4 a cada lado.

9) Configurando a persistência dos dados

1. Você deve ter notado que o contador de visitantes do website, por algum motivo, não está funcionando. Isso se deve ao fato que o serviço do Redis, que fornece uma espécie de banco de dados em arquivo, não está operacional. Vamos corrigir isso: acesse a máquina `docker1` como `root`.

```
# hostname ; whoami
docker1
root
```

2. Para adicionar um serviço para o Redis, edite o arquivo de configuração `/root/docker/docker-compose.yml` com o comando `sed` a seguir:

```
# sed -i '/^networks:$/i\
  redis:\
    image: redis\
    ports:\
      - "6379:6379"\
    volumes:\
      - "/root/data:/data"\
    deploy:\
      placement:\
        constraints: [node.role == manager]\
    command: redis-server --appendonly yes\
    networks:\
      - webnet' ~/docker/docker-compose.yml
```

O comando acima irá:

- Adicionar um novo serviço, `redis`, usando a imagem de container `redis` baixada do *registry* global do Docker Hub.
- Mapear a porta externa 6379 para a porta interna 6379, dentro do contexto do container. Como esse serviço não será acessado externamente, não será necessário criar novas regras no firewall `ns1`.
- Criar um mapeamento de volume do diretório `/root/data` na máquina *host* para o caminho `/data` dentro do container. A persistência de dados de vistantes do website será armazenada nesse diretório.
- Cria-se uma restrição de alocação desse container, que deve rodar exclusivamente em nodos que possuam a *role* de *manager* no *cluster*. Assim, todos os containers do *cluster* terão a

mesma visão dos dados em persistência.

- Finalmente, conecta-se o serviço à mesma rede `webnet` que havíamos criado antes.

Naturalmente, temos que criar o diretório `/root/data`, que ainda não existe:

```
# mkdir ~/data
```

3. Faça o *redeploy* da *stack*:

```
# cd ~/docker ; docker stack deploy -c docker-compose.yml pyhello-stack
Updating service pyhello-stack_visualizer (id: kzzk52hi201lz8jlsoif86p45)
Creating service pyhello-stack_redis
Updating service pyhello-stack_web (id: fld46go1hn34kirew3xn3019v)
```

4. Em sua máquina física, aponte o navegador para o IP público do *datacenter*, o endereço do interface `enp0s3` da máquina `ns1` na porta 7080. Note que o contador de visitas está, agora sim, sendo contabilizado corretamente:

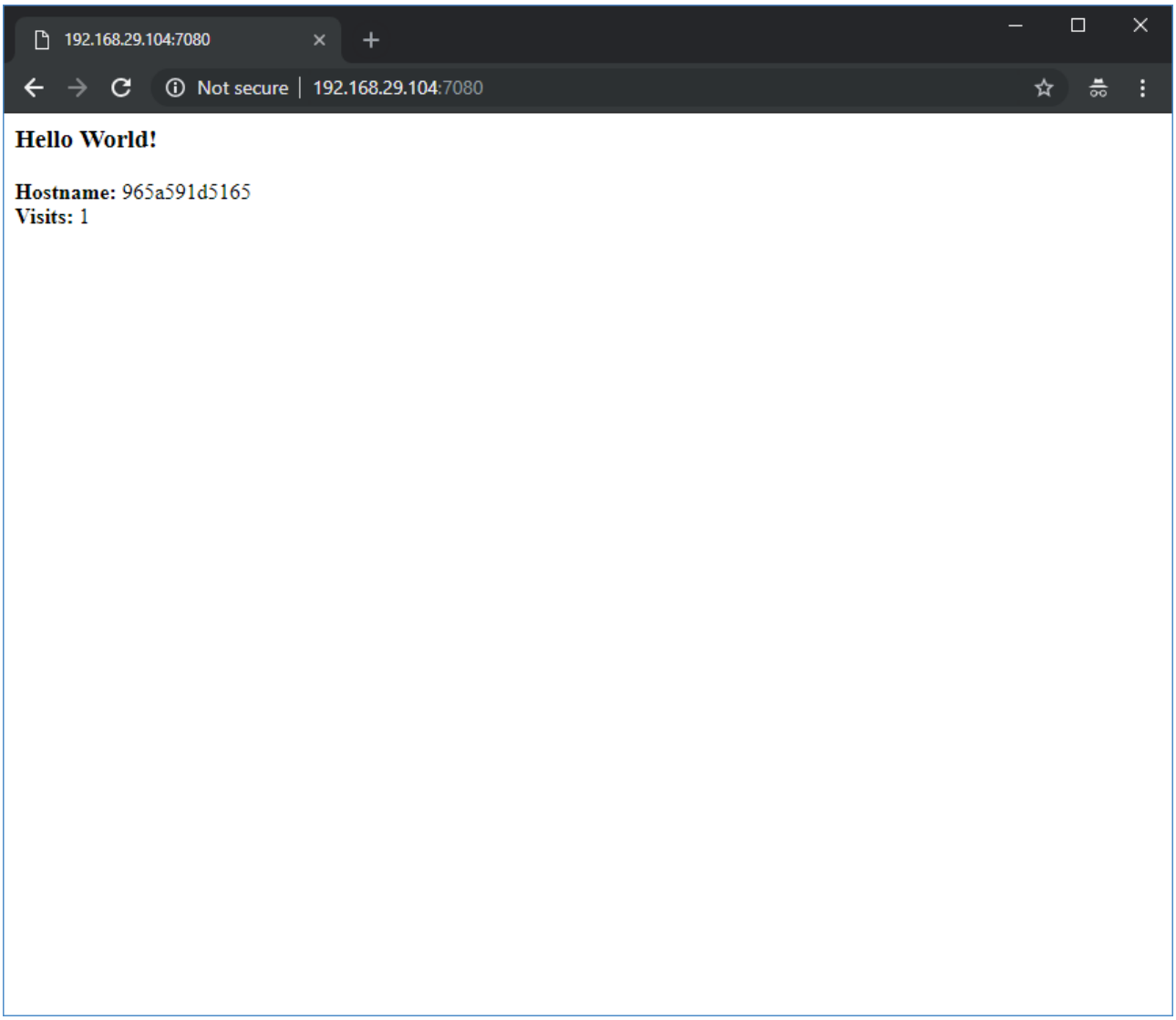

Figura 10. Contador de visitas operacional

Recarregue a página algumas vezes, e observe que o contador continua aumentando independentemente do fato de estarmos sendo atendidos por um container localizado na máquina `docker1` ou na máquina `docker2`. Como todos os containers tem a mesma visão da "verdade", isto é, a base de dados compartilhada no diretório `/root/data` da máquina `docker1`, conseguimos obter o valor correto e incrementá-lo sem importar de onde está partindo a requisição.

Observando o `visualizer`, que opera na porta 8080, note que o container do Redis executa exclusivamente na máquina `docker1`, como configurado:

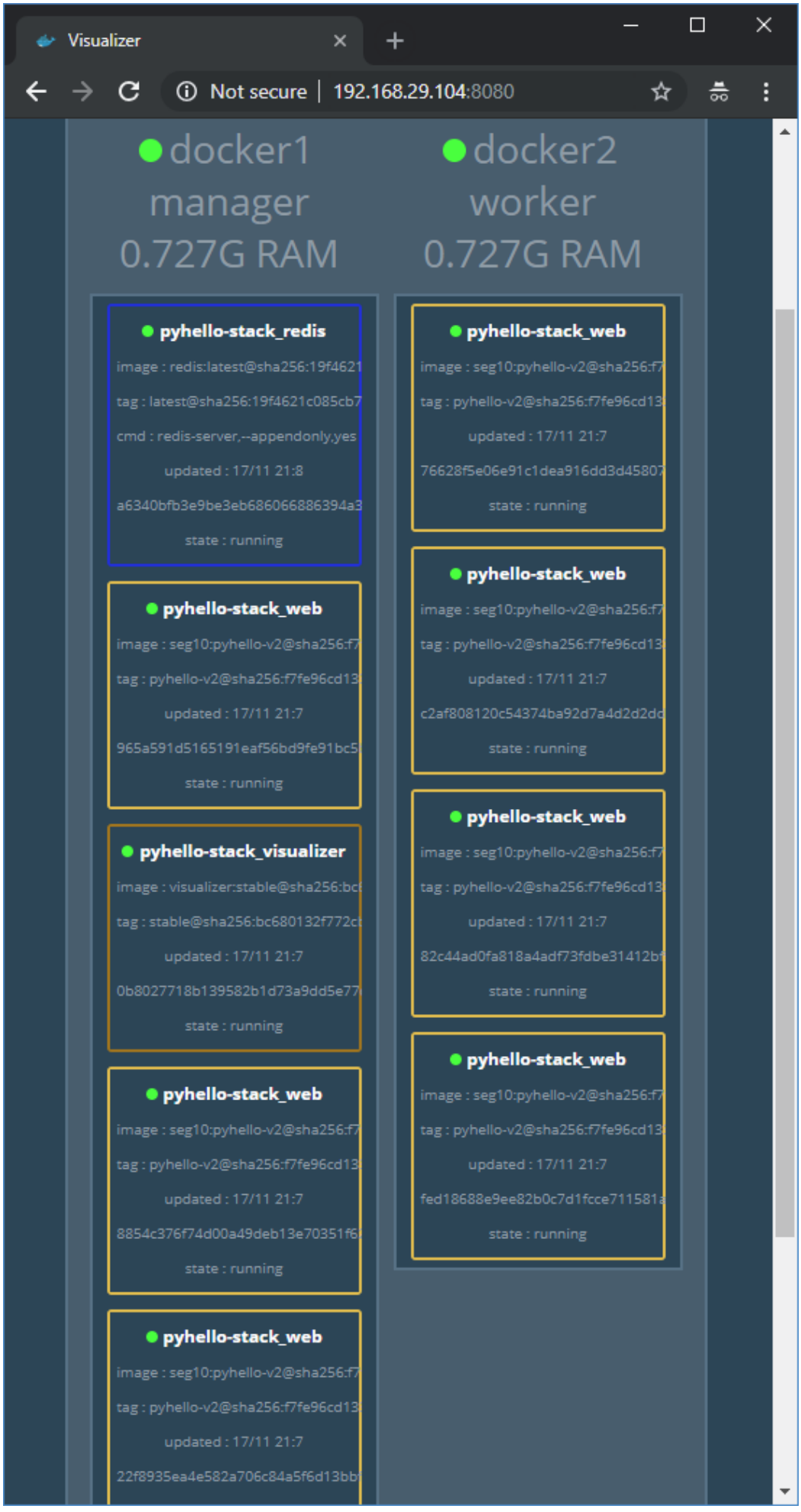

Figura 11. Máquina docker1 executando visualizador e Redis

5. Encerradas as nossas atividades com containers, encerre o *stack* e *swarm* em ambas as máquinas `docker1` e `docker2`, e desligue-as. Para manter reduzido o uso de recursos durante o decorrer das próximas sessões, é interessante que tenhamos o mínimo de VMs operacionais.