Tópicos Avançados em Arquiteturas Distribuídas de Software
Relatório Trabalho 1

Grupo 8: Lucas Penteado, Michele Argolo, Sofia Silveira

Primeiros Passos

Inicialmente, foi necessário executar os comandos a seguir para clonar o repositório `slice-enablers`. Esse passo é importante, pois é na pasta `arquivos` que se encontram as permissões necessárias para o acesso à cloud da UFSCar, sendo possível se conectar ao `ubuntu@master`.

```
$ cd ~
$ git clone https://github.com/dcomp-leris/slice-enablers.git
$ cd slice-enablers
$ chmod 400 arquivos/cloud_ufscar_rsa.dms
```

Em seguida, para a realização do trabalho o grupo criou um repositório no GitHub. O número do nosso grupo é o 8, porém a VM 8 estava com problemas, portanto passamos a utilizar a VM 12 para este trabalho. Para acessá-la quando estamos conectados na rede da própria UFSCar, campus Sorocaba, basta digitar no terminal:

```
$ bash acessar.sh 12
```

Já para acessá-la remotamente é necessário primeiro acessar a VM Master com o código no terminal:

```
$ chmod 400 arquivos/cloud_ufscar_rsa.dms
$ ssh -i cloud_ufscar_rsa.dms ubuntu@200.136.252.136
```

Em seguida, devemos ir até a pasta `slice-enablers` e, em seguida, até `arquivos` antes de acessar a VM específica do grupo (número 12):

```
$ cd slice-enablers/arquivos
$ bash acessar.sh 12
```

Após esses passos, precisamos clonar a pasta do repositório criado no GitHub dentro da nossa máquina virtual:

```
git clone https://github.com/lucaspsacchi/Trab1-TAAD.git
```

Infraestrutura Vagrant

Depois, fizemos a criação do arquivo Vagrantfile com as configurações das VMs, uma master com 4GB de memória e uma worker com 2GB de memória, com seus

respectivos IPs, e configuramos a rede como privada. Era necessário que a instalação do Docker fosse feita automaticamente após subir as VMs com o Vagrant, então, no Vagrantfile foi adicionada uma linha de código para cada uma das VMs, um para master e um para worker, para chamar um script externo que irá instalar o Docker:

```
containerhost01.vm.provision "shell", path:"../scripts/init_server.sh"
containerhost02.vm.provision "shell", path: "../scripts/init_client.sh"
```

Os arquivos `init_server.sh` e `init_client.sh` irão realizar a atualização padrão do sistema, clonar a pasta do repositório do GitHub em cada uma das VMs, fazer download e update do Docker e Docker Swarm. Em seguida, foi realizada a criação (build) da imagem do Docker. Dessa forma, são iniciadas as aplicações de servidor e cliente, respectivamente, dentro dos containers.

Inicialmente, estávamos chamando estes script por meio de `s.inline`. Porém, não estava sendo possível acessá-los dessa forma, já que não estávamos conseguindo passar o caminho correto. Então, por isso, decidimos mudar para `path`.

Dockerfile

No Dockerfile optamos por utilizar a imagem do Ubuntu, pois ela é muito mais leve do que a imagem de Python. Ambos Dockerfile, tanto do cliente quanto do servidor, são parecidos, porém o do cliente instala Python e biblioteca pip com `python-pip`, a biblioteca docker e Flask, enquanto o do servidor instala Python e Flask. Flask foi a framework de web escolhida por estar escrita em Python, já que essa foi a linguagem que decidimos utilizar para a implementação do projeto.

Aplicações client e server

A aplicação server é uma API REST e possui os métodos GET_INFO, GET_INFO/id e POST_INFO implementados no arquivo `server.py`, ou seja, este arquivo cria uma aplicação REST em Flask que retorna dados (métodos GET_INFO) ou insere os dados (método POST_INFO) com rotas específicas. O método GET_INFO se difere do método GET_INFO/id por apresentar toda a lista de dados, enquanto que o GET_INFO/id retorna apenas aquele com o id passado. Já o client, possui o arquivo `interface.py`, que possui a interface para acessar as 3 possíveis rotas, isto é, duas por GET e uma por POST.

O grupo optou por não utilizar uma base de dados para armazenar os dados, mas sim utilizar tasks que na verdade são dicionários, sendo possível guardar o id da task, id do container, timestamp, uso da CPU e uso de memória.

No início, estávamos tentando executar o código `info.py` dentro do container, porém de dentro dele não era possível obter as informações do `docker stats`. Após tentar de várias formas e falhar, o grupo conversou com o Professor André e decidiu executar o codigo `info.py` fora do container no worker para que fosse possível pegar externamente as informações do docker stats e realizar o método POST.