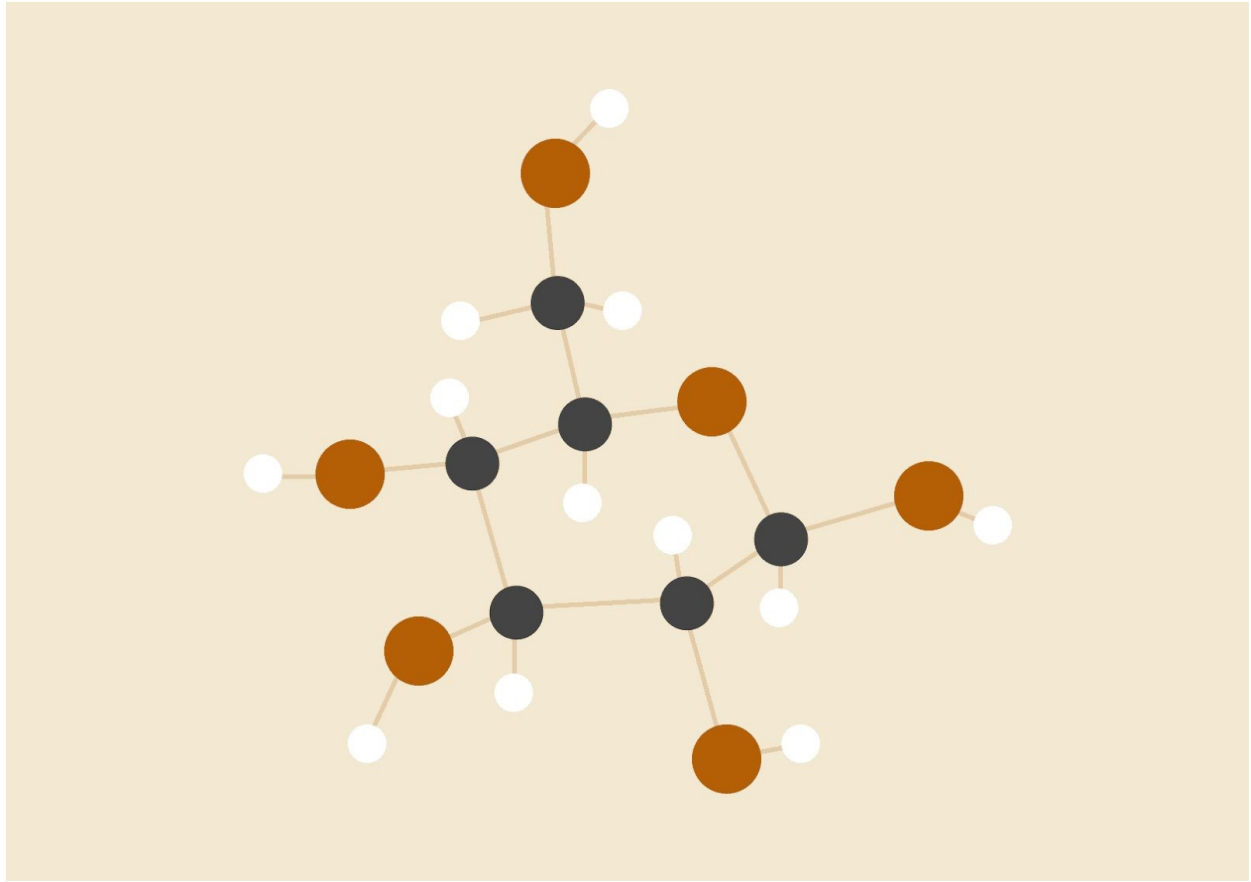


# Apache Mesos

*Uma introdução ao Apache Mesos*



**Eli Jose Abi Ghosn, Pedro Azambuja e Rafael Vieira**

18.11.2019

Megadados 6º Semestre

## INTRODUÇÃO

Aplicações/Sistemas distribuídos estão cada vez mais populares, uma vez que as aplicações sendo desenvolvidas atualmente não cabem mais em um único computador. Uma das explicações para isso é a aplicação gerar uma quantidade muito grande de dados ou possuir muitos usuários pelo mundo. Quando o hardware falha em acompanhar as crescentes demandas de CPU, memória e espaço em disco, deve se recorrer às tecnologias da computação em nuvem.

Nesse contexto, o Apache Mesos se apresenta como uma solução de gerenciamento de componentes distribuídos em rede.

## O QUE É

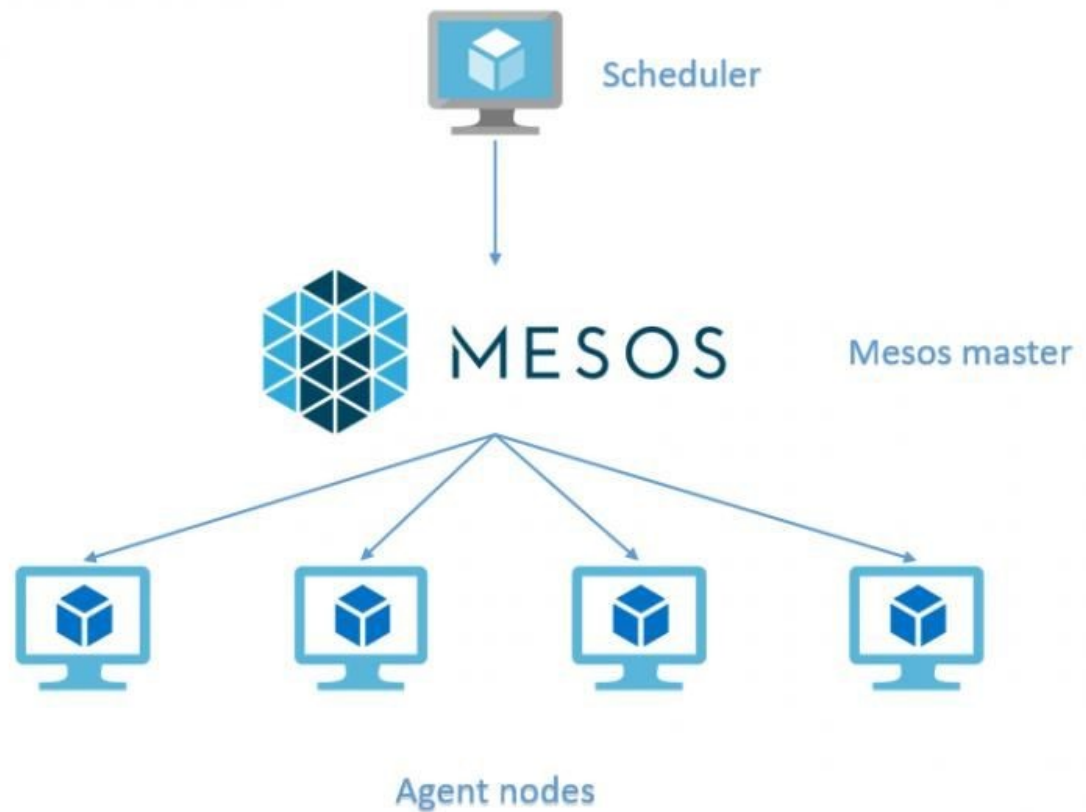
O Apache Mesos é um gerenciador de cluster open source que ajuda a administrar os clusters, fornecendo alocação eficiente de recursos e isolamento entre os componentes de sistemas/aplicações distribuídas.

Algumas funcionalidades do Mesos incluem escalabilidade de até 10.000 nodes, isolamento de recursos para tarefas por meio de containers Linux, agendamento de recursos de processamento e memória de forma eficiente, alta disponibilidade do nó master por meio do Apache ZooKeeper e interface gráfica em web para monitorar o estado do cluster e das tarefas.

Também é importante notar que o Mesos é o oposto da virtualização, porque na virtualização um recurso físico é dividido em múltiplos recursos virtuais, enquanto no Mesos, múltiplos recursos físicos são concatenados em um único recurso virtual.

## COMO FUNCIONA

O Mesos introduz uma camada de abstração voltada ao gerenciamento dos recursos físicos. No diagrama mostrado abaixo, o scheduler está sincronizado com “n” nodes agentes. Ao introduzir o Mesos, ao invés do scheduler se comunicar diretamente com todos os nodes, ele irá apenas se comunicar com o Mesos, o qual irá administrar todas as máquinas.

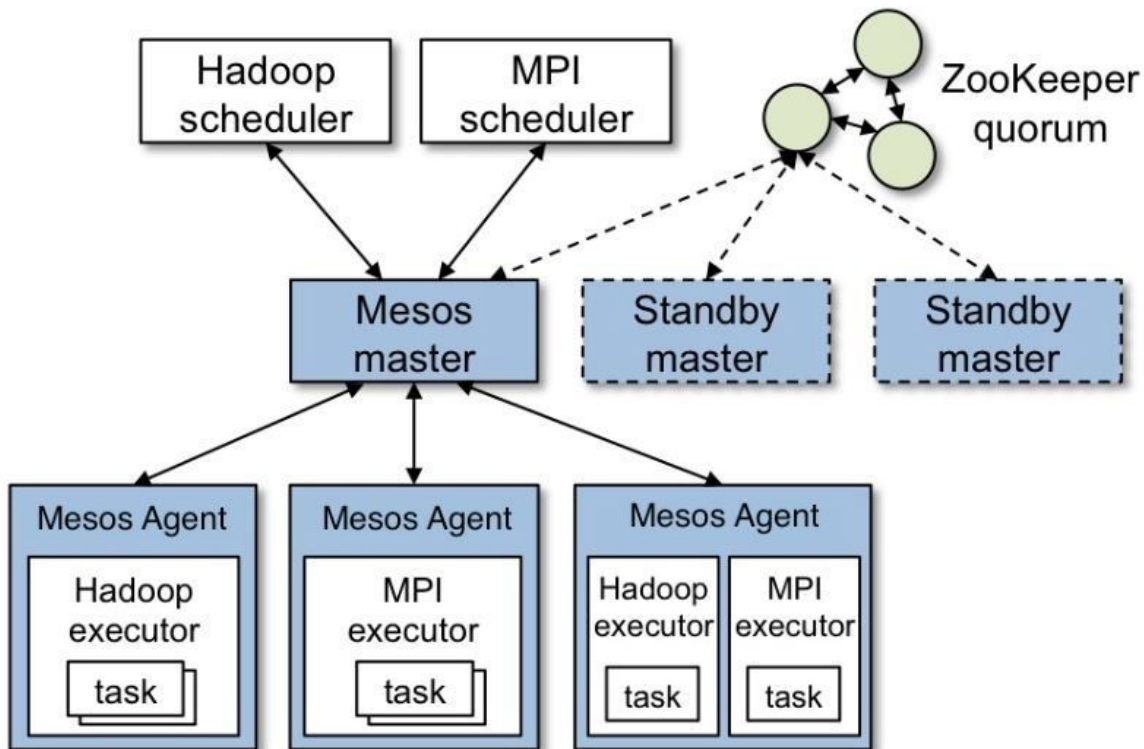


O gerenciamento de uma aplicação distribuída possui, em geral, as seguintes funcionalidades: distribuição de tarefas, monitoramento de tarefas, limpeza das tarefas, detecção de falhas, inicialização de tarefas, encerramento de tarefas e etc. Todas essas funcionalidades estão disponíveis no Mesos por meio de uma API. Além disso, o Mesos torna fácil rodar múltiplos sistemas distribuídos em um único cluster por compartilhar os recursos de forma dinâmica e eficiente.

## ARQUITETURA

Para melhor compreender a arquitetura do Mesos, vamos analisar a seguinte imagem.

## Mesos Architecture



Essa imagem faz parte da documentação oficial do Mesos. Nesse exemplo, Hadoop e MPI são duas aplicações que dividem o cluster do Mesos.

Nas seções seguintes serão analisados cada um dos componentes mostrados aqui.

- **Mesos Master**

Master é o componente principal da configuração. Ele armazena o estado atual dos recursos no cluster. Além disso, ele age como um orquestrador entre os agentes e as aplicações por passar informações sobre tarefas e recursos.

Como qualquer falha no master resultaria na perda do estado dos recursos e tarefas, é feito o seu deploy em uma configuração de alta disponibilidade. Como pode ser visto no diagrama acima, o Mesos faz o deploy de outros masters em standby. Esses masters em aguardo dependem do Zookeeper para recuperar o estado em caso de falha.

- **Agentes Mesos**

Um cluster Mesos deve necessariamente rodar um agente em cada máquina. Esses agentes relatam o estado de seus recursos periodicamente ao master, assim como recebem tarefas que uma aplicação agendou para serem rodadas. Esse ciclo se repete após a tarefa agendada ser completada ou perdida.

Nas seções seguintes será explicado melhor como as aplicações agendam e executam as tarefas nesses agentes.

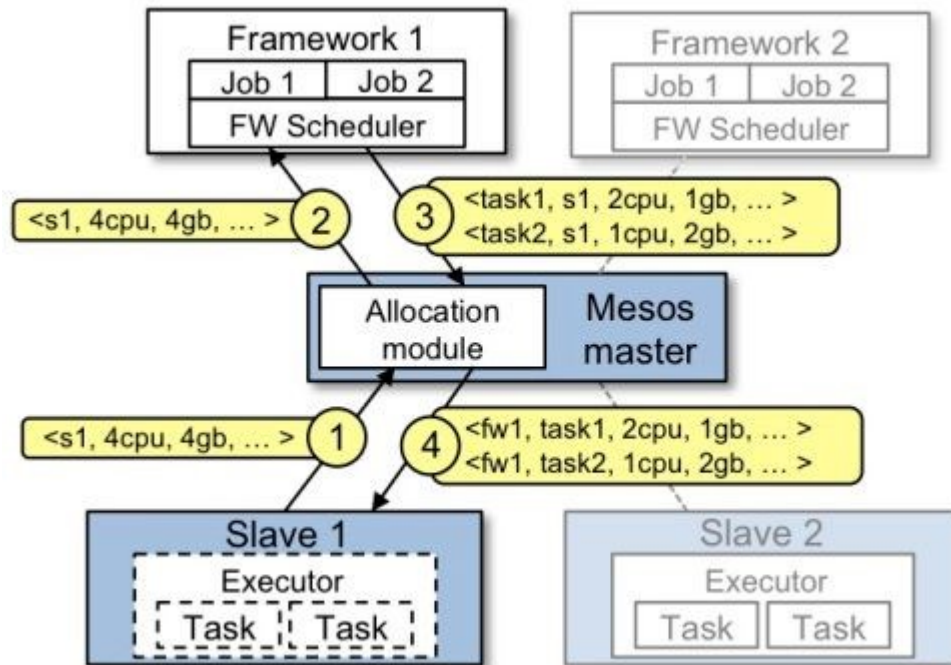
- **Framework Mesos**

O Mesos permite que seja implementado um componente abstrato nas aplicações, chamado de framework, que interage com a Master para receber os recursos disponíveis no cluster, além de tomar decisões de agendamento baseado nessas informações de recursos

Um framework Mesos é composto de dois sub-componentes:

- **Scheduler:** Permite que as aplicações realizem agendamentos de tarefas baseado nos recursos disponíveis nos agentes.
- **Executador:** Roda em todos os agentes e contém as informações necessárias para executar quaisquer tarefas agendadas naquele agente.

Esse processo de agendamento/execução é ilustrado com o seguinte fluxo:



Primeiro os agentes relatam seus recursos ao Master. Nesse instante, o master oferece esses recursos a todos os schedulers registrados. Esse processo é conhecido como oferta de recurso.

O scheduler então escolhe o melhor agente e executa diversas tarefas nele, através do master. Assim que um executor completa a tarefa designada, os agentes reportam seus recursos ao master. O Master repete esse processo de compartilhamento de recursos para todos os frameworks no cluster.

## MESOS vs. KUBERNETES

Uma das principais ferramentas similares ao Mesos é o Kubernetes. Para resumir, o Kubernetes é uma plataforma de orquestração de containers desenvolvida originalmente pelo Google que em 2015 se tornou open source e totalmente gratuita para uso. Ele é baseado em uma arquitetura de Nodes, que são as máquinas onde os containers são criados, e um Master, que gerencia e mantém o estado desses containers, seus deploys e seus serviços.

Uma das principais diferenças entre os dois é que o Mesos, além de oferecer uma plataforma de orquestração de containers por meio do Marathon, também possibilita a execução de tarefas em um ambiente não containerizado. Dessa maneira, é possível ter um mesmo cluster do Mesos executando aplicações clusterizadas e não clusterizadas simultaneamente, enquanto o Kubernetes é totalmente baseado na orquestração de containers.

Outro ponto importante é a escalabilidade, enquanto o Kubernetes suporta a execução de até 5000 nodes simultaneamente, o Mesos tem um desempenho melhor nesse quesito, podendo chegar até 10000 agentes.

Além disso, o Kubernetes é uma ferramenta menos complexa e de aprendizado mais fácil do que o Mesos e pode-se afirmar que a comunidade do Kubernetes é bem maior que a do Mesos. Portanto, o primeiro tem uma implantação mais rápida e fácil, tendo um maior suporte em decorrência do tamanho da comunidade.

Abaixo, um resumo das principais diferenças entre os dois e por que escolher cada um como plataforma:

1. Por que escolher o Mesos: Oferece a possibilidade de rodar em um ambiente não clusterizado conjuntamente com aplicações baseadas em containers. Além disso, o Mesos aguenta monitorar até 10.000 agentes ao mesmo tempo (o Kubernetes aguenta apenas 5.000 nodes);
2. Por que escolher o Kubernetes: Oferece mais recursos por meio de APIs e uma comunidade maior para suporte. Além disso, é mais flexível e rápido de ser lançado para novas aplicações;
3. Rodar o Kubernetes sobre o Mesos, utilizando-o como um framework para executar as aplicações desejadas no cluster de maneira rápida e eficiente.

## MESOS vs. YARN

Um outro forte competidor do Mesos é o Yarn (Yet Another Resource Negotiator). Essa plataforma de gerenciamento de recursos foi desenvolvida para processar dados e realizar tarefas do Hadoop (também desenvolvida pela Apache). Por esse motivo, esta plataforma é muito mais utilizada para serviços do Hadoop, especificamente, do que aplicações no geral. Assim como o Mesos, o YARN faz o gerenciamento de recursos distribuídos para otimizar o processamento de dados e garantir high availability e velocidade.

Porém, a forma com que o Yarn é organizado, difere muito do Mesos no que se diz ao scheduler. Enquanto o Yarn é organizado de forma monolítica, ou seja, o scheduler tem o papel de decidir completamente sozinho onde os jobs devem ser processados, o scheduler do Mesos se organiza de forma não monolítica, e por isso faz ofertas para os jobs serem feitos conforme os recursos disponíveis que mais se adequam àquele job. Por isso, o Mesos acaba sendo muito mais escalável que o YARN.

Além disso, enquanto o Mesos foi programado em C++, o YARN foi totalmente desenvolvido em Java.

Abaixo, um resumo das principais diferenças entre os dois e por que escolher cada um como plataforma:

1. Por que escolher o Mesos: Maior escalabilidade para aplicações muito grandes e diversificadas;
2. Por que escolher o YARN: Foi criado para realizar processamento de serviços Hadoop, embora possa ser utilizado em uma variedade muito grande de aplicações;
3. Usar o YARN e Mesos conjuntamente pode ser feito por meio do projeto Myriad, que integra o scheduler de ambos para obter o processamento mais eficiente possível de dados Hadoop e não Hadoop.

## APACHE MESOS TUTORIAL

Um tutorial de implantação do Mesos foi criado e está presente nas próximas páginas. Além da configuração e execução dos serviços do Mesos também está detalhada a criação de um framework para executar uma aplicação sobre essa infraestrutura. Esse tutorial também está disponível [neste link](#).



## TUTORIAL DE DEPLOY DO APACHE MESOS

Para realizar esse tutorial, é necessária uma máquina rodando Ubuntu server, podendo estar rodando na plataforma de sua escolha: AWS, Azure ou VirtualBox na sua própria máquina.

Nesse começo trataremos sobre como criar uma instância na AWS adequada para rodar o Mesos.

### CRIANDO A INSTÂNCIA NA AWS EC2

1. Realize o login no AWS Console e navegue para a seção do EC2.
2. Clique em **Launch Instance** e selecione a imagem **Ubuntu Server 16.04 LTS AMI (Eligible for Free tier)**.


 **Ubuntu Server 16.04 LTS (HVM), SSD Volume Type** - ami-04763b3055de4860b (64-bit x86) / ami-02ca3cadbc293e21 (64-bit Arm)

Ubuntu Server 16.04 LTS (HVM),EBS General Purpose (SSD) Volume Type. Support available from Canonical (<http://www.ubuntu.com/cloud/services>).

Free tier eligible

Root device type: ebs    Virtualization type: hvm    ENA Enabled: Yes

3. Em **Instance Type** selecione **t2.micro** e continue para **Configure Instance Details**.

	General purpose	t2.micro Free tier eligible	1	1	EBS only
--	-----------------	--------------------------------	---	---	----------

4. Selecione para todos os campos a opção **default** e continue para **Add Storage**.

#### Step 3: Configure Instance Details

Configure the instance to suit your requirements. You can launch multiple instances from the same AMI, request Spot instances to take advantage of the lower pricing, assign an access management role to the instance, and more.

Number of instances	1	<a href="#">Launch into Auto Scaling Group</a>
Purchasing option	<input type="checkbox"/> Request Spot instances	
Network	vpc-c32d39b8 (default)	<a href="#">Create new VPC</a>
Subnet	No preference (default subnet in any Availability Zone)	<a href="#">Create new subnet</a>
Auto-assign Public IP	Use subnet setting (Enable)	

5. Em storage mantenha a opção padrão: **8GB General Purpose SSD**.

Volume Type	Device	Snapshot	Size (GiB)	Volume Type	IOPS	Throughput (MB/s)	Delete on Termination	Encryption
Root	/dev/sda1	snap-0b7112419d20ddeb4	8	General Purpose SSD (gp2)	100 / 3000	N/A	<input checked="" type="checkbox"/>	Not Encrypted
<a href="#">Add New Volume</a>								

6. Adicione uma tag de **Name** na instância para indentifica-la.

Key	Value	Instances	Volumes
Name	tutorial-mesos	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<a href="#">Add another tag</a> (Up to 50 tags maximum)			

7. Na seção de **Security Groups**, crie um com as portas **22**, **2181**, **5050** abertas. Caso planeje instalar o Marathon posteriormente, abra a porta 8080 também para acessar o Marathon UI.

Type ⓘ	Protocol ⓘ	Port Range ⓘ	Source ⓘ
Custom TCP F ▼	TCP	8080	Custom ▼ 0.0.0.0, ::/0
Custom TCP F ▼	TCP	5050	Custom ▼ 0.0.0.0/0
SSH ▼	TCP	22	Custom ▼ 0.0.0.0/0
Custom TCP F ▼	TCP	2181	Custom ▼ 0.0.0.0/0

8. Por fim, clique em **Review and Launch** e selecione uma key pair (caso já possua uma) ou crie e baixe uma para acessar a instância.
9. Assim que a instância estiver rodando, acesse-a por meio de uma conexão SSH utilizando o **Public IP** e a **key pair** existente ou criada.

---

Public DNS (IPv4)	ec2-34-201-73-112.compute-1.amazonaws.com
IPv4 Public IP	34.201.73.112
IPv6 IPs	-

10. Se você estiver usando **Windows**, será necessário utilizar o **Putty** para gerar um arquivo **.ppk** a partir do arquivo baixado **.pem** e para acessar a instância.
11. Se estiver usando **Linux**, é possível se conectar diretamente do terminal utilizando o arquivo **.pem** e o **Public IP** da instância. Para isso basta utilizar os seguintes comandos, substituindo o `[/dir/da/keypair.pem]` pelo diretório onde o arquivo `.pem` da keypair está :

```
$ chmod 400 /dir/da/keypair.pem
$ ssh -i /dir/da/keypair.pem ubuntu@[IP-DA-INSTANCIA]
```

## CONFIGURANDO O AMBIENTE DA INSTÂNCIA

1. Após se conectar a instância, é preciso instalar o Java JDK 8.

```
$ sudo apt-get update
$ sudo apt install openjdk-8-jdk openjdk-8-jre
```

2. Então, execute os seguintes comandos para adicionar as chaves OpenPGP para o pacote do Mesos.

Caso o primeiro comando falhe, tente com o segundo.

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv DFDCBE BF
OU
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv E F
```

3. Depois adicione o repositório do Mesos específico para a versão do Ubuntu sendo utilizada, como estamos utilizando a versão 16.04, seu codinome é **Xenial**, como escrito abaixo.

```
sudo sh -c 'echo "deb http://repos.mesosphere.com/ubuntu xenial main" >> /etc/apt/sources.list.d/mesosphere.list'
sudo apt-get update
```

4. Caso tudo tenha ocorrido bem, podemos agora instalar o Mesos do pacote adicionado.

```
sudo apt-get -y install mesos
```

Antes de começar a rodar o Mesos é preciso configurar algumas variáveis e arquivos.

## CONFIGURANDO O MESOS

1. Execute o comando:

```
hostname -f
```

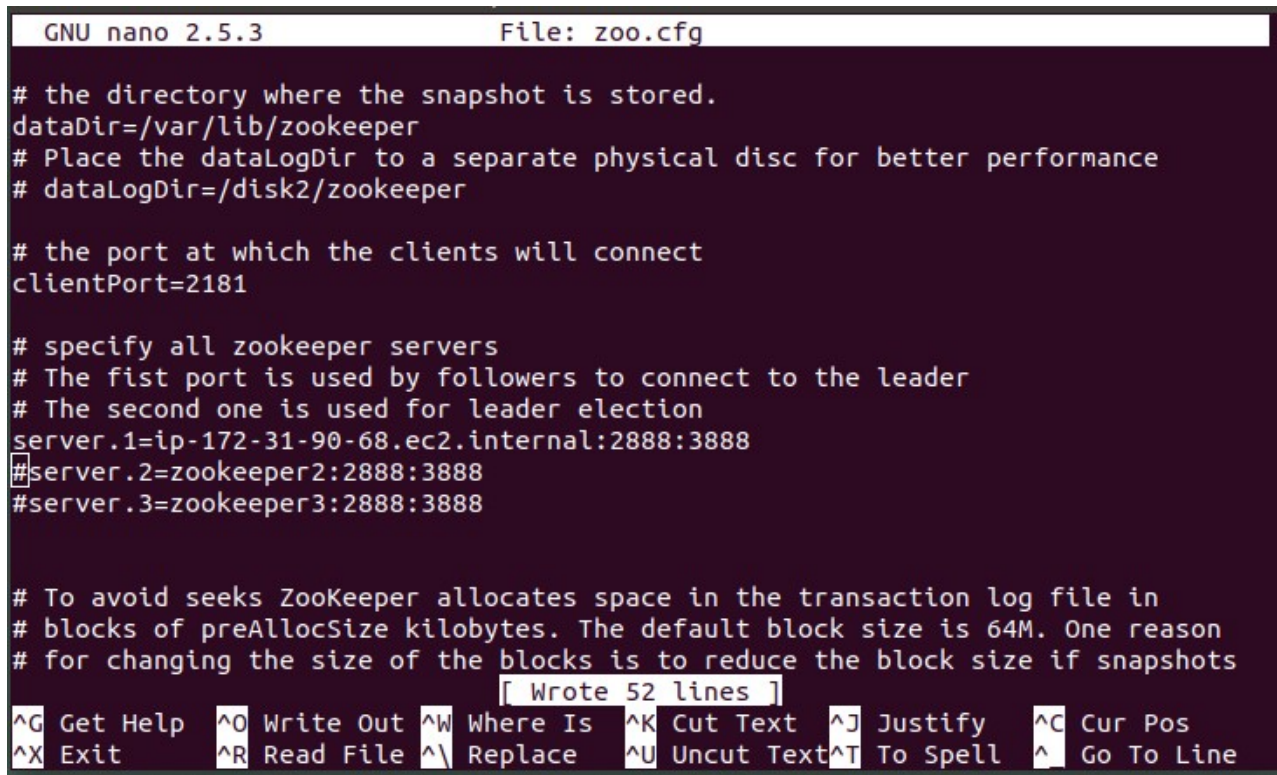
Para obter o hostname da Instância onde o Mesos Master será iniciado.

```
ubuntu@ip-172-31-90-68:/etc/zookeeper/conf$
ubuntu@ip-172-31-90-68:/etc/zookeeper/conf$
ubuntu@ip-172-31-90-68:/etc/zookeeper/conf$
ubuntu@ip-172-31-90-68:/etc/zookeeper/conf$
ubuntu@ip-172-31-90-68:/etc/zookeeper/conf$ hostname -f
ip-172-31-90-68.ec2.internal
ubuntu@ip-172-31-90-68:/etc/zookeeper/conf$
```

2. Navegue para o diretório: **/etc/zookeeper/conf** e edite o arquivo **zoo.cfg** como mostrado abaixo. Descomente a linha que se inicia com **server.1** e substitua **zookeeper1** com o hostname obtido

anteriormente. Como estamos utilizando um zookeeper em um única máquina, podemos substituir por localhost.

```
cd /etc/zookeeper/conf  
sudo nano zoo.cfg
```



```
GNU nano 2.5.3 File: zoo.cfg  
  
# the directory where the snapshot is stored.  
dataDir=/var/lib/zookeeper  
# Place the dataLogDir to a separate physical disc for better performance  
# dataLogDir=/disk2/zookeeper  
  
# the port at which the clients will connect  
clientPort=2181  
  
# specify all zookeeper servers  
# The first port is used by followers to connect to the leader  
# The second one is used for leader election  
server.1=ip-172-31-90-68.ec2.internal:2888:3888  
server.2=zookeeper2:2888:3888  
server.3=zookeeper3:2888:3888  
  
# To avoid seeks ZooKeeper allocates space in the transaction log file in  
# blocks of preAllocSize kilobytes. The default block size is 64M. One reason  
# for changing the size of the blocks is to reduce the block size if snapshots  
  
[ Wrote 52 lines ]  
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos  
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line
```

3. Rode o seguinte comando para editar o arquivo **myid** dentro do diretório **/etc/zookeeper/conf**. Aqui estamos adicionando o número 1 ao arquivo. Esse número é o server id para cada instância zookeeper do cluster, ele não pode ser repetido e varia de 1 a 255.

```
cd /etc/zookeeper/conf  
sudo sh -c 'echo -n "" >> myid'
```

4. Navegue para o diretório: **/etc/mesos** e edite o arquivo **zk** para apontar para o IP interno da instância do zookeeper como mostrado na imagem abaixo, novamente, como estamos utilizando uma única máquina pode-se usar localhost. Caso haja mais de uma instância rodando um mesos-master, o mesmo deverá ser feito em todas.

```
cd /etc/mesos
```

```
sudo nano zk
```

```
GNU nano 2.5.3      File: zk
zk://localhost:2181/mesos
█
```

5. Em seguida, navegue para o diretório: **/etc/mesos-master** e edite o arquivo **quorum** com um valor. Esse valor deve ser definido de uma maneira que 50% dos masters devem estar “presentes” para que uma decisão seja tomada. Assim, ele deve ser maior que o número de masters dividido por 2, como estamos rodando apenas uma instância com um master, definimos como 1.

```
cd /etc/mesos-master
```

```
sudo nano quorum
```

```
GNU nano 2.5.3      File: quorum
1
█
```

6. Ainda é preciso configurar o hostname e endereço IP para nosso mesos-master. Crie 2 arquivos **ip** e **hostname**, dentro do diretório **/etc/mesos-master** com os valores corretos.

```
cd /etc/mesos-master
sudo sh -c 'echo "[IP-INTERNO-DA-INSTANCIA]" >> ip'
sudo sh -c 'echo "[IP-INTERNO-DA-INSTANCIA]" >> hostname'
cat ip
cat hostname
```

```
ubuntu@ip-172-31-90-68:/etc/mesos-master$
ubuntu@ip-172-31-90-68:/etc/mesos-master$
ubuntu@ip-172-31-90-68:/etc/mesos-master$
ubuntu@ip-172-31-90-68:/etc/mesos-master$ cat ip
172.31.90.68
ubuntu@ip-172-31-90-68:/etc/mesos-master$ cat hostname
172.31.90.68
ubuntu@ip-172-31-90-68:/etc/mesos-master$ █
```

**Observação:** Caso o mesos seja executado em uma subnet privada, utilize o IP Interno da instância para ambos os arquivos. Mas caso esteja em uma subnet pública, utilize o IP público da instância.

## INICIANDO O ZOOKEEPER, MESOS-MASTER E MESOS-SLAVE

Agora que o ambiente e as variáveis estão todas configuradas, podemos iniciar o processo do mesos-master, mesos-slave e do zookeeper. Para isso, utilize os seguintes comandos:

```
$ sudo service zookeeper start
$ sudo service mesos-master start
$ sudo service mesos-slave start
```

Para verificar que tudo está rodando sem erros utilize os seguintes comandos para obter o status dos processos:

```
$ sudo service zookeeper status
$ sudo service mesos-master status
$ sudo service mesos-slave status
```

O output deve ser parecido com o mostrado abaixo para o zookeeper:

```
● zookeeper.service - LSB: centralized coordination service
   Loaded: loaded (/etc/init.d/zookeeper; bad; vendor preset: enabled)
   Active: active (running) since Qua 2019-11-13 14:33:51 UTC; 27s ago
     Docs: man:systemd-sysv-generator(8)
  Process: 18392 ExecStop=/etc/init.d/zookeeper stop (code=exited, status=0/SUCCESS)
  Process: 18406 ExecStart=/etc/init.d/zookeeper start (code=exited, status=0/SUCCESS)
    Tasks: 15
   Memory: 48.6M
      CPU: 863ms
   CGroup: /system.slice/zookeeper.service
           └─18421 /usr/bin/java -cp /etc/zookeeper/conf:/usr/share/java/jline.jar:/usr/share/java

Nov 13 14:33:51 ip-172-31-90-68 systemd[1]: Stopped LSB: centralized coordination service.
Nov 13 14:33:51 ip-172-31-90-68 systemd[1]: Starting LSB: centralized coordination service...
Nov 13 14:33:51 ip-172-31-90-68 systemd[1]: Started LSB: centralized coordination service.
~
~
~
~
~
~
```

---

Para o mesos-master:



```

● mesos-master.service - Mesos Master
   Loaded: loaded (/lib/systemd/system/mesos-master.service; disabled; vendor preset: enabled)
   Active: active (running) since Qua 2019-11-13 14:34:01 UTC; 1m1n 23s ago
     Main PID: 18443 (mesos-master)
        Tasks: 21
       Memory: 20.1M
          CPU: 205ms
     CGroup: /system.slice/mesos-master.service
             └─18443 /usr/sbin/mesos-master --zk=zk://localhost:2181/mesos --port=5050 --log_dir=/var/log/mesos --hostname=172.31.90.68 --ip=172.31.90.68 --quorum=1 --work_dir=/var/lib/mesos
               └─18459 logger -p user info -t mesos-master[18443]
                 └─18460 logger -p user err -t mesos-master[18443]

Nov 13 14:34:10 ip-172-31-90-68 mesos-master[18460]: I1113 14:34:10.023685 18468 registrar.cpp:544] Successfully updated the registry in 5.957888ms
Nov 13 14:34:10 ip-172-31-90-68 mesos-master[18460]: I1113 14:34:10.023742 18468 registrar.cpp:416] Successfully recovered registrar
Nov 13 14:34:10 ip-172-31-90-68 mesos-master[18460]: I1113 14:34:10.023939 18468 master.cpp:1819] Recovered 1 agents from the registry (3298); allowing 10mins for agents to reregister
Nov 13 14:34:10 ip-172-31-90-68 mesos-master[18460]: I1113 14:34:10.023970 18468 coordinator.cpp:348] Coordinator attempting to write TRUNCATE action at position 14
Nov 13 14:34:10 ip-172-31-90-68 mesos-master[18460]: I1113 14:34:10.024061 18468 replica.cpp:541] Replica received write request for position 14 from __req_res__(3)@172.31.90.68:5050
Nov 13 14:34:10 ip-172-31-90-68 mesos-master[18460]: I1113 14:34:10.033177 18468 replica.cpp:695] Replica received learned notice for position 14 from log-network(1)@172.31.90.68:5050
Nov 13 14:34:10 ip-172-31-90-68 mesos-master[18460]: I1113 14:34:10.980072 18462 master.cpp:7439] Received reregister agent message from agent 94987e3b-f4ae-4c80-ab5d-30473e071c53-50 at slave(1)@172.31.90.68:5050
Nov 13 14:34:10 ip-172-31-90-68 mesos-master[18460]: I1113 14:34:10.980556 18462 master.cpp:7980] Re-registered agent 94987e3b-f4ae-4c80-ab5d-30473e071c53-50 at slave(1)@172.31.90.68:5051 (ip-172-31-90-68.ec2.i
Nov 13 14:34:10 ip-172-31-90-68 mesos-master[18460]: I1113 14:34:10.980695 18462 hierarchical.cpp:854] Added agent 94987e3b-f4ae-4c80-ab5d-30473e071c53-50 (ip-172-31-90-68.ec2.internal) with cpus:1; mem:495; dt
Nov 13 14:34:10 ip-172-31-90-68 mesos-master[18460]: I1113 14:34:10.982476 18467 master.cpp:8487] Ignoring update on agent 94987e3b-f4ae-4c80-ab5d-30473e071c53-50 at slave(1)@172.31.90.68:5051 (ip-172-31-90-68.
-

```

## Para o mesos-slave:

```

ubuntu@ip-172-31-90-68:/etc/mesos-master$ ^C
ubuntu@ip-172-31-90-68:/etc/mesos-master$ sudo service mesos-slave status
● mesos-slave.service - Mesos Slave
   Loaded: loaded (/lib/systemd/system/mesos-slave.service; disabled; vendor preset: enabled)
   Active: active (running) since Qua 2019-11-13 14:34:09 UTC; 1m1n 56s ago
     Main PID: 18486 (mesos-slave)
        Tasks: 16
       Memory: 7.1M
          CPU: 235ms
     CGroup: /system.slice/mesos-slave.service
             └─18486 /usr/sbin/mesos-slave --master=zk://localhost:2181/mesos --log_dir=/var/log/mesos --work_dir=/var/lib/mesos
               └─18496 logger -p user info -t mesos-slave[18486]
                 └─18497 logger -p user err -t mesos-slave[18486]

Nov 13 14:34:10 ip-172-31-90-68 mesos-slave[18497]: I1113 14:34:10.136004 18500 slave.cpp:1351] New master detected at master@172.31.90.68:5050
Nov 13 14:34:10 ip-172-31-90-68 mesos-slave[18497]: I1113 14:34:10.136024 18500 slave.cpp:1405] No credentials provided. Attempting to register without authentication
Nov 13 14:34:10 ip-172-31-90-68 mesos-slave[18497]: I1113 14:34:10.136036 18500 slave.cpp:1416] Detecting new master
Nov 13 14:34:10 ip-172-31-90-68 mesos-slave[18497]: I1113 14:34:10.136051 18500 task_status_update_manager.cpp:181] Pausing sending task status updates
Nov 13 14:34:10 ip-172-31-90-68 mesos-slave[18497]: I1113 14:34:10.136059 18500 status_update_manager_process.hpp:379] Pausing operation status update manager
Nov 13 14:34:10 ip-172-31-90-68 mesos-slave[18497]: I1113 14:34:10.982153 18502 slave.cpp:1700] Re-registered with master master@172.31.90.68:5050
Nov 13 14:34:10 ip-172-31-90-68 mesos-slave[18497]: I1113 14:34:10.982239 18502 slave.cpp:1746] Forwarding agent update ("operations": {}, "resource_providers": {}, "resource_version_oid": {"value": "u190IeI+Rx2QC0d
Nov 13 14:34:10 ip-172-31-90-68 mesos-slave[18497]: I1113 14:34:10.982832 18498 task_status_update_manager.cpp:186] Resuming sending task status updates
Nov 13 14:34:10 ip-172-31-90-68 mesos-slave[18497]: I1113 14:34:10.982852 18498 status_update_manager_process.hpp:385] Resuming operation status update manager
Nov 13 14:35:10 ip-172-31-90-68 mesos-slave[18497]: I1113 14:35:10.070348 18503 slave.cpp:7406] Current disk usage 48.59%. Max allowed age: 2.898959710091586days
lines 1-22/22 (END)

```

Quando todos os serviços estiverem rodando corretamente, utilize seu browser e acesse o endereço

**[IP\_PUBLICO\_INSTANCIA]:5050** para entrar no dashboard do mesos, ilustrado abaixo:

**Observação:** Se seu computador estiver conectado a uma rede pública, como de faculdades, esse endereço pode ser bloqueado no browser.



The screenshot shows the Mesos Master web interface. The top navigation bar includes links for Frameworks, Agents, Roles, Offers, and Maintenance. The main content area is divided into several sections:

- Cluster Information:** Cluster: (Unnamed), Leader: 172.31.90.68:5050, Version: 1.9.0, Built: 2 months ago by ubuntu, Started: 5 minutes ago, Elected: 5 minutes ago.
- Leading Master Log:** Download | View
- Agents:** A table showing the status of agents. The 'Activated' status has a count of 1, while 'Deactivated' and 'Unreachable' have counts of 0.
- Tasks:** A table showing the status of tasks. All statuses (Staging, Starting, Running, Unreachable, Killing, Finished, Killed, Failed, Lost) have a count of 0.
- Resources:** A table showing resource usage. The 'Total' row shows 1 CPU, 0 GPUs, 495 MB Mem, and 3.8 GB Disk. 'Allocated' and 'Offered' rows show 0 for all resources.
- Active Tasks:** A table with columns: Framework ID, Task ID, Task Name, Role, State, Health, Started, Host. It shows 'No active tasks.'
- Unreachable Tasks:** A table with columns: Framework ID, Task ID, Task Name, Role, Started, Agent ID. It shows 'No unreachable tasks.'
- Completed Tasks:** A table with columns: Framework ID, Task ID, Task Name, Role, State, Started, Stopped, Host. It shows 'No completed tasks.'

Como não rodamos nenhum framework ainda, a lista de frameworks estará vazia.

Para executar um comando simples na infraestrutura recém-criada, para testá-la podemos utilizar a seguinte sintaxe:

```
$ mesos-execute --master=<IP_PUBLICO>:5050 --name="echo-test" --command=echo "Hello, World"
```

```
ubuntu@ip-172-31-90-68:~$ mesos-execute --master=34.201.73.112:5050 --name="echo-test" --command=echo "Hello, World"
I1113 15:01:34.161592 1761 scheduler.cpp:189] Version: 1.9.0
I1113 15:01:34.165388 1769 scheduler.cpp:342] Using default 'basic' HTTP authenticator
I1113 15:01:34.165576 1769 scheduler.cpp:525] New master detected at master@34.201.73.112:5050
Subscribed with ID b7e3aa64-2820-4f1c-8da3-564894574c49-0000
Submitted task 'echo-test' to agent '94987e3b-f4ae-4c80-ab5d-30473e071c53-S0'
Received status update TASK_STARTING for task 'echo-test'
  source: SOURCE_EXECUTOR
Received status update TASK_RUNNING for task 'echo-test'
  source: SOURCE_EXECUTOR
Received status update TASK_FINISHED for task 'echo-test'
  message: 'Command exited with status 0'
  source: SOURCE_EXECUTOR
ubuntu@ip-172-31-90-68:~$
```

## CRIANDO UM FRAMEWORK

Como explicado anteriormente, para executar tarefas sobre a infraestrutura gerenciada pelo mesos master é preciso de um framework que define essa intermediação. Assim, vamos explicar abaixo a composição e configuração de um framework em **Python**.

Para esse framework utilizamos a biblioteca **pymesos** disponível para Python e realizamos os seguintes imports no arquivo python:

```
from pymesos import MesosSchedulerDriver, Scheduler, encode_data
import uuid
from addict import Dict
import socket
import getpass
from threading import Thread
import signal
import time
import enum
import os
```

Primeiramente temos uma classe que define as **Tasks**, estas recebem em sua inicialização um **taskId**, um **comando** a ser executado e os **resources necessários (cpu e memória)**.

```
class Task:
    def __init__(self, taskId, command, cpu, mem):
        self.taskId = taskId
        self.command = command
        self.cpu = cpu
        self.mem = mem
```

Então temos um conjunto de métodos para essa classe que são utilizados para aceitar uma oferta de recursos para executar a tarefa:

- O método **\_\_getResource** retorna valor de um recurso específico disponibilizado em uma oferta, por exemplo, **memória**, ele é utilizado na etapa de definir se os recursos de uma oferta são suficientes para rodar a tarefa.

```
def __getResource(self, res, name):
    for r in res:
        if r.name == name:
            return r.scalar.value
    return .
```

- O método **\_\_updateResource** atualiza o valor de um recurso específico, ele é utilizado quando uma oferta é aceita. Assim, ele basicamente subtrai o valor necessário de certo recurso para a tarefa do valor total de certo recurso na oferta.

```
def __updateResource(self, res, name, value):
    if value <= 0:
        return
    for r in res:
        if r.name == name:
            r.scalar.value -= value
    return
```

- O método **acceptOrder** determina se uma oferta recebida é adequada para executar a tarefa. Assim, a partir dos métodos acima ele verifica se o valor de recursos da oferta é maior que o valor de recursos necessários para a execução da tarefa e aceita ou não esta oferta, atualizando os recursos disponíveis caso a oferta seja aceita.

```

def acceptOffer(self, offer):
    accept = True
    if self.cpu != 0:
        cpu = self.__getResource(offer.resources, "cpus")
        if self.cpu > cpu:
            accept = False
    if self.mem != 0:
        mem = self.__getResource(offer.resources, "mem")
        if self.mem > mem:
            accept = False
    if(accept == True):
        self.__updateResource(offer.resources, "cpus", self.cpu)
        self.__updateResource(offer.resources, "mem", self.mem)
        return True
    else:
        return False

```

Em seguida temos uma classe que define o **Scheduler**, ela é inicializada com listas e dicionários vazios que são usados para armazenarem as tarefas disponíveis, em inicialização, em execução e terminadas. Além disso, neste caso na inicialização da classe já inserimos na lista de tarefas disponíveis as tarefas a serem executadas, mas poderíamos criar um método para que a classe recebesse as tarefas dinamicamente. Essas tarefas serão melhor explicadas mais a frente neste guia.

```
class PythonScheduler(Scheduler):  
    def __init__(self):  
        self.idleTaskList = []  
        self.startingTaskList = {}  
        self.runningTaskList = {}  
        self.terminatingTaskList = {}  
  
        self.idleTaskList.append(Task("taskHelloWorld", "echo HelloWorld", .1, 100))  
        self.idleTaskList.append(Task("taskDIR", "mkdir /home/ubuntu/HelloMesos", .1,  
100))
```

Então, temos um método ***resourceOffers*** que verifica se há alguma tarefa pendente na lista e caso haja, ele cria uma oferta para essa tarefa por meio do método ***acceptOffer*** da classe **Task**. Assim, caso a oferta seja aceita ele executa a tarefa no cluster e a adiciona na lista correspondente.

```
def resourceOffers(self, driver, offers):  
    logging.debug("Received new offers")  
    logging.info("Recieved resource offers: {}".format([o.id.value for o in offers]))  
  
    if(len(self.idleTaskList) == 0):  
        driver.suppressOffers()  
        logging.info("Idle Tasks List Empty, Suppressing Offers")  
        return  
  
    filters = {'refuse_seconds': 1}  
  
    for offer in offers:  
        taskList = []  
        pendingTaksList = []  
        while True:  
            if len(self.idleTaskList) == 0:
```

```

        break

    Task = self.idleTaskList.pop(0)

    if Task.acceptOffer(offer):

        task = Dict()

        task_id = Task.taskId

        task.task_id.value = task_id

        task.agent_id.value = offer.agent_id.value

        task.name = 'task {}'.format(task_id)

        task.command.value = Task.command

        task.resources = [

            dict(name='cpus', type='SCALAR', scalar={'value': Task.cpu}),

            dict(name='mem', type='SCALAR', scalar={'value': Task.mem}),

        ]

        self.startingTaskList[task_id] = Task

        taskList.append(task)

        logging.info("Starting task: %, in node: %" % (Task.taskId, offer.agent_id.hostname))

    else:

        pendingTaksList.append(Task)

    if(len(taskList)):

        driver.launchTasks(offer.id, taskList, filters)

    self.idleTaskList = pendingTaksList

```

Por fim, há o método ***statusUpdate*** que é utilizado para logar o status de execução da tarefa e atualizar sua lista e estado.

```
def statusUpdate(self, driver, update):  
    if update.state == "TASK_STARTING":  
        Task = self.startingTaskList[update.task_id.value]  
        logging.debug("Task %s is starting." % update.task_id.value)  
    elif update.state == "TASK_RUNNING":  
        if update.task_id.value in self.startingTaskList:  
            Task = self.startingTaskList[update.task_id.value]  
            logging.info("Task %s running in %s. Moving to running list" %
```



```

        (update.task_id.value, update.container_status.network_infos[0].ip_addresses[0].ip_address))

        self.runningTaskList[update.task_id.value] = Task
        del self.startingTaskList[update.task_id.value]
    elif update.state == "TASK_FAILED":
        Task = None
        if update.task_id.value in self.startingTaskList:
            Task = self.startingTaskList[update.task_id.value]
            del self.startingTaskList[update.task_id.value]
        elif update.task_id.value in self.runningTaskList:
            Task = self.runningTaskList[update.task_id.value]
            del self.runningTaskList[update.task_id.value]
        if Task:
            logging.info("Uni task: %s failed." % Task.taskId)
            self.idleTaskList.append(Task)
            driver.reviveOffers()
        else:
            logging.error("Received task failed for unknown task: %s" % update.task_id.value )
        else:
            logging.info("Received status %s for task id: %s" % (update.state, update.task_id.value))

```

O código completo do **scheduler.py** está presente aqui

Descrevendo agora as tarefas que foram criada na inicialização da classe do **Scheduler**, criamos duas tarefas a serem executadas sobre os agentes do mesos:

```

Task("taskHelloWorld", "echo HelloWorld", .1, 100)
Task("taskDIR", "mkdir /home/ubuntu/HelloMesos", .1, 100)

```

A primeira simplesmente executa um comando "***echo HelloWorld***", porém não é possível observar o output do comando, uma vez que ele roda no agente. Então, criamos uma segunda tarefa para conseguirmos observar de fato sua execução, nela há um comando para que uma pasta ***HelloMesos*** seja criada no diretório ***/home/ubuntu***. É importante notar que na infraestrutura do Mesos que criamos nesse tutorial tanto o master quanto os agentes estão na mesma máquina, por isso conseguimos observar o resultado do comando dessa segunda tarefa.

Finalmente, para rodar o framework criado basta criar um arquivo scheduler.py na máquina onde o tutorial de implantação foi executado ou clonar diretamente do github e configurar uma variável de ambiente com o IP público da máquina do Mesos Master:

```
$ nano scheduler.py
OU
$ git clone https://github.com/elijose55/Mesos-Tutorial.git

$ export MESOS_MASTER_IP=34.201.73.112
$ python3 scheduler.py
```

Então, após executar o **scheduler** você deverá receber um output como abaixo:

```
ubuntu@ip-172-31-90-68:~$ python3 scheduler.py
Scheduler running, Ctrl+C to quit.
DEBUG:root:Received new offers
INFO:root:Recieved resource offers: ['b7e3aa64-2820-4f1c-8da3-564894574c49-039']
INFO:root:Starting task: taskHelloWorld, in node: ip-172-31-90-68.ec2.internal
INFO:root:Starting task: taskDIR, in node: ip-172-31-90-68.ec2.internal
DEBUG:root:Task taskHelloWorld is starting.
DEBUG:root:Task taskDIR is starting.
INFO:root:Task taskHelloWorld running in 172.31.90.68. Moving to running list
INFO:root:Task taskDIR running in 172.31.90.68. Moving to running list
INFO:root:Received status TASK_FINISHED for task id: taskDIR
INFO:root:Received status TASK_FINISHED for task id: taskHelloWorld
DEBUG:root:Received new offers
INFO:root:Recieved resource offers: ['b7e3aa64-2820-4f1c-8da3-564894574c49-040']
INFO:root:Idle Tasks List Empty, Suppressing Offers
```

E como se pode observar o diretório ***HelloMesos*** é criado após a execução da tarefa no scheduler:

```
ubuntu@ip-172-31-90-68:~$ ls
HelloMesos  scheduler.py
```

Além disso, é possível ver o framework criado na dashboard do Mesos, com seus detalhes e tarefas executadas.

Frameworks

Agents

Roles

Offers

Maintenance

Master

Frameworks

Active Frameworks

ID ▼

Host

User

Name

Roles

Principal

Active Tasks

CPUs

GPUs

Mem

Disk

Max Share

Registered

Re-Registered

...f125560db748

ip-172-31-90-68

ubuntu

InspierMesos

\*

0

0

0

0 B

0 B

0%

just now

just now

Clicando no **ID** do framework detalhamos suas tarefas:

Mesos

Frameworks

Agents

Roles

Offers

Maintenance

Master

Framework

742bbf32-abd0-40cb-9a3e-f125560db748

Name: InspierMesos

User: ubuntu

Roles: \*

Principal:

Registered: just now

Re-registered: just now

Active tasks: 0

Tasks

Staging

0

Starting

0

Running

0

Unreachable

0

Killing

0

Finished

2

Killed

0

Failed

0

Lost

0

Resources

Allocated

Offered

CPUs

0

1

GPUs

0

0

Memory

0 B

495 MB

Disk

0 B

495 MB

Active Tasks

ID ▼	Name	Role	State	Health	Started	Host	
------	------	------	-------	--------	---------	------	--

Unreachable Tasks

ID ▼	Name	Role	Started	Agent ID
------	------	------	---------	----------

Completed Tasks

▼

Find...

ID ▼	Name	Role	State	Started	Stopped	Host	
taskHelloWorld	task taskHelloWorld	*	FINISHED	just now	just now	ip-172-31-90-68.ec2.internal	<a href="#">Sandbox</a>
taskDIR	task taskDIR	*	FINISHED	just now	just now	ip-172-31-90-68.ec2.internal	<a href="#">Sandbox</a>

## REFERÊNCIAS

1. <https://linuxacademy.com/guide/25034-introduction-to-apache-mesos/>
2. <https://dzone.com/articles/introduction-to-apache-mesos>
3. <https://www.baeldung.com/apache-mesos>
4. <https://www.baeldung.com/mesos-kubernetes-comparison>
5. <https://mapr.com/blog/apache-mesos-vs-hadoop-yarn-whiteboard-walkthrough/>
6. <http://mesos.apache.org/documentation/latest/app-framework-development-guide/>
7. <https://github.com/smurli/pymesos-sample>
8. <https://linuxacademy.com/guide/25034-introduction-to-apache-mesos/>
9. <https://github.com/mesosphere/RENDLER/tree/master/python>