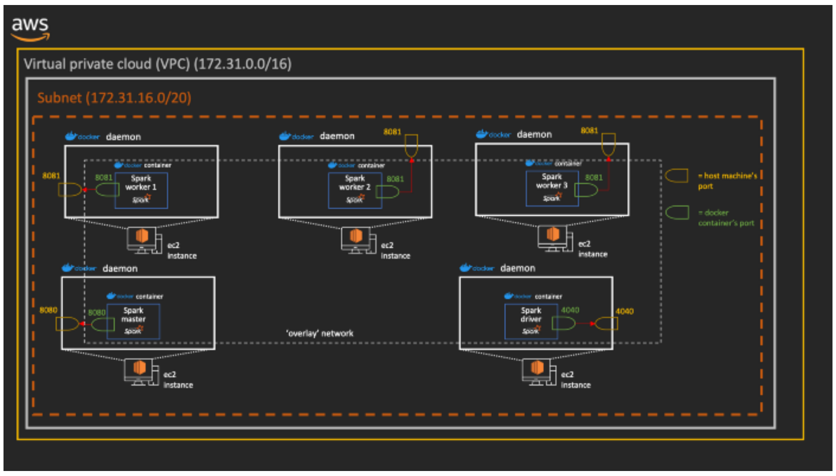
Published in Towards Data Science

Shane De Silva Follow
May 7, 2020 · 16 min read ★ · Listen

DATA SCIENCE FOR ALL

DIY: Apache Spark & Docker

Affectionately known as `Dark’ or `Spocker’



Fully distributed Spark cluster running inside of Docker containers

Introduction

Two technologies that have risen in popularity over the last few years are Apache Spark and Docker.

Apache Spark provides users with a way of performing CPU intensive tasks in a distributed manner. It’s adoption has been steadily increasing in the last few years due to its speed when compared to other distributed technologies such as Hadoop. In 2014 Spark won the Gray Sort Benchmark test in which they sorted 100TB of data 3x faster using 10x fewer machines then a Hadoop cluster previously did.

Docker on the other hand has seen widespread adoption in a variety of situations. Docker provides users the ability to define minimal specifications of environments meaning you can easily develop, ship, and scale applications. Furthermore, due to its use of linux containers users are able to develop Docker containers that can run be run simultaneously on a single server whilst remaining isolated from each other. Finally, Docker provides an abstraction layer called the Docker Engine that guarantees compatibility between machines that can run Docker solving the age-old headache of “it works on my machine, I don’t know why it doesn’t on yours”.

With the rise of Big Data these two technologies are a matched made in heaven. Apache Spark providing the analytics engine to crunch the numbers and Docker providing fast, scalable deployment coupled with a consistent environment.

I assume knowledge of Docker commands and terms as well as Apache Spark concepts. Therefore, I do not recommend this article if either of these two technologies are new to you. With considerations of brevity in mind this article will intentionally leave out much of the detail of what is happening. For a full drawn out description of the architecture and a more sequential walk through of the process I direct the reader to my github repo.

I also assume that you have at least basic experience with a cloud provider and as such are able to set up a computing instance on your preferred platform.

The rest of this article is going to be a fairly straight shot at going through varying levels of architectural complexity:

1. Docker container networking — local machine
2. Docker container networking — multiple machines
3. Apache Spark cluster — local machine
4. Docker & Spark — local machine
5. Docker & Spark — multiple machines
6. BONUS: Docker Stack & Spark

Let's get to it!

[source]

Docker — Local Machine

First we need to get to grips with some basic Docker networking. We will do this with the containers running on the same machine in the first instance.

1. The first thing to do is to either build the docker images using the Dockerfiles from my repo or more conveniently just pull the docker images using the following commands

NOTE: For the purpose of this section any images will do.

2. Create a bridge network by running

```
docker network create --driver bridge spark-net-bridge
```

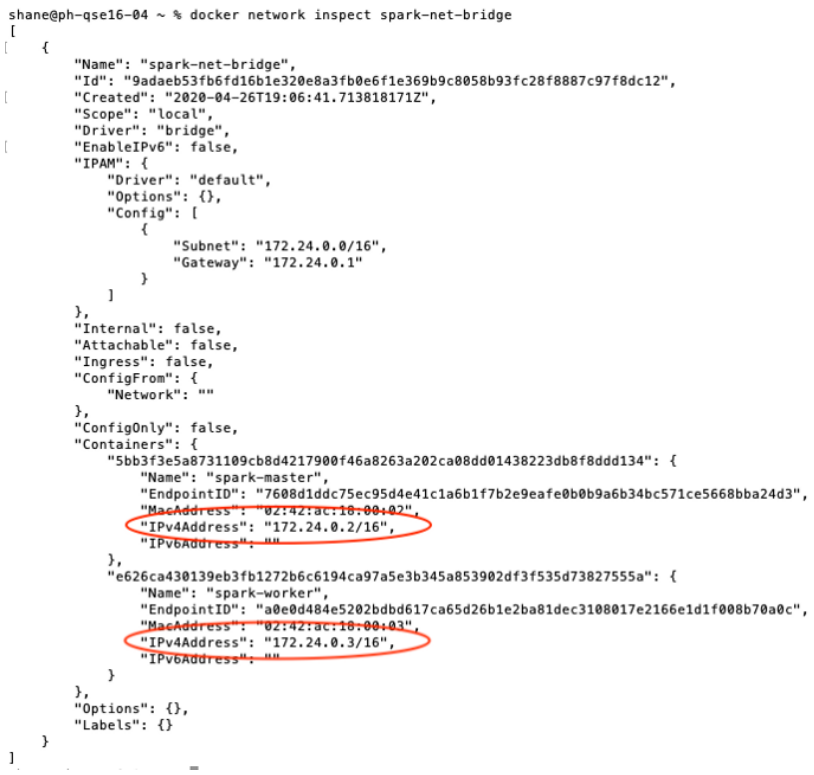
3. Run two containers on this user-defined bridge network by running

```
docker run -dit --name spark-master --network spark-net-bridge --
entrypoint /bin/bash sdesilva26/spark_master:0.0.2
docker run -dit --name spark-worker1 --network spark-net-bridge --
entrypoint /bin/bash sdesilva26/spark_worker:0.0.2
```

4. Inspect the network and find the IP addresses of the two containers

```
docker network inspect spark-net-bridge
```

The output of the above should look like the image below



```
shane@ph-qse16-04 ~ % docker network inspect spark-net-bridge
[
    {
        "Name": "spark-net-bridge",
        "Id": "9adaeb53fb6fd16b1e320e8a3fb0e6f1e369b9c8058b93fc28f8887c97f8dc12",
        "Created": "2020-04-26T19:06:41.713818171Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.24.0.0/16",
                    "Gateway": "172.24.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "5bb3f3e5a8731109cb8d4217900f46a8263a202ca08dd01438223db8f8ddd134": {
                "Name": "spark-master",
                "EndpointID": "7608d1ddc75ec95d4e41c1a6b1f7b2e9eafe0b0b9a6b34bc571ce5668bba24d3",
                "MacAddress": "02:42:ac:18:00:02",
                "IPv4Address": "172.24.0.2/16",
                "IPv6Address": ""
            },
            "e626ca430139eb3fb1272b6c6194ca97a5e3b345a853902df3f535d73827555a": {
                "Name": "spark-worker",
                "EndpointID": "a0e0d484e5202bdbd617ca65d26b1e2ba81dec3108017e2166e1d1f008b70a0c",
                "MacAddress": "02:42:ac:18:00:03",
                "IPv4Address": "172.24.0.3/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```

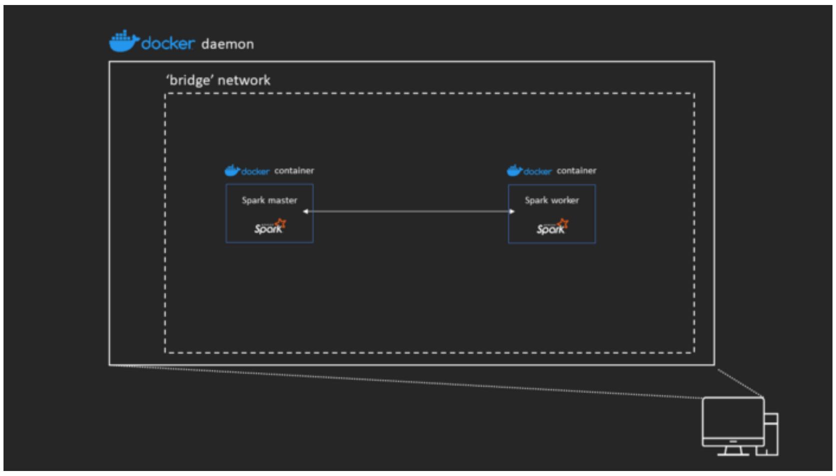
5. Attach to the spark-master container and test it's communication to the spark-worker container using both it's IP address and then using its container name

```
ping -c 2 172.24.0.3
ping -c 2 spark-worker
```

name. This is called **automatic service discovery** and will be a great help to us later.

Architecture

With the above commands we have created the following architecture.



Docker — Multiple Machines

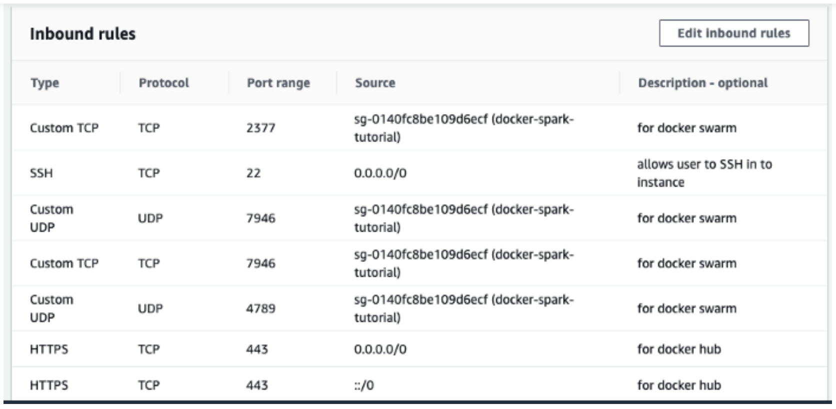
1. Go ahead and setup 2 instances on your favourite cloud provider
2. When configuring the instance's network make sure to deploy them into the same subnet
3. Open up the following ports for the containers to communicate with each other and for overlay network traffic (inbound and outbound);

```
Protocol  |  Port(s)  | Source
TCP       |  2377     | <your-security-group-or-subnet-name>
TCP       |  7946     | <your-security-group-or-subnet-name>
UDP       |  7946     | <your-security-group-or-subnet-name>
UDP       |  4789     | <your-security-group-or-subnet-name>
```

4. Open up the following ports for the instance to communicate with docker hub (inbound and outbound);

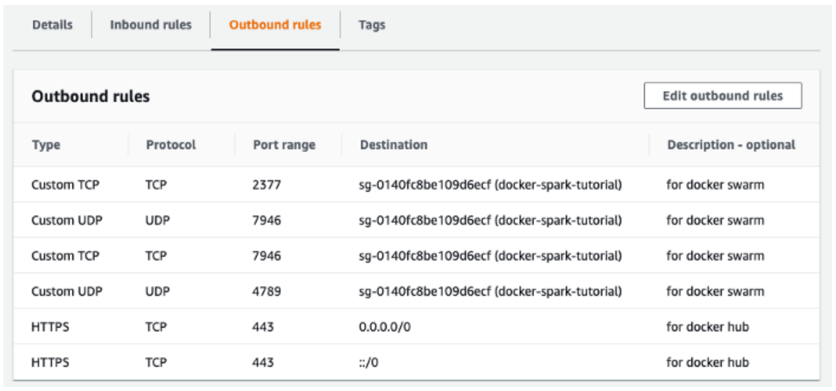
```
Protocol  |  Port(s)  | Source
HTTPS     |  443      | 0.0.0.0/0, ::/0
```

For example, on AWS, my security group which my two instances are deployed in have the following security group settings



Inbound rules

Type	Protocol	Port range	Source	Description - optional
Custom TCP	TCP	2377	sg-0140fc8be109d6ecf (docker-spark-tutorial)	for docker swarm
SSH	TCP	22	0.0.0.0/0	allows user to SSH in to instance
Custom UDP	UDP	7946	sg-0140fc8be109d6ecf (docker-spark-tutorial)	for docker swarm
Custom TCP	TCP	7946	sg-0140fc8be109d6ecf (docker-spark-tutorial)	for docker swarm
Custom UDP	UDP	4789	sg-0140fc8be109d6ecf (docker-spark-tutorial)	for docker swarm
HTTPS	TCP	443	0.0.0.0/0	for docker hub
HTTPS	TCP	443	::/0	for docker hub



Outbound rules

Type	Protocol	Port range	Destination	Description - optional
Custom TCP	TCP	2377	sg-0140fc8be109d6ecf (docker-spark-tutorial)	for docker swarm
Custom UDP	UDP	7946	sg-0140fc8be109d6ecf (docker-spark-tutorial)	for docker swarm
Custom TCP	TCP	7946	sg-0140fc8be109d6ecf (docker-spark-tutorial)	for docker swarm
Custom UDP	UDP	4789	sg-0140fc8be109d6ecf (docker-spark-tutorial)	for docker swarm
HTTPS	TCP	443	0.0.0.0/0	for docker hub
HTTPS	TCP	443	::/0	for docker hub

where “sg-0140fc8be109d6ecf (docker-spark-tutorial)” is the name of the security group itself, so only traffic from within the network can communicate using ports 2377, 7946, and 4789.

5. Install docker.

```
sudo yum install docker -y
sudo service docker start
sudo usermod -a -G docker ec2-user # This avoids you having to use
sudo everytime you use a docker command (log out and then in to your
instance for this to take affect)
```

6. On instance 1, pull a docker image of your choice.

```
docker pull sdesilva26/spark_master:0.0.2
```

7. Pull another image onto instance 2.

```
docker pull sdesilva26/spark_worker:0.0.2
```

8. Initialise a docker swarm and make instance 1 the swarm manager by running

```
docker swarm init
```

10. On instance 1 (the swarm manager) create an overlay network

```
docker network create -d overlay --attachable spark-net
```

11. On instance 1, run a container

```
docker run -it --name spark-master --network spark-net --entrypoint
/bin/bash sdesilva26/spark_master:0.0.2
```

12. On instance 2, run a container within the overlay network created by the swarm manager

```
docker run -it --name spark-worker --network spark-net --entrypoint
/bin/bash sdesilva26/spark_worker:0.0.2
```

13. From inside the container on instance 2 check the container communication by pinging the container running on instance 1

```
ping -c 2 spark-master
```

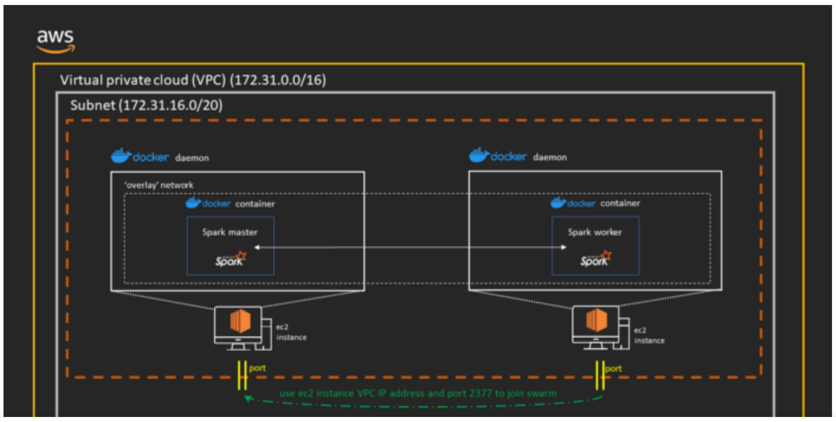
14. Similarly, check the backwards connection from the container in instance 1 to the container in instance 2

```
ping -c 2 spark-worker
```

As before, the containers are able to resolve each other's IP address using only the container name since they are within the same overlay network.

Architecture

After following the instructions above, you have created an architecture similar to the one below.



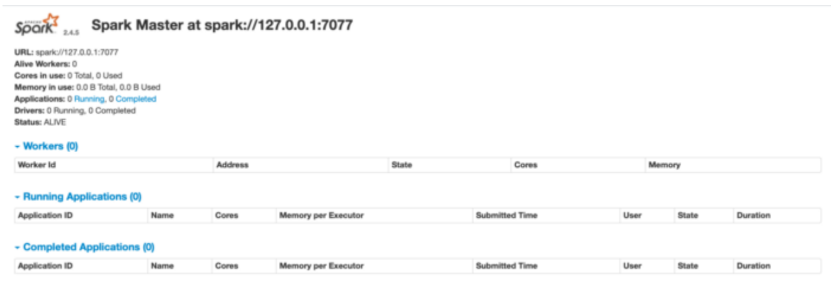
Apache Spark — Local Machine

Now that we have a handle on how to get two different docker hosts to communicate, we will get started on creating a Spark cluster on our local machine.

1. Install Spark from their website
2. From the command line navigate to the bin directory of your Spark installation
3. Setup a Spark master node

```
./spark-class org.apache.spark.deploy.master.Master
```

4. Check your master node has successfully been deploy by navigating to http://localhost:8080. You should see the following



Spark 2.4.5 **Spark Master at spark://127.0.0.1:7077**

URL: spark://127.0.0.1:7077
Alive Workers: 0
Cores in use: 0 Total, 0 Used
Memory in use: 0.0 B Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

- Workers (0)

Worker Id	Address	State	Cores	Memory

- Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration

- Completed Applications (0)

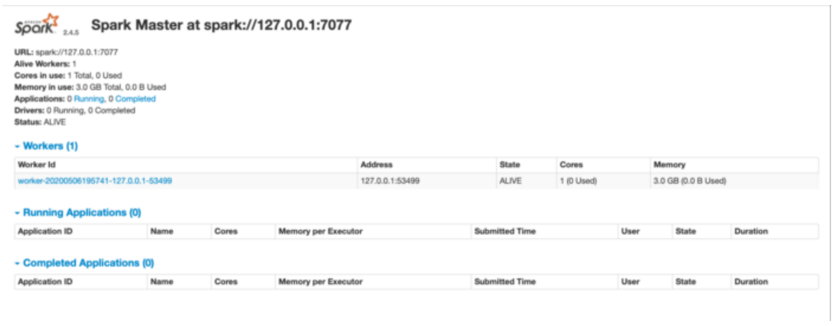
Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration

5. Attach a worker node to the cluster

```
./spark-class org.apache.spark.deploy.worker.Worker -c 1 -m 3G
spark://localhost:7077
```

where the two flags define the amount of cores and memory you wish this worker to have. The last input is the address and port of the master node prefixed with "spark://" because we are using spark's standalone cluster manager

6. Check that the worker was successfully registered with the master node by going back to http://localhost:8080. You should now see the worker node as a resource of the cluster. (You can also check the UI of the worker by going to http://localhost:8081)



Spark 2.4.5 **Spark Master at spark://127.0.0.1:7077**

URL: spark://127.0.0.1:7077
Alive Workers: 1
Cores in use: 1 Total, 0 Used
Memory in use: 3.0 GB Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

- Workers (1)

Worker Id	Address	State	Cores	Memory
worker-20200506195741-127.0.0.1-53499	127.0.0.1:53499	ALIVE	1 (0 Used)	3.0 GB (0.0 B Used)

- Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration

- Completed Applications (0)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration

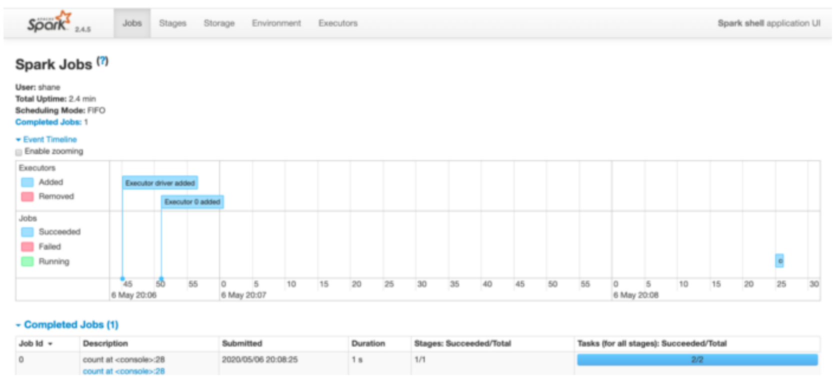
```
./spark-shell --master spark://localhost:7077
```

and running

```
val NUM_SAMPLES=10000
var count = sc.parallelize(1 to NUM_SAMPLES).filter { _ =>
  val x = math.random
  val y = math.random
  x*x + y*y < 1
}.count() * 4/(NUM_SAMPLES.toFloat)
```

This will return an estimate of the value of pi.

8. Check the UI of the application by going to http://localhost:4040. You should see something similar to



Spark Jobs (?)

User: shane
Total Uptime: 2.4 min
Scheduling Mode: FIFO
Completed Jobs: 1

Completed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	count at <console>:28 count at <console>:28	2020/05/06 20:08:25	1 s	1/1	2/2

You now have a fully functioning spark cluster!

Spark & Docker — Local Machine

Now it's time to start tying the two together. We will now learn to walk before running by setting up a Spark cluster running inside Docker containers on your local machine

1. Create a user defined bridge network (if you haven't done so already)

2. Create a Spark master node inside of the bridge network

```
docker run -it --name spark-master --network spark-net -p 8080:8080
sdesilva26/spark_master:0.0.2 bash
```

3. Check the container successfully started a Spark master node by navigating to

```
docker run -dit --name spark-worker1 --network spark-net -p 8081:8081
-e MEMORY=2G -e CORES=1
  sdesilva26/spark_worker:0.0.2 bash
```

By default the sdesilva26/spark_worker:0.0.2 image, when run, will try to join a Spark cluster with the master node located at spark://spark-master:7077.

If you change the name of the container running the Spark master node (step 2) then you will need to pass this container name to the above command, e.g. -e <MASTER_CONTAINER_NAME>. See the dockerfile here.

5. Again, verify the worker has successfully registered with the master node by navigating to http://localhost:8080 and http://localhost:8081.

6. Attach a second spark worker to the cluster

```
docker run -dit --name spark-worker2 --network spark-net -p 8082:8081
-e MEMORY=2G -e CORES=1
  sdesilva26/spark_worker:0.0.2 bash
```

The only change we had to make from the command in step 4 was that we had to give the container a unique name and also we had to map port 8081 of the container to port 8082 of the local machine since the spark-worker1 container is already using your local machines port 8081.

7. Spin up a Spark submit node

```
docker run -it --name spark-submit --network spark-net -p 4040:4040
sdesilva26/spark_submit bash
```

You should now be inside of the spark-submit container.

8. Open a scala shell and connect to the Spark cluster

```
$SPARK_HOME/bin/spark-shell --conf spark.executor.memory=2G --conf
spark.executor.cores=1 --master spark://spark-master:7077
```

As before, if you have a different name than spark-master for the container running your Spark master node then you would change the above command with — master spark://<YOUR_MASTER_CONTAINER>:7077.

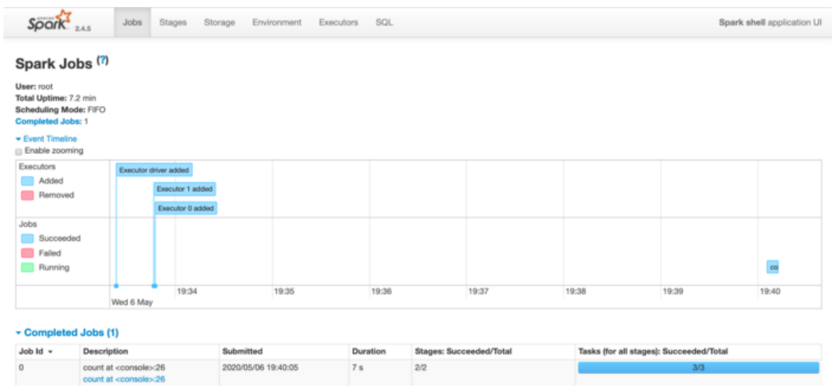
The above command also asks that on your cluster, you want each executor to contain 2G of memory and 1 core. The Spark master node will allocate these executors, provided there is enough resource available on each worker to allow this. For an explanation of executors and workers see the following article.

9. Run an example job in the interactive scala shell

```
val divisBy2 = myRange.where("number % 2 = 0")

divisBy2.count()
```

10. Check the application UI by navigating to http://localhost:4040. You should see the following



Spark Jobs (?)

User: root
Total Uptime: 7.2 min
Scheduling Mode: FIFO
Completed Jobs: 1

▾ Event Timeline
Enable zooming

▾ Completed Jobs (1)

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	count at <console>:26 count at <console>:26	2020/05/06 19:40:05	7 s	2/2	3/3

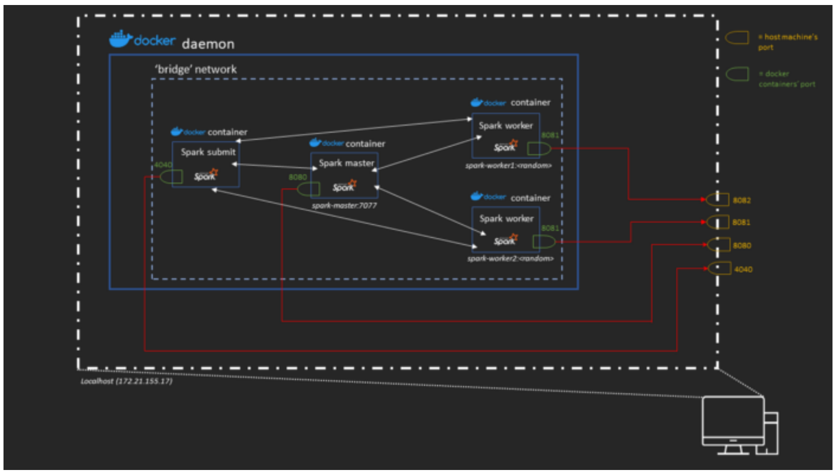
You have just run a Spark job inside of Docker containers. Spocker is born!

Architecture

What we have done in the above is created a network within Docker in which we can deploy containers and they can freely communicate with each other.

The white arrows in the diagram below represents open communication between containers. Ports on the containers are shown in green and the ports of your local machine are shown in yellow.

You can see that all the container are deployed within the bridge network. If we now deploy a container outside of this network, it would not be able to resolve the IP addresses of the other containers just by using their container names.



Docker & Spark — Multiple Machines

Now let's wrap everything together to form a fully distributed Spark cluster

```
docker pull sdesilva26/spark_master:0.0.2
docker pull sdesilva26/spark_worker:0.0.2
docker pull sdesilva26/spark_submit:0.0.2
```

You can also build them yourself by downloading the Dockerfiles

1. Start up an instance on a cloud provider of your choice and make it a docker swarm manager

```
docker swarm init
```

2. Copy and paste the output of the above command to at least 2 other instances. I have done this on 4 other instances — 3 will act as Spark workers and 1 will be my Spark submit node

3. On instance 1, create an overlay network as we did before

```
docker network create -d overlay --attachable spark-net
```

4. Run the spark_master image to create a container that will be the Spark master node

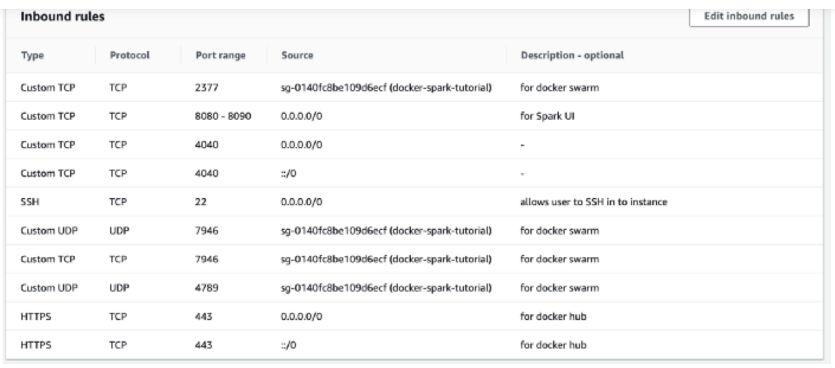
```
docker run -it --name spark-master --network spark-net -p 8080:8080
sdesilva26/spark_master:0.0.2
```

5. Open up ports 8080–8090 and 4040 by adding the following to your security group's inbound rules

```
Protocol    |  Port(s)   | Source
Custom TCP  |  8080-8090 | 0.0.0.0/0
Custom TCP  |    4040    | 0.0.0.0/0, ::/0
```

NOTE: In AWS security groups are stateful, so the return traffic from the instance to users is allowed automatically, so you don't need to modify the security group's outbound rules. On other cloud providers you may have to add a similar rule to your outbound rules.

My inbound security group rules now look like this


Inbound rules Edit inbound rules

Type	Protocol	Port range	Source	Description - optional
Custom TCP	TCP	2377	sg-0140fc8be109d6ecf (docker-spark-tutorial)	for docker swarm
Custom TCP	TCP	8080 - 8090	0.0.0.0/0	for Spark UI
Custom TCP	TCP	4040	0.0.0.0/0	-
Custom TCP	TCP	4040	::/0	-
SSH	TCP	22	0.0.0.0/0	allows user to SSH in to instance
Custom UDP	UDP	7946	sg-0140fc8be109d6ecf (docker-spark-tutorial)	for docker swarm
Custom TCP	TCP	7946	sg-0140fc8be109d6ecf (docker-spark-tutorial)	for docker swarm
Custom UDP	UDP	4789	sg-0140fc8be109d6ecf (docker-spark-tutorial)	for docker swarm
HTTPS	TCP	443	0.0.0.0/0	for docker hub
HTTPS	TCP	443	::/0	for docker hub

6. Check the Spark master node UI at http://<PUBLIC_IPv4_ADDRESS_OF_INSTANCE>:8080. You should see the same UI that we saw earlier.

7. Now on another one of your instances run the following to attach a Spark worker node to the cluster

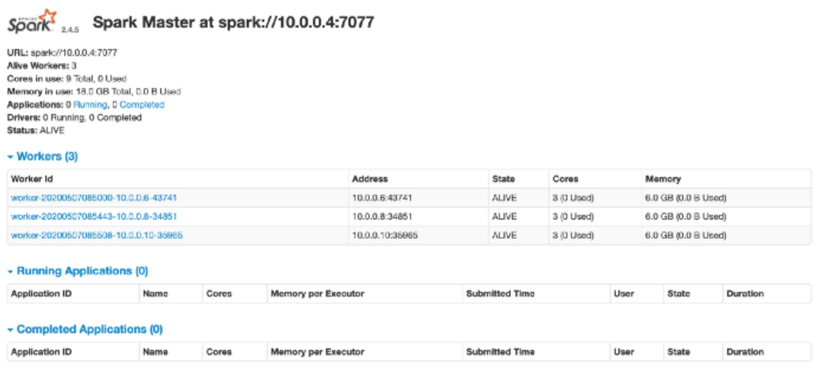
```
docker run -it --name spark-worker1 --network spark-net -p 8081:8081 -
e MEMORY=6G -e CORES=3 sdesilva26/spark_worker:0.0.2
```

NOTE: As a general rule of thumb start your Spark worker node with memory = memory of instance-1GB, and cores = cores of instance - 1. This leaves 1 core and 1GB for the instance's OS to be able to carry out background tasks.

8. Again, check the master node's web UI to make sure the worker was added successfully.

9. Rinse and repeat step 7 to add as many Spark workers as you please. Make sure to increment the name of the container though from spark-worker1 to spark-worker2, and so on.

I have connected 3 workers and my master node's web UI looks like this


Spark 2.4.5 **Spark Master at spark://10.0.0.4:7077**

URL: spark://10.0.0.4:7077
Alive Workers: 3
Cores in use: 9 Total, 0 Used
Memory in use: 18.0 GB Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (3)

Worker Id	Address	State	Cores	Memory
worker-20200507085030-10.0.0.6-43741	10.0.0.6:43741	ALIVE	3 (0 Used)	6.0 GB (0.0 B Used)
worker-20200507085443-10.0.0.8-34851	10.0.0.8:34851	ALIVE	3 (0 Used)	6.0 GB (0.0 B Used)
worker-20200507085508-10.0.0.10-35965	10.0.0.10:35965	ALIVE	3 (0 Used)	6.0 GB (0.0 B Used)

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration

Completed Applications (0)

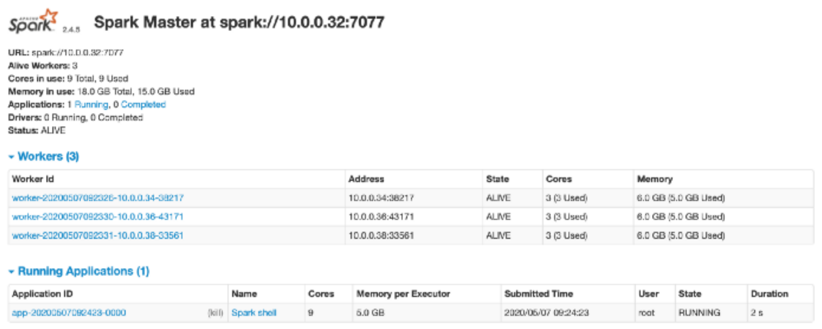
Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration

10. In another instance, fire up a Spark submit node

```
docker run -it --name spark-submit --network spark-net -p 4040:4040
sdesilva26/spark_submit:0.0.2 bash
```

NOTE: You specify the resources you would like each executor to have when connecting an application to the cluster by using the — conf flag. The topic of Spark tuning is a whole post in itself so I will not go into any detail here. This two part Cloudera blog post I found to be a good resource for understanding resource allocation: part 1 & part 2. Also a blog post by Anthony Shipman from C2FO.io I found very useful and also includes a handy excel sheet to work out settings for memory, cores, and parallelization.

12. Check the submit node has successfully connected to the cluster by checking both the Spark master node's UI and the Spark submit node's UI. They should look like the images below.



Spark Master at spark://10.0.0.32:7077

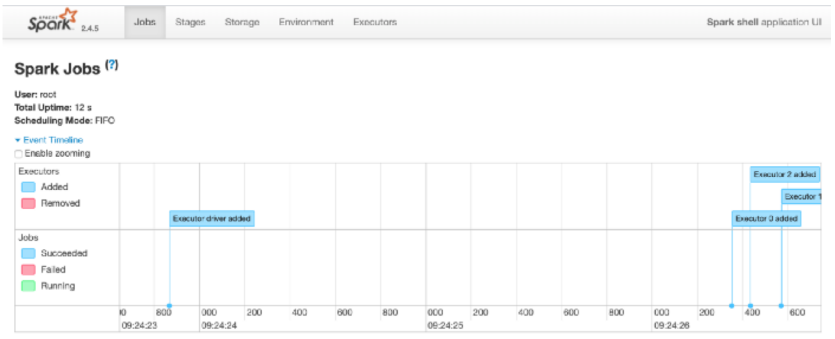
URL: spark://10.0.0.32:7077
Alive Workers: 3
Cores in use: 9 Total, 9 Used
Memory in use: 18.0 GB Total, 15.0 GB Used
Applications: 1 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (3)

Worker Id	Address	State	Cores	Memory
worker-20200507092326-10.0.0.34-38217	10.0.0.34:38217	ALIVE	3 (3 Used)	6.0 GB (5.0 GB Used)
worker-20200507092330-10.0.0.36-43171	10.0.0.36:43171	ALIVE	3 (3 Used)	6.0 GB (5.0 GB Used)
worker-20200507092331-10.0.0.38-33561	10.0.0.38:33561	ALIVE	3 (3 Used)	6.0 GB (5.0 GB Used)

Running Applications (1)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20200507092423-0000 (kill)	Spark shell	9	5.0 GB	2020/05/07 09:24:23	root	RUNNING	2 s



Spark Jobs (?)

User: root
Total Uptime: 12 s
Scheduling Mode: FIFO

Event Timeline

13. Run a sample job from the pyspark shell

```
from random import random

def inside(p):
    x, y = random(), random()
    return x*x + y*y < 1

NUM_SAMPLES = 100000

count = sc.parallelize(range(0, NUM_SAMPLES)).filter(inside).count()

print("Pi is roughly {:0.4f}".format(4.0 * count / NUM_SAMPLES))
```

The more common way to submit jobs to a Spark cluster is by using the spark-submit script which is included with your spark installation. Let's also do this.

14. Exit out of pyspark and submit a program to executor on the cluster

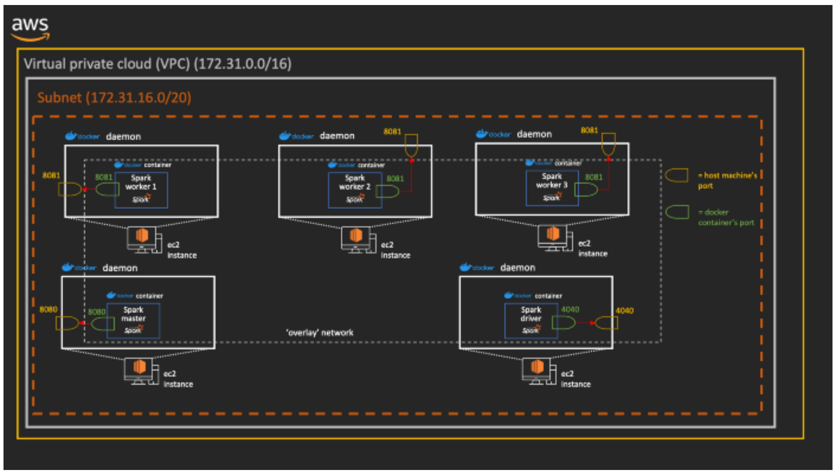
```
$SPARK_HOME/examples/src/main/python/pi.py 20
```

It's pretty much the same syntax as before except we are calling the spark-submit script and we are passing it a .py file along with any other configurations for the file to execute.

Happy days! We have now created a fully distributed Spark cluster running inside of Docker containers and submitted an application to the cluster.

Architecture

The architecture we have just created looks like the following



Each Spark worker node and the master node is running inside a Docker container located on its own computing instance. The Spark driver node (spark submit node) is also located within its own container running on a separate instance. All the Docker daemons are connected by means of an overlay network with the Spark master node being the Docker swarm manager in this case.

Within the overlay network, containers can easily resolve each other's addresses by referencing container names which utilises automatic service discovery.

More Spark worker nodes can be fired up on additional instances if needed.

Welcome to the finish line!

Photo by Jonathan Chng on Unsplash

Conclusion

In this tutorial we have managed to sequentially step through the varying levels of complexity in setting up a Spark cluster running inside of Docker containers.

We first started off with some simple Docker networking principles both on our local machine, in which we used a bridge network, and then on distributed machines using an overlay network with a docker swarm.

Next we set up a Spark cluster running on our local machine to get to grips with adding workers to a cluster.

Then we introduced Docker back in to the mix and set up a Spark cluster running inside of Docker containers on our local machine.

Finally, we brought everything together to construct a fully distributed Spark cluster running in Docker containers.

Hopefully you've gained a clear understanding of how these two technologies link together and could provide benefit to your specific problem or project.

Now is the time for you to start experimenting and see what you can learn using this architecture.

Enjoy!

BONUS

The above steps of manually creating a cluster was more informative than practical as it required a lot of manual typing and repeated commands.

For small(ish) problems where you only need the resources of maybe 4 or 5 computing instances this amount of effort is probably below your pain threshold.

However, as the data becomes truly large and the computing power needed starts to increase, following the above steps will turn you into a full-time cluster creator.

A much more practical and elegant way of setting up a cluster is by taking advantage of **Docker compose**

Docker compose

For those of you new to Docker compose, it allows you to launch what are called "**services**".

A service is made up of a single Docker image, but you may want multiple containers of this image to be running. For example, running multiple Spark worker containers from the docker image sdesilva26/spark_worker:0.0.2 would constitute a single service.

To launch a set of services you create a docker-compose.yml file which specifies everything about the various services you would like to run.

However, Docker compose is used to create services running on a single host. It does not support deploying containers across hosts.

Enter **Docker stack.**

The Docker stack is a simple extension to the idea of Docker compose. Instead of running your services on a single host, you can now run your services on multiple hosts which are connected as a Docker swarm.

Best of all, if you have a Docker compose file very little modifications need to be made in order for it to work with the Docker stack commands.

Let's see how to create our distributed Spark cluster running inside of Docker containers using a compose file and docker stack.

1. The first step is to label the nodes in your Docker swarm. From the Docker swarm manager list the nodes in the swarm.

```
docker node ls
```

You should get a similar output to the image below.

```
zsn16pl3pn3ow3a0gfsdg1hjv *   ip-172-31-24-244     Ready               Active              Leader              18.09.9-ce
```

2. For any instances you wish to be Spark workers, add a label to them

```
docker node update --label-add role=worker x5kmfd8akvvtnsfvmxybcjb8w
```

3. Now label the instance you wish to run the Spark master node with the role of master

```
docker node update --label-add role=master
```

4. Create a docker-compose.yml file or pull the one I have created.

```
version: "3"
services:
  # Create a service called spark-master
  spark-master:
    image: "sdesilva26/spark_master:0.0.2"
    ports:
      - "8080:8080"
    networks:
      - spark-net
    deploy:
      placement:
        # set node labels using docker node update --label-add key=value <NODE ID> from s
        constraints:
          - node.labels.role==master
  # Create a second service called spark-worker
  spark-worker:
    image: "sdesilva26/spark_worker:0.0.2"
    ports:
      - "8081:8081"
    environment:
      - CORES=3
      - MEMORY=15G
    deploy:
      placement:
        # set node labels using docker node update --label-add key=value <NODE ID> from s
        constraints:
          - node.labels.role==worker
      # Deploy 3 containers for this service
      replicas: 3
    networks:
      - spark-net
# Create the spark-net network
networks:
  spark-net:
    driver: overlay
```

docker-compose.yml hosted with ❤ by GitHub view raw

In this compose file I have defined two services — spark-master and spark-worker.

The first service deploys a single container onto any node in the swarm that has the label “role=master”.

The second service will deploy 3 containers of the sdesilva26/spark_worker:0.0.2 image onto nodes with the label “role=worker”. If 3 suitable nodes in the swarm aren’t found it will deploy as many as is possible

5. Copy the docker-compose.yml into the instance which is the swarm manager.

```
scp -i <YOUR_KEY>.pem /path/to/docker-compose.yml ec2-
user@<PUBLIC_IP_ADDRESS_OF_INSTANCE>:/home/ec2-user/docker-compose.yml
```

[see here for alternative ways of doing this]

6. Finally, run your docker stack from the swarm manager and give it a name.

```
docker stack deploy --compose-file docker-compose.yml sparkdemo
```

NOTE: the name of your stack will be prepended to all service names. So the service "spark-master" becomes "sparkdemo_spark-master". You can check the services that are running by using

```
docker service ls
```

7. Check that the spark_worker image is running on the instances you labelled as 'worker' and that the spark_master image is running on the node labelled as 'master'.

Congratulations, we have just simplified all of the work from this article into a few commands from the Docker swarm manager.

The best thing about Docker services is that it is very easy to scale up. For example, if later on we added another instance to the Docker swarm and we then wished to scale up the "sparkdemo_spark-worker" service, we can simply run

```
docker service scale sparkdemo_spark-worker=4
```

and you now have 4 Spark workers in your cluster!

Docker FTW.

Photo by J E W E L M I T CH E L L on Unsplash