Docker Overlay Network

Mitch Dresdner

Table of Contents

Using Docker overlay networks for communicating across hosts

Summary

When we get started using Docker, the typical configuration is to create a standalone application on our desktop.

For the most part, it's usually not practical to run all your applications on a single machine and when it's not you'll need an approach for distributing the applications across many machines. This is where a Docker Swarm comes in.

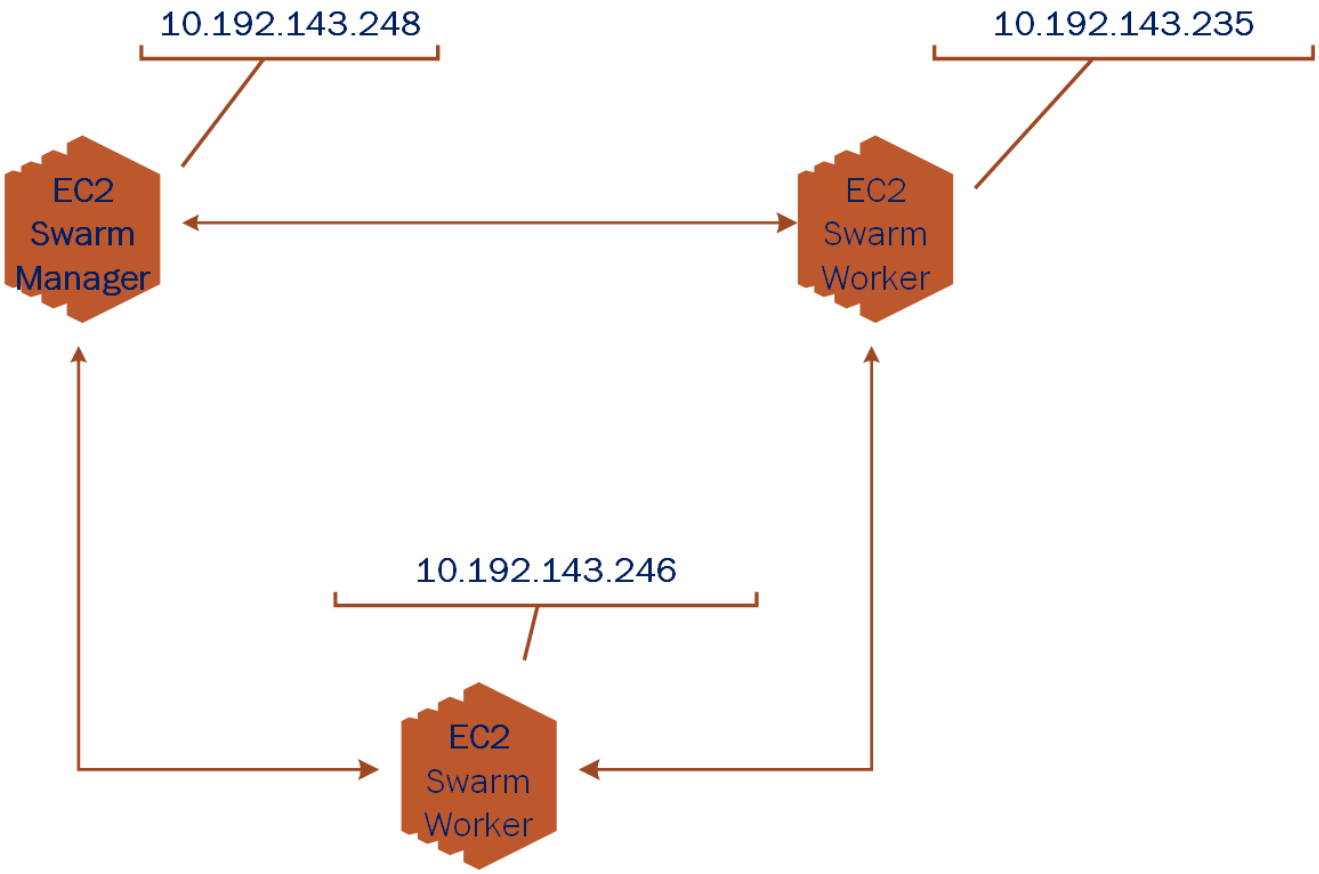
Docker Swarm provides capabilities for clustering, scalability, discovery and security to name a few. In this article, we'll create a basic Swarm configuration and perform some experiments to illustrate discovery and connectivity.

In this demo we'll create a Swarm overlay cluster that will consist of a Swarm manager and a worker, for convenience it will be running in AWS.

If you feel you're in need of a refresher on Docker Swarm or configuring your AWS account see Dzone article: Fun with Docker Swarm here.

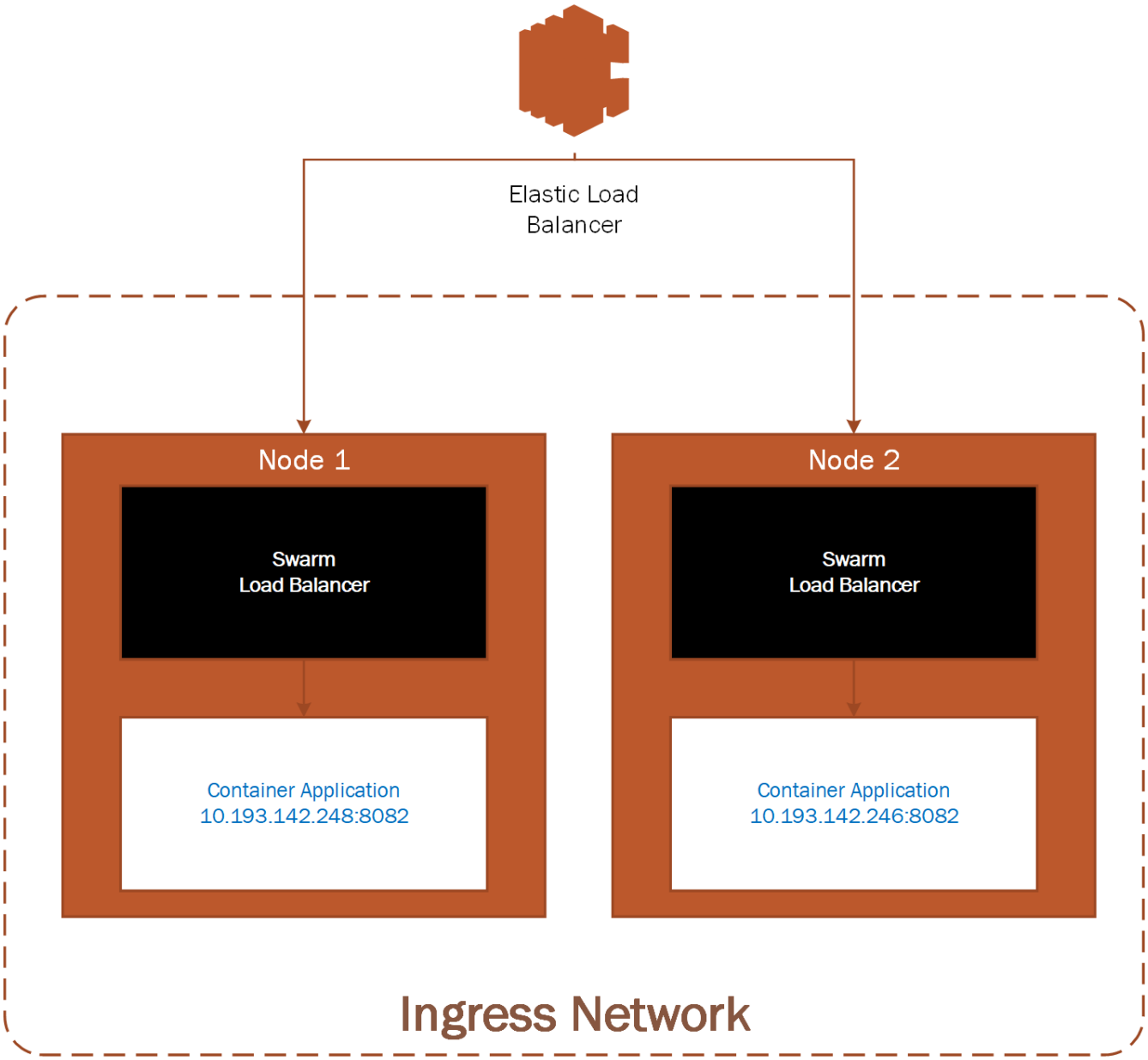
Architecture

Our target Architecture will consist of a couple of Docker containers running inside AWS AMI images on different EC2 hosts. The purpose of these examples is to demonstrate the concepts of how a Docker swarm can be used to discover services running on different host machines and communicate with one another.



In our hypothetical network above, we depict the inter connections of a Docker swarm manager and a couple of swarm workers. In the examples which follow we'll use a single manager and a single worker to keep complexity and cost low. Keep in mind that your real configurations will likely consist of many swarm workers.

Here's an example of what a potential Use Case may look like. An AWS load balancer configured to distribute load to a Docker swarm running on 2 or more EC2 instances.



We'll show in the examples below how you can create a Docker swarm overlay network that will allow DNS discovery of members and allow members to communicate with one another.

Prerequisites

We assume you're somewhat familiar with Docker and have some familiarity setting up EC2 instances in AWS.

If you're not confident with AWS or would like a little refresher please review the following articles:

Some refreshers before getting started

- Provision a free tier EC2 instance

- Configure Docker on your EC2 instance

Caveats

Some AWS services will incur charges, be sure to stop and/or terminate any services you aren't using. Additionally, consider setting up billing alerts to warn you of charges exceeding a threshold that may cause you concern.

Configuration

Begin by creating (2) EC2 instances, free tier should be fine, and install Docker on each EC2 instance. Refer to the Docker Supported platforms section for Docker installation guidance and instructions for your instance.

AWS ports to open to support Docker swarm and our port connection test:

Table 1. Open ports in AWS Mule SG

Type	Protocol	Port Range	Source	Description
Custom TCP Rule	TCP	2377	10.193.142.0/24	Docker swarm management
Custom TCP Rule	TCP	7946	10.193.142.0/24	Container network discovery
Custom UDP Rule	UDP	4789	10.193.142.0/24	Container ingress network
Custom TCP Rule	TCP	8083	10.193.142.0/24	Demo port for machine to machine communications

For our examples we'll use the following ip addresses to represent Node 1 and Node2.

- Node 1: *10.193.142.248*
- Node 2: *10.193.142.246*

Before getting started lets take a look at the existing Docker networks.

List Docker networks

```
docker network ls
```

The output of the network list should look at least like the listing below if you've never added a network or initialized a swarm on this Docker daemon. Other networks may be shown as well.

.Results of Docker network listing

```
NETWORK ID          NAME                DRIVER              SCOPE
fa977e47b9f3        bridge              bridge              local
705fc078c278        host                host                local
bd4caf6c1751        none                null                local
```

From Node 1 lets begin by initializing the swarm.

Create the swarm master node

```
docker swarm init --advertise-addr=10.193.142.248
```

You should get a response that looks like the one below, we'll use the token provided to join our other node to the swarm.

```
Swarm initialized: current node (v9c2un5lqf7iapnv96uobag00) is now a manager.

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

    docker swarm join --token SWMTKN-1-
5bbh9ksinfmajdqnsuef7y5ypbwj5d9jazt47urenz3ksuw9lk-227dtheygwbxt8dau8ul791a7
10.193.142.248:2377

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

It make take a minute or two for the Root CA Certificate to synchronize through the swarm, if you get an error give it a few minutes and try again.

If you happen to misplace the token, as of Docker you can use the *join-token* argument to list tokens for manager and workers. For example, on Node 1 run the following.

Manager token for Node 1

```
docker swarm join-token manager
```

Next, lets join the swarm from Node 2.

Node 2 joins swarm

```
docker swarm join --token SWMTKN-1-5bbh9ksinfmajdqnsuef7y5ypbwj5d9jazt47urenz3ksuw9lk-
227dtheygwbxt8dau8ul791a7 10.193.142.248:2377
This node joined a swarm as a worker.
```

From Node 1 the swarm master we can now look at the connected nodes

On Master, list all nodes

```
docker node ls
```

Results of listing nodes

```
ID                            HOSTNAME            STATUS    AVAILABILITY   ENGINE
VERSION
2quenyegseco1w0e5n1qe58r3     ip-10-193-142-248   Ready     Active         18.03.1-ce
wrjk02g909c6fnuxlepmksuz4     ip-10-193-142-246   Ready     Active         18.03.1-ce
```

Also, notice that an Ingress network has been created, this provides an entry point for our swarm network.

Results of Docker network listing

```
NETWORK ID          NAME                DRIVER              SCOPE
fa977e47b9f3        bridge              bridge              local
705fc078c278        host                host                local
bd4caf6c1751        none                null                local
qrppfipdu098        ingress             overlay             swarm
```

Lets go ahead and create our Overlay network for standalone containers

Create overlay network from Node 1

Overlay network creation on Node 1

```
docker network create --driver=overlay --attachable my-overlay-net

docker network ls
```

Results of Docker network listing

```
NETWORK ID          NAME                DRIVER              SCOPE
fa977e47b9f3        bridge              bridge              local
705fc078c278        host                host                local
bd4caf6c1751        none                null                local
qrppfipdu098        ingress             overlay             swarm
vn12jyorp1ey        my-overlay-net      overlay             swarm
```

Note the addition of our new overlay network to the swarm

Join the overlay network from Node 1

Run our container, join the overlay net

```
docker run -it --name alpine1 --network my-overlay-net alpine
```

Join the overlay network from Node 2, we'll open port *8083* to test connectivity into our running container.

Run our container, join the overlay net

```
docker run -it --name alpine2 -p 8083:8083 --network my-overlay-net alpine
```

Verify our overlay network connectivity

With our containers running we can test that we can discover our hosts using DNS configured by the swarm.

From Node 2 lets ping the Nod 1 container.

Node 2 pings Node 1, listens on port 8083

```
ip addr   # show our ip address
ping -c 2 alpine1

# create listener on 8083
nc -l -p 8083
```

From Node 1 lets ping the Node 2 container and connect to it's open listener on port *8083*.

Node 1 pings Node 2, connect to Node 2 listener on port 8083

```
ip addr   # show our ip address
ping -c 2 alpine2

# connect to alpine2 listener on 8083
nc alpine2 8083
Hello Alpine2
^C
```

Cleanup

With our testing complete we can tear down the swarm configuration.

Remove Node 2 swarm

```
docker container stop alpine2
docker container rm alpine2

docker swarm leave
```

Remove Node 1 swarm

```
docker container stop alpine1
docker container rm alpine1

docker swarm leave --force
```

There you have it, you created a tcp connection from Node 1 to Node 2 and sent a message. Similarly, your services can connect with and exchange data when running in the Docker overlay cluster. With these fundamental building blocks in place you're ready to apply these principles to real workd designs.

This concludes our brief examples with creating Docker Overlay Networks. With these fundamental building blocks in place, you now have the essential pieces necessary for building larger, more complex Docker container interactions.

Be sure to remove any AWS assets you may have used in these examples so you don't incur any ongoing costs.

I hope you enjoyed reading this article as much as I have enjoyed writing it, i'm looking forward to your feedback!

About the Author:

Mitch Dresdner is a Senior Mule Consultant at TerraThink