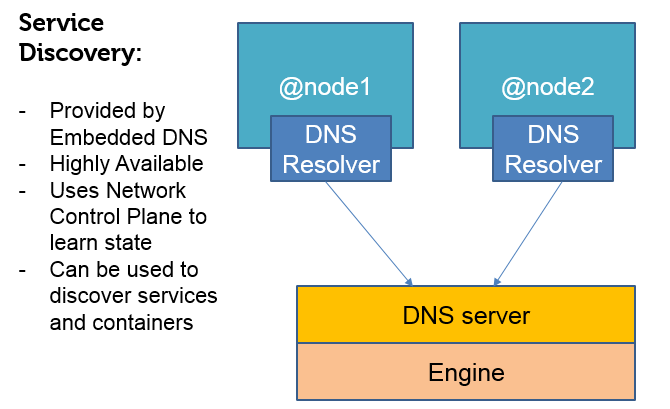
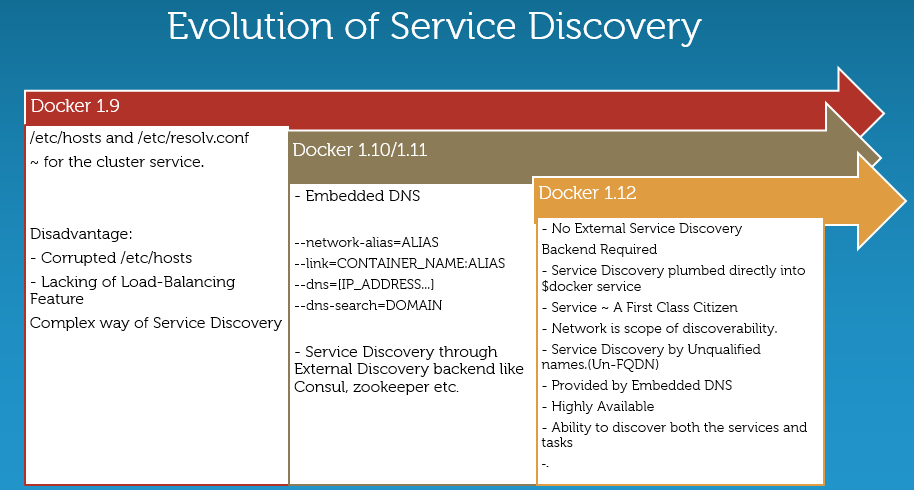
Prior to Docker 1.12 release, setting up Swarm cluster needed some sort of [service discovery backend](https://docs.docker.com/v1.11/swarm/discovery/). There are multiple discovery backends available like hosted discovery service, using a static file describing the cluster, etcd, consul, zookeeper or using static list of IP address.

[](https://collabnix.com/wp-content/uploads/2016/07/pic-intro.png)

**Thanks to Docker 1.12 Swarm Mode**, we don’t have to depend upon these external tools and complex configurations. [Docker Engine 1.12](https://github.com/docker/docker/releases/tag/v1.12.0-rc5) runs it’s own internal DNS service to route services by name. Swarm manager nodes assign each service in the swarm a unique DNS name and load balances running containers. You can query every container running in the swarm through a DNS server embedded in the swarm.

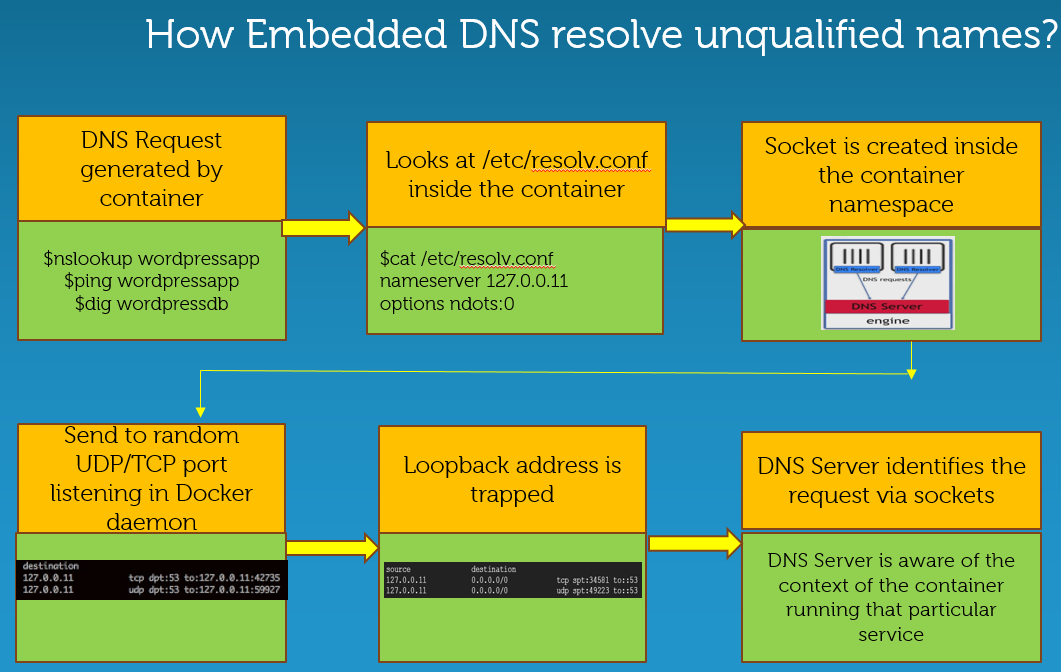
**How does it help?**

When you create a service and provide a name for it, you can use just that name as a target hostname, and it’s going to be automatically resolved to the proper container IP of the service. In short, within the swarm, containers can simply reference other services via their names and the built-in DNS will be used to find the appropriate IP and port automatically. It is important to note that if the service has multiple replicas, **the requests would be round-robin load-balanced**. This would still work if you didn’t forward any ports when you created your docker services.

[](https://collabnix.com/wp-content/uploads/2016/07/Pic10.png)

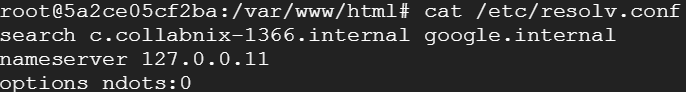
Embedded DNS is not a new concept. It was first included under Docker 1.10 release. Please note that DNS lookup for containers connected to user-defined networks works differently compared to the containers connected to default bridge network. As of Docker 1.10, the docker daemon implements an embedded DNS server which provides built-in service discovery for any container created with a valid name or net-alias or aliased by link. Moreover,container name configured using --name is used to discover a container within an user-defined docker network. The embedded DNS server maintains the mapping between the container name and its IP address (on the network the container is connected to).

**How does Embedded DNS resolve unqualified names?**

[](https://collabnix.com/wp-content/uploads/2016/07/Pic22.png)

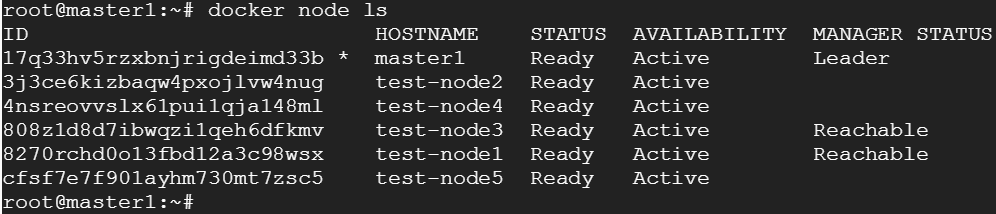
With Docker 1.12 release, a new API called “service” is being included which clearly talks about the functionality of service discovery.  It is important to note that Service discovery is scoped within the network. What it really means is –  If you have redis application and web client as two separate services , you combine into single application and put them into same network. If you try build your application in such a way that you are trying to reach to redis through name “redis”,it will always resolve to name “redis”. Reason – both of these services are part of the same network. You don’t need to be inside the application trying to resolve this service using FQDN. Reason – FQDN name is not going to be portable which in turn, makes your application non-portable.

Internally, there is a listener opened inside the container itself. If we try to enter into the container which is providing a service discovery and look at /etc/resolv.conf, we will find that the nameserver entry holds something really different like 127.0.0.11.This is nothing but a loopback address. So, whenever resolver tried to resolve, it will resolve to 127.0.0.11 and this request is rightly trapped.

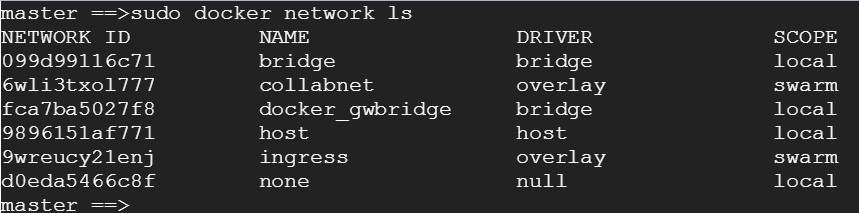
[](https://collabnix.com/wp-content/uploads/2016/07/Pic-12.png)

Once this request is trapped, it is sent to particular random UDP / TCP port currently being listened under the docker daemon. Consequently, the socket is to be created inside the namespace. When DNS server and daemon gets the request, it knows that this is coming from which specific network, hence gets aware of  the context of from where it is coming from.Once it knows the context, it can generate the appropriate DNS response.

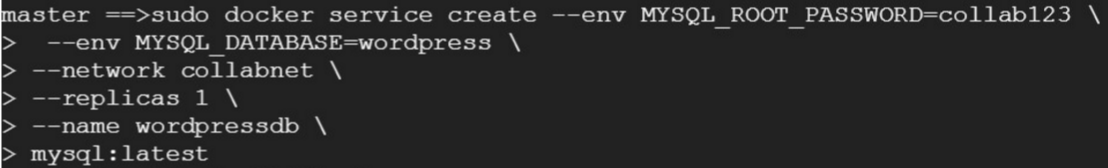
To demonstrate Service Discovery  under Docker 1.12, I have upgraded Docker 1.12.rc5 to 1.12.0 GA version. The swarm cluster look like:

[](https://collabnix.com/wp-content/uploads/2016/07/Pico01.png)

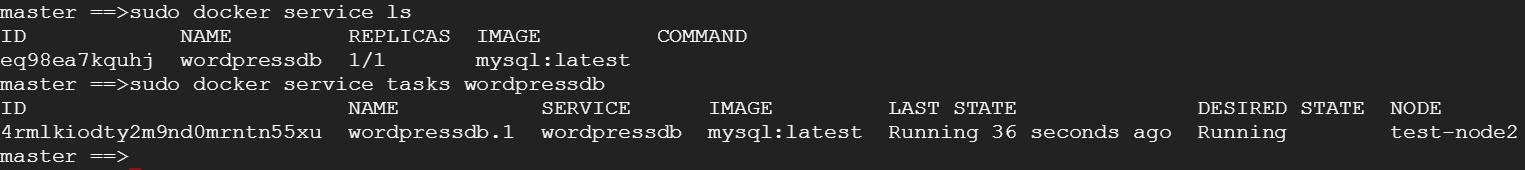
I have created a network called “collabnet” for the new services as shown below:

[](https://collabnix.com/wp-content/uploads/2016/07/Pic-2.jpg)

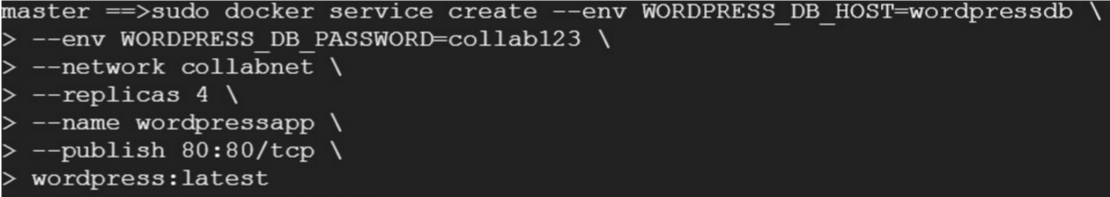
Let’s create a service called “wordpressdb” under collabnet network :

[](https://collabnix.com/wp-content/uploads/2016/07/pico-mysql.png)

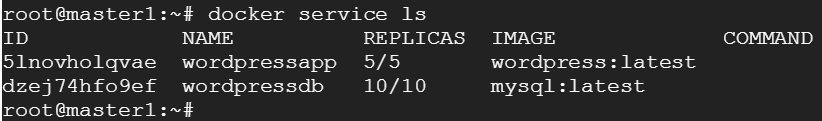
You can list the running tasks(containers) and the node on which these containers are running on:

[](https://collabnix.com/wp-content/uploads/2016/07/Pic-4.png)

Let’s create another service called “wordpressapp” under the same network:

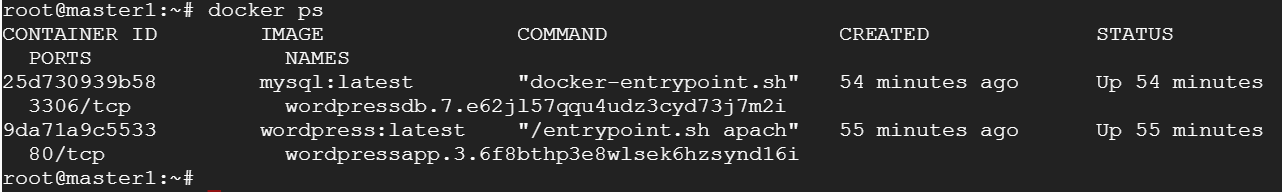
[](https://collabnix.com/wp-content/uploads/2016/07/pico-app.png)

Now, we can list out the number of services running on our swarm cluster as shown below.

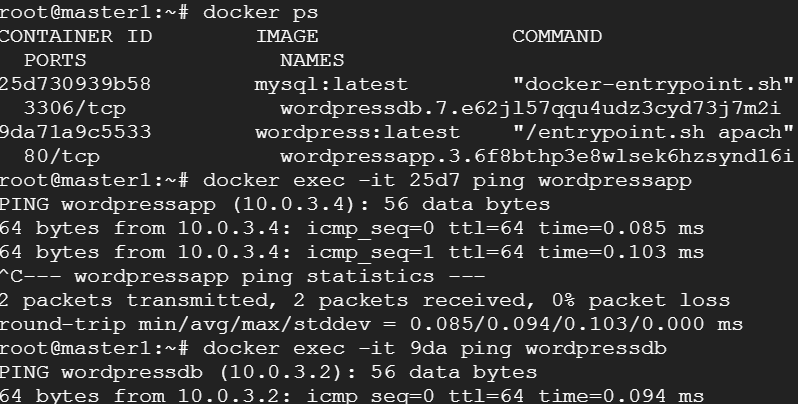
[](https://collabnix.com/wp-content/uploads/2016/07/pico-2.png)

I have scaled out the number of wordpressapp and wordpressdb just for demonstration purpose.

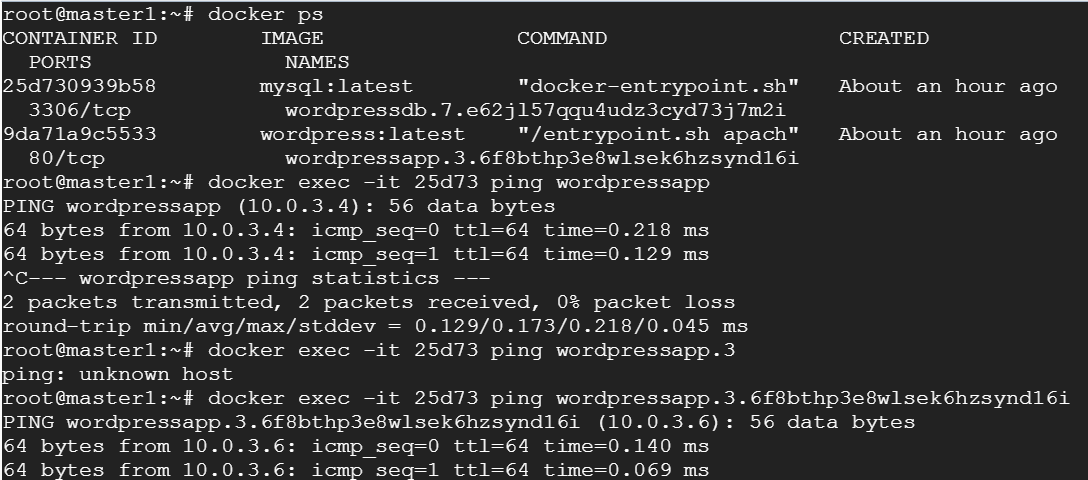
Let’s consider my master node where I have two of the containers running as shown below:

[](https://collabnix.com/wp-content/uploads/2016/07/Pico-1.png)

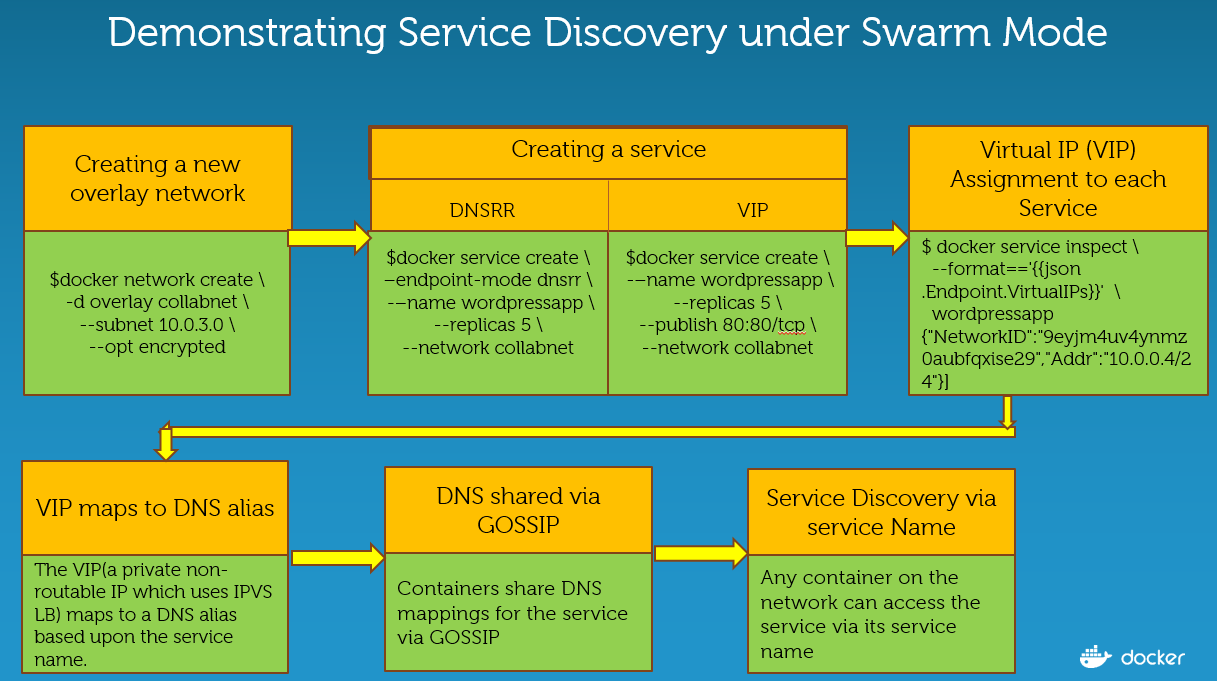
I can reach out one service(wordpressapp) from another service(wordpressapp) through just service-name as shown below:

[](https://collabnix.com/wp-content/uploads/2016/07/pico-last.png)

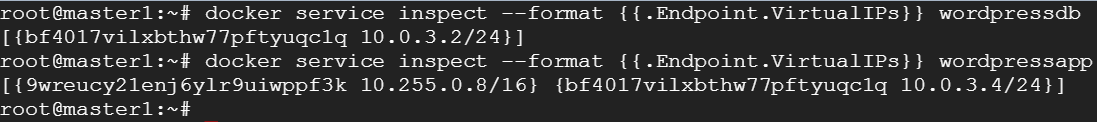
Also, I can reach out to particular container by its name from other container running different service but on the same network. As shown below, I can reach out to wordpressapp.3.6f8bthp container via wordpressdb.7.e62jl57qqu running wordpressdb.

[](https://collabnix.com/wp-content/uploads/2016/07/pico-tasktoo.png)

The below picture depicts the Service Discovery in a nutshell:

**[](https://collabnix.com/wp-content/uploads/2016/07/Pic23.png)**

Every service has Virtual IP(VIP) associated which can be derived as shown below:

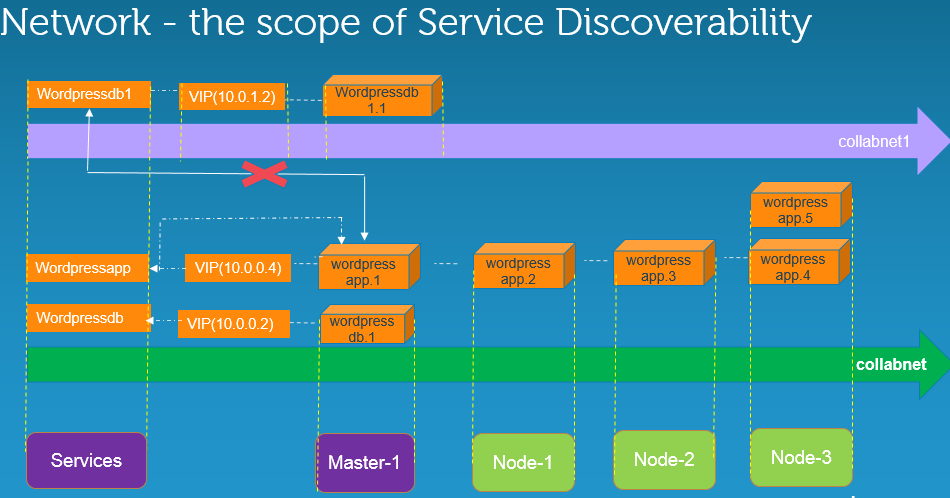
[](https://collabnix.com/wp-content/uploads/2016/07/pic-list.png)

As shown above, each service has an IP address and this IP address maps to multiple container IP address associated with that service. It is important to note that service IP associated with a service does not change even though containers associated with the service dies/ restarts.

Few important points to remember:

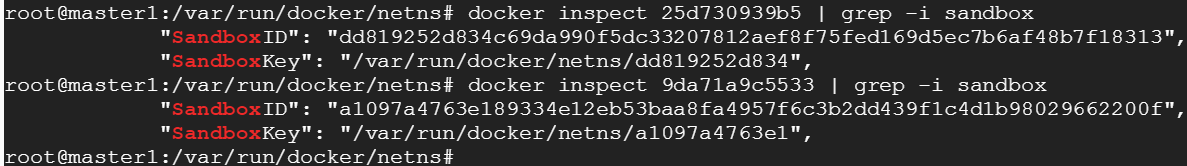
* VIP based services use Linux IPVS load balancing to route to the backend containers. This works only for TCP/UDP protocols. When you use DNS-RR mode services don’t have a VIP allocated. Instead service names resolves to one of the backend container IPs randomly.
* Ping not working for VIP is as designed. Technically, IPVS is a TCP/UDP load-balancer, while ping uses ICMP and hence IPVS is not going to load-balance the ping request.
* For VIP based services the reason ping works on the local node is because the VIP is added a 2nd IP address on the overlay network interface
* You can any of the tools like  dig, nslookup or wget -O- <service name> to demonstrate the service discovery functionality

Below picture depicts that the network is the scope of service discoverability which means that when you have a service running on one network , it is scoped to that network and won’t be able to reach out to different service running on different network(unless it is part of that network).

[](https://collabnix.com/wp-content/uploads/2016/07/SD.png)

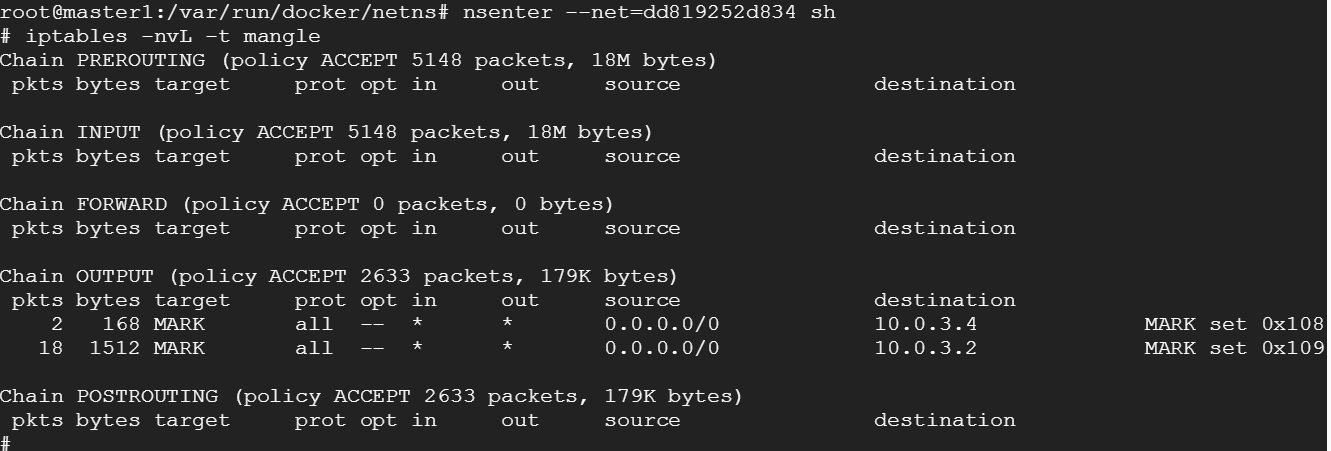
Let’s dig little further introducing Load-balancing aspect too. To see what is basically enabling the load-balancing functionality, we can go into sandbox of each containers and see how it has been resolved.

Let’s pick up the two containers running on the master node. We can see the sandbox running through the following command:

[](https://collabnix.com/wp-content/uploads/2016/07/pico-namespace.png)

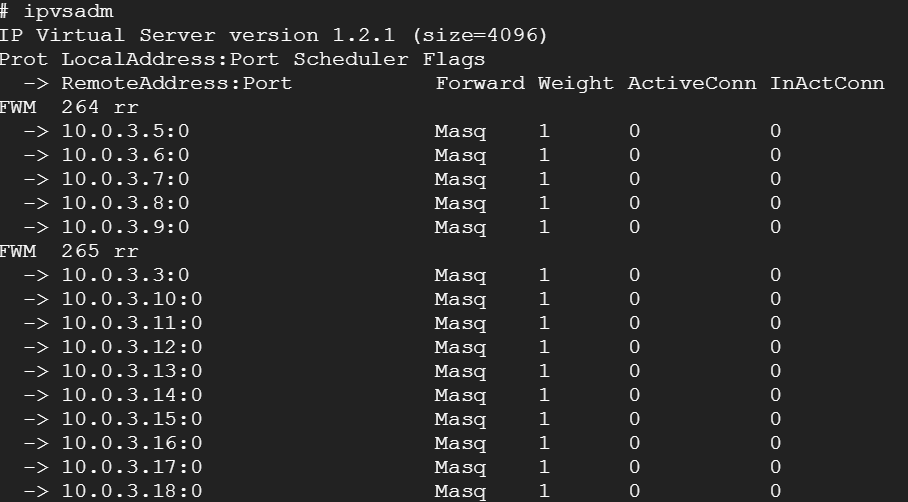
Under /var/run/docker/netns, you will find various namespaces. The namespaces marked with x-{id} represents network namespace managed by the overlay network driver for its operation (such as creating a bridge, terminating vxlan tunnel, etc…). They don’t represent the container network namespace. Since it is managed by the driver, it is not recommended to manipulate anything within this namespace. But if you are curious on the deep dive, then you can use the “nsenter” tool to understand more about this internal namespace.

We can enter into sandbox through the nsenter utility:

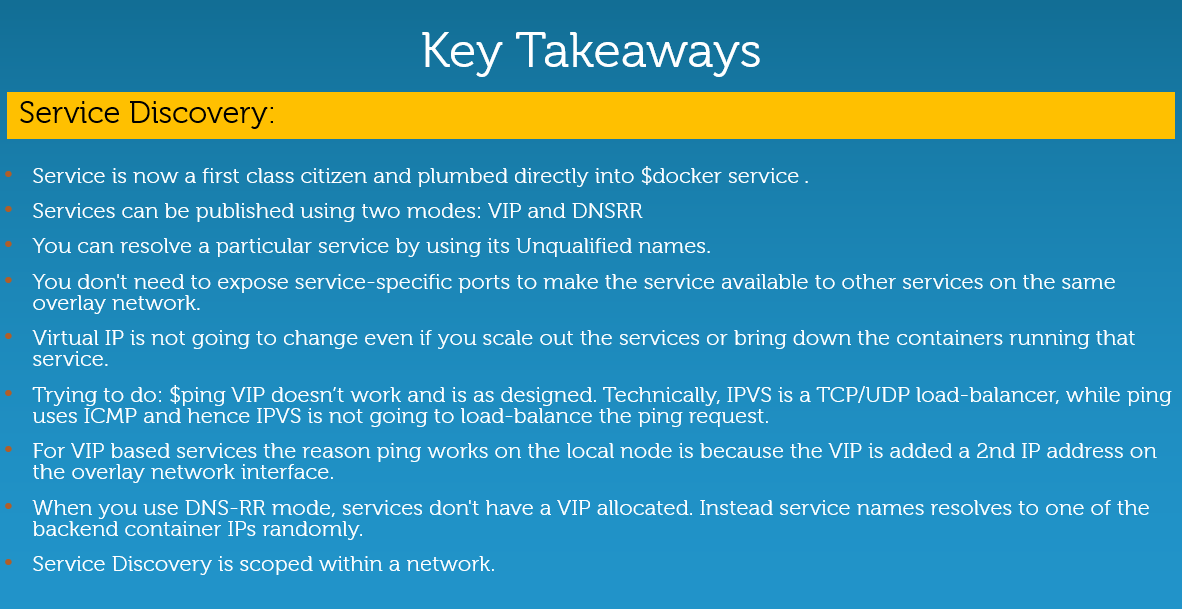
[](https://collabnix.com/wp-content/uploads/2016/07/pico-mangle.png)

In case you faced an error stating “nsenter: reassociate to namespace ‘ns/net’ failed: Invalid argument”, I suggest to look at [this](http://tinyurl.com/gu5rsw9) workaround.

10.0.3.4 service IP is marked 0x108 using iptables OUTPUT chain. ipvs uses this marking and load balances it to containers 10.0.3.5 and 10.0.3.6 as shown below:

[](https://collabnix.com/wp-content/uploads/2016/07/ipvs.png)

Here are key takeaways from this entire post:

[](https://collabnix.com/wp-content/uploads/2016/07/Pic34.png)