DOCKER NETWORKING OVERVIEW

(As of Docker 17.06 version)

Presenter's Name: Sreenivas Makam
Presented At: Cisco Systems
Presentation Date: July 5th, 2017

About me

- Senior Engineering Manager at Cisco Systems Data Center group
- Author of “Mastering CoreOS” https://www.packtpub.com/networking-and-servers/mastering-coreos/)

- Docker Captain(https://www.docker.com/community/docker-captains)

- Blog: https://sreeninet.wordpress.com/
- Projects: https://github.com/smakam
- LinkedIn: https://in.linkedin.com/in/sreenivasmakam
- Twitter: @srmakam

Terminology

- Unmanaged containers
  - No orchestration
  - Created using “docker run”
- Managed services
  - Orchestration using Swarm
  - Created using “docker service create”
- Legacy Swarm refers to pre Docker 1.12 Swarm mode
- Swarm refers to post Docker 1.12 Swarm mode

Note:

- All examples in this slide deck use Docker version 17.06 and below.
- Primary focus is on Docker Linux Networking

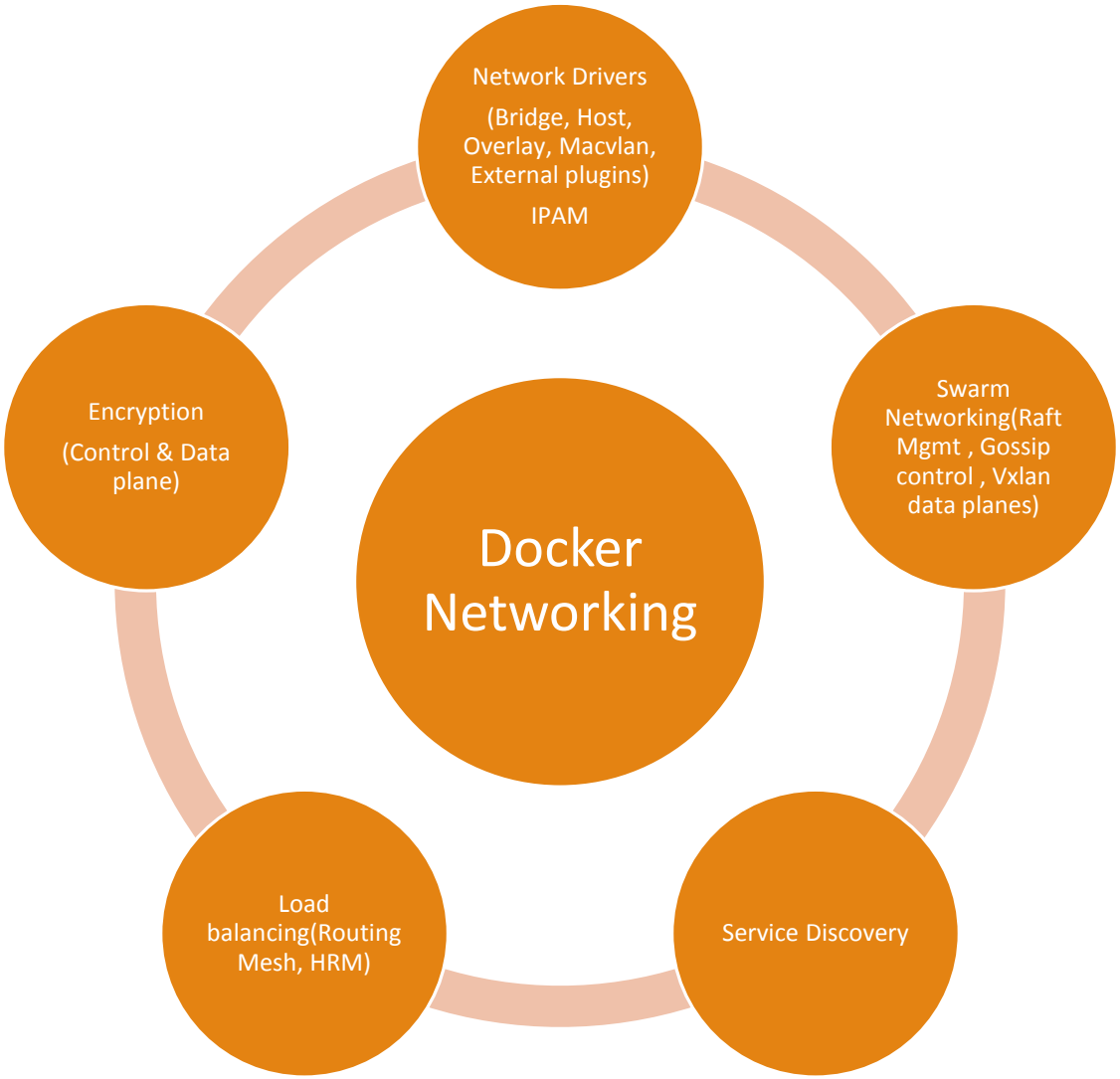
Why we need Container Networking?

- Containers need to talk to external world.
- Reach Containers from external world to use the service that Containers provides.
- Allows Containers to talk to host machine.
- Inter-container connectivity in same host and across hosts.
- Discover services provided by containers automatically.
- Load balance traffic between different containers in a service
- Provide secure multi-tenant services

Compare Container Networking with VM Networking

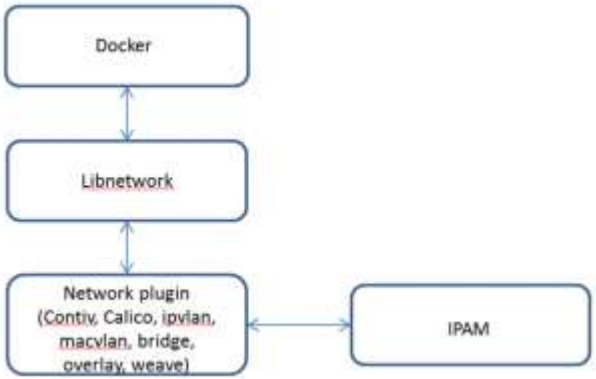
Feature	Container	VM
Isolation	Network isolation achieved using Network namespace.	Separate networking stack per VM
Service	Typically, Services gets separate IP and maps to multiple containers	Multiple services runs in a single VM
Service Discovery and Load balancing	Microservices done as Containers puts more emphasis on integrated Service discovery	Service Discovery and Load balancing typically done outside
Scale	As Container scale on a single host can run to hundreds, host networking has to be very scalable.	Host networking scale needs are not as high
Implementation	Docker Engine and Linux bridge	Hypervisor and Linux/OVS bridge

Docker Networking components



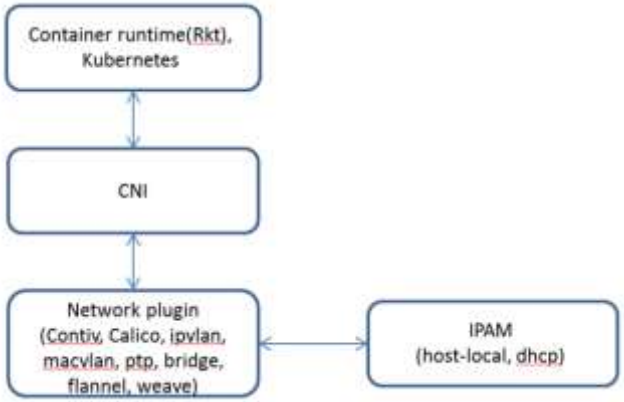
CNI and CNM – Standards for Container Networking

CNM



- Project started by Docker.
- Keep networking as a library separate from the Container runtime.
- Networking implementation will be done as a plugin implemented by drivers.
- IP address assignment for the Containers is done using local IPAM drivers and plugins.
- Supported local drivers are bridge, overlay, macvlan, ipvlan. Supported remote drivers are Weave, Calico, Contiv etc.

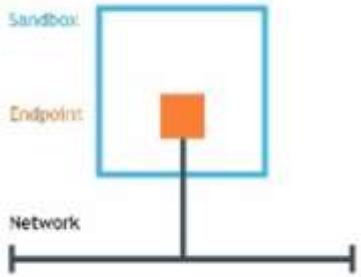
CNI



- Project started by CoreOS. Used by Cloudfoundry, Mesos and Kubernetes.
- The CNI interface calls the API of the CNI plugin to set up Container networking.
- The CNI plugin calls the IPAM plugin to set up the IP address for the container.
- Available CNI plugins are Bridge, macvlan, ipvlan, and ptp. Available IPAM plugins are host-local and DHCP.
- External CNI plugins examples – Flannel, Weave, Contiv etc

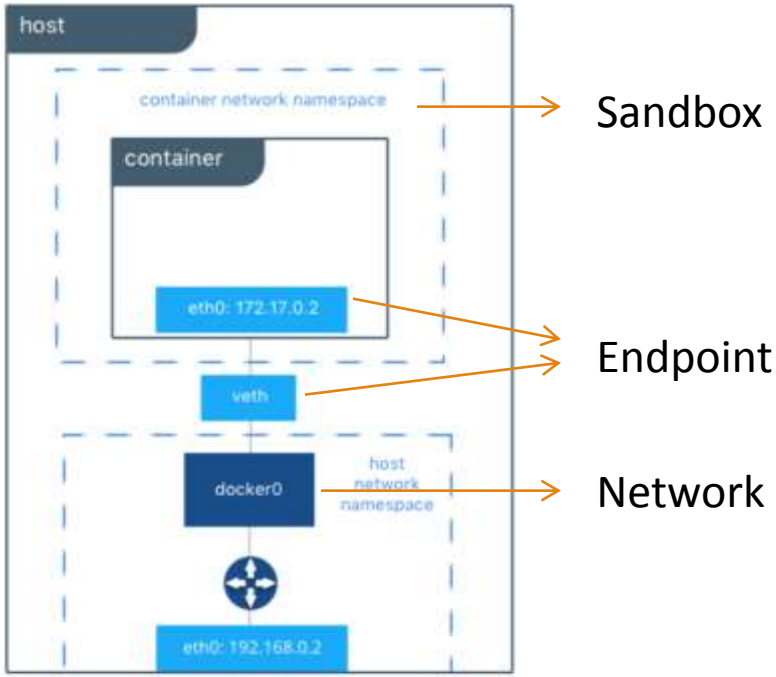
CNM and Libnetwork

CNM constructs



Picture from Docker white paper

CNM usage in Docker



Sandbox — A Sandbox contains the configuration of a container's network stack. In Docker example, Container network namespace is the equivalent of Sandbox.
Endpoint — An Endpoint joins a Sandbox to a Network. Eth0 and veth are the endpoints in above example.
Network — Multiple endpoints share a network. In other words, only endpoints located in same network can talk to each other. In above example, docker0 is the bridge network.

Compare Docker Network driver types

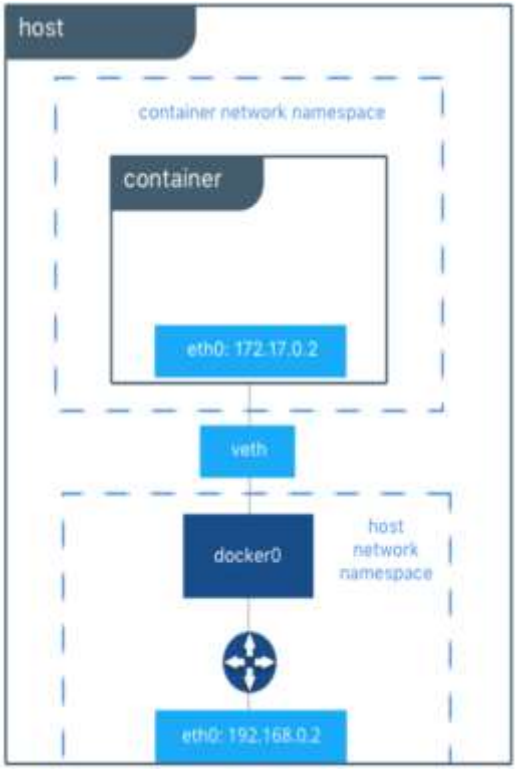
Driver/ Features	Bridge	User defined bridge	Host	Overlay	Macvlan/ipvlan
Connectivity	Same host	Same host	Same host	Multi-host	Multi-host
Service Discovery and DNS	Using “links”. DNS using /etc/hosts	Done using DNS server in Docker engine	Done using DNS server in Docker engine	Done using DNS server in Docker engine	Done using DNS server in Docker engine
External connectivity	NAT	NAT	Use Host gateway	No external connectivity	Uses underlay gateway
Namespace	Separate	Separate	Same as host	Separate	Separate
Swarm mode [1]	No support yet	No support yet	No support yet	Supported	No support yet
Encapsulation	No double encap	No double encap	No double encap	Double encap using Vxlan	No double encap
Application	North, South external access	North, South external access	Need full networking control, isolation not needed	Container connectivity across hosts	Containers needing direct underlay networking

1 – 17.06 introduces Swarm mode for all network drivers

Bridge Driver

- Used by “docker0” bridge and user-defined bridges.
- “docker0” bridge is created by default. User has the choice to change “docker0” bridge options by specifying them in Docker daemon config.
- User-defined bridges can be created using “docker network create” with “bridge” driver.
- Used for connectivity between containers in same host and for external North<->South connectivity.
- Services running inside Containers can be exposed by NAT/port forwarding.
- External access is provided by masquerading.

docker run -d -p 8080:80 --network bridge --name web nginx

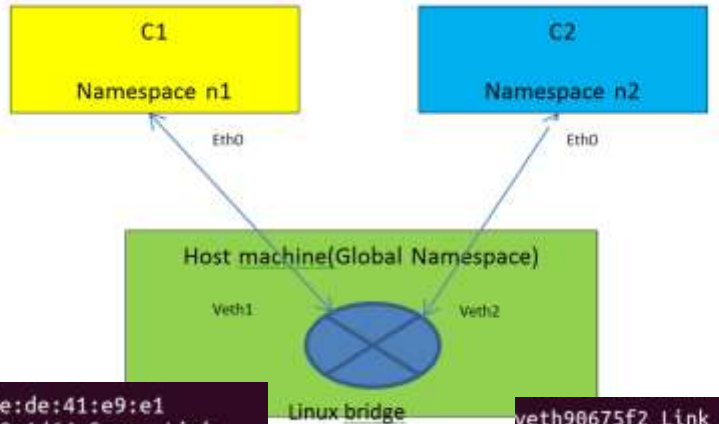


Picture from Docker white paper

Docker Container Networking – Bridge driver

```
root@bd212dd98815:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:2a:02
          inet addr:172.17.42.2  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::42:acff:fe11:2a02/64 Scope:Link
          UP BROADCAST RUNNING  MTU:1500  Metric:1
          RX packets:6 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:508 (508.0 B)  TX bytes:598 (598.0 B)
```

```
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:2a:03
          inet addr:172.17.42.3  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::42:acff:fe11:2a03/64 Scope:Link
          UP BROADCAST RUNNING  MTU:1500  Metric:1
          RX packets:3 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:258 (258.0 B)  TX bytes:348 (348.0 B)
```




```
veth3f5b988 Link encap:Ethernet  HWaddr 8e:2e:de:41:e9:e1
          inet6 addr: fe80::8c2e:deff:fe41:e9e1/64 Scope:Link
          UP BROADCAST RUNNING  MTU:1500  Metric:1
          RX packets:9 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:738 (738.0 B)  TX bytes:648 (648.0 B)
```

```
veth90675f2 Link encap:Ethernet  HWaddr 9a:bc:0a:4a:fd:b4
          inet6 addr: fe80::98bc:aff:fe4a:fdb4/64 Scope:Link
          UP BROADCAST RUNNING  MTU:1500  Metric:1
          RX packets:9 errors:0 dropped:0 overruns:0 frame:0
          TX packets:17 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:738 (738.0 B)  TX bytes:1386 (1.3 KB)
```

```
sreeni@ubuntu:~$ sudo brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.8e2ede41e9e1       no              veth3f5b988
                                                        veth90675f2

sreeni@ubuntu:~$
```

```
docker0   Link encap:Ethernet  HWaddr 26:90:8f:3a:1f:d5
          inet addr:172.17.42.1  Bcast:0.0.0.0  Mask:255.255.255.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

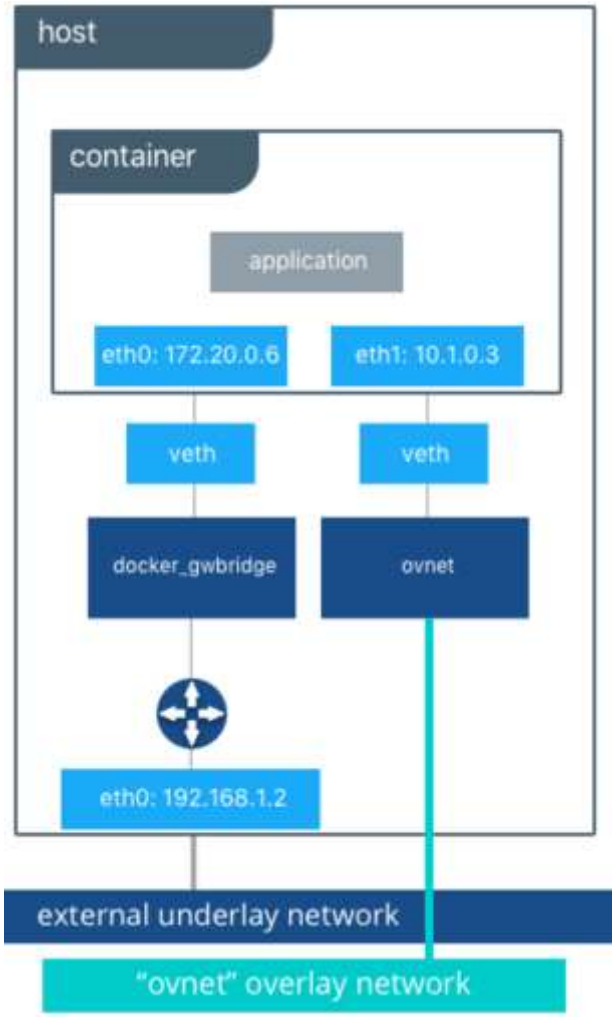
Overlay Driver

- Used for container connectivity across hosts.
- Before Docker 1.12 version, Overlay driver needed external KV store. After 1.12, external KV store is not needed.
- Containers connected to overlay network also get connected to “docker_bwbridge” for external access.
- Vxlan is used for encapsulation.

docker network create --driver overlay onet

docker run -ti --name client --network onet smakam/myubuntu:v4 bash

docker run -d --name web --network onet nginx



Picture from Docker white paper

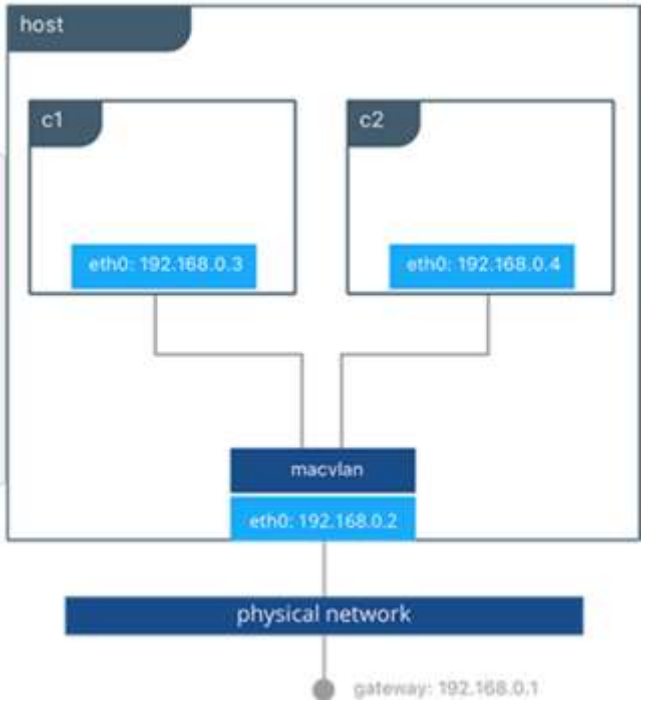
Macvlan driver

- Macvlan allows a single physical interface to have multiple mac and ip addresses using macvlan sub-Interfaces
- Macvlan driver allows for containers to directly connect to underlay network. Works well for connectivity to legacy applications.
- Provides connectivity within a single host as well as across hosts.

docker network create -d macvlan --subnet=192.168.0.0/16 --ip-range=192.168.2.0/24 -o macvlan_mode=bridge -o parent=eth1 macvlan1

docker run -d --name web1 --network macvlan1 nginx

docker run -d --name web2 --network macvlan1 nginx



Picture from Docker white paper

Docker Network plugins

- Extends functionality of Docker networking by using plugins to implement networking control and data plane.
- Docker provides batteries included approach where user has a choice of using Docker network drivers or plugins provided by other vendors.
- Using plugins, switch vendors can get Docker integrated with their custom switches having special features or with custom features like policy based networking.
- Docker network plugins follow CNM(libnetwork) model.
- Docker 1.13.1+ included support for global scoped network plugins that allows network plugins to work in Swarm mode.
- Following network plugins are available now:
  - Contiv – Network plugin from Cisco. Supports L2 and L3 physical topology. Integrates with Cisco ACI. Also provides policy based networking.
  - Calico – Follows Layer3 rather than overlay approach. Uses policy based networking.
  - Weave – Follows Overlay approach
  - Kuryr – Uses openstack Neutron to provide container networking

IP Address management

- Docker does the IP address management by providing subnets for networks and IP addresses for containers.
- For default “bridge” network, custom subnet can be specified in Docker daemon options.
- Users can specify their own subnet while creating networks and specify IP when creating containers. Following example illustrates this.

  docker network create --subnet=172.19.0.0/16 mynet

  docker run --ip 172.19.0.22 -it --network mynet smakam/myubuntu:v4 bash

- Using remote IPAM plugin, IP addresses can be managed by external application instead by Docker.
- Remote IPAM plugin can be specified using “--ipam-driver” option while creating Docker network. Infoblox is an example of external Docker IPAM plugin.
- Docker also supports assignment of IPV6 addresses for Containers.

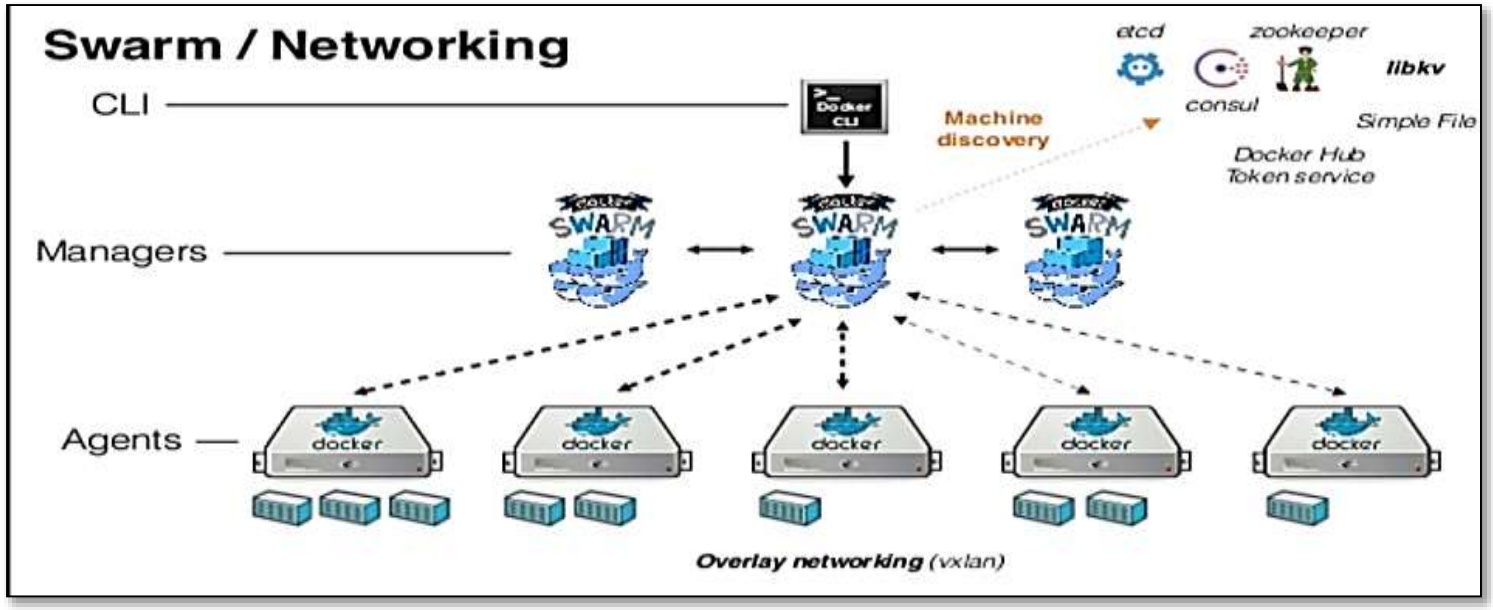
Default Networks created by Docker

```
sreeni@ubuntu:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
7acb2b0d26ce        bridge              bridge              local
aa0c5ef9b316        docker_gwbridge     bridge              local
71525ff22fb2        host                host                local
6zi2aihqnd4s        ingress             overlay             swarm
7d71c04fe4e1        none                null                local
```

- “bridge” is the default Bridge network
- “docker_gwbridge” is used by multi-host networks to connect to outside world
- “host” network is used for having containers in host namespace
- “ingress” network is used for routing mesh
- “none” network is used when Containers don’t need any networking
- “Scope” signifies if the network is local to host or across the Swarm cluster

Legacy Swarm mode

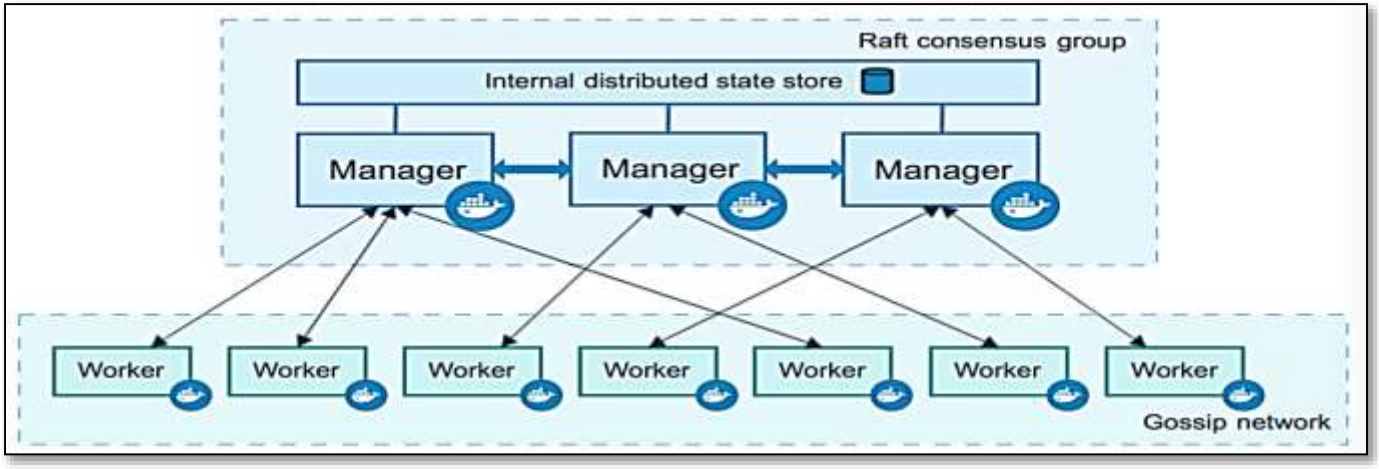
- In this mode, Swarm is not integrated with Docker engine and it runs as a separate container.
- Needs separate KV store like Consul, etcd.
- Supported in Docker prior to version 1.12
- This is a legacy mode that is deprecated currently



https://www.slideshare.net/Docker/docker-online-meetup-28-productionready-docker-swarm

Swarm Mode

- Orchestration supported by Docker from 1.12 version.
- Using Raft protocol, Managers maintain state of Swarm nodes as well as services running on them.
- Gossip protocol is used by workers to establish control plane between them. Only workers in same network exchange state associated with that network.
- Control plane is encrypted by default. Data plane can be optionally encrypted using "--opt encrypted" when creating network.
- No separate KV store is needed with Swarm mode.
- Prior to Docker 17.06, Swarm mode was supported only with Overlay driver. Post 17.06, all network drivers are supported with Swarm mode.



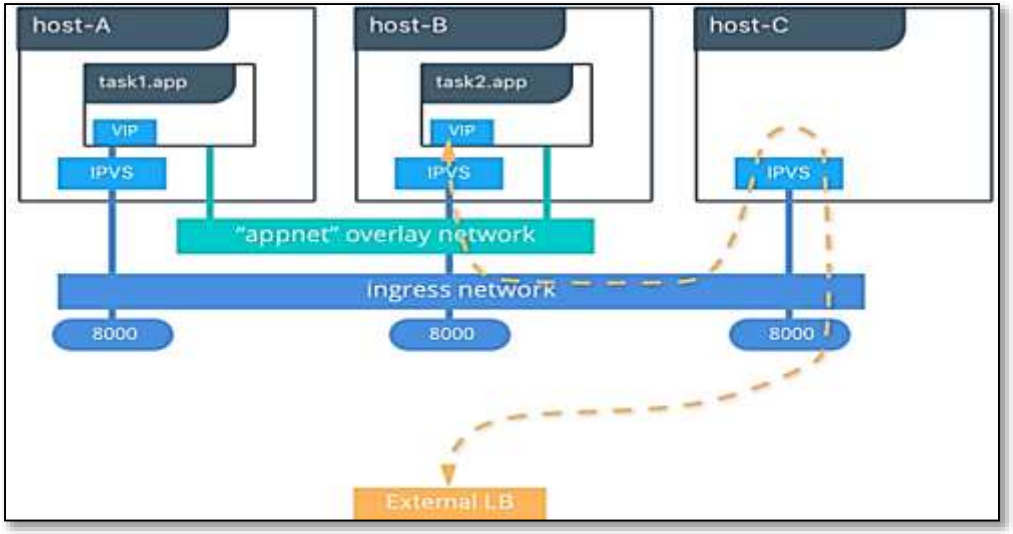
https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/#manager-nodes

Service Discovery

- Service discovery is provided by DNS server available in Docker engine.
- For unmanaged containers, container name resolves to container IP. Alias names can be also be used.
- For services using service IP(endpoint mode=vip), service name resolves to service IP which in turn forwards the request to containers. In this case, ipvs based L4 load balancing is done.
- For services using direct DNS(endpoint mode=dnsrr), service name directly resolves to container IP. In this case, DNS round robin load balancing is done.
- Service Discovery is network scoped. Only containers in same network can discover each other.

Load balancing

- For unmanaged containers, load balancing is done using simple round robin load balancing. Using aliases, a single alias can load balance to multiple unmanaged containers .
- Docker takes care of load balancing internal services to the containers associated with the services.
- For services using service IP(endpoint mode=vip), ipvs and iptables are used to load balance. This provides L4 based load balancing. Ipvs is Linux kernel load balancing feature.
- For services using direct DNS(endpoint mode=dnsrr), DNS round robin balancing is used.
- For services exposed externally, Docker uses routing mesh to expose the service on all Swarm nodes. Routing mesh uses “ingress” network to connect all nodes.
- For HTTP based load balancing, HRM(HTTP Routing mesh) can be used. This is supported only with Docker EE.



Picture from Docker white paper

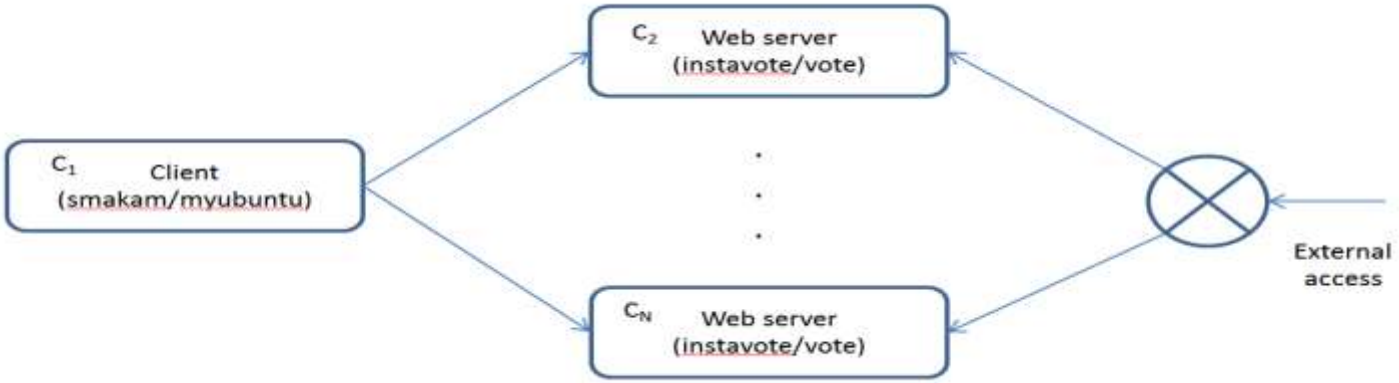
Swarm Networking - Sample application detail

- The application will be deployed in 2 node Swarm cluster.
- "client" service has 1 client container task. "vote" service has multiple vote container tasks. Client service is used to access multi-container voting service. This application is deployed in a multi-node Swarm cluster.
- "vote" services can be accessed from "client" service as well as from outside the swarm cluster.

docker network create -d overlay overlay1

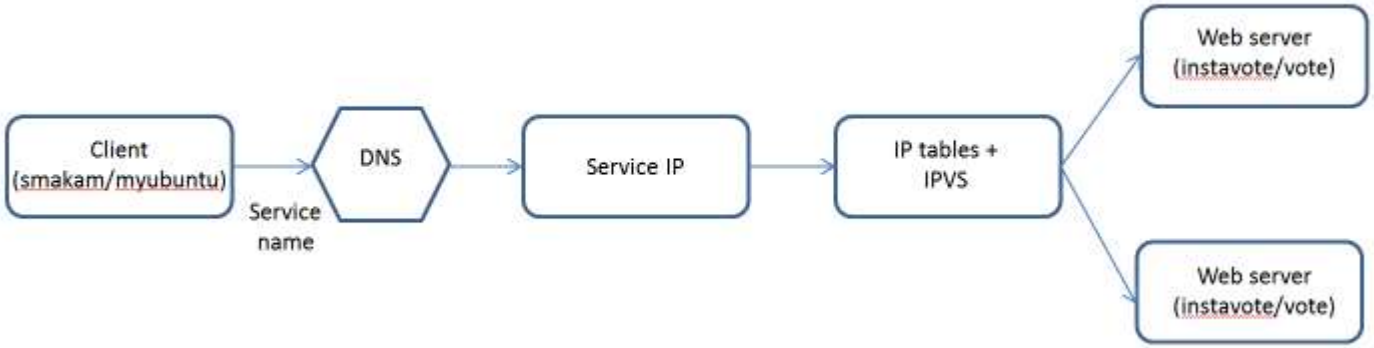
docker service create --replicas 1 --name client --network overlay1 smakam/myubuntu:v4 sleep infinity

docker service create --name vote --network overlay1 --mode replicated --replicas 2 --publish mode=ingress,target=80,published=8080 instavote/vote

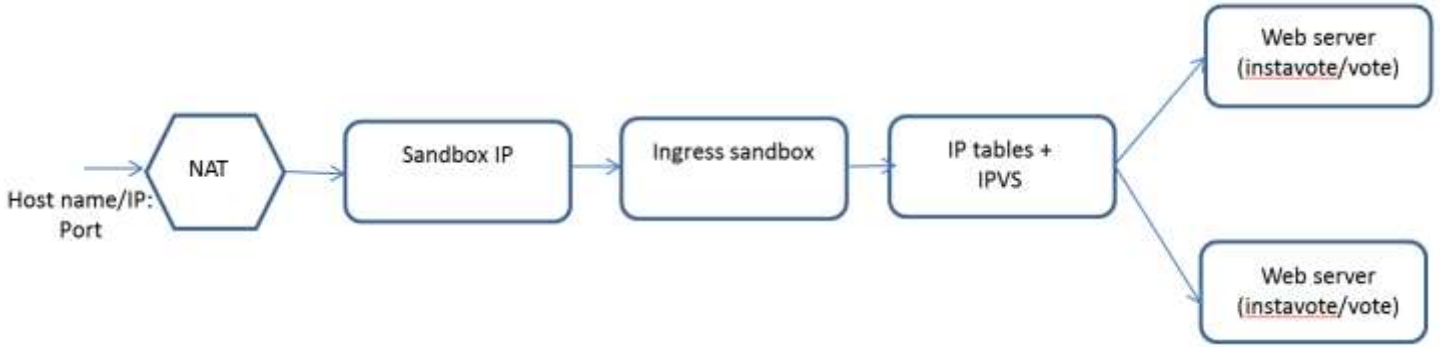


Swarm Networking - Application access flow

“Client” service accessing “vote” service using “overlay” network

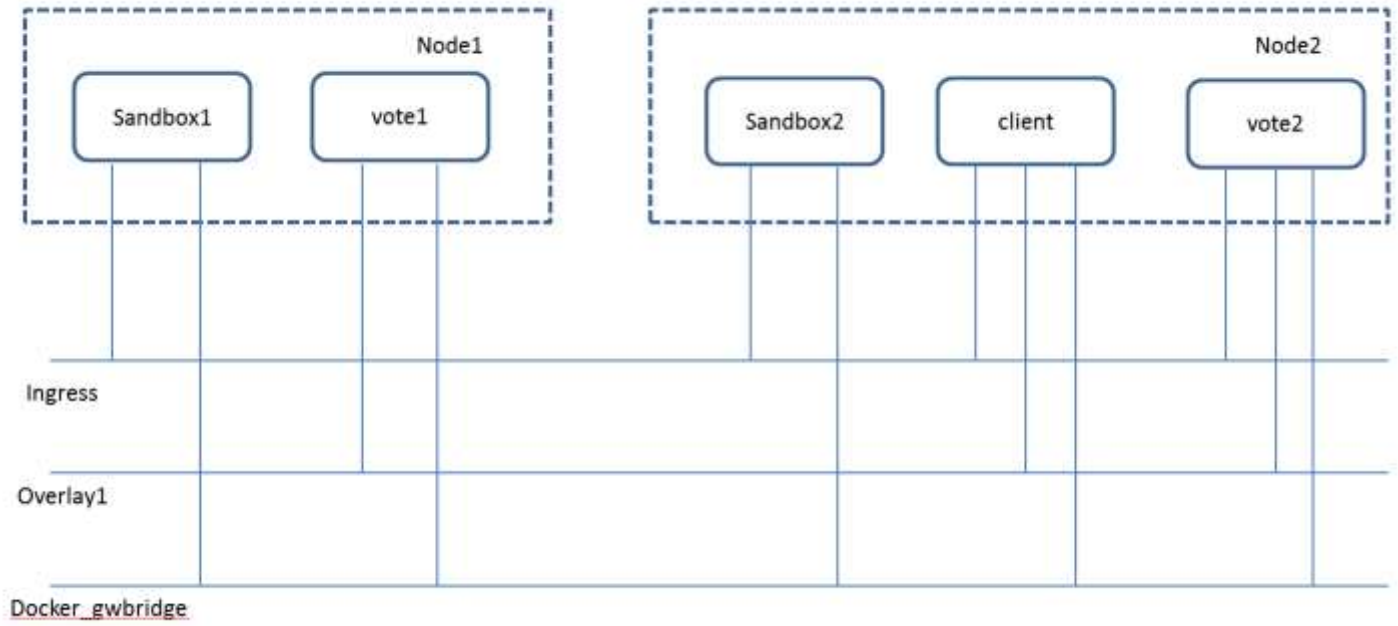


Accessing “vote” service using “ingress” network externally



Swarm Application - Networking detail

- Sandboxes and “vote” containers are part of “ingress” network and it helps in routing mesh.
- “client” and “vote” containers are part of “overlay1” network and it helps in service connectivity.
- All containers are part of the default “docker_gwbridge” network. This helps for external access when services gets exposed using publish mode “host”



Compare Docker and Kubernetes Networking

Feature	Docker	Kubernetes
Abstraction	Container	Pod
Standard	CNM	CNI
Service discovery	Embedded DNS	Kube-dns
Internal load balancing	Iptables and ipvs	Iptables and Kube-proxy
External load balancing	Routing mesh	Nodeport
External plugins	Weave, Calico, Contiv	Flannel, Weave, Calico, Contiv

Note: Implementation differences are not captured

Docker Network debug commands

- Basic Swarm debugging:

Docker node ls

- Service and Container debugging:

Docker service logs <service name/id>

Docker service inspect <service name/id>

Docker container logs <container name/id>

Docker container inspect <container name/id>

- Network debugging:

Docker network inspect <network name/id>

Use "-v" option for verbose output

Troubleshooting using debug container

- All Linux networking tools are packaged inside “nicolaka/netshoot”(https://github.com/nicolaka/netshoot) container. This can be used for debugging.
- Using this debug container avoids installation of any debug tools inside the container or host.
- Linux networking tools like tcpdump, netstat can be accessed from container namespace or host namespace.

Capture port 80 packets in the Container:

docker run -ti --net container:<containerid> nicolaka/netshoot

tcpdump –i eth0 –n port 80

Capture vxlan packets in the host:

docker run -ti --net host nicolaka/netshoot

tcpdump –i eth1 –n port 4789

- Debug container can also be used to get inside container namespace, network namespace and do debugging. Inside the namespace, we can run commands like “ifconfig”, “ip route”, “brctl show” to debug further.

Starting nsenter using debug container:

docker run -it --rm -v /var/run/docker/netns:/var/run/docker/netns --privileged=true nicolaka/netshoot

Getting inside container or network namespace:

nsenter –net /var/run/docker/netns/<networkid> sh

References

- White paper on Docker networking
- HRM and UCP White paper
- Docker Networking Dockercon 2017 presentation
- Docker blogs by me
- Docker Networking – common issues and troubleshooting techniques

DEMO