DOCKER NETWORKING

COMMON ISSUES AND TROUBLESHOOTING TECHNIQUES

Presenter's Name: Sreenivas Makam

Presented At: Docker Meetup, Bangalore

Presentation Date: July 1st, 2017 (Last update: Aug 2nd)

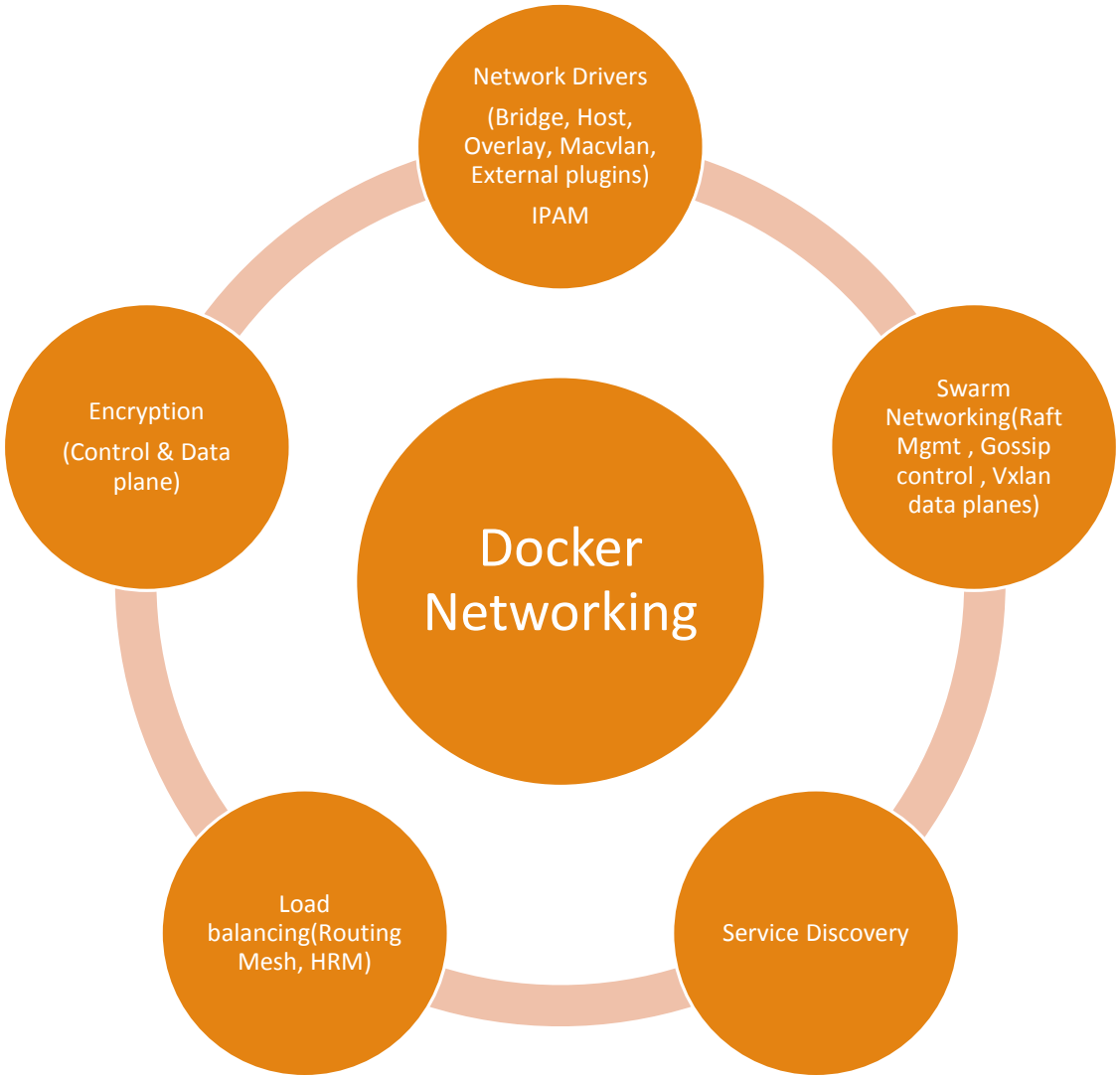
About me

- 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

Docker Networking components



Terminology

- Unmanaged containers
  - No orchestration
  - Created using “docker run”
- Managed services
  - Orchestration using Swarm
  - Created using “docker service create”
- 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

Version of this document:

V0 – July 1, 2017

V1 – Aug 5, 2017

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

Docker Daemon

Accessing Docker daemon remotely and securely

- Remote Docker engine can be accessed by setting “DOCKER_HOST” variable. Port 2375 has to be opened up in the host.

export DOCKER_HOST=<remote IP>:2375

- To access remote Docker daemon securely, https remote connection to port 2376 can be used. This needs certificate and SSL key setup.
- If there is a ssh connection available to remote host, we can tunnel the Docker 2375 connection over ssh. This option avoids setting up 1 more secure keypair and can be used for development purpose.

ssh -i <key> -fNL 2375:localhost:2375 username@<ip>
export DOCKER_HOST=:2375

- Above tunneling approach can also be used to tunnel Docker connection across Jump hosts if the remote Docker daemon has to be accessed through the jump host.
- When using public cloud and doing remote Docker access, it is needed to open up firewall ports 2375 and 2376 based on the need.

Using Docker behind Corporate firewall

- Docker specific proxy settings are needed in addition to setting up usual proxy for allowing external access.
- Below steps will allow Docker daemon to use “proxy address” to access Docker.io to download Docker images

/etc/default/docker:
Add line: export http_proxy="<proxy address>"
Restart Docker daemon

- For building and running Containers, following environment variables needs to be added to “Dockerfile”. This allows package manager to download software during the process of building Containers. The same proxy environment is also used inside containers.

ENV http_proxy <proxy address>
ENV https_proxy <proxy address>

- If we dont want to change Dockerfile, we can use following as arguments to “docker build” when building Docker image.

--build-arg http_proxy <proxy address>

Using Docker behind Corporate firewall(Contd)

- There are scenarios where proxy intercepts and does certificate signing. In this case, following error will be seen when fetching Docker images.

x509: certificate signed by unknown authority

- The error happens when Docker does not trust the proxy signature. To solve this problem, proxy certificate needs to be added to list of certificates that Docker trusts. The actual procedure varies based on Linux distribution type.
- For Debian/Ubuntu flavors, following procedure can be used to update trust:

sudo cp yourcert.crt /usr/local/share/ca-certificates/
sudo update-ca-certificates
sudo service docker restart

- Following links has more details on this issue:
  - *http://www.devops-insight.com/2014/11/using-docker-with-a-proxy.html*
  - *https://stackoverflow.com/questions/20267339/docker-behind-proxy-that-changes-ssl-certificate*

Docker Networking with VirtualBox

- Virtualbox uses a boot2docker VM to run Container inside boot2docker VM.
- When services are exposed to host machine from containers, they are not accessible from host machine since there is a boot2docker VM between host machine and container.
- To access exposed ports from host machine, configure port forwarding under NAT in Virtualbox and forward exposed ports to host.

Docker run –d –p 8080:80 nginx -> Container running inside boot2docker VM

To access this web server from host machine, configure NAT port forwarding to map 8080 port from VM to 8080 port in host machine.

- There could be cases where Containers need to access some service running directly in the host machine. To achieve this, we can either use NAT gateway or host-only adapter gateway based on the adapter type being used in Virtualbox. I used the following option to “docker run” to map “localhost” to host machine IP.

--add-host="localhost:10.0.2.2“ (for NAT)

--add-host="localhost:192.168.99.1“ (for host-only)

After above step, reference to “localhost” inside container will reference to host machine. Actual gateway address can vary in your environment.

- For macvlan and ipvlan network, we need to use “host-only” network with promiscuous mode turned on to talk between containers running on 2 different Virtualbox VM. Parent interface of macvlan network should be the “host-only” interface. In this example, its “eth2”

docker network create -d macvlan --subnet=192.168.0.0/16 --ip-range=192.168.2.0/24 –gateway=192.168.99.1 -o macvlan_mode=bridge -o parent=eth2 macvlantest

Preventing iptable modification

- Docker adds iptable rules for doing masquerading and NAT.

$ sudo iptables -t nat --list
Chain POSTROUTING (policy ACCEPT)
target prot opt source destination
MASQUERADE all -- 172.17.0.0/16 anywhere
DNAT tcp -- anywhere anywhere tcp dpt:8085 to:172.17.0.2:80

- There might be cases where its needed to prevent Docker from modifying iptables. This could be because of security reasons or because user might be using a different firewall manager on top of iptables.

Disabling iptables:

Set "--iptables=false" in /etc/default/docker(for Ubuntu)
Restart Docker daemon

- If we disable iptable modification by Docker, we need to add the NAT rules by ourselves for external forwarding to work.

Cleanup unused networks

- To cleanup unused networks, following command can be used. This command will remove all networks which does not have an attached container.

Docker network prune

- Following command cleans up unused containers, volumes in addition to networks.

Docker system prune

Container networking

Restrict Container access to Internet

- For security reasons, it might be needed to prevent internet access from/to container.
- Network created with "--internal" flag does not allow internet access. This option can be used with any network driver.

docker network create --internal --driver bridge intbridge

- Internet access can be blocked for “bridge” driver by disabling masquerading.

docker network create -o "com.docker.network.bridge.enable_ip_masquerade"="false" my-network

- Inter-container connectivity for “bridge” driver can be prevented using icc flag.

docker network create -o "com.docker.network.bridge.enable_icc"="false" my-network

- Packet forwarding between containers and internet access can be prevented with following option at Docker daemon level. This only applies to default “docker0” bridge.

/etc/default/docker:
--ip-forward=false
Restart Docker daemon

Connect Container/Service to multiple networks

- There are scenarios where a Container or Service needs to connect to multiple networks. An example is an application container needing to talk to both frontend and backend container.
- When starting a container, it is not possible to specify multiple networks in the same CLI. We need to do this as 2 steps.
- The reasoning for 2 steps is that there are other options that are specific to a single network and it becomes difficult to group in a single CLI.
- Following example will connect “web” container to “bridge” and “testbr” bridge networks.

docker network create –driver bridge testbr
docker run -d -p 8080:80 --network bridge --name web nginx
docker network connect testbr web

- For services, we can similar approach to connect to multiple networks.
- Following example will connect “web” service to “oneta” and “onetb” overlay networks.

docker network create --attachable --driver overlay oneta
docker network create --attachable --driver overlay onetb
docker service create --name web --network oneta --replicas 2 nginx
docker service update --network-add onetb web

Connecting containers – Don't use "--link"

- Connecting containers using "--link" option is not advisable. This option is referred to as legacy and might get deprecated anytime.
- Links have disadvantages like being uni-directional, not being able to dynamically replace linked containers, difficulty with linking multiple containers etc.
- Link based containers are supported only with default bridge which supports only primitive dns and service discovery. With user defined networks, link functionality is automatically supported when containers are in the same network.

old way:

docker run -d --name mysql -e MYSQL_ROOT_PASSWORD="mysql" mysql

docker run --name mywordpress -p 8081:80 --link mysql:mysql -d wordpress

new way:

docker network create backend

docker run -d --name mysql --network backend -e MYSQL_ROOT_PASSWORD="mysql" mysql

docker run --name mywordpress -p 8081:80 --network backend -e WORDPRESS_DB_PASSWORD="mysql" -d wordpress

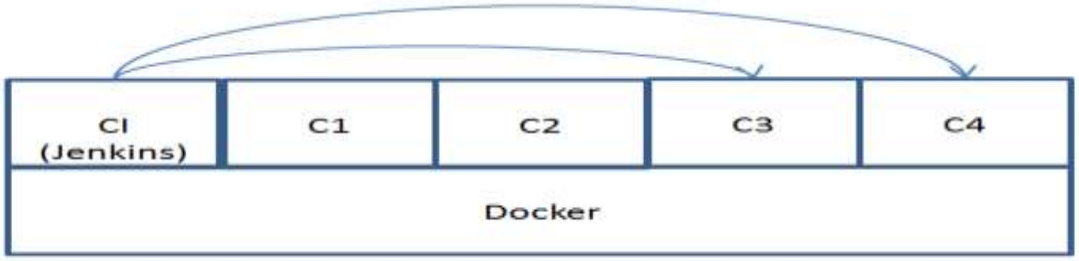
- One feature that user-defined networks do not support that can be done with "--link" is sharing environmental variables between containers. However, other mechanisms such as volumes can be used to share environment variables between containers.

Spinning Container inside Container and talking to it

- There are use-cases where we need to spin containers inside container and be able to talk to it.
- This use case is not same as Docker in Docker(DinD). An example is Continuous integration(CI) use case where it is needed to build Containers from CI system like Jenkins which is running as Container.
- For this scenario, it is not needed to have Docker engine running within Jenkins Container. It is needed to have Docker client in Jenkins container and use Docker engine from host machine. This can be achieved by mounting "/var/run/docker.sock" from host machine.

docker run --rm --user root --name myjenkins -v /var/run/docker.sock:/var/run/docker.sock -p 8080:8080 -p 50000:50000 jenkins

- If we spin a new Container inside Jenkins Container, that will be in same bridge network of host machine and both containers can talk to each other directly over bridge network.



Container name vs Service name vs Host name

- Docker container hostname is not used for Service discovery. Container name or Service name is used for Service discovery.

docker run -d --name web --network frontend --hostname webserver -d nginx
docker run -ti --rm --name client --network frontend smakam/myubuntu:v4 bash

- From client container, we can access webserver by using “web” but not “webserver”. “web” is container name, “webserver” is hostname of container.
- For services, service name is used for Service discovery.

Container and Service alias names

- Using network scoped aliases, a single container or service can be referred by different names.
- When multiple containers have the same alias name, primitive load balancing can be achieved since Docker would round robin the alias access between multiple containers.
- In following example, wordpress container is referred as “wordfront” in “frontend” network and as “wordback” in “backend” network.

docker run -d --name mysql --network backend -e MYSQL_ROOT_PASSWORD="mysql" mysql

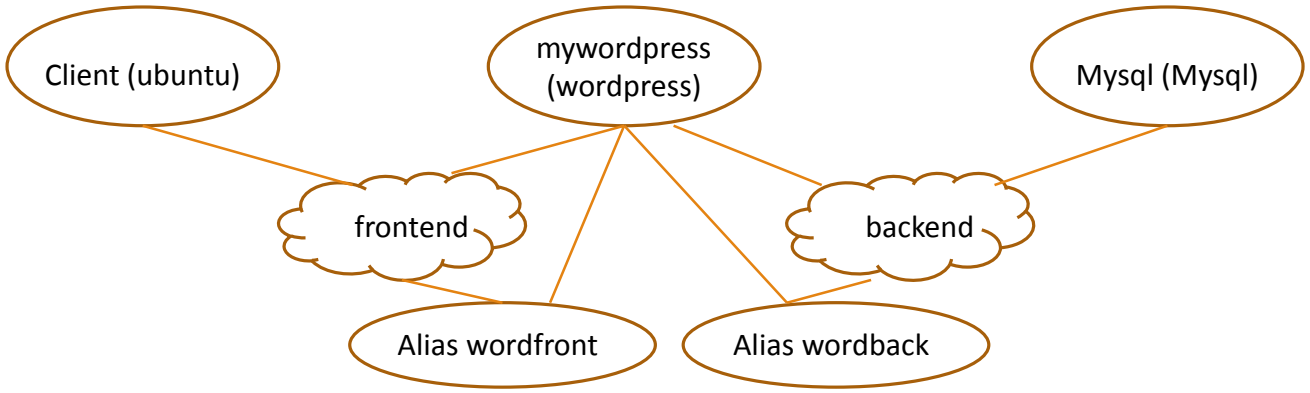
docker network create frontend

docker network create backend

docker run --name mywordpress -p 8081:80 --network frontend --network-alias wordfront -e WORDPRESS_DB_PASSWORD="mysql" -d wordpress

docker network connect --alias wordback backend mywordpress

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



Container and Service alias names(contd..)

- Service names can also be aliased.
- “docker service create” does not yet support “--network-alias” option. We can create network alias using Docker api or stack file.
- Following example creates 2 aliases for “wordpress” service similar to previous example using stack file. In this case, alias is created for service compared to container in previous example.

```
version: "3"

services:
  mysql:
    image: mysql:latest
    environment:
      - "MYSQL_ROOT_PASSWORD=mysql"
    networks:
      - backend

  wordpress:
    image: wordpress:latest
    environment:
      - "WORDPRESS_DB_PASSWORD=mysql"
    ports:
      - "8000:80"
    networks:
      frontend:
        aliases:
          - wordfront
      backend:
        aliases:
          - wordback

  client:
    image: smakam/myubuntu:v4
    command: [ping, docker.com]
    networks:
      - frontend

networks:
  frontend:
    driver: overlay
  backend:
    driver: overlay
```

Swarm/Service networking

Expose Swarm ports in the Cloud

- Docker Swarm uses following ports for communication between Swarm nodes:
  - TCP port 2377 for RAFT management plane
  - TCP and UDP port 7946 for gossip control plane
  - UDP port 4789 for VXLAN data plane
- When creating Swarm cluster in AWS using default VPC, we need to tie the instances to security group that opens up above ports.
- In Google cloud and Azure cloud, port based communication between nodes is not blocked as long as the nodes are in the same virtual private network. There is no need of separate firewall rules in this case.
- Opening up firewall ports(eg: with iptables) would be needed in non-cloud scenarios based on where Swarm gets deployed.

Connect Container to service running in local machine

- There are cases where Container needs to connect to a service running in local machine. Following are some examples.
  - Database running in local machine that is not containerized.
  - Service like Consul that can run as Container which is normally run as "--net=host"

Following are some possible options for this connectivity:

Containers running with bridge network:

- Run database in docker0 bridge IP and connect container to database port of docker0 bridge IP
- Run database in public IP and connect container to public IP and database port. Bridge networks do provide external connectivity.
- Mount mysql daemon socket(/var/run/mysqld/mysqld.sock) to the client container

Containers running with host network:

For containers running with "--host" option, we can directly access database using "localhost" This is because "localhost" inside container would refer to host machine since there is only 1 network namespace.

Connect managed services to unmanaged containers

- Following are some use cases where we need connectivity between managed services and unmanaged containers.
  - Running some privileged containers as unmanaged and connecting to services.
  - Quick debugging of services using unmanaged container
  - Connecting Swarm mode service with legacy unmanaged containers already running in production.

- By using “attachable” option when we create overlay network, we can attach unmanaged container to the same network.

docker network create --attachable --driver overlay oneta

docker service create --name web --network oneta --replicas 2 nginx

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

- In the above example, “oneta” provides connectivity between managed service “web” and unmanaged container “client”.

Preserve source IP of Container when accessed externally

- When containers need access to external world, we typically use “bridge” driver with NAT. In this case, container IP is translated to host IP when packets exit the host.
- Some applications(like monitoring app) use the source IP to segregate sessions and they cannot differentiate sessions if source IP is same.
- Following are 2 options to overcome this problem:
  1. Create IP alias to have multiple IP address per interface. Map each Container to 1 of the alias IP.
  2. Use “macvlan” or “ipvlan” network driver. This allows the Container to get IP addresses directly from the underlay network and be part of underlay network.

Preserve Source IP(contd.) –IP alias

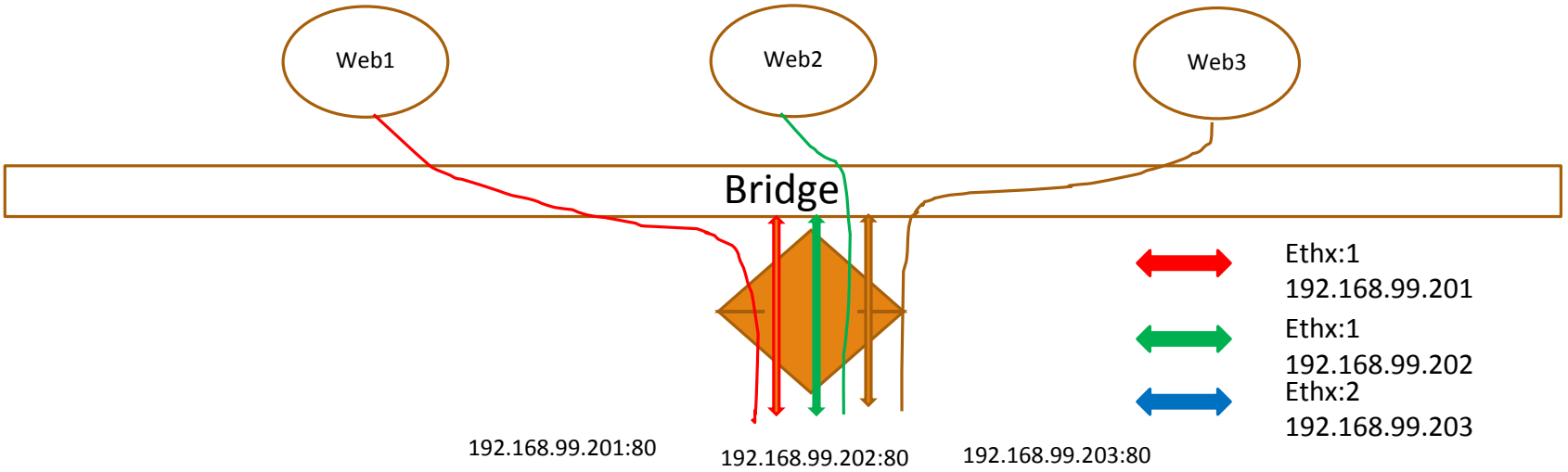
- Create multiple IP address on same interface using IP aliasing feature.

sudo ifconfig eth0:1 192.168.241.101 up
sudo ifconfig eth0:2 192.168.241.101 up
sudo ifconfig eth0:3 192.168.241.101 up

- Create container and map the port to individual IP.

docker run --name web1 -d -p 192.68.241.101:80:80 nginx
docker run --name web2 -d -p 192.68.241.102:80:80 nginx
docker run --name web3 -d -p 192.68.241.103:80:80 nginx

- Based on the destination IP address, the request will be directed towards the specific container. In the return direction, source IP of container would be preserved. 1 of the alias IP would be used as source IP.



Preserve Source IP(contd.) - Macvlan

- Macvlan Docker network driver exposes underlay or host interfaces directly to Containers running in the host.
- Macvlan allows a single physical interface to have multiple mac and ip addresses using macvlan sub-interfaces. The containers are attached to macvlan sub-interfaces.
- Create macvlan network with parent interface as eth1

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

- Create 2 containers in the macvlan network:

docker run -d --name web1 --network macvlan1 nginx
docker run -d --name web2 --network macvlan1 nginx

- The 2 containers “web1” and “web2” will get address in “192.168.2.0/24” subnet and can be directly accessed in the underlay network. The source IP of container would be preserved in this case when there is external access.

Overcoming Routing mesh issues

- With routing mesh, same port gets exposed in all nodes of the Swarm cluster. The service request can come to any node and it will get load balanced to all replicas of the service.

Issues with Routing mesh:

- For services using routing mesh, source IP of packets reaching container gets modified to “ingress” network when the packets reach the container. This creates problem for applications like monitoring which likes the source IP to be preserved.
- Load balancing cannot be done based on application state since routing mesh operates at L3.

Using publish mode “host”:

- Using publish mode “host”, the services gets exposed directly on the host without routing mesh.
- With this approach, external load balancer is needed.
- Following example publishes the service “vote” in all nodes of the cluster using “global” mode. In addition, port 8080 is exposed in every individual node.

docker service create --name vote --network overlay1 --mode global --publish mode=host,target=80,published=8080 instavote/vote

Using dns round-robin instead of Service IP

- Swarm mode uses Service IP based approach by default.
- There are some applications(like Elasticsearch) where this causes issues. (http://derpturkey.com/elasticsearch-cluster-with-docker-engine-swarm-mode/)
- The alternative to Service IP is DNS RR where service names directly resolves to container IP. If there are multiple container replicas, DNS will provide the multiple container IP and basic round robin load balancing will be done.
- Currently, ingress routing mesh is not supported with DNS RR because of some complications with implementation(https://github.com/moby/moby/issues/25016). The workaround is to use publish mode “host” when using DNS RR.
- Following is an example of “vote” service that uses DNS RR with publish mode “host”

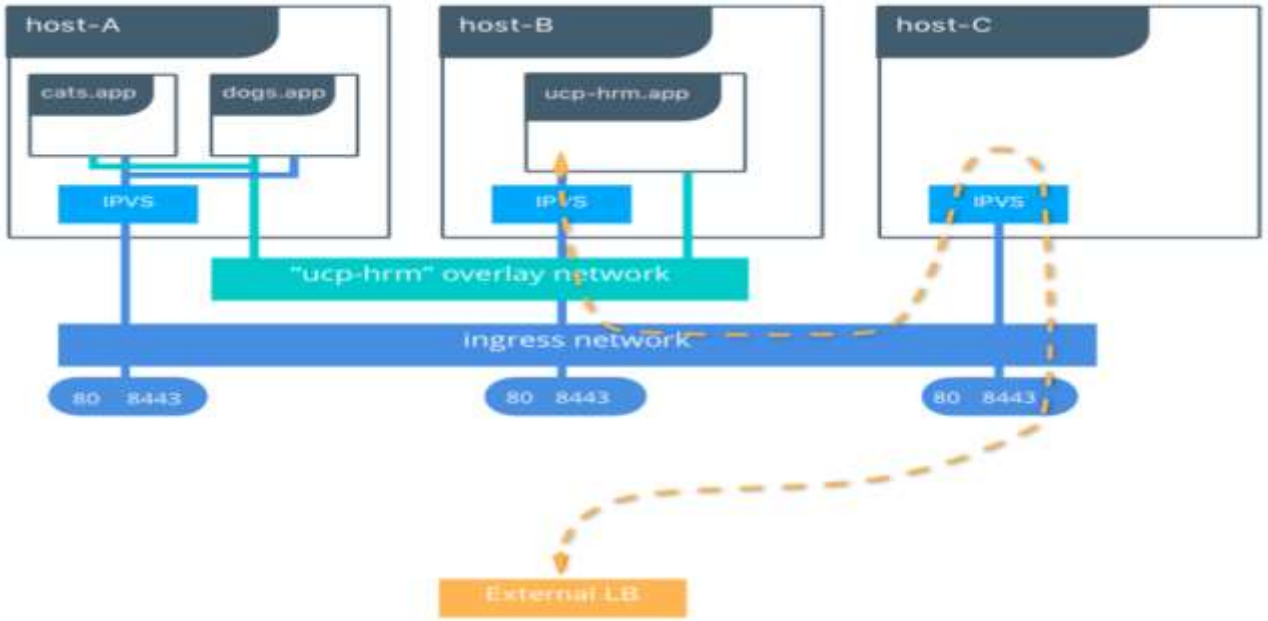
docker service create --endpoint-mode dnsrr --replicas 1 --name client --network overlay1 smakam/myubuntu:v4 ping docker.com

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

- DNS RR approach solves Elasticsearch deployment in Swarm mode. In this case, there is a need to have external load balancer for load balancing.

Routing Mesh vs HRM

- Routing mesh uses transport level load balancing.
- For HTTP based load balancing, we can use HRM(HTTP Routing mesh). This is supported only with Docker EE.
- Routing mesh is used to direct traffic towards HRM container which in turn does HTTP based load balancing.
- Following picture illustrates how HRM works.



Picture from Docker white paper

Managed services with non-overlay network drivers

- Currently Swarm mode services is supported only in Overlay network.
- With Docker 17.06, managed services can be created with bridge, host and macvlan network.(https://github.com/moby/moby/pull/32981)
- Following example creates a Swarm service over macvlan network.

Create config-only network in all nodes:

master:

*docker network create --config-only --subnet=192.168.0.0/16 *
*--ip-range=192.168.2.0/24 *
-o macvlan_mode=bridge -o parent=eth1.70 mvconf

worker:

*docker network create --config-only --subnet=192.168.0.0/16 *
*--ip-range=192.168.3.0/24 *
-o macvlan_mode=bridge -o parent=eth1.70 mvconf

Create macvlan network in swarm master:

docker network create -d macvlan --scope=swarm --config-from mvconf swarm-mv-nw

Create services:

docker service create --replicas 1 --name client --network swarm-mv-nw --detach=false smakam/myubuntu:v4 sleep 10000

docker service create --name vote --network swarm-mv-nw --mode replicated --replicas 1 --publish mode=ingress, target=80, published=8080 instavote/vote

Troubleshooting

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>

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 Overview

BACKUP

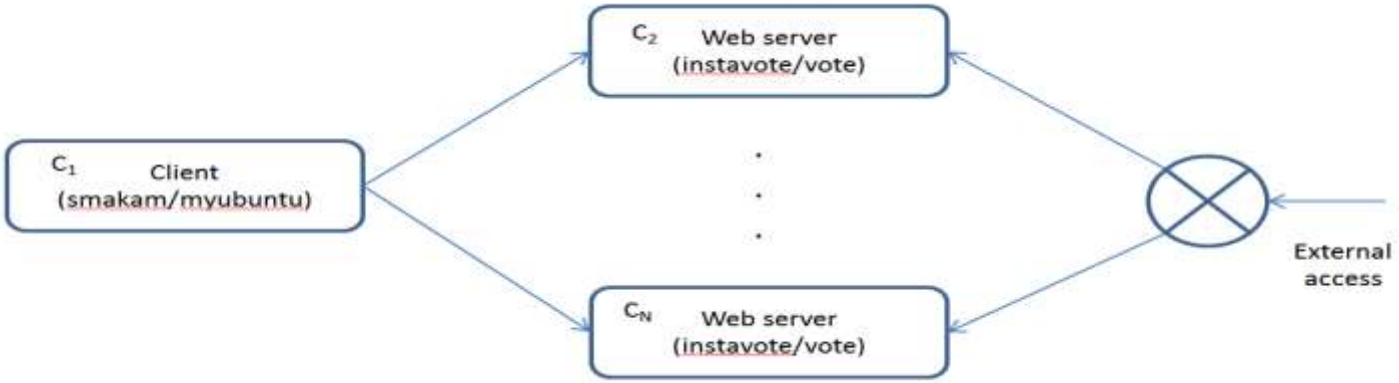
Swarm Networking - Sample application detail

- 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

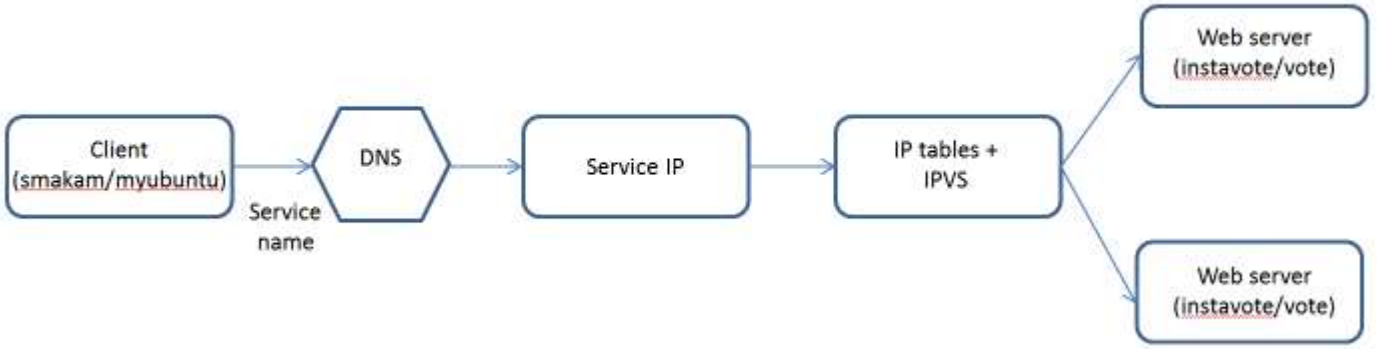
docker service create --replicas 1 --name client --network overlay1 smakam/myubuntu:v4 sleep infinity

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

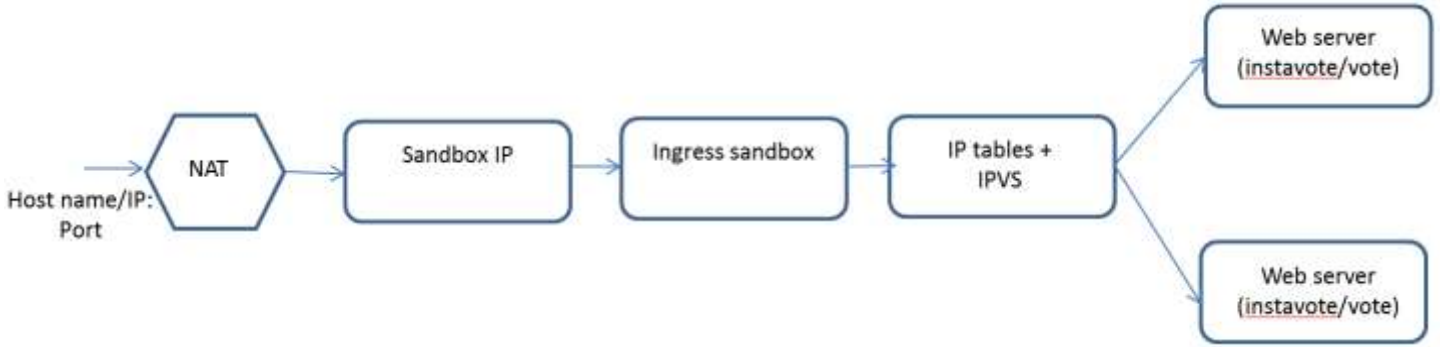


Swarm Networking - Application access flow

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

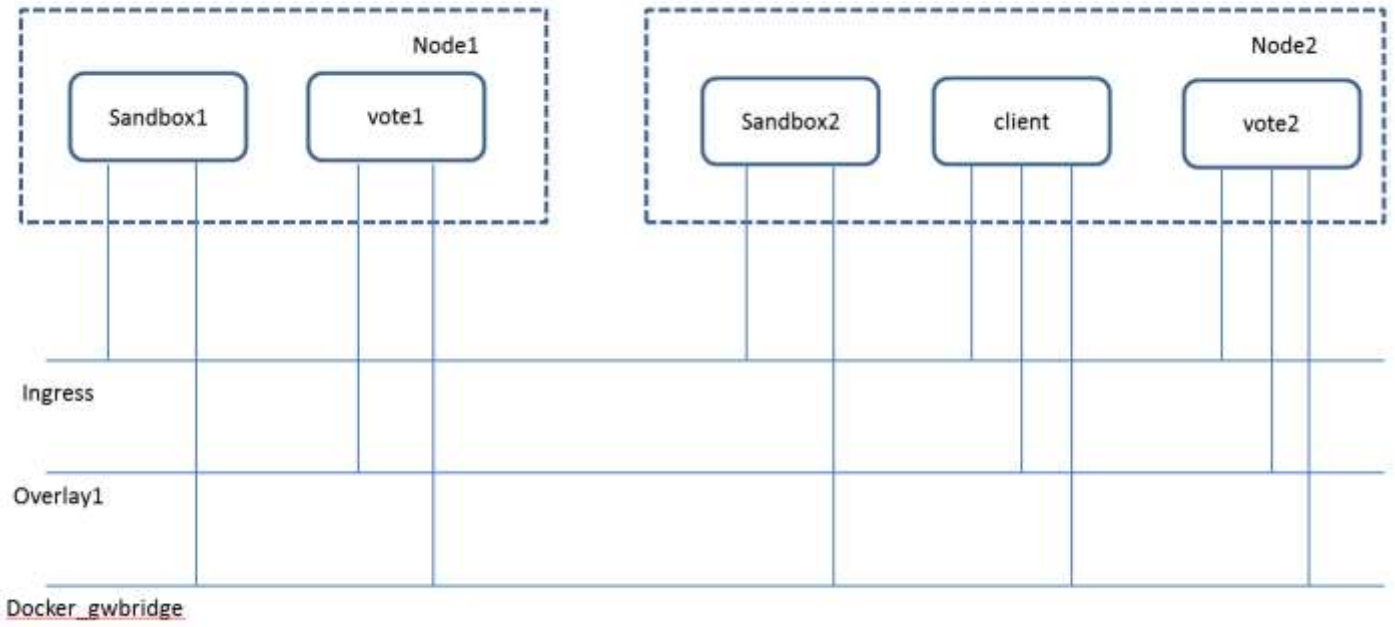


Accessing “vote” service using “ingress” network externally



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 ports gets exposed using publish mode “host”



Linux monitoring tools