nigelpoulton.com

A former big-iron guy now containerized and re-written in Go!

Iconic One Theme | Powered by Wordpress

Demystifying Docker overlay networking

y Nigel Poulton | October 12, 2016 33 Comme

Docker overlay networking is insanely simple to configure. I mean insanely simple! But lurking beneath the simplicity of the setup are a bunch of moving parts that you really wanna understand if you're gonna deploy this stuff in your prime-time production estate.

Anyway… last week I attended the Docker Open Systems Summit in Berlin and got the chance to hang out with some of the networking gods at Docker. My eyes were opened while at the same time my mind was blown! It was intense, but I learned a shed load!

So… I thought I'd write up what I learned and add it as a networking chapter in my book Docker Deep Dive. What follows here is a major excerpt from that chapter.

Enjoy!

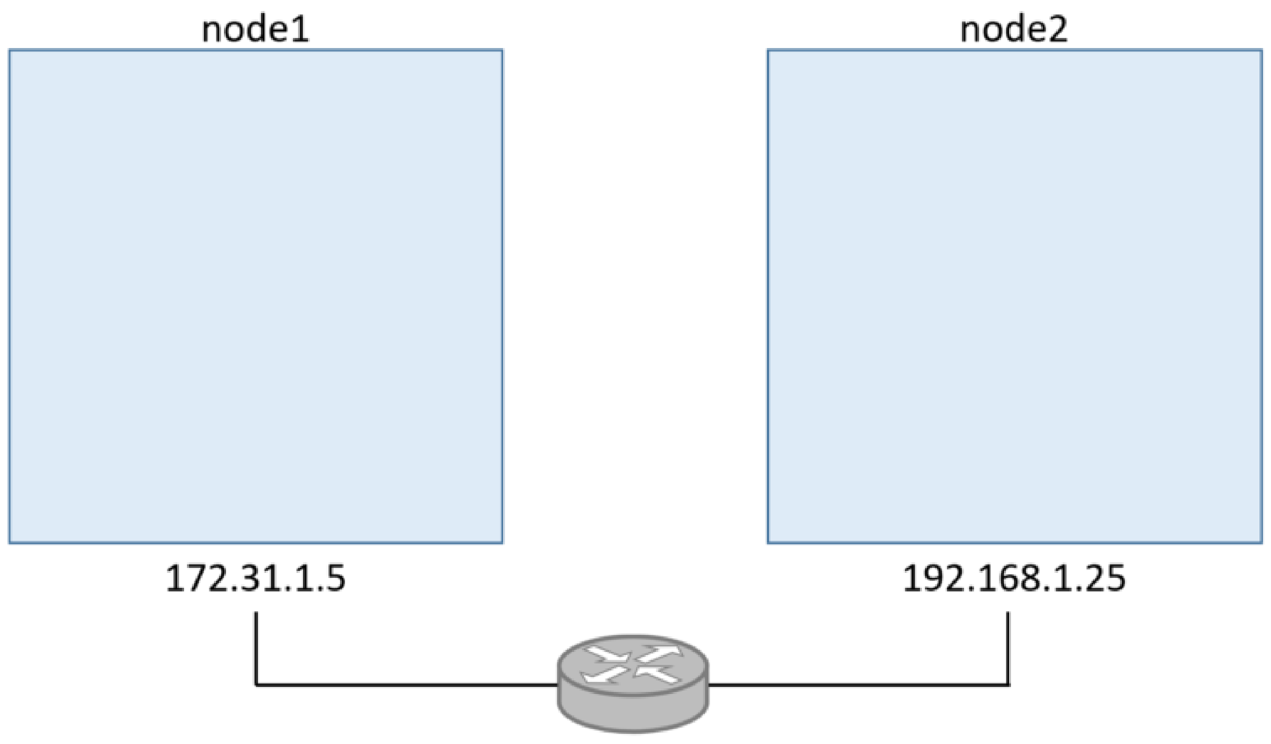
. . . . .

The rest of this chapter will be broken into two parts:

– Part 1: we'll build and test a Docker overlay network in swarm mode
– Part 2: We'll explain the theory behind how it works.

Part 1: Build and test a Docker overlay network in swarm mode

For the following examples we'll use two Docker hosts on two separate Layer 2 networks connected by a router as shown below



Each host is running Docker 1.12 or higher and a 4.4 Linux kernel (newer is always better).

Build a swarm

The first thing we'll do is configure the two hosts into a two-node Swarm. We'll run the `**docker swarm init**` command on **node1** to make it a *manager*, and then we'll run the `**docker swarm join**` command on **node2** to make it a *worker*.

> Warning: If you are following along in your own lab you'll need to swap the IP addresses, container IDs, tokens etc. with the correct values for your environment.

Run the following command on **node1**.

```
$ docker swarm init
Swarm initialized: current node (1ex3...o3px) is now a manager.

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

docker swarm join \
 --token SWMTKN-1-0hz2ec...2vye \
 172.31.1.5:2377
```

Run the next command on **node2**.

```
$ docker swarm join \
> --token SWMTKN-1-0hz2ec...2vye \
> 172.31.1.5:2377
This node joined a swarm as a worker.
```

We now have a two-node Swarm where **node1** is a manager and **node2** is a worker.

Create a new overlay network

Now let's create a new *overlay network* called **uber-net**.

Run the following command from **node1**.

```
$ docker network create -d overlay uber-net
c740ydi1lm89khn5kd52skrd9
```

That's it! You've just created a brand new overlay network that is available to all hosts in the swarm and has its control plane encrypted with TLS!

You can list all networks on each node with the `**docker network ls**` command.

```
$ docker network ls
NETWORK ID    NAME             DRIVER   SCOPE
ddac4ff813b7  bridge           bridge   local
389a7e7e8607  docker_gwbridge  bridge   local
a09f7e6b2ac6  host             host     local
ehw16ycy980s  ingress          overlay  swarm
2b26c11d3469  none             null     local
c740ydi1lm89  uber-net         overlay  swarm
```
```

The network we created is at the bottom of the list called **uber-net**.
The other networks were automatically created when Docker was installed and when we created the swarm. We're only interested in the **uber-net** overlay network.

If you run the `**docker network ls**` command on **node2** you'll notice that it can't see the **uber-net** network. This is because new overlay networks are only made available to worker nodes that have containers using the overlay. This reduces the scope of the network gossip protocol and helps with scalability.

## Attach a service to the overlay network

Let's create a new *Docker service* and attach it to the **uber-net** overlay network. We'll create the service with two replicas (containers) so that one runs on **node1** and the other runs on **node2**. This will automatically extend the **uber-net** overlay to **node2**.

Run the following commands from **node1**.

```
$ docker service create --name test \
--network uber-net \
--replicas 2 \
ubuntu sleep infinity
```
```

The command creates a new service called **test**, attaches it to the **uber-net** overlay network, and creates two containers (replicas) running the `**sleep infinity**` command. This command makes sure the containers don't immediately exit.

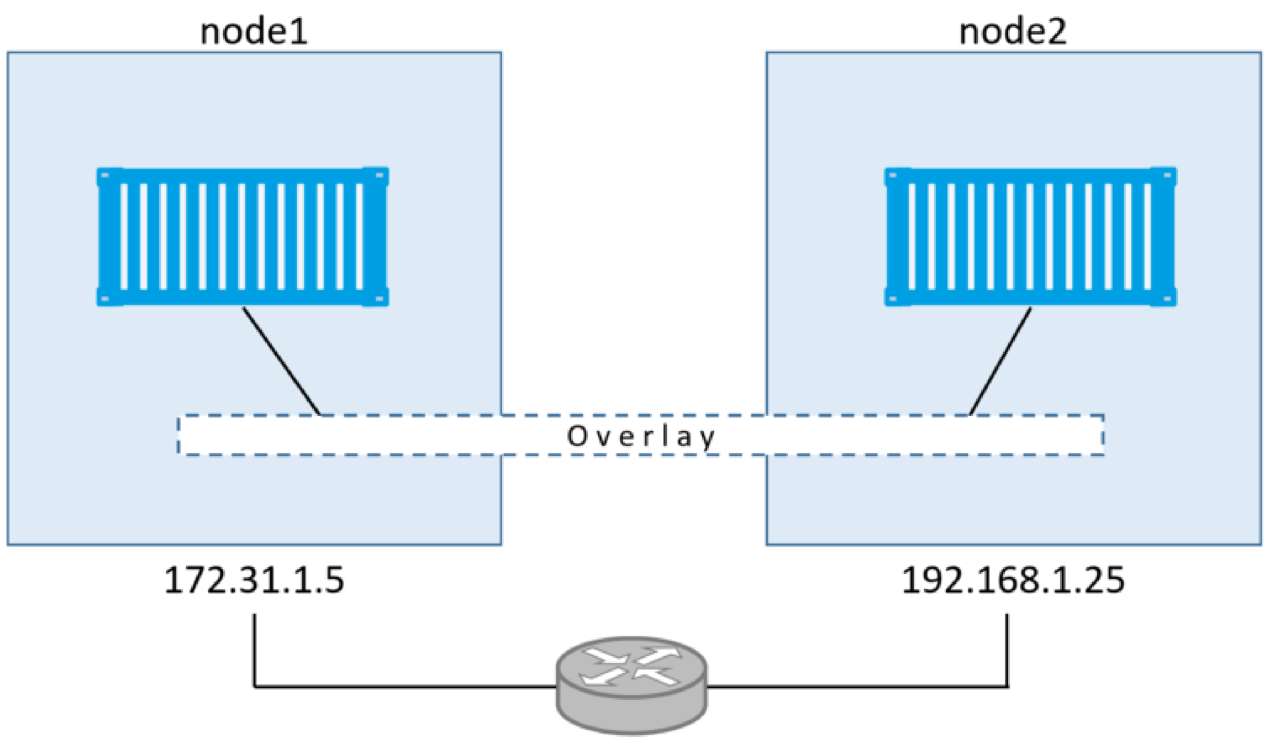
Because we're running two containers (replicas) and the Swarm has two nodes, one container will run on each node.

Verify the operation with a `**docker service ps**` command.

```
$ docker service ps
ID          NAME    IMAGE   NODE   DESIRED STATE    CURRENT STATE
77q...rkx   test.1  ubuntu  node1  Running          Running
97v...pa5   test.2  ubuntu  node2  Running          Running
```

When Swarm starts a container on an overlay network it automatically extends that network to the node the container is running on. This means that the **uber-net** network is now visible on **node2**.

Congratulations! You've created a new overlay network spanning two nodes on separate physical underlay networks, and you've scheduled two containers to use the network. How simple was that!



Test the overlay network

Now let's test the overlay network with the ping command.

In order to do this, we need to do a bit of digging around to get each container's IP address.

From **node1** run a `**docker network inspect**` to see the **Subnet** assigned to the overlay.

```
$ docker network inspect uber-net
[
  {
    "Name": "uber-net",
    "Id": "c740ydi1lm89khn5kd52skrd9",
    "Scope": "swarm",
    "Driver": "overlay",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": null,
      "Config": [
        {
          "Subnet": "10.0.0.0/24",
          "Gateway": "10.0.0.1"
        }
<Snip>
```

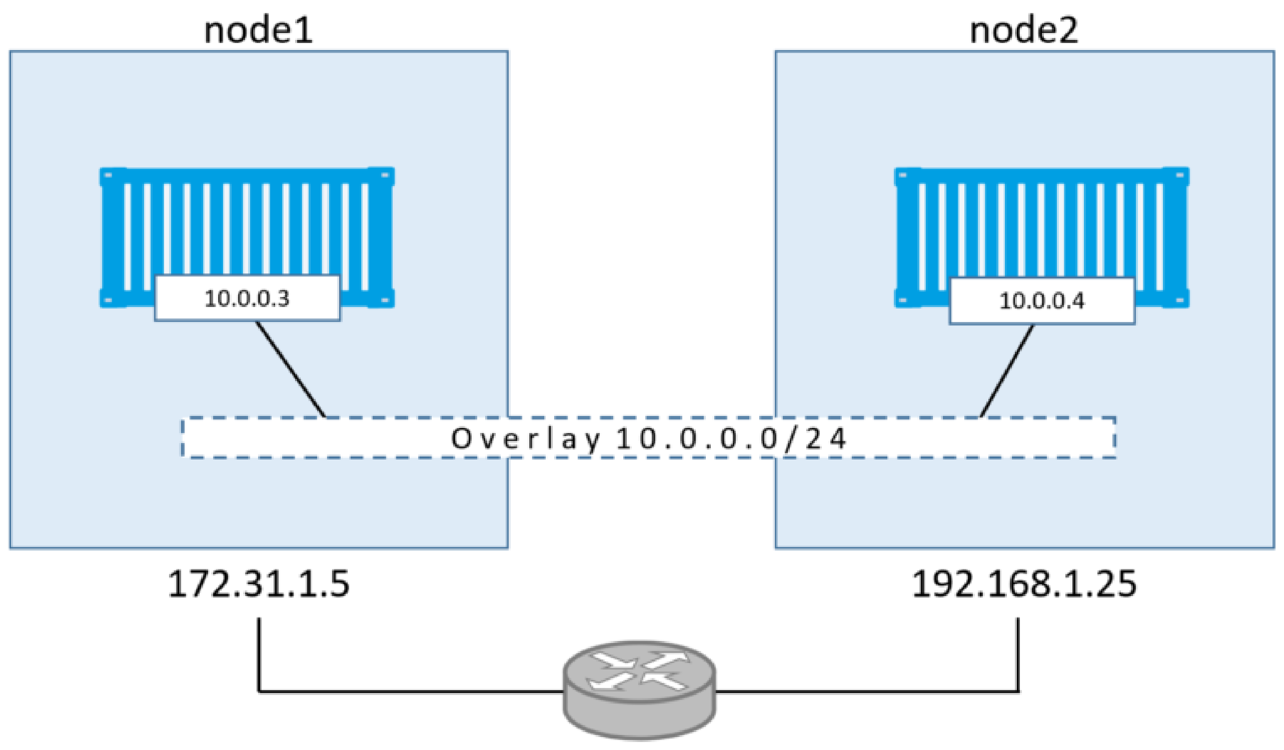
The output above shows that **uber-net**'s subnet is `**10.0.0.0/24**`. Note that this does not match either of the physical underlay networks (`172.31.1.0/24` and `192.168.1.0/24`).

Run the following two commands on **node1** and **node2** to get the container ID's and their IP addresses.

```
$ docker ps
CONTAINER ID  IMAGE           COMMAND            CREATED       STATUS
396c8b142a85  ubuntu:latest   "sleep infinity"   2 hours ago   Up 2 hrs
$
$ docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' 396c8b142a85
10.0.0.3
```

Make sure you run these commands on both nodes to get the IP addresses of both containers.

The diagram below shows the configuration so far.



As we can see, there is a Layer 2 overlay network spanning both hosts, and each container has an IP address on this overlay network. This means that the container on **node1** will be able to ping the container on **node2** using it's `**10.0.0.4**` address from the overlay network. This works despite the fact that both nodes are on separate Layer 2 underlay networks. Let's prove it.

Log on to the container on **node1** and install the `**ping**` utility and then ping the container on **node2** using its `**10.0.0.4**` IP address.

If you're following along, the container ID used below will be different in your environment.

```
$ docker exec -it 396c8b142a85 bash
root@396c8b142a85:/#
root@396c8b142a85:/#
root@396c8b142a85:/# apt-get update
<Snip>
root@396c8b142a85:/#
root@396c8b142a85:/#
root@396c8b142a85:/# apt-get install iputils-ping
Reading package lists... Done
Building dependency tree
Reading state information... Done
<Snip>
Setting up iputils-ping (3:20121221-5ubuntu2) ...
Processing triggers for libc-bin (2.23-0ubuntu3) ...
root@396c8b142a85:/#
root@396c8b142a85:/#
root@396c8b142a85:/# ping 10.0.0.4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=1.06 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=1.07 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=1.03 ms
64 bytes from 10.0.0.4: icmp_seq=4 ttl=64 time=1.26 ms
^C
root@396c8b142a85:/#
```

As shown above, the container on **node1** can ping the container on **node2** using the overlay network.

If you install `**traceroute**` on the container, and trace the route to the remote container, you'll see only a single hop (see below). This proves that the containers are talking directly over the overlay network and are blissfully unaware of any underlay networks being traversed.

```
$ root@396c8b142a85:/# traceroute 10.0.0.4
 traceroute to 10.0.0.4 (10.0.0.4), 30 hops max, 60 byte packets
 1 test-svc.2.97v...a5.uber-net (10.0.0.4) 1.110ms 1.034ms 1.073ms
```

So far we've created an overlay network with a single command. We then added containers to the overlay network on two hosts on two different Layer 2 networks. Once we worked out the container's IP addresses, we proved that they could talk directly over the overlay network.

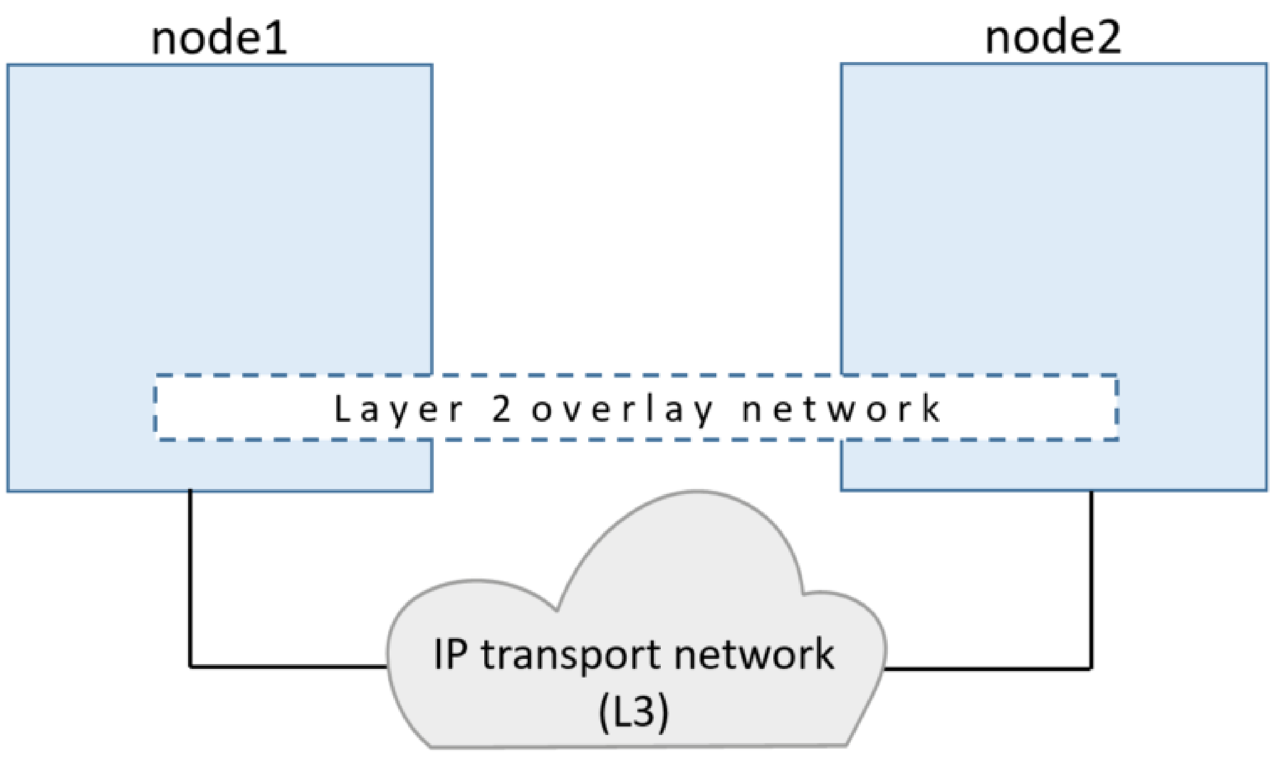
Part 2: The theory of how it all works

Now that we've seen how to build and use a container overlay network, let's find out how it's all put together behind the scenes.

VXLAN primer

First and foremost, Docker overlay networking uses VXLAN tunnels as the underlying for creating virtual Layer 2 overlay networks. So before we go any further, let's do a quick primer on VXLAN technology.

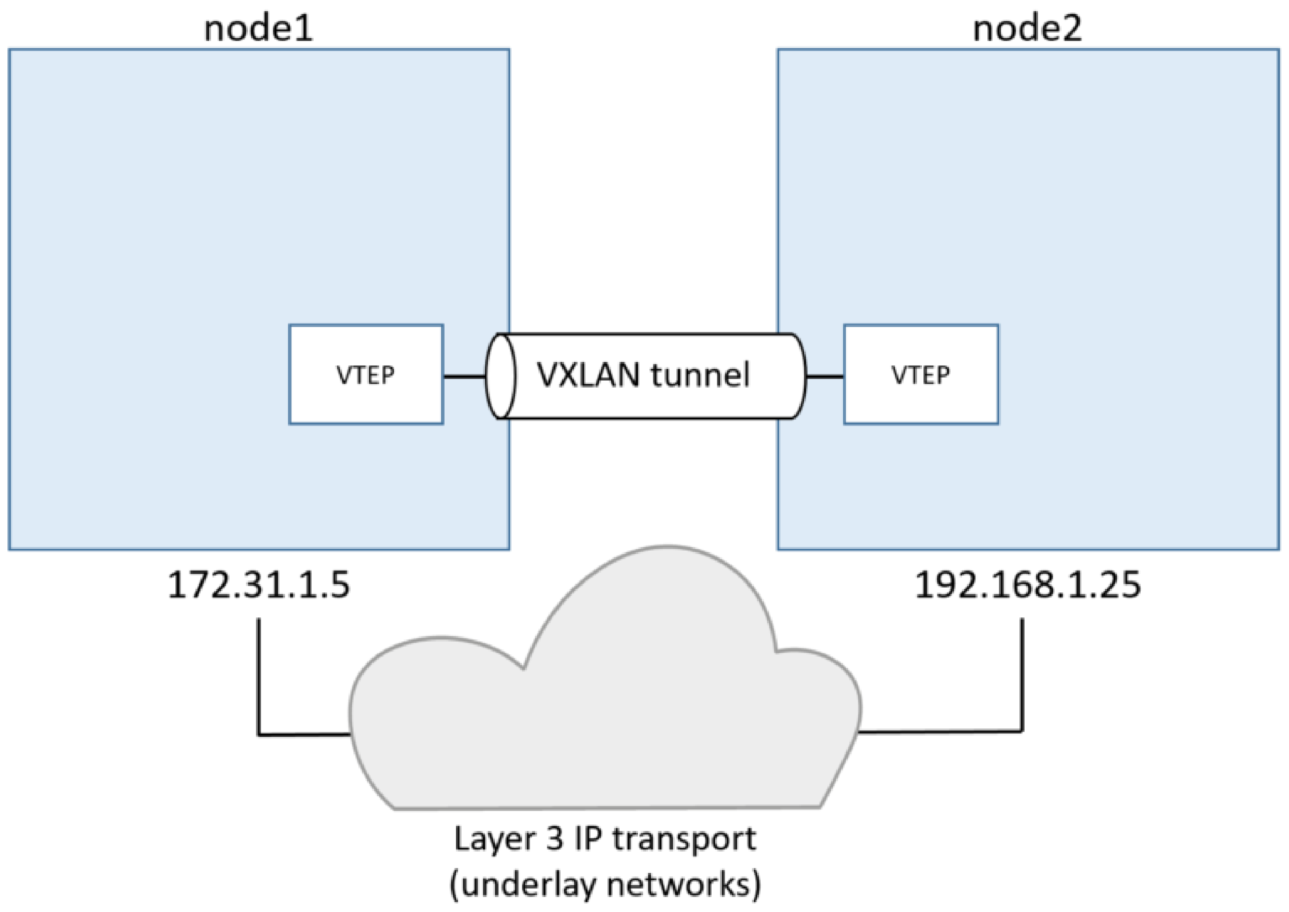
At the highest level, VXLANs let you create a virtual Layer 2 network on top of an existing Layer 3 infrastructure. The example we used earlier created a new 10.0.0.0/24 network on top of a Layer 3 IP network comprising two Layer 2 networks – 172.31.1.0/24 and 192.168.1.0/24. This is shown below.



The beauty of VXLAN is that existing routers and network infrastructure just see the VXLAN traffic as regular IP/UDP packets and handle them without issue.

To create the virtual Layer 2 overlay network a VXLAN *tunnel* is created through the underlying Layer 3 IP infrastructure. You might hear the term *underlay network* used to refer to the underlying Layer 3 infrastructure.

Each end of the VXLAN tunnel is terminated by a VXLAN Tunnel Endpoint (VTEP). It's this VTEP that performs the encapsulation/de-encapsulation and other magic required to make all of this work. See below.

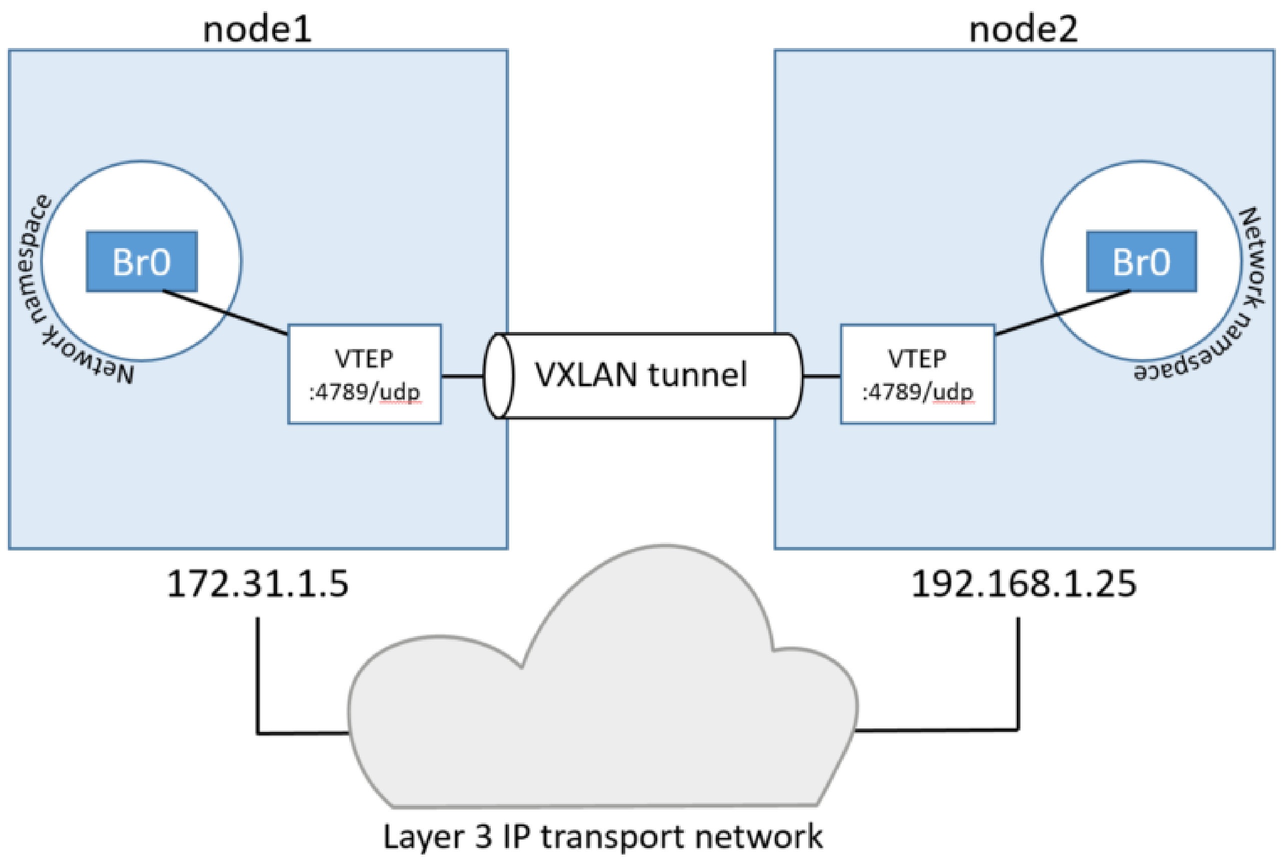


Walk through our two-container example

In the example we built earlier, we had two hosts connected via an IP network. Each host ran a single container, and we created a single VXLAN overlay network for the containers to use.

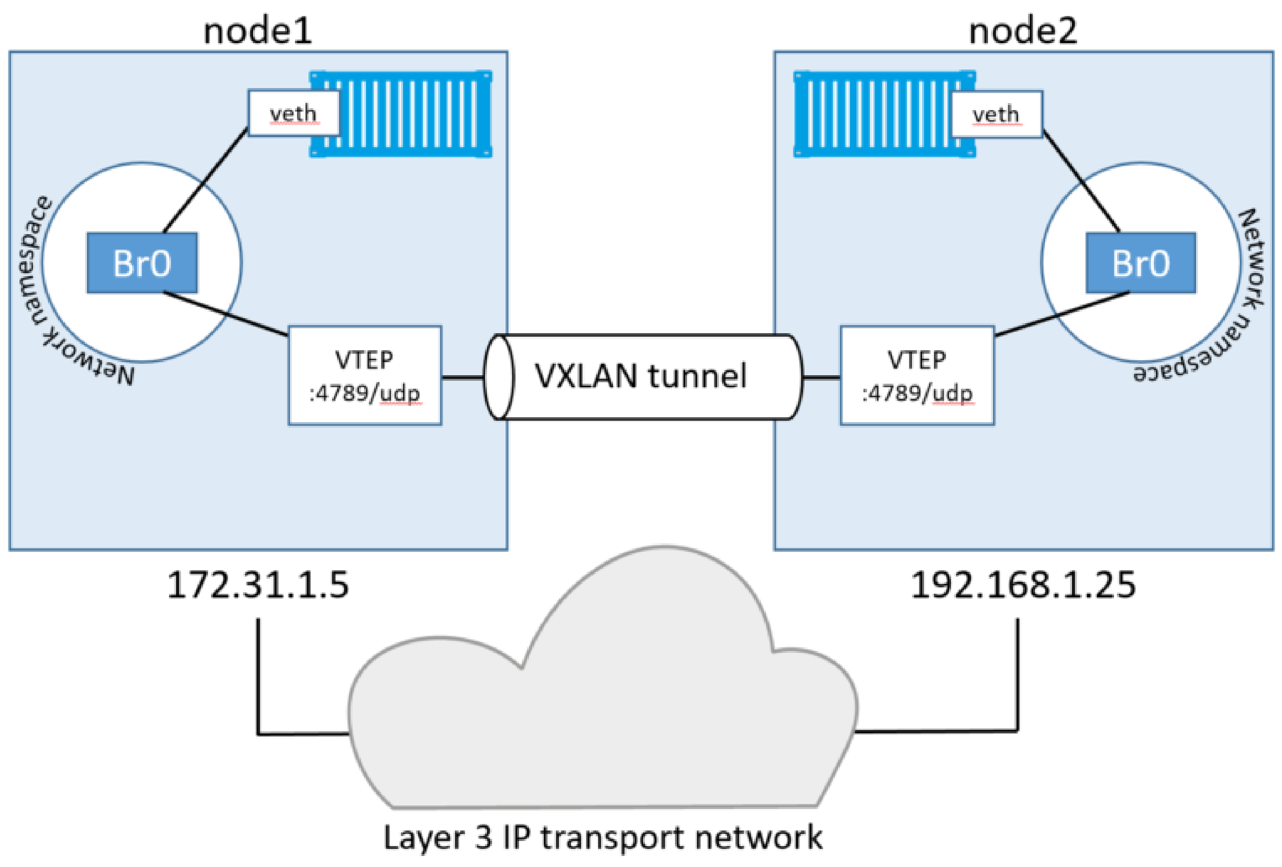
To accomplish this the a new *network namespace* was created on each host. A *network namespace* is like a container, but instead of running an application it runs an isolated network stack – one that's sandboxed from the network stack on the host itself.

A virtual switch (a.k.a virtual bridge) called **Br0** is created inside the network namespace. A VTEP is also created with one end plumbed into the **Br0** virtual switch, and the other end plumbed into the host network stack. The end in the host network stack gets an IP address on the underlay network the host is connected to and is bound to a UDP socket on port 4789. The two VTEPs on each host create the overlay via a VXLAN tunnel as shown below.



This is essentially the VXLAN overlay network created and ready for use.

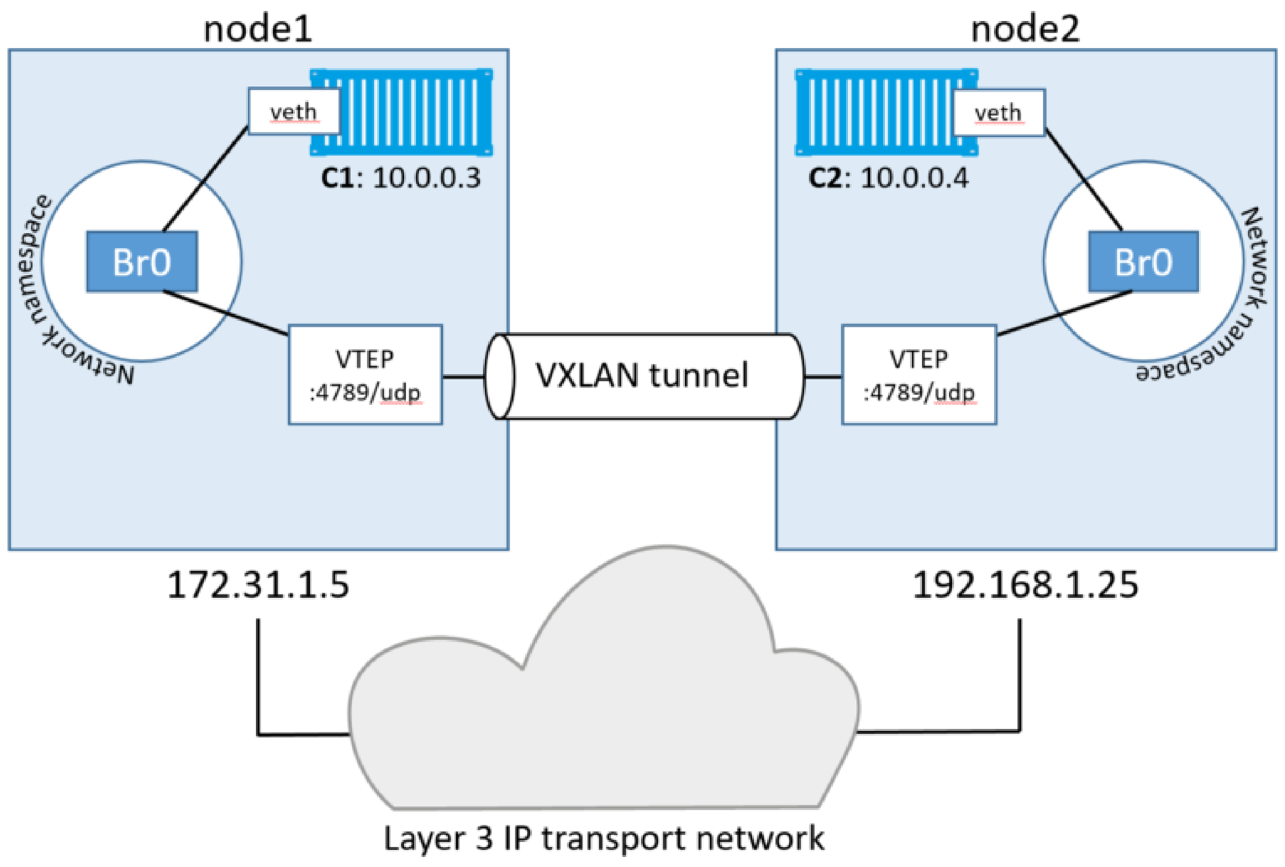
Each container then gets its own virtual Ethernet (**veth**) adapter that is also plumbed into the local **Br0** virtual switch. The topology now looks like the image below, and it should be getting easier to see how the two containers can communicate over the VXLAN overlay network despite their hosts being on two separate networks.



Communication example

Now that we've seen the main plumbing elements let's see how the two containers communicate.

For this example, we'll call the container on node1 "**C1**" and the container on node2 "**C2**". And let's assume **C1** wants to ping **C2** like we did in the practical example earlier in the chapter.



Container **C1** creates the ping requests and sets the destination IP address to be the `10.0.0.4` address of **C2**. It sends the traffic over its **veth** interface which is connected to the **Br0** virtual switch. The virtual switch doesn't know where to send the packet as it doesn't have an entry in its MAC address table (ARP table) that corresponds to the destination IP address. As a result, it floods the packet to all ports. The VTEP interface connected to **Br0** knows how to forward the frame so responds with its own MAC address. This is a **proxy ARP** reply and results in the **Br0** switch *learning* how to forward the packet and it updates its ARP table mapping 10.0.0.4 to the MAC address of the VTEP.

Now that the **Br0** switch has *learned* how to forward traffic to **C2** all future packets for **C2** will be transmitted directly to the VTEP interface. The VTEP interface knows about **C2** because all newly started containers have their network details propagated to other nodes in the swarm using the network's built-in gossip protocol.

The switch then sends the packet to the VTEP interface which encapsulates the frames so they can be sent over the underlay transport infrastructure. At a fairly high level this encapsulation includes adding a VXLAN header to the Ethernet frame. The VXLAN header contains the **VXLAN network ID (VNID)** which is used to map frames from VLANs to VXLANs and vice versa. Each VLAN gets mapped to VNID so that on the receiving end the packet can be de-encapsulated and forwarded on to the correct VLAN. This obviously maintains network isolation. The encapsulation also wraps the frame in a IP/UDP packet with the IP address of the VTEP on node2 in the *destination IP field* and the UDP port 4789 socket information. This encapsulation allows the data to be sent across the underlying networks without the underlying networks having to know anything about VXLAN.

When the packet arrives at **node2**, the kernel sees that it's addressed to UDP port 4789. The kernel also knows that it has a VTEP interface bound to that socket. As a result, it sends the packet to the VTEP which reads the VNID, de-encapsulates the packet and sends it on to its own local **Br0** switch on the VLAN that corresponds the VNID. From there it is delivered to container **C2**.

That's the basics of how VXLAN technology is leveraged by native Docker overlay networks.

We're only scratching the surface here, but it should be enough for you to be able to start the ball rolling with any potential production Docker deployments. It should also give the knowledge required to talk to your networking team about the networking aspects of your Docker infrastructure.

One final thing to mention about Docker overlay networks is that Docker also supports Layer 3 routing within the same overlay network. For example, you can create an overlay network with two subnets, and Docker will take care of routing between them. The command to create a network like this could be `**docker network create –subnet=10.1.1.0/24 –subnet=11.1.1.0/24 -d overlay prod-net**`. This would result in two virtual switches **Br0** and **Br1** being created inside the *network namespace* and routing happens by default.

This has been an excerpt from chapter from my book Docker Deep Dive. I also cover Docker networking in my online video coureses Docker Networking, and Docker Deep Dive (brand new updated course – January 2018).

Category: Books containers Docker Education Networking