Docker ARP Analysis Documentation

Release v0.2

Charles Uneze

Feb 11, 2024

TABLE OF CONTENTS:

I've been immersing myself in CI/CD pipeline studies lately, and now that I've gotten a good grasp of it, I can finally dedicate my full attention to understanding container technologies. To start, I decided to focus on networking, which is the stack I'm most familiar with. In this simple how-to guide, I'll walk you through observing the Address Resolution Protocol (ARP) in action within a Docker environment. An Address Resolution Protocol(ARP) allows a container to learn the MAC address of another device (in this lab, it's the bridge and other containers) dynamically. Without an ARP, a ping between containers, external networks, or even a web request between containers and other devices fails.

CHAPTER

ONE

REQUIREMENTS

I recently made the switch to a Mac and found using Multipass from Canonical simpler than spinning up a VM with tools like Virtualbox. However, when it comes to container networking labs, non-Linux systems are not recommended. While I downloaded Docker Desktop for Mac, I had trouble seeing all the Docker network interfaces. For Windows users, I will advise you to use WSL2 instead, it's easier to deploy and manage compared to having to use a virtual machine like Virtualbox. Keep in mind that the Ubuntu installation requirement is only for Mac and Linux users. This guide will use three terminal tabs throughout.

1.1 Install Homebrew and Multipass

Download Homebrew

```
# Terminal-1 Mac/Linux
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/
↪install.sh)"
```

1.2 Download Multipass

```
# Terminal-1 Mac/Linux
brew install multipass
```

1.3 Download an Ubuntu Server from Multipass

This Ubuntu server will be customized to run the same specifications as an AWS EC2-T2 Micro instance type.

- 1 CPU
- 1 GB of RAM
- 8 GB of disk
- version 22.04

Simple and fast, right?

```
# Terminal-1 Mac/Linux
multipass launch jammy --name=ubuntu --cpus=1 --disk=8G --memory=1G
```

jammy is the image name for Ubuntu server version 22.04.

1.4 Go to the Shell of the Ubuntu Server

```
# Terminal-1 Mac/Linux
multipass shell ubuntu
```

1.5 Install Docker

Download docker on the Ubuntu server.

```
# Terminal-1 Ubuntu
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-
↪compose-plugin

# Close this Ubuntu shell.
# Sometimes you might need to use the exit command severally
# to successfully exit the shell.
exit
```

1.6 Mount a Local Folder to an Ubuntu Directory

Mount the *Documents/* folder in your Mac, or other machine to the *mnt/* directory in the Ubuntu server running inside Multipass.

```
# Terminal-1 Mac/Linux
multipass mount /Users/apple/Documents /mnt/
Verify Ubuntu Server Installation
```

```
apple@Charles-MBP ~ % multipass info ubuntu
Name:           ubuntu
State:          Running
Snapshots:      0
IPv4:           192.168.64.4
                172.17.0.1
Release:        Ubuntu 22.04.3 LTS
Image hash:     9dxa2awl28c8 (Ubuntu 22.04 LTS)
CPU(s):         1
Load:           0.00 0.00 0.00
Disk usage:     2.3GiB out of 7.7GiB
Memory usage:   201.4MiB out of 951.6MiB
Mounts:         /Users/apple/Documents => /mnt
                  UID map: 501:default
                  GID map: 20:default
```

1.7 Install Wireshark

Choose your preferred machine and download.

CHAPTER

TWO

ARCHITECTURE DEPLOYMENT GUIDE

I have a simple architecture that deploys two docker containers in two different subnets. In docker, you can attach one container to several subnets. This is achieved by a new interface being created and assigned to that subnet.

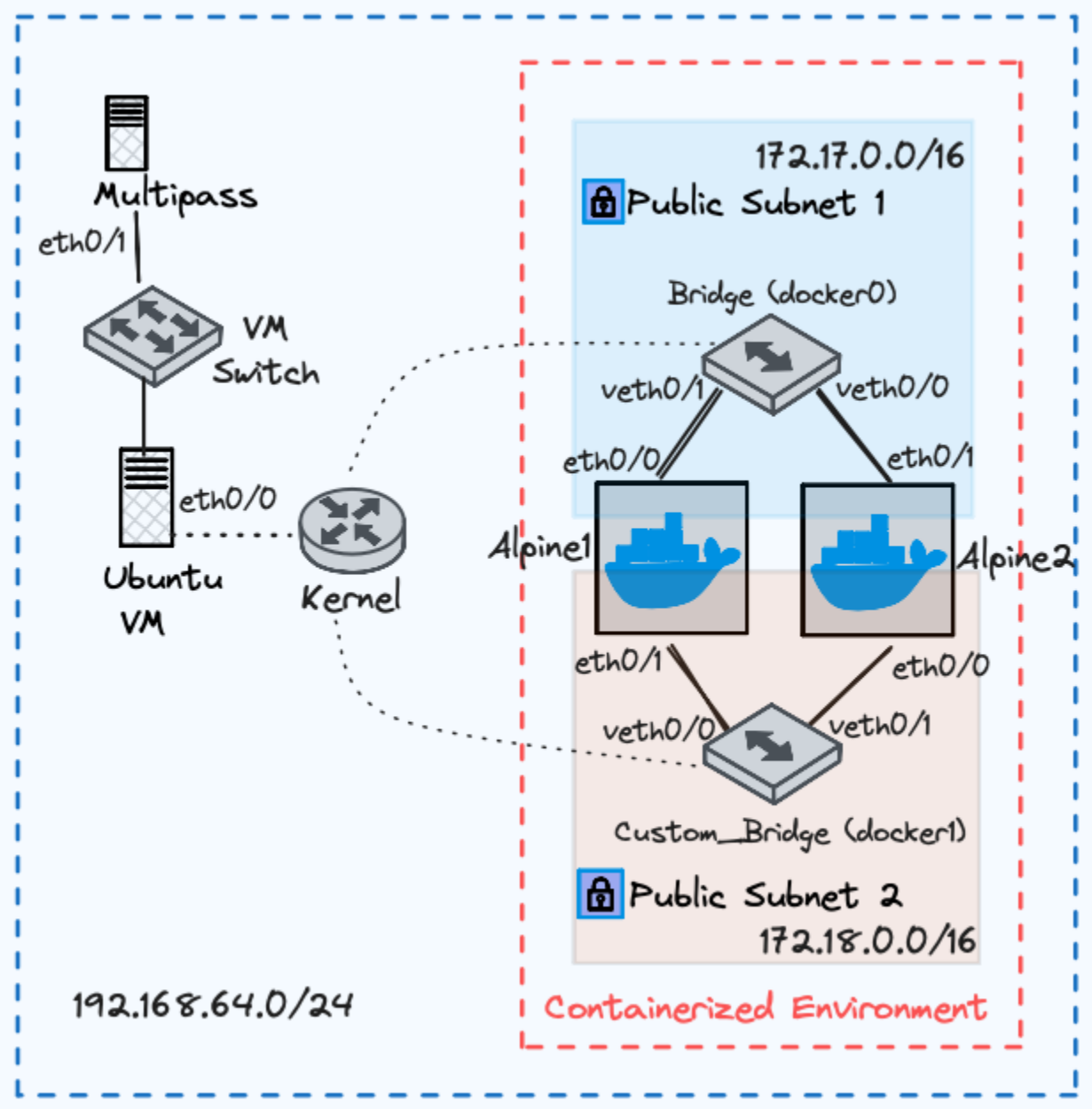
Multipass
eth0/1
VM
Switch
eth0/0
Ubuntu
VM
Kernel
192.168.64.0/24
172.17.0.0/16
Public Subnet 1
Bridge (docker0)
veth0/1
veth0/0
eth0/0
eth0/1
Alpine1
Alpine2
eth0/1
eth0/0
veth0/0
veth0/1
Custom_Bridge (docker1)
Public Subnet 2
172.18.0.0/16
Containerized Environment

Table 1: Network Architecture Table

Device Name	Interface	IP Address	Subnet Mask
Alpine1	eth0/0	172.17.0.2	255.255.0.0
Alpine1	eth0/1	172.17.0.4	255.255.0.0
Alpine2	eth0/0	172.18.0.3	255.255.0.0
Alpine2	eth0/1	172.18.0.5	255.255.0.0
Kernel/Ubuntu VM	eth0/0	192.168.64.4	255.255.255.0
Multipass	eth0/1	192.168.64.1	255.255.255.0
Bridge	docker0	172.17.0.1	255.255.0.0
Bridge	veth0/0	–	–
Bridge	veth0/1	–	–
Custom_Bridge	docker1	172.18.0.1	255.255.0.0
Custom_Bridge	veth0/0	–	–
Custom_Bridge	veth0/1	–	–

2.1 Enter into Ubuntu Shell

Enter Into the Shell of the Ubuntu Server Again

```
# Terminal-1 Mac/Linux
multipass shell ubuntu
Create a New Network
```

I noticed that in docker, you only specify a subnet and mask. This makes sense because if you are deploying this on AWS, a VPC will be defined already, all you need to do is create a new subnet mask for your containerized environment.

2.2 Create a Custom Bridge

```
# Terminal-1 Ubuntu
docker network create \
-o com.docker.network.bridge.name=docker1 \
--subnet=172.18.0.0/24 \
--gateway=172.18.0.253 \
custom_bridge
```

2.3 Verify New Bridge

```
# Terminal-1 Ubuntu

# This is a simulated command and output
ubuntu@ubuntu:~$ docker network ls
NETWORK ID     NAME            DRIVER    SCOPE
6e5ffkvms8c3   bridge          bridge    local
d3b0029faed7   custom_bridge   bridge    local <==
e9f55dsdf605   host            host      local
e9708nj5179a   none            null      local
```

2.4 Pull Alpine Image

I love using Alpine Linux because it's lightweight.

```
# Terminal-1 Ubuntu
docker pull alpine
Open a New Terminal
```

2.5 Open New Terminal Tabs and Capture Packets in Each Bridge

2.5.1 docker0

Execute this command to open a new tab. $+T$

Then enter the Ubuntu shell

```
# Terminal-2 Mac
multipass shell ubuntu
Listen for ARP Packets in Each Bridge
```

Now that the Ubuntu shell has been initialized, execute the below command to capture all packets.

```
# Terminal-2 Ubuntu
sudo tcpdump -i docker0 -w capture_docker_0.pcap
```

Open a third terminal tab $+T$

2.5.2 docker1

Execute another command to listen for all packets in the docker1 bridge interface.

```
# Terminal-3 Ubuntu
sudo tcpdump -i docker1 -w capture_docker_1.pcap
```

2.6 Create 2 Containers in the Default Bridge, Also Connect Them to the Custom Bridge

2.6.1 Create 2 Containers in the Default Bridge

```
# Terminal-2 Ubuntu

# Create containers in the default bridge
docker run -itd \
--name=alpine1 \
--ip=172.17.0.2 \
alpine
```

(continues on next page)

(continued from previous page)

```
docker run -itd \
--name=alpine2 \
--ip=172.17.0.4 \
alpine
```

2.6.2 Connect Containers to Another Network

Connect new interfaces in the containers to another network.

```
# Connect alpine1 to custom_bridge with IP 172.18.0.3
docker network connect --ip=172.18.0.3 custom_bridge alpine1

# Connect alpine2 to custom_bridge with IP 172.18.0.5
docker network connect --ip=172.18.0.5 custom_bridge alpine2
```

2.7 Send Pings to the Internet From the First Interface

Ping google.com four times in each container from *bridge*.

```
# Ping from alpine1 with IP 172.17.0.2
docker exec -it alpine1 ping -I 172.17.0.2 -c 2 google.com

# Ping from alpine2 with IP 172.17.0.4
docker exec -it alpine2 ping -I 172.17.0.4 -c 2 google.com
```

2.8 Send Pings to the Internet From the Second Interface

Ping google.com four times in each container from *custom_bridge*.

```
# Ping from alpine1 with IP 172.18.0.3
docker exec -it alpine1 ping -I 172.18.0.3 -c 2 google.com

# Ping from alpine2 with IP 172.18.0.5
docker exec -it alpine2 ping -I 172.18.0.5 -c 2 google.com
```

2.9 View MAC Addresses of Each Bridge Interface

View MAC addresses of Docker0 and Docker1 bridge interfaces.

Note: The Organizationally Unique Identifier (OUI) of all Docker network adapters is **02:42**. So expect all docker container MAC addresses to begin with that.

```
# Terminal-1 Ubuntu
ip --brief link | grep -E 'docker0|docker1' | awk '{print $1, $3}'
```

Note: Jump to *View Packet Captures*

Output:

```
docker0 02:42:28:a8:cb:f5
docker1 02:42:7c:61:6d:f0
```

2.10 View MAC Addresses of Each Container Interface

View MAC addresses of containers in *bridge* and *custom_bridge* networks.

2.10.1 Bridge network

```
# Terminal-1 Ubuntu
docker network inspect bridge --format '{{range .Containers}}{{.Name}}: {{.MacAddress}}{{
↪"\n"}}{{end}}'
```

Note: Jump to *View Packet Captures*

Output-1:

```
alpine1: 02:42:ac:11:00:02
alpine2: 02:42:ac:11:00:03
```

2.10.2 Custom bridge network

```
# Terminal-2 Ubuntu
docker network inspect custom_bridge --format '{{range .Containers}}{{.Name}}: {{.
↪MacAddress}}{{"\n"}}{{end}}'
```

Output-2:

```
alpine1: 02:42:ac:12:00:03
alpine2: 02:42:ac:12:00:05
```

2.11 End Packet Captures

2.11.1 Packet Capture 1

Stop ARP packet capture in Terminal-2 Ubuntu.

```
# Terminal-2 Ubuntu
"control + c"
```

2.11.2 Packet Capture 2

Stop ARP packet capture in Terminal-3 Ubuntu.

```
# Terminal-3 Ubuntu
"control + c"
```

2.12 Move the Files to the Local Directory

Move captured packet files to the local directory.

```
# Terminal-1, 2, or 3 Ubuntu
mv capture_docker_0.pcap /mnt
mv capture_docker_1.pcap /mnt
```

CHAPTER

THREE

VIEW PACKET CAPTURES

Note: Go to, *View MAC Addresses of Each Bridge Interface* and *View MAC Addresses of Each Container Interface* to confirm the MAC addresses of each device when analyzing the packet capture.

3.1 Docker0 Bridge Interface

Now, I located and opened the packet capture.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	02:42:ac:11:00:02	Broadcast	ARP	42	Who has 172.17.0.1? Tell 172.17.0.2
2	0.000037	02:42:28:a8:cb:f5	02:42:ac:11:00:02	ARP	42	172.17.0.1 is at 02:42:28:a8:cb:f5
3	0.000044	172.17.0.2	192.168.64.1	DNS	70	Standard query 0xab82 A google.com
4	0.001381	172.17.0.2	192.168.64.1	DNS	70	Standard query 0xacb4 AAAA google.com
5	0.073192	192.168.64.1	172.17.0.2	DNS	86	Standard query response 0xab82 A google.com A 142.250.184.14
6	0.074337	192.168.64.1	172.17.0.2	DNS	98	Standard query response 0xacb4 AAAA google.com AAAA 2a00:1450:4003:808::200e
7	0.074821	172.17.0.2	142.250.184.14	ICMP	98	Echo (ping) request id=0x002a, seq=0/0, ttl=64 (reply in 8)
8	0.319984	142.250.184.14	172.17.0.2	ICMP	98	Echo (ping) reply id=0x002a, seq=0/0, ttl=52 (request in 7)
9	1.149902	172.17.0.2	142.250.184.14	ICMP	98	Echo (ping) request id=0x002a, seq=1/256, ttl=64 (reply in 10)
10	1.343037	142.250.184.14	172.17.0.2	ICMP	98	Echo (ping) reply id=0x002a, seq=1/256, ttl=52 (request in 9)
11	1.441313	02:42:ac:11:00:03	Broadcast	ARP	42	Who has 172.17.0.1? Tell 172.17.0.3
12	1.441348	02:42:28:a8:cb:f5	02:42:ac:11:00:03	ARP	42	172.17.0.1 is at 02:42:28:a8:cb:f5
13	1.441354	172.17.0.3	192.168.64.1	DNS	70	Standard query 0x9d49 A google.com

We noticed something interesting: The default MAC addresses of *alpine1: 02:42:ac:11:00:02* and *alpine2: 02:42:ac:11:00:03* were requesting the MAC address for the bridge's interface, *docker0: 02:42:28:a8:cb:f5*, to send an Ethernet frame to *google.com*. The bridge's interface was used as the next hop.

We also see that the DNS name server lookup for *google.com* could be possible only after an ARP reply from *docker0: 02:42:28:a8:cb:f5*.

3.2 Docker1 Bridge Interface

3	539.454556	02:42:7c:61:6d:f0	Broadcast	ARP	42	Who has 172.18.0.3? Tell 172.18.0.1
4	539.454595	02:42:ac:12:00:03	02:42:7c:61:6d:f0	ARP	42	172.18.0.3 is at 02:42:ac:12:00:03
5	539.454618	142.250.184.14	172.18.0.3	ICMP	98	Echo (ping) reply id=0x002f, seq=0/0, ttl=52
6	540.582198	142.250.184.14	172.18.0.3	ICMP	98	Echo (ping) reply id=0x002f, seq=1/256, ttl=52
7	540.848621	02:42:7c:61:6d:f0	Broadcast	ARP	42	Who has 172.18.0.5? Tell 172.18.0.1
8	540.848844	02:42:ac:12:00:05	02:42:7c:61:6d:f0	ARP	42	172.18.0.5 is at 02:42:ac:12:00:05
9	540.848879	142.250.184.14	172.18.0.5	ICMP	98	Echo (ping) reply id=0x0011, seq=0/0, ttl=52
10	541.862551	142.250.184.14	172.18.0.5	ICMP	98	Echo (ping) reply id=0x0011, seq=1/256, ttl=52

The same also applies here. the custom MAC addresses of *alpine1: 02:42:ac:12:00:03* and *alpine2: 02:42:ac:12:00:05* were requesting the mac-address for the bridge's interface *docker1: 02:42:7c:61:6d:f0* so it can send an ethernet frame destined to *google.com*. The bridge's interface is used as the next hop.

CHAPTER

FOUR

CLEAN-UP

4.1 Stop and Remove the Container

```
# Stop containers alpine1 and alpine2
docker stop alpine1 alpine2

# Remove containers alpine1 and alpine2
docker rm alpine1 alpine2
```

4.2 Delete Custom Bridge

```
# Remove custom_bridge network
docker network rm custom_bridge
```

4.3 Verify Network List

```
root@ubuntu:/home/ubuntu# docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
2eabde4e866c   bridge    bridge    local
e9f54b21c605   host      host      local
e9708605179a   none      null      local
```

4.4 Stop Ubuntu Server

```
# Stop Ubuntu server
multipass stop ubuntu
```

CHAPTER

FIVE

CONCLUSION

Now, I understand that everyone can't stop talking about Kubernetes, but a lot of senior engineers have advised that it'd be best to learn docker before picking kubernetes up. Though I've played with Kubernetes severally, I struggled. However, each new day I keep spending with docker makes understanding kubernetes a piece of cake. This guide serves as a strong foundation for analyzing ARP packets in containers.

CHAPTER

SIX

RESOURCES

I enjoyed these two articles from Hank Preston, a principal engineer at Cisco. The last one is from me.

- Exploring Default Docker Networking Part 1
- Exploring Default Docker Networking Part 2
- A Routers Intimacy With MAC Addresses (I wrote this a few years ago. It discusses ARP and ICMP in a Cisco environment.)