Implementing a Virtual Network System among Containers

Songlin Jiang
songlin.jiang@aalto.fi

Tutor: Tuomas Aura

Abstract

This paper investigates the method of implementing a virtual network system among containers. We first implement a testbed environment for VPN in IPv4 (site-to-site, host-to-host) and IPv6 (site-to-site). Then we compare VirtualBox-based Vagrant with Docker Compose in realizing the same networking features. The result shows that migrating the test from the Virtual Machines (VM) to the Docker containers can save nearly 90% of CPU time and 94% of memory for the VPN system. At the same time, the host machine still has no security risk increase.

KEYWORDS: Container, Network, Cloud, VPN

1 Introduction

In recent years, container technologies, such as Docker, have received much attention from the industry and academia as applications are moving to the cloud. Containers are much more efficient and lightweight than virtual machines because containers share the Linux kernel with the host. In contrast, virtual machines employ hardware virtualization and have their own kernel instance, which consumes more resources [16].

Virtual Private Networks (VPN) are typically used in complex network-

ing environments with many different network components. It is still a common practice [11] to build and test network systems configurations using virtual machines, which allows the network engineer to experiment in a virtual environment before setting up the physical system. In addition, virtual networks are sometimes also needed for automated integration tests, as developers may want to test the usability and performance of their software under some specific network topology.

However, creating a virtual computing environment can be slow and troublesome. The problems can worsen when testing VPN systems with a large number of nodes on one host machine, as each network component and host needs a virtual machine instance. It can be memory-consuming to simultaneously run many virtual machine instances on one host machine to simulate the network environment, which significantly troubles testers working on personal computers short of memory. Moreover, Mac M1 / M2 chips are based on the ARM64 architecture, and virtual machine hypervisors currently have limited support for ARM64 hardware virtualization. In contrast, ARM64 is well supported by container runtimes [12]. Furthermore, Containers are easy to launch on demand in the cloud, and the cost is low because they can run within one virtual machine. Running with virtual machines can significantly increase the cost of Continuous Integration / Continuous Delivery (CI/CD) implementations in the cloud, as using nested virtualization with limited resources is hard. Most importantly, building, storing, and managing multi-platform virtual machine images in the cloud is also challenging. In contrast, Docker supports building multi-platform images [8] and uploading them to Docker Hub.

Due to little research on implementing network systems using containers, this paper investigates the possibilities of implementing virtual network systems based on Docker containers to overcome the disadvantages of virtual machines mentioned above. This paper also analyzes the functional and security limitations when virtual networks are implemented this way.

This paper is organized as follows. Section 2 reviews the current technologies used for container networking. Section 3 explains our case study of implementing a VPN system using Docker containers, while Section 4 explains the details of our implementation. Section 5 compares the VPN system performance when implemented with the virtual machines and containers. Finally, Section 6 provides the concluding remarks.

2 Docker Networking System

This section discusses the choice of the Docker container network driver for implementing the virtual network system. It also discusses how to enable routing, firewalls and IPv6 inside a container.

2.1 Network Drivers

Docker employs a pluggable networking subsystem. Several drivers can provide the Docker network functionality. The default drivers include the bridge, host, overlay, IPVLAN, MACVLAN, and none. We can choose one of them to implement the virtual network system.

The none, host, and overlay drivers do not cater for our needs. Firstly, the none driver disables all networking. Secondly, the host driver is unnecessary as it can have security implications due to its nature of sharing the same network stack with the host machine. In addition, the overlay is not an appropriate option as we are simulating the network system only on one host machine.

As a result, the possible candidates are bridge, IPVLAN, and MACVLAN. When examining the details, the bridge learns Media Access Control (MAC) addresses by checking the frame headers sent by the communicating hosts. On the other hand, the MACVLAN is a trivial bridge that does not need to learn as it already knows every MAC address it can receive [4]. IPVLAN is similar to MACVLAN, except IPVLAN assigns the same MAC address to all containers attached to it. In contrast, MACVLAN assigns a different MAC address to each attached Docker container [5].

MACVLAN is the best choice for our needs. As we only use the driver to implement the internal network, there is no need to use advanced flood control and forwarding database manipulation that are specific to the bridge driver. MACVLAN bridge mode allows the testbed network to run layer 2 (data link layer) protocols, such as the Address Resolution Protocol (ARP) and Link Layer Discovery Protocol (LLDP). MACVLAN also supports address configuration and discovery protocols, as well as other multicast protocols, such as Dynamic Host Configuration Protocol (DHCP) [17] and Precision Time Protocol (PTP) [1]. Moreover, according to Docker documentation related to networking [9], MACVLAN networks are the best choice when migrating from a VM setup, as MACVLAN makes the container appear as a physical device with its own MAC address. Furthermore, Gundall et al. [10] conduct benchmarks for different virtualiza-

tion technologies for the networking overhead. The result shows that the MACVLAN driver performs best in throughput while requiring the least CPU resources compared to the other network drivers.

2.2 Routing and Firewall

By default, Docker containers do not allow manipulating container network devices and setting routing tables or firewalls inside the containers. These features can be enabled by assigning the net_admin capability to the container.

According to the capabilities man page [13], assigning the net_admin capability to containers allows the following network-related operations:

1. Interface configuration;
2. Administration of IP firewall, masquerading, and accounting;
3. Modifying routing tables;
4. Binding to any address for transparent proxying;
5. Setting the type-of-service (TOS);
6. Clearing driver statistics;
7. Setting the promiscuous mode;
8. Enabling multicasting;
9. Using setsockopt(2) to set the following socket options: SO_DEBUG, SO_MARK, SO_PRIORITY (for a priority outside the range 0 to 6), SO_RCVBUFFORCE, and SO_SNDBUFFORCE.

As we use the MACVLAN to build the internal network, all the network manipulations mentioned above only work for the network components that belong to the corresponding network namespace. There are no security implications to the host network if we give net_admin capability to the container with its network namespace.

2.3 IPv6

Configuring IPv6 networking in Docker [7] is also possible. Docker disables the IPv6 support by default. We can add the following content to the daemon configuration file (default location at: `/etc/docker/daemon.json`) or corresponding settings in Docker Desktop:

```
{ "ipv6": true, "fixed-cidr-v6": "fd00::/80" }
```

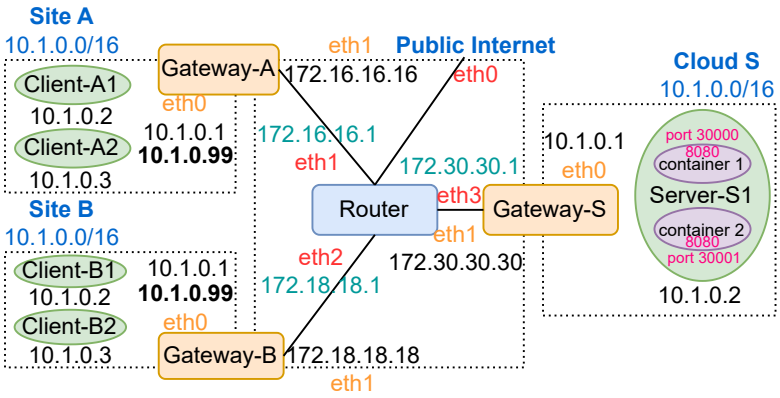

Figure 1. Host to Host

3 Case Study: VPN

This paper simulates a scenario where the IoT devices (clients) in two sites, A and B, would like to connect to the server in the cloud S. The topology is based on the Aalto University CS-E4300 Network Security 2022-2023 instance Project 2 [3], where site A, B, and cloud S both use the private IP addresses to improve the security and save the IPv4 addresses. Gateways A, B, and S connect sites A, B, and S to the public Internet. The router in the topology represents the Internet between the sites and the cloud. The address space between the gateway and the router simulates public, routable IPv4 addresses, although they are all private. Site A, B, and cloud S use the router to access the Internet.

In order to make the clients in both site A and site B connect to the cloud server safely, this paper attempts to implement a virtual network testbed for this networking exercise based on Docker containers. We experiment with two types of VPN: site-to-site and host-to-host, using strongSwan, a VPN implementation based on Internet Protocol Security (IPsec).

3.1 Host to Host

A host-to-host VPN connects different gateways together. IP packets then get routed to different clients within the corresponding site according to the routing table of the gateway [14].

Figure 1 shows the topology and corresponding address spaces under such circumstances.

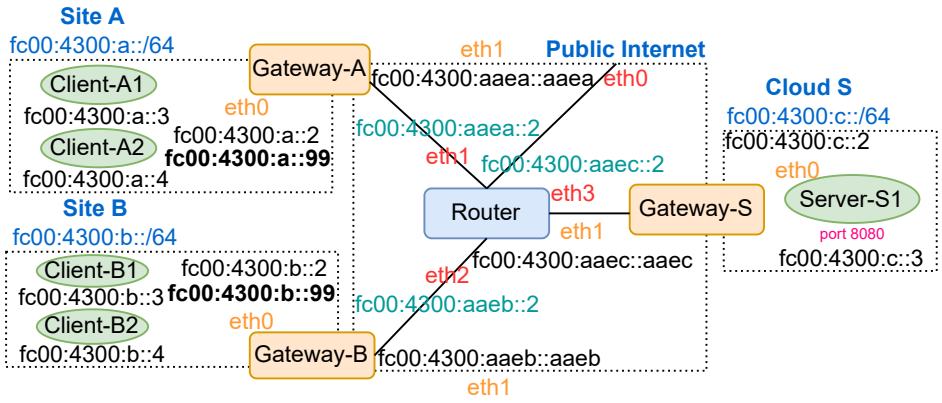

Figure 2. Site to Site in IPv6

3.2 Site to Site

A site-to-site VPN connects different network systems located at different sites together directly [2]. In this case, the address spaces of sites A, B and the cloud network S should not overlap [15].

Our site-to-site VPN topology is almost the same as figure 1, except that the address space for site B is 10.2.0.0/16, and cloud S is 10.3.0.0/16.

3.3 Site to Site in IPv6

Site to Site in IPv6 is similar to Site to Site VPN in IPv4 but replaces all the address space in IPv6.

Figure 2 shows the topology and corresponding address space under such circumstances.

4 Implementation

To manage the Docker containers, we use Docker Compose for orchestration. Based on the figures above, we make each network component a separate container, assign the net_admin capability to each container, and implement the network with the MACVLAN driver. We only connect the router to the Docker default bridge network to enable connections to the public Internet. Finally, we set up the routing and firewall and configured the strongSwan IPsec with certificates.

We use Network Address Translation (NAT) masquerade for the gateway interface to prevent leaking local IP addresses outside their subnets for routing. We bind the preconfigured local server IP address 10.1.0.99 or 10.2.0.99 to the interface eth0 of corresponding gateway A and B accord-

ing to their subnets.

We use strict firewall rules on the clients, assuming there should be no need for the clients (IoT devices) to visit the Internet. We use iptables to set up gateway A and B firewall rules for input and output. We accept Internet Key Exchange (IKE) and Encapsulating Security Payload (ESP) traffic (port 500 and 4500 in UDP) from and to the cloud. Finally, we drop everything else, including the connection from and to the Internet.

As explained in the following sections, there are also some differences between the two kinds of VPN setups, and between IPv4 and IPv6.

4.1 Host to Host

About routing, for gateway A and B, we redirect the traffic from the original local server address 10.1.0.99 of port 8080 to cloud gateway S of port 8080 with Destination NAT. For gateway S, we redirect the traffic from the client gateway A and B of port 8080 to corresponding ports (30000 and 30001) on the server s1 address.

There are overlapping network address spaces for host-to-host VPN. Suppose we specify the IP address and subnet directly through Docker Compose. In that case, there will be errors notifying us that 'Pool overlaps with another one on this address space'. Although technically this should not be a problem, as we are creating a separate internal network without direct routing, Docker still forbids us.

We have addressed the issue of overlapping address space through a method that can be likened to IP address spoofing. When we do not specify the IP address of the network in Docker Compose, it will assign a random address from the Docker address pool to the network interface. Now we can modify the IP address and subnet of the interface to our desired one, and no error will be thrown now.

In addition, for server S1, we use the Docker-in-Docker image to run Docker containers inside the server S1 container. Running that image requires the container to be privileged according to the documentation [6]. As a result, in practice, we must ensure the software running in server S1 is benign and poses no risk to the host machine. It can be acceptable in a testbed network but not in production. Otherwise, we recommend having a separate VM for the servers.

4.2 Site to Site

Concerning routing, for gateways A and B, we redirect the traffic from the local address (10.1.0.99 and 10.2.0.99) of port 8080 to cloud server S1 on port 8080 with Destination NAT.

For the firewall on gateways A and B, we also have to accept the post-routing traffic to the cloud server 10.3.0.3.

There is no overlapping network address space within the site-to-site VPN. Thus, we can specify the IP address and subnet directly through Docker Compose. Additionally, we avoid using the IP address ending in ".1", as Docker does not allow us to assign that address to any container. These addresses are reserved for gateways or routers on a particular network (although we are simulating the Gateway and Router).

4.3 Site to Site in IPv6

Site to Site in IPv6 is similar to Site to Site in IPv4. Just replacing all the IPv4 addresses with IPv6 would complete the job. The only difference is that, in addition to the existing configurations, we also have to allow ICMPv6 traffic at the gateways for firewall rules. In IPv6, Neighbor Discovery is a necessary component, replacing Address Resolution Protocol (ARP) in IPv4. This way, IPv6 Neighbor Discovery can work, and different containers can communicate within their subnet. We also need ICMPv6 for Destination Unreachable messages.

5 Evaluation

This section summarizes the result of our experiment. It makes a comparison between the virtual-machine-based testbed and our container-based testbed.

5.1 Usability and Portability

We can use Docker Buildx [8] to create the testing environment images for multiple platforms with only one machine, then upload them to Docker Hub for reusing. Developers do not need to be aware of the different processor architectures when starting the container-based testbed. In contrast, VM monitors, such as VirtualBox, usually do not have a centralized image hub. The developer must choose the right image based on the ar-

Table 1. Performance Test Result in Average

Solution	Boot Time[1]	Memory[2]
Docker Compose	75 s	278 MB
Vagrant + VirtualBox	689 s	4.5 GB

chitecture before running the VM testbed.

Most importantly, the Docker setup can run in M1/M2-based macOS with Docker Desktop. In contrast, the VM ones cannot be run due to the limited support of VirtualBox for ARM64.

The shell script commands for all the routing and firewall configurations in Docker containers and VMs are identical. Hence migrating configurations from the VMs to the Docker containers is easy. We can simultaneously start the site-to-site and host-to-host VPN setup in Docker without interfering with each other.

5.2 Performance

We use Docker Engine 23.0.1 to run the containers and VirtualBox 7.0.6 to run the Virtual Machines. Table 1 reflects the average situation for all the three implementations. Table 1 shows that our container solution significantly reduces the fresh boot time[1] for the whole VPN system by nearly 90%. It dramatically reduces the memory consumption[2] by nearly 94% as well.

5.3 Security

We can check the current virtual network devices with the command `ls /sys/devices/virtual/net -l`. It shows different results when executing from the host machine and inside the container, indicating that the network stacks inside containers are entirely isolated from the host machine.

5.4 Limitations

There are also some limitations of the Docker networking model, which cause several observable differences compared to VMs. However, these limitations generally have workarounds to bypass and will not stop us from adopting container solutions.

[1]Also include the running environment building time for the host platform
[2]Maximum value during the whole running process

1. We cannot have overlapped IP address ranges in different virtual network segments assigned by Docker. The only way to do that is to configure the IP addresses manually inside containers.
2. We are not allowed to assign the IP address ending in ".1" to a Docker container, and we will only get a warning if we do that in a VM. Similar to the previous limitation, a workaround is to assign the IP addresses ending in ".1" manually inside containers.

In addition, if we want to test the scalability of the network software and run Docker containers inside a Docker container, that container needs to be privileged. Such requirements may cause some security risks, but this is unnecessary for implementing virtual networks.

6 Conclusion

This paper investigates how to construct the testbed network environment through the case study of container-based and VM-based VPN configurations. Containers are much more lightweight than virtual machines. As a mature virtualization technology, containers can realize every functionality we require, similar to virtual machines, while increasing no security risk to the host machine.

We hope this paper can inspire researchers and engineers to migrate their testing environment related to network systems from VMs into containers.

Source code for this paper: *https://github.com/HollowMan6/Implement-VPN-System-with-Containers/tree/main/src*

References

[1] Giuliano Albanese, Robert Birke, Georgia Giannopoulou, Sandro Schönborn, and Thanikesavan Sivanthi. Evaluation of networking options for containerized deployment of real-time applications. In *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2021.

[2] Aung, Si Thu and Thein, Thandar. Comparative Analysis of Site-to-Site Layer 2 Virtual Private Networks. In *2020 IEEE Conference on Computer Applications(ICCA)*, 2020.

[3] Aura, Tuomas and Peltonen, Aleksi and Bui, Thanh. Tuomaura/CS-e4300_testbed: TESTBED network setup for Student Projects, Dec 2022. GitHub Repository.

[4] Cha, Jae-Geun and Kim, Sun Wook. Design and Evaluation of Container-based Networking for Low-latency Edge Services. In *2021 International Conference on Information and Communication Tech Convergence (ICTC)*, pages 1287–1289, 2021. IEEE.

[5] Claassen, Joris and Koning, Ralph and Grosso, Paola. Linux containers networking: Performance and scalability of kernel modules. In *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, pages 713–717, 2016.

[6] Docker contributors. Docker-in-Docker image README, February 2023. Docker Hub.

[7] Docker contributors. Enable ipv6 support, February 2023. Docker Documentation.

[8] Docker contributors. Multi-platform images, March 2023. Docker Documentation.

[9] Docker contributors. Networking overview, February 2023. Docker Documentation.

[10] Gundall, Michael and Reti, Daniel and Schotten, Hans D. Application of Virtualization Technologies in Novel Industrial Automation: Catalyst or Show-Stopper? In *2020 IEEE 18th International Conference on Industrial Informatics (INDIN)*, volume 1, pages 283–290, 2020.

[11] Hauser, Frederik and Häberle, Marco and Schmidt, Mark and Menth, Mich. P4-IPsec: Site-to-Site and Host-to-Site VPN With IPsec in P4-Based SDN. *IEEE Access*, 8:139567–139586, 2020.

[12] Kaiser, Shahidullah and Haq, Md. Sadun and Tosun, Ali Saman and Korkmaz, Turgay. Container Technologies for ARM Architecture: A Comprehensive Survey of the State-of-the-Art. *IEEE Access*, 10:84853–84881, 2022.

[13] Linux contributors. *Capabilities(7)*, February 2023. Linux Man Page.

[14] Du Meng. Implementation of a host-to-host vpn based on udp tunnel and openvpn tap interface in java and its performance analysis. In *2013 8th International Conference on Computer Science & Education*, pages 940–943. IEEE, 2013.

[15] Oğuzhan Akyıldız and İbrahim Kök and Feyza Yıldırım Okay and Suat Özdemir. A P4-assisted task offloading scheme for Fog networks: An intelligent transportation system scenario. *Internet of Things*, 22:100695, 2023. Elsevier.

[16] Sharma, Prateek and Chaufournier, Lucas and Shenoy, Prashant and Tay, Y. C. Containers and Virtual Machines at Scale: A Comparative Study. In *Proceedings of the 17th International Middleware Conference*, Middleware '16. ACM, 2016.

[17] Arne Wendt and Thorsten Schüppstuhl. Proxying ros communications — enabling containerized ros deployments in distributed multi-host environments. In *2022 IEEE/SICE International Symposium on System Integration (SII)*, pages 265–270, 2022.