

C++



by author

Docker Container Networking. C++ client — server app.



Markus Buchholz · [Follow](#)

7 min read · May 26, 2021



4



In following article I am going to give you general overview about communication between docker containers. Verification of docker communication will be performed by running C++ client — server application and standard ‘ping’ command. Two separate docker images will be build and run.

Since I will pay attention on docker networking side, the C++ details will not be provided. There are one file and easily understandable programs. I assume also that the reader understands the main concept of docker. In this article I do not exhaust complete concept of docker networking. For further studying I recommend to use documentation, which can be found [here](#).

Docker concept in a simple terms is an open platform allowing software engineers for developing, integration, and running applications with the help of containers, build according to specification in Dockerfile.

Docker developing policy enables users to separate applications from infrastructure, utilise efficiently resources and deliver applications as built and planned.

First, I will explain and give you complete information about communicating two docker images, which can be deployed on one Host (this type of docker networking communication will use docker **BRIDGE**).

Secondly, I will display docker concept **OVERLAY NETWORK**, where docker images will be deployed on separate Hosts (and separate networks).

Communication approaches will be verified by the same client — server C++ application or command ping.

Please note, the configuration of networks (depicted in following article) have to be considered as a simple example. Normally in each case architecture of network can be organised differently and connecting many different containers.

I verified also overlaid network in the Cloud, connecting Host on cloud Linode server with my local machine (not depicted in this article since I have to publish all IP addresses).

Bridge network in Docker

Consider below image and notice that this type of communication is used when connecting containers running on **one Host**. In our example, we will run C++ client in one container and C++ server in other (program included in this article). Communication between containers is possible by supporting a docker bridge (communication pipeline), which can be associated as internal ETH network (seen only for images running on the same Host).

Containers you can build according to Dockerfiles (separately one for client and one for server). While checking container communication you need to start first server (run server container). The server will wait for the client, which you start from second container.

The client send to server random number from 0–100. Server collects received information sent from client and (as a string) add: 3.1415 and resent package to client.

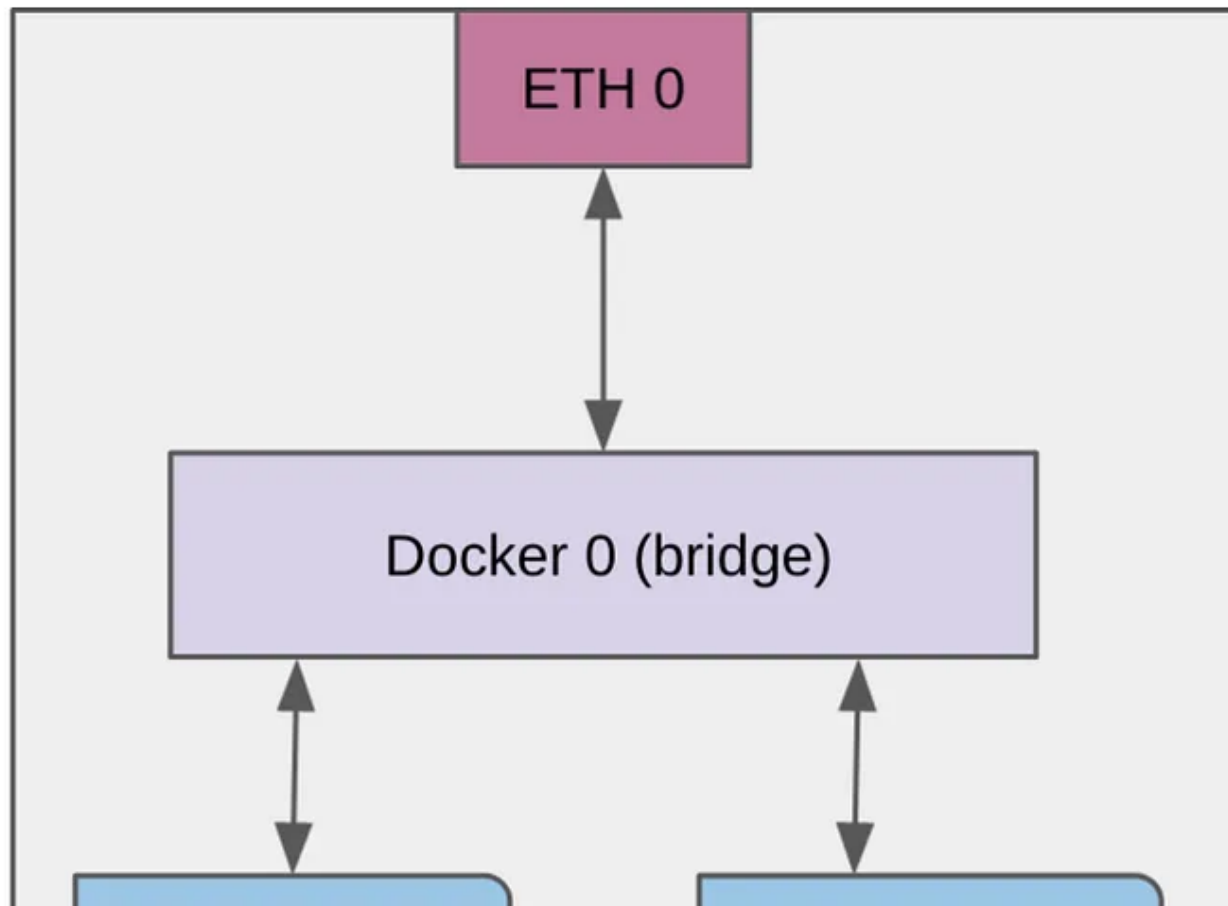
Following examples (docker images and C++ filers) represent my specific

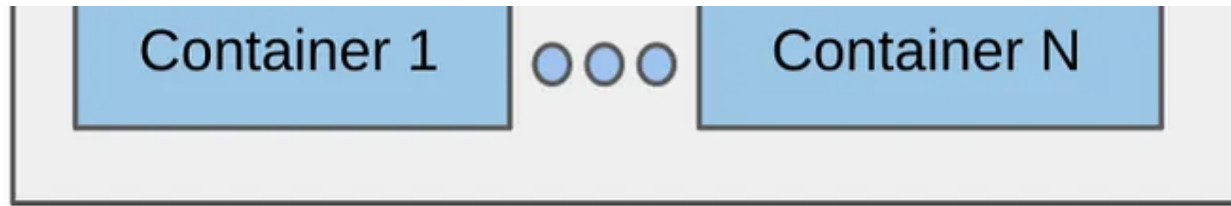
setup of SW and HW. You will have to (or not) change the IP address in client.
The port I use is 5555 (you can adjust according to you preferences).

Private bridge

(“virtual” network between Docker containers running on one Host)

Host





by author

CLIENT++

```

1
2 // Client inspired by GeeksforGeeks
3
4 #include <stdio.h>
5 #include <sys/socket.h>
6 #include <arpa/inet.h>
7 #include <unistd.h>
8 #include <string>
9 #include <string.h>
10 #include <iostream>
11 #include <stdlib.h>
12 #include <time.h>
13
14 #define PORT 5555
15
16 int main()
17 {
18     int sock = 0, valread;
19     struct sockaddr_in serv_addr;
20     srand(time(NULL));
21
22     char buffer[1024] = {0};
23     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
24     {
25
26         std::cout << "Socket creation error" << std::endl;
27         return -1;
28     }
29
30     serv_addr.sin_family = AF_INET;
31     serv_addr.sin_port = htons(PORT);
32
33     if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) //LOCAL
34     //if(inet_pton(AF_INET, "172.17.0.2", &serv_addr.sin_addr)<=0) //CLIENT DOES NOT RUN

```



```
35 //if(inet_pton(AF_INET, "172.21.0.1", &serv_addr.sin_addr)<=0) //CONTAINER
```

[Sign up](#)[Sign In](#)[Write](#)

```
40 }
41
42 if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
43 {
44     std::cout << "Connection Failed. Try again! ..." << std::endl;
45     return -1;
46 }
47
48 int number = rand() % 100;
49
50 std::cout << "check : " << number << std::endl;
51 std::string str = std::to_string(number);
52
53 char *cstr = &str[0];
54
55 send(sock, cstr, strlen(cstr), 0);
56 std::cout << "Message sent " << std::endl;
57 valread = read(sock, buffer, 1024);
58 std::cout << buffer << std::endl;
59
60 return 0;
61 }
```

medium_client.cpp hosted with ❤ by GitHub

[view raw](#)

Dockerfile client

```
FROM ubuntu:bionic #pull ubuntu
FROM gcc:latest #pull gcc

#just in case, you can install gcc and cmake

#RUN apt-get update && apt-get -y install build-essentials gcc cmake

ADD . /usr/src # add (copy) all from local folder to /usr/src

WORKDIR /usr/src

EXPOSE 5555

RUN g++ medium_client.cpp -o medium_client

# comment this and un - comment other if you would like to run your #
program manually from shell

CMD ["/usr/src/medium_client"]
#CMD ["/bin/bash"]
```

Perform following commends on terminal (build container).

```
sudo docker build . -t client:1
```

SERVER C++

```
1
2 // Server side C/C++ program to demonstrate Socket programming
3 // Server - inspired by GeeksforGeeks
4 #include <iostream>
5 #include <unistd.h>
6 #include <stdio.h>
7 #include <sys/socket.h>
8 #include <stdlib.h>
9 #include <netinet/in.h>
10 #include <string>
11 #include <string.h>
12
13 #define PORT 5555
14
15 int main()
16 {
17     int server_fd, new_socket, valread;
18     struct sockaddr_in address;
19     int opt = 1;
20     int addrlen = sizeof(address);
21
22     char buffer[1024] = {0};
23
24     if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
25     {
26         std::cout << "socket failed" << std::endl;
27         exit(EXIT_FAILURE);
28     }
29
30     if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT,
31                   &opt, sizeof(opt)))
32     {
33         std::cout << "socket failed" << std::endl;
34         exit(EXIT_FAILURE);
```

```

35     }
36     address.sin_family = AF_INET;
37     address.sin_addr.s_addr = INADDR_ANY;
38     address.sin_port = htons(PORT);
39
40     if (bind(server_fd, (struct sockaddr *)&address,
41             sizeof(address)) < 0)
42     {
43         std::cout << "bind failed" << std::endl;
44         exit(EXIT_FAILURE);
45     }
46     if (listen(server_fd, 3) < 0)
47     {
48         std::cout << "listen" << std::endl;
49         exit(EXIT_FAILURE);
50     }
51     if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
52                             (socklen_t *)&addrlen)) < 0)
53     {
54         std::cout << "accept" << std::endl;
55         exit(EXIT_FAILURE);
56     }
57
58     double pi = 3.1415;
59     std::string str1 = "server => ";
60     std::string str2 = std::to_string(pi);
61
62     valread = read(new_socket, buffer, 1024);
63     std::cout << buffer << std::endl;
64     std::string str = str1 + " : " + str2 + " and " + buffer;
65     char *cstr = &str[0];
66
67     send(new_socket, cstr, strlen(cstr), 0);
68     std::cout << "Message has been sent!" << std::endl;

```

```
68     std::cout << "message has been sent!" << std::endl;
69
70     return 0;
71 }
```

medium_server.cpp hosted with ❤ by GitHub

[view raw](#)

Server Dockerfile

```
FROM ubuntu:bionic
FROM gcc:latest

ADD . /usr/src

WORKDIR /usr/src

EXPOSE 5555

RUN g++ medium_server.cpp -o medium_server

# comment this and un - comment other if you would like to run your #
# program manually from shell

CMD ["/usr/src/medium_server"]
#CMD ["/bin/bash"]
```

Perform following command on terminal (build container).

```
sudo docker build . -t server:1
```

Perform following commands to create bridge and check communication. I assume you build before you containers(client and server.

```
// inspect available docker networks (here is my setup) - see bridge  
is running as default
```

```
sudo docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
46ef2f2901bd	bridge	bridge	local
f78ee1293783	cedalo-platform	bridge	local
1da7030bdac6	docker-nginx_backend	bridge	local
87cd39e47b26	docker-nginx_frontend	bridge	local
dcd1495544dd	docker_gwbridge	bridge	local
d0e057f5b327	home_net	bridge	local
08407441ae3d	host	host	local
56oul5xxbi5r	ingress	overlay	swarm
051116ed2b09	none	null	local
q6euxpd16iso	overnet	overlay	swarm

by author

Next, you need to inspect your bridge and capture the IP address of the bridge (container). Subnet network is the network you need to have inside

your server cpp program. If you have the other, change and re run you container build.

by author

Now you need to create your own network (bridge is a provider) and specify your name, here I use home_net.

```
sudo docker network create --driver bridge home_net  
// confirm creation running again  
sudo docker network ls
```

Your network has been created but in order to provide communication for the containers, both containers have to be connected (to network). Perform following command in order to run container and attach to you network home_net.

Please note you should build your both containers for /bin/bash (see comments in Dockerfile) in order to run your container first with the command line (without running your program which has been compiled while building a container).

```
sudo docker run -it -p 5555:5555 --network=home_net server:1
```

Next inspect again your network again. See that your container has been attached to your network.

```
sudo docker network inspect home_net
```

by author

Do the same with other container.

```
sudo docker run -it --network=home_net client:1
```

and check (run following command).

```
sudo docker network inspect home_net
```

Now your two containers are running and are connected to your network.

by author

Now, perform following commands (in your running containers). Start from server.

```
./server_medium # container with server  
./client_medium # container with client
```

Now you can see some communication between containers.

Overlay network in Docker

When the docker container you would like run is at different physical location (run on different physical Host) you can still connect your container to other containers. Here we are going to use **overlay network** concept in Docker.

You can imagine that docker containers can, similarly to bridge communication, send and receive information across separated, virtual network, called overlay network.

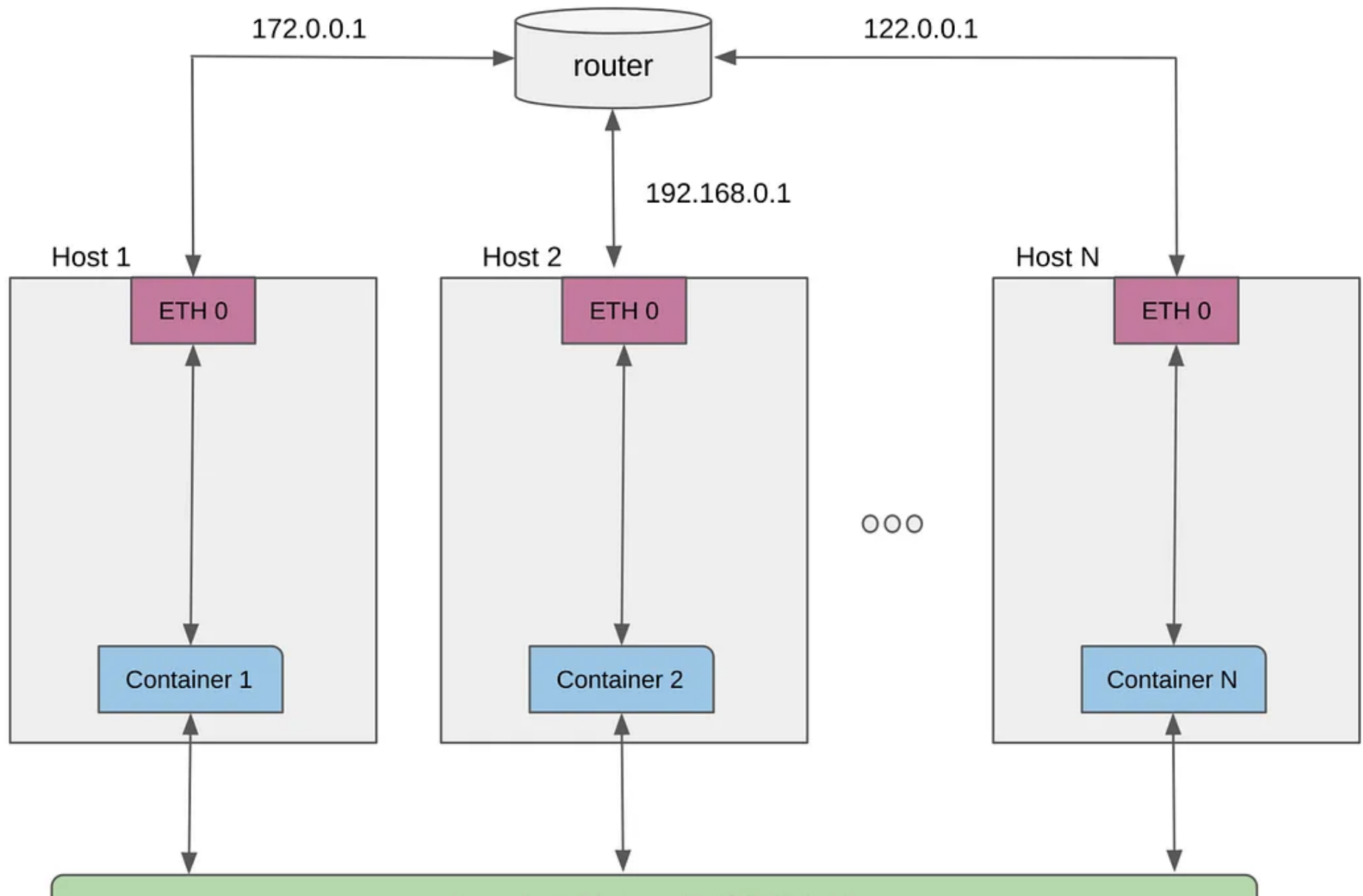
It means we can build private virtual network for our containers which can interact completely transparently. Please consider below figure. In this example we can reuse the images and C++ client-server application (the IP

addresses has to be change accordingly).

Please note, in both cases (bridge and overlay network) the communication between containers can be verified also by running command: *ping* *<host_IP_address>*. For current network (overlay network) will perform this type of test. However, the author run C++ application (client — server) in this type of docker architecture also. Regarding application was also verified while the Host server was deployed in the cloud (Linode server).

Overlay Network

("virtual" network - overlay Network between Docker containers running on different Hosts)



Overlay Network (10.0.0.1)

by author

For both cases (bridge, overlay) you have 4 possible choices to can run your application (depending on the “location” of client).

Consider below IP address specification which you need to provide for medium_client.cpp before you build client image.

```
//CLIENT AND SERVER RUN LOCAL (NOT IN CONTAINERS)
if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0)

//CLIENT DOES NOT RUN IN CONTAINER. SERVER IN CONTAINER
//if(inet_pton(AF_INET, "172.17.0.2", &serv_addr.sin_addr)<=0)

//BRIDGE COMMUNICATION
//if(inet_pton(AF_INET, "172.21.0.1", &serv_addr.sin_addr)<=0)

//OVERLAY COMMUNICATION
//if(inet_pton(AF_INET, "10.0.9.1", &serv_addr.sin_addr)<=0)
```

Communication between containers, using the overlay network will be verified by command ping, however as I mentioned before presented previously C++ application works as well.

Run following commands (#Host 1 and #Host 2)

#Host 1

First you to initialise the network manager with the IP address of Host 1 network.

```
sudo docker swarm init --advertise-addr 192.168.0.101
```

In my case I received output as follows.

```
markus@markus:~$ sudo docker swarm init --advertise-addr 192.168.0.101
Swarm initialized: current node (vyopapgrbi9rqaratsj2iu9ih) is now a manager.

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

    docker swarm join --token SWMTKN-1-4pmk157a83i2nungc6jdtmh7vp4ujqich1pho141kox1dzxeiu-c6usd1wik2kl6p3ot7fhx4pd1 192.168.0.101:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

markus@markus:~$
```

by author

#Host 2

Copy and run command:

```
docker swarm join --token SWMTKN-1-  
4pmk157a83i2nungc6jdtmh7vp4ujqich1pho141kox1dzxeiu-  
c6usd1wik2kl6p3ot7fhx4pd1 192.168.0.101:2377
```

You should receive information as follows.

```
markus@markus:~$ sudo docker swarm join --token SWMTKN-1-4pmk157a83i2nungc6jdtmh  
7vp4ujqich1pho141kox1dzxeiu-c6usd1wik2kl6p3ot7fhx4pd1 192.168.0.101:2377  
This node joined a swarm as a worker.
```

by author

#Host 1

Run following command and verify the swarm manager (one of the image above displays the swarm and bridge)

```
sudo docker network ls
```

Now create you overlay network, here our name is **overnet**.

```
sudo docker network create -d overlay overnet
```

by author

Inspect the network you created. Verify ID, etc.

```
sudo docker network inspect overnet
```

by author

Pull two Ubuntu images (replicas), activate you network and created a new service myservice. Following command send also the image for the # Host 2

```
sudo docker service create --name myservice --network overnet --  
replicas 2 ubuntu sleep
```

```
markus@markus:~$ sudo docker service create --name myservice --network overnet --replicas 2 ubuntu sleep infinity  
deme9h7m153agpf2nxl52wme4  
overall progress: 2 out of 2 tasks  
1/2: running [=====>]  
2/2: running [=====>]  
verify: Service converged
```

by author

Confirm what you have done running following commands.

```
sudo docker service ls  
sudo docker service ps myservice
```

Now confirm if you created properly overlay network and both containers are connected. The # Host 1 has IP address 10.0.1.4 .

Run command

```
sudo docker network inspect overnet
```

```
markus@markus:~$ sudo docker service ls
ID            NAME          MODE          REPLICAS  IMAGE          PORTS
deme9h7m153a  myservice     replicated    2/2        ubuntu:latest
markus@markus:~$ sudo docker service ps myservice
ID            NAME          IMAGE          NODE       DESIRED STATE  CURRENT STATE          ERROR          PORTS
pvrv9qztlyz   myservice.1   ubuntu:latest  markus     Running        Running about a minute ago
qul0m5hy9z7t  myservice.2   ubuntu:latest  markus     Running        Running about a minute ago
markus@markus:~$
```


Host 2

Run the same (I do not show image since it is similar to above). This host received address IP 10.0.1.5

```
sudo docker network inspect overnet
```



```
"Attachable": false,
"Ingress": false,
"ConfigFrom": {
  "Network": ""
},
"ConfigOnly": false,
"Containers": {
  "20b0857d223d41daf75edffd1bfcea8d5090a83090d057babef0559594940b6d": {
    "Name": "myservice.2.qul0m5hy9z7t4iedgla4dr8dz",
    "EndpointID": "e10af1d716750b5a6a1dfffb27f350ff5b4b88601afc42bc18d6d0eac5728c15a",
    "MacAddress": "[REDACTED]",
    "IPv4Address": "10.0.1.4/24",
    "IPv6Address": ""
  },
  "lb-overnet": {
    "Name": "overnet-endpoint",
    "EndpointID": "7d3a272848cef6a6f93631e3ef339bb4ab54d3290a3ebbad094be6fba8013f64",
    "MacAddress": "[REDACTED]",
    "IPv4Address": "10.0.1.6/24",
    "IPv6Address": ""
  }
},
"Options": {
  "com.docker.network.driver.overlay.vxlanid_list": "4097"
},
"Labels": {},
"Peers": [
  {
    "Name": "a64ac2a7e965",
    "IP": "192.168.0.101"
  },
  {
    "Name": "d8ed9140db84",
    "IP": "192.168.0.106"
  }
]
```



by author

#Host 1 and #Host 2

Run following command to install 'ping command'.

```
apt-get update && apt-get install -y iputils-ping
```

On both #Host 1 and 2 run command to capture name of container

```
sudo docker ps
```

and connect to docker shell in order to run ping

```
sudo docker exec -it 94835734987 sh
ping 10.0.1.5 # from host 1
ping 10.0.1.4 # from host 2
```

Thank you for reading.

Docker

Cpp

Programming




Written by Markus Buchholz

465 Followers

Follow



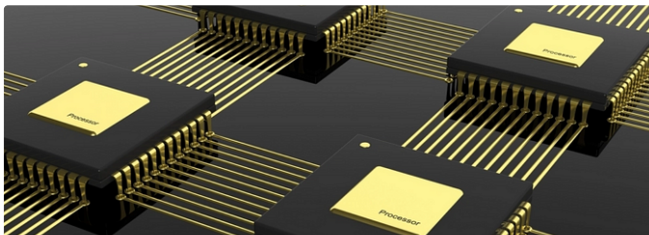
More from Markus Buchholz


 Markus Buchholz

Eigen Value Problem For Coupled Oscillators. Simulation in C++...


The study of oscillator systems is crucial in understanding the dynamic behavior of...

5 min read · May 22



 Markus Buchholz

Parallel Computation in C++

 Markus Buchholz in Geek Culture

Digital Filter Design in Python and C++

In following article I will demonstrate a general approach of digital filters design. Th...

13 min read · Sep 4, 2021



 Markus Buchholz in Geek Culture

2D and 3D Bézier Curves in C++

INTRODUCTION

Often we consider the application we often think about performance, meaning how fast...

4 min read · Aug 28, 2021


8 min read · Feb 4

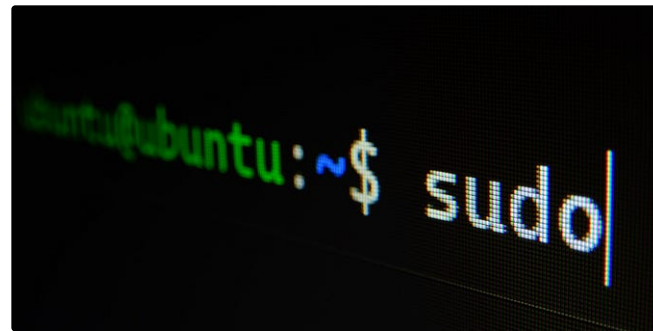



See all from Markus Buchholz

Recommended from Medium



 Dimitris Pouloupoulos in Towards Data Science



 Sung Kim in Geek Culture

The Power of Linux Cgroups: How Containers Take Control of Their...

Optimizing Container Resource Allocation with Linux Control Groups

★ · 8 min read · Jan 10

👏 45 💬 1



Enable SSH Access to WSL from a Remote Computer

Setup SSH Server on Windows Subsystems for Linux (Ubuntu) on Windows 11 and Enabl...

★ · 9 min read · Jan 4

👏 84 💬 5



Lists



Stories to Help You Grow as a Software Developer

19 stories · 115 saves



Leadership

30 stories · 54 saves



How to Run More Meaningful 1:1 Meetings

11 stories · 48 saves



Stories to Help You Level-Up at Work

19 stories · 95 saves





 Tate Galbraith in DevOps.dev

Using Private Repositories With Docker Compose & BuildKit

Put those SSH keys to work inside your images

✦ · 3 min read · Feb 16




Love Sharma in Dev Genius

System Design Blueprint: The Ultimate Guide

Developing a robust, scalable, and efficient system can be daunting. However,...

✦ · 9 min read · Apr 20



 Martin Heinz in Better Programming

The Right Way to Run Shell Commands From Python

These are all the options you have in Python for running other processes—the bad, the...

✦ · 7 min read · Jun 5



○ Carlos Aldea

5 Reasons To Become an Embedded SW Developer

And keep yourself as far as possible from web development

★ · 2 min read · Jan 13



[See more recommendations](#)