North American Journal of Engineering and Research

Est. 2020

# Eliminate the Noisy Neighbor Problem in Docker using Resource Limits

**Pallavi Priya Patharlagadda**

*Email: Pallavipriya527.p@gmail.com*

## Abstract

Container technologies are highly popular these days. It is evident that a wide range of applications are operating in containers. Its increased computing usage due to resource sharing and/or isolation is one of the main causes for its widespread adoption. Sharing resources is important, but a careless setup can result in a noisy neighbor scenario. A noisy neighbor scenario in computing is when a process or collection of processes uses up too many resources on a host. This condition has an adverse effect on other processes that are operating on the same host. In the end, the host's low resources cause it to become less responsive, which lowers the performance of the entire system. This paper will go over how to reduce the noisy neighbor issue in Docker by setting memory and CPU restrictions.

## Introduction

It's now really simple to deploy new containers. Scaling becomes considerably simpler once containers are up and running. A single keystroke can double or even triple the number of containers that are running, but can your container infrastructure support it?

similar to vehicles traveling on a freeway. Each car is unique in terms of its size, performance, and shape. The amount of space on the road for cars is restricted. In response to demand, freeways cannot yet expand or contract in size. A bottleneck will eventually form when there are more cars on a freeway. For this reason, laws like those prohibiting halting and setting speed limits are in place to aid in traffic flow. The container infrastructure is generally not as dynamic as the containers themselves. It is possible to easily stretch the container infrastructure's computational capacity. The industry is averaging 10 containers per host, with some installations having up to 95 containers on a host, according to the Sysdig annual container survey. Whoa! It's normal to have some troublemakers in the mix, even if the total number of containers stays the same. For instance, some programs gradually increase their memory and CPU usage. This is usually acceptable, but what would happen if every container in your entire infrastructure began consuming more resources? In the subsequent sections, let's look at how to solve the problem of heavy resource consumption.

## All about Docker:

Docker is an open platform for application development, delivery, and execution. Docker allows you to rapidly release software by separating your apps from your infrastructure. You can use Docker to manage your infrastructure in the same manner that you do your apps. You may cut down on the amount of time it takes between writing code and putting it into production by utilizing Docker's shipping, testing, and deployment processes.

With Docker, you can bundle and execute an application in a container—a loosely isolated environment. You can execute multiple containers concurrently on a single host thanks to the isolation and security. You can use containers instead of depending on what is installed on the host because they are lightweight and come with everything required to run the application. While working, you can share containers and ensure that each person you share with gets the identical container that functions uniformly.

Docker gives you the tools and a platform to control your containers' lifecycle:

- Use containers to develop your application and its supporting components.
- The container serves as the distribution and testing hub for your program.
- When you're prepared, launch your application as an orchestrated service or in a container in your production environment. Regardless of whether your production environment is a cloud provider, a local data center, or a combination of the two, this operates in the same way.

**Docker uses**:

Delivering your applications in a timely and reliable manner:

By enabling developers to operate in standardized settings with local containers that host their applications and services,

Docker simplifies the development lifecycle. Workflows involving continuous integration and continuous delivery (CI/CD) benefit greatly from containers.

Look at the following example situation:

- Using Docker containers, your engineers collaborate with one another while writing code locally.
- They execute both automatic and manual tests on their applications by pushing them into a test environment using Docker.
- In order to verify and validate their fixes, developers can redeploy their fixed code to the test environment after fixing it in the development environment.
- After testing, it's just a matter of uploading the revised image to the production environment to get the patch to the customer.

Responsive deployment and scaling:

Highly portable workloads are possible using Docker's container-based technology. A developer's laptop, real or virtual machines in a data center, cloud providers, or a combination of environments can all execute Docker containers.

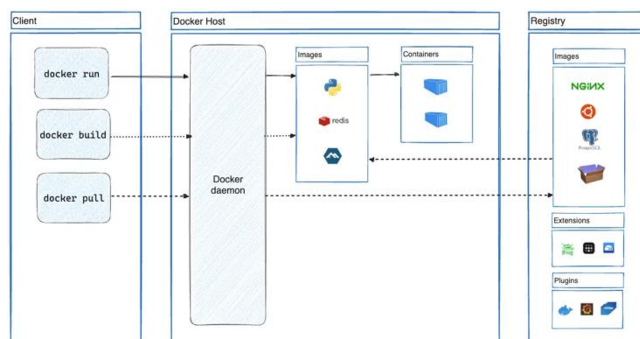
Because of its lightweight design and mobility, Docker also makes it simple to manage workloads dynamically, quickly scaling up or down services and applications based on business requirements.

Running more workloads on the same hardware:

Docker is quick and light-weight. It offers a practical, affordable substitute for virtual machines based on hypervisors, allowing you to utilize more of your server's capacity to meet your company's objectives. When you need to accomplish more with fewer resources in small and medium deployments and high density situations, Docker is ideal.

Docker architecture

The architecture of Docker is client-server based. The Docker daemon, which builds, manages, and distributes your Docker containers, is the one with whom the Docker client communicates. It is possible for the Docker client and daemon to operate simultaneously on the same machine or to link a Docker client to a remote Docker daemon. UNIX sockets, a network interface, or a REST API are the ways in which the Docker client and daemon exchange information. Docker Compose is an additional Docker client that facilitates the manipulation of applications composed of many containers.




The Docker daemon

In addition to managing Docker objects like images, containers, networks, and volumes, the Docker daemon, or (dockerd), is in charge of listening for Docker API calls. To manage Docker services, a daemon can converse with other daemons.

The Docker client

For many Docker users, the main interface to interact with Docker is the Docker client (docker). The client transmits commands to dockerd, which executes them, when you use commands like docker run. Utilizing the Docker API is the docker command. Multiple daemons can be in communication with the Docker client.

Docker Desktop

You can create and distribute containerized apps and microservices with Docker Desktop, an easy-to-install program for Linux, Windows, and Mac. Together with Docker Compose, Docker Content Trust, Kubernetes, Credential Helper, and the Docker daemon (dockerd), Docker Desktop also contains the Docker client (docker).

Docker registries

Docker images are kept in a registry. Docker searches Docker Hub by default for images, and Docker Hub is a publicly accessible registry. Even managing your own private registry is possible.

When you use the docker pull or docker run commands, Docker pulls the required images from your configured registry. When you use the docker push command, Docker pushes your image to your configured registry.

Docker objects

Creating and utilizing images, containers, networks, volumes, plugins, and other things is what Docker is all about. Here's a quick rundown of a few of those items.

Images

An image is a read-only template that contains Docker container creation instructions. Images are frequently modified versions of one another. You may, for instance, create an image that is based on the Ubuntu image and includes your application and the Apache web server installed, together with the configuration information required for your application to function.

You may make your own pictures or you might limit yourself to using only those that other people have made and uploaded to a register. You construct a Dockerfile with straightforward syntax to define the procedures required to produce and run your own image. A Dockerfile's instructions each generate a layer in the image. Rebuilding the image after making changes to the Dockerfile only affects the modified layers. This contributes to the lightweight, small size, and speed of pictures in comparison to other virtualization systems.

Containers

An image's executable instance is known as a container. The Docker API and CLI allow you to create, stop, transfer, and destroy containers. Not only can you attach storage to a container, but you can also link it to one or more networks and use its existing state to produce a new image.

A container's default level of isolation from its host computer and other containers is rather high. The degree of isolation between a container's network, storage, and other underlying subsystems and the host computer or other containers is something you can adjust.

An image and any configuration variables you give it during creation or startup define a container. Any modifications made to a container's state that aren't kept in persistent storage vanish when it is deleted.

Now that we've learned about Docker, let's look at how to mitigate the noisy neighbor situation,

1. Creating Container Without Limits:

Docker does not impose any resource limitations on a container by default. It gives the container access to every resource on the system. Let's use an example to better grasp this.

Start by displaying the host machine's CPU and memory configuration:

```
$ docker info | grep -iE "CPUs|Memory"
CPUs: 4
Total Memory: 7.714GiB
```

The output shown above indicates that the host machine has 4 CPUs and 7.714 GiB of memory.

Now, let's create a new container and use docker stats command to view its resource limits:

```
$ docker container run --rm -it -d --name example nginx:alpine

$ docker stats example --no-stream --format "{{ json . }}" | python3 -m json.tool
```


```
{
"BlockIO": "0B / 12.3kB",
"CPUPerc": "0.00%",
"Container": "web-server",
"ID": "2def8ff5e138",
"MemPerc": "0.05%",
"MemUsage": "3.734MiB / 7.714GiB",
"Name": "example",
"NetIO": "5.46kB / 0B",
"PIDs": "5"
}
```


The memory limit, 7.714 GiB, is the same as the host's RAM, as the result above indicates. You can see that in the MemUsage box.

To print the result in a nice format in this example, it is sent to the Python interpreter.

2. Setting Resources Limits when Creating a Container:

We can construct a container with Docker in the following ways:

The container run child command

The docker-compose.

You can use either of these options to impose resource restrictions in Docker.

2.1. Using the container, run the child command.

Setting resource limitations with the container run child command is the easiest method. We can specify memory and CPU limits with this command.

2.1.1. Setting Memory Limits

Use the container run child command with the --memory argument to set a hard memory restriction. Following the setting of this limit, Docker prevents a container from using more RAM, either system or user.

```
$ docker container run --rm -it -d --name test-mem-limit --memory=512m nginx:alpine
```

Now, let's check its memory limits:

```
$ docker stats test-mem-limit --no-stream --format "{{ json . }}" | python3 -m json.tool
```


```
{
"BlockIO": "0B / 12.3kB",
"CPUPerc": "0.00%",
"Container": "test-mem-limit",
"ID": "b46befb6d196",
"MemPerc": "1.45%",
"MemUsage": "3.711MiB / 512MiB",
"Name": "test-mem-limit",
"NetIO": "3.83kB / 0B",
"PIDs": "5"
```

}

We can observe that Docker has imposed a 512 MiB RAM limit in the output above. That's what the MemUsage field says. A positive integer may be entered as an input for the –memory option, followed by the suffixes b, k, m, or g to represent bytes, kilobytes, megabytes, or gigabytes, respectively

Furthermore, we can create soft memory restrictions with Docker. If the kernel doesn't detect memory contention, This option lets a container utilize as much memory as it needs. Use the container run child command with the –memory-reservation argument to set soft memory limits:

$ docker container run --rm -it -d --name soft-mem-limit --memory=2g --memory-reservation=512m nginx:alpine

The hard memory constraints in this example are set to 2GiB, while the soft memory limits are set at 512MiB.

Note: It is important to remember that soft memory limitations need to be lower than hard memory restrictions. as there is no assurance that the container won't surpass it and it is a soft limit.

Let's now confirm the container's hard memory limits:

```
$ docker stats soft-mem-limit --no-stream --format "{{ json . }}" | python3 -m json.tool
{
  "BlockIO": "135kB / 12.3kB",
  "CPUPerc": "0.00%",
  "Container": "soft-mem-limit",
  "ID": "9b748ec04b2a",
  "MemPerc": "0.38%",
  "MemUsage": "3.863MiB / 2GiB",
  "Name": "soft-mem-limit",
  "NetIO": "3.98kB / 0B",
  "PIDs": "5"
}
```

There is no command in Docker to display the soft memory constraints. But these specifics are available in the cgroups.

Let's enter the container and make a list of the contents of /sys/fs/cgroup/memory.minimal file:

```
$ docker exec -it soft-mem-limit sh
# cat /sys/fs/cgroup/memory.low
536870912
# exit
```

There is an integer value in this file. It has the same 512 MiB byte representation as the soft limitations.

2.1.2. Setting CPU Limits

By default, containers have unrestricted access to the host computer's CPU cycles. Use the container run command with the –cpus argument to restrict it.

$ docker container run --rm -it -d --name test-cpu-limit --cpus=2 nginx:alpine

Docker ensures that a container running with this configuration can only use two CPU at a time.

We may also establish limitations on a certain CPU with Docker. Use the –cpuset-cpus option to do this:

$ docker container run --rm -it -d --name test-cpu-set --cpus=1 --cpuset-cpus=2 nginx:alpine

We have placed restrictions on the third CPU in this case.

Let's now enter the container and list the contents of the /sys/fs/cgroup/cpuset.cpus file in order to confirm this:

```
$ docker exec -it test-cpu-sets sh
# cat /sys/fs/cgroup/cpuset.cpus
2
# exit
```

There is an integer value in this file. It stands for a CPU figure. It is significant to remember that the CPU numbering begins at 0.

2.2. Using docker-compose

We can also use docker-compose to define resource restrictions using Docker. Let's use an example to better grasp this.

Start by generating a configuration file called docker-compose.yml.

```
$ cat docker-compose.yml
version: "3.9"
services:
  web-server:
    image: nginx:alpine
    container_name: compose-res-limits
    mem_limit: "2g"
    mem_reservation: "512m"
    cpus: "1"
  cpuset: "2"
```

In this setup:

The term "mem_limit" refers to the hard memory restrictions. 2 GiB is the value we have set for mem_reservation, which stands for the soft memory restrictions. The maximum CPU amount, which is

512MiB, is what we have set. As it reflects the maximum on a particular CPU, we have it set to 1. It is now running on the third CPU.

Now let's implement the setup to establish a container with the given boundaries:

```
$ docker-compose up -d
```

Let's now examine the container's CPU and RAM limits:

```
$ docker stats compose-res-limits --no-stream --format "{{ json . }}" | python3 -m json.tool
{
  "BlockIO": "0B / 12.3kB",
  "CPUPerc": "0.00%",
  "Container": "compose-res-limits",
  "ID": "0d7055f8eb31",
  "MemPerc": "0.18%",
  "MemUsage": "1.836MiB / 2GiB",
  "Name": "compose-res-limits",
  "NetIO": "11.2kB / 0B",
  "PIDs": "2"
}
$ docker exec -it compose-res-limits sh
# cat /sys/fs/cgroup/memory.low
536870912
# cat /sys/fs/cgroup/cpuset.cpus
2
# exit
```

The right setting of CPU and RAM limits via docker-compose is evident here.

3. Setting Resources Limits on a Running Container

In the previous section, we defined resource limitations using the commands container run and docker-compose. Nevertheless, a significant limitation of these instructions is their inability to provide dynamic limit updates. It implies that when building a container, we must declare every limit. The container run command's inability to put restrictions on more than one container is another drawback.

Use the container update command to get around all of these restrictions. This command modifies the various containers' configurations dynamically. So long as the containers are operating, we can use it.

First, establish a container with no resource restrictions in order to comprehend this:

```
$ docker container run --rm -it -d --name test-no-limits nginx:alpine
```

Let's now use the container update command to set the CPU and RAM limits:

```
$ docker update --memory=2g --memory-reservation=512m --cpus=1 --cpuset-cpus=2 --memory-swap -1 test-no-limits
```

We've used the –memory-swap -1 option in this case. Up to the amount of memory on the host system, this option raises the swap memory limit. Without this choice, the command produced the following error:

Error response from daemon: Cannot update container: Memory limit should be smaller than already set memoryswap limit, update the memory swap at the same time

Let's confirm that the container's CPU and RAM limitations were changed correctly:

```
$ docker stats test-no-limits --no-stream --format "{{ json . }}" | python3 -m json.tool
{
  "BlockIO": "0B / 12.3kB",
  "CPUPerc": "0.00%",
  "Container": "test-no-limits",
  "ID": "ef6bd714038a",
  "MemPerc": "0.36%",
  "MemUsage": "3.691MiB / 2GiB",
  "Name": "test-no-limits",
  "NetIO": "5.29kB / 0B",
  "PIDs": "5"
}
$ docker exec -it test-no-limits sh
# cat /sys/fs/cgroup/memory.low
536870912
# cat /sys/fs/cgroup/cpuset.cpus
2
# exit
```

## Conclusion

Specifying Docker container CPU limitations is equally important as specifying memory limits. Luckily, Docker Desktop has global restrictions that stop users from running too many Docker Images on their PC and completely overloading it. Nonetheless, I believe it is best practice to explicitly define resource limitations and to treat your local

development workstation the same as your Kubernetes cluster or cloud vm.

## References


[1] https://howtodoinjava.com/devops/docker-memory-and-cpu-limits/

[2] https://docs.docker.com/guides/docker-overview/

[3] https://www.56k.cloud/en/blog/put-the-brakes-on-docker-containers

[4] https://www.thorsten-hans.com/docker-container-cpu-limits-explained/

[5] https://faun.pub/understanding-docker-container-memory-limit-behavior-41add155236c