Running 1,000 Containers in Docker Swarm

Last updated: 2017-05-08

by Tit Petric | 4 Comments

Development

Reading Time: 9 minutes

Docker has been touted as the holy grail of on-premises software container solutions. Docker Swarm is the orchestration upgrade that allows you to scale your containers from a single host to many hosts, and from tens of containers into thousands of them.

But does it deliver on that promise? Let's run 1,000 containers to find out.

> "Does Docker Swarm deliver on its orchestration upgrade promises?" via @titpetric
>
> CLICK TO TWEET

Prepare the Network

I'm starting with a three-node cluster, running the latest version of Docker.

```
root@swarm1:~# docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
2jrjw0f0t6k2hbf2w0d41db2x *  swarm1    Ready   Active        Reachable
aet822iruebj35gu4vlkn70bb    swarm3    Ready   Active        Leader
l4bdxu989ktteb61cte66dqbc    swarm2    Ready   Active        Reachable
```

We will need to set up a multihost network. This network is the common communication channel between the containers, or any other containers. For example, if you would run a webserver and a database, you would put them on the same network, so they can talk to each other.

> *Using Docker Engine running in Swarm mode, you can create an overlay network on a manager node. The swarm makes the overlay network available only to nodes in the swarm that require it for a service. When you create a service that uses an overlay network, the manager node automatically extends the overlay network to nodes that run service tasks.*

In order for containers to be discoverable between hosts in a Docker swarm, we will need to create an `overlay` network on one of the manager nodes.

```
# docker network create \
  --driver overlay \
  --subnet 10.0.0.0/20 \
  --attachable \
  party-swarm
```

Since we can use a higher number of containers, we can declare a larger number of IPs available, using the CIDR notation. A bit mask with 20 bits leaves 12 bits available for IPs — 4,096 of them can be run on this network.

The switch `--attachable` creates this network in a way that may be used by non-swarm containers. This provides access to the network from the classic `docker run` way of running your containers.

Prepare the Host

Due to the very large number of containers, which means an even larger number of system processes on your Docker host, we will need to tune some settings in order to avoid instability when running our service at such a scale.

Various Linux subsystems, especially in regards to networking, have many exposed configuration flags that are managed by sysctl. The

defaults of these flags are more suited for laptops than servers that would run thousands of processes.

ARP cache

When I first tried to run 1,000 containers in a service, I hit a limitation that resulted in the Docker swarm becoming unresponsive, with occasional connectivity interruptions. When inspecting this, I found many messages like the following one in output of `dmesg`.

```
neighbour: arp_cache: neighbor table overflow!
```

The ARP cache is a look-up table that connects IP addresses (OSI Layer 3) to MAC addresses (OSI Layer 2). When we're running 1,000 containers and assigning them IPs, we will inevitably declare a large number of IP/MAC address pairs. The issue with connectivity interruptions was caused by hitting the ARP cache limits, where garbage collection is performed.

Keep in mind, when we start a service, the service will attach to three networks:

- `party-swarm` – the shared network we set up
- `ingress` – where traffic comes from; we will forward a port to the service
- `docker_gwbridge` – the public internet access network

Some of these networks will contain multiple IP addresses in each container. On my count, it means about 5,000 network addresses allocated for a 1,000 containers.

A bit of digging around led me to some sysctl tunable settings which can be used to work around the problem, basically by increasing the thresholds where ARP cache collection occurs.

Issue this on every Docker swarm host:

```
sysctl -w net.ipv4.neigh.default.gc_thresh1=8096
sysctl -w net.ipv4.neigh.default.gc_thresh2=12288
sysctl -w net.ipv4.neigh.default.gc_thresh3=16384
```

And this is what the values do:

- `gc_thresh1` – the minimum number of entries to keep in the ARP cache
- `gc_thresh2` – the soft maximum number of entries to keep in the ARP cache
- `gc_thresh3` – the hard maximum number of entries to keep in the ARP cache

The operative value is `gc_thresh2`, as garbage collection there happens five seconds after the ARP cache crosses this many entries. The default value was 512, which caused very frequent garbage collection runs and related system instability.

The reason why I chose this value is that I needed a sane default that would not be hit. Generally the recommendation is that you should multiply all the `gc_thresh*` values by 2 until your system will behave nicely. As is the case most of the time, the sysctl tunable settings are very poorly documented, so figuring out what kind of impact they have in terms of memory use or other pitfalls is possible only in the rarest occasions.

Sysctl tunables

There are other sysctl tunable settings which might be useful for you:

```
# Have a larger connection range available
net.ipv4.ip_local_port_range=1024 65000

# Reuse closed sockets faster
net.ipv4.tcp_tw_reuse=1
net.ipv4.tcp_fin_timeout=15

# The maximum number of "backlogged sockets".  Default is 128.
net.core.somaxconn=4096
net.core.netdev_max_backlog=4096

# 16MB per socket - which sounds like a lot,
# but will virtually never consume that much
```

```
# but will virtually never consume that much.
net.core.rmem_max=16777216
net.core.wmem_max=16777216

# Various network tunables
net.ipv4.tcp_max_syn_backlog=20480
net.ipv4.tcp_max_tw_buckets=400000
net.ipv4.tcp_no_metrics_save=1
net.ipv4.tcp_rmem=4096 87380 16777216
net.ipv4.tcp_syn_retries=2
net.ipv4.tcp_synack_retries=2
net.ipv4.tcp_wmem=4096 65536 16777216
#vm.min_free_kbytes=65536

# Connection tracking to prevent dropped connections (usually issue on LBs)
net.netfilter.nf_conntrack_max=262144
net.ipv4.netfilter.ip_conntrack_generic_timeout=120
net.netfilter.nf_conntrack_tcp_timeout_established=86400

# ARP cache settings for a highly loaded docker swarm
net.ipv4.neigh.default.gc_thresh1=8096
net.ipv4.neigh.default.gc_thresh2=12288
net.ipv4.neigh.default.gc_thresh3=16384
```

These sysctl settings deal mainly with how many connections can be open at a given time, which ports are used for the connections, how fast connections are recycled… I can't imagine a high-traffic, load-balancer host running without at least some of these.

Check Resource Usage

For our purposes, my Swarm cluster has three nodes, which each have six CPU cores and 8GB of RAM. I have a bit more than 100GB of disk space available, but if you're following along, 10GB of available disk space will be enough to run 1,000 containers.

Calculating needed disk space

In order to calculate the theoretical capacity of your system for a single Docker container, we can look at the image size. A Docker image is created as a composite of many layers.

```
# docker history titpetric/sonyflake
IMAGE          INFO                            SIZE
af387743bcd5   ENTRYPOINT ["/sonyflake"]        0 B
fdf6859f28f4   ADD file:62a837b350878b1d75      6.107 MB
2a13f6fd8e6c   MAINTAINER Tit Petric <blac      0 B
88e169ea8f46   ADD file:92ab746eb22dd3ed2b      3.98 MB
```

For example, the image I'm using uses `alpine:3.5` as a base (4MB) and adds a compiled application which is about 6MB in size. The application is an ID generator written in Go (source code available on titpetric/sonyflake GitHub).

```
capacity = available resources (10GB) / size of a container (10MB)
```

Using this simple calculation, we can estimate that we can run about 1,000 containers on a single host with 10GB of available disk space.

Calculating memory use

Memory use is a bit harder to calculate, because it can vary from the state when your application just starts (cold) or after the state when your application is serving real traffic for a while. The basic calculation is pretty much the same, but I do caution that you keep some buffer when doing calculations with more fluid resources.

```
# docker stats sonyflake
CONTAINER   MEM USAGE / LIMIT
sonyflake   5.273 MiB / 7.792 GiB
```

I want to reserve about 20 percent of memory for other uses, so I assume that I only have 6GB of available resources. I round up the memory usage of the container as well.

```
capacity = available resources (6GB) / size of a container (6MB)
```

It just so happens that we can comfortably run 1,000 containers on a single host because of the low memory footprint of the sonyflake app.

Calculating CPU use

You don't. You should use a monitoring system that will monitor CPU usage of your hosts and containers. There are some free and cool tools you can use. For example, ctop gives you a real-time view of your Docker containers on a single host. Netdata provides an overview of real-time data as well as limited historical data and alerting.

You can also resort to benchmarks, with software like wrk or siege, to estimate what kind of capacity you can expect with your setup.

Running 1,000 Containers

After we get all the configuration and tuning out of the way, we're at a point where we can run our service and create 1,000 instances. For this example, I've chosen the sony/sonyflake ID generator and prepared an HTTP server following 12-factor app guidelines. The source code for the HTTP server is on titpetric/sonyflake GitHub.

```
# docker service create \
>       --replicas 1000 \
>       --network party-swarm \
>       --update-parallelism 5 \
>       --name sonyflake \
>       -p 80:80 titpetric/sonyflake
je2np5ab1s6ztf55qj62im72g
```

It takes about ten minutes to start up all the containers. You can inspect the service as it's starting up containers:

```
# docker service ls
ID            NAME       MODE         REPLICAS  IMAGE
je2np5ab1s6z  sonyflake  replicated  371/1000  titpetric/sonyflake:latest
```

The service is already usable as soon as one container is online.

```
# curl -s http://127.0.0.1
{"id":134286618554008122,"machine-id":570,"msb":0,"sequence":0,"time":8004106197}
```

After all the containers have started, there will be a nice 1000/1000 replicas column in the output of `docker service ls`.

```
# docker service ls
ID            NAME       MODE         REPLICAS   IMAGE
je2np5ab1s6z  sonyflake  replicated  1000/1000  titpetric/sonyflake:latest
```

Achievement unlocked!

Let's give it a kick with a benchmark tool just to see that it works. As wrk is also available as a Docker container, I'll be running it to attach to the custom network which we created and to benchmark the `sonyflake` service.

```
# docker run --net=party-swarm --rm williamyeh/wrk -t 6 -c 30 http://sonyflake
Running 10s test @ http://sonyflake
  6 threads and 30 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     2.82ms    5.26ms 104.23ms   89.64%
    Req/Sec     3.48k     1.31k    7.92k    67.57%
  207348 requests in 10.07s, 40.36MB read
  Socket errors: connect 0, read 0, write 0, timeout 37
Requests/sec:  20598.27
Transfer/sec:      4.01MB
```

I'm feeling pretty nervous, so I'll also scale down the service to 30 containers, wait about ten minutes until it completes that, and rerun the benchmark.

```
# docker service scale sonyflake=30
sonyflake scaled to 30
```

You can inspect the state of your containers by issuing something like this:

```
# docker ps -a --format "{{.Status}}"  | sort | uniq -c
      1 Created
    164 Removal In Progress
      2 Up 11 minutes
      1 Up 12 minutes
      2 Up 13 minutes
      2 Up 7 minutes
      3 Up 9 minutes
```

When "Removal in Progress" drops to 0 on all hosts, it should be okay to rerun the benchmark.

```
# docker run --net=party-swarm --rm williamyeh/wrk -t 6 -c 30 http://sonyflake
Running 10s test @ http://sonyflake
  6 threads and 30 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     0.89ms    1.37ms  37.99ms   93.22%
    Req/Sec     5.69k     1.83k    9.61k    60.83%
  340455 requests in 10.04s, 66.30MB read
  Socket errors: connect 0, read 0, write 0, timeout 46
Requests/sec:  33904.03
Transfer/sec:      6.60MB
```

Here we can see the effect that process scheduling on Linux has on containers. When we run 1,000 containers, it means that the operating system has to divide all available time between them. Figuring out which process needs CPU time is a completely different problem when you're running 1,000 containers or when you're running only 30.

This is why the benchmark for fewer containers is faster. But consider that increasing containers from 30 to 1,000 containers is a 30x increase, and the penalty is still just 30 percent of your total capacity. Amazing.

Conclusion

Running a service in Docker Swarm feels very natural and simple. Scaling to 1,000 containers was a fun challenge, with honestly not very many obstacles.

Of course, it very much depends on what kind of application you are scaling. If you follow the 12-factor application guidelines, it makes it very easy to go from 1 to 1,000 containers, but as soon as you're thinking of adding on another thousand containers, some systems settings should be revisited.

If you need some `sysctl` settings inside your container — for example, if you have a high-traffic reverse proxy or Redis recommends some settings for you — it seems that you need to stick with `docker run` for a while longer.