How to Scale Docker Swarm Horizontally in Production

Link: https://betterstack.com/community/guides/scaling-docker/horizontally-scaling-swarm/

Auto ScalingDockerHigh Availability

Cristovao Cordeiro
Updated on January 9, 2024

Contents

With any software service deployment, there might come a time when you need to think about scaling your service due to meet changing demands. Such an action may be triggered by multiple factors including the following:

- Your service is under heavy load which can ultimately impact its performance. Therefore, you'll want to add more instances of the service throughout the cluster, such that the incoming load can be re-distributed evenly.
- Your service does not have enough instances running throughout multiple availability zones of your cluster. This means it might be subject to a single point of failure which compromises its availability if one or more nodes in the cluster go down. Therefore, you'll want to add more instances to your service, distributed more equally throughout the cluster to increase its redundancy.
- Your service is experiencing a low demand from your users, which means you might be hosting and maintaining more instances of your service than what is needed to fulfill the demand. Therefore, you'll want to decrease the number of active service instances to reduce your maintenance efforts and compute costs.

In this tutorial, you will learn how to horizontally scale in/out an existing Docker service, and you'll also experiment with scalability constraints and auto scaling solutions. For future reference, when we talk about *Scaling In* and *Scaling Out*, we mean the following:

- **Scaling In**: removing instances from an existing service.
- **Scaling Out**: adding more instances in parallel to spread out the service load.

Prerequisites

Before proceeding with this article, ensure that you've read and performed all the steps from the previous tutorial on Setting up Docker Swarm High Availability in Production. This means you should have four replicas of an NGINX service running on a 5-node cluster with three managers and two workers where all managers have been drained:

```
docker node ls
```

Output

```
ID                            HOSTNAME    STATUS    AVAILABILITY   MANAGER STATUS   ENGINE
VERSION
9r83zto8qpqiazt6slxfkjypq     manager-1   Ready     Drain          Reachable        20.10.18
uspt9qwqnzqwl78gbxc7omja7 *   manager-2   Ready     Drain          Leader           20.10.18
txrdxwuwjpg5jjfer3bcmtc5r     manager-3   Ready     Drain          Reachable        20.10.18
kaq8r9gec4t58yc9oh3dc0r2d     worker-1    Ready     Active                          20.10.18
vk1224zd81xcihgm1iis2703z     worker-2    Ready     Active                          20.10.18
```

We are currently working on the `manager-2` node, which is the leader and is not schedulable (as it was drained in the previous tutorial). Go ahead and execute the following command to see the active services on the cluster:

```
docker service ls
```

Output

```
ID             NAME          MODE         REPLICAS   IMAGE          PORTS
euqfhji2qpco   nginx_nginx   replicated   4/4        nginx:latest   *:8088->80/tcp
```

Currently, there is only one service with four replicas in the entire cluster. For now, ensure that you are connected to the Leader node (`manager-2` in this example). If you do not have access to a Docker cluster as required to follow this tutorial, we recommend you use the online Docker playground.Getting started

In this tutorial, we'll mainly use the `docker service` management command to perform all the service inspections and scalability actions. Knowledge of this command is crucial when scaling your services since it allows you to perform the following actions through various subcommands:

- `inspect` all the information about your service, including its previous and current states.
- `list` (or `ls`) all the services running in your cluster.
- list (`ps`) all the tasks within a service. Each task will correspond to a replica of your service, so this information is critical when scaling in/out.
- `update` a service's specification (i.e., its placement policies, number of replicas, etc.).
- `rollback` a service to its previous specification.
- `scale` one or more services concurrently, in, or out, which is pretty much the same as running a service `update` to change the number of service replicas.

To see these subcommands in action, let's take on a 5-step scenario where we'll analyze our current cluster and services, and then scale in and out a service while also applying scalability constraints.

Step 1 — Checking the current distribution of tasks

Docker services are composed of tasks which are scheduled into the cluster nodes based on the provided placement preferences and constraints. In the previous tutorial, our NGINX service was deployed via a Docker Stack, which was declared with the `nginx.yaml` compose file. Let's confirm this through the command below:

```
docker stack ls
```

You should observe just one stack which was deployed from our `nginx.yaml` compose file:

Output

```
NAME      SERVICES   ORCHESTRATOR
nginx     1          Swarm
```

Next, list all the services in the `nginx` stack using the command below:

```
docker stack services nginx
```

Output

```
ID             NAME          MODE         REPLICAS   IMAGE          PORTS
euqfhji2qpco   nginx_nginx   replicated   4/4        nginx:latest   *:8088->80/tcp
```

The above command produces all the services defined in the `nginx.yaml` file and deployed as part of the `nginx` stack above. In this case, it's the same as running `docker service ls` since there are no other stacks or services in the cluster.

It is easy to read through our `nginx.yaml` file to verify that we haven't defined any placement policies for this service. Take a look at the `deploy` section of the file:

nginx.yaml

```
version: '3.7'

networks:
  nginx:
    external: false

services:

  # --- NGINX ---
  nginx:
      image: nginx:latest
      ports:
        - '8088:80'
```

deploy:

```

```

replicas: 4

```

```

update_config:

parallelism: 2

order: start-first

failure_action: rollback

delay: 10s

rollback_config:

parallelism: 0

order: stop-first

restart_policy:

condition: any

delay: 5s

max_attempts: 3

window: 120s

```
    healthcheck:
      test: ["CMD", "service", "nginx", "status"]
    networks:
      - nginx
```

We can also confirm this by using `docker service inspect` to get detailed information about the NGINX service. Because this inspection can return an exhaustive amount of information, we can format the output using a Go template. In our case, we know that the service's placement policies are detailed under the service's `Spec->TaskTemplate->Placement` field, so we can format our inspection output by running the following:

```
docker service inspect nginx_nginx --format '{{json .Spec.TaskTemplate.Placement}}'
```

Output

```
{"Platforms":[{"Architecture":"amd64","OS":"linux"},{"OS":"linux"},{"OS":"linux"},{"Architectu
re":"arm64","OS":"linux"},{"Architecture":"386","OS":"linux"},{"Architecture":"mips64le","OS":
"linux"},{"Architecture":"ppc64le","OS":"linux"},{"Architecture":"s390x","OS":"linux"}]}
```

As you can see, the service does not have any placement policies. We should expect that Docker is smart enough to distribute the four replicas of this service throughout all schedulable nodes of the cluster as evenly as possible. We have four schedulable nodes in our cluster, so in theory, we should have two NGINX replicas per worker node (since all three manager nodes are drained). Let's confirm this:

```
docker service ps nginx_nginx # list the tasks of the nginx_nginx service
```

Output

```
ID             NAME                IMAGE          NODE        DESIRED STATE   CURRENT
STATE           ERROR     PORTS
ikjo7su8ooo6   nginx_nginx.1       nginx:latest   worker-2    Running         Running 27 hours
ago
inel921owq9b    \_ nginx_nginx.1   nginx:latest   manager-2   Shutdown        Shutdown 28
hours ago
y0y5mvj5n44r   nginx_nginx.2       nginx:latest   worker-1    Running         Running 27 hours
ago
uk18jo6wwlq6    \_ nginx_nginx.2   nginx:latest   manager-3   Shutdown        Shutdown 28
hours ago
```

```
ijbalagf7isy   nginx_nginx.3       nginx:latest   worker-1    Running         Running 27 hours
ago
drgu007baw8z   nginx_nginx.4       nginx:latest   worker-2    Running         Running 27 hours
ago
```

As you can see, both worker nodes `worker-1` and `worker-2` are running two replicas of the NGINX service each. You can also see that the replicas that were running on `manager-2` and `manager-3` were shut down and reassigned to `worker-2` and `worker-1` respectively when both manager nodes were drained in step 8 of the previous tutorial.

Step 2 — Scaling out the service to 5 replicas

Let's begin this section by assuming our NGINX service is under a high load. We'll need to add another NGINX instance (aka replica) to cope with the increased demand. We can use the `docker service scale` command for this purpose. We only need to state the service name and the number of desired replicas.

```
docker service scale nginx_nginx=5
```

Output

```
nginx_nginx scaled to 5
overall progress: 4 out of 5 tasks
1/5: running   [==================================================>]
2/5: starting  [============================================>      ]
3/5: running   [==================================================>]
4/5: running   [==================================================>]
5/5: running   [==================================================>]
```

Once the service is stable, we'll find a new replica in our service:

```
docker service ps nginx_nginx
```

Output

```
ID             NAME                IMAGE          NODE        DESIRED STATE   CURRENT
STATE            ERROR     PORTS
```

```
ikjo7su8ooo6   nginx_nginx.1       nginx:latest   worker-2    Running         Running 28 hours
ago
inel921owq9b    \_ nginx_nginx.1   nginx:latest   manager-2   Shutdown        Shutdown 29
hours ago
y0y5mvj5n44r   nginx_nginx.2       nginx:latest   worker-1    Running         Running 28 hours
ago
uk18jo6wwlq6    \_ nginx_nginx.2   nginx:latest   manager-3   Shutdown        Shutdown 29
hours ago
ijbalagf7isy   nginx_nginx.3       nginx:latest   worker-1    Running         Running 28 hours
ago
drgu007baw8z   nginx_nginx.4       nginx:latest   worker-2    Running         Running 28 hours
ago
```

2cd89o4cybmn nginx_nginx.5 nginx:latest worker-2 Running Running 11 seconds ago

This new `nginx_nginx.5` task is now running alongside `nginx_nginx.1` and `nginx_nginx.4` in `worker-2`.

We could've also added a new NGINX replica by running:

```
docker service update nginx_nginx --replicas 5
```

The outcome would've been the same as:

```
docker service scale nginx_nginx=5
```

Nonetheless, we recommend always using the `docker service scale` command as it can scale multiple services simultaneously. For example:

```
docker service scale my_service=3 my_other_service=4 ...
```

On the other hand, `docker service update` can only update one service at a time.

Step 3 — Undoing a scaling operation

In this step, we will assume that we scaled our service by accident in the previous section and we'd like to revert our changes. As mentioned earlier, the `docker service scale` command is pretty much the same as a `docker service update` that only targets the number of service replicas. What happens under the hood is that the service is taking in a new specification where the number of replicas differs from the current one.

So how can we return our service to its previous state? Do we need to remember its entire previous specification by heart? Gladly we don't! Docker is smart enough to keep the service's previous specification within its definition. By running `docker service inspect`, we can examine a service's current and previous states:

```
docker service inspect nginx_nginx
```

Output

```
[
    {
        "ID": "euqfhji2qpco",
        . . .
```

"Spec": {

"Name": "nginx_nginx",

. . .

"TaskTemplate": {

"ContainerSpec": {

"Image": "nginx:latest@sha256:2834dc507516af02784808c5f48b7cbe38b8ed5d0f4837f16e78d00deb7e7767",

"Labels": {

"com.docker.stack.namespace": "nginx"

},

. . .

},

. . .

},

"Mode": {

"Replicated": {

"Replicas": 5

}

},

. . .

"RollbackConfig": {

"Parallelism": 0,

"FailureAction": "pause",

"Monitor": 5000000000,

"MaxFailureRatio": 0,

"Order": "stop-first"

},

. . .

},

"PreviousSpec": {

"Name": "nginx_nginx",

. . .

"TaskTemplate": {

"ContainerSpec": {

"Image":
"nginx:latest@sha256:2834dc507516af02784808c5f48b7cbe38b8ed5d0f4837f16e78d00deb7e7767",

"Labels": {

"com.docker.stack.namespace": "nginx"

},

. . .

},

. . .

},

"Mode": {

"Replicated": {

"Replicas": 4

}

},

. . .

"RollbackConfig": {

"Parallelism": 0,

"FailureAction": "pause",

"MaxFailureRatio": 0,

"Order": "stop-first"

},

. . .

},

```
        . . .
    }
]
```

The above command results in a large output (thus truncated), but we can clearly see two top-level keys, `Spec` and `PreviousSpec`, which look identical at first, but if you look closely, `PreviousSpec` defines a `Replicated` service with `4` replicas, while `Spec` sets this number to `5` (as expected, given the scale out we've performed in the previous step).

Since we have this `PreviousSpec` information, we can determine what our service will look like once we revert to the previous state. But the question now is: how do we undo the last scaling

operation?

Once again, Docker comes to the rescue! The `docker service` management command has a `rollback` subcommand for these situations. There's not much to know about it, you simply run `docker service rollback` on the desired service, and Docker will automatically update it to its `PreviousSpec` configuration. Go ahead and try it out as shown below:

```
docker service rollback nginx_nginx
```

Output

```
nginx_nginx
rollback: manually requested rollback
overall progress: rolling back update: 4 out of 4 tasks
1/4: running   [>                                                  ]
2/4: running   [>                                                  ]
3/4: running   [>                                                  ]
4/4: running   [>                                                  ]
verify: Service converged
rollback: rollback completed
```

If the rollback is successful, we should now see our initial four replicas instead of the five we scaled out to in the previous step:

```
docker service ps nginx_nginx
```

Output

```
ID             NAME                IMAGE          NODE        DESIRED STATE   CURRENT
STATE           ERROR     PORTS
ikjo7su8ooo6   nginx_nginx.1       nginx:latest   worker-2    Running         Running 29 hours
ago
inel921owq9b    \_ nginx_nginx.1   nginx:latest   manager-2   Shutdown        Shutdown 30
hours ago
y0y5mvj5n44r   nginx_nginx.2       nginx:latest   worker-1    Running         Running 29 hours
ago
uk18jo6wwlq6    \_ nginx_nginx.2   nginx:latest   manager-3   Shutdown        Shutdown 30
hours ago
ijbalagf7isy   nginx_nginx.3       nginx:latest   worker-1    Running         Running 29 hours
ago
```

```
drgu007baw8z   nginx_nginx.4       nginx:latest   worker-2    Running         Running 29 hours
ago
```

And there we go, our service is back to where it was in Step 1, running with four replicas.

Step 4 — Scaling in the service to 3 replicas

Let's now assume we've been able to absorb the high service loads with our previous scale out and are now experiencing low demand for our service. To free up some space and potentially reduce our compute costs, let's shrink our service from four to three replicas:


```
docker service scale nginx_nginx=3
```

Output

```
nginx_nginx scaled to 3
overall progress: 3 out of 3 tasks
1/3: running   [==================================================>]
2/3: running   [==================================================>]
3/3: running   [==================================================>]
verify: Service converged
```

The command is pretty much the same as in step 2, except for the desired number of replicas. Docker will take care of shutting down one of the replicas, leaving us with the desired number, evenly distributed throughout the cluster once again:


```
docker service ps nginx_nginx
```

Output

```
ID             NAME                IMAGE          NODE        DESIRED STATE   CURRENT
STATE           ERROR     PORTS
ikjo7su8ooo6   nginx_nginx.1       nginx:latest   worker-2    Running         Running 29 hours
ago
inel921owq9b    \_ nginx_nginx.1   nginx:latest   manager-2   Shutdown        Shutdown 30
hours ago
y0y5mvj5n44r   nginx_nginx.2       nginx:latest   worker-1    Running         Running 29 hours
```

```
ago
uk18jo6wwlq6    \_ nginx_nginx.2   nginx:latest   manager-3   Shutdown        Shutdown 30
hours ago
ijbalagf7isy   nginx_nginx.3       nginx:latest   worker-1    Running         Running 29 hours
ago
```

Notice that other replicas weren't disturbed by this action — they are still running, uninterruptedly, for 29 hours.

Step 5 — Defining constraints while scaling a service

Finally, sometimes you might want to ensure that a certain condition is met, even after your service is already running. For example, let's assume we now need to scale out to five NGINX replicas again, but we discovered that our worker nodes are not capable of hosting more than two NGINX replicas concurrently without a performance degradation.

In such scenario, we can no longer use `docker service scale` because it does not allow us to specify other configurations besides the number of replicas. This is where the `docker service update` command comes in, as it offers a wide range of options for configuring the new state of our service.

We need a new number of replicas, plus one condition (no more than two replicas per node), so we can use the following options:

```
docker service update --help
```

Output

```
. . .
--constraint-add list                Add or update a placement constraint
. . .
--replicas uint                      Number of tasks
--replicas-max-per-node uint         Maximum number of tasks per node (default 0 = unlimited)
. . .
```

Go ahead and run the following service update command to scale out to five replicas once more:

```
docker service update nginx_nginx --replicas 5 --replicas-max-per-node 2
```

Output

```
nginx_nginx
overall progress: 2 out of 5 tasks
1/5: starting  [============================================>      ]
2/5: running   [==================================================>]
3/5: starting  [============================================>      ]
4/5: running   [==================================================>]
```

5/5: no suitable node (max replicas per node limit exceed; scheduling constrain…

```
```

You'll observe that Docker will rightfully complain about your decision, because you've restricted each worker to only two replicas which means the fifth one will have no where to go. Since the above command was executed in the foreground, Docker will keep indefinitely, trying to fix this situation. You'll see something like this:

Output

```
nginx_nginx
overall progress: 4 out of 5 tasks
1/5: running   [==================================================>]
2/5: running   [==================================================>]
3/5: running   [==================================================>]
4/5: running   [==================================================>]
5/5: no suitable node (max replicas per node limit exceed; scheduling constrain…
```

We can `Ctrl+c` to interrupt the process (the service update will continue in the background), and then verify that this new task is pending, waiting for a suitable node to show up:

```
docker service ps nginx_nginx --no-trunc # use --no-trunc to see the whole output
```

Output

```
ID                          NAME
IMAGE
NODE        DESIRED STATE   CURRENT STATE
ERROR
 PORTS
ikjo7su8ooo6b7wmh4r9uj1lq   nginx_nginx.1
```

```
nginx:latest@sha256:0b970013351304af46f322da1263516b188318682b2ab1091862497591189ff1   worker-
2    Running         Running 3 days ago
inel921owq9bjq4zotkz590dz    \_ nginx_nginx.1
nginx:latest@sha256:0b970013351304af46f322da1263516b188318682b2ab1091862497591189ff1
manager-2   Shutdown        Shutdown 3 days ago
y0y5mvj5n44r23b5y548hrvsm   nginx_nginx.2
nginx:latest@sha256:0b970013351304af46f322da1263516b188318682b2ab1091862497591189ff1   worker-
1    Running         Running 3 days ago
uk18jo6wwlq6qfrv2lr6zk1xz    \_ nginx_nginx.2
nginx:latest@sha256:0b970013351304af46f322da1263516b188318682b2ab1091862497591189ff1
manager-3   Shutdown        Shutdown 3 days ago
ijbalagf7isyu1drhm0hu8z95   nginx_nginx.3
nginx:latest@sha256:0b970013351304af46f322da1263516b188318682b2ab1091862497591189ff1   worker-
1    Running         Running 3 days ago
lqa493b94iu8lpmc8yqtu4jd1   nginx_nginx.4
nginx:latest@sha256:0b970013351304af46f322da1263516b188318682b2ab1091862497591189ff1   worker-
2    Running         Running 4 minutes ago
```

txc8rjd3xpojvl7ckz8zgohx9 nginx_nginx.5 nginx:latest@sha256:0b970013351304af46f322da1263516b188318682b2ab1091862497591189ff1 Running Pending 5 minutes ago "no suitable node (3 nodes not available for new tasks; max replicas per node limit exceed)"

```
```

To move the highlighted `Pending` task above to `Running`, you can take either of the following actions:

1. Redefine the scheduling constraints by allowing the cluster to run more than two replicas per node (as defined in the `docker service update` command above).
2. Horizontally scaling the cluster by provisioning additional worker nodes (this will be covered in the next tutorial).

Auto scaling Docker services (optional)

When talking about scaling container services, "auto scaling" often comes up. Now, the reason why this is a bonus (and optional) step, is because **Docker doesn't really offer any auto scaling mechanisms** (unlike Kubernetes).

Nonetheless, you might be wondering how to introduce some level of automation to help scale in and out Docker services. Let's face it, there is no right answer for this, nor there is any "one size

fits all" solution that can help you. Because we cannot auto scale directly with Docker, we must resort to workarounds that might involve third-party tools, custom scripts, etc.

There are some tools out there (such as Orbiter) that are designed specifically to fill this gap in Docker. But depending on the use case and auto scaling rules, you could build a custom Docker auto scaler with just a few lines of Bash and a cronjob!

Our NGINX service from above currently has five replicas. Regardless of their state and placement policies, let's say we simply want to **add** and **delete** service replicas every time a worker joins or leaves the Docker Swarm cluster, such that we always have two replicas per worker.

Start by building a script to monitor the cluster size and scale the service when necessary. As a superuser, create a new file called "/opt/autoscaler.sh" and open it with your favorite text editor:

```
sudo nano /opt/autoscaler.sh
```

Paste the following text into the file:

/opt/autoscaler.sh

```
#!/bin/sh

service_name=nginx_nginx

# 1
## let's see how many workers we have, by listing and counting all
## node IDs corresponding to the workers in the cluster
num_workers=$(docker node ls --filter role=worker -q | wc -l)

# 2
## then, let's infer the current number of replicas our service has
num_replicas=$(docker service inspect $service_name --format '{{json
.Spec.Mode.Replicated.Replicas}}')

# 3
## let's then assess how many replicas we should have (2 per node)
let "target_replicas = 2*$num_workers"

# 4
## and finally, let's compare the target replicas
## with the ones we currently have, and if they are different
```

```
## we scale the service to the number of target_replicas
if [[ $num_replicas -ne $target_replicas ]]
then
  echo "Autoscaling Docker service ${service_name} to ${target_replicas} replicas"
  docker service scale $service_name=$target_replicas -d
fi

## else, there's nothing to do since we already have
## the desired amount of replicas
```

The above file defines a simplistic auto scaling rule, but applicable nonetheless. Next, you need to run this script periodically via a cronjob (say once every hour). To do that, execute the following command to open the crontab config file in your text editor:

```
crontab -e
```

Add the following line at the end of the file, then save and close it.

```
0 * * * * sh /opt/autoscaler.sh
```

With this setup in place, the `/opt/autoscaler.sh` script will be executed once per hour by the system's `cron` daemon. While this autoscaling infrastructure isn't production quality, it is enough to bootstrap your understanding of how to work around the autoscaling limitations in Docker.

Conclusion and next steps

In this tutorial, you learned about scaling a Docker Swarm service in and out, and how to undo a scaling operation. You also learned how to inspect a service so that you can make educated decisions about your scaling constraints. Finally, we touched base on auto scaling, giving you some tips on how to automate the scaling operations for your service.

In a future tutorial, we'll learn about horizontally scaling the cluster itself, such that a suitable node shows up to serve the pending task from step five.

This article was contributed by guest author Cristovao Cordeiro, a Docker certified Engineering Manager at Canonical. He's an expert in Containers and ex-CERN engineer with 9+ years of experience in Cloud and Edge computing.

Revision #1
Created 22 March 2026 19:28:55 by Administrador

Updated 22 March 2026 19:30:39 by Administrador