Technology Conversations

Scaling To Infinity with Docker Swarm, Docker Compose and Consul (Part 1/4) – A Taste of What Is To Come

This series is split into following articles.

- A Taste of What Is To Come
- Manually Deploying Services
- Blue-Green Deployment, Automation and Self-Healing Procedure
- Scaling Individual Services

Previous articles put a lot a focus on **Continuous Delivery** and **Containers with Docker**. In **Continuous Integration, Delivery or Deployment with Jenkins, Docker and Ansible** I explained how to continuously build, test and deploy micro services packaged into containers and do that across multiple servers, without downtime and with the ability to rollback. We used Ansible, Docker, Jenkins and few other tools to accomplish that goal.

Now it's time to extend what we did in previous articles and scale services across any number of servers. We'll treat all servers as one **server farm** and deploy containers not to predefined locations but to those that have the least number of containers running. Instead of thinking about each server as an individual place where we deploy, we'll treat all of them as one unit.

We'll continue using some of the same tools we used before.

- **Vagrant** with **VirtualBox** will provide an easy way to create and configure lightweight, reproducible, and portable virtual machines that will act as our servers.
- **Docker** will provide an easy way to build, ship, and run distributed applications packaged in containers.
- **Ansible** will be used to setup servers and deploy applications.
- We'll use **Jenkins** to detect changes to our code repositories and trigger jobs that will test, build and deploy applications.
- Finally, nginx will provide proxy to different servers and ports our micro services will run on.

On top of those we'll see some new ones like following.

- **Docker Compose** is a handy tool that will let us run containers.
- **Docker Swarm** will turn a pool of servers into a single, virtual host.
- Finally, we'll use **Consul** for service discovery and configuration.

In order to follow this article, set up **Vagrant** with **VirtualBox**. Once done, please install vagrant-cachier plugin. While it's not mandatory, it will speed up VM creation and installations.

Some users reported difficulties running Ansible inside Vagrant VM on Windows. Seems that the problem exists only on older versions of Vagrant. Please make sure that the latest Vagrant is installed.

```
vagrant plugin install vagrant-cachier
```

With prerequisites out of our way, we're ready to start building our server farm.

Servers Setup

We'll create four virtual machines. One (swarm-master) will be used to orchestrate deployments. Its primary function is to act as **Docker Swarm master node**. Instead of deciding in advance where to deploy something, we'll tell Docker Swarm what to deploy and it will deploy it to a node that has the least number containers running. There are other strategies that we could employ but, as a demonstration, this default one should suffice. Besides Swarm, we'll also set up **Ansible**, **Jenkins**, **Consul** and **Docker Compose** on that same node. Three additional virtual machines will be created and named **swarm-node-01**, **swarm-node-02** and **swarm-node-03**. Unlike **swarm-master**, those nodes will have only Consul and Swarm agents. Their purpose is to host our services (packed as Docker containers). Later on, if we need more hardware, we would just add one more node and let **Swarm** take care of balancing deployments.

We'll start by bringing up Vagrant VMs[1]. Keep in mind that four virtual machines will be created and that each of them requires 1GB of RAM. On a 8GB 64 bits computer, you should have no problem running those VMs. If you don't have that much memory to spare, please try edit the **Vagrantfile** by changing `v.memory = 1024` to some smaller value.

All the code is located in the vfarcic/docker-swarm GitHub repository.

```
git clone https://github.com/vfarcic/docker-swarm.git
cd docker-swarm
vagrant up
vagrant ssh swarm-master
```

We can set up all the servers by running **infra.yml** Ansible playbook.

```
ansible-playbook /vagrant/ansible/infra.yml -i /vagrant/ansible/hosts/prod
```

First time you run Ansible against one server, it will ask you whether you want to continue connecting. Answer with **yes**.

A lot of things will be downloaded (Jenkins container being the biggest) and installed with this command so be prepared to wait for a while. I won't go into details regarding Ansible. You can find plenty of articles about it both in the official site as well as in other posts on this blog. Important detail is that, once the execution of the Ansible playbook is done, **swarm-master** will have **Jenkins**, **Consul**, **Docker Compose** and **Docker Swarm** installed. The other three nodes received instructions to install only **Consul** and **Swarm agents**. For more information please consult the Continuous Integration, Delivery or Deployment with Jenkins, Docker and Ansible and other articles in Continuous Integration Delivery and Deployment.

Throughout this article, we will never enter any of the **swarm-node** servers. Everything will be done from the single location (**swarm-master**).

Now let us go through few of the tools we haven't used before in this blog; **Consul** and **Docker Swarm**.

Consul

Consul is a tool aimed at easy service discovery and configuration of distributed and highly available data centers. It also features easy to set up failure detection and key/value storage.

Let us take a look at **Consul** that was installed on all machines.

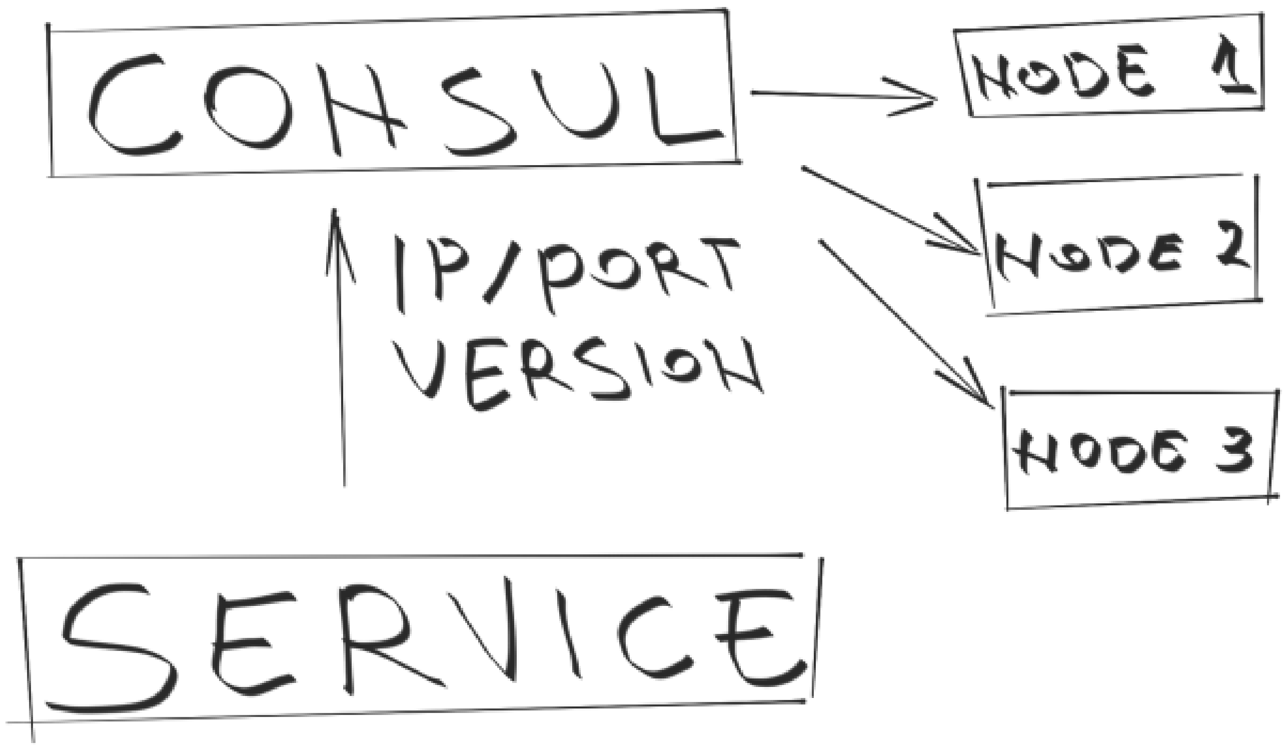
For example, we can see all members of our cluster with the following command.

```
consul members
```

The output should be something similar to the following.

```
Node           Address              Status  Type    Build  Protocol
swarm-master   10.100.199.200:8301  alive   server  0.5.0  2
swarm-node-01  10.100.199.201:8301  alive   client  0.5.0  2
swarm-node-02  10.100.199.202:8301  alive   client  0.5.0  2
swarm-node-03  10.100.199.203:8301  alive   client  0.5.0  2
```

With Consul running everywhere we have the ability to store information about applications we deploy and have it propagated to all servers. That way, applications store data locally and do not have to worry about location of a central server. At the same time, when an application needs information about others, it can also request it locally. Being able to propagate information across all servers is an essential requirement for all distributed systems.



Another way to retrieve the same information is through Consul's REST API. We can run following command.

```
curl localhost:8500/v1/catalog/nodes | jq .
```

This produces following JSON output formatted with **jq**.

```
[
  {
    "Node": "swarm-master",
    "Address": "10.100.199.200"
  },
  {
    "Node": "swarm-node-01",
    "Address": "10.100.199.201"
  },
  {
    "Node": "swarm-node-02",
    "Address": "10.100.199.202"
  },
  {
    "Node": "swarm-node-03",
    "Address": "10.100.199.203"
  }
]
```

Later on, when we deploy the first application, we'll see **Consul** in more detail. Please take note that even though we'll use Consul by running commands from Shell (at least until we get to health section), it has an UI that can be accessed by opening http://10.100.199.200:8500.

Docker Swarm

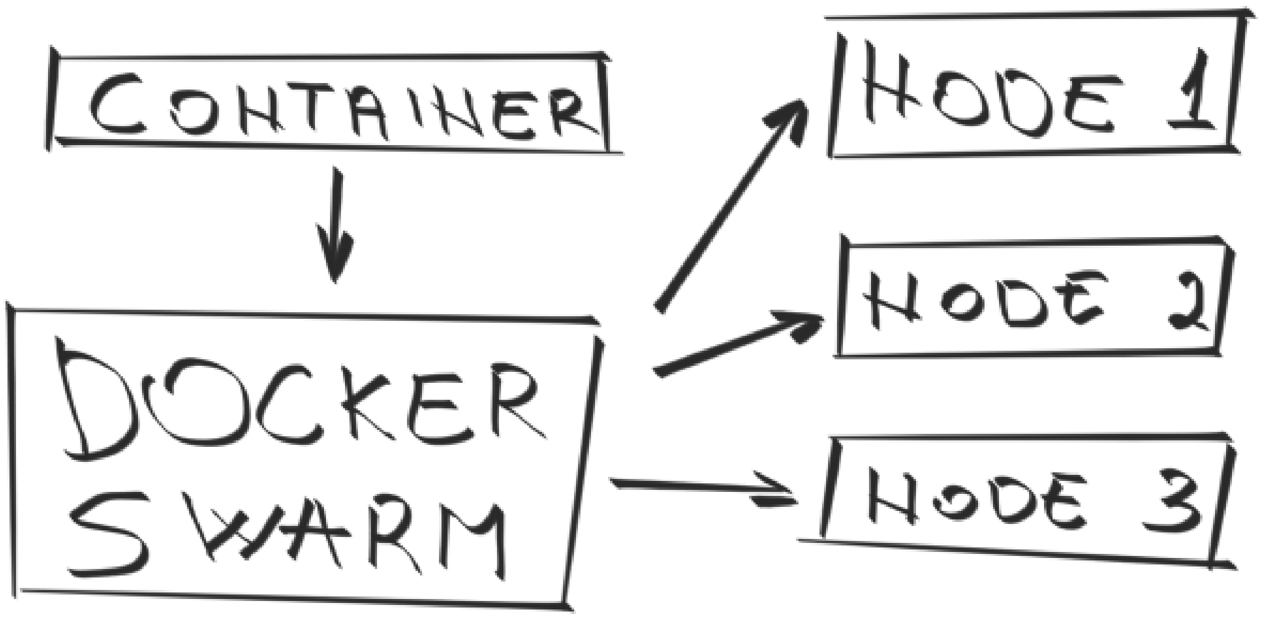
Docker Swarm allows us to leverage standard Docker API to run containers in a cluster. the easiest way to use it is to set the DOCKER_HOST environment variable. Let's run Docker command **info**.

```
export DOCKER_HOST=tcp://0.0.0.0:2375
docker info
```

The output should be similar to the following.

```
Containers: 9
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 3
 swarm-node-01: 10.100.199.201:2375
  └ Containers: 3
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.019 GiB
 swarm-node-02: 10.100.199.202:2375
  └ Containers: 3
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.019 GiB
 swarm-node-03: 10.100.199.203:2375
  └ Containers: 3
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.019 GiB
```

We get immediate information regarding number of deployed containers (at the moment 9), strategy Swarm uses to distribute them (spread; runs on a server with the least number of running containers), number of nodes (servers) and additional details for each of them. At the moment, each server has one Swarm Agent and two Consul Registrators deployed (nine in total). All those deployments were done as part of the **infra.yml playbook** that we run earlier.



Deployment

Let us deploy the first service. We'll use Ansible playbook defined in books-service.yml.

```
ansible-playbook /vagrant/ansible/books-service.yml -i /vagrant/ansible/hosts/pı
```

Running this, or any other playbook from this article is slow because we're pulling images to all nodes, not only the one we'll deploy to. The reason behind this is that, in case a node fails, we want to have everything ready for as fast as possible re-deployment to a different node. Good news is that next time you run it, it will be much faster since images are already downloaded and Docker will only pull differences.

The playbook that we just run follows the same logic as the one we already discussed in the blue/green deployment article. The major difference is that this time there are few things that are unknown to us before playbook is actually run. We don't know the IP of the node service will be deployed to. Since the idea behind this setup is not only to distribute applications between multiple nodes but also to scale them effortlessly, port is also unknown. If we'd define it in advance there would be a probable danger that multiple services would use the same port and clash.

Right now we'll go through what this playbook does and, later on in the next article, we'll explore how it was done.

Books service consists of two containers. One is the application itself and the other contains MongoDB that the application needs. Let's see where they deployed to.

```
docker ps | grep booksservice
```

The result will differ from case to case and it will look similar to the following. Docker **ps** command will output more information than presented below. Those that are not relevant for this article were removed.

```
vfarcic/books-service:latest  10.100.199.203:32768->8080/tcp   swarm-node-03/boo
mongo:latest                  10.100.199.201:32768->27017/tcp  swarm-node-01/boo
```

We can see that the application container was deployed to **swarm-node-03** and is listening the port **32768**. Database, on the other hand, went to a separate node **swarm-node-01** and listens to the port **32768**. The purpose of the **books service** is to store and retrieve books from the Mongo database.

Let's check whether those two containers communicate with each other. When we request data from the application container (**booksservice_blue_1**) it will retrieve it from the database (**booksservice_db_1**). In order to test it we'll request service to insert few books and then ask it to retrieve all store records.

```
curl -H 'Content-Type: application/json' -X PUT -d '{"_id": 1, "title": "My Firs
curl -H 'Content-Type: application/json' -X PUT -d '{"_id": 2, "title": "My Seco
curl -H 'Content-Type: application/json' -X PUT -d '{"_id": 3, "title": "My Thir
curl http://10.100.199.200/api/v1/books | jq .
```

The result of the last request is following.

```
[
  {
    "_id": 1,
    "title": "My First Book",
    "author": "John Doe"
  },
  {
    "_id": 2,
    "title": "My Second Book",
    "author": "John Doe"
  },
  {
    "_id": 3,
    "title": "My Third Book",
    "author": "John Doe"
  }
]
```

All three books that we requested the service to put to its database were stored. You might have noticed that we did not perform requests to the IP/port where the application is running. Instead of doing `curl` against **10.100.199.203:32768** (this is where the service is currently running) we performed requests to **10.100.199.200** on the standard HTTP port **80**. That's where our **nginx** server is deployed and, through the "magic" of Consul, Registrator and Templating, **nginx** was updated to point to the correct IP and port. Details of how this happened are explained in the next article. For now, it is important to know that data about our application is stored in Consul and freely accessible to every service that might need it. In this case, that service is **nginx** that acts as a reverse proxy and load balancer at the same time.

To prove this, let's run the following.

```
curl http://localhost:8500/v1/catalog/service/books-service-blue | jq .
```

Since we'll practice **blue/green deployment**, name of the service is alternating between **books-service-blue** and **books-service-green**. This is the first time we deployed it so the name is **blue**. The next deployment will be **green**, than **blue** again and so on.

```
[
  {
    "Node": "swarm-node-03",
    "Address": "10.100.199.203",
    "ServiceID": "swarm-node-03:booksservice_blue_1:8080",
    "ServiceName": "books-service-blue",
    "ServiceTags": null,
    "ServiceAddress": "",
    "ServicePort": 32768
  }
]
```

We also have the information stored as **books-service** (generic one, without blue or green) with IP and port that should be accessible to public.

```
curl http://localhost:8500/v1/catalog/service/books-service | jq .
```

Unlike previous outputs that can be different from case to case (IPs and ports are changing from deployment to deployment), this output should always be the same.

```
[
  {
    "Node": "swarm-master",
    "Address": "10.100.199.200",
    "ServiceID": "books-service",
    "ServiceName": "books-service",
    "ServiceTags": [
      "service"
    ],
    "ServiceAddress": "10.100.199.200",
    "ServicePort": 80
  }
]
```

No matter where we deploy our services, they are always accessible from a single location **10.100.199.200** (at least until we start adding multiple load balancers) and are always accessible from the default HTTP port 80. **nginx** will make sure that requests are sent to the correct service on the correct IP and port.

We can deploy another service using the same principle. This time it will be a front-end for our books-service.

```
ansible-playbook /vagrant/ansible/books-fe.yml -i /vagrant/ansible/hosts/prod
```

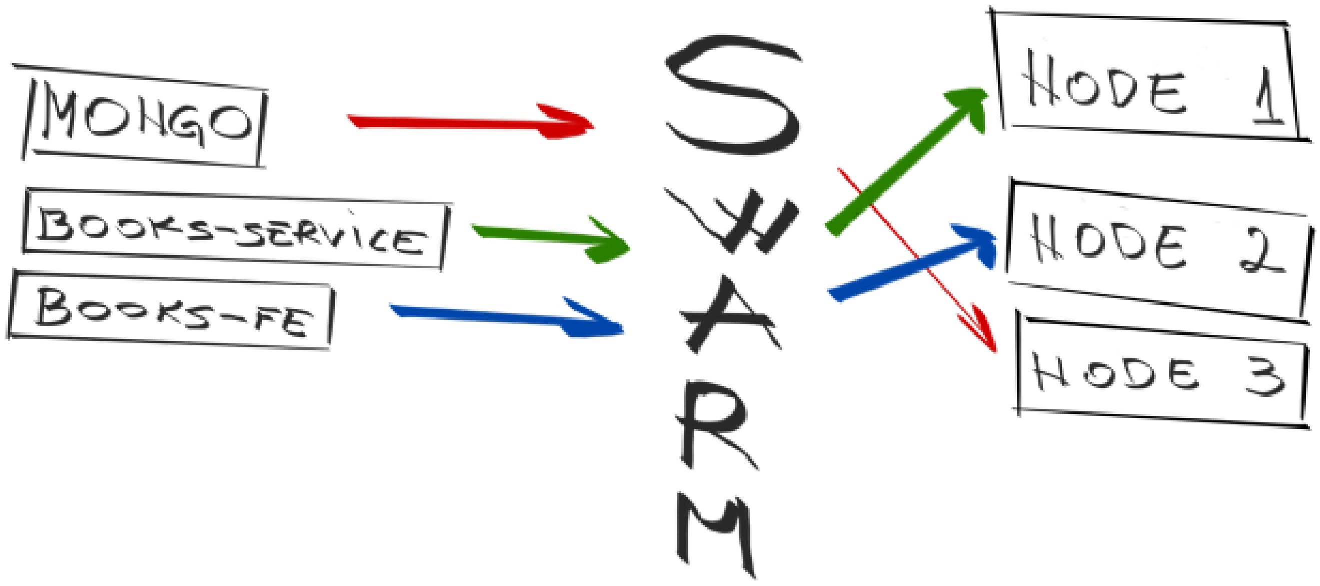
You can see the result by opening http://10.100.199.200 in your browser. It's an **AngularJS UI** that uses the service we deployed earlier to retrieve all the available books. As with the **books-service**, you can run following to see where the container was deployed.

```
docker ps | grep booksfe
curl http://localhost:8500/v1/catalog/service/books-fe-blue | jq .
```

The output of both commands should be similar to the following.

```
vfarcic/books-fe:latest         10.100.199.201:32769->8080/tcp    swarm-node-0

[
  {
    "Node": "swarm-node-01",
    "Address": "10.100.199.201",
    "ServiceID": "swarm-node-01:booksfe_blue_1:8080",
    "ServiceName": "books-fe-blue",
    "ServiceTags": null,
    "ServiceAddress": "",
    "ServicePort": 32769
  }
]
```



Now let us imagine that someone changed the code of the books-service and that we want to deploy a new release. The procedure is exactly the same as we did before.

```
ansible-playbook /vagrant/ansible/books-service.yml -i /vagrant/ansible/hosts/pı
```

To verify that everything went as expected we can query Consul.

```
curl http://localhost:8500/v1/catalog/service/books-service-green | jq .
```

The output should be similar to the following.

```
[
  {
    "Node": "swarm-node-02",
    "Address": "10.100.199.202",
    "ServiceID": "swarm-node-02:booksservice_green_1:8080",
    "ServiceName": "books-service-green",
    "ServiceTags": null,
    "ServiceAddress": "",
    "ServicePort": 32768
  }
]
```

While **blue** release was on IP **10.100.199.203**, this time the container was deployed to **10.100.199.202**. **Docker Swarm** checked which server had the least number of containers running and decided that the best place to run it is **swarm-node-02**.

You might have guessed that, at the beginning, it's easy to know whether we deployed blue or green. However, we'll loose track very fast with increased number of deployments and services. We can solve this by querying Consul keys.

```
curl http://localhost:8500/v1/kv/services/books-service/color | jq .
```

Values in Consul are stored in base64 encoding. To see only the value, run following.

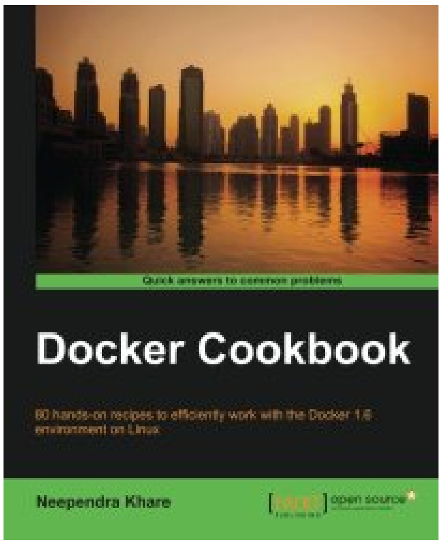
```
curl http://localhost:8500/v1/kv/services/books-service/color?raw
```

The output of the command is `green`.

Jenkins

The only thing missing for fully implemented Continuous Deployment is to have something that will detect changes to our source code repository and then build, test and deploy containers. With Docker it's very easy to have all builds, testing and deployments follow the same standard. For this article, I created only jobs that do the actual deployment. We'll use them later in the next article when we explore ways to recuperate from failure. Until then, you can take a look at the running Jenkins instance by opening http://10.100.199.200:8080/.

To Be Continued


Quick answers to common problems
Docker Cookbook
80 hands-on recipes to efficiently work with the Docker 1.6 environment on Linux
Neependra Khare
open source

In the next article we're exploring additional features of Consul and how we can utilize them to recuperate from failures. Whenever some container stops working, Consul will detect it and send a request to Jenkins which, in turn, will redeploy the failed container. While Jenkins jobs created for this article only deploy services, you could easily extend them to, let's say, send an email when the request comes from Consul indicating a failure.

We'll go in-depth how all this was accomplished and show manual commands with Docker Compose, Consul-Template, Registrator, etc. Their understanding is a prerequisites for explanation of Ansible playbooks that we saw (and run) earlier.

Finally, we'll explore how we could scale multiple instances of same applications.

You got the taste of **what** and now it's time to understand **how**.

The story continues in the Manually Deploying Services article.

1. If you run into issues with Ansible complaining about executable permissions, try modifying the `Vagrantfile`'s `synced_folder` entry from `config.vm.synced_folder ".", "/vagrant"` to `config.vm.synced_folder ".", "/vagrant", mount_options: [“dmode=700,fmode=600”]`

This entry was posted in Continuous Integration, Delivery and Deployment, Docker, Tutorial and tagged Ansible, Consul, Continuous delivery, Continuous integration, Continuous Integration, Delivery and Deployment, Data Center, Docker, Docker Compose, Docker Swarm, jenkins, Microservices, nginx, Scaling, Server Farm, Vagrant on July 2, 2015 [http://technologyconversations.com/2015/07/02/scaling-to-infinity-with-docker-swarm-docker-compose-and-consul-part-14-a-taste-of-what-is-to-come/] by Viktor Farcic.

38 thoughts on "Scaling To Infinity with Docker Swarm, Docker Compose and Consul (Part 1/4) – A Taste of What Is To Come"

Pingback: Links & Reads from 2015 Week 27 | Martin's Weekly Curations

Hugh Messenger
July 13, 2015 at 5:25 pm

Really want to run through this tutorial, but have run in to two issues.

1) I'm using Vagrant / Virtual Box on Windows, so I immediately hit this problem:

https://github.com/ansible/ansible/issues/10068

… which (I think) I've worked round by copying /vagrant/ancible to /etc/ancible and chmod -x'ing the hosts/prod file.

2) When running ansible-playbook, I'm getting an SSH failure ..

```
fatal: [10.100.199.200] => SSH Error: Permission denied (publickey,password).
while connecting to 10.100.199.200:22
```

I'm currently wading through the ssh -vvv output to see what's going on there!

Viktor Farcic Post author
July 13, 2015 at 5:44 pm

Unfortunately I don't have Windows on any of my machines and cannot fix this problem. If you find the solution I would appreciate if you could create a pull request and/or send me text that I could put into the article.

Tordek
July 14, 2015 at 1:58 am

The second error is because the key seems to not be stored in the repo; I fixed it by manually copying the private key from…

https://github.com/mitchellh/vagrant/blob/master/keys/vagrant

into /vagrant/.vagrant/machines/swarm-master/virtualbox/private_key and setting the permissions to 400.

Viktor Farcic Post author
July 14, 2015 at 1:51 pm

Did the rest work for you on Windows?

Tordek
July 15, 2015 at 7:21 am

I didn't do it in Windows; I used Debian Unstable.

cheesegrits
July 14, 2015 at 12:16 am

The ssh problem was related to the ansible_ssh_private_key_file directive in /etc/ansible/host_vars/10.100.199.20x …

ansible_ssh_private_key_file: /vagrant/.vagrant/machines/swarm-master/virtualbox/private_key

… which doesn't exist for any of the four hosts. So I created a standard (pass phrase-less) key pair for the vagrant user on swarm-master …

ssh-keygen -t dsa
cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys

… and used vagrant ssh to then copy it to authorized_keys on the other three nodes, and commented out that ansible_ssh_private_key_file line in all four host_vars files.

That got me a bit further, but the ancible-playbook now fails for one of the nodes:

TASK: [swarm | Swarm node is running] **

failed: [10.100.199.201] => {"changed": false, "failed": true}
msg: ConnectionError(MaxRetryError("UnixHTTPConnectionPool(host='localhost', por
t=None): Max retries exceeded with url: /run/docker.sock/v1.12/images/swarm/json
(Caused by : [Errno 2] No such file or directory)",),)

FATAL: all hosts have already failed — aborting

PLAY RECAP **

```
    to retry, use: --limit @/home/vagrant/infra.retry
```

10.100.199.200 : ok=19 changed=1 unreachable=0 failed=0
10.100.199.201 : ok=9 changed=0 unreachable=0 failed=1
10.100.199.202 : ok=1 changed=0 unreachable=0 failed=0
10.100.199.203 : ok=1 changed=0 unreachable=0 failed=0

Viktor Farcic Post author
July 14, 2015 at 12:38 am

That's strange because Vagrant creates SSH keys to that location.

In the current setting, Ansible is run in serial mode (one node after another) instead of parallel. It seems that it fails on one of the hosts but that is because the rest (swarm-node-0[2-3]) is not even run. I suspect that in this case the problem is not related with Vagrant. Can you try again something like following:

```
vagrant destroy
vagrant up –no-provision
vagrant ssh swarm-master
$ ssh-keygen
$ cd /vagrant
$ ./bootstrap.sh
$ ./bootstrap_ansible.sh
$ exit

vagrant ssh swarm-node-01
$ ssh-keygen
$ cd /vagrant
$ ./bootstrap.sh
$ exit
[REPEAT FOR SWARM-NODE-0[2-3]

vagrant ssh swarm-master
$ ssh-copy-id 10.100.199.200
$ ssh-copy-id 10.100.199.201
$ ssh-copy-id 10.100.199.202
$ ssh-copy-id 10.100.199.203
[copy file from /vagrant to somewhere else and change permissions]
[run commands from the article but from the directory where files are copied]
```

cheesegrits
July 14, 2015 at 1:52 am

I'll try that later tonight. Just FYI, I tore everything down and started from scratch, with my fixes for 1) and 2), but it still failed with the connection error on node-01.

albacerrada
July 14, 2015 at 1:48 pm

I've got the windows based installation much further along by adding the following:

```
config.vm.synced_folder ".", "/vagrant",
mount_options: ["dmode=700,fmode=600"]
```

But after much waiting (for jenkins) it finally failed. The docker seems to be failing to run (keeps respawning on the first node)

```
TASK: [docker | Add Docker repository and update apt cache] *******************
changed: [10.100.199.201]

TASK: [docker | Docker is present] ********************************************
changed: [10.100.199.201]

TASK: [docker | Python-pip is present] ****************************************
changed: [10.100.199.201]

TASK: [docker | Docker-py is present] *****************************************
changed: [10.100.199.201]

TASK: [swarm | Directories are present] ***************************************
skipping: [10.100.199.201] => (item=/data/scripts)

TASK: [swarm | Scripts are present] *******************************************
skipping: [10.100.199.201] => (item={'dest': '/data/scripts/swarm_get_ip.sh', 'src': 'sw

TASK: [swarm | Obsolete Docker init.d service is removed] *********************
changed: [10.100.199.201]

TASK: [swarm | Configuration is present] **************************************
changed: [10.100.199.201]

TASK: [swarm | Service is restarted] ******************************************
changed: [10.100.199.201]

TASK: [swarm | Swarm node is running] *****************************************
failed: [10.100.199.201] => {"changed": false, "failed": true}
msg: ConnectionError(MaxRetryError("UnixHTTPConnectionPool(host='localhost', port=None):
no 2] No such file or directory)",),)

FATAL: all hosts have already failed -- aborting

PLAY RECAP ********************************************************************
           to retry, use: --limit @/home/vagrant/infra.retry

10.100.199.200             : ok=19   changed=10   unreachable=0    failed=0
10.100.199.201             : ok=10   changed=7    unreachable=0    failed=1
10.100.199.202             : ok=1    changed=0    unreachable=0    failed=0
10.100.199.203             : ok=1    changed=0    unreachable=0    failed=0
```

gistfile1.txt hosted with ❤ by GitHub view raw

```
$ vagrant ssh swarm-node-01
snip
vagrant@swarm-node-01:~$ sudo tail /var/log/syslog
Jul 14 11:28:58 swarm-node-01 kernel: [ 1612.580460] init: docker main process ended, respawning
Jul 14 11:28:58 swarm-node-01 kernel: [ 1612.677191] init: docker main process ended, respawning
Jul 14 11:28:59 swarm-node-01 kernel: [ 1612.769985] init: docker main process ended, respawning
Jul 14 11:28:59 swarm-node-01 ansible-docker: Invoked with publish_all_ports=False tty=False
insecure_registry=False links=None domainname=None lxc_conf=None image=swarm pid=None
docker_api_version=No
ne tls_client_key=None tls_ca_cert=None password=NOT_LOGGING_PASSWORD hostname=None
docker_url=None use_tls=None state=started tls_client_cert=None dns=None volumes_from=None
net=None email=None usern
ame=None memory_limit=0 env=None expose=None stdin_open=False tls_hostname=None
registry=None detach=True memory_swap=0 count=1 pull=missing name=swarm-node
restart_policy=None privileged=False comman
d=join –addr=10.100.199.201:2375 token://a1e87a8f8b369ba66b94f11fd3443e20
restart_policy_retry=0 volumes=None debug=False ports=None
Jul 14 11:28:59 swarm-node-01 kernel: [ 1612.854461] init: docker main process ended, respawning
Jul 14 11:28:59 swarm-node-01 kernel: [ 1612.911004] init: docker main process ended, respawning
Jul 14 11:28:59 swarm-node-01 kernel: [ 1612.969680] init: docker main process ended, respawning
Jul 14 11:28:59 swarm-node-01 kernel: [ 1613.027900] init: docker main process ended, respawning
Jul 14 11:28:59 swarm-node-01 kernel: [ 1613.084894] init: docker main process ended, respawning
Jul 14 11:28:59 swarm-node-01 kernel: [ 1613.142051] init: docker respawning too fast, stopped
vagrant@swarm-node-01:~$
```

cheesegrits
July 14, 2015 at 6:42 pm

Exactly the same here on node-01. 10 respawns in about a second, then "respawning too fast".

I was about to post about that fmode on the shared folder workaround. Hopefully the PR on ansible that fixes the underlying execute permission issue will get accepted at some point.

Trevorn
July 14, 2015 at 3:48 pm

I had problems with the SSH keys too, then realised I was running an older version of Vagrant. After updating to Vagrant 1.7.3 I could run the ansible script without any problems

albacerrada
July 14, 2015 at 6:35 pm

So – I'm a little perplexed, as the swarm-master runs docker fine, but all the nodes have the above respawning issue.

If I vagrant ssh into a node I can start docker manually without upstart and it works fine?!

Viktor Farcic Post author
July 14, 2015 at 7:24 pm

I just destroyed Vagrant VMs and run it again and it worked. I followed exactly the same steps as in the article until the "Deployment" section. All 9 containers deployed with Swarm are up and running.

The only thing I can think of as the cause of the problem is Vagrant version or Windows itself. The rest is handled by Ansible and installed versions should be the same everywhere.

I'm downloading evaluation version of Windows 8.1 Enterprise. I'm a bit rusty with Windows but I'll give me best to try to figure out the cause of the problem.

albacerrada
July 14, 2015 at 7:59 pm

Great – the main issue I had was with the permissions in the synced_folder which I detail how I got around above. Why docker is playing up on the nodes and not the master though I just don't understand. I took a look at the upstart/ansible config and it all looks fine to me and I take it there's no difference between the master and the nodes or is there?

cheesegrits

July 14, 2015 at 8:19 pm

Did you use the cachier plugin?

Viktor Farcic Post author
July 14, 2015 at 9:39 pm

I just finished creating Windows VM and inside it installed VirtualBox, Vagrant and Git. Than I run "vagrant up" only to discover that VB does not support 64bits VMs inside VM. In other words, I was hoping to reproduce Windows problems by running it inside VM only to realize that Vagrant can not be run like that. Sorry guys… In a month from now I'll be joining a new company and they'll give me a Win. laptop. Until then I don't have a decent way to reproduce this.

cheesegrits
July 14, 2015 at 9:01 pm

Well, I got through the basic install by upgrading to the latest Vagrant, which fixes the ansible executable issue and the ssh key issue. However, I'm still getting issues with docker respawning, and in the startup log on node-01, I'm seeing:

```
vagrant@swarm-node-01:~$ sudo more /var/log/upstart/docker.log
Waiting for /var/run/docker.sock
INFO[0000] Listening for HTTP on unix (/var/run/docker.sock)
ERRO[0000] Udev sync is not supported. This will lead to unexpected behavior, data loss and errors.
For more information, see https:
//docs.docker.com/reference/commandline/cli/#daemon-storage-driver-option
ERRO[0000] 'overlay' not found as a supported filesystem on this host. Please ensure kernel is new
enough and has overlay support lo
aded.
WARN[0000] Running modprobe bridge nf_nat failed with message: insmod /lib/modules/3.13.0-55-
generic/kernel/net/llc/llc.ko
insmod /lib/modules/3.13.0-55-generic/kernel/net/802/stp.ko
insmod /lib/modules/3.13.0-55-generic/kernel/net/bridge/bridge.ko
insmod /lib/modules/3.13.0-55-generic/kernel/net/netfilter/nf_conntrack.ko
insmod /lib/modules/3.13.0-55-generic/kernel/net/netfilter/nf_nat.ko
```

```
, error: exit status 1
/var/run/docker.sock is up
WARN[0000] Your kernel does not support swap memory limit.
```

… which doesn't look like a Good Thing .

cheesegrits
July 14, 2015 at 9:04 pm

Oh, and of course I had to apply the mount_options fix to the Vagrant file. And just FYI, I tried 644 rather than 600, but of course the ssh directives then fail because the private key file located on the share is group/world readable. So it does indeed have to be 600.

cheesegrits
July 14, 2015 at 9:20 pm

Just as a sanity check on the box itself …

```
$ vagrant box update
==> swarm-master: Checking for updates to 'ubuntu/trusty64'
swarm-master: Latest installed version: 20150609.0.10
swarm-master: Version constraints:
swarm-master: Provider: virtualbox
==> swarm-master: Box 'ubuntu/trusty64' (v20150609.0.10) is running the latest version.
```

albacerrada
July 15, 2015 at 12:15 pm

ok – we need to remove the -r=true and one of the -H entries in /etc/default/docker to get this to work. The upstart error makes this clear now. The version of docker has probably changed and the options you can apply like -r and multiple -H is no longer supported.

swarm-master isn't running any extra DOCKER_OPTS and that's why it runs. So this issue is not related to windows from what I can tell. I'll need to try and find the bit in ansible where it does this.

Viktor Farcic Post author
July 15, 2015 at 12:59 pm

Docker does have a tendency to release new versions are not fully compatible with the old ones. However, I'm not sure that is the case in this example. I run everything from zero and it worked. On nodes, Docker 1.7 is installed with /etc/default/docker changed (with -r and two -H). Swarm seems to be working correctly. I deployed books-service with its MongoDB container without any problem. There is no trace of docker respawning in syslog on node-01. The only difference is OS and even that should affect only what's in /vagrant directory. That's the only thing that is shared with the host OS.

---

albacerrada
July 15, 2015 at 12:54 pm

```
$ cat ./ansible/roles/swarm/files/docker.cfg
DOCKER_OPTS="-r=true -H tcp://0.0.0.0:2375 ${DOCKER_OPTS}"
```

I've edited to look like this

```
DOCKER_OPTS="-H unix:// -H tcp://0.0.0.0:2375"
```

I then re-provisioned the nodes

```
$ vagrant provision swarm-node-01 swarm-node-02 swarm-node-03
```

removed the ~/.ssh/known_hosts on swarm-master and kicked off the ansible-playbook

And it works!!!

```
PLAY RECAP ************************************************************************
10.100.199.200 : ok=46 changed=18 unreachable=0 failed=0
10.100.199.201 : ok=25 changed=12 unreachable=0 failed=0
10.100.199.202 : ok=25 changed=12 unreachable=0 failed=0
10.100.199.203 : ok=25 changed=17 unreachable=0 failed=0
```

Viktor Farcic Post author
July 15, 2015 at 1:03 pm

That's great news. It means that "-r" was causing problems (DOCKER_OPTS variable is empty so it should have any affect). I'll change the docker.cfg and start over…

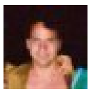
cheesegrits
July 16, 2015 at 12:58 am

```
PLAY RECAP ***********************************************************************
10.100.199.200 : ok=46 changed=27 unreachable=0 failed=0
10.100.199.201 : ok=25 changed=17 unreachable=0 failed=0
10.100.199.202 : ok=25 changed=17 unreachable=0 failed=0
10.100.199.203 : ok=25 changed=17 unreachable=0 failed=0
```

The only gotcha I still found was with vagrant-cachier. For some weird reason I'm getting errors from apt-get about "archive directory /var/cache/apt/archive/partial is missing", even when it's clearly there. I had to uninstall the plugin to get things going.

Thanks, everyone!

Pingback: Required Reading – 2015-07-17 | We Build Devops



Pinco Pallo (@mscavazzin)
August 5, 2015 at 11:01 am

Hello Viktor, I got everything running (with small tweaks because I am behind a corporate proxy) but it looks like everything is running on the swarm-master node. Moreover my output of docker info is not looking like yours but I noticed that it looks a lot like the one that you have when you run docker-machine. By the way I don't understand how you can run a docker swarm without any need of docker-machine ? Am I missing something ? Thank you very much !

Viktor Farcic Post author
August 5, 2015 at 8:27 pm

Docker Machine is a handy way to create virtual machines. It does nothing that you can not do without it. It is only one more tool that can be used to create VMs. I, personally, prefer using Ansible to orchestrate servers because it gives me much more freedom.

Dennis
October 9, 2015 at 5:47 pm

Hello Viktor, I have a issue with a Windows installation. The “ansible-playbook /vagrant/ansible/books-service.yml -i /vagrant/ansible/hosts/prod” command continues to fail with the following error. Any suggestions on how to get around this error would be helpful.

```
…
TASK: [service | Compose template is present] *********************************
changed: [10.100.199.200]

TASK: [service | Service is applied] *********************************************
changed: [10.100.199.200]

TASK: [service | DB container is pulled] ****************************************
failed: [10.100.199.200] => {“changed”: true, “cmd”: “docker-compose pull db”, “delta”:
“0:01:05.085311”, “end”: “2015-10-09 15:38:30.432541”, “rc”: 255, “start”: “2015-10-09
15:37:25.347230”, “warnings”: []}
stderr: Pulling db (mongo:latest)…
Traceback (most recent call last):
File “”, line 3, in
File “/code/build/docker-compose/out00-PYZ.pyz/compose.cli.main”, line 31, in main
File “/code/build/docker-compose/out00-PYZ.pyz/compose.cli.docopt_command”, line 21, in
sys_dispatch
File “/code/build/docker-compose/out00-PYZ.pyz/compose.cli.command”, line 27, in dispatch
File “/code/build/docker-compose/out00-PYZ.pyz/compose.cli.docopt_command”, line 24, in dispatch
File “/code/build/docker-compose/out00-PYZ.pyz/compose.cli.command”, line 59, in
perform_command
File “/code/build/docker-compose/out00-PYZ.pyz/compose.cli.main”, line 231, in pull
File “/code/build/docker-compose/out00-PYZ.pyz/compose.project”, line 221, in pull
File “/code/build/docker-compose/out00-PYZ.pyz/compose.service”, line 518, in pull
File “/code/build/docker-compose/out00-PYZ.pyz/docker.client”, line 735, in pull
File "/code/build/docker-compose/out00-PYZ.pyz/docker.client", line 79, in _post
File "/code/build/docker-compose/out00-PYZ.pyz/requests.sessions", line 425, in post
File "/code/build/docker-compose/out00-PYZ.pyz/requests.sessions", line 383, in request
File "/code/build/docker-compose/out00-PYZ.pyz/requests.sessions", line 486, in send
File "/code/build/docker-compose/out00-PYZ.pyz/requests.adapters", line 394, in send
File "/code/build/docker-compose/out00-PYZ.pyz/requests.models", line 679, in content
File "/code/build/docker-compose/out00-PYZ.pyz/requests.models", line 619, in generate
requests.exceptions.ChunkedEncodingError: IncompleteRead(234 bytes read)

FATAL: all hosts have already failed — aborting

PLAY RECAP ********************************************************************
to retry, use: –limit @/home/vagrant/books-service.retry

10.100.199.200 : ok=23 changed=3 unreachable=0 failed=1

vagrant@swarm-master:~$
```

Viktor Farcic Post author
October 9, 2015 at 8:29 pm

I never saw that error. I might be related to some networking issues. Can you enter the VM and try to pull the image manually?

```
sudo docker pull mongo
```

With that we'll at least discard network as the possible culprit. Also, you can contact me on HangOuts and we can debug it together.

spinebone
October 31, 2015 at 5:15 pm

Would these steps work on dedicated servers as well? Also, do I have to run virtual box on the server as well? This tutorial is based on running it on a personal computer, I just wonder how different it would be on an actual server.

Thanks, amazing knowledge shared

Viktor Farcic Post author
October 31, 2015 at 5:36 pm

Hi,

I used VMs in this article because most of us cannot afford real servers to practice things. In a “real world” situation you would deploy one or more Swarm masters and as many Swarm nodes as the number of servers (minus Swarm master). Ideally, you would not use any VM since it only represents an overhead. Deploying containers directly on server OS is most efficient. However, there are cases when VMs are practical or unavoidable. No matter the case, Swarm works the same no matter whether you’re using your laptop or servers, with or without VMs. You’d need to change few IPs and not much more. Docker is about to come up with very important changes that will make Docker in general and tools around it (including Swarm) much more usable when running a cluster. I will not reveal details until it is officially released (probably during next week).

Peter
November 4, 2015 at 1:04 pm

I don’t understand one part. Everything works properly but my mono instance had been deployed to localhost/booksservice_db_1. Why is that so? Why not to one of 3 nodes? So one of node can be master node?

Viktor Farcic Post author
November 4, 2015 at 1:08 pm

Can you post the complete output of “docker ps -a” and “docker info”?

Peter
November 4, 2015 at 1:12 pm

```
SWARM-MASTER
docker ps -a

0fc7967e4b7f vfarcic/books-service:latest "/run.sh" 9 minutes ago Up 9 minutes
10.100.199.201:32768->8080/tcp swarm-node-01/booksservice_green_1
4aa3d9812d66 vfarcic/books-service:latest "/run.sh" About an hour ago Exited (137) 9 minutes ago
swarm-node-02/booksservice_blue_1
756768faeec2 gliderlabs/registrator "/bin/registrator -ip" About an hour ago Up About an hour
swarm-node-02/registrator-kv
318e25d9514d gliderlabs/registrator "/bin/registrator -ip" About an hour ago Up About an hour
swarm-node-02/registrator
06bca749e57b gliderlabs/registrator "/bin/registrator -ip" About an hour ago Up About an hour
swarm-node-01/registrator-kv
7480363217d2 gliderlabs/registrator "/bin/registrator -ip" About an hour ago Up About an hour
swarm-node-01/registrator
e8c1d788292e swarm "/swarm join –addr=1" About an hour ago Up About an hour 2375/tcp swarm-
node-02/swarm-node
a1ddcaceabc7 swarm "/swarm join –addr=1" About an hour ago Up About an hour 2375/tcp swarm-
node-01/swarm-node
6a3c5a8e00e6 gliderlabs/registrator "/bin/registrator -ip" 4 hours ago Exited (1) 4 hours ago
localhost/registrator-kv
236b357c5527 gliderlabs/registrator "/bin/registrator -ip" 4 hours ago Exited (1) 4 hours ago
localhost/registrator
0664140d53a8 nginx "nginx -g 'daemon off" 4 hours ago Up 4 hours 80/tcp, 443/tcp localhost/nginx
5bc6ff4c1fb5 swarm "/swarm join –addr=d" 4 hours ago Up 4 hours 2375/tcp localhost/swarm-node
bfb3116f0ee6 mongo:latest "/entrypoint.sh mongo" 21 hours ago Up 21 hours
104.237.138.145:32815->27017/tcp localhost/booksservice_db_1
107093d3511c swarm "/swarm join –addr=1" 35 hours ago Up 35 hours 2375/tcp localhost/swarm-
node
47c5854ce45e gliderlabs/registrator "/bin/registrator -ip" 35 hours ago Up 35 hours
localhost/registrator-kv
5e6aa977ec6d gliderlabs/registrator "/bin/registrator -ip" 35 hours ago Up 35 hours
localhost/registrator

Containers: 20
Images: 36
Role: primary
Strategy: spread
Filters: health, port, dependency, affinity, constraint
Nodes: 5
localhost: 104.237.138.145:2375
 └ Containers: 4
 └ Reserved CPUs: 0 / 2
 └ Reserved Memory: 0 B / 2.05 GiB
 └ Labels: executiondriver=native-0.2, kernelversion=4.1.5-x86_64-linode61,
operatingsystem=Ubuntu 14.04.1 LTS, storagedriver=devicemapper
localhost: demo.abide.is:2375
 └ Containers: 4
 └ Reserved CPUs: 0 / 2
 └ Reserved Memory: 0 B / 2.05 GiB
 └ Labels: executiondriver=native-0.2, kernelversion=4.1.5-x86_64-linode61,
operatingsystem=Ubuntu 14.04.1 LTS, storagedriver=devicemapper
swarm-node-01: 10.100.199.201:2375
 └ Containers: 4
 └ Reserved CPUs: 0 / 1
 └ Reserved Memory: 0 B / 1.019 GiB
 └ Labels: executiondriver=native-0.2, kernelversion=3.13.0-66-generic, operatingsystem=Ubuntu
14.04.3 LTS, storagedriver=devicemapper
swarm-node-02: 10.100.199.202:2375
 └ Containers: 4
 └ Reserved CPUs: 0 / 1
 └ Reserved Memory: 0 B / 1.019 GiB
 └ Labels: executiondriver=native-0.2, kernelversion=3.13.0-66-generic, operatingsystem=Ubuntu
14.04.3 LTS, storagedriver=devicemapper
swarm-node-03: 10.100.199.203:2375
 └ Containers: 4
 └ Reserved CPUs: 0 / 1
 └ Reserved Memory: 0 B / 1.019 GiB
 └ Labels: executiondriver=native-0.2, kernelversion=3.13.0-66-generic, operatingsystem=Ubuntu
14.04.3 LTS, storagedriver=devicemapper
CPUs: 7
Total Memory: 7.157 GiB
```

Peter
November 4, 2015 at 4:27 pm

Ok I get this. Actually I don’t but I know why that happened to me. I had machines from previous course “jenkins-docker-ansible” declared in my virtualbox, but they were non running. Probably those two showed up in my master but they were not registerd by swarm and somehow were working locally. That’s a moment I don’t understand how swarm could know about address ip and so on and what’s more I didn’t see mongo service in consul.

Peter
November 4, 2015 at 4:56 pm

Ehh this is more weird. Actually there are still different containers created 8 or even 38 hours ago. I don't know how and why. I cannot remove them, cause containers are not found... I don't have any others virtual machines running aprat from these 4 in the article. I uninstalled vagrant-cachier but this didn't give me anything

---

Viktor Farcic Post author
November 4, 2015 at 5:26 pm

I think that the problem you are experiencing is related to the Swarm Cluster ID defined in https://github.com/vfarcic/docker-swarm/blob/master/ansible/roles/swarm/defaults/main.yml. You should generate a unique ID and recreate the cluster in order to be sure that it does not clash with anything else. Even better, use Consul for that purposes. You can find more info in https://docs.docker.com/swarm/discovery/. If you continue having problems, please contact me on HangOuts (you can find my email in the About section) and we can chat and debug it together.

:)