Hot Topics in ISE - Topic 5
Comparison of Container Schedulers

Armand Grillet

Technische Universität Berlin

Abstract. This report compares three popular solutions to schedule containers: Docker Swarm, Google Kubernetes and Apache Mesos (using the framework Marathon). After explaining the basics of scheduling and containers, it explores the schedulers' features and compare them through two use cases: a website that only needs two containers and a scalable voting application that can accommodate arbitrarily large scale.

Keywords: scheduling, containers, Swarm, Kubernetes, Mesos, Marathon

1 What is scheduling? What are containers?

1.1 Scheduling

A cluster scheduler has multiple goals: using the cluster's resources efficiently, working with user-supplied placement constraints, scheduling applications rapidly to not let them in a pending state, having a degree of "fairness", being robust to errors and always available.

To reach these goals, there are three main scheduler architectures presented in the white-paper concerning Omega, a scalable scheduler for large compute clusters developed by Google [48]:

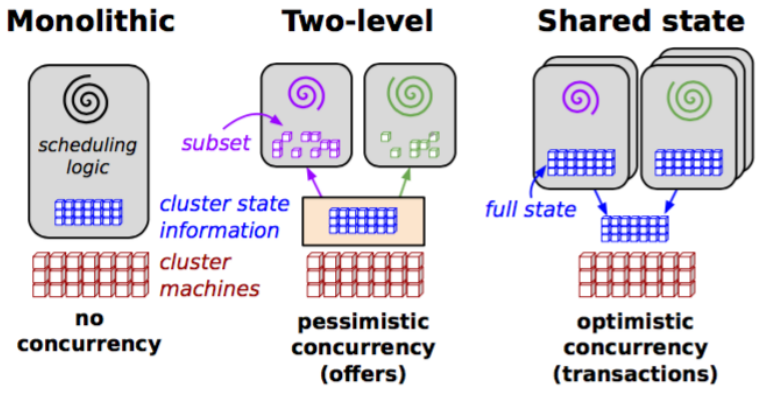

Fig. 1. Schematic overview of scheduling architectures, © Google, Inc. [48]

Monolithic scheduling

Monolithic schedulers are composed of a single scheduling agent handling all the requests, they are commonly used in high-performance computing. A monolithic scheduler generally applies a single-algorithm implementation for all incoming jobs thus running different scheduling logic depending on job types is difficult.

Apache Hadoop YARN [55], a popular architecture for Hadoop that delegates many scheduling functions to per-application components, is a monolithic scheduler architecture due to the fact that the resource requests from application masters have to be sent to a single global scheduler in the resource master.

Two-level scheduling

A two-level scheduler adjusts the allocation of resources to each scheduler dynamically using a central coordinator to decide how many resources each subcluster can have, it is used in Mesos [50] and was used for Hadoop-on-Demand (now replaced by YARN).

With this architecture, the allocator avoids conflicts by offering a given resource to only one framework at a time and attempts to achieve dominant resource fairness by choosing the order and the sizes of the resources it offers. Only one framework is examining a resource at a time thus the concurrency control is called pessimistic, a strategy that is less error-prone but slower compared to an optimistic concurrency control offering a resource to many frameworks at the same time.

Shared-state scheduling

Omega grants each scheduler full access to the entire cluster, allowing them to compete in a free-for-all manner. There is no central resource allocator as all of the resource-allocation decisions take place in the schedulers. There is no central policy-enforcement engine, individual schedulers are taking decisions in this variant of the two-level scheme.

By supporting independent scheduler implementations and exposing the entire allocation state of the schedulers, Omega can scale to many schedulers and works with different workloads with their own scheduling policies [54].

What is the best scheduler?

The development of multiple cluster schedulers by different companies is because there is not an unique solution to solve every problems with cluster computing. Google (main contributor of Omega and Kubernetes) wants an architecture that gives control to the developers, assuming that they respect the rules concerning the priority of their jobs in the cluster, while Yahoo! (main contributor for YARN) wants an architecture that enforces capacity, fairness and deadlines.

These different needs result in different ways to schedule applications. Now that we know how classic schedulers are designed we can put our focus on containers and container schedulers to see what are the needs in this domain and the solutions offered.

1.2 The container revolution

Containers are an alternative to virtual machines for helping developers to build, ship, deploy, and instantiate applications [3]. A container is a set of processes that are isolated from the rest of the machine encapsulating its dependencies.

Containers run in isolation, sharing an operating system (OS) instance. They do not need an entire guest operating system, making them way lighter by an order of magnitude compared to virtual machines.

As they can start in a matter of seconds, more quickly than virtual machines, containers are made to take a limited amount of resources (less than 2GB of RAM) and scale to satisfy the demand. Containers are often used in micro-services architectures where each container represents a service, connected to the other services through the network. This architecture allows each component to be deployed and scaled independently of the others.

This amount of resources used and the expected lifespan of containers are the main differences for a normal scheduler and a container scheduler. Whereas the design of traditional clusters like Hadoop is focused on running massive jobs [55], container clusters will run dozens of small instances that need to be organized and networked to optimize how they share data and computational power.

Docker

Docker is a popular type of containers that was first based on LinuX Containers (LXC) [2] but is now powered by runC [46], a CLI tool for spawning and running containers built by the Open Containers Initiative [36]. Docker containers have a layered file-system to share the operating system kernel of the host operating system. This feature means that even if your Docker image is based on an operating system of 1GB, running ten instances on the same host it will not take 10GB contrary to a virtual machine that needs ten times a full guest OS.

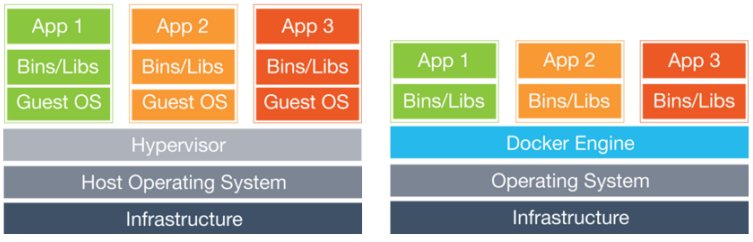

Fig. 2. Virtual machines architecture compared to containers, © Docker Inc. [23]

Docker is based on images, i.e. a snapshot of an operating system. To create a new image you start with a base image, make your changes and commit them. It can be shared on public or private registries [10] to be used by other developers who just need to pull the image.

Images are extremely convenient to create snapshots of the operating system and then use them to create new containers, they are lightweight and easy to use and share as it is the core of Docker CLI and business model [12]. Containers contain everything they need to run (code, runtime, system tools and libraries), Docker guarantees a lightweight and stable environment to create and run jobs quickly.

2 Description of container schedulers

The main task of a container scheduler is to start containers on the most appropriate host and connect them together. It has to handle failures by doing automatic fail-overs and it needs to be able to scalable containers when there is too many data to process/compute for a single instance.

This report compare three popular container schedulers: Docker Swarm [13], Apache Mesos (running the Marathon framework) [50] and Google Kubernetes [31]. This section describes each scheduler's design and features.

2.1 Docker Swarm

Docker Swarm is a container scheduler developed by Docker. The development of this cluster solution being managed by Docker offers advantages like the use of the standard Docker API [17]. The architecture of a swarm is composed of two elements:

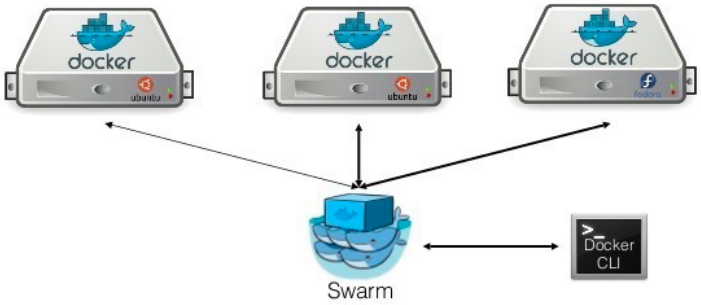

Fig. 3. Docker Swarm architecture, © Alexandre Beslic (Docker Inc.) [14]

One machine plays the role of the manager, it runs an image of Swarm (like it would run any other Docker images), it is represented on the image by the whales

and is responsible for scheduling the containers on the agents [4]. Swarm uses the same API as the Docker standard API, this means that running a container on a Swarm requires the same command as running it on a single machine. New flags are available but a developer can use a Swarm cluster without changing its workflow.

The swarm is also composed of multiple agents, also called nodes, that are just machines having a Docker remote API available for the Swarm manager by opening the correct ports when starting the Docker daemon [5]. Three of them are represented on the image. These machines will pull and run different images depending on the containers that the swarm scheduler assigns to them.

Each agent can have labels assigned when starting the docker daemon, these labels are key/value pairs giving details concerning the agent. These labels are used to filter the cluster when running a new container, further details are given later in this part.

There is containers in Swarm is possible by specifying a global strategy when starting the manager or specifying filters when running a new container.

Strategies

Three strategies (e.g. ways for the manager to select a node to run a container) are available in Swarm [22]:

Strategy name	Node selected
spread	Has the fewest containers, disregarding their states
binpack	Most packed (i.e. has the minimum amount of free CPU/RAM)
random	Chosen randomly

If multiple nodes are selected by the strategy, the scheduler chooses a random node among those. The strategy needs to be defined when starting the manager or the "spread" strategy will be used by default.

Filters

To schedule containers on a subset of nodes, Swarm offers two node filters (constraint and health), and three container configuration filters (affinity, dependency and port).

Constraint filter

Key/value pairs are associated to particular nodes. To tag a node with a specific set of key/value pairs you must pass a list of `--label` options when the docker daemon is starting on the node. When running a container in a production environment you can then specify the constraints required, a container will only

start on a node tagged with the key value environment:prod. If no nodes are meeting the requirements, the container will not be started.

A standard set of constraints is available like the node's operating system, you do not need to specify them when starting the nodes [19].

Health filter

The health filter prevents scheduling containers on unhealthy nodes. Little information is available concerning the notion of what is a healthy node even after reading Swarm source code [21].

Affinity filter

The affinity filter is here to create "attractions" when running a new container. Three types of affinities exist concerning containers, images and labels.

For containers, you only need to give the name of the container (or container's ID) you want to be linked with when running the new one and they will run next to each other. If one container ends up, all the containers linked to it will also stop.

The image affinity schedules a container to run on nodes where a specific image is already pulled.

The label affinity works with containers' labels. To schedule a new container next to a specific container the only affinity required when running the new one is `affinity:container>==<container_name>`.

Affinities and constraints use a syntax that accept negation and soft enforcement to run a container even if it is not possible to meet all the requirements specified by the filters and constraints [18].

Dependency filter

A dependency filter can be used to run a container that is dependent of another container. Being dependent means having a shared volume with another container, needing to be linked to another container or being on the same network stack.

Port filter

If you want to run a container on a node that has a specific free port you can use this filter. If there is no agents with this port available in the Swarm cluster an error message will appear.

2.2 Apache Mesos & Mesosphere Marathon

The purpose of Mesos is to build a scalable and efficient system that supports a wide array of both current and future frameworks. This is also the main issue:

frameworks like Hadoop and MPI are developed independently thus it is not possible to do fine-grained sharing across frameworks [35].

The proposition of Mesos is to add a thin resource-sharing layer giving frameworks a common interface for accessing clusters resources. Mesos is delegating control over scheduling to the frameworks because many frameworks already implement sophisticated scheduling.

Four types of frameworks exist depending on the type of jobs you want to run on your cluster [52] and some of them have native Docker support like Marathon [39]. The support of Docker containers has been added in Mesos 0.20.0 [51].

We will focus on the use of Mesos with Marathon because it is a framework actively maintained by Mesosphere [41] that offers many features in terms of scheduling such as constraints [38], health checks [40], service discovery and load balancing [42].

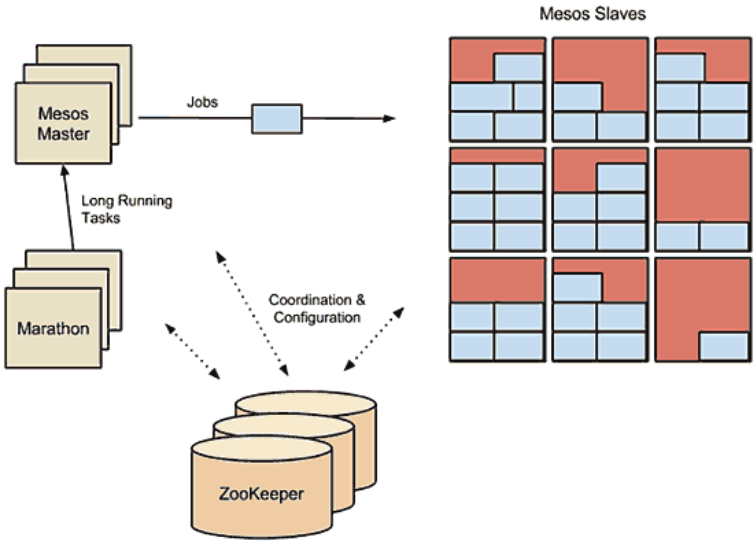

Fig. 4. Apache Mesos architecture using Marathon, © Adrian Mouat [49]

As we can see on the image, there are four elements in the cluster. ZooKeeper helps Marathon to look up the address of the Mesos master [53], multiple instances are available to handle failure. Marathon starts, monitors and scales the containers. The Mesos master sends the tasks assigned to a node and make offers to Marathon when a node has some free CPU / RAM. The Mesos slaves run the container and submit a list of their available resources.

Constraints

Constraints give operators control over where apps should run, they are made up of three parts: a field name (can be the slave hostname or any Mesos slave attribute), an operator, and an optional value. Five operators exist:

Operator	Role
UNIQUE	Forces attributes' uniqueness, e.g. the constraint `["hostname", "UNIQUE"]` ensures that there is only one application task running on each host.
CLUSTER	Run your application on slaves that share a certain attribute, e.g. the constraint `["rack_id", "CLUSTER", "rack-1"]` obliges the application to run on rack-1 or to be in a pending state to wait for some free CPU/RAM on rack-1.
GROUP_BY	Evenly spread your application across nodes with a particular attribute like a specific host or rack.
LIKE	Ensures that the app runs only on slaves having a certain attribute. When there is only one value it is working like CLUSTER but many values can be matched because the argument is a regular expression.
UNLIKE	Does the contrary of LIKE

Constraints can be combined as the constraints parameter when running a new application on Mesos is an array.

Health checks

Health checks are application dependent and needs to be implemented manually as only the developers know what constitutes a healthy state in your app (a difference between Swarm and Mesos).

Many options are provided to declare the number of seconds to wait between each health check or the number of consecutive health check failures after which the unhealthy task should be killed.

Service discovery and load balancing

To send data to the running applications, we need to have a service discovery. Apache Mesos offers a DNS based discovery called Mesos-DNS [44] that works in clusters composed of multiple frameworks (not just Marathon).

For a cluster that is only composed of nodes running containers, Marathon is enough to manage it. In that case each host can run a TCP proxy to forward connections to the static service port to the individual app instances. Marathon will ensure that all dynamically assigned service ports by the are unique which is better than doing it manually because multiple containers running the same image thus needing the same port can be on the same host.

Marathon offers two TCP/HTTP proxy. A simple shell script [37] and a more complex one called marathon-lb that has more features [43].

2.3 Google Kubernetes

Kubernetes is an orchestration system for Docker containers using the concepts of "labels" and "pods" to group containers into logical units. Pods are the main difference between Kubernetes and the two other solutions, they are collections of co-located containers forming a service deployed and scheduled together [28]. This approach simplifies the management of the cluster comparing to an affinity-based co-scheduling of containers (like Swarm and Mesos).

The Kubernetes scheduler's task is to watch for pods having an empty `PodSpec.NodeName` value and attribute them a value to schedule the container somewhere in the cluster [32]. This is a difference compared to Swarm and Mesos as Kubernetes allows developers to bypass the scheduler by defining a `PodSpec.NodeName` when running the pod.

The scheduler uses predicates [29] and priorities [30] to define on which nodes a pod should run. The default values of these parameters can be overridden using a new scheduler policy configuration [33].

By using the command-line flag `--policy-config-file` pointing to a JSON file (appendix A) describing the predicates and priorities to use when starting Kubernetes, the scheduler will use the policy defined by the administrator.

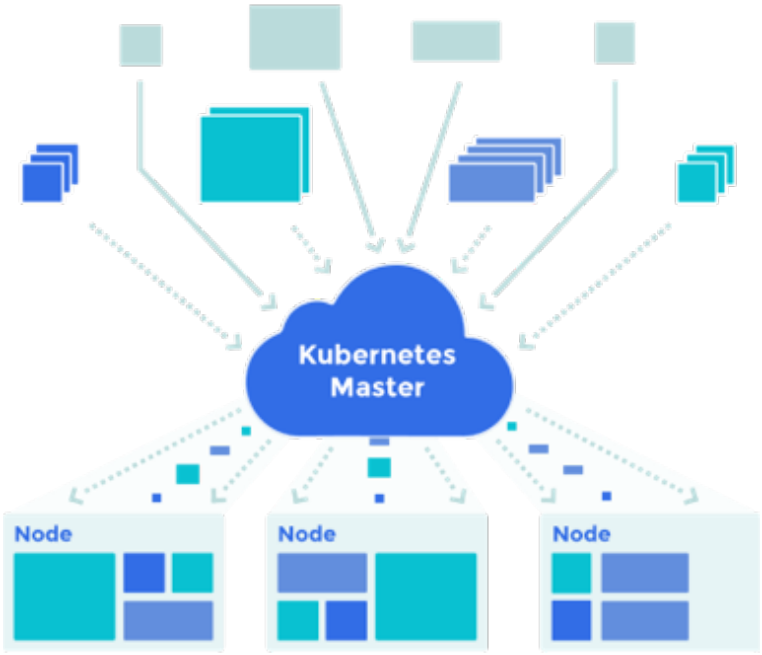

Fig. 5. Kubernetes architecture (containers in grey, pods in color), © Google Inc. [31]

Predicates

Predicates are mandatory rules to schedule a new pod on the cluster. If no machine corresponds to the predicates required by the pod it will remain in pending state until a machine can satisfy them. The predicates available are:

Predicate	Node's requirements
PodFitsPorts	Needs to be able to host the pod without any port conflicts.
PodFitsResources	Has enough resources to host the pod.
NoDiskConflict	Has enough space to fit the pod and the volumes linked.
MatchNodeSelector	Match the selector query parameter defined in the pod's description
HostName	Has the name of the host parameter defined in the pod's description.

Priorities

If the scheduler finds multiple machines fitting the predicates the priorities will then be used to find what is the most suitable machine to run the pod. A priority is a key/value representing the name of the priority from the list of existing ones and its weight (importance of the priority). The priorities available are:

Priority	Node(s) considered as the best(s)
LeastRequestedPriority	Calculates the percentage of memory and CPU requested by the pods that are already on the node. The node with the minimum percentage is the best.
BalancedResourceAllocation	Nodes that have a similar memory and CPU usage.
ServiceSpreadingPriority	Prefers the nodes that have different pods using them.
EqualPriority	Only used for testing, give an equal priority to all the nodes in the cluster.

2.4 Conclusion

The three containers described offer different features and rules to personalize the scheduler's logic. It already appears from this section that Swarm is the simplest to use due to its API that can work like Docker works on a single machine.

Running containers the Docker way [15] means thinking that containers should be ephemeral and that they should run only one process per container.

With this philosophy it appears extremely common to have multiple containers working together to offer a service or to represent an application.

Being able to orchestrate and schedule containers is thus a bigger priority than when handling regular applications; it explains why there are already so many schedulers available offering different features and options even if the technology is fairly young.

3 Comparison of container schedulers

3.1 Applications used to compare the schedulers

We have seen in the last section that container schedulers are necessary in many cases as containers have to work together in order to be real services.

We will first review each scheduler in a simple case with two containers running in a scheduler. We will use the sample project from the Tutorial Docker for Beginners to run a food trucks website [47] and how to deploy it depending on the cluster.

We will then compare the deployment of schedulers at scale with another example: a voting application running on top of Amazon Web Services (AWS). This example is based a tutorial made by Docker to "try Swarm at scale"[7]. Every component of the application run inside of containers, the containers are in different nodes and the application's design is made to scale if needed:

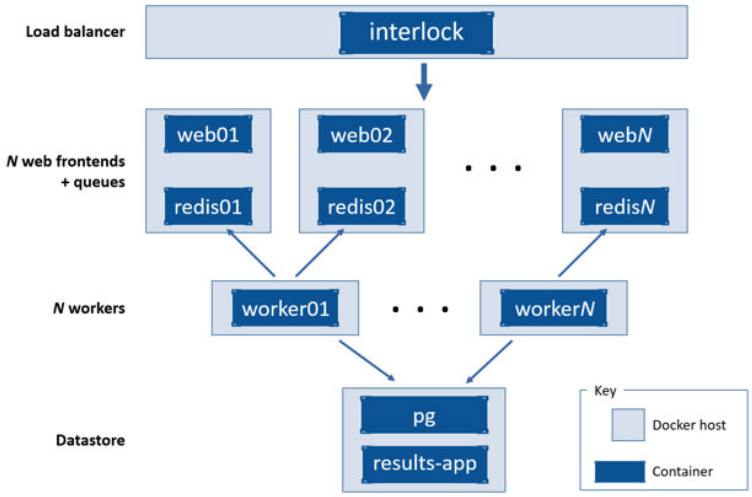

Fig. 6. Voting application architecture, © Docker Inc.

The load balancer manages a scalable number of web servers and associates queues running a Flask[1] application. The worker tier scans the queues of the

Redis containers, dequeues the votes and commits the deduplicated votes to a Postgres container running on another node.

3.2 Food Trucks Application

The features compared in this section are the default configuration of each scheduler, the management of a bottleneck in one container due to a spike in users' demand and how is handle a container that needs to be restarted.

The multi-container environment we want to run on the cluster is composed of one container running a personalized Flask process [1] and another running the Elasticsearch (ES) process [24].

Swarm

Docker Inc. offers multiple tools, we have already seen the engine and Swarm but the key for multi-containers environments is Docker Compose. With this tool we can define and run multi-container Docker applications using only one file describing it.

For our example, we use a `docker-compose.yml` specifying the two images that need to run (a custom image for Flask and the Elasticsearch image) with a link between them directly with Swarm.

The main problem is that Swarm can build an image from a Dockerfile just like a single-host Docker instance can, but the resulting image will only live on a single node and won't be distributed to the other nodes of the cluster [9]. Thus the application is considered as one container, an approach that is not fine-grained.

If we are scaling one of the container part of the `docker-compose.yml` (appendix B) afterwards using `docker-compose scale`, the new containers will be scheduled following the scheduler rules.

Docker Swarm does not auto-scale containers if they're too much used, something that can is a problem in our use case thus we need to check regularly if the number of users viewing the websites is not creating a bottleneck.

If a container fails, Swarm will not keeps track of how many instances of one services are supposed to be running thus it will not create a new container. It is also not able to roll an update of some containers in the clusters, a feature that could be extremely useful as the Docker philosophy is to be able to do fast stop/start of stateless containers.

Mesos & Marathon

Instead of using directly the `docker-compose.yml` file, we need to give a specific definition made for Marathon. The first definition for Elasticsearch (appendix C) does not use scheduler features, letting everything to default; in that case the definition is simple and similar to what we have in the previous `docker-compose.yml`. The definition of Flask (appendix D) uses more features from Mesos, with CPU/RAM information specified and health checks.

Mesos offers a more powerful definition file as the container is described as well as its requirements in the cluster. Compared to Swarm, the approach is less simple but it easy to scale the service, e.g. by changing the instances of a container in the app definition (set to 2 in the Flask app definition).

Kubernetes

Kubernetes use another type of description to describe a pod in YAML or JSON (appendix E). It includes a `ReplicationController` that makes sure that at least one instance of the app is always running. Once we create the app pod in a Kubernetes cluster, we need to create a Kubernetes service that is a named load balancer that proxies traffic to one or more containers. Using a service with only one instance is still useful as it gives us a deterministic way to route to the pod using an elastic IP.

Compared to Swarm, Kubernetes adds a logic of pods and replicas. This more complex architecture offers new possibilities for the scheduler and the orchestration tool such as load balancing and the possibility to scale the app up or down. You can also deliver updates to your running instances of your app using Kubernetes, a useful feature following the Docker philosophy.

Conclusion

Docker Compose is the standard way to prepare the scheduling of multi-containers environments and Docker swarm can use it directly. For Mesos and Kubernetes a different description file combining this standard description and adding information is needed but it offers a better scheduling in our use case.

With Mesos we can see that the scheduler would not have an issue to work with something else than containers, making it a more powerful solution. However, for the use case defined at the beginning of the section, the best way to run the micro-services architecture would be Kubernetes. By combining a description very similar to a Compose description file and offering replication controllers, Kubernetes is the best way to start a reliable service and scale it if needed.

3.3 Voting application

The architecture deployed is the same as the one proposed by Docker in its tutorial [7] with small variations depending on the scheduler. To host the cluster we create an Amazon Virtual Public Cloud (VPC) and deploy the nodes in the VPC. We use Amazon Web Services because it supports the three schedulers and Docker offers a deployment file to start a dockerised cluster using it [8].

Swarm

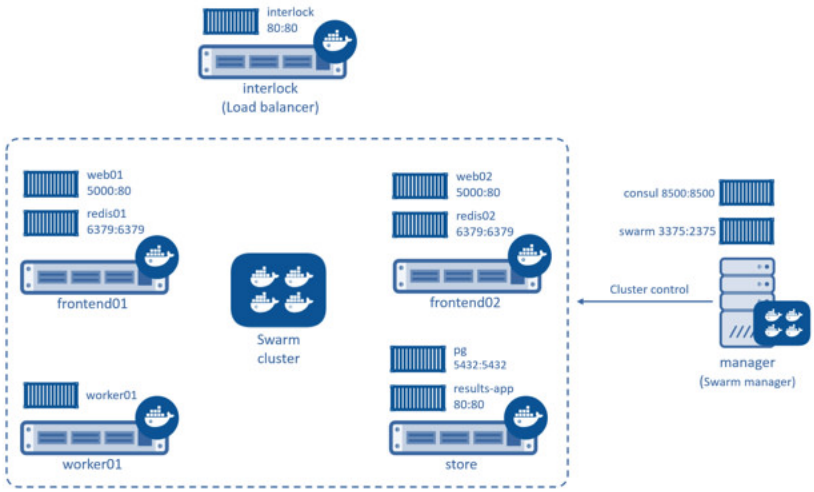

Fig. 7. Voting application detailled organization, © Docker Inc.

Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS	Public IP
manager	i-0a40e7c9	t2.micro	us-west-1c	running	2/2 checks ...	None	ec2-54-183-184-183.us...	54.183.184.183
interlock	i-0c40e7cf	t2.micro	us-west-1c	running	2/2 checks ...	None	ec2-52-53-220-140.us-...	52.53.220.140
store	i-0d40e7ce	m3.medium	us-west-1c	running	2/2 checks ...	None	ec2-52-53-221-193.us-...	52.53.221.193
frontend02	i-124fe8d1	t2.micro	us-west-1c	running	2/2 checks ...	None	ec2-54-67-107-61.us-w...	54.67.107.61
worker01	i-874ee944	t2.micro	us-west-1c	running	2/2 checks ...	None	ec2-52-53-233-214.us-...	52.53.233.214
frontend01	i-b14fe872	t2.micro	us-west-1c	running	2/2 checks ...	None	ec2-52-53-241-12.us-w...	52.53.241.12

Fig. 8. EC2 machines to run the scalable applications in a cluster.

The main steps to create the cluster are joining all the nodes of the cluster, creating of an overlay network to easily communicate between nodes (similar to what Kubernetes offers automatically using pods) and the deployment of the application by running images on specific containers [1].

We use the command-line flag `--restart=unless-stopped` to allow the docker daemon running on the manager to restart a container if it unexpectedly stops. If an entire node fails its containers will not be restarted in another node.

The cluster has a load balancer [25] to stop routing requests to a node that does not exist anymore thus if `frontend1` fails all the requests will go to `frontend2`. As the load balancer is itself a container, running it with the flag `--restart=unless-stopped` makes sure that it will restart if it fails.

The main issue with the cluster deployed is that, as the Postgres node is not replicated, if the machine fails then the entire application will be down. To improve the fault tolerance of the cluster we should also add another manager for the Swarm scheduler, in case the first one crashes.

The efficiency of the scheduler to schedule containers on the cluster is similar to running the containers directly on a single machine. The scheduling strategies

[1] https://gist.github.com/ArmandGrillet/c928a1df55fccd17e36e

of Swarm are simple (as we have seen on the previous section) thus the scheduler selects nodes nearly as fast as if there was only one machine. Its performances can be seen in the Docker scale testing of Swarm by running 30,000 containers [16].

Containers in the clusters	Scheduler delay
15000	230ms
27000	250ms
29700	400ms

Mesos & Marathon

Mesos & Marathon is supported by commercial solutions that are also offering deployment services. Mesosphere [45] offers a community edition (i.e. the solution with no support from the company) to create a highly available cluster in minutes. We only need to give the number of masters, public agent nodes and private agent nodes and a Mesos cluster will be deployed on a cloud provider.

The configuration of a Mesos cluster being more complicated than Swarm due to the number of elements in it (a Mesos marathon, Mesos slaves, Marathon and Zookeeper instances), pre-configured clusters are a good way to start and the availability to directly start with a high-availability cluster (with three masters) is an advantage to create a fault-tolerant cluster.

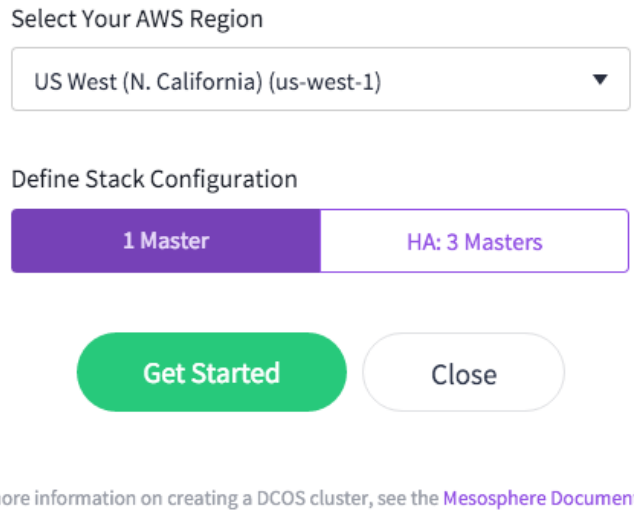

Fig. 9. Deploying a cluster using Mesos & Marathon can be done in 2 minutes.

Once the cluster is started, its state can be analyzed through a Web interface provided by the Mesos master giving all the information about the cluster. Running containers on the cluster requires similar operation as on Swarm and for the previous example.

The fault tolerance is better than Swarm as Mesos uses health checks that can be described in the JSON representing the application we run. The cluster representing the application can not scale automatically as auto-scaling rules are fairly business-specific but solutions such as Amazon EC2 Auto Scaling Groups exist to do it.

Kubernetes

Kubernetes has a command line administration tool and a cluster start-up script to start the cluster. Kubernetes offers a user interface [34] similar to what Mesosphere offers but it is directly part of the scheduler.

We need to create a replication controller which defines the pod's containers and the minimum number of pods to run at any given time. As we have seen in the first example, we can describe the entire cluster with information about the replications in one description file.

The cluster can be scaled using the scheduler's policies and Google states that Kubernetes supports clusters up to 100 nodes with 30 pods per node with 1-2 containers per pod [27]. The performance of the scheduler is worse than Swarm due to its more complex architecture and Mesos as it is less bare metal but companies are working on improving it [2].

4 Future work

There are no of benchmarks available to review their scalability due to the novelty of the schedulers compared. Future work should add a comparison of the schedulers in an environment requiring a high number of connections between clusters that fail often.

The schedulers compared are mainly used to create scalable Web services. This use case requires an important fault tolerance that exists in all the solutions compared but the speed of the schedulers when handling thousands of containers is not documented. For example, there is no exact numbers concerning the scalability of Mesos & Marathon, the only use cases about it gives examples with a cluster composed of 80 nodes and 640 virtual CPUs[3].

All the schedulers compared in this report are actively supported by commercial companies, a benchmark implying a large number of nodes and applications running at the same time could be done in an impartial manner, comparing the same use cases with the same hardware used to host the containers. This new benchmark would give information about the possibility to use containers schedulers for other use cases such as batch processing.

[2] https://coreos.com/blog/improving-kubernetes-scheduler-performance.html
[3] https://mesosphere.com/blog/2015/11/30/arangodb-benchmark-dcos/

5 Conclusion

Docker Swarm is the simplest scheduler to use, with comprehensible strategies and filters, it does not provide features to handle failures of the nodes making it not recommendable in a production environment. Swarm is perfectly integrated in the Docker environment, it uses the same API as the Docker engine and works with Docker Compose description files thus it can be used by developers that are not familiar with scheduling solutions (such as Mesos frameworks).

Docker Swarm is lightweight and offers several drivers to use it with all the trending clusters solutions [11]. It is a scheduling solution that is easy to star with and offers a high degree of configuration for developers that want very specific workflows. Docker Swarm is not attached to a specific cloud provider, is fully open-source and has a strong community.

Using Mesos with Marathon is an excellent combination if you already have a Mesos cluster. The fact that it works like any Mesos framework when scheduling a new task and has a description file like Docker Compose to specify tasks makes it a great way for developers to start using containers in a cluster. The complete solution offered by Mesosphere [45] is also a simple and powerful way to have a container scheduler used for production.

Kubernetes has a logic that is different from the standard Docker philosophy but its concept of pods and services is an interesting way to use containers while still thinking about what their combinations with others. The fact that Google uses this solution and provides easy ways to use Kubernetes on its cluster solution [26] makes it a logical choice for developers already using the Google ecosystem.

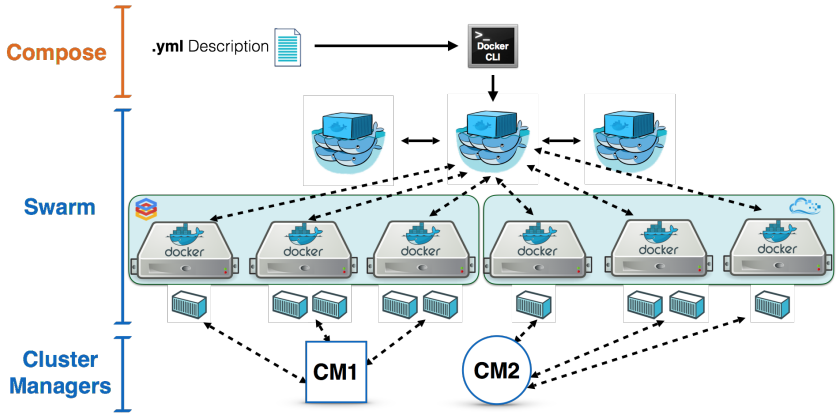

Fig. 10. Swarm frontends, © Docker Inc. [6]

There is no best solution for scheduling containers, the ones reviewed are not even in real competition as we can see with Swarm frontends [20], a project

allowing the deployment of Kubernetes and Mesos + Marathon on top of Swarm that will soon allow Cloud Foundry, Nomad, and more container schedulers to work on top of Swarm. Depending on your needs and your cluster, you should be able to select a scheduler easily.

As you can see in the last figure, a Swarm cluster can be managed by other managers such as Mesos & Marathon and Kubernetes with the containers dispatched on different clusters. These combinations allow containers to be scheduled and orchestrated as you want.

A Appendix - policy-config-file.json

```
{
  "kind": "Policy",
  "apiVersion": "v1",
  "predicates": [
  {
    "name": "PodFitsPorts"
  },
  {
    "name": "PodFitsResources"
  },
  {
    "name": "NoDiskConflict"
  },
  {
    "name": "MatchNodeSelector"
  },
  {
    "name": "HostName"
  }],
  "priorities": [
  {
    "name": "LeastRequestedPriority",
    "weight": 1
  },
  {
    "name": "BalancedResourceAllocation",
    "weight": 1
  },
  {
    "name": "ServiceSpreadingPriority",
    "weight": 1
  },
  {
    "name": "EqualPriority",
```

```
    "weight": 1
  }]
}
```

B Appendix - docker-compose.yml

```
es:
  image: elasticsearch
  container_name: "es"
web:
  image: prakhar1989/foodtrucks-web
  command: python app.py
  ports:
    - "5000:5000"
  volumes:
    - .:/code
```

C Appendix - Mesos app definition (Elasticsearch)

```
{
  "id": "es",
  "container": {
    "type": "DOCKER",
    "docker": {
      "network": "HOST",
      "image": "elasticsearch"
    }
  }
}
```

D Appendix - Mesos app definition (Flask)

```
{
  "id": "web",
  "cmd": "python app.py",
  "cpus": 0.5,
  "mem": 64.0,
  "instances": 2,
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "prakhar1989/foodtrucks-web",
      "network": "BRIDGE",
      "portMappings": [
        { "containerPort": 5000, "hostPort": 0, "servicePort": 5000, "protocol":
```

```
      ]
    },
    "volumes": [
      {
        "containerPath": "/etc/code",
        "hostPath": "/var/data/code",
        "mode": "RW"
      }
  },
  "healthChecks": [
    {
      "protocol": "HTTP",
      "portIndex": 0,
      "path": "/",
      "gracePeriodSeconds": 5,
      "intervalSeconds": 20,
      "maxConsecutiveFailures": 3
    }
  ]
}
```

E Appendix - Kubernetes pod definition

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: app
  labels:
    name: app
spec:
  replicas: 1
  selector:
    name: app
  template:
    metadata:
      labels:
        name: app
    spec:
      containers:
      - name: es
        image: elasticsearch
        ports:
        - containerPort: 6379
      - name: web
        image: prakhar1989/foodtrucks-web
        command:
```

```
  - python app.py
volumeMounts:
- mountPath: /code
  name: code
ports:
- containerPort: 5000
```

References

[1] Armin Ronacher. *Flask, a microframework for Python*. 2014. URL: http://flask.pocoo.org/ (visited on 01/23/2016).

[2] Canonical. *Linux Containers*. 2015. URL: https://linuxcontainers.org/ (visited on 01/21/2016).

[3] Cisco; Red Hat. "Linux Containers : Why They' re in Your Future and What Has to Happen First". In: (2014). URL: http://www.cisco.com/c/dam/en/us/solutions/collateral/data-center-virtualization/openstack-at-cisco/linux-containers-white-paper-cisco-red-hat.pdf.

[4] Docker Inc. *Create a swarm for development - configure a manager*. 2015. URL: https://docs.docker.com/swarm/install-manual/#configure-a-manager (visited on 01/13/2016).

[5] Docker Inc. *Create a swarm for development - create swarm nodes*. 2015. URL: https://docs.docker.com/swarm/install-manual/#create-swarm-nodes (visited on 01/15/2016).

[6] Docker Inc. *Deploy and Manage Any Cluster Manager with Docker Swarm*. 2015. URL: https://blog.docker.com/2015/11/deploy-manage-cluster-docker-swarm/ (visited on 01/23/2016).

[7] Docker Inc. *Docker Compose*. 2015. URL: https://docs.docker.com/swarm/swarm_at_scale/ (visited on 02/21/2016).

[8] Docker Inc. *Docker Compose*. 2015. URL: https://github.com/docker/swarm-microservice-demo-v1/blob/master/AWS/cloudformation.json (visited on 02/21/2016).

[9] Docker Inc. *Docker Compose - Swarm*. 2015. URL: https://github.com/docker/compose/blob/master/SWARM.md (visited on 01/23/2016).

[10] Docker Inc. *Docker Hub*. 2015. URL: https://hub.docker.com/explore/ (visited on 01/21/2016).

[11] Docker Inc. *Docker Machine - Drivers*. 2015. URL: https://docs.docker.com/machine/drivers/ (visited on 01/21/2016).

[12] Docker Inc. *Docker Pricing*. 2015. URL: https://www.docker.com/pricing (visited on 01/21/2016).

[13] Docker Inc. *Docker Swarm*. 2015. URL: https://github.com/docker/swarm (visited on 01/15/2016).

[14] Docker Inc. *Docker Swarm*. 2015. URL: http://www.slideshare.net/Docker/docker-swarm-020 (visited on 01/15/2016).

[15] Docker Inc. *Docker Swarm*. 2015. URL: https://docs.docker.com/engine/articles/dockerfile_best-practices/ (visited on 01/21/2016).

[16] Docker Inc. *Docker Swarm*. 2015. URL: https://blog.docker.com/2015/11/scale-testing-docker-swarm-30000-containers/ (visited on 01/15/2016).
[17] Docker Inc. *Docker Swarm API*. 2015. URL: https://docs.docker.com/swarm/api/swarm-api/ (visited on 01/15/2016).
[18] Docker Inc. *Docker Swarm filters - how to write filter expressions*. 2015. URL: https://docs.docker.com/swarm/scheduler/filter/#how-to-write-filter-expressions (visited on 01/15/2016).
[19] Docker Inc. *Docker Swarm filters - use a constraint filter*. 2015. URL: https://docs.docker.com/swarm/scheduler/filter/#use-a-constraint-filter (visited on 01/15/2016).
[20] Docker Inc. *Docker Swarm frontends*. 2015. URL: https://github.com/docker/swarm-frontends (visited on 01/23/2016).
[21] Docker Inc. *Docker Swarm - health.go*. 2015. URL: https://github.com/docker/swarm/blob/master/scheduler/filter/health.go (visited on 01/15/2016).
[22] Docker Inc. *Docker Swarm strategies*. 2015. URL: https://docs.docker.com/swarm/scheduler/strategy/ (visited on 01/15/2016).
[23] Docker Inc. *How is Docker different from virtual machines?* 2015. URL: https://www.docker.com/what-docker (visited on 01/21/2016).
[24] Elasticsearch Inc. *Elasticsearch*. 2016. URL: https://www.elastic.co (visited on 01/23/2016).
[25] Evan Hazlett. *Interlock*. 2015. URL: https://github.com/ehazlett/interlock (visited on 02/21/2016).
[26] Google. *Google Container Engine*. 2015. URL: https://cloud.google.com/container-engine/docs/ (visited on 01/23/2016).
[27] Google. *Kubernetes Large Cluster*. 2015. URL: http://kubernetes.io/v1.1/docs/admin/cluster-large.html (visited on 01/15/2016).
[28] Google. *Kubernetes - pods*. 2015. URL: http://kubernetes.io/v1.1/docs/user-guide/pods.html (visited on 01/15/2016).
[29] Google. *Kubernetes predicates*. 2015. URL: https://github.com/kubernetes/kubernetes/blob/release-1.1/plugin/pkg/scheduler/algorithm/predicates/predicates.go (visited on 01/15/2016).
[30] Google. *Kubernetes priorities*. 2015. URL: https://github.com/kubernetes/kubernetes/blob/release-1.1/plugin/pkg/scheduler/algorithm/priorities/priorities.go (visited on 01/15/2016).
[31] Google. *Kubernetes scheduler*. 2015. URL: http://kubernetes.io (visited on 01/15/2016).
[32] Google. *Kubernetes scheduler*. 2015. URL: http://kubernetes.io/v1.1/docs/devel/scheduler.html (visited on 01/15/2016).
[33] Google. *Kubernetes scheduler policy config*. 2015. URL: http://kubernetes.io/v1.1/examples/scheduler-policy-config.json (visited on 01/15/2016).
[34] Google. *Kubernetes User Interface*. 2015. URL: http://kubernetes.io/v1.1/docs/user-guide/ui.html (visited on 01/15/2016).

[35] Benjamin Hindman et al. "Mesos: A platform for fine-grained resource sharing in the data center". In: (2011). URL: https://www.cs.berkeley.edu/~alig/papers/mesos.pdf.

[36] Linux Foundation. *Open Containers Initiative*. 2015. URL: https://www.opencontainers.org/ (visited on 01/21/2016).

[37] Mesosphere. *Marathon - bridge*. 2015. URL: https://github.com/mesosphere/marathon/blob/master/bin/haproxy-marathon-bridge (visited on 01/15/2016).

[38] Mesosphere. *Marathon - constraints*. 2015. URL: https://mesosphere.github.io/marathon/docs/constraints.html (visited on 01/15/2016).

[39] Mesosphere. *Marathon - GitHub repository*. 2015. URL: https://github.com/mesosphere/marathon (visited on 01/15/2016).

[40] Mesosphere. *Marathon - health checks*. 2015. URL: https://mesosphere.github.io/marathon/docs/health-checks.html (visited on 01/15/2016).

[41] Mesosphere. *Marathon - releases*. 2015. URL: https://github.com/mesosphere/marathon/releases (visited on 01/15/2016).

[42] Mesosphere. *Marathon - service discovery and load balancing*. 2015. URL: https://mesosphere.github.io/marathon/docs/service-discovery-load-balancing.html (visited on 01/15/2016).

[43] Mesosphere. *Marathon-lb*. 2015. URL: https://github.com/mesosphere/marathon-lb (visited on 01/15/2016).

[44] Mesosphere. *Mesos DNS*. 2015. URL: https://github.com/mesosphere/mesos-dns (visited on 01/15/2016).

[45] Mesosphere. *Mesosphere*. 2016. URL: https://mesosphere.com/ (visited on 01/23/2016).

[46] Open Containers Initiative. *runC*. 2015. URL: https://github.com/opencontainers/runc (visited on 01/21/2016).

[47] Prakhar Srivastav. *FoodTrucks: San Francisco's finger-licking street food now at your fingertips.* 2016. URL: https://github.com/prakhar1989/FoodTrucks (visited on 01/23/2016).

[48] Malte Schwarzkopf et al. "Omega: flexible, scalable schedulers for large compute clusters". In: *SIGOPS European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, 2013, pp. 351–364. URL: http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf.

[49] "Swarm v. Fleet v. Kubernetes v. Mesos". In: (2015). URL: http://radar.oreilly.com/2015/10/swarm-v-fleet-v-kubernetes-v-mesos.html.

[50] The Apache Software Foundation. *Apache Mesos*. 2015. URL: http://mesos.apache.org/ (visited on 01/15/2016).

[51] The Apache Software Foundation. *Apache Mesos - Docker Containerizer*. 2015. URL: http://mesos.apache.org/documentation/latest/docker-containerizer/ (visited on 01/15/2016).

[52] The Apache Software Foundation. *Apache Mesos - Frameworks*. 2015. URL: http://mesos.apache.org/documentation/latest/frameworks/ (visited on 01/15/2016).

[53] The Apache Software Foundation. *Apache ZooKeeper*. 2015. URL: https://cwiki.apache.org/confluence/display/ZOOKEEPER/ProjectDescription (visited on 01/21/2016).

[54] Abhishek Verma et al. "Large-scale cluster management at Google with Borg". In: *Proceedings of the Tenth European Conference on Computer Systems - EuroSys '15* (2015), p. 13. URL: http://dl.acm.org/citation.cfm?doid=2741948.2741964.

[55] Vinod Kumar Vavilapalli et al. "Apache Hadoop YARN : Yet Another Resource Negotiator". In: (2013), p. 16. URL: http://www.socc2013.org/home/program/a5-vavilapalli.pdf.