TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Department of Computer Science, Institute of Systems Architecture

Chair of Systems Engineering

Research Project Development
Analysis of a Research Project

Kompose evaluation

Thomas Hareau

22nd October 2017

Contents

1 Introduction

Containerized applications can be deployed in clusters using software like Docker Swarm or Kubernetes. Both software present pros and cons, and migrating from a solution to another is sometimes needed.

Thanks to Docker, it is possible to create a container that is independent from the operating system and with good performances. Thereafter, several containers can be aggregated into a new application. The tool *docker-compose* can manage all the containers at once. With this option,

the local code can be used as well as remote images, downloaded from a Docker registry. It also manages the system requirements, and connects the components together. To do so, a *docker-compose file* has to be defined. *docker-compose* aims at providing a facility to test a solution on a local cluster, before deploying it to a production cluster.

The use of a cluster allows improvement in availability, performances, scalability or resource management. On the one hand, *Docker Swarm* is the logical next step of docker-compose, since it also accepts a docker-compose file. On the other hand, *Kubernetes* may be preferred for some reason (finer adjustment, more powerful API, ...). However, this tool does not recognize the compose file. Instead, a Kubernetes manifest must be used.

Moving from one cluster to another may be difficult, especially for complex projects. *Kompose* was created to facilitate the migration from one solution to another. It can be used in two ways. Firstly, it can directly deploy the compose file to a cluster. In this case, the command line instructions are very similar to those offered by docker-compose. Secondly, it can convert the compose file into a Kubernetes manifest, in order to have a finer node management, or simply to migrate from one solution to the other.

It can be very interesting to manage a Kubernetes or a Swarm cluster using the same configuration. The Kompose project may perform this role. The goal of this report it to determine whether this project may be used in this manner. The quality of the generated component will also be evaluated.

2 Background and definition

Before moving to the experiments, some definitions must be given. Therefore this part may be skipped for an experimented user of Kubernetes, docker-compose or Docker Swarm.

2.1 Docker and the container management

Docker is a central technology of the cluster management. It can package applications and software into a container, which can thereafter be reused in any type of OS.

To do so, an image is produced from the sources of the application, which can be uploaded to a *repository*. Other users will then be able to download from the repository and reuse the image on any OS which supports docker.

2.2 Architecture of a cluster

The cluster management can be done in various manners. Since this project is about Docker Swarm and Kubernetes, only these types of architectures will be taken into account, but the global concept stays true for most of the cluster.

The goal of a cluster is to execute a program on several nodes, which can be a computer or a virtual machine. For testing purposes, a single node cluster can be used (as docker-compose or minikube).

The *manager*, which can also be called *master*, is the main node of the cluster. It is in charge of scheduling the tasks to other nodes while taking into account their states (CPU usage, memory usage, ...). It also manages the resources of the nodes. It is often the first created node. Other nodes which join the cluster can be called *minion*, or *worker*. It is possible to have several masters. In this case, the scheduling is separated between all the masters.

In each node, *containers* can be launched. For this, Docker images can be used, even if it is also possible to use other technologies in the case of Kubernetes. Master nodes can also welcome containers, depending on the configuration (default behavior for Docker Swarm; configuration can be set up for Kubernetes). It is also possible to run several components on a node.

Several types of containers can be created. On the highest level, a deployment[1] can be created. It defines a set of other containers which constitute an application.

The different containers communicate together using *network*. For this, different drivers can be used, with their own efficiency. Several networks can be created inside a cluster to create a better communication separation.

The data management is done using the concept of *volume*. The container can access a file or a directory of the host node using this, but it can also be used to ensure data is accessible from different nodes.

2.3 The different types of cluster

Depending on the usage, different types of clusters may be used.

Docker-Compose can be used to connect several containers together. For instance, a web server may communicate with a Postgres database. To do so, a *docker-compose file* can be defined. The file defines the communication between the different nodes (network definition, IP definition, port exposure, how to find other services), and also which resources to allocate to which component (CPU, RAM, volumes, ...). Then, using the command `docker-compose up`, all the component will be created on the local machine. The result is assimilable to a local node cluster. It is mainly used for testing purposes. To migrate to a production environment, Docker Swarm can be used.

Docker Swarm is the cluster version of Docker-Compose. It can have several nodes and several master nodes. A *docker-compose file* can be deployed using `docker-stack`, after tiny modifications (no build management, the local volume must be locally synchronized).

The cluster creation is straightforward. The master should firstly initialize the swarm using the command `docker swarm init`. This will set up the environment and generate a token. This key is necessary for other nodes to join the cluster, using the command `docker swarm join --token <token> <master-ip>`. Nothing else is necessary.

Kubernetes is a competing cluster solution. It can also use Docker to back the components. As for Docker-Compose, it is possible to create single node clusters (using minikube for instance). This can be used for testing purposes. It is also possible to manage several nodes, as for Docker Swarm, with tools like *kubeadm*.

Creating a cluster using kubeadm is mainly like using Docker Swarm. There is only one more step: installing the networking solution. For this project, *kube-router* was used.

2.4 The volume management

As said before, a volume can be used to manage data. In containers, it is a folder or a file. It can be bound to an existing directory on the hosting node, but can also be used to ensure the data is safe against container restarts.

The volume management can be very complex, especially for Kubernetes. This part will briefly explain the main concepts.

With Docker-Compose and Docker Swarm

Only one type of volume can be defined in a docker-compose file. It is simply named a *volume*. In this, the mounting path, the host path or the driver type can be defined. It is really straightforward but is quite rigid.

[1] The term *deployment* is specific to Kubernetes. Its equivalent on Docker Compose is *service*. Since this term defines another concept in Kubernetes, deployment will be used instead in this report to avoid confusion.

With Kubernetes

Kubernetes defines more than 20 types of volumes. This allows a close resource management. Some volumes are local to a cluster, others are external. With this system, it is, for instance, possible to easily access a Google Persistent Disk.

The volume definition is divided into two locations. First, inside the component definition, what is local to the pod must be written. It can be the mount path, for instance. A name will also be given, which will be used to associate it with the desired volume. The second part is the volume definition itself: which type of volume must be used, which storage space is needed, ... And in this part, there are a lot of different choices.

emptyDir represents an empty folder, which will not be accessible from the host, and deleted after the teardown of the hosting component. Using this, it would not be possible to inject data in this volume before the creation. Nor would it be possible to access the data from the node. However, it is persistent across failures, meaning that if the hosting component fails and restarts, it would still be able to retrieve its data.

hostPath bind the component mounting directory and its node file system. Therefore, the data in the node repository will be accessible when the component starts and what was computed can be retrieved after its shutdown. However, there is a lot of complications if this volume type is used as is (root rights needed from the component, for example). Typically, it can be used to synchronize some administration files.

The couple PV-PVC: this is the most complex but the most flexible solution. Two elements have to be defined: a Persistent Volume (PV) and a Persistent Volume Claim (PVC).

The **PVC** is the object bound to the component. It can be seen as a pod whose only goal is to furnish a link to the volume. It will define how much storage it needs, and other component oriented information. However, it is not in charge of storing the data. This is the task of the **PV**. Here every storage oriented information must be defined: how much space it offers, how to behave when shutting down, etc. The storage itself is done by using any other volume type, typically a hostPath (since the PV manages the volume, the root right sketched before is overridden and any user on the component can use it).

Since the two components are defined separately, the scheduler is in charge of binding a PV to a PVC. To do so, it will associate them using the provided requirement. It is also possible to define a *StorageClass* in both definitions. In this case, the PVC would only be linkable to a PV having the same StorageClass. This is especially useful if the PVC must use a specific node directory: it is not possible to define it on this side, so the StorageClass must be used. If two PV have the same specifications, it is not possible to predict which one will be bound to a PVC.

3 Use case and Requirements

The initial motivation for this project is to be able to manage different types of clusters using the same configuration. *Sentry*, a logging software, can be deployed to a Docker Swarm cluster. It could be interesting to use the same configuration and deploy it to Kubernetes.

The project *sentry* provides a centralized logging solution: several projects can be plugged to sentry, and send their logs to the service thereafter. It is especially useful in the case of cluster applications: independent code will be launched on different nodes. Writing to a common file system is difficult, because this file would have to be synchronized between the nodes. Also, since the clock of each node may not be synchronized, local log files cannot be completely trusted. Indeed, it is not possible to know which event arrived first.

Sentry can be used for this purpose. Rather than sending the logs to a file, they are sent to sentry, which can reliably proceed them. The application itself consists of several different services (a frontend, databases, workers, ...), which can be deployed to a cluster. These components are defined through a compose file.

This solution is easily deployable to a swarm cluster but is not compatible with Kubernetes. Creating a new configuration is complicated, and moreover, any changes in one of the two configurations should be duplicated to the second. The result would be error-prone and not reliable.

In this case, generating the configuration on the fly using Kompose may be a solution. The developers would only have to maintain the compose file, and then to choose during the installation between a Kubernetes or a Docker Swarm deployment.

From this example, two requirements stand out:

- The solution deployed by Kompose is directly usable, or some tiny modification has to be done. In the last case, the modifications have to be scriptable.
- The solution deployed using Kompose is sensibly as efficient as the original solution.

To know if this requirement can be checked out, some experiments were launched on three different projects.

4 Migrating from Docker to Kubernetes

To evaluate the solution generated by Kompose, the tool was used on three different projects: sock shop, example-voting-app and scone-cli (sentry).

Sock Shop (microservices demonstration) is a simulation of a commercial application. It aims to provide a demonstration and a testing utility of the microservices and cloud technologies (according to the official documentation[2]). Therefore, it is available in several versions, as Kubernetes, docker-compose and Docker Swarm.

Among all the goals of the project, *Demonstrate microservice best practices* make this application very interesting to test Kompose. A very complex architecture was voluntary created, splitting the concept in as much microservices as possible (see figure 1).

Every component of the architecture is provided by an independent docker image. Most of them are written in different languages to show another advantage of the microservices architecture. They communicate together using REST over HTTP, and discover each other using the DNS.

example-voting-app is an official example from the docker-compose documentation. It simulates a voting application, where the user selects one of two choices. Here five component are defined (see figure 2): two frontends (voting, where the user can vote, and result, where the result can be retrieved), two databases (redis and db) and a worker, which computes the received votes from redis, and stores them in db.

It can be natively deployed to docker-compose and Docker Swarm, the first version representing a development environment, whereas the second can be used in production. Both versions are interesting, the first to test the local code and image management, the second to evaluate the performances.

scone-cli (sentry) is a tool which, among other utilities, can be used to deploy sentry on a swarm cluster. It manages volume sharing, installs the different tools and finally deploys sentry to the swarm node. Being able to launch it with Kubernetes is the main goal of this project.

[2] see microservices-demo architecture

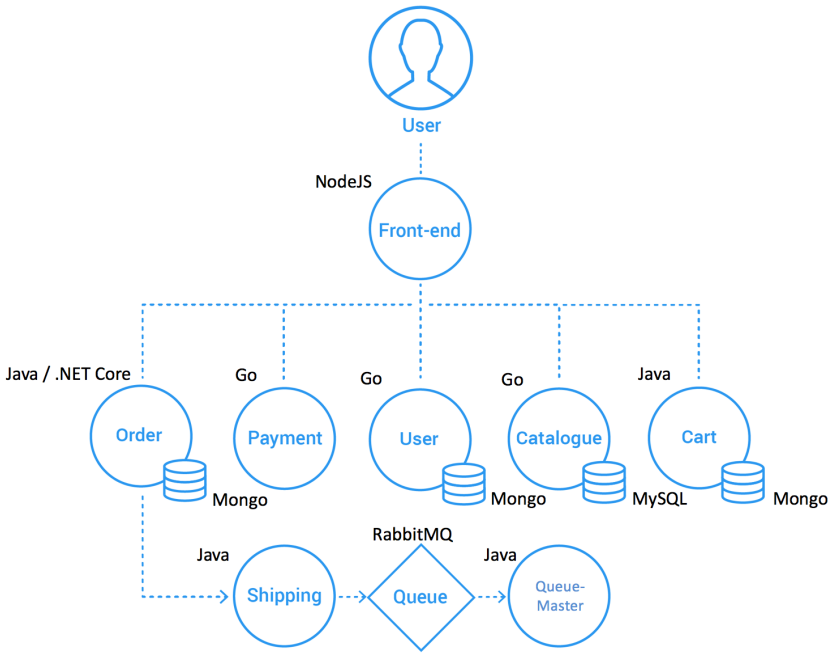

Figure 1: Sock Shop architecture

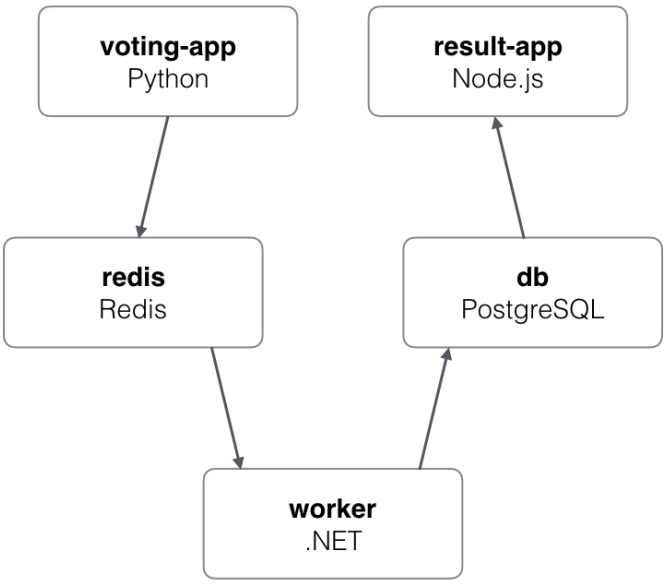

Figure 2: Sock Shop architecture

4.1 The volume management

The implementation of the Kubernetes and docker-compose volumes is too different and renders it impossible to automate the translation. Indeed, docker-compose only has a single type of volume, whereas Kubernetes provides several volume types, permitting a more precise management. Since choosing the volume type depends on the usage, Kompose turns all the volumes into PVCs or into empty directories, depending on a flag, and lets the user create the PV if needed. The main issue is when a node path is provided: it is totally ignored. Depending on the usage, there are three different options:

- translating all the volumes into empty directories. As a result, the data in the volumes will be empty when starting the deployment and will be deleted after the component tears down. It will also not be possible to access the data during the runtime. However, the data will stay consistent across failures. In this case, the kompose command *up* can be used directly, with the flag `--volumes` set to `emptyDir`;
- using the generated PVCs. Since they do not have a StorageClass, it is not possible to assign a specific PVC to a specific PV. It will not be possible, for instance, to mount the PV at a specific path on the node. Also, after several restarts, a PVC may not be bound to the same PV as before. A situation where this behavior is not an issue is rare, but allows the use of the command `kompose up`. Also, the different PVs must be created, unless dynamic storage is available;
- modifying the generated files to add the StorageClass, to ensure the PV is correctly bound to the relevant PVC. Another possibility is to replace the PVC with a `hostPath`, but this is less scalable.

In all three projects, the third option was used. Indeed, a volume is frequently used to inject some data or to share the data between the nodes. In these cases, accessing the data from the host machine is a need. However, these modifications were always the same. It could be imagined to create a new script to be launched after `kompose convert`, and which will modify the manifest.

4.2 The tmpfs

The `tmpfs` is a specific property of docker compose. It provides a temporary volume, stored in the host RAM, and removed after the deletion of the container. In the definition, it is also possible to add the mode, by adding it after the volume path.

This feature is not supported in Docker Swarm nor in Kubernetes. Therefore, in both cases, it has to be adapted. When deploying a docker-compose file into a Docker Swarm cluster, the tool simply ignores the temporary file system.

In Kubernetes, the closest volume is `emptyDir`. The volume is not necessarily created on the RAM, but on any type of medium (even if it can be specified to use the RAM). Also, it is not possible to specify the mode in this case.

Thus, Kompose translates the `tmpfs` into a `emptyDir`. However, the mode was kept in the translated manifest. It is a bug[3], which will be solved in one of the future releases.

For now, if a docker-compose file contains a `tmpfs` with a mode, it would not be possible to use `kompose up`. Instead, `kompose convert` has to be used, and the generated files must be edited to remove the mode after the path.

4.3 The build management

Docker-Compose can be used in a development environment. In this case, it is possible to replace the keyword `image` with `build`. Instead of an image name, a path to a directory containing a

[3] https://github.com/kubernetes/kompose/issues/807

Docker file has to be provided. When deploying the application, docker-compose will build the image and use it to create the component.

The keyword `build` is not supported in Docker Swarm nor in Kubernetes. In both cases, an image must be built, pushed to a repository, and finally used instead of the initial `build` value.

If wanting to use the `build` keyword on a regular Kubernetes cluster is unlikely, it is more likely to do so with a single node cluster as minikube. In this case, it is quite simple to achieve the equivalent, by setting the cluster to use the local docker registry. However, the user would still have to push the image to the local registry manually.

On a multiple-node cluster, it has to be configured to connect to a local registry. Two possibilities exist, either installing a registry, which can be impossible on certain configuration types, or by using the addon registry. Finally, the generated manifests have to be edited in order to add the registry URL before the name of the image.

The project Kompose plans to support this feature better. Indeed, the issue 636 plans to support it in the version 1.3 (even if it was already postponed twice). But for now, there is no real solution for this issue, excepting not using it for a multiple-node cluster (which makes a lot of sense, this is definitively a development purpose feature).

4.4 Miscellaneous

Some other tiny issues exist. They depend either on the cluster configuration or on the usage made of Kubernetes.

- The restart policy in docker-compose can be `on-failure`. In this case, no service will be generated, only a pod. It may create complications if the user wants to use a service simultaneously. The restart section of the user guide can bring more information.
- To expose a port in Kubernetes, specifying the port may not be enough, it may also be necessary to define the service type to create. Since this is not available in docker-compose, it is possible to add a label in the docker-compose file. It would be ignored with docker-compose, but used by Kompose to generate the manifest. More information can be found in the user guide.
- The port exposure in the translated file may not be the same. In every case, a port will be exposed, but maybe not the expected one, depending on the cluster configuration. With minikube for instance, the real URL can be accessed by using the command: `minikube service yourservice`.
- If a script is used to install some elements on the node, not only will the compose file have to be adapted but also will the cluster commands. For example, in *scone-cli*, the nodes could be retrieved using the command `docker node ls`. Of course, this will not work on Kubernetes, and this command has to be adapted (`kubectl get nodes`).

As a conclusion, some features are not supported by Kompose. Most of them are not very important, and some of them will eventually be corrected. However, regarding the volume management, it will most likely stay as it. One solution for this is to create a script to run after Kompose, and which would update the files.

5 Evaluation

After these experiences, it is possible to perform a performance analysis. The goal is to determine if the conversion alters the quality of the component.

The testing architecture is the following: the cluster has 5 virtual-machines (VM): one master and 4 workers, with each 3 Go RAM and 2 CPU. The Docker Swarm cluster was created using the

integrated command `docker swarm init`. The Kubernetes cluster was created using the `kubeadm` utility (the master was not detached, therefore it did not welcome any containers), with kube-router as a pod network[4]. The tests were launched from another VM, not linked to the cluster (no VM of the cluster was on the same physical machine as the benchmarking VM).

For the tests, the tool wrk2 was used. It is an HTTP based benchmarking tool, which ensures to produce a constant throughput load. It sends a known amount of requests per second, during a certain duration, and computes the mean latency.

Tests were launched on a set of different throughputs, going from 10 to 10^6. Two projects were used: the swarm version of example-voting-app and microservices-demo.

Since example-voting-app is entirely asynchronous, sending POST requests would not represent real computation from the cluster. Consequently, a simple GET request was sent to the *vote* frontend.

microservices-demo is mainly asynchronous but also has some parts which require computation, like ordering what is in the cart: in this case, some information about the order. To do so, socks were put on the cart, and a POST request was sent to the route `/order`. Since this action creates an order without clearing the cart,the same cart can be used many times. The application was entirely reset between each throughput test, to be sure that the state in the beginning is always consistent.

Results

The following graphs represent the evolution of the latency in relation with the throughput. On each graph, the two curves represent the performances of the original docker-compose solutions on the Swarm cluster, and of the Kompose generation result on the Kubernetes cluster.

example-voting-app Figure 3 shows the evolution for example-voting-app. Both curves for Kubernetes and Docker Swarm start by decreasing slowly, with a low standard deviation, until being close to zero. Then the curves stay stable until a dramatic raise of the latency and of its associated standard deviation. After reaching a high latency level, the curves stay there for any higher throughput. So, there are four stages: the decreasing stage, the low level stage, the increasing stage and finally the high level stage. The decreasing stage shows a lighter but better behavior for Docker Swarm since it is lower; the low level stage shows comparable results; the increasing stage happens sooner for Docker Swarm; the final stage is once again comparable.

The main difference in both curves is the third stage. It highlights the throughput each solution can reach until they begin to saturate, which is also indicated by the high values of the standard deviation. The cruves reveal better performances for the Kubernetes solution, doubtlessly due to the networking efficiency, since no computation is required. Also, the increasing stage is smoother for Kubernetes. It could have been shown that the Docker Swarm networking solution is more dependent on the external bandwidth usage than Kubernetes, maybe explaining the difference.

On the other side, the low level stage, which represents the optimum throughput condition, is the same for both the curves. This shows that in optimal condition, the performances are alike.

microservices-demo Figure 4 presents the evolution for microservices-demo. The three last steps of the benchmark for example-voting-app are also present: a low level stage, an increasing step (this time slower than before) with an increasing standard deviation, and finally a high level stage, with a high standard deviation.

As before, the saturation is reached sooner for the Docker Swarm version, due to the networking efficiency. The smallest slope shows also that the efficiency of the container is also involved. However, in either case, the slope is alike, not permitting to determine if the translated solution is more or less efficient than the original.

[4] kube-router documentation

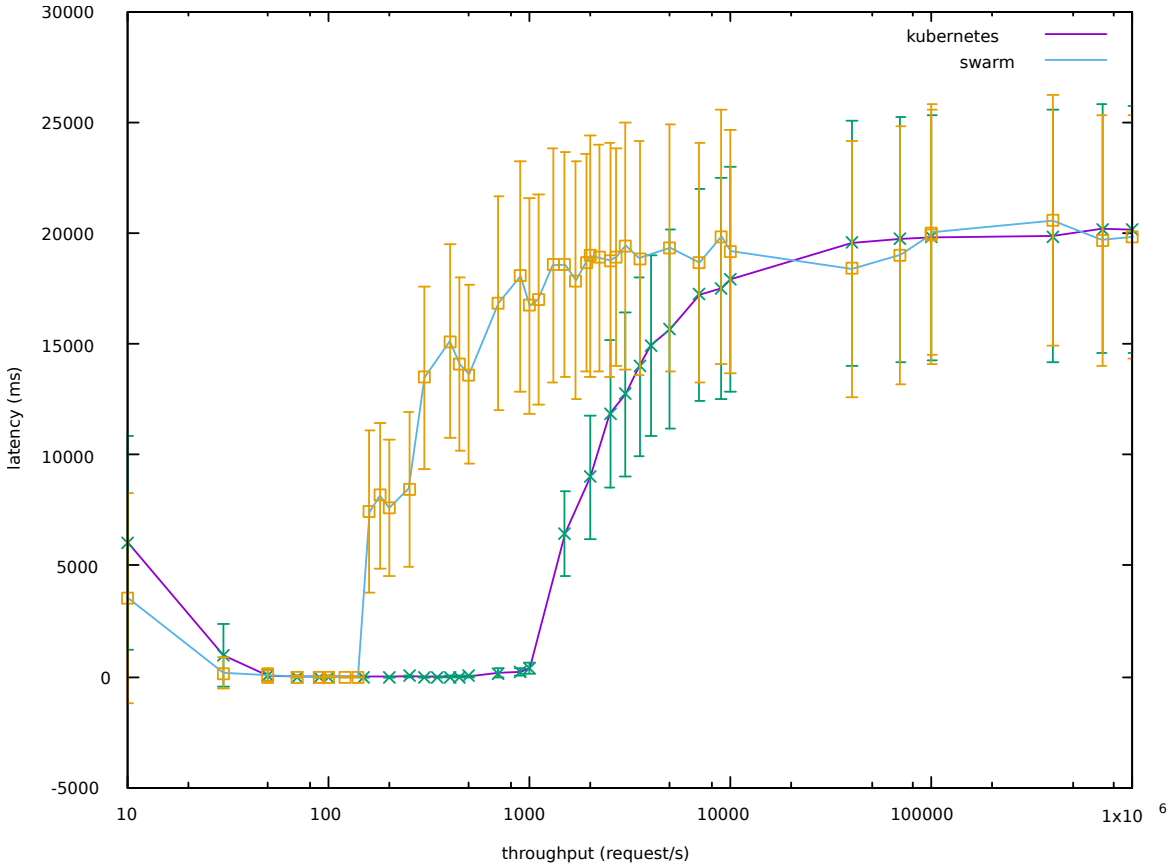

Figure 3: Evolution of the latency against the throughput for example-voting-app

The low level stage is very short for the Swarm cluster, it became therefore difficult to exploit it. However, the value is close, suggesting alike performances.

Conclusion

The comparison of the throughput for an original swarm application and it's converted version does not show a clear result. The networking efficiency is not the same in the different clusters, making it difficult to decide if the converted version suffers from a loss of efficiency. However, the different curves do not highlight real performance issues: in optimal throughput conditions, the efficiency is alike.

6 Conclusion

In this report, the Kompose project was investigated in terms of functionality and efficiency. The main weakness of the project is the volume management. If involved, the translated manifest will need to be highly modified. On the other side, the translated application does not show a loss of efficiency, suggesting that Kompose can safely be used.

A solution to manage an application using either Docker Swarm or Kubernetes is to create another tool, which would manage the volume management. Indeed, despite being complicated, the possible modifications are always the same: using a PVC linked to a PV and bound to a hostPath on the local node, for instance. This action could be totally automatized, solving the main issue of Kompose.

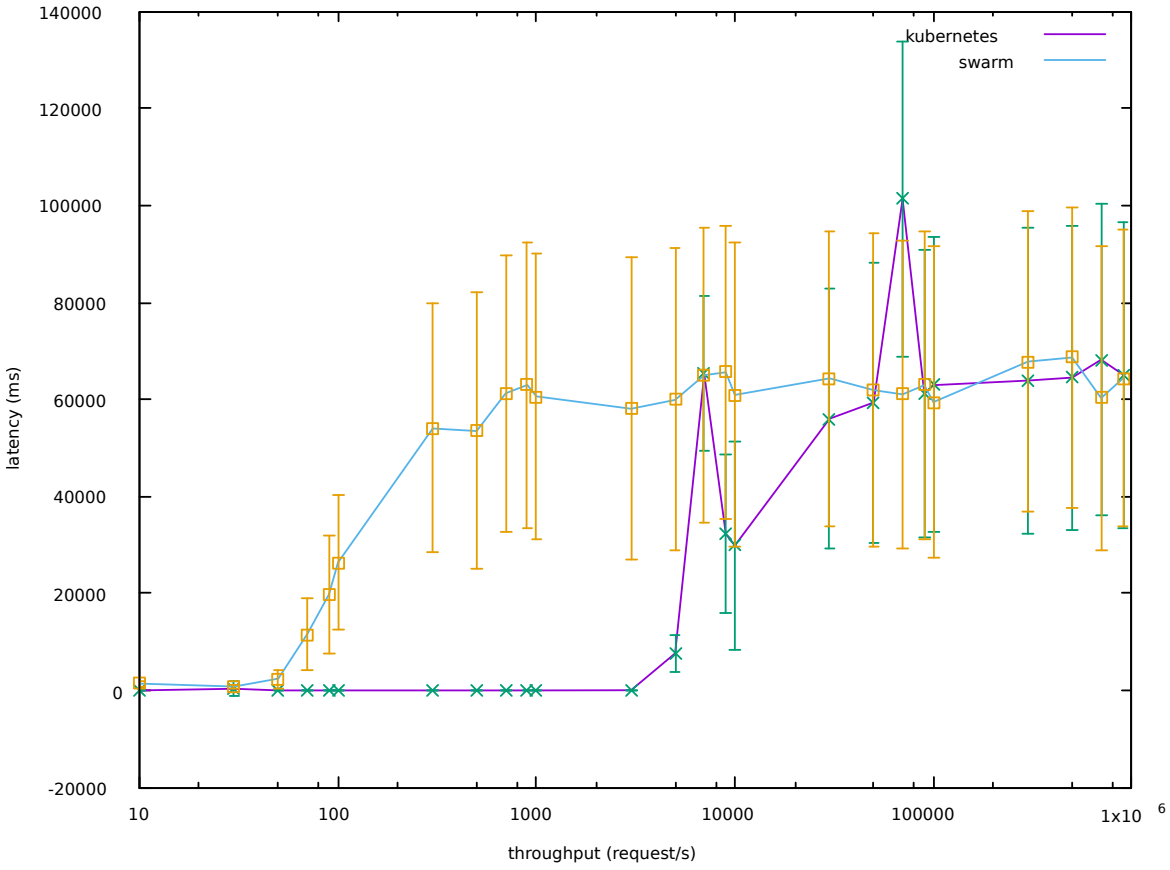

Figure 4: Evolution of the latency against the throughput for microservices-demo