Containers

107021129 黃明瀧

Cloud Computing Homework 2

A. Screenshots for Task 1

Figure 1: List containing the container for task 1

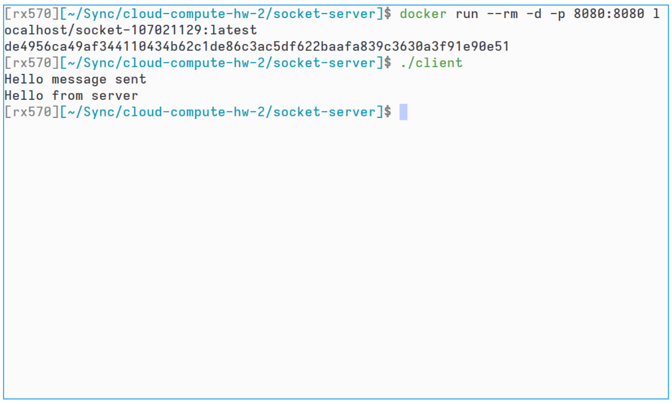


Figure 2: Running the container and the client

B. Screenshots for Task 3

The client was modified to read destination IP and port from stdin.

```
s107021129@cc-jump:~/task1$ bat socket-deployment.yml

File: socket-deployment.yml
Size: 446 B

apiVersion: apps/v1
kind: Deployment
metadata:
  name: s107021129-deployment
spec:
  selector:
    matchLabels:
      app: s107021129-socket
  replicas: 5
  template:
    metadata:
      labels:
        app: s107021129-socket
        tier: backend
        track: stable
    spec:
      containers:
      - name: s107021129-socket
        image: socket-107021129:latest
        imagePullPolicy: Never
        ports:
        - containerPort: 8080

s107021129@cc-jump:~/task1$
```

Figure 3: Deployment configuration

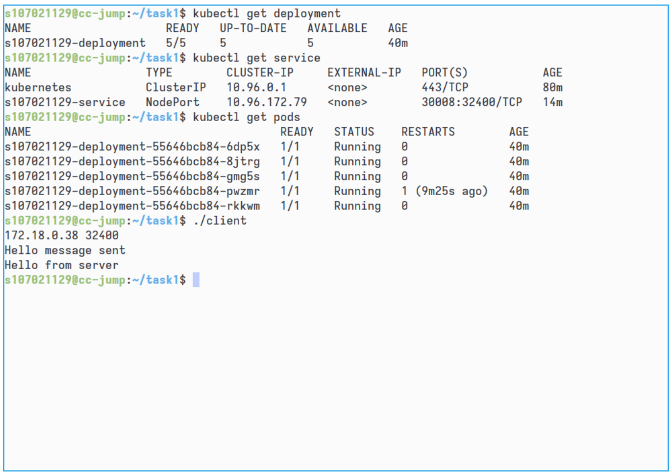
```
s107021129@cc-jump:~/task1$ bat socket-service.yml

File: socket-service.yml
Size: 192 B

apiVersion: v1
kind: Service
metadata:
  name: s107021129-service
spec:
  type: NodePort
  selector:
    app: s107021129-socket
  ports:
  - protocol: TCP
    targetPort: 8080
    port: 30008

s107021129@cc-jump:~/task1$
```

Figure 4: Service configuration



```
s107021129@cc-jump:~/task1$ kubectl get deployment
NAME                   READY   UP-TO-DATE   AVAILABLE   AGE
s107021129-deployment  5/5     5            5           40m
s107021129@cc-jump:~/task1$ kubectl get service
NAME                 TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)           AGE
kubernetes           ClusterIP   10.96.0.1      <none>        443/TCP           80m
s107021129-service   NodePort    10.96.172.79   <none>        30008:32400/TCP   14m
s107021129@cc-jump:~/task1$ kubectl get pods
NAME                                     READY   STATUS    RESTARTS        AGE
s107021129-deployment-55646bcb84-6dp5x   1/1     Running   0               40m
s107021129-deployment-55646bcb84-8jtrg   1/1     Running   0               40m
s107021129-deployment-55646bcb84-gmg5s   1/1     Running   0               40m
s107021129-deployment-55646bcb84-pwzmr   1/1     Running   1 (9m25s ago)   40m
s107021129-deployment-55646bcb84-rkkwm   1/1     Running   0               40m
s107021129@cc-jump:~/task1$ ./client
172.18.0.38 32400
Hello message sent
Hello from server
s107021129@cc-jump:~/task1$
```

Figure 5: `kubectl` and client outputs

C. Performance of Container vs VM (Task 2)

All the experiments are ran for at least 3 times, and the average values is presented in the following table.

Item	VM	Container
Sysbench CPU (Events/s)	18486	18630
Sysbench Memory (Events/s)	61141029	65703919
Sysbench File IO (Events/s)	17737	1066502
iperf (Gbps)	1.19	9.68

In terms of CPU and memory performance, there is no dramatic difference between the VM and the container. With the help of KVM, the VM inside QEMU run pretty much as well as the container, with very minor performance hit.

File IO speed in the VM is severely impacted by the fact that the file operations goes through the qcow2 disk image format, which, as described in the documentation of QEMU, "has the largest overhead compared to raw images when it needs to grow the image". Since the experiments were done in a fresh VM, the images had not allocated the space required for the files written by sysbench, and did the allocation on-the-fly while sysbench runs, causing file IO to be very slow.

`iperf` shows that the container has a huge advantage (with more than 8 times better throughput) in communicating with the host machine via network, since the network traffic is essentially travelling between processes on the host itself.

D. Difference between Docker Container and VM

As opposed to virtual machines, which are themselves fully virtualized systems, Docker containers are essentially a bunch of carefully managed and restricted processes running on the same host. This enables containers to be much more lightweight and enables better sharing of the computational resources on the host (for example, memory allocation and overcommitting is much more easier), at the cost of having less isolation and security.

Docker containers typically have different use cases than virtual machines. Docker containers are nowadays mainly used for deploying stable development and production environments, which is not commonly done using virtual machines due to the bloat of virtual machine images.

E. Explain “Deployment”, “Service”and “Pod”

Deployment

Deployment in Kubernetes is a declarative way to specify how to create and manage the *pods* that constitutes a container app. Typically things like the container images to be used and the number of replicas are specified in a deployment.

Service

A *service* in Kubernetes groups one or more pods in a cluster as a logical application, presenting them to the outside world as a network service. In this assignment, a `NodePort` service is used to expose the socket service on all nodes.

Pod

A *pod* in Kubernetes is an instance (or several instances) of running Docker containers, and is the smallest unit of execution. When there are multiple containers inside a single pod, the containers share network and storage resources.

F. Kubernetes

Kubernetes is a system for managing containerized applications and services. It provides us with many features such as *service discovery* (exposing container with a DNS name), *rollouts* and *rollbacks* (gradually reaping old instances and creating new ones), *fault tolerance* (restarts unheathy instances), *load-balancing*, and more.

The need of Kubernetes stems from the fact that managing containers *across multiple servers* at scale is hard. Kubernetes presents a tightly integrated interface and declarative configurations, which allows the operators to do so in a maintainable way.

G. Container Technology

Container technology is a kind of *OS-level* virtualization that runs multiple userspaces, with restrictions on the resources they can access, while the userspaces are referred to as *containers*. Container technology is widely used in today's cloud computing environments for several reasons:

- Container is lightweight.
- Container bundles all the dependencies together, making shipping easy.
- Deployment with containers is fast.
- Deployment is more scalable than VM's.
- Container can run everywhere, reducing lock-in's.

H. Container Data

By default, the storage presented to the container instances are *empheral* local storage, which is implemented by using union filesystem on top. With help of the storage driver behind the scenes, the data written to the root of different container instances are actually written to different (temporary) actual locations, and the lifetime is tied to the instance, making them deleted upon container exit.

To actually persist the files, Docker provides *volumes* and *bind mounts* for the purpose. *Volumes* exposes storages managed by Docker to the containers, while *bind mounts* exposes a directory on the host directly into the containers. These can be done, respectively, by

```
docker volume create my-volume
docker run --mount source=my-volume,target=/app ...
```

and

```
docker run -v /path/to/host/app:/app ...
```

.

I. Why Pods?

Kubernetes uses Pod as a thin wrapper around a container (or several containers). Within a pod, the containers are able to *share the local network* and other resources, which must all *be deployed together* onto a specific executing node. Without the notion of pods, one cannot express this idea in the configurations when using Kubernetes, which is why pods exists.