Project 1 Report

Team “RemCode”:
Matvey Plevako, Mohamad Ziad AlKabakibi

Introduction

Topics, covered in the project:

- Understanding Docker-machine and Docker Swarm deployment
- Container Docker Cluster farm deployment
- Docker Container migration - Application Distribution
- Memory management
- Image size compression
- Building distributed Interactive web application

Application description:
RemCode is a website for remote code execution. Users can input code text, stdin to pass to the program and select language from the list. Then, they can view the results of their program.

Requirements for application:
- Web applications shouldn’t execute any user code.
- Web applications should submit tasks to the task queue.
- Workers should connect to the task queue and pick tasks as they appear.
- Workers should execute the code with stdin passed with source code.
- After code execution, both successful and failed, workers should save results to the database.
- Users can view results of their program in a new window.
- Both Web application and Worker should be **scalable.**

Project

Understanding Docker-machine and Docker Swarm deployment

1. What is Docker-machine and what is it used for?

Docker-machine is a tool that is used for installing, setting and managing remote Docker hosts easily, moreover these hosts can be placed on different physical machines.
We can use docker-machine when we want to create a deployment environment for our application and manage all the micro-services running on it. With different drivers, docker-machine can be executed on different platforms for remote or local hosting.

2. What is Docker Swarm, what is it used for and why is it important in Containers Orchestration?
Docker Swarm is a system that is used for merging a set of Docker containers into a single coherent cluster and managing it. One of the key benefits associated with the operation of a docker swarm is the high level of availability offered for applications. In a docker swarm, there are typically several worker nodes and at least one manager node that is responsible for handling the worker nodes' resources efficiently and ensuring that the cluster operates efficiently. Moreover, it provides a convenient interface for managing containers on these hosts, services and distributing them among connected worker and manager nodes.

3. Install Docker-machine based on your virtualization platform (VirtualBox, Hyper-V, VMware), create a Machine (named Master), and collect some relevant information for you.

a. docker-machine driver is virtual box

b. Creating a new docker-machine host. we select virtualbox driver and name it “Master”

```
demo@myvm:~$ docker-machine create --driver virtualbox Master
Running pre-create checks...
Creating machine...
(Master) Copying /home/demo/.docker/machine/cache/boot2docker.iso to /home/demo/.docker/machine/machines/Master/boot2docker.iso...
(Master) Creating VirtualBox VM...
(Master) Creating SSH key...
(Master) Starting the VM...
(Master) Check network to re-create if needed...
(Master) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this virtual machine, run: docker-machine env Master
```

c. Running docker-machine ls after creating several docker-machines

```
demo@myvm:~$ docker-machine ls
NAME      ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER     ERRORS
Master    -        virtualbox   Running   tcp://192.168.99.100:2376           v19.03.12
Worker1   -        virtualbox   Running   tcp://192.168.99.101:2376           v19.03.12
Worker2   -        virtualbox   Running   tcp://192.168.99.102:2376           v19.03.12
```

Stopping and start docker-machine

```
demo@myvm:~$ docker-machine stop Worker2
Stopping "Worker2"...
Machine "Worker2" was stopped.
demo@myvm:~$ docker-machine ls
NAME      ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER     ERRORS
Master    -        virtualbox   Running   tcp://192.168.99.100:2376           v19.03.12
Worker1   -        virtualbox   Running   tcp://192.168.99.101:2376           v19.03.12
Worker2   -        virtualbox   Stopped                                       Unknown
demo@myvm:~$ docker-machine start Worker2
Starting "Worker2"...
(Worker2) Check network to re-create if needed...
(Worker2) Waiting for an IP...
Machine "Worker2" was started.
Waiting for SSH to be available...
Detecting the provisioner...
Started machines may have new IP addresses. You may need to re-run the `docker-machine env` command.
```

docker-machine rm

```
demo@myvm:~$ docker-machine rm Worker1
About to remove Worker1
WARNING: This action will delete both local reference and remote instance.
Are you sure? (y/n): y
Successfully removed Worker1
```

4. Create two Workers as well. Later we will connect them into one swarm. Make a screenshot for docker-machine ls command. You should have 3 running machines.

```
demo@myvm:~$ docker-machine ls
NAME      ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER     ERRORS
Master    -        virtualbox   Running   tcp://192.168.99.100:2376           v19.03.12
Worker1   -        virtualbox   Running   tcp://192.168.99.101:2376           v19.03.12
Worker2   -        virtualbox   Running   tcp://192.168.99.102:2376           v19.03.12
```

Container Docker Cluster farm deployment

5. Now that Docker Swarm is enabled, deploy a true container cluster farm across many Dockerized virtual machines. (One master and two workers). Verify the Docker Swarm status, identify the Master node(s), and how many workers active exist. Take as many screenshots as you need to explain the process.

Firstly, we need to initiate a Docker swarm with address and port to join.

```
docker@Master:~$ docker swarm init --advertise-addr 192.168.99.100:2377
Swarm initialized: current node (n6nqjh7tao9gxqo891gsti8s9) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-1dy8cu0t4lqgms592rnvocwvsbxqm7yrtbl62h416io5j6j41v-8g23nmzxtm268u3pnvfj003m0 192.168.99.100:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

Then, we can check info about our swarm cluster

```
docker@Master:~$ docker info
Client:
 Debug Mode: false

Server:
 Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
 Images: 0
 Server Version: 19.03.12
 Storage Driver: overlay2
  Backing Filesystem: extfs
  Supports d_type: true
  Native Overlay Diff: true
 Logging Driver: json-file
 Cgroup Driver: cgroupfs
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog
 Swarm: active
  NodeID: thj3ytc2o73pljwgio18adb4a
  Is Manager: true
  ClusterID: ac2mttba01k7kfkcqeldi7c57
  Managers: 1
  Nodes: 1
  Default Address Pool: 10.0.0.0/8
  SubnetSize: 24
  Data Path Port: 4789
  Orchestration:
```

Join worker nodes to swarm (commands for joining are too long, so they are compressed)

```
<dy8cu0t4lqgms592rnvocwvsbxqm7yrtbl62h416io5j6j41v-8g23nmzxtm268u3pnvfj003m0 192.168.99.100:2377
This node joined a swarm as a worker.
docker@Worker1:~$ exit
logout
demo@myvm:~$ docker-machine ssh Worker2
   ( '>')
  /) TC (\   Core is distributed with ABSOLUTELY NO WARRANTY.
 (/-_--_-\)           www.tinycorelinux.net

<dy8cu0t4lqgms592rnvocwvsbxqm7yrtbl62h416io5j6j41v-8g23nmzxtm268u3pnvfj003m0 192.168.99.100:2377
This node joined a swarm as a worker.
```

list all active nodes in swarm

```
docker@Master:~$ docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS      ENGINE VERSION
n6nqjh7tao9gxqo891gsti8s9 *   Master              Ready               Active              Leader              19.03.12
oek8fn09t7eayw037tqoeyz2z     Worker1             Ready               Active                                  19.03.12
y0mt9uulugck7s4cfu5nxhgdx     Worker2             Ready               Active                                  19.03.12
```

6. How can a Worker be promoted to Master and vice versa? Please explain if special requirements are needed to perform this action? Perform the process and explain it.

We can promote a worker node to the manager role using manager node. This is useful when a manager node becomes unavailable or if we want to take a manager offline for maintenance. Similarly, we can demote a manager node to the worker role.
Special requirements are we must always maintain a quorum of manager nodes in the swarm, in case of master node failures.

1. To promote worker node to the manager node:

```
docker@Master:~$ docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS      ENGINE VERSION
n6nqjh7tao9gxqo891gsti8s9 *   Master              Ready               Active              Leader              19.03.12
oek8fn09t7eayw037tqoeyz2z     Worker1             Ready               Active                                  19.03.12
y0mt9uulugck7s4cfu5nxhgdx     Worker2             Ready               Active                                  19.03.12
docker@Master:~$ docker node promote Worker1
Node Worker1 promoted to a manager in the swarm.
docker@Master:~$ docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS      ENGINE VERSION
n6nqjh7tao9gxqo891gsti8s9 *   Master              Ready               Active              Leader              19.03.12
oek8fn09t7eayw037tqoeyz2z     Worker1             Ready               Active              Reachable           19.03.12
y0mt9uulugck7s4cfu5nxhgdx     Worker2             Ready               Active                                  19.03.12
```

   we promoted worker node and now we can perform cluster management from this node

2. To demote manager to worker:

```
docker@Master:~$ docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS      ENGINE VERSION
n6nqjh7tao9gxqo891gsti8s9 *   Master              Ready               Active              Leader              19.03.12
oek8fn09t7eayw037tqoeyz2z     Worker1             Ready               Active              Reachable           19.03.12
y0mt9uulugck7s4cfu5nxhgdx     Worker2             Ready               Active                                  19.03.12
docker@Master:~$ docker node demote Worker1
Manager Worker1 demoted in the swarm.
docker@Master:~$ docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS      ENGINE VERSION
n6nqjh7tao9gxqo891gsti8s9 *   Master              Ready               Active              Leader              19.03.12
oek8fn09t7eayw037tqoeyz2z     Worker1             Ready               Active                                  19.03.12
y0mt9uulugck7s4cfu5nxhgdx     Worker2             Ready               Active                                  19.03.12
```

7. Deploy a simple Web page, e.g Nginx, showing the hostname of the host node it is running upon, and validate that its instances are spreading across the servers previously deployed on your farm.

Solution: We created a simple html page and displayed $hostname variable that is passed from nginx and served it with nginx server.

Created Dockerfile:

```
FROM nginx:alpine
COPY ./nginx.conf /etc/nginx/nginx.conf
COPY ./index.html /www/media/index.html
```

Created HTML page:

```
<html>
    <body>

    <!--# echo var="hostname" default="unknown_host" -->:<!--# echo var="server_port" default="unknown_port" -->

    </body>
</html>
```

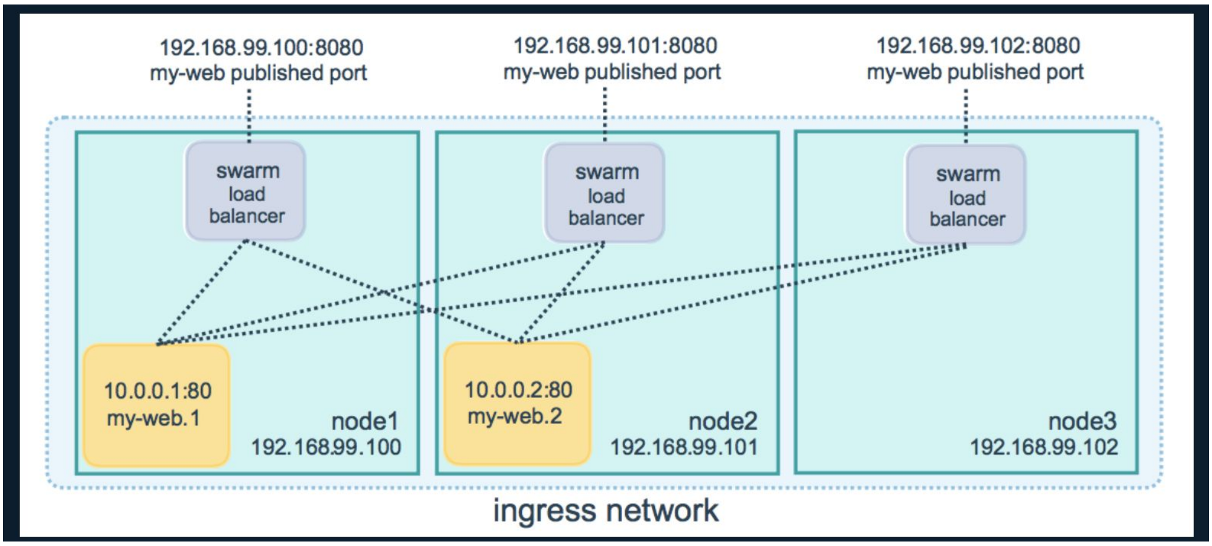
Created nginx.conf and enabled ssi to display $hostname variable

```
http {
    server {
        listen 80 default_server;
        listen [::]:80 default_server;

        location / {
            ssi on;
            root /www/media;
        }
    }
}
events { }
```

Spreading web-app for different hosts

```
docker@Master:~$ docker service create --name my-web --publish published=8080,target=80 --replicas 3 remcodeds/nginx-page:latest
qrbfzke51l3rcbeec80wxcwzc
overall progress: 3 out of 3 tasks
1/3: running   [==================================================>]
2/3: running   [==================================================>]
3/3: running   [==================================================>]
verify: Service converged
docker@Master:~$ docker service ps my-web
ID                  NAME                IMAGE                         NODE                DESIRED STATE       CURRENT STATE            ERROR
     PORTS
tnz67oxxl1zk        my-web.1            remcodeds/nginx-page:latest   Master              Running             Running 13 seconds ago

vxphcir5wei5        my-web.2            remcodeds/nginx-page:latest   Worker1             Running             Running 14 seconds ago

sw2sgjgagica        my-web.3            remcodeds/nginx-page:latest   Worker2             Running             Running 14 seconds ago
```

When we publish a port from swarm the following happens: Each request will be processed by swarm load balancer and addressed to node according to the routing algorithm. Such a mesh provides a fault-tolerant system for the application.



Let's ping our service from master node.

```
docker@Master:~$ curl 0.0.0.0:8080
<html>
<body>

134bdb77eb15:80

</body>
</html>
docker@Master:~$ curl 0.0.0.0:8080
<html>
<body>

185c628e640e:80

</body>
</html>
docker@Master:~$ curl 0.0.0.0:8080
<html>
<body>

fb086571b64e:80

</body>
</html>
docker@Master:~$ curl 0.0.0.0:8080
<html>
<body>

134bdb77eb15:80

</body>
</html>
docker@Master:~$
```

As we can see, each time our request gets processed by a new node.

Let’s scale down service to 2 nodes and check how requests will be distributed

```
docker@Master:~$ docker service scale my-web=2
my-web scaled to 2
overall progress: 2 out of 2 tasks
1/2: running   [==================================================>]
2/2: running   [==================================================>]
verify: Service converged
docker@Master:~$ curl 0.0.0.0:8080
<html>
<body>

185c628e640e:80

</body>
</html>
docker@Master:~$ curl 0.0.0.0:8080
<html>
<body>

fb086571b64e:80

</body>
</html>
docker@Master:~$ curl 0.0.0.0:8080
<html>
<body>

185c628e640e:80

</body>
</html>
```

Let’s also test how requests are redistributed if we query each machine in swarm:

```
demo@myvm:~$ docker-machine ls
NAME      ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER     ERRORS
Master    -        virtualbox   Running   tcp://192.168.99.100:2376           v19.03.12
Worker1   -        virtualbox   Running   tcp://192.168.99.101:2376           v19.03.12
Worker2   -        virtualbox   Running   tcp://192.168.99.102:2376           v19.03.12
demo@myvm:~$ ^C
demo@myvm:~$ curl 192.168.99.100:8080
<html>
<body>

d33bdf505278:80

</body>
</html>
demo@myvm:~$ curl 192.168.99.101:8080
<html>
<body>

d33bdf505278:80

</body>
</html>
demo@myvm:~$ curl 192.168.99.102:8080
<html>
<body>

30fcb4f0cd2c:80

</body>
</html>
```

As we can see, each machine can be queried and we will get a response even if there are only 2 instances running these services.

8. How to scale instances in the Docker Swarm? Could it be done automatically?
We can scale services manually using
docker service scale <service-name>=<number-of-instances>

```
docker@Master:~$ docker service scale my-web=2
my-web scaled to 2
overall progress: 2 out of 2 tasks
1/2: running   [==================================================>]
2/2: running   [==================================================>]
verify: Service converged
```

There are no built-in functions in docker swarm for automated scaling, because hosts are created with docker-machine with different drivers. However, it can be automated by scripts using metrics.

Docker Container migration - Application Distribution

9. Validate that when a node goes down a new instance is launched. Show how the redistribution of the instances can happen when the dead node comes back alive.

Initial state:

```
docker@Master:~$ docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS      ENGINE VERSION
n6nqjh7tao9gxqo891gsti8s9 *   Master              Ready               Active              Leader              19.03.12
oek8fn09t7eayw037tqoeyz2z     Worker1             Ready               Active                                  19.03.12
y0mt9uulugck7s4cfu5nxhgdx     Worker2             Ready               Active                                  19.03.12
```

```
docker@Master:~$ docker service ps my-web
ID                  NAME                IMAGE                          NODE                DESIRED STATE       CURRENT STATE            ERROR
     PORTS
bvtma8naaite        my-web.1            remcodeds/nginx-page:latest    Worker2             Running             Running 21 seconds ago

w4deoptdbzai        my-web.2            remcodeds/nginx-page:latest    Master              Running             Running 21 seconds ago

rc68a0h808hm        my-web.3            remcodeds/nginx-page:latest    Worker1             Running             Running 21 seconds ago
```

Next, we will stop Worker2 and see distribution of instances

```
demo@myvm:~$ docker-machine stop Worker2
Stopping "Worker2"...
Machine "Worker2" was stopped.
demo@myvm:~$ docker-machine ls
NAME      ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER     ERRORS
Master    -        virtualbox   Running   tcp://192.168.99.100:2376           v19.03.12
Worker1   -        virtualbox   Running   tcp://192.168.99.101:2376           v19.03.12
Worker2   -        virtualbox   Stopped                                       Unknown
```

```
docker@Master:~$ docker service ps my-web
ID                  NAME                IMAGE                          NODE                DESIRED STATE       CURRENT STATE            ERROR
     PORTS
9z9o8d5ze6zv        my-web.1            remcodeds/nginx-page:latest    Master              Running             Running 25 seconds ago

bvtma8naaite         \_ my-web.1        remcodeds/nginx-page:latest    Worker2             Shutdown            Running 2 minutes ago

w4deoptdbzai        my-web.2            remcodeds/nginx-page:latest    Master              Running             Running 2 minutes ago

rc68a0h808hm        my-web.3            remcodeds/nginx-page:latest    Worker1             Running             Running 2 minutes ago
```

As we can see, master node increases the number of instances it runs.

Next, we will start Worker2 again.

```
demo@myvm:~$ docker-machine start Worker2
Starting "Worker2"...
(Worker2) Check network to re-create if needed...
(Worker2) Waiting for an IP...
Machine "Worker2" was started.
Waiting for SSH to be available...
Detecting the provisioner...
Started machines may have new IP addresses. You may need to re-run the `docker-machine env` command.
```

According to docker documentation, when node goes up, automatic redistribution does not happening in order to prevent interrupting running services

Force the swarm to rebalance

Generally, you do not need to force the swarm to rebalance its tasks. When you add a new node to a swarm, or a node reconnects to the swarm after a period of unavailability, the swarm does not automatically give a workload to the idle node. This is a design decision. If the swarm periodically shifted tasks to different nodes for the sake of balance, the clients using those tasks would be disrupted. The goal is to avoid disrupting running services for the sake of balance across the swarm. When new tasks start, or when a node with running tasks becomes unavailable, those tasks are given to less busy nodes. The goal is eventual balance, with minimal disruption to the end user.

In Docker 1.13 and higher, you can use the `--force` or `-f` flag with the `docker service update` command to force the service to redistribute its tasks across the available worker nodes. This causes the service tasks to restart. Client applications may be disrupted. If you have configured it, your service uses a rolling update.

Automatic task distribution is not happening, so to have the same number of containers on each host we have to force rebalance nodes.

```
docker@Master:~$ docker service update --force my-web
my-web
overall progress: 3 out of 3 tasks
1/3: running   [==================================================>]
2/3: running   [==================================================>]
3/3: running   [==================================================>]
verify: Service converged
docker@Master:~$ docker service ps my-web
ID             NAME           IMAGE                         NODE       DESIRED STATE   CURRENT STATE            ERROR
     PORTS
h9ggp00n952i   my-web.1       remcodeds/nginx-page:latest   Worker2    Running         Running 3 minutes ago

9z9o8d5ze6zv    \_ my-web.1   remcodeds/nginx-page:latest   Master     Shutdown        Shutdown 3 minutes ago

bvtma8naaite    \_ my-web.1   remcodeds/nginx-page:latest   Worker2    Shutdown        Shutdown 8 minutes ago

n9bi6vnwlpfq   my-web.2       remcodeds/nginx-page:latest   Master     Running         Running 3 minutes ago

w4deoptdbzai    \_ my-web.2   remcodeds/nginx-page:latest   Master     Shutdown        Shutdown 3 minutes ago

l5rprhyv8s2j   my-web.3       remcodeds/nginx-page:latest   Worker1    Running         Running 3 minutes ago

rc68a0h808hm    \_ my-web.3   remcodeds/nginx-page:latest   Worker1    Shutdown        Shutdown 3 minutes ago
```

As we can see, container was taken down on Master and was created on Worker 2

10. Perform some update in your application, a minor change in your sample application for example. How to replicate the changes in the rest of the farm servers?

We need to tell docker to update all nodes that don't correspond to, for example, the latest image of service. The update will happen as follows:

1. Update will happen sequentially, so service will keep running on other nodes, while updating other
2. Task will be shutted down on running node
3. Task will be recreated with new image and updater will proceed to other nodes
4. After all nodes are updated, the updater will verify that they are running without error for at least 5 seconds and finish.

```
docker@Master:~$ docker service update --image remcodeds/nginx-page:latest my-web
my-web
overall progress: 2 out of 2 tasks
1/2: running   [==================================================>]
2/2: running   [==================================================>]
verify: Service converged
```

11. It is a good practice to monitor performance and logs on your servers farm. How can this be done with Docker Swarm? Could it be just CLI or maybe GUI?

There are good GUI tools for visualizing performance such as https://docs.docker.com/config/daemon/prometheus/ can be used.
We create this service on already running nodes and it monitors performance metrics of Docker containers running on different hosts.

Playing with Memory

12. Please explain what is "Out Of Memory Exception (OOME)", how it could affect Docker services, and which configuration can be set to avoid this issue?

We can refer to the official Docker documentation. OOME is dangerous because it can make the system unstable and kill other processes.

Understand the risks of running out of memory

It is important not to allow a running container to consume too much of the host machine's memory. On Linux hosts, if the kernel detects that there is not enough memory to perform important system functions, it throws an `OOME`, or `Out Of Memory Exception`, and starts killing processes to free up memory. Any process is subject to killing, including Docker and other important applications. This can effectively bring the entire system down if the wrong process is killed.

Most obvious solution is we can prevent OOME by limiting use of memory by containers, but we also have to follow these rules to prevent it:

You can mitigate the risk of system instability due to OOME by:

- Perform tests to understand the memory requirements of your application before placing it into production.
- Ensure that your application runs only on hosts with adequate resources.
- Limit the amount of memory your container can use, as described below.
- Be mindful when configuring swap on your Docker hosts. Swap is slower and less performant than memory but can provide a buffer against running out of system memory.
- Consider converting your container to a service, and using service-level constraints and node labels to ensure that the application runs only on hosts with enough memory

We can enforce Docker to limit memory usage with the following commands:

Limit a container's access to memory

Docker can enforce hard memory limits, which allow the container to use no more than a given amount of user or system memory, or soft limits, which allow the container to use as much memory as it needs unless certain conditions are met, such as when the kernel detects low memory or contention on the host machine. Some of these options have different effects when used alone or when more than one option is set.

Most of these options take a positive integer, followed by a suffix of `b`, `k`, `m`, `g`, to indicate bytes, kilobytes, megabytes, or gigabytes.

Option	Description
`-m` or `--memory=`	The maximum amount of memory the container can use. If you set this option, the minimum allowed value is `4m` (4 megabyte).
`--memory-swap` *	The amount of memory this container is allowed to swap to disk. See `--memory-swap` details.
`--memory-swappiness`	By default, the host kernel can swap out a percentage of anonymous pages used by a container. You can set `--memory-swappiness` to a value between 0 and 100, to tune this percentage. See `--memory-swappiness` details.
`--memory-reservation`	Allows you to specify a soft limit smaller than `--memory` which is activated when Docker detects contention or low memory on the host machine. If you use `--memory-reservation`, it must be set lower than `--memory` for it to take precedence. Because it is a soft limit, it does not guarantee that the container doesn't exceed the limit.
`--kernel-memory`	The maximum amount of kernel memory the container can use. The minimum allowed value is `4m`. Because kernel memory cannot be swapped out, a container which is starved of kernel memory may block host machine resources, which can have side effects on the host machine and on other containers. See `--kernel-memory` details.
`--oom-kill-disable`	By default, if an out-of-memory (OOM) error occurs, the kernel kills processes in a container. To change this behavior, use the `--oom-kill-disable` option. Only disable the OOM killer on containers where you have also set the `-m/--memory` option. If the `-m` flag is not set, the host can run out of memory and the kernel may need to kill the host system's processes to free memory.

13. Deploy a docker container with at least 15% of CPU every second for memory efficiency.

From the Docker documentation, we can use cpus parameter to achieve this.

`--cpus=<value>`	Specify how much of the available CPU resources a container can use. For instance, if the host machine has two CPUs and you set `--cpus="1.5"`, the container is guaranteed at most one and a half of the CPUs. This is the equivalent of setting `--cpu-period="100000"` and `--cpu-quota="150000"`. Available in Docker 1.13 and higher.

To deploy my-web and limit it to 15% usage:

```
demo@myvm:~$ docker run --cpus="0.15" my-web
```

Compression

14. Verify the size of the Docker images that you're working with. Can this size be reduced and how can we achieve this?

Verify the size of my-web:

```
]
docker@Master:~$ docker images
REPOSITORY            TAG       IMAGE ID       CREATED        SIZE
remcodeds/nginx-page  latest    b42cb789c1ff   18 hours ago   133MB
remcodeds/nginx-page  <none>    85683af5f261   18 hours ago   133MB
remcodeds/nginx-page  <none>    06ee0ce0d979   19 hours ago   133MB
```

There are some common advices for reducing image size:

1. Change base image to alpine.
2. (e.g) FROM nginx:alpine
3. Use multi-stage builds to minimize unused artifacts
4. Don’t install unnecessary for production debug tools
5. Reduce number of layers and perform operations simultaneously
6. run installs with --no-install-recommends to install only main dependencies

After applying these suggestions to my-web image:

```
docker@Master:~/step7$ docker images
REPOSITORY            TAG       IMAGE ID       CREATED         SIZE
remcodeds/nginx-page  latest    5838c1c1f978   7 seconds ago   22.1MB
remcodeds/nginx-page  <none>    b42cb789c1ff   19 hours ago    133MB
```

So we reduced size from 133MB to 22.1MB

Web application

Cluster is currently Up and running on http://34.77.110.39:8888/

First, let's start with creating python scripts for Web-application and Worker.
Web-application is written with Django.

We will present here only important parts of code.

view.py

```
from django.http import HttpResponseRedirect, HttpResponse
from django.shortcuts import render
import hashlib
import time

from redis import from_url

from .forms import GetCode


def get_data(request):
  r = from_url("redis://redis:6379")
  # if this is a POST request we need to process the form data
  if request.method == 'POST':
     # create a form instance and populate it with data from the request:
     form = GetCode(request.POST)
     # check whether it's valid:
     if form.is_valid():
        code = form.cleaned_data['code']
        stdin = form.cleaned_data['stdin']
        lang = form.cleaned_data['language']
        token = hashlib.sha256(code.encode()).hexdigest()
        r.lpush("tasks", f"{token}\n{lang}\n{stdin}{token}{code}")
        url = f'/token/{token}/'
        time.sleep(1)
        return HttpResponseRedirect(url)

  # if a GET (or any other method) we'll create a blank form
  else:
     form = GetCode()

  return render(request, 'code_submit.html', {'form': form})


def get_code_result(request, token):
  r = from_url("redis://redis:6379")
  res = r.get(token)
  if res:
     return HttpResponse(res.decode())
  else:
     return render(request, 'wait_for_code.html')
```

Important points:

1. get_data process data, that user inputed and redirects user to the page with result
2. get_code_results returns program output after execution.

worker.py

```python
from redis import from_url
import subprocess
import sys

import logging


def run(cmd, stdin=""):
    proc = subprocess.Popen(cmd,
                            stdin=subprocess.PIPE,
                            stdout=subprocess.PIPE,
                            stderr=subprocess.PIPE,
                            shell=True
                            )

    stdout, stderr = proc.communicate(input=stdin.encode())

    return proc.returncode, stdout, stderr


def execute_code(code_text, stdin, language, token):
    r = from_url("redis://redis:6379")
    if language == "python3":
        exit_code, out, err = run_py(code_text, stdin)
    else:
        exit_code, out, err = run_gcc(code_text, stdin)

    logging.info(out)
    logging.info(err)
    logging.info(exit_code)
    print("out: '{}'".format(out))
    print("err: '{}'".format(err))
    print("exit: {}".format(exit_code))
    if exit_code == 0:
        if out:
            r.set(token, out)
        else:
            r.set(token, "no output was produced by code")
    else:
        r.set(token, err)


def run_py(code_text, stdin):
    with open("tmp.py", "w") as file:
        file.write(code_text)

    exit_code, out, err = map(lambda x: x.decode() if type(x) == bytes else x,
                              run([f"{sys.executable} tmp.py"], stdin))
    run(["rm tmp.py"])

    return exit_code, out, err


def run_gcc(code_text, stdin):
    with open("tmp.c", "w") as file:
        file.write(code_text)
```

```
    exit_code, out, err = map(lambda x: x.decode() if type(x) == bytes else x,
                              run(["gcc tmp.c && ./a.out"], stdin))
    run(["rm tmp.c"])
    run(["rm a.out"])

    return exit_code, out, err


def main():
    while True:
        r = from_url("redis://redis:6379")
        token, lang_input_code = r.brpop("tasks")[1].decode().split("\n", 1)
        lang, input_code = lang_input_code.split("\n", 1)
        stdin, code = input_code.split(token, 1)
        execute_code(code, stdin, lang, token)


if __name__ == '__main__':
    main()
```

Important points:

1. program blocks on redis list to avoid pulling
2. After token is received, it is processed and passed to execute_code
3. execute_code depending on the language, execute the source program
4. result is written into the redis database

Dockerfile for Web-App

```
FROM python:3
ENV PYTHONUNBUFFERED 1
RUN mkdir /code
WORKDIR /code
COPY requirements.txt /code/
RUN pip install -r requirements.txt
COPY . /code/
```

Dockerfile for Worker

```
FROM python:3
ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update && apt-get -y install gcc mono-mcs && rm -rf /var/lib/apt/lists/*
RUN pip install redis
COPY worker.py .
CMD python worker.py
```

docker-compose.yml

```
version: '3'

services:
  postgres:
    image: postgres:alpine
    environment:
      - POSTGRES_DB=postgres
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres

  redis:
```

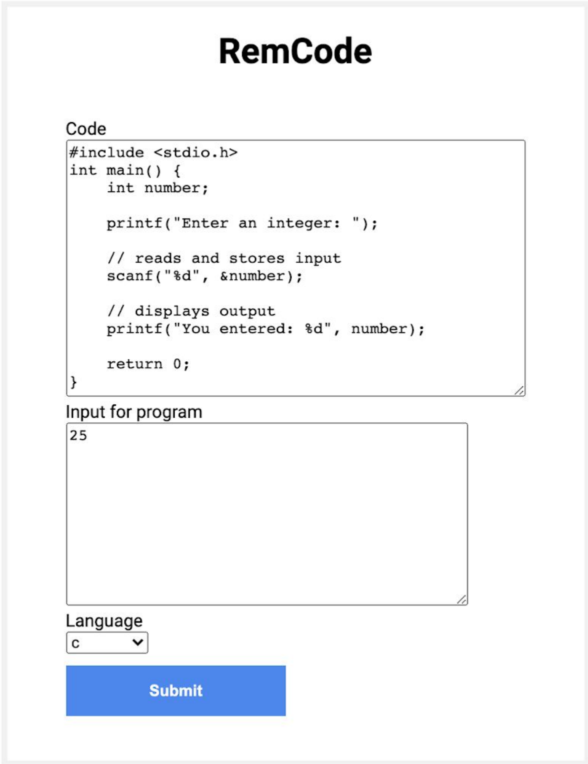
```
  image: redis:alpine

webapp:
  build: remCodeWeb
  command: python manage.py runserver 0.0.0.0:8000
  ports:
    - "8000:8000"
  depends_on:
    - postgres
    - redis

worker:
  build: remCodeWorker
  depends_on:
    - redis
```
it will be used with docker stack deploy

Deployment:

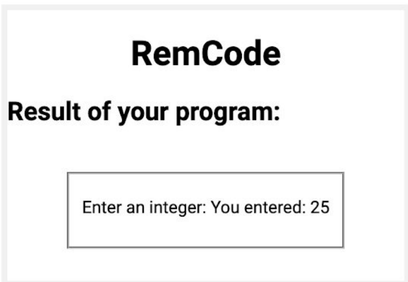
1. push all locally build images to the repository
   a. docker push remcodeds/worker
   b. docker push remcodeds/web-app
2. replace all locally build images in docker-compose.yml with pushed ones
3. deploy stack
```
docker@Master:~/step7$ docker stack deploy --compose-file docker-compose.yml remcode
Creating network remcode_default
Creating service remcode_webapp
Creating service remcode_worker
Creating service remcode_postgres
Creating service remcode_redis
```
4. Check status of stack
```
docker@Master:~/step7$ docker stack services remcode
ID              NAME               MODE         REPLICAS   IMAGE                        PORTS
ex9nxwslrl32    remcode_redis      replicated   1/1        redis:alpine
hys5jvxpqeh9    remcode_postgres   replicated   1/1        postgres:alpine
npu0v9wl5voz    remcode_worker     replicated   1/1        remcodeds/worker:latest
oas4j56a3bhb    remcode_webapp     replicated   1/1        remcodeds/web-app:latest     *:8000->8000/tcp
```
   as we can see, every service has been replicated=1 and web-app has opened port
5. As most of the workload will be on workers, we can scale them to 4 instances.
```
docker@Master:~/step7$ docker service scale remcode_worker=4
remcode_worker scaled to 4
overall progress: 4 out of 4 tasks
1/4: running   [==================================================>]
2/4: running   [==================================================>]
3/4: running   [==================================================>]
4/4: running   [==================================================>]
verify: Service converged
```
6. We can also scale web-app to 2 instances.
```
docker@Master:~/step7$ docker service scale remcode_webapp=2
remcode_webapp scaled to 2
overall progress: 2 out of 2 tasks
1/2: running   [==================================================>]
2/2: running   [==================================================>]
verify: Service converged
```
7. The final picture of deployed services looks like this:
```
docker@Master:~/step7$ docker stack services remcode
ID              NAME               MODE         REPLICAS   IMAGE                        PORTS
a901d3xsttrx    remcode_postgres   replicated   1/1        postgres:alpine
hjwd5a0pp3c2    remcode_worker     replicated   4/4        remcodeds/worker:latest
uicd7a31h202    remcode_redis      replicated   1/1        redis:alpine
wqgehcspqtex    remcode_webapp     replicated   2/2        remcodeds/web-app:latest     *:8000->8000/tcp
```

Web interface

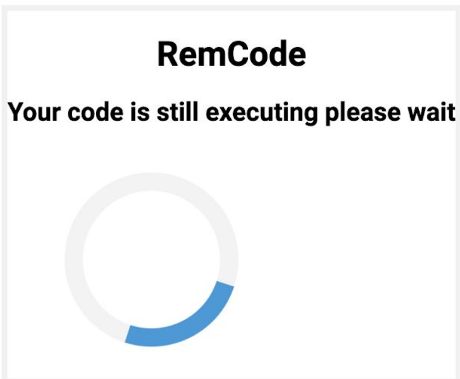
The main page:



Page with results of program:



Page with message if results are not ready yet:



Conclusion

What we learned

1. What is and how to use docker-machine
2. How to deploy our own swarm cluster
3. How to manage nodes in swarm and deploy new Manager nodes
4. How to build scalable applications
5. How to deploy applications in docker cluster and how to scale them
6. How to limit resources used by containerized application

Difficulties

1. Come up with scalable architecture for application
2. How to pass data between containers
3. How to build scalable worker application