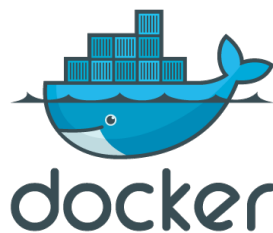




Process monitoring with containers in



David Prelogović, University of Zagreb, Croatia

September 7, 2017

Abstract

Description of software containers in general, what they are, differences with respect to virtual machines. After describing basic advantages of Docker containers and how to use them, we will take a look into monitoring containers and other processes on the host system.

Contents

1. Introduction	3
1.1. Container v.s. VM	3
1.2. Docker image structure	4
1.2.1. Namespaces	5
1.2.2. Control groups	5
1.3. My first container	5
2. Docker containers - deeper dive	6
2.1. Networking	6
2.1.1. Port publishing	8
2.2. Volumes mounting	8
2.2.1. CVMFS	9
2.3. Dockerfile	9
2.3.1. Cern Root example	9
2.4. Registry	11
3. Monitoring containers	12
3.1. Collecting stats - cAdvisor	12
3.2. Processing stats - Logstash	14
3.3. Saving stats - Elasticsearch	14
3.4. Showing stats - Kibana	15
4. Conclusion	16
5. Acknowledgements	16
Appendix A. Monitoring condor processes	17
Appendix B. Setting up Logstash container	19

1. Introduction

Reader might ask herself/himself if this project will be about dockers and containers in harbours, ships, how to put containers on them, etc. So I would like to cheer you up (or disappoint) from the start by saying it won't be. Our subject is far from any ships, containers or any metal in general. Here we will talk about software containers and how they can make someone's life easier (maybe). Or at least less painful. Our main goal will be a bit farther from that point though, as we will try to see how containers can help us in monitoring processes.

As one could suspect, naming software containers as containers is not just a coincidence, but indeed, a matter of convenience. Because they do exactly what one would say: they contain software in a closed package, which you can very easily 'ship' - using software called *Docker*. As with containers in 'real life', one can ship more goods, or in our case applications, more quickly and easily. (There are also other software solutions for containers besides *Docker*, for example *Singularity*, with some differences, advantages and disadvantages, but here our main focus will be on *Docker*.)

1.1. Container v.s. VM

Both containers and virtual machines serve the purpose of isolating processes, resources, network. Because most of the readers probably had some experience, or at least heard of virtual machines, the best way to explain what containers are, is to see what differs them from VM's.

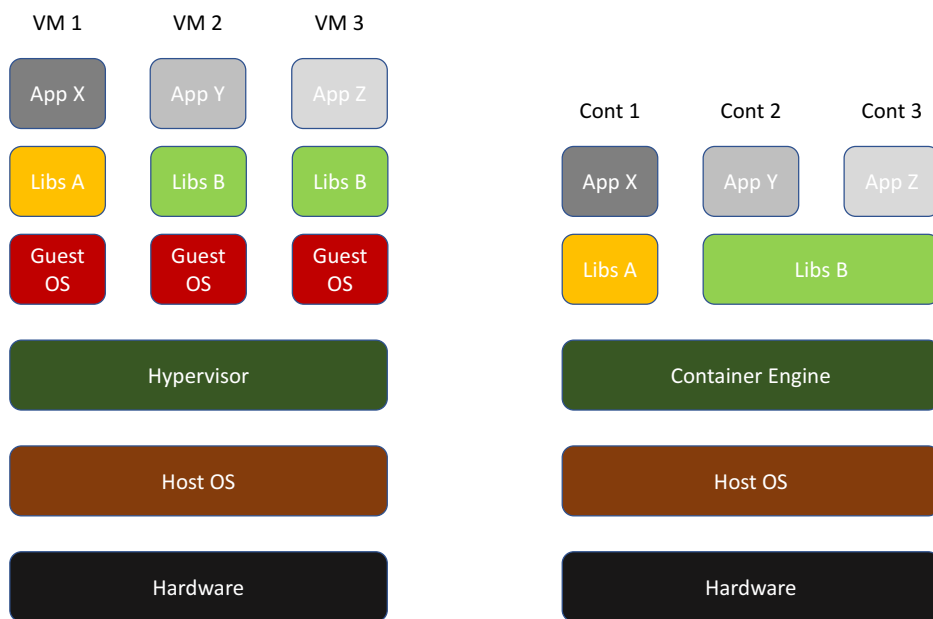


Figure 1: Structural difference of processes (apps) running inside of containers and VM's on a single host.

Fig. 1 shows an example for the difference in structure between three processes running on the host in virtual machines and in containers. We demand separation of those processes. Two main differences are the following: first, apps inside of containers are running directly on the host OS with help of a Container Engine, where in the case of VM's, apps are running inside of guest OS ran by Hypervisor - therefore, each app requires whole OS just for itself. (It has to be mentioned though, that apps inside of a container are constrained by the kernel version of the host OS.) Second difference is that with containers, apps can share libraries, which is essential for the storage efficiency of containers.

Before starting our first container, let's briefly explain how Container Engine creates a container from image.

1.2. Docker image structure

To run a process inside of virtual machine, you need to install Guest OS inside of it, and then install all libraries you need to run processes in it. Similarly for containers, you need a base image. Most commonly that is stripped-down version of some operating system, for example Ubuntu, Centos, Debian. Based on this images, you can build your own, with all libraries installed and services properly configured. Later, we will take a look how to properly create our own image from base one, in particular we will build an image which by default runs *Cern Root* software and is based on *CentOS 7* image.

For now, I would just like to describe how an image looks like, and what happens when Container Engine starts a container from that image.

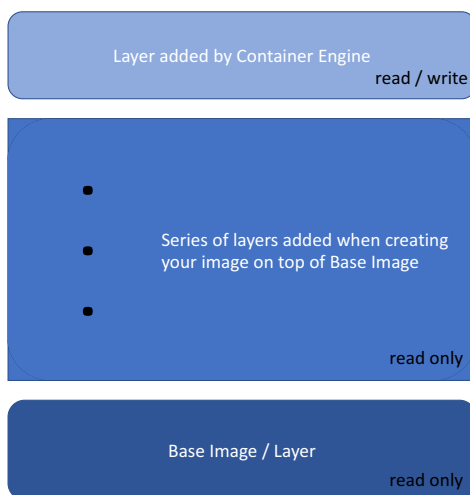


Figure 2: Scheme of image layers.

Fig. 2 shows structure of a container after Engine has started it. It differs from an image just in top **read/write** layer. Docker is controlling all layers of a container with special type of file system called *Union File System*. Details of how it works can be found

[here](#). What is worth mentioning is that this feature gives to Docker containers storage efficiency. If you would like to start a new container from the same image, just one new **read/write** layer would be created, nothing more.

Besides mentioned *UnionFS*, there are few more underlying technologies, which give to containers an isolation from the rest of the host OS, or more precisely they create container with it's content and state inside of host OS.

1.2.1. Namespaces

Docker uses **namespaces** to provide the isolated workspace - container. When you run a container, Docker creates a set of namespaces for that container.

These namespaces provide a layer of isolation. Each aspect of a container runs in a separate namespace and its access is limited to that namespace. Docker Engine uses namespaces such as the following:

- **pid namespace**: Process isolation (PID: Process ID).
- **net namespace**: Managing network interfaces (NET: Networking).
- **ipc namespace**: Managing access to IPC resources (IPC: InterProcess Communication).
- **mnt namespace**: Managing filesystem mount points (MNT: Mount).

1.2.2. Control groups

Docker Engine on Linux also relies on *control groups* (**cgroups**). A cgroup limits an application to a specific set of resources. Control groups allow Docker Engine to share available hardware resources to containers and optionally enforce limits and constraints. For example, you can limit the memory available to a specific container, CPU share, and so on.

Usage of **cgroups** is important for monitoring containers, because it gives live data about resources for every process on the host OS, and therefore also for each container.

1.3. My first container

Let us finally try to start our first container. After [installing Docker](#), just run in terminal:

```
docker run -it centos /bin/bash
```

With this command what happend is the following. Docker daemon (**dockerd**) received command from terminal for starting container based on **centos:latest** image (this is default behaviour, otherwise you could specify version of **centos**). It looked if the image exists in local files, if not, it pulled it from the *Docker Hub* [4] and started **/bin/bash**

process inside of container, interactively (`-it` part of a command). So you have found yourself inside of a container, where you can run `bash` commands. Instead of `bash` you could also run any other process, or more of them.

`dockerd` is controlling all Docker operations. From processing user's requests, making and holding containers, monitoring them, pulling images from registries. *Docker Hub* is official Docker registry where anyone can publicly store his/her own images, or pull other's. But if you wish to, you can easily set up and run your own registry, for storing personal images. More about this topic will be covered later.

2. Docker containers - deeper dive

Now we will take a look into details of Docker containers, how to connect them, how to store and save data from them, how to build images, etc.

For the beginning let's take a look as an example of how isolation works in container for the `pid` namespace isolation. For this purpose we will use `pstree` command on the host OS and inside of a container, which shows all processes visible in parent-child format. I have started container and in it few `sleep` processes. The following is the output of the command on the two, respectively.

```
systemd--cron
  | -dbus-daemon
  | -dhclient
  | -dockerd--docker-containe--docker-containe--bash---5*[sleep]
  |           |                               |
  |           |                               `--8*[{docker-containe}]
  |           |
  |           `--9*[{dockerd}]
  | -login---zsh---pstree
  |
  ...
```

```
bash--pstree
  `--5*[sleep]
```

As we can clearly see, processes inside of a container are isolated. But if we for some reason need to change this behaviour and break `pid` namespace isolation, we can easily do that by passing `--pid=host` flag when starting (running) a container. Then, the previous command gives the same output. What's also visible from the previous output is that `dockerd` sits as a parent of all container processes.

2.1. Networking

Docker by default gives us a lot of functionality when comes to networking inside of containers. When you are creating a container, you can add it to a network. To see

(list) all currently available networks just run `docker network ls`. By default there are three of them: `bridge`, `host` and `none`. It should be an easy guess that `host` network represents host OS network, corresponding to the network card usually named `eth0`. Similar guess works for the `none` network also. When one installs Docker, new network device is added, `docker0` (you can inspect that by running `ip addr` or `ifconfig` in your terminal), which corresponds to `bridge` network listed before. This is a default network to which every container is connected if you do not specify another one explicitly. Also, `dockerd` creates new network device for every container which you run.

So let's finally see how would you connect two containers so that you can communicate between them. There are two possible ways. The first one is following.

```
docker run -it --detach=true --name=cont1 some_image
docker run -it --detach=true --name=cont2 --link=cont1 some_image
```

We have two containers named `cont1/2` running in `detached` mode. Because we didn't specify the network, they are both connected to the default one. That means that, with any further settings, containers are basically connected, because they are part of the same network, with local IP addresses. That means if we find out IP addr of each container we could communicate between them. One can do that very easily, just by running `docker network inspect bridge` - to see all containers connected to `bridge` network and their info, or one can run `docker inspect cont1` - to see all info about particular container.

But that sounds, and indeed it is very tedious. So there is where `--link=cont1` part comes into play. Basically it just maps `cont1` IP address with it's name, so one could run `ping cont1` inside of `cont2` and it should work properly.

There is more elaborate way of connecting containers, and that is by using user-defined networks inside Docker. Let's create one network.

```
docker network create --driver=bridge my_network
```

Now if we list all networks, we will see the new one on the list with it's name and defined driver. And also if we list all network devices available on our system we will see one more. (There are more types of networks in Docker besides `bridge`, they are used in more complicated setups with more hosts, but the basic principles are the same)

Let's add some containers to our network.

```
# -d <=> --detach=true
docker run -itd --network=my_network --name=cont1 some_image
docker run -itd --network=my_network --name=cont2 some_image
```

What is really nice with containers connected to the same user-defined network is they are all discoverable by their names inside of other containers. That is possible because `dockerd` runs DNS (Domain Name System) server for every user-defined network.

It should be noted that containers can communicate with outside world, just as your host machine, but one could easily disable it so that the whole world visible to a container would be just the network to which it is connected.

2.1.1. Port publishing

If we have a container connected to some user-defined network and we wish that someone from the outside can reach that container, then we would use port publishing. E.g., `nginx` is a web server which by default runs simple web page on port 80. There is a Docker image for it, so we can run it inside of container.

```
docker run -d --name=nginx --network=my_network --publish=8888:80 nginx
```

Now, we can type in our browser `localhost:8888`, and should see default `nginx` message, which means that port 8888 on the host is forwarded to port 80 in container.

2.2. Volumes mounting

When we start a new container, `read/write` layer is created on the top of an image. When we remove it, the whole layer is removed and no data is saved. That is totally fine if we run some processes inside of it, so we don't care about data inside of it. But if we want to save data produced by some process inside of a container, or if we need some data for running the process in the first place (let's say some script or code), then what we could do is mount a path from our host to some path inside of a container.

```
docker run -d --volume=/path/to/docs:/some/path/to/docs some_image
```

This way, everything we write in `/path/to/docs` on the host will be visible inside of a container at `/some/path/to/docs` and vice versa.

If I don't care about the path on my host where container stores data, but I just care it's somewhere, I could leave `dockerd` do it for me and just run:

```
# you can omit the first command, since volume is created if there is none
docker volume create --name=some_volume
docker run -d --volume=some_volume:/some/path/to/docs some_image
```

In most cases, you don't care about where is your data exactly - as long as you know it's on some isolated location and that you can access it if you want to. What you get

in return with this approach is portability, because you don't have hard-coded paths mounted inside of a container.¹

2.2.1. CVMFS

[CernVM File System](#) [2] is used by majority of particle physicists and others around the World, and the question is can you use it inside of a container. Because it's much more complex than some folder on your PC, it has to be mounted properly.

```
docker run --volume=/cvmfs:/some/path/to/cvmfs:shared some_image
```

2.3. Dockerfile

Until now we talked about how to use an image. In this section we will see how to build one, and in the next one how to save it on some remote location.

How one could basically create his own image is just by starting, let's say CentOS 7 container, installing all libraries he needs, exit container and run `docker commit container_id`, `docker tag --tag=some_image_name image_id`. And there you have it, your very own image would be ready. But then if one would need to make some small change: install new version of package, update to CentOS 8, or something similar, odds are he would need to do all steps by himself from the beginning.

That's way Dockerfile comes into play. Basically what you do is put all configuration commands into a file and run

```
docker build --tag image_name .
```

2.3.1. Cern Root example

For my own exercise I've created Dockerfile with which you can build image for Cern Root software.

```
1  #Dockerfile for cern-root
2
3  #number of commands equals number of image layers, therefore it's good to
4  #combine some of them into one
5
```

¹ In some earlier versions of Docker, there was no option for creating named volumes, and therefore you would use data-only containers, where you would create container with volumes and then mount them all inside desired container. With an option to create named volumes, this practice becomes obsolete.

```

6  #base image centos:latest
7  FROM centos
8
9  #downloading root and installing dependencies
10 RUN cd /opt \
11     && curl -O \
12     https://root.cern.ch/download/root_v6.10.02.Linux-centos7-x86_64-gcc4.8.tar.gz \
13     && tar -xzf root_v6.10.02.Linux-centos7-x86_64-gcc4.8.tar.gz \
14     && rm root_v6.10.02.Linux-centos7-x86_64-gcc4.8.tar.gz \
15     && yum install -y make gcc-c++ gcc binutils \
16     libX11-devel libXpm-devel libXft-devel libXext-devel \
17     && yum clean all
18
19 #installing additional packages
20 RUN yum install -y gcc-gfortran openssl-devel pcre-devel \
21     mesa-libGL-devel mesa-libGLU-devel glew-devel ftgl-devel mysql-devel \
22     fftw-devel cfitsio-devel graphviz-devel \
23     avahi-compat-libdns_sd-devel libldap-dev python-devel \
24     libxml2-devel gsl-static \
25     && yum clean all
26
27 #setting up user and ownerships for some folders
28 ENV HOME /home/cern-root-user
29 RUN useradd --create-home --home-dir $HOME cern-root-user \
30     && chown -R cern-root-user:cern-root-user $HOME \
31     && mkdir /data \
32     && chown -R cern-root-user:cern-root-user /data
33
34 #working directory
35 WORKDIR /data
36 #defining user
37 USER cern-root-user
38 #adding post-install script to the .bashrc which defines all variables for
39 #cern-root, and then adding command which runs actual root
40 RUN echo "source /opt/root/bin/thisroot.sh" >> $HOME/.bashrc \
41     && echo "root" >> $HOME/.bashrc

```

As it says in comment (3 – 4) number of commands equals number of layers in final image. With “FROM” we define base image. The following “RUN” commands are just what we would write in terminal when installing Cern Root. Part (27 – 32) defines user and gives proper privileges to it for some folders. For security reasons, one should always run processes in a container as a user, ie. not root (if not necessary).

We can now build this image as:

```
docker build -tag=cern-root .
```

Additionally, if we want to have display output from the container, we should pass

DISPLAY variable and mount `.X11` folder. What would also be nice, is to have some data or code inside of a container for analysis with `root`. We can therefore run container such way that folder in which we are currently in, is mounted on `/data` inside of a container. All together, we would run our container from `cern-root` image as following.

```
docker run -it \  
  --rm \  
  --volume=/tmp/.X11-unix:/tmp/.X11-unix \  
  --env=DISPLAY=unix$DISPLAY \  
  --name=root<F3> \  
  --volume=$PWD:/data \  
  cern_root
```

2.4. Registry

In this section we will take a look how to setup basic, unsecure registry where you can push your images. Unsecure, because anyone with an access to IP address of the machine on which it is running will be able to push and pull from it. For more advanced setup with `https` support take a look [here](#).

Setting up basic registry is very easy once you have host which you can discover either by IP address or domain name. There is already an image on *Docker Hub*, you just have to start a container.

```
docker run \  
  --detach=true \  
  --publish=5000:5000 \  
  --restart=always \  
  --name=registry \  
  --volume=registry:/var/lib/registry \  
  registry:2
```

Flag `--restart=always` instructs `dockerd` to restart container in all circumstances. Now if you want to push images from you PC to that registry, just put the following into `/etc/docker/daemon.json`:

```
{  
  "insecure-registries" : ["address_of_registry:5000"]  
}
```

where `address_of_registry` is IP address, or domain name of registry host. To push `some_image`:

```
docker tag some_image address_of_registry:5000/username/some_image
docker push address_of_registry:5000/username/some_image
```

3. Monitoring containers

We have finally covered all needed for our main goal, to monitor containers or processes in them. Comparing process running on the host and the one running inside of a container, besides the advantage of being isolated, process in a container is much easier to monitor. Furthermore, because every container has dedicated network device, you can record how much data your process is pulling or uploading to the network, which is something you cannot record or constrain for normal process.

If you want to monitor a process on your machine, you would use **cgroups**, and the same thing you use for containers. Our goal here is not constraining containers by how much CPU or memory should they use, but just to capture how much of the resources they are using at the moment, saving those stats somewhere for later processing and showing some graphical output of it.

For monitoring we will use a set of tools, which all can be run inside of a container, so we will have completely containerized environment. In next sections we will in short introduce those tools and at the end combine them into one compact environment which you can run on each host to collect data for all containers running on it.

3.1. Collecting stats - cAdvisor

There are few possible ways to collect stats about containers running on a host. You could use built-in **docker stats** command or similar tools from Docker, or you could use some other tool. But at the end it all eventually come down to reading **cgroups** and somehow combining that data.

Therefore we won't reinvent the wheel but will use a tool called *cAdvisor*. What's even more, you can run it as a container, and the metrics it collects are not constrained just to docker containers, but to all processes in **cgroups**. It collects live data which you can see on web interface. An example of CPU usage for one container is shown in Fig. 3. But we are not particularly interested in web interface, we want to use cAdvisor to collect data, process it and then send it in some database, from where we can pull it and display it as we wish. When starting cAdvisor container, one needs to mount some of the host's folders so it can access data it needs.

```
docker run \
  --volume=/:/rootfs:ro \
  --volume=/var/run:/var/run:rw \
```

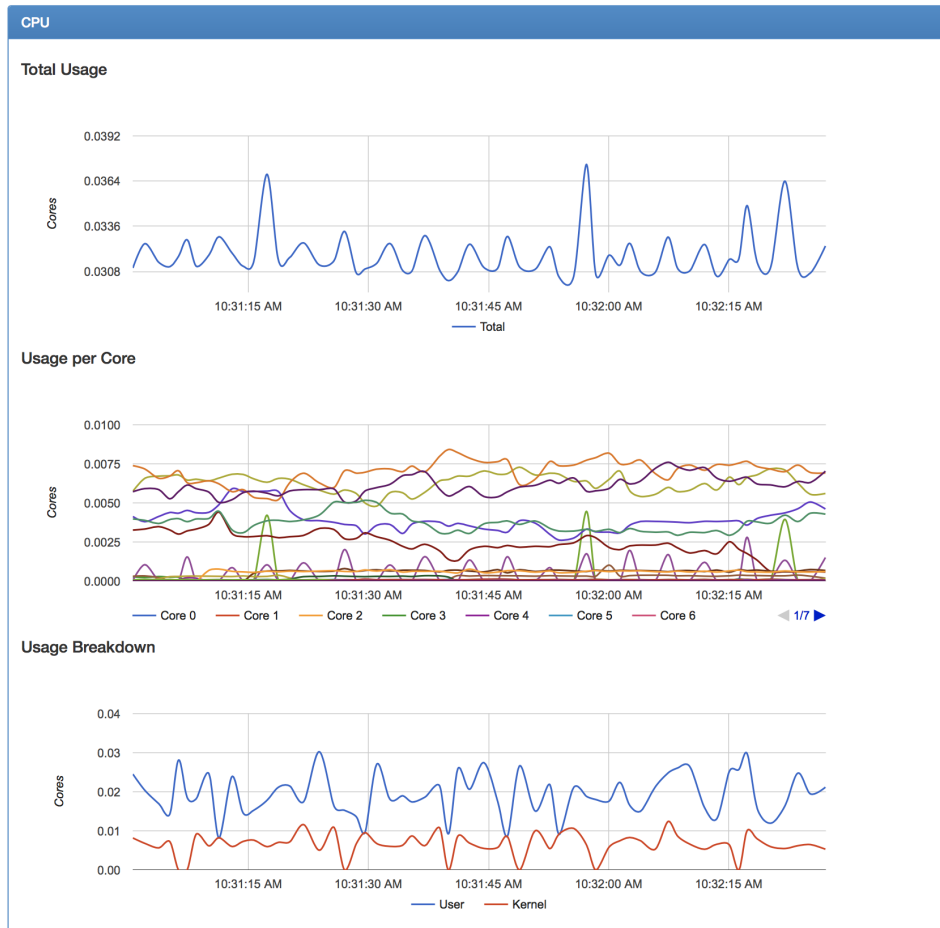


Figure 3: CPU usage for container in cAdvisor's web interface.

```
--volume=/sys:/sys:ro \
--volume=/var/lib/docker/./var/lib/docker:ro \
--volume=/dev/disk/./dev/disk:ro \
--publish=8080:8080 \
--detach=true \
--name=cadvisor \
google/cadvisor:latest
```

All statistics visible on Fig. 3 together with the ones for memory, network, etc., are available on cAdvisor's web API in form of `.json` file. For example if we would want to get stats for all Docker containers currently running on the host, we could type:

```
curl -XGET 'localhost:8080/api/v1.3/docker'
```

supposing cAdvisor is running on localhost:8080. v1.3 is API version. The same in Python would be:

```
import requests
import json
stats = requests.get('http://localhost:8080/api/v1.3/docker').json()
```

In Appendix A you can find script which prints out stats about all `condor` processes running on a machine. What is `condor` exactly is not important at the moment, all you need to know is that it can start any process (also any Docker container) on the machine in a `condor cgroup`, and from all processes running on the host I pick just ones started by `condor`.

3.2. Processing stats - Logstash

With script in Appendix A, we extracted stats about processes of our interest in form of `.json 'blob'`, now we would like to send this data to some database. We could send data directly from python script to adress where we have our database running, but the better way is to use Logstash. Logstash in general can be used to filter data first, and then sending it to database, but here, we will use it just as a mediator. Reason for this is ease of use, portability and scalability. We will set up logstash container with all Python scripts already in it, so all you will need to do is start a container and it would have already collected some stats and sent it to desired database location.

What's even more, with Logstash container, it is very easy to add some additional data. For that, we would mount host system inside of a container, for example as `--volume=/:/rootfs`. Therefore with additional effort one could extract data not visible at cAdvisor. For it, it's often necessary to break `pid` isolation.

Dockerfile and logstash settings can be seen in Appendix B.

3.3. Saving stats - ElasticSearch

The database we use is *ElasticSearch* [5]. It works very well with Logstash, it is NoSQL (no sequel) type of database, which means you can save data of different structures, and it does data selection on your request very well. To start ElasticSearch instance as Docker container you could run:

```
docker run \
  --detach=true \
  --volume=elasticsearch/data:/usr/lib/elasticsearch/data \
  --volume=elasticsearch/logs:/usr/lib/elasticsearch/logs \
  --publish=9200:9200 \
  --publish=9300:9300 \
  --name=elasticsearch \
  elasticsearch:1.7.0
```

Now you just have to write IP address where your ES is running in `logstash.conf` from Appendix B and you are ready to go.

All data saved on it is available on web interface, where you can also check if ElasticSearch instance is running properly by visiting: `elasticsearch:9200/`, where `elasticsearch` is IP address of it.

3.4. Showing stats - Kibana

Now, at the end, collecting data from containers and other processes works and is send to our ElasticSearch instance. We would like to see it in some nicer format, with some graphs of CPU usage and so on. For it we can use *Kibana*. Basically it's graphical interface which can send queries to ElasticSearch, receive data from it, process it and display it in nice format.

Running Kibana is pretty straightforward, just start container:

```
docker run \
  --restart=always \
  --detach=true \
  --name=kibana \
  --env=ELASTICSEARCH_URL=http://elasticsearch:9200 \
  --publish=5601:5601 \
  kibana:4.1.1
```

where again `elasticsearch` is IP address or domain name of ElasticSearch instance, and Kibana is running on port 5601.

On Fig. 4 there is a basic Kibana setup with graph about amount of data sent to ElasticSearch in last 12h. At the bottom you can see beginning of a `.json` 'blob' from last event.

Appendix A Monitoring condor processes

Following is a Python script for pulling stats from cAdvisor about all processes on the host started by condor. Prerequisite is to have cAdvisor running on `cadvisor:8080`.

```
1  import requests
2  import json
3  import datetime
4
5  import condor_job_statistics
6
7  def timestamp_to_time(timestamp):
8      date, microseconds = timestamp.split('.')
9      timestamp_time = datetime.datetime.strptime(date, "%Y-%m-%dT%H:%M:%S")
10     timestamp_time += datetime.timedelta(seconds = int(microseconds[0] > '4'))
11     return timestamp_time
12
13 def datetime_diff(t1, t2):
14     delta = (t1 - t2).total_seconds()
15     if delta < 0.:
16         return -delta
17     else:
18         return delta
19     return False
20
21 def find_closest_time_index(timestamps, time):
22     min_time = datetime_diff(time, timestamp_to_time(timestamps[0]))
23     c_t_i = 0
24     for i in range(1, len(timestamps) - 1):
25         temp = datetime_diff(time, timestamp_to_time(timestamps[i]))
26         if temp < min_time:
27             min_time = temp
28             c_t_i = i
29     return c_t_i
30
31
32 try:
33     condor_data = requests.get(
34         'http://cadvisor:8080/api/v1.3/containers/system.slice/condor.service').json()
35 except:
36     condor_data = requests.get(
37         'http://cadvisor:8080/api/v1.3/containers/htcondor').json()
38
39
40 try:
41     subcontainers = condor_data['subcontainers']
42 except:
43     quit()
44
45
```

```

46 for subcontainer in subcontainers:
47     all_process_data = requests.get(
48         'http://cadvisor:8080/api/v1.3/containers' + subcontainer['name']).json()
49
50     time = datetime.datetime.utcnow() - datetime.timedelta(seconds = 30)
51     timestamps = [i['timestamp'] for i in all_process_data['stats']]
52     closest_time_index = find_closest_time_index(timestamps, time)
53
54     process = {}
55     process['name'] = all_process_data['name']
56     process['spec'] = all_process_data['spec']
57     process['stats'] = all_process_data['stats'][closest_time_index]
58
59     #adding additional data
60     additional_process_data = condor_job_statistics.getCgroupStats(process['name'])
61     process['process_info'] = json.loads(json.dumps(additional_process_data))
62
63     print json.dumps(process)

```

Appendix B Setting up Logstash container

```
1  # Dockerfile
2  FROM docker.elastic.co/logstash/logstash:5.5.2
3
4  USER root
5  RUN yum install -y python-requests && yum clean all
6  USER logstash
7
8  RUN mkdir /usr/share/logstash/documents
9  ADD condor_job_statistics.py /usr/share/logstash/documents/
10 ADD condor_services_logstash.py /usr/share/logstash/documents/
11
12 ADD logstash.conf /usr/share/logstash/pipeline/logstash.conf
```

Notice that we have to have all files in the same directory as Dockerfile so to be able to copy them (with ADD command) to container.

```
1  #logstash.conf
2
3  input {
4    exec {
5      command => "/usr/share/logstash/documents/condor_services_logstash.py"
6      interval => 300
7      codec => json_lines
8      type => "condor_job_monitoring_info"
9    }
10 }
11 filter {
12   if [type] == "condor_job_monitoring_info" {
13     json {
14       source => "message"
15     }
16     date {
17       match => [ "timestamp", "YYYY-MM-dd HH:mm:ss" ]
18       timezone => "CET"
19     }
20   }
21 }
22 output {
23   elasticsearch {
24     hosts => ["elasticsearch:9620"]
25     index => "stats-%{type}-%{+YYYY.MM.dd}"
26   }
27 }
```

From this .conf file we can easily see structure of Logstash process as input -> filter -> output.

References

- [1] Using Docker: Developing and Deploying Software with Docker, *Adrian Mouat*, O'Reilly Media Inc., 2016.
- [2] [CernVM File System](#)
- [3] [Docker Documentation](#)
- [4] [Docker Hub](#)
- [5] [ElasticSearch](#)