SWitCH – Sistemas de Computação e Redes (SCOMRED)

2021/2022

Laboratory 04

BASH programming – Algorithmics.

Scheduled commands execution in Linux – CRON.

Command-line utilities - using and managing Docker containers.

1. Practical exercise: a BASH script for network monitoring

Create a BASH script to check hosts availability on the network, such checking will be implemented by performing ICMP echo tests (ping).

1.1. The script is to meet the following requirements:

- The script file to be created is **/root/bin/check-hosts**.
- The script must be designed to run unattended, possibly in background, therefore it shouldn't interact with the terminal (stdin, stdout or stderr).
- The hosts to be checked are stored in the text file **/etc/check-hosts.conf**. Hosts may be represented by an IP address or a DNS name. Several hosts may be placed in the same line separated by spaces. Any line containing a hash signal should be ignored.
- A host availability checking is performed by sending to its address **a single ICMP echo request**, if an ICMP echo response is received within 2 seconds the test is successful and the host is regarded as being available, otherwise as unavailable.
- The script should log its actions into the file **/var/log/check-hosts.log**, this means appending text lines describing what is doing and each achieved result, bear in mind timestamps are mandatory on log files.
- Whenever a host is unavailable, a message should be sent to the **root** user in the system. For this purpose, the **write** command is to be used. The write command has some outstanding limitation, the target user is required to have an active terminal session on the system, otherwise the message is lost. One other issue with the write command is that if the user has several terminal sessions open, then the message is delivered only to one of those sessions.
- When the script is run, it will create an HTML page and store it on file **/var/www/html/check-hosts.html**. (Thus, this page will be available as **https://vsX.dei.isep.ipp.pt/check-hosts.html**, where **vsX** is your virtual server's DNS name). This file's content is replaced each time the script is run, the contents must encompass at least a timestamp for when the file was generated, the list of hosts being checked and for each the status at that time (available/unavailable).

1.2. Implementation guidelines and hints.

- Use good programming practices, e.g., avoid constants spread around your code, always store them on variables on the first lines of the script, and of course this includes filenames.

- Use incremental programming, this algorithm encompasses a main loop than will go through the hosts list, then for each host several features are required to be implemented. Start by implementing the main loop, then other features can be gradually added and tested one by one at a time.

- Regarding the test itself, you may use the **ping** command. You could focus on the command's output, but a much better option is focusing on its return value (exit code).

2. Task scheduling in Linux – the CRON service

Tasks scheduling is an important capability most operating systems make available to users, it allows a user to schedule a task execution (e.g., a program or a script) to a given time and date, and the user is not required to be logged in at the time of execution.

This is especially useful for systems administrators as they can schedule the periodic execution of unattended maintenance operations, most often at a time of low load on systems. Some examples of such operations are software updates and data backups.

CRON is the most popular tasks scheduler for Linux and other UNIX like operating systems, CRON is a service running in background (in UNIX such processes and services are commonly referred to as **daemons**).

2.1. The /etc/crontab configuration file

For every minute tick, the CRON service wakes and reads the content of the **/etc/crontab** configuration file, then it matches the current date/time with each entry on the file, for every match the corresponding task is executed.

Each line in **/etc/crontab** has at least seven attributes separated by spaces, the first five are a time/date pattern, if the current time matches that pattern, then CRON will **execute the command specified as 7th attribute using the username specified as 6th attribute**. After the 7th attribute arguments to be passed to the command may be used.

The first five attributes (time/data pattern) are:

1st – minute – legal values are integers from 0 up to 59.

2nd – hour – legal values are integers from 0 to up to 23.

3rd – day of the month – legal values are integers from 1 up to 31.

4th – month – legal values are integers from 1 to up to 12.

5th – day of the week – legal values are integers from 0 up to 7. Both 0 and 7 stand for Sunday.

For each of these attributes, a single value or a list of comma-separated alternative values may be specified, a match will happen if the corresponding current date/time value is present in the list.

Instead of a comma separated list of values, for a set of consecutive values an inclusive range may be specified, with the first and last values separated by a dash.

The asterisk character represents the comma separated list of all possible values for an attribute, so it's equivalent to an inclusive range between the lowest possible value and the highest possible value.

The asterisk character (all possible values) may be associated by a forward slash with a step to create a subset of all possible values. For instance, ***/20** in the minute specification is equivalent to **0,20,40**.

For a match to happen, a match on every attribute is required, so if no specific value is required for an attribute, the asterisk should be used.

Instead of the first five fields (time/data pattern), some special strings may be used, one very useful is **@reboot** which means run the task when the CRON service is started, something that happens when the server starts, but also whenever the CRON service is restarted. Nevertheless, it is a good solution to execute something on the server boot.

2.2. An example of a CRON configuration file

Here is an example for a **/etc/crontab** file content:

```
*/5 * * * *  root /usr/bin/prog1 /etc/pptt.cfg
5,35 * * * * root /root/update -g all
45 2 * 8 6   root /root/make-backup
```

Try figuring out when and at what time each of these tasks will get executed.

2.3. Schedule the periodic execution of today's class BASH script.

The hosts availability checking script, we have previously developed, may now be automatically executed from time to time.

Create a new configuration line in **/etc/crontab**, such as the hosts availability checking script is executed by the **root** user, every ten minutes, from Monday to Friday.

3. Using and managing Docker containers

The **Docker** software is one of the most popular PaaS (Platform as a Service) solutions, it uses the operating system virtualization concept to establish **containers**. A container may be seen as a closed box inside which programs can be run safely without interfering with the exterior world, namely the host operating system and other applications and services running on it.

Containers are useful in many ways, for instance when testing applications under development, it is rather simple to create a set of containers to simulate the production environment where the applications will run later.

As you know, the virtual server each student has created for the SCOMRED lab classes is itself a docker container, even though is looks much like a normal Linux Server running several applications and services. In fact, one of such services is the docker service, as you may check:

```
systemctl status docker
```

So yes, you can run docker containers inside your virtual server (itself a Docker container).

Some important concepts about containers and Docker:

Containers are created from images; the main content of an image is the filesystem. The operating system kernel is not in the image because containers use operating system virtualization, this means applications running inside the container will use the operating system kernel that is hosting the container.

For Docker, many images are available at the internet, for instance at **Docker Hub** (https://hub.docker.com/), they are fetched on demand and stored locally so that they become available to create and run containers locally.

Running (starting) a Docker container is in fact running a specific application inside the container, it must be an executable file somewhere in the filesystem inside the container, **when that application exits, the container stops running**. The application to be run is defined in the image, however that may be override, and another application inside the container may be run instead.

Containers may be ephemeral, an ephemeral container is created with the purpose of running only once, when it stops it's immediately destroyed.

3.1. Let's now test these concepts with the docker command.

3.1.1. Listing locally available Docker images:

```
docker image ls
```

Most likely there is none.

You can get help about a docker command by using the **--help** option, for instance:

```
docker image --help
```

3.1.2. Listing local Docker containers:

```
docker ps -a
```

Again, most likely none. The **-a** flag instructs the **docker ps** command to list stopped containers as well, by default, only running containers are listed.

3.1.3. Creating and running containers

The **docker run** command creates a new container from an image and starts it. If the requested image is not available locally, it will be downloaded from Docker Hub. Then, a container is created from it, and it is started. By default, docker containers are not ephemeral, this means they are kept after stopping.

To be identifiable, within a host each container should have a unique name, for non-ephemeral containers it's best to baptise the container when it is created by using the **--name** option.

Let's create and start a container using the **hello-world** image, this is a test image, it simply prints a hello world greeting along with some other information and stops running.

Images are referred by their name, but they also have an additional sub identifier called tag that may be used as suffix, separated by the image name by a colon. **Tags are most often used to identify different versions of the same image**.

If no tag is used to refer an image, then the **latest** tag is used by default. For instance, when we refer the **hello-world** image, the image that will be used is **hello-world:latest**.

Let's try it, the new container will be named as **myHello**, try to understand what is happening:

```
docker run --name myHello hello-world
```

Now let's list again existing images and containers:

```
docker image ls

docker ps -a
```

The **hello-world** image has been downloaded, and is being used by the **myHello** container, therefore the image can't be removed. However, even if no containers were using the image, it would not be automatically removed while there was no need to free disk space.

Now, let's create and start another container using the same image:

```
docker run --name myHello2 hello-world
```

This time there was no need to download the image. Check that both containers exist:

```
docker ps -a
```

If the purpose is running once only, we can create an ephemeral container instead, we can do that by simply adding the --**rm** option to the docker run command:

```
docker run --rm --name myHello3 hello-world
```

Check that this last container has not been preserved:

```
docker ps -a
```

One other way to create a Docker container is by using the **docker create** command it's identical to the **docker run** command, however, once created, the container is not started. The **docker create** command also supports the **--name** option, but not the **--rm** option.

3.1.4. Starting existing containers

A non-ephemeral, previously created, stopped container may be started by using the **docker start** command. Unlike with the **docker run** command that will usually execute in foreground (interactive mode), the **docker start** command will usually execute in background, to force the interactive mode, the **-i** option should be used. This container prints data to the **stdout**, and in background mode you wouldn't see anything.

Let's try starting the **myHello2** container in interactive mode:

```
docker start -i myHello2
```

3.1.5. Removing stopped containers

Non-ephemeral, stopped containers we no longer need can be removed with the **docker rm** command, let´s remove the containers created so far (**myHello** and **myHello2**):

```
docker rm myHello myHello2
```

By default, containers must be stopped before being removed, however, the **docker rm** command accepts the **-f** option that will force the running containers to stop (SIGKILL) before removing them.

3.2. Networking with Docker

In your server, every Docker container is connected to a **virtual switch** (software bridge) under control of the Docker service. This virtual switch is a **private network** within your server, IP packets are forward to your server's **eth0** interface (the outside world) by applying NAT (Network Address Translation).

Remember nodes in a private network are allowed to access the internet, however they are not addressable from the internet because they use private addresses. So, at first glance, a server (e.g., Apache) in a private network is not reachable from the outside world. To overcome this issue, most devices applying NAT allow administrators to set static port redirections from the public address to private addresses, and Docker also allows that.

When creating a Docker container (e.g., run and create commands) you may use the **-p** option to specify which port numbers at the public address should be redirected to a port number at the container, for this purpose you will use **-p PORT1:PORT2**, where PORT1 stands for the source port number at the public interface, and **PORT2** stands for the destination port number at the container.

3.2.1. Running a network service inside a container

Let's now use the **httpd** image, it's a minimal image with the Apache 2 web server. Unlike the previously used **hello-world** image than simply prints something and then exits, a container running **a network service runs forever and never stops** unless requested to do so (**docker stop** command).

One other difference between the **hello-world** image and a typical network service is that unlike the former, **a network service runs in background** (non-interactive mode).

We are now going to use the **docker run** command to create an ephemeral container (**--rm**), named as **myApp**, and run it in background (**-d** option, standing for detach STDIN, STDOUT, and STDERR).

The Apache server running inside this container serves HTTP clients at port number 80, to make it accessible to the outside world we will redirect port number **8080** of your server to port number **80** of the container (**-p 8080:80**).

Here is the full command-line:

```
docker run --rm -d --name myApp -p 8080:80 httpd
```

Check that the container is running:

```
docker ps -a
```

Now let's test if the Apache server running inside the container is accessible to the outside world. At your laptop, **with the DEI VPN connected**, and using your favourite web browser, type:

```
http://vsXXX.dei.isep.ipp.pt:8080
```

Replacing **vsXXX** with the DNS name of your server.

If everything went ok, you are expected to get something like:

It works!

This is the default web page (index.html) of the Apache web server running inside the **myApp** container.

3.2.2. Executing commands inside a running container

While a container is running you can execute commands inside it by using the **docker exec** command. Though remember the single purpose of this specific image is running the Apache server, so many commands will be missing.

Let's try running some commands inside your **myApp** container:

```
docker exec myApp pwd

docker exec myApp ls -l

docket exec myApp ls -l htdocs

docker exec myApp cat /usr/local/apache2/htdocs/index.html
```

Right, here is the default page presented before at your web browser.

Which Linux distribution is being used inside your **myApp** container?

Many distributions have that information in the text file **/etc/issue.net**, with the purpose of being shown at the terminal before the user logins.

So, let's check, first your server, and then your **myApp** container:

```
cat /etc/issue.net

docker exec myApp cat /etc/issue.net
```

We can also **start a shell session inside the container**, but again, remember that your **myApp** container is not a full distribution, it contains only what is required for the Apache server to run.

To be able to properly interact with the shell inside the container, two options of the **docker exec** command should be used, **-t** to create a terminal environment, and **-i** to interact through STDIN.

Presuming the **bash** is available inside this image, let's try:

```
docker exec -t -i myApp bash
```

You are now in a command-line session inside your container, try some basic bash commands.

To exit you can use the **exit** command and you will be back on your server's command line.

3.3. Using volumes with Docker containers

While we were at the command line inside the container, we could have changed the Apache pages contents, and other configurations, in fact, because there's no text editor inside this image, we would also have to install one.

However, there is a big catch, this is an ephemeral container, so when it stops is immediately destroyed, and all those changes and date stored inside the container would be lost.

Stop your container and check that it no longer exists:

```
docker stop myApp

docker ps -a
```

Does this mean it's pointless to change the configuration or store data inside an ephemeral container?

No, the most popular workaround to this issue is volumes, volumes are folders and files at the host system (outside the container) that are mounted inside the container replacing the ones existing in the image being used.

The **-v** option, available with the **docker run** and **docker create** commands, establishes objects, most often folders, to be mounted. The basic syntax is **-v OUTSIDE-FOLDER:INSIDE-FOLDER**.

From the previous exploration to this image, we now know that the web documents being served by Apache are stored at the **/usr/local/apache2/htdocs** folder inside the container. Therefore, if we want to continue using an ephemeral container and at the same time being able to manage and preserve the contents being served, that folder must be a mounted volume.

Let's put this idea into practice, first we will create a folder to store the web files, we will also create an index.html file in this folder. Later when the container is created, we will use the **-v** option to replace the **/usr/local/apache2/htdocs** folder inside the container by this external folder.

```
mkdir /root/my-web

nano /root/my-web/index.html
```

Add some content to the **index.html** file, for instance:

```
<html><body bgColor=gray>
<h1>This web page is on a volume</h1>
</body></html>
```

Now we can create and start the ephemeral container as before, but now replacing the **/usr/local/apache2/htdocs** folder inside the container, mind that this is a single command-line:

```
docker run --rm -d --name myApp -p 8080:80 -v
/root/my-web:/usr/local/apache2/htdocs httpd
```

Test the server. At your laptop, **with the DEI VPN connected**, and using your web browser, type:

`http://vsXXX.dei.isep.ipp.pt:8080`

Replacing **vsXXX** with the DNS name of your server. The page we have just created should now be shown.

Notice that you can now manage the documents being served by the container, from outside the container. It will have the same result as managing them from inside the container.

Edit the **/root/my-web/index.html** file, for instance:

```
<html><body bgColor=gray>
<h1>This web page is on a volume</h1>
<h2>Changed from outside the container</h2>
</body></html>
```

Reload the page at your web browser.

```
http://vsXXX.dei.isep.ipp.pt:8080
```

Stop the container:

```
docker stop myApp

docker ps -a
```

It no longer exists, but the content of the **/usr/local/apache2/htdocs** folder is not lost.

Create and start the container again:

```
docker run --rm -d --name myApp -p 8080:80 -v
/root/my-web:/usr/local/apache2/htdocs httpd
```