Cloud Computing Fall 2019
Assignment 1: Building an Automatically Scaling Web Application

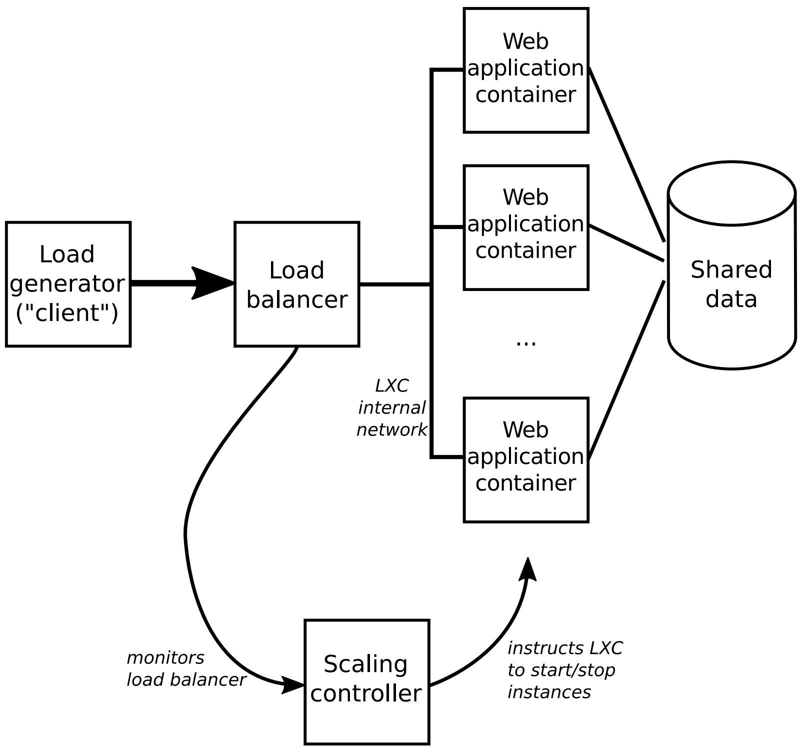
Deadline: Monday, November 25, 2019

1 Aim and Scope

In this assignment we will build a small automatically scaling testbed for a (very) trivial Web application. The testbed should be built on top of a minimal "IaaS environment", namely Linux Containers (LXC). The goal of the assignment is to become familiar with all facets of scaling Web applications, which will increase your understanding of the low-level / fundamental implementation details of Cloud systems.

We will keep things simple and constrain ourselves to a single host on which containers can be started. In order to simulate saturated servers, the Web application will be rate limited on purpose. The model that we use can be easily extended to a distributed system consisting of multiple hosts at the expensive of a larger time investment, which is why we choose not to do this[1].

As has been discussed during the lecture, a number of components are required to create an automatically scaling Web application. These components are summarized in the following figure:


Web application container
Web application container
...
Web application container
Load generator ("client")
Load balancer
Shared data
LXC internal network
monitors load balancer
Scaling controller
instructs LXC to start/stop instances

Let us now discuss these components one by one:

1. As an actual Web application to scale you will develop a small and trivial object store API. This application will be rate limited on purpose. You may develop this application in a programming language of choice.

[1]One of the (tricky) things that needs to be arranged, for instance, is to extend the virtual network on which the instances are hosted to cover multiple nodes.

2. The "hypervisor" technology that you will be using is Linux Containers (LXC). This technology has been selected because it is easier to work with than a fully fledged virtual machine hypervisor and requires (significantly) less memory and disk space for hosting multiple instances of the developed Web application. LXC also automatically handles the virtual network. Additionally, LXC provides an easy way to store state outside of the containers in order to make our containers hosting the Web application stateless — important to simplify scaling!

   Note that within this assignment you are required to use LXC using the low-level commands and API. You are *not allowed* to use e.g. LXD or Docker.

3. In order to distribute the load over multiple instances and to provide clients with a single entry point, a load balancer is needed. You are allowed to write your own (simple) load balancer, but are also free to pick an existing one. We suggest you use HAProxy.

4. To perform the actual automatic scaling, a scaling controller needs to be developed. Within this assignment you must write your own scaling controller. The controller needs to monitor the state of the system. If you chose to use HAProxy, you can monitor the haproxy daemon which has an option to provide you with statistics. Based on this information scaling decisions can be made, for example using a rule-based method. Subsequently, the scaling decision (to create or to stop instances) is carried out by executing LXC commands. The scaling controller may be developed in a programming language of choice. However, note that LXC provides a C and Python API. Therefore, Python seems a reasonable choice.

5. To be able to test the automatic scaling capabilities of the system a client, or rather HTTP request generator, is required. You are allowed to either write your own request generator or use an existing one. Existing projects that you can use include jMeter (Java based) or Locust (Python based and can operate as command-line utility).

Your setup needs to be well documented and the architecture must be described in a report that accompanies your submission. Finally, at least two meaningful experiments must be conducted that demonstrate the effectiveness of the automatic scaling. These experiments *must* investigate the response of the system to an increasing and decreasing number of client requests as to demonstrate that the system will automatically scale up and down. Investigate the response time of the scaling system and try to improve this to show that the number of failed requests can be minimized or (preferably) fully eliminated. Things to experiment with include changing the rules for scaling decisions used by the scaling controller, optimizing the method or timing of spawning new containers (such that they can be spawned in less time), reducing oscillation, etc.

2 Development Environment

In order to complete this assignment you need administrative (root) access to a Linux installation. This could be your own laptop or workstation, however we *strongly* recommend to create a dedicated virtual machine for this assignment. By doing so, you avoid the trouble of accidentally introducing problems in your regular OS installation (which you may need for other courses). The entire setup that was described in the previous section can be built within a single virtual machine. Run the virtual machine on your Linux laptop or workstation using qemu/KVM, or if you are using a different host OS, use a suitable virtualization product (VMWare, Virtual Box, Parallels, etc.). The required disk space for the virtual machine depends on the OS you want to use within the containers (see also the Appendix). In the case of the small Alpine OS, 4 to 5 GB should be enough. However, if you want to use Ubuntu or CentOS as OS within the container rootfs, create a VM with a disk of at least 10 GB.

If you want to work on the University workstations you must create a virtual machine (as you do not have root access on these machines). Virtual machines can be run on the University workstations, but as KVM may not be enabled on Linux you might encounter decreased performance. The performance is most likely still sufficient however.

You are free to make your own choice for a Linux distribution to use within this virtual machine. The lecturer did a prototype implementation on Debian 10 and all recommendations described in the Appendix work on Debian 10. Your mileage may vary on other distributions.

3 Web Application API

As Web application we ask you to implement a trivial object store API. The straightforward way to implement this is by designing a RESTful API. Objects are stored by name (key or object ID) in a directory on the file system. This object ID is unique. You do not have to implement support for buckets, having one global pool of objects is good enough.

In order to make the containers that host the Web application stateless, an external directory will be mounted within the containers (so all containers have access to the same external directory and thus to the same collection of objects). To keep things simple you do not have to consider problems arising from concurrent access to the objects such as race conditions in this assignment (we will leave this for another course).

The following is an example of a suitable API:

`GET /`	Return a list of object IDs.
`DELETE /`	Delete all objects.
`GET /objs/<obj_id>`	Return content of object with ID `<obj_id>` or 404 if object does not exist.
`PUT /objs/<obj_id>`	Store provided content within object with ID `<obj_id>`. Creates a new object if an object with this ID does not yet exist, otherwise it overwrites the existing object.
`DELETE /objs/<obj_id>`	Delete the object with specified ID. Returns 404 if the specified object does not exist.

Once the application is working, do not forget to rate limit it on purpose to facilitate the implementation and testing of the scaling functionality.

This Web application may be implemented in a programming language of choice. The lecturer did a quick prototype in Python using the Flask web framework. Within this framework extension modules exist that make it trivial to implement a RESTful API and rate limiting. See the Tips & Tricks appendix for details.

4 Submission Details

Teams may be formed that consist of at most *two* persons. Assignments must be submitted through BlackBoard. Note that an individual submission from each class participant is expected, also if you have completed the assignment in a team. In the case of teams, a mandatory individual questionnaire to evaluate the team work must be completed upon assignment submission.

Ensure that all files that are submitted include names and student IDs. In case your submission is larger than 10 MB in size, please use SURFfilesender or SurfDrive and include download links in the submission text box.

Deadline: Monday, November 25, 2019.

As with all other course work, keep assignment solutions to yourself. Do not post the code on public Git or code snippet repositories where it can be found by other students. All code and reports that are submitted may be subjected to automated plagiarism checks.

The following needs to be submitted:

Web application	Source code
Hypervisor	• LXC Configuration files • Rootfs for the web application container and/or instructions to generate this rootfs.
Load balancer	• Configuration file • If run within a container, rootfs and/or instructions to generate this rootfs.
Scaling controller	Source code.
Request generator	• Source code (in case you write our own) • Configuration files
Report	Report in PDF format (NO WORD FILES!!!).

The maximum grade that can be obtained is 10. The grade is the sum of the scores for the following components, maximum score between square brackets:

- [**6 out of 10**] Completeness and functionality of the submission.
- [**2 out of 10**] Quality of the content and layout of the report. The report must include a brief explanation of the implemented architecture, choices made during implementation and the conducted experiments (assessed separately).
- [**2 out of 10**] Quality and depth of the conducted experiments.

A Tips & Tricks

This section contains a number of tips, tricks and interesting tidbits to save you time. It is based on the quick prototype implementation done by the lecturer. As was noted earlier, the lecturer did a prototype implementation on Debian 10, so the tips are based on Debian 10 and details for other Linux distributions may vary. Do not feel obliged to use all of these tips; the main purpose of this section is to save you time.

A.1 Python Web application tips

Note that you are free to choose a programming language and framework for the Web application, so do not feel obliged to use the framework described here.

Flask (`https://palletsprojects.com/p/flask/`) is a popular Web framework based on the Python programming language. Extension modules exist for Flask to help with the implementation of RESTful APIs (Python package `flask-RESTful`, website with quickstart guide `https://flask-restful.readthedocs.io/en/latest/quickstart.html#a-minimal-api`) and rate limiting (Python package `flask-limiter`, website `https://flask-limiter.readthedocs.io/en/stable/`).

A.2 Working with LXC

Before you can commence working with LXC, it needs to be installed and configured:

1. Install the LXC packages, on Debian this is accomplished by running as root: `apt-get install lxc lxc-templates`.

2. Verify there is enough storage space on the partition hosting `/var/lib/lxc`. How much storage space you need depends on what Linux distribution you want to use within your containers. If this is a tiny distribution like Alpine, then 200 to 300 MB storage space should be enough. If you want to use a distribution like Debian, Ubuntu or CentOS, count on 2 to 3 GB.
3. Check the default network configuration in `/usr/lib/x86_64-linux-gnu/lxc/lxc-net`. Depending on your local configuration, you may need to choose another subnet to use.
4. To enable bridged networking, you need to execute the following command (on Debian-based systems): `echo 'USE_LXC_BRIDGE="true"' > /etc/default/lxc-net`
5. Edit the network settings in `/etc/network/default.conf` to enable networking by default for new containers:

   ```
   #lxc.net.0.type = empty
   lxc.net.0.type = veth
   lxc.net.0.link = lxcbr0
   lxc.net.0.flags = up
   lxc.net.0.hwaddr = 00:16:3e:xx:xx:xx
   ```

6. Execute `systemctl restart lxc-net`

Your system should now be ready to create containers that have network access by default.

A.2.1 Commands to set up containers

A new container can be created from scratch by installing a base OS image using an existing template. After container creation, you can attach to the container and continue configuration of the container. For the scaling web application, you need to create and configure a container image that will be copied each time to create new instances (see later on).

The templates that are available are listed in `/usr/share/lxc/templates`. In the interest of saving disk space, the lecturer decided to use Alpine Linux. This is a (very) small Linux distribution that is often used within containers.

To create a container based on Alpine Linux with the name `testcontainer`:

```
lxc-create -t alpine -n testcontainer
```

Subsequently, the container can be started:

```
lxc-start -n testcontainer
```

Attaching to the container gives you a prompt with root access, so you can make modifications to the container:

```
lxc-attach -n testcontainer
```

Or specify a command to run instead of getting a shell:

```
lxc-attach -n testcontainer -- ls /etc
```

Within the container, you might want to install additional packages (Python anyone?). The exact instructions depend on the distribution you have selected. In case you are working with Alpine, you can install Python 3 by executing the following commands:

```
apk update
apk add python3
```

You can search the package database as follows:

```
apk list '*haproxy*'
```

(yes, Alpine also has HAProxy packages in case you want to create a container for your load balancer).

A container is stopped (shut down) using `lxc-stop -n testcontainer` or can be rebooted with `lxc-stop -r -n testcontainer`. Finally, you can obtain a list of containers with
`lxc-ls --fancy`.

A.2.2 Mounting a directory within a container

For the Web application you need to ensure that all containers have access to the same directory in which the objects are stored. First make sure you have created such a directory on the host (we use `/srv/objects`). Now in your container, you want to create a mountpoint (we use `/objects`:

```
lxc-attach -n testcontainer -- mkdir /objects
```

We also need to change the configuration file of the container. For a container with the name `testcontainer` this configuration file is located at `/var/lib/lxc/testcontainer/config`. Open this file and add the following line:

```
lxc.mount.entry = /srv/objects objects none bind 0 0
```

And restart the container. The external directory should now be mounted within the container at `/objects`.

A.2.3 Starting the Web application on container startup

Do not forget to ensure that your Web application is started on container startup. The exact instructions to achieve this depends on the OS you have chosen for your container. In the case of Alpine, you need to create a script in `/etc/init.d`. The following is an example to run the Python built-in webserver which serves files from the specified directory:

```
#!/sbin/openrc-run

command="python3"
command_args="-m http.server 8000 --directory /tmp"
pidfile="/run/$RC_SVCNAME.pid"
command_background="yes"

depend()
{
        need localmount
}
```

Assume the above file is stored as `/etc/init.d/test`. In order to ensure this `test` service is started on system startup, the following command needs to be executed (once): `rc-update add test default`.

A.2.4 Copying containers

To "scale up" the Web application, you need to create additional instances of the application container. This can be achieved using the `lxc-copy` command. By default, this command will create a full copy of the container and copy the entire rootfs. This can be quite time consuming.

The container for your Web application is supposed to be stateless. This implies that when the container is no longer necessary (scale down), it can be safely deleted as no valuable data is stored within the container. LXC supports *ephemeral* containers for this purpose and these containers are automatically deleted upon container shutdown. Furthermore, as the container is stateless a separate copy is not necessary. A copy-on-write container is sufficient. LXC also supports this using `overlay` as backing store.

In order to use ephemeral copy-on-write containers, the source container that serves as template (in our case `testcontainer`) ***must be stopped***! Then, containers can be instantiated with the following command:

```
lxc-copy -e -B overlay -n testcontainer
```

A.2.5 LXC Python API

LXC can also be controlled via its C and Python API. In this section, we give a number of examples of the Python API. Before you can use this API, the module needs to be imported with `import lxc`.

Obtain a list of names of defined containers:

```
containers = lxc.list_containers()
```

Get a handle on a container and if it is running print the IP addresses of this container:

```
c = lxc.Container("testcontainer")
if c.running:
    print(c.get_ips())
```

Containers can be stopped with the `.stop()` method. The method `.wait('RUNNING')` waits (blocking) until a container is running. The `.get_ips()` method also has a timeout argument that waits the given number of seconds until the IP addresses are available (useful if the container was just started, as the network interface is not brought up instantly).

To run a command within a container (`stdout` and `stdin` arguments can be used to redirect output):

```
c = lxc.Container("testcontainer")
if c.running:
    c.attach_wait(lxc.attach_run_command, ["ls", "/etc"])
```

A final **important** note: ephemeral copy-on-write containers cannot be easily created using the LXC API. So, to achieve this simply run the `lxc-copy` shell command. Within Python this can be achieved using `os.system` but preferably using `subprocess.run`.

A.3 Dealing with HAProxy

You may write your own load balancer or use an existing one such as HAProxy. As you may have read above, Alpine does provide HAProxy packages. Therefore, it is relatively easy to set up a container in which you can run the load balancer.

HAProxy allows you to read out statistics via HTTP. To enable this, you need to add a section such as the following to the configuration file:

```
frontend stats
    bind *:9999
    stats enable
    stats uri /stats
    stats refresh 1s
```

After restarting HAProxy, you can connect to port 9999 to read statistics. For instance, connect to `http://10.0.3.6:9999/stats` (of course replace the IP address with the correct one). The following URL returns machine-readable CSV output, which will be useful for your scaling controller: `http://10.0.3.6:9999/stats;csv`.

A.4 Tips for the scaling controller

The scaling controller consists of two parts that can be programmed independently (make good use of your team's resources) before these are integrated. The monitoring part should monitor the load balancer and/or the container instances running the web application. It needs to retrieve the information required to make scaling decisions (should we scale up, scale down? and if so by how many instances?).

The LXC part needs to be capable of creating new container instances, stopping instances, listing all instances (and their IP addresses) and all other container management utilities you need in order to make the scaling controller work.

After instructing LXC to start or stop a container, you also need to update the configuration of the load balancer. How exactly this should be done depends on the load balancer have you chosen to use. In the case of HAProxy there is no clear runtime API to update the list of servers. The most straightforward way to achieve this is to have your scaling controller regenerate the HAProxy configuration file, send this configuration to the load balancer container and reload HAProxy. Hacky, but it works and it appears this is used in practice (!).

A.5 Testing the system

In order to test the completed system, you want to use a HTTP load generator. Options are jMeter, Locust or something you wrote yourself.

Locust is Python-based and works as a command-line utility. First, you need to write a locust file (refer to the website `https://docs.locust.io/en/stable/quickstart.html` for an example). After that you can start locust:

```
locust -f mylocustfile.py --no-web -c 10 -r 0.5
```

Where `-c` configures the number of (concurrent) clients and `-r` configures the rate in which clients are spawned. The values for the parameters that are given are just examples, you should set up your own experiment.