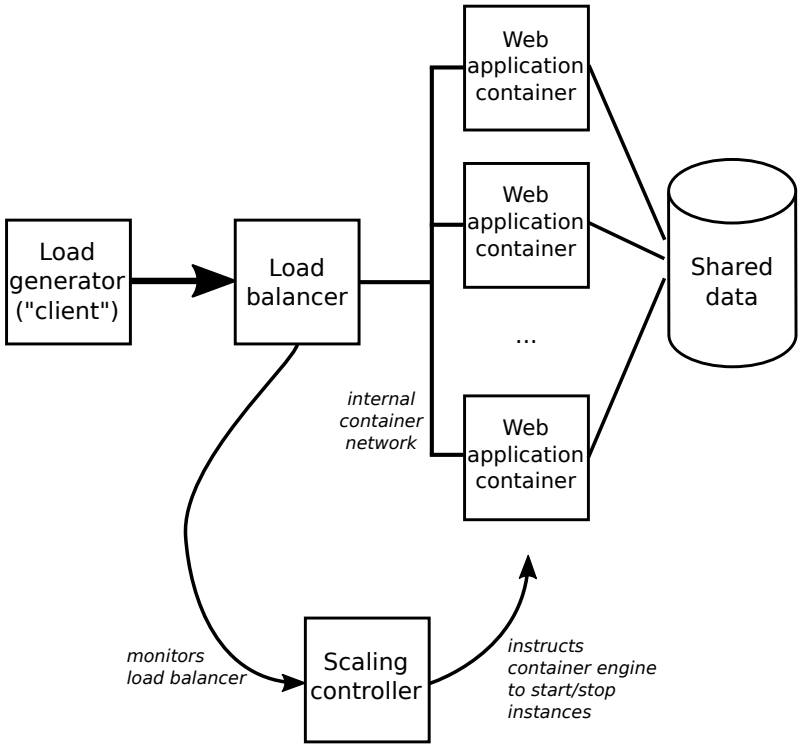
# Cloud Computing Spring 2022
## Assignment 1: Building an Automatically Scaling Web Application

**Deadline:** Friday, April 29, 2022

## 1 Aim and Scope

In this assignment we will build a small automatically scaling testbed for a (very) trivial Web application. 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. As we have discussed in class, we could deploy such a web application within a virtual machine or within containers. To keep things manageable on a single workstation, or laptop, we will constrain ourselves to a single host and make use of containers. In order to simulate saturated servers, the Web application will be single-threaded and can be rate limited on purpose. Do note that this assignment can be implemented similarly using virtual machines, and also can be easily extended to a distributed system consisting of multiple hosts. Both of these are out of scope of this assignment however.

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




Let us now discuss these components one by one:

1. As an actual Web application to scale you will be provided with a small and trivial object store API. This application is single-threaded and can be rate limited on purpose (this is configurable). Additionally, it has the capability to insert random delay in serving HTTP requests, to mimic a server under load. Both of these properties will be of use for the experimentation. The application has been written in Python.

2. We will be using *podman* as container engine. As described above, we have chosen to use containers for this assignment 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. *podman* automatically handles the virtual network and additionally provides an easy way to store state outside of the containers in order to make our container that hosts the Web application stateless — important to simplify scaling!

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 *podman* commands. The *podman* commands or API calls can be blocking, so you might want to make your scaling controller multi-threaded (or asynchronous). This way, you can continue monitoring, while the *podman* commands are being executed in a different thread.

   The scaling controller may be developed in a programming language of choice. *podman* also provides a REST API that can be used, and Python bindings to control *podman* exist. 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. We strongly recommend you to first design your experiments and then see whether an existing project can used to perform your experiments. If this is not the case, write your own request generator. Existing projects that you can consider 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. Also experiments must be included in this report. These consist of a number of preliminary "calibration" experiments:

1. Determine the saturation point of a single container. Play with rate limiting and the random delays to see what effects this has on response time (latency), request throughput, CPU utilization, etc.
2. A similar experiment, with different amounts of containers, to get an impression of how the performance would scale.
3. Determine the time required to spawn new containers.

With the information obtained through calibration you can design and tune a scaling algorithm for your scaling controller. For instance, design rules/thresholds for scaling decisions. It might be worthwhile to first set a particular target response time, so to make a Service Level Agreement, that you want to achieve also under increasing client load. Finally, *two* meaningful experiments must be conducted that demonstrate the effectiveness of the automatic scaling that you have implemented. 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 such that the SLA is adhered to. 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.). Make sure to assign multiple cores to this virtual machine, this is important for the experiments. 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. You are free to make your own choice for a Linux distribution to use within this virtual machine (but do document it in your report). The lecturer prototyped the assignment on Debian 10 / 11 based systems.

Although it is possible to run virtual machines on the University workstations, KVM is not enabled for non-root users and you might run into performance problems. We therefore recommend you to run a virtual machine on your own hardware. If you want to work together on this project in a team of two, exchange instructions on how to set up the virtual machine, such that both team members have a (mostly) equal environment.

## 3 Web Application API

As Web application you are provided with a trivial object store application based on a RESTful API. The application is written in Python using the Flask web framework. Objects are stored by name (key or object ID) in a directory on the file system. This object ID is unique. The application only works with text files (so not with binary image or PDF files), keep that in mind in case you want to experiment with large files.

To deploy the application within a container, we recommend to install Python 3 packages within the container image first. After that, create a Python `venv`[1] and install `flask_limiter` and `flask_restful` using `pip`.

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 we did not consider problems arising from concurrent access to the objects such as race conditions (we will leave this for another course).

The API implemented is as follows:

| | |
|---|---|
| `GET /`<br>`DELETE /` | Return a list of object IDs.<br>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. |
| `GET /objs/<obj_id>/checksum` | Return SHA512 checksum of object with ID `<obj_id>` or 404 if object does not exist. |

The application allows rate limiting and random delays to be configured. These customization opportunities are there to support the experiments to test the scaling functionality. For the

[1] `https://docs.python.org/3/tutorial/venv.html`

configuration of rate limiting, refer to the bottom of the file. By default random delays are introduced when serving HTTP requests, to 'emulate' a server under load. These can be configured (and also be disabled) using the variables just above the definition of the function `random_delay`.

# 4 Submission Details

Teams may be formed that consist of at most *two* persons. Assignments must be submitted through BrightSpace. For each team a single submission is expected. Ensure that all files that are submitted include names and student IDs. In case there are problems with the team work, contact the lecturer by e-mail.

**Deadline:** Friday, April 29, 2022

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. Cases of plagiarism will be reported to the Board of Examiners.

The following needs to be submitted:

| | |
|---|---|
| **Web Application** | • Commands, or Dockerfile to generate the container rootfs for the web application.<br>• Web application source, if modifications were made. |
| **Load balancer** | • Configuration file.<br>• If run within a container, commands or Docker file to generate this container. |
| **Scaling controller** | Source code. |
| **Request generator** | • Source code (in case you write our own).<br>• Configuration files. |
| **Report** | Report in PDF format (NO WORD FILES!!!), in which the following is described:<br>• Description and explanation of the implemented architecture.<br>• Choices made during implementation.<br>• Report on the 'calibration' experiments.<br>• Report on the conducted experiments to evaluate the effectiveness of your scaling controller. |

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:

- **[3.5 out of 10]** Completeness and functionality of the submission.
- **[2 out of 10]** Quality of the content and layout of the report.
- **[1 out of 10]** Calibration experiments.
- **[2.5 out of 10]** Quality and depth of the conducted experiments to evaluate the effectiveness of the scaling controller.
- **[1 out of 10]** Own initiatives and ideas that clearly surpass the assignment's requirements.

# 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 uses Debian 10 / 11 for prototyping. The 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 Working with *podman*

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

1. Install the packages, on Debian this is accomplished by running as root: `apt-get install podman`.
2. Verify there is enough storage space on the partition hosting `/var/lib/containers`. 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. Container networking should work out of the box.

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

### A.1.1 Commands to create container images

*podman* can create containers using Dockerfiles. Many use another tool called *buildah* (which actually came before *podman*) to build container images. *buildah* can be installed on Debian with a single `apt-get install buildah`. *buildah* can handle Dockerfiles, but can also build images from scratch.

A new container image can be created from scratch by starting with a base OS image. After image creation, commands can be run inside the container to continue configuration. For the scaling web application, you will create containers that are based on a particular container image (see later on).

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 initialize a new container image based on Alpine Linux:

```
container=$(buildah from alpine)
```

Subsequently, we can run commands within the container[2]:

```
buildah run $container -- 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:

```
buildah run $container -- apk update
buildah run $container -- apk add python3
```

You can search the package database as follows:

```
buildah run $container -- apk list '*haproxy*'
```

[2]On Debian 11 this does not appear to work out of the box, the following environment variable is required: `export BUILDAH_RUNTIME=/usr/bin/runc`.

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

Files can be copied into the container using the `copy` subcommand, e.g.:

```
buildah copy $container myfile.txt /root
```

Finally, it is important that the changes made are committed to a named image. In this case we will use `testcontainer` as name. After commit, the image will be visible within *podman* as well.

```
buildah commit $container testcontainer
```

Note that the same can be achieved using a Dockerfile with `RUN` and `COPY` verbs. The command to build containers from Dockerfiles is `buildah bud`.

**Additional Resources** For more information on *buildah*, refer to for instance:
`https://appfleet.com/blog/everything-you-need-to-know-about-buildah/`
`https://github.com/containers/buildah/tree/main/docs/tutorials`

### A.1.2 Starting the Web application on container startup

Do not forget to ensure that your Web application is started on container startup. This can be achieved by configuring a container command or entrypoint. For example, to start the Python built-in webserver, which serves files from the specified directory, upon container start up:

```
buildah config --cmd "" $container
buildah config --entrypoint "python3 -m http.server 8000 --directory /tmp" $container
buildah commit $container testcontainer
```

### A.1.3 Starting container instances

From images, container instances can be started. Multiple container instances can be started from a single image, and this is exactly what we need to 'scale out' the Web application. This is achieved using the *podman* command. Note that this will not create full copies of the container image, rather, the image remains read-only and an overlay file system is placed on top of it to catch any writes.

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. A special command line flag is present for this purpose, such that a container is automatically deleted upon container shutdown. Using the following command, such a container can be instantiated and the entrypoint will be launched automatically:

```
podman run --rm --name mycontainer testcontainer
```

As you can see the `testcontainer` image is used to create a container named `mycontainer`. You will note that the container entrypoint process remains in the foreground. To avoid this, add the `-d` command line flag.

The currently active containers can be inspected using the command `podman ps`. Now, how can we access the Python process running on the internal container network? We need to retrieve the internal IP address. To do so, you can use the command:

```
podman container inspect mycontainer
```

This gives a lot of information about the container. Search for `IPAddress` to obtain the internal IP address. Through this IP, you should be able to access the Python webserver on port `8000`. You can also map the container's port to the host port using the `-p` option. But note that in the context of this assignment, you will only want to expose the port of the load balancer, the ports of the Web application instances remain internal!

Finally, to stop a container you can use: `podman stop`.

### A.1.4 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`). In your container image, you want to create a mountpoint, for example `/objects`. With this in place, add the following command line option to the `podman run` command to mount the host directory `/srv/objects` within the container:

```
-v /srv/objects:/objects
```

### A.1.5 *podman* APIs

*podman* can also be controlled via its RESTful API. This API is documented in detail: `https://docs.podman.io/en/v3.2.3/_static/api.html`.
While you can target this RESTful API directly, fortunately also language bindings exist for at least Python and Go. We give a small example of the Python API. Before you can use this API, the module needs to be installed with `pip3 install podman`.

Obtain a list of names of defined containers:

```
from podman import PodmanClient
client = PodmanClient(base_url="unix:///run/podman/podman.sock")

l = [c.name for c in client.containers.list()]
```

Get a handle on a container and if it is running print the IP addresses of this container[3]:

```
from podman import PodmanClient
client = PodmanClient(base_url="unix:///run/podman/podman.sock")

c = client.containers.get("testcontainer")
if c.status == 'running':
    print(c.attrs['NetworkSettings']['Networks']['podman']['IPAddress'])
```

Containers can be stopped with the `.stop()` method. The method `.wait(condition='running')` waits (blocking) until a container is running.

The Python module has extensive documentation, use the `help` command in an interactive Python shell. The lecturer could not find this documentation online.

## A.2 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`.

[3]The `podman` in the dictionary access is the name of the default container network, see the command `podman network ls`.

## A.3 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 *podman* 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. *podman* calls might be blocking by default, in which case you want to look into asynchronous calls or multi-threading in order to make your scaling controller more responsive.

After instructing *podman* 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 (or use a volume mount?) and reload HAProxy. Hacky, but it works and it appears this is used in practice (!).

## A.4 Testing the system

In order to test the completed system, you want to use a HTTP load generator. Options are writing something yourself, jMeter, Locust or something else you find to be suitable. For your experiments it is important to first design your experiments and after that decide on a load or traffic pattern generator to use. You do not want to be limited in your experiments by a previously selected load generator.

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.