Cloud Computing - Lab Assignment 1

Srishankar Sundar - 3151298

May 1, 2022

1 Overview

This report covers the implementation of a scaling controller for a sample web application. The application is deployed as a container or multiple containers using *podman*, a container engine, behind HAProxy, a reverse-proxy.

The idea behind the scaling controller is to automatically scale the application containers up and down as per usage. The entire setup is described, with each component elaborated upon. Initial experiments were run on static single and multiple-container setups (with no scaling), the results of which helped shape the rules used by the scaling controller. This is followed by an evaluation of the system as a whole, and some discussions derived over the course of the implementation.

2 Architecture

A schematic diagram depicting the architecture of the setup is given below:

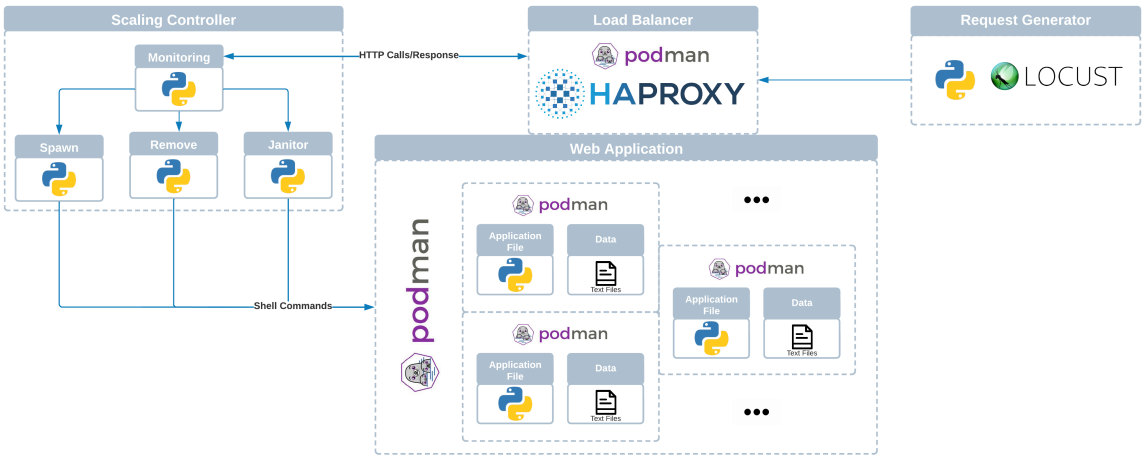


Figure 1: Solution Architecture

The components are elaborated upon in the coming sub-sections.

2.1 Application

The application is a simple object store with APIs exposed to store objects, retrieve objects and delete objects, besides the capability to retrieve a list of objects. This is implemented through a .py file that was provided. The HTTP command was used to send text data in a 'content' field, which was placed in a file titled with the alphanumeric string that the command would provide. For example, a PUT request to /objs/123 with content 'hello world' would create a file titled '123' containing the text 'hello world'.

The specifics of the containerization are given below:

- A container image was built from the .py file using *buildah*. *buildah* commands were used on a an *alpine* Linux base image, on top of which the required packages and libraries were installed, and the application file was copied in
- The data is volume mounted from a fixed directory in the host system to ensure that the data is consistent among multiple containers if spawned
- Once the container is committed as an image, *podman* is used to run the image to spawn a container. This action is eventually controlled and executed by the scaling controller
- The application listens on the container port 5000. The running containers do not expose any ports to the host machine, as they all communicate instead with the reverse proxy which is explained in the coming sub-section

2.2 Load Balancer

As explained earlier, the web application containers run on the internal 5000 port of the containers, which is not exposed to the host machine. In order to ensure that user requests can reach containers and establish load-balancing between these containers, *HAProxy* is used in tandem with the container names instead of their ports by taking advantage of the networking provided by *podman*. The specifics of containerizing the load balancer and facilitating communication are given below:

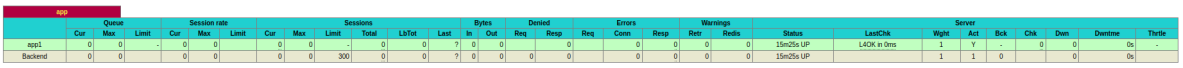
- *HAProxy* is run as a container too. In order to facilitate communication between *HAProxy* and the application containers, a new *podman* network was created. The *HAProxy* container and the subsequent application containers are all attached to this same network. The HAProxy container is specifically run with a restart policy of 'always', which enables automatic restart once the container is killed. This is integral to the solution and is elaborated upon in the Scaling Controller section.
- The HAProxy container is built using a Dockerfile as shown at HAProxy Docker Alpine. (The Dockerfile and required commands are also elaborated upon in the README file as part of the submission)
- To tell *HAProxy* that a specific container is intended as a backend that it can route traffic to, an entry is made to the configuration file of *HAProxy*. An example of one such entry is given below:

```
server  app1 mycontainer:5000 check
```

  This creates a backend on HAProxy named 'app1', which is routed to the application container named 'mycontainer' and specifically to the container port 5000

An important aspect of the load balancer that is integral to the working of the scaling controller is the fact that HAProxy exposes the statistics of its traffic on a port (9999 in the case of the current implementation). This port is published, ensuring that the scaling controller can hit the given endpoint to derive traffic statistics. This is used to make decisions with regards to spawning or removing containers.

Finally it is to be noted that the load balancer itself uses **round-robin** as the balancing scheme, which implies that the load is distributed evenly among all running containers as it comes.



app	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
app1	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	15m25s UP	L4OK in 0ms	1	Y	-	0	0	0s	-
Backend	0	0		0	0		0	0	300	0	0	?	0	0	0	0		0	0	0	0	15m25s UP		1	1	0		0	0s	

Figure 2: HAProxy Statistics for a single container backend

2.3 Scaling Controller

The scaling controller runs as a python script on the host machine, from where it communicates with podman via shell commands and consumes the load balancer's statistics.

The scaling controller can be conceptually split into four sub-components - monitoring, spawning containers, deleting containers and the janitor.

2.3.1 Monitoring

The monitoring method is invoked when the scaling controller is started. It hits the aforementioned HAProxy stats endpoint, which yields data at one given instant. Thus the script hits the endpoint at an approximately one second interval. After hitting the endpoint, the data is filtered through and specifically the total number of requests hitting the back-end (overall requests for all containers) is grabbed.

The scaling decision is made based on the **difference between the number of requests at two instances**. This is further elaborated upon in the Scaling Rule section.

When the decision is made, the monitoring part of the script calls the spawn or the remove containers methods, depending on the decision.

2.3.2 Spawn

The spawn method takes in the number of containers currently running and the number of containers to create, and iteratively carries out two steps -

- The method first runs a shell command to create a container. The naming scheme of the containers is *mycontainer<containerNumber>*. For example, if 3 containers are currently running and 1 container is to be created, the script will always create the next container as *mycontainer4*
- The method then adds a new entry to the HAProxy configuration file. The configuration file is set up such that the backend section is at the very end of the file, therefore any changes to this file simply involves appending the given line or removing one or more lines from the end. Similar to the container names being created on the fly based on the number of containers currently present, the backend app names are also created the same way (app1, app2, app3 and so on). The exact line to be added to the file is given in the Load Balancer section.

These two steps are repeated for each container to be created. Once this is completed, the load balancer container is killed, and since the restart policy of the container is set to 'always' and the configuration file is volume mounted, it restarts with the new configuration. It is to be noted that the actual action of **spawning and removing containers are prevented if the load balancer has just been reloaded the previous instant**. This is discussed further in the Limitations section.

2.3.3 Remove

The remove method takes in the number of containers to be removed, and similar to the spawn method carries out the following two steps

- The required number of lines (number of containers to be removed) is removed from the HAProxy configuration file, after which the HAProxy container is killed and subsequently restarts. As each line is removed, the method grabs the container name from each removed line and stores them in a list
- The method iteratively runs the *podman kill* command on the list of containers created in the previous step, thus removing the containers

2.3.4 Janitor

The janitor is called if there are more than 20 consecutive instances where no requests are encountered. In this case, the entire system is essentially 'wiped clean'. All containers except one are removed and the load balancer is restarted with the initial configuration. The reason for implementation is elaborated upon in the Limitations section.

2.4 Request Generator

The request generator consists of a *locustfile* where the requests are defined, and a shell script that triggers this file to start Locust with various configurations, thus yielding an overall pattern.

2.4.1 Requests

For each request, Locust is made to pick with equal probability between two possible requests.

- A **PUT** request that adds a file with a randomized name and text content *'spawn attack2!'*
- A **GET** request on the '/' endpoint that retrieves the list of files at the present moment

It was decided that only these two types of requests would be picked from to ensure ease of interpretability of the behavior of the system regardless of the length of the experiments that the system was tested on. More requests with varying processing loads would mean that it is possible that one run could enjoy lower loads and another could suffer higher loads, so this was eliminated, as the focus could then be on how many (more uniform) requests the containers could serve.

2.4.2 Pattern

Locust takes in two main arguments - the maximum number of concurrent users at any given moment and the ramp-up rate of said users. The pattern used for the evaluation for the controller is the following - (1,1), (3,1), (5,1), (3,1), (1,1), where each pair represents the maximum number of users and the ramp up rate.

Each configuration is run for about 20 seconds using the *timeout* command in the shell script.

The rationale behind the pattern was to study both the scaling up and scaling down behavior of the controller.

3 Calibration Experiments

Experiments were carried out to get an idea of baseline performance of containers and certain aspects of the system. These are elaborated upon below.

3.1 Saturation Point of a Single Container

With the requests described in the Request Generator sub-section, a single-container setup was first tested, without the scaling controller running in the background. The following visualizations are given by *Locust*.

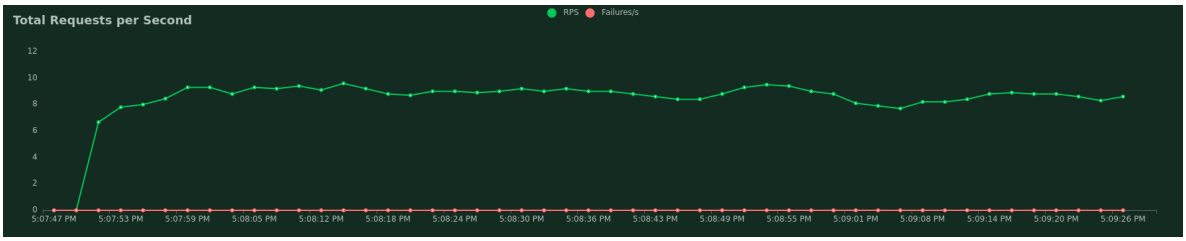


Figure 3: Failed and successful requests for a single container setup with about 8 requests per second

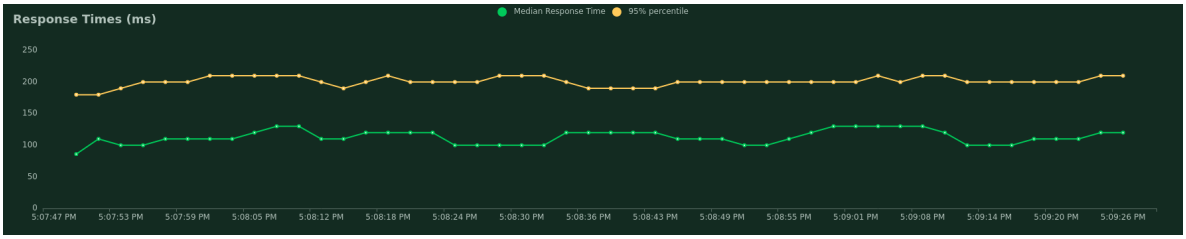


Figure 4:
Response times for a single container setup with about 8 requests per second

As the visualizations show, a single user hitting the container with about 8 requests per second is handled well by the setup, with no failures and a response time well under 250ms.

However, if this is turned up instead to about 20 requests per second, we start to see different results.

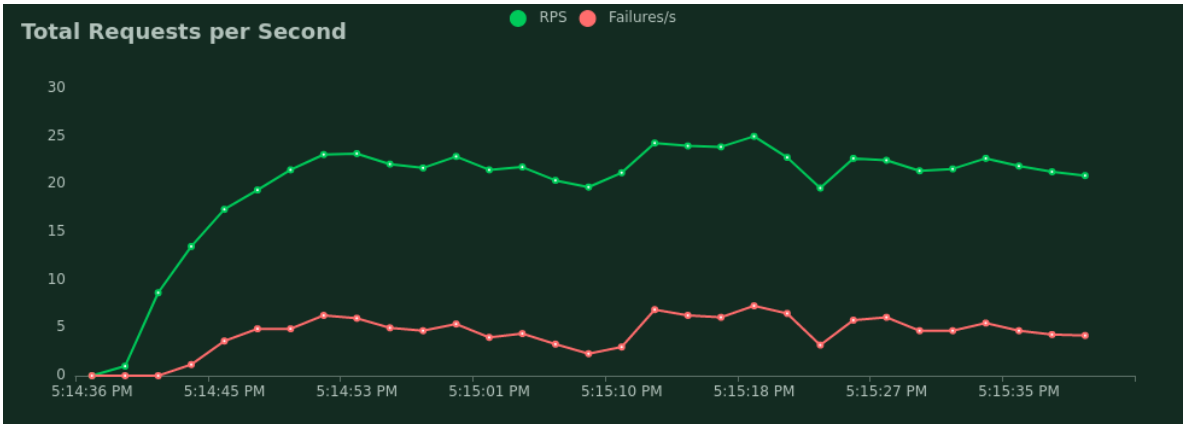


Figure 5: Failed and successful requests for a single container setup with about 20 requests per second

We start to see failed requests that spike up with moments of increased load. This specifically starts to happen at or just after 20 requests per second, so it can be ascertained that this is the saturation point of the container. It is important to note that 20 requests per second is not a generic number for the container itself and is contingent on the requests sent by *Locust*. In this case we have a PUT request with small text content and a GET request. With larger requests, it can be posited that the saturation point would be lower in terms of requests per second.

3.2 Saturation points for 2 and 3 container setups

Similar experiments were carried out for 2 and 3 container setups. For 2 and 3 containers, it was found that about 40 and 60 requests per second respectively were the saturation points.

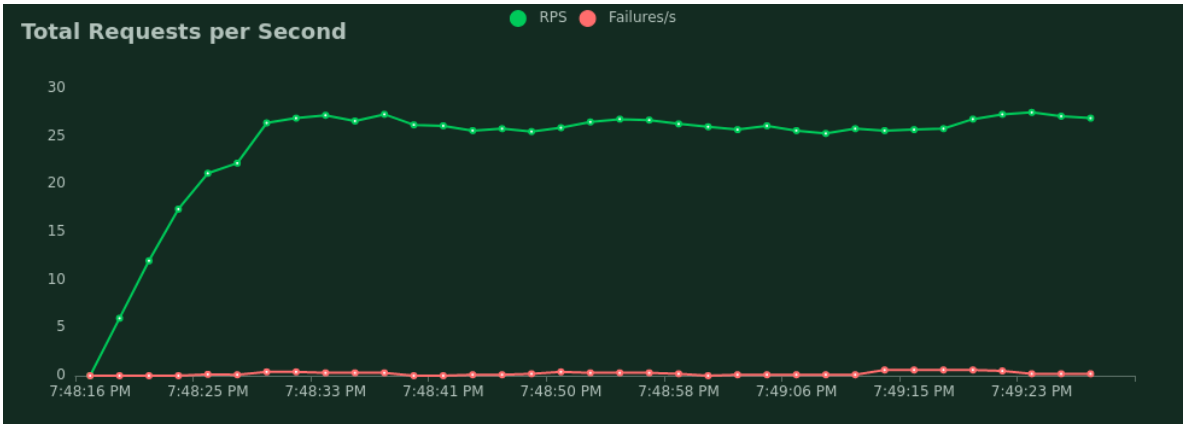


Figure 6: Failed and successful requests for a 2 container setup with about 25 to 30 requests per second

As can be seen in the above figure, a 2 container setup is easily able to handle loads that the single container setup could not.

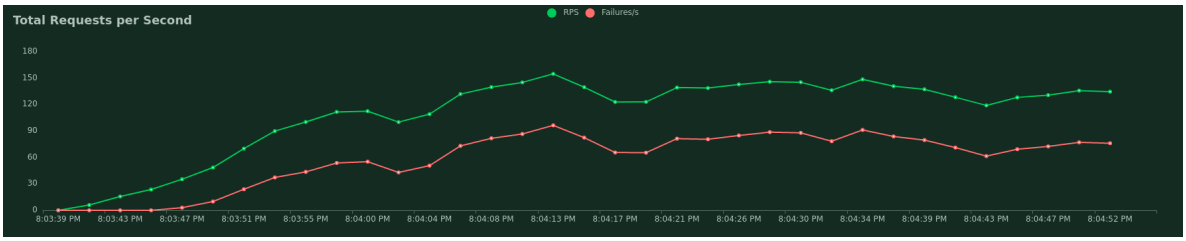


Figure 7: Failed and successful requests for a 3 container setup with about 120 requests per second

The figure above shows what saturation looks like, with 120 requests a second being too much for the 3 container setup to handle.

It is interesting to see that with 1, 2 and 3 containers, there is a linear relationship with the saturation points, i.e 20, 40 and 60 requests per second respectively. This is observed further with even 7 container setups taking about 140 to 150 requests per second before saturating. Further study could possibly be done on whether this linear relationship holds as the scale increases.

3.3 Time required to spawn and remove containers

For a single container, the script takes between 0.3 to 0.6 seconds to create and 0.2 to 0.3 seconds to remove it. The plot below illustrates the times for spawning and removing 1 to 10 containers.

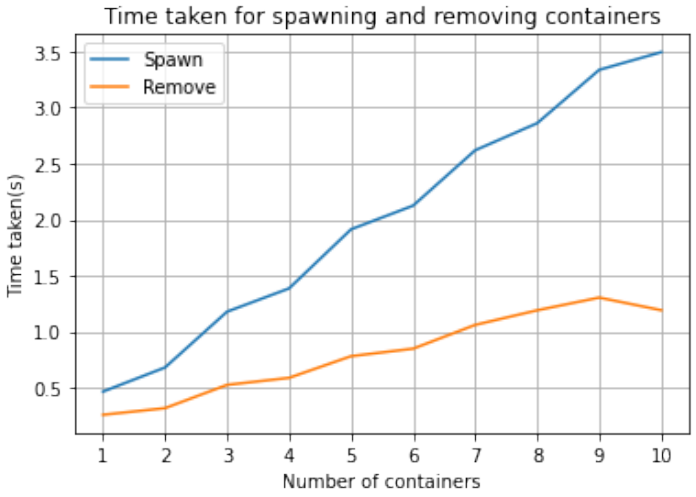


Figure 8: Execution times for spawning and removing containers

It is observed that the relationships are fairly linear so far. A possible area of further study would be as to how this relationship works for a much higher number of containers.

4 Scaling Rule

From the calibration experiments, we observe that each container is able to handle about 20 requests per second. This forms the basis of the rule used for scaling up and down. This is described in the points below:

- At every second, the monitor method calculates the number of new requests HAProxy encounters by subtracting the total requests at the previous instant from the total requests at that instant
- It then divides this number by 20 to estimate how many containers *should* exist at that moment. For example, if it encounters 46 new requests, the method expects 3 containers to serve the requests (46/20 + 1)
- It checks how many containers are currently present by checking the number of entries in the HAProxy statistics that have the name 'app'. If there are four containers present, for example, there will be five entries in the data list containing the 'app' in the name field
- It then calls the spawn method if the required number of containers is higher than the current container count, and the remove method for the opposite case
- If more than 20 seconds pass by with no requests, the janitor method is called and the system is 'wiped clean'

4.1 Remove Method

A few different aspects were experimented with for the scaling down.

4.1.1 Approach to scaling down - Remove-one vs. Remove-many

One aspect is as to whether the controller should be allowed to scale down multiple containers at once. Due to the controller working at a 1-second cadence, it seems very 'reactive' with respect to scaling up and scaling down. Being reactive in scaling up is not as much of a problem when it comes to serving requests, however being reactive when scaling down seems to negatively impact the whole system.

For example, there are sequences where there is sudden sharp drop in the number of new requests for just one or two instances, and the number of requests pick back up again. With the remove-many approach, the controller estimates how many containers are needed to serve those requests and scales

down the system harshly, reducing the number of containers from 15 to say 2, and two seconds later when the requests pick back up it has to scale up another 9 containers.

To address this, an alternate approach, i.e remove-one was tried. Here, when the controller wants to scale down at any instant, **it is only allowed to scale down one container at a time**. Thus, to scale 15 containers down to 1 container, it would take the controller a minimum of 14 seconds, during which if there is an increased demand the scaling down is naturally halted.

This yielded better performance of the system as a whole, with lesser failed requests and was thus used as the final approach. It is to be noted that the remove-one approach is indeed heavier on resources, as it does not remove containers as quickly. In cases where the constraint for developing a scaling controller is that it should be very sensitive to resources, the remove-many approach may be more appropriate, or even the core remove-one approach but with a higher number of containers being removed, for example 3 containers removed every instant instead of 1.

4.1.2 *podman kill* vs. *podman remove*

The containers are started with the *–rm* option, which makes the *kill* and the *rm* commands yield the same outcome. However, it was found that *podman rm -f* resulted in much longer execution times compared to *podman kill*, sometimes even ten times more. Thus *podman kill* was used as the command to remove containers, provided they were started with *–rm*. Further study could provide more insight into this disparity.

5 Evaluation of the Scaling Controller

The load pattern described in the Request Generator section was 'imitated' on Locust, and the results are given below -

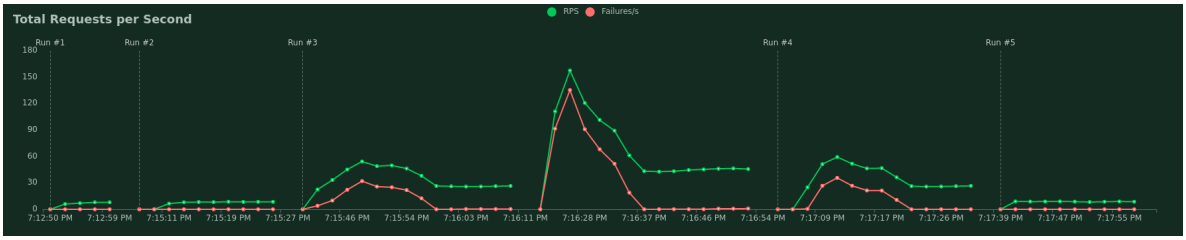


Figure 9: Failed and successful requests with the scaling controller

The curves correspond to a sequence of 1,1,3,5,3,1 maximum concurrent users respectively. We see that there is a sudden spike usually towards the beginning of each setting, which the controller tries to address but results in many failures. However, when the spike settles, the requests are usually served flawlessly, meaning the number of containers the controller has scaled to is appropriate in terms of handling the load post the spikes.

The spawning and removal of containers is illustrated through the curve below that tracks the number of containers over the course of the execution of the pattern.

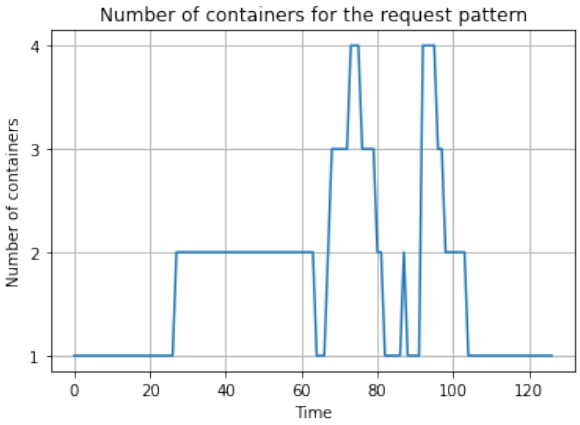


Figure 10: Number of containers during the request pattern with the scaling controller

We can see that between 20 and 60 seconds, the controller decides to keep a steady 2 containers, but between 60 to 100 seconds where the 5 and 3 concurrent users respectively are configured, the controller is quite reactive, choosing to scale containers up and down frequently. This indicates that there are quite a few spikes and drops in loads in this part of the pattern, which the controller is adapting to to some extent.

Finally, the scaling controller also maintains the same response times as initially discussed with around 200ms.

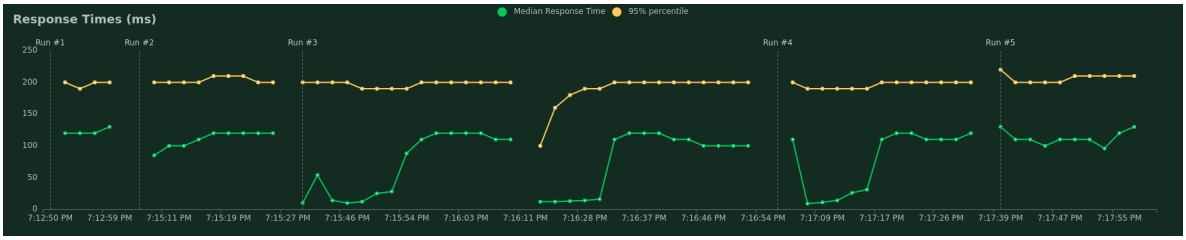


Figure 11: Response times with the scaling controller

Thus we can conclude that the scaling controller is, to some extent, effective in addressing varying loads while maintaining the same initial response times, but also fails to address harsh spikes effectively, resulting in many failed requests.

5.1 Limitations

Several limitations were observed over the course of designing and evaluating this system.

- The primary problem faced while writing the controller was as to how 'reactive' the controller should be. The spikes are not handled well as the controller is too 'slow' to react to them. Thus when the number of new requests suddenly goes from 20 to 150, the controller takes about two instances to spawn say 7 more containers, during which many of those requests overwhelm the single container that is running. However, to address this, if the cadence of querying for the scaling controller was increased, it instead became too 'reactive', with containers continuously being removed and created in cases where it would be beneficial to wait.

  One solution here could be to write separate logic to detect spikes early and handle them faster than whatever the current cadence is at that moment. The query cadence itself could be varied, with increased cadence (and subsequently increased 'reactivity') when a possible spike is detected. *podman* could also have been interacted with to gain more container statistics that could help inform decisions

- The system's scaling rules are entirely built only on a single metric. This was due to a limitation faced on the stats page of HAProxy, which seemingly **did not update the number of failed requests** at all. Ideally, the number of failed requests would be another metric to be considered while framing the rule

- The manner of reloading the HAProxy configuration is unwieldy, primarily because we lose all the statistics prior to the reload. Thus the total number of requests that the backend has encountered drops to 0. This causes some issues with the scaling controller, as it is built on grabbing the number of 'new' requests at every instant, and during the time of reload, this is possibly wrong. To counter this, the controller is updated but **prevented from taking any action right after a reload**. However, this comes with its own problems as it impacts the controller's reactivity. The ideal approach here would be to use an API if available to communicate with a load balancer rather than alter the load balancer container directly.

- The janitor should ideally not exist, i.e t**he controller should not encounter a case where containers are left 'orphaned' from the load balancer**. This seems to happen in cases where a large number of containers, i.e more than 20, are spawned and eventually before all 20 are deleted, more requests come in. This results in a few of the containers not being killed, but being removed from the HAProxy configuration file.

  Thus this is a brute-force measure that bypasses addressing a core flaw in the system and was done more in the interest of time rather than as a design decision.

6 Conclusion

The scaling controller is essentially a layer that acts above *podman* and controls it based on the traffic encountered by the load balancer. Over the course of implementation of the controller several questions were encountered, including the manner of scaling, the specifics with removing containers, different approaches to scaling down. While the results are positive to an extent, it is clear that there is room for a lot more sophistication in addressing these questions. Implementing the controller provided an inkling of the aspects that state-of-the-art tools consider. Finally, it also provided a chance to implement a small and simple system as a whole, which called for aspects of design as well, apart from the core solution to the problem of scaling.