NSA
University of Oslo
Oslo and Akershus Univeristy College of Applied Sciences
Autumn 2014

Lars Haugan, s171201

MS016A - Lab Report

Contents

Chapter 1

Dynamic cloud-based scaling web services

Keywords: Virtualization, Cloud computing, performance, scripting

Abstract

This report takes a look at the implementation of a dynamic setup in cloud-based web-services that scale with the load. With the load balancer HAProxy and implemented through OpenStack APIs.

1.1 Introduction

Problem statement

The given problem statement in this project was as follows:

Build a cloud-based web service which is able to adjust the number of webservers based on the incoming rate of user requests.

1.2 Background

Cloud computing is becoming more and more popular, and is being implemented in many parts of the industry (OpenStack, 2014c). You can rent resources from large providers like Amazon which provides a public cloud infrastructure or by building your own cloud infrastructure. This can be done with open source tools like OpenStack. There is also a third alternative that is the hybrid cloud, which enables easy transaction from using both a private cloud and public clouds.

Having these cloud infrastructures, make it possible to have services that scale over multiple locations, and to utilize the resources that are available. When having spanning clouds we can use this to our advantage and create solutions to create more sturdy solutions, that will scale web services in a more cost efficient way.

Web services with many consumers or with a high demand for calculation power will need to be able to use multiple servers to provide service to the clients. The number of servers needed is proportionate to the number of visitors and calculations needed to provide the service. If there is not enough servers to handle the load, there will a result in long response times or even loss of service.

To handle this we need a way of scaling the number of servers in a way that will give the expected result for the consumers, but at the same time use the bare minimum amount needed in able to save money.

This will make the basis for this chapter where we will look into a solution to scale a web service over multiple servers in an OpenStack environment.

1.2.1 Cloud solution with OpenStack

OpenStack is a free open source cloud software, that can be used to provide infrastructure as a service (IaaS). OpenStack proclaims to be one of the fastest growing open source communities in the world, backed by some of the biggest names in the industry like RedHat and HP (OpenStack, 2014b). It is built up of multiple services where each is responsible to handle a part of the operation in the cloud. Most notably of these are nova, which handles the instances it selves and the communication with the vitalization hypervisor. There are a total of 13 services which handles everything from identities, storage, networking, orchestration and much more.

OpenStack is a viable cloud solution due to the large scale implementation, and the large community supporting further development. One of the features provided and result of the open source software are the APIs that are available. *Python-novaclient* implementation that provides almost full integration with nova. Other implementations for the other OpenStack services are also available. Since OpenStack are implemented in Python, there has been more work on these API implementations, than what you might expect from a open source project. According to

(OpenStack, 2014a) the nova API is compatible with the implementation from Amazon (AWS). This means that it is possible to use *python-novaclient* also with AWS. This is powerful when developing tools that are supposed to work with clouds.

1.2.2 HAProxy for load balancing

HAProxy (High Availability proxy) is a free, very fast and reliable solution offering high availability, load balancing, and proxying for TCP and HTTP-based applications (HAProxy, 2014b). It is used by large sites like Reddit, Stack Overflow and Twitter (HAProxy, 2014c). Some of the features it provides in the latest version is native SSL/TLS termination, which is lacking from most other freely available load balancers, full HTTP keep-alive, IPv6 support, health checks and much more (HAProxy, 2014b). There are other free load balancers that can be used, such as apache with mod, nginx, pound and varnish. Varnish is mostly used only for caching and does not support SSL/TLS termination. HAProxy appears to be a de facto standard when it comes to open source load balancers.

These solutions creates a background for the solution to be created that can make a service scale in a cloud environment. All this so we can save money, and serve solutions that will prevail with large amounts of requests. It is also worth noting that it will not only save the provider money, but ultimately lower energy usage that could lower the environmental impact of a service.

1.3 Approach

This study will focus on creating a application that enables scaling of a web service in a cloud environment. It will explore the possibility of web scaling by implementing the possibilities of OpenStack and HAProxy in a Python application. This will make it possible to make an application that is tightly integrated with both OpenStack and HAProxy and make good use of the tight integration.

1.3.1 Setup

To be able to develop a application for this study, there is a need for a setup that will supported the needed features. This is provided with the usage of OpenStack and HAProxy. The base setup will need to be based on at least 3 instances, where we have one load balancer, which should also run the Webscaler application and the HAProxy software as shown in figure 1.1. We also need at least one client to send requests to the instance, and at least one backend instance which actually holds the website.

From the application it will be possible to connect with both the load balancer and OpenStack,

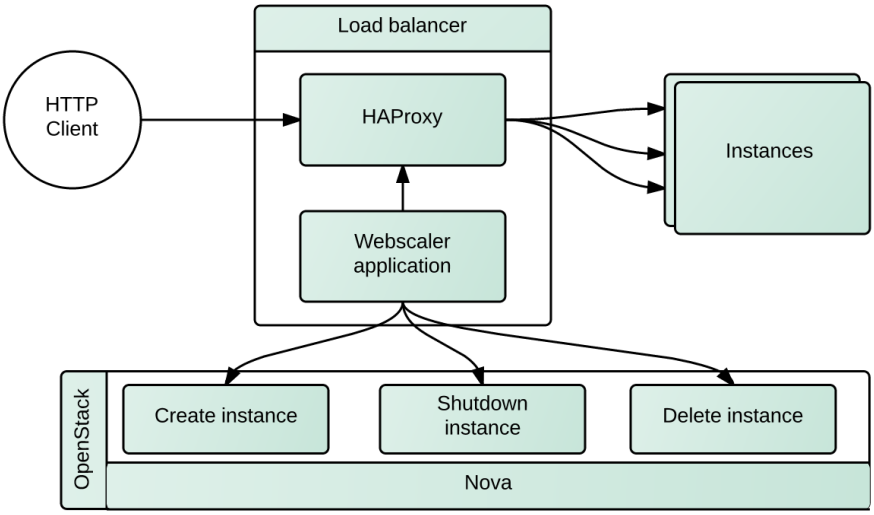

Figure 1.1: Overview of the setup

and one requirement for this is that the application must run at the load balancer instance.

1.3.2 OpenStack integration

OpenStack provides excellent possibilities for integration when programming in Python. With the use of *python-novaclient* (OpenStack, 2014d) it is possible to do almost anything you can do with nova through either the CLI or OpenStack Horizon.

It is to be expected that the integration can provide all the needed functionality of handling the instances. This means that it is possible to handle the creation of new instances and provide the needed information so that OpenStack handles the installation of the needed software on the new servers. This can be done through the usage of cloud-data which is served with a metadata service provided by OpenStack.

With this the application can scale up completely new instances which is configured for the service within a short amount of time. When the instances are no longer needed, the integration can shutdown or delete the instance altogether.

1.3.3 HAProxy

The integration part to OpenStack is only a small part for administering the instances to be created, but a way to communicate and get information from HAProxy is needed. It is possible to get data from HAProxy by issuing commands to the socket interface. There is also a web

interface which provides a overview of the status of the proxy itself and the stats for each of the services and backends provided. The interface 1.2 lists every node and service provided, and shows the different metrics available.

web-service

	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
Frontend				11	11	-	1	1	2 000	278			16 897	121 748	0	0	0					OPEN								

nodes

	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
node-1	0	0	-	5	6		0	1	-	139	139	0s	8 479	61 994		0		0	0	0	0	22h7m UP	L7OK/200 in 1ms	1	Y	-	0	0	0s	-
node-2	0	0	-	6	6		1	1	-	139	139	0s	8 418	59 754		0		0	0	0	0	22h7m UP	L7OK/200 in 0ms	1	Y	-	0	0	0s	-
node-3	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	22h7m DOWN	* L4TOUT in 2000ms	1	Y	-	1	1	22h7m	-
Backend	0	0		11	11		1	1	200	278	278	0s	16 897	121 748	0	0		0	0	0	0	22h7m UP		2	2	0		0	0s	

Figure 1.2: HAProxy web console

Everything in this stats page is available from the command line with the use of a local socket connection to the proxy 1.1.

Listing 1.1: Getting statistics from HAProxy with sockets

```
echo "show stat" | socat /var/run/haproxy/admin.sock stdio
```

The socket interface supports many different commands which will be useful to create a integrated service. This includes setting the backends as *disable*, *enable* or *draining* which can administer which backends that should be used. *Draining* mode is specially useful as what it does is to disable new connection to the proxy, but enables the existing connections to finish. This is powerful when shutting down backends.

1.3.4 Data collection: Acquiring and analysing results

To enable the scaling there is a need to have a picture of the current state of service. This is what the load balancer can give with the use of the statistics given through the socket interface. There are many different metrics to look at when the goal is to get the rate of user requests. It is possible to get metric from the web-service, which shows the frontend of the load balancer, or the backends which can show individual metrics for each backend. The data could be total transfered bytes, or the more relevant sessions which gives both current number of sessions for the hole service, or total over time.

For this study the most relevant is the total over time for the service. This will enable the scaler application to calculate the difference over time and present it as a rate of incoming requests. This should be done to calculate the number of requests per second. Thereafter this metric can be used to calculate the needed servers, when setting a baseline-limit for connections per server per second.

The difference is then calculated with the following formula:

$$\frac{(newsessions) - (previoussessions)}{sleeptime} = Sessions/second \tag{1.1}$$

This can then base on the calculation of the needed servers:

$$\frac{ceil(lastdiff)}{serverthreshold} = neededservers \tag{1.2}$$

This means that there are multiple metrics that need to be stored during the lifetime of the scaling application.

- Timestamp
- Acumulated sessions/requests
- Difference between last acumulated
- Needed machines
- Active machines (Available in OpenStack)
- Active machines in HAProxy

From the data collected it should be possible to show how the scaling is progressing. The data should therefor be stored in a csv (comma separated) file so it is possible to import the data into graphing programs, or a spreadsheet program. Based on this it is possible to generate a picture of how the process flow is unfolding.

The data could be gathered on a rapid scale, but since it takes time for new machines to be created or started, this needs to be taken into account. How rapid the data gathering and scaling should be done is not that important, since the data will be gathered on an average between the datapoints. A timeframe of one minute is therefore chosen as it will be enough time to get a good average, but also enough time for new machines to come up.

1.3.5 Simulation for presentation

To test how the load balancer and the webbalancer application will work during load, a simulation is needed. The needed function for the simulation is the possibility to vary the amount of requests to the load balancer. In the first place the simulation will use httperf to generate load on the load balancer, but since a variation is needed, this can be implemented by running multiple httperf. This is implemented as 1.2.

Listing 1.2: Simulator of request increased and descreased

```
httperf --server balance2 --port 80 --num-conns 1200 --rate 1 & #20min
sleep 120
httperf --server balance2 --port 80 --num-conns 4800 --rate 10 & #8min
sleep 60
httperf --server balance2 --port 80 --num-conns 3600 --rate 10 & #6min
sleep 60
httperf --server balance2 --port 80 --num-conns 3000 --rate 10 & #5min
sleep 60
httperf --server balance2 --port 80 --num-conns 9600 --rate 40 & #4min
sleep 120
httperf --server balance2 --port 80 --num-conns 4800 --rate 10 &
httperf --server balance2 --port 80 --num-conns 4800 --rate 10 &
httperf --server balance2 --port 80 --num-conns 9600 --rate 40 & #4min
```

With the results presented by the application it will be possible to see a correlation between the needed scaling of servers based on the incoming http requests.

With this approach it will be possible to create a webscaler by this design that can scale up new instances in OpenStack based on the number of requests gathered from HAProxy. It is a reactive design that will scale the machines after the needed resources. The servicelevel will therefore depend on the threshold that is set for one server. This is of course a limitation to the design, and could be resolved by over provisioning or by predicting the needed level of resources. That would be more of a proactive approach, and not a reactive approach as this study intends.

1.4 Result

This study presents a implementation of the desired application, that can scale servers by the incoming rate of web requests. It is totally written in Python, and uses libraries available to integrate with OpenStack and HAProxy.

The application runs on the load balancer server, as this gives access to the local socket connection made available by HAProxy. Through this socket connection the script can communicate with the load balancer and get the current metrics. This makes the basis for the data that is needed to calculate the needed amount of web servers.

The program is built to be run continuous, meaning it will always be on. It could simply be transformed into a service, but at the time being it is running as a user started application.

1.4.1 Program flow

The program works with on a few main principles. This is to run every *60* seconds, and do the calculation to either boot up new servers, or shut down the existing once. This is done by

running a continuous loop, that every 60 seconds, gathers new performance data from the load balancer.

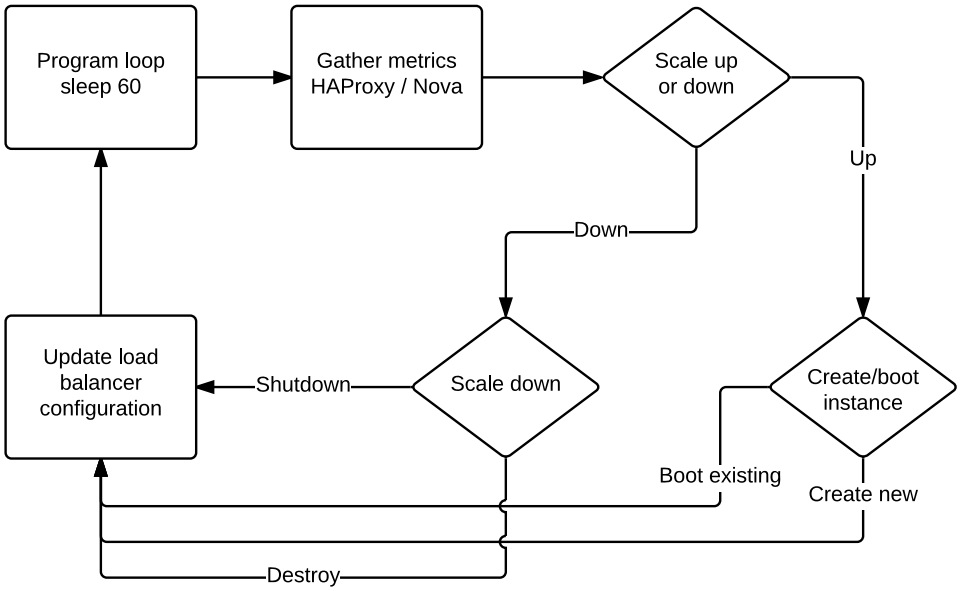

Figure 1.3: Program flow

As the program runs every minute, it will create an average of the cumulated web requests on a minute basis. As seen on the figure 1.3 this metric allows the application to define what should happen. The different possibilities are a result of the need of more or less servers. If there are more servers needed, the application will either start existing shutdown instances, or create new instances. If there is no need for servers, the instances will either be shutdown or destroyed. This ensures that the absolute minimum of needed servers are in action, but at the same time, if the need arises, an additional instance will be available faster than when creating new instances.

There are two essential parts of gathering the data. First of all, the application starts by running two rounds without doing anything. This is to be able to establish a baseline for what to do. After the baseline is established, the new metrics are calculated.

With the gathered metrics a simple calculation can be done to calculate the need servers. This is shown in the code 1.3, and is an essential part of the main Webscaler application.

Listing 1.3: Calculation of needed servers

```
1 def needed_servers(acu=None,diff=None):
2     last = metrics[-1]
3     # calculate servers for the load
4     # requests per sec / threshold
5     # ceil: rounds up
6     #needed = int(ceil(float(last['diff']) / float(server_threshold)))
```

```
7    print diff
8    needed = int(ceil(int(diff) / float(server_threshold)))
9    return needed
```

With the code 1.3 it is possible to test if there is need for more or less servers. This is tested in the main loop of the application, that executes the functions *scale_up(Needed)* or *scale_down(Needed)* according to the amount of servers that are running in comparison to what is needed.

After the instance count is handled, the load balancer will get an updated configuration with the new servers.

1.4.2 Program structure

The program structure is based on a folder structure to create a cleaner working environment. This means that all the code are separated into a folder called *bin*, and all the configuration is separated into a folder called *etc*. The main program *webscaler.py* is at the root of the application folder alongside the *requirements.txt*. The requirements file is added to ensure that all the packages needed can be installed with the use of the package manager *pip* (Python Package Index). With the combination of virtual environments this application can run anywhere where Python can run. This makes it easy to get up and running, with the following four commands:

```
$ virtualenv /path/to/new/environment
$ source /path/to/new/environment/bin/activate
$ pip install -r requirements.txt
(env)$ python webscaler.py
```

Webscaler
- **bin**
  - **haconn.py 1.14**
  - **haproxy.py 1.12**
  - **hastats.py 1.13**
  - **__init__.py**
  - **nova.py 1.11**
- **etc**
  - **clouddata.txt 1.9**
  - **haproxy.cfg 1.8**
  - **openstack.cfg**
- **README.md**
- **requirements.txt**
- **webscaler.py 1.10**

The *webscaler.py* file contains the main logic of the scaler application, see Appendix 1.10: Webscaler main application. This program does not work without the other files under the *bin* directory. These files include some central functionality for operating and communicating with the load balancer and OpenStack. The implementation of this is described in more detail here.

1.4.3 OpenStack integration

The OpenStack integration uses a module to Python called the *python-novaclient*. This is built on the same module that is possible to use from the command line to handle OpenStack nova operations. This requires a configuration file in the *etc* directory called *openstack.cfg*. This configuration is a pythonified nova api configuration, that is supplied by OpenStack that presents the AUTH endpoint and the user credentials. This configuration file is read in when the *openstack* class in the nova.py file is referenced. See 1.11: Nova.py: OpenStack integration.

This code handles all the integration with OpenStack, and provides important functions like listing the existing instances, shutdown instances, creation of new instances and operations on them.

To create a new instance there are two possible functions to use, the *create_multiple(number)* and *create_backend()*. The first is always calling the second function, but can iterate over multiple new instances, threading each new creation shown in 1.4. The name is derived from the existing instances, so that the names are human-readable in the manner of *node-1, node-2, node-3* .... This ensures easy lookup when debugging, as the nodes will always have the same order, also in the load balancer.

Listing 1.4: Nova.py: Creation of multiple instances

```
1  def create_multiple(self, number):
2      backends = self.backends()
3      namenr = self.get_instance_number(nodes=backends, next=True)
4      instances = []
5      threads = []
6      for i in range(0,number):
7          name = 'node-%s' % str(namenr + i)
8          #print "start %s" % name
9          thread = Thread(target=self.create_backend, kwargs={'nextname':name})
10         thread.start()
11         threads.append(thread)
12
13     for thread in threads:
14         thread.join()
```

A new instance is created with the function *create_backend()* 1.5. This defines all the needed parameters for the new instances to be created. One of the things that is important to handle when using OpenStack, is the time it takes after the instance has been created until the IP address is associated with the new instance. This is done by sleeping while the instance is beeing built, and after this get the new version. To prime the new instances the instance is injected with *clouddata* 1.9 which is a bash script that installs Apache and PHP.

Listing 1.5: nova.py: Function to create a backend

```
1  def create_backend(self, nextname=None):
2      """ Creates a instance in Openstack """
3      backends = self.backends()
4
5      name = ''
6      if nextname:
7          #print nextname
8          name = nextname
9      else:
10         name = 'node-%s' % str(self.get_instance_number(nodes=backends, next=True))
11
12     keypair = self.nova.keypairs.find(name='hlarshaugan')
13     image = self.nova.images.find(name='ubuntu-12.04')
```

```
14      #flavor = self.nova.flavors.find(name='m1.medium')
15      flavor = self.nova.flavors.find(name='m1.tiny')
16      net = self.nova.networks.find(label='MS016A_net')
17      nics = [{"net-id": net.id, "v4-fixed-ip": ''}]
18      f = open(path.join(self.p, 'etc/clouddata.txt'), 'r')
19
20      # try/except novaclient.exceptions.OverLimit
21      try:
22          server = self.nova.servers.create(name = name,
23                                  image = image.id,
24                                  flavor = flavor.id,
25                                  nics = nics,
26                                  key_name = keypair.name,
27                                  userdata=f)
28      except novaclient.exceptions.OverLimit:
29          print "No more available resources"
30          return None
31
32      status = server.status
33      while status == 'BUILD':
34          time.sleep(5)
35          instance = self.nova.servers.get(server.id)
36          status = instance.status
```

The integration is by itself generic, and as long as other cloud solution gives a supported API or other integration similar to OpenStack, this application is quite portable. This means that if the python-novaclient library cannot be used, the only change needed is the nova-openstack python class created in the *nova.py* file.

1.4.4 HAProxy integration

There are two ways this application works with HAProxy. This is through the socket connection that are established by the *HAconn.py* file, 1.14: haconn.py: HAProxy socket connection class, and with the generation of the configuration file haproxy.conf, 1.8:HAProxy configuration file jinja template, located under */etc/haproxy/haproxy.conf*. This uses the Python template library *Jinja* to make a template file that can be injected with the different nodes. This is done since the socket connection to HAProxy not currently supports adding or removing servers (HAProxy, 2014a).

What the socket connection is perfect for, is to gather metrics and data about the current status. It is also possible to enable, disable or drain the instances. This is important, especially the drain mode which makes it possible to ensure that no connections are lost in mid-transit. When a backend, as the webserver is called, is in draining mode, no more connections will be established to the host, but the existing will close when done. This is perfect when scaling down the amount of servers, and this is done in the *handle_scaledown* function in *Webscaler.py* 1.10.

In the *hastats.py* the statistical metrics is gathered from HAProxy socket connection, and parsed to a dictionary. This is what is being used to find the cumulated requests, that are needed to

calculate the needed backends.

1.4.5 Output of the *Webscaler*

In the approach it were specified which metrics were needed to be able to see that the dynamic scaling application performed as expected 1.3.4. These data are written every time the application closes, creating a timestamped file with the gathered data.

Listing 1.6: Data output

```
1 diffpt,haactive,active,epoch,acu,date,diff,needed
2 0,2,2,0,0,2014-09-28 15:49:58.808737,0,
```

The output is generated automatic in csv format based on the keys and values in the list *metrics* which is updated for every run. This does that it is easy to change the script to write to file for every run, and not just when the application is done. The fields describes the following:

Name	Description
diffpt	Accumulated difference since last run
haactive	Active servers in HAProxy
active	Active servers in OpenStack
epoch	Seconds since the script were initiated
acu	Accumulated requests since last load balancer restart
date	Date timestamp in ISO format
diff	Requests per second in the last time period
needed	The amount of servers needed

This is the basis of the application, how it gathers the data, and uses the data and OpenStack to decide and build the new instances. It is built on the basis that we want a minimum of servers to ensure stability, but also speed with having sleeping machines. This would be relevant in a ideal world, where you pay money for the machines you have running, and pay less for the machines that are shutdown. You are after all still using storage space.

1.5 Analysis

To ensure that the application runs as designed it has been run with the script described in 1.3.4. This load generating script is meant to illustrate a rapid growth of webrequests, so that there is

a need for more servers. The output of the webscaler is the following when this loadtest has finished 1.7.

Listing 1.7: Data output

```
diffpt,haactive,active,epoch,acu,date,diff,needed
0,2,2,0,0,2014−09−28 15:49:58.808737,0,
0,2,2,61,0,2014−09−28 15:50:59.071109,0,0
60,2,2,121,10,2014−09−28 15:51:59.330578,1,1
60,2,2,183,60,2014−09−28 15:53:01.526958,1,1
248,2,2,246,262,2014−09−28 15:54:03.836814,4,1
868,2,2,308,1113,2014−09−28 15:55:06.224331,14,2
1525,2,2,370,2593,2014−09−28 15:56:08.021714,25,3
2835,3,3,433,5411,2014−09−28 15:57:11.322713,45,5
4320,4,5,514,4266,2014−09−28 15:58:32.031523,72,8
7920,7,8,595,7867,2014−09−28 15:59:53.072102,132,14
5880,10,13,685,5861,2014−09−28 16:01:23.347892,98,10
4356,8,10,752,10210,2014−09−28 16:02:30.298435,66,7
1740,7,7,822,1684,2014−09−28 16:03:39.453189,29,3
1320,3,3,894,1262,2014−09−28 16:04:51.603086,22,3
1364,3,3,956,2570,2014−09−28 16:05:53.873466,22,3
1220,3,3,1018,3752,2014−09−28 16:06:55.722346,20,2
60,3,3,1086,60,2014−09−28 16:08:03.852704,1,1
60,3,3,1148,60,2014−09−28 16:09:06.335401,1,1
62,3,3,1211,122,2014−09−28 16:10:08.712067,1,1
124,3,3,1273,185,2014−09−28 16:11:11.296698,2,1
62,3,3,1336,223,2014−09−28 16:12:13.611447,1,1
0,3,3,1398,223,2014−09−28 16:13:16.051720,0,0
0,2,2,1462,223,2014−09−28 16:14:20.002880,0,0
0,2,2,1527,223,2014−09−28 16:15:24.452591,0,0
0,2,2,1589,0,2014−09−28 16:16:27.101334,0,0
```

The metrics that are displayed 1.7 can be presented as a graph where the rate of requests, active servers ,pending servers and the needed servers are illustrated fig: 1.4.

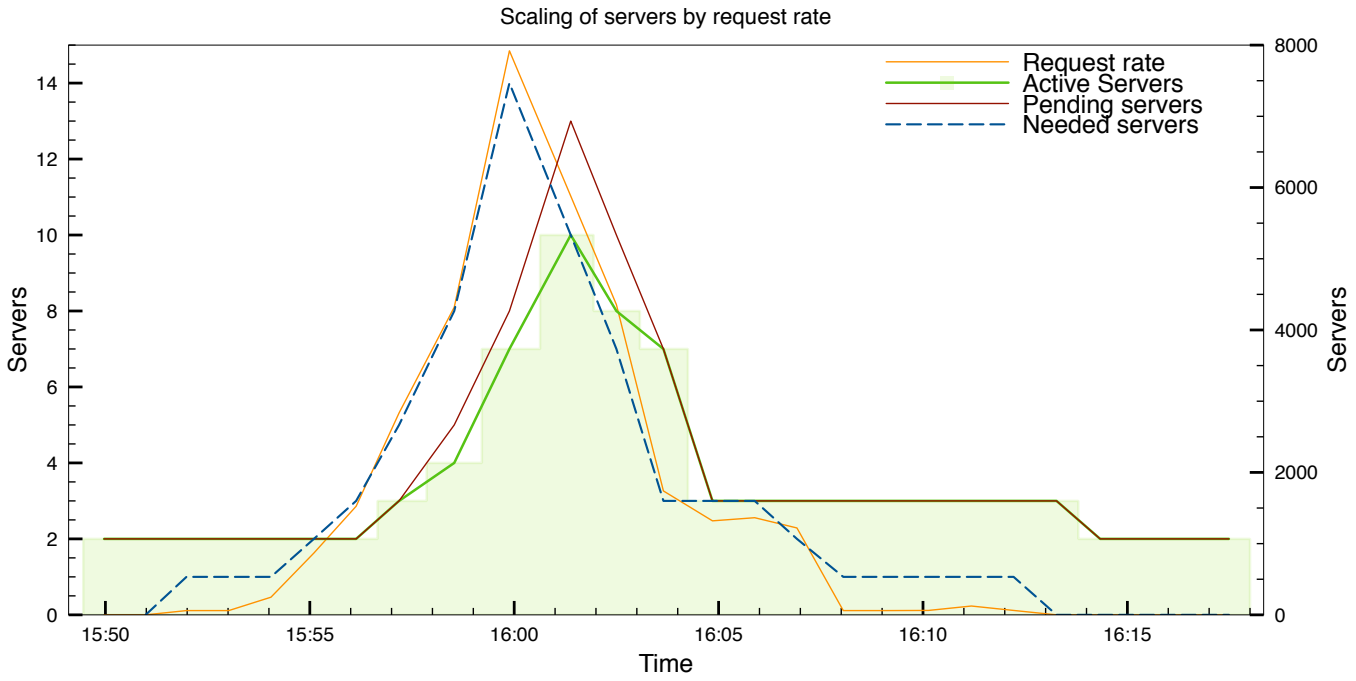

Figure 1.4: Scaling of servers

This shows that the scaling algorithm works, and that it can scale up based on the total number of requests.

1.6 Discussion and conclusion

The goal of this paper were to write an application, which could scale web servers according to the amount of requests. To solve this, a static theoretical amount of connection, were set as a soft limitation for the web servers, meaning that a single web server were designated to handle a specific amount of requests. In real life situations this could be based on the benchmarked performance of a web application. Each web application do have its own saturation point as they do different tasks, and this is therefore a highly variable factor.

The webscaler implementation is based on a reactive method. This means that the number of servers started or stopped is based on the difference of the last time period, which is by default 60 seconds. Since the time period is over 60 seconds the number of servers are based on the mean number of connections, and not the peak traffic in the period. As we can see on the graph 1.4 we see that the webscaler does handle different connection loads, but lags behind with the time it takes for a new server to spawn.

The new servers are generated on a need-at-the-time basis, meaning that new servers are generated on the fly. This enables the scaler to create servers when needed, and a faulty server would not be running for long. If the server has faulted, the healthcheck of the load balancer will fail, but the server will still be active. As the new servers are created at runtime, the generation is speed up through forking of the generation process. This means that every new server are generated simultaneous. Since the scaler uses the integration to the nova APIs, the script need to wait for a IP address to be assigned to the new servers. For this reason, the forking is perfect, as it will reduce the time for new servers to be taken to approximately the same as it would take to generate one server.

By default the webscaler will start and hold the minimum number of servers active. This means that with this configuration there will always be two servers running. This is to ensure that there is fault tolerance if one of the machines should fail.

1.6.1 Improvements

There are many improvements that can be made to both the application and the way of thinking about the scaling of web servers. The most notably improvement that can be made is through the conclusions of the requests data. This is outside the scope of this task, but a scaler script should use machine learning techniques to find the perfect number of servers it should run. This would require a lot of data to be gathered, and a calculation of expected deviation, but it is quite possible that this would result in better scaling, and more predictability. This is based on the reactive handling the scaler uses, as to the proactive handling of web server scaling a machine learning or even a prefixed baseline scaling would have.

This application only handles the mean number of connections over the last minute. It does not take into account a longer period of time, or the peaks experienced through the time period. Through different measuring modes, flapping of servers, where they are created and deleted, would be limited.

1.6.2 Conclusion

Though there are improvements, the application works well in regards to scaling servers based on the incoming load. Through the integration to both the HAProxy load balancer and OpenStack it is able to communicate well with the different tools needed to create a good and effective scaling solution. The time it takes for new machines to be added is the time OpenStack uses to create them.

With this it can be concluded that the webscaler does fulfill the specifications of the problem statement.

1.7 Appendix

Listing 1.8: HAProxy configuration file jinja template

```
global
    log /dev/log  local0
    log /dev/log  local1 notice
    chroot /var/lib/haproxy
    stats socket /run/haproxy/admin.sock mode 660 level admin
    stats timeout 30s
    user haproxy
    group haproxy
    daemon

    # Default SSL material locations
    ca-base /etc/ssl/certs
    crt-base /etc/ssl/private

    # Default ciphers to use on SSL-enabled listening sockets.
    # For more information, see ciphers(1SSL).
    ssl-default-bind-ciphers kEECDH+aRSA+AES:kRSA+AES:+AES256:RC4-SHA:!kEDH:!LOW:!EXP:!MD5:!aNULL:!eNULL

defaults
    log global
    mode  http
    option  httplog
    option  dontlognull
    timeout connect 4s
    timeout client 1m
    timeout server 1m
    timeout queue 1m
    timeout http-request 5s
    errorfile 400 /etc/haproxy/errors/400.http
    errorfile 403 /etc/haproxy/errors/403.http
    errorfile 408 /etc/haproxy/errors/408.http
    errorfile 500 /etc/haproxy/errors/500.http
    errorfile 502 /etc/haproxy/errors/502.http
    errorfile 503 /etc/haproxy/errors/503.http
    errorfile 504 /etc/haproxy/errors/504.http

listen stats :2000
    mode http
    stats enable
    stats hide-version
    stats realm Haproxy\ Statistics
    stats uri /

frontend web-service
    bind *:80
    default_backend nodes

backend nodes
{%- for node in nodes %}
    server {{node.name}} {{node.ip}}:80 id {{node.id}} check
{%- endfor %}
    http-check expect string It\ works!
    option httpchk GET /
```

Listing 1.9: Cloud data injected in instnace

```
#!/bin/bash
apt-get update --fix-missing
apt-get install apache2 libapache2-mod-php5 -y
```

Listing 1.10: Webscaler main application

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

# Python program for scaling webservices with HAproxy
from bin import haproxy, hastats
```

```
6  from bin.nova import openstack
7  from math import ceil
8  from threading import Thread, Lock
9  import time
10 import datetime
11 import csv
12
13 import sys, traceback
14
15 sleeptime = 60
16 min_backends = 2
17 server_threshold = 10
18 current_backends = 0
19 pending_backends = []
20 quota_limit = 13
21 ha_reloaded = False
22 ha_last_reload = None
23 epoch_start = datetime.datetime.now()
24
25 metrics = []
26 actions = {}
27
28 def write_data():
29     with open('%s-webscaler' % (datetime.datetime.now().strftime("%Y-%m-%dT%H%M")), 'wb') as f:
30         w = csv.DictWriter(f, metrics[0].keys)
31         header = metrics[0].keys()
32         #header = sorted([k for k, v in metrics[0].items()])
33         csv_data = [header]
34         for d in metrics:
35             csv_data.append([d[h] for h in header])
36         w.writer.writerows(csv_data)
37
38
39 def scale_up(Number=1):
40     stack = openstack()
41     sleeping = stack.sleeping_machine()
42
43     if len(stack.backends()) == (quota_limit):
44         return False
45
46     scaled = 0
47     if sleeping:
48         print sleeping
49         if len(sleeping) > 1:
50             for node in sleeping:
51                 if scaled < Number: # and not 'powering-on' in node.state:
52                     node.start()
53                     scaled += 1
54         else:
55             if not 'ACTIVE' in sleeping[0].status:
56                 sleeping[0].start()
57                 scaled += 1
58
59     else:
60         if scaled < Number:
61             thread = Thread(target=stack.create_multiple(Number-scaled))
62             thread.start()
63
64 def scale_down(Needed=1):
65     ha = haproxy.HAproxy()
66     stack = openstack()
67     active = stack.active_backends()
68     passive = stack.passive_backends()
69     instance = None
70
71     print "active servs: %s, min bakends: %s, needed: %s" % (str(len(active)), str(min_backends), str(Needed))
72     toremove = len(active) - Needed
73     removed = 0
74     threads = []
75
76     if len(passive) > 1:
77         # Delete the stopped nodes, and leave one
78         for node in passive[1:][::-1]:
79             handle_scaledown(node, delete=True)
80
81     if toremove > 0:
```

```
82          print "Want to remove %s nodes" % str(toremove)
83          if len(active) <= min_backends:
84              return False
85          elif (len(active) - toremove) > min_backends:
86              for i in range(1, toremove + 1):
87                  if 'ACTIVE' in active[-i].status:
88                      handle_scaledown(active[-i], stop=True)
89          else:
90              recalculate = len(active) - (Needed + min_backends)
91              for i in range(1, recalculate+1):
92                  if 'ACTIVE' in active[-i].status:
93                      handle_scaledown(active[-i], stop=True)
94
95      else:
96          print "No nodes to stop/delete"
97          return False
98      return True
99
100 def handle_scaledown(instance, delete=False, stop=False):
101     print "Starting to handle scaledown of %s" % instance.name
102     ha = haproxy.HAproxy()
103     stack = openstack()
104     # Set the instance in draining mode.
105     # No new conns. Active finishes
106     ha.drain(instance)
107     # Operating with short draintime (only static page)
108     time.sleep(1)
109     try:
110         if stop:
111             print "Stopping node %s" % instance.name
112             instance.stop()
113         elif delete:
114             print "Deleting node %s" % instance.name
115             instance.delete()
116     except:
117         print "Cant stop/delete instnace %s" % instance.name
118         traceback.print_exc(file=sys.stdout)
119
120 def update_conf():
121     """ Do we need to update the configuration? """
122     global ha_reloaded
123     stats = hastats.get_stat_backends()
124     stack = openstack()
125     backends = stack.backends()
126     if not len(backends) == len(stats):
127         ha = haproxy.HAproxy()
128         ha.compile(backends)
129         global ha_last_reload
130         ha_last_reload = datetime.datetime.now()
131         if ha.restart():
132             ha_reloaded = True
133         return True
134     return False
135
136 def initiate():
137     # Boot first machines if not active:
138     stack = openstack()
139     backends = stack.backends()
140
141     # Gathering first data
142     data = {}
143     data['acu'] = hastats.get_backend_cum_requests()['stot']
144     data['diff'] = 0
145     data['diffpt'] = 0
146     data['date'] = datetime.datetime.now()
147     data['active'] = len(stack.active_backends())#len(hastats.get_backends_up())
148     data['haactive'] = len(hastats.get_backends_up())
149     data['needed'] = None
150     data['epoch'] = (datetime.datetime.now()-epoch_start).seconds
151     metrics.append(data)
152     print metrics
153     time.sleep(sleeptime)
154     last = data
155     data = {}
156     data['acu'] = hastats.get_backend_cum_requests()['stot']
157     data['diff'] = int((float(data['acu']) - float(last['acu'])) / float(sleeptime))
```

```
158     data['diffpt'] = data['diff'] * sleeptime
159     data['date'] = datetime.datetime.now()
160     data['needed'] = needed_servers(acu=data['acu'], diff=data['diff'])
161     data['active'] = len(stack.active_backends())#len(hastats.get_backends_up())
162     data['haactive'] = len(hastats.get_backends_up())
163     data['epoch'] = (datetime.datetime.now()-epoch_start).seconds
164     metrics.append(data)
165     time.sleep(sleeptime)
166
167 def needed_servers(acu=None,diff=None):
168     last = metrics[-1]
169     # calculate servers for the load
170     # requests per sec / threshold
171     # ceil: rounds up
172     #needed = int(ceil(float(last['diff']) / float(server_threshold)))
173     print diff
174     needed = int(ceil(int(diff) / float(server_threshold)))
175     return needed
176
177 def new_metrics(current_cumulated, hareset=False):
178     global ha_reloaded
179     global ha_last_reload
180     current = {}
181     current['acu'] = current_cumulated
182     current['date'] = datetime.datetime.now()
183
184     if ha_reloaded:
185         last_cumulated = 0
186         difference = int(ceil((float(current_cumulated) - float(last_cumulated)) \
187                 / float((current['date'] - ha_last_reload).seconds)))
188         diffpt = int(difference) * (current['date'] - ha_last_reload).seconds
189
190     try:
191         print "Current new cumulated connections: %s" % str(current_cumulated)
192         print "Calculation: float(%s) - float(%s) / float(%s-%s.seconds (%s))" % \
193                 (str(current_cumulated), metrics[-1]['acu'], str(current['date']), str(metrics[-1]['date']),\
194                 str((current['date'] - metrics[-1]['date']).seconds))
195
196         if ha_reloaded:
197             current['diff'] = difference
198             current['diffpt'] = diffpt
199             ha_reloaded = False
200         else:
201             current['diff'] = int(ceil((float(current_cumulated) - float(metrics[-1]['acu'])) \
202                     / float((current['date']-metrics[-1]['date']).seconds)))
203             current['diffpt'] = current['diff'] * (current['date']-metrics[-1]['date']).seconds
204     except ZeroDivisionError:
205         current['diff'] = 0
206
207     stack = openstack()
208     current['needed'] = needed_servers(acu=current['acu'], diff=current['diff'])
209     current['active'] = len(stack.active_backends())#len(hastats.get_backends_up())
210     current['haactive'] = len(hastats.get_backends_up())
211     current['epoch'] = (datetime.datetime.now()-epoch_start).seconds
212
213     metrics.append(current)
214     return current
215
216 def main():
217     # Starting the first time
218     # getting current cum connections
219     try:
220         if not metrics:
221             print("Gathering initial data...")
222
223             # Gathering first data
224             initiate()
225
226         while True:
227             current = new_metrics(hastats.get_backend_cum_requests()['stot'])
228             print metrics[-1]
229             print "Needed servers: %s" % str(needed_servers(diff=current['diff']))
230
231             # What to do? Scale up/down or are we happy?
232             stack = openstack()
233             active_backends = stack.active_backends()
```

```
234             up_backends = hastats.get_backends_up()
235
236             needed = needed_servers(diff=current['diff'])
237             if needed > len(active_backends):
238                 print "Scaling up"
239                 scale_up(needed-len(active_backends))
240             elif needed < len(active_backends):
241                 print "Scaling down"
242                 if not scale_down(Needed=needed):
243                     print "Lowest number"
244             else:
245                 # Sleeping
246                 print "Sleeping one more round"
247
248             if update_conf():
249                 print "HAproxy config reloaded"
250                 print ha_last_reload
251
252             for line in hastats.get_stat_backends():
253                 print line['svname'] + ', ' + line['status']
254
255             time.sleep(sleeptime)
256
257     except KeyboardInterrupt:
258         write_data()
259
260 if __name__ == '__main__':
261     main()
```

Listing 1.11: Nova.py: OpenStack integration

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  from novaclient import client, v1_1
5  import novaclient
6  from os import environ, path
7  from threading import Thread
8  from natsort import humansorted
9  import haproxy, hastats
10 import time
11 import ConfigParser
12
13 class openstack:
14
15     def __init__(self):
16         self.p = path.split(path.dirname(path.abspath(__file__)))[0]
17         opens = path.join(self.p, 'etc/openstack.cfg')
18         if path.isfile(opens):
19             config = ConfigParser.ConfigParser()
20             config.read(opens)
21         else:
22             logging.error("Missing configuration file 'etc/openstack.cfg'")
23             exit(1)
24
25         self.cred = {}
26         self.cred['version'] = config.get('main', 'API_VERSION')
27         self.cred['username'] = config.get('main', 'USERNAME')
28         self.cred['api_key'] = config.get('main', 'PASSWORD')
29         self.cred['auth_url'] = config.get('main', 'AUTH_URL')
30         self.cred['project_id'] = config.get('main', 'TENANT_NAME')
31
32         self.nova = client.Client(**self.cred)
33
34     def reload_conf(self):
35         self.__init__(self)
36
37     def backends(self):
38         """ Get the virtual machines that are backends
39             Returns: List of server objects
40         """
41         nodes = self.nova.servers.list()
42         backends = []
43
44         for node in nodes:
```

```
45          if "node" in node.name:
46              backends.append(node)
47
48      obj = humansorted(backends, key=lambda x: x.name, reverse=True)
49
50      return obj
51
52  def active_backends(self):
53      active = []
54      for node in self.backends():
55          if node.status in 'ACTIVE':
56              active.append(node)
57
58      return active
59
60  def passive_backends(self):
61      passive = []
62      for node in self.backends():
63          if not node.status in 'ACTIVE':
64              passive.append(node)
65      return passive
66
67  def create_multiple(self, number):
68      backends = self.backends()
69      namenr = self.get_instance_number(nodes=backends, next=True)
70      instances = []
71      threads = []
72      for i in range(0,number):
73          name = 'node-%s' % str(namenr + i)
74          #print "start %s" % name
75          thread = Thread(target=self.create_backend, kwargs={'nextname':name})
76          thread.start()
77          threads.append(thread)
78
79      for thread in threads:
80          thread.join()
81          #print "next2"
82          #self.create_backend()
83
84  def create_backend(self, nextname=None):
85      """ Creates a instance in Openstack """
86      backends = self.backends()
87
88      name = ''
89      if nextname:
90          #print nextname
91          name = nextname
92      else:
93          name = 'node-%s' % str(self.get_instance_number(nodes=backends, next=True))
94
95      keypair = self.nova.keypairs.find(name='hlarshaugan')
96      image = self.nova.images.find(name='ubuntu-12.04')
97      #flavor = self.nova.flavors.find(name='m1.medium')
98      flavor = self.nova.flavors.find(name='m1.tiny')
99      net = self.nova.networks.find(label='MS016A_net')
100     nics = [{"net-id": net.id, "v4-fixed-ip": ''}]
101     f = open(path.join(self.p, 'etc/clouddata.txt'), 'r')
102
103     # try/except novaclient.exceptions.OverLimit
104     try:
105         server = self.nova.servers.create(name = name,
106                                 image = image.id,
107                                 flavor = flavor.id,
108                                 nics = nics,
109                                 key_name = keypair.name,
110                                 userdata=f)
111     except novaclient.exceptions.OverLimit:
112         print "No more available resources"
113         return None
114
115     status = server.status
116     while status == 'BUILD':
117         time.sleep(5)
118         instance = self.nova.servers.get(server.id)
119         status = instance.status
120
```

```
121         #return instance
122 
123     def start(self, instance):
124         if isinstance(instance, v1_1.servers.Server):
125             instance.start()
126         else:
127             self.nova.servers.findall(name=instance)[0].start()
128 
129     def shutdown(self, instance):
130         if not instance:
131             return False
132         if isinstance(instance, v1_1.servers.Server):
133             instance.stop()
134         else:
135             self.nova.servers.find(name=instance).stop()
136 
137     def delete(self, instance):
138         """ Terminates a instance """
139         if isinstance(instance, v1_1.servers.Server):
140             instance.delete()
141         else:
142             self.nova.servers.findall(name=instance)[0].delete()
143 
144     def get_instance_number(self, nodes=None, next=False,lowest=False):
145         if not nodes:
146             nodes = self.backends()
147         numbers = []
148         for node in nodes:
149             numbers.append(int(node.name.split('-')[1]))
150 
151         if next:
152             if numbers:
153                 return max(numbers) + 1
154             else:
155                 return 1
156         elif lowest:
157             return min(numbers)
158         else:
159             return numbers
160 
161     def sleeping_machine(self):
162         """ Returns the first shutoff machine """
163         backends = self.backends()
164         ret = []
165         for node in backends:
166             if node.status in 'SHUTOFF':
167                 ret.append(node)
168         return ret
169 
170 def main():
171     stack = openstack()
172     print stack.backends()
173     print stack.create_backend()
174 
175     #print nova.limits.get().to_dict()
176     #print nova.quotas.get('59a46c9fcf174ec3890211cc86e0836b', user_id='s171201').instances
177 
178 if __name__ == '__main__':
179     main()
```

Listing 1.12: haproxy.py: HAProxy integration

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3 
4  import subprocess
5  from haconn import HAconn
6  from nova import openstack
7  from os import path
8  from jinja2 import Environment, PackageLoader, FileSystemLoader
9 
10 socket = "/var/run/haproxy/admin.sock"
11 class HAproxy:
12     def __init__(self):
13         self.subnet = 'MS016A_net'
```

```
14          pass
15
16      def restart(self):
17          """ """
18          # Need to collect the current sessions before restart!
19          # service haproxy reload # < will reload the config with minimal service impact
20          # Needs to be run with root privileges
21          pr = subprocess.Popen("service haproxy reload".split(), stdout=subprocess.PIPE)
22          output, err = pr.communicate()
23
24          if not err:
25              return True
26          else:
27              return False
28
29      def compile(self, serverlist):
30          #http://jinja.pocoo.org/docs/dev/api/
31          p = path.dirname(path.abspath(__file__))
32          env = Environment(loader=FileSystemLoader(path.split(p)[0] + '/etc/'))
33          template = env.get_template('haproxy.cfg')
34
35          nodes = []
36          counter = 1
37          for server in serverlist:
38              s = {}
39              s['name'] = server.name
40              try:
41                  s['ip'] = server.addresses[self.subnet][0]['addr']
42              except KeyError:
43                  stack = openstack()
44                  ip_activate = False
45                  while not ip_activate:
46                      try:
47                          s['ip'] = stack.nova.servers.find(name=server.name).addresses[self.subnet][0]['addr']
48                          ip_activate = True
49                      except KeyError:
50                          continue
51
52              s['id'] = counter
53              counter += 1
54              nodes.append(s)
55
56          # Example
57          # nodes=[ {'name': 'node01', 'ip': '192.168.128.48','id':1},]
58          with open('/etc/haproxy/haproxy.cfg', 'wb') as f:
59              f.write(template.render(nodes=nodes))
60
61      def set_online(self, instancename):
62          conn = HAconn()
63          ret = conn.send_cmd('enable server nodes/%s\r\n' % (instancename))
64          conn.close()
65          return ret
66
67      def set_offline(self,instancename):
68          conn = HAconn()
69          ret = conn.send_cmd('disable server nodes/%s\r\n' % (instancename))
70          conn.close()
71          return ret
72
73      def drain(self, instance):
74          conn = HAconn()
75          ret = conn.send_cmd('set server nodes/%s state drain\r\n' % instance.name)
76          conn.close()
77          return ret
78
79
80  def main():
81      ha = HAproxy()
82      ha.compile()
83
84  if __name__ == '__main__':
85      main()
```

Listing 1.13: hastats.py: HAProxy statistical integration

```
#!/usr/bin/env python
# −∗− coding: utf−8 −∗−

from haconn import HAconn
import pprint

def get_info():
    con = HAconn()
    data = con.send_cmd('show info\r\n')
    con.close()

    return dict([ line.split(': ') for line in data.splitlines() if line])

def get_stat(output=None):
    con = HAconn()
    data = con.send_cmd('show stat\r\n')

    lines = data.splitlines()
    header = lines.pop(0).split('# ')[1].split(',')
    #print header
    l = []
    for line in lines:
        if len(line.split(',')) > 1:
            l.append(dict(zip(header, line.split(','))))
    con.close()

    if output:
        s = ''
        for key, value in l[0].iteritems():
            s += 'key (%s)' % key
            for i in l:
                if key and len(i) > 1:
                    s += i[key] + ','
            s += '\n'
        print s

    return l

def get_stat_backends():
    stats = get_stat()
    backends = []
    for node in stats:
        if 'node' in node['svname']:
            backends.append(node)

    return backends

def get_backend_cum_requests():
    stats = get_stat()
    for node in stats:
        if 'nodes' in node['pxname'] and 'BACKEND' in node['svname']:
            return node

def get_backends_up():
    backends = get_stat_backends()
    backs = []
    for node in backends:
        if 'UP' in node['status']:
            backs.append(node)

    return backs

def cum_req():
    pass

def previous_req():
    pass

def main():
    #print get_info()
    #print get_cur_req()
    get_stat(output=True)

if __name__ == '__main__':
    main()
```

Listing 1.14: haconn.py: HAProxy socket connection class

```
#!/usr/bin/env python
# −∗− coding: utf−8 −∗−

from socket import socket, AF_UNIX, SOCK_STREAM
#from haproxy import const

HA_BUFSIZE = 8192

class HAconn:
    def __init__(self, sockfile=None):
        if not sockfile:
            self.sockfile = '/var/run/haproxy/admin.sock'
        else:
            self.sockfile = sockfile
        self.sock = None
        self.open()

    def open(self):
        self.sock = socket(AF_UNIX, SOCK_STREAM)
        self.sock.connect(self.sockfile)

    def send_cmd(self, cmd):

        res = ""
        self.sock.send(cmd)
        output = self.sock.recv(HA_BUFSIZE)

        while output:
            res += output
            output = self.sock.recv(HA_BUFSIZE)

        return res

    def close(self):
        """Closes the socket"""
        self.sock.close()

def main():
    #con = HAconn('/var/run/haproxy/admin.sock')
    ##print con.send_cmd('show stat\r\n')
    #print con.send_cmd('enable server nodes/node02\r\n')
    #con.close()
    con = HAconn()
    print con.send_cmd('show info\r\n')
    con.close()
    con = HAconn()
    print con.send_cmd('show stat\r\n')
    con.close()
    con = HAconn()
    print con.send_cmd('show sess\r\n')
    con.close()
    con = HAconn()
    print con.send_cmd('show table\r\n')
    con.close()

if __name__ == '__main__':
    main()
```

Chapter 2

Comparing HAProxy and Pound load balancers

Keywords: Webservers, Performance, Analysis, Load Balancers, Scripting

Abstract

This chapter is a comparison of the two load balancers HAProxy and Pound, where they are compared in the fields of Configurability, performance and scalability. Through the testing of the different load balancers, a comparative study will present the values of each load balancer.

2.1 Introduction

When thinking about the Internet today, most people associate this with the Wold Wide Web. The reason for this is simple, as this is what everyday users use daily. The main components to running the world wide web is the webservers, which make all the content available through the HTTP and HTTPS protocols. These protocols have been around for a long while, where the first real implementation were presented in 1992 with the first function of the GET options (Berners-Lee, 1992). The protocol were designed to transfer text files over a ẗelnet-style internet protocol, but the protocol has evolved much from the first iterations.

Last years there has been a shift in the way people use the Internet, and with new streaming services like Spotify, Netflix and many other streaming services, the webservers are getting more and more important. Other services like social network, online banking, communications and almost anything imaginable of information is mostly available. Although the information

is out there, this is not the only worry anymore. The information should be available almost instant, and at the users demand. This is also included in one of the most common security phrases; the CIA attributes: confidentiality, integrity and **availability**.

All these services utilizes the power of the webservers, and the HTTP and HTTPS protocols. But it is not possible to rely on one single machine to provide the information to the masses. Not only because of general failure, but due to high demand for the information, or the complexity of presenting it. A single server can shutdown, but the information is still expected to be available.

To solve this problem, a layer between the end users and the webservers, called a loadbalancer is introduced. This solution has the intention of spreading the workload among the available servers and much more. These are the facility for providing a more reliable World Wide Web, and to facilitate for high availability and flexibility.

Problem statement

The given problem statement in this project was as follows:

Setup and evaluate and compare the HAproxy and pound load balancers for a web- service with regard to:

- *Configureability*
- *Performance*
- *Scalability*

2.2 Background

Through this chapter we will focus on two different load balancers for load balancing HTTP traffic. To be able to compare the load balancers, a short introduction to the two different load balancers are needed, along with a introduction to the concepts of balancing http traffic. How does a web server work in short, and how is it possible to test the performance.

2.2.1 Load balancing HTTP and HTTPS

Load balancers are important to most of the high end websites that are available. Ensuring high availability and scalability. They can be a part of even the smallest sites that are prone to fail, ensuring high uptime, or to balance the load over multiple web servers, or even database servers.

There are many different load balancers available. Many of these are open source and free, but with varying quality and support. There are also many enterprise load balancers or reverse proxy's as they are usually referenced to, like Alteon product from Radware and Big IP from F5. But a load balancer could be as simple as a single web server running apache or nginx.

The most basic feature of a load balancer is the ability to forward HTTP traffic to the configured backends. The load balancer acts as a reverse proxy, redirecting requests to one of the configured web servers. The algorithm used for choosing server is normally some sort of least-connections or round-robin distribution, distributing the load over the different web servers. The traffic is then returned from the web server through the load balancer, to the client that requested the specific page. Users do not see any of the actions of the load balancer, making it a transparent service enabling high availability.

HTTP is the most common protocol for communicating with web servers, although it does not provide any security measures. The security part is only implemented in the HTTPS protocol, which implements security features like encryption and web site verification with certificates. Some load balancers have the possibility of terminating SSL connections. This would take the overhead of encryption and decryption of traffic away from the web servers, and enable the web servers to focus on the work needed to present the data.

SSL termination is just one of the many possibilities a load balancer for HTTP/S brings to the table. It makes it possible to masquerade many servers with different areas of expertise, like specific static content servers, and dynamic content servers seem to be the same address, without the specific handling in code. There are countless of possibilities, and load balancing is now a must have for larger sites or site with the need for high availability.

2.2.2 HAProxy

HAProxy is a powerful load balancer and reverse proxy. It has lots of configuration possibilities, and large sites, like Reddit and Stackexchange use this tool to load balance their requests. (HAProxy, 2014c). It is a free software, that now comes packaged with the most common Linux distributions used, and is also supported by companies like Red Hat. HAProxy is written in C, and is therefore both lightwait and fast enabling high performance load balancing. It can also be used to balance the TCP protocol in general which means you can use it to proxy eg. databases.

2.2.3 Pound

Pound is a open-source reverse proxy and HTTP load balancer designed to secure applications. It is developed by Apsis, a security company, and created to enable distribution of load over multiple web servers, and also to enable SSL wrapping, for servers not offering this natively. It is a very small program which is easily audited for security issues (APSIS, n.d.).

2.2.4 Benchmarking with Httperf

To benchmark both web servers and load balancers, a tool which can perform a high number of connections to the web server at a short time interval is needed. A tool like this is the Httperf tool created by HP. With this tool it is possible to create a high number of HTTP or HTTPS requests to a website. This is needed to see how the different component works under high pressure. At one point there will be something that cannot handle the high load, but the question is what the magical number would be. The key performance areas should therefor be measured and evaluated, and this can be done with the output from httperf.

The quality of the different performance benchmarking tools for HTTP, that are openly available, are very variable. Httperf is a tool which does the job, but it is old, and contain some bugs which you may find. Though httperf actually can handle SSL connections, it does so only up to the standard of SSL 3.0. All the versions of SSL/TLS up until version TLS 1.1 is broken with specific ciphers (RC4). The performance testing of the handling of SSL/TLS is important, but due to the differences in the SSL/TLS versions, and the unsupported usage of httperf with these versions, this is not included in this comparison.

The best way to benchmark the different load balancers is therefor to test the performance handling of HTTP with different loads. The easiest way to get httperf is to install through the package manager in debian based systems. It can also be compiled manually. This is needed to do testing with SSL/TLS, as httperf needs the OpenSSL library. Httperf can be used to test the performance with different connection rates, number of calls per connection and over a longer period. An example of such a test is run with the following command:

Listing 2.1: Httperf output example

```
httperf --hog --server balance2 --port 80 --uri /perf.php --num-conn 6000 --rate 100 --num-call 1 --
    timeout=5
httperf --hog --timeout=5 --client=0/1 --server=balance2 --port=80 --uri=/perf.php --rate=100 --send-
    buffer=4096 --recv-buffer=16384 --num-conns=6000 --num-calls=1
httperf: warning: open file limit > FD_SETSIZE; limiting max. # of open files to FD_SETSIZE
Maximum connect burst length: 1

Total: connections 6000 requests 6000 replies 6000 test-duration 59.995 s

Connection rate: 100.0 conn/s (10.0 ms/conn, <=3 concurrent connections)
Connection time [ms]: min 2.3 avg 3.1 max 60.5 median 2.5 stddev 1.1
Connection time [ms]: connect 0.5
Connection length [replies/conn]: 1.000

Request rate: 100.0 req/s (10.0 ms/req)
Request size [B]: 69.0

Reply rate [replies/s]: min 100.0 avg 100.0 max 100.0 stddev 0.0 (11 samples)
Reply time [ms]: response 2.6 transfer 0.0
Reply size [B]: header 167.0 content 8.0 footer 0.0 (total 175.0)
Reply status: 1xx=0 2xx=6000 3xx=0 4xx=0 5xx=0

CPU time [s]: user 30.92 system 28.78 (user 51.5% system 48.0% total 99.5%)
Net I/O: 23.8 KB/s (0.2*10^6 bps)
```

```
Errors: total 0 client−timo 0 socket−timo 0 connrefused 0 connreset 0
Errors: fd−unavail 0 addrunavail 0 ftab−full 0 other 0
```

The output of httperf gives information about the connection metrics, request metrics, reply metrics, system metrics and errors during the test. This information is important in finding the metrics of how the system is performing. The command above can be explained as to run a test which lasts for 60 seconds, where total of 6000 connections are established, where 100 connections is established each second. It uses the *–hog* option which unlimits the usage of only 1024 ports for httperf to a range from 1024 to 5000. Httperf uses this to use as many TCP ports as needed. The *–server* option specifies which server to connect to. The other options are important to specify the number of connections and the time it will take for the test to run. It is important to keep the rate of total connections and rate constant.

2.3 Approach

The operationalization asks for a study of two load balancers which can be observed under the same conditions. There is therefor a need to have a configuration and setup that is as similar as possible, so that the data collected are comparative.

The two load balancers in question is th HAProxy and Pound load balancers. They will be compared in regards to configurability, performance and scalability.

2.3.1 Configurability

The configurability of the two load balancers will be tested after the installation and configuration. This will result in experience with the two solutions and provide a view on how they are different. How the configuration of the two solutions are will be based on the configuration options available, with a special focus on the documentation provided. This especially since good documentation is key to correct configuration of the system.

Through the documentation, the configuration possibilities are provided, giving a view on the different possibilities the system gives to the user. There should be some expected key points in the configuration where a given set of pre defined options should be compared.

How the syntax of the configuration files is structured and viewable is important to a system to ensure that misconfiguration does not happen. This is a measure which is not directly quantifiable, but could be measured by counting the number of characters used for specific operations. This should be used with care, as more characters does not directly mean bad syntax. Through this the usage of special characters could be used as a measure.

The configurability should also address the possibility of integrating outside code into the load balancer, for either gathering metrics, or doing configuration changes.

With this the following aspects needed to be addressed:

- Configuration options
- Documentation and quality of documentation
- Syntax of configuration
- Integration possibilities

With the configuration options there is the need to set some specific configurational attributes. The configuration options things that should be present in the load balancers.

- HTTPS termination
- Management/Integration interface
- Support
- Logging posibilities
- Graphical interface
- Documentation
- Requirements
- Balancing modes (algorithms)
- Cookie support
- IPv6

2.3.2 Performance

The benchmarking of load balancers is tricky. The expected result is not to actually find the saturation point. HAProxy as an example has been reported to successfully handle the load of a full 1 gigabit ethernet connection, and does also handle the loads of large pages (HAProxy, 2014b). To test the performance of the different load balancers, the key aspects should be the usage of system resources. High amount of forking or many system calls can actually lead to slower performance of the system, and cause the hardware or platform to be more powerful. This is a key aspect to the performance. It is not expected to find the breaking point, but with

this it is possible to find the surrogate values that could provide an answer to if the application could fail in the feature and with higher load.

While doing the performance testing, a webpage is needed on each of the backends. Since it is the load balancer that is tested, it is important that the backend servers will not get saturated through high load. This can be done by using a php script which uses the sleep function. This ensures that the web requests take longer time, but still does not cause high CPU iterations on the backends. Some tests of different web pages shows that a normal response time is about 0.2 seconds. This is also around the time for cognitive response to be observed, and if a request take longer than this time limit, the user would observe the page as loading. The value of 0.2 seconds is therefor chosen, and the two load balancers should be tested with this sleep time. The response time relative to this should be tested, so that the basic user experience can be shown.

To test the performance of the load balancers, httperf will be used. To be able to collect the information gathered from httperf, a script that can parse the output of the tool is needed.

2.3.3 Scalability

There are different parts of the object of scalability. The load balancer should be able to handle many backend servers, but should also handle the possibility of failover from the running instance, enabling multiple load balancers to run. Having only one load balancer running, would result in a single point of failure.

One site may have one or more load balancers, but there are usually a lot more applications than just the single web applications that are expected here. It should therefor be possible to create multiple frontends in the load balancer along with multiple backends, and map these respectively.

The quantitatively measure of this, could possibly be found in the documentation of the load balancers if the documentation is good. This will provide a input on how many frontends, backends and possible backend servers that are configurable.

2.4 Result

2.4.1 Configurability

Through the points set in the approach the different points of the two load balancers can be viewed.

HAProxy

HAProxy is a well formed load balancer, which is beautifully implemented. The documentation on the website **??** is well written, and complete with performance tuning options and a lot of information. The documentation is split into different parts; Configuring HAProxy, Gloabl parameters, Proxies, Bind and Server options, HTTP header manipulation, Using ACLs and fetching samples, Logging and Statistics and monitoring is the main parts. A lot of performance tuning options are available where an advanced options can be tuned to get the best performance for your system. The tests that has been done in this report does not take advantage of any of these.

The configuration file is split neatly into three main parts. The *global*, *frontend* and *backend*. The global part contains all the global configuration 2.5 on page 42. The frontend part is extendible, and for each application, a new frontend should be added. For the backends it is the same, as a backend represents a list of available servers for a frontend.

The syntax of the configuration is nice, where indentation is used. There is not a lot of special characters, making the configuration file easy to read.

Integration is also possible through the socket connection available. This requires an application running on the same server, as this is a local socket, and not a network socket. Through this socket connection, there is options to set the server in maintenance mode, and other modes runtime. There is no option for adding servers through the socket connection. This is OK, as it will maintain a complete configuration file, and ensuring that every configuration option is stored in the configuration file.

Pound

Pound is a more lightweight load balancer compared to HAProxy, and has less records of success posted on their web page. The documentation of pound is sparse (APSIS, n.d.). Although it has a well formed man page, with configuration examples and some descriptions of the different options.

Though there is performance tuning options in the configuration documentation, it seems like the most important function for a small scale site is present, making it possible to create a high available site.

The configuration file of pound, 2.6 on page 43, is compared to HAProxy messy. It Uses many of the same features, but have some other naming convention. It also practices start / end blocks of each option, shown in the minimal configuration. There is no excessive usage of special characters which is good, but it uses on the other hand keywords in camel casing, which makes the configuration file look more messy. It gets bad when adding many different backends, as each backend is followed by a block with the naming *END*.

For runtime configuration, there is a tool called poundctl which enable connection to the socket presented. This gives the same functions as available with the socket interface natively. This includes enable/disable a listener, service or backend. The tool is old, but still works.

Comparison of features

A short comparison of the different features found in HAProxy and Pound.

Feature	HAProxy	Pound
HTTPS termination	Yes	Yes
Socket connection	Yes	Yes
Logging possibilities	Configurable to capture traffic and much more	Syslog or stdout
Graphical interface	Web statistics overview	None
Documentation	Really good documentation	Sparese
Requirements	OpenSSL for SSL termination	OpenSSL for SSL termination
Balancing modes	10 different. Incl. most common	Unknown
Session/ Cookie support	Yes	Yes
IPv6	Yes - IPv6 to IPv4 support	Yes - IPv6 to IPv4 support

2.4.2 Performance

To be able to do the tests needed to run httperf with an incremental mode, and getting the data from httperf, a opensource project on Github has been used. This project called Httperfpy (Joshua Mervine, 2014) can based on the input given to the class constructor, run httperf with the options given, and if specified parse the output. Httperpy did have some things missing, like the possibility of passing *–hog* to httperf. The changes necessary for adding this option were added, and submittet back to the project based on the following pull request (Haugan, 2014), which is added and now available in the pypi package.

The script used for running the performance tests is added as appendix 2.3 on page 41. The script takes the output of httperf and parses it, before it appends it to a csv file for the current measurement. This makes the project repeatable, and there is not a lot of prerequisites needed to perform the tests. The csv file created will have the following format, where the new measurements are appended to the file:

Listing 2.2: Data output of a combined run

```
reply_time_transfer,errors_addr_unavail,connection_rate_ms_conn,reply_time_response,connection_time_min,request_size,
    errors_ftab_full,net_io_kb_sec,errors_client_timeout,reply_rate_stddev,connection_length,reply_rate_max,
    cpu_time_total_pct,cpu_time_system_pct,reply_status_3xx,connection_rate_per_sec,errors_conn_refused,errors_total,
    reply_status_2xx,connection_time_avg,errors_conn_reset,request_rate_ms_request,cpu_time_user_sec,
    errors_fd_unavail,cpu_time_system_sec,connection_time_max,reply_rate_min,reply_status_5xx,
```

connection_time_stddev,reply_rate_avg,reply_size_total,reply_size_header,errors_other,reply_size_content,
request_rate_per_sec,reply_rate_samples,connection_time_connect,total_connections,cpu_time_user_pct,
reply_size_footer,max_connect_burst_length,errors_socket_timeout,total_replies,net_io_bps,reply_status_1xx,
reply_status_4xx,command,connection_time_median,total_requests,total_test_duration
0.0,0,10.0,2.9,2.5,79.0,0,24.8,0,0.1,1.000,100.0,99.6,47.8,0,100.0,0,0,6000,3.6,0,10.0,31.12,0,28.66,23.2,99.8,0,0.8,100.0,
175.0,167.0,0,8.0,100.0,12,0.7,6000,51.9,0.0,1,0,6000,0.2∗10^6,0,0,httperf −−hog −−client=0/1 −−server=
balance1 −−port=80 −−uri=/perf.php?sleep=0.2 −−rate=100 −−send−buffer=4096 −−recv−buffer=16384 −−
num−conns=6000 −−num−calls=1,3.5,6000,59.998

The following graph is a plotting of the response times gotten from the csv file above.

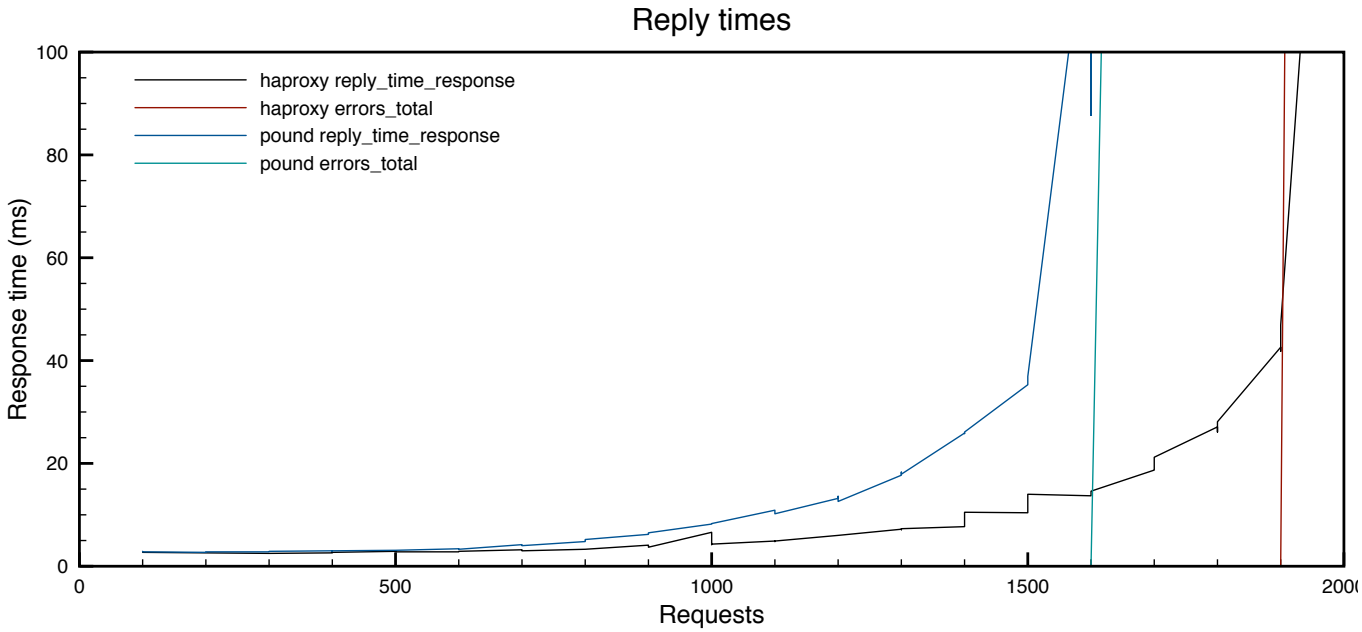

Figure 2.1: Reply times from the two load balancers

At the same time as the tests have been running, the CPU usage have been monitored on the load balancers. From the following graph the CPU utilization is visible. The test that has been running with this graph is an incremental test from 100 until 2000 requests a second at a incremental value of 100 requests. Each test were run 3 times. To measure these values, the *sar* command has been used to capture the CPU usage. This has simply been done by using the following command, which captures the CPU usage every second second.

```
$ sar -u 2 4000 >> cpu_mon_balance*
```

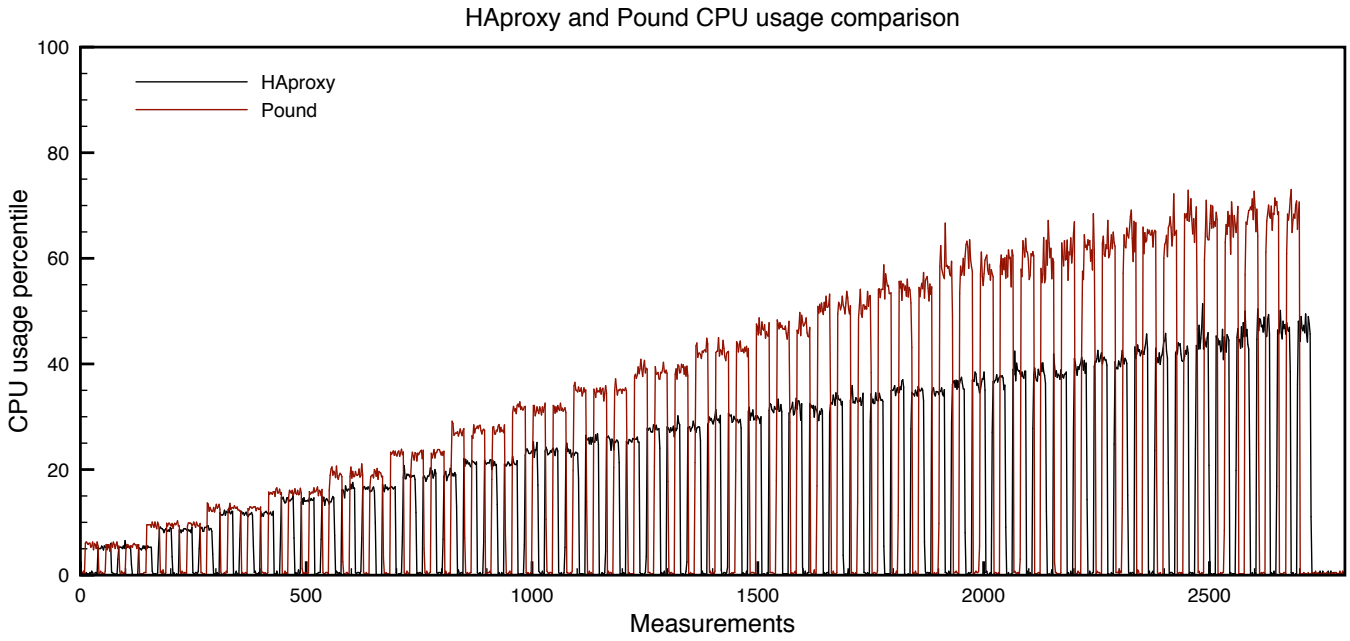

Figure 2.2: Cpu usage of HAProxy and Pound under different loads

2.4.3 Scalability

There are many possibilities with both Pound and HAProxy. HAProxy is probably the most mature of the two, where it is possible to synchronize the connection table of HAProxy to another HAProxy instance. With the usage of keepalived, it is possible to create a high availability setup with HAProxy. This is documented well by Red Hat (RedHat, n.d.). This is not very well documented in regards to Pound, so if this is possible, it is a good kept secret.

How many frontends and backends it is possible to add or use, and the performance impact of said configuration is something that is not covered well here. The documentation of either does not state the limitations on this aspect, which leads to assume that there is a lot more than what is necessary for normal usage.

2.5 Analysis

The data gathered in the results, presents to important graphs. We can from 2.1 on the previous page see that there is a difference in the way that HAProxy and Pound manages to respond to the requests. Already at 1000 requests per second, there is a noticeable difference, ant at 1500

requests per second Pound does not follow at all. The reason for this could be that the fd limit is set on the balancer, and this is also shown when the errors come. These errors are fd limit errors, and is not a key value for saying anything about how the balancer is handling.

With the information gathered about system resources we can see that there is a significant difference in the way that HAProxy and Pound work. Shown in 2.2 on the preceding page it shows that Pound uses as much as 20% more CPU than what HAProxy does. A reason for this could be that Pound forks new threads for each new connection, and has therefore a lot of system calls for each connection, whereas HAProxy is a single threaded process of effective code that has to do a lot less system calls than what of Pound.

2.6 Discussion and conclusion

A lot of data has been captured to be able to gather some information about the load balancers. This has mostly been gathered from httperf, but also system values with tools like sar. There is a lot of data that is captured, and more of this data could be graphed, as they make little meaning by themselves. This is one of the improvements that is needed to further sturdy the comparison of these load balancers.

It is important to note that the data gathered from the usage of httperf is done with the package manager of httperf with the fd_limit set. This can be a problem, and the errors has therefor been plottet in the graphs. The load balancers are also configured with the fd_limit set, meaning that both balancers will notice these errors, but will handle them differently.

It is not possible to be sure about any saturation point for both balancers, as the data gathered most likely show some sort of external effect from the operating system. This means that if the operating system and the load balancers were perfectly configured, a more accurately test could be done.

2.6.1 Improvements

There is a lot of improvements that could be done to this task, and there is still a lot of work to be done to be 100% sure of the results that are gathered. Some of the main improvements are the usage of SSL to test one of the most important features of these load balancers. The test does also only test the connection rate, and does not look at how the load balancers behave when higher sizes are forwarded. This would be relevant to video streaming and the like. The system should be configured for higher load to be able to accuracy measure the performance of the load balancers, as the setup today does not provide a perfect solution.

There are many other improvements that could be done to this testing, and although this gives a picture on the two load balancers, it is not a complete picture.

2.6.2 Conclusion

In summary, the task has been completed to some degree. There is still a lot of work left to have a complete finished performance comparison, and evaluation of the two laod balancers, but it is possible to get some key points from the information gathered. It is possible to conclude that HAProxy seems to be the best tool for the job in both smaller scale, and larger scale when it comes to configurability, both due to the amount of configuration possible, the good documentation and the beautiful configuration file.

Performance wise, HAProxy also has a small leap in front of Pound, where it has better response times at higher load, and also uses a lot less processor power.

Scalability is a topic that is not widely touch on here, but it is possible to see that HAProxy has an advantage in the possibilities of using keepalived.

2.7 Appendix

Listing 2.3: Script for running httperf through httperfpy

```
1  #!/usr/bin/env python
2  # −∗− coding: utf−8 −∗−
3
4  from httperfpy import Httperf
5  import csv
6  import os
7  import time
8  import argparse
9
10 # saturation:
11 # /usr/bin/httperf −−hog −−server balance1 −−port 80 −−uri="/perf.php?sleep=0.2" −−num−conns=15000 −−rate
       =2000 −−num−call=1
12
13 def run_httperf(server, num_conns=None, rate=None, num_calls=None, sleep=None):
14     if sleep:
15         uri = '/perf.php?sleep={}'.format(sleep)
16     else:
17         uri = '/perf.php?sleep=0.2'
18     perf = Httperf('hog', path='/usr/bin/httperf', server=server, port=80, uri=uri,
19             num_conns=num_conns, rate=rate, num_calls=num_calls)
20     perf.parser = True
21     results = perf.run()
22
23     print results
24
25     return results
26
27 def write(filename, results):
28     csv_data = []
29     header = results.keys()
30     if not os.path.isfile(filename):
31         csv_data.append(header)
32
```

```
    with open(filename, 'a') as f:
        w = csv.DictWriter(f, results.keys)
        csv_data.append([results[x] for x in results])
        w.writer.writerows(csv_data)

def main(server):
    if server:
        servername = server
    else:
        servername = 'balance1'
    length = 60 # sec
    increase_by = 100
    rate = 100
    max_rate = 2000
    num_calls = 1
    times = 3
    sleep = 1

    filename = 'perf_%s_combined' % (servername)
    while rate <= max_rate:
        #filename = 'perf_%s_r%s' % (servername,rate)
        num_conns = rate * length

        for t in range(0,times):
            #result = run_httperf(servername, num_conns=1000, rate=25, num_calls=1)
            result = run_httperf(servername, num_conns=num_conns, rate=rate, num_calls=num_calls,sleep=sleep)
            write(filename, result)
            time.sleep(30)

        rate += increase_by

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description="Performance testing VIP")
    parser.add_argument('--server', metavar='server', default=None, help="Server to test")
    args = parser.parse_args()
    main(args.server)
```

Listing 2.4: Webpage used for benchmarking

```
<?php
    $rounds = $_GET['rounds'];
    $sleep = $_GET['sleep'];
    $random = $rounds * rand(1,10);
    echo "Rounds $random";
    sleep($sleep);

?>
```

Listing 2.5: HAProxy configuration file

```
global
    log /dev/log  local0
    log /dev/log  local1 notice
    chroot /var/lib/haproxy
    stats socket /run/haproxy/admin.sock mode 660 level admin
    stats timeout 30s
    user haproxy
    group haproxy
```

```
    daemon

    # Default SSL material locations
    ca-base /etc/ssl/certs
    crt-base /etc/ssl/private

    # Default ciphers to use on SSL-enabled listening sockets.
    # For more information, see ciphers(1SSL).
    ssl-default-bind-ciphers kEECDH+aRSA+AES:kRSA+AES:+AES256:RC4-SHA:!kEDH:!LOW:!EXP:!MD5:!
        aNULL:!eNULL

defaults
    log global
    mode  http
    option  httplog
    option  dontlognull
    timeout connect 4s
    timeout client 1m
    timeout server 1m
    timeout queue 1m
    timeout http-request 5s
    errorfile 400 /etc/haproxy/errors/400.http
    errorfile 403 /etc/haproxy/errors/403.http
    errorfile 408 /etc/haproxy/errors/408.http
    errorfile 500 /etc/haproxy/errors/500.http
    errorfile 502 /etc/haproxy/errors/502.http
    errorfile 503 /etc/haproxy/errors/503.http
    errorfile 504 /etc/haproxy/errors/504.http

listen stats :2000
    mode http
    stats enable
    stats hide-version
    stats realm Haproxy\ Statistics
    stats uri /

frontend web-service
    bind *:80
    default_backend nodes

backend nodes
    server web-1 {{node.ip}}:80 id {{node.id}} check
    server web-2
    http-check expect string It\ works!
    option httpchk GET /
```

Listing 2.6: Pound configuration file

```
## Minimal sample pound.cfg
##
## see pound(8) for details


######################################################################
## global options:

User    "www-data"
Group   "www-data"
#RootJail "/chroot/pound"

```

```
13 ## Logging: (goes to syslog by default)
14 ##  0 no logging
15 ##  1 normal
16 ##  2 extended
17 ##  3 Apache-style (common log format)
18 LogLevel  1
19
20 ## check backend every X secs:
21 Alive   30
22
23 ## use hardware-accelleration card supported by openssl(1):
24 #SSLEngine  "<hw>"
25
26 # poundctl control socket
27 Control "/var/run/pound/poundctl.socket"
28
29
30 ######################################################################
31 ## listen, redirect and ... to:
32
33 ## redirect all requests on port 8080 ("ListenHTTP") to the local webserver (see "Service" below):
34 ListenHTTP
35   Address 0.0.0.0
36   Port  80
37   HeadRemove "X-Forwarded-For"
38   ## allow PUT and DELETE also (by default only GET, POST and HEAD)?:
39   #xHTTP    0
40 End
41 #ListenHTTPS
42 # Address 0.0.0.0
43 # Port 443
44 # AddHeader "X-Forwarded-Proto: https"
45 # HeadRemove "X-Forwarded-Proto"
46 # HeadRemove "X-Forwarded-For"
47 # Cert "/etc/pound/mycert.pem"
48 #End
49 Service
50   BackEnd
51     Address 192.168.128.187
52     Port  80
53   End
54   BackEnd
55     Address 192.168.128.185
56     Port  80
57   End
58   BackEnd
59     Address 192.168.128.227
60     Port  80
61   End
62   BackEnd
63     Address 192.168.128.228
64     Port  80
65   End
66 End
```

Listing 2.7: Script for getting csv data into plot

```
1 # Script for importing csv files to plot2 with comments
2 open(IN,$file);
3 @descr = @array;
```

```
4  $line=0;
5  while(<IN>) {
6      @column=split(',', $_);
7      if($column[0] == "nr"){
8          @descr = @column;
9      }else{
10         &log("Iterating data line $line\n");
11
12         for $i (1 .. $#column){
13             $data[$i-1][$line][0]=$column[0];
14             $data[$i-1][$line][1]=$column[$i];
15             $source[$i-1]="$file";
16             $comment[$i-1]="$descr[$i]";
17         }
18         $line++;
19     }
20 }
21 close(IN);
22 &log("$line number of points");
```

Chapter 3

A monitoring tool for /proc/diskstats

Keywords: Storage, Performance

Abstract

In this report, a tool for monitoring the */proc/diskstats* file is developed, storing the content for later analysis.

3.1 Introduction

Problem statement

The given problem statement in this project was as follows:

Write a script which is able to collect data from /proc/diskstats and store it in an orderly manner for later analysis.

3.2 Background

While running a system it is sometimes important to be able to gather metrics on how the system is performing. One of this metrics are the data from disk operations. This can help identify bot-

tlenecks in the system, and give a better understanding of how the system is using its resources.

To understand the usage and need for a tool that can monitor the file */proc/diskstat* a short introduction to the files under */proc* and the values stored inside is needed.

3.2.1 The /proc folder

/proc is a virtual filesystem, which sometimes is reffered to as a process information pseudo-file system (TLDP, n.d.). The files in the folder are not *real*, but is rather a means of getting information from the kernel.

3.2.2 /proc/diskstats

The file /proc/diskstats contain information about each block device on the system. In newer kernels (from 2.6 and newer) this information is also available under the path /sys/block/. Each line has the different information about each block device, which could either be the disk itself, or the partitions on it. The information for each block device displays the I/O statistics like reads, writes and time spent doing the different tasks.

The different fields of the file means the following:

1. major number
2. minor mumber
3. device name
4. reads completed successfully
5. reads merged
6. sectors read
7. time spent reading (ms)
8. writes completed
9. writes merged
10. sectors written
11. time spent writing (ms)
12. I/Os currently in progress
13. time spent doing I/Os (ms)
14. weighted time spent doing I/Os (ms)

These are then representing each column in the file which can look like the following:

Listing 3.1: Contents of /proc/diskstats

```
253       0 vda 31410 806 1020922 2564596768 1465561 1382324 54703840 3579730636 0 7229708 40955272
253       1 vda1 31182 334 1015322 2564547860 1465459 1382324 54703024 3579730152 0 7229220 40954784
```

There are other tools which enables the logging of different metrics from block devices as well, like the *sar* tool and *iostat* (ricklind@us.ibm.com, n.d.). One of the benefits the monitoring of /proc/diskstats holds over the usage of other tools, is the low overhead, and no need for additional packages.

This tool will enable the continuous logging of the kernel counters for the block devices specified, and prepare the data for later analysis.

3.3 Approach

The operationalization asks for a tool, which can read from the /proc/diskstats file and store the data appropriately for later analysis. The tool should be developed in Python, as this is one of the common scripting languages used. It also enables easy file operations functionality which is important to this task, as the information gathered will be in the format of a file.

The intention should be that the tool could be used in two ways. The normal usage, which would be user execution of the tool. The other way is the possibility of running the tool as a cron job. If the tool is run with a specific interval executed by the user, this would result in a process staying alive in the background the hole time. Running the tool with cron, would on the other hand only require the tool to be running at the time of execution, while the user would need to have a session open over time if executed by the user. This is easily done with applications like *screen* or *tmux*, but the usage of cron is a better way of doing it if the application should be running over longer periods of time.

3.3.1 The format of data

As shown in the background 3.2.2 the content of /proc/diskstats is containd in lines and columns. Since each line represents a single block device, the columns of this line contains the information about the device. This enables the usage of iterations in python, where each column can be splitted on the separators, which is multiple spaces.

The data can therefor be represented as a dictionary in Python. This is a key:value based data structure. A single block device can therefor have a single dictionary with each value of the /proc/diskstats file beeing represented as column:data. Each column get its name from the documentation for /proc/diskstats shown on 3.2.2 on the preceding page.

3.3.2 Storing the data

To be able to analyze the results at a later point, the output of the tool should be stored to file. One of the criteria for storing the data to file would be the possibility of using other tools to analyze the data at a later time. Wether the tool is written in Python, Perl or any other language should not matter. The most common possibilities for storing to file would be *json*, *xml* or *csv*. Both *json* and *xml* uses a key:value representation, but also makes it harder to format the data without the usage of external libraries. There are also numerous ways of interpreting these different formats, as there are multiple standards. The choice is therefor to use *csv* which is a simple text format which uses the comma notation to separate each value. The top of the file should contain the description of each column, and each field should be represented with either a comma or 0 if there is no data.

If we takes the previous examples given in the background 3.2.2 on the previous page, the final file will look something like the following:

Listing 3.2: Possible output of monitoring tool

```
date, major number,minor mumber,device name,reads completed successfully,reads merged,sectors read ...
date, 253,0,vda,31410,806,1020922,2564596768,1465561,1382324,54703840,3579730636,0,7229708,40955272
date, 253,1,vda1,31182,334,1015322,2564547860,1465459,1382324,54703024,3579730152,0,7229220,40954784
```

Including the output from /proc/diskstats, the time stamp of when the test has been run is important to include, as this will provide the time interval between the two following tests to be calculated. The collected data could with this format be imported into any scripting language or applications like libreOffice for further work.

3.3.3 Different command line options

As described in the beginning of this section, the tool should be able to be run in two different modes, the user-mode and cron-mode. The tool does not need to be implemented any specific way for it to work in the different modes, other than actually taking some different command line options. The needed options is to specify if the application should run the gatherings in a loop or not. This is needed only in user-mode and can then take the parameter as for how long the tool should sleep between each data gathering. This is not need in cron-mode as this is specified in the cron job itself.

The other needed option is to specify which devices that the tool should gather information on. By default the tool should store information about all the devices, but as a command line option it should also be possible to specify a list of devices to focus on.

3.4 Result

This section will present the actual tool that has been developed. It has been developed in Python, and does not require any additional modules to work. This enables it to work at any distribution, with basic Python installed.

Based on the findings in the approach, the tool has been implemented with the following command line options, which were described in 3.3.3 on the preceding page. This different options is visible with the use of *-h* or *–help* when running the tool.

Listing 3.3: Command line options of disksts.py

```
$ python diskstats.py -h
usage: diskstats.py [-h] [-l sleep] [-d name [name ...]]

Parser tool for /proc/diskstats

optional arguments:
  -h, --help            show this help message and exit
  -l sleep, --loop sleep
                        Runs the program in a loop. Takes the looptime as
                        option.
  -d name [name ...], --device name [name ...]
                        Names, like sda, sda1...
```

The tool can therefore be run with the intention of running every second and only look at the information for the block device vda. The command for this would be the following.

Listing 3.4: Execution of disksts.py

```
$ python diskstats.py -l 1 -d vda
Gathering data for ['vda']
Gathering data for ['vda']
```

When running with this options, a file with the name *mon_vda.csv* would be created. The contents of the file would be in csv format, and formated as described in the approach.

Listing 3.5: Gathered data stored in file from diskstst.py

```
datetime,major_number,minor_number,device_name,read_completed_successfully,reads_merged,sectors_read,
    time_spent_reading(ms),writes_completed,writes_merged,sectors_written,time_spent_writing_(ms),
    IO_currently_in_progress,time_spent_doing_IO_(ms),weighted_time_spent_doing_IO
2014-12-14T14:27:07.878920,253,0,vda
    ,5964436,13925,540485938,2768828,1289917,1519024,48384848,9417637,0,1382484,12180809
2014-12-14T14:27:08.881174,253,0,vda
    ,5964436,13925,540485938,2768828,1289934,1519049,48385184,9417668,0,1382487,12180840
```

As described it should also be possible to run the script as a cron job.

Listing 3.6: Cronjob example

```
# Diskstats cronjob
# This cronjob would execute every minute, and capture only the information of vda
*/1 * * * * /usr/bin/python /path/to/diskstats.py -d vda
```

The final version of the tool is attached in the appendix 3.7 on the next page. Most of the logic that has been implemented is related to the different command line options, as the tool needs different

3.5 Analysis

The aim of this tool were to be able to collect the information that was possible to be gathered from the /proc/diskstats file. It is able to collect the data, and store it as a csv file, which is a readable format that is supported by many different softwares. The script does also implement many different function beyond what is outlined in the initial description of the needed tool, as the possibility for it being run as a cron job, and with variable sleeping lengths.

However there are one issue that has been discovered. When running the tool as a cron job, there is no way to specify the path to where the file should be stored. The output of the tool is therefor lost.

This has been fixed, by adding another command line option, *–outfilepath* or *-o*.

3.6 Discussion and conclusion

This tool is intended to enable the user to store the contents of the file /proc/diskstats. This tool enables the gathering of the information for later analysis, by storing all the specified content of the file. By default the application runs in a simple manner, but also includes aditional functionality, making it possible to use the tool in many different situations where debugging of file system is important.

This tool does the same thing as many other tools available, but at a lower level. Most of the tools available, will do some calculations on the data, but this tool only gathers the data for later analysis.

This is in many cases a benefit. This will also make less of an impact on the general performance of the system. This said, the performance impact of other tools have not been tested here.

3.6.1 Conclusion

In summary, the task has been completed to the specified requirements by solving the following main points.

- Collecting the information from */proc/diskstats*
- Storing the data in a orderly manner for later analysis

This conclude a simple tool that can capture the data from the kernel file system for later analysis. The tool work as intended, and can capture the content of the file over longer periods of time.

3.7 Appendix

Listing 3.7: diskstats.py

```
#!/usr/bin/env python
# −∗− coding: utf−8 −∗−
from datetime import datetime
import os, re, csv
import argparse
from time import sleep

# Tool for getting /proc/diskstats
DISKSTATS_PATH = '/proc/diskstats'
HEADERS = ['datetime', 'major_number', 'minor_number', 'device_name', 'read_completed_successfully',
    'reads_merged', 'sectors_read', 'time_spent_reading(ms)', 'writes_completed',
    'writes_merged', 'sectors_written', 'time_spent_writing_(ms)', 'IO_currently_in_progress',
    'time_spent_doing_IO_(ms)', 'weighted_time_spent_doing_IO']

lastread_timestamp = None

def read_diskstats():
  global lastread_timestamp
  devs = []
  with open(DISKSTATS_PATH, 'r') as f:
    for line in f.readlines():
      # check if line is disk
      devs.append(dict(zip(HEADERS[1:], re.split('\s+', line.strip()))))
  lastread_timestamp = datetime.now()

  return devs

def find_device(data, device):
  return (item for item in data if item['device_name'] == device).next()

def write_diskstats(data, device=None, path=""):
  global lastread_timestamp
  csv_data = []

  if device:
    if len(device) > 1:
```

```
37          filename = "{}mon_diskstats_{}".format(path, "_".join(device))
38          if not os.path.isfile(filename):
39              csv_data = [HEADERS]
40          d = []
41          for dev in device:
42              d.append(find_device(data, dev))
43
44          if len(d) < 1:
45              print "No devices found by filter {}".format(" ".join(device))
46              return False
47          else:
48              data = d
49
50              for dev in data:
51                  tmp = [dev[h] for h in HEADERS[1:]]
52                  tmp.insert(0, lastread_timestamp.isoformat())
53                  csv_data.append(tmp)
54
55      else:
56          data = find_device(data, device[0])
57          filename = "{}mon_{}".format(path, device[0].strip())
58          if not os.path.isfile(filename):
59              csv_data = [HEADERS]
60          tmp = [data[h] for h in HEADERS[1:]]
61          tmp.insert(0, lastread_timestamp.isoformat())
62          csv_data.append(tmp)
63
64  else:
65      filename = "{}mon_diskstats".format(path)
66      if not os.path.isfile(filename):
67          csv_data = [HEADERS]
68      for dev in data:
69          tmp = [dev[h] for h in HEADERS[1:]]
70          tmp.insert(0, lastread_timestamp.isoformat())
71          csv_data.append(tmp)
72
73  with open(filename, 'a') as f:
74      w = csv.DictWriter(f, HEADERS)
75      w.writer.writerows(csv_data)
76
77  def gather_and_write(devices):
78      data = read_diskstats()
79      write_diskstats(data, devices)
80      return data
81
82  def main():
83      parser = argparse.ArgumentParser(description='Parser tool for /proc/diskstats')
84      parser.add_argument('-l', '--loop', type=int, metavar='sleep',
85          help='Runs the program in a loop. Takes the looptime as option.')
86      parser.add_argument('-d', '--device', metavar='name', nargs='+', default=None, help='Names, like sda, sda1...')
87      parser.add_argument('-o', '--outdir', metavar='outdir', default=None, help='Path to store files in. End with /')
88      args = parser.parse_args()
89
90      try:
91          if args.loop:
92              while True:
93                  print "Gathering data for {}".format(args.device)
94                  data = gather_and_write(args.device)
95                  sleep(args.loop)
96          else:
97              data = gather_and_write(args.device)
98              print find_device(data, 'vda')
```

```
99      except KeyboardInterrupt:
100         print "Aborting..."
101
102 if __name__ == '__main__':
103     main()
```

References

APSIS. (n.d.). *Pound - reverse-proxy and load-balancer.* Retrieved from **http://www.apsis.ch/pound/**

Berners-Lee, T. (1992). *Hypertext transfer protocol.* Retrieved from **http://www.w3.org/History/19921103-hypertext/hypertext/WWW/Protocols/HTTP.html**

HAProxy. (2014a). *Configuration manual.* Retrieved from **www.cbonte.github.io/haproxy-dconv/configuration-1.5.html**

HAProxy. (2014b). *Haproxy description.* Retrieved from **http://www.haproxy.org/#desc**

HAProxy. (2014c). *They use it.* Retrieved from **www.haproxy.org/they-use-it.html**

Haugan, L. (2014). *Pull request of httperfpy changes.* Retrieved from **https://github.com/jmervine/httperfpy/pull/4**

Joshua Mervine, j. (2014). *Httperfpy.* Retrieved from **https://github.com/jmervine/httperfpy**

OpenStack. (2014a). *Documentation nova/apifeaturecomparison.* Retrieved from **https://wiki.openstack.org/wiki/Nova/APIFeatureComparison**

OpenStack. (2014b). *Openstack frontpage.* Retrieved from **http://www.openstack.org/community/**

OpenStack. (2014c). *Openstack user stories.* Retrieved from **http://www.openstack.org/user-stories/**

OpenStack. (2014d). *Python bindings to the openstack nova api.* Retrieved from **http://docs.openstack.org/developer/python-novaclient/**

RedHat. (n.d.). *Load balancing.* Retrieved from **https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Load_Balancer_Administration/ch-lvs-overview-VSA.html**

ricklind@us.ibm.com. (n.d.). *I/o statistics fields.* Retrieved from **https://www.kernel.org/doc/Documentation/iostats.txt**

TLDP. (n.d.). *Linux filesystem hierarchy: /proc.* Retrieved from **http://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html**