UiO : University of Oslo

A Dynamically Scaled Cloud-based Web-Service using Docker-Swarm, HAProxy and OpenStack

by

Samiul Saki Chowdhury (589795)
HiOA: (S316611)

Supervisor
Kyrre Begnum

Service Management and Developer Operations (MS019A / INF4019NSA)

Network and System Administration (Department of Informatics)
Faculty of Mathematics and Natural Sciences
University of Oslo
Norway

Oslo
August 2, 2017

Abstract

In this report, a cluster of cloud-based web-service is built in order to adjust the number of Docker-based web-servers (containers) in order to manage the incoming rate of user requests. The main focus of this cloud-based web-services is to prevent service depletion due to heavy traffic from outside network and automate the nodes to generate new services to adapt the requests approaching for accessing those web-services. To achieve this infrastructure, several tools such as Docker-Swarm, Docker-Machine, HAProxy have been used. Additionally, Grafana, InfluxDB and cAdvisor have been used to monitor the traffic pattern as well as CPU, memory and file-system usage of the nodes. Upon successful testing with the ApacheBench to exhaust the traffic handling capability of the load-balancer, the services are auto-scaled in order to adjust the concurrent connection requests and to stabilize the network overload.

Contents

List of Figures

Chapter 1

Introduction

In this project, we are given the task to create a dynamically scaling cloud-based infrastructure to automate the number of web-services based on the incoming requests towards the web-servers. In the cloud computing business, the number of interest on having an infrastructure on a cloud-based servers are ascending. Having the cloud-based platform give the opportunities to the users/developers to work on a hardware constraint platform. Although the cost of devices are less paying for uptime of the virtual machines (VMs) are still a big concern. The demand for having a cluster of VMs which can be deployed in a second and can be monitored and scaled to certain requirements are demanding among the users. The servers designed to work as services are in need to maintained in order to reduce cost and avoid saturation. In this project, a cloud-based web service need to be create which can regulate the number of docker-based web-servers on OpenStack, based on the incoming rate of user requests. This recalls the scaling of the web-services to adapt to the user demand.

To achieve such framework, a number of requirements we need to consider in this project:

1. A load-balancer is required in order to distribute the payload (user requests in this case) among the all web-server.
2. Based on the user request to access the web-services the number of nodes/web-servers need to be adjusted.
3. A scripted approach can be utilised to adjust these required number of web-services. The information can be harvested from the load-balancer itself.
4. A monitoring service have to be in place for observing the traffic pattern and alert the developers based on the conditions given.
5. Finally a numerical test needed to be conducted to demonstrate the increasing number of servers based on the increasing numbers of incoming requests have been made.

1.1 Problem Statement

The number of server can be spawned or deleted based on requests in a simple and easy manner. But taking into account that the uptime for a VM in cloud environment it is a complex method to avoid creating numerous VMs for a few number of web-requests. The technique to utilise most of a single VM using simple tools is a difficult process and the design for such infrastructure need to be carefully outlined. The problem arise when the increasing number of requests demands new services. Additionally, the services/servers also have to be removed once the connection request rate goes down or the web-services are no longer needed. The latter is to ensure to avoid server saturation. A simple but effective solution is required to solve this issue.

1.2 Project Objectives

Our task in this project is to create a framework where a load-balancer can successfully distribute the user requests among all the the servers and return the reply messages to the requester machines. The main focus of this project is the scaling. The main goal is to having a setup where the web-services can be created and removed based on the ongoing traffic pattern between servers/nodes and the load-balancer. We need to create a cluster of docker-machine nodes as web-servers which have docker-swarm cluster in place in order to minimized the space requirements for such design and use HAProxy to use as load-balancer. Monitoring tools such as Grafana along with InfluxDB and cAdvisor is to be deployed to observe the traffic pattern and to take action based on the conditions. Apache AutoBench is used to exhaust the web-servers by imitating the web requests in real-life platform. Finally, a scripted approach is adapted which will run and observe the incoming connection rate information yielded by HAProxy and add/remove docker web-services in the swarm accordingly.

1.3 Report Outline

Rest of this report is organised as follows:

- In Chapter 2, the necessary features of the referred tools and enabling technologies which are adapted to this project are briefly discussed.
- In Chapter 3, the infrastructure of the project design is described and framework setup is elaborated in order to achieve the project goals.
- In Chapter 4, the implementation of technical design is thoroughly followed up using flow diagrams to give the readers an extensive idea behind the design principle. The proposed algorithm applied to the cluster is also explained thoroughly in this chapter.
- In Chapter 5, the overall performance results of the testing of project design is discussed along with potential usage of the framework.
- And finally, Chapter 6 outlines the conclusion of this project.

Chapter 2

Background

The tools that used to deploy a well-structured server varies on user requirements. In order to achieve and reach the desired goal a system administrator need to accommodate different tools/-software for individual tasks. In the modern day web framework and configuration management, developers tend to choose the right device that can provide them to not only to deploy a service in the cloud but also maintain and monitor the services in order to accomplish a stable foundation The stability of the foundation depends on the quality and effectively of the tools that have been utilised. To avoid large project deployment and bulky resource consuming systems, the best tools that fits well for the projects are chosen. The fundamental of web-service is to making sure the services never get depreciated. Additionally, the developers need to confirm that the servers are to be monitored in regular manner and if needed it would automate the healing process itself from the situation that can cause the services to be downgraded. Most importantly, it is mostly required to run an autonomous system which is self-sufficient to response to complex circumstances and can be configured can be maintained without help of any human interaction. This type of framework are getting more famous among companies and businesses which provide services based on their clients' requests. The number of requests for services are mostly based upon the number of clients and their service subscriptions. The unique value proposition of Cloud Computing creates new opportunities to align IT and business goals. Cloud computing is essentially a powerful computing paradigm in which tasks are assigned to a combination of connections, software and services accessed over a network. The vast processing power of Cloud Computing is made possible though distributed, large-scale computing clusters, often in concert with server virtualisation software and parallel processing. Several different types of mechanism and appliances are used by the vast majority of the service providers and operations management teams. In this Chapter, the tools to implement our cloud-based web-servers are discussed.

2.1 Dynamic Scaling Cloud

Scalability is critical to the success of many enterprises currently involved in doing business on the web and in providing information that may vary drastically from one time to another. Cloud computing provides a powerful computing model that allows users to access resources on-demand. Maintaining sufficient resources just to meet peak requirements can be costly. A dynamic scaling algorithm for automated provisioning of VMs resources based on threshold number of active session is needed in this case. Capabilities on-demand for the cloud services to provision and allocate resources dynamically to the users are one of the main goal of the dynamic scaling cloud. In this section, the compelling benefits of the Cloud which can handle sudden load surges, deliver IT-resources to its clients and maintain its standard of high resource utilization is discussed.

There is some of the key features need to be considerate while creating a dynamically scaled cloud cluster:

- Auto-scaling is needed to be included to scale dynamically. How to scale in response to changing demand need to be set carefully. If the web applications runs in certain numbers of instances a new or some few instances need to spawned as soon as the load on the current instances rises over certain threshold. Then again the number of instances needed to decreased down when the load level goes down under some threshold. The level of thresholds will vary on user specification, i.e., service providers' advertised policies.
- A alarming monitor can be useful when setting up dynamic scaling cloud which in general will used to alert and trigger a certain scaling policy and associated with the scaling group. An alert is an object that watches over a single metric (e.g. current session connection rate of the instances) over a period of time. When the value of the metric reach over a certain threshold (that is defined) in the specific time interval, the alert will perform some actions in order to scale in or scale out the number of web-servers (instances).
- Different types of applications requires different resources. Application requires different scaling for different tiers (some heavy on storage and some on computing). Certain application can be beneficial from scaling up when other apps can use a good scaling down algorithm. A well-balanced scaling algorithm need to be in place. Application built for scale out can add new servers within a few minutes and make them the part of the cluster. On the other hand, scale in approach takes advantage of performance and efficiencies inherent power systems that allows execution of unexpected and dynamic workload with linear performance gains while at the same time the efficient use of server capacity is uncompromised

2.1.1 Benefits of dynamic auto scaling clouds

Some benefits of auto scaling clouds are mentioned below:

- In a application architecture adding auto scaling feature can maximize the benefits of cloud systems.
- Scaling can launch instances (if needed on demand) to support the infrastructure stability.
- Better fault tolerance can be achieved by detecting when an instance is unhealthy.
- A better cost management is established when auto scaling dynamically increase and decrease capacity as needed. Since customers need to pay the by the uptime, the appropriate number of VMs as necessary can reduce the uptime cost.
- Computing at the scale of the Cloud allows users to access supercomputer-level computing power which gives the users access to the enormous and elastic resources whenever they need them. For the same reason, cloud computing is also described as on-demand computing.

The call for setting up a good framework for dynamic auto-scaled cloud cluster a number of tools need to be perfectly coordinated. In the next section, the tools utilised for this project are briefly described:

2.2 Docker

Docker is an open-source technology that is vastly embraced by proprietary software companies such as Microsoft and mostly UNIX systems. The most important reason behind using docker as has lower system requirements than of VM hypervisors, such as Hyper-V, KVM and Xen. Containers spawned in docker use shared operating systems, that means they are much more efficient than hypervisors when in comes the term for system resources. Containers rest on top of the Linux instances instead of virtualising hardware which means it is possible to leave behind all the bulky VM junks and just with a small capsule containers with a distinct application/s. Therefore with a perfectly tuned docker container system anyone can have as many as 4-6 times of the number of server applications instances than hypervisors. Although, docker is a new name in the developing field but spawning containers is quite old idea. This idea dates back at least year 2000. Other companies such as Oracle Solaris also has a similar concept called Zones but companies like Docker and Google concentrate more on open-source projects as OpenVZ and LXC (Linux Containers) to make it work better and more secured.

Docker is built on top of LXC and has its own file system, storage, CPU, RAM and so on. The key difference between containers and VMs is that the hypervisors abstracts an entire device while containers only abstract the operating system kernels. Hypervisors can use different operating systems and kernel such as Microsoft Azure can run on both Windows Server and SUSE Linux Enterprise Server but with the Docker, all the containers must use the same operating system and kernel. On the other hand, Docker containers can be used to get most server application instances running on the least amount of hardware. This approach can save a large amount of resources for the cloud providers or data centres. Using Docker the deployment of containers become easier and safer. Developers around the globe are using Docker to pack, ship and run any application as portable, self sufficient, lightweight containers in virtually anywhere. Docker containers are easy to deploy in the cloud. It has been designed to incorporate into most DevOps applications, including puppet, Chef, Ansible or just on its own to manage development environments. It makes easy to create and run ready containers with applications and it makes managing and deploying application much easier.

2.2.1 Benefits of using Docker

In the IT world running application instead of bulky virtual machines is a fast gaining momentum. The technology is one of the fastest growing in recent history due to its adaptation in industry level along with software vendors. Docker as a company and its software have grown immensely in technology field due to its usability.

Simple and fast configurations

One of the key benefit of Docker is the way it simplifies matter. VMs are allowing the users to run any platform with its own configurations on top of the infrastructure of the user where Docker has concentrated on the same benefit except reducing the overhead of a VM. An user can take their own configuration, put in into codes and deploy it in quick and easy manner. Docker containers can be used in wide variety of environments since the infrastructure requirements are no longer an issue for the environment of the application.

Increased productivity

Two major goals arises when when it comes to working in a developer environment, they are the bringing of the product as close to production as possible. It is done by running all the services on its own VM to show the application functionality However, any overhead needed to be avoid when compiling the application. The second goal is to make the development environment as fast as possible for interactive use. Receiving the feedback after tests is important in production level. Docker shows its functionality by not adding to the memory footprint and by allowing much more services at the same time.

Rapid deployment

The appearance of VMs took bringing up the hardware down to minutes. However, Docker manages to reduce this deployment time to mere seconds. The reason behind is that Docker containers creates every process but does not boot the OS. The cost of bringing up the system is next to null while creating and destroying the data in the Docker containers. The resources can be allocated in more aggressive manner with containers instances.

2.2.2 Docker-Machine

Docker machine is the tool that enable installation of Docker-Engine on a virtual hosts to manage the hosts with docker-machine commands. Using this tool Docker hosts can be created in the various platforms such as local machines, virtual box or even on cloud provider such as OpenStack. Using the docker-machine commands an instance can be started, inspected, stopped, restarted. It can also be possible to upgrade the Docker clients and daemon using this tool. Using the docker-machine env (environment) commands the developers can load a Docker host (a different instance even) as default and make configuration changes on that particular host. Different other apps and services can be created, deployed and maintained though docker-machine without having to accessing the Docker clients individually. This makes the developer to use a single master server (instance) to control or manage the entire network in the cluster of Docker hosts. Although Docker Engine runs on natively on all Linux systems, Docker Machine enables users to provision multiple remote Docker hosts on a network, in the cloud or even locally. The main difference between Docker Engine and Machine is that typically for the client-server application Engines (or commonly known as just Docker) are Docker daemon, a REST API that specifies interfaces for interacting with the daemon and a command line interface (CLI) client that can talk to the daemon (through the REST API wrapper). On the other hand, Docker Machine is a tool for provisioning and managing your Dockerized hosts (with Docker Engine on them) [1]. Fig. 2.1 gives a brief taste of how docker-machine works in principal.

2.2.3 Docker-Swarm

Docker-Swarm is the Docker cluster management and orchestration features which is embedded in Docker-Engine. It is built using Swarm-Kit A swarm mode can be enabled by either initialising swarm or joining an existing swarm. It is a collection of nodes with Docker-Engines which is used to deploy services. Using swarm mode in Docker containers, services are setup and those swarm services and stand-alone containers can be run on the same Docker instances. A swarm node is an instance of the Docker Engine participating in the swarm. One or more nodes can be run on a single physical computer or cloud server, but production swarm deployments typically include Docker nodes distributed across multiple cloud machines. Usually, to deploy a application

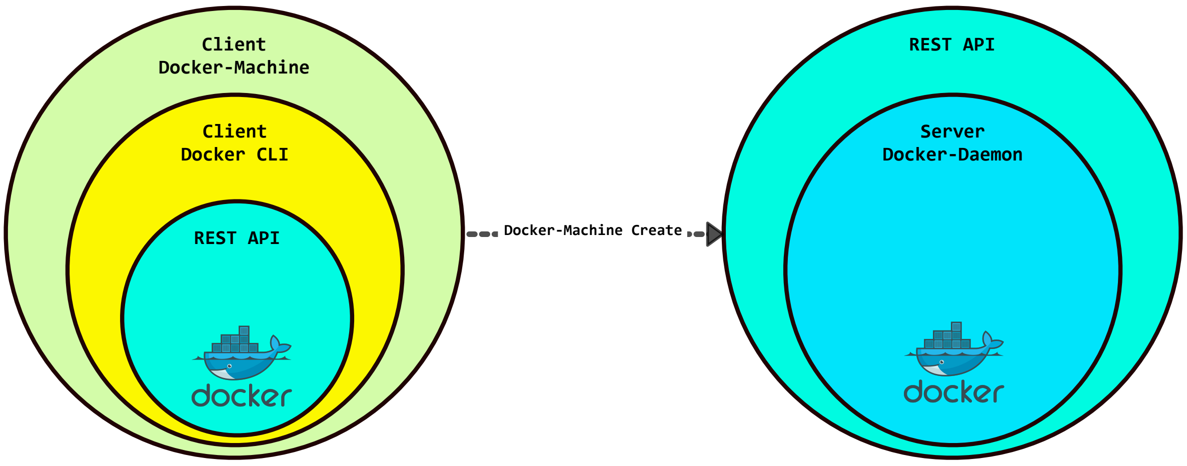


Figure 2.1: Illustration of Docker-Machine Mechanism [1]

a service definition need to be submitted to a manager node. The manager nodes then dispatches unit of work called tasks to worker nodes. Manager nodes also performs the orchestration and cluster management functions required to maintain the desired state of the swarm where the tasks are conducted by an elected single leader. Worker nodes executes received tasks dispatched from manager nodes. An agent runs on each worker nodes and reports on the tasks assigned to it. The worker nodes then notifies the manager node of the current state so that the swarm-manager can maintain the desired state of each worker. However, services can be deployed to manager only as well.

A service is the definition of the tasks to execute on the worker nodes. It is the central structure of the swarm system and the primary source of user interaction with the swarm. Upon creating services the developers can choose which container to use and which commands to be executed. In the replicated services model, the swarm manager distributes a specific number of replica tasks among all the node based upon the required scaling goal. A service as a task runs on Docker container and the commands runs inside the containers. A scheduling unit activates using swarm where manager node assign tasks to worker nodes according to the number of replicas set in the service scale. The swarm manager initialize an ingress load-balancing to expose the services that is required to make available externally to the swarm. All nodes is a swarm participate in an ingress routing mesh. The routing mesh enables each node in the swarm to accept connections on published ports for any service running in the swarm, even if there is no task running on the node. The algorithm routes all incoming requests to published ports on available nodes to an active container. As a swarm-manager the node can automatically (can be chosen manually) publish ports for the services. The automatically chosen default port range is between (30000 - 32767). Swarm mode has an internal DNS component that automatically assigns each service in the swarm a DNS entry. Swarm manager uses internal load balancing to distribute requests among services within the cluster based upon the DNS name of the service [2]. A quick overview of the Docker swarm load-balancing principal can be shown in the Fig. 2.2.

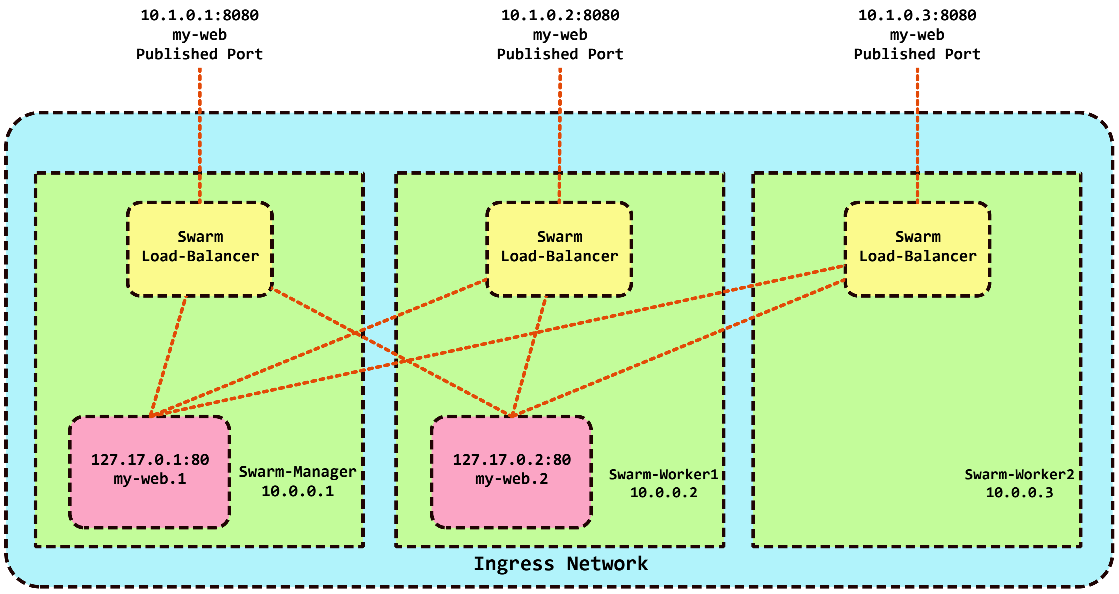


Figure 2.2: Illustration of Docker Swarm Load-Balancing Algorithm [2]

2.3 HAProxy

HAProxy is single-threaded, event driven, non-blocking engine combining a very fast I/O later with a priority based scheduler. The architecture is designed to move data as quickly as possible with the least possible operation keeping in the forwarding of data in mind. Data does not reach the higher level in OS model while performing offering a bypass mechanism at each level. HAProxy let the kernel do that most processing work and avoid certain operation when it guesses they could be grouped later. Typically 15% of the processing time spent in HAProxy versus 85% in the kernel TCP or HTTP close mode. Also the HTTP keep-alive mode is about 30% for HAProxy versus 70% for the kernel [3]. A single process in HAProxy can run as much as 300000 distinct proxies which requires only one process for all the instances. It is also possible to run it over multiple processes. Usually this tool scale very well for HTTP keep-alive mode but the performance that can be achieved out of single process generally outperforms common needs by an order of magnitude. HAProxy only requires the haproxy executables and a configuration file to run which makes it easy to use for the service providers. a syslog daemon need to accurately configured for logging services and rotation services for logs. The configuration files are parsed before stating and HAProxy tries to bind all the listening sockets. If anything fails then it refuses to start. The run-times failure is next to none once HAProxy accepts to start.

HAProxy is an open source software used for TCP/HTTP load balancing. It runs on Linux, FreeBSD, and Solaris. It is famous and widely used for its ability to keeps servers up by distributing the load across multiple servers. Among many terminologies used in HAProxy Access Control List (ACL) contains a set of rules which has to be checked so that it can carry out some predefined actions which can be blocking some requests or selecting the server to forward the request based on the conditions. ACL rules applies to all the incoming traffic which increase the flexibility of the traffic forwarding based on different factors such as some connection to the back end and pattern matching. Back end is the group of servers that have been established for load-balancers. It contains the IP addresses of the servers along with the port numbers)(if necessary) and chooses the load-balancing algorithm for efficient processing of the web-server access requests. Another attribute of HAProxy is front end where the configuration define how the requests have to be forwarded to the back end servers with ACL. The definition of front end contains IP addresses and port numbers of the servers as well. HAProxy also have a health check-up feature which in simple way checks the back end servers' health. It is carried out by simply sending a TCP request and find whether server listens to the specific ports and IP. Upon no response the load-balancer can fire up a forwarding request to another server which is healthy. The unhealthy back end server does not get any further request until it checks up as healthy server again. At Least, one server should be healthy to process the request.

Among many load-balancing algorithms in HAProxy, the commonly used algorithm is round-robin algorithm. The algorithm chooses the server sequentially in the list. Once it reaches the end of the server list, the algorithm forwards the next request to the first server in the list again. The weighted round-robin algorithm uses the weight allocation to the server to forward the request while dynamic round-robin algorithm uses the real-time updated weight list of the server. The least connection algorithm is also being used in HAProxy which selects the server with few active transactions and then forwards the user requests to the back end. The source algorithm also selects the server based on source IP addresses using the hash to connect to the machinating server. Server overloading can be reduced using high availability algorithm which gets activated when the primary (active) load-balancer gets overloaded. It fires up the secondary (passive) load-balancer if the primary load-balancer fails. In this project, a primary load-balancer is used with round-robin algorithm in order to forward all the incoming request to the back end server [4].

```
   192.168.1.1       192.168.1.11 - 192.168.1.14      192.168.1.2
-------+----------+-----------+----------+----------+--------+----
       |          |           |          |          |       _|_db
    +--+--+     +-+-+       +-+-+      +-+-+      +-+-+    (___)
    | LB1 |     | A |       | B |      | C |      | D |    (___)
    +-----+     +---+       +---+      +---+      +---+    (___)
    HAProxy               Back End Web Servers

                        A Simple HAPoxy Setup
```

Along with other important tools mentioned above several tools are used for monitoring and testing purposes. The following tools are also utilised to setup the dynamic scaled cloud-cluster with docker services.

2.4 Grafana

Grafana allows you to query, visualize, alert on and understand your metrics no matter where they are stored. Create, explore, and share dashboards with your team and foster a data driven culture. It is a beautiful dashboard for displaying various graphite metrics through a web browser. Grafana is simple in use and easy to setup and maintain. It is almost like Kibana display style but with many other features included. The Grafana project started at the beginning of the 2014 and since then it has shown an enormous amount of potential usage and functionality One of the nicest feature of this tool is that the behind scenes for details or intricacies of how all the graphite components works together are not needed to be worried about. This tool give abilities to bolt all kids of alerting for the graphs which are based on various famous databases.

2.5 InfluxDB

InfluxDB is a time-series database built on LevelDB. It is designed to support horizontal as well as vertical scaling and it written in the language Go. InfluxDB was very easy to set up and fully integrated with Grafana. It has an end-to-end platform which is ready to be deployed on the cloud or via download. The tool is free of external dependencies, yet it can open a complex deployment which is flexible enough to use. InfluxDB gives visibility with real-time access which helps to find certain value in the data in quick fashion. Identity pattern, control systems, turning insight into action even predicting the future of a node can be implied with this useful tool. It uses a powerful engine which is fast and meets the demands of even the largest monitoring deployments.With the native clustering functionality InfluxDB offers high availability while eliminating single points of failure and simple scale out.

2.6 cAdvisor

cAdvisor (Container Advisor) provides container users an understanding of the resource usage and performance characteristics of their running containers. It is a running daemon that collects, aggregates, processes, and exports information about running containers. Specifically, for each container it keeps resource isolation parameters, historical resource usage, histograms of complete historical resource usage and network statistics. This data is exported by container and machine-wide. cAdvisor has native support for Docker containers and should support just about any other container type out of the box.

2.7 ApacheBench

Load testing is a good idea before any deployment. It's nice to quickly establish a best-case scenario for a project before running more detailed tests down the road. The ApacheBench tool (ab) can load test servers by sending an arbitrary number of concurrent requests. This especially shows how many requests per second the Apache installation is capable of serving. This is a tool for benchmarking any Apache HTTP server. It is designed to give an impression of how the current Apache installation performs. AB is a single-threaded command line computer program comes bundled with the standard Apache source distribution, and like the Apache web server itself, is free, open source software and distributed under the terms of the Apache License. ApacheBench will only use one operating system thread regardless of the concurrency level (specified by the -c parameter). In some cases, especially when benchmarking high-capacity servers, a single instance of ApacheBench can itself be a bottleneck. When using ApacheBench on hardware with multiple processor cores, additional instances of ApacheBench may be used in parallel to fully saturate more the target URL.

In the next Chapter, the technical design needed to meet this project goal which is to build a framework with cluster of instances in the cloud (Docker swarm) with dynamic auto-scaling feature enabled to handle the number of web-service requests.

Chapter 3

Technical Design

The main focus of the technical design in this report are on the design principle and model to achieve a cloud-based Dockerized swarm to deploy auto-scaled web services dynamically. The design include several tools and their features such as Docker-Machine, Docker-Swarm, HAProxy, InfluxDB, cAdvisor and Grafana. There is other alternatives tools exists with which similar framework can be deployed. As it is discussed earlier in Chapter 2, the benefits of adapted tools have a flexible and rudimentary factor which have high recommendation. It is a easy to setup architecture that can be applied to any set of node in a network. The following sections explains the design principle of the dynamic scaled cloud:

3.1 Design Principle

To start with the setup process individual steps have been carried to achieve the project goal. These steps are briefly descried in following:

- A server has been created in ALTO (**OpenStack**) cloud in order to deploy the Docker nodes, setup Docker-Swarm among those nodes, running services and managing them from one single point of control box. This server is selected as Master server and selected **Automata** as the hostname of the machine. A floating IP will be associated to this server.
- Docker, Docker Machine and Docker Compose will be installed in Automata server.
- Using the Docker machine commands three Docker-Machines will be created on ALTO. The machines will be identified as **Swarm-Manager**, **Swarm-Worker1** and **Swarm-Worker2**. Docker will be pre-installed in all these three machines.
- A Docker-Swarm will be created using the above three machines and the Swarm-Manager will be chosen as the swarm-leader. Additionally, Swarm-Worker1 and Swarm-Worker2 will be joining the same swam to make a cluster.
- In the Automata server monitoring tools **Grafana**, **InfluxDB** and **cAdvisor** will be deployed as Docker-services.
- On the Swarm-Manager all the monitoring services will run but in the Swarm-Worker nodes only the cAdvisor will be running to collect container metrics. This metrics will be collected and viewed as graph in a Grafana which can also be used to alert other services to response.
- A load-balancer (**HAProxy**) will be running on master server.

- A **httpd** service will be running in the swarm as a web service. The services will be deployed to handle the incoming web request for other servers. This web-service is scalable on request demand.
- A bash script will be running on the master to check the current session for connection rate metrics collected from HAProxy socket. This script makes the services scalable.
- If the incoming request to the web-services rises above certain threshold Swarm-Master would scale-up to generate more web-services.
- If the incoming requests goes down under certain threshold then it Swarm-Master would scale in to reduce the number of web-services.
- Additionally, a Docker **Visualiser** container will be setup (on Swarm-Manager) to visualize the services running on the Docker-Swarm.
- Finally, **ApacheBench** will be used to set a test-bench for the swarm's auto-scaling capabilities on the cloud. The results will be collected and measure the performance of the scaled cloud cluster.

3.2 Design Model

The model (Fig. 3.1) explains the principle behind implemented technical design of this project.

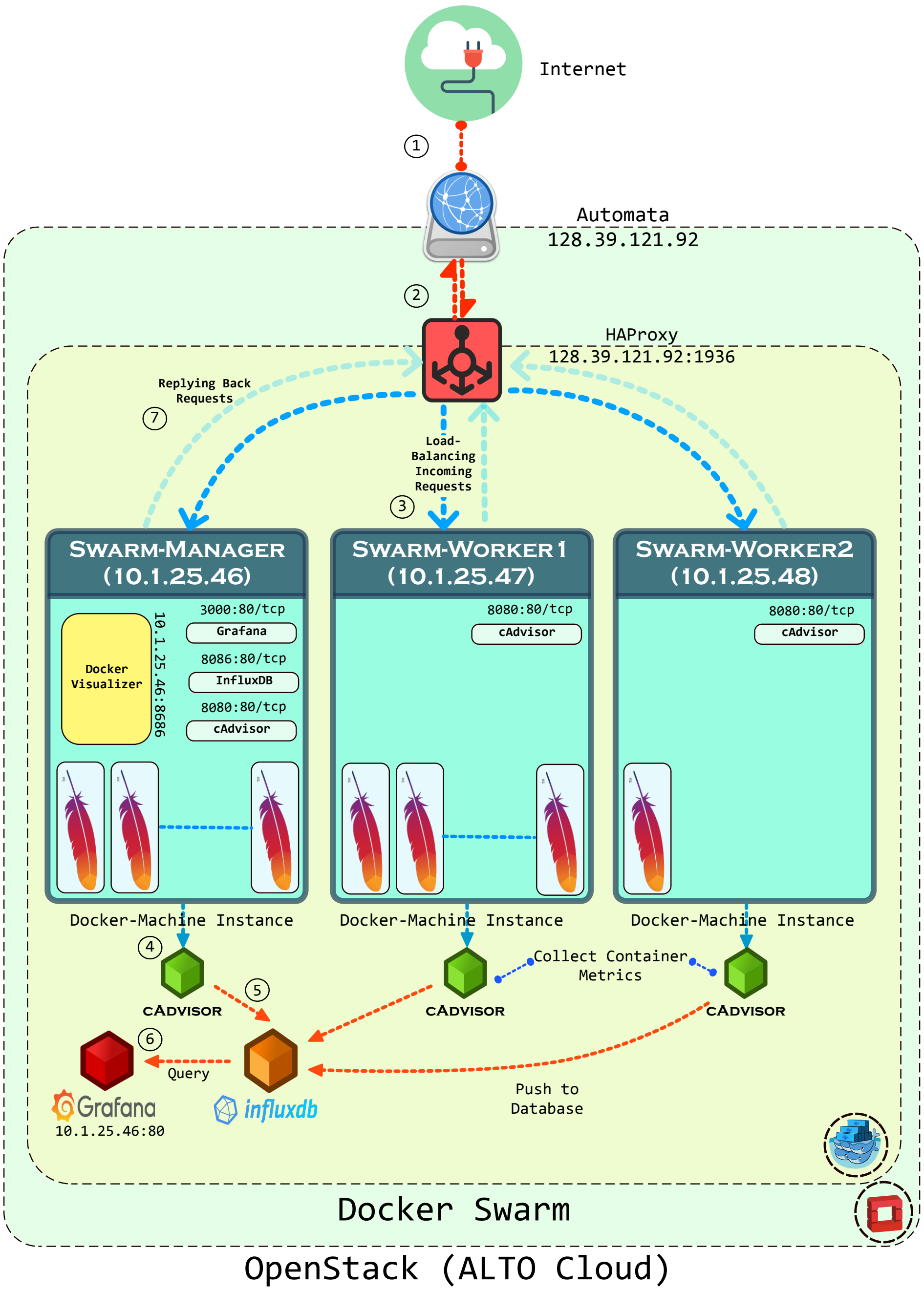


Figure 3.1: Illustration of the Technical Design of the Dynamic Scaling Cloud Framework

The above diagram represents the design principal for a network dynamically scaled Docker cloud framework for web-services. The following section explains the software integration process that has been adapted to fit the design principal in this project.

3.3 Design Architecture

This section is integrated into several subprocesses in order to cover implementation in different part of the project. This part will give the readers thorough perception on the design setup and the adjustment done on the tools used to attain the project goals.

Server setup

In the Alto cloud a server is created as instance in order to setup the tools for the technical implementation. The instance name is set to **Automata** as it fits the profile of the project task. The server is created from a Ubuntu 16.04 64 bit image and given a RAM amount of ca 8 GB. It has a large disk capacity with 80 GB of total disk space and 4 VCPUs. The instance is given the connection to the **nsa_master_net**. To make it accessible over internet a floating IP (**128.39.121.92**) is associated to this server. The mandatory software and services will be installed and running in this server. It is an essential server for the entire setup as it is the heart of cloud cluster in the network. Additionally, the a custom TCP rule is added to the ports **1936/2377/3000/4000/8086/8687/8690** in the default security group of the OpenStack in order to make other container and services to communicate each other and to the other side of the internet. This rule makes the ports available/open for the server to use. Later the use of this rule will be discussed thoroughly. That is all needed to setup the server for the desired project goals.

Docker setup

Docker is installed in the Automata server for building Docker-containers and services. Docker services will all be spawned using Docker-Engine. The instruction to setup Docker for Ubuntu Xenial 16.04 is taken from the official Docker website [5]. The setup requires to install linux-image-extra-* packages, which allow Docker to use the aufs storage drivers. Then the packages to allow apt to use a repository over HTTPS needed to be installed followed by adding the Docker's official GPG key. And finally setting up the stable repository from Docker for Ubuntu Xenial distribution and update the server before the Docker-Engine is installed. Docker is now running in the master server.

Docker-Machine is also installed by running the following commands. This feature of Docker is necessary since the nodes (instances) in swarm are going to be spawned or created as Docker-Machine as we discussed earlier.

```
curl -L https://github.com/docker/machine/releases/download/v0.10.0/docker-machine
-‘uname -s‘-‘uname -m‘ >/tmp/docker-machine &&
chmod +x /tmp/docker-machine &&
sudo cp /tmp/docker-machine /usr/local/bin/docker-machine
```

Finally, the Docker-compose feature will be installed in the master server to manage the Docker-services to be deployed, and managed.

```
curl -L https://github.com/docker/compose/releases/download/1.13.0-rc1/docker-comp
ose-‘uname -s‘-‘uname -m‘ > /usr/local/bin/docker-compose
chmod +x /usr/local/bin/docker-compose
```

For the further setup the following versions of Docker tools are recommended:

- Docker: version 1.13 or higher, to support Docker Compose File version 3 and Swarm mode.
- Docker Machine: version 0.8 or higher.
- Docker Compose: version 1.10 or higher, to support Docker Compose file version 3.

Initial server setup

For this setup OpenStack driver are used to create the instances in the ALTO cloud. There are many other drivers which are supported by Docker-Machine such as Amazon Web Services, Microsoft Azure, Digital Ocean, Exoscale, Google Compute Engine, Generic, Oracle VirtualBox etc. In this project, OpenStack driver has been used.

To create a instance with Docker following commands need to be run first in this format along with the desired host/server names.

```
docker-machine create -d openstack \
--openstack-flavor-name <flovor_name> \
--openstack-image-id <image_id> \
--openstack-sec-groups <security_group_name> \
--openstack-net-name <nsa_master_net> \
--openstack-ssh-user <user_name> \
--openstack-keypair-name <keypair_name> \
--openstack-private-key-file <path_to_private_key> \
<machine_name>
```

To save the IP of the server this command can be used:

```
export IP=$(docker-machine ssh <machine_name> ’ifconfig ens3 | grep "inet addr:"
| cut -d: -f2 | cut -d" " -f1’)
```

Following the previous steps, three nodes Swarm-Manager, Swarm-Worker1 and Swarm-Worker2 are created which comes with Docker-Engine pre-installed. These nodes can be viewed via **docker node ls** command in master server.

Swarm setup

Now context have to be switched to use the docker engine in the swarm-manager. This have be done throughout rest of the setup demonstration in the docker engine of swarm-manager and not in the Automata system. To do this, run the command **eval ‘docker-machine env swarm-manager‘**. Now the docker engine is in Swarm-Manager node. To deploy the Swarm using swarm-manager as the manager/leader the following command has been run:

```
docker swarm init --advertise-addr ‘docker-machine ip swarm-manager‘
```

Followed by the these commands to the two workers:

```
docker-machine ssh swarm-worker1 docker swarm join --token ‘docker swarm join-token
-q worker‘ ‘docker-machine ip swarm-manager‘:2377
docker-machine ssh swarm-worker2 docker swarm join --token ‘docker swarm join-token
-q worker‘ ‘docker-machine ip swarm-manager‘:2377
```

This passes the Join Token and the published IP when the swarm was created. **docker swarm join -q worker** command get the token as worker. The default port for joining the swarm is set to 2377. **docker node ls** command now shows the manager status of the swarm.

Monitor setup

Docker-compose (in version 3 deploy parameter for each service is introduced) can define the entire stack with the deployment strategy with a single file and deploy it with one command. The parameter defines where and how the containers are to be deployed. The docker-compose file **docker-stack.yml** (Appendix: A) for the monitoring is run with the following command:

```
docker stack deploy -c docker-stack.yml monitor
```

The **yml**l file uses the InfluxDB image, create a volume named **influx** for persistent storage, mount the volume to **/var/lib/influxdb** folder in the container. A deploy copy of InfluxDB will be placed in the swarm-manager set in the deploy key part. Since the Docker-Engine is running on the Swarm-Manager the command can be executed from here itself. **depends_key** part defines that all other services will be running InfluxDB. The service Grafana section uses the **grafana/grafana** image and expose the port 3000 of the container to the port 80 of the host. The router mesh algorithm of Swarm will be implied to access Grafana from port 80 of any host in the swarm. A new volume called **grafana** created and mounted to the **/var/lib/grafana** folder of the container and a copy of Grafana will be deployed in the swarm-manager. At the end of the file, cAdvisor configuration has been set. The cAdvisor agent will be running on every container and collect all metrics from the node and the containers running on them. With the value **.NODE.ID** hostnames are determined. When cAdvisor sends metrics to InfluxDB, it also sends a tag machine that contains the hostname of cAdvisor container. After matching it with the ID of the node running it, Docker stacks allows templating its name. Containers' names are used since the origin of the metrics are important to know [6]. The **logtostderr** value redirects the logs generated by cAdvisor to **stderr** which then becomes easy to debug. **docker_only** flag is to specify the service runs in Docker containers only and the rest of the parameters are to define where the metrics are to be pushed for storage. cAdvisor pushes the metrics to database in InfluxDB server listening on port 8086. Influx service in the slack gets the metrics. All the ports are exposed in stack. A volume is then set to collect the metrics from host and Docker system and it is deployed globally (one instance of cAdvisor per node) in the swarm. Volume key are set with the Influx and Grafana volumes. All the volumes are stored in the swarm-manager.

The docker stack command starts the services and downloads the necessary image and configuration files. In the stack it is named as **monitor**. To store the metrics is InfluxDB a database named **cadvisor** is created:

```
docker exec ‘docker ps | grep -i influx | awk ’{print $1}’‘ influx -execute
’CREATE DATABASE cadvisor’
```

docker stack ls monitor shows the list of services named as monitor.
Now to setup Grafana with InfluxDB database IP of the Swarm-Manger need to be opened. To add the InfluxDB as data-source in Grafana a data-source link must be created. Choose the type as InfluxDB, the URL is http://influx:8086 and Access is proxy. This will point to the port listened by the InfluxDB container. Finally, the database cadvisor is selected and click the Save and Test button. This should give the message Data source is working. A **dashboard.json** (Appendix: B) [7] is imported to the Grafana via Menu>Dashboard>Import Option.

Web-Service setup

A docker service has been created in purpose of the serve a web-service for the users. It is a scalable service made with simple **httpd** Docker image. Port 8080 of the container is exposed to the port 8080 of the host in each swarm-node (mainly whichever the services are currently distributed and running on). A volume with small html file is attached to this service to define the different servers. The command to create and run the web service is as follows:

```
docker service create --name my-web --publish 8080:80 --replicas 1 --mount type=
bind,src=/home/ubuntu/web/,dst=/usr/local/apache2/htdocs/ httpd
```

Later the web-services in the Docker Swarm will be used to auto-scale the services using the assigned algorithm in Sec. 4.1.

Load-Balancer setup

Load-balancing service has been setup in order to balance (i.e., forward) incoming request to server to accessible container in the swarm nodes. The round-robin algorithm has been set for the back end servers. The load-balancing tool HAProxy has been installed and setup to forward the traffic from front end to back end. The configuration file of the HAProxy is included in Appendix: C. The file configures HAProxy to do set the metrics as follows:

1. In the global setting, dumping all the socket stats to the **admin.sock** file in **/var/run/haproxy** folder which has 660 mode permission for use admin and the default settings has been set.
2. The frontend are named **mysite**, listens to host port 80, set mode to http and pointed to the back-end named containers.
3. The back-end **containers** routes incoming connection request to server swarm-manager, swarm-worker1 and swarm-worker2 in round-robin fashion.
4. Finally, it listens to port 1936 to show the statistics of HAProxy gathered in a URI. It also sets the username and password for the URI.

More discussion of HAProxy settings will be discussed on the next Chapter 4. Additional software **hatop** is installed in Automata server in order to monitor HAProxy in action in real time (since the browser version is not auto-refreshed).

Scaling script

A script to automatically scale the Docker web-services in the swarm is attached to Appendix: D. The application called Socket CAT (**socat**) has been used with this bash script which is like Netcat but with security in mind to support chrooting. Socat works over various protocols and through files, pipes, devices, TCP sockets, UNIX sockets, SSL etc. It is a command line based utility that establishes two bidirectional byte streams and transfers data between them. Because of the streams can be constructed from a a large set of different types of data sinks and sources, socat can be used for different purposes.

The script starts with running every 10 secs and measure the metrics from HAProxy. Among various parameters it checks for **CurrConns*** and **MaxConn*** value of the front-end site. The socat command is used to gather metrics of a particular attributes of HAProxy:

```
echo "show info;show stat" | socat unix-connect:/var/run/haproxy/admin.sock stdio |
grep '<query_parameter>' | awk '{print $2}'
```

In the same manner, the script checks for **ConnRate*** and **SessRate*** value of the back-end server. By applying a simple algorithm (Sec. 4.1) the script sets the new connection limit and connection rate limit for HAProxy using socat. The following socat command sets the new value for HAProxy:

```
socat /var/run/haproxy/admin.sock - <<< "set maxconn <parameter> $new_value"
```

Using the above socat configuration management the developers can avoid the requirement to restarting the load-balancer while it is on the operation mode. At the end of each loop (10 secs) the script sets the number of web-services based on 10% of the new connection rate limit set using the following Docker command:

```
docker service scale <web-service>=$(new_value * 0.1)
```

The script is to implement the assigned algorithm which is discussed in more details in the next Chapter 4 along with the flow diagram of the design principle.

Test-Bench setup

Choosing any of the local server or server located in different cloud ApacheBench need to be setup for testing purposes. The test includes the performance of the web-servers in the Swarm and the their incoming request handling capabilities. Installing Apache2 and Apache2-Utils will initialise AutoBench by default. No further setup is necessary for this test-bench.

The initial system setup have been done and the technical design is ready to be conducted in this infrastructure. The next Chapter will describe the technical design implementation of this project.

* *CurrConns=Current Connections per Session, MaxConn=Maximum Connection per Session Limit, ConnRate=Connection Rate, SessRate=Session Rate*

Chapter 4

Design Principle

In the previous Chapter, the overall setup that has been done to scale a dynamic cluster of cloud serving web-services. Tools such as Docker, HAProxy are integrated to work in auto-scaling mode and to server monitoring purposes. The following sections discuss the auto-scaled cloud workflow principle in a real-life scenario:

4.1 Algorithm

To prove the proof of concept an algorithm has been set to utilize auto-scaling in this framework. Among various metrics collected by HAProxy many diverse options can be used to configure and reconfigure the HAProxy settings. Measured values such as CPU usage, memory and power consumption, incoming and outgoing traffic etc. can be observed and modify their inter-related configuration to make the load-balancer work for this dynamic cloud cluster. Since the project goal is to setup a scaled cloud based on incoming requests to query this algorithm is set to accommodate incoming current connection and session settings of the load-balancer. It is no doubt the front-end of the load-balancer is the first line of incoming request handling mechanism where the frontend settings will transfer or forward the requests to the back-end servers. The number of current connection in session are the big factor in this case. How many connection for each sessions front-end will handle is up to the algorithm to decide how to tune the load-balancer to adapt to this. After forwarding requests have been sent to back-end the number of connection per nodes (swarm-nodes) are also need further adjusting. Below the algorithm set to fine tune the load-balancer with the script (Appendix: D) is discussed. Certain values are chosen to set parameters for testing purposes only:

i If the ***Current Connection Limit*** is ***Greater*** than and ***Equal to Maximum Current Connection Default*** (This is set to a certain value. See the footer[†]) then the algorithm will proceed to next step.

ii If the difference between ***Current Connection Limit*** and ***Variable Current Connection*** is ***Greater*** than and ***Equal to Maximum Current Connection Difference***[†] then ***Current Connection Limit*** will be deducted by certain value[†] and set it as new value ***Current***

[†] *Maximum Current Connection Default=2000, Maximum Current Connection Default=2000, Minimum Service Number Default = 20, Minimum Current Connection Difference=500, Maximum Current Connection Difference=1000, Minimum Connection Rate Difference=1000, Maximum Connection Rate Difference=2000, Digit=1000, Service Scaling Ratio=100*

Connection Limit. The ***Maximum Connection*** parameter of ***Global*** and ***Front-end Server*** of load-balancer will be set to this new current connection limit.

iii If the difference between ***Current Connection Limit*** and ***Variable Current Connection*** is ***Less*** than and ***Equal to Minimum Current Connection Difference***† then ***Current Connection Limit*** will be added with certain value† and set it as new value ***Current Connection Limit***. The ***Maximum Connection*** parameter of ***Global*** and ***Front-end Server*** of load-balancer will also be set to this new current connection limit.

iv If the ***Current Connection Rate Limit*** is ***Greater*** than and ***Equal to Maximum Current Connection Rate Default***† ***AND*** the difference between current connection rate limit and ***Variable Current Connection Rate*** are ***Greater*** than ***Maximum Connection Rate Difference***† then the ***Connection Rate Limit*** will be set by deducting a certain value† from it. The ***Current Rate Limit*** of the ***Connections*** and ***Sessions*** parameter of global settings in load-balancer are set to this new value.

v ***Docker Service Scale In*** the web-services to the certain ratio based on new connection rate limit value with the ***Connection per Service Ratio*** value†.

vi If the ***Current Connection Rate Limit*** is ***Greater*** than and ***Equal to Maximum Current Connection Rate Default***† ***AND*** the difference between current connection rate limit and ***Variable Current Connection Rate*** are ***Less*** than ***Minimum Connection Rate Difference***† then the ***Connection Rate Limit*** will be set by adding a certain value† to it. The ***Current Rate Limit*** of the ***Connections*** and ***Sessions*** parameter of global settings in load-balancer are set to this new value.

vii ***Docker Service Scale Up*** the web-services to the certain ratio based on division of new connection rate limit value with the connection per service ratio value as well.

The following algorithm (Algorithm 1) is the pseudo code representation of the above procedure:

Algorithm 1: Proposed Algorithm for Automated Service

```
Data: HAProxy Socket
Result: Fetched network status and adjust the number of services
initialization
while Current Connection Limit ≥ Maximum Current Connection Default; do
    if (Current Connection Limit - Variable Current Connection) ≥ Maximum Current Connection Difference then
        Connection Limit -= 1000
        Global Maximum Connection = Connection Limit
        Front-end Server Maximum Connection = Connection Limit
        Connection Limit Rightarrow Reset
    if (Current Connection Limit - Variable Current Connection) ≤ Maximum Current Connection Difference then
        Connection Limit += 1000
        Global Maximum Connection = Connection Limit
        Front-end Server Maximum Connection = Connection Limit
        Connection Limit Rightarrow Reset
    if Current Connection Rate Limit ≥ Maximum Current Connection Rate Default then
        if (Current Connection Rate Limit - Variable Current Connection Rate) >Maximum Connection Rate Difference then
            Connection Rate Limit -= 1000
            Global Connections Rate Limit = Connection Rate Limit
            Global Sessions Rate Limit = Connection Rate Limit
            Scale = (Connection Rate Limit / Service Scale Ratio)
            Number of Services = Scale
            Connection Rate Limit ⇒ Reset
        if (Current Connection Rate Limit - Variable Current Connection Rate) <Minimum Connection Rate Difference then
            Connection Rate Limit += 1000
            Global Connections Rate Limit = Connection Rate Limit
            Global Sessions Rate Limit = Connection Rate Limit
            Scale = (Connection Rate Limit / Service Scale Ratio)
            Number of Services = Scale
            Connection Rate Limit ⇒ Reset
        else
            continue
    else
        continue
done
if Current Connection Limit < Maximum Current Connection Default then
    Global Maximum Connection = Maximum Current Connection Default
    Front-end Server Maximum Connection = Maximum Current Connection Default
    Global Connections Rate Limit = Maximum Current Connection Rate Default
    Global Sessions Rate Limit = Maximum Current Connection Rate Default
    Number of Services = Minimum Docker Service
else
    continue
```

4.2 Flow Diagram

Fig. 4.1 shows the flow diagram of the implementation of the infrastructure design. These are the steps taken when a request to access web-servers are made to the cloud cluster:

Step: 1 The incoming web-service access request comes from the other side of the internet (or locally) and reaches the Automata master server.

Step: 2 In the front-end setting of the load-balancer, the value of current connection limit is checked.

Step: 3 If the value is more than certain value (default value set to 2000) the current connection limit and variable ongoing session is checked.

Step: 4 If the difference between the current connection limit and variable incoming session is more than certain value (set to 1000) then new current connection limit is set by decreasing with certain value (set to 1000). However, If the difference is less than certain value (set to 500) the new current connection limit is set by increasing with certain value (set to 1000 also). Both global and front-end server maximum connection limit is are now set to new connection limit.

Step: 5 Load-balancer distributes the load in round-robin fashion between the Docker swarm nodes.

Step: 6 In the back-end of setting of the load-balancer, the current connection rate limit is checked.

Step: 7 If the value is more than certain value (default value set to 2000) the current connection rate limit and variable ongoing connection rate is checked.

Step: 8 cAdvisor collects metrics from swarm nodes, send to InfluxDB and creates graphs in Grafana (this step is only for purpose of monitoring the cluster).

Step: 9 If the difference between the current connection rate limit and variable ongoing connection rate is more than certain value (set to 2000) then new current connection rate limit is set by decreasing with certain value. Nevertheless, if the difference is less than certain value (set to 1000) the new current connection rate limit is set by increasing with certain value as well (set to 1000). Both rate limit of connections and sessions of the global settings of the load-balancer are set to new connection rate limit in either case.

Step: 10 If the difference is too high, the number of Docker web-services are scaled up to certain ratio (set to 100:1). On the other hand, if the difference is too low then the number of Docker web-services are scaled in to certain ratio again. The incoming requests are then sent (distributed) to different Docker web-services in the swarm cluster.

Step: 11 Web-services then replies back to the request.

Step: 12 The replies are then sent back to the client/s.

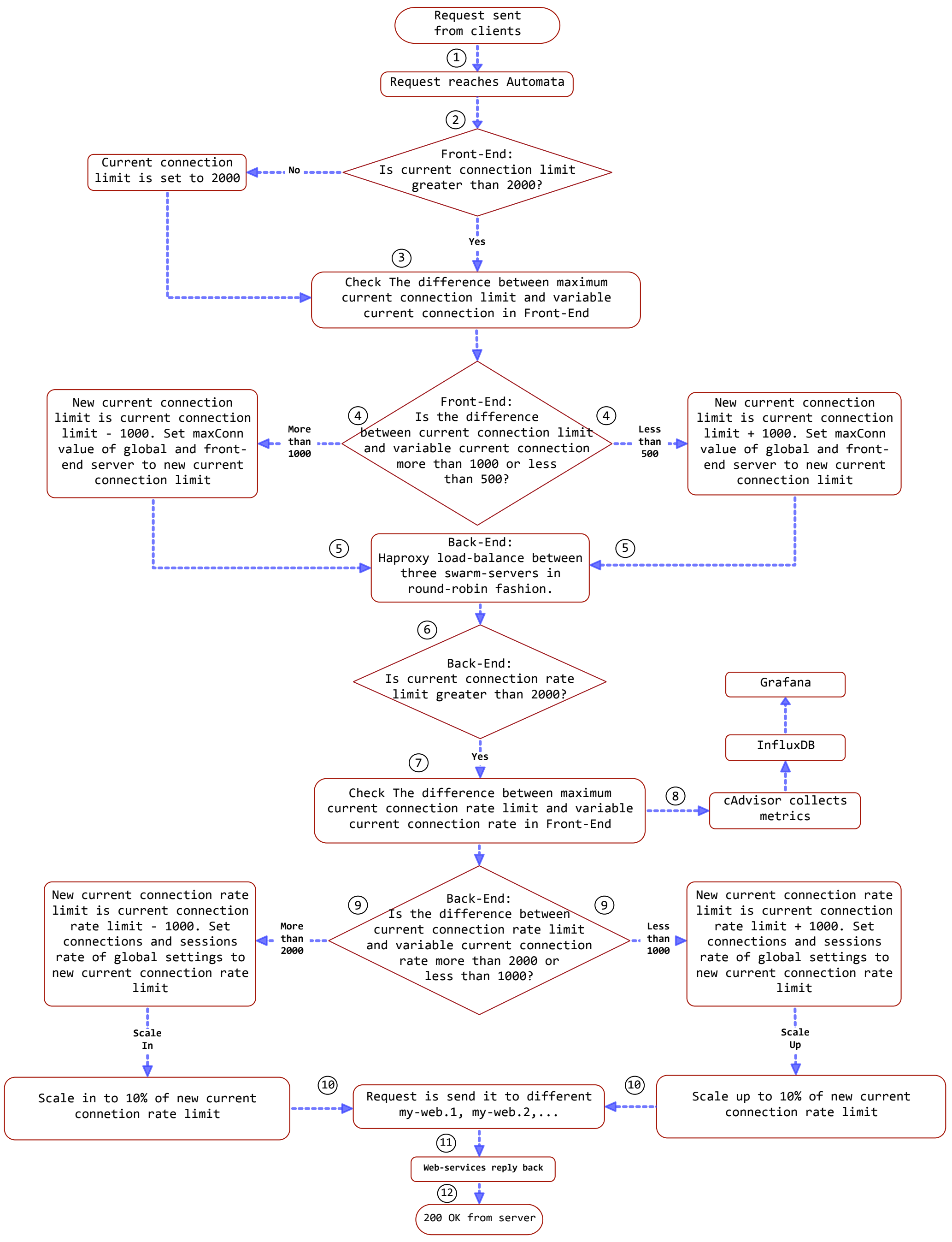

Request sent from clients
1
Request reaches Automata
2
Front-End: Is current connection limit greater than 2000?
No
Current connection limit is set to 2000
Yes
3
Check The difference between maximum current connection limit and variable current connection in Front-End
4
Front-End: Is the difference between current connection limit and variable current connection more than 1000 or less than 500?
More than 1000
New current connection limit is current connection limit - 1000. Set maxConn value of global and front-end server to new current connection limit
Less than 500
New current connection limit is current connection limit + 1000. Set maxConn value of global and front-end server to new current connection limit
5
Back-End: Haproxy load-balance between three swarm-servers in round-robin fashion.
6
Back-End: Is current connection rate limit greater than 2000?
Yes
7
Check The difference between maximum current connection rate limit and variable current connection rate in Front-End
8
cAdvisor collects metrics
InfluxDB
Grafana
9
Back-End: Is the difference between current connection rate limit and variable current connection rate more than 2000 or less than 1000?
More than 2000
New current connection rate limit is current connection rate limit - 1000. Set connections and sessions rate of global settings to new current connection rate limit
Less than 1000
New current connection rate limit is current connection rate limit + 1000. Set connections and sessions rate of global settings to new current connection rate limit
Scale In
Scale in to 10% of new current connetion rate limit
Scale Up
Scale up to 10% of new current connection rate limit
10
Request is send it to different my-web.1, my-web.2,...
11
Web-services reply back
12
200 OK from server

Figure 4.1: Illustration of the Flow Diagram of Design Implementation in the Framework

Chapter 5

Discussion

In this Chapter, the overall success and future possible strategy can be taken are discussed. The script auto-scaling.sh determines how many web-services (my-web) will be running after each iteration of system check. The various results are collected using the monitoring services Grafana, InfluxDB that are used in this project. The monitoring services can be observed in the link **http://128.39.121.92:8687**[‡] and HAProxy statistics can be observed in **http://128.39.121.92:1936**. Docker visualiser is showing in the **http://128.39.121.92:8690** (optional monitoring).

5.1 Test Scenario

ApacheBench were setup to test the auto-scaling capabilities of our design architecture in a small scale. The test scenario is the smaller scale of real-life scenario which gives the readers a small projection of functionalities of the cloud system. The test was carried by running the following command from a different server:

```
ab -n <number-of-requests> -c  <number-of-concurrency> [http[s]://]hostname[:port]
/path
```

Here, **-n** option is for number of requests to perform and **-c** option is for number of multiple requests to make at a time. We decided to send **100000** requests to perform with **1000** concurrency to make at a time to the IP address (**128.39.121.92**) on port 80 which will redirect and balance each load to Docker-Swarm nodes in round-robin fashion. Multiple tests were ongoing simultaneously for the test to enforce the real-life test bench scenarios on the web-servers. While the test runs, the script adjusts the number of web-services based on incoming requests thus the dynamic scaled cloud-based network in achieved.

5.2 Test Results

The test was successful in a small scale and the script effectively scale the network in dynamical way. The test results are included in with some figures and brief description is given for readers

[‡]username:*guest*, password:*lifeishard*

account.

Fig. 5.1 shows the network status of the Docker-Swarm nodes when the number of incoming requests are increased and adapted. This value is collected by cAdvisor and with InfluxDB query to Grafana the network status of the cluster shows up in the Grafana graph. It shows the network status of the hosts and the container with services increases and decrease according to the number of connection to the Docker-Swarm cloud. From the Fig. 5.1 it is clear that the number of services are auto-scaled up when the number of connection request from clients rises up and also auto-scaling in method bring down the service numbers of the web-service when the connection requests goes down.

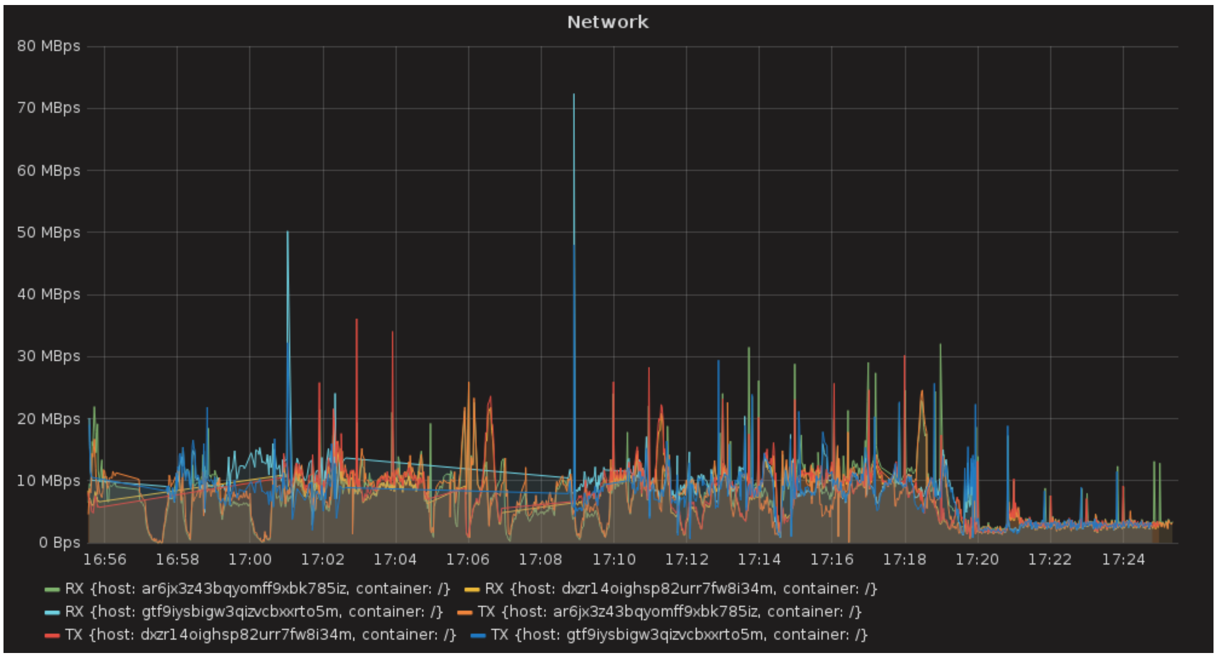


Figure 5.1: Network State of the Docker-Swarm

Fig. 5.2 and 5.3 shows the CPU and Memory status respectively of the Docker-Swarm nodes while the test is running which also can be found on Grafana. The results shows the performance of the setup while we increase the concurrency of the test bench with ApacheBench and the CPU and Memory consumption of the Docker-Machines along with their assigned random containers respectively. The containers/services were spread assigned to random Swarm nodes by the Swarm-Manager when the new scaling requests have been made. The services stays at the same node until it is assigned to removed while scaling down. The graph gives us a good insight of the system summarization and helps us monitor the servers accurately.

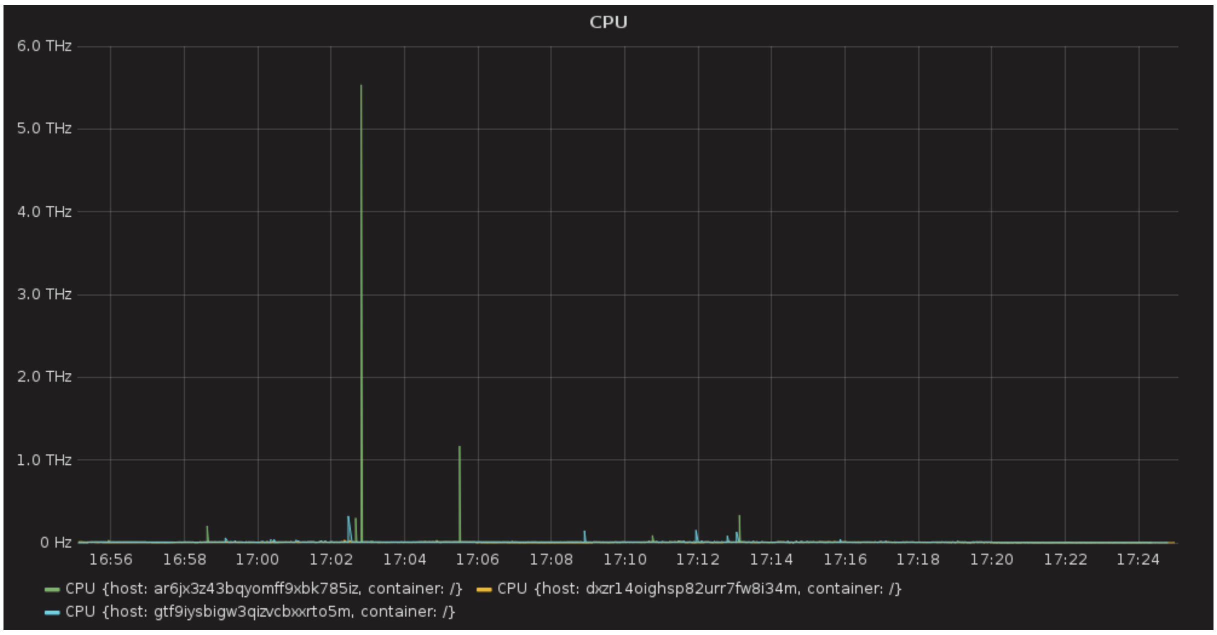


Figure 5.2: CPU State of the Docker-Swarm

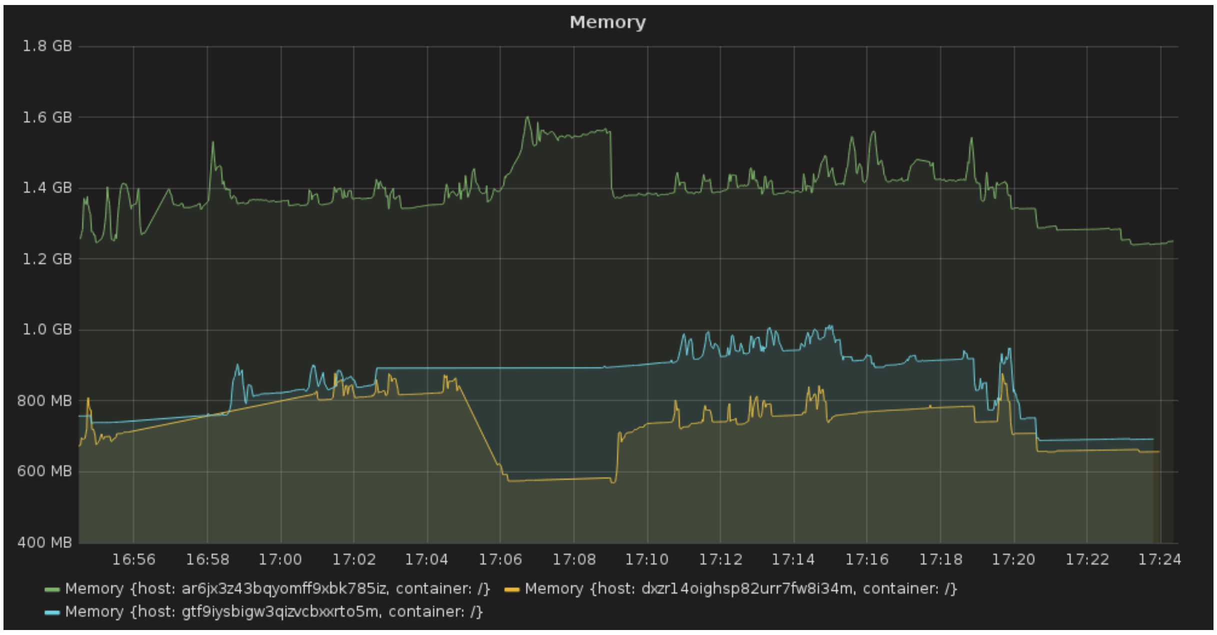
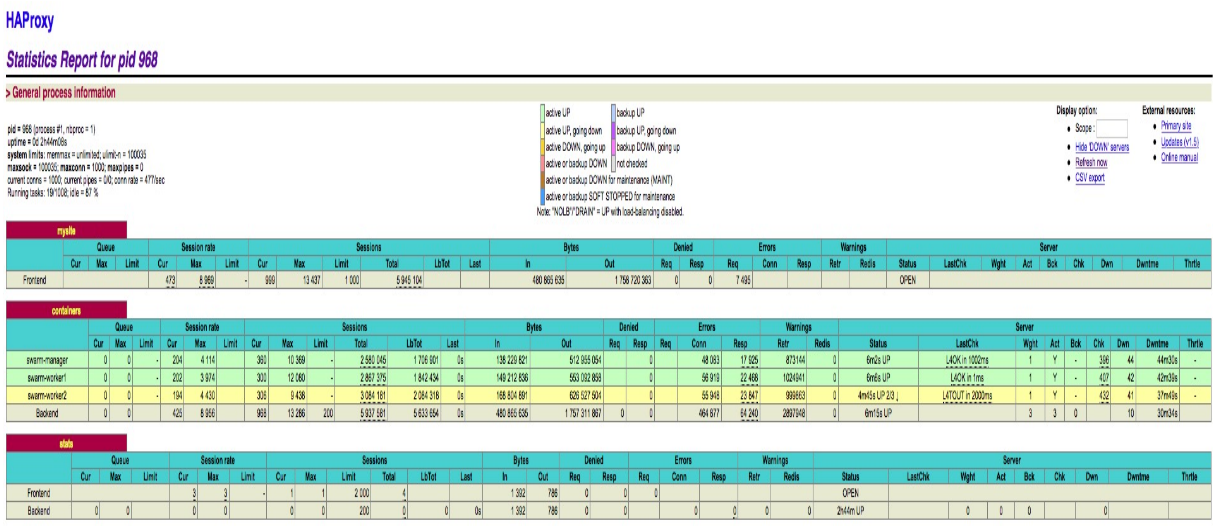


Figure 5.3: Memory State of the Docker-Swarm

In the Fig. 5.4 the status of HAProxy is shown for the Docker-Machine Cluster when the test in running against the test bench. The performance of the load-balancer can be observed from the statistics of the HAProxy which is also being adapted in order to script handle the self-regulating scaling concept. The figure (Fig. 5.5) shows the status of the Docker-Swarm with to give our readers a simple view of what is the recent situation of the swarm.



HAProxy

Statistics Report for pid 968

> General process information

pid = 968 (process #1, nbproc = 1)
uptime = 0d 2h44m08s
system limits: memmax = unlimited; ulimit-n = 100035
maxsock = 100035; **maxconn** = 1000; **maxpipes** = 0
current conns = 1000; current pipes = 0/0; conn rate = 477/sec
Running tasks: 19/1008; idle = 87 %

🟩 active UP	🟦 backup UP
🟨 active UP, going down	🟪 backup UP, going down
🟧 active DOWN, going up	🟪 backup DOWN, going up
🟥 active or backup DOWN	⬜ not checked
🟫 active or backup DOWN for maintenance (MAINT)	
🟦 active or backup SOFT STOPPED for maintenance	

Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.

Display option:
- Scope :
- Hide 'DOWN' servers
- Refresh now
- CSV export

External resources:
- Primary site
- Updates (v1.5)
- Online manual

mysite

	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				473	8 969	-	999	13 437	1 000	5 945 104			480 865 635	1 758 720 363	0	0	7 495					OPEN								

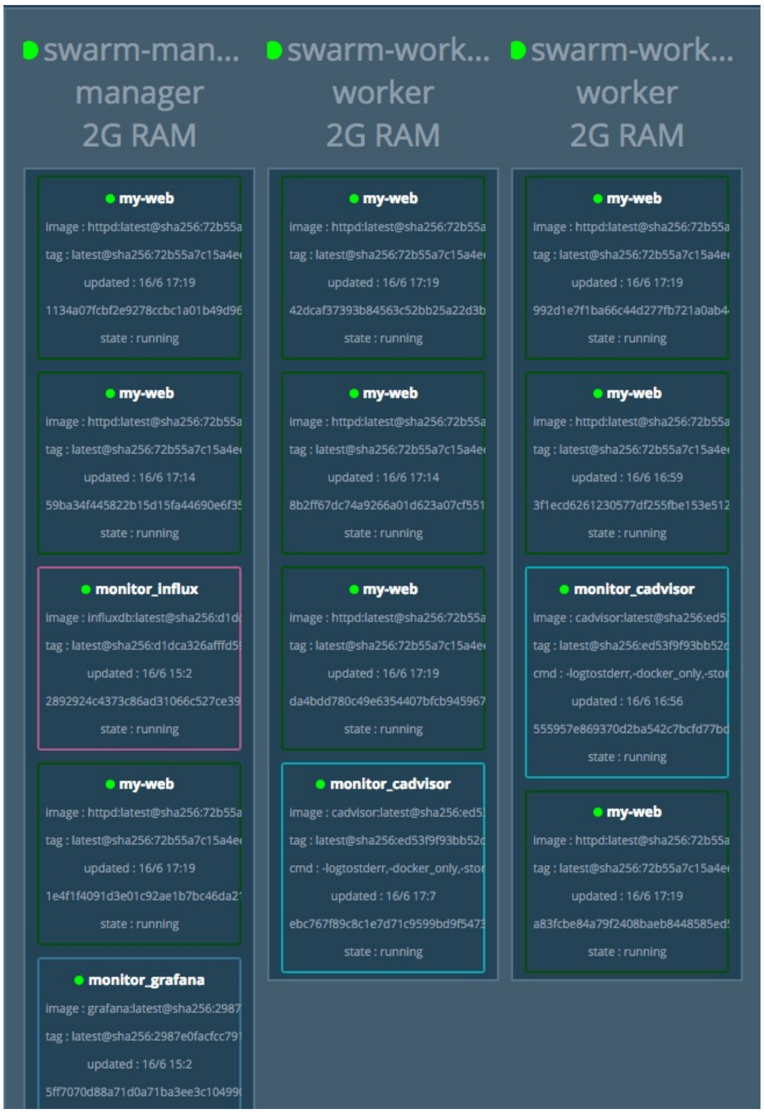
containers

	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
swarm-manager	0	0	-	204	4 114		360	10 369	-	2 580 045	1 706 901	0s	138 229 821	512 955 054		0		48 083	17 925	873144	0	6m2s UP	L4OK in 1002ms	1	Y	-	396	44	44m30s	-
swarm-worker1	0	0	-	202	3 974		300	12 080	-	2 867 375	1 842 434	0s	149 212 836	553 092 858		0		56 919	22 468	1024941	0	6m6s UP	L4OK in 1ms	1	Y	-	407	42	42m39s	-
swarm-worker2	0	0	-	194	4 430		306	9 438	-	3 084 181	2 084 318	0s	168 804 891	626 527 504		0		55 948	23 847	999863	0	4m45s UP 2/3 ↓	L4TOUT in 2000ms	1	Y	-	432	41	37m49s	-
Backend	0	0		425	8 956		968	13 286	200	5 937 581	5 633 654	0s	480 865 635	1 757 311 867	0	0		464 877	64 240	2897948	0	6m15s UP		3	3	0		10	30m34s	

stats

	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				3	3	-	1	1	2 000	4			1 392	786	0	0	0					OPEN								
Backend	0	0		0	0		0	0	200	0	0	0s	1 392	786	0	0		0	0	0	0	2h44m UP		0	0	0		0		

Figure 5.4: HAProxy Status of the Cluster



swarm-man...
manager
2G RAM

my-web
image : httpd:latest@sha256:72b55a
tag : latest@sha256:72b55a7c15a4e
updated : 16/6 17:19
1134a07fcbf2e9278ccbc1a01b49d96
state : running

my-web
image : httpd:latest@sha256:72b55a
tag : latest@sha256:72b55a7c15a4e
updated : 16/6 17:14
59ba34f445822b15d15fa44690e6f35
state : running

monitor_influx
image : influxdb:latest@sha256:d1d
tag : latest@sha256:d1dca326afffd5
updated : 16/6 15:2
2892924c4373c86ad31066c527ce39
state : running

my-web
image : httpd:latest@sha256:72b55a
tag : latest@sha256:72b55a7c15a4e
updated : 16/6 17:19
1e4f1f4091d3e01c92ae1b7bc46da2
state : running

monitor_grafana
image : grafana:latest@sha256:2987
tag : latest@sha256:2987e0facfcc79
updated : 16/6 15:2
5ff7070d88a71d0a71ba3ee3c10499

swarm-work...
worker
2G RAM

my-web
image : httpd:latest@sha256:72b55a
tag : latest@sha256:72b55a7c15a4e
updated : 16/6 17:19
42dcaf37393b84563c52bb25a22d3b
state : running

my-web
image : httpd:latest@sha256:72b55a
tag : latest@sha256:72b55a7c15a4e
updated : 16/6 17:14
8b2ff67dc74a9266a01d623a07cf551
state : running

my-web
image : httpd:latest@sha256:72b55a
tag : latest@sha256:72b55a7c15a4e
updated : 16/6 17:19
da4bdd780c49e6354407bfcb945967
state : running

monitor_cadvisor
image : cadvisor:latest@sha256:ed5
tag : latest@sha256:ed53f9f93bb52d
cmd : -logtostderr,-docker_only,-stor
updated : 16/6 17:7
ebc767f89c8c1e7d71c9599bd9f547
state : running

swarm-work...
worker
2G RAM

my-web
image : httpd:latest@sha256:72b55a
tag : latest@sha256:72b55a7c15a4e
updated : 16/6 17:19
992d1e7f1ba66c44d277fb721a0ab4
state : running

my-web
image : httpd:latest@sha256:72b55a
tag : latest@sha256:72b55a7c15a4e
updated : 16/6 16:59
3f1ecd6261230577df255fbe153e512
state : running

monitor_cadvisor
image : cadvisor:latest@sha256:ed5
tag : latest@sha256:ed53f9f93bb52d
cmd : -logtostderr,-docker_only,-stor
updated : 16/6 16:56
555957e869370d2ba542c7bcfd77bd
state : running

my-web
image : httpd:latest@sha256:72b55a
tag : latest@sha256:72b55a7c15a4e
updated : 16/6 17:19
a83fcbe84a79f2408baeb8448585ed
state : running

Figure 5.5: Visualisation of the Docker-Swarm

To give readers a notion of how the scripting adjust the values of HAProxy, the following Table 5.1 is included. The table gives an example output of when the script adjust the value of different network parameters in different time points.

Table 5.1: Example of Auto-Scaling with HAProxy

	Front-End		**Back-End**		
Time Point	**CurrConns**	**MaxConn**	**CurrConnRateLimit**	**CurrConnRate**	**Number of Services**
A	2000	5000	2000	400	20
B	3000	1600	3000	1200	30
C	3000	2300	3000	1500	30
D	2000	800	2000	800	20

The time graph is represented with the following figure (Fig. 5.6):

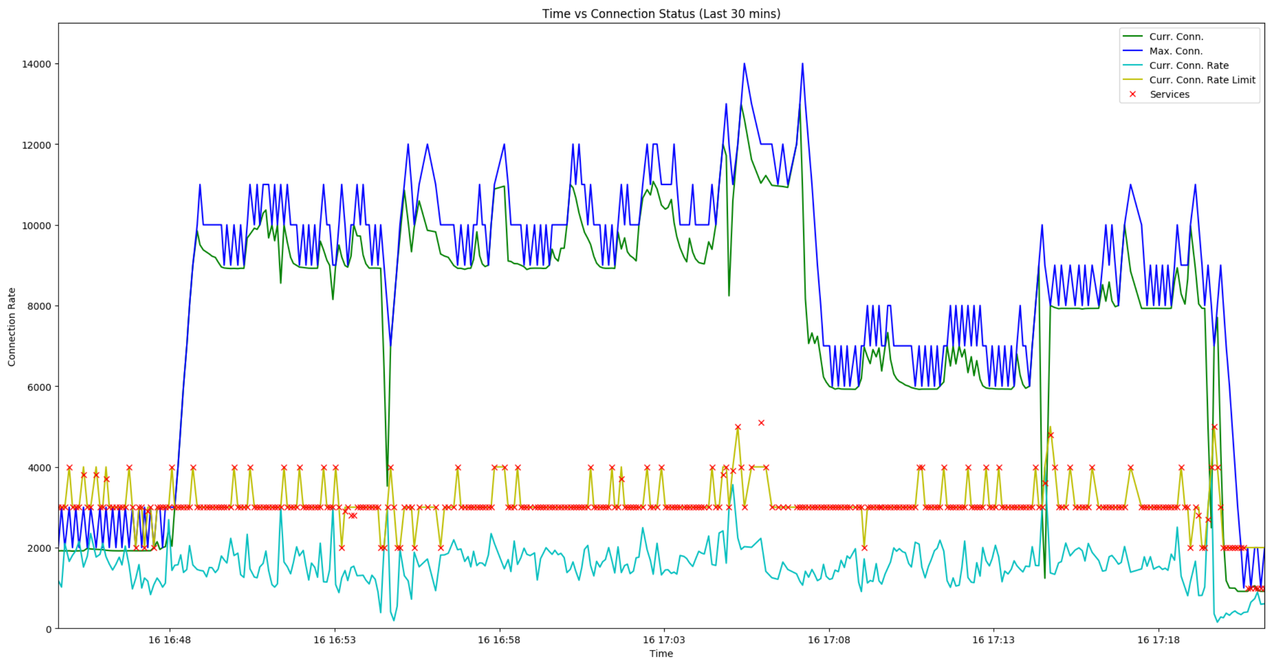


Figure 5.6: Time vs Connection Status

In the graph the last 30 minutes of network status of the cluster have been included. It clearly shows the how the maximum connection limit (blue line) of load-balancer has been adjusted according to the current connection (green line) status in a given time. At the same time, the connection connection rate limit (yellow line) has been adjust according to the number incoming current connection rate (cyan line). And finally, the web-services (red x) are scaled to up and in according to the current connection rate limit that was set for the current connection rate.

5.3 Possible Future Work

- Ideally we should have use a monitoring service which can watch over the nodes and based on some alerts will trigger the scaling of the services. Monitoring tools such as Consul or similar can be used to check the health of the nodes and change the configuration files of load-balancer and restart the services. This should be a optimized setup for such framework.
- The number of Docker-Swarm nodes can be also adjusted depending the number of requests. This requires the script to spawn or add and likewise delete the nodes when the load-balancer reaches its global maximum configuration limits. For real-life scenario the not only the number of services but also the number of nodes needed to be scaled as well.
- As we discussed earlier, in the script we could have consider other metrics of load-balancing attributes rather than just considering network settings. This will make this cluster more vibrant and dynamic.
- A backup/secondary load-balancer could be added in case of primary load-balancer fails or shuts down.

Chapter 6

Conclusion

In this project, we learned the basic principle behind the auto-scaling of the cloud-based services and advantage of using appropriate load-balancing of the cloud-cluster. In the process of adapting automation of scaling services we used the some of the most recognised and well-structured tools like Docker-Engine, HAProxy along with the OpenStack cloud-platform. There is not just one exact way to complete this project instead we found many other different ways to complete this assignment. The approach we chose is not only simple to setup but also an efficient way to achieve the project goals.

This particular project enable us to dive deep into the cluster infrastructure which help us understanding the complex mechanism behind web-services design and load-balancer algorithms and engineering that ties the different enabling technology together. We have learned thoroughly the procedure of using auto-scaling in real-life Docker based cloud system environment which can help us to do better research in this noble field of web-service management and developer operations.

References

[1] Docker Developer Team, "*Docker Machine Overview*", [online] Available: https://docs.docker.com/machine/overview/

[2] Docker Developer Team, "*Swarm Mode Overview*", [online] Available: https://docs.docker.com/engine/swarm/

[3] HAProxy Developer Team, "*How HAProxy Works*", [online] Available: https://cbonte.github.io/haproxy-dconv/1.8/intro.html#3.2

[4] Mitchell Anicas, "*An Introduction to HAProxy and Load Balancing Concepts*", [online] Available: https://www.digitalocean.com/community/tutorials/an-introduction-to-haproxy-and-load-balancing-concepts

[5] Docker Team, "*Get Docker for Ubuntu*", [online] Available: https://docs.docker.com/engine/installation/linux/ubuntu/

[6] Moby, "*Create services using templates*", [online] Available: https://github.com/moby/moby/blob/master/docs/reference/commandline/service_create.md#create-services-using-templates

[7] Botleg, "*Swarm-Monitoring*", [online] Available: https://github.com/botleg/swarm-monitoring/blob/master/dashboard.json

Appendices

Appendix A

docker-stack.yml

```
version: '3'

services:
  influx:
    image: influxdb
    volumes:
      - influx:/var/lib/influxdb
    deploy:
      replicas: 1
      placement:
        constraints:
          - node.role == manager

  grafana:
    image: grafana/grafana
    ports:
      - 0.0.0.0:80:3000
    volumes:
      - grafana:/var/lib/grafana
    depends_on:
      - influx
    deploy:
      replicas: 1
      placement:
        constraints:
          - node.role == manager

  cadvisor:
    image: google/cadvisor
    hostname: '{{.Node.ID}}'
    command: -logtostderr -docker_only -storage_driver=influxdb -storage_driver_db
        =cadvisor -storage_driver_host=influx:8086
    volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:rw
      - /sys:/sys:ro
      - /var/lib/docker/:/var/lib/docker:ro
    depends_on:
      - influx
    deploy:
      mode: global

```

```
volumes:
  influx:
    driver: local
  grafana:
    driver: local
```

Appendix B

dashboard.json

```
{
  "__inputs": [
    {
      "name": "DS_INFLUX",
      "label": "influx",
      "description": "",
      "type": "datasource",
      "pluginId": "influxdb",
      "pluginName": "InfluxDB"
    }
  ],
  "__requires": [
    {
      "type": "grafana",
      "id": "grafana",
      "name": "Grafana",
      "version": "4.2.0"
    },
    {
      "type": "panel",
      "id": "graph",
      "name": "Graph",
      "version": ""
    },
    {
      "type": "datasource",
      "id": "influxdb",
      "name": "InfluxDB",
      "version": "1.0.0"
    }
  ],
  "annotations": {
    "list": []
  },
  "editable": true,
  "gnetId": null,
  "graphTooltip": 0,
  "hideControls": false,
  "id": null,
  "links": [],
  "refresh": false,
  "rows": [
    {
      "collapse": false,
      "height": 250,
      "panels": [
        {
          "aliasColors": {},
          "bars": false,
          "datasource": "${DS_INFLUX}",
          "fill": 1,
          "id": 1,
          "legend": {
            "avg": false,
            "current": false,
            "max": false,
            "min": false,
            "show": true,
            "total": false,
            "values": false
          },
          "lines": true,
          "linewidth": 1,
          "links": [],
          "nullPointMode": "null",
          "percentage": false,
          "pointradius": 5,
          "points": false,
          "renderer": "flot",
          "seriesOverrides": [],
          "span": 6,
          "stack": false,
          "steppedLine": false,
          "targets": [
            {
              "alias": "Memory {host: $tag_machine, container: $tag_container_name
                 }",
              "dsType": "influxdb",
              "groupBy": [
                {
                  "params": [
                    "machine"
                  ],
                  "type": "tag"
                },
                {
                  "params": [
                    "container_name"
                  ],
                  "type": "tag"
                }
              ],
              "measurement": "memory_usage",
              "policy": "default",
              "query": "SELECT \"value\" FROM \"memory_usage\" WHERE \"
                  container_name\" =~ /^$container$/ AND \"machine\" =~ /^$host$/
                  AND $timeFilter",
              "rawQuery": false,
              "refId": "A",
              "resultFormat": "time_series",
              "select": [
                [
                  {
                    "params": [
                      "value"
                    ],
                    "type": "field"
                  }
                ]
              ],
              "tags": [
                {
                  "key": "container_name",
                  "operator": "=~",
                  "value": "/^$container$*/"
                },
                {
                  "condition": "AND",
                  "key": "machine",
                  "operator": "=~",
                  "value": "/^$host$/"
                }
              ]
            }
          ],
          "thresholds": [],
          "timeFrom": null,
          "timeShift": null,
          "title": "Memory",
          "tooltip": {
            "shared": true,
            "sort": 0,
            "value_type": "individual"
          },
          "type": "graph",
          "xaxis": {
            "mode": "time",
            "name": null,
            "show": true,
            "values": []
          },
          "yaxes": [
            {
              "format": "decbytes",
              "label": null,
              "logBase": 1,
              "max": null,
              "min": null,
              "show": true
            },
            {
              "format": "short",
              "label": null,
              "logBase": 1,
              "max": null,
              "min": null,
              "show": true
            }
          ]
        },
        {
          "aliasColors": {},
          "bars": false,
          "datasource": "${DS_INFLUX}",
          "fill": 1,
          "id": 2,
          "legend": {
            "avg": false,
            "current": false,
            "max": false,
            "min": false,
            "show": true,
            "total": false,
            "values": false
          },
          "lines": true,
          "linewidth": 1,
          "links": [],
          "nullPointMode": "null",
          "percentage": false,
          "pointradius": 5,
          "points": false,
          "renderer": "flot",
          "seriesOverrides": [],
          "span": 6,
          "stack": false,
          "steppedLine": false,
          "targets": [
            {
              "alias": "CPU {host: $tag_machine, container: $tag_container_name}",
              "dsType": "influxdb",
              "groupBy": [
                {
                  "params": [
                    "machine"
                  ],
                  "type": "tag"
                },
                {
                  "params": [
                    "container_name"
                  ],
                  "type": "tag"
                }
              ],
              "measurement": "cpu_usage_total",
              "policy": "default",
              "refId": "A",
              "resultFormat": "time_series",
              "select": [
                [
                  {
                    "params": [
                      "value"
                    ],
                    "type": "field"
                  },
                  {
                    "params": [
                      "10s"
                    ],
                    "type": "derivative"
                  }
                ]
              ],
              "tags": [
                {
                  "key": "container_name",
                  "operator": "=~",
                  "value": "/^$container$*/"
                },
                {
                  "condition": "AND",
                  "key": "machine",
                  "operator": "=~",
                  "value": "/^$host$/"
                }
              ]
            }
          ],
          "thresholds": [],
          "timeFrom": null,
          "timeShift": null,
          "title": "CPU",
          "tooltip": {
            "shared": true,
            "sort": 0,
            "value_type": "individual"
          },
          "type": "graph",
          "xaxis": {
            "mode": "time",
            "name": null,
            "show": true,
            "values": []
          },
          "yaxes": [
            {
              "format": "hertz",
              "label": null,
              "logBase": 1,
              "max": null,
              "min": null,
              "show": true
            },
            {
              "format": "short",
              "label": null,
              "logBase": 1,
              "max": null,
              "min": null,
              "show": true
            }
          ]
        }
      ],
      "repeat": null,
      "repeatIteration": null,
      "repeatRowId": null,
      "showTitle": false,
      "title": "Dashboard Row",
      "titleSize": "h6"
    },
    {
      "collapse": false,
      "height": 250,
      "panels": [
        {
          "aliasColors": {},
          "bars": false,
          "datasource": "${DS_INFLUX}",
          "fill": 1,
          "id": 3,
          "legend": {
            "avg": false,
            "current": false,
            "max": false,
            "min": false,
            "show": true,
            "total": false,
            "values": false
          },
          "lines": true,
          "linewidth": 1,
          "links": [],
          "nullPointMode": "null",
          "percentage": false,
          "pointradius": 5,
          "points": false,
          "renderer": "flot",
          "seriesOverrides": [],
          "span": 6,
          "stack": false,
          "steppedLine": false,
          "targets": [
            {
              "alias": "Usage {host: $tag_machine, container: $tag_container_name
                }",
              "dsType": "influxdb",
              "groupBy": [
                {
                  "params": [
                    "container_name"
                  ],
                  "type": "tag"
                },
                {
```

```
                "params": [
                  "machine"
                ],
                "type": "tag"
              }
            ],
            "measurement": "fs_usage",
            "policy": "default",
            "refId": "A",
            "resultFormat": "time_series",
            "select": [
              [
                {
                  "params": [
                    "value"
                  ],
                  "type": "field"
                }
              ]
            ],
            "tags": [
              {
                "key": "machine",
                "operator": "=~",
                "value": "/^$host$/"
              },
              {
                "condition": "AND",
                "key": "container_name",
                "operator": "=~",
                "value": "/^$container$*/"
              }
            ]
          },
          {
            "alias": "Limit {host: $tag_machine, container: $tag_container_name
                }",
            "dsType": "influxdb",
            "groupBy": [
              {
                "params": [
                  "container_name"
                ],
                "type": "tag"
              },
              {
                "params": [
                  "machine"
                ],
                "type": "tag"
              }
            ],
            "measurement": "fs_limit",
            "policy": "default",
            "refId": "B",
            "resultFormat": "time_series",
            "select": [
                [
                  {
                    "params": [
                      "value"
                    ],
                    "type": "field"
                  }
                ]
              ],
              "tags": [
                {
                  "key": "machine",
                  "operator": "=~",
                  "value": "/^$host$/"
                },
                {
                  "condition": "AND",
                  "key": "container_name",
                  "operator": "=~",
                  "value": "/^$container$*/"
                }
              ]
            }
          ],
          "thresholds": [],
          "timeFrom": null,
          "timeShift": null,
          "title": "File System",
          "tooltip": {
            "shared": true,
            "sort": 0,
            "value_type": "individual"
          },
          "type": "graph",
          "xaxis": {
            "mode": "time",
            "name": null,
            "show": true,
            "values": []
          },
          "yaxes": [
            {
              "format": "decbytes",
              "label": null,
              "logBase": 1,
              "max": null,
              "min": null,
              "show": true
            },
            {
              "format": "short",
              "label": null,
              "logBase": 1,
              "max": null,
              "min": null,
              "show": true
            }
```

```
        ]
      },
      {
        "aliasColors": {},
        "bars": false,
        "datasource": "${DS_INFLUX}",
        "fill": 1,
        "id": 4,
        "legend": {
          "avg": false,
          "current": false,
          "max": false,
          "min": false,
          "show": true,
          "total": false,
          "values": false
        },
        "lines": true,
        "linewidth": 1,
        "links": [],
        "nullPointMode": "null",
        "percentage": false,
        "pointradius": 5,
        "points": false,
        "renderer": "flot",
        "seriesOverrides": [],
        "span": 6,
        "stack": false,
        "steppedLine": false,
        "targets": [
          {
            "alias": "RX {host: $tag_machine, container: $tag_container_name}",
            "dsType": "influxdb",
            "groupBy": [
              {
                "params": [
                  "container_name"
                ],
                "type": "tag"
              },
              {
                "params": [
                  "machine"
                ],
                "type": "tag"
              }
            ],
            "measurement": "rx_bytes",
            "policy": "default",
            "refId": "A",
            "resultFormat": "time_series",
            "select": [
              [
                {
                  "params": [
                    "value"
                  ],
                  "type": "field"
                },
                {
                  "params": [
                    "10s"
                  ],
                  "type": "derivative"
                }
              ]
            ],
            "tags": [
              {
                "key": "machine",
                "operator": "=~",
                "value": "/^$host$/"
              },
              {
                "condition": "AND",
                "key": "container_name",
                "operator": "=~",
                "value": "/^$container$*/"
              }
            ]
          },
          {
            "alias": "TX {host: $tag_machine, container: $tag_container_name}",
            "dsType": "influxdb",
            "groupBy": [
              {
                "params": [
                  "container_name"
                ],
                "type": "tag"
              },
              {
                "params": [
                  "machine"
                ],
                "type": "tag"
              }
            ],
            "measurement": "tx_bytes",
            "policy": "default",
            "refId": "B",
            "resultFormat": "time_series",
            "select": [
              [
                {
                  "params": [
                    "value"
                  ],
                  "type": "field"
                },
                {
                  "params": [
                    "10s"
                  ],
                  "type": "derivative"
                }
              ]
            ],
            "tags": [
              {
                "key": "machine",
                "operator": "=~",
                "value": "/^$host$/"
              },
              {
                "condition": "AND",
                "key": "container_name",
                "operator": "=~",
                "value": "/^$container$*/"
              }
            ]
          }
        ],
        "thresholds": [],
        "timeFrom": null,
        "timeShift": null,
        "title": "Network",
        "tooltip": {
          "shared": true,
          "sort": 0,
          "value_type": "individual"
        },
        "type": "graph",
        "xaxis": {
          "mode": "time",
          "name": null,
          "show": true,
          "values": []
        },
        "yaxes": [
          {
            "format": "Bps",
            "label": null,
            "logBase": 1,
            "max": null,
            "min": null,
            "show": true
          },
          {
            "format": "short",
            "label": null,
            "logBase": 1,
            "max": null,
            "min": null,
            "show": true
          }
        ]
      }
    ],
    "repeat": null,
    "repeatIteration": null,
    "repeatRowId": null,
    "showTitle": false,
    "title": "Dashboard Row",
    "titleSize": "h6"
  }
],
"schemaVersion": 14,
"style": "dark",
"tags": [],
"templating": {
  "list": [
    {
      "allValue": "",
      "current": {},
      "datasource": "${DS_INFLUX}",
      "hide": 0,
      "includeAll": true,
      "label": "Host",
      "multi": false,
      "name": "host",
      "options": [],
      "query": "show tag values with key = \"machine\"",
      "refresh": 1,
      "regex": "",
      "sort": 0,
      "tagValuesQuery": "",
      "tags": [],
      "tagsQuery": "",
      "type": "query",
      "useTags": false
    },
    {
      "allValue": null,
      "current": {},
      "datasource": "${DS_INFLUX}",
      "hide": 0,
      "includeAll": false,
      "label": "Container",
      "multi": false,
      "name": "container",
      "options": [],
      "query": "show tag values with key = \"container_name\" WHERE machine =~
         /^$host$/",
      "refresh": 1,
      "regex": "/([^.]+)/",
      "sort": 0,
      "tagValuesQuery": "",
      "tags": [],
      "tagsQuery": "",
      "type": "query",
      "useTags": false
    }
  ]
},
"time": {
  "from": "now-1h",
  "to": "now"
},
"timepicker": {
  "refresh_intervals": [
    "5s",
    "10s",
    "30s",
    "1m",
    "5m",
    "15m",
    "30m",
    "1h",
    "2h",
    "1d"
  ],
  "time_options": [
    "5m",
    "15m",
    "1h",
    "6h",
    "12h",
    "24h",
    "2d",
    "7d",
    "30d"
  ]
},
"timezone": "browser",
"title": "cAdvisor",
"version": 3
}
```

Appendix C

haproxy.cfg

```
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
  maxconn 50000
  # 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). This list is from:
  #  https://hynek.me/articles/hardening-your-web-servers-ssl-ciphers/
  ssl-default-bind-ciphers ECDH+AESGCM:DH+AESGCM:ECDH+AES256:DH+AES256:ECDH+AES128
      :DH+AES:ECDH+3DES:DH+3DES:RSA+AESGCM:RSA+AES:RSA+3DES:!aNULL:!MD5:!DSS
  ssl-default-bind-options no-sslv3

defaults
  log global
  mode  http
  option  httplog
  option  dontlognull
        timeout connect 5000
        timeout client  50000
        timeout server  50000
  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

# Configure HAProxy to listen on port 80
frontend mysite
  bind *:80
  mode http
  default_backend containers
```

```

# Configure HAProxy to route requests to swarm nodes on port 8080
backend containers
  mode http
  balance roundrobin
  server swarm-manager 10.1.25.46:8080 check
  server swarm-worker1 10.1.25.47:8080 check
  server swarm-worker2 10.1.25.48:8080 check

# Configure HAProxy to listen on port 1936 and show the statistics in uri
listen stats
        bind *:1936
        stats enable
        stats uri /
        stats hide-version
        stats auth samiul:lifeishard
```

Appendix D

auto-scaling.sh

```
#!/bin/bash
clear

VarCurrConn=$(echo "show info;show stat" | socat unix-connect:/var/run/haproxy/admin.sock stdio | grep 'CurrConns' | awk '{print $2}')
MaxCurrConn=$(echo "show info;show stat" | socat unix-connect:/var/run/haproxy/admin.sock stdio | grep 'Maxconn' | awk '{print $2}')
VarConnRate=$(echo "show info;show stat" | socat unix-connect:/var/run/haproxy/admin.sock stdio | grep '^ConnRate:' | awk '{print $2}')
MaxConnRate=$(echo "show info;show stat" | socat unix-connect:/var/run/haproxy/admin.sock stdio | grep '^ConnRateLimit:' | awk '{print $2}')
DockerPs=$(docker service ps my-web | grep "my-web.*" | grep "Running" | wc -l)

echo "Variable Current Connection :" $VarCurrConn
echo "Maximum Current Connection Limit :" $MaxCurrConn
echo "Variable Connetion Rate : "$VarConnRate
echo "Maximum Connection Rate Limit :" $MaxConnRate

printf "`date +'%d-%m-%y-%T'`,$VarCurrConn,$MaxCurrConn,$VarConnRate,$MaxConnRate,$DockerPs\n" >> file.csv

MaxConnDefault=2000
MaxConnRateDefault=2000
ServScaleRatio=100
MinDockerServ=10
MinConnDiff=500
MaxConnDiff=1000
MinConnRateDiff=1000
MaxConnRateDiff=2000
Digit=1000

if [ "$((MaxCurrConn - VarCurrConn))" -le  "$MinConnDiff" ] ; then
  printf "MaxCurrConn is less than VarrCurrConn\n"
  ConnLimit="$((MaxCurrConn + Digit))"
  echo "Connection Limits are now :" $ConnLimit
  SetGlobal=$(socat /var/run/haproxy/admin.sock - <<< "set maxconn global $ConnLimit")
        SetFrontend=$(socat /var/run/haproxy/admin.sock - <<< "set maxconn frontend mysite $ConnLimit")
  printf "Session Limits are set for Global and Frontend to %d (increased)\n\n" $ConnLimit
  ConnLimit=0
```

```
fi

if [ "$((MaxCurrConn - VarCurrConn))" -ge  "$MaxConnDiff" ] ; then
  printf "MaxCurrConn is greater than VarrCurrConn\n"
  ConnLimit="$((MaxCurrConn - Digit))"
  echo "Connection Limits are now :" $ConnLimit
  SetGlobal=$(socat /var/run/haproxy/admin.sock - <<< "set maxconn global
      $ConnLimit")
  SetFrontend=$(socat /var/run/haproxy/admin.sock - <<< "set maxconn frontend
      mysite $ConnLimit")
  printf "Session Limits are set for Global and Frontend to %d (decreased)\n\n"
      $ConnLimit
  ConnLimit=0
fi

if  [ "$MaxConnRate" -ge "$MaxConnRateDefault" ] && [ "$((MaxConnRate -
    VarConnRate))" -lt  "$MinConnRateDiff" ] ; then
  ConnRateLimit="$((MaxConnRate + Digit))"
  SetRateConn=$(socat /var/run/haproxy/admin.sock - <<< "set rate-limit
      connections global  $ConnRateLimit")
  SetRateSess=$(socat /var/run/haproxy/admin.sock - <<< "set rate-limit sessions
      global  $ConnRateLimit")
  printf "Connections and Sessions Rate Limits are set for Global to %d (increased
      )\n\n"  $ConnRateLimit
  Scale=$(expr $ConnRateLimit / $ServScaleRatio)
  echo "Scale :"$Scale
  docker service scale my-web=$Scale
  ConnRateLimit=0
fi

if  [ "$MaxConnRate" -ge "$MaxConnRateDefault" ] && [ "$((MaxConnRate -
    VarConnRate))" -gt  "$MaxConnRateDiff" ] ; then
  ConnRateLimit="$((MaxConnRate - Digit))"
  SetRateConn=$(socat /var/run/haproxy/admin.sock - <<< "set rate-limit
      connections global  $ConnRateLimit")
  SetRateSess=$(socat /var/run/haproxy/admin.sock - <<< "set rate-limit sessions
      global  $ConnRateLimit")
  printf "Connections and Sessions Rate Limits are set for Global to %d (decreased
      )\n\n"  $ConnRateLimit
  Scale=$(expr $ConnRateLimit / $ServScaleRatio)
  echo "Scale :" $Scale
  docker service scale my-web=$Scale
  ConnRateLimit=0
fi

if [ "$MaxCurrConn" -lt "$MaxConnDefault" ]; then
  printf "Maximum Current Connection is Lower\n"
  printf "Changing to default values........."
  SetGlobal=$(socat /var/run/haproxy/admin.sock - <<< "set maxconn global
      $MaxConnDefault")
  SetFrontend=$(socat /var/run/haproxy/admin.sock - <<< "set maxconn frontend
      mysite $MaxConnDefault")
  SetRateConn=$(socat /var/run/haproxy/admin.sock - <<< "set rate-limit
      connections global  $MaxConnRateDefault")
  SetRateSess=$(socat /var/run/haproxy/admin.sock - <<< "set rate-limit sessions
      global  $MaxConnRateDefault")
  SetDockerService=$(docker service scale my-web=$MinDockerServ)
  sleep 2s
fi
```