LINUX NETWORKING

AUTOSCALING AS A SERVICE

Milestone 2 updates

MUKUL VASHIST (mvashis2)
SANDEEP GHANTA (sghanta)
SHANTANU BHOYAR (sbhoyar)
VAIBHAV NAGVEKAR (vpnagvek)

Project Description

Introduction

Before cloud computing, it was very difficult to scale a website, let alone figure out a way to automatically scale a server setup. In a traditional, dedicated hosting environment, we were limited by our hardware resources. Once the server resources were maxed out, the site would inevitably suffer from a performance perspective and possibly crash, thereby causing us to lose data.

Today, cloud computing has made it possible to build a fully scalable and managed server setup. Of the many features, auto-scaling is a way to automatically scale in or out the number of compute resources that are being allocated to the application based on its needs at any given time. It allows us to set up and configure the necessary triggers so that we can create an automated setup to automatically manage resources when thresholds are crossed(upper/lower).

Benefits of Auto Scaling:

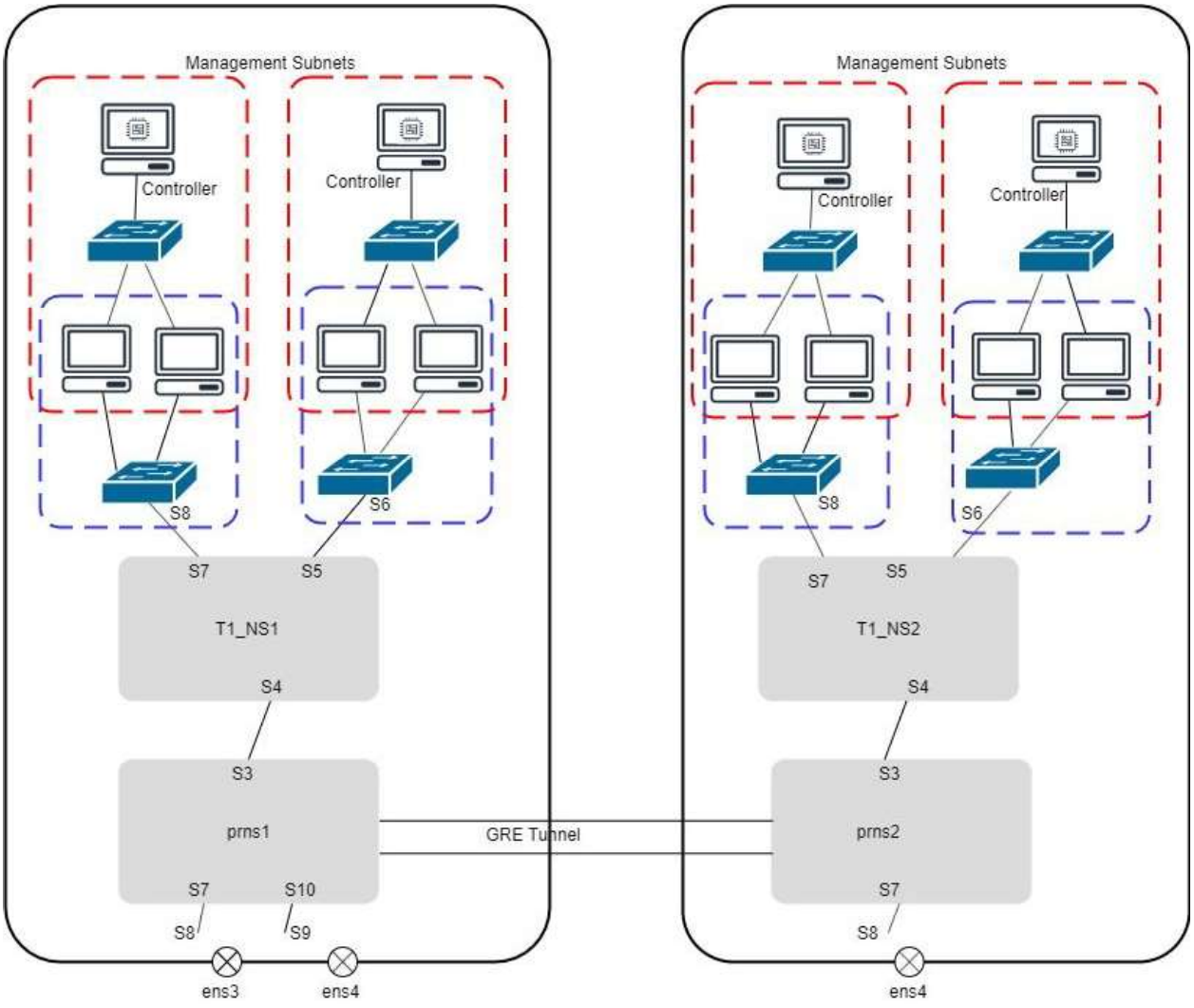
1. Better Performance Control: Autoscaling helps control the tradeoff between cost and performance.
2. Better cost management: Dynamically increases and decreases capacity as needed. Because one pays for the instances used, they can save money by launching instances when needed and terminating them when they are not in use.

Multi-cloud VPC environment

With many customers having their workloads/servers spanning multiple cloud providers, a solution for managing resources in a hybrid VPC environment can be really useful for driving down costs, at the same time improving asset value.
Our project will be an auto-scaling solution capable of managing workloads across different cloud providers(GCP, AWS) and privately owned resources(NCSU VPN). This autoscaling service would spawn new servers on demand and shutdown servers when not in use to keep costs low. Since the objective is to reduce costs while maintaining high availability, the Auto Scaling Service will be triggered when the performance metrics exceed/fall below a certain threshold which would be set by the client based on the service priority.

Implementation Architecture

VM Based Deployment:



Each Tenant would have a VPC in each cloud and these VPCs would be connected through the GRE tunnel. Each VPC would have a Load Balancer which would balance load within its VPC. However, there would be one master Load Balancer which would have a public endpoint. Any traffic intended for the Tenant would first come here. And this Master Load Balancer would distribute traffic between the multiple cloud environments and also load balance within its VPC. The load balancer in another cloud would load balance traffic in its VPC. We would use iptables for load balancing.

Each Tenant would have a controller VM/container in each cloud environment. The Controller would collect data on performance metrics to decide when, where and how much to scale while making the corresponding load balancers aware of every scaling activity.

We have decided to implement this design based on our current understanding and would refine the design further in future.

Related Work

Many popular cloud solution providers offer auto-scaling features for their infrastructure services. We are particularly interested in compute scaling services. We looked into the following ones to get a better understanding of the current industry standard:

Amazon EC2 Auto Scaling:

Amazon offers a comprehensive auto-scaling solution called Amazon EC2 Auto Scaling under the AWS Auto Scaling umbrella.

Functional features:

- Scaling is done for a collection of EC2 instances, called auto-scaling groups
- Has different configuration options like minimum/maximum/desired capacity.
- It uses hot/cold starting vms(spot instances and on-demand instances). This means unused vms are kept in a kind of hibernation mode instead of shutting down to allow rapid deployment when needed again.
- They also support auto-scaling in a VPC. In this case the EC2 instances are launched in a subnet.
- This service can monitor the health of running instances periodically and replace unhealthy ones.
- Policies:
    - Maintaining desired capacity - The ASG(auto-scaling group) maintains the specified number of active healthy instances in the ASG.
    - Manual Scaling(static): User manually updates the minimum/maximum/desired capacity settings and they are reflected in the ASG.
    - Scheduled scaling(static): User specifies the schedule of when to scale up/down.
    - On-Demand scaling(dynamic): Dynamic scaling based on different parameters like CPU utilization, memory consumption etc.
    - Predictive scaling(dynamic): Uses machine learning to supplement decision making in proactive/reactive scaling.
- Within dynamic scaling, following policies are supported:
    - Target tracking - maintains capacity according to a specified metric like 60% CPU utilization.
    - Step scaling: Scaling in increments as the threshold is breached.
    - Simple scaling: Scales in one go instead of steps/increments.
- We also looked into how AWS integrates auto-scaling with load balancing; particularly the network load balancer that reads packets TCP/IP or UDP/IP packets:
    - They have the load balancer act as the end-point to which the client connects. It routes traffic to a VPC.
    - The load balancer itself offers redundancy by having a shadow/mirror load balancer to take over in case the active one fails.
    - Load balancer can distribute traffic across availability zones.

- The auto-scaling itself can re-balance the instance distribution by launching new instances and the terminating older ones(these may have failed health checks, the availability zones had some issues, user changed availability zones/settings etc).
- Allows vertical scaling as well.
- AWS allows more than the maximum capacity to exist for a short period when it is replacing VMs in the auto-scaling group.

Management feature/ Other observations:

- Management can be done using browser-based console, language-APIs and SDK CLI console.
- Offers ability to create launch configurations/templates to ease creating Auto scaling groups.
- Offers ability to cap billing as per affordability.
- Users can customize how many On-demand instances make up the total initial capacity.
- Users are also able to customize the type of instances they want as per their processing/memory requirements, within an auto scaling group.
- Auto-scaling can span across Availability Zones, which are independent data centers. This gives this solution high availability.
- Auto-scaling doesn't span across multiple regions. This means poor availability if an entire AWS region goes down, although the chance of this happening is rare.
- Right now there is no way to customize capacity units for different instance types. Although this feature is on the roadmap.
- Feature and stability-wise we noticed AWS auto-scaling was quite ahead of its competitors.

Azure Virtual Machine Scale Sets

Microsoft added auto-scaling support in 2013.

Functional Features:

- Like AWS, has Availability zones to ensure high availability.
- Resource Groups equivalent of ASG.
- No support for providing a range of minimum and maximum capacity.
- Has built-in support for monitoring host metrics using Azure Diagnostic Extension. These metrics can be directly accessed without third-party tools(using AZure provided APIs).
- As of this writing supports 9 different metrics for determining scaling threshold. Any metric can be monitored on these dimensions: average, minimum, maximum, total, last, count
- Offers increase/decrease by count or percentage.
- Allows vertical scaling to increase VM instance capacity/size.
- The options are limited when choosing on the type of scaling policy:
    - static (scheduled scaling)
    - Simple dynamic scaling - in response to demand(reactive). No proactive auto scaling offered.
- Supports Virtual networks with subnets.

- Azure also offers Availability sets that are different from Scale sets. In an Availability set, the VMs are discrete, each with its own namespace and properties.

Management Features/ Other observations:

- Management can be done using Azure portal, Azure Powershell, Azure CLI, Azure template.
- Has Azure Monitor for VMS to allow users to automate the collection of important CPU, memory, disk, and network performance metrics.
- Allows simulation of user traffic for testing using Availability Tests.
- Doesn't support creating images from VMs in a scale set.
- Azure doesn't allow assigning public IPv6 addresses to VMs in a scale set.
- It over-provisions VMs when launching a scale set to compensate for failure during the VM spin up. But this can lead to issues when the application cannot handle extra VMs.
- Azure has a limit of 1000 VMs per scale set.

Google Compute Engine

Google came out with its auto scaling offering in 2014.

Functional Features

- Google Compute Engine(GCE) offers auto-scaling as a feature of what it calls managed instance groups, an equivalent to auto-scaling groups in AWS and Resource Groups in Azure.
- It too offers the ability to use templates to create the instance groups using *instance templates*.
- When a new VM is loaded, there is a configurable *cooldown* period of boot-up to allows the VM to properly finish initialization.
- It also offers a *stabilization period,* where the scale-in happens based on a moving window of peak load in the last 10minutes. As per our understanding this feature can't be opted out of.
- It allows configuration of policies for scaling based on CPU utilization, traffic usage and other GCP(Google Cloud Platform) StackDriver metrics.
- This service also offers the possibility of a tighter integration of the auto-scaler and the load balancer. This feature is offered as the HTTP(S) load balancing policy for the auto scaler. Under this, the auto-scaler can scale in/out based on the network load on the system as measured by the load balancer. However, this means that auto scaling with this poiilcy can't work when there is a backup pool of instances. Specifically, when scaling in, load balancer health check start failing and as a result it redirects the traffic to the back up pool. This causes the main instance pool utilization to go to 0 and this could cause the auto-scaler to scale in much more than required.
- No support for scheduled scaling, step scaling or simple scaling. It also doesn't offer termination protection for EC2 instances.

Management features / Other observations

- GCP offers auto-scaling Charts as a visual tool to monitor auto-scaling activity.
- One can use the GCP web console, Google Cloud CLI and language APIs.
- While GCP does support proactive instance distribution it doesn't support proactive scaling. This means the redistributions across multiple zones occurs after the scaling is done. This results in a two step process and leads to higher VM load times and potentially more work.
- A lot of the features are in beta. This can be attributed to the fact that this service is the lates of the three discussed so far.

Across the board we noticed that while there are easy options for getting started quickly, the process to get the right setup can be tedious due to the amount of configuration involved.

Functional Features:

Autoscaling Policies which we would like to implement are

1) Static Autoscaling Policy:
    a) The User can predefined at what time and for how long a certain number of VMs are required.
2) Dynamic Autoscaling Policy:
    a) The user can set the min and max number of VMs which can be deployed.The user will also set the avg threshold value for certain metrics (CPU, memory etc) over a time interval (5min,10min,15min etc). Autoscaling would be done to ensure the user always gets the required resources.
    b) The user can choose either/both Proactive Autoscaling and Reactive Autoscaling:
    c) Proactive Autoscaling:
    Proactive Autoscaling is done by analyzing the metrics set by the user. If those metrics utilization is increasing/decreasing fast and closer to the threshold then Autoscaling is done beforehand to ensure optimal resource allocation
    d) Reactive Autoscaling:
    Reactive Autoscaling is done when the metrics exceed/fall below the threshold. Now to consider how many VMs to scale in/out.

$$\text{Scale up} = \frac{[\text{load avg of metric} - \text{threshold}]}{(\text{threshold})} * (\text{current number of VMs})$$

$$\text{Scale in} = \frac{[\text{threshold} - \text{load avg of metric}]}{(\text{threshold})} * (\text{current number of VMs})$$

Eg: Let us say the user has 9 VMs running and has set the 5 load min avg threshold for CPU usage as 60%. And current 5min CPU load avg is 80% for all the VMs so then the autoscaling action would create [(80-60)/60]*9 = 3 new VMs.

e) In our Service we are considering that each VM created has a fixed set of standard specifications (CPU, Memory etc).

3) Custom Autoscaling Policy:
   a) The user can create his own policy by incorporating both Static and Dynamic Auto Scaling Policy.
   b) Whenever there is a conflict between the Static and Dynamic Auto Scaling Policy the min number of VMs for Dynamic Autoscaling would be to changed to be equal to the number of VMs required in the Static Autoscaling Policy.

Another issue which needs to be handled is *where* to scale.Since our architecture is a MultiCloud VPC environment. We would allow the user to set the scaling preference for each Cloud which could be based on cost, service reliability etc. When an Autoscaling action is triggered we would scale up in the Cloud Environment with highest priority while it still has resources available and when it cannot support additional VMs, move on to the next in that priority order. And any Scale in action would be done in the opposite order, such that VMs in the least preferred Cloud Environment are shut down first.

One important feature which we would like to consider is the amount of buffer time which should elapse before the scaling is done again after autoscaling once. If the buffer time is too long then there is a chance that the user would be deprived of the required resources, but if it is too short there is a high chance of overprovisioning resources at least for a short time. So we would like to allow the user to set the buffer time depending on how critical the load is. The default value would be around 2 mins.

Also before a VM is being shutdown we would ensure a cool-down Period during which no new requests would be sent to that VM. This would ensure graceful shutdown of the VM without leading to any disruption in service.

Management Features:

Log Management

The solution would enable central storage of daily system logs from all hosts. Customer's administrator would be able to fetch these logs in .csv format and analyze them for the purpose of troubleshooting and fault management.

We are collecting all the CPU load metrics of all VMs the Tenant has and also maintain a list of all the autoscaling actions (scale out/scale in) performed for the tennant.

Configurability

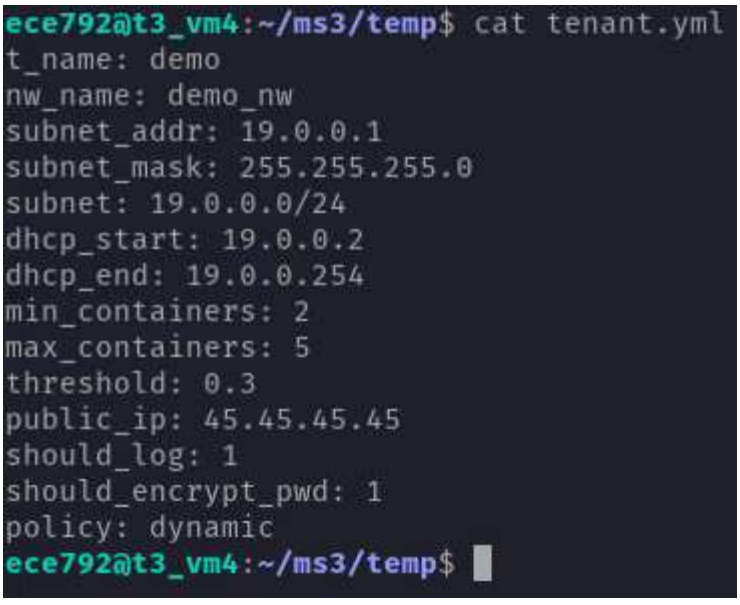
The solution would allow the customers'/tenants' administrators to configure the network attributes like subnet details and their application use-case specific autoscaling policy parameters such as:

- Type of metric to be used
- Threshold
- Cooldown buffer
- Maximum and Minimum number of VMs in the scaling group
- Specific time for scheduled scaling

Manageability

Customers would have the choice to modify their network and autoscaling policy at any time by changing the tennant.yml file.

Northbound:



The Northbound interface includes the tenant.yml file which is edited by each tennant. This file has all the subnet details,Tennant Name,CPU threshold, min number of VMs and max number of VMs. This gives the administrator the flexibility to configure the auto scaling logic.

Southbound:

The southbound interface consists of mainly four components:

1. Setup
2. Create Resources
3. Delete Resources
4. Collect Performance Metrics

Setup:
In setup we have 2 scripts for network and vm creation in roles/network-setup and roles/vm-setup directories. We also have another network.py file which is creating the namespaces, generating routes and performing load balancing.

***Create* Resources:**
Scale_up.py is a southbound script that creates a VM on the specified hypervisor. This script also updates a yaml file to report this change in this state. Load balancing rules have also been set up in this script to accommodate this new change. Previous iptables rules are flushed out and new iptable rules have been added.

***Delete* Resources:**
Scale_down.py is a southbound script that deletes a VM on the specified hypervisor. This script also updates a yaml file to report this change in this state. Load balancing rules have also been set up in this script to accommodate this new change. Previous iptables rules are flushed out and new iptable rules have been added.

Logic Layer

***Reading* Performance Metrics:**

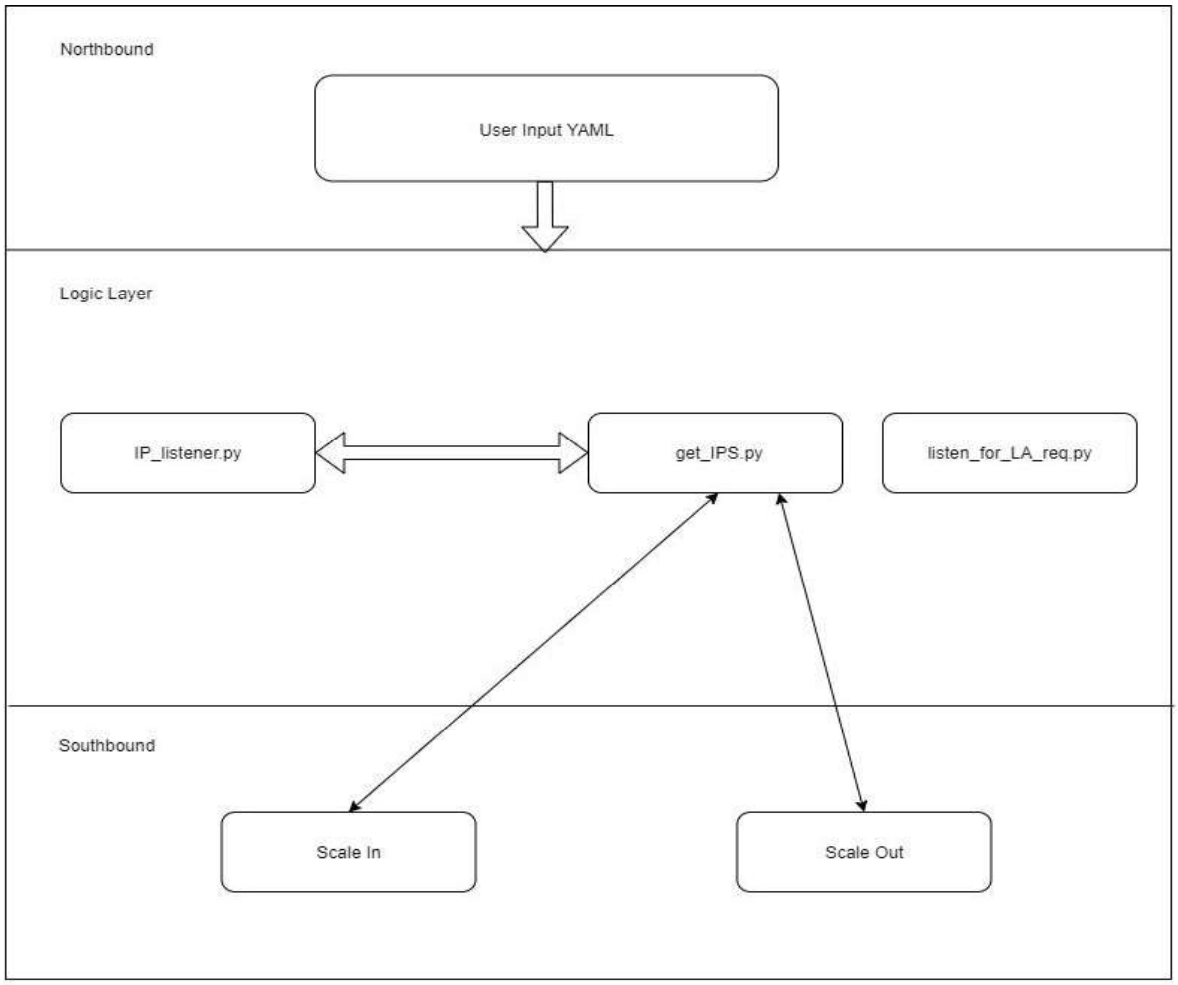
The controller VM polls the other customer VMs every 2 minutes for an output of the top command using the get_ips.py script. It calculates the average of all the currently active VMs and compares it with the user-defined threshold. If load is greater than threshold and at least one vm can be spawned, we trigger the south-bound script to create a new vm(scale_up.py). Similarly, we call scale_down.py to delete that last created VM.

All communications across all machines happen using TCP sockets.

***Update* IP Lists:**

The controller always maintains the lastest VM IPs. It polls the hypervisor to get the latest IP list for VMs belonging to the customer.

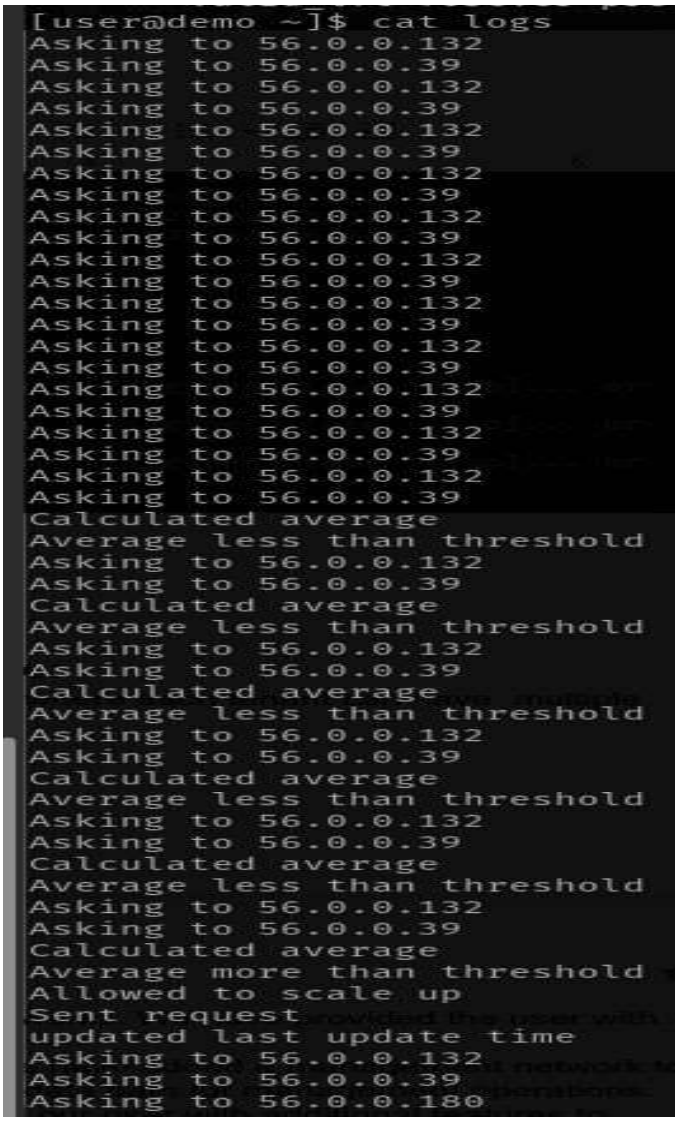
Load:
We have defined our load as all new ICMP traffic for the Tenant. We are load balancing the



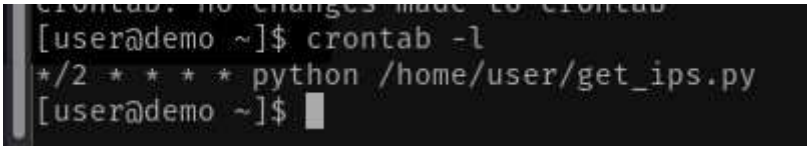
Logic Layer Separation

incoming traffic at the Transit Namespace equally between the hypervisors. Within each hypervisor the load is divided equally among all the existing Tenant VMs.

Controller monitoring in action:



Cron-job on controller:



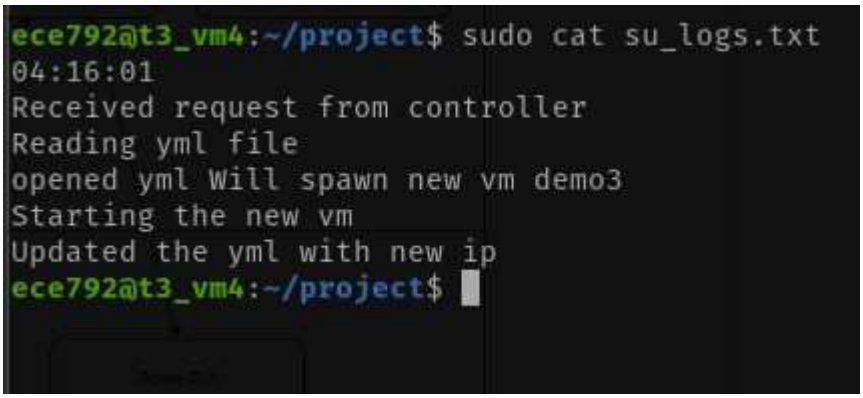
VMs listening for incoming poll from controllers:

Hypervisor listening for incoming requests for latest ips:

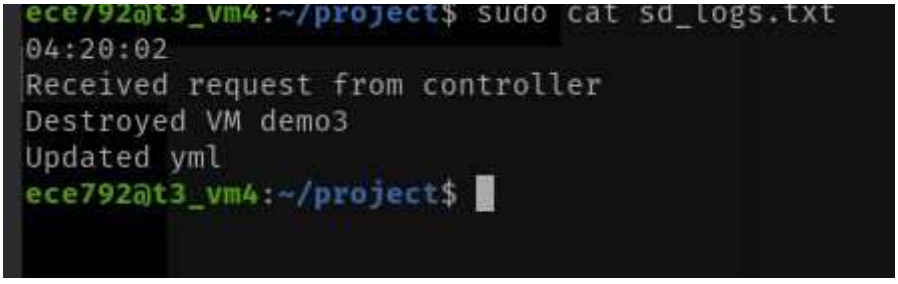
Hypervisor listening for scale up requests:

Hypervisor listening for scale down requests:

Scale up logs:



Scale down logs:



Improvements from previous milestone:

1. We have modified our configuration to better represent a VPC where each tenant can have multiple subnets.
2. Our VMs and containers are now part of the customer and the management networks separately. The management network is used by the controller to poll the customer devices and takes decisions on scale up/scale down.
3. We have provided the user with the option to connect his resources to multiple subnets.
4. Controller has now been removed from the customer's data path.We have added a management network to our VMs and containers. Previously we were using the data path for management operations.
5. To provide our project as a service we have provided our user with additional features to customize the project.
6. SouthBound and Logic Layer have been delineated.
7. We also made our project more service-oriented by allowing the customer to set preferences for features such as logging, password encryption, auto-scaling policy
8. We have given the user the capability to choose the policy he wants to inculcate in his service.

Container Specific Outputs:

Initial setup:

```
ece792@t3_vm4:~/ms3/temp$ sudo python initial_setup.py
Sending build context to Docker daemon  15.36kB
Step 1/7 : FROM ubuntu:18.04
 ---> 775349758637
Step 2/7 : RUN apt update && apt upgrade -y
 ---> Using cache
 ---> e2d027a58d2b
Step 3/7 : RUN apt install net-tools
 ---> Using cache
 ---> 14fc3a699045
Step 4/7 : RUN apt install -y iproute2
 ---> Using cache
 ---> 1537ad33c141
Step 5/7 : RUN apt install -y iputils-ping
 ---> Using cache
 ---> 6e6655be75bc
Step 6/7 : RUN apt install -y tcpdump
 ---> Using cache
 ---> eb620b258094
Step 7/7 : CMD ["/bin/bash"]
 ---> Using cache
 ---> 34c6e0053332
Successfully built 34c6e0053332
Successfully tagged ln:hw5
efcac988bf949f671b7914844f0c052426f7320fbf67fce536de0c5a9ff467ae
cac785fd1e835c854891220a14522a1993db2bfb82e1716b4cb4265b3bfba32d
b5e8d0860a4c1673e23da268eb1866e0b687873921a448c7b3483412a2b6c8c1
Created containers
Created veth pairs
Moved veths
Set devices up
added routes
172.17.0.4
172.17.0.5
sh: 1: kill: Usage: kill [-s sigspec | -signum | -sigspec] [pid | job]... or
kill -l [exitstatus]
sh: 1: kill: Usage: kill [-s sigspec | -signum | -sigspec] [pid | job]... or
kill -l [exitstatus]
sh: 1: kill: Usage: kill [-s sigspec | -signum | -sigspec] [pid | job]... or
kill -l [exitstatus]
ece792@t3_vm4:~/ms3/temp$
```

Scale up:

```
ece792@localhost:~/vaibhav$ sudo python container_scale_up.py T1_br "19.0.0.0/24"
Sending build context to Docker daemon  11.26kB
Step 1/6 : FROM ubuntu:16.04
 ---> 5f2bf26e3524
Step 2/6 : RUN apt-get update
 ---> Using cache
 ---> 9f6fb70e0512
Step 3/6 : RUN apt-get install net-tools
 ---> Using cache
 ---> 6de22e27f9cc
Step 4/6 : RUN apt-get install -y iputils-ping
 ---> Using cache
 ---> ff6a8c97c1ce
Step 5/6 : RUN apt-get install -y iproute2
 ---> Using cache
 ---> 679e2843134b
Step 6/6 : RUN apt-get install iputils-ping
 ---> Using cache
 ---> 4641b6273afc
Successfully built 4641b6273afc
Successfully tagged bottleimage:latest
360969c62782db4cbf71fdf993e2ad5fbc92a53ade72bf02283ac634d38c9185
Container has been successfully created
```

```
ece792@localhost:~$ sudo docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
360969c62782        bottleimage         "/bin/bash"         About a minute ago  Up About a minute                       T1_demo
_nw_3
0054a237a260        bottleimage         "/bin/bash"         4 hours ago         Up 4 hours                              T13
b411857b7c45        con1                "/bin/bash"         6 hours ago         Up 6 hours                              CON1
cb71550325e2        new:latest          "/bin/bash"         4 days ago          Up 4 days                               S7
caed515bc10a        new:latest          "/bin/bash"         4 days ago          Up 4 days                               S6
962e2d50b000        new:latest          "/bin/bash"         4 days ago          Up 4 days                               S5
b32a9f7164d6        new:latest          "/bin/bash"         4 days ago          Up 4 days                               S4
058483e5aa85        new:latest          "/bin/bash"         4 days ago          Up 4 days                               S3
eeae6599cc7a        new:latest          "/bin/bash"         4 days ago          Up 4 days                               S2
6b213f97ac3c        new:latest          "/bin/bash"         4 days ago          Up 4 days                               S1
b175dfd11566        new:latest          "/bin/bash"         4 days ago          Up 4 days                               X5
```

Scale down:

```
ece792@localhost:~$ sudo python cont_scale_down.py T1_br 19.0.0.0/24
T1_demo_nw_3
T1_demo_nw_3
VM has been deleted
```

```
ece792@localhost:~$ sudo docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
0054a237a260        bottleimage         "/bin/bash"         4 hours ago         Up 4 hours                              T13
b411857b7c45        con1                "/bin/bash"         7 hours ago         Up 6 hours                              CON1
cb71550325e2        new:latest          "/bin/bash"         4 days ago          Up 4 days                               S7
caed515bc10a        new:latest          "/bin/bash"         4 days ago          Up 4 days                               S6
962e2d50b000        new:latest          "/bin/bash"         4 days ago          Up 4 days                               S5
b32a9f7164d6        new:latest          "/bin/bash"         4 days ago          Up 4 days                               S4
058483e5aa85        new:latest          "/bin/bash"         4 days ago          Up 4 days                               S3
eeae6599cc7a        new:latest          "/bin/bash"         4 days ago          Up 4 days                               S2
6b213f97ac3c        new:latest          "/bin/bash"         4 days ago          Up 4 days                               S1
b175dfd11566        new:latest          "/bin/bash"         4 days ago          Up 4 days                               X5
ace27e77908e        new:latest          "/bin/bash"         4 days ago          Up 4 days                               X4
8f17e58daf3e        new:latest          "/bin/bash"         4 days ago          Up 4 days                               X3
8907e7f8bcac        new:latest          "/bin/bash"         4 days ago          Up 4 days                               X2
5cbd02e84e1b        new:latest          "/bin/bash"         4 days ago          Up 4 days                               X1
07ddfcf294b3        new:latest          "/bin/bash"         4 days ago          Up 4 days                               B6
```

As we can see that the last created container was deleted when we ran the vm.

Future Work:

We will move the infrastructure from VM based deployment to Container Based deployment
We have currently implemented only Reactive Autoscaling and would like to implement other policies such as Static AutoScaling, Proactive Autoscaling.
We would like to create logs for each Tenant to maintain a list of all the autoscaling actions and the load averages for each VM to get better manageability.
We would also like to improve upon our infrastructure design

References

- AWS autoscaling resources [https://aws.amazon.com/ec2/autoscaling/resources/]
- Azure autoscaling documentation [https://docs.microsoft.com/en-us/azure/virtual-machine-scale-sets/overview]
- Google Compute Engine Autoscaling documentation [https://cloud.google.com/compute/docs/autoscaler/]
- How to create snapshot in Linux KVM VM/Domain [https://www.cyberciti.biz/faq/how-to-create-create-snapshot-in-linux-kvm-vmdomain/]