Fitness-aware Brokerage of Hosted Containerized Environments

By Ishani Ghose

Introduction

Containers are increasingly becoming vehicles for deploying production workloads. However, containerization of mission-critical workloads is not yet main stream. Virtual machines are often the environment of choice to host SLA'ed workloads. The project works on a method to host SaaS application in a cost effective manner by engineering their dynamic deployment on best-fit container hosting environments.

Problem Statement

Internal IT departments of industries are moving to an Everything-as-a-service (XaaS) model, where in a cloud broker (or in general, a digital market place) maintains a portfolio of IT services. There is a for- a-smart-fulfilment engine within the cloud that automatically deploys a chosen catalogued IT service into a best-available-fit-container out of a set of available container engines hosted on various supported underlying clouds.

More broadly, the problem statement is to supply Best-fit Container Deployment as a service to applications that are a part of a brokers's Saas catalogue, in an on demand, pay as-you-go manner.

General Methodology

1.Expose the designed offering as part of a SaaS catalogue in a digital market place such as that of a cloud broker.

2. Implement an 'integration agent' that programmatically containerizes the catalogue element and places it under the best-available-fit container engine on underlying cloud as:

i. Kubernetes on GCP

ii. Ec2 container services on Amazon Cloud

iii. Docker Swarm Engine on IBM Softlayer Cloud

Experimental Work

The following modules perform the functions as described below:

1.CapacityPlan/Benchmarking/Scalability/

i.ECS_scale.py: Measures the time taken to scale the number of containers on an Ecs cluster of 10 t2.micro, t2.medium, m4.xlarge instances from 1to 5 containers and 1 to 10 containers.

ii.SL_scalability.py: Measures the time taken to scale the number of containers on a docker swarm cluster of 10 VMs from 1to 5 containers and 1 to 10 containers.

iii. gcp_scalability.py: Measures the time taken to scale the number of containers on a gcp cluster of 10 instances from 1to 5 containers and 1 to 10 containers.

2. CapacityPlan/Benchmarking/Speed/Linpack:

GCP_linpack.py,linpack.py,SL_linpack.py is used to run the linpack benchmark on an ec2,gcp instance and docker swarm engine to measure the number of GFLOPS of the container.

3.CapacityPlan/Benchmarking/Speed/Linpack/Apache:

GCP_linpack.py,linpack.py,SL_linpack.py is used to run the apache benchmark on an apache web server containerized on ec2,gcp instance and docker swarm engine to measure the time taken to complete specified number of requests.

4. CapacityPlan/Benchmarking/Results:

Contains all the files that were used to store the results while collecting data.

Forming the comparable container cluster node sets:

The results of the Linpack benchmark were used to form the three comparable container cluster node sets.

The sets formed were:

S[1] = 2-core(Sandy Bridge) 4G,GCP Asia2-core 4G,AWS Ireland t2.micro

S[2] = 4-core(Sandy Bridge) 4G, GCP Europe 4-core 3.6G, AWS Ireland t2.medium

S[3] = 8-core(Sandy Bridge)8G, GCP Europe 8-core 7.2G, AWS Ireland m4.2xlarge

Problems faced and how they were solved:

General problems faced:

1. Running the benchmarks from a geographically different location would require the use of the external network and due to a large number of hops on a congested network, the time taken to run the benchmarks was much higher for AWS, GCP than for IBM softlayer.

Hence while measuring the benchmarking time for Apache web server, the readings for GCP and ECS had to be taken by running a container that would run the benchmark locally for the server using the local ip address.

Problems faced while using AWS:

1. Connecting to the instance and continue executing commands on the instance.

Method used: boto.manage.cmdshell module

Problem:ec2 instances in clusters created using the ecs cli are not assigned public domain names by default due to which it was not possible to connect to the instance using the chosen method.

Solution:The Dns support attribute and Dns enable attribute has to be set to "yes" using the AWS console for the ec2 instances to be assigned a public domain name. Without a public domain name for the ec2 instances, the above method cannot be used.

The commands for aws cli given in the documentation are:

aws ec2 modify-vpc-attribute --vpc-id vpc-a01106c2 --enable-dns-support "{\"Value\":false}"

aws ec2 modify-vpc-attribute --vpc-id vpc-a01106c2 --enable-dns-hostnames "{\"Value\":false}"

However these commands did not work and the settings had to be changed manually.

Other methods for connecting to the ec2 instance programmatically:

1)ec2 Run command: https://aws.amazon.com/blogs/aws/new-ec2-run-command-remote-instance-management-at-scale/

2) Using the fabric module

2.Scaling containers on Ecs cluster:

Problem: By default, one task is assigned to one ec2 container

Solution: Running multiple tasks on one ec2 container:

i. Use an application load balancer and change the following settings:

a)Set networking mode to "bridge"

b) Set the "host port" to 0 to make it dynamic.

c)While defining the health check in the ALB task group , omit the health check port so that it defaults to using the dynamic port.

3.Clearing the cloud stack

1)All resources created by ecs cli have to be deleted using the ecs cli. The aws console cannot be used to delete these resources.

If the aws console is used, the auto scaling group does not get deleted along with the cluster and has to be deleted separately using the console else you will see new container instances being spawned automatically every time you delete them.

2) Ecs cli can be used to delete resources only created by the ecs cli.

Eg: If any resource has been created using boto3 functions, they have to be deleted using boto3 functions.

Results

Linpack benchmark results:

Experimental Results on Cluster Node Composition-Linpack Benchmark
S[1] = 2-core (Sandy Bridge) 4G, GCP Asia 2-core 4G, AWS Ireland t2.micro
S[2] = 4-core (Sandy Bridge) 4G, GCP Europe 4-core 3.6G, AWS Ireland t2.medium
S[3] = 8-core (Sandy Bridge) 8G, GCP Europe 8-core 7.2G, AWS Ireland m4.2xlarge
SP factor: 0.98

	Amazon Web Services Compute T-Shirt Size	Google Compute Engine Compute T-Shirt Size	IBM SoftLayer Compute T-Shirt Size	GFLOPS (Avg)
S[1]	25.40GFLOPS, 0.226	26.58GFLOPS, 0.236	34.86GFLOPS, 0.310	28.95GFLOPS, 0.257
S[2]	72.89GFLOPS, 0.649	72.45GFLOPS, 0.645	72.96GFLOPS, 0.650	72.76GFLOPS, 0.648
S[3]	109.4GFLOPS, 0.974	101.9GFLOPS, 0.907	107.61GFLOPS, 0.958	106.30GFLOPS, 0.94633

Apache Benchmark Results:

Load	Apache Web Server Brokered to Google Container Engine (GCE) [Average response times from 15 deployments over 1 month]	Apache Web Server Brokered to Amazon EC2 Container Services (ECS) [Average response times from 15 deployments over 1 month]	Apache Web Server Brokered to Docker Engine on IBM SoftLayer (SL) [Average response times from 15 deployments over 1 month]	Apache Web Server Brokered Randomly Across GCE, ECS and SL [Average response times from 15 deployments over 1 month]	Apache Web Server Brokered Using our Best-fit Algorithm Across GCE, ECS & SL [Average response times from 15 deployments over 1 month]
100 Connections <i=1>	0.0216 secs	0.022 secs	0.0186 secs	0.019 secs(SL)	0.019 s(SL)
1000 Connections <i=1>	0.249 secs	0.253 secs	0.194 secs	0.248 secs(GCP)	0.195 s(SL)
10K Connections <i=2>	1.212 secs	1.620 secs	0.793 secs	1.636 secs(ECS)	0.79 s(SL)
100K Connections <i=3>	7.772 secs	7.218 secs	7.38 secs	7.218 secs(ECS)	7.218 s(ECS)

Scalability Results:

scalability quotient = 0.02

ecs	scale to 5 containers	scale to 10 containers
t2.micro	1.92s, 0.0188	2.95s, 0.018
t2.medium	12.8s, 0.012	13.45s, 0.011
m4.2xlarge	30.4s, 0.001	31.43s, 0.0003

gcp	scale to 5 containers	scale to 10 containers
2-core 4G	4.8s, 0.017	3.42s, 0.017
4-core 3.6G	12.8s, 0.012	11.86s, 0.012
8-core 7.2G	19.2s, 0.008	20.05s, 0.007

metal	scale to 5 containers	scale to 10 containers
2-core	3.44s, 0.016	6.92s, 0.015
4-core	4.02s, 0.017	8.73s, 0.014
8-core	6.31s, 0.015	13.15s, 0.011

Calculations for fitness quotient:

SP factor = 0.98
SC factor = 0.02

AWS:
S[1] : 0.98 * (25.40/110) + 0.02 * (32-2.95)/32
S[2] : 0.98 * (72.89/110) + 0.02 * (32-13.45)/32
S[3] : 0.98 * (109.4/110) + 0.02 * (32-31.43)/32

GCP:
S[1] : 0.98 * (26.58/110) + 0.02 * (32-3.42)/32
S[2] : 0.98 * (72.45/110) + 0.02 * (32-11.86)/32
S[3] : 0.98 * (101.9/110) + 0.02 * (32-20.05)/32

SL:
S[1] : 0.98 * (34.86/110) + 0.02 * (32-6.92)/32
S[2] : 0.98 * (72.96/110) + 0.02 * (32-8.73)/32
S[3] : 0.98 * (107.61/110) + 0.02 * (32-13.15)/32

Fitness Quotient Values:

	Amazon Web Services Compute T-Shirt Size	Google Compute Engine Compute T-Shirt Size	IBM SoftLayer Compute T-Shirt Size	GFLOPS (Avg)
S[1]	0.243	0.253	0.325	SL
S[2]	0.660	0.657	0.664	SL
S[3]	0.974	0.914	0.969	ECS

Conclusion

The data required to implement the Container Fitness-awareness Algorithm has been collected.

From the results, IBM softlayer is the choice for S[1] and S[2] and Ecs for S[3]. However these results are dynamic.

The next steps would be to:

i. implement asynchronous algorithm thread

ii.Implement synchronous algorithm thread

iii. Add the cognitive algorithm described