Autoscaling using Docker Swarm

Group 14
Madhushree Nijagal mnijagal@wisc.edu
Priyavarshini Murugan pmurugan@wisc.edu
Tanisha Hegde tghegde@wisc.edu

Overview

We aim to perform autoscaling using Docker Swarm and observe the trends it follows when performing autoscaling. To achieve this, we have auto-scaled based on:

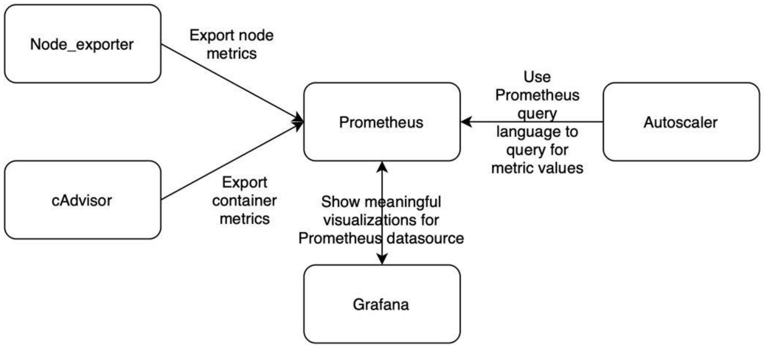
- HTTP Requests
- CPU Load
- Memory Consumption

To plot the graphs, we use multiple images from docker hub that help poll the data generated by docker.

The official images used for the project and their use is summarized below:

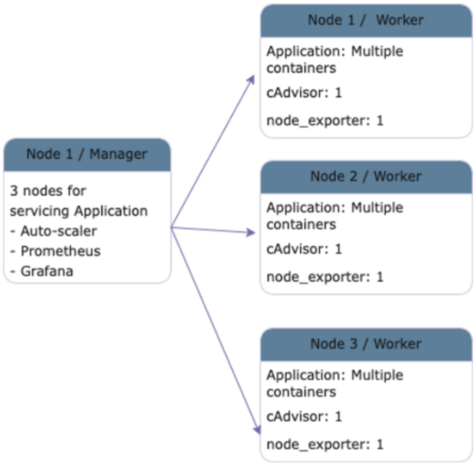
Image	Use
Prometheus	Query metrics value, used for autoscaling
cAdvisor	Export container metrics
Node_exporter	Export node metrics
Grafana	Plot graphs and create dashboards

The Prometheus service is the central point of the services, Node_exporter and cAdvisor collect and pass data to. The Autoscalar service uses these metrics from Prometheus using PromQL (Prometheus Query Language) to help swarm decide to scale up or scale down based on threshold values. Grafana scrapes these metrics based on a scrape interval to generate graphs. The diagram summarizes the interaction of all the services.


Node_exporter
Export node metrics
Prometheus
Use Prometheus query language to query for metric values
Autoscaler
cAdvisor
Export container metrics
Show meaningful visualizations for Prometheus datasource
Grafana

Design and Architecture

The docker swarm architecture consists of 3 nodes, each capable of running multiple containers. The manager node orchestrates the working of the worker nodes. It also behaves like a worker node by taking load when running the application. The manager node uses the metrics received by Prometheus to scale up / down based on the autoscaling logic. This requires the Prometheus image to reside only on the manager node. The cAdvisor and node_exporter export container and node specific details respectively. This requires each of these services to be deployed on each node.



Applications for load Generation

We tried autoscaling based on CPU Load, memory utilization and HTTP Requests. We required different applications that would stress the containers individually on these parameters.
We used available images –

1. Progrium/stress - stress the memory utilization of a container.
2. Apache Benchmark - send multiple Http Requests
3. Own Script - stress CPU load.

1. Progrium/stress

Progrium/stress generates load by performing multiple memory allocations (similar to malloc).
Example:

```
$ docker run -d --name stress-test progrium/stress --vm-bytes 512M --vm-keep -m 1
```

here stress-test is the container that generates load by allocating 512MB of virtual memory (--vm) for 1 minute (-m).

We generated a load of 50GB for 1min by passing these values to the docker-compose file for the progrium/stress service.

2. Apache Benchmark

Apache Benchmark generate HTTP request load by specifying the target URL and the number of requests to send.

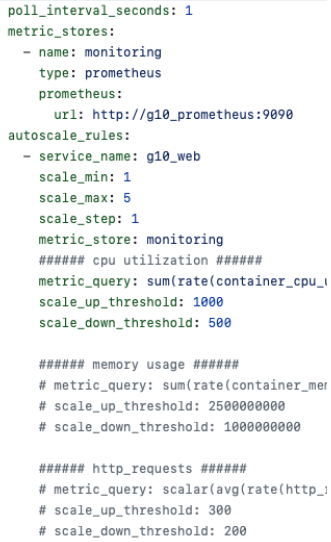
```
$ docker --rm ab -n 500000 -c 1 -t 600 http://127.0.0.1:8080/
```

We generated 50k requests in 10seconds

3. Own Script

To increase the load of CPU we performed a heavy computation (e.g., sqrt(64*64*64*64*64)) for a specified interval of time.
To reduce the load, we halt the computation in the middle of the interval.
We have used two scenarios:
Scenario 1: Script to generate load to increase CPU utilization by 30% for 3 mins
Scenario 2: Script to generate above CPU load for 1 min and reduce it for the next 2 mins

Autoscaling Logic



```
poll_interval_seconds: 1
metric_stores:
  - name: monitoring
    type: prometheus
    prometheus:
      url: http://g10_prometheus:9090
autoscale_rules:
  - service_name: g10_web
    scale_min: 1
    scale_max: 5
    scale_step: 1
    metric_store: monitoring
    ###### cpu utilization ######
    metric_query: sum(rate(container_cpu_
    scale_up_threshold: 1000
    scale_down_threshold: 500

    ###### memory usage ######
    # metric_query: sum(rate(container_me
    # scale_up_threshold: 2500000000
    # scale_down_threshold: 1000000000

    ###### http_requests ######
    # metric_query: scalar(avg(rate(http_
    # scale_up_threshold: 300
    # scale_down_threshold: 200
```

We are fetching the metrics, `scale_up` and `scale_down` values from the `autoscaler.yml` along with the `metric_query`. We run the `metric_query` to get the current CPU Utilization / Memory Utilization / Http Requests and compare them against the threshold value. We scale up or down the number of running replicas using docker_client.scale_service from the `DockerAPIBasedClient`library.

1. CPU Load

Metric_Query: sum(rate(container_cpu_usage_seconds_total{container_label_com_docker_swarm_service_name='g10_web'}[5m])) BY (container_label_com_docker_swarm_service_name,instance)*100

Scale up threshold: 30% (total of 40 CPUs)
Scale down threshold: 12.5% (total of 40 CPUs)

2. Memory Utilization

Metric_Query: sum(rate(container_memory_usage_bytes{container_label_com_docker_swarm_task_name=~' .+'}[30s]))BY(container_label_com_docker_swarm_service_name,instance)

Scale up threshold: 2.5GB (of 187GB)
Scale down threshold: 1GB (of 187GB)

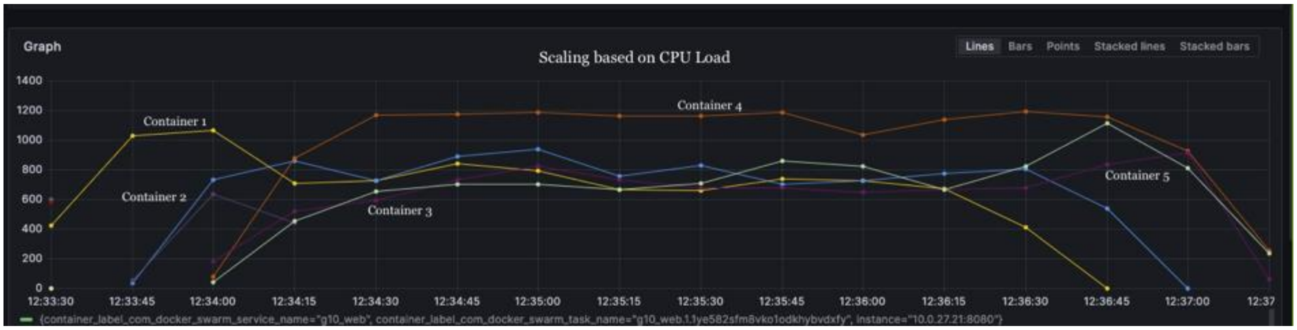
3. Http Requests

Metric_Query: scalar(avg(rate(http_requests_total{job="web"}[30s])))
Scale up threshold: 300
Scale down threshold: 200

Observations

1. CPU Load

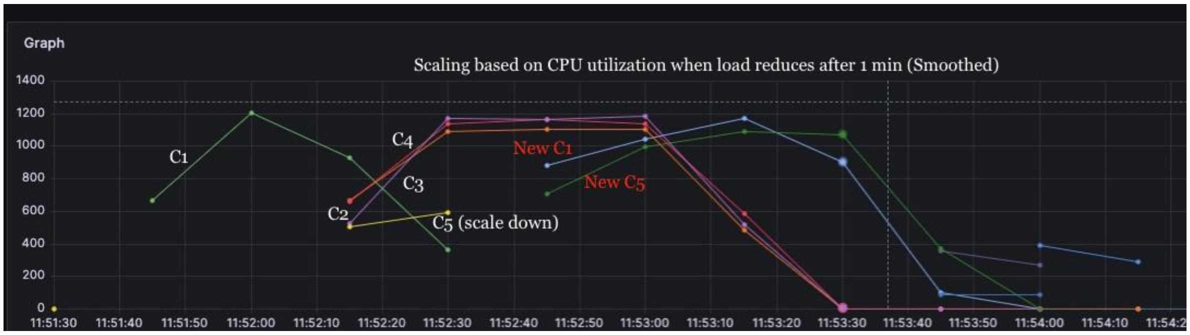
a. Scenario 1: CPU Load generated for interval of 3 minutes
Max Containers: 5
Scale up: 30% (1200)
Scale down: 12.5% (500)



We observed that as the CPU load increases for Container 1 and reaches the threshold of 1200%, Container 2 is created (timestamp 12:33:45). As Container 1 and Container 2 still are overloaded Container 3, Container 4 and Container 5 are created (timestamp 12:34:60). Since the limit of scale max is 5 no new containers were created after this. It can be observed that as one container is always overloaded, there is no scale down between the load generation that is created. At the end of 3 mins, all containers gradually scale down as the job ends.

b. Scenario 2: CPU Load generated for interval of 1 minute followed by 2 mins sleep

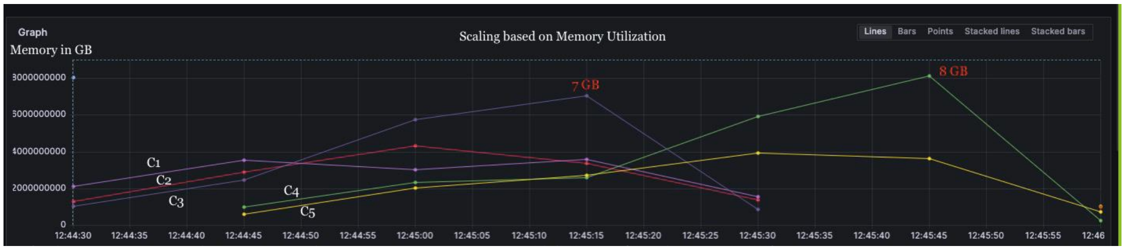
Max Containers: 5
Scale up: 30% (1200)
Scale down: 12.5% (500)



With reference to scenario 1, as a clear scale down in the middle of the job was not visible, we created scenario 2. Since load is generated for 1 min followed by 2 mins of no load, we observe that C1 begins and ends ~1m. As threshold is reached C2, C3, C4 and C5 are created. C1 and C5 however are scaled down as the load reduces than the minimum value. New containers (New C1, New C2) are recreated as C2, C3 and C4 are overloaded.

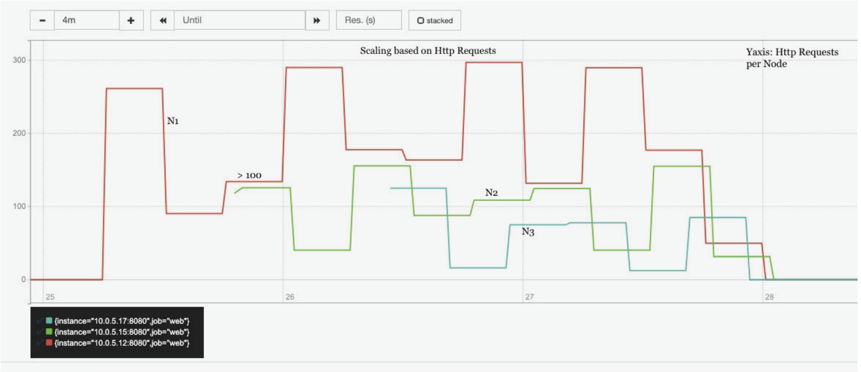
2. Memory Utilization

Max Containers: 5
Scale up: 2.5GB
Scale down: 1GB



Before plotting the graphs for autoscaling due to memory utilization our assumption was that a graph, similar to CPU load should be observed. We however, noticed that even though a few containers had crossed the maximum threshold, new container were not created. This can be seen in the graph where C4 and C5 reach to 7G and 4GB respectively but no new containers were created.

3. Http Requests



For Http Requests we plotted the graph node wise as our query measured the total http requests per node to scale up/down. We see that after reaching the maximum threshold load a sudden spike in the load is seen for N1 for a few milliseconds but reduces as load is distributed among N1 and N2. New node N3 with similar trends where a similar spike followed by a balance is observed.

Conclusion

We noticed that docker swarm auto scaling behavior is different for various types of metrics (CPU, memory, HTTP load). We also observed that the Docker swarm handles load balancing as a result of auto scaling. Secondly, we noticed that some of the containers were always loaded, we suspected that it could be due to the nature of our load generation. We also plan to further learn about how auto scaling works at the container level and which processes are responsible to manage the swarm.