Dynamic Service Scaling Using Kubernetes

Fernando Rego
Faculty of Engineering
University of Porto
Porto, Portugal
up201905951@edu.fe.up.pt

José Costa
Faculty of Engineering
University of Porto
Porto, Portugal
up201907216@edu.fe.up.pt

***Abstract*—It is common for web services to not be under constant load levels throughout their life cycle. The load may vary with the time of the day, and time of the year among other unspecified and many times unknown a priori variables. Web service providers provide scaling based on Kubernetes deployments in order to make scaling easy and repeatable. However, these deployments rely on externally provided services to function correctly and bare-metal clusters cannot assure all of them. Scaling implemented by Kubernetes is also very limited regarding data sources and metrics that it can use, not being able to scale purely on HTTP-based metrics. This paper discusses a possible bare-metal Kubernetes cluster configuration with support for scaling HTTP workloads using HTTP metrics from a Prometheus metrics database, gathered from a Traefik proxy server, in the context of a server farm of an organization.**

I. Introduction

With the advent of containerization and cloud computing, web deployments have never been easier. Kubernetes[1] emerged as the main project to support containers in a cloud environment, providing an easy framework for web service provider companies to offer infrastructure as a service (IaaS) products, providing scalable hosting solutions according to service loads. Before, the solution to scale a service horizontally was to set up another machine and add it to a load balancer, which wasn't easily automatable and reproducible.

In this paper, we discuss a possible configuration for a bare-metal Kubernetes cluster inside an organization network. The organization has to ensure access to the internal servers only to the clients inside the organization network and to ensure that both collaborators and external users can access the services available at the server farm, where the Kubernetes cluster handles all requests.

Kubernetes engines made available by web service providers use external services to address tasks like load balancing and exposing services to the outside of the cluster.

Currently, bare-metal clusters need tools like MetalLB[2] to circumvent this issue. MetalLB internally uses an L2 load balancer to expose the services available at the cluster; what effectively happens is that the network interface on the node acts as if it has various IPs assigned, making the services available on the node network.

The scaling implemented by Kubernetes, Horizontal Pod Autoscaler (HPA)[3], is limited to the metrics available at the internal cluster metrics server. These metrics don't provide enough information to scale on HTTP workloads. To gather the rate of HTTP requests, a proxy service with support to Kubernetes ingresses needs to route the requests and gather this metric. In this implementation, Traefik[4] was used to act as the proxy and later, a Prometheus[5] server scrapes the metrics from Traefik. Then an autoscaler with support for Prometheus queries, KEDA[6] in this implementation, calculates the rate of requests.

II. Virtual Machines

To develop this project three different virtual machines were used in order to divide work and simulate a server farm network (Virtual Machine B) and a private organization (Virtual Machine C). Virtual Machine A was used only with the purpose of simulating external clients that request the services provided by the server farm and acting as a pseudo Internet Service Provider (ISP). Figure 1 represent the three virtual machines connected to the *192.168.88.100/24* network.

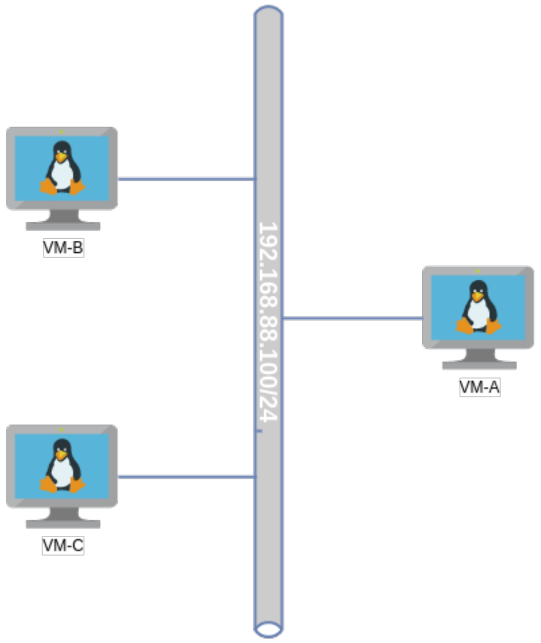


Fig. 1. Virtual Machines

III. Instrumental tools

This section aims to present and explain tools outside of the scope of the main problem that were used for testing purposes and to ease the set-up process of the experiment.

A. Docker and Docker compose

Docker[7] was the container engine used to emulate hosts in various networks. Docker compose was used to describe and configure each of the networks and respective services, avoiding tedious and more error-prone script files.

B. Kind - Kubernetes in Docker

A single Kubernetes engine can be used per host. As only one virtual machine with limited resources was available to emulate the server farm where the Kubernetes cluster provides services, Kind[8] was used. Kind uses containers with custom images to act as nodes in a Kubernetes cluster.

C. Locust

Locust[9] is a Python-based distributed load-testing tool. It has a master-worker architecture, allowing several workers with different network locations to be controlled from a single easy-to-use Web GUI. Load tests are defined in code, so can be written to be as complex as desirable, emulating user behaviours. In our simple test cases, we only have used *GET* requests to the available services.

IV. Internet Service Provider

Virtual machine A has a 2-fold objective. It serves as an ISP to the other virtual machines and also emulates external incoming traffic to the server farm, for load testing purposes.

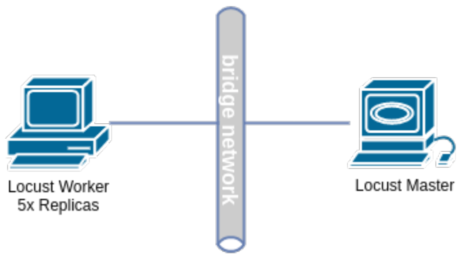


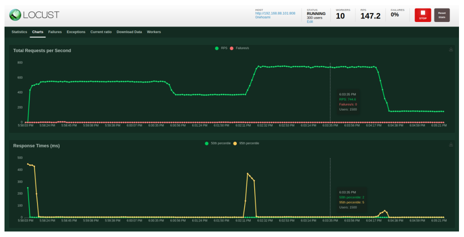
Fig. 2. Virtual Machine A

Figure 2 represent the bridge network created by Docker in the virtual machine to expose the Locust services to the network. This machine hosts 5 Locust workers and the Locust master instance.

Using *iptables* to rewrite network requests, the Locust master, deployed in this machine, and the Traefik service, deployed in the Kubernetes cluster in the server farm (Virtual Machine B), were exposed to the external network, making them accessible from upstream hosts, easing testing.

- Locust master: http://192.168.109.156:8089/
- Traefik service: http://192.168.109.156:8080/dashboard/

The figure 3 shows the graphical interface of locust when requests are being made to a certain service.



HOST http://192.168.88.101:8080/whoami
STATUS RUNNING 300 users Edit
WORKERS 10
RPS 147.2
FAILURES 0%

6:03:35 PM
RPS: 744.6
Failures/s: 0
Users: 1500

6:03:35 PM
50th percentile: 2
95th percentile: 5
Users: 1500

Fig. 3. Locust

V. Server Farm

The server farm network is abstracted as a docker network created by Kind. The virtual machine B, apart from hosting the Kubernetes nodes as containers, can be thought of as a router for the server farm network.

A. Kubernetes Cluster

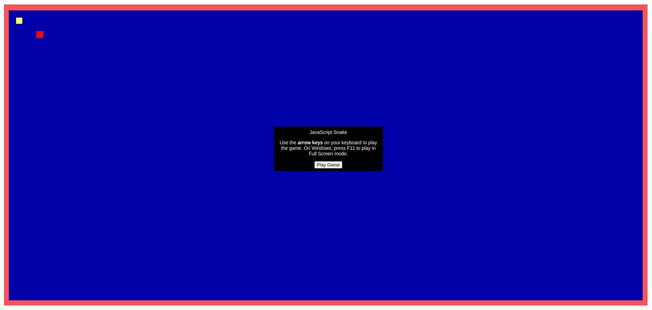
The cluster was configured with the MetalLB L2 load balancer so services configured inside the cluster could be exposed to the network. Note that Kind is being used to emulate the nodes, so exposing the services exposes them to the running containers. These need to be then exposed to the virtual machine itself so they're accessible from other locations.

As discussed in the Introduction, the Traefik proxy is needed to gather the number of HTTP requests being made to each service. Traefik is exposed to the outside world using the method described above.

Prometheus server was configured to scrape Traefik every 2 seconds. For debug purposes, it was also exposed to the virtual machine B network, allowing for queries through the Web GUI.

KEDA was also installed in the cluster. There is no global configuration for it, as the configuration is done on a per-service basis.

Then each service was configured using a deployment specification for describing a service pod; a service specification, describing the port mappings and deployment to use; an ingress, to map the URL that will be used externally and the target service; and a middleware to rewrite the URL so it is in the format expected by the deployed image. The KEDA configuration for each service was also used, so as to configure metrics source, Prometheus query to use, thresholds, pod number, timings and scale-down policies. As an example, the figure 4 shows the service *snake* which hosts a web application to play the traditional snake game.


JavaScript Snake

Use the **arrow keys** on your keyboard to play the game. On Windows, press F11 to play in Full Screen mode.

Fig. 4. Snake game

As requests reach the Traefik proxy, they are routed to the respective target service. Prometheus scrapes the proxy for the number of requests to each service every 2 seconds. KEDA checks the rate of requests in the last 2 minutes every 5 seconds, making use of the values stored by Prometheus. When there are more than 100 requests per pod the KEDA trigger is activated, and another pod is spun up, up to 10 pods, allowing for a rate of 1000 requests before stopping scaling up. The cluster chooses what pod handles each request. If for more than 10 seconds the rate of request per pod is lower than the threshold, a pod is spun down every 5 seconds.

Figure 5 depicts how the various services communicate inside the Kubernetes cluster.

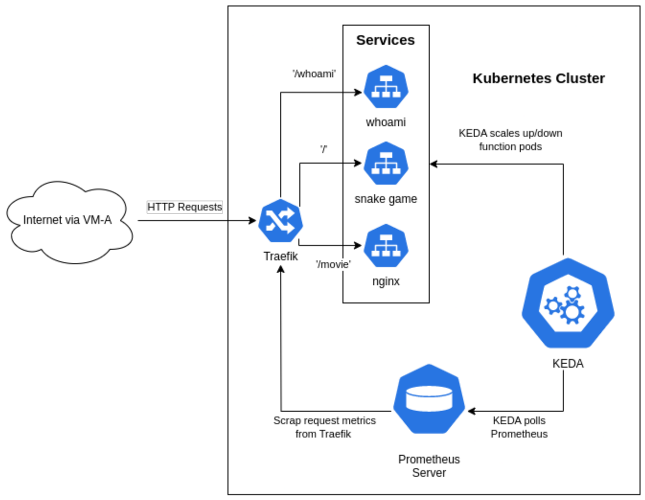


Fig. 5. Virtual Machine B

VI. Organization's Network

Finally, Virtual Machine C represents a private organization network. The organization is composed of one router, connected with two private networks, and one public network that communicates with the ISP (Virtual Machine A). The two private networks correspond to the client network and the server network.

- Client network - *10.0.1.0/24*
  - The clients' network is connected to the router and clients of the private organization. The main goal is not only to provide internet services but also the private services and resources of the organization that are present in the organization's servers.
- Servers network - *10.0.2.0/24*
  - Five Nginx servers are providing a simple web service to the clients, and the load balancer redistributes the requests across the servers so that the load on each server is balanced. These servers are only accessible by the clients inside the organization network.

Besides the organization network, 5 Locust Workers were added to Virtual Machine C. These workers connect with the Locust Master present in Virtual Machine A in order to distribute the workload between workers.

Figure 6 visually represents both the organization network and the Locust workers present in Virtual Machine C.

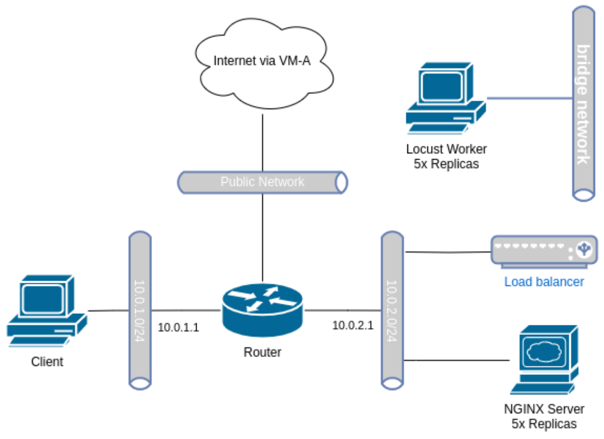
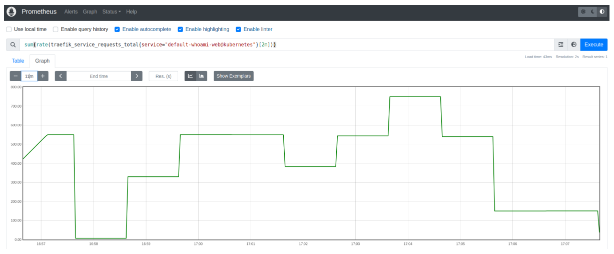


Fig. 6. Virtual Machine C

VII. Synthetic Load Testing

The service scale-up processes behave as expected: when the threshold is met, the trigger is activated and a new pod instance is created. On the other hand, during scale-down, some of the requests that were already routed to the pod leaving the service are lost, which is undesirable, as a user could think the service is unavailable. A mechanism to detect these failures and silently retry these requests should be explored, resulting only in more latency instead of a request failure.

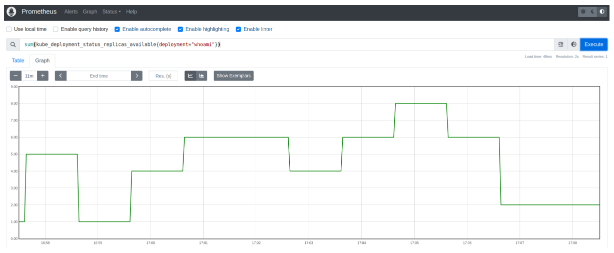
Figure 7 represents the rate of requests to a service, *Whoami* service in this case. The calculated metric is used to scale up or down pods of the service at running time. When comparing with figure 8, both graphs are very similar, which shows that the number of pods will grow with the rate of requests.



```
sum(rate(traefik_service_requests_total{service="default-whoami-web@kubernetes"}[2m]))
```

Load time: 43ms Resolution: 2s Result series: 1

Fig. 7. Whoami service: Number of requests



```
sum(kube_deployment_status_replicas_available{deployment="whoami"})
```

Load time: 48ms Resolution: 2s Result series: 1

Fig. 8. Whoami service: Number of pods

VIII. Future Work

To further evolve the dynamic service scaling, real-world timings, the composition of different metrics and the usage of AI models should be put into practice in order to correctly scale up or down the services. Another important feature that can help to correctly evaluate is monitoring the important metrics for the system. This can be achieved with tools like Grafana [10] that provide a user-friendly graphical interface with programmatic real-time graphs.

Another aspect that would bring this configuration closer to real-world scenarios is the usage of humanly recognizable URLs. For this experiment only the machines' IPs were used, relying on matching the path part to disambiguate to the target service. Kubernetes ingresses and Traefik allow the usage of different host URLs to match different services, however, this would bring the need to also configure the DNS service so clients could correctly resolve these names.

IX. Conclusion

Kubernetes allowed a revolution in cloud hosting services and provisioning in general. From this experiment, it is clear to picture the benefits of using such solutions when compared to common server provisioning: flexibility, ease of use, fast reconfiguration, easy scaling, security and containerization of services.

It is also clear that Kubernetes were thought with cloud environments in mind, as bare-metal implementations lack support for some crucial features and are clearly second-class citizens in the Kubernetes world.

Implementation is available at https://github.com/Sirze01/feup-grs-autoscale.

References

[1] The Kubernetes Authors, *Kubernetes documentation*, kubernetes.io. https://kubernetes.io/docs/home/ (accessed Jun. 9, 2023).
[2] The MetalLB Contributors, *MetalLB concepts*, metallb.universe.tf. https://metallb.universe.tf/concepts/ (accessed Jun. 9, 2023).
[3] The Kubernetes Authors, *Horizontal Pod Autoscaling*, kubernetes.io. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (accessed Jun. 9, 2023).
[4] Traefik Labs, *Traefik documentation*, traefik.io. https://doc.traefik.io/traefik/ (accessed Jun. 9, 2023).
[5] Prometheus Authors, *Traefik documentation*, prometheus.io. https://prometheus.io/ (accessed Jun. 9, 2023).
[6] KEDA Authors, *KEDA Homepage*, keda.sh. https://keda.sh/ (accessed Jun. 9, 2023).
[7] Docker Inc., *Docker documentation*, docker.com. https://docs.docker.com/ (accessed Jun. 9, 2023).
[8] The Kubernetes Authors, *Kind homepage*, kind.sigs.k8s.io. https://kind.sigs.k8s.io/ (accessed Jun. 9, 2023).
[9] Locust Authors, *Locust Homepage*, locust.io https://locust.io/ (accessed Jun. 9, 2023).
[10] Grafana Labs, *Grafana Homepage*, grafana.com https://grafana.com/ (accessed Jun. 9, 2023).