Adaptive scaling of Kubernetes pods

David Balla, Csaba Simon, Markosz Maliosz
High Speed Networks Laboratory
Budapest University of Technology and Economics
Budapest, Hungary
{simon | balla | maliosz}@tmit.bme.hu

***Abstract*—Scalability is the ability of a resource or an application to be expanded to handle the increasing demands. One of the most emphasized aspects of cloud computing is the ability of scaling up or down. Scaling can be performed by either an operator or automatically. Autoscaling algorithms are getting more and more emphasized in the field of cloud computing. In this paper we propose an adaptive autoscaler, Libra, which automatically detects the optimal resource set for a single pod, then manages the horizontal scaling process. Additionally, if the load or the underlying virtualized environment changes, Libra adapts the resource definition for the pod and adjusts the horizontal scaling process accordingly.**

***Index Terms*—Cloud Computing, Resource Management, Scalability**

I. Introduction

Scalability is the ability of a resource or an application to be expanded to handle the increasing demands. One of the most emphasized aspects of cloud computing is the ability of scaling up or down. Scaling can be performed by either an operator or automatically. Autoscaling algorithms are getting more and more emphasized in the field of cloud computing.

Due to the dynamic nature of the load in the cloud, both the optimal resource allocation for a given pod and the autoscaling process are challenging tasks. Currently the most used and trending cloud native container orchestration framework is Kubernetes, which proposes two distinct auto scaling mechanisms, providing both horizontal and vertical scaling, the Horizontal and Vertical Pod Autoscaling (HPA and VPA), respectively. By default, both HPA and VPA are based on the measured CPU usage of the pods, but the definition of the triggering conditions are rather static. In this paper we propose a novel autoscaling mechanism, that jointly manages the vertical and horizontal scaling processes of Kubernetes pods, and adapts itself to the resource and load conditions of the virtualized environment. In the next section we present the scaling architecture of Kubernetes and several autoscaler proposals. Then we describe our proposal, and motivate our architectural decisions. We have implemented our scaling method in a prototype, and in section IV we present a measurement based evaluation of the performance of our solution.

II. Kubernetes and autoscaling

In Kubernetes applications are running inside containers. A Kubernetes pod is a group of one or more containers. Containers in a pod share the same resources and are able to communicate via localhost or by using methods of inter-process communication. Communication between pods is implemented by using the virtual network of the Kubernetes architecture [6]. Kubernetes uses replicasets to ensure that a specified number of pods are running in the system [7]. Kubernetes deployments are controllers that are providing declarative updates for pods and replicasets [8]. Deployments represent a set of multiple, identical pods with no unique identities [9].

A. Kubernetes resource management

In Kubernetes, the amount of compute resources can be assigned to the containers when a pod is specified. According to the terminology of Kubernetes we can separate resource requests and limits. Request are referring to the minimum value of the given compute resource that has to be guaranteed to the pod, while the limit is the maximum utilization that can be used by the pod. Initially the supported compute resources were CPU units and memory [3]. Additional custom resource metrics, like network or disk usage can be defined by using third party extensions.

B. Kubernetes Horizontal Pod Autoscaling

Kubernetes Horizontal Pod Autoscaling scales the number of pods of a deployment based on the utilization of compute resources (see Fig. 1). HPA receives user defined parameters related to resource usage thresholds. To be able to utilize HPA, resource requests and limits should be assigned to the containers running in the pods. The HPA scaling calculates the desired number of pods according to the algorithm in equation 1 [4].

$$desiredReplicas{=}\left\lceil currentReplicas{*}\frac{currentMetricValue}{desiredMetricValue}\right\rceil \tag{1}$$

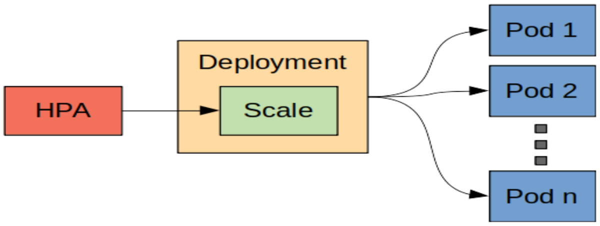

Fig. 1. Architecture of Kubernetes HPA

978-1-7281-4973-8/20/$31.00 © 2020 IEEE

C. Kubernetes Vertical Pod Autoscaling

Kubernetes Vertical Pod Autoscaler automatically sets the appropriate amount of resources for the pod without requiring any user cooperation [10]. The VPA works by collecting historical data from the pods and sets the resource requests accordingly [11], [12]. VPA recommends resource requests to the pods, calculated from the previously gathered historical data-series. VPA also recommends an upper and lower bound with the recommended value which can indicate the confidence of the recommendation. However, VPA only sets the resource requests values, thus the usable resource are not limited for the applications which can lead to cases when the pods running the applications would be killed by Kubernetes.

D. Knative project for Kubernetes

Knative is a third party extension package for Kubernetes that is essential to build source-centric, container-based applications [13]. However, in this paper, we only put our focus on the autoscaling module of Knative. The default autoscaling algorithm of Knative is based on the average inbound requests per pod (concurrency). To control scaling an average target concurrency can be set. In case of exceeding this concurrency level, the autoscaler scales the number of pods accordingly. However, the algorithm also maintains a 6 seconds panic window. If the level of concurrency exceeds the twice of the average concurrency level during the panic window, the algorithm enters panic mode and operates on a more sensitive 6 seconds long window. Once the panic conditions are no longer met, after a 60 seconds interval the algorithm returns to operate with the initial 60 seconds long window [14].

E. Alternative autoscaling proposals

The above overview of the Kuberenetes-based autoscaling solutions reveal their great deficiency: they require a static condition based on which they take their scaling decisions, and cannot adapt to the changes of their environment. There are several proposals in the literature to alleviate this problem, as scaling is an essential feature of cloud systems. Due to lack of space we cannot give a detailed overview of the topic, instead we refer the reader to some thorough reviews on this field. Papers [17] and [19] present scaling mechanisms, detail their theoretical backgrounds and offer a categorization of them. Most of these are generic theoretical proposals, and use either some machine learning or time series based methods to predict the load. In what follows, we focus on Kuberentes related proposals. Papers [18], [20] use an adaptive threshold based approach, but limit their work on vertical scaling, only. Zhao et al [5] implements an autoscaler for Kubernetes based on response delays by applying empirical mode decomposition and ARIMA model that predicts the load directed to the pods and sets the number of pods accordingly. Recently, several Function as a Service (FaaS) projects rely on Kubernetes to implement their services. Most of them use Kubernetes HPA for autoscaling, but some of them define custom scaling algorithms. In this paper we will use OpenFaaS as a benchmark since OpenFaaS is a well known and widely used, mature open-source serverless project, with a proprietary autoscaling module. By default, OpenFaaS scales on a requests per second basis [15]. Scaling is performed in case of the average number of served requests exceeds a given limit, which is hardcoded to 5 for a 10 seconds interval [16].

III. Libra

In order to provide an automated scaling mechanism in Kubernetes, we propose Libra, an autoscaler for applications running on top of Kubernetes. The main idea behind Libra is that the traditional autoscaling algorithms are taking the scaling levels from the user and trigger a scaling event in case of exceeding these limits. However, assigning the appropriate scaling levels to the services is not trivial to human operators. Usually, these scaling levels are set after long-time monitoring of services which can show an approximation of scaling level values. The problem with this approach is the appearance of new services. The behavior of these new services is unknown by the operators, thus a reasonable scaling level cannot be assigned. Libra mixes the advantages of horizontal and vertical autoscaling.

A. How it works

1) Phase 1 - Finding the appropriate CPU limit: In contrast to the traditional approach where the resource limit can be under or over-estimated, the scaling algorithm of Libra first calculates the least required, but sufficient enough CPU limit for a pod with which it can serve the actual load. Thus, in this phase Libra works as a vertical autoscaler. For this, Libra increases the CPU limit of the pod and collects usage metrics for thirty seconds. We have observed that the average number of served requests and the latency of serving times are converging to a value as the CPU limit is getting increased. We show in Fig. 2 the change of serving time latency (left axis) and of the number of served requests (right axis) as a function of CPU limit assigned to the pod. Starting from low resource values at the beginning, the increase in CPU resources improves the performance of the pod. Nevertheless, after reaching a threshold, it can be seen that the performance gain is not evolving further. Libra calculates this threshold by taking the last 5 measurement points of the 30 second long monitoring window and applies linear regression on both the number of served requests and on the serving times. If the steepness is below |5%|, Libra returns the minimum CPU limit with which the current load can be served. Based on our measurements, the shape of these curves is similar for very different type of applications. Fig. 2 presents the values obtained with a simple web service, whereas Fig. 3 shows the behavior of a compute intensive task (random sized matrix multiplication).

In this phase Libra maintains two kinds of PODs, a canary pod and a production pod. The canary pod is responsible for calculating the appropriate CPU usage for the current load, while the production pod is used to continuously serve the requests. Libra distributes the inbound requests between the canary and production pods on a 25-75% basis as it is shown in Fig. 4.

We selected the value of 25% of traffic directed to the canary pods as we believe that results coming from lower values of the traffic (e.g. 5-10%) might be affected by the noisy behavior of the system.

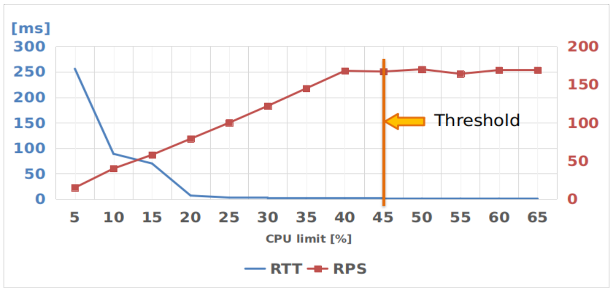

Fig. 2. Phase1: Finding the appropriate CPU resource limit for a simple web service

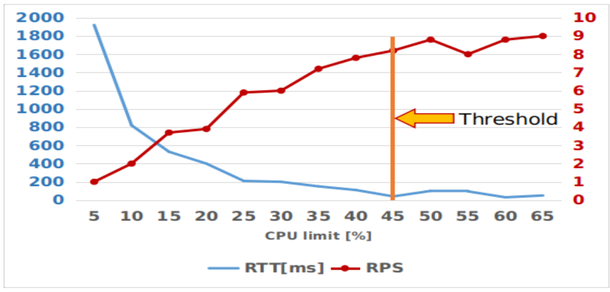

Fig. 3. Phase1: Finding the appropriate CPU resource limit for a service returns the result of random sized matrix multiplication

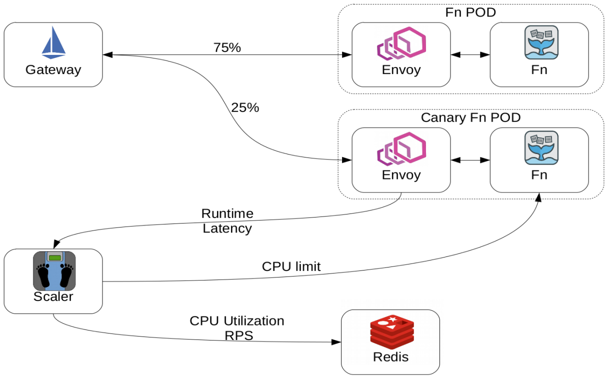

Fig. 4. Phase1: Sampling and Vertically scaling the Canary function pod

2) Phase 2 - Horizontal scaling: After calculating the sufficient CPU limit, Libra configures the production pod with the calculated CPU limit and routes all incoming requests to the production pod. Libra saves the calculated CPU value, as well as the number of served requests per seconds and the average duration of serving a single request. In this phase, Libra acts as a horizontal autoscaler and increases the number of pods according to the load until the number of pods reaches a predefined limit. After exceeding the limits of horizontal scaling Libra starts to act as a vertical autoscaler and finds a new CPU limit with which the pods can serve the increased load.

In the horizontal scaling phase, Libra increases the number of pods if the the number of served requests approaches the 90% of number of served requests related to the actual CPU limit value, to avoid pushing the system to an over-driven state, and scales down the number of pods if the number of served requests falls under the 40% of the number of served requests related to the CPU limit used by the pod, as in this case, not even the half of the resources are utilized. Libra also scales up the number of pods if the average serving time of a single request is at least 2 times higher than the calculated one.

B. Monitoring CPU parameters

Libra saves the CPU limits as well as the corresponding number of served requests per seconds performed by all of the pods that are responsible for running the same service. To assess these numbers we have installed a front proxy through which all requests shall pass. Therefore, it is available to monitor all the requests served by a given service. Libra updates the number of served requests per seconds for the corresponding CPU limit values if the current value is greater than the actually stored one. Monitoring the number of served requests also lets us to change the limit of CPU utilization of the pods according to the number of served requests.

C. Libra implementation details

The autoscaling architecture of Libra consists of the following parts (see Fig. 4). The main component is the autoscaler which fulfills vertical and horizontal autoscaling tasks and monitors the serving pods and the number of totally served requests. The worker pods are currently equipped with Envoy proxy instances running as a sidecar container next to the container that serves the requests. We have applied the Envoy proxy for being able to collect runtime metrics from the serving pods. We have chosen Envoy for this task since it is a lightweight proxy written in C++ with negligible overhead. Libra collects the calculated scaling levels and stores in a Redis database. We only needed to store the scaling levels in Redis to make Libra fault tolerant and to be able to restore its structures during a failover. For distributing the inbound traffic during the phase of vertical scaling we use Istio and its functions implementing the canary deployment pattern. To be able to monitor the total number of served requests by an application we have set up a front proxy which is implemented by an Envoy proxy instance.

IV. Evaluation

For the measurements we used a simple HTTP server that returns a "Hello" string. We implemented our web service in a Kubernetes pod, that we have used in our earlier work [1], in which we have examined performance tuning scenarios in open source FaaS systems.

We have compared how Libra performs compared to the builtin Kubernetes HPA, and we have also compared Libra's performance to the autoscaler implemented in the OpenFaaS project.

Libra implements a mixed scaling algorithm of both vertical and horizontal scaling techniques. However both Kubernetes HPA and the autoscaling mechanism of OpenFaaS are implementing horizontal scaling. Therefore in this section we compare the gains of using of Libra to the selected autoscaling algorithms.

Since in this case Libra works as a horizontal autoscaler, and in order to have fair comparison, we have configured the same CPU resource request and limit values in the other two cases (HPA and OpenFaaS), as well. Therefore we set a limitation for the test service to use only 15% of the total available CPU resources available on the given host (see IV-C for motivation).

For the measurements we used the Hey HTTP load-generator [2]. For our 10 minutes long measurements we generated a load with the concurrency level of 50. Our test environment consists of 4 XL-170 nodes supported by CloudLab. Each node is equipped with 10-core Intel E5-2640v4 CPUs, 64 GB of RAM and connected by 25 GB Mellanox ConnectX-4 network adapters. We ran Kubernetes v. 1.15.4 on top of Ubuntu 18.04 server operating system. Our Kubernetes setup consists of a single master and three worker nodes.

A. Kubernetes HPA

In case of Kubernetes HPA we set the CPU request resource parameters of the container implementing the service to 15%, which was identical to the value calculated by Libra. We set the threshold for scaling out to 90%. Therefore, we could simulate identical behavior to the horizontal scaling phase of Libra. According to the results, by using Kubernetes HPA the service scaled up to four pods, and was able to serve nearly 150 requests/sec after reaching the final number of pods, which can be seen in Fig. 5.

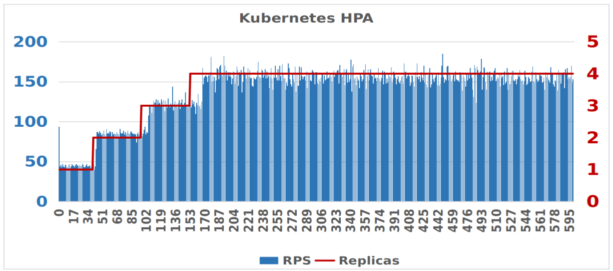

Fig. 5. Scaling performance of Kubernetes HPA

B. OpenFaaS builtin autoscaler

OpenFaaS implements an autoscaling algorithm that triggers if the rate of successful queries (HTTP 200) is greater than 5/sec for 10 seconds. However, these hard-coded values might lead to the over-driving of the architecture. We used the default maximum number of service pods which is set to 20. According to the results, OpenFaaS quickly scales out to the maximum number of pods, however after reaching the number of 10 serving pods the amount of served requests does not increase, see Fig. 6.

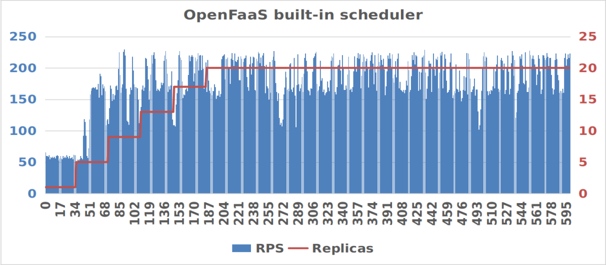

Fig. 6. Scaling performance of the OpenFaaS autoscaling mechanism.

C. Libra

The CPU limitation value of the service had been calculated as we had shown in III-A1, as a result, Libra set the CPU limit of the serving pod to 15%. Then the autoscaling phase resulted in scaling up to 8 pods, being able to serve 250 requests/sec as it is shown in Fig. 7.

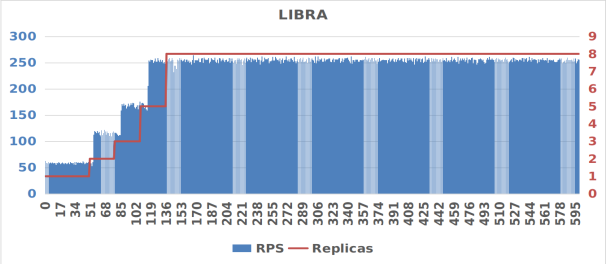

Fig. 7. Scaling performance of the Libra autoscaling mechanism.

V. Conclusion

Single threaded applications can increase their resource pool beyond the limits of a full single CPU core only by horizontally scaling out. However, multi-threaded applications can make advantage of vertical scaling, since they can efficiently use the resources of multiple CPU cores, as well. In order to exploit all the advantages of the scaling strategies, a combination of these two scaling mechanisms is desired.

We have proposed and implemented an adaptive scaling algorithm for Kubernetes pods called Libra, which does not need input parameters assigned by the user. We have shown that by coupling Libra with our sample web service application we outperformed both HPA, the default Kubernetes autoscaling mechanism, and the autoscaling facility of OpenFaaS.

Libra implements a mixed autoscaling algorithm of both vertical and horizontal scaling solutions, therefore it can improve the performance of both single and multi-threaded applications running on top of Kubernetes.

Acknowledgment

The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications.

The authors would like to thank the support and valuable comments of Dániel Géhberger (Ericsson Research Hungary).

References

[1] D. Balla, M. Maliosz, Cs. Simon, D. Gehberger, Tuning Runtimes in Open Source FaaS, In International Conference on Internet of Vehicles (pp. 250-266). Springer, 2020

[2] github.com/rakyll/hey

[3] The Kubernetes resource model, https://github.com/kubernetes/community/blob/master/contributors/design-proposals/scheduling/resources.md

[4] Kubernetes, Horizontal Pod Autoscaler https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[5] Zhao et al, Research on Resource Prediction Model Based on Kubernetes Container Auto-scaling, 2019 Technology

[6] Kubernetes Documentation, Pods, https://kubernetes.io/docs/concepts/workloads/pods/pod/

[7] Kubernetes Documentation, Replicasets, https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/

[8] Kubernetes Documentation, Deployments, https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

[9] Google Cloud, Google Kubernetes Engine, Deployment

[10] Vertical Pod Autoscaling, Google cloud, Kubernetes engine documentation, 2019

[11] Vertical pod autoscaler, Banzai Cloud blog, 2018, https://banzaicloud.com/blog/k8s-vertical-pod-autoscaler/

[12] Vertical Pod Autoscaler, kubernetes documentation page, 2019, https://github.com/kubernetes/community/blob/master/contributors/design-proposals/autoscaling/vertical-pod-autoscaler.md#resource-policy

[13] Knative documentation, 2019, https://github.com/knative/docs

[14] Knative serving autoscaling documentation, 2019, https://github.com/knative/serving/blob/master/docs/scaling/DEVELOPMENT.md

[15] OpenFaaS documentation, Auto-scaling, 2019, https://docs.openfaas.com/architecture/autoscaling/

[16] OpenFaaS [computer software], 2019, https://github.com/openfaas/faas/blob/3bcc10a07e264684d81355ea08a7403673b76bb1/prometheus/alert.rules.yml

[17] Al-Dhuraibi, Yahya, et al. "Elasticity in cloud computing: state of the art and research challenges." IEEE Transactions on Services Computing 11.2 (2017): 430-447.

[18] Lorido-Botran, Tania, Jose Miguel-Alonso, and Jose A. Lozano. "A review of auto-scaling techniques for elastic applications in cloud environments." Journal of grid computing 12.4 (2014): 559-592.

[19] Al-Dhuraibi, Yahya, et al. "Autonomic vertical elasticity of docker containers with elasticdocker." 2017 IEEE 10th international conference on cloud computing (CLOUD). IEEE, 2017.

[20] Rattihalli, Gourav, et al. "Exploring Potential for Non-Disruptive Vertical Auto Scaling and Resource Estimation in Kubernetes." 2019 IEEE 12th International Conference on Cloud Computing (CLOUD). IEEE, 2019.