Autoscaling in Kubernetes

Liam Nguyen
Department of Computer Science
CSULB
liam.nguyen@student.csulb.edu

Neha Bhoi
Department of Computer Science
CSULB
Neha.Bhoi@student.csulb.edu

***Abstract*—Container is a standalone, self-contained units that package software and its dependencies together. As the cloud industry adopts container technologies for internal usage and as commercial offering, a large-scale system of containers opens many challenges to manage such system. Kubernetes, introduced in 2015, is a container orchestration platform from Google that promises to run distributed system resiliently and effectively. One of its core responsibilities is to utilize and balance resources among containers and cluster of containers by auto-scaling against certain metrics. In this paper, we will discuss how Kubernetes gathers metrics and perform its auto-scaling actions. In addition, we also explore proposed improvements in scaling techniques to handle the sharing of resources among containers more fair, robust and efficiently.**

I. Introduction

Prior to the introduction of containerization, virtualization technology dominates the application world. Each virtual encloses all dependencies that an application needs and establish isolation and security controls among applications. However, Virtual Machine (VM) take up a lot of resources because each VM runs a full copy of an OS. This adds up a lot of expensive resources such as CPU and RAM.

As an alternative, containers, popularized by Docker, is light-weight (only few megabytes in size) and fast to respond to changes (few seconds to start). Containers store an application or application components with all the library dependencies, the binaries. They allow developers a high level of abstraction for the process life-cycle management, with the ability to start, stop or upgrade a new version of the application or micro-service seamlessly. However, the containerized applications are scaled with micro services architecture, the management and coordination of the container become challenging and complex.

Kubernetes, a container orchestration platform, provides functionalities such as load balancing, deployment and scaling of wide range of workloads. Most importantly, it can automatically restart the failed containers and reschedule them even when the hosts die.

II. Fundamental Concepts

[8]Kubernetes provides a framework to run a distributed systems. It provides load-balancing, storage orchestration, automated rollouts and rollbacks, self-healing. Its architecture includes a hardware side and a software side.

A. Hardware

A node is the smallest unit of computing hardware in Kubernetes. It is a representation of a single machine. This machine can be a either a physical machine or a virtual machine hosted on a cloud provider. When nodes are pooled together their resources, they form a more powerful machine called cluster, the gray rectangle on the right of figure 1.

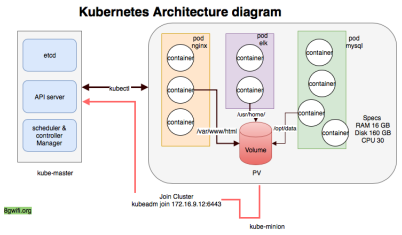

Fig. 1. Kubernetes high-level architecture diagram

A cluster is dynamic throughout the life of the program since it will expand or contract with different number of nodes determined by Kubernetes. As a result, programs shouldn't rely on data in arbitrary place in the file system.

Instead, persistent volumes should be used and required to act as a file system to store persistent data. This volume can be a cloud base or local drives, and not associated with any particular node.

B. Software

Container is the standard unit of software that packages up code and its dependencies. A container is built from image or layer of images by a Docker engine. However, Kubernetes doesn't run containers directly.

Instead, a pod is the smallest deploy-able unit that is managed by a node. A pod represents a unit of deployment: a single instance of an application in Kubernetes, which might consist of either a single container or a small number of tightly coupled containers [3]. Pods are typically not directly launched onto a cluster.

Instead, pods are usually managed by deployment which declare how many replicas of a pod should be running at a time.

III. Architecture

Figure 2 shows master-slave architecture of Kubernetes.

The slave nodes (or worker nodes) are managed by masters. On each node, Kubelet is an agent that acts as a bridge between the master and its node. Within Kubelet, cAdvisor is an analysis agent that discovers all containers in its node and collect reports on CPU, memory, file system, and network usage statistics.

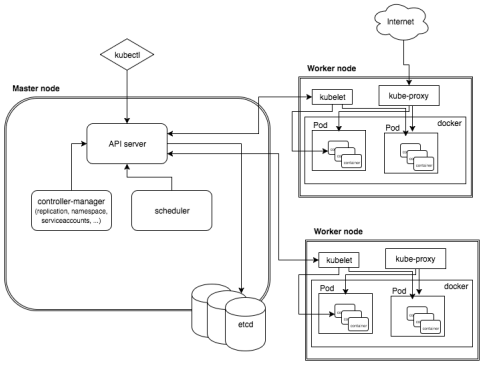

Fig. 2. Architecture of a Kubernetes

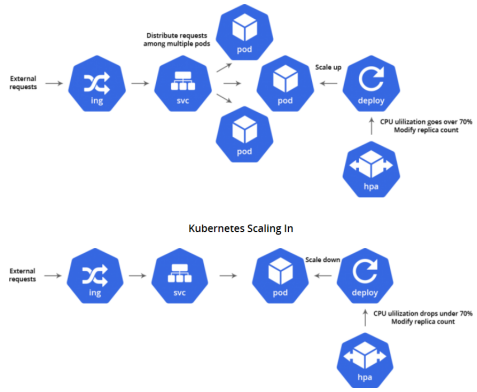

Fig. 3. Kubernetes scaling out and scaling in

This report is then sent back to the masters. Figure 2 shows that there are 4 components in the master:

1) etcd: store Kubernetes cluster data. For security reason, it is only accessible from the API server.
2) kube-apiserver: a central management entity that receives all REST request for modifications (pod, services, replication sets/controllers), served as front-end to the cluster.
3) kube-controller-manager: runs a number of controller. processes in the background, to regulate the shared state of cluster and perform routine tasks.
4) kube-scheduler: help schedule the pods on various node based on resource utilization.

In this paper, we will focus on Kube-controller-manager. It received the report from Kubelets and compare it against its metrics to make changes to attempting to move the current state towards a desire state.

With the arrangement of pods and clusters, there are also two layers of scaling in Kubernetes. In relation to pods, Kubernetes has Horizontal Pod Autoscaler(HPA) and Vertical Pod Autoscaler(VPA). In addition, cluster can be scaled as well. Cluster autoscaler (CA) scales up or down the number of nodes inside a cluster.

IV. Horizontal Pod Autoscaler

Horizontal pod autoscaler (HPA) changes the shape of your Kubernetes workload by automatically increasing or decreasing the number of Pods in response to the workload's CPU or memory consumption, or in response to custom metrics reported from within Kubernetes or external metrics from sources outside of your cluster. [6] (Refer fig: 3)

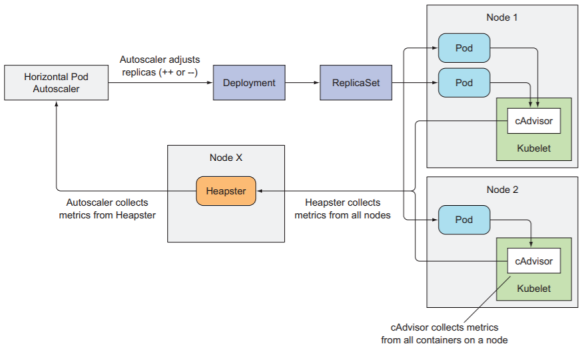

Fig. 4. How the autoscaler obtains metrics and rescales the target deployment

We can create a new autoscaler using kubectl create command.

```
>> kubectl autoscale rs foo
--min=2 --max=5 --cpu-percent=80
```

this command will create an autoscaler for replication set foo, with target CPU utilization set to 80 percent and the number of replicas between 2 and 5

A. How it works [7]

HPA is implemented as a control loop, with a period controlled by the controller manager's `--horizontal-pod-autoscaler-sync-period` flag (with a default value of 15 seconds). HPA can automatically scale the number of Pods in your workload based on one or more metrics of the following types [5]:

- Actual resource usage: when a given Pod's CPU or memory usage exceeds a threshold. This can be expressed as a raw value or as a percentage of the amount the Pod requests for that resource.
- Custom metrics: based on any metric reported by a Kubernetes object in a cluster, such as the rate of client requests per second or I/O writes per second.
- External metrics: based on a metric from an application or service external to your cluster.

The auto-scaling process can be split into three steps:

- Obtain metrics of all pods managed by scaled resource object.
- Calculate the number of pods required to bring the metrics to specified target value.
- Update the replicas field of the scaled resource

Fig: 4 shows how the autoscaler obtains metrics and rescales the target deployment. The autoscaler doesn't perform the gathering of the pod metrics itself. It gets the metrics from a different source. The horizontal pod autoscaler controller gets the metrics of all the pods by querying heapster through REST calls. The arrows leading from the pods to the cAdvisors, which continue on to heapster and finally to the Horizontal Pod Autoscaler, indicate the direction of the flow of metrics

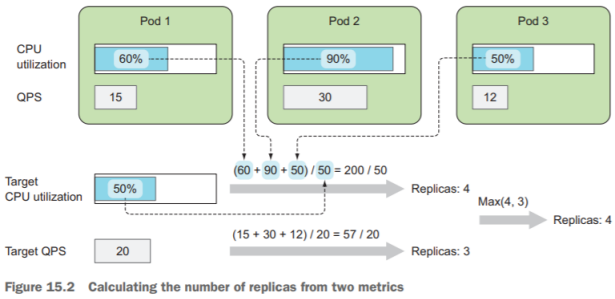

Figure 15.2 Calculating the number of replicas from two metrics

Fig. 5. Calculating the number of replicas from two metrics

Fig. 6. The Horizontal Pod Autoscaler modifies only on the Scale sub-resource

data. It's important to be aware that each component gets the metrics from the other components periodically.

Once the autoscaler has metrics for all the pods belonging to the resource the autoscaler is scaling, it can use those metrics to figure out the required number of replicas. The input to this calculation is a set of pod metrics and the output is a single integer.

When the Autoscaler is configured to consider only a single metric, calculating the required replica count is simple. All it takes is summing up the metrics values of all the pods, dividing that by the target value set on the Horizontal PodAutoscaler resource, and then rounding it up to the next-larger integer.

Fig: 5 shows how to calculate the number of replicas from two metrics.When auto-scaling is based on multiple pod metrics (for example, both CPU usage and Queries-Per-Second [QPS]), the calculation isn't that much more complicated. The autoscaler calculates the replica count for each metric individually and then takes the highest value.

Fig: 6 shows how The Horizontal Pod Autoscalar modifies only on the Scale sub-resource.The final step of an auto-scaling operation is updating the desired replica count field on the scaled resource object and then letting the ReplicaSet controller take care of spinning up additional pods or deleting excess ones.

V. Vertical Pod Autoscaler

A. Purpose

Vertical Pods Autoscaler aims to allocate more (or less) CPU or memory to existing pods. VPA works for both stateful or stateless pods but generally, it is built for stateful services. When VPA kicks in, it requires the pods to be restarted to change the allocated cpu and memory [2]. When VPA restarts pods, it respects pods distribution budget (PDB) on deployments to make sure there is always a minimum required number of pods.

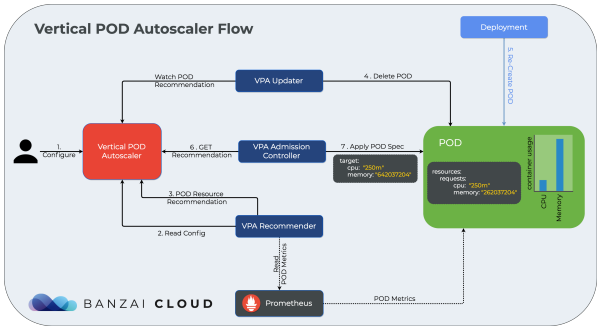

Fig. 7. VPA Flow

B. Internal Architecture

Figure 7 shows the internal of VPA. VPA consists of 3 main components: A VPA Updater, VPA Admission Controller and VPA Recommender.

- Recommender: monitor the current and past resource consumption to provides recommended values containers' CPU and memory requests
- Updater: check which of the managed pods have the correct resources set, and if not, kills them so that they can be recreated by their controllers with the updated requests.
- Admission Controller: sets the correct resource requests on new pods.

There are also 3 important configurations for VPA:

- Label Selector: determines which pods will be scaled according to the given VPA policy. The Recommender will aggregate signals for all pods matched by a given VPA, so it's important that the user set labels to group similar behaving pods under one VPA.
- Update Policy: determine how VPA applies changes. There are three modes that can be set:
  1) "Initial": VPA only assigns resources on Pod creation and does not change them during lifetime of the Pod.
  2) "Auto" (default): VPA assigns resources on Pod creation and additionally can update them during lifetime of the Pod, including evicting / rescheduling the Pod.
  3) "Off": VPA never changes Pod resources. The recommender still sets the recommended resources in the VPA object. This can be used for a "dry run".
- Resource Policy: controls how VPA computes the recommended resources. Here, the user can extend VPA capability by using different recommended algorithm to their specific use-case.

C. How it works

After VPA is configured, on startup, the recommender fetches historical resource utilization of all pods together with history of pod's out of memory (OOM) events. It aggregates this data and keep it in the memory [1]. It also watches all pod and all VPA objects in the cluster through real time metrics API from the metrics server. For every pod that is matched by some VPA selector, the recommender computes the recommended

resources and sets the recommendation on the VPA object. Since one VPA object has one recommendation, it is expected to use one VPA to control pods with similar resource usage patterns.

VPA Updater periodically fetches recommendations for the pods that are controlled by VPA by calling Recommender API. When the recommended resources significantly diverge from actually configured resources, the updater will update the pod. That can mean evicting pods to recreate them with new recommended resources. The Updater relies on other mechanisms (such as Replica Set) to recreate a deleted Pod. While terminating Pods is disruptive and generally undesired, it is sometimes justified in order to (1) avoid CPU starvation (2) reduce the risk of correlated OOMs across multiple Pods at random time or (3) save resources over long periods of time. Apart from its own policy on how often a Pod can be evicted, the Updater also respects the Pod disruption budget, by using Eviction API to evict Pods. The Updater only touches pods that point to a VPA with updatePolicy.mode set to ”Auto”.

Then VPA Admission Controller intercepts pod creation requests. The controller will fetch VPA configuration and if the pod is matched with mode not set to ”off”, the controller rewrites the request by applying the recommended resources to the pod spec. Otherwise, it will leave the pod unchanged.

D. Recommended Model

The request is calculated based on analysis of the current and previous runs of the container and other containers with similar properties. The model assumes that the memory and CPU consumption are independent random variables with distribution equal to the one observed in the last N days (recommended value is N=8 to capture weekly peaks). A more advanced model in future could attempt to detect trends, periodicity and other time-related patterns.

For CPU the objective is to keep the fraction of time when the container usage exceeds a high percentage (e.g. 95%) of request below a certain threshold (e.g. 1% of time). In this model the ”CPU usage” is defined as mean usage measured over a short interval. The shorter the measurement interval, the better the quality of recommendations for spiky, latency sensitive workloads. Minimum reasonable resolution is 1/min, recommended is 1/sec.

For memory the objective is to keep the probability of the container usage exceeding the request in a specific time window below a certain threshold (e.g. below 1% in 24h). The window must be long (*GTE* 24h) to ensure that evictions caused by OOM do not visibly affect (a) availability of serving applications (b) progress of batch computations (a more advanced model could allow user to specify SLO to control this).

VI. Cluster Autoscaler

Cluster Autoscaler scales the cluster nodes based on pending pods. It checks at a default interval of 10 seconds to determine whether there are any pending pods and increases the size of the cluster if more resources are needed and if the scaled-up cluster is within the user-defined constraints. When the node

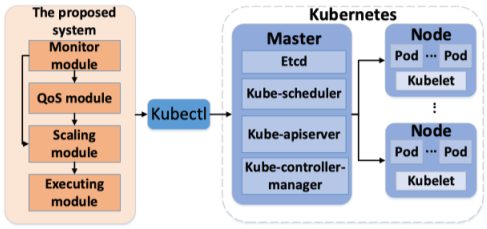

Fig. 1: Architecture of a Kubernetes cluster with the proposed system

Fig. 8. High-level CA and HPA

```
apiVersion: v1
kind: Pod
metadata:
 name: pod1
spec:
 containers:
 - name: demo1
 image: demo/demo1
 resources:
  requests:
    memory: "400Mi"
    cpu: "100m"
  limits:
    memory: "650Mi"
    cpu: "200m"
```

```
apiVersion: v1
kind: Pod
metadata:
name: pod2
spec:
 containers:
 - name: demo2
 image: demo/demo2
 resources:
  requests:
    memory: "100Mi"
    cpu: "300m"
    limits:
      memory: "300Mi"
      cpu: "400m"
```

Fig. 9. Pod configuration

is granted by the cloud provider, the node is joined to the cluster and serve pods. The whole process of scaling up is about 30 seconds. On the other hands, CA has a cool-down of 10 minutes before scales down when nodes are not needed.

VII. What is the issue with current scheduling

When a pod is deployed to Kubernetes, a yaml configuration is submitted to Kubernetes. In the configuration, the pod specifics its request and limit (max value) of CPU and memory that it might need. An example is shown in figure 9. If the resource limits are not specified during the pod creation level, a pod may consume all the resources of a node, leading to the starvation of other pods. Some applications are intensive in CPU or memory which means that they need more resources be able to run in the cluster.

Consider a node with the capacity of (900m,1800mi) in which m and mi represent CPU and memory units respectively. As shown in pod configuration figure 9, there are two pods each has single container. Generally, Kubernetes tries to allocate resource to each pod based on resource request [9]. Therefore, pod1 consume up to (200m,650mi) of the node resources and pod2 consume up to (400m, 300mi) of the node resources. The remain (300m,850mi) of the node resources is unused due to Kubernetes is not able to manage resources dynamically.

VIII. Solution for scheduling issue

If Kubernetes allows dynamical allocation of resources, several algorithms are proposed to fairly assigns resource limits based on resource requests and with respect to the given node. The resource limits should be determined based on the contribution of pods in the node.

A. DRF [4] : Dominant Resource Fairness

A user's dominant resource is the resource he/she has the biggest share of. Here is an example:

```
Total node resource <900m,1800mi>
User allocation <200m,650mi>
Dominant resource is memory because
(CPU Ratio = 2/9 < Memory Ratio = 1/3)
```

Consider two users A and B submit tasks with demand vector (d_cA,d_mA) and (d_cB, d_rB) respectively, where c and m denote CPU and memory. The the capacity of the resources is indicated by C_c and C_m. Calculation of dominant resources for both users are calculated as follows:

- $dom_A = max(d_cA/C_c, d_mA/C_m)$
- $dom_B = max(d_cB/C_c, d_mB/C_m)$

The allocation of resources are determined as the following optimization problem: maximize (A,B)

subject to $c_A + c_B < C_c$
$m_A + m_B \leq m$
$d_mA/C_m = d_cB/C_c$

Example from figure 9, we have:

- (d_cA,d_mA) : (200m, 650mi)
- (d_cB,d_rB) : (400m, 300mi)
- (C_c, C_m) : (900m, 1800mi)
- dom_A = max(200/900, 650/1800) = 0.36. Hence, dominant resource for pod1 is memory.
- dom_B = max(400/900, 300/1800) = 0.44. Hence, dominant resource for pod2 id CPU.
- DRF assigns (300m, 1200mi) resource limits for pod 1 and (600m, 200mi) for pod 2.

B. MLF-DRS [4] : Multi-level Fair Dominant Resource Scheduling

MLF-DRS tries to share the available resources equally among the dominant shares. MLF-DRS determines non-dominant resources as well. MLF-DRS considers fair share of resources, indicated by $f_c = C_c/n$ and $f_m = C_m/n$ where n denotes the total number of users. Consider two users A and B submit tasks with demand vector (d_cA, d_mA) and (d_cB, d_rB) respectively, where c and m denotes CPU and memory. At the first stage, dominant resources get a fair share of resources if ($dA_c <= f_c$, $dA_m <= f_m$) and ($dB_c <= f_c$, $dB_m <= f_m$). Otherwise, dominant resources get initial requested resources. Similarly, non-dominant resources are allocated only what they have requested.

MLF-DRS calculates non-dominant resources of users as follows:

nondomA = min(d_cA/C_c,d_mA/C_m)
nondomB = min(d_cB/C_c,d_mB/C_m)

For the next level, considering that the allocated resources for users A and B are denoted by (X_cA,X_mA) and (X_cA,X_mA) respectively, the allocation can be calculated as follows:

CPU allocation for dominants:

- UserA = $((C_c - (X_cA + X_cB)) * f_c)/(X_cA + X_cB)$
- UserB = $((C_c - (X_cA + X_cB)) * f_c)/(X_cA + X_cB)$

CPU allocation for non-dominants:

- $UserA : ((C_c - (X_cA + X_cB)) * X_cA)/(X_cA + X_cB)$
- $UserB : ((C_c - (X_cA + X_cB)) * X_cB)/(X_cA + X_cB)$

Memory allocation for dominants :

- $UserA : ((C_m - (X_mA + X_mB)) * f_c)/(X_mA + X_mB)$
- $UserB : ((C_m - (X_mA + X_mB)) * f_c)/(X_mA + X_mB)$

Memory allocation for non-dominants :

- $UserA : ((C_m - (X_mA + X_mB)) * X_mA)/(X_mA + X_mB)$
- $UserB : ((C_m - (X_mA + X_mB)) * X_mB)/(X_mA + X_mB)$

Using the configuration from figure 9, where (C_c, C_m) : (900m, 1800mi) and there are two user A

C. FFMRA [4]: A Fully Fair Multi-Resource Allocation Algorithm

FFMRA is the generalization of DRF and proportionality. FFMRA calculate the allocation as follows. It determines dominant and non-dominant resources as follows:

- $dom_A = max(d_cA/C_c, d_mA/C_m)$
- $dom_B = max(d_cB/C_c, d_mB/C_m)$
- $nondom_A = min(d_cA/C_c, d_mA/C_m)$
- $nondom_B = min(d_cB/C_c, d_mB/C_m)$

It sums up together all dominant resources of all users of the entire server.

$S_{dom} = dom_A + dom_B$

It sums up together all non-dominant resources of all users of the entire server.

$S_{nondom} = nondom_A + nondom_B$

capacity of the resources of entire resource pool and both dominant and non-dominant resources respectively.

$S_C = C_c + C_m$

$S_t = S_{dom} + S_{nondom}$

Total capacity of the resource pool is divided proportionally among dominant and non-dominant resources indicated by P_{dom} and P_{nondom} respectively as follows:

$P_{dom} = (S_c * S_{dom})/S_t$

$P_{nondom} = (S_c * S_{nondom})/S_t$

The divided share for CPU and memory in the resource pool denoted by Sh_c and Sh_m respectively are determined as follows:

$Sh_c = (P_{dom} * C_c)/S_C$

$Sh_m = (P_{dom} * C_m)/S_C$

The allocated resource to each user is calculated (Sh_c, Sh_m) as the final stage of MLF-DRS

Fig. 10. The configuration of four pods to be applied in experiment

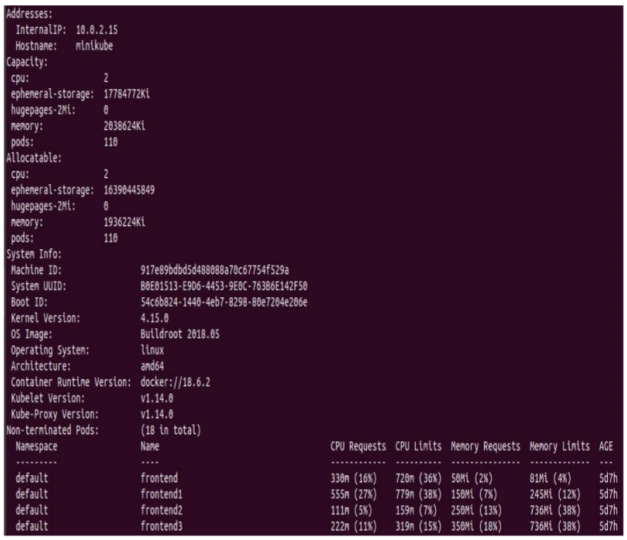

Fig. 11. An experiment consists of four pods along with resource limit assignments

IX. Proposed Solution

A. Mathematical implementation in Kubernetes [4]

we can show the problem as follows:

Let P is a set of created pods by user U. R indicates the number of resources that can be requested by user U.N represents the number of nodes.

```
|P| = p1,p2,...,pn.
|R| = 1,2,3...,r.
|N| = 1,2,3...,n.
```

Let LP_r and xp_r refer to the resource limits and requests of a pod where the requested resources should be less than or equal to the resource limit (xp_r LTE lp_r).

The scheduled pod in a specific node is represented by pi which is the scheduled pod p in a corresponding node i.

In Kubernetes the available resources for pods in a node are defined as allocatable resources.

```
Allocatable = nodecapacity -
(Kubereserved + systemreserved
+ evictionthreshold)
```

Allocatable resources r of a particular node i can be definedas AL_{ri}. Dominant and nondominant resources of each pod considering the capacity of resources with respect to each node calculated as follows:

$Dp_{ir} = max(xp_r/Al_{r_i})$

$Ndp_{ri} = min(xp_r/Al_i)$

Let's say $S(LP_r)$=Lp_r refers to the sum of resource limits of a specific pod p, then the new resource limits assignment could be determined as follows:

maximize (p1,p2,...,pn)

subject to $xp_r \leq C_{ir}$.

$s(lp_r) \leq C_{ir}$.

B. Practical Example [4]

A single cluster is using a single node with capacity of (2000mi, 2000m) using minikube. The example 11 represents pods and assigned resource limits in administrator's side. The results shown in Figure 11 based on pods configurations in Figure 10 indicates resource requests and limits as well as the running pods. The total capacity of the CPU and Memory are 2 and 2,038,624ki respectively of which 2 is the number of CPU core equals to 2000m and the memory equals to (2038624/1024 = 1990mi).

According to the mathematical implementations, frontend and frontend1 have dominant resource in CPU, while frontend2 and frontend 3 have dominant resource in memory. Resource limits are evenly divided among the applications so that first two applications with dominant resource in CPU get 36% and 38% respectively. Last two applications which have dominant resource in Memory, they get exactly 38% of resource limits.

X. Conclusion

In general, Kubernetes auto-scaling measures CPU and Memory resources to determine its actions. Although, its scaling capabilities are well-tested and used in various workflow and applications, there are still room for improvements. Kubernetes doesn't consider fairness at all. We have analyzed fair allocation algorithm, DRF, MLF-DRS and FFMRA. Based on these policies, we found an experiment which defines dynamic resource limit calculation in Kubernetes. This experiment assumes fairness in administrator's point of view and supposed that he/she is aware of the corresponding node capacity. However, since this experiment is in implementation level it does not consider real case example including many pods and containers.

References

[1] "Kubernetes," https://github.com/charlespwd/project-title, 2019.

[2] M. Ahmed, "Kubernetes autoscaling 101: Cluster autoscaler, horizontal pod autoscaler, and vertical pod autoscaler," July 2018. [Online]. Available: https://medium.com/magalix/kubernetes-autoscaling-101-cluster-autoscaler-horizontal-pod-autoscaler-and-vertical-pod-2a441d9ad231/

[3] V. P. Emiliano Casalicchio, "Auto-scaling of containers: the impact of relative and absolute metrics," 2017.
[4] K. K. Hamed Hamzeh, Sofia Meacham, "A new approach to calculate resource limits with fairness in kubernetes," 2019.
[5] M. H. C. L. Hanqing Zhao, Hyunwoo Lim, "Predictive container auto-scaling for cloud-native applications," 2019.
[6] M. T. F. K. Leila Abdollahi Vayghan, Mohamed Aymen Saied, "Deploying microservice based applications with kubernetes: Experiments and lessons learned," 2018.
[7] M. Luksa, *Kubernetes in Action*, 2nd ed. Manning, 2020.
[8] R. B. Maria A. Rodriguez, "Containers orchestration with cost-efficient autoscaling in cloud computing environments."
[9] L. L. S. Q. G. X. Qiang Wu, Jiadi Yu, "Dynamically adjusting scale of a kubernetes cluster under qos guarantee," 2019.