A New Approach to Calculate Resource Limits with Fairness in Kubernetes

Hamed Hamzeh[1], Sofia Meacham[1] andKashaf Khan[2]
[1]*Faculty of Science and Technology, Bournemouth University, UK.*
hamzehh@bournemouth.ac.uk, smeacham@bournemouth.ac.uk
[2]*British Telecom, Ipswich, UK, kashaf.khan@bt.com*

***Abstract*—Containerization has become a new approach that facilitates application deployment and delivers scalability, productivity, security, and portability. As a first promising platform, Docker was proposed in 2013 to automate the deployment of applications. There are many advantages of Docker for delivering cloud native services. However, its widespread use has revealed problems such as performance overhead. In order to deal with those problems, Kubernetes was introduced in 2015 as a container orchestration platform to simplify the management of containers. Kubernetes simplifies managing a large scale number of docker containers, however, the fairness is a missing point in the Kubernetes that has been applied in other platforms such as Apache Hadoop, YARN and Mesos. Assigning resource limits fairly among the pods in kubernetes becomes a challenging issue as some applications may require intensive resources such as CPU and memory that should be maximized to satisfy them. In order to do that, in this paper, we practice a novel way to assign resource limits fairly among the pods in the Kubernetes environment.**

***Keywords*-Cloud computing; dominant; non-dominant; fairness; Kubernetes ; resource; Scheduling.**

I. Introduction

Cloud computing provides many facilities and opportunities for organizations and especially for DevOps to move their operations into the virtual world. The aim of virtualization in the cloud is to attain a good level of elasticity in terms of resources so that Virtual Machines (VMs) and containers provide a specific level of virtualization. While virtual machines provide resources in Infrastructure as a Service(IaaS) layer, the virtualization in containers happens in operating system level in which multiple containers run on top of the OSKernal [1]. Containerization aims to provide isolation between management and developing applications without concerning of migrating them from one environment to another. Docker is one of the most popular containerization platforms which was proposed in 2013 to package the applications along with their dependencies. Within the Docker, codes are transformed easily to other environments[2].

However, as users's demands increase and the containerized applications are scaled, the management and coordination of the containers become challenging. To overcome those problems, Kubernetes[3] was proposed in 2015 as a container orchestration platform in order to orchestrate different workloads in terms of computing and networking operations. Additionally, Kubernetes provides different functionalities such as load balancing, deployment, and scaling of a wide range of workloads. Fairness algorithms are generally used in well-known platforms such as Apache Hadoop,YARN[4] and Mesos[5] such as Max-Min[6] and proportional fairness[7].

Hadoop is an open-source software system which is used to process the huge amount of data with the cluster of computers. Hadoop Fair Scheduler(HFS)[8]assigns resources in slots to all incoming jobs. HFS considers a queue of jobs that allows short jobs to be processed without starving long ones. Fair resource allocation in HFS applies priorities as weights of each user's job to calculate the exact amount of compute resources that could be allocated in each time. Apache Mesos[9] is a cluster manager developed by the University of Berkeley that provides efficient resource sharing on a huge scale between distributed applications and data centers. It lays between the application and operating system layers in order to provide an easy and efficient approach for the application deployment and management in large-scale clustered environments. Mesos uses DRF to allocate and schedule resources across multiple users, aiming the high resource utilization. Apache Hadoop YARN as a necessary component for enterprise Hadoop aims to facilitate resource management to provide consistent operations, security, and data governance tools across the Hadoop clusters[10]. FIFO, Fair, and Capacity are widely used schedulers in YARN. Similar to Hadoop Fair policy takes into account only utilization of memory for each job and intents to assign memory in equal shares, while the DRF tries to ensure all jobs to get an equal share of resources based on their dominant resource requirements.

Kubernetes has an approach however we believe it can be improved as it is not fair in all circumstances. Basically, each pod may have intensive resource requests over one of the resource types either in CPU or memory. In this case, maintaining fairness among the pods would be a core criterion in which the maximization of allocations for each pod becomes an optimal solution.

The concept of fairness was initially considered in a multi-resource cloud environment by introducing DRF algorithm[11] in 2012. Technically, DRF equalizes dominant shares of users to achieve a fair allocation along with efficient utilization of resources while the objective is to maximize the allocations of users' shares. According to our knowledge, there is no any specific work considering fairness in Kubernetes, however, DRF has been considered in kube-batch project as a batch scheduler in Kubernetes which aims to share resources among different tenants[12]. DRF as a plugin in kube-batch is just for registering some callbacks for actions such as the compare function to sort jobs and event handler.

We believe that setting up resource limits without considering fairness is not an appropriate solution in an environment in which pods are competing to get more resources. Basically, in the Kubernetes, 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. So, each pod should normally get a specific amount of resource. In other words, some applications are intensive in CPU or memory which means that they need more resources be able to run in the cluster.

Kubernetes has an approach, however, we believe that it can be improved as it is not fair in all circumstances. The fairness problem in Kubernetes has only been taken into account in terms of resource quota. The resource quota is applicable when many number of users or teams share the cluster with a certain number of nodes. In that case, the resource quota feature, prevents a team to use more than its fair share of resources. Although this method is a way to avoid an unfair situation in the Kubernetes, however, it is only for the limited number of cases not for a general situation. Furthermore, the resource quota and limit allocations should follow dynamic settings which means that each pod should receive its share based on what it has asked for and what is the resource requirements. Basically, each pod may have intensive resource requests over one of the resource types either in CPU or memory. In this case, maintaining fairness among the pods would be a core criterion in which the maximization of allocations for each pod becomes an optimal solution.

Taking into account the above-mentioned problems, In this paper, the first attempt is to model and integrate three different fair allocation algorithms, MLF-DRS[13] and FFMRA[14] as well as DRF in Kubernetes, trying to assign resource limits fairly among different pods running in a specific node.

The paper is organized as follows: In section II, a background study is considered. Section III, explains the motivation of the research. The proposed model is introduced in section IV as well as examples. Finally, section V presents the conclusion and summary of the paper.

II. Background study

A. Kubernetes architecture

Kubernetes (K8s) is a widely accepted open-source platform for the container orchestration, automating deployment, scaling, and management of containerized applications. It gathers together all containers to keep up applications into logical units to do an easy management and discovery operations. The high level architecture of Kubernetes[15] has been presented in Figure 1. Cluster is the highest-level concept in Kubernetes which is composed of a bunch of running machines to manage the containers. Unlike already mentioned platforms, Kubernetes does not apply any fair allocation and scheduling mechanisms.

CPU and Memory are two types of resources in Kubernetes known as computing resources for containers. Containers are placed in the simplest and the fundamental units of Kubernetes known as pods. Pods are scheduled in the selected nodes by the scheduler. Nodes are a kind of virtual machines Which are not created by the Kubernetes, however, they are delivered by cloud providers such as Google cloud, Amazon Web Services (AWS) and Microsoft Azure. Basically, the Kubernetes consists of different components listed as follow:

- Master: As the main control plane of the kubernetes, the master component provides different functionalities to worker nodes and also users. The main components of master are categorized as follows:

1) Etcd: is a key value storage and backup agent in kubernetes in which all the information and configurations regarding the cluster are stored and accessible through the API server by the worker nodes.
2) Kube-apiserver: The API server as the front end of the kubernetes control panel, is one of the critical services within kubernetes master component and as a bridge among different components. The API is served using a json file and the communication inside it is handled by kubeconfig package. As the principal management component of the entire cluster, a user is allowed to configure Kubernetes workloads and organizational units. The API server is eligible to manipulate the state of different objects like pods and services.
3) Kube-controller-manager: The share state of the cluster in kubernetes is monitored by a controller via the API server and it tries to change the current state to the optimal state. The existing controllers in kubernetes are endpoints, replication, namespace and service accounts controllers. To reduce the complexity, all those controllers are integrated and compiled in a single binary.
4) Kube-scheduler: This component is responsible for deploying pods and services in the suitable nodes. Different parameters are considered during the scheduling, resource requirements and limits, quality of services of pods, affinity and anti-affinity. The main function of a node in kubernetes is to check the requested resources of all pods to make sure that their requested amount doesnt exceed the total capacity of the corresponding

node.

- Node:As the worker machine managed by the master component in the kubernetes, a node is created by the cloud providers in order to run the containers. It consists of different objects as follows:

1) Kubelet: Is one of the main components of the kubernetes that places into each worker node to run the pods and checks them regularly in order to make sure that they are working properly. Kubelet also runs health checks for all running pods and then it interacts with API server and reports regularly the state of the node and running pods inside it.
2) Kube proxy: As an object running in each worker node, kube proxy checks regularly the changes in pods and services to keep the network up to date to make sure that the environment is accessible.
3) Container runtime: This object is placed at the lowest layer of the node in order to start and stop pods and services. Docker is the most well-known container run-time that we mentioned in previous sections.

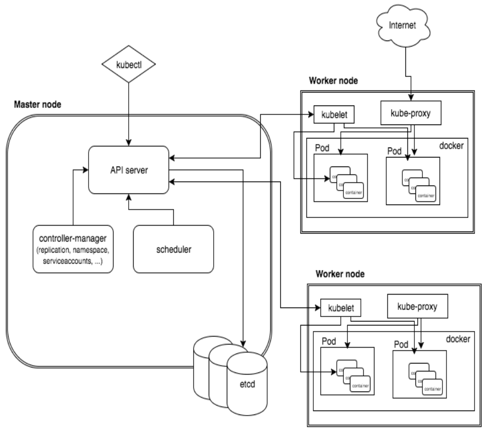


Figure 1: Kubernetes architecture

B. Scheduling in kubernetes:

When a user creates a pod, resource requests and limits known as PodSpecs are included in that pod. The scheduler considers different parameters such as predicates, priorities, and Quality of Service of pods. Predicates maintain functions and they get PodSpecs and node information like available resources and capacity of the node and return a boolean value to indicate whether a pod can be fitted in a specific node. The priority of pods depends on the QoS classes of those pods. The importance of a pod is determined by the priority. Typically, pods with the lowest priority is the best candidate to be evicted (preemption) in order to schedule the higher priority pods. Therefore, pods are categorized into three different levels: Guaranteed, burstable and best-effort. The guaranteed class has the highest priority in which pods have the same resource limits and requests. In a case where the resource limit is set up above requested resources, it refers to the burstable class so that a pod can consume the resources up to the specified limit. Finally, in the best effort class, both resource requests and limits are not specified which makes a pod as the lowest priority one.

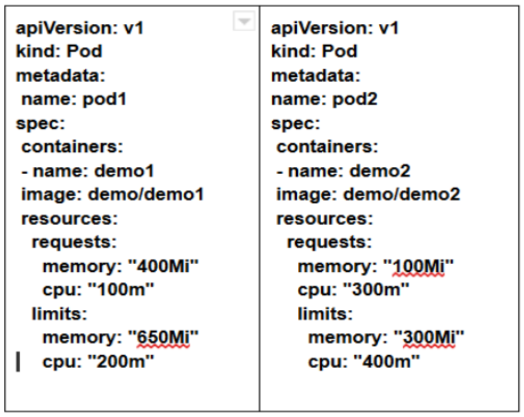
1) Afinity and anti-afinity: nodeSelector introduces the best and simple approach to limit pods to nodes with specific labels called affinity/anti-affinity property which develops constraints so that a user may represent. The affinity property involves two different types, node affinity and inter-pocd affinity/anti-affinity. Node affinity is similar to the nodeSelector object, while inter-pod affinity/anti-affinity limits opposed to pod labels instead of node labels, plus having the first and second features mentioned above. Node affinity is perceptually like nodeSelector which allows anyone limits which pod is eligible to be scheduled in a particular node, according to the labels on the node. Node affinity is determined as field nodeAffinity of field affinity in the PodSpec.

2) Resource quota and namespaces: If many teams are working in a shared cluster, it is necessary to assign namespaces for each of those teams. Namespaces guarantees isolation of each teams resources from the rest of the cluster and for each namespace, resource quotas should be setup that constraints resource consumption of pods in each namespace. Resource quotas can be setup in different practices. Typically, kubernetes, considers resource limits based on what can be seen in figure 2. In original kubernetes, the namespaces are placed directly inside the cluster with specific resource quotas while according to figure 3, in Rancher[16] as the recent development of the kubernetes, the project object and project quota has been specified within the cluster and then quotas have been assigned to the existing namespecase.

III. Fairness in Kubernetes

Resource allocation only based on resource limits and requests and the specified QoS levels of pods may not satisfy the fairness property in a multi-resource environment such as Kubernetes where users may submit pods consist of either intensive CPU or memory tasks. However, in kubernetes, there is no any strategy to deal with such tasks or pods.

As an example based on figure 2, there are two pods each has one single container. Also, there is a node with the capacity of $(900m, 1800mi)$ in which m and mi represent CPU and memory units respectively. The kubernetes tries to allocate resources to each pod according to resource requests. Therefore, based on the default policy and what has been set up in resource requests and limits, pod1 can consume up to $(200m, 650mi)$ of the node resources and pod2 ables to utilize maximum $(400m, 300mi)$. Consequently, at least $(300m, 850mi)$ remain unused that can be allocated to other pods. Unfortunately, Kubernetes is



```
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"
```

Figure 2: Pod configuration

not able to manage resources dynamically and users may require to assign resource limits when creating pods. It is a challenging problem to allow kubernetes to set resource limits after scheduling based on resource requests and node information. To be more clear, when users create pods, they do not know exactly what proportion of resource limits should be assumed. Also, the scheduler does not consider resource limits in scheduling time. Consequently, a problem may occur. The scheduler selects a node for a sort of pods with different resource limits. If the total amount of resource limits becomes greater than the total capacity of the node, and each node tries to consume resources up to the specified limit, then the Linux OS kernel will ignore resource limits and allocate only the requested resource or the resource limit will be assigned based on the resource limit ranges specified by the kubernetes. The logical way is that after scheduling, Kubernetes assigns resource limits based on resource requests and with respect to the given node. However, in cloud environments, some users may have intensive tasks in some resources. Here in kubernetes, each pod may have one intensive resource either in CPU or memory. So, after scheduling a pod in a specific node, the resource limits should be determined fairly based on the contribution of pods in the node. This gives an accurate resource limit assignment while the fairness is considered. Typically, when pods are created by users, resource limits are also specified.

A. *Planning fairness algorithms in Kubernetes*

There are different fair allocation algorithms in the cloud, and in this section, we introduce three algorithms, DRF, MLF-DRS and FFMRA that we mentioned in section I that we plan to integrate them in Kuberentes.

1) DRF: As an example based on Figure 2, If 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 subsequently and the capacity of the resources is indicated by C_c and C_m, then dominant resources for both users are calculated as follows:

$$dom_A = max(d_{cA}/C_c, d_{mA}/C_m) \tag{1}$$

$$dom_B = max(d_{cB}/C_c, d_{mB}/C_m) \tag{2}$$

Consequently, based on (1) and (2) as well as the example in figure 1, the dominant resource for pod 1 is memory and pod 2 is CPU. Therefore, the allocation of resources are determined as the following optimization problem:

$$\begin{aligned} \text{maximize} \quad & (A, B) \\ \text{subject to} \quad & cA + cB \leq \ C_c. \\ & mA + mB \leq_m . \\ & d_{mA}/C_m = d_{cB}/C_c. \end{aligned} \tag{3}$$

By solving(3), DRF assigns $(300m, 1200mi)$ resource limits for pod 1 and $(600m, 200mi)$ for pod 2. Typically, DRF equalizes dominant resource to calculate the allocations.

2) MLF-DRS:: While DRF takes into account only dominant resources, MLF-DRS determines non-dominant resources as well. Compared to DRF, MLF-DRS tries to allocate all the resources of the resource pool to reach the highest resource utilization and a desirable fair allocation. It also guarantees that all users with dominant resource get desirable amount of allocation while in DRF and in scenarios with more than two users, some of them are not able to maximize their allocations. Consequently, starvation may have happened for those users with dominant shares. So, according to the same configuration in 4.1, 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. Considering that (dA_c, dA_m) and (dB_c, dB_m) are demand vectors for users A and B respectively, 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. Despite of DRF, MLF-DRS calculates non-dominant resources of users based on (4) and (5) as follows:

$$nondom_A = min(d_{cA}/C_c, d_{mA}/C_m) \tag{4}$$

$$nondom_B = min(d_{cB}/C_c, d_{mB}/C_m) \tag{5}$$

Similarly, non-dominant resources are allocated only what they have requested. 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 :
$UserA : ((C_m - (X_{mA} + X_{mB})) * X_{mA})/(X_{mA} + X_{mB})$
$UserB : ((C_m - (X_{mA} + X_{mB})) * X_{mB})/(X_{mA} + X_{mB})$

3) FFMRA:: FFMRA is the generalization of DRF and proportionality. Same as MLF-DRS, it considers both dominant and non-dominant resources, however, it tries to equalize both dominant and non-dominant resources in entire resource pool to provide a totally fair allocation of resources. FFMRA maintains the balance of the system by evenly distributing system resources among the bounce of dominant resources of users. By keeping the similar configurations in the previous policies, FFMRA calculate the allocation as follows. First of all, it determines dominant and non-dominant resources and then it sums up together all dominant resources of all users of the entire server. This process also applies for all non-dominant resources. Then, the proportion of resources in the server is calculated for all dominant and non-dominant resources as follows:

$$S_{dom} = dom_A + dom_B \tag{6}$$

$$S_{nondom} = nondom_A + nondom_B \tag{7}$$

$$S_C = C_c + C_m \tag{8}$$

$$S_t = S_{dom} + S_{nondom} \tag{9}$$

Based on (6),(7),(8),and(9) S_{dom}, S_{nondom},S_c and S_t denote sum of dominant, non-dominant, capacity of the resources of entire resource pool and both dominant and non-dominant resources respectively. According to (10)and(11), the 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:

$$P_{dom} = (S_c * S_{dom})/S_t \tag{10}$$

$$P_{nondom} = (S_c * S_{nondom})/S_t \tag{11}$$

Accordingly, it is necessary to divide P_{dom} and P_{nondom} to corresponding resources in the resource pool. This process guarantees the balanced distribution of resources among the users. The divided share for CPU and memory in the resource pool denoted by Sh_c and Sh_m respectively are determined as follows:

$$Sh_c = (P_{dom} * C_c)/S_C \tag{12}$$

$$Sh_m = (P_{dom} * C_m)/S_C \tag{13}$$

Finally, according to (12) and (13), the allocated resource to each user is calculated as the final stage of MLF-DRS.

IV. Proposed solution

A. Mathematical implementation in Kubernetes

Basically, the purpose of this paper is to maximize the resource limits assignment among the pods. Generally, we can show the problem as follows:

Given that P is a set of created pods by user U, say $|P| = p_1, p_2, , p_n$ and R indicates the number of resources that can be requested by user U, $|R| = 1, 2, ..., r$ and N represents the number of nodes where $|N| = 1, 2, ..., n$. We assume that LP_r and xp_r refer to the resource limits and requests of a pod created by user U where the requested resources should be less than or equal to the resource limit ($xp_r \leq lp_r$). The scheduled pod in a specific node is represented by p_i which is the scheduled pod p in a corresponding node i. Accordingly, dominant and non-dominant resources of each pod considering the capacity of resources with respect to each node indicated by C_{ir} based on the following formulations:

$$Dp_{ir} = max(xp_r/C_{ri}) \tag{14}$$

$$Ndp_{ri} = min(xp_r/cr_i) \tag{15}$$

Basically, in Kubernetes the available resources for pods in a node are defined as allocatable resources. So, without loss of generality, it can be calculated as follows:

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

Therefore, allocatable resources r of a particular no0de i can be defined as AL_{ri}. Hence, (14) and (15) can be changed to (16) and (17):

$$Dp_{ir} = max(xp_r/Al_{ri}) \tag{16}$$

$$Ndp_{ri} = min(xp_r/Al_i) \tag{17}$$

However, if every pod has the resources associated with a weight lets say, in the case the definition of dominant and non-dominant resources can be changed to the following formulations:

$$d_{rp} = max(xp_r/f_{rp}) \tag{18}$$

$$nd_{rp} = min(xp_r/f_{rp}) \tag{19}$$

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

$$\begin{aligned} \text{maximize} \quad & (p_1, p_2, ..., p_n) \\ \text{subject to} \quad & xp_r \leq \ C_{ir}. \\ & s(lp_r) \leq \ C_{ir}. \end{aligned} \tag{20}$$

This is note that in the examples in figures 3 and 4, we assume that $C_{ir} = AL_{ri}$.

B. System Design

In order to integrate our proposed solution, we described the system both structure and behaviour at the system level using SysML modelling language[17]. To have an overall approach of the system, SysML Block Definition Diagrams were described for the structure and SysML Activity Diagrams for the behaviour. A professional tool, the Cameo Systems Modeller was used to describe and validate the diagrams[18]

1) Block diagram system structure: In order to design a system to integrate mentioned policies in section IV, it is essential to have in-depth information from different component in Kubernetes. A quick review of different objects is depicted in figure 3 and how the proposed model works. First of all, a **user** with a unique ID creates a **pod** using **kubectl** along with requested CPU and memory. The created pod along with other information are passed to the **API server** which serves the the kubernetes **API**, version **v1**. The API consists of different component in which the most important objects are **nodes**, **pods**, **services** and **types**. When a node is created by a particular cloud provider like Amazon Web Services(AWS), Google Cloud Platform(GCP) and Microsoft Azure, that are references within **cloud provider** object, the information is sent to the **API server**. Then other objects get the **node information** through the API server. In the model both **scheduler** and **policy broker** get required information from **API server**. In our proposed model, since, we are looking for changing and updating resource limits after scheduling, it is necessary to extract some important information. So, we consider two objects in the model which are **Node information** and **resource limits**. In node information object, different component are considered that are described as follows:

- Node Information

1) AllowedPod(): It is an integer value in orde to store the number of pods that can be run in the corresponding node.
2) RequestedResource(): Stores all requested resources of pods in the node.
3) AllocatableReosurce(): It stores allocatable resource in a given node which can be equal or less than total capacity of that node. We will cover it later in mathematical implementation.
4) VolumeLimits: The maximum number of volumes that can be considered to a particular node by different cloud providers.

- ResourceLimits

1) GetResourceLimits(): Computes resourcelimits for input pod.

The policy **broker object** has different allocation policies that are defined as functions. A suitable policy can be requested by API server based on the existing state of the system and requirements. Consequently, the broker calls the requested function and then, resource limits are calculated and assigned to pods in a particular node. Then, the broker updates the pods information with API server. At the next stage, the API server passes the information to OS kernel in order to allocate resources.

2) Behaviour-SysMl Activity diagram: Figure 4 represents the activity diagram which depicts the possibility of applying fairness in Kubernetes. Activity or behaviour diagram delivers a good picture of working a system in a sequential way. Each step has been isolated from each other. Three phases of the model which are User, Scheduler and Fairness policies are categorized with their specific operational levels.

V. PRACTICAL EXAMPLE

The implementation of the proposed model is being done in go programming language[19]. Since the work is in the implementation phase, we only refer to a small-scale example. Hence, we design a practical case using four different submitted pods. Accordingly, we setup a single cluster using minikube[20], taking into account a single node with the total capacity of $(2000mi, 2000m)$. Despite the real world cases, we assume that the administrator knows everything regarding the node and applies fairness and resource limits to all pods before sending to the scheduler. In other words, based on the selected policy, the administrator investigates that the sum of the requested resources of all pods are not greater than the node's capacity and based on that information, she specifies resource limits for each pod. This is note that in the real cases that will be included in the next paper, all the processes will be done in a dynamic way using the mentioned algorithms. The examples in this paper are pretty static and only reflects the main concept of the work. The example in figure 5 represents pods and assigned resource limits in administrator's side.

The results shown in Figure 5 based on pods configurations in Figure 6 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). However, the allocatable and actual capacity of the memory is different than the total capacity of the given node. According to the mathematical implementations, in this example, frontend and frontend1 have dominant resource in CPU, while frontend2 and frontend 3 have dominant resource in memory. So, based on the output, the considered policy, resource limits

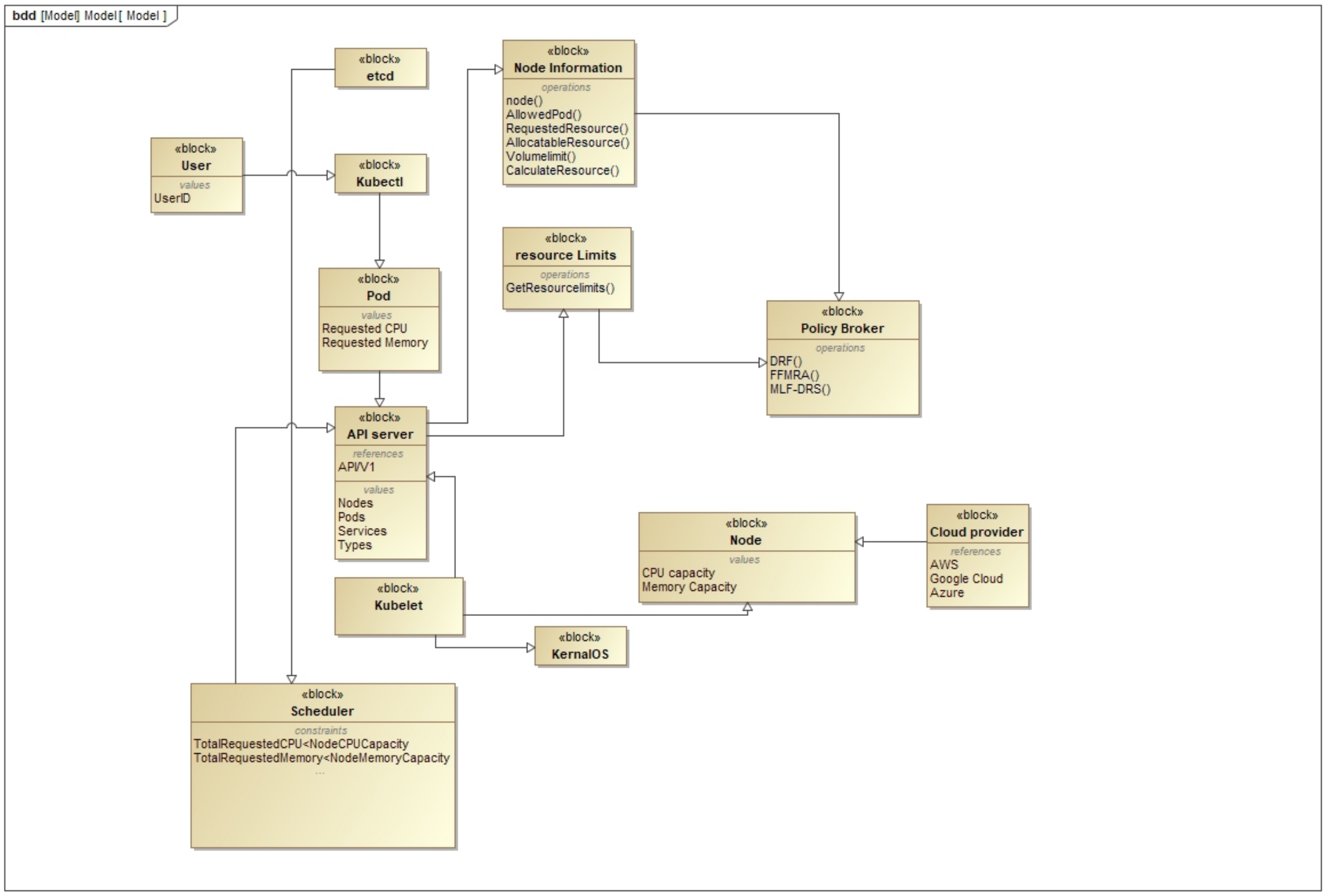


Figure 3: The integration of different fair allocation policies in kubernetes

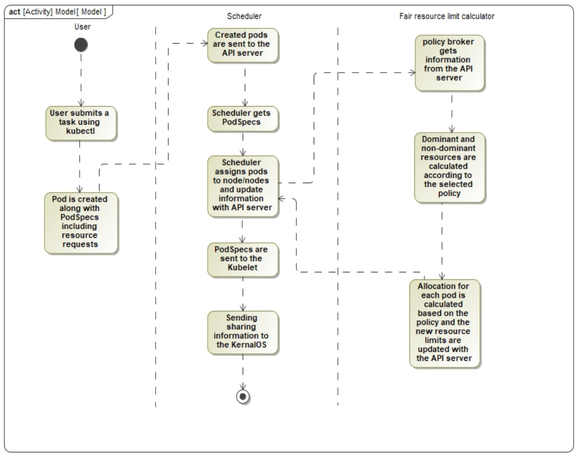

act [Activity] Model [Model]
User
Scheduler
Fair resource limit calculator
User submits a task using kubectl
Pod is created along with PodSpecs including resource requests
Created pods are sent to the API server
Scheduler gets PodSpecs
Scheduler assigns pods to node/nodes and update information with API server
PodSpecs are sent to the Kubelet
Sending sharing information to the KernalOS
policy broker gets information from the API server
Dominant and non-dominant resources are calculated according to the selected policy
Allocation for each pod is calculated based on the policy and the new resource limits are updated with the API server

Figure 4: The Activity diagram of applying fairness in Kubernetes

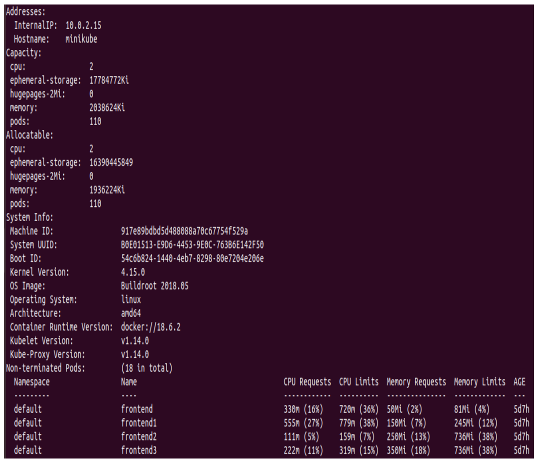
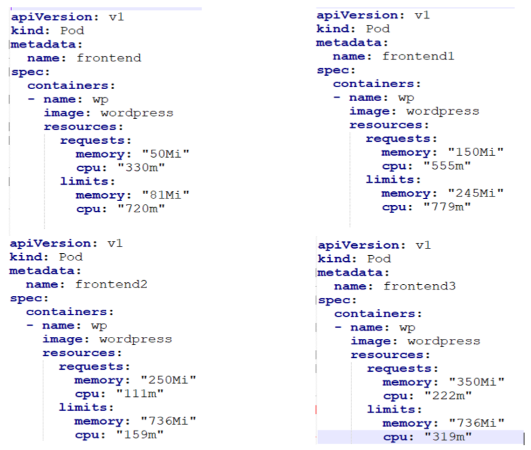

```
Addresses:
  InternalIP:  10.0.2.15
  Hostname:    minikube
Capacity:
 cpu:                2
 ephemeral-storage:  17784772Ki
 hugepages-2Mi:      0
 memory:             2038624Ki
 pods:               110
Allocatable:
 cpu:                2
 ephemeral-storage:  16390445849
 hugepages-2Mi:      0
 memory:             1936224Ki
 pods:               110
System Info:
 Machine ID:                 917e89bdbd5d488088a70c67754f529a
 System UUID:                B0E01513-E9D6-4453-9E0C-763B6E142F50
 Boot ID:                    54c6b824-1440-4eb7-8298-80e7204e206e
 Kernel Version:             4.15.0
 OS Image:                   Buildroot 2018.05
 Operating System:           linux
 Architecture:               amd64
 Container Runtime Version:  docker://18.6.2
 Kubelet Version:            v1.14.0
 Kube-Proxy Version:         v1.14.0
Non-terminated Pods:         (18 in total)
  Namespace                  Name                                  CPU Requests  CPU Limits  Memory Requests  Memory Limits  AGE
  ---------                  ----                                  ------------  ----------  ---------------  -------------  ---
  default                    frontend                              330m (16%)    720m (36%)  50Mi (2%)        81Mi (4%)      5d7h
  default                    frontend1                             555m (27%)    779m (38%)  150Mi (7%)       245Mi (12%)    5d7h
  default                    frontend2                             111m (5%)     159m (7%)   250Mi (13%)      736Mi (38%)    5d7h
  default                    frontend3                             222m (11%)    319m (15%)  350Mi (18%)      736Mi (38%)    5d7h
```

Figure 5: An experiment consists of four pods along with resource limit assignments

are evenly divided among the applications so that first two applications with dominant resource in CPU get 36% and 38% respectively. For last two applications which have dominant resource in Memory, they get exactly 38% of resource limits. Since the efficiency is a basic concept in Kubernetes, we only take care about resource limits and how



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: wp
    image: wordpress
    resources:
      requests:
        memory: "50Mi"
        cpu: "330m"
      limits:
        memory: "81Mi"
        cpu: "720m"
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: frontend1
spec:
  containers:
  - name: wp
    image: wordpress
    resources:
      requests:
        memory: "150Mi"
        cpu: "555m"
      limits:
        memory: "245Mi"
        cpu: "779m"
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: frontend2
spec:
  containers:
  - name: wp
    image: wordpress
    resources:
      requests:
        memory: "250Mi"
        cpu: "111m"
      limits:
        memory: "736Mi"
        cpu: "159m"
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: frontend3
spec:
  containers:
  - name: wp
    image: wordpress
    resources:
      requests:
        memory: "350Mi"
        cpu: "222m"
      limits:
        memory: "736Mi"
        cpu: "319m"
```

Figure 6: The configuration of four pods to be applied in experiment

to assign them fairly among the pods. A pod may consume the maximum amount of resources up to the specified limit, while another one may not consume all possible amount of resources based on specified limit. Basically, according to the Kubernetes, the remaining resources will be allocated to other pods that are waiting to be scheduled.

The conducted experiment is based on the administrator's perspective that can be done in minimum scales. However, in real cases, such as big data scenarios, we need dynamic allocation policies in order to calculate resource limits automatically based on different parameters mentioned in the previous section.

VI. Conclusion and future work

In this paper we proposed a model to calculate and assign resource limits fairly among the pods. We then investigated different platforms that fairness has been considered and discussed that Kubernetes doesn't consider fairness at all. In the designing phase, we integrated three different fair allocation policies, DRF, MLF-DRS and FFMRA in Kubernetes original infrastructure. As practical perspective, we assumed fairness in administrator's point of view and supposed that she is aware of the corresponding node capacity. However, since the model is in implantation level, we couldn't give a real case example including many pods and containers. For the future works, we will implement and analyzed the proposed architecture in real cases considering big data scenarios. Also, we will investigate to propose fair scheduling and load balancing algorithms to be applicable in Kubernetes.

References

[1] C. Pahl, Containerization and the PaaS Cloud, IEEE Cloud Computing, vol. 2, no. 3, pp. 2431, 2015.

[2] S. Singh and N. Singh, Containers Docker: Emerging roles future of Cloud technology, 2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT), 2016.

[3] https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

[4] https://hadoop.apache.org/

[5] http://mesos.apache.org/

[6] X. Huang, Max-min fairness bandwidth allocation and scheduling in wireless ad-hoc networks.

[7] T. Bonald, L. Massouli, A. Proutire, and J. Virtamo, A queueing analysis of max-min fairness, proportional fairness and balanced fairness, Queueing Systems, vol. 53, no. 1-2, pp. 6584, 2006.

[8] https://hadoop.apache.org/docs/r2.7.4/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[9] B, Hindman., A, Konwinski., M, Zaharia., A, Ghodsi., A. D., Joseph., R, Katz., S, Shenker., I, Stoica, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center", Electrical Engineering and Computer SciencesUniversity of California at Berkeley, May 26, 2010.

[10] https://hortonworks.com/apache/yarn/

[11] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, *Dominant resource fairness: Fair allocation of multiple resource types.*inProc. USENIX NSDI , vol. 11, 2011, pp. 2424.

[12] https://github.com/kubernetes-sigs/kube-batch

[13] Hamzeh, H., Meacham, S., Virginas, B., Khan, K. and Phalp, K. T., 2019. MLF-DRS: A Multi-level Fair Resource Allocation Algorithm in Heterogeneous Cloud Computing Systems. In: 2019 IEEE 4th International Conference on Computer and Communication Systems, 23-25 February 2019, Singapore.

[14] Hamzeh, H., Meacham, S., Khan, K., Phalp, K. and Stefanidis, A., 2019. FFMRA: A Fully Fair Multi-Resource Allocation Algorithm in Cloud Environments. In: The 3rd IEEE Symposium on Software Engineering for Smart Systems (SSESS) 2019 19-23 August 2019 Leicester. IEEE.

[15] https://x-team.com/blog/introduction-kubernetes-architecture/

[16] https://rancher.com/docs/rancher/v2.x/en/project-admin/resource-quotas/

[17] http://www.omgsysml.org/

[18] https://www.nomagic.com/products/cameo-systems-modeler

[19] J. Newmarch,Overview of the Go Language, Network Programming with Go, pp. 2127, 2017.

[20] https://kubernetes.io/docs/setup/learning-environment/minikube/