National College of Ireland

A Novel Approach for Fair Resource Allocation in Kubernetes

MSc Research Project
Cloud Computing

Anjalee
Student ID: x19107803

School of Computing
National College of Ireland

Supervisor: Manuel Tova-Izquierdo

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Anjalee
Student ID:	x19107803
Programme:	Cloud Computing
Year:	2018
Module:	MSc Research Project
Supervisor:	Manuel Tova-Izquierdo
Submission Due Date:	20/12/2018
Project Title:	A Novel Approach for Fair Resource Allocation in Kubernetes
Word Count:	XXX
Page Count:	24

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

I agree to an electronic copy of my thesis being made publicly available on TRAP the National College of Ireland's Institutional Repository for consultation.

Signature:	Anjalee
Date:	28th September 2020

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

A Novel Approach for Fair Resource Allocation in Kubernetes

Anjalee
x19107803

Abstract

The computing systems and networks commonly uses resource allocation standards and protocols to secure efficiency and fairness when allocating resources among different users. A problem arrives while selecting allocation rules when resources are limited and not enough to fully meet the demand on its entirety. It is significant to consider their QoS in Kubernetes when designing resource allocation rules. By fairness, it means that all users obtain impartial amount of system resources. Instead of providing fairness to users with good channel conditions, a trade-off is required between maximising the fairness between users and maximising the system throughput. Previous works in the literature of resource allocation algorithms overlook the fairness criteria and enables users possessing good channel conditions to acquire most of the resources. In this research CPU and Memory requests and limits are configured at the time of creation in case of Kubernetes default scheduler. When using Fully Fair Multi-Resource Allocation (FFMRA) algorithm, the resource limits will be applied after the scheduling for better resource allocation. Using this custom scheduler improves the resource utilization when two web applications are considered as proved in the results. The Grafana is used for monitoring the nodes and pods activities.

1 Introduction

Cloud Computing has been uniformly undertaken as one of technological solution for various problems in multiple businesses and areas. The technology presents a pool of computing storage and several other resources that can be obtained over network. Furthermore, cloud applications have become part of everyday lives and are composed from cloud services which are generally built on essence of service-oriented architecture(Joseph and Chandrasekaran; 2020).

Kubernetes is widely used open source technology that manages containerized applications in distributed environments and provides scaling, maintenance and deployment of applications (Beltre, Saha, Govindaraju, Younge and Grant; 2019). The smallest deliverable unit of computing that can be managed and created are termed as Kubernetes pods. These pods are used to bind together the containers that belong to same application. The resources utilised by pods are controlled by settings of requests and limits; requests being the guaranteed amount of resources that can be achieved and limits being the maximal resources that can be obtained. There is a mechanism called cpu-shares which

happens during resource contention in a node, resulting in resource allocation among the pods as per the relative weight of their requests. Additionally, scaling is another way to manage the available resources in an application. Kubernetes presents default autoscalers namely, Vertical Pod Autoscaler (VPA) and Horizontal Pod Autoscaler (HPA). Nevertheless, VPA comes across various limitations that curbs its real time usefulness (Verreydt et al.; 2019).

In a Kubernetes(k8s) cluster, KubeSphere serves as middleware component that accepts various users' tasks and transmits them to a connected k8s cluster based on fairness metrics and custom defined policy. A conventional k8s cluster allows users to submit tasks directly to Kubernetes control plane and permits k8s master to launch the tasks appropriately. Contrarily, KubeSphere comes up with another control layer where cluster admin introduces fairness policies to control and dispatch various tasks for the users (Huang et al.; 2019). FFMRA, the advanced version of DRF is used for allocating of resources with Nginx Ingress Controller in Kubernetes to provide better efficiency (NGINX; 0009).

For the regular work at constant workloads, the default scheduler works great. They also supports feature like pod affinity, priority, tolerations, predicates, and taints that will allow various scheduling principles. With the advancement in cloud computing, the requirement for dynamic allocation and mapping the newly created pods to nodes were needed. Therefore, from Kubernetes 1.6, the multiple and custom scheduler concept were developed. In this study, in case of multi-cluster environment default scheduler is not as effective as the scheduler developed based on FFMRA.

The Research question is :- **Can the resource allocation while scheduling of pods can be improved with FFMRA?**
The objective is to enhance allocation strategy in kube-scheduler by using efficient load balancer and a better scheduler. For this study, small scale application is considered with two submitted pods with different CPU and Memory resources. On the basis of CPU and Memory requests and limits, scheduling is done on nodes. The concept of fairness that is already used in Hadoop, Mesos, Kube-batch and YARN (Hamzeh, Meacham and Khan; 2019). The scheduler is developed in Go language assuming the pods are of equal priority so the decision is made based on the requests and limits allocated. This means calculating the accurate and definite resource requirement at any particular time. The comparison is made between default scheduler and the proposed custom scheduler. The limitations of this research is that scheduling of pods on nodes are done on a single cluster. The theory has proved that the proposed scheduler will work effectively in a multi-cluster environment, but practically not proven. Although, multi-container and multi-node scenarios are covered in this study.

This research is divided into various sections describing the significance of the research done. The Section 2 specifies the research work done in this field, where different algorithms proposed by the scholars are compared and analyzed to identify the best one. Section 3 methodology describes the steps performed in the research with the description of specifications used. Section 4 design depicts the underlying the framework or architecture along with the specifications used for implementing. The practical implementation is briefed in Section 5 with languages and tools used for the research and also the models

developed. In Section 6, the results and the output from the implementation section is analyzed. The last section 7 include the conclusion and future work is discussed.

2 Related Work

In Cloud Computing, obtaining efficient and fair resource allocation is always required along with resource mapping, provisioning, and adaption. This research paper focuses on the fair resource allocation. The methods for distribution of resources among the users with distributed demands have been researched a lot. The fairness is one of the different quality parameters that has been introduced and is still a challenging issue. Therefore, more research is required in this area (Lopez-Pires and Baran; 2017). The equality and fairness are two different concepts, the quality is maintaining the same quantity whereas the fairness is the quality of having an unprejudiced condition (Fleurbaey; 2012).

The growth in containerization has modified the developing, maintaining, and deploying of software. The main features of containers are flexibility and are lightweight. With this approach, different application services are packaged into different containers, and these containers are deployed over physical or virtual machines. Therefore, there was a requirement of container orchestration tool, that will automate and manage deployment, improves scalability and networking of container-based applications (Yegulalp; 2019).

Kubernetes, an open source framework that is widely used for managing, deploying and scaling of containerized applications in a distributed environment. The smallest computing unit that is created and managed is referred as pod. The containers associated with same application containers are assembled in pods. There can be a limitation in resource utilization while setting its limits and requests. The request is the resource amount that is certain to get and the limit is the maximal resource amount a pod can obtain. In case of resource conflict in any node, then there is a division of resources among the pods on the basis of their relative weight as the requests. This technique is referred as cpu-shares. The other way of resource management is scaling of resources and the default scalers are Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA) (Verreydt et al.; 2019).

The main components of Kubernetes are master and nodes. The master equipped components are Master and with node components are referred as Node. The Master, which consist of Kubectl, Controller manager, Api-Server, Scheduler and Etcd as described in the Fig controls the complete operation of the platform. The Api-server maintains interfaces for resources to perform various operations, Etcd act as a storage for the data in a cluster, Scheduler governs the scheduling of resources, Kubectl act as an administrative tool to manage all the components, Controller-Manager is accountable for implementing different controllers. Node component coordinating with Master in managing the resources. The Kubelet and Kube proxy are the components of the Node. The responsibility of Kubelet is operating and maintaining the pods. The responsibility of Kube proxy is to monitor information transmission between pods in the particular cluster(pods that are linked to same Master) (Huang et al.; 2019).

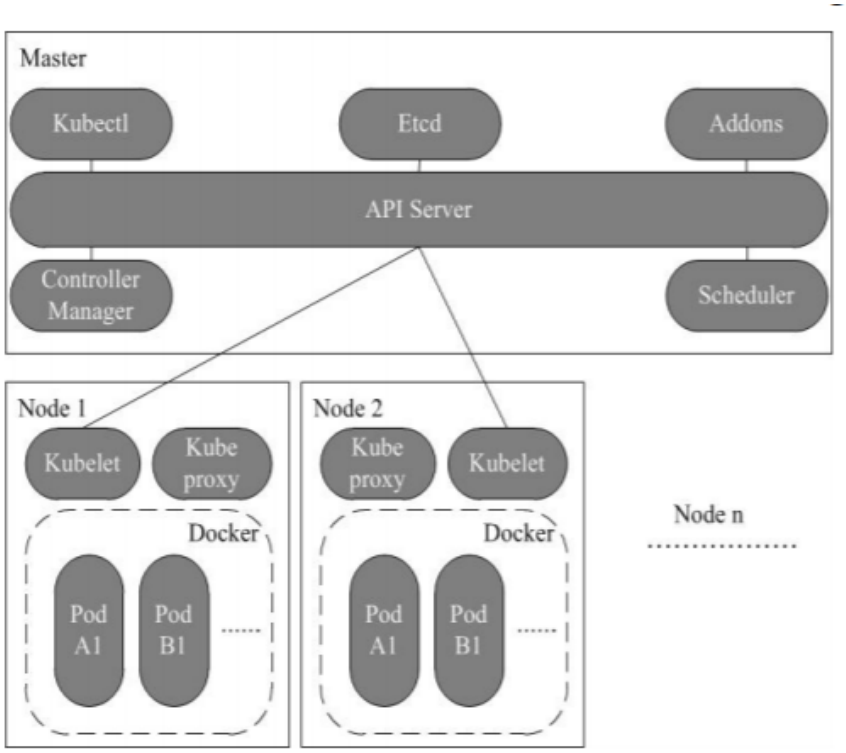

Figure 1: Kubernetes Architecture

The challenges in kubernetes are to position resource limit based on node information and resource requests after scheduling. And Kubernetes cannot perform dynamic resource management. So, there is a problem when two nodes has a single container and the resources are allocated as there resource requests. In case the pods consume less than the capacity assigned to node by default, there is a wastage of resources. Also, the resource limits for scheduling time is not considered by the scheduler. Therefore, after scheduling of pods in particular node, the fair resource limits should be determined on the basis of the participation of pods in that node. (Hamzeh, Meacham and Khan; 2019)

2.1 Kubernetes vs. Docker

Kubernetes is not an alternative or substitute for Docker but is a replacement of technologies developed around Docker. That technology is Docker Swarm. Kubernetes is more complicated than Swarm but in long run, it provides better resilient and managed application infrastructure. Docker Swarm can be preferred while working with the small container clusters (Marathe et al.; 2019). A swarm is a group of machines merged with operating Docker, that are handled by swarm administrator. The Docker Swarm is responsible for clustering along with scheduling of Docker containers. The request execution is based FIFO strategy. The Docker Swarm selects the suitable machine on the basis of their CPU cores usage. The strategies used are: (i) Speed strategy, that is by default used, where a node is selected with least containers, (ii) Random strategy, where node is determined randomly, (iii) Binpack strategy, choose node with overloaded containers (Cérin et al.; 2017).

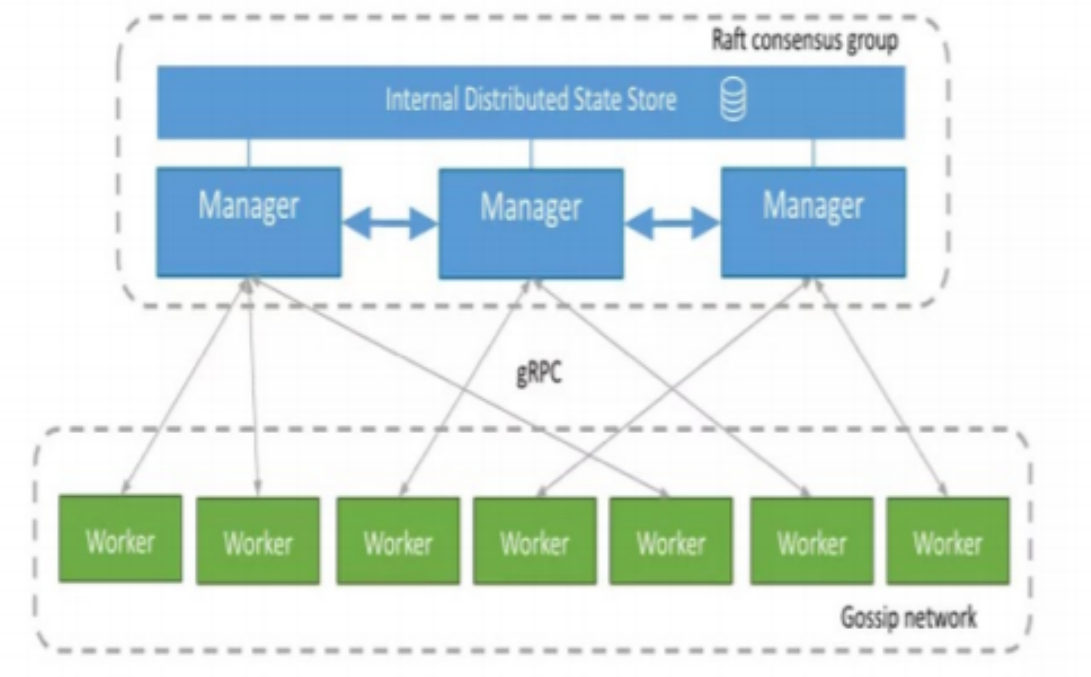

Figure 2: Swarm Cluster

The research has proved that Kubernetes performance and flexibility is much better than Docker Swarm. One more advantage with Kubernetes is that it is fault tolerant and self-healing.

2.2 Kube-scheduler

Kubernetes default scheduler kube-scheduler, that will allocate pods on nodes. For selection of destination node, it follows a two- step process. First step is to identify the worker nodes on which the pods will be running based on certain filter set (predicates). As an instance, if a pod is identified as key-value pair, then scheduler will choose nodes which matches the pair. Therefore, not matched nodes are discarded. The second step is to rank the unused nodes by the priority set functions. Each of this function attach a score with every node and the summation of these scores is calculated as a final score. This way each of the remaining node is assigned a score and the one with highest score is selected to execute the pod. In case the same score is achieved by many nodes, then selection process is random (Rossi et al.; 2020).

2.3 Scheduling of Pods

Scheduling of pods includes identifying the pods that is needed to be run on the nodes in a cluster. The basic steps for scheduling of pods on a node is shown in the below figure. Kubernetes scheduler is the process that works beside different components as API server and observe API for the pods to be scheduled. First, to find an appropriate node for pod scheduling, then schedule one pod at a particular time. Scheduling is decided by the policy that includes priority and predicates function (Vohra; 2017).

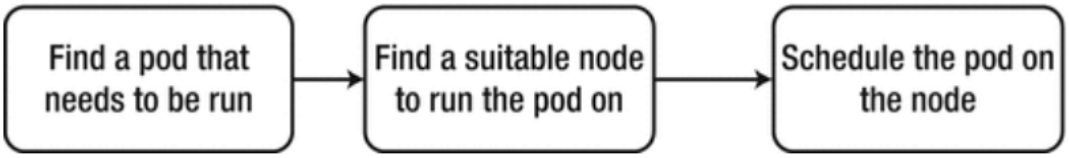

Figure 3: Pod Scheduling Steps

The research has also shown that there is three ways to scheduling:

1. Default Scheduler.
2. Custom scheduler implementation along with the default scheduler.
3. "scheduler extender" implementation which is called by standard scheduler at a last step while taking decisions.

2.4 Resource Allocation

The dynamic resource allocation algorithm can be described as five below steps:

1. Collect the requirement of cluster's resource, memory and CPU.
2. If the utilization is smaller than the threshold, then proceed with step (3) else with step (5).
3. Additional pods and nodes are calculated based on the basis of the ratio and the least number of nodes added.
4. The present pods and worker nodes number are added with the additional ones calculated in step (3).
5. The final count of pods and worker nodes are returned (Surya and Imam Kistijantoro; 2019).

2.5 Max-Min Fairness

A well-admired policy termed as Max-min fairness is an approach for defining fairness in resource allocation by assigning shared resources among conflicting demands. The policy focuses to assign as many resources to low demand jobs and then equally allocate resources among the remaining jobs with high or competing demands.With some refined properties of Max-min fairness such as sharing-incentive, pareto efficiency, strategy-proofness and envy-freeness, the policy is always attainable on a single machine. These properties are traditional notions of fairness. Sharing incentive indicates that every job should thrive on sharing resources, rather than statically splitting the resources. Pareto efficiency denotes that in order to increase the allocation of a specific job, allocation of another job needs to be decreased. Strategy-proofness property allows incentive compatibility by not increasing allocation for a job with misleading demands. Envy- freeness interprets common need of fairness which says a job would not favour allocation of another job (Shafiee and Ghaderi; 2020).

2.6 Dominant Resource Fairness

In computing systems, sharing of resources among multiple users are an elementary research area. DRF is an extension to max-min fairness when it comes to variety of resources. Another concept on fairness is Proportional Fairness (PF). DRF fairness policy is far better than PF as they can be represented as α -fairness family member (α represents fair resource allocation which indicate optimization. The highest the value of α indicates the fairer allocation of resources. For PF, $\alpha = 1$ and for DRF $\alpha = \infty$ and therefore, it can be concluded that DRF is fairer than PF (Youngmi Jin and Hayashi; 2016).

The dominant share of resources is defined as the maximum ratio of the resources distributed in a server per user and are equalized. But with heterogeneity of the servers, dominant resource solution is imprecise. The improved version is DRFH, transitioning from single server to many heterogeneous servers, where resource in the form of storage, processing, and memory are pooled by heterogeneous servers (Wang et al.; 2014).

Dominant resource fairness (DRF) is a multi-resource fairness policy and called to be successful tool in terms of an effective resource allocation (Zhao et al.; 2018). The better mechanism where secondary resource are used along with dominant one which supports efficiency greater than DRF, but only pareto-optimal and envy-freeness are achieved (Jiang and Wu; 2018). Kubebatch allows Kubernetes to create scheduling decisions based on DRF. Although, DRF does not include other fairness metrics of a shared cluster, but considers its usage along with resource demand and average waiting time for a task. Another policy driven meta-scheduler has been developed for a Kubernetes cluster, termed as KubeSphere. The policy enables efficient scheduling of individual user's tasks based on respective user's resource consumption and overall resource demands.

The ks8 default scheduler is well informed with task queues and cluster resources in order to include the custom fairness policy. In addition to the default scheduler, the main goal of KubeSphere is to include another layer of scheduling mechanism. However, an alternative could be modifying the Kubernetes allocation module, but that demands persistent community support for building and managing the code. Therefore, KubeSphere can be utilised as an elective module along with off-the-shelf Kubernetes (Beltre, Saha and Govindaraju; 2019).

DRF is very efficient for single server and not much effective for multiple servers because it may result to inefficient allocation. As a result, considering the problems from multiple servers and as a generalisation of Max-min fairness algorithm, a multi-level fair resource scheduling and allocation algorithm (MLF-DRS) could be a solution. MLF-DRS provides allocation in the same rate for dominants and non-dominants as set of resources. Additionally, this solution seeks dynamic allocation policy relative to server specifications such as resource capacity pool and users total requested resources (Hamzeh, Meacham, Virginas, Khan and Phalp; 2019).

2.7 FFMRA

FFMRA works for both dominant and non-dominant type of resources. The dominant is the user's maximum allocation of resources and the non-dominant is the user's minimum resource share. The allocation is done without interfering the other users. The fairness properties met by FFMRA algorithm are pareto efficient (user cannot increase its allocation by reducing the other user's allocation), envy-free (user can't select other user's allocation), strategy-proof and sharing incentive (Hamzeh, Meacham, Khan, Phalp and Stefanidis; 2019).

DRFH also assures the properties mentioned. While using multi-server environment, the users are assigned dominant type resources on various servers and total resource for each user is calculated. DRFH equalizes each user's dominant resource share, that is the highest ratio of any type of resources allocated to the user in the complete resource pool. DRFH is proved to pareto-optimal and envy-free. It also supports bottleneck fairness, single-server DRF and resource fairness (Wang et al.; 2014).The existing problems in DRF can be solved by increasing per-server resource functions implementing classes by which the efficiency and fairness can be fulfilled. This research included virtual dominant type of resources also. Overall, it will increase the efficiency but still not full utilization is achieved (Khamse-Ashari et al.; 2018).

The other algorithm in this field is the combination of Nash-Bargaining (NB) and Lexicographically-Max-Min-Fair (LMMF). They support maximum resource usage and fair allocation when more than one tasks are submitted by the user. There combination satisfies pareto-optimality, envy-freeness and the max-min fairness (Zarchy et al.; 2015).

In order to provide better efficiency, Ant Colony algorithm (ACO) along with Particle Swarm Optimization (PSO) (Wang et al.; n.d.) mechanism was introduced to schedule pod for better results. The ACO works on distributing the containers and balancing the resource on docker. The latter mechanism works parallel with the weight and the random factor to lower the total cost (Kaewkasi and Chuenmuneewong; 2017).

The Improved version of ACO, where the ants were assigned to the node for which pheromone concentration are maximum and after that become inactive. In this, few ants receive the highest of the pheromone and the remaining one receives some random value that are accessible locally. That is how random nodes are selected for deployment. There is also an improved version of PSO, where for every repeating particle weight ratio calculation is done based on that allocation is determined. The weight of the particles will determine its next iteration. The information is moved between the particles. On integrating these mechanism the overall cost is reduced with efficient scheduling and speed (Wei-guo et al.; 2018).

2.8 Conclusion

The algorithms that are introduced in this research are combination of ACO and PSO, DRF, DRFH, integration of NB and LMMF, MLF-DRS and FFMRA. The table 1 below depicts the fairness property supported by each algorithm. The definition of the properties is already there in the above paragraphs.As for ACO and PSO, do prevent the faults of

the local inspecting of the pods and provide cost effective solutions. The drawback is when there is increase in load, then maintaining the efficiency with reduced cost becomes difficult. When considering fairness, it didn't support SI and EF. The NB and LMMF integration support only EF and PO, that means the user have the permission to run the programs that require higher resources than the allocated one and there is a chance of incorrect resource allocation. DRF can easily be replaced by DRFH that provide a better efficiency in heterogenous environment. The SI property well supported by DRFH where user can dynamically change the resource allocated based on the requirement. The MLF-DRS and FFMRA are better than other algorithms as they work for both dominant and non-dominant type of resources and also supports multi-server environment. They also provide full resource utilization and the better performance. But, the implementation of FFMRA is quite simpler when compared to MLF-DRS and therefore, considered for this research.

Table 1: Different Algorithm Comparison

Property	**ACO & PSO**	**DRF**	**DRFH**	**NB & LMMF**	**MLF-DRS**	**FFMRA**
Sharing incentive (SI)	NO	NO	YES	NO	YES	YES
Envy freeness (EF)	NO	YES	YES	YES	YES	YES
Pareto optimality (PO)	YES	YES	YES	YES	YES	YES
Strategy proofness (SP)	NO	YES	NO	NO	YES	YES

3 Methodology

This research methods revolves around the scheduling of pods. In the pod scheduling lifecycle, first step is the pod creation after which it will look for other component to get started and the state is stored in etcd along with the name of node unfilled. The scheduler will detect that a new pod is created without any node attached. It will find the best fitted pod. The apiserver will bind the pod to node and then save the state to etcd component. Kubelets with apiserver administers these pods and then start the containers. The logic to identify the best node can be customized. The components shown

in the figure 4 are loosely coupled. The unbound pods inside the cluster are watched in a loop by the apiserver. The best node is selected for a pod. The request is made to connect endpoint to the apiserver.

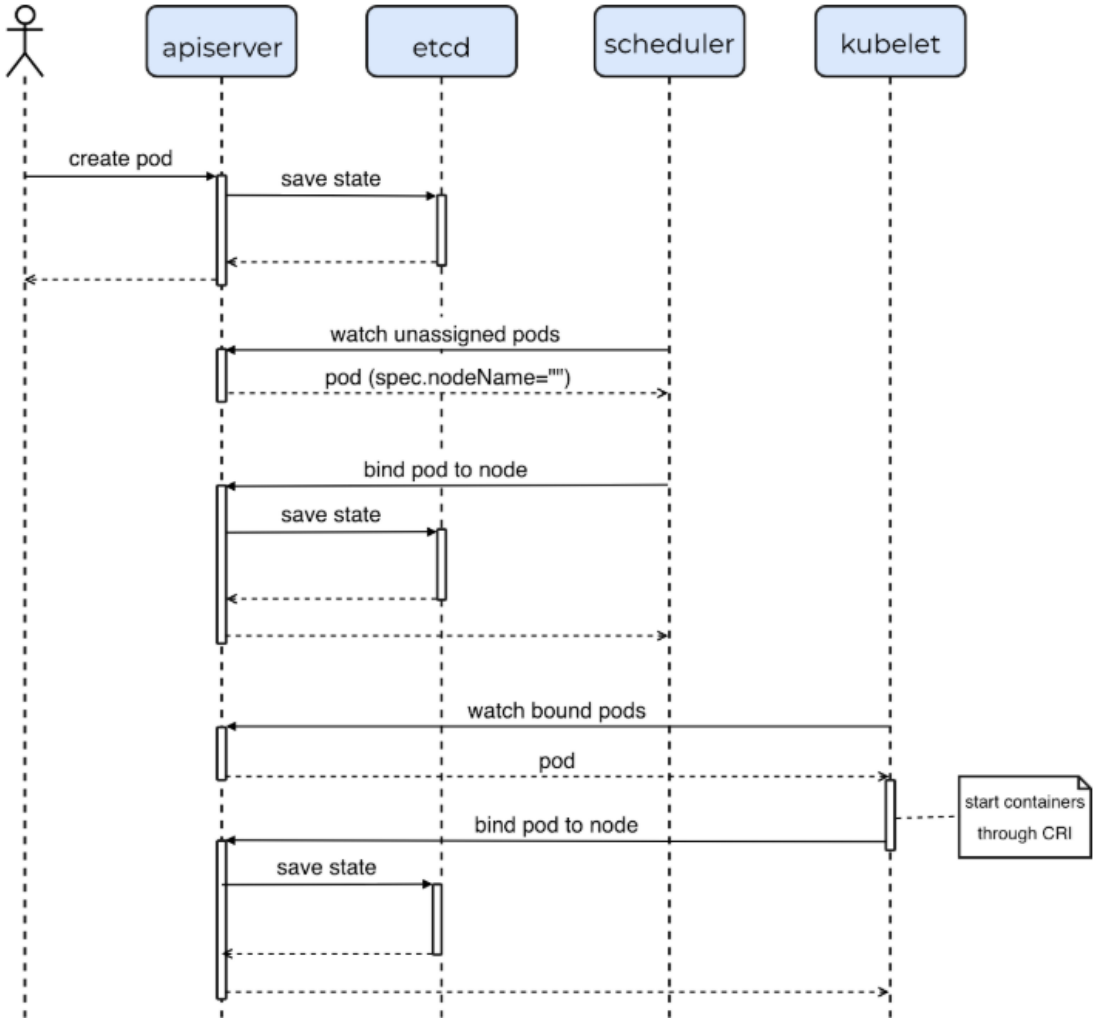

Figure 4: Pod lifecycle

3.1 Digital Ocean Kubernetes (DOKS)

[1] The containerized app are run with DOKS, that supports cluster autoscaling, tokenized authentication, minor version upgrades, and the latest Kubernetes release. User can connect to DOKS clusters using your DigitalOcean API access token, in addition to the previously supported certificates.The Kubernetes cluster are created from the control plane in DOKS. The Docker is used for containerization. And the comparison is done with respect to the Kubernetes default scheduler performance and resource utilization.

[1] `https://www.digitalocean.com/products/kubernetes/`

3.2 DockerHub

[2] With DockerHub, the installation of Docker on VM is quite simpler and also the set up of repositories and tags. The latest docker version is available providing networking with containers and the security with built in scheduling property.

3.3 Nginx Ingress controller

[3] As shown in the below figure 5, a component named as Ingress is passed between the different services and the loadbalancer. In this scenario only single loadbalancer point is used to connect to the clients. In case there is traffic on the loadbalancer, then Ingress will back up and will be used as a host. The path attached with the request will route it to the appropriate service. This is quite better than the default loadbalancer as in case of traffic different loadbalancer will be added for every service. The other approach is to use Traefik, which is simpler to implement but doesn't support complex projects.

Figure 5: Nginx Ingress

3.4 Monitoring Tool

[4] The DOKS provide many software for monitoring, Kubernetes dashboard is available without any extra set up to be done. Grafana and Prometheus, both are open-source can be set up in the cluster to analyze the pods and nodes activities as seen as the graphs. The information can be pulled over http and data can be viewed in a timely manner for the resources like CPU and Memory, schedulers, clusters, nodes, pods and different views are selected. Grafana is accessed only by port forwarding and not available publicly. The Prometheus will show the logs for CPU, or Memory. The alert can also be set on specified mail.

[2] `https://hub.docker.com/editions/community/docker-ce-server-ubuntu`
[3] `https://kubernetes.io/docs/concepts/services-networking/ingress/`
[4] `https://grafana.com/docs/grafana/latest/features/datasources/prometheus/`

4 Design Specification

To perform this research, oracle virtual box 6.1 is used for setting up the virtual machine with Ubuntu 20.04 LTS. The virtual machine with 2 CPUs and 4GB RAM is created on virtual box. Digital Ocean (DO) is used as cloud server and for containerisation, Docker is used. Docker Hub is used for creating The two web applications app1 and app2 are deployed. Kubernetes cluster is created on DO with 2 nodes. The kubectl version v1.18.3 and doctl tools are installed in the local virtual machine. With doctl, DO controls can be accessed. With kubectl, pods status and names, cluster information can be checked. The below table will show the design specifications used for this research set up:

Table 2: System Configuration

Tool/Platform	Specification
Oracle Virtual Box	6.1
Ubuntu	20.04 LTS
Docker	19.03.12
Kubernetes	1.18.3
Ngnix	30.0
Doctl	1.46.0-release
Grafana	6.7.3
Prometheus Operator	0.38.1
Code Language Used	GO
Manifest Language	YAML
Application Container	Node.js

4.1 Proposed Design

After the cluster is set up on DO, kubectl and doctl is installed to manage the cluster from local machine. For getting the access, one api token was generated in the control panel that needs to be set while installing it.

Kubectl is used as a client for configuring the cluster from the local virtual machine as a REST API command line tool. Kubectl search for config file in $HOME/.kube directory. The kubernetes Master and Node is configured with the DO and the local virtual machine. The important yaml used for intercommunication are:

1. secret.yaml: This file is created inside k8s directory. Kubernetes uses this to fetch the images from the pods created.

2. deployment.yaml The deployment.yaml and service.yaml are created for both the apps app1 and app2.

3. service.yaml The name mentioned in the secret.yaml is used while deployment and in service.yaml.

4. ingress.yaml and cloud-generic.yaml for creating a load balance service.

5. scheduler.yaml for implementing the FFMRA scheduler in GO language.

4.2 Proposed Architecture

The activity diagram depicts a sequential steps taken for the research. First the cloud provider is set up, following with configuring the pod and next phase is scheduling and determining the efficient and fair resource utilization. This depends on the CPU and memory resources requests and limits assigned to the pods to be scheduled. The API server functions as cluster gateway. All the cluster configuration and information are accessed through API server. The kubectl act as a client and is then attach to API server as sown in the figure.

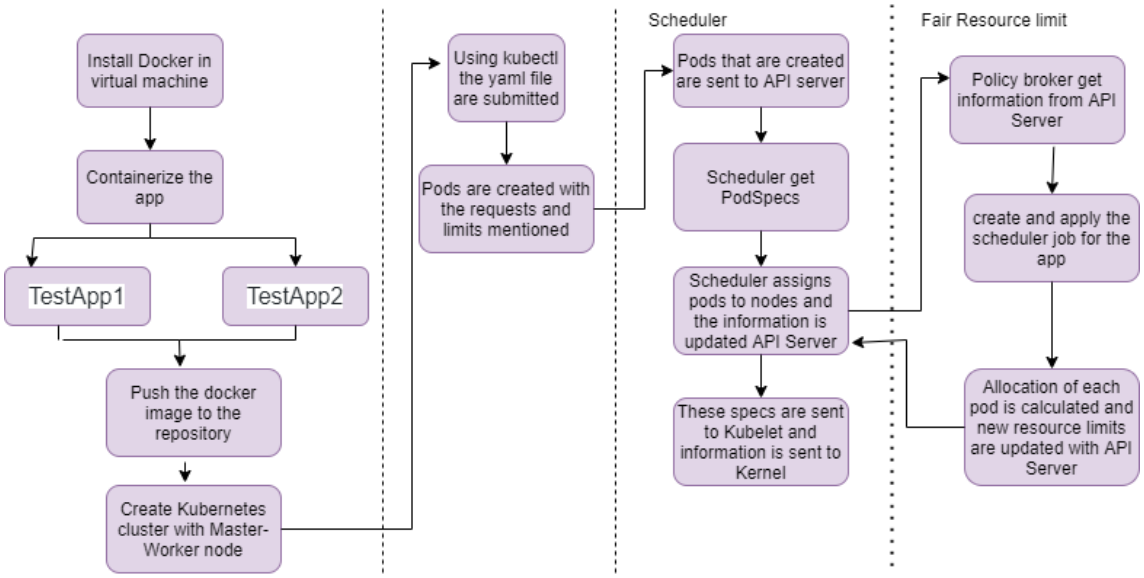

Figure 6: Activity Diagram of the Architecture

5 Implementation

When a pod is scheduled, the containers must have sufficient resources to proper execution. If a large application is scheduled with the insufficient resources then there can be out of resources problem and the node will stop functioning. There is also a possibility for applications to use resources more than they asked for. This can be due to many replicas or the poor configuration that lead the application to perform unexpectedly.

Kubernetes control resources with limits and requests mechanisms. The container resources that are assured to get are defined as the requests. Kubernetes will schedule the container on node only if it has sufficient resource. Limits are to assure that container should not exceed certain value. The container can use resource upto the limits defined and after that it is deprived. The relation between requests and limits is that request are always greater than limit. If not, then there is an error and the container will stop. Pods generally include single container, but there can be multiple containers. Every container has its own request and limit, as every time pods are scheduled in a group, requests and limits of each container are added to get an combined value. For handling the requests and limits, quotas are set at namespace and container level.

DOKS is used to deploy and manage Kubernetes clusters avoiding the complexity of administrating containerized applications. These clusters will be integrated with DO block storage and Load balancers. The cluster created on DO with the CPU resource as 4vCPUs and memory resource as 8 GB as shown in the below figure:

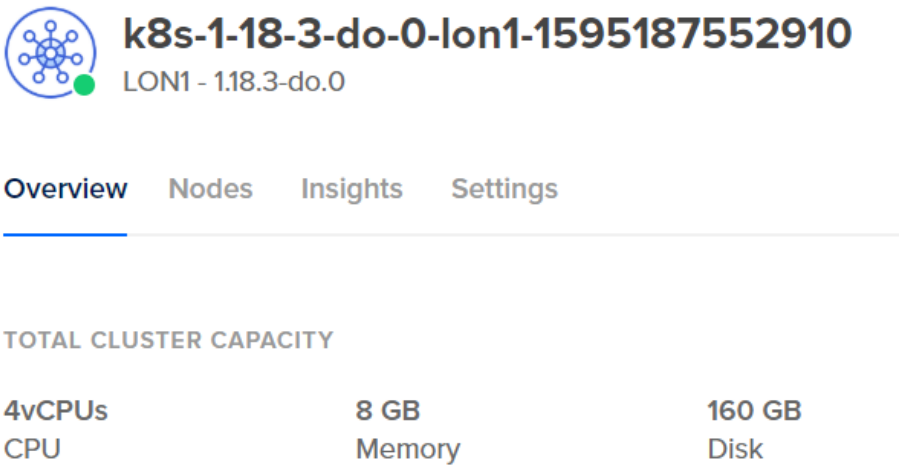

Figure 7: Kubernetes Cluster

To check the health of Kubernetes component- controller, etcd and scheduler, following command should be used: **kubectl get cs** and As shown in the figure all the components are installed properly and are healthy to use.

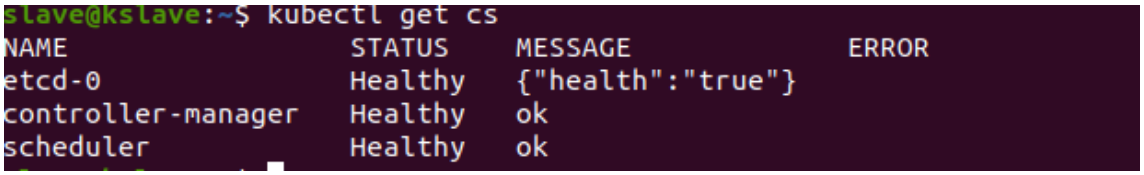

Figure 8: Kubernetes Component Status

The two TestApp1 and TestApp2 web app are created in which app1.yaml and app2.yaml are there where CPU and memory requests and limits are assigned. The CPU is described in millicores and the memory in bytes. To containerize the app, "Dockerfile" is created. Also, "Makefile" is created to automate the docker commands. As seen in the Figure 9, the CPU and memory resources used for the experiment are:

Table 3: Resources

App1	CPU	Memory
Requests	200m	400Mi
Limits	350m	650Mi
App2	CPU	Memory
Requests	100m	300Mi
Limits	400m	300Mi

The aggregate of CPU and memory is less than that of cluster including the replica. So, no issue will be there in resource management and proper scheduling is done.

Figure 9: Pod Configuration

The class diagram below depicts the connections and flow of all the entities used in the study. As shown in the figure, Digital Ocean is the cloud provider which will create the cluster and ubuntu will interact with Kubernetes kubectl. The resources specifications and limitations used in the experiment is already described above. The default scheduler kube-scheduler is used where Node CPU capacity and Node Memory capacity is greater than Total Requested CPU and Total Requested Memory respectively.

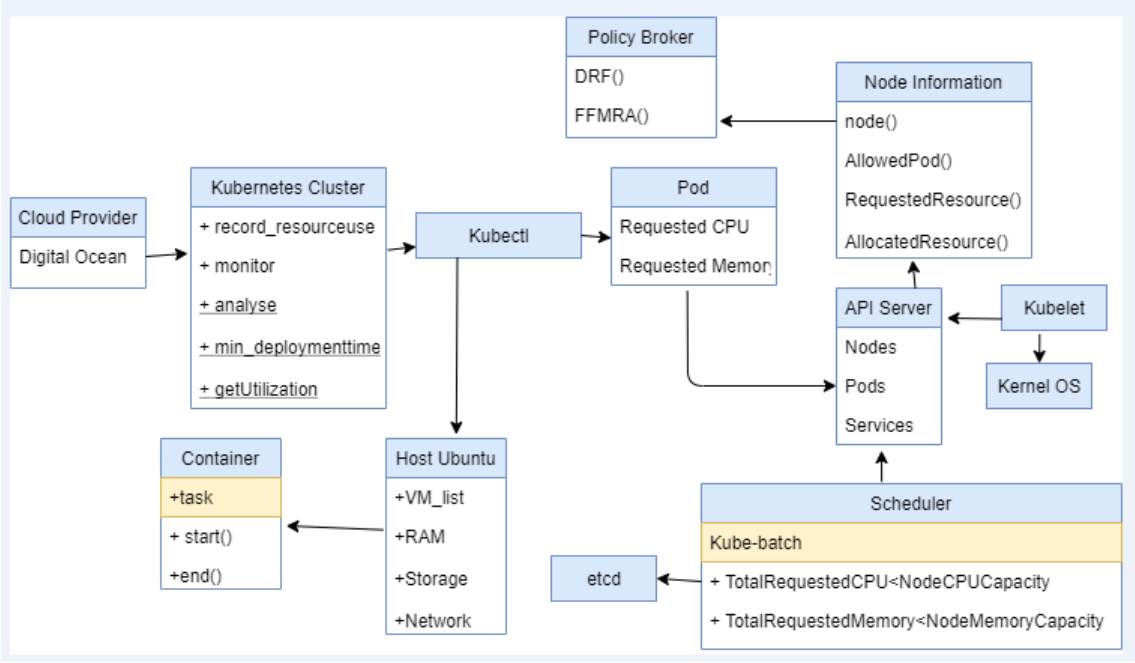

Figure 10: Class Diagram

Each and every container that exist in the pod are configured to have unique port and IP addresses (called as ClusterIP). Every node has its private and public IP addresses and the ports assigned to the nodes are referred as NodePort, from where these pods can be

accessed. As it can be seen in the output, the address 10.245.230.192 and 10.245.147.108 are the ClusterIPs assigned to app1 and app2 respectively. With these address, the applications are available internally at port 80. The traffic will be forwarded to container Port to 3000 and 3001 on the selected pods.

```
NAME                 TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
service/app1         ClusterIP   10.245.230.192   <none>        80/TCP
service/app2         ClusterIP   10.245.147.108   <none>        80/TCP
service/kubernetes   ClusterIP   10.245.0.1       <none>        443/TCP
```

Figure 11: Class Diagram

The default scheduler in Kubernetes as described guarantees that nodes with sufficient resources will only be placed for pods to deploy. For efficient resource utilization, DRF and FFMRA scheduling algorithm that are established on the fairness principle. The custom scheduler code is written in go language and also the default scheduler. The experiment shows the full resource management with fairness in case of scheduling of multiple pods and containers in the node. It equalizes the resources that are dominant and non-dominant to provide fairness in multi-tenant environment. This is done by taking total resource capacity, consumed resource and user's dominant and non-dominant share as an input. The allocations are done by mapping of jobId with the resource capacity. If there is no node that fulfills all the described demands of the pod, then preemption principle activated for the remaining pod. With this principle, preemption will try to locate a node and remove the pods with lower priority. If the node is found, lower priority are removed from node.

From Digital Ocean Kubernetes dashboard, the details of pods, nodes and resource utilization that are deployed on the cluster can be analyzed and for graphical representation, Prometheus and Grafana are deployed on Kubernetes.

6 Evaluation

To illustrate the practical usage of FFMRA scheduler and if the resource allocation can be improved this experiment was performed. The two Node.js application containers were deployed with their different pod configuration in yaml file on the nodes. As discussed above the aggregate of CPU and Memory requests should be less than that of the Kubernetes cluster CPU and Memory respectively. The proper analysis is done when compared to the default Kubernetes Scheduler. Grafana and Kubernetes dashboard is used for graphical representational of usage of the resources. Prometheus is set up to get the logs.

6.1 Using Default Kube-Scheduler

In this set up no extra implementation is done. Only the Nginx Ingress load balancer is added and no change in the scheduler component. Therefore, pods are chosen randomly

and Node.js application are deployed as shown in Figure 12, where every pod will be started. Figure 10 below depicts that 25% of the Requests are allocated for CPU and 26% for the Memory when considered for one of the application.

```
slave@kslave:~/Documents/K8sTest/K8s$ kubectl get pods
NAME                    READY   STATUS    RESTARTS   AGE
app1-78bd6c6c9b-scxtv   1/1     Running   0          4d23h
app1-78bd6c6c9b-zk9qf   1/1     Running   0          4d23h
app2-565bd4b94f-n7279   1/1     Running   0          4d23h
app2-565bd4b94f-wj8xx   1/1     Running   0          4d23h
```

Figure 12: Default Scheduler Deployment

```
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests     Limits
  --------           --------     ------
  cpu                512m (25%)   202m (10%)
  memory             832Mi (26%)  125Mi (4%)
  ephemeral-storage  0 (0%)       0 (0%)
  hugepages-2Mi      0 (0%)       0 (0%)
```

Figure 13: Default Scheduler Resource Allocation

The CPU and Memory usage for the particular node as shown by the Grafana dashboard is as:

Figure 14: Node Utilization

6.2 Using Fair Scheduler

The Kubernetes clusters favours various schedulers, default scheduler can be used along with custom scheduler that can be used for special pods. The custom scheduler for this research is implemented in Go language that will find a node for every new pod. With Nginx Ingress load balancer and the scheduler the resource utilization can be improved.

When compared with the default scheduler, the CPU Requests allocation increased to 60% and the Memory increased to 26 %. However, with FFMRA resource limits are also changed. The CPU limits is now 40% and Memory limits is increased to 19%.

```
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests     Limits
  --------           --------     ------
  cpu                1202m (60%)  802m (40%)
  memory             829Mi (26%)  618Mi (19%)
  ephemeral-storage  0 (0%)       0 (0%)
  hugepages-1Gi      0 (0%)       0 (0%)
  hugepages-2Mi      0 (0%)       0 (0%)
Events:              <none>
```

Figure 15: FFMRA Scheduler Resource Allocation

The above figure shows the allocation for one of the app. Using Grafana, both "app1" and "app2" can be analyzed by the figure, where CPU usage for app2 is around 60% and that of app1 is 28%.

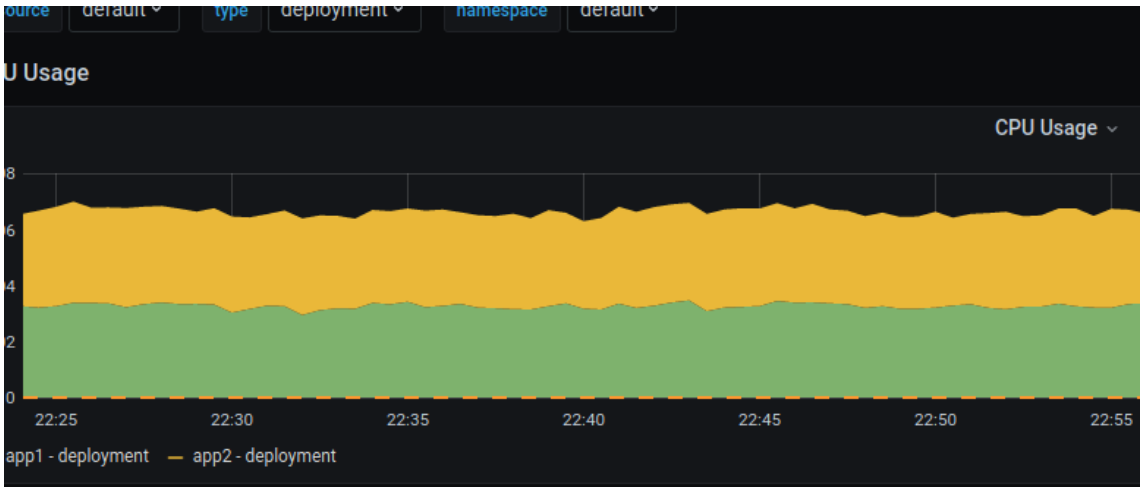

Figure 16: CPU Usage for the Application

The Memory Usage for the Node.js applications "app1" and "app2" is shown as:

The Kubernetes dashboard supported by the DO also shows that ffmra-scheduler is running along with "app1" and "app2" and their replicas as described in the pod configuration.

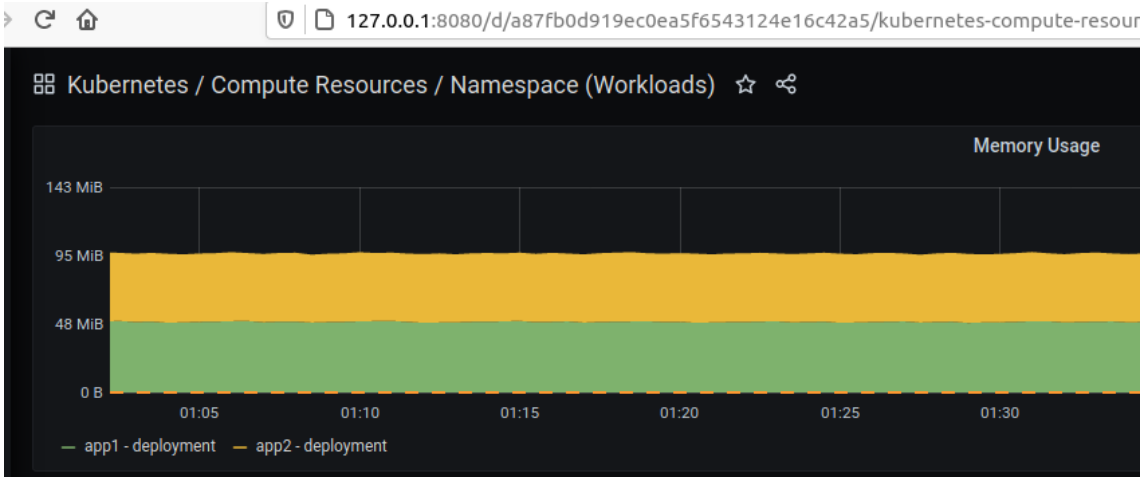

Figure 17: Memory Usage for the Application

Pods

	Name	Namespace	Labels	Node	Status
✓	ffmra-scheduler-59447648cf-tdzgz	default	app: ffmra-scheduler pod-template-hash: 59447648cf	pool-71atsyh52-3jrhx	Running
✓	app2-565bd4b94f-wj8xx	default	app: app2 pod-template-hash: 565bd4b94f	pool-71atsyh52-3jrhy	Running
✓	app2-565bd4b94f-n7279	default	app: app2 pod-template-hash: 565bd4b94f	pool-71atsyh52-3jrhx	Running
✓	app1-78bd6c6c9b-scxtv	default	app: app1 pod-template-hash: 78bd6c6c9b	pool-71atsyh52-3jrhx	Running
✓	app1-78bd6c6c9b-zk9qf	default	app: app1 pod-template-hash: 78bd6c6c9b	pool-71atsyh52-3jrhy	Running

Figure 18: Kubernetes Dashboard

6.3 Discussion

As described in the experiment of resource allocation using default and custom scheduler, there is an increment of utilization for CPU resources request as well as for limits. But, in case of Memory, there is no change for resource request but the limits are changed. Kubernetes's Kube-batch has drf as plugin and is used for providing the fairness among the pods. That means it is already proven that drf practically provides fairness. But the drawback of DRF as described in the literature review is overcome by FFMRA. So theoretically, it is proven that FFMRA is better than drf and in future can be replaced in the Kube-batch implementation as it provides full utilization of resources. However, when implemented practically on two Node.js application container and deployed on Kubernetes environment, the resource utilization was not 100% as expected.
When using the default scheduler, the pods are deployed randomly by this Kubernetes component. As shown in Figure 9, "app1", "app2", and their replica set are deployed and their status is changed from "Pending" to "Running". The only criteria that is fulfilled by the pods to be scheduled are that the aggregate of the resource requests of CPU and Memory should be less than that of the Kubernetes cluster. The limits defined in the YAML file is the amount after which the pod will stop running. The difference between the Figure 10 and Figure 12 of resource allocation is that, with ffmra-scheduler (this name is defined in the ffmra.go file) CPU Requests utilization is incremented by 35% while the Memory remains the same. The resource limits for CPU is increased by 30% and for Memory by 15%. The Figure 15 prove that ffmra-scheduler is working as expected. Figure 13 and Figure 14 provides information for the comparison of the CPU and the Memory usage of the app1 and app2. The light green colour is for the app1 usage and the yellow is for app2. Figure 11 determines the whole activity of the Kubernetes node. Overall, FFMRA is efficient than Kube-Scheduler.

7 Conclusion and Future Work

The kube-scheduler starts automatically after all the other components are initiated. But, for implementing the custom scheduler, there require a design change. The above study do satisfy the objective of this research. And, it has been proved that resource allocation can be improved by using custom scheduler: ffmra-scheduler in Kubernetes. Although, there was no full utilization of resources but there is an improvement in the allocation policy and thus the efficiency. Through this research it has been highlighted that FFMRA scheduling algorithm runs effieciently with Nginx Inress loadbalancer. Kubernetes default scheduler is implemented using Go language and also, the ffmra-scheduler. The results shows that there is some limitation in case of Memory resource type, but overall the efficiency is improved.

Generally, the containers with some dependencies are meant to be allocated on same node, so that latency is reduced. Working with this pattern require further research where dependant containers will run on same node. The algorithm can be implemented in case of multi-cluster environment and can be analyzed. Also, it can added in kube-batch plugin where already drf is running successfully.

References

Beltre, A. M., Saha, P., Govindaraju, M., Younge, A. and Grant, R. E. (2019). Enabling hpc workloads on cloud infrastructure using kubernetes container orchestration mechanisms, *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, Denver, CO, USA, USA, pp. 11–20. 2019 paper, ranking not available.

Beltre, A., Saha, P. and Govindaraju, M. (2019). Kubesphere: An approach to multi-tenant fair scheduling for kubernetes clusters, *2019 IEEE Cloud Summit*, pp. 14–20.

Cérin, C., Menouer, T., Saad, W. and Abdallah, W. B. (2017). A new docker swarm scheduling strategy, *2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2)*, pp. 112–117.

Fleurbaey, M. (2012). *Fairness, Responsibility, and Welfare*.

Hamzeh, H., Meacham, S. and Khan, K. (2019). A new approach to calculate resource limits with fairness in kubernetes, *2019 First International Conference on Digital Data Processing (DDP)*, London,United Kingdom, pp. 51–58.

Hamzeh, H., Meacham, S., Khan, K., Phalp, K. and Stefanidis, A. (2019). Ffmra: A fully fair multi-resource allocation algorithm in cloud environments., *2020 IEEE 4th International Workshop on Software Engineering for Smart Systems (SESS)*, Leicester, UK, pp. 279–286. CORE Ranking:B.

Hamzeh, H., Meacham, S., Virginas, B., Khan, K. and Phalp, K. (2019). Mlf-drs: A multi-level fair resource allocation algorithm in heterogeneous cloud computing systems, *2019 IEEE 4th International Conference on Computer and Communication Systems (ICCCS)*, pp. 316–321.

Huang, Y., cai, K., Zong, R. and Mao, Y. (2019). Design and implementation of an edge computing platform architecture using docker and kubernetes for machine learning, *Proceedings of the 3rd International Conference on High Performance Compilation, Computing and Communications*, HP3C '19, Association for Computing Machinery, New York, NY, USA, p. 29–32.
URL: *https://doi.org/10.1145/3318265.3318288*

Jiang, S. and Wu, J. (2018). 2-dominant resource fairness: Fairness-efficiency tradeoffs in multi-resource allocation, *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*, Orlando, FL, USA, USA, pp. 1–8. CORE Ranking:B.

Joseph, C. T. and Chandrasekaran, K. (2020). Intma: Dynamic interaction-aware resource allocation for containerized microservices in cloud environments., *Journal of Systems Architecture* **111**.
URL: *http://search.ebscohost.com/login.aspx?direct=trueAuthType=ip,cookie,shibdb=edselpAN=S1*
livescope=sitecustid=ncirlib

Kaewkasi, C. and Chuenmuneewong, K. (2017). Improvement of container scheduling for docker using ant colony optimization, *2017 9th International Conference on Knowledge and Smart Technology (KST)*, Chonburi, Thailand, Thailand, pp. 254–259. Core Rank:Not Ranked.

Khamse-Ashari, J., Lambadaris, I., Kesidis, G., Urgaonkar, B. and Zhao, Y. (2018). An efficient and fair multi-resource allocation mechanism for heterogeneous servers, *IEEE Transactions on Parallel and Distributed Systems* **29**(12): 2686–2699.

Lopez-Pires, F. and Baran, B. (2017). Cloud computing resource allocation taxonomies, *International Journal of Cloud Computing* **6**: 238.

Marathe, N., Gandhi, A. and Shah, J. M. (2019). Docker swarm and kubernetes in cloud computing environment, *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*, pp. 179–184.

NGINX, I. (0009). Nginx announces ingress controller solution for load balancing on red hat openshift container platform., *Business Wire (English)* .
URL: *http://search.ebscohost.com/login.aspx?direct=trueAuthType=ip,cookie,shibdb=bwhAN=bizwi livescope=sitecustid=ncirlib*

Rossi, F., Cardellini, V., Lo Presti, F. and Nardelli, M. (2020). Geo-distributed efficient deployment of containers with kubernetes, *Computer Communications* **159**: 161 – 174.
URL: *http://www.sciencedirect.com/science/article/pii/S0140366419317931*

Shafiee, M. and Ghaderi, J. (2020). On max-min fairness of completion times for multi-task job scheduling, *2020 IFIP Networking Conference (Networking)*, pp. 100–108.

Surya, R. Y. and Imam Kistijantoro, A. (2019). Dynamic resource allocation for distributed tensorflow training in kubernetes cluster, *2019 International Conference on Data and Software Engineering (ICoDSE)*, pp. 1–6.

Verreydt, S., Beni, E. H., Truyen, E., Lagaisse, B. and Joosen, W. (2019). Leveraging kubernetes for adaptive and cost-efficient resource management, *Proceedings of the 5th International Workshop on Container Technologies and Container Clouds*, WOC '19, Association for Computing Machinery, New York, NY, USA, p. 37–42.
URL: *https://doi.org/10.1145/3366615.3368357*

Vohra, D. (2017). Scheduling pods on nodes., *Kubernetes Management Design Patterns* p. 199.
URL: *http://search.ebscohost.com/login.aspx?direct=trueAuthType=ip,cookie,shibdb=edbAN=12213 livescope=sitecustid=ncirlib*

Wang, Q., Fu, X.-L., Dong, G.-F. and Li, T. (n.d.). Research on cloud computing task scheduling algorithm based on particle swarm optimization., *Journal of Computational Methods in Sciences Engineering* **19**(2): 327. Impact Factor:Not found.

Wang, W., Li, B. and Liang, B. (2014). Dominant resource fairness in cloud computing systems with heterogeneous servers, *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, Toronto, ON, Canada, pp. 583–591. CORE Ranking:A*.

Wei-guo, Z., Xi-lin, M. and Jin-zhong, Z. (2018). Research on kubernetes' resource scheduling scheme, *Proceedings of the 8th International Conference on Communication and Network Security*, ICCNS 2018, Association for Computing Machinery, New York, NY, USA, p. 144–148. CORE Rank:C.
URL: *https://ezproxy.ncirl.ie:2079/10.1145/3290480.3290507*

Yegulalp, S. (2019). What is kubernetes? container orchestration explained., *InfoWorld.com* .
URL: *http://search.ebscohost.com/login.aspx?direct=trueAuthType=ip,cookie,shibdb=edsggoAN=eds livescope=sitecustid=ncirlib*

Youngmi Jin and Hayashi, M. (2016). Efficiency comparison between proportional fairness and dominant resource fairness with two different type resources, *2016 Annual Conference on Information Science and Systems (CISS)*, pp. 643–648.

Zarchy, D., Hay, D. and Schapira, M. (2015). Capturing resource tradeoffs in fair multi-resource allocation, *2015 IEEE Conference on Computer Communications (INFOCOM)*, pp. 1062–1070.

Zhao, L., Du, M. and Chen, L. (2018). A new multi-resource allocation mechanism: A tradeoff between fairness and efficiency in cloud computing, *China Communications* **15**(3): 57–77. Impact Factor:1.514.