Dynamic Resources allocation using Priority Aware scheduling in Kubernetes

MSc Research Project
MSc in Cloud computing

Shelar Prasad Lahu
Student ID: X18137342

School of Computing
National College of Ireland

Supervisor: Mohammad Iqbal

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Prasad Lahu Shelar
Student ID:	X18137342
Programme:	Msc in Cloud computing
Year:	2018-2019
Module:	Research in Computing
Supervisor:	Mohammad Iqbal
Submission Due Date:	12 Dec 2019
Project Title:	Dynamic Resources allocation using Priority Aware scheduling in Kubernetes
Word Count:	7086
Page Count:	40

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.

Signature:	
Date:	11/12/2019

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):	

Dynamic Resource Allocation using Priority Aware algorithm in Kubernetes

Prasad Lahu shelar
X18137342

Abstract

In current IT infrastructure, Microservices based architecture provides the loosely coupled services for the development, deployment of the application. In a microservice architecture container-based application runs on the cloud service provider to achieve business continuity. Hundreds of container-based applications are deployed on a daily basis in the service provider. In order to manage the containers and also to scale the application, there is a need for container orchestrator and currently the leading orchestrator tools in the market are Kubernetes, Docker swarm and Apache Mesos. By default, Kubernetes schedule containers use the Bin-packing algorithm. The scheduler allocates the resources as per the availability vs demand in First fit first manner. In this research, we are trying to explore Application-aware scheduling. The mechanism involves allocating the containers based on priority with the custom python-based scheduler in Kubernetes architecture . Our results show that priority aware scheduler can allocate the resources without affecting the services as well as Kubernetes default scheduler.

1 Introduction

The cloud-native approach is a more desirable approach for deploying application in distribute environment. Virtualization plays a major role to deploy the application in cloud infrastructure. Operating system virtualization is a more suitable technique used to deploy a distributed application called as Micro servicesLarrucea et al. (2018). Small virtual instances called containers that do not require a guest operating system on the virtualization layer. This is the reason due to which container base application is widely used to run microservices on a large scale. Google and Amazon-like cloud-native infrastructure service providers use container technology to run a micro service-based application Alam et al. (2018).

Docker[1] and Linux[2] containers are extensively used to deploy microservices. The reason of popularity that it provides more tool and straight forward workflow to use application. Container technology enables GPU computing and InfiniBand which helps to build block for a High-performance computing environment. Containers provides identical solution as virtual machine in multi tenant architecture. Sharing of the resources are the major role to illustrate the virtual machine and container base deployment. In virtual machine base deployment resources are shared based on Virtual machines and container

[1]https://docs.docker.com/get-started/
[2]https://www.redhat.com/en/topics/containers/whats-a-linux-container

base technology resource sharing is carried out by sharing Operating system. In high performance computing container base application increased utilization rate as well as have less transfer delay Li and Kanso (2015).

Currently, Rapid integration of container base application in IOT, web services enables the need of effective organization and management of containers in large scale clusterRamalho and Neto (2016). In container base technology docker has defacto standard to run microservices. Container orchestrator software required to manage the container in large scale cluster. Kubernetes, Docker swarm and and apache Mesos are leading open source container orchestrator software available for container provisioning, resource allocation and configuration purpose. Effective container scheduling and resource optimization is the necessity in cloud service provider Netto et al. (2017a).

Bin packing algorithm is used to place the pods in container orchestrator tools like Kubernetes and Docker Swarm. Application-aware scheduling, autoscaling and rescheduling and cost-effective scheduling are the main key area of research in container orchestration which was put forth by Rajkumar Bhuya Rodriguez and Buyya (2018).

1.1 Research objective

Application base scheduling approach is used to manage load in data center. Critically of an application and bandwidth are the main factor on which application aware scheduling can be used to allocate the resource or pod placement in container orchestration. Priority aware scheduling and Network bandwidth aware algorithm is used in cloud environments to reduce response time and increase resource optimization in container base application. In this research we propose Priority aware and Network bandwidth algorithm in Software define networking environment to allocate the resources in Virtual cluster using Container orchestrator tool like Kubernetes. Hence the our research question -

Can dynamic resource allocation of kubernetes be improve the performance of system and Quality of service via priority aware algorithm ?

Priority aware algorithm places the POD according as per the predefined application criticality and Network resources allocated for an application.

2 Related Work

This section we can have given brief information of the section a) Microservices implementation using containers b) container orchestration Software or platform c) Importance of application aware scheduling.

2.1 Containers virtualization in Micro-services

Containerization or container base virtualization enables the isolation for an application rather than hardware emulation. It allowed to the user for the creation of multiple user spaces over the same operating system kernel level. Docker has the defacto standards in container virtualization because it facilitated the application for auto-creation, deployment, and execution. Docker host is a lightweight host which requires the minimum resources as compared to the actual virtual machine Naik (2017). As shown in Fig 1. Each container has abstraction of libraries of an application for an executing in isolated environment. This isolation layer can be achieved through the Linux features namespaces and groups. Name spaces allowed the container for restricting resources allocated to it.

When new containers are initiated that allowed user to create the **system calls** which helps to create abstraction from an existing namespace. Cgroups is used to identify the resources. Docker supports the Linux features Cgroups and namespaces, but it has additional feature called Advance Multi-layered Unification system (AUFS) for management of containers. It provides the capability to docker produce multiple container from the single base image. Updation of image can be easily tracked in docker Jha et al. (2019).

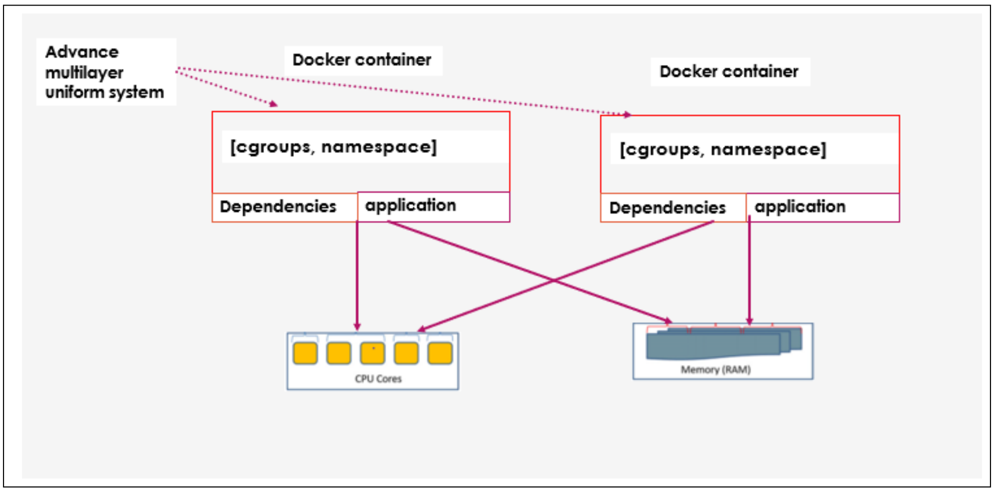


Figure 1: Docker container architecture in microservices environment

2.2 Container Orchestration platform

Containers are formed by static images which are stateless in nature. Containers are terminated when they are shut down in the system. Due to this nature some orchestrating tool was required for optimizing and effective use of containers in large scale deployment. Google used the containerization of application from last 10 years to run application over internet. Kubernetes was developed by Borg to manage the containers in large scale environment Schwarzkopf et al. (2013). Cloud native computing foundation improve the Kubernetes features so it can be easily accepting in cloud service provider.

Creating standards in Kubernetes technology is the objective of this foundation. In 2014, Kubernetes technology has rapid development from open source communities such as Red hat and VMware etc. Kubernetes enables the feature like self-healing capability which can remove unwanted or unused containers from virtual cluster. Resources are optimized due to self-healing capability. Scheduling in Kubernetes are categorized into two different stages i.e. predicting and prioritize. Adequate resources are decided according the ports and disk are used and containers are placed according to the fit and specification of Containers. Adaptive scheduling approached are taken by Open source community to improve performance of containers Netto et al. (2017b).

Docker swarm is the proprietary scheduling tool for Docker containers. It schedules the containers according to the First in First out manner. Docker swarm allocate CPU cores and memory resources according to different strategies like Spread strategy, Bin pack strategy and Random strategy . Bin pack strategy deals with container according to fitting mechanism. If container is fitted on the node then it has been placed to scheduled node. Spread strategy can be carried out according the least no of containers in node. Random strategy is used to allocate the CPU random manner. Docker swarm supports the filtering mechanism which can allocate the container according to the container affinity and priority which has the significance role in Enterprise cloud services Cérin et al. (2018).

Apache Mesos scheduler has the centralize scheduling mechanism which can be take input from the frameworks, resource availability and policies defined for clusters. After considering all this factor apache Mesos calculate the resources for all tasks. Apache Mesos have complex scheduling mechanism which has negative impact on scalability and resilience. It has abstraction called as Resource offer which implied that resource is encapsulated with the number of resources offer to each framework. Apache Mesos supports task base scheduling with respect to host on which task is executed. Apache Mesos provide the functionality like apache Hadoop, apache spark, with distributed cluster for e.g. Dkron and Chronos. Hindman et al. (n.d.)

YARN is two level schedulers in which scheduler assign the job on the base of per job bases. It sends the requests according to the resource masters. Resource master has the role to check the task and assign the task to appropriate virtual machine. Application masters have responsibility to allocate the services rather than scheduling. YARN scheduler has the functionality to allocate resources at a time, but it allows to user for accessing the multiple API's. Application level scheduling can be taken easily from YARN scheduler due to availability of multi functionality of APIs.Rodriguez and Buyya (2019) Kubernetes has the functionality to allow multiple APIs and have feasibility to work on auto scale mode. Scale in and scale out features allow to users for deployment of services in virtual cluster. It has multi containers features on which scheduler can schedule containers like Docker, Linux, rocket. Third party support-ability is the main reason due to which Kubernetes is popular in open source community.Table 1 shows the comparison between kuberntes scheduler with Docker swarm and Apache mesos.

The application of Kubernetes platform is used for the fog computing in network service provider. Kubernetes has the capability to take decision on the current state of the system . In network infrastructure ,number of nodes are increases rapidly . By using Kubernetes platform Node connectivity and node affinity is achieved Santos et al. (2019).

2.3 Scheduling Mechanism with related work:-

Container orchestration system is based on the Google Borg system which is used to deploy thousands of applications on the virtual cluster in Data center. It is based on the task base scheduling with process level performance. It has the capability to provide support in run time environment. Borg enable the user to provide design decision and quantitative examination of operational experience Verma et al. (2015).

Scalability is the concern in container base scheduling. Resource allocation for on demand requests raise by consumer can be achieve through the effective use of Resource manager. Xu apply the game theoretic method to use of effective binding between physical resources with container resource need. Xu et al. (2015)

Resource demand is varying according to cloud consumer to the consumer in cloud infrastructure. Non-uniform distribution resources increase the complexity of the distri-

Table 1: Comparing schedulers of Kubernetes,Docker swarm, Apache mesos , YARN Truyen et al. (2018)

Characteristics	**kubernetes Scheduler**	**Docker swarm scheduler**	**Apache mesos Scehduler**	**YARN Scheduler**
Container supportability	Docker,rkt , CRI , API implemnetation , OCI-compliant runtimes	Docker	mesos containers and Docker	Linux group based , Docker
Application deployment model	workload supportaility	Long time job and task base scheduling	Schedules per task per time and colocated task	monolithic for task scheduling and bach process scheduling
Scheduler architecture	Distributed and monolithic	Distributed and monolithic	two level but offer base schdeuling	two level but offer base schdeuling
cluster Elasticity and scalability	Elastic but manual supports autoscaling functionality	Elastic but manual supports autoscaling functionality	Elastic but manual supports autoscaling functionality	Elastic but manual supports autoscaling functionality
Supports Hypervisor isolation	yes	no	no	no
Port mapping and IP per port features	Scheduler can not map port with IP	Scheduler can not map port with IP	Port mapping with Ips are supports	Does not supports Port mapping
Computer perfomance isolation and Resource Quota per user bases	Yes	No	Yes	yes

bution of resources. For achieving, optimization of resources in a heterogeneous cloud environment, Kawase proposes the ant colony algorithm with Docker swarm. Resources distribution is depending upon the probability distribution function. Some group of containers, it allocated the resources according to the random manner and the rest of the container allocation is based on the ant behavior Kaewkasi and Chuenmuneewong (2017).

Container automation and management have issues while implementation in large scale environment. Container automation and management have issues while deployment in a large-scale environment. Guerrero highlighted that Container automation proposed the genetic algorithm . This algorithm helps in achieving effective system provisioning, optimization, and failure detection.Apart from Resource allocation, cost reduction is a major challenge in the cloud service provider. Brownout proposes that if microservices are not running then some of the components need to be shut which can effectively save resource consumption of an application. Brownout proposes the model which is responsible for fine-grained control in containers. Several scheduling policies had been applied to the containers. Docker swarm is used to implement this model which has functionality to activate and deactivate the microservices Tao et al. (2017). For Dynamic resource allocation, Xuedong propose the node selection logic. In this node selection logy fuzzy logic implemented for prediction of node where containers can be deployed. The objective of this research was the optimum use of resource configuration and improves the performance of the cluster Xu et al. (2019).Kubernetes scheduling architecture is used in CNC system which is advance processing system in manufacturing industry. By using scheduler strategy CNC various task are distributed across the kubernetes component Jin et al. (2019).Weaver architecture is propose the SQL queries raised in the cluster. Monitoring of this queries with the help of kubernetes propose by Lalit from Vmware.Suresh et al. (2019)

Elastic provisioning of virtual machines for container deployment which has taken heterogeneous configuration into account. Optimization of QOS is the main concern of this deployment. Elastic provisioning is illustrated with the help of linear programming model Nardelli et al. (2017). In 2018, Rajkumar Bhuyya proposed priority aware scheduling algorithm in Software defined networking to allocate the virtual machine in Data center. This algorithm based on the placement of the application criticality which helps to increase the QOS and resources optimization of the system . Closest proximity of virtual machines is the main key aspect of experimental setup Son and Buyya (2018).

3 Methodology

3.1 Architecture Design

We have proposed the priority aware scheduling mechanism with using Kubernetes existing default scheduling mechanism. Architecture proposed to implement Priority aware algorithm required one master with multiple slaves configuration. Kubernetes K8s scalar platform described as shown in figure 2. Proposed architecture is divided into the different sections.Section 1 included a brief overview of the Kubernetes master component whether section 2 has primarily focused on slave node design. Apart from master-slave architecture, we have explained open source monitoring tools like Prometheus and Grafana.

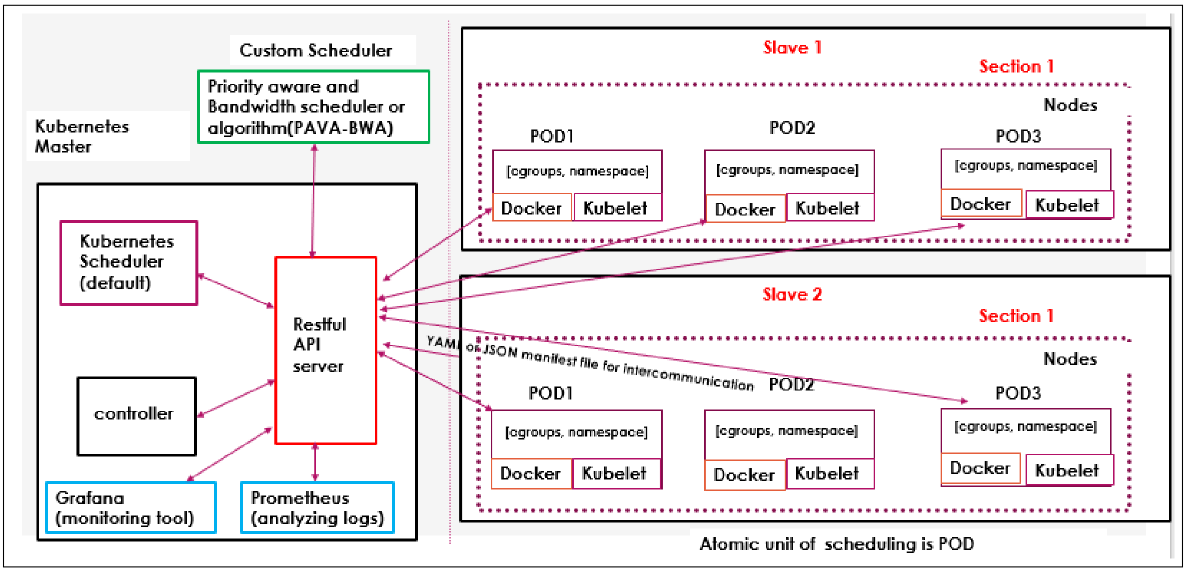


Figure 2: Priority aware Scheduling mechanism with the Kubernetes Platform

Kubernetes Master[3]:-

In section 1, we are discussing about Kubernetes master configuration which has major role to allocate the Pod in Kubernetes cluster and it is centralize unit through which all components are monitored and allocated the task. Kubernetes master consist of API server, Controller and Kubernetes default scheduler. For deployment of Kubernetes master and slave we must allocate the dedicated virtual machines with IP reach ability.The functions of components which are used in our deployment. Kubernetes master component are etcd, default scheduler, API server and control managerChang et al. (2017)

- **Etcd**:– Etcd is used to store the stage of the system. Pod status and logs are the stored in etcd system. After watching etcd state scheduler takes the action for execution of Task.
- **Default scheduler**:Kubernetes default scheduler is used to allocate the pod with worker node or slave. According to CPU and memory score Kubernetes allocate the pods.
- **Controller Manager**: The controller manager has the responsibility to analyses the components of the system. If the state of the system changes then the controller manager forced to Kubernetes component for maintaining the desired state.

In Section 2 Besides of the Master node, Kubernetes has the slave nodes which can be run other virtual machines. Pods and services are deployed on worker node. Kubelet and POD are the component of Kubernetes worker node which is described as below.

- **POD**: - Pod is a basic atomic element in Kubernetes which has allocated one or multiple containers. Each Pod must assign a significance IP address using the flannel network. In our case we are using flannel network with IP pool of 40.168.0.0/16..
- **Kubelet**:- Kubelet is the node agent that is installed on each worker nodes. It has the responsibility to monitor the POD specification through the masters and slaves. Resource utilization, pod status, and node events are highlighted by Kubelet.it exposes the information on port number 10255.
- **Kubernetes namespaces**: - Process isolation is important factor while deployment service. In kubernetes cluster,Name spaces has the responsibility to isolate the pods or containers. Pods in same namespaces can be communicate to each others.
- **Replication controller**: - Replication controlled is used to control number of replicas of pod in kubernetes cluster, Minimum number of replicas we have set to be 1.

Monitoring Module:- we are using the below three software for implementation and analysis of nodes and pods statistics. This three modules are grafana , Prometheus and Kubernetes dashboard services.

- **Grafana**[4] is pluggable data source model which has the supportability of time series database like Graphite and Prometheus and OpenTSBDB. It has monitoring support with cloud service provider like Google and Amazon.

[3]https://kubernetes.io/docs/concepts/overview/components/

[4]https://grafana.com/

- **Prometheus**[5] is open source monitoring system and having alter managing system. It scraps and stores the time series data. It pulls information over http and target discovery can be take place using service discovery and Static configuration.

- **Kubernetes dashboard**[6] is used to identify the cluster as well as allocate the resources and Pods creation and deletion. It is a web interface through which we can modify and troubleshoot the containerized application. It also provides the logs of the existing nodes like CPU, memory .

3.2 Priority aware Algorithm implementation using Kubernetes orchestrator

In cloud service provides, Placement of container can be taking place according Bin packing algorithm. Bin pack algorithm fits the application according to the resource availability at nodes. This criteria of deployment of application may be unsuitable to deploy the critical application. The bin pack algorithm first fits on the first bases. Priority Aware application state that we must consider the priority into account to deploy the services. This algorithm proposed that if an application with the closest proximity improve the user experience in SDN environment. In our research work, we implement this priority aware algorithm to the placement of POD using the help of Kubernetes default scheduling and policy base schedule Son and Buyya (2018). Application with the highest priority needs to be scheduled first in the queue. Key highlights of this algorithm are below.

- The algorithm states that the nodes which have the closest proximity can be preferred to the allocation of application to avoid network congestion. Closest proximity can be defined using the states of the workers in the system.

- The highest priority of an application carrying pod should be the first scheduler and create a group of scheduled pods with the priority according to group or policy.

- If we place the application with the same group, then we can assign it to the same nodes. We must consider the current resource consumption or availability of memory and CPU to allocate the Pod carrying application.

In Figure 3 represents the pseudo code of priority aware algorithm which shows the application aware scheduling. Application running on container can assign the priority and this priority is carried out through the sorting mechanisms.Queue group is created to align the container on the basis of priority. If priority of application is high then the containers needs to be place on the first come first bases.Nodes proximity are main objective of Priority Aware scheduling.

[5] https://prometheus.io/docs/introduction/overview/
[6] https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/

Algorithm 1: Priority Aware container allocation algorithm

```
Input: Con : container to be placed
Input rd: resource demand of container
Input app: application information of container
Input host: list of hosts in virtual cluster.
Host-group :List of host for placement
Qh = Queue for available hosts in nodes
if app is a higher-priority application then
    Hostapp - List of host connected for other containers in app ;
    if Hostapp is not an empty then
        Qh.enqueue(Hostapp)
        for each host in hostapp do
        Host-edge group where host is included
        Qh.enqueues(Host-edge);
        end for
else if Sor Host-group with available capacity in higher to lower order then
    Qh.enqueues(Host-group);
    while Qh is not empty and placed=ffalse do
        hostq = Qh.dequeue();
        Capacity-host ———- (free resources available in host' Queue)
        if rd < capacity-Host then
            Placement of container will be done ;
            Capacity-host ← rd;
            placement=true
        else

        end
        end while
    end
else
    Kubernetes default scheduling mechanism will work
end
```

Figure 3: Psuedo code for Priority Aware algorithm

4 Design Specification

In our experimental setup we are used the oracle virtual box 6.0 to create the virtual machine with Ubuntu operating system 18.0.4. Master and slave configuration required for Kubernetes this is achieved through the interconnection of Virtual machine. We are created additional interface ***enp0s8*** which is used for control message and API interaction between master and slave.Kubernetes uses ***kubectl version v1.16.3*** as command line interface for checking ,modification and creation of pods, deployment, and services. we are allocated 4GB RAM and minimum two virtual cpu allocated for each node in cluster. Kubernetes required 2 CPU with 2GB RAM minimum for running POD and services. For experimental setup design prospective the list of component with specification is highlighted in Table 3.

Whenever virtual machines will be evoked it shows the error of localhost or port is not defined. For resolving this error we have to off the partion using **swappoff -a** command or we can do permanent in partion of linux library /etc/fstab. Proxy setting of the ubutu machines need to be check. Kubernetes by default proxy works on the port

Table 2: System configuration for Kubernetes Cluster

Deployment essentials	Description
Virtualization software for VM creation	Oracle Virtual Box 6.0
Container orchestrator software	Kubernetes 1.16.0
container software	Docker 19.0.3
Application container	Ngnix, Redis
operating system	Ubuntu 18.04
Process configuration	AMD ryzen 5 with 2.0 GHZ
No of vcpu reuired for master and slave	7 vcpu
RAM	16 GB minimum
Hardisk	1 TB HDD
Custom scheduler code language	Python 2.7 used
Manifiest language for intercommunication	YAML

number 8001.We have to evoke this proxy using command kubectl proxy –port=8001.Inter pod communication can be form using flannel network. We have defined the ip pool 40.168.0.0/32 for flannel network as shown in figure 4.

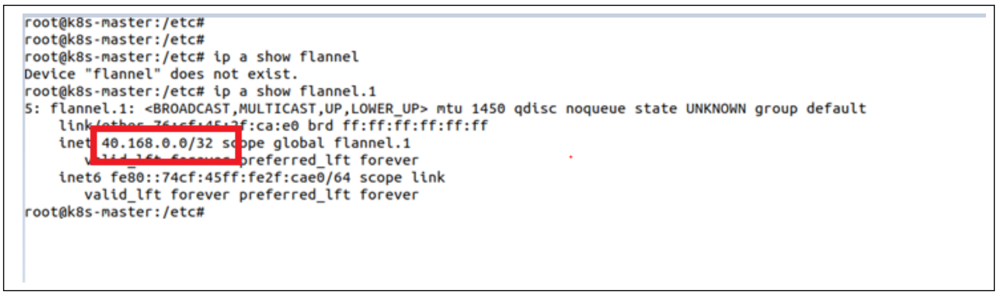


Figure 4: Flannel Network for Pod communication

5 Implementation

5.1 Kubernetes Master and Slave Formation

Docker's 19.03 version had been installed before running any command of the Kubernetes cluster. we have to enable the docker version on both master and slave nodes. Kubernetes master and slave formation can be achieved through installation of Kubernetes administrator package as ***kubeadm* v1.16.3 in nodes**. ***kubeadm*** has the responsibility to consider all preflight check which consists of partitioning disabled or not, docker version and enabled or not. It also creates the *kubeadm* join request which initiate the slave to join master. As shown in figure 5, Kubectl get nodes command show the availability of nodes in cluster and slave with there versions.

```
root@k8s-master:/etc/kubernetes#
root@k8s-master:/etc/kubernetes#
root@k8s-master:/etc/kubernetes# kubectl get nodes
NAME         STATUS   ROLES    AGE     VERSION
k8s-master   Ready    master   7h47m   v1.16.3
slave-1      Ready    <none>   7h23m   v1.16.2
slave-2      Ready    <none>   7h18m   v1.16.3
root@k8s-master:/etc/kubernetes#
```

Figure 5: Master and slave configuration

5.2 Priority Aware Custom scheduler Implementation

For implementation of priority aware scheduling, we have to create the PAVA.py file. Kubernetes default scheduler code is written in go language. We are trying to implement the python base custom scheduler which can interact with Kubernetes API for getting live statistics and perform the sort pod using priority bases. We have followed step by step approach as shown in figure 6 to run redis and container application on kubernetes platform.

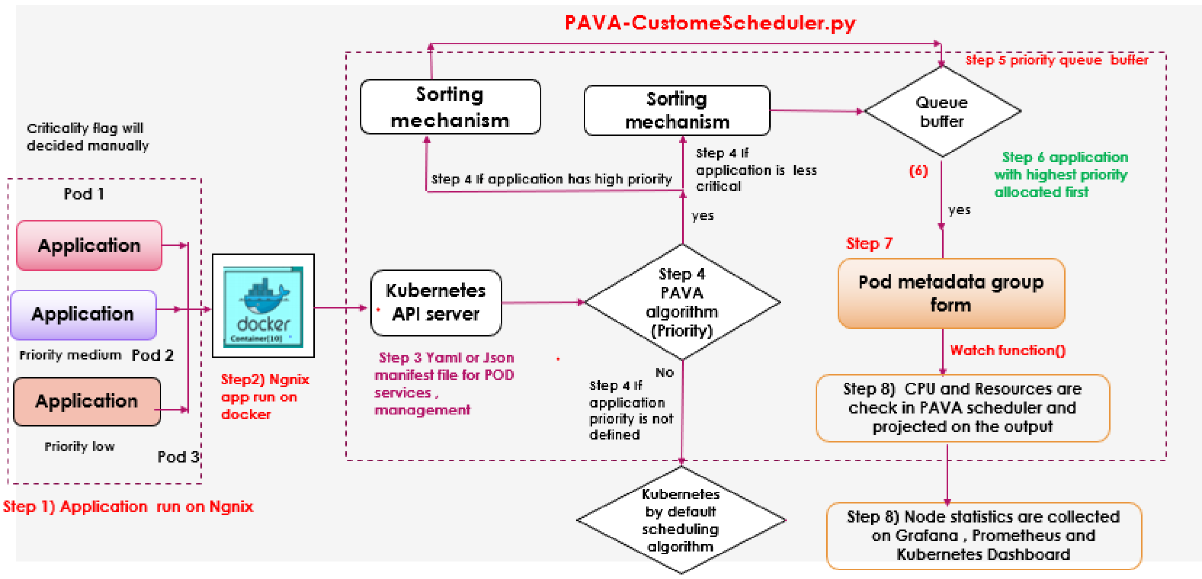
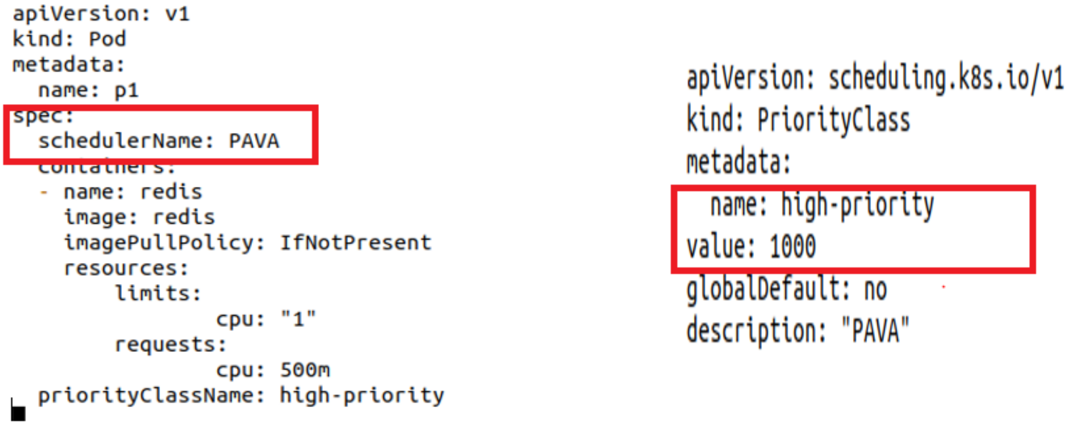


Figure 6: Implementation of Priority aware algorithm in Kubernetes

Step 1:- Pods has been created using the ***pod.yaml*** files as shown in figure 7 on which redis or ngnix containers are running.

Step 2:-We are created five yaml files. Pod 1 ,2 and 3 are defined with priority as high medium and low whether Pod 4 and 5 are without priority field .As shown in Figure 7. *pod. Yaml* , we are added the Scheduler name field as Priority aware (PAVA). This fields is used to identifier for scheduling.If scheduler name is not defined then Kubernetes schedule the pod using default scheduling mechanism.Virtual CPU is set 1 in redis application. In request filed we have to set ram information for running application.



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: p1
spec:
  schedulerName: PAVA
  containers:
  - name: redis
    image: redis
    imagePullPolicy: IfNotPresent
    resources:
        limits:
                cpu: "1"
        requests:
                cpu: 500m
  priorityClassName: high-priority
```

```yaml
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority
value: 1000
globalDefault: no
description: "PAVA"
```

Figure 7: *Pod.yaml* and *pod.class* file configuration for deployment of redis application

Policy map functionality we have to defined in separate yaml file. This class. Yaml files helps to mapping the priority with respective pods. There are 5 pods. Yaml file for creation of pods and 3 policy map files for define the priority. We have to install Kubernetes python libraries for intercommunication between python client to api server. Pip install Kubernetes command is used for importing Kubernetes package in python. We are using config, watch and client from Kubernetes library for implementation purpose. In pod. Yaml we are adding the **Scheduler name** field as Priority aware **PAVA**. This fields is used to identified the scheduler name. If scheduler name is not defined then kubernetes default scheduler starts deploying pods on slaves.

Step 3 :- Policy map functionality we have to defined in separate yaml file. This class. Yaml files helps to mapping the priority with respective pods. There are 5 pods. Yaml file for creation of pods and 3 policy map files for define the priority. We have to install Kubernetes python libraries for intercommunication between python client to api server. Pip install Kubernetes command is used for importing Kubernetes package in python. We are using **config, watch and client** library from Kubernetes library for implementation purpose. We are uses V1 api version and pod as kind in yaml file. Config load kube config is the main function of Kubernetes, which helps for running custom scheduler with Kubernetes components.

Figure 8 denotes the state diagram of pod binding process,In this First step is pods creation, which can be responsible to carry ngnix or Redis container for deployment of services. kubectl command sends the request to Kubernetes api server at 192.168.56.101 to start the pods in yaml. This state is saved in storage element of Kubernetes that's is etcd. There are three states of pods which are pending, running and evicted. Pending state denotes pods are not ready to deploy. Running state denotes pods are ready to run the services. Running is the ideal state of the pod. Evicted state denotes that's pods are not running due to resource availability, API issue or else Medel et al. (2018).

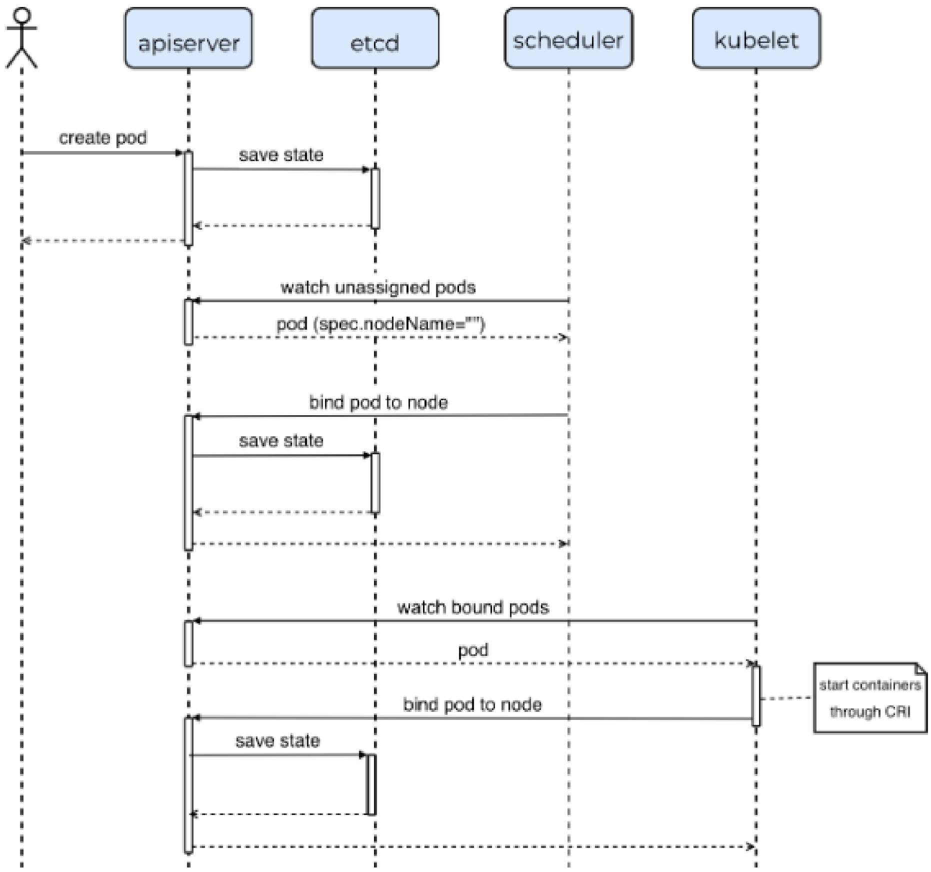


Figure 8: State diagram of pod creation

Step 4 :- We are using sorting functionality in PAVA scheduler fields with priority value 100, 500 .1000 in pod1 and pod2 and pod3 respectively.The input from **client.CoreAPi()** of kubernetes give brief information of the namespace ,metadata and scheduler name, priority etc. On the basis of this information empty array of scheduling priority and group pod is created .

```
################################## priority aware algorithm works ##########################

print("\n \n \n \n scheduling pod details are as below \n \n  ")

for i in ret.items:
    Scheduling_priority =[]
    if i.spec.scheduler_name=='PAVA':
        print("%s\t%s\t%s\t%s\t%s\t%s\t%s" % (i.status.pod_ip, i.metadata.namespace,
         i.metadata.name,i.spec.priority,i.spec.scheduler_name,i.status.phase,
         i.spec.priority_class_name))
        group_priority.append(i.spec.priority)
        group_name.append(i.metadata.name)
        group_pod=dict(zip(group_name,group_priority))
        Sorted_poddict=sorted(group_pod.items(),key=lambda kv:(kv[1],kv[0]),reverse=True)

print ("\n \n  pods details captured in dictionary  before  sorting  %s \n \n " %group_pod)
```

Figure 9: Priority base sorting function

Step 5:- As shown in figure 9, PAVA scheduler uses zip function for combining group name and group priority. This function help to stored the pods in the form of Queue buffer group_pod. Sorting function generally use lowest to highest priority, so we have to set **reverse flag** value as **true** in sorted dictionary variable.In the **watch** function, watch method checked the readiness of the slaves.

step 6:- After sorting function and slave node readiness, Binding function is evoke to bind slave with pods using target and Body parameter. Target significance to slave information whether the body denotes the pod's information which includes metadata, Api version etc..

Step 7) Important parameter throughout this process is namespace. The namespace is the unique process ID that can be used for binding function. We are setting the value of namespace as "default". There is an internal namespace called Kube-system which can be used for the communication between a component of the system. We cannot use this namespace for our scheduling and processing of the pods. V1 create binding function is used for attachment of pods. Binding function is in python in development sometimes its response does not attach to Kubernetes binding parameter. This is the reason which causes that error comes from the system is Target not found. Alternative solution for this problem is delete pods and create pods with scheduler name as default. we have to disable SSL setting in kubernetes for binding of the pods in node.Below are the setting needs to be add in library.

config = client. Configuration()
config. verify_ssl=False

Binding of the pod can be take place using **v.client_name spaced_binding()** function in which we have defined the body of the pods which includes the **apiVersion** field as **V1**,**metadata name** as pod name. Target is defined whether we have to send our pods .we can customized the pod by setting target value as slave name.. In target field we have to defined kind value as node then only response of binding function given to the API server.

Step 8:-We have calculated available CPU score and memory score and store using function *compute_ allocated_ resources* function.

step 9:-By using Kubernetes dashboard services, we are analysing the node statistics, pods status and services deployed on the virtual cluster. CPU statistics and Network traffic information is a more detail manner in Prometheus and Grafana. We are using node exporter services for nodes stats and HTTP requests running on the node. Kubernetes dashboard give brief information of pods status, phase and CPU, memory statistics. We deployed . **Prometheus version-2.13.1 Linux amd** and **Grafana 6.4.4 version** are deployed in the Kubernetes cluster for graphical representation of logs collected by node exporter

.

6 Evaluation

Experimental setup To demonstrate the potential capabilities of priority aware scheduling, we have performed experiments with the Kubernetes framework. In this experimental setup, we have deployed the Redis application on the container and assign the priority by applying the policy-map respective pods as described before in implementation. Evaluation can be carried out with a maximum number of 5 pods. The priority of pods is defined as 1000,500 and 100 respectively. we uses grafana for Graphical representation of CPU and memory statistics received from Prometheus server. We have performed 3 cases for evaluation priority aware algorithm as follow.

6.1 Case 1:- Pods without priority with kubernetes scheduler

In the first experiment, we have not assigned any priority to scheduled or nonscheduled pods. Nodes' choice is also random. When default scheduler creates pods using *kubectl* command pods runs with the redis application.The status of pods as shown in figure 10. In this scenario pods are deployed in random manner.

```
pod/pod5 created
root@k8s-master:/etc# kubectl get pods
NAME    READY    STATUS     RESTARTS    AGE
pod1    1/1      Running    0           107s
pod2    1/1      Running    0           104s
pod3    1/1      Running    0           101s
pod4    1/1      Running    0           81s
pod5    1/1      Running    0           78s
```

Figure 10: Pod status while default scheduling works

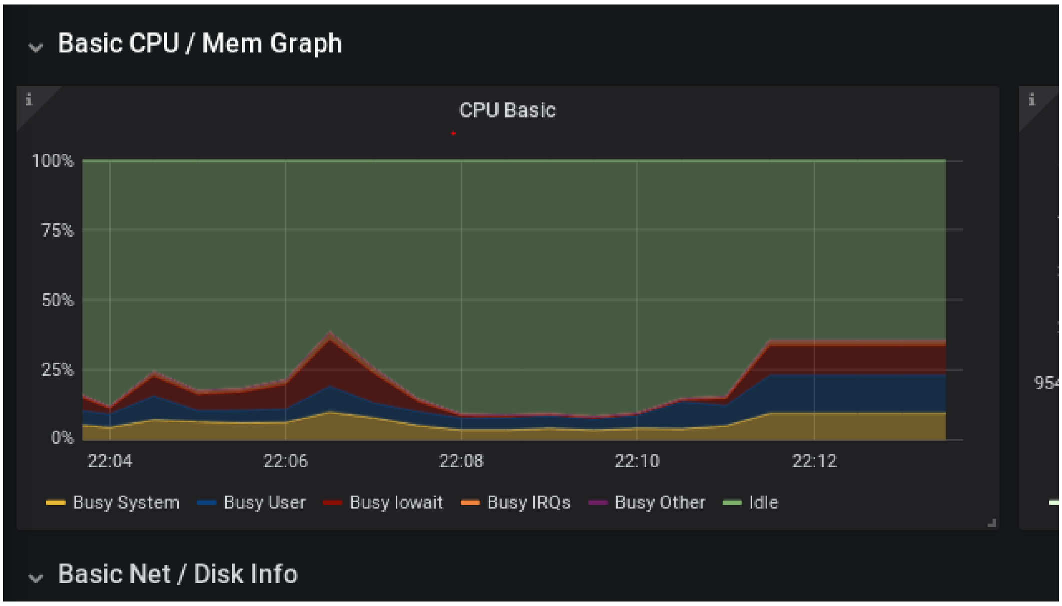


Figure 11: Grafana status of CPU performance

As shown in figure 11, we have observed that system CPU utilization is vary between 25 to 50 range when default scheduling works.After 5 minute of interval system become stable at 39 percentage.we are concern on the CPU statistics of master as compared to slave.

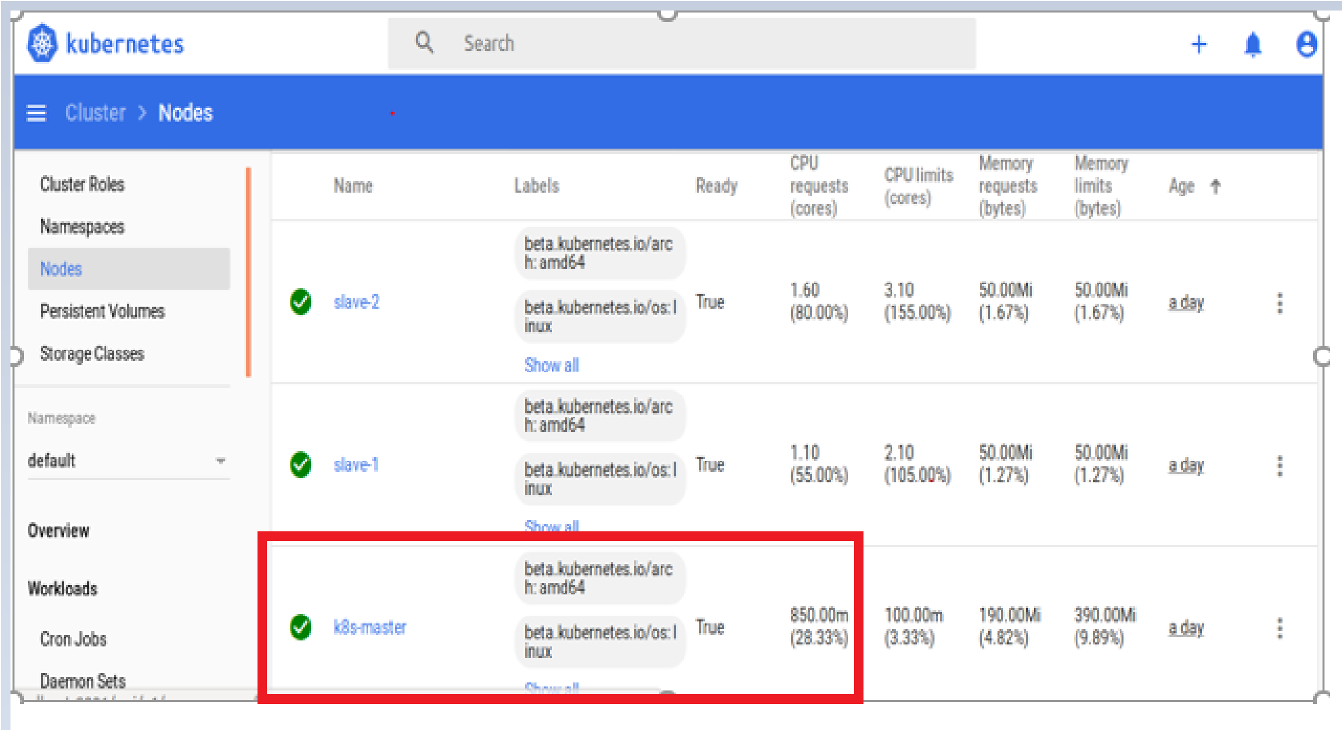


Figure 12: Kubernetes dashboard shows CPU performance

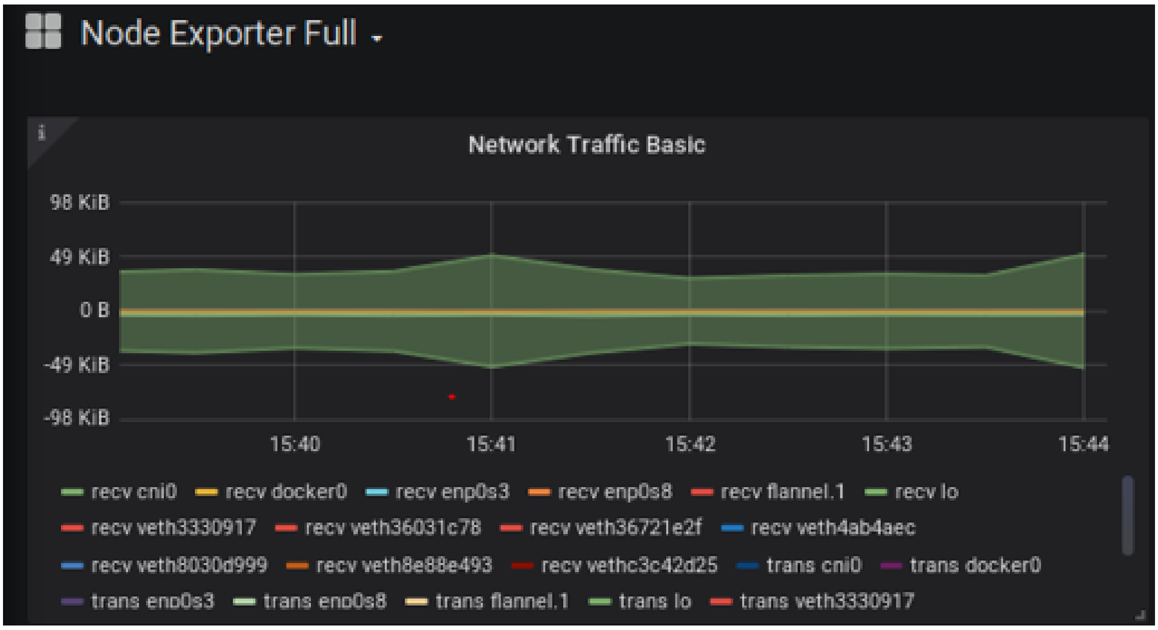


Figure 13: Netstat for default scheduler

Figure 13 showes that Netstat interface running on the masters. The graph represent that CNI interface traffic. CNI interface is used for kubectl traffic. The receive traffic from slaves towards master is 49kilobytes in default scheduling.

6.2 Case 2: Priority base scheduling with default kubernetes scheduler

In second experiment we have configure pod 1,2 and 3 are 1000,500 and 100 respectively. This experiments statistics are taken on the bases of Default scheduling to check whether the changes in performance occurred or not. As shown in figure 14 , CPU performance is vary between 25 to 50 range.In kubernetes dashboard showes the same 28 percent CPU utilization.

Figure 14: Priority with default scheduler

When we apply the priority on the pods there is a rise in receiving traffic of Kubernetes CNI interface as shown in figure 15. This rise in Kubernetes traffic has increased the CPU spike in CPU/memory inputs as shown in the figure. CNI interface receiving traffic is almost double after priority assign to pods. Overall CPU utilization for the master is the same but on the slave, CPU utilization is increased due to the placement of pods.

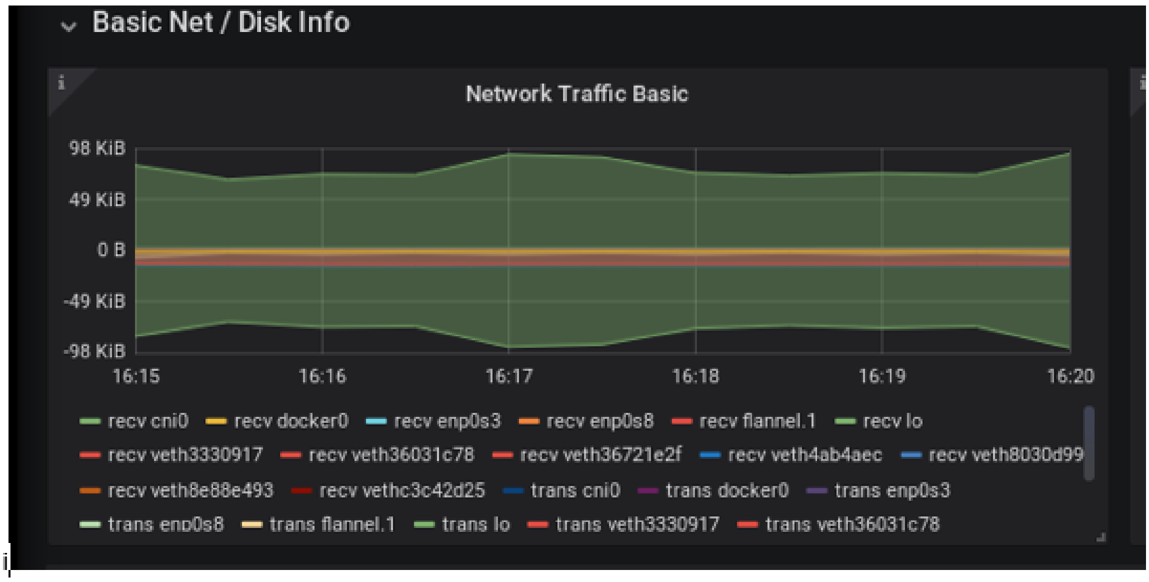
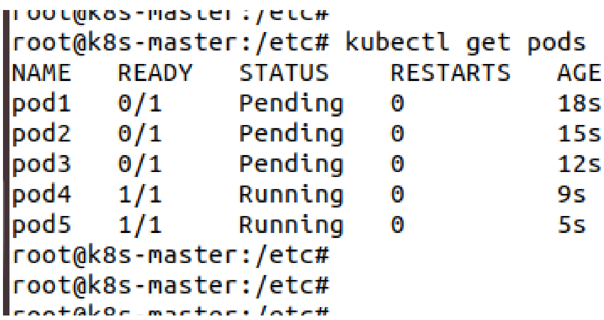


Figure 15: Priority with default scheduler

6.3 Case 3 :- Priority aware scheduling with custom scheduler

In this experiment setup, We have set the priority of the pods pod1=1000, Pod2=500, Pod3=100, and scheduler name as "PAVA" in *pod.yaml* (Priority aware). Pod3 and pod4 are set zero with default scheduling mechanism.The time interval is 15 min to take logs of the system. our pods are initiate container and pod status goes to the pending state as shown in figure . Kubernetes default scheduler unscheduled this pod1, pod2, and pod3. Kubectl describes pods information to give brief information on Kubelet. Kubelet give container status of information on the bases of binding functionality of pods.



```
root@k8s-master:/etc# kubectl get pods
NAME    READY   STATUS    RESTARTS   AGE
pod1    0/1     Pending   0          18s
pod2    0/1     Pending   0          15s
pod3    0/1     Pending   0          12s
pod4    1/1     Running   0          9s
pod5    1/1     Running   0          5s
root@k8s-master:/etc#
root@k8s-master:/etc#
```

Figure 16: Priority with default scheduler

After running Priority Aware scheduler pods changes its state from pending to running state. CPU spikes while running due to PAVA schedule. Python process of API calling and binding requires some additional resources due to that CPU utilization increases as shown in figure 17.

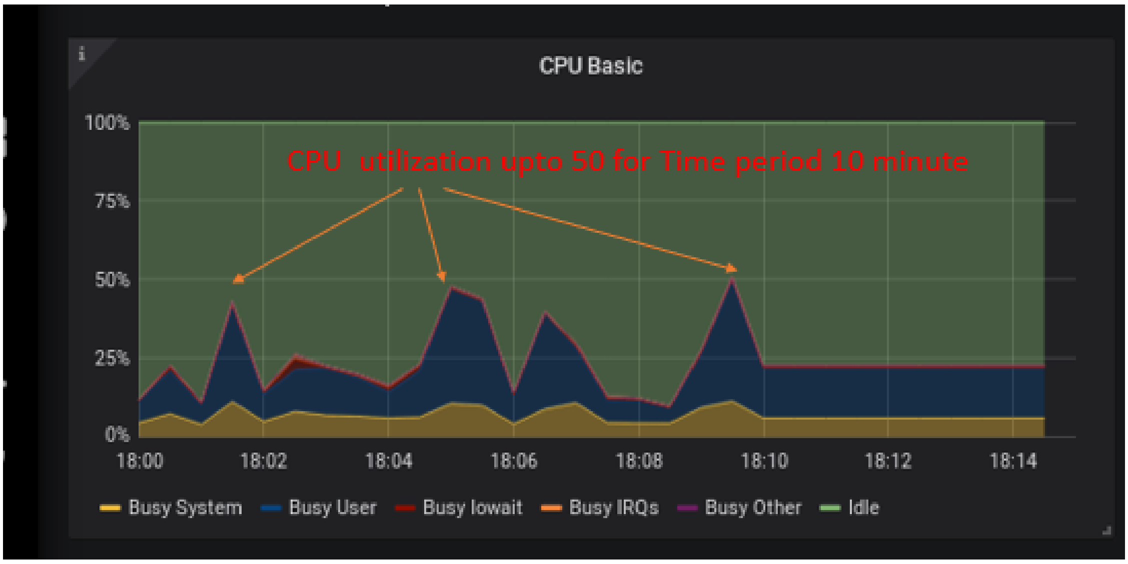


Figure 17

After some 10 minute of interval process is stable. Kubelet sends information through of status so there is additional traffic is shown in netstat graph. In figure 18 CNi interface receive traffic is above 100 Kilobytes for 5 minutes of interval.

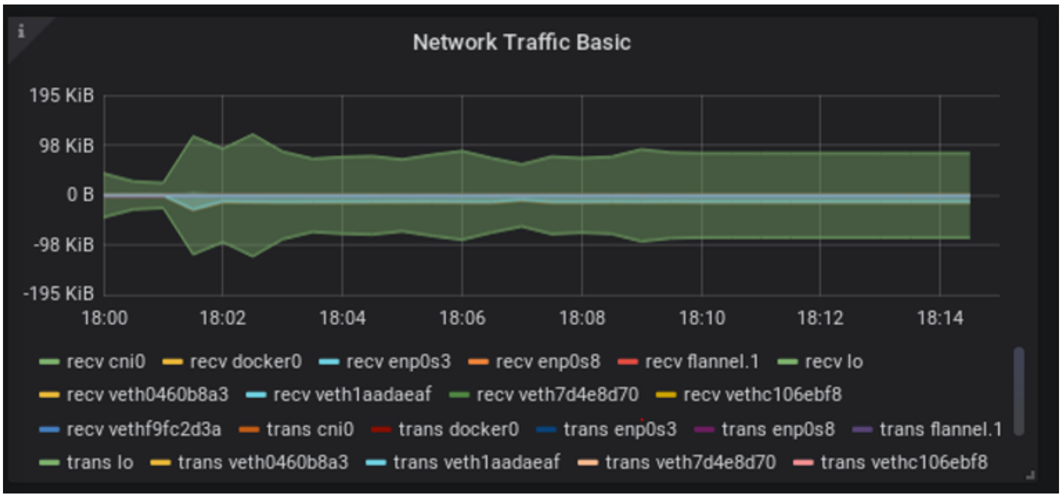


Figure 18: Priority with default scheduler

Kubectl give status of slave and update to master. The events happen which can be analyze by command line interface **kubectl** as shown in figure 19. Its check the status of container deployment on the node and according to that container creation steps will be initiated.

```
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type    Reason   Age   From               Message
  ----    ------   ----  ----               -------
  Normal  Pulled   115s  kubelet, slave-1   Container image "redis" already present on machine
  Normal  Created  114s  kubelet, slave-1   Created container redis
  Normal  Started  114s  kubelet, slave-1   Started container redis
root@k8s-master:/etc#
```

Figure 19: Events for deploying pods in Priority aware algorithm

6.4 Results and Discussion

Figure 20 showes the overview of results on the bases of experiment. Experiment 1 gives statistics of the threshold value of CPU utilization and network traffic in the CNi0 interface while the default scheduling is performed. This experiment gives the minimum threshold value which useful to determine whether our custom scheduler is over-processed or not. Default mechanism is bin packing algorithm which enables the pods in first come first bases and randomly assign. CPU utilization is 28 percent according Kubernetes dashboard services in default mechanism. Second experiment is used for scheduler pods with Kubernetes priority mechanism and this experiment cni0 interface through which control traffic received become doubled .CPU utilization is still constant throughout the process. We have not targeted any particular node for deploying the nodes.

In third experiment we have set the priority and scheduled the pods using the PAVA scheduler. This scheduling is achieved through the scheduler name as PAVA in yaml file. Pod 1,2 and 3 are scheduled by PAVA custom scheduler whether pods 4 and pods 5 are scheduled using default one. In this state pods become in pending state CPU utilization is increases due to additional process working with scheduler. The rise in CPU is upto 50 percent but within 10 minutes it stabilizes as per the default scheduler. Pending state

Cases	Priority	CPU utilization at Grafana	CNI0 interface	POD status	Scheduler name
Default scheduling	not defined	25 to 50 percentage (36 %)	49 KBps	Running	Default
Default scheduling	pod1,2,3 defined only	25 to 50 percentage (36 %)	90 KBps	Running	Default
Priority Aware scheduler	pod1,2,3 defined only	50 percentage	110kbps	pending first then ruuning	PAVA

Figure 20: Events for deploying pods in Priority aware algorithm

cause additional cni0 interface traffic upto 100 kBites. Kubernetes pods events denote that scheduler name is not shown in events because binding functionality defined by Kubelet not Kubernetes scheduler.

7 Conclusion and Future Work

Kubernetes Scheduler have capability to deal with custom scheduler mechanism. If we have to run priority aware (PAVA) custom scheduler then we have to add some additional resources . Additional advantage of priority aware scheduling , we can schedule the application parallel with default Kubernetes scheduler. We can modify the sequence , deletion ,creation of pods according to application requirements. For example, we have done priority for scheduling purpose , we can use network statistics or IP base pods binding on selective node or random manner.

In whole experimental setup challenge is the api integration of python client and libraries and platform dependencies like partitioning . Kubernetes default scheduling is written in go language so python base api library or binding function is in development phase. This can create API exception issues or mismatch parameters of API. We have to take accountability of security parameters. For example, when SSL settings is off then only the Custom Priority aware scheduler can be communicated with Kubernetes core API. Node affinity and node selectivity with network parameters like bandwidth, port base POD forwarding will be the area to be explore.

8 Acknowledgement

I would like to express special thanks to my supervisor Mr. Mohammad Iqbal one who encourages and guide us in every single stage of research work. His valuable inputs and problem-solving ability help us in our implementation phase. I also like to thanks my family and friends one who constantly supports me in this journey.

References

Alam, M., Rufino, J., Ferreira, J., Ahmed, S. H., Shah, N. and Chen, Y. (2018). Orchestration of Microservices for IoT Using Docker and Edge Computing, *IEEE Communications Magazine* **56**(9): 118–123.

Cérin, C., Menouer, T., Saad, W. and Abdallah, W. B. (2018). A New Docker Swarm Scheduling Strategy, *Proceedings - 2017 IEEE 7th International Symposium on Cloud and Service Computing, SC2 2017* **2018-January**: 112–117.

Chang, C. C., Yang, S. R., Yeh, E. H., Lin, P. and Jeng, J. Y. (2017). A Kubernetes-Based

Monitoring Platform for Dynamic Cloud Resource Provisioning, *2017 IEEE Global Communications Conference, GLOBECOM 2017 - Proceedings* **2018-January**: 1–6.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S. and Stoica, I. (n.d.). 2011_Benjamin Hindman_Benjamin Hindman_Mesos A Platform for Fine-Grained Resource Sharing in the Data Center.

Jha, D. N., Garg, S., Jayaraman, P. P., Buyya, R., Li, Z., Morgan, G. and Ranjan, R. (2019). A study on the evaluation of HPC microservices in containerized environment, *Concurrency Computation* (April): 1–18.

Jin, H., Wang, Y., Wang, Q., Liu, J., Wang, S., Zhang, J., Hao, S. and Fu, H. (2019). Architecture modelling and task scheduling of an integrated parallel cnc system in docker containers based on colored petri nets, *IEEE Access* **7**: 47535–47549.

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: Crunching Information of Everything, KST 2017* pp. 254–259.

Larrucea, X., Santamaria, I., Colomo-Palacios, R. and Ebert, C. (2018). Microservices, *IEEE Software* **35**(3): 96–100.

Li, W. and Kanso, A. (2015). Comparing containers versus virtual machines for achieving high availability, *Proceedings - 2015 IEEE International Conference on Cloud Engineering, IC2E 2015* pp. 353–358.

Medel, V., Tolosana-Calasanz, R., Bañares, J. Á., Arronategui, U. and Rana, O. F. (2018). Characterising resource management performance in Kubernetes, *Computers and Electrical Engineering* **68**(May 2017): 286–297.
URL: *https://doi.org/10.1016/j.compeleceng.2018.03.041*

Naik, N. (2017). Docker container-based big data processing system in multiple clouds for everyone, *2017 IEEE International Symposium on Systems Engineering, ISSE 2017 - Proceedings* .

Nardelli, M., Hochreiner, C. and Schulte, S. (2017). Elastic Provisioning of Virtual Machines for Container Deployment, pp. 5–10.

Netto, H. V., Lung, L. C., Correia, M., Luiz, A. F. and Sá de Souza, L. M. (2017a). State machine replication in containers managed by Kubernetes, *Journal of Systems Architecture* **73**: 53–59.
URL: *http://dx.doi.org/10.1016/j.sysarc.2016.12.007*

Netto, H. V., Lung, L. C., Correia, M., Luiz, A. F. and Sá de Souza, L. M. (2017b). State machine replication in containers managed by Kubernetes, *Journal of Systems Architecture* **73**: 53–59.
URL: *http://dx.doi.org/10.1016/j.sysarc.2016.12.007*

Ramalho, F. and Neto, A. (2016). Virtualization at the network edge: A performance comparison, *WoWMoM 2016 - 17th International Symposium on a World of Wireless, Mobile and Multimedia Networks* pp. 1–6.

Rodriguez, M. A. and Buyya, R. (2018). Containers Orchestration with Cost-Efficient Autoscaling in Cloud Computing Environments.
URL: *http://arxiv.org/abs/1812.00300*

Rodriguez, M. A. and Buyya, R. (2019). Container-based cluster orchestration systems: A taxonomy and future directions, *Software - Practice and Experience* **49**(5): 698–719.

Santos, J., Wauters, T., Volckaert, B. and De Turck, F. (2019). Towards network-Aware resource provisioning in kubernetes for fog computing applications, *Proceedings of the 2019 IEEE Conference on Network Softwarization: Unleashing the Power of Network Softwarization, NetSoft 2019* pp. 351–359.

Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M. and Wilkes, J. (2013). Omega: Flexible, scalable schedulers for large compute clusters, *Proceedings of the 8th ACM European Conference on Computer Systems, EuroSys 2013* pp. 351–364.

Son, J. and Buyya, R. (2018). Priority-Aware VM Allocation and Network Bandwidth Provisioning in Software-Defined Networking (SDN)-Enabled Clouds, *IEEE Transactions on Sustainable Computing* **4**(1): 17–28.

Suresh, L., Loff, J., Kalim, F., Narodytska, N., Ryzhyk, L., Gamage, S., Oki, B., Lokhandwala, Z., Hira, M. and Sagiv, M. (2019). Automating Cluster Management with Weave.
URL: *http://arxiv.org/abs/1909.03130*

Tao, Y., Wang, X., Xu, X. and Chen, Y. (2017). Dynamic resource allocation algorithm for container-based service computing, *2017 IEEE 13th International Symposium on Autonomous Decentralized System (ISADS)*, pp. 61–67.

Truyen, E., Van Landuyt, D., Lagaisse, B., Joosen, W. and Bruzek, M. (2018). Evaluation of Container Orchestration Systems for Deploying and Managing NoSQL Database Clusters, *IEEE International Conference on Cloud Computing, CLOUD* **2018-July**: 468–475.

Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E. and Wilkes, J. (2015). Large-scale cluster management at Google with Borg, *Proceedings of the 10th European Conference on Computer Systems, EuroSys 2015* .

Xu, M., Toosi, A. N. and Buyya, R. (2019). ibrownout: An integrated approach for managing energy and brownout in container-based clouds, *IEEE Transactions on Sustainable Computing* **4**(1): 53–66.

Xu, X., Yu, H. and Pei, X. (2015). A novel resource scheduling approach in container based clouds, *Proceedings - 17th IEEE International Conference on Computational Science and Engineering, CSE 2014, Jointly with 13th IEEE International Conference on Ubiquitous Computing and Communications, IUCC 2014, 13th International Symposium on Pervasive Systems, Algorithms, and Networks, I-SPAN 2014 and 8th International Conference on Frontier of Computer Science and Technology, FCST 2014* pp. 257–264.

Appendix

1. Introduction

Kubernetes cluster formation require step by step approach. This document helps to deploy Kubernetes cluster with Priority Aware application custom scheduler . Our configuration manual is classified into three section. Section 2 includes system configuration whether section 3 is brief overview of Priority aware scheduler code and step 4 represent evaluation software tool like Grafana and Prometheus

2 System configurations :-

Kubernetes cluster is deployed on the local virtual machine with ubuntu operating system. The details configuration has been listed below. Virtual machines are managing with oracle virtual box software 6.0 platform.

Deployment essentials	Description
Container orchestrator software	Kubernetes 1.16.0
container software	Docker 19.0.3
Application container	Ngnix , Redis
operating system	Ubuntu
Process configuration	AMD ryzen 5 with
No of vcpu reuired for master and slave	6 vcpu
RAM	16 GB minimum
Hardisk	40 GB SSD
Custom scheduler code language	Python 2.7 used
Manifiest language for intercommunication	YAML

2. Master and Slave configuration :-

We are deployed the master with 2 slave nodes in Kubernetes cluster which has the functionality to deploy the container base application for example Redis , ngnix. This master and slave are deployed on different virtual machine with specific configuration as listed below.

Virtual machine configuration for Implementation :-

Master VM configuration: -

- Virtual machine Ubuntu 18.04
- vCPU -3
- RAM =8 GB

- Storage =40 GB

Slave VM configuration: -

Virtual machine Ubuntu 18.04

VCPU -3

RAM =4 GB

Storage -40

Step 2) **Installation of Docker container on master and Slave**

In this we must enable the docker version in existing ubuntu Linux virtual machine. We have to enable docker in Linux operating system by below command. we are using docker version **19.03.2** for OS/architecture **linux amd 64**

#sudo apt-get install docker.io

sudo systemctl enable

```
prasad@k8s-master:~$ docker version


Client: Docker Engine - Community
 Version:           19.03.2
 API version:       1.40
 Go version:        go1.12.8
 Git commit:        6a30dfc
 Built:             Thu Aug 29 05:29:11 2019
 OS/Arch:           linux/amd64
 Experimental:      false
Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Get http://%2Fvar%2Frun%2Fdocker.soc
k/v1.40/version: dial unix /var/run/docker.sock: connect: permission denied
```

Step 3) Curl package and GPG key for installation for Kubernetes [1]

 # sudo apt-get install curl

---

[1] https://www.docker.com/

 # curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add[2]

Step 4) we have to add this key in google repository.

#sudo apt-add-repository "deb http://apt.kubernetes.io/ kubernetes-xenial main“

Step 5) **Installation of Kubernetes component that is Kubeadm , Kubectl**
Kubeadm –
Kubeadm is a tool built to provide **kubeadm** init and **kubeadm** join as best-practice “fast paths” for creating Kubernetes clusters
Kubectl - line interface for running commands against **Kubernetes** clusters.

```
#sudo apt-get install kubeadm
```

Check Kubectl version using command **# kubectl version**

```
prasad@k8s-master:~$
prasad@k8s-master:~$ kubectl version
Client Version: version.Info{Major:"1", Minor:"16", GitVersion:"v1.16.3", GitCommit:"b3cbbae08ec52a7fc73d334838e18d17e8512749", GitTreeState:"
clean", BuildDate:"2019-11-13T11:23:11Z", GoVersion:"go1.12.12", Compiler:"gc", Platform:"linux/amd64"}
```

Step 6) Partitioning need to be on the system using command. if not happen then local host. error can come. There is need to be one or 2 minutes required to reflect effect. Kubernetes cannot start without partitioning of Virtual machines.

 # swap off -a

```
prasad@k8s-master:~$
prasad@k8s-master:~$ sudo su
root@k8s-master:/home/prasad# swapoff -a
root@k8s-master:/home/prasad#
root@k8s-master:/home/prasad#
root@k8s-master:/home/prasad#
```

Step 7) Assignment of extra interface for intercommunication of Virtual machines. Check reachability of Virtual machines.

```
enp0s8: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.56.101  netmask 255.255.255.0  broadcast 192.168.56.255
        inet6 fe80::a4a1:3f56:e5ed:1fd6  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:d9:ac:b0  txqueuelen 1000  (Ethernet)
        RX packets 1338  bytes 194038 (194.0 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 894  bytes 256996 (256.9 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

```
[1]+  Stopped                 ping 192.168.56.101
root@k8s-master:/etc# ping 192.168.56.102
PING 192.168.56.102 (192.168.56.102) 56(84) bytes of data.
64 bytes from 192.168.56.102: icmp_seq=1 ttl=64 time=0.370 ms
64 bytes from 192.168.56.102: icmp_seq=2 ttl=64 time=0.573 ms
64 bytes from 192.168.56.102: icmp_seq=3 ttl=64 time=0.627 ms
^Z
[2]+  Stopped                 ping 192.168.56.102
root@k8s-master:/etc# ping 192.168.56.103
PING 192.168.56.103 (192.168.56.103) 56(84) bytes of data.
64 bytes from 192.168.56.103: icmp_seq=1 ttl=64 time=0.342 ms
64 bytes from 192.168.56.103: icmp_seq=2 ttl=64 time=0.581 ms
64 bytes from 192.168.56.103: icmp_seq=3 ttl=64 time=0.546 ms
^Z
[3]+  Stopped                 ping 192.168.56.103
root@k8s-master:/etc#
```

Step 8) Each Pod required IP address to allocate the resource. Kubernetes support multiple networks like flannel and calico network. We have to define some of private IP pool address to deploy POD which is not visible in public network. In our case we are using 40.168.0.0 /16 IP pool which can be used for allocating 65536 Ip address to POD.

Command to allocate to allocate the IP pool range and API server IP address is as below.

Kubeadm need to be installed on master node only we have set APIserver Ip address as 192.168.56.101 interface address.

Command :- # sudo kubeadm init --pod-network-cidr=<ip pool of Pods> --apiserver-advertise-address=<interface address>

```
root@k8s-master:/etc# sudo kubeadm init --pod-network-cidr=40.168.0.0/16 --apiserver-advertise-address=192.168.56.101
[init] Using Kubernetes version: v1.16.3
[preflight] Running pre-flight checks
        [WARNING SystemVerification]: this Docker version is not on the list of validated versions: 19.03.2. Latest validated version: 18.09
[preflight] Pulling images required for setting up a Kubernetes cluster
[preflight] This might take a minute or two, depending on the speed of your internet connection
[preflight] You can also perform this action in beforehand using 'kubeadm config images pull'
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
```

Step 9) we have to allow the access to local user for running kubectl command. Blow are the configuration needs to be run on the master node only.

mkdir -p $HOME/.kube

sudo c[3]p -i /etc/kubernetes/admin.conf $HOME/.kube/config

sudo chown $(id -u):$(id -g) $HOME/.kube/config

[3] https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/

```
[addons] Applied essential addon: kube-proxy

Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 192.168.56.101:6443 --token xj7u9q.xj6raqp4o3um2uyu \
    --discovery-token-ca-cert-hash sha256:fba6ec7167720f50b8eff9620485ecf80eb23641012aaa4676da872348c58e80
root@k8s-master:/etc#
```

Command output of Kubeadm needs to be save on notepad and execute the slave node of the cluster. we are executing this command slave 1 and Slave 2.

Step 10) **Kubernetes token** is created which can be help for authentication and signing

Below is the command is used for verification of the token.

kubeadm token list

```
root@k8s-master:/etc# kubeadm token list
TOKEN                     TTL       EXPIRES                USAGES                   DESCRIPTION
 EXTRA GROUPS
xj7u9q.xj6raqp4o3um2uyu   23h       2019-12-06T19:13:06Z   authentication,signing   The default bootstrap token generated by 'kubeadm init'.
 system:bootstrappers:kubeadm:default-node-token
```

Step 10) Token creation done using below command through which we can access Kubernetes dashboard.

 # openssl x509 -pubkey -in /etc/kubernetes/pki/ca.crt | openssl rsa -pubin -outform der 2>/dev/null | openssl dgst -sha256 -hex | sed 's/^. * //'

```
root@k8s-master:/etc# openssl x509 -pubkey -in /etc/kubernetes/pki/ca.crt | openssl rsa -pubin -outform der 2>/dev/null | openssl dgst -sha256
 -hex | sed 's/^.* //'
fba6ec7167720f50b8eff9620485ecf80eb23641012aaa4676da872348c58e80
root@k8s-master:/etc#
```

Step 11) Create **Flannel network** for intercommunication of PODs. Cluster role bindings are defined in this cluster.

sudo kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml[4]

```
root@k8s-master:/etc# sudo kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml
podsecuritypolicy.policy/psp.flannel.unprivileged created
clusterrole.rbac.authorization.k8s.io/flannel created
clusterrolebinding.rbac.authorization.k8s.io/flannel created
serviceaccount/flannel created
configmap/kube-flannel-cfg created
daemonset.apps/kube-flannel-ds-amd64 created
daemonset.apps/kube-flannel-ds-arm64 created
daemonset.apps/kube-flannel-ds-arm created
daemonset.apps/kube-flannel-ds-ppc64le created
daemonset.apps/kube-flannel-ds-s390x created
```

Step 12) **Master configuration verification :--**

Kubectl get nodes command is used to check whether master is created or not.

#kubectl get nodes

```
root@k8s-master:/etc# kubectl get nodes
NAME         STATUS   ROLES    AGE   VERSION
k8s-master   Ready    master   33m   v1.16.3
root@k8s-master:/etc#
```

 # kubectl get namespaces for checking isolation of process. Kube-node-lease,kube-public and kube-system are the Kubernetes internal namespaces .

```
See 'kubectl get -h' for help and examples
root@k8s-master:/etc# kubectl get namespaces
NAME              STATUS   AGE
default           Active   35m
kube-node-lease   Active   35m
kube-public       Active   35m
kube-system       Active   35m
root@k8s-master:/etc#
```

ip a show flannel command is used to check the flannel ip pool status.

[4] https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

```
root@k8s-master:/etc#
root@k8s-master:/etc#
root@k8s-master:/etc# ip a show flannel
Device "flannel" does not exist.
root@k8s-master:/etc# ip a show flannel.1
5: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN group default
    link/ether 76:cf:45:2f:ca:e0 brd ff:ff:ff:ff:ff:ff
    inet 40.168.0.0/32 scope global flannel.1
       valid_lft forever preferred_lft forever
    inet6 fe80::74cf:45ff:fe2f:cae0/64 scope link
       valid_lft forever preferred_lft forever
root@k8s-master:/etc#
```

Step 13**) Dashboard creation on the Kubernetes**

Kubernetes dashboard contain the cluster node binding , deployment and services bindings.

kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/v1.10.1/src/deploy/recommended/kubernetes-dashboard.yaml

```
root@k8s-master:/etc# kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v1.10.1/src/deploy/recommended/kubernetes-dashbo
ard.yaml
secret/kubernetes-dashboard-certs created
serviceaccount/kubernetes-dashboard created
role.rbac.authorization.k8s.io/kubernetes-dashboard-minimal created
rolebinding.rbac.authorization.k8s.io/kubernetes-dashboard-minimal created
deployment.apps/kubernetes-dashboard created
service/kubernetes-dashboard created
```

Step 14) **Kubernetes slave configuration**: -

In this we are repeating step from step 1 to step 6 after that we must use join request created on the master K8-master node. we must use join command to slave nodes.

 # Kubectl get nodes

```
root@k8s-master:/etc# kubectl get nodes

NAME                STATUS   ROLES    AGE    VERSION
k8s-master          Ready    master   128m   v1.16.3
prasad-virtualbox   Ready    <none>   18m    v1.16.2
slave-virtualbox    Ready    <none>   15m    v1.16.3
root@k8s-master:/etc#
root@k8s-master:/etc#
```

Step 15) Token creation required to form Kubernetes dashboard services, we have to form admin account in dashboard services. We have form default setting to access the   Kubernetes Dashboard services.

Command to create Dashboard setting is as Below.

Kubernetes create serviceaccount dashboard -n default

```
token}" | base64 --decode
Error from server (NotFound): serviceaccounts "dashboard" not found
root@k8s-master:/home/prasad# kubectl create serviceaccount dashboard -n default
serviceaccount/dashboard created
```

Step 16) In this step we have to create cluster role binding with Kubernetes dashboard service. This binding is required to access all nodes pods information.

```
root@k8s-master:/home/prasad# kubectl create clusterrolebinding dashboard-admin -n default --clusterrole=cluster-admin --serviceaccount=defaul
t:dashboard
clusterrolebinding.rbac.authorization.k8s.io/dashboard-admin created
```

Step17) Kubectl proxy command is used to allowed to start proxy server. If proxy server is not working, then we have to check whether 8001 port is bind with some other service or not.

Command for checking port binding: -

netstat -tulp| grep 8001

 # Kubectl proxy --port=8001

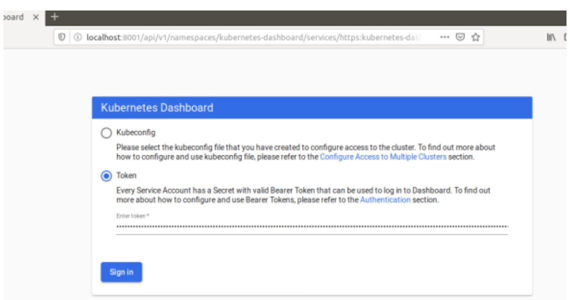
```
root@k8s-master:/home/prasad# kubectl proxy --port=8001
Starting to serve on 127.0.0.1:8001
```

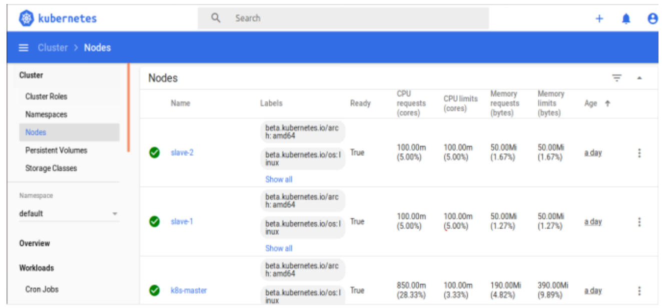
Step 18) Kubernetes dashboard can be access from below link on web brower. We have to use token base authentication command .

kubectl get secret $(kubectl get serviceaccount dashboard -o jsonpath="{.secrets[0].name}") -o jsonpath="{.data.token}" | base64 –decode

http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/ [5]

[5] https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/





PODs and Priorty class configuration for scheduling

Section 3) Priority Aware schdeuler   deplouyment

First step of   implemnetration of PAVA scheduler is creation of Pods.yaml file.We are used 5 fives pods for deployment and evaluation puroposes.

Step 18) Pod creation we have to define api version as V1 and kind should be in Pod as shown in figure. In Metadata name filed is used to identification pupose whether in specification. We have to defined the **schedulerName** as **PAVA**.

```
apiVersion: v1
kind: Pod
metadata:
  name: p2
spec:
  schedulerName: PAVA
  containers:
  - name: redis
    image: redis
    imagePullPolicy: IfNotPresent
    resources:
        limits:
                cpu: "1"
        requests:
                cpu: 500m
  priorityClassName: medium-priority

~
```

```
apiVersion: v1
kind: Pod
metadata:
  name: p3
spec:
  schedulerName: PAVA
  containers:
  - name: redis
    image: redis
    imagePullPolicy: IfNotPresent
    resources:
        limits:
                cpu: "1"
        requests:
                cpu: 500m
  priorityClassName: low-priority
~
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: p1
spec:
  schedulerName: PAVA
  containers:
  - name: redis
    image: redis
    imagePullPolicy: IfNotPresent
    resources:
        limits:
                cpu: "1"
        requests:
                cpu: 500m
  priorityClassName: high-priority
```

For Pods pod4 and pod5 are scheduler by default scheduler mechanism. For yaml file we have to define the **scheduler name** as **default**.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod5
spec:

  containers:
  - name: redis
    image: redis
    imagePullPolicy: IfNotPresent
    resources:
        limits:
                cpu: "1"
        requests:
                cpu: 500m
  priorityClassName: high-priority
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod4
spec:

  containers:
  - name: redis
    image: redis
    imagePullPolicy: IfNotPresent
    resources:
        limits:
                cpu: "1"
        requests:
                cpu: 500m
~
~
```

Step 19) we have to create policy map for assigning the policy map to define scheduling the pods. We are set priority as 100,500,1000. According to this priority values PAVA scheduler the nodes.

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: low-priority
value: 100
globalDefault: no
description: "PAVA"
```

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: medium-priority
value: 500
globalDefault: false
description: "PAVA"
```

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority
value: 1000
globalDefault: no
description: "PAVA"
```

Step 20) After creation of pods , we will create the pods using the # **kubectl create -f <podname.yaml>.** The state of pods is shown in command line interface when we execute the command #kubectl get pods

```
root@k8s-master:/etc# kubectl get pods
NAME    READY   STATUS     RESTARTS   AGE
pod1    1/1     Running    0          17m
pod2    1/1     Running    0          23m
pod3    0/1     OutOfcpu   0          55s
pod4    1/1     Running    0          22m
pod5    1/1     Running    0          22m
root@k8s-master:/etc#
```

Step 21) Experimental code for Priority Aware scheduling Algorithm is as below .

We have to install the python kuberntes library to interatct python command lines the kuberntes core api. Pip install kubnertes command in useses for API interaction only .

The libaries that are using for Kuberntes API interaction purpose are kuberntes client,config and watches. Without this libraries our client api can not works.

#config.load_kube_config() – we are using for the loading kubenrtes all core class config.

C1=client.configuration

C1.verify_ssl= This configuration we are using to avaoid ssl issue . Kubernetes API interaction can be taken without ssl from this configuration.

Schdeuler_name as PAVA (priority aware we had set for matching purpose)

```
import re
import sys
import kubernetes.client
from pint        import UnitRegistry
from collections import defaultdict
from kubernetes import client, config, watch
from kubernetes.client.rest import ApiException
import time
import requests

config.load_kube_config()
c1=client.Configuration()
c1.verify_ssl=False
scheduler_name="PAVA"
###################### Core api of Kubernetes  from which we are gathering data in YAML ##
v1=client.CoreV1Api()
ret = v1.list_pod_for_all_namespaces(watch=False)
Scheduler_name='foobar'
group_priority =[]
dict_obj1=[]
group_pod={}
group_name=[]
```

- We are using for loop for taking the metatdata,namespace and priority field from the vore api of Kubernetes. As shown in digaram below priority aware algorithm sorting functionality returns to sort pods name against the priority and store this value **in sorted_pod dictionary**.

```
########################################  available PODs from core api server  of Kubernetes for scheduling  #######

print ("\n \n \n \n Running pod on current kubernetes cluster \n \n \n \n  ")
for j in ret.items:
    print(" %s\t%s\t%s\t%s\t%s\t%s" % (j.status.pod_ip, j.metadata.namespace, j.metadata.name,j.spec.priority,
        j.spec.scheduler_name,j.status.phase))


################################### priority aware algorithm works ####################################


print("\n \n \n \n scheduling pod details are as below \n \n  ")

for i in ret.items:
    Scheduling_priority =[]
    if i.spec.scheduler_name=='PAVA':
        print("%s\t%s\t%s\t%s\t%s\t%s\t%s" % (i.status.pod_ip, i.metadata.namespace,
         i.metadata.name,i.spec.priority,i.spec.scheduler_name,i.status.phase,
         i.spec.priority_class_name))
        group_priority.append(i.spec.priority)
        group_name.append(i.metadata.name)
        group_pod=dict(zip(group_name,group_priority))
        Sorted_poddict=sorted(group_pod.items(),key=lambda kv:(kv[1],kv[0]),reverse=True)

print ("\n \n  pods details captured in dictionary  before  sorting  %s \n \n " %group_pod)
```

- We are setting the Reverse flag as True which assign the pods name and value in Highest to lowest manner.

 # reverse=True

```
############################ Scheduled POD with highest priority first############################################################

print("sorted PODs namespaces and priority value %s"%sorted(group_pod.items(),key=lambda kv:(kv[1],kv[0]),reverse=True))
```

- Compute allocated resoources are function which can is used for calculation of memory and CPU score utilizaed by the kubernte master and slave.We are stored this memory score and cpu score in **data1 variable**.

```
########################## Selecting best slave nodes  for deployment on the basis of CPU
__all__ = ["compute_allocated_resources"]

def compute_allocated_resources():
    ureg = UnitRegistry()
    ureg.load_definitions('kubernetes_units.txt')

    Q_   = ureg.Quantity
    data = {}

    # doing this computation within a k8s cluster
 #   config.load_incluster_config()
    config.load_kube_config()
    core_v1 =client.CoreV1Api()

    for node in core_v1.list_node().items:
        stats          = {}
        node_name      = node.metadata.name
        allocatable    = node.status.allocatable
        print allocatable
        max_pods       = int(int(allocatable["pods"]) * 1.5)
        field_selector = ("status.phase!=Succeeded,status.phase!=Failed," +
                          "spec.nodeName=" + node_name)

        stats["cpu_alloc"] = Q_(allocatable["cpu"])
        print stats["cpu_alloc"]
        stats["mem_alloc"] = Q_(allocatable["memory"])

        pods = core_v1.list_pod_for_all_namespaces(limit=max_pods,
                                                   field_selector=field_selector).items
```

```
        stats["cpu_req"]     = sum(cpureqs)
        print stats["cpu_req"]
        stats["cpu_lmt"]     = sum(cpulmts)
        print stats["cpu_lmt"]
        stats["cpu_req_per"] = (stats["cpu_req"] / stats["cpu_alloc"] * 100)
        print stats["cpu_alloc"]
        print stats["cpu_req_per"]
        stats["cpu_lmt_per"] = (stats["cpu_lmt"] / stats["cpu_alloc"] * 100)
        stats["mem_req"]     = sum(memreqs)
        stats["mem_lmt"]     = sum(memlmts)
        stats["mem_req_per"] = (stats["mem_req"] / stats["mem_alloc"] * 100)
        stats["mem_lmt_per"] = (stats["mem_lmt"] / stats["mem_alloc"] * 100)

        data[node_name] = stats

    return data

data1= compute_allocated_resources()
print (data1)
```

- Watch functionality is used to create watch stream which checks the status of pods whether it is pending and macthces with schdeuler name as “PAVA “ . After matching two conditioons scheduler check status of node to bind the pods.

```
################################################Watch function to check POD and append value #######################
w = watch.Watch()
for event in w.stream(v1.list_namespaced_pod,"default"):
    if event['object'].status.phase == "Pending"  and event['object'].status.conditions == None and
    event['object'].spec.scheduler_name == scheduler_name:
       # print event['object']
        r1 = event['object'].metadata.name
        #print event['object'].namespace
        print r1
        ready_nodes=[]
        for n in v1.list_node().items:
            for status in n.status.conditions:
                if status.status == "True" and status.type == "Ready":
                    ready_nodes.append(n.metadata.name)
        node="slave-1"
        print group_pod
        group_name=list(group_pod.keys())
        for item in group_name:
            name=item
            print name
            namespace="default"
            target=client.V1ObjectReference(kind='Node',api_version = 'v1', name = node,namespace="default")
            meta=client.V1ObjectMeta(name=name)
            body= kubernetes.client.V1Binding(target=target,metadata=meta)
            print body
            try:
                res = v1.create_namespaced_binding(namespace=namespace,body=body,_preload_content=False)
                print res
            except Exception as a:
                print ("Exception when calling CoreV1Api->create_namespaced_binding: %s\n" % a)
```

group_pod stores the pod metadata name and value as priority field. We are running binding function for each pod against metadata nam **("group_name")**

We have bind the pods with namespace as default , api_version= "v1 and execute the v1.create_namespaced_binding function.

Section 4:- Monitoring tool Grafana and prometheus setup

Prometheus setup:- prometheus is open source tool which is intergrated with node exported software tool to extract the linux information while running kubernetes master. In our experimental setup we are using prometheus 2.13.1-linux amd version .

Step 23) Prometheus software can be get from wget https://prometheus.io/download/prometheus-2.13.1.linux-amd64.tar.gz

```
root@k8s-master:/home/prasad/promentheus# ls
grafana-6.4.4                   node_exporter-0.18.1.linux-amd64        prometheus-2.13.1.linux-amd64        prometheus-files
grafana-6.4.4.linux-amd64.tar.gz  node_exporter-0.18.1.linux-amd64.tar.gz  prometheus-2.13.1.linux-amd64.tar.gz
root@k8s-master:/home/prasad/promentheus#
```

```
root@k8s-master:/home/prasad/promentheus/prometheus-2.13.1.linux-amd64# ls
charts  config-map.yaml  consoles  data  kubernetes-prometheus  LICENSE  NOTICE  p1.yaml  prometheus  prometheus.yml  promtool  tsdb
root@k8s-master:/home/prasad/promentheus/prometheus-2.13.1.linux-amd64#
root@k8s-master:/home/prasad/promentheus/prometheus-2.13.1.linux-amd64#
root@k8s-master:/home/prasad/promentheus/prometheus-2.13.1.linux-amd64#
```

Step 24) Node exporter   download from the below official link of prometheus

```
wget https://github.com/prometheus/node_exporter/releases/download/v*/node_exporter-*.*-amd64.tar.gz

tar xvfz node_exporter-*.*-amd64.tar.gz

cd node exporter-*.*-amd64

./node exporter
```

Step 24) we have to edit the configuration in File name iof   prometheus.yaml at /etc/prometheus

```
# Alertmanager configuration
#alerting:
#  alertmanagers:
#  - static_configs:
#    - targets:
      # - alertmanager:9093

# Load rules once and periodically evaluate them according to the glob
rule_files:
  # - "first_rules.yml"
  # - "second_rules.yml"

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeserie
  - job_name: 'prometheus'

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
    - targets: ['localhost:9090']
  - job_name: 'node_exporter'
    static_configs:
            - targets: ['localhost:9100']
```

Step 25) Grafana setup for prometheus as Datasource.

Grafan download from the the official wensite http://grafana.com

- wget https://dl.grafana.com/oss/release/grafana_6.5.1_amd64.deb
- sudo dpkg -i grafana_6.5.1_amd64.deb

```
root@k8s-master:/home/prasad/promentheus# ls
grafana-6.4.4                         node_exporter-0.18.1.linux-amd64
grafana-6.4.4.linux-amd64.tar.gz      node_exporter-0.18.1.linux-amd64.tar.gz
root@k8s-master:/home/prasad/promentheus#
```

- sudo systemctl daemon-reload ---------- For demone reloaded
- sudo systemctl start grafana-server ------ for starting grafana server
- sudo systemctl status grafana-server ------ To check the status of Grafana

- sudo systemctl status grafana-server ---- Starting of Grafna server

```
root@k8s-master:/home/prasad/promentheus/grafana-6.4.4/bin#
root@k8s-master:/home/prasad/promentheus/grafana-6.4.4/bin# ./grafana-server
INFO[12-11|07:56:15] Starting Grafana                 logger=server version=6.4.4 commit=092e514 branch=HEAD compiled=2019-11-06T12:04
:33+0000
INFO[12-11|07:56:15] Config loaded from               logger=settings file=/home/prasad/promentheus/grafana-6.4.4/conf/defaults.ini
INFO[12-11|07:56:15] Path Home                        logger=settings path=/home/prasad/promentheus/grafana-6.4.4
INFO[12-11|07:56:15] Path Data                        logger=settings path=/home/prasad/promentheus/grafana-6.4.4/data
INFO[12-11|07:56:15] Path Logs                        logger=settings path=/home/prasad/promentheus/grafana-6.4.4/data/log
INFO[12-11|07:56:15] Path Plugins                     logger=settings path=/home/prasad/promentheus/grafana-6.4.4/data/plugins
INFO[12-11|07:56:15] Path Provisioning                logger=settings path=/home/prasad/promentheus/grafana-6.4.4/conf/provisioning
INFO[12-11|07:56:15] App mode production              logger=settings
INFO[12-11|07:56:15] Initializing SqlStore            logger=server
```

Step 25) Server will be start at http:\\localhost:3000

Conclusion :-

In this manual we are explaniing brief overview of step by step approach to deploy the kubernetes cluster with monitoring tool grafana and Prometheus. The necessary results are shown and highlighted . Code explanation has been given in such a way that user will understand the basic understanding of the code.