Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing applications

Abdul Ahad Ayaz

Summer term 2020

Abstract -

1 Introduction

In recent years with the evolution of technology, Internet of Things (IoT) devices are increasing day by day. According to Ericsson mobility report[], there will be 17% (approx. 22.3 billion) increase in IoT devices by 2024. Functionally IoT is defined as *"The Internet of Things allows people and things to be connected Anytime, Anyplace, with Anything and Anyone"* [**European commission 2008**]. IoT devices have served mankind in many ways such as from smart houses to smart cities, smart transportation systems and many medical applications. These IoT applications enables many devices connected to network and generates alot of heterogeneous data also known as BigData which requires special data processing models and Infrastructure support. Processing BigData required alot of resources and cloud computing theoratically provides it unlimited resources[**fog-comp-survey**]. But there is downside of using cloud computing for such complex computation as it is more costly when it comes to computation power, storage and bandwidth. Computation need to be performed at the node level and only the aggregated data need to send to central node for further computations and analysis. This de-centralized approach will save alot of computation power as well as bandwidth requriments[**fog-comp-survey**]. To overcome the downside of cloud computing, the terminology fog computing is used. Fog computing allows the computation at the egde of network instead of central core.

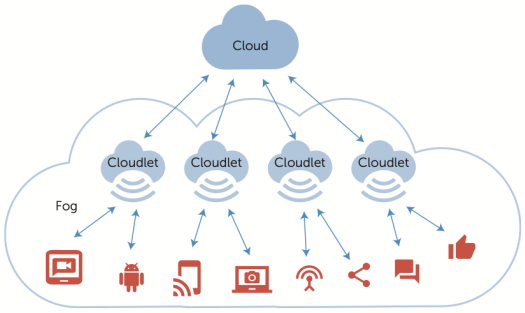

Figure 1: Fog computing: Top-level overview[8]

Fog computing is defined as *"an archietecture that uses one or a collaborative multitude of end-user clients or near-user edge devices to carry out a substantial amount of storage (rather than stored primarily in cloud data centers), communication (rather than routed over the internet backbone), and control, configuration, measurement and management (rather than controlled primarily by network gateways such as those in the LTE (telecommunication) core)"*[**Chiang 2015; Aazam and Huh 2014**]. Traditionally, user applications running in cloud access the cloud core network through access points for data exchange to fetch data from data-centers [8]. In fog computing these access points also serves as resource providers such as computation power and storage etc. and are called "cloudlets"[8]. Figure 1 show the top-level archietecture of the fog computing.

Fog computing is responsible for providing resources to IoT devices for processing. Traditionally these resources are allocated as VMs from different cloud infrastructures such as AWS, Google, OpenStack, etc. to run the applications. VMs are considered resource greedy and require more computational resources. Alternate is to use the Containers such as Docker which are light-weight, requires less resources and based on micro-service architecture. Large applications are split into containers based on the main processes of the application. This increasing number of containers per application required the proper monitoring for health check and resource consumption. The most commonly used orchestrator for containers is Kubernetes.

Kubernetes act as IaaS for fog computing to provide resource for IoT applications. Kubernetes is an open-source platform for management, deployment and scaling of containers. In Kubernetes, applications are deployed as pod consisting of multiple containers. When the configuration of deploying application is passed to Kubernetes, it checks for the availability of resources and deploys afterward. Kubernetes default resource scheduler monitor and deploys the pod using computation power-based scheduling mechanism and does not consider latency and available bandwidth, which is considered important while dealing with data-centric application. Example of data-centric application is weather forecast that receives data from scattered IoT devices and provide prediction. If the data is lost or delayed due higher latency and poor bandwidth, timely decisions cannot be made that leads to disaster. To overcome this drawback of Kubernetes, author proposed an alternate Kubernetes scheduler that consider network resources along with computational resources.

2 Background

This section explains about the Kubernetes main components, working as Orchestrator and built-in resource provisioning techniques.

2.1 Kubernetes Main Components

Kubernetes is an open-source project that manages the container-based applications deployed over multiple hosts. Kubernetes act as Orchestrator which is responsible for deploying, managing, scaling of container-based application[**kubernetes-github-repo**]. Figure 2 shows the main components coupled to work as one unit called kubernetes. It is based on master-slave model, consisting of one *master node* and multiple *worker nodes* [6]. *worker nodes* can be either physical or virtual resource such as physical servers or virtual machines. *master node* communicates with *worker nodes* using *API* calls. *API server* uses RESTFul API, for managing all the *API* calls is also part of *master node*. End-users communicates with kubernetes cluster using *Kubectl*, which forward user requests to *API server* and intern gets the result. *Etcd* stores the data as key-value pair,which is used to store all configurations, states. It is one of the main component of kubernetes, which maintians the state across the cluster for synchorization of data. *Control Manager* is resposible for monitoring of *Etcd*. For any state change of cluster, *Control Manager* forward the new state request using *API server*. *Kube Scheduler* is discussed later in section 2.3. On *worker node*, node agent known

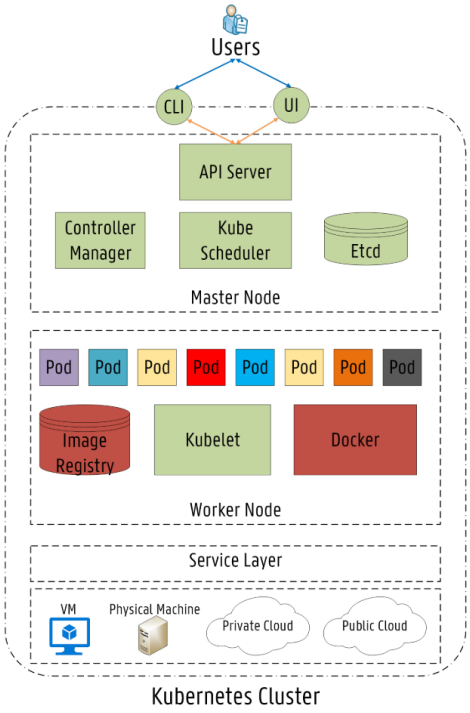

Figure 2: Kubernetes Cluster: Main Components[6]

as *Kubelet* which is resposible for maintain state based on *API server* request. For any state change communicated by *API server*, *Kubelet* performs the desired operation such as starting or deleting of Docker containers. *Image Registry* is resposible for managing the images required to create the container applications. *Pod* is main component of *worker node* where all the applications are deployed. Single *Pod* represents the application which consists of multiple containers based on the services of application. *Pod* is the collections of containers, volumes in an isolated environment which means there is no cross communication between two *Pods*. Containers running in a *Pod* share the same IP Address [6]. Containers communicates using different ports, hence there is a limitaion to this apporach as two containers listening on same port cannot be in same *Pod*[6].

2.2 Kubernetes as Orchestrator

Orchestrator is responsible for automating the processes that requires alot of human effort. As discussed in [1], Orchestrator is responsible for following:

- Starting or stopping of different applications.
- Ensure Scalabilty of application for high usage demands.
- Management of load across different nodes to avoid resource overhead.
- Monitoring health of applications.

There are many Orchestrator currently available, but the most widely used are OpenStack and Kubernetes.

1. **OpenStack Orchestration:** It provides template-based orcestration for cloud application to run on OpenStack. Template allows to create resources such as Virtual machines(Instances), Storage (Volumes), Networks etc. These resources are coupled together as OpenStack *Project* to run cloud application[2].
2. **Kubernetes Orchestration:** It is responsible for automating deployment, scaling and management of container-based applications. *master node* orcestrates the application across various *worker node* based on resource availability.

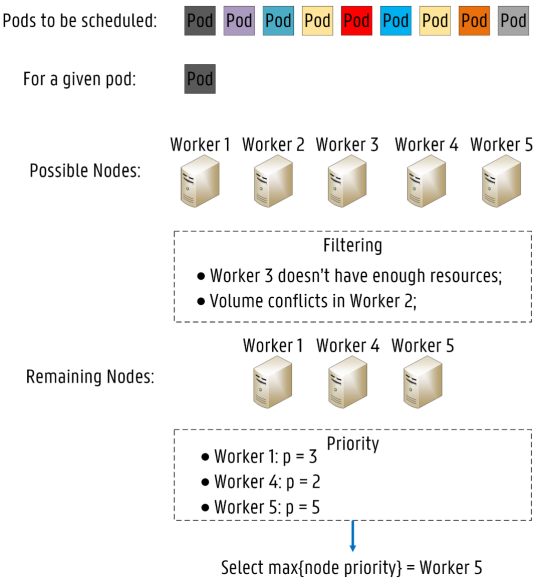

Figure 3: Kubernetes Scheduler: Working[6]

2.3 Kubernetes Resource Provisioning

When the user provides the configuration for creating new *pod* using *Kubectl*. *Pod* is added to the waiting queue with all the other *pods*. *Kube-Scheduler* which is the default scheduler of kubernetes, decides which *pod* deploys on which *worker node* based on some criteria. Figure 3 shows the default scheduling mechanism where *pod* is deployed by passing through following steps: *node filtering* and *node priority or scoring*[6]. In the kubernetes cluster*worker nods* meeting the requirement of *ped* are called *feasible nodes*[3].

2.3.1 *Node Filtering*

The first step of deploying *pod* is *node filtering* in which *Kube-Scheduler* will select the *feasible nodes* based on the *pod* configuration by applying some filters. These filter are also called *predicates*. Following is the list of *predicates* that are supported by *Kube-Scheduler*[3]:

1. **PodFitsHostPorts:** This filters checks the *worker node* for the ports requested by the *pod*.

2. **PodFitHost:** This filter checks for *worker node* with hostname mentioned in *pod* configuration.

3. **PodFitsResources:** This filter checks for the available resources i.e. CPUs and Memory to run the *pod*.

4. **NoDiskConflict:** This filter checks the *worker node* for the volumes requested by the *pod* and are already mounted.

5. **CheckNodeMemoryPressure:** This filter checks the *worker node* for over-utilization of Memory.

6. **CheckNodeDiskPressure:** This filter checks the *worker node* disk space and filesystem, sufficient to run the *pod*.

7. **CheckNodeCondition:** This filter checks the *worker node* for available disk space, networking configuration and that of *Kubelet* is reachable or not.

8. **PodMatchNodeSelector:** This filter search for the *worker node* based on the label mentioned in *pod* configuration. These labels allows the user to deploy the *pod* on specfic *worker node*(node-affinity)[6]. Other usecase of using label is to restrict the *pod* deployment based on other *pod* already deployed on that *worker node* (pod-anti-affinity) [6]. These affinity rules are based on *Tolerations* and *Taints* which are defined as key-value pair along with their effects. *Tolerations* are defined in *pod* configration whereas *Taints* are set for *worker node*. Both *Tolerations* and *Taints* work together to ensure *pod* is not deployed on inapproriate *worker node* [3].

Using the above mentioned filters(*predicates*), *Kube-Scheduler* returns the *feasible node* for *pod* deployment. If no *feasible node* is found, *pod* remains unscheduled and error message is generated for failed deployment [6]. If the list of *feasible node* is returned as the result of applying filters then *Kube-Scheduler* moves to second step *node priority or scoring*.

2.3.2 *Node Priority/Scoring*

Kube-Scheduler assign rank to each *worker node* that passes the *node filtering* stage. These ranks/priorities sort the list of *worker node* based on best-fit for *pod* deployment. These priorities are set based on following criteria[3]:

1. **SelectorSpreadPriority:** "This priority algorithm tries to minimize the number of deployed pods belonging to the same service on the same node or on the same zone/rack"[6].

2. **InterPodAffinityPriority:** This priority sets the score for *worker node* based on the pod-affinity rule mentioned above.

3. **LeastRequestedPriority:** This priority sets the score for *worker node* based on the higher available resources i.e. CPU and Memory.

4. **MostRequestedPriority:** This priority sets the score for *worker node* based on the minimum resource requirement for *pod* deployment.

5. **RequestedToCapacityRatioPriority:** This priority sets the score for *worker node* based on request to capacity using ResourceAllocationPriority.

6. **BalancedResourceAllocation:** This priority selects the *worker node* with balanced resource utilization.

7. **NodeAffinityPriority:** This priority selects the *worker node* based on node-affinity rule. *Worker node* with the required label will be given priority.

8. **TaintTolerationPriority:** This priority sets the score for *worker node* based on their *taints* with respect to *tolerations* mentioned in *pod* configuration[6].

9. **ImageLocalityPriority:** This priority sets the score for *worker node* based on the availability of the image on *worker node* required to build the containers for *pod*.

10. **EqualPriority:** This priority sets the equal weight to all the *worker nodes*.

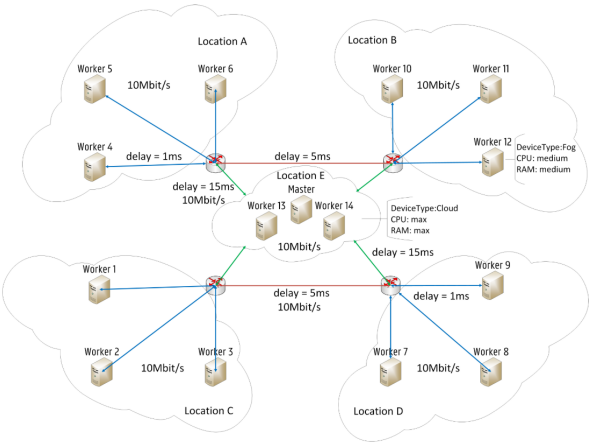

Figure 4: Kubernetes-based Fog Computing Infrastructure[6]

3 Kubernetes Network-based Resource Provisioning

The default *Kube-Scheduler* works efficiently for the resources such as CPU, Memory and storage but does not consider the networking resource which is consider critical resource in many use-case scenarios. Considering one application of Fog Computing such as IoT based smart cities which is data sensitive use-case. Ensuring that no data is lost, networking resource need to be configured properly[6]. Default *Kube-Scheduler* does not check the network latency and available bandwidth for *worker node*. In order to cater this drawback, author in [6] proposed a scheduler thats checks for the network resources along with the dedault *Kube-Scheduler*.

Kubernetes allows three ways to extend the *Kube-Scheduler* to allow the network-based resource provisioning[3].

- Extending *Kube-Scheduler* by adding new *filter/predicates* or *priority/scoring*.
- Build the new scheduler that replaces the default *Kube-Scheduler* or two schedulers work together.
- Define the scheduling process that can be called by the default *Kube-Scheduler* before scheduling the resources.

As per [6], author used the third approach that allows the default *Kube-Scheduler* to apply *filters* and calculate *priority* of the *worker nodes* afterwards the external scheduling process function is called. When the scheduling process is called, two function calls are generated[6], first one for the list of *worker nodes* as the result of *filtering/predicates* of *Kube-Scheduler*[6]. Second for the list of *worker nodes* after calculating the *priority* using *Kube-Scheduler*[6]. For Kubernetes to work as Fog Computing infrastructure, "affinity/anit-affinity" rules and node labeling is used as shown figure 4. The infrastructure consists of 1 *master node* and 14 *worker nodes*. All nodes are labeled with {*Min, High, Medium*} for resources such as {*CPU, Memory*}[6]. These nodes are further labeled by {*DeviceType*} based on their their functionality and geographical positioning by *tains* such as {*Cloud, Fog*}[6]. These rules and node labeling will help in efficient deployment of *pods* on certain *worker nodes*. Considering the delay-sensitive application scenario, data collecting node which is near to data processing node is taken into account due to time dependency[6]. For improving *pod* deployment based on above scenario, all *worker nodes* are further labeled for Round Trip Time(RTT) from the *master node* as shown in figure5.

The external scheduling process further calls two schedulers and these schedulers will filter out *worker nodes* based on networking resources.

1. **Random Scheduler:** This scheduler will get the input as list of *worker nodes* from *Kibe-Scheduler* after applying *filters/predicates* and output will be the random picked *worker node*

Node	RTT-A	RTT-B	RTT-C	RTT-D	RTT-E
Master	32 ms	32 ms	32 ms	32 ms	4 ms
Worker 1	64 ms	64 ms	4 ms	14 ms	32 ms
Worker 2	64 ms	64 ms	4 ms	14 ms	32 ms
Worker 3	64 ms	64 ms	4 ms	14 ms	32 ms
Worker 4	4 ms	14 ms	64 ms	64 ms	32 ms
Worker 5	4 ms	14 ms	64 ms	64 ms	32 ms
Worker 6	4 ms	14 ms	64 ms	64 ms	32 ms
Worker 7	64 ms	64 ms	14 ms	4 ms	32 ms
Worker 8	64 ms	64 ms	14 ms	4 ms	32 ms
Worker 9	64 ms	64 ms	14 ms	4 ms	32 ms
Worker 10	14 ms	4 ms	64 ms	64 ms	32 ms
Worker 11	14 ms	4 ms	64 ms	64 ms	32 ms
Worker 12	14 ms	4 ms	64 ms	64 ms	32 ms
Worker 13	32 ms	32 ms	32 ms	32 ms	4 ms
Worker 14	32 ms	32 ms	32 ms	32 ms	4 ms

Figure 5: Location-based RTT values of *nodes* in Fog Computing Infrastructure[6]

Input: Remaining Nodes after Filtering Process in
Output: Node for the service placement out

```
1: //Handle a provisioning request
2: handler(http.Request){
3:     receivedNodes = decode(http.Request);
4:     receivedPod = decodePod(http.Request);
5:     node = selectNode(receivedNodes, receivedPod);
6:     return node
7: }
8: //Return the best candidate Node (recursive)
9: selectNode(receivedNodes, receivedPod){
10:    targetLocation = getLocation(receivedPod);
11:    minBandwidth = getBandwidth(receivedPod);
12:    min = math.MaxFloat64;
13:    copy = receivedNodes;
14:    // find min RTT
15:    for node in range receivedNodes{
16:       rtt = getRTT(node, targetLocation);
17:       min = math.Min(min, rtt);
18:    }
19:    // find best Node based on RTT and minBandwidth
20:    for node in range receivedNodes{
21:       if min == getRTT(node, targetLocation){
22:          if minBandwidth ≤ getAvBandwidth(node){
23:             return node;
24:          }
25:          else
26:             copy = removeNode(copy, node);
27:       }
28:    }
29:    if copy == null
30:       return null, Error("No suitable nodes found!");
31:    else
32:       return selectNode(copy, receivedPod);
33: }
```

Figure 6: Network-Aware scheduling Algorithm[6]

from the input list.

2. **Network-Aware Scheduler:** This scheduler is based on algorithm as shown in figure6. Based on the algorithm, input will be list of *worker node* from *Kube-Scheduler* after applying *filters/predicates*. After getting the deploy location from *pod* configration file, this scheduler will make use of RTT labels to pick the best-fit *worker node* having minimum RTT value[6]. Apart from RTT based selection, this scheduler also look for the bandwidth label and check the *pod* configration file for bandwidth requirement. If no bandwidth requirement is specfied then the scheduler consider 250KBit/s by default and returns the *worker node* having minimum RTT and more bandwidth[6].

4 Performance Evaluation

In order to test the network-based resource provisioning for Fog Computing, Smart city scenario was considered [6]. This scenario collects the air quality data of Antwerp city for organic compounds in the atmosphere [6].

4.1 Expermentation Setup

This smart city scenario was tested using the infrastructure as shown in figure4. This infrastructure was setup at IDLab,Belgium[6]. The proposed network-based resource provisioning/network-aware scheduler was developed using Go and used as *pod* in Kubernetes cluster[6]. Figure7 shows the *pod* configration that consists of two containers, first container "extender" performs the network-ware resource scheduling operations whereas second container "network-aware-scheduler" is the *Kube-Scheduler* itself[6]. Configuration can be seen in figure9, where it can be seen that first *Kube-scheduler* operations are performed afterward "extender" is called that performs the network-aware scheduling based on algorithm as shown in figure6.

In smart city scenario, many services were deployed as shown in figure9.All the services were deployed either single or multiple *pods*. Figure9 shows all the defined paramerters in configuration file against each *pod*. The data collection of air quality was done through algorithm and implemented as

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
 labels:
  component: scheduler
  tier: control-plane
 name: network-aware-scheduler
 namespace: kube-system
spec:
 selector:
  matchLabels:
   component: scheduler
   tier: control-plane
 replicas: 1
 template:
  metadata:
   labels:
    component: scheduler
    tier: control-plane
    version: second
  spec:
   tolerations:
   - key: "function"
    operator: "Equal"
    value: "master"
    effect: "NoSchedule"
   serviceAccountName: network-aware-scheduler
   containers:
   - name: extender
    image: jpedro1992/network-aware-scheduler:0.0.3
    ports:
    - containerPort: 8100
   - name: network-aware-scheduler
    image: mirrorgooglecontainers/kube-scheduler:v1.12.3-beta.0
    command:
    - /usr/local/bin/kube-scheduler
    - --address=0.0.0.0
    - --leader-elect=false
    - --scheduler-name=network-aware-scheduler
    - --policy-configmap=network-aware-scheduler-config
    - --policy-configmap-namespace=kube-system
    livenessProbe:
     httpGet:
      path: /healthz
      port: 10251
     initialDelaySeconds: 15
    readinessProbe:
     httpGet:
      path: /healthz
      port: 10251
    resources:
     requests:
      cpu: '0.1'
    securityContext:
     privileged: false
    volumeMounts: []
   hostNetwork: false
   hostPID: false
```

Figure 7: *Pod* Configuration for Scheduler[6]

```
kind: ConfigMap
apiVersion: v1
metadata:
 name: network-aware-scheduler-config
 namespace: kube-system
data:
 policy.cfg: |
  {
    "kind" : "Policy",
    "apiVersion" : "v1",
    "metadata" : {
       "name": "network-aware-scheduler-config",
       "namespace": "kube-system"
    },
    "predicates" : [
       {"name" : "PodFitsResources"},
       {"name" : "PodFitsHostPorts"},
       {"name" : "NoDiskConflict"},
       {"name" : "NoVolumeZoneConflict"},
       {"name" : "PodToleratesNodeTaints"},
       {"name" : "MatchInterPodAffinity"}
    ],
    "extenders": [
       {
          "urlPrefix": "http://127.0.0.1:8100",
          "apiVersion": "v1",
          "filterVerb": "filter",
          "enableHttps": false
       }
    ]
  }
```

Figure 8: *Kube-Scheduler* Configuration[6]

container which is deployed as *pod* namely "birch-api"[6]. The *pod* configuration of "birch-api" is similar to one shown in figure7. Each *pod* of the service had some additional parameters in *pod* configration file such as "*targetLocation*" that will define the deploy location in Fog Infrastructure as shown in figure4. Another parameter "*bandwidthReq*" that defines the minimum required bandwidth for *pod* deployment. Furthermore in *pod* configuration file, *affinity* parameter was set to *podAntiAffinity* which limits the *pods* of same service deploying on same *worker node*[6].

4.2 Analysis of Kubernetes Default and Network-based Resource Provisioning

In order to check the Performance of network-based resource provisioning scheduler, services in figure9 where deployed using default *Kube-Scheduler(KS)*, Random Scheduler(RS) and Network-Aware Scheduler(NAS) as shown in figure10. It can be seen in figure10 that both *Kube-Scheduler* and Random Scheduler performs poorly by not taking the deploy location factor into account this leads to increased network latency[6]. Taking an example of

Service Name	Pod Name	CPU Req/Lim (m)	RAM Req/Lim (Mi)	Min. Bandwidth (Mbit/s)	Replication Factor	Target Location	Dependencies
Birch	birch-api birch-cassandra	100/500 500/1000	128/256 1024/2048	2.5 5	4 3	A	birch-cassandra birch-api
Robust	robust-api	200/500	256/512	2	4	B	none
Kmeans	kmeans-api kmeans-cassandra	100/500 500/1000	128/256 1024/2048	2.5 5	5 3	C	kmeans-cassandra kmeans-api
Isolation	isolation-api	200/500	256/512	1	2	D	none

Figure 9: Smart city deployed services[6]

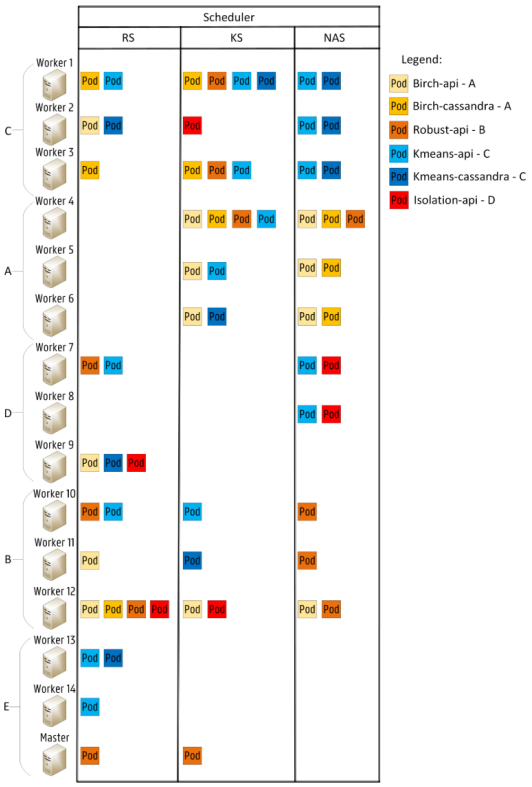

Figure 10: Service deployment using different scheduling techniques[6]

Scheduler	Extender decision	Scheduling decision	Binding operation	Pod Startup Time
KS	-	2.14 ms	162.7ms	2.02 s
RS	5.32 ms	7.71 ms	178.2ms	3.04 s
NAS	4.82 ms	6.44 ms	173.1ms	2.10 s

Figure 11: Processing time of different scheduling techniques[6]

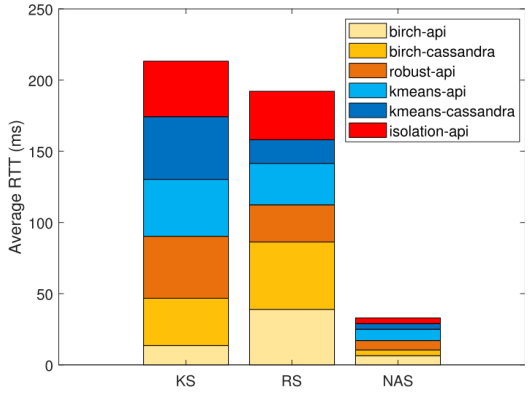

Figure 12: Average RTT of service deployment using different scheduling techniques[6]

"*Isolation-api*", whose desired deploying position was Location D, whereas it was not deployed properly by both *Kube-Scheduler* and Random Scheduler[6]. Figure11 shows the processing time of different schedulers for deploying *pods*. *Kube-Scheduler* on average takes 2.14ms for taking scheduling decisions and in case of Random Scheduler and Network-aware Scheduler takes around 7.71ms and 6.44ms respectively[6]. This added delay is due to external process call as discussed in section3. After scheduling decision, for resource provisioning and starting *pod* containers on *worker node* time taken by *Kube-Scheduler* and Network-Aware Scheduler was average 2 seconds whereas for Random schedulers it was 3 seconds[6]. Further figure10 shows that using *Kube-Scheduler* and Random Scheduler overloads the *worker node* interms of bandwidth requirement as defined bandwidth per *worker node* is 10Mbit/s[6]. For example *Kube-Scheduler* deploys 4 *pods* on *worker node*1 and 4 exceeding the bandwidth of *worker nodes* this leads to service interuption due to bandwidth[6]. This issue was re-

solved when using Network-Aware Scheduler as it allows add minimum bandwidth requirement using "*bandwidthReq*" in *pod* configration file. Figure12 shows the average RTT taken by different schedulers for *pod* deployments. It shows thats Network-based resource scheduling/Network-Aware Scheduler outstands the *Kube-Scheduler* and Random Scheduler by having very less RTT for each *pod* deployment[6]. For instance, average RTT for "*Isolation-api*" is 4ms when deployed using Network-Aware Scheduler and its very high for *Kube-Scheduler* and Random Scheduler that is 39ms and 34ms respectively[6]. Results show that by adding few miliseconds for Network-Aware scheduling performance can be improved for service deployment interm of network latency compare to *Kube-Scheduler*[6].

5 Comparison of Network-based Resource Provisioning Solutions

- Compare different solutions based on the following criteria:

5.1 Orchestrator

- write about the differences between Kubernetes(main-paper)[6] and other available cloud solutions such as Fogernetes[7] and [5].

5.2 Resource Provisioning Techniques

- difference between different resource scheduling techniques such as [8], [4] etc.

6 Conclusion

7 Further Research Topics

- after writing the seminar, if there is any improvement that can be done, will be added in this section.

References

[1] *Container Journal - What is an Orchestrator?* URL: https://containerjournal.com/features/whats-orchestrator-orchestrators-arent/.

[2] *OpenStack - Orchestration service overview*. URL: https://docs.openstack.org/mitaka/install-guide-rdo/common/get_started_orchestration.html.

[3] *Production-Grade Container Orchestration - Kubernetes*. URL: https://kubernetes.io/.

[4] D. Haja, M. Szalay, B. Sonkoly, G. Pongracz, and L. Toka. "Sharpening Kubernetes for the Edge". In: https://dl.acm.org/doi/10.1145/3342280.3342335. SIGCOMM 2019 - Proceedings of the 2019 ACM SIGCOMM Conference Posters and Demos, Part of SIGCOMM 2019, 2019.

[5] A. Reale, P. Kiss, M. Tóth, and Z. Horváth. *Designing a decentralized container based Fog computing framework for task distribution and management*. Tech. rep. http://www.naun.org/main/UPress/cc/2019/a022012-044.pdf. 2019.

[6] J. Santos, T. Wauters, B. Volckaert, and F. De Turck. "Towards network-Aware resource provisioning in kubernetes for fog computing applications". In: http://physics.nist.gov/Document/sp811.pdf. IEEE Conference on Network Softwarization Unleashing the Power of Network Softwarization, NetSoft 2019, 2019.

[7] C. Wöbker, A. Seitz, H. Mueller, and B. Bruegge. "Fogernetes: Deployment and management of fog computing applications". In: https://ieeexplore.ieee.org/document/8406321. IEEE/IFIP Network Operations and Management Symposium: Cognitive Management in a Cyber World, NOMS 2018, 2018.

[8] L. F. Bittencourt, J. Diaz-Montes, R. Buyya, O. F. Rana, and M. Parashar. "Mobility-Aware Application Scheduling in Fog Computing". In: (2017). https://ieeexplore.ieee.org/document/7912261.