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

Abdul Ahad Ayaz

Summer term 2020

Abstract - Internet of Things (IoT) is emerging rapidly in recent years, IoT applications are producing huge amount of heterogeneous data. This can lead to disaster as IoT applications have minimum computation power. Fog Computing resolves this issue by providing not only computational resources but also proper management and Orchestration for these IoT applications. But providing proper resource scheduling for these applications is still a challenge in Fog Computing. This seminar paper discusses the solution available for proper scheduling of applications based on network resources such as latency, bandwidth, etc. Kubernetes is used as Fog Computing infrastructure and applications are deployed as *pod* (set of containers). Default Kubernetes scheduler only considered the computational resources and does not consider the network resources which is important in the case of data-sensitive applications. Results show that network-based resource provisioning performs better than the default scheduling technique. Detail comparison is provided between using network-based resource provisioning and other available solutions in the market.

1 Introduction

In recent years with the evolution of technology, the Internet of Things (IoT) devices are increasing day by day. According to Ericsson mobility report[], there will be a 17% (approx. 22.3 billion) increase in IoT devices by 2024. Functionally IoT is

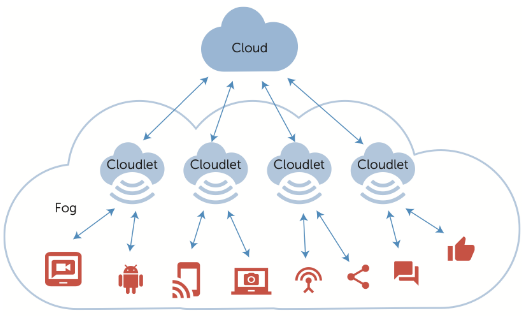


Figure 1: Fog computing: Top-level overview[10]

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 enable many devices connected to the network and generate a lot of heterogeneous data also known as BigData which requires special data processing models and Infrastructure support. Processing BigData required a lot of resources and cloud computing theoretically provides it unlimited resources[**fog-comp-survey**]. But there is a downside of using cloud computing for such complex computation as it is more costly when it comes to computation power, storage, and bandwidth. Computation needs to be performed at the node level and only the aggregated data need to send to the central node for further computations and analysis. This de-centralized approach will save a lot 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 edge of the network instead of central core.

Fog computing is defined as *"An architecture 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 the cloud access the cloud core network through access points for data exchange to fetch data from data-centers [10]. In fog computing these access points also serves as resource providers such as computation power and storage etc. and are called "cloudlets"[10]. Figure 1 show the top-level architecture 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. An alternate is to use the Containers such as Docker which is light-weight, requires fewer 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 resources 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 the deploying application is passed to Kubernetes, it checks for the availability of resources and deploys afterward. Kubernetes default resource scheduler monitors 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 the data-centric application. An example of a data-centric application is a weather forecast that receives data from scattered IoT devices and provide prediction. If the data is lost or delayed due to higher latency and poor bandwidth, timely decisions cannot be made that lead to disaster. To overcome this drawback of Kubernetes, the author proposed an alternate Kubernetes scheduler that considers network resources along with computational resources.

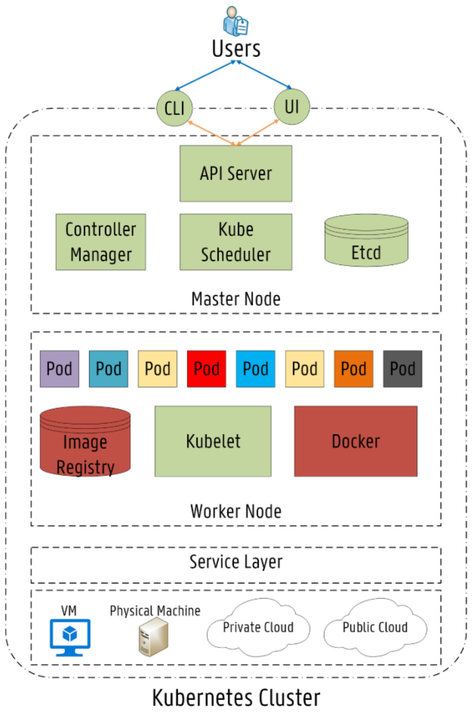


Figure 2: Kubernetes Cluster: Main Components[7]

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* [7]. *worker nodes* can be either physical or virtual resources 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 communicate with the 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 components of Kubernetes, which maintains the state across the cluster for synchronization of data. *Control Manager* is responsible for monitoring of *Etcd*. For any state change of cluster, *Control Manager* forwards the new state request using *API server*. *Kube Scheduler* is discussed later in section 2.3. On *worker node*, node agent known as *Kubelet* which is responsible 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 responsible for managing the images required to create the container applications. *Pod* is the main component of *worker node* where all the applications are deployed. Single *Pod* represents the application that consists of multiple containers based on the services of the 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 [7]. Containers communicate using different ports, hence there is a limitation to this approach as two containers listening on same port cannot be in same *Pod*[7].

2.2 Kubernetes as Orchestrator

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

- Starting or stopping of different applications.
- Ensure the Scalability of application for high usage demands.
- Management of load across different nodes to avoid resource overhead.
- Monitoring the 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* orchestrates the application across various *worker node* based on resource availability.

2.3 Kubernetes Resource Provisioning

When the user provides the configuration for creating new *pod* using *Kubectl*. The *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 the *pod* is deployed by passing through following steps: *node filtering* and *node priority or scoring*[7]. In the kubernetes cluster, *worker nods* meeting the requirement of *ped* are called *feasible nodes*[3].

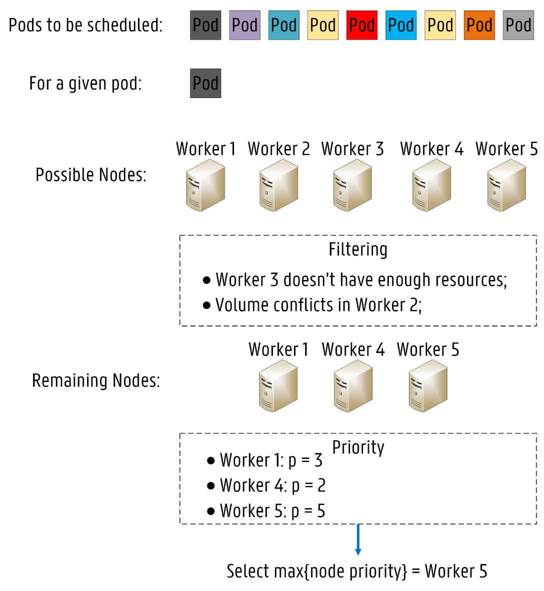


Figure 3: Kubernetes Scheduler: Working[7]

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 filters are also called *predicates*. Following is the list of *predicates* that are supported by *Kube-Scheduler*[3]:

1. **PodFitsHostPorts:** This filter checks the *worker node* for the ports requested by the *pod*.
2. **PodFitHost:** This filter checks for *worker node* with hostname mentioned in a *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 is already mounted.
5. **CheckNodeMemoryPressure:** This filter checks the *worker node* for the 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 allow the user to deploy the *pod* on specific *worker node*(node-affinity)[7]. Another use case of using a label is to restrict the *pod* deployment based on other *pod* already deployed on that *worker node* (pod-anti-affinity) [7]. These affinity rules are based on *Tolerations* and *Taints* which are defined as key-value pairs 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 inappropriate *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 [7]. If the list of the *feasible nodes* 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 assigns a rank to each *worker node* that passes the *node filtering* stage. These ranks/priorities sort the list of *worker nodes* based on best-fit for *pod* deployment. These priorities are set based on the 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"[7].
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 a 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 the 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[7].

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

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

3 Kubernetes Network-based Resource Provisioning

The default *Kube-Scheduler* works efficiently for resources such as CPU, Memory, and storage but does not consider the networking resource which is considered a critical resource in many use-case scenarios. Considering one application of Fog Computing such as IoT based smart cities which is a data-sensitive use case. Ensuring that no data is lost, networking resources need to be configured properly[7]. Default *Kube-Scheduler* does not check the network latency and available bandwidth for the *worker node*. In order to cater to this drawback, the author in [7] proposed a scheduler that checks for the network resources along with the default *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 [7], the author used the third approach that allows the default *Kube-Scheduler* to apply *filters* and calculate the *priority* of the *worker nodes* afterward the external scheduling process function is called. When the scheduling process is called, two function calls are generated[7], first one for the list of *worker nodes* as the result of *filtering/predicates* of *Kube-Scheduler*[7]. Second, for the list of *worker nodes* after calculating the *priority* using *Kube-Scheduler*[7]. For Kubernetes to work as Fog Computing infrastructure, "affinity/anit-affinity" rules and node labeling is used as shown in 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*}[7]. These nodes are further labeled by {*DeviceType*} based on their functionality and geographical positioning by *tains* such as {*Cloud, Fog*}[7]. These rules and node labeling will help inefficient deployment of *pods* on certain *worker nodes*. Considering the delay-sensitive application scenario, the data collecting node which is near to data processing node is taken into account due to time dependency[7]. For improving *pod* deployment based on the 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 a list of *worker nodes* from *Kibe-Scheduler* after applying *filters/predicates* and output will be the randomly picked *worker node* from the input list.

2. **Network-Aware Scheduler:** This scheduler is based on the algorithm as shown in figure6. Based on the algorithm, input will be a list of

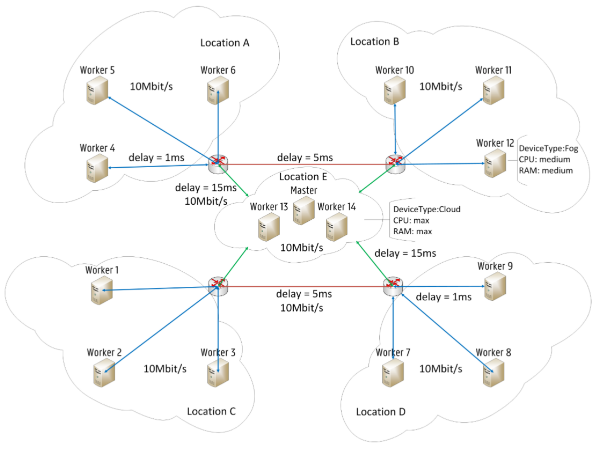


Figure 4: Kubernetes-based Fog Computing Infrastructure[7]

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[7]

worker nodes from *Kube-Scheduler* after applying *filters/predicates*. After getting the deploy location from the *pod* configuration file, this scheduler will make use of RTT labels to pick the best-fit *worker node* having minimum RTT value[7]. Apart from RTT based selection, this scheduler also look for the bandwidth label and check the *pod* configuration file for bandwidth requirement. If no bandwidth requirement is specified then the scheduler consider 250KBit/s by default and returns the *worker node* having minimum RTT and more bandwidth[7].

```
Input: Remaining Nodes after Filtering Process in
Output: Node for the service placement out
//Handle a provisioning request
handler(http.Request){
    $receivedNodes = decode(http.Request)$;
    $receivedPod = decodePod(http.Request)$;
    $node$ = selectNode(receivedNodes, receivedPod);
    return $node$
}
//Return the best candidate Node (recursive)
selectNode(receivedNodes, receivedPod){
    $targetLocation = getLocation(receivedPod)$;
    $minBandwidth = getBandwidth(receivedPod)$;
    $min = math.MaxFloat64$;
    $copy = receivedNodes$;
    // find min RTT
    for $node$ in range $receivedNodes\{$
        $rtt = getRTT(node, targetLocation)$;
        $min = math.Min(min, rtt)$;
    }
    // find best Node based on RTT and minBandwidth
    for $node$ in range $receivedNodes\{$
        if $min == getRTT(node, targetLocation)\{$
            if $minBandwidth \leq getAvBandwidth(node)\{$
                return $node$;
            }
            else
                $copy = removeNode(copy, node)$;
        }
    }
    if $copy == null$
        return $null$, $Error$("No suitable nodes found!");
    else
        return selectNode(copy, receivedPod);
}
```

Figure 6: Network-Aware scheduling Algorithm[7]

4 Performance Evaluation

In order to test the network-based resource provisioning for Fog Computing, Smart city scenario was considered [7]. This scenario collects the air quality data of Antwerp city for organic compounds in the atmosphere [7].

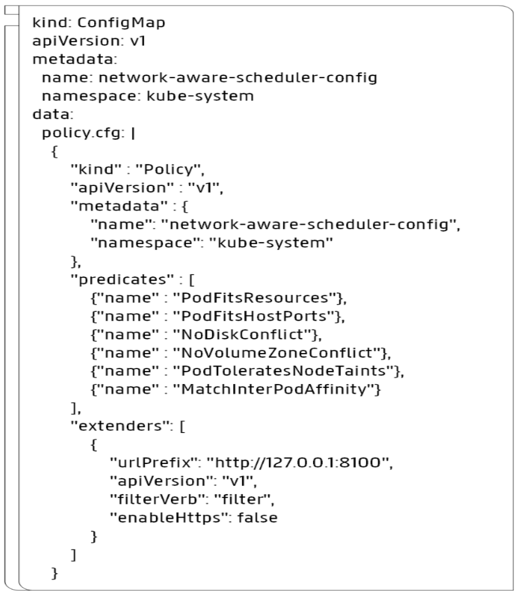
4.1 Expermentation Setup

This smart city scenario was tested using the infrastructure as shown in figure4. This infrastructure was setup at IDLab, Belgium[7]. The proposed network-based resource provisioning/network-aware scheduler was developed using Go and used as a *pod* in Kubernetes cluster[7]. Figure7 shows the *pod* configuration that consists of two containers, the first container "extender" performs the network-aware resource scheduling operations whereas the second container "network-aware-scheduler" is the *Kube-Scheduler* itself[7]. 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 the 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 parameters in the configuration file against each *pod*. The data collection of air quality was done through the algorithm and implemented as a container which is deployed as *pod* namely "birch-api"[7]. The *pod* configuration of "birch-api" is similar to one shown in figure7. Each *pod* of the service had some additional parameters in a *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, the *affinity* parameter was set to *podAntiAffinity* which limits the *pods* of same service deploying on same the *worker node*[7].

```
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[7]



```
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[7]

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 were 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[7]. Taking an example of "*Isolation-api*", whose desired deploying position was Location D, whereas it was not deployed properly by both *Kube-Scheduler* and Random Scheduler[7]. Figure11 shows the processing time of different schedulers for deploying *pods*. *Kube-Scheduler* on average takes 2.14ms for making scheduling decisions and in case of Random Scheduler and Network-aware Scheduler takes around 7.71ms and 6.44ms respectively[7]. This added delay is due to an external process calls 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[7]. Further figure10 shows that using *Kube-Scheduler* and Random Scheduler overloads the *worker node* in terms of bandwidth requirement as defined bandwidth per *worker node* is 10Mbit/s[7]. 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[7]. This issue was resolved when using Network-Aware Scheduler as it allows add minimum bandwidth requirement using "*bandwidthReq*" in *pod* configuration 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[7]. 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[7]. Results show that by adding a few miliseconds for Network-Aware scheduling performance can be improved for service deployment in terms of network latency compare to *Kube-Scheduler*[7].

5 Comparison of Network-based Resource Provisioning Solutions

In this section, the Network-based resource provisioning technique is compared with the other solutions that allow resource provisioning in Fog Computing. This comparison is done based on the available orchestrator for fog computing and resource provisioning techniques.

5.1 Orchestrator

There are many orchestrators currently available, one such orchestrator is Fogernetes[9]. Fogernetes[9] platform is specially built for fog and edge applications and make use of heterogeneous devices. Fogernetes[9] has three layers namely cloud, fog, and edge. Similar to the Kubernetes-based solution[7], it also uses the labeling system to identify the layer, location, performance, and connectivity[9].

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[7]

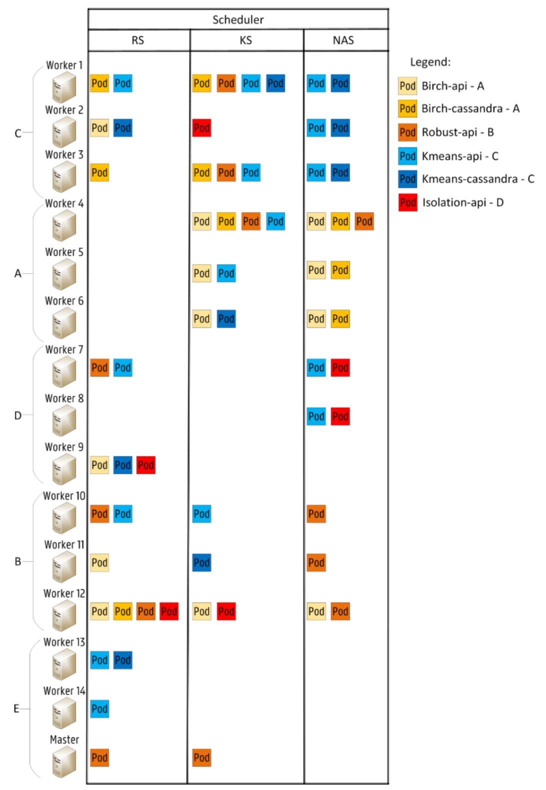


Figure 10: Service deployment using different scheduling techniques[7]

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[7]

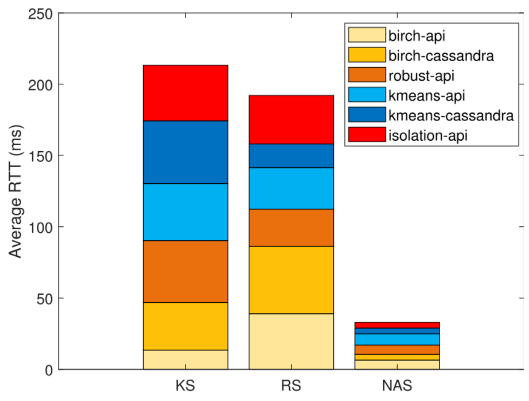


Figure 12: Average RTT of service deployment using different scheduling techniques[7]

Fogernetes[9] platform was tested against surveillance application, that collects data from multiple cameras. Kubernetes was used for the implementation of Fogernetes[9]. Three Kubernetes nodes were used one for each layer of Fogernetes[9]. Out of three nodes, 2 nodes were servers that were labeled as cloud and fog and the third node was Rasberry Pi labeled as edge. Camera works on edge layer that transfers the data to fog layer for processing and afterward it is stored in cloud layer. Monitoring of these layers can be done through the Kubernetes Dashboard or external Dashboard Grafana. Compare to the solution of base-paper[7], Fogernetes[9] tested the Fog Computing application in true manner, by deploying services on heterogeneous devices.

Another orchestrator is presented in [6] build on top of Docker Swarm. This orchestrator considers the dynamic nature of the network and automates the distribution of the services across the nodes in the Fog Computing environment[6]. This solution is based on peer-to-peer collaborative computation network that tries to map available resources, plans the deployment, move the services across the nodes, and monitoring of overall infrastructure[6]. In base-paper solution[7] Kubernetes was used as the infrastructure whereas in [6] Docker Swarm was used. One key feature of using a solution[6] is that it allows the *live migration* of service when the node is exhausted in terms of computation, network latency. This feature is currently lagging in base-paper solution[7].

5.2 Resource Provisioning Techniques

There are many resource provisioning techniques/schedulers currently available for Kubernetes-based Fog Computing infrastructure. One such scheduler is presented in [5], that deploys the services taking network delay into account. This scheduler[5] has the following features: checks for node latency periodically and updates the node label with the latest latency value, supports *pod* re-scheduling in-case of network delays. This Scheduler[5] selects the *worker node* using the "greedy heuristic algorithm". This scheduler[5] performs better than Network-Aware Scheduler[7] in two ways: first, continuously updating the latency that helps in the efficient deployment of *pods* and second the re-scheduling of the *pod* from one *worker node* to other in terms of *worker node* failure and network delay. These two features are currently not present in Network-Aware Scheduler.

DYSCO[8] is yet another scheduler for Kubernetes resource provisioning. DYSCO takes following contextual elements to for service deployment across nodes[8]: "Node Name", "Compute Domain"(cloud, fog, edge), "Location", "Computing Power", "Processor Architecture"(amd64, arm, etc.), "Memory" and "Network Connectivity"(GSM, 3G, LTE, DSL, etc.). DYSCO is also fault-tolerant in case of node failure and network breakdown[8]. Compare to Network-Aware Scheduler[7], DYSCO has additional filters such as processor type and network-connectivity type but it does not tackle the latency and bandwidth issues.

The Scheduler proposed in [4] works alongside the default *Kube-Scheduler* as an external asynchronous module. *Kube-Scheduler* performs the operations in two steps: *node filtering* and *node scoring* whereas This Scheduler[4] performs the operation in a single step and works as follows: continuously collects the real-time data such as CPU usage, available Memory, latency, packet loss, etc. for all *worker nodes*. Afterward the scheduling is applied for *pod* deployment. Scheduling algorithm consists of the following steps: the list of pending *pods* to be deployed. Re-arrange them in descending order meaning *pod* with high resource requirement comes first. As mentioned in [4], "Knapsack problem greedy approach" was used which is an optimization technique that will arrange *worker nodes* in a list based on best-fit criteria of *pod* resource requirement fulfillment. The next step, each node is checked for Liveness(status of node: busy, ready, etc.), available CPU, and available Memory[4]. Lastly, node scoring is done based on the real-time resource utilization data from the *worker node*. The *worker node* with less resource utilization value is considered the best candidate[4]. Compare to Network-Aware Scheduler[7], this scheduler[4] is a complete scheduler itself, whereas Network-aware Scheduler[7] is an extension of default *Kube-Scheduler*. The proposed scheduler in [4] is more complex but uses the optimization techniques that result in efficient scheduling.

6 Conclusion

The aim of Fog Computing is to bring computational resources nearer to the end-user with minimum network utilization. Fog Computing enabled many IoT applications but there is no proper management and orchestration. This seminar paper presented the Kubernetes-based solution for IoT applications. All the applications are deployed as *pod* consist of single or multiple Docker containers. Default Kubernetes scheduler does not consider the network utilization while deploying the *pods* which goes against the Fog Computing. In order to cater the network utilization, this seminar paper also discussed about the new scheduling technique Network-Aware Scheduler that works along with the default scheduler and allows the *pod* deployment based on network parameters such as RTT, latency, and bandwidth. The results mentioned in this seminar paper shows that the performance of Network-Aware Scheduler is far better compared to the default Kubernetes scheduler.

7 Further Research Topics

Extend the Network-Aware Scheduler to re-schedule the *pod* in case of node failure and network breakdown more efficiently. Adding more networking filters such as selecting connection types such as LAN, 3G, LTE, fiber, etc.

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] M. Chima Ogbuachi, C. Gore, A. Reale, P. Suskovics, and B. Kovács. "Context-Aware K8S Scheduler for Real Time Distributed 5G Edge Computing Applications". In: `https://ieeexplore.ieee.org/document/8903766`. 2019 International Conference on Software, Telecommunications and Computer Networks (SoftCOM), 2019.

[5] 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.

[6] 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.

[7] 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.

[8] L. Mittermeier, F. Katenbrink, A. Seitz, H. Muller, and B. Bruegge. "Dynamic scheduling for seamless computing". In: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8567371`. Proceedings-8th IEEE International Symposium on Cloud and Services Computing, SC2 2018, 2018.

[9] 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.

[10] 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`.