

# 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 theoretically 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 requirments[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 in-

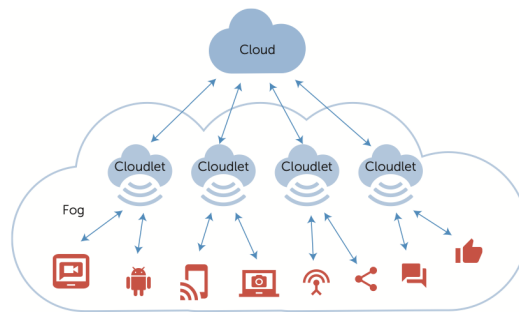


Figure 1: Fog computing: Top-level overview[8]

stead 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 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 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, Open-Stack, 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.

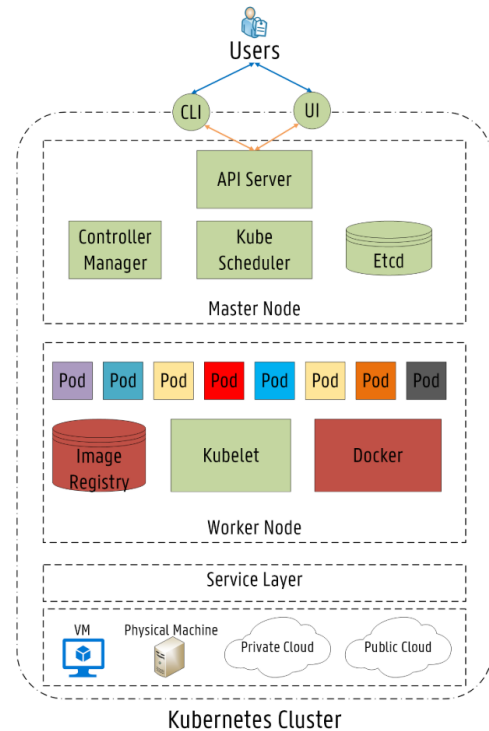


Figure 2: Kubernetes Cluster: Main Components[6]

## 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. *Etd* 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 maintains the state across the cluster for synchronization of data. *Control Manager* is responsible for monitoring of *Etd*. 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 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 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 limitation to this approach 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 a lot of human effort. As discussed in [1], Orchestrator is responsible for following:

- Starting or stopping of different applications.

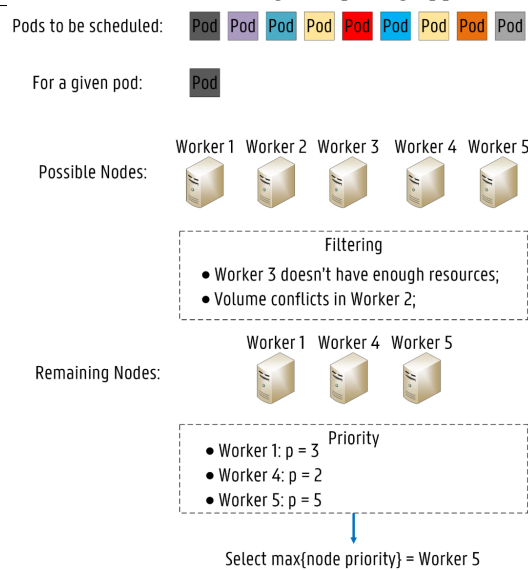


Figure 3: Kubernetes Scheduler: Working[6]

- Ensure Scalability 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 orchestration 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*. *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 nodes* meeting the requirement of *pod* 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 specific *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* configuration 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 [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*.

### 3 Kubernetes Network-based Resource Provisioning

- write about why we need network-based resource provisioning
- main factors consideration (e.g bandwidth and latency)
- workflow of network-based scheduler

## 4 Performance Evaluation

- Write about the considered use-case of Fog Computing for Evaluation

### 4.1 Experimentation Setup

- setup of Kubernetes base on the mentioned use-case of Fog Computing with diagram

### 4.2 Analysis of Kubernetes Default and Network-based Resource Provisioning

- write about the Performance difference between default Kubernetes scheduler and network based scheduler with supporting result tables and graphs

## 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](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>.