# Kubernetes Ip Hash Set for Enhanced Cpu Utilization

**Kishore Kumar Jinka[1], Dr. B. Purnachandra Rao[2]**

[1]GlobalLogic Inc VA, USA
[2]Sr. Solutions Architect, HCL Technologies, Bangalore, Karnataka, India.

**Abstract**
Kubernetes is a platform for automating the deployment, scaling, and management of containerized applications. Kubernetes automates the orchestration of containers, enabling seamless scaling, load balancing, and fault tolerance in a highly dynamic environment. In Kubernetes, iptables is commonly used to support networking functionalities like kube-proxy, Manages networking rules to route traffic to the appropriate backend pods. Service discovery and load balancing. Kubernetes uses iptables rules to direct requests for a service to the correct pod(s) based on IPs. Network policies,
IP Tables plays a key role in how networking is managed, particularly in terms of routing traffic to Pods and Services. Kubernetes uses IPTables in several key components to ensure smooth communication within the cluster and to external systems. When you create a Service, Kubernetes sets up IPTables rules to route traffic to the correct set of Pods. IP Tables use Hash Table to store rules and connections for fast lookups. Linked lists to manage chains of rules and connections, trees to optimize rule matching and bitmaps for compactly store flags and options.
IP Hashset is commonly used to optimize IPTables performance, especially in large-scale environments with extensive IP filtering needs, like Kubernetes clusters. When dealing with numerous IP addresses or IP ranges, the hashset data structure allows more efficient storage and quicker lookups than a list, especially with large datasets. Hashsets are typically implemented through ipset in Linux, which works in conjunction with iptables. Single IP hash set**s** in iptables can lead to significant challenges, especially as cluster sizes and network complexity grow. A single IP hash set can struggle to handle the scale in large Kubernetes clusters, where thousands of IP addresses are stored. It becomes more memory-intensive and slower in lookups due to higher collision rates in the hash set. As the hash set grows, search and update operations slow down, creating latency in packet processing, high cpu utilization. This affects cluster performance, as any latency in the network layer impacts the efficiency of service communication. In this paper we will address cpu utilization issue by using multi IP hash set.

**Keywords**: Kubernetes (K8S), Cluster, Nodes, Deployments, Pods, ReplicaSets, Statefulsets, Service, Service Abstraction, IP-Tables, HashTable, IP Hash Set, Single IP Hash Set, Multiple IP Hash Set.

## INTRODUCTION

Kubernetes consists of several components that work together to manage containerized applications. Master Node: This controls the overall cluster, handling scheduling and task coordination.API Server: Frontend that exposes Kubernetes functionalities through RESTful APIs. Scheduler: Distributes work across the nodes based on workload requirements..Controller Manager: Ensures that the current state matches the desired state by managing the cluster's control loops.etcd: Kube-proxy: Manages network communication within and outside the cluster.

Pod is the smallest deployable unit in Kubernetes, encapsulating one or more containers [1] with shared storage and network resources. All containers in a pod run on the same node. Namespace, these are used to create isolated environments within a cluster. They allow teams to share the same cluster resources without conflicting with each other. Deployment: A higher-level abstraction that manages the creation and scaling of Pods. It also allows for updates, rollbacks, and scaling of applications. ReplicaSet [2] ensures a specified number of replicas (identical copies) of a Pod are running at any given time. StatefulSet: Designed to manage stateful applications, where each Pod has a unique identity and persistent storage, such as databases. DaemonSet, ensures that a copy of a Pod is running on all (or some) nodes. This is useful for deploying system services like log collectors or monitoring agents. Job, A Kubernetes resource that runs a task until completion. Unlike Deployments or Pods, a Job does not need to run indefinitely. CronJob runs Jobs at specified intervals, similar to cron jobs in Linux.

## LITERATURE REVIEW

### Kubernetes Cluster

A **cluster** [3] refers to the set of machines (physical or virtual) that work together to run containerized applications. A cluster is made up of one or more **master nodes** (control plane) and **worker nodes**, and it provides a platform for deploying, managing, and scaling containerized workloads.

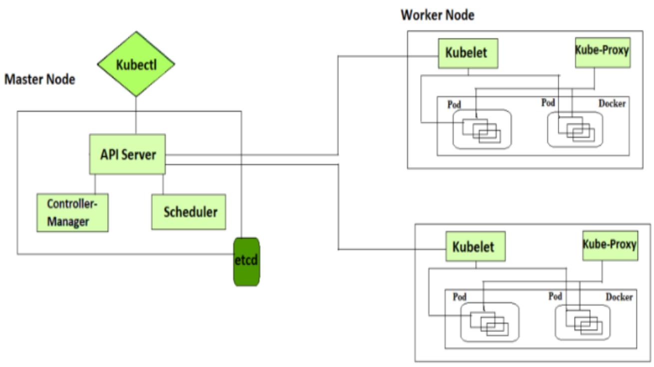



**Fig 1. Kubernetes cluster Architecture**

Client kubectl will connect to API server [4] (part of Master Node) to interact with Kubernetes resources like pods, services, deployment etc. Client will be authenticated through API server having different stages like authentication and authorization. Once the client is succeeded though authentication [5] and authorization (RBAC plugin) it will connect with corresponding resources to proceed with further operations. Etcd [6] is the storage location for all the kubernetes resources. Scheduler will select the appropriate node for scheduling the pods unless you have mentioned node affinity (this is the provision to specify the particular node for accommodating the pod). Kubelet is the process which is running on all nodes of the kubernetes cluster and it will manage the mediation between api server and corresponding node. Communication between any entity with master node is going to happen only through api server.

***Key Components of a Kubernetes Cluster:***

**Control Plane (Master Node)**:

API Server exposes Kubernetes APIs. All interactions with the cluster (e.g., deploying applications, scaling, etc.) go through the API server.

Etcd is a distributed key-value store [7] that holds the state and configuration of the cluster, including information about pods, services, secrets, and configurations.

Controller Manager ensures that the cluster's desired state matches its actual state, by managing different controllers (like deployment, replication, etc.).

Scheduler is the one which Assigns workloads to worker nodes based on resource availability, scheduling policies, and requirements.

Worker Nodes:

Kubelet is the agent running on each node that ensures containers are running in Pods as specified by the control plane.

Container Runtime [8] is the software responsible for running containers (e.g., Docker, containerd).

Kube-proxy manages network traffic between pods and services, handling routing, load balancing, and network rules.

**How a Kubernetes Cluster Works:**

Pods: The smallest deployable units in Kubernetes, consisting of one or more containers. They run on worker nodes and are managed by the control plane.

Nodes: Physical or virtual machines in the cluster that host Pods and execute application workloads.

Services: Provide stable networking and load balancing for Pods within a cluster.

***Cluster Operations:***

Kubernetes clusters can automatically scale up or down by adding/removing nodes or pods.

Resilience: Clusters are designed for high availability and can automatically restart failed pods or reschedule them on healthy nodes.

Kubernetes ensures traffic is evenly distributed across Pods within a Service.

Self-Healing: The control plane continuously monitors the state of the cluster and acts to correct failures or discrepancies between the desired and current state.

***Service Abstraction:***

Service Abstraction [9] in Kubernetes provides a way to define a logical set of Pods and a policy by which to access them. This abstraction enables communication between different application components without needing to know the underlying details of each component's location or state.

Stable Network Identity: Services provide a stable IP address and DNS name that can be used to reach Pods, which may be dynamically created or destroyed.

Load Balancing: Kubernetes services automatically distribute traffic to the available Pods, providing a load balancing mechanism. When a Pod fails, the service can route traffic to other healthy Pods.

Service Types: Kubernetes supports different types of services:

ClusterIP: The default type, which exposes the service on a cluster-internal IP. Only accessible from within the cluster.

NodePort: Exposes the service on each Node’s IP at a static port (the NodePort). This way, the service can be accessed externally.

Kubernetes automatically provisions a load balancer for the service when running on cloud providers. ExternalName maps the service to the contents of the externalName field (e.g., an external DNS name).

***Iptables Coordination:***

**Iptables** [10] is a user-space utility program that allows a system administrator to configure the IP packet filter rules of the Linux kernel firewall. In the context of Kubernetes, iptables is used to manage the networking rules that govern how traffic is routed to the various services.

| SNo | IP Address | Port |
|---|---|---|
| 1 | 10.3.4.3, 10.3.4.5,10.3.4.7 | 8125 |
| 2 | 10.3.5.3, 10.3.5.5,10.3.5.7 | 8081 |
| 3 | 10.3.6.3, 10.3.6.5,10.3.6.7 | 8080 |
| 4 | 10.3.2.3, 10.3.2.5,10.3.2.7 | 5432 |
| 5 | 10.3.7.3, 10.3.7.5,10.3.7.7 | 6212 |
| 6 | 10.3.8.3, 10.3.8.5,10.3.8.7 | 6515 |

**Table 1: IP Tables Storage Structure**

Key Functions:

Traffic Routing: Iptables rules direct incoming traffic to the correct service IP based on the defined service configurations.

NAT (Network Address Translation): Iptables can be configured to rewrite the source or destination IP addresses of packets as they pass through, which is crucial for services that need to expose Pods to external traffic.

Connection Tracking: Iptables tracks active connections and ensures that replies to requests are sent back to the correct Pod.

***Service and IP Table***:

Service Request: A request is sent to the service's stable IP address.

Kubernetes Networking [11] uses iptables to manage the routing of this request. It sets up rules to map the service IP to the IP addresses of the underlying Pods.

Load Balancing: Ip tables distributes incoming traffic among the Pods that match the service's selector, ensuring load balancing. Return Traffic: When a Pod responds, iptables ensures that the response goes back through the same network path, maintaining connection tracking.

Service abstraction in Kubernetes provides a simplified and stable interface for accessing application components, while iptables coordination ensures that the network traffic is efficiently routed to the right Pods. Together, they form a robust networking framework that is fundamental to the operation of Kubernetes clusters.

Three node , four node , five node , six node , seven node , eight node , nine node and ten node clusters have been configured with 32 CPU, 64 GB and 500GB for master node and 24 CPU , 32 GB and 350 GB for all worker nodes. The existing IP table has been implemented using IP Hash set, it is a data structure typically used in networking and firewall applications to store IP addresses using a hash table [12][13][24][25][26]. We have collected different samples for performance , scaling , network policy enforcement parameters.

```
apiVersion: v1
kind: Pod
metadata:
  name: single-iphash-metrics
spec:
  containers:
  - name: single-iphash-metrics
    image: busybox
    command: ["sh", "-c"]
    args:
    - while true; do
      kubectl top pod --all-namespaces --metrics-server;
      sleep 5;
      done
  restartPolicy: Always
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: single-iphash-metrics-config
data:
  metrics-server.config: |
    metrics:
    - iphash_cpu_usage
    - iphash_memory_usage
    - iphash_network_latency
    - iphash_packet_loss_rate
    - iphash_context_switching_rate
```

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: single-iphash-metrics-daemon
spec:
  selector:
    matchLabels:
      app: single-iphash-metrics
  template:
    metadata:
      labels:
        app: single-iphash-metrics
    spec:
      containers:
      - name: single-iphash-metrics
        image: busybox
        command: ["sh", "-c"]
        args:
        - while true; do
          kubectl get --raw /apis/metrics.k8s.io/v1beta1/nodes/${NODE_NAME
          }/metrics | jq '.metrics[] | select(.name == "iphash_cpu_usage")';
          sleep 5;
          done
        env:
        - name: NODE_NAME
          valueFrom:
            fieldRef:
              fieldPath: spec.nodeName
```

```
kubectl get --raw /apis/metrics.k8s.io/v1beta1/nodes/${NODE_NAME}
/metrics | jq '.metrics[] | select(.name == "iphash_cpu_usage")'
```

To retrieve these metrics manually, you would run the above command.

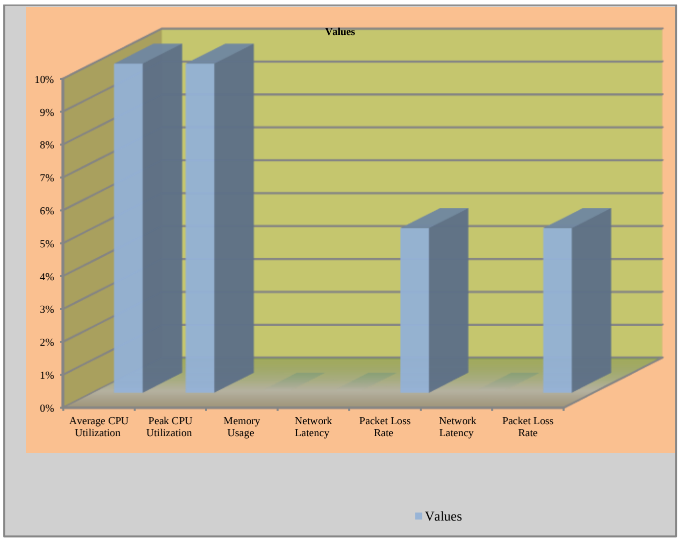
We have deployed the pod having the busy box container, defined all the metrices at configMap and deployed the same. And then DaemonSet to collect IP hash-related CPU usage from each node. This enables node-specific metric collection, which can be useful for monitoring IP tables or custom CPU metrics related to IP sets. DaemonSet runs the same Pod on every node in the cluster, ensuring node-level metrics are gathered. The args define a while loop that, every 5 seconds, fetches node metrics in raw JSON format and filters for the metric named iphash_cpu_usage using jq. The NODE_NAME environment variable dynamically retrieves the name of the node on which the Pod is running.

We have deployed daemon set at default namespace, it is getting accommodated at all nodes in the cluster. It doesn't matter about dynamic increase or decrease of the number of nodes. We have taken the busybox image and ran the commands inside the docker container and invoking the single IPHash set and collected cpu usage, memory usage , network latency, packet loss rate and context switching rate. We can collect all these metrics using the command which we are running inside the container.

| Metric | Values |
|---|---|
| Average CPU Utilization | 40% |
| Peak CPU Utilization | 70% |
| Memory Usage | 2.5 GB |
| Network Latency | 50 ms |
| Packet Loss Rate | 5% |

**Table 2: Single IP Hash Set-1**

Table 2 shows the parameters like average cpu utilization , peak cpu utilization, memory usage, network latency , and packet loss rate. We need to collect the statistics using the code which we have provided previously. Just for the sake of collecting the metrics , I have provided all the metrics at only one table. Actually all these parameters are having different units.

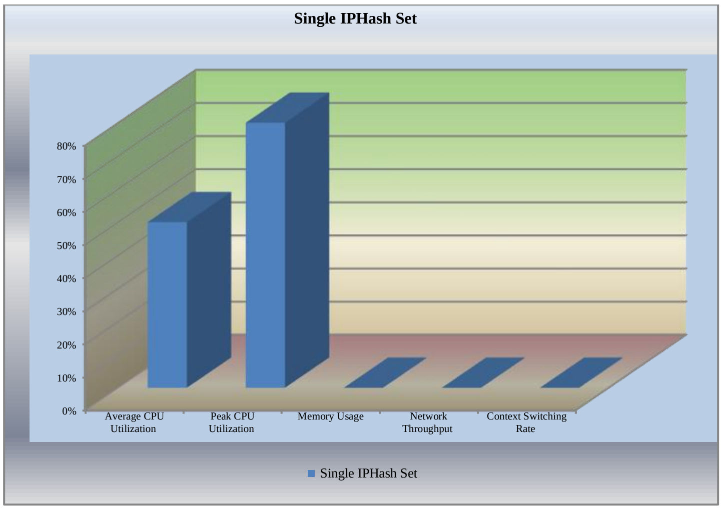



**Graph 1: Single IP Hash Set-1**

| Metric | Values |
|---|---|
| Average CPU Utilization | 50% |
| Peak CPU Utilization | 80% |
| Memory Usage | 4 GB |
| Network Throughput | 10 Gbps |
| Context Switching Rate | 1000/s |

**Table 3: Single IP Hash Set-2**

Table 3 shows the parameters like average cpu utilization , peak cpu utilization, memory usage, network latency , and packet loss rate. We need to collect the statistics using the code which we have provided previously. Just for the sake of collecting the metrics , I have provided all the metrics at only one table. Actually all these parameters are having different units. Average CPU utilization is 50% , Peak CPU utilization is 80%, Memory usage is 4GB , Network throughput is 10 Gbps and context switching rate is 1000 rules per second.

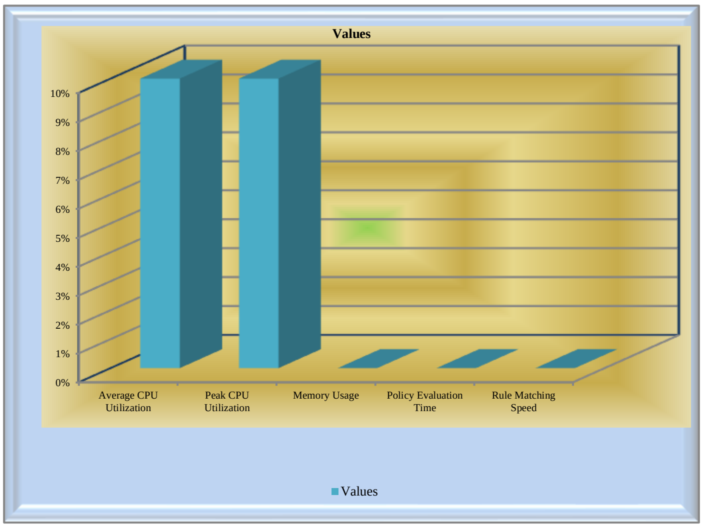



**Graph 2: Single IP Hash Set-2**

| Metric | Values |
|---|---|
| Average CPU Utilization | 35% |
| Peak CPU Utilization | 60% |
| Memory Usage | 1.8 GB |
| Policy Evaluation Time | 500 µs |
| Rule Matching Speed | 1000 rules/s |

**Table 4: Single IP Hash Set-3**

Table 4 shows the parameters like average cpu utilization , peak cpu utilization, memory usage, network latency , and packet loss rate. We need to collect the statistics using the code which we have provided previously. Just for the sake of collecting the metrics , I have provided all the metrics at only one table. Actually all these parameters are having different units. Average CPU utilization is 35% , Peak CPU utilization is 60%, Memory usage is 1.8GB , policy evaluation time is 500 us and rule matching speed is 1000 rules per second.

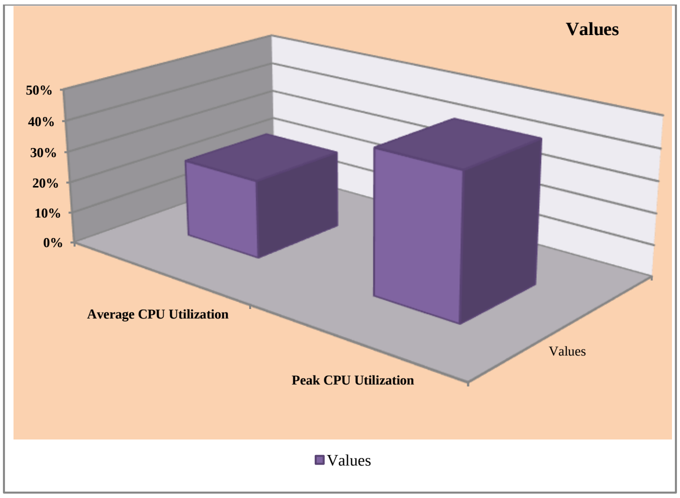



**Graph 3: Single IP Hash Set-3**

| Metric | Values |
|---|---|
| Average CPU Utilization | 25% |
| Peak CPU Utilization | 45% |
| Memory Usage | 1.2 GB |
| Average Response Time | 50 ms |
| Pod Scaling Time | 5 min |

**Table 5: Single IP Hash Set-4**

Table 5 shows the parameters like average cpu utilization , peak cpu utilization, memory usage, network latency , and packet loss rate. We need to collect the statistics using the code which we have provided previously. Just for the sake of collecting the metrics , I have provided all the metrics at only one table. Actually all these parameters are having different units. Average CPU utilization is 25% , Peak CPU utilization is 45%, Memory usage is 1.2GB , average Response Time is 50 ms and pod scaling time is 5 mins.

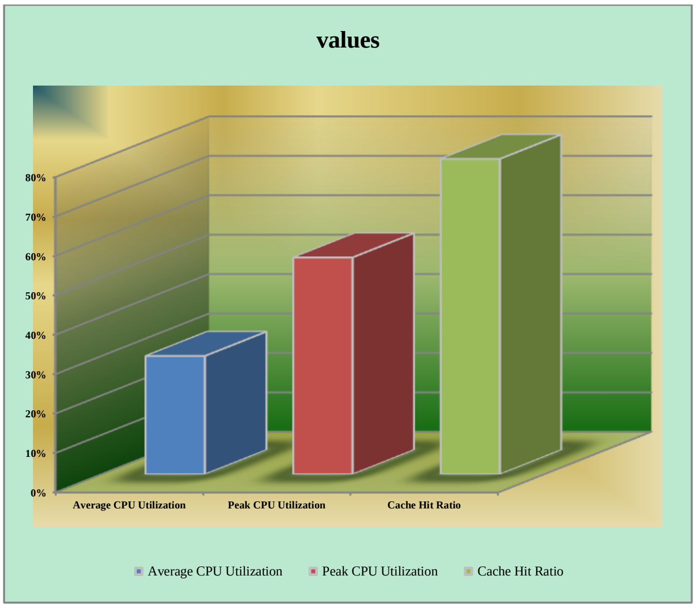



**Graph 4: Single IP Hash Set-4**

| Metric | Values |
|---|---|
| Average CPU Utilization | 30% |
| Peak CPU Utilization | 55% |
| Memory Usage | 800 MB |
| Network Bandwidth | 100 Mbps |
| Cache Hit Ratio | 80% |

**Table 6: Single IP Hash Set-5**

Table 6 shows the parameters like average cpu utilization , peak cpu utilization, memory usage, network latency , and packet loss rate. We need to collect the statistics using the code which we have provided previously. Just for the sake of collecting the metrics , I have provided all the metrics at only one table. Actually all these parameters are having different units. Average CPU utilization is 30% , Peak CPU utilization is 55%, Memory usage is 800 MB , Network Bandwidth 100 Mbps and cache hit ratio is 80%.

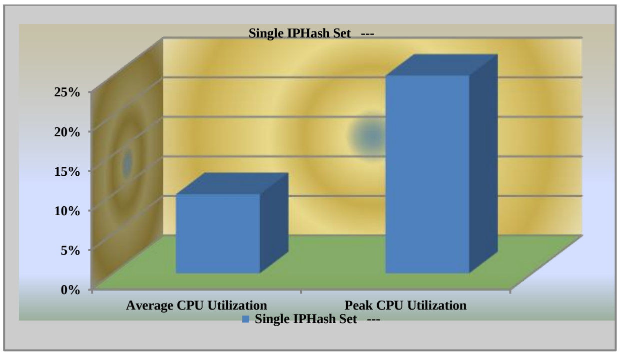



**Graph 5: Single IP Hash Set-5**

| Metric | Values |
|---|---|
| Average CPU Utilization | 10% |
| Peak CPU Utilization | 25% |
| Memory Usage | 400 MB |
| Average Latency | 20 ms |
| Packet Processing Rate | 500 packets/s |

**Table 7: Single IP Hash Set-6**

Table 7 shows the parameters like average cpu utilization , peak cpu utilization, memory usage, network latency , and packet loss rate. We need to collect the statistics using the code which we have provided previously. Just for the sake of collecting the metrics , I have provided all the metrics at only one table. Actually all these parameters are having different units. Average CPU utilization is 10% , Peak CPU utilization is 25%, Memory usage is 400 MB , Average Latency 20ms and packet processing Rate 500 packets.

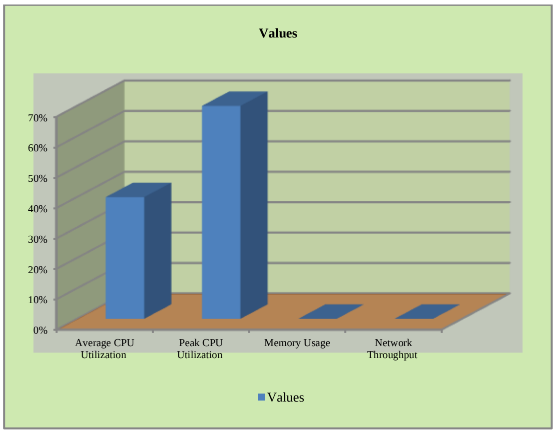



**Graph 6: Single IP Hash Set-6**

| Metric | Values |
|---|---|
| Average CPU Utilization | 40% |
| Peak CPU Utilization | 70% |
| Memory Usage | 2 GB |
| Network Throughput | 5 Gbps |
| Context Switching Rate | 500/s |

**Table 8: Single IP Hash Set-7**

Table 8 shows the parameters like average cpu utilization , peak cpu utilization, memory usage, network latency , and packet loss rate. We need to collect the statistics using the code which we have provided previously. Just for the sake of collecting the metrics , I have provided all the metrics at only one table. Actually all these parameters are having different units. Average CPU utilization is 40% , Peak CPU utilization is 70%, Memory usage is 2Gb , Network throughput is 5Gbps, and context switching rate is 500 per second.




**Graph 7: Single IP Hash Set-7**

## PROPOSAL METHOD

### Problem Statement

Service abstraction is using IP tables to store the rules of services and provide matching to the incoming request to the IP tables. The existing IP tables have been implemented using Single IP Hash Set for storing the IP addresses, It can struggle to handle the scale in large Kubernetes clusters, where thousands of IP addresses are stored. It becomes more memory-intensive and slower in lookups due to higher collision rates in the hash set. As the hash set grows, search and update operations slow down, creating latency in packet processing. This affects cluster performance, as any latency in the network layer impacts the efficiency of service communication and cpu utilization. To resolve all these issues we need to work on the sets of the IP Hash set.

### Proposal

Multiple IP Hash Set is the one which we can use to overcome the issues like faster lookup times, multiple IP Hash sets reduce lookup times , improving network performance. Smaller IPHash sets minimize latency , ensuring faster packet processing. Multiple smaller IPHash sets consume less memory. Processing smaller IPHash sets decreases CPU utilization

| SNo | IP Address |
|---|---|
| 1 | 10.3.4.3, 10.3.4.5,10.3.4.7 |
| 2 | 10.3.5.3, 10.3.5.5,10.3.5.7 |
| 3 | 10.3.6.3, 10.3.6.5,10.3.6.7 |
| 4 | 10.3.2.3, 10.3.2.5,10.3.2.7 |
| 5 | 10.3.7.3, 10.3.7.5,10.3.7.7 |
| 6 | 10.3.8.3, 10.3.8.5,10.3.8.7 |

**Table 9: Single IP Hash Set Storage**

| | | | |
|---|---|---|---|
| **000** | 10.3.4.3, 10.3.4.5 | **110** | 10.3.4.3, 10.3.4.5 |
| **001** | 10.3.6.3, 10.3.6.6 | **111** | 10.3.6.3, 10.3.6.6 |
| **010** | 10.3.6.7, 10.3.6.9 | **1000** | 10.3.6.7, 10.3.6.9 |
| **011** | 10.3.6.6, 10.3.6.0 | **1001** | 10.3.6.6, 10.3.6.0 |
| **100** | 10.3.6.1, 10.3.6.2 | **1010** | 10.3.6.1, 10.3.6.2 |
| **101** | 10.3.6.11, 10.3.6.12 | **1011** | 10.3.6.11, 10.3.6.12 |

**Table 10: Multiple IP Hash Set Storage**

Table 9 shows the storage of IP addresses at IP Tables. In this it uses the Single IP Hash set to access the IP Addresses.

Table 10 shows the storage of IP addresses at IP Tables. In this it uses the Single IP Hash set to access the IP Addresses. In the multi IP Hash set storage process , as per the octal representation the addresses will be stored at each tag. Based on the tag we can access multiple ip addresses at the same location. That is why we will get high performance by using this storage.

## IMPLEMENTATION

Three node , four node , five node , six node , seven node , eight node , nine node and ten node clusters have been configured with 32 CPU, 64 GB and 500GB for master node and 24 CPU , 32 GB and 350 GB for all worker nodes.

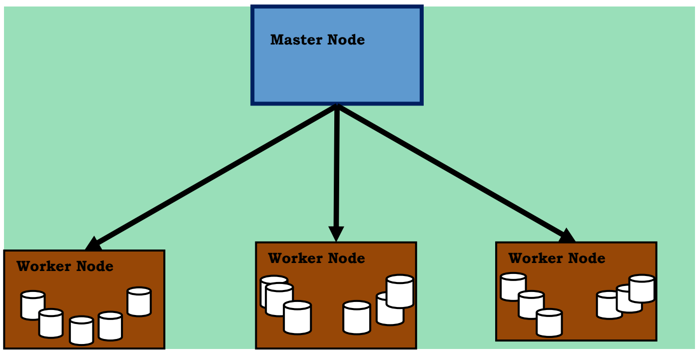



**Fig. 4. Four Node Cluster One Master and Three worker Nodes.**

Fig. 4 shows the four node cluster, one node is the master node and the remaining three are the worker nodes. Master node will have control plane and all other kubernetes core libraries toi manage the cluster. Each node in the cluster having the kubelet process , this is the agent at all the machines which is taking care of connecting with other nodes. Docker and containerd are running at each machine along with kubelet agent. Kube proxy the process which is available at all machines to manage the IP Tables. Kubelet is responsible for managing the node health status and reporting to master node.
API server is available at master node (Control Plane) and it is the point of contact between worker nodes and other components of the control plane. When ever kuberenetes client want to do to some operation at Master it will send request to API server. This will validate the request by authenticating the client and verifies the authorization of the operation what the client wants to do at the cluster or node level.
Once the authentication is successful It will work with etcd to do the expected operation. If it is update of the existing manifest file It will update the copy of the file and stores at etcd. Etcd is the key value store , it is consistent data store for kubernetes cluster. If Kuberbetes cluster client wants to delete pod from the specific namespace it will get triggered to API server. API server will authenticate the client , if it is successful then it will verify that the client is having necessary permissions to delete the pod in that namespace. If both are successful the pod will get deleted from the namespace and parallelly it will get updated at ectd datastore. Please find the API lifecycle at the Fig. 4.

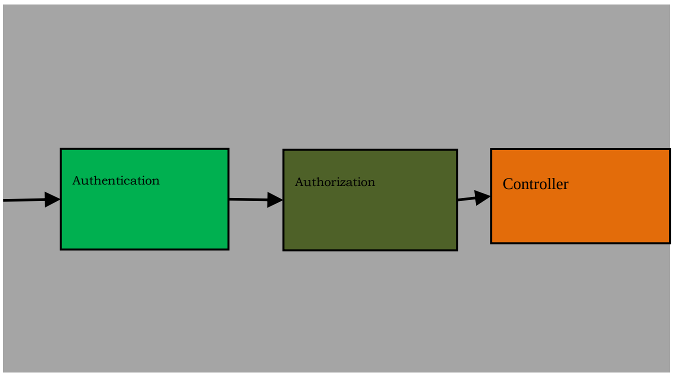



**Fig. 5. API Server Life Cycle**

Fig. 5 ,6 , 7 and 8 shows the clusters for five node . six node , seven node and eight nodes.

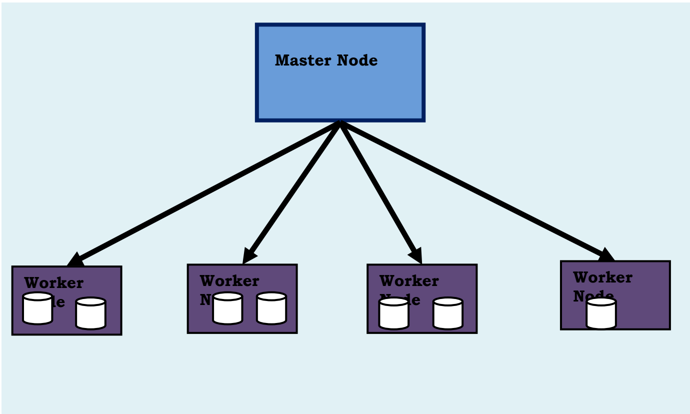



**Fig. 6. Five Node Cluster One Master and Four worker Nodes.**

Pod will get deployed to specific node if there is any node affinity enabled , or else it will get scheduled to any node based on the scheduling algorithm used by the scheduler. Container network interface is the library which will take care of assigning the IP address to pod based on allowable ips from the specific node ips. Ecah node is having different range of ips , and it will get managed by CNI [15]. Flannel is the plugin from the CNI which we have used to implement this functionality. Calico is one more alternative for flannel which we can use. As soon as pods gets deployed to node , kubelet starts reporting to control plane on the health status of the pod.

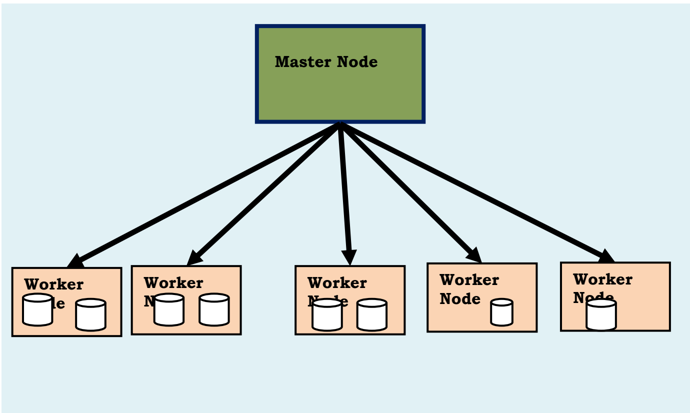



**Fig. 7. Six Node Cluster One Master and Five worker Nodes.**

If we are not defining any storage location to pod , it will get automatic storage inside the container. But the data will get lost for each restart of the pod. This is the reason we can have number of storage classes , where we can attach the volume from the local disk to container. What ever the files we are having at local to node , they will get exposed to container. Changes will get reflected automatically if we do something at the local files. Converse of this is always true.

If there any environment parameters [16] [22] [30] , we can pads them through env section of the pod manifest files. If there are any changes in the parameters we need to redeploy the pod for each update in the manifest files. To avoid this type of overhead we can deploy them using the configMap object of the kuberentes. This is what is called separation [17] of the parameters from the manifest files. We can do the changes at parameters independent of the pod deployment. The changes will get reflected automatically without having to redeploy the pod.

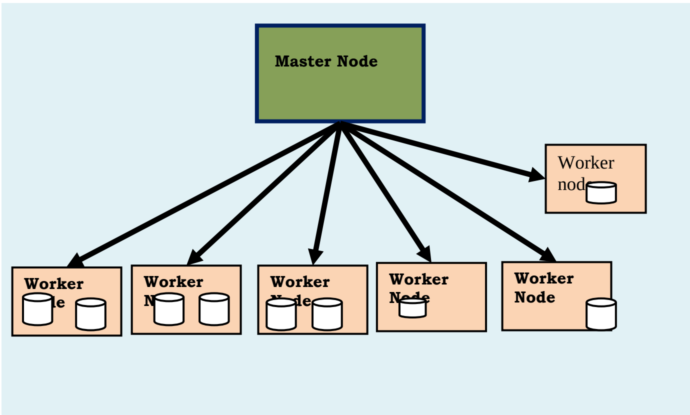



**Fig. 8. Seven Node Cluster One Master and Six worker Nodes.**

We have different types of volumes [18] [21] which we can attach to pod. Need to create the volume (folder) at the node where the pod is getting scheduled. Using Node affinity we can schedule the pod in the expected location. If there is any chance of mismatch in the pod schedule , this architecture will not workout. We can use dynamic volume creation if there is any deployment in production and if we doesn't have access to prod location.

The volume gets created automatically as soon as we deploy the pod. Container Storage Interface [19] [20] [21] will take care of creation of volumes.

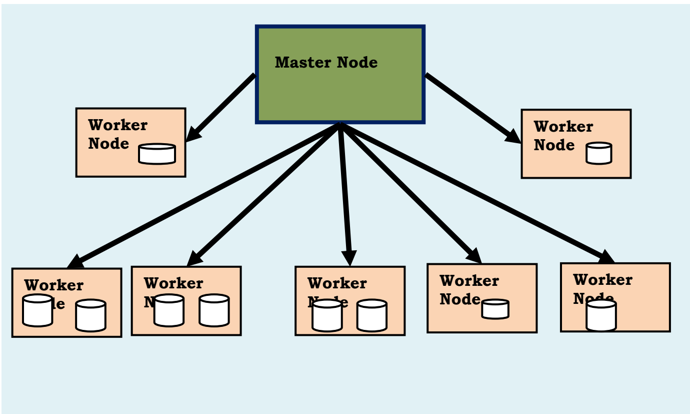



**Fig. 9. Eight Node Cluster One Master and Seven worker Nodes.**

We can connect github location files as well to container using the volume mount plugin in yaml file. We can manage the pod to pod communication using the service abstraction. Since the pod ip is ephermal we need to use service abstraction to connect to pod.

Number of nodes in the cluster is no way related to size of the IP table, but if the number of services , ingress controllers are high in count , it will directly proportional to size of the IP Table. We have three types of probes in Kubernetes liveness probe , readiness probe and startup probe. First one checks if the application is still running, second one checks if the application is ready to server the traffic and last one checks if the application has started properly.

We have configured different sizes of cluster and with different configurations on volumes like hostPath, gitRepo, emptyDir, nfs.

If there are number of pods working of interconnected functionalty like one pod is working on calculation , second pod is collecting the info from the first pod, where as third pod needs to record the log files. In this each pod needs to have access to another pods storage location or volume.

In this case instead of using the volume at each node , it would be better to define the volume at master location and make it available at all nodes in the cluster. This is what is called Network File System sharing mechanism. We have implemented this service as well.

The size of the IP Table depends on the number of services , as well as the number of pods , network policies , and ingress rules in the cluster irrespective of what is the implementation we are using for IP Tables [26] [27] [28] [29][30].

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: iphash-metrics
spec:
  containers:
  - name: iphash-metrics
    image: busybox
    command: ["sh", "-c"]
    args:
    - while true; do
      kubectl top pod --all-namespaces --metrics-server;
      sleep 5;
      done
  restartPolicy: Always
```

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: iphash-metrics-config
data:
  metrics-server.config: |
    metrics:
    - iphash_cpu_usage
    - iphash_cpu_utilization
```

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: iphash-metrics-daemon
spec:
  selector:
    matchLabels:
      app: iphash-metrics
  template:
    metadata:
      labels:
        app: iphash-metrics
    spec:
      containers:
      - name: iphash-metrics
        image: busybox
        command: ["sh", "-c"]
        args:
        - while true; do
          kubectl get --raw /apis/metrics.k8s.io/v1beta1/nodes/${NODE_NAME}/
          metrics | jq '.metrics[] | select(.name == "iphash_cpu_usage")';
          sleep 5;
          done
        env:
        - name: NODE_NAME
          valueFrom:
            fieldRef:
              fieldPath: spec.nodeName
```

```
kubectl get --raw /apis/metrics.k8s.io/v1beta1/nodes/${NODE_NAME}/
metrics | jq '.metrics[] | select(.name == "iphash_cpu_usage")'
```

```yaml
          # Time Deletion
          start_deletion=$(date +%s%3N)
          ipset del iphashset_groupA 10.0.1.1
          end_deletion=$(date +%s%3N)
          deletion_time=$((end_deletion - start_deletion))
          echo "Deletion Time: $deletion_time ms" >> /metrics/iphash_metrics.log

          # Time Policy Evaluation (using iptables)
          start_policy=$(date +%s%3N)
          iptables -C INPUT -m set --match-set iphashset_groupA src -j ACCEPT -exist
          end_policy=$(date +%s%3N)
          policy_time=$((end_policy - start_policy))
          echo "Policy Evaluation Time: $policy_time ms" >> /metrics/iphash_metrics.log

          # Keep the container running for logging
          sleep infinity
        volumeMounts:
          - name: metrics-volume
            mountPath: /metrics
        resources:
          requests:
            cpu: 100m
            memory: 50Mi
        volumeMounts:
        - mountPath: /lib/modules
          name: modules
          readOnly: true
      volumes:
      - name: metrics-volume
        emptyDir: {}
      - name: modules
        hostPath:
          path: /lib/modules
      hostNetwork: true
      hostPID: true
```

This setup will allow each node to maintain IP hash sets using ipset, useful for managing groups of IP addresses for access control or other network policies. This DaemonSet runs a pod on each Kubernetes node, setting up and managing IP hash sets directly on each node's network stack. Using ipset create, it creates two hash sets: iphashset_groupA and iphashset_groupB.

The -exist option ensures idempotency, avoiding errors if the set already exists. IP addresses are added to each hash set to specify allowed addresses for each group. iptables rules are added for each hash set, allowing traffic from IPs in iphashset_groupA and iphashset_groupB. hostNetwork: true ensures the iptables and ipset configurations apply to the node directly. The privileged: true security context grants the necessary permissions.

We have deployed the pod having the busy box container, defined all the metrices at configMap and deployed the same. After deploying the daemonset , it deploys the pods on each node. After applying, you can check the logs for successful execution or troubleshoot any issues using kubectl log command. This approach allows efficient, node-specific IP-based traffic management and ensures a scalable, maintainable setup across Kubernetes nodes.

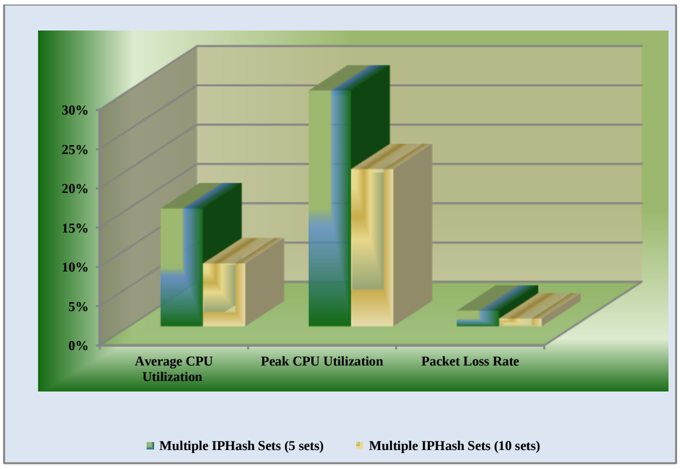
It doesn't matter about dynamic increase or decrease of the number of nodes. We have taken the busybox image and ran the commands inside the docker container and invoking the multiple IPHash set and collected CPU utilization , peak cpu utilization time , memory usage , policy evaluation time,

network throuput , context switching rate , network latency, cache hit ratio. These parameters are getting at all machines in the corresponding log files.
We have collected all these metrics and formatted graphs and tables in the following sections to get the better view of the results and to finalize on the paper discussion.
Please find the different parameters which we had using the previous code.

| Metric | Multiple IPHash Sets (5 sets) | Multiple IPHash Sets (10 sets) |
|---|---|---|
| Average CPU Utilization | 15% | 8% |
| Peak CPU Utilization | 30% | 20% |
| Memory Usage | 1.2 GB | 800 MB |
| Network Latency | 20 ms | 10 ms |

**Table 11: Multiple IP Hash Set-1**

Table 11 shows the parameters like Average CPU utilization, Peak CPU Utilization, Memory Usage and Network latency . Avg CPU utilization is 15% at 5 sets configuration where it is 8% at multiple IPHash sets 10 sets configuration. Peak CPU Utilization is 30% at 5 sets configuration and it is 20% at 10 sets configuration. Memory usage is 1.2GB at 5 sets configuration and it is 800 MB at 10 sets configuration. Network latency is 20 ms at 5 sets configuration and it is 10 ms at 10 sets configuration.




**Graph 8**: **Multiple IP Hash Set-1**

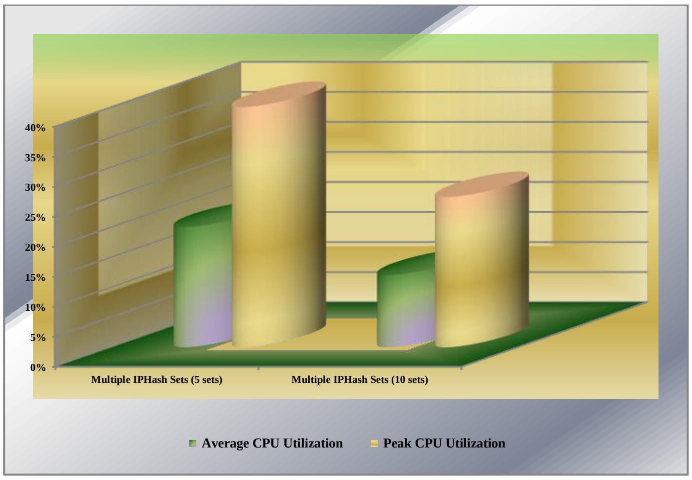
Graph 8 clearly exhibits that the average CPU utilization , peak CPU utilization and Packet loss ratio will be decreased by increasing the number of sets in the IP Hashset configuration. Different shades in the bar representing the values is no way related to volume of the value.

| Metric | Multiple IPHash Sets (5 sets) | Multiple IPHash Sets (10 sets) |
|---|---|---|
| Average CPU Utilization | 20% | 12% |
| Peak CPU Utilization | 40% | 25% |
| Memory Usage | 2 GB | 1.5 GB |

| | | |
|---|---|---|
| Network Throughput | 20 Gbps | 30 Gbps |
| Context Switching Rate | 500/s | 200/s |

**Table 12: Multiple IP Hash Set-2**

Table 12 shows the parameters like Average CPU utilization, Peak CPU Utilization, Memory Usage and Network Througput and context switching. . Avg CPU utilization is 40% at 5 sets configuration where it is 25% at multiple IPHash sets 10 sets configuration. Peak CPU Utilization is 30% at 5 sets configuration and it is 20% at 10 sets configuration. Memory usage is 2GB at 5 sets configuration and it is 1.5GB at 10 sets configuration. Network throughput is 20 Gbps at 5 sets configuration and it is 30 Gbps ms at 10 sets configuration. Context switching rate is 500 per second at 5 sets configuration and it is 200 per second in 10 sets configuration.



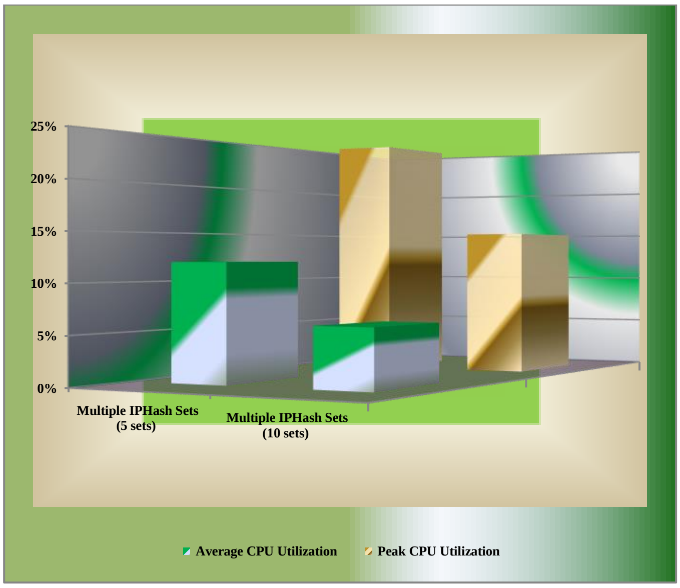

**Graph 9: Multiple IP Hash Set-2**

Graph 9 shows the Peak CPU utilization is going down when the number of sets are going high where avg CPU utilization is going down when the number of sets are going to high. Different shades in the bar representing the values is no way related to volume of the value.

| Metric | Multiple IPHash Sets (5 sets) | Multiple IPHash Sets (10 sets) |
|---|---|---|
| Average CPU Utilization | 12% | 6% |
| Peak CPU Utilization | 25% | 15% |
| Memory Usage | 900 MB | 600 MB |
| Policy Evaluation Time | 200 µs | 100 µs |
| Rule Matching Speed | 5000 rules/s | 10,000 rules/s |

**Table 13: Multiple IP Hash Set-3**

Table 13 shows the parameters like Average CPU utilization, Peak CPU Utilization, Memory Usage , policy evaluation time and rule matching speed. Avg CPU utilization is 12% at 5 sets configuration where it is 6% at multiple IPHash sets 10 sets configuration. Peak CPU Utilization is 25% at 5 sets configuration and it is 15% at 10 sets configuration. Memory usage is 900 MB at 5 sets configuration and it is 600 MB at 10 sets configuration. Policy evaluation time is 200us , where as it is 100us in 10 sets configuration, and Rule matching speed is 5000 rules per second and it is 10000 rules per second at 10 sets configuration.



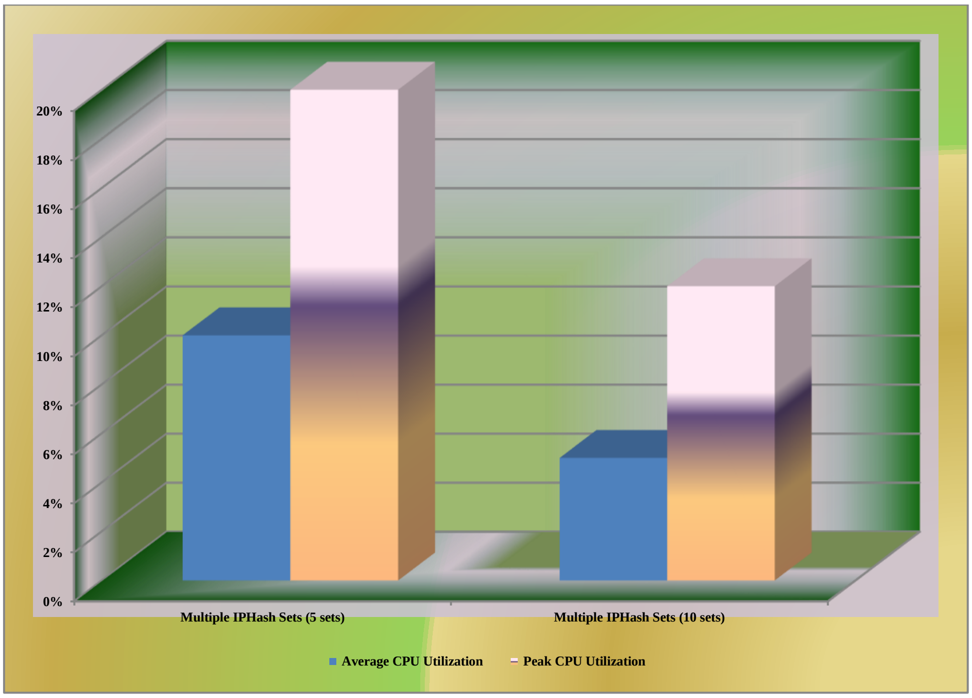

**Graph 10**: **Multiple IP Hash Set-3**

Graph 10 shows avg cpu utilization and peak cpu utilization for 5 set and 10 set configuration. Avg CPU utilization is going down as it raises with number of sets and peak CPU utilization is also follows the same. Different shades in the bar representing the values is no way related to volume of the value. Different shades in the bar representing the values is no way related to volume of the value.

| Metric | Multiple IPHash Sets (5 sets) | Multiple IPHash Sets (10 sets) |
|---|---|---|
| Average CPU Utilization | 10% | 5% |
| Peak CPU Utilization | 20% | 12% |
| Memory Usage | 600 MB | 400 MB |
| Average Response Time | 20 ms | 10 ms |
| Pod Scaling Time | 2 min | 1 min |

**Table 14: Multiple IP Hash Set-4**

Table 14 shows the parameters like Average CPU utilization, Peak CPU Utilization, Memory Usage , avg response time and pod scaling time. Avg CPU utilization is 10% at 5 sets configuration where it is 5% at multiple IPHash sets 10 sets configuration. Peak CPU Utilization is 20% at 5 sets configuration and it is 12% at 10 sets configuration. Memory usage is 600 MB at 5 sets configuration and it is 400 MB at 10 sets configuration. Average response time is 20ms , where as it is 10ms in 10 sets configuration, and pod scaling timer is 2 min at 5 sets configuration and it is 1 min for 10 sets configuration.



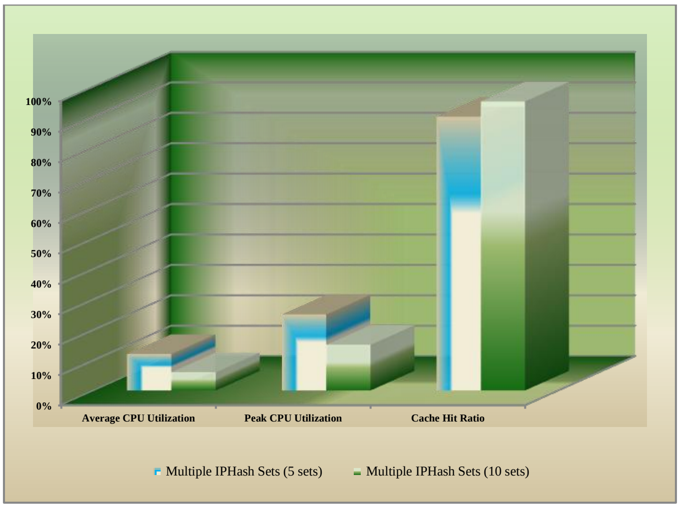

**Graph 11: Multiple IP Hash Set-4**

Graph 11 shows the Avg CPU utilization time and Peak CPU Utilization time for 5 sets configuration and 10 sets configuration. Average CPU utilization time is getting down while going from 5 to 10 sets in the configuration. Peak CPU utilization is going down while using from 5 sets to 10 sets configuration. Different shades in the bar representing the values is no way related to volume of the value.

| Metric | Multiple IPHash Sets (5 sets) | Multiple IPHash Sets (10 sets) |
|---|---|---|
| Average CPU Utilization | 12% | 6% |
| Peak CPU Utilization | 25% | 15% |
| Memory Usage | 400 MB | 200 MB |
| Network Bandwidth | 500 Mbps | 1 Gbps |
| Cache Hit Ratio | 90% | 95% |

**Table 15: Multiple IP Hash Set-5**

Table 15 shows the parameters like Average CPU utilization, Peak CPU Utilization, Memory Usage , network bandwith, and cache hit ratio. Avg CPU utilization is 12% at 5 sets configuration where it is 6% at multiple IPHash sets 10 sets configuration. Peak CPU Utilization is 25% at 5 sets configuration and it is 15% at 10 sets configuration. Memory usage is 400 MB at 5 sets configuration and it is 200 MB at 10 sets configuration. Netwrok Bandwidth is 500 Mbps for 5 sets configuration, 1 Gbps is for 10 sets configuration , cache hit ratio is 90% is for 5 sets configuration and 95% is for 10 sets configuration.



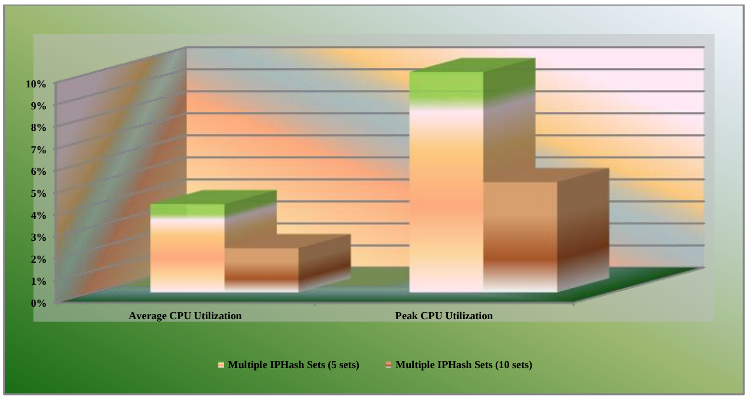

**Graph 12: Multiple IP Hash Set-5**

Graph 12 shows the avg CPU utilization , peak CPU utilization , and cache hit ratio . While we are using high set configuration CPU utilization is going down , cache hit ratio is going high while sets are going high. Different shades in the bar representing the values is no way related to volume of the value.

| Metric | Multiple IPHash Sets (5 sets) | Multiple IPHash Sets (10 sets) |
|---|---|---|
| Average CPU Utilization | 4% | 2% |
| Peak CPU Utilization | 10% | 5% |
| Memory Usage | 200 MB | 100 MB |
| Average Latency | 10 ms | 5 ms |
| Packet Processing Rate | 2000 packets/s | 5000 packets/s |

**Table 16: Multiple IP Hash Set-6**

Table 16 shows the parameters like Average CPU utilization, Peak CPU Utilization, Memory Usage , average latency, ansd packet processing. Avg CPU utilization is 4% at 5 sets configuration where it is 2% at multiple IPHash sets 10 sets configuration. Peak CPU Utilization is 10% at 5 sets configuration and it is 5% at 10 sets configuration. Memory usage is 200 MB at 5 sets configuration and it is1200 MB at 10 sets configuration. Average latency is 10 ms at 5 sets configuration and it is 5 ms at 10 sets configuration. Packet processing rate is 2000 per second at 5 sets configuration and it is 5000 per second at 10 sets configuration.

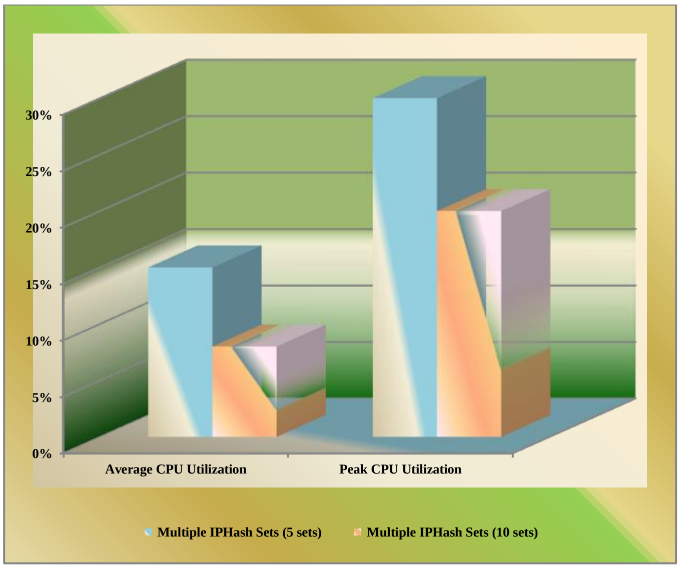



**Graph 13: Multiple IP Hash Set-6**

| Metric | Multiple IPHash Sets (5 sets) | Multiple IPHash Sets (10 sets) |
|---|---|---|
| Average CPU Utilization | 15% | 8% |
| Peak CPU Utilization | 30% | 20% |
| Memory Usage | 1 GB | 600 MB |
| Network Throughput | 10 Gbps | 20 Gbps |
| Context Switching Rate | 200/s | 100/s |

**Table 17: Multiple IP Hash Set-7**

Table 17 shows the parameters like Average CPU utilization, Peak CPU Utilization, Memory Usage , average latency, ansd packet processing. Avg CPU utilization is 15% at 5 sets configuration where it is 8% at multiple IPHash sets 10 sets configuration. Peak CPU Utilization is 30% at 5 sets configuration and it is 20% at 10 sets configuration. Memory usage is 1Gb at 5 sets configuration and it is 600 MB at 10 sets configuration. Network througput is 10 Gbps at 5 sets configuration and it is 20 Gbps at 10 sets configuration. Context switching rate is 200 per second at 5 sets configuration and it is 100 per second at 10 sets configuration.

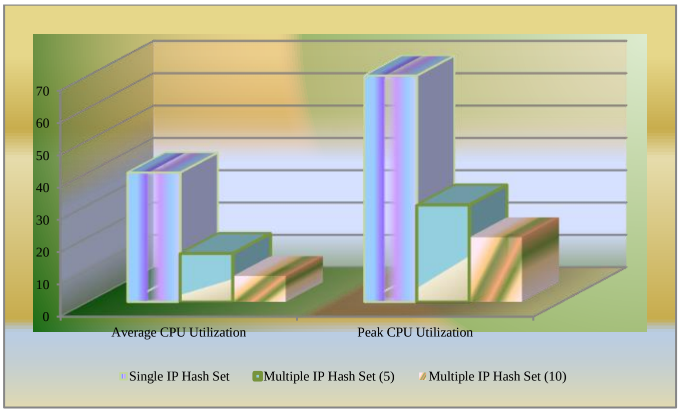



**Graph 14**: **Multiple IP Hash Set-7**

Graph 14 shows the avg CPU utilization , peak CPU utilization . While we are using high set configuration CPU utilization is going down. Different shades in the bar representing the values is no way related to volume of the value.

| Metric | Single IP Hash Set | Multiple IP Hash Set (5) | Multiple IP Hash Set (10) |
|---|---|---|---|
| Average CPU Utilization | 40 | 15 | 8 |
| Peak CPU Utilization | 70 | 30 | 20 |

**Table 18: Single** Vs **Multiple IP Hash Set-1**

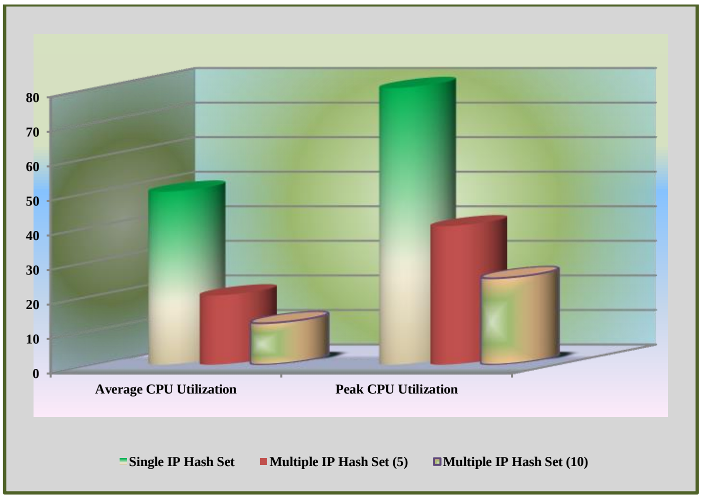



**Graph 15**: **Single** Vs **Multiple IP Hash Set-1**

Table 18 and Graph 15 shows that avg and peak CPU utilizations are going down while increasing the sets in IP Hash Set configuration.

| Metric | Single IP Hash Set | Multiple IP Hash Set (5) | Multiple IP Hash Set (10) |
|---|---|---|---|
| Average CPU Utilization | 50 | 20 | 12 |
| Peak CPU Utilization | 80 | 40 | 25 |

**Table 19: Single** Vs **Multiple IP Hash Set-2**

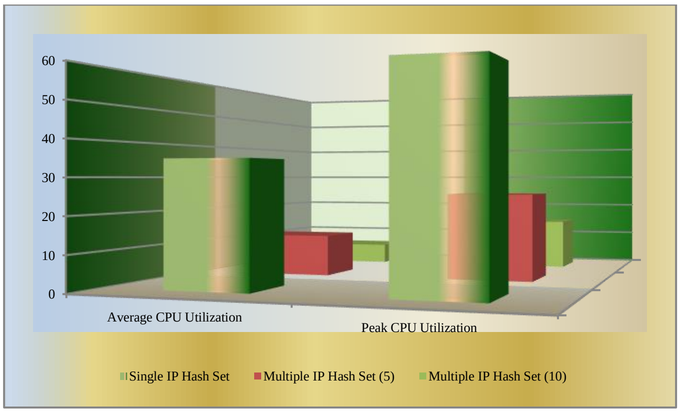



**Graph 16**: **Single** Vs **Multiple IP Hash Set-2**

Table 19 and Graph 16 shows that average and peak CPU utilizations are going down from 35 to 12 and 6 , 60 to 25 and 15 while increasing the sets in IP Hash Set configuration.

| Metric | Single IP Hash Set | Multiple IP Hash Set (5) | Multiple IP Hash Set (10) |
|---|---|---|---|
| Average CPU Utilization | 35 | 12 | 6 |
| Peak CPU Utilization | 60 | 25 | 15 |

**Table 20: Single** Vs **Multiple IP Hash Set-3**

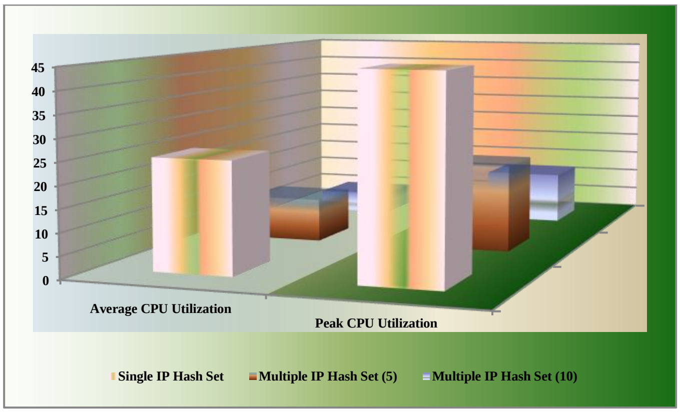



**Graph 17**: **Single** Vs **Multiple IP Hash Set-3**

Table 20 and Graph 17 shows that the single IP Hash set is having higer values at average CPU and peak cpu utilization compared to 5 sets and 10 sets IP Hash Set configuration.

| Metric | Single IP Hash Set | Multiple IP Hash Set (5) | Multiple IP Hash Set (10) |
|---|---|---|---|
| Average CPU Utilization | 25 | 10 | 5 |
| Peak CPU Utilization | 45 | 20 | 12 |

**Table 21: Single** Vs **Multiple IP Hash Set-4**

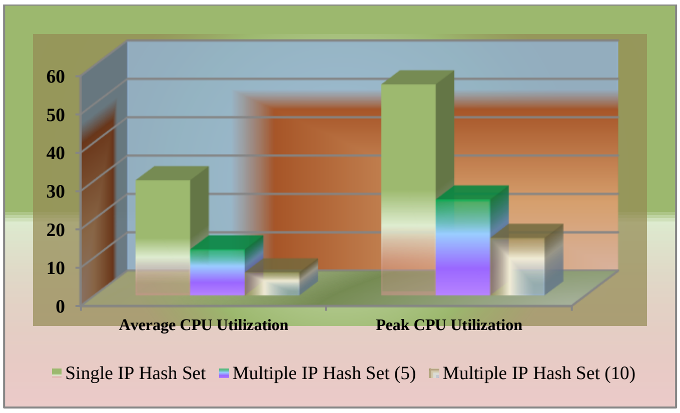



**Graph 18**: **Single** Vs **Multiple IP Hash Set-4**

Table 21 and Graph 18 shows that the Average CPU utilization is going down from 25 to 10 and again to 5 , peak CPU utilization is going down from 45 to 25 and 12.

| Metric | Single IP Hash Set | Multiple IP Hash Set (5) | Multiple IP Hash Set (10) |
|---|---|---|---|
| Average CPU Utilization | 30 | 12 | 6 |
| Peak CPU Utilization | 55 | 25 | 15 |

**Table 22: Single** Vs **Multiple IP Hash Set-5**

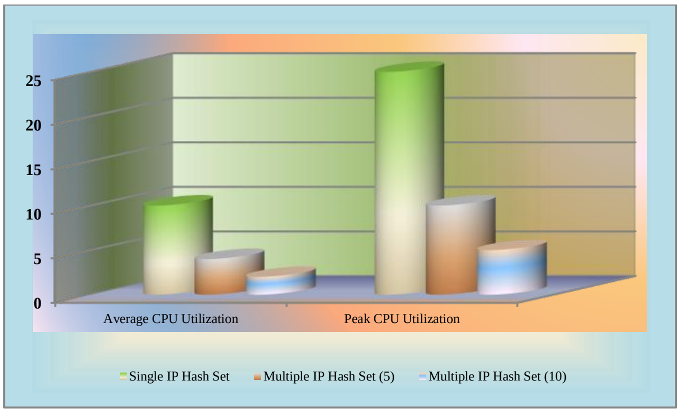



**Graph 19**: **Single** Vs **Multiple IP Hash Set-5**

Table 22 and Graph 19 shows that the Single Hash Set configuration is having the values for avg CPU utilization and peak cpu utilization are 30 and 25 where as it is 12 and 6 , 25 , 15 for 5 set IP hash set configuration and 10 set IP hash Set configuration.

| Metric | Single IP Hash Set | Multiple IP Hash Set (5) | Multiple IP Hash Set (10) |
|---|---|---|---|
| Average CPU Utilization | 10 | 4 | 2 |
| Peak CPU Utilization | 25 | 10 | 5 |

**Table 23: Single** Vs **Multiple IP Hash Set-6**

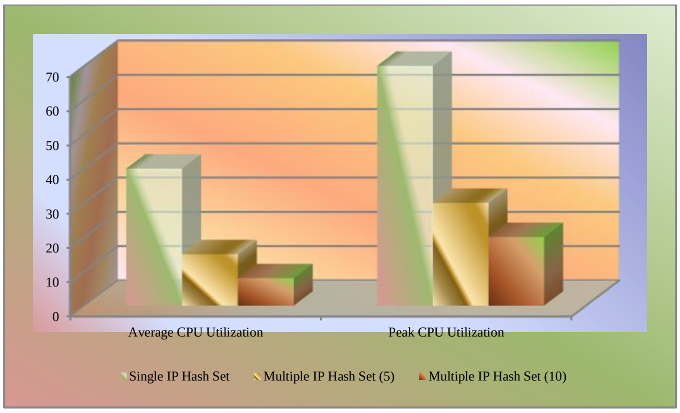



**Graph 20**: **Single** Vs **Multiple IP Hash Set-6**

Table 23 and Graph 20 shows that the Single Hash Set configuration is having the values for avg CPU utilization and peak cpu utilization are 10 and 25 where as it is 4 and 2 , 10 , 5 for 5 set IP hash set configuration and 10 set IP hash Set configuration.

| Metric | Single IP Hash Set | Multiple IP Hash Set (5) | Multiple IP Hash Set (10) |
|---|---|---|---|
| Average CPU Utilization | 40 | 15 | 8 |
| Peak CPU Utilization | 70 | 30 | 20 |

**Table 24: Single** Vs **Multiple IP Hash Set-7**




**Graph 21**: **Single** Vs **Multiple IP Hash Set-7**

Table 24 and Graph 21 shows that the Single Hash Set configuration is having the values for avg CPU utilization and peak cpu utilization are 40 and 70 where as it is 15 and 8 , 30 , 20 for 5 set IP hash set configuration and 10 set IP hash Set configuration.

## EVALUATION

In sample 1 , Avg CPU utilization is reduced to 8 from 40 where as Peak value is reduced to 20 from 70. In sample 2 the values are coming from 50 to 12 , and peak is coming from 80 to 25. In sample 3 cpu utilization is coming form 35 to 6 , and peak is coming from 60 to 15. In sample 4 cpu utilization is coming form 25 to 5 , and peak is coming from 45 till 12. In sample 5 the values rae coming form 30 to 6 and 55 to 15. In sample 6 the values are coming from 10 to 2 and 25 till 5.

## CONCLUSION

We have configured three node , four node , five node , six node , seven node , eight node , nine node and ten node clusters with 32 CPU, 64 GB and 500GB for master node and 24 CPU , 32 GB and 350 GB for all worker nodes.

IP Table size is no way related to cluster size (number of nodes). size of the IP table is influenced by the number of pods, services, and network policies**,** it is not a direct measure of the cluster size (i.e., the number of nodes). Instead, it is more closely related to the complexity of the network configuration in the cluster.

A larger cluster with many pods and services, especially if there are complex network policies or ingress configurations, will likely result in a larger IP table.

We have tested the performance of ip tables having single IP Hash set and multiple IP Hash set , and shown in the table and graphs multi IP Hash set configuration of IP Tables is giving high performance compared to single hash set IP configuration. The main reason here is it is following the set associative concept , where it is storing all the related items in one set , so that it is easy to access the set number like index number and scanning the reaming number already stored at the set. With this we can conclude that implementing the IP Tables using Multi Set IP Hash set configuration gives best CPU utilization either it is average cpu utilization or peak cpu utilization.

We didn’t cover the concept of log management in this paper. The future work includes working on the log management for smaller IP Hash Set wrt Higher IP Hash Set configuration.

## REFERENCES

1. Kuberenets in action by Marko Liksa , 2018.
2. Kubernetes and Docker - An Enterprise Guide: Effectively containerize applications, integrate enterprise systems, and scale applications in your enterprise by Scott Surovich and Marc Boorshtein, 2020.
3. Kubernetes Patterns, Ibryam , Hub
4. Kubernetes Best Practices , Burns, Villaibha, Strebel , Evenson.
5. Learning Core DNS, Belamanic, Liu.
6. Core Kubernetes , Jay Vyas , Chris Love.
7. A Formal Model of the Kubernetes Container Framework. GianlucaTurin, AndreaBorgarelli, SimoneDonetti, EinarBrochJohnsen, S.LizethTapiaTarifa, FerruccioDamiani Researchreport496,June202
8. Kubernetes Container Orchestration as a Framework for Flexible and Effective Scientific Data Analysis, IEEE Xplore, 13 February 2020.
9. A survey of Kubernetes scheduling algorithms, Khaldoun Senjab, Sohail Abbas, Naveed Ahmed & Atta ur Rehman Khan Journal of Cloud Computing volume, 12 , 2023.
10. Research and Implementation of Scheduling Strategy in Kubernetes for Computer Science Laboratory in Universities, by Zhe Wang 1,Hao Liu ,Laipeng Han ,Lan Huang and Kangping Wang.
11. Study on the Kubernetes cluster mocel, Sourabh Vials Pilande. International Journal of Science and Research , ISSN : 2319-7064.
12. Network Policies in Kubernetes: Performance Evaluation and Security Analysis, Gerald Budigiri; Christoph Baumann; Jan Tobias Mühlberg; Eddy Truyen; Wouter Joosen, IEEE Xplore 28 July 2021.
13. Networking Analysis and Performance Comparison of Kubernetes CNI Plugins, 28 October 2020, pp 99–109, Ritik Kumar & Munesh Chandra Trivedi.
14. Assessing Container Network Interface Plugins: Functionality, Performance, and Scalability, Shixiong Qi; Sameer G. Kulkarni; K. K. Ramakrishnan, 25 December 2020 , IEEEXplore.
15. Kubernetes and Docker Load Balancing: State-of-the-Art Techniques and Challenges, International Journal of Innovative Research in Engineering & Management, Indrani Vasireddy, G. Ramya, Prathima Kandi
16. Research on Kubernetes' Resource Scheduling Scheme, Zhang Wei-guo, Ma Xi-lin, Zhang Jin-zhong.
17. Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned, Leila Abdollahi Vayghan Montreal, Mohamed Aymen Saied; Maria Toeroe; Ferhat Khendek, IEEE XPlore.
18. Improving Application availability with Pod Readiness Gates https://orielly.ly/h_WiG
19. Kubernetes Best Practices: Resource Requests and limits https://orielly.ly/8bKD5
20. Configure Default Memory Requests and Limits for a Namespace https://orielly.ly/ozlUi1
21. Kubernetes CSI Driver for mounting images https://orielly.ly/OMqRo
22. Modelling performance & resource management in kubernetes by Víctor Medel, Omer F. Rana, José Ángel Bañares, Unai Arronategui.
23. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases, Viktor Leis, Alfons Kemper, Thomas Neumann.

24. Trie: Mathematical and Computer Modelling An Alternative Data Structure for Data Mining Algorithms F. BODON AND L. R~NYAI Computer and Automation Institute, Hungarian Academy of Sciences.
25. Distributed Kubernetes Metrics Aggregation, 23 September 2022, pp 695–703, Mrinal Kothari, Parth Rastogi, Utkarsh Srivastava, Akanksha Kochhar & Moolchand Sharma, Springer.
26. An Analysis on the Performance of Tree and Trie based Dictionary Implementations with Different Data Usage Models, M. Thenmozhi1 and H. Srimathi, Indian Journal of Science and Technology, Vol 8(4), 364–375, February 2015.
27. A Portable Load Balancer for Kubernetes Cluster, 28 January 2018, Kimitoshi Takahashi, Kento Aida, Tomoya Tanjo, Jingtao SunAuthors Info & Claims.
28. A reduction algorithm based on trie tree of inconsistent system, Xiaofan Zhang, IEEEXplore
29. Predicting resource consumption of Kubernetes container systems using resource models, Gianluca Turin , Andrea Borgarelli , Simone Donetti , Ferruccio Damiani , Einar Broch Johnsen , S. Lizeth Tapia Tarifa.
30. TRIE DATA STRUCTURE, Pallavraj SAHOO. 2015, Research Gate.