Exploring the Features of Kubernetes

Pradnya Kandarkar
kandarkarp@gmail.com

Farheen Jamadar
farheen.jamadar158@gmail.com

Afif Bin Kamrul
afifbinkamrul@gmail.com

Yashvi Pithadia
yashvi.pithadia.it@gmail.com

ABSTRACT

This project explores the following aspects of distributed systems using Kubernetes: load distribution, scalability, high availability, failure recovery. Kubernetes is a container orchestration system. A Kubernetes cluster has control plane and worker nodes and provides support for deploying and managing applications using these nodes. Above features are studied using a local multi-node Kubernetes cluster.

1 PROJECT SETUP

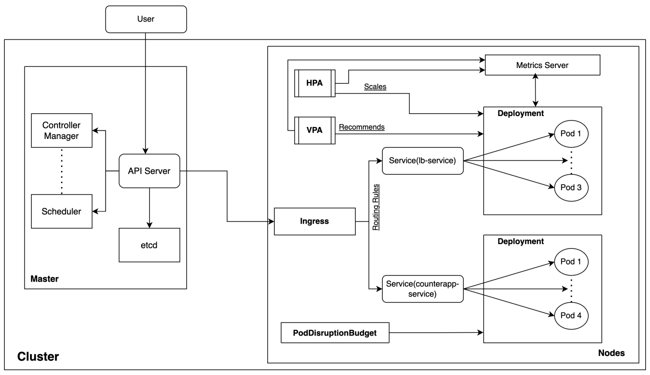


Figure 1: System Diagram

Minikube offers a lightweight implementation of Kubernetes. A local Kubernetes cluster consisting of 4 nodes is set up using minikube: one master node and three worker nodes. Every node has one or more pods. Pods consist of containerized applications built using Docker. Two containerized applications are deployed in the cluster, each having more than one replica. Kubernetes has Deployment objects that are used to specify details about how applications are to be deployed, such as number of replicas/pods for the application, how the pods should be assigned to nodes. Service objects are used to expose these applications in order to allow external traffic to flow in. An ingress object is used to specify rules that are followed to route traffic to appropriate services in the cluster. The selected aspects of distributed systems (load distribution, scalability, high availability, failure recovery) for Kubernetes are discussed in the subsequent sections.

2 LOAD DISTRIBUTION

Kubernetes provides Service and Ingress objects to expose applications running in a cluster as network services. A service object is used to expose a single network service. Services can be published in different ways using different service types (Kubernetes Service-Types). The following is a brief introduction to available service types:

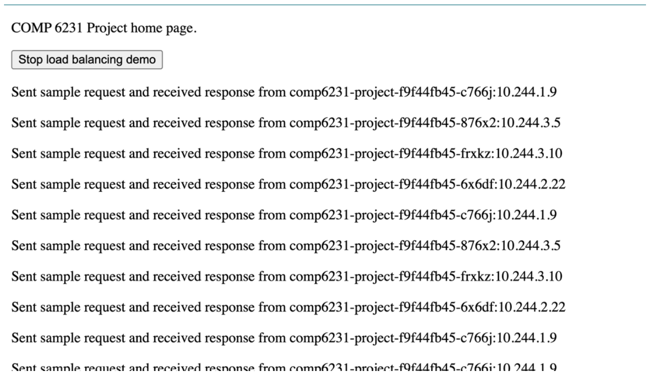
- ClusterIP: This can be used to make a service available inside a cluster. This is the default option for service type.
- NodePort: The service is exposed at a static port (default range: 30000-32767) for every node IP. A NodePort routes traffic to a ClusterIP which is automatically created. External traffic can access services exposed using NodePort.
- LoadBalancer: The service is exposed to external traffic through an external load balancer. NodePort and ClusterIP services, to which the load balancer routes, are automatically created.
- ExternalName: The service is mapped to the contents of the "externalName" field (e.g., abc.example.com). This is useful when representing an external service, such as an external database, in a cluster.

An ingress resource can be used to expose multiple network services in a cluster and provide load distribution among them based on a specified set of rules. LoadBalancer and Ingress are used in the project setup for load distribution and are described below:

2.1 LoadBalancer

When LoadBalancer is used to expose a service, an external load balancer is used to direct incoming traffic to the appropriate backend pods. When using this service type on cloud-based clusters, where an external load balancer is supported, an external load balancer should be supplied for accessing the service. When LoadBalancer is used on a cluster implemented using minikube, the service is made available through the "minikube service" command.
In the project setup, both application deployments are exposed using the LoadBalancer service type. Based on the configuration specified in the Deployment objects for applications, multiple pods for an application may be assigned to the same node or each pod may get assigned to a different node. Load distribution across pods is observed regardless of whether different pods are assigned to the same node or different nodes. When the number of pods increases/decreases, load distribution adapts to this change and load is balanced among the available pods. Figure 2 shows a working demonstration of load distribution across 4 replicas of an application in the project setup.

Each service exposed with LoadBalancer requires an IP address. Kubernetes Ingress can be used to expose multiple services using a single IP address.



COMP 6231 Project home page.

```
Sent sample request and received response from comp6231-project-f9f44fb45-c766j:10.244.1.9
Sent sample request and received response from comp6231-project-f9f44fb45-876x2:10.244.3.5
Sent sample request and received response from comp6231-project-f9f44fb45-frxkz:10.244.3.10
Sent sample request and received response from comp6231-project-f9f44fb45-6x6df:10.244.2.22
Sent sample request and received response from comp6231-project-f9f44fb45-c766j:10.244.1.9
Sent sample request and received response from comp6231-project-f9f44fb45-876x2:10.244.3.5
Sent sample request and received response from comp6231-project-f9f44fb45-frxkz:10.244.3.10
Sent sample request and received response from comp6231-project-f9f44fb45-6x6df:10.244.2.22
Sent sample request and received response from comp6231-project-f9f44fb45-c766j:10.244.1.9
Sent sample request and received response from comp6231-project-f9f44fb45-c766j:10.244.1.9
```

Figure 2: System Diagram

2.2 Ingress

Ingress is not a type of Service object. An ingress object manages external access to services in a cluster (typically HTTP) and can provide load balancing along with other services. An ingress resource can consist of rules to route HTTP(S) traffic to different services. An ingress controller is needed to continuously monitor and ensure that traffic is forwarded as per the rules specified in ingress. An ingress without an ingress controller is not sufficient.
Using ingress, traffic can be load-balanced across different services by forwarding requests with specific paths (URLs) to specific services. A default backend can be configured where all requests that do not match any rule in the ingress object can be forwarded. Default backend configuration option should be present in the ingress controller and not the ingress resources. An ingress with no rules will end up sending all traffic to a single default backend.
Ingress objects do not work with arbitrary protocols. For exposing services using protocols other than HTTP and HTTPS, LoadBalancer or other available service types can be used.
Ingress object acts as the entry point for the project system and is used to route HTTP traffic to appropriate services based on the provided path in the requests. It is observed that traffic is forwarded to services as expected according to the paths specified in the HTTP requests.

3 AUTOSCALING

Kubernetes enables optimization of resource usage and resolves management and commercial problems by autoscaling. There are three methods of autoscaling that Kubernetes provides, namely, Horizontal Pod Autoscaling, Vertical Pod Autoscaling, and Cluster Autoscaling.

3.1 Horizontal Pod Autoscaling

Horizontal Pod Autoscaler automatically updates the workload resource to adapt to the changing workload requirements. In other words, it is a scaling-out activity that manages the number of pods based on the application resource usage. Kubernetes Horizontal Pod Autoscaler is a control loop that monitors the state of the cluster and acts to achieve the desired state. It decides the desired number of pods based on the threshold set by the cluster operator against the chosen metrics. These metrics could be obtained from resource metrics API, which is used for per-pod-resource metrics, custom metrics API, which are application-specific metrics deployed in the Kubernetes cluster, and external metrics API, which is reported from the application that is not running in Kubernetes cluster but its performance impacts the aforementioned application. Kubernetes HorizontalPodAutoscaler(HPA) calculates the desired replicas by taking the ceiling value of the product of the current replica count with the division value of current and desired metric value.

3.2 Vertical Pod Autoscaling

Vertical Pod Autoscaler is a scaling-up operation that calculates the desired resource parameters (CPU and memory) based on the resource usage and recommends or automatically configures the deployment. It minimizes the overhead of setting or updating resource requests and limit for their containers and improves resource utilization. Kubernetes provides VerticalPodAutoscaler(VPA) with different modes of operation, Auto, Recreate, Initial, Off. Kubernetes implementation of VPA has three main components:

1. **VPA Recommender**: This component monitors all the Pods, extracts corresponding metrics from the metrics server, and calculates recommendations for them
2. **VPA Objects**: VPA recommender uses VPA Objects to save its recommendations
3. **VPA admission controller**: It overrides the pod's resources with the respective recommendations

3.3 Cluster Autoscaling

Cluster Autoscaler manages the nodes in the cluster based on the workload. It will add nodes to the Kubernetes cluster if the existing nodes are unable to host new pods due to insufficient memory. It will remove them if they remain underutilized for a long time. Also, there is a possibility of shifting their workload to existing nodes. It decides to add or remove the nodes based on the resource requests of Pods running on a node in the node pool.

3.4 Implementation

Project implementation consists of a VPA and HPA based on the resource metrics API. They rely on the metrics server. It is enabled as an add-on in Minikube.

HorizontalPodAutoscaler monitors the CPU after a regular interval of time which has been set to 5 seconds using –horizontal-pod-autoscaler-sync-period. In the event of underutilization, HPA will scale down the required amount of pods by considering the highest recommendation from all the recommendations within a certain period. This period has been set to 10 seconds using –horizontal-pod-autoscaler-downscale-stabilization. This HPA configuration leads to thrashing or flapping, which is frequent fluctuation in the replica count, because of short synchronization and downscale window. Hence, a cluster manager has to carefully configure the respective time windows based on the workload in the deployed application. The minimum allowed replica count is set to 1, and the maximum count is set to 10. Horizontal Pod Autoscaler implemented using only the resource metrics may not be an efficient solution for this implementation. Scaling based on multiple custom metrics, for example, HTTP requests count with latency in the requests would fit better to the use case.

In Kubernetes, VerticalPodAutoscaler is a custom resource definition object, and has been implemented in Off mode. It is not recommended to use VPA and HPA with the same resource metrics. This may lead to race conditions and also, fluctuations in replicas. Here, VPA has not been implemented in auto mode as it will change the workload resource based on the CPU and memory and this will clash with the HPA implementation eventually.

A separate local flask application provides a User Interface to present HPA and VPA output as the Minikube dashboard does not provide a user-friendly way to monitor the process. It utilizes Kubernetes API and kubectl to fetch the needed outcomes.

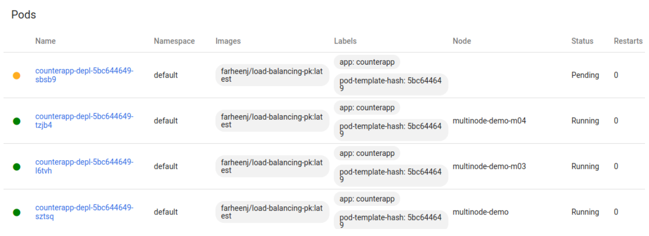
Project implementation does not include Cluster autoscaler as Minikube does not support automatic addition or reduction of nodes. Node count has to be specified in Minikube during startup.

4 HIGH AVAILABILITY

High Availability is one of the most desired requirements of any cloud system. There are many cases to be optimized for ensuring high availability for a kuberbetes cluster. For example: rounding downtime as close to zero all the time and while updating the application server, ensuring tolerance against faulty nodes, ensuring the server is up and running after a voluntary or involuntary disruption etc. For our system we investigated how to safely update the system and save a system against voluntary disruption.

4.1 Save System against Voluntary Disruption

Cluster administrator may have to drain node for maintenance willingly. But if the administrator unwillingly drains all the nodes, the system will be unavailable to the users. In order to safeguard this voluntary disruption, a Pod Disruption Budget will work in between the administrator and the system. In our case, we set the minAvailable to 75 percent and we implemented the Pod Disruption Budget rule to a counterapp-depl.yaml deployment file. In order to meet a certain high availability condition of having as much replicas of pod as possible, we set the replicas to 4. We also implemented Affinity rule so that 4 different pods would land on 4 separate nodes. When we tried to drain more than one node like an administrator, it straight up refused to do so because it would violate Pod Disruption Budget rule. Normally the evicted pods would land on a different node.


Pods

	Name	Namespace	Images	Labels	Node	Status	Restarts
●	counterapp-depl-5bc644649-sbsb9	default	farheenj/load-balancing-pk:latest	app: counterapp pod-template-hash: 5bc644649		Pending	0
●	counterapp-depl-5bc644649-tzjb4	default	farheenj/load-balancing-pk:latest	app: counterapp pod-template-hash: 5bc644649	multinode-demo-m04	Running	0
●	counterapp-depl-5bc644649-l6tvh	default	farheenj/load-balancing-pk:latest	app: counterapp pod-template-hash: 5bc644649	multinode-demo-m03	Running	0
●	counterapp-depl-5bc644649-sztsq	default	farheenj/load-balancing-pk:latest	app: counterapp pod-template-hash: 5bc644649	multinode-demo	Running	0

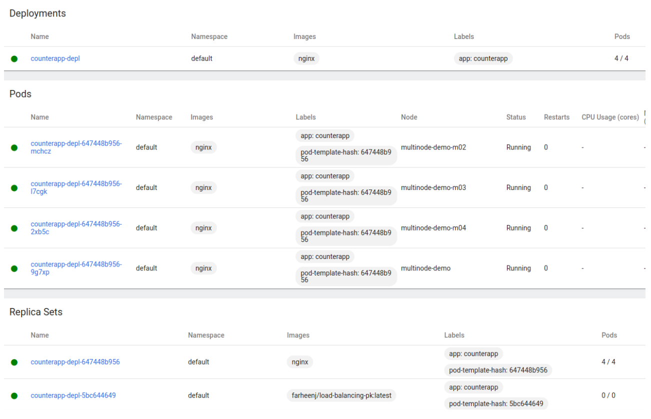
Figure 3: One pod is pending to be evicted when multinode-demo-m02 is drained but in a pending state as it can't land on another node

From the minikube dashboard shown in 3 we can demonstrate that one pod is in pending state and it can not land on another node. If the administrator tried to drain another node, an error will be popped up.

In order to ensure a super stable system, we could set minAvailable to 100 percent so that no node could be drained and no pod could be evicted. If system update is required, Cluster Administrator will have to contact the authority.

4.2 Rolling Update without Downtime

It is common practice in software development procedure to update a system with a new version. But it is a challenge to ensure zero downtime while rolling the update and rolling back in case any problem occurs. For our system, we tested the same counterapp-depl.yaml file with 4 replicas of pod. we set the strategy of update to rollingupdate and set the maxSurge and maxUnavailable to 25 percent. The whole idea behind was to update the docker image from one to another and accordingly only one pod could go down at a time while rolling the update. In this way, the other 3 pods could be functional and the system would be functional. In the


Deployments

	Name	Namespace	Images	Labels	Pods
●	counterapp-depl	default	nginx	app: counterapp	4 / 4

Pods

	Name	Namespace	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	
●	counterapp-depl-647448b956-mchcz	default	nginx	app: counterapp pod-template-hash: 647448b956	multinode-demo-m02	Running	0	-	-
●	counterapp-depl-647448b956-l7cgk	default	nginx	app: counterapp pod-template-hash: 647448b956	multinode-demo-m03	Running	0	-	-
●	counterapp-depl-647448b956-2xb5c	default	nginx	app: counterapp pod-template-hash: 647448b956	multinode-demo-m04	Running	0	-	-
●	counterapp-depl-647448b956-9g7xp	default	nginx	app: counterapp pod-template-hash: 647448b956	multinode-demo	Running	0	-	-

Replica Sets

	Name	Namespace	Images	Labels	Pods
●	counterapp-depl-647448b956	default	nginx	app: counterapp pod-template-hash: 647448b956	4 / 4
●	counterapp-depl-5bc644649	default	farheenj/load-balancing-pk:latest	app: counterapp pod-template-hash: 5bc644649	0 / 0

Figure 4: System has been updated to a different image

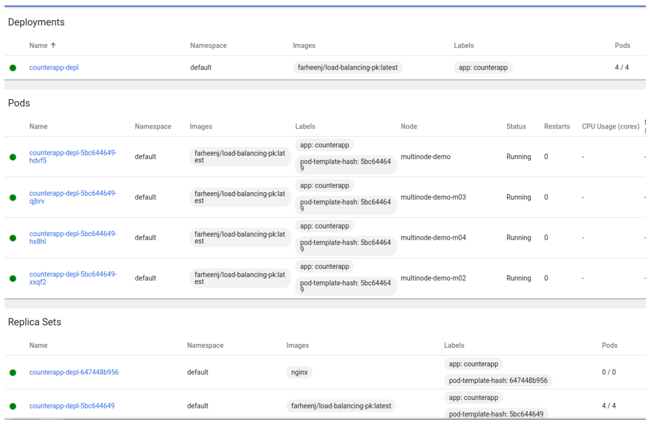
same way, if we want to rollback the update for some malfunction, the system will rollback to previous version by following the same previous rule. For both of the case, if some unresolvable case inside the cluster happens, it will surge one pod to scale up and balance as maxSurge is set to 25 percent.

5 FAILURE RECOVERY

For maintaining the availability of deployed microservices, Kubernetes provides health checks and repair actions. We have demonstrated three such features. First, Kubernetes detects pod failures at the pod level and responds according to the set restart policy. Second, Kubernetes uses distributed daemons to monitor the cluster's nodes for node failure detection at the node level. If the node hosting a pod goes down, the pod is rescheduled to another node that is up and running. Third, Kubernetes stores its objects in the etcd distributed database, which may be backed up to enable quick recovery in the event of a failure or data loss.

5.1 Pod Failure

When a pod is deployed, it creates an additional container, the pod container, in addition to the application containers. It's possible



Deployments

	Name ↑	Namespace	Images	Labels	Pods
●	counterapp-depl	default	farheenj/load-balancing-pk:latest	app: counterapp	4 / 4

Pods

	Name	Namespace	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	
●	counterapp-depl-5bc644649-hdvf5	default	farheenj/load-balancing-pk:latest	app: counterapp pod-template-hash: 5bc644649	multinode-demo	Running	0	-	-
●	counterapp-depl-5bc644649-qjbrv	default	farheenj/load-balancing-pk:latest	app: counterapp pod-template-hash: 5bc644649	multinode-demo-m03	Running	0	-	-
●	counterapp-depl-5bc644649-hx8hl	default	farheenj/load-balancing-pk:latest	app: counterapp pod-template-hash: 5bc644649	multinode-demo-m04	Running	0	-	-
●	counterapp-depl-5bc644649-xxqf2	default	farheenj/load-balancing-pk:latest	app: counterapp pod-template-hash: 5bc644649	multinode-demo-m02	Running	0	-	-

Replica Sets

	Name	Namespace	Images	Labels	Pods
●	counterapp-depl-647448b956	default	nginx	app: counterapp pod-template-hash: 647448b956	0 / 0
●	counterapp-depl-5bc644649	default	farheenj/load-balancing-pk:latest	app: counterapp pod-template-hash: 5bc644649	4 / 4

Figure 5: System has rolled back to previous image

that it'll go down. The failure is mimicked in this case by removing the pod. The Kubelet recognises that the pod container is no longer present when the pod container process is stopped. A new pod is formed in this situation to meet the required number of pods.

5.2 Node Failure

After a node failure, it stops transmitting status updates to the master. kubectl get nodes will report a NotReady state. The statuses of all pods operating on the NotReady node will change to Unknown or NodeLost. This is based on the pod eviction timeout settings, which are set at five minutes by default. Kubernetes will evict the pod on the failed node and attempt to reschedule it on healthy nodes . We're draining a node to evict all pods on it to simulate node failure.

5.3 Etcd Failure

etcd is a reliable and highly available key-value store that Kubernetes uses to back up all cluster data. We have a Kubernetes cluster with three etcd members operating. By tolerating member failures, the etcd cluster achieves high availability. However, we should replace unsuccessful members as soon as possible to improve the cluster's general health. Replacing a failing member entails removing the failed member and replacing it with a fresh one. It's critical to back up the etcd cluster data on a regular basis to restore Kubernetes clusters in the event of a disaster, such as losing all nodes. All of Kubernetes' states and crucial information are contained in the snapshot file. To retrieve data from a failing cluster, a restore procedure is used.

6 REFERENCES

- https://kubernetes.io/docs/concepts/services-networking/service/
- https://kubernetes.io/docs/tutorials/hello-minikube/
- https://kubernetes.io/docs/concepts/services-networking/ingress/
- https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
- https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/
- https://enterprisersproject.com/article/2021/3/kubernetes-autoscaling-explanation
- https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler
- https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-autoscaler
- https://cloud.google.com/kubernetes-engine/docs/concepts/custom-and-external-metrics
- https://kubernetes.io/docs/tasks/manage-daemon/update-daemon-set/
- https://kubernetes.io/docs/tasks/manage-daemon/rollback-daemon-set/
- https://kubernetes.io/blog/2018/04/30/zero-downtime-deployment-kubernetes-jenkins/
- https://kubernetes.io/docs/concepts/workloads/pods/disruptions/
- https://microk8s.io/high-availability
- https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/
- https://kubernetes.io/docs/tasks/run-application/configure-pdb/
- https://www.youtube.com/watch?v=X48VuDVv0do&t=6537s
- https://www.youtube.com/watch?v=xRifmrap7S8
- https://www.youtube.com/watch?v=09Wkw9uhPak
- https://www.youtube.com/watch?v=e2HjRrmXMDw
- https://arxiv.org/ftp/arxiv/papers/1901/1901.04946.pdf
- https://www.velotio.com/engineering-blog/the-ultimate-guide-to-disaster-recovery-for-your-kubernetes-clusters
- https://rafay.co/the-kubernetes-current/etcd-kubernetes-what-you-should-know/
- https://kubernetes.io/docs/home/
- https://etcd.io/docs/v3.5/
- https://www.ibm.com/docs/en/cloud-paks/cp-management/2.0.0?topic=restore-backing-up-restoring-etcd
- https://www.ibm.com/docs/en/spectrum-connect/3.5.0
- https://devops.com/decoding-the-self-healing-kubernetes/
- https://www.upnxtblog.com/index.php/2021/06/22/what-happens-when-one-of-your-kubernetes-nodes-fails/