ISSN: 2577-2996

PROGRESS IN MEDICAL SCIENCES, 2022
VOL 6, NO. 3, PAGE 1 – 6
DOI: doi.org/10.47363/PMS/2022(6)E108

REVIEW ARTICLE

Open Access

Resolve Bottlenecks at Kubernetes using Load Balancer Services

Pallavi Priya Patharlagadda

USA

ABSTRACT

Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery. During high load, Kubernetes performs the auto-scaling of the backend pods. But if the Kubernetes service itself is getting overloaded, it contains bottlenecks that affect the overall application performance. This paper speaks in detail about the bottleneck situation and how we can overcome it.

ARTICLE HISTORY

Received May 02, 2022
Accepted May 09, 2022
Published May 23, 2022

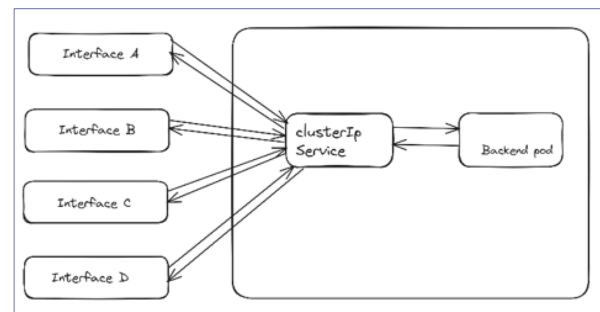
Introduction

Kubernetes is a powerful open-source orchestration tool, designed to help you manage microservices and containerized applications across a distributed cluster of computing nodes. Kubernetes was initially developed by Google and later supported by the Cloud Native Computing Foundation (CNCF).

Kubernetes is a platform designed to completely manage the life cycle of containerized applications and services using methods that provide predictability, scalability, and high availability. The way applications should be run and interact with other applications or outside the world can be defined by the user. Kubernetes helps you with scaling the services, rolling updates, and switching traffic between different application versions and deployments. Kubernetes provides interfaces and composable platform primitives that allow you to define and manage your applications with high degrees of flexibility, power, and reliability.

Problem Statement

Kubernetes has a lot of components. Consider a scenario where we are getting traffic from multiple interfaces. Let's say Interface A, Interface B, Interface C, and Interface D. All these interfaces are connected to a Service. During high traffic, the load on the service is increasing, and the traffic is dropping. We can configure the backend pods based on the load, but we cannot scale the Kubernetes service. Below is a picture of the problematic scenario. I have a lot more services, like Ingress and Replica Set etc. For simplicity's sake, I haven't shown all these in the diagram.



This paper speaks on how to resolve this issue

Kubernetes Architecture and Components

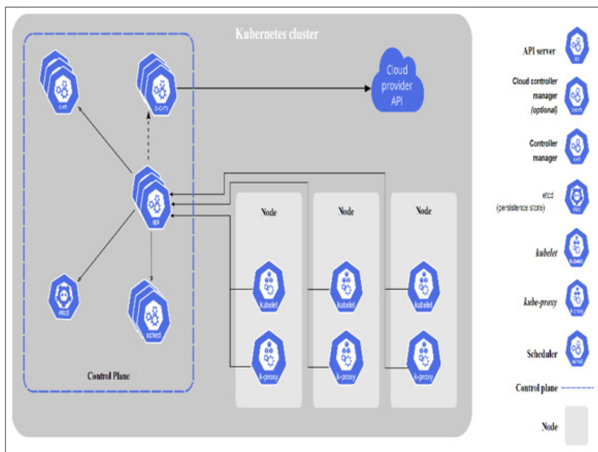
Kubernetes architecture is a client-server architecture.

The server side of Kubernetes is known as the control plane. By default, there is a single control plane server that acts as a controlling node and point of contact. This server consists of components including the Kube-apiserver, etc. storage, Kube-controller-manager, cloud-controller-manager, Kube-scheduler, and Kubernetes DNS server.

The client side of Kubernetes comprises the cluster nodes—these are machines or the worker nodes on which Kubernetes can run containers. Node components include the kubelet and kube-proxy.

Contact: Pallavi Priya Patharlagadda, USA.

© 2022 The Authors. This is an open access article under the terms of the Creative Commons Attribution NonCommercial ShareAlike 4.0 (https://creativecommons.org/licenses/by-nc-sa/4.0/).



Kubernetes Master Node Components

The components on the master node work together to accept user requests, determine the best ways for scheduling the pods, authenticate clients and nodes, adjust cluster-wide networking, and manage scaling and health checking responsibilities. Below are the main components found on the control plane node.

ETCD

etcd is a simple, lightweight, distributed key-value store that can be configured to span across multiple nodes. Kubernetes uses etcd to store configuration data that can be accessed by each of the nodes in the cluster. The etcd data can be used for service discovery, API objects, and cluster data (like pods, their states, namespaces, etc.). It also helps maintain cluster state with features like leader election and distributed locking. etcd enables notifications to the cluster about configuration changes with the help of watchers. Notifications are API requests on each etcd cluster node to trigger the update of information in the node's storage. For security reasons, the Kubernetes API server is the only component of Kubernetes that can directly connect to, etcd. The API server acts as a gateway between etcd and the rest of the Kubernetes system. Other Kubernetes components must go through the API server to interact with the cluster state.

Kube-Apiserver

The Kubernetes API server is the central management entity that receives all requests for modifications (to pods, services, replication sets/controllers, and others). The API server implements a RESTful interface to communicate with other Kubernetes components and serves as a frontend to the cluster. It is also responsible for making sure that the etcd store and the service details of deployed containers are in agreement. A client called kubectl is available as a default method of interacting with the Kubernetes cluster from a local computer.

Kube-Controller-Manager

The controller manager manages different controllers that regulate the state of the cluster, manage workload life cycles, and perform routine tasks. For instance, a replication controller ensures that the number of replicas (identical copies) defined for a pod matches the number currently deployed on the cluster. The endpoints controller populates endpoint objects like services and pods. The details of these operations are written to etcd, where the controller manager watches for changes through the API server. When a change is seen, the controller reads the new information and implements the procedure that fulfills the desired state. This can involve scaling an application up or down, adjusting endpoints, etc.

Kube-Scheduler

Kube-scheduler is a process that schedules the pods (a co-located group of containers inside which our application processes are running) on the various nodes based on resource utilization. It reads the service's operational requirements and schedules it on the best-fit node.

For example, if the application needs 1 GB of memory and 2 CPU cores, then the pods for that application will be scheduled on a node with at least those resources. The scheduler runs each time there is a need to schedule pods. The scheduler must know the total resources available as well as resources allocated to existing workloads on each node.

Cloud Controller Manager

Kubernetes can be deployed in many different environments and can interact with various infrastructure providers to manage the state of resources in the cluster. While Kubernetes works with generic representations of resources like attachable storage and load balancers, it needs a way to map these to the actual resources provided by non-homogeneous cloud providers.

Cloud controller managers act as the glue that allows Kubernetes to interact with providers with different capabilities, features, and APIs while maintaining relatively generic constructs internally. This allows Kubernetes to update its state information according to information gathered from the cloud provider, adjust cloud resources as changes are needed in the system, and create and use additional cloud services to satisfy the work requirements submitted to the cluster. For example, when a controller needs to check if a node was terminated or set up routes, load balancers, or volumes in the cloud infrastructure, all that is handled by the cloud controller-manager.

Kubernetes Worker Node Components

A worker node is a Kubernetes worker machine managed by the control plane, which can run one or more pods. The Kubernetes control plane is scheduling pods between nodes in the cluster. Worker nodes have a few requirements that are required for communicating with master components, configuring the container networking, and running the actual workloads assigned to them. Automatic scheduling in the control plane takes into account the resources available on each node and other constraints, such as affinity and taints, which define the desired running environment for different types of pods.

Below are the main components found on a Kubernetes worker node

Container Runtime

The first component that each node must have is a container runtime. Usually, this requirement is satisfied by installing and running Docker, but alternatives like rkt and runc are also available. The container runtime is responsible for starting and managing containers, applications encapsulated in a relatively isolated but lightweight operating environment. Each unit of work on the cluster is, at its basic level, implemented as one or more containers that must be deployed. The container runtime

on each node is the component that finally runs the containers defined in the workloads submitted to the cluster.

Kubelet

Kubelet is the main service that manages the container runtime (such as containerd or CRI-O). It controls the container runtime to launch or destroy containers as needed. The kubelet service regularly communicates with the master components to authenticate to the cluster and takes in new or modified pod specifications (primarily through the kube-apiserver). It ensures that pods and their containers are healthy and running in the desired state. This component also reports to the master on the health of the host where it is running.

Kube-Proxy

It is a proxy service that runs on each worker node to deal with individual host subnetting, make services available to other components, and expose services to the external world. It performs request forwarding to the correct pods/containers, does primitive load balancing, and is generally responsible for making sure the networking environment is predictable and accessible across the various isolated networks in a cluster.

Kubernetes Objects

Containers are the underlying mechanism used to deploy containerized applications. But Kubernetes uses additional layers of abstraction over the container interface to provide scaling, resiliency, and life cycle management features. Instead of managing containers directly, users define and interact with instances composed of various primitives provided by the Kubernetes object model.

PODS

A pod is the basic and smallest unit of management in a Kubernetes cluster. It represents one or more containers that constitute a functional component of an application. Pods consist of containers that operate closely together, share a life cycle, and should always be scheduled on the same node. They are managed entirely as a unit and share their environment, volumes, and IP space.

Usually, pods consist of a main container that performs the actual application functionality and, optionally, some helper containers that facilitate closely related tasks. These are programs that benefit from being run and managed in their own containers but are tightly tied to the main application. For example, a pod may have one container running the primary application server and a helper container pulling down files to the shared file system when changes are detected in an external repository. Horizontal scaling is generally discouraged at the pod level because there are other higher-level objects more suited for the task. It is not recommended for users to manage their own pods. Kubernetes suggests users work on higher-level objects that use pods or pod templates as base components but implement additional functionality.

Deployments

Deployments are high-level objects designed to ease the life-cycle management of replicated pods. Deployments can be updated easily by configuration changes, and Kubernetes will adjust the replica sets, manage transitions between different application versions, and optionally maintain event history and undo capabilities automatically.

Replication Controller

A replication controller is an object that defines a pod template and control parameters for horizontal scaling of the pods. The replication controller is responsible for ensuring that the number of pods deployed in the cluster matches the number of pods in its configuration. This helps with load distribution and availability within Kubernetes. If there is any failure, the controller will compensate by starting the new pods. If the number of replicas in a controller's configuration changes, the controller either starts up or kills containers to match the desired number. The replication controller performs rolling updates to roll over a set of pods to a new version one by one, minimizing the impact on application availability.

Replica Set

A Replica set is an improvement on the replication controller architecture. Replica sets provide the controller with more flexibility in identifying the pods that need to be managed. Due to their superior replica selection capabilities, replication sets are starting to take on the role of replication controllers; nevertheless, unlike replication controllers, replication sets cannot do rolling updates to upgrade backends to a new version. Replication sets are intended to be utilized instead of higher-level, supplementary units that offer that feature.

Stateful Sets

Stateful sets are specialized pod controllers that provide guaranteed ordering and uniqueness. These are mostly utilized for stable networking, persistent data, or deployment ordering. Stateful sets, for example, are frequently linked to data-oriented applications such as databases, which require access to the same volumes in the event of a rescheduling to a different node.

Daemon Sets

Daemon sets are a type of pod controller that run a copy of a pod on each worker node in the Kubernetes cluster. This is useful for deploying and performing maintenance pods.

For example, gathering and sending logs, combining metrics, etc. Daemon sets can skip scheduling rules, as they are needed throughout the application lifetime. For instance, due to its special duties, the master server is often set up to be unavailable for regular pod scheduling; however, daemon sets can overcome this restriction pod-by-pod in order to ensure that critical services continue to function.

Kubernetes services

A service in Kubernetes is an abstraction that represents a set of application logical pods that provide the same functionality. Actual pods are ephemeral; Kubernetes guarantees the availability of pods and replicas specified but not the liveliness of each individual pod. This means that other pods that need to communicate with this application or component cannot rely on the underlying individual pod's IP addresses.

A service has a cluster IP, which is a virtual IP address, and it exists until it is specifically terminated. The service acts as a reliable endpoint for inter-component or application communication as

the requests gets routed to the relevant pods. The end user would send the request to Kubernetes service using service name or IP, and those Requests would be forwarded to the backend pods. Requests can also be made through an API in Kubernetes' apiserver, which automatically exposes and maintains the real pod endpoints for applications.

There are four types of Kubernetes services: ClusterIP, NodePort, LoadBalancer, and ExternalName. The type property in the Service's specification determines how the service is exposed to the network. Let's discuss on each of them in detail.

Cluster IP

A ClusterIP service is the default type of service in Kubernetes. It creates a service inside the Kubernetes cluster that can be accessed by other applications in the cluster. Kubernetes will assign a cluster-internal IP address to ClusterIP service. One can optionally set cluster IP in the service definition file. Cluster IP services don't provide access to the external world.

Below is the sample YAML file

apiVersion: v1

kind: Service

metadata:

name: my-app-service

spec:

clusterIP: 10.20.30.40

selector:

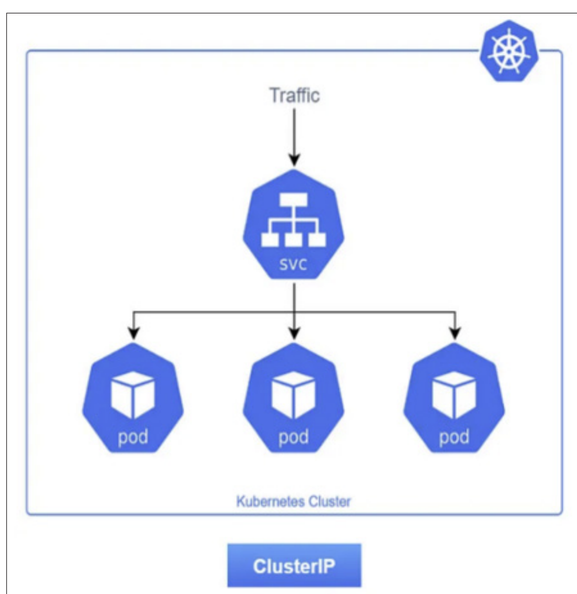
app.kubernetes.io/name: App

ports:

protocol: TCP

port: 80

targetPort: 9111



Usecases

- ClusterIP is mainly used for communication within a Kubernetes cluster. For example, communication between the front-end and back-end components of the application.

Nodeport

NodePort is an extension of ClusterIP service. It exposes the service outside of the cluster by adding a cluster-wide port on top of ClusterIP. Every node in the cluster configures itself to listen on that assigned port and forwards the traffic to one of the ready endpoints associated with that Service. Kubernetes will automatically assign a port, but it also lets the user provide a custom node port. Node port should be in the range of 30000–32767. Please make sure that the port a user is assigning is not in use by another service.

Below is the sample YAML file

apiVersion: v1

kind: Service

metadata:

name: my-app-service

spec:

type: NodePort

selector:

app.kubernetes.io/name: App

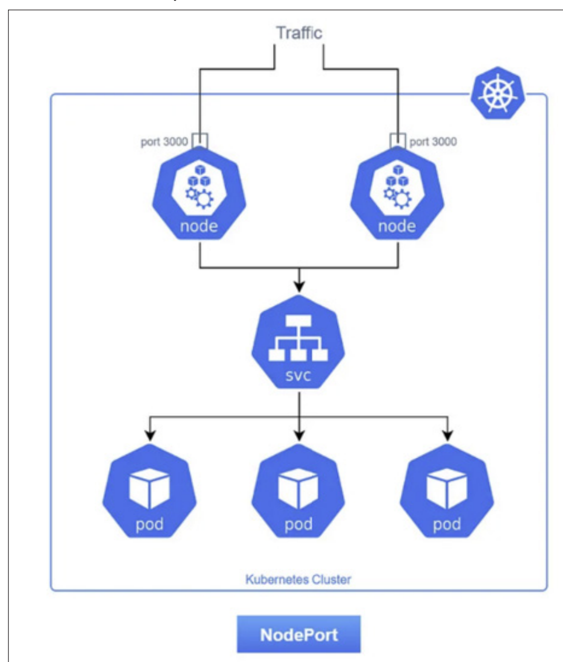
ports:

protocol: TCP

port: 80

targetPort: 9111

nodePort: 31000 #Optional field



Usecases

- When external connectivity is required for your service.
- Having a NodePort gives you the ability to set up a customized load balancing solution, to configure environments that are not fully supported by Kubernetes, or even to expose one or more nodes' IPs directly.

Loadbalancer

LoadBalancer is an extension of NodePort. A load balancer is a standard way to expose a Kubernetes service externally so it can be accessed over the internet. Any load balancer service would create NodePort and ClusterIP Services. LoadBalancer service integrates with cloud-based load balancers. The load balancer algorithm is specific to each cloud provider (AWS, Azure, GCP, etc.) implementation. The cloud provider will create a load balancer, which automatically routes requests to your Kubernetes Service. Every time a service needs to be exposed to the outside world, a new load balancer needs to be created and get an IP address.

Below is the sample YAML file

apiVersion: v1

kind: Service

metadata:

name: my-app-service

spec:

type: LoadBalancer

clusterIP: 10.0.30.40

loadBalancerIP: 172.38.228.23

selector:

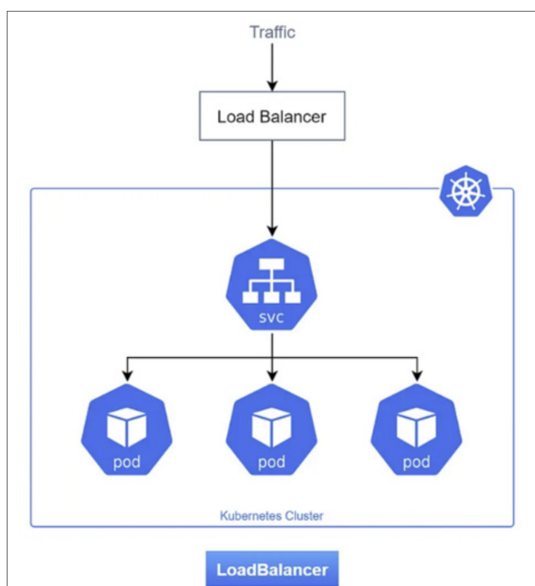
app.kubernetes.io/name: App

ports:

protocol: TCP

port: 80

targetPort: 9111



Usecases

- When you want to load balance a service,
- When you are using a cloud provider to host the Kubernetes cluster,

D. External Name

Services of type ExternalName are used to access an application outside of the Kubernetes cluster, like mapping a Service to a DNS name. It is not like a regular service where we use the selector of an endpoint object. You specify these Services with the `spec.externalName` parameter, which maps the Service to the contents of the externalName field (e.g., sample.demo.example.com) by returning a CNAME record with its value.

Below is the sample YAML file

apiVersion: v1

kind: Service

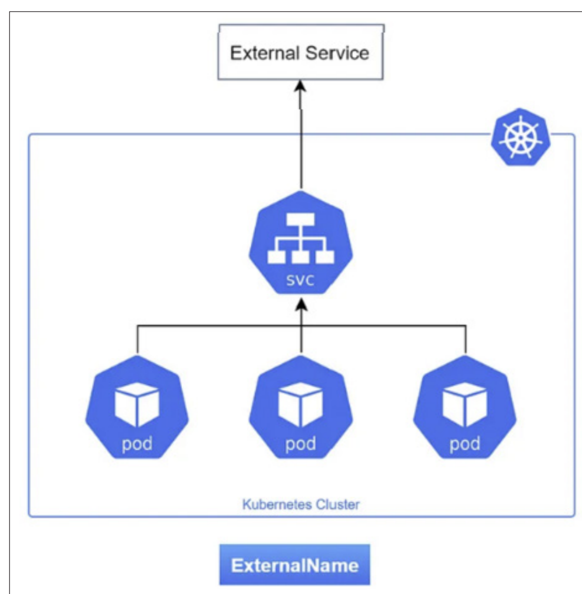
metadata:

name: my-app-service

spec:

type: ExternalName

externalName: my.db.sampleexample.com



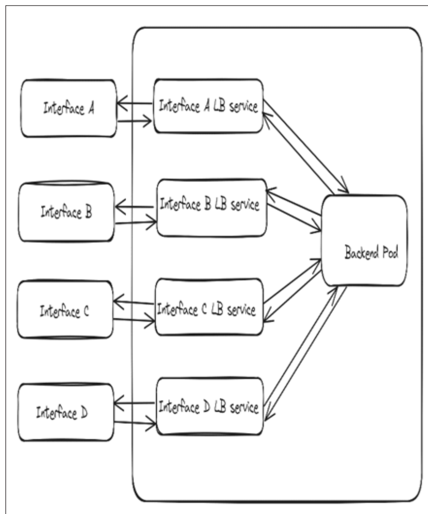
Usecases

- This is commonly used to create a service within Kubernetes representing an external datastore.
- This can also be created when pods from one namespace need to communicate with a service in another namespace.

Solution

Since we have learned about the different kinds of Kubernetes services, let's discuss the problem statement: a single service receiving requests from multiple interfaces is getting overloaded with too many requests. The better solution in this case would be to create a load balancer service. In our specific scenario, we have Four different interfaces: Interface A, Interface B, Interface C and Interface D. So, we have created four different load-balancer

services. All these services need to have an Ip configured and these services are mapped to the same backend pod as the source code, so the functionality that needs to be done is the same. These backend pods are configured such that if the cpu load reaches 50%, they would scale horizontally [1-5].



Conclusion

In scenarios where the service is getting overloaded, we can have a load balancer service. This would load-balance the traffic and resolve the bottleneck scenarios.

References

[1] https://www.aquasec.com/cloud-native-academy/kubernetes-101/kubernetes-complete-guide/.

[2] https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes.

[3] https://www.densify.com/kubernetes-autoscaling/kubernetes-service-load-balancer/.

[4] https://medium.com/devops-mojo/kubernetes-service-types-overview-introduction-to-k8s-service-types-what-are-types-of-kubernetes-services-ea6db72c3f8c.

[5] https://medium.com/google-cloud/kubernetes-from-load-balancer-to-pod-3f2399637b0c.