1

Outline

- Overview of Services
- Kubernetes ClusterIP vs NodePort vs LoadBalancer
- Accessing internal services
- Scaling up and scaling down the application replicas

2

Overview of Services

3

Kubernetes Networking Model

- All containers within a pod can communicate with each other unimpeded.
- All Pods can communicate with all other Pods without NAT.
- All nodes can communicate with all Pods (and vice-versa) without NAT.
- The IP that a Pod sees itself as is the same IP that others see it as.

4

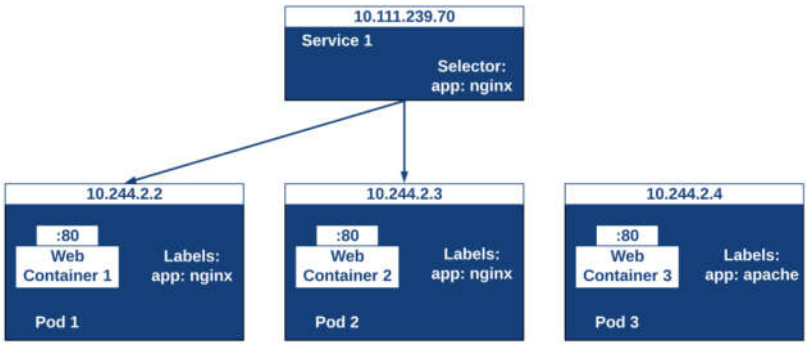
Fundamentals applied

- **Container-to-Container**
  - Containers within a pod exist within the same network namespace and share an IP.
  - Enables intra pod communication over localhost.
- **Pod-to-Pod**
  - Allocated cluster unique IP for the duration of its life cycle.
  - Pods themselves are fundamentally ephemeral
- **Pod-to-Service**
  - managed by **kube-proxy** and given a persistent cluster unique IP
  - exists beyond a Pod's lifecycle.
- **External-to-Service**
  - Handled by k**ube-proxy**.
  - Works in cooperation with a cloud provider or other external entity (load balancer).

5

Exposing Pods

- **We might be asking how the service knows which pods it should be providing a front end for ?**
  - A Kubernetes Service can select the pods it is supposed to abstract through a **label** selector.
  - The two pods have labels named "app: nginx" and the Service has a label selector that is looking for those same labels.



6

Services

- **Services give us a stable endpoint to connect to a pod or a group of pods**
  - Durable resource (unlike Pods)
    - static cluster-unique IP
    - static namespaced DNS name (**<servicename>.<namespace>.svc.cluster.local**)
  - Target Pods using equality based selectors
  - kube-proxy provides simple load-balancing.
- **An easy way to create a service is to use `kubectl expose`**
  - If we have a deployment named my-little-deploy, we can run:

    ```
    kubectl expose deployment my-little-deploy --port=80
    ```
  - ... and this will create a service with the same name (`my-little-deploy`)
  - Services are automatically added to an internal DNS zone
    - In the example above, our code can now connect to `http://my-little-deploy/`

7

Kubernetes ClusterIP vs NodePort vs LoadBalancer

8

Services and endpoints

- **A service has a number of "endpoints"**
- **Each endpoint is a host + port where the service is available**
  - Check the endpoints that Kubernetes has associated with our httpenv service:

```
kubectl describe service httpenv
```

- **Viewing endpoint details**
  - When we have many endpoints, our display commands truncate the list

```
kubectl get endpoints
```

  - If we want to see the full list, we can use one of the following commands:

```
kubectl describe endpoints httpenv
kubectl get endpoints httpenv -o yaml
```


9

The DNS zone

- **In the kube-system namespace, there should be a service named kube-dns**
- **This is the internal DNS server that can resolve service names**
- **The default domain name for the service we created is default.svc.cluster.local**
- **Get the IP address of the internal DNS server:**

```
IP=$(kubectl -n kube-system get svc kube-dns -o
jsonpath={.spec.clusterIP})
```

- **Resolve the cluster IP for the httpenv service:**

```
host httpenv.default.svc.cluster.local $IP
```


10

Service Types

- There are 4 major service types:
  1. ClusterIP (default)
  2. NodePort
  3. LoadBalancer
  4. ExternalName
- Services can also have optional external IPs
- There is also another resource type called Ingress (specifically for HTTP services)

11

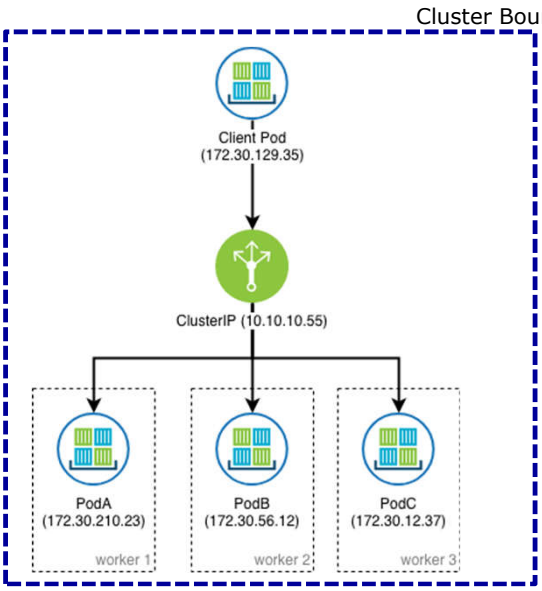
ClusterIP Services

- It is the default service type
- A virtual IP address is allocated for the service
- This IP address is reachable only from within the cluster (nodes and pods)
- Perfect for internal communication, within the cluster

```yaml
apiVersion: v1
kind: Service
metadata:
  name: example-prod
spec:
  type: ClusterIP
  selector:
    app: nginx
    env: prod
  ports:
  - protocol: TCP
    port: 8080
    targetPort: 80 # should meet the
                Container port
```


12

ClusterIP Services



13

NodePort Services

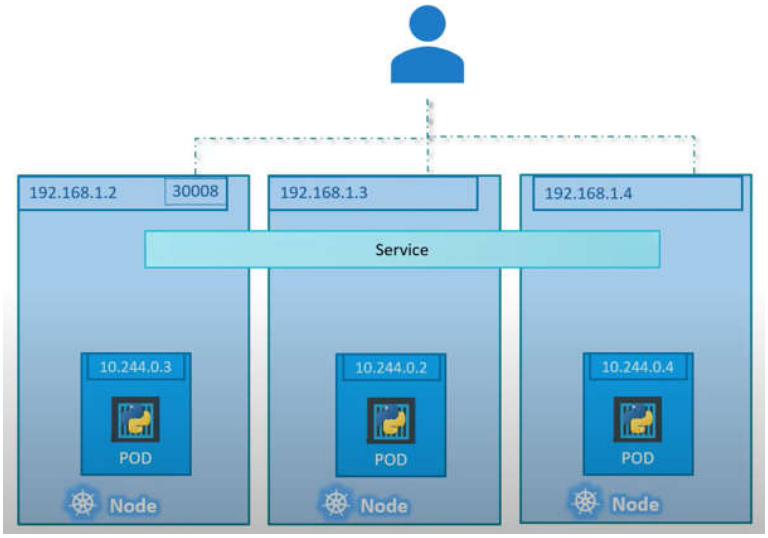
- **NodePort services extend the ClusterIP service.**
  - Exposes a port on every node's IP.
- **Port can either be statically defined, or dynamically taken from a range between 30000-32767.**

```yaml
apiVersion: v1
kind: Service
metadata:
  name: example-prod
spec:
  type: NodePort
  selector:
    app: nginx
    env: prod
  ports:
  - port: 80
    targetPort: 80
```


14

Exposing
and Scaling applications

NodePort services



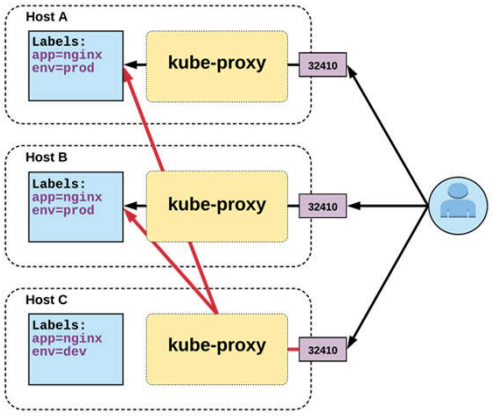
M.Romdhani, 2020

15

15

Exposing
and Scaling applications

NodePort Services




```
Name:        example-prod
Selector:    app=nginx,env=prod
Type:        NodePort
IP:          10.96.28.176
Port:        <unset>  80/TCP
TargetPort:  80/TCP
NodePort:    <unset>  32410/TCP
Endpoints:   10.255.16.3:80,
             10.255.16.4:80
```

- User can hit any host in cluster on NodePort IP and get to service
- Does introduce extra hop if hitting a host without instance of the pod

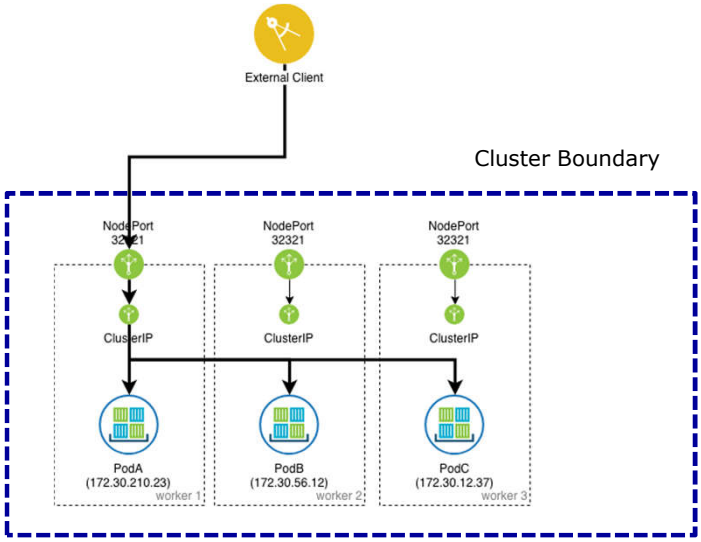
M.Romdhani, 2020

16

16

Exposing
and Scaling applications

NodePort Services



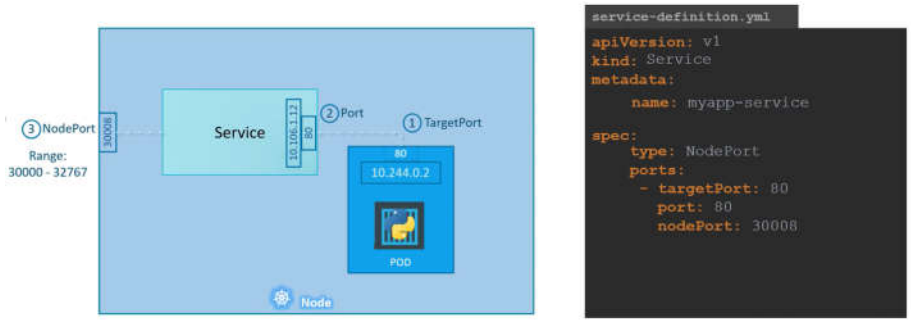
M.Romdhani, 2020

17

17

Exposing
and Scaling applications

nodePort, port, targetPort



M.Romdhani, 2020

18

18

Exposing and Scaling applications

LoadBalancer Services

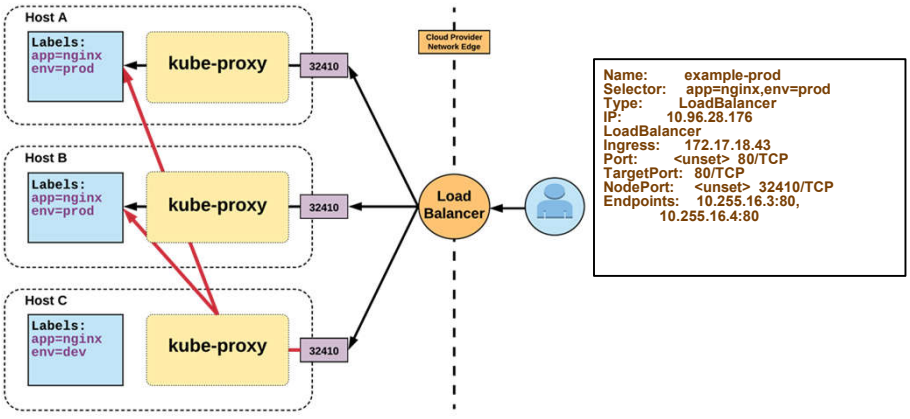
- LoadBalancer services extend NodePort.
- Works in conjunction with an external system to map a cluster external IP to the exposed service (typically a cloud load balancer, e.g. ELB on AWS, GLB on GCE ...)

```
apiVersion: v1
kind: Service
metadata:
  name: example-prod
spec:
  type: LoadBalancer
  selector:
    app: nginx
    env: prod
  ports:
    protocol: TCP
    port: 80
    targetPort: 80
```


19

Exposing and Scaling applications

LoadBalancer Services

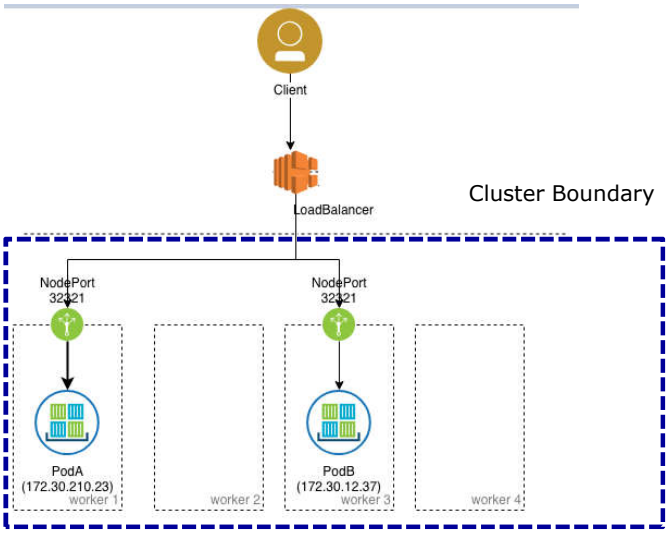


- Gets external IP from provider

20

Exposing
and Scaling applications

LoadBalancer services



21

Exposing
and Scaling applications

ExternalName Services

- **Services of type ExternalName are quite different**
- **No load balancer (internal or external) is created**
- **Only a DNS entry gets added to the DNS managed by Kubernetes**
- **That DNS entry will just be a CNAME to a provided record**
- **Example:**
  - Create a CNAME k8s pointing to kubernetes.io

```
kubectl create service externalname k8s
        --external-name kubernetes.io
```

  - Check the CNAME

```
nslookup k8s.default.svc.cluster.local
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: example-prod
spec:
  type: ExternalName
  selector:
    app: nginx
    env: prod
  ports:
    protocol: TCP
    port: 80
    targetPort: 80
```


22

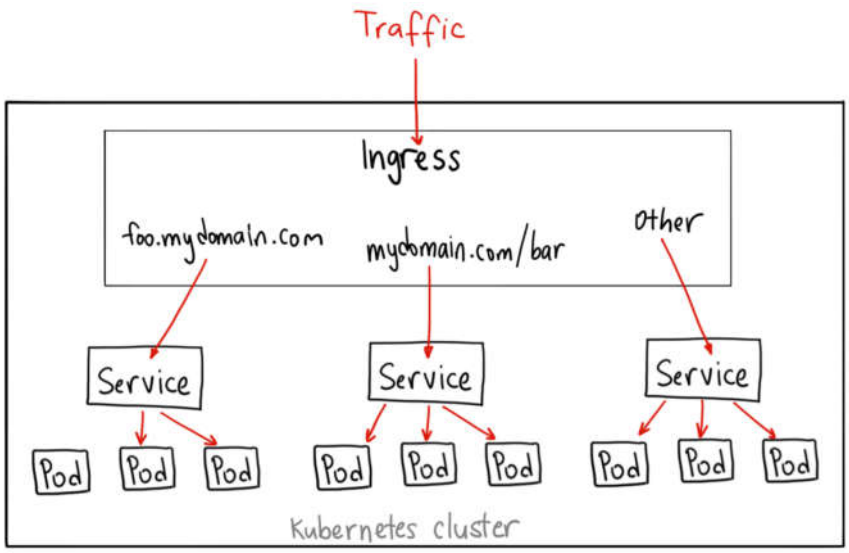
Ingres

- **An Ingres is an API object that manages external access to the services in a cluster**
  - Provides load balancing, SSL termination and name/path-based virtual hosting
  - Gives services externally-reachable URLs
- **They are specifically for HTTP services(not TCP or UDP)**
- **They can also handle TLS certificates, URL rewriting ...**

```
# Path based  routing Examle
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: simple-fanout-example
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        backend:
          serviceName: service1
          servicePort: 4200
      - path: /bar
        backend:
          serviceName: service2
          servicePort: 8080
```


23

Ingess



24

Accessing Internal Services

25

Accessing internal services

- **When we are logged in on a cluster node, we can access internal services**
  - As per the Kubernetes network model: all nodes can reach all pods and services)
- **When we are accessing a remote cluster, our local machine won't have access to the cluster's internal subnet. To overcome this:**
  - **kubectl proxy**: gives us access to the API, which includes a proxy for HTTP resources
  - **kubectl port-forward**: allows forwarding of TCP ports to arbitrary pods, services, ...

26

kubectl proxy

- **Running `kubectl proxy` gives us access to the entire Kubernetes API**
  - The API includes routes to proxy HTTP traffic
  - By default, the proxy listens on port 8001
- **These routes look like the following:**
  - `/api/v1/namespaces/<namespace>/services/<service>/proxy`
- **We just add the URI to the end of the request, for instance:**
  - `/api/v1/namespaces/<namespace>/services/<service>/proxy/index.html`
- **We can access services and pods this way !**
- ***Security considerations : kubectl proxy is intended for local use***
  - Running kubectl proxy openly is a huge security risk
  - It is slightly better to run the proxy where you need it (and copy credentials, e.g. ~/.kube/config, to that place)
  - It is even better to use a limited account with reduced permissions

27

kubectl port-forward

- **What if we want to access a TCP service?**
  - We can use **kubectl port-forward** instead
  - It will create a TCP relay to forward connections to a specific port (of a pod, service, deployment...)
- **The syntax is:**

```
kubectl port-forward service/name_of_service local_port:remote_port
```

  - If only one port number is specified, it is used for both local and remote ports

28

Scaling up and scaling down the application replicas

29

Scaling Up and Down Applications

- **Scaling out a Deployment will ensure new Pods are created and scheduled to Nodes with available resources.**
  - Scaling will increase the number of Pods to the new desired state. Kubernetes also supports autoscaling of Pods
- **You can scale a Deployment by using the following command:**

```
kubectl scale deployment nginx-deployment --replicas=10
```

- **To check the deployment, use the get command**

```
kubectl get deployments nginx-deployment
```

- **Running multiple instances of an application will require a way to distribute the traffic to all of them.**
  - Services have an integrated load-balancer that will distribute network traffic to all Pods of an exposed Deployment.
  - Services will monitor continuously the running Pods using endpoints, to ensure the traffic is sent only to available Pods.

30

What is the Horizontal Pod Autoscaler, or HPA?

- **It is a controller that can perform horizontal scaling automatically**
- **Horizontal scaling**
  - Changing the number of replicas (adding/removing pods)
- **Vertical scaling**
  - Changing the size of individual replicas (increasing/reducing CPU and RAM per pod)
- **Cluster scaling**
  - Changing the size of the cluster (adding/removing nodes
- **HPA's Principle of operation**
  - Each HPA resource (or "policy") specifies:
    - which object to monitor and scale (e.g. a Deployment, ReplicaSet...)
    - min/max scaling ranges (the max is a safety limit!)
    - a target resource usage (e.g. the default is CPU=80%)
  - The HPA continuously monitors the CPU usage for the related object
  - It scales the related object up/down to this target number of pods

31

Creating Horizontal Pod Autoscaler

- **Imperative style**

kubectl autoscale deployment nginx-deployment --cpu-percent=50 --min=1 --max=10

  - This command creates an Horizontal Pod Autoscaler that maintains between 1 and 10 replicas of the Pods controlled by the php-apache deployment.
  - Roughly speaking, HPA will increase and decrease the number of replicas (via the deployment) to maintain an average CPU utilization across all Pods of 50%

- **The declative style use the object HorizontalPodAutoscaler**

```yaml
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: nginx
spec:
  maxReplicas: 10
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx
  metrics:
  - type: Resource
    resource:
      name: cpu
      targetAverageUtilization: 50
  - type: Resource
    resource:
      name: memory
      targetAverageValue: 100Mi
```


32