**Distributed systems and ectd alternatives…**

Distributed. Kubernetes architecture is distributed, which means each platform component has a well-defined role and clear mechanism of communication (via API). It is can run on multiple machines, which makes it more resilient and fault-tolerant.

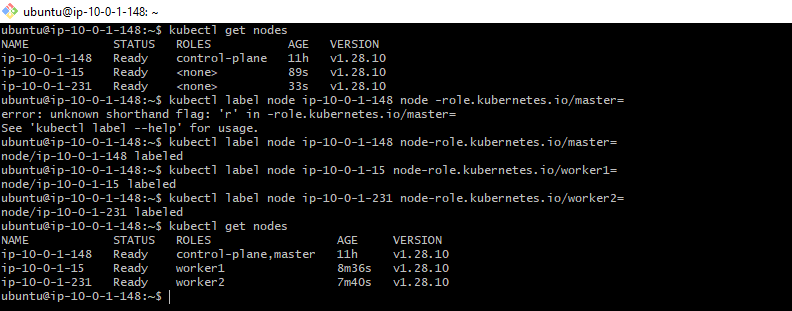
**Types of Distributed Systems**

* Client/Server Systems.
* Peer-to-Peer Systems.
* Middleware.
* Three-tier.
* N-tier.

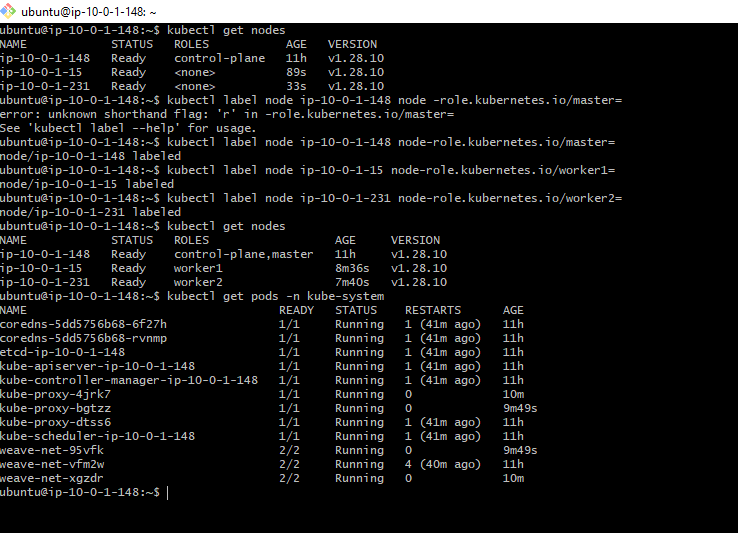
In Kubernetes, etcd is the default key-value store used for storing cluster state and configuration data. However, there are alternatives to etcd that can be used depending on specific requirements. Here are a few alternatives:

1. **Consul**: Consul is a distributed service mesh and key-value store designed for modern, elastic architectures. It provides features like service discovery, health checking, and distributed key-value storage. Consul can be integrated with Kubernetes using its official Helm chart or by deploying it as a standalone service and using its DNS interface for service discovery.
2. **ZooKeeper**: ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. While not as commonly used with Kubernetes as etcd or Consul, ZooKeeper can still be integrated using its client libraries and Kubernetes primitives like ConfigMaps.
3. **Redis**: Redis is an open-source, in-memory data structure store used as a database, cache, and message broker. While it's not designed specifically for Kubernetes, Redis can be used as a key-value store in Kubernetes clusters. However, it may not provide the same level of distributed consensus and fault tolerance as dedicated solutions like etcd, Consul, or ZooKeeper.
4. **SQLite**: SQLite is a self-contained, serverless, zero-configuration, transactional SQL database engine. While typically used in embedded systems or as an application file format, SQLite could be used as a lightweight alternative for storing configuration data in Kubernetes, particularly in development or testing environments.
5. **HTTP Status Codes**: HTTP status codes are standardized responses used by web servers to indicate the outcome of a request. They are three-digit numerical codes, grouped into five categories:
   1. 1xx: Informational responses
   2. 2xx: Success responses
   3. 3xx: Redirection responses
   4. 4xx: Client error responses
   5. 5xx: Server error responses Common examples include 200 (OK), 404 (Not Found), and 500 (Internal Server Error).
6. **HTTP Request Methods**: HTTP defines several request methods (also known as verbs) to indicate the desired action to be performed on a resource. Some common request methods include:
   1. GET: Retrieve data from the server.
   2. POST: Submit data to be processed by the server.
   3. PUT: Update a resource on the server.
   4. DELETE: Remove a resource from the server.
   5. PATCH: Apply partial modifications to a resource.
   6. OPTIONS: Get information about the communication options available for the target resource.
   7. HEAD: Retrieve metadata about the resource without the response body. These methods define the operations that can be performed on web resources.
7. **Configure Readiness Probe**: In Kubernetes, a readiness probe is used to determine if a container is ready to serve traffic. It is configured in the pod specification and periodically checks the health of the container. If the readiness probe fails, the container is removed from service until it becomes healthy again. Readiness probes are crucial for ensuring that only healthy containers receive traffic, preventing service disruptions due to unhealthy containers.
8. **CrashLoopBackOff Phase**: In Kubernetes, the CrashLoopBackOff phase indicates that a pod has crashed and is continuously restarting in a loop due to a persistent error. This error could be related to issues such as misconfiguration, resource constraints, or application bugs. Kubernetes automatically restarts the pod in an attempt to recover from the failure, but if the underlying issue persists, the pod remains stuck in the CrashLoopBackOff phase.
9. **ImagePullErr Phase**: The ImagePullErr phase occurs when Kubernetes is unable to pull the container image specified in the pod specification. This could happen due to various reasons, such as incorrect image name or tag, authentication issues with the container registry, or network connectivity problems. When a pod enters the ImagePullErr phase, Kubernetes will continuously attempt to pull the image until it succeeds or reaches the maximum number of retries. If the image cannot be pulled successfully, the pod remains in the ImagePullErr phase and is unable to star

**Kubernetes Cluster setup …**

****

**Kubernetes systems**

****

**Setup horizontal autoscaling in kubernetes…**

Here's a step-by-step guide to setting up horizontal autoscaling in Kubernetes:

1. **Enable Metrics Server**: The Horizontal Pod Autoscaler requires metrics from the Metrics Server to function. Ensure that the Metrics Server is installed and running in your cluster. You can install it using **kubectl apply -f**
2. **Deploy Your Application**: You need to have a deployment, replica set, or replication controller running your application.

**Deploy Your Application**: You need to have a deployment, replica set, or replication controller running your application

**apiVersion: apps/v1**

**kind: Deployment**

**metadata:**

**name: my-app**

**spec:**

**replicas: 3**

**selector:**

**matchLabels:**

**app: my-app**

**template:**

**metadata:**

**labels:**

**app: my-app**

**spec:**

**containers:**

**- name: my-app**

**image: my-image:latest**

**resources:**

**limits:**

**cpu: 500m**

**requests:**

**cpu: 200m**

**Create a Horizontal Pod Autoscaler (HPA)**: Define an HPA object that specifies the scaling behavior. You can create an HPA using a YAML manifest or by using **kubectl autoscale** command.

**apiVersion: autoscaling/v2beta2**

**kind: HorizontalPodAutoscaler**

**metadata:**

**name: my-app-hpa**

**spec:**

**scaleTargetRef:**

**apiVersion: apps/v1**

**kind: Deployment**

**name: my-app**

**minReplicas: 3**

**maxReplicas: 10**

**metrics:**

**- type: Resource**

**resource:**

**name: cpu**

**target:**

**type: Utilization**

**averageUtilization: 50**

1. the HPA scales the number of replicas of the "my-app" deployment based on CPU utilization. It ensures that the average CPU utilization across all pods remains at or below 50%.
2. **Apply the HPA Configuration**: Apply the HPA configuration to your Kubernetes cluster using **kubectl apply -f hpa.yaml**.
3. **Monitor Autoscaling**: Once the HPA is applied, Kubernetes will monitor the CPU utilization of the pods and adjust the number of replicas accordingly. You can monitor the HPA status using **kubectl get hpa**.