Introduction to Kubernetes!

Savitha Raghunathan

Introduction

Savitha Raghunathan

- Senior Software Engineer @ Red Hat
- Kubernetes Contributor
- CNCF Ambassador

Agenda

Part 1: Containers & Docker

Part 2: Kubernetes

Part 3: Kubernetes Architecture

Part 4: Kubernetes Components

Part 5: Hands-On Demo

Part 6: Wrap-Up and Q&A

Part 1: Containers & Docker

Traditional Virtual Machines and Local Setups

Traditional virtual machines (VMs) virtualize an entire operating system, and running applications locally on a single OS can lead to conflicts.

Disadvantages:

- Heavy Resource Usage: VMs replicate the whole operating system, consuming substantial resources (RAM, CPU) on each VM instance.
- Lack of Consistency: Running apps locally can result in version conflicts (dependencies, environment variables)
- Inefficient Scaling: Spinning up a new VM is slower and requires more storage

What are Containers?

Containers package an application with its dependencies, letting it run consistently across any environment.

Key Points:

- Containers occupy a smaller footprint compared traditional VMs.
- Containers allow consistency by packaging everything the application needs.
- Containers can be moved between environments without issues.

Why Use Containers?

Benefits:

- Portability: Containers can work across all computers and servers.
- Efficiency: Containers only carry what's needed for the app, making them smaller and faster than full virtual machines.
- Resource Isolation: Containers are independent, preventing conflicts.

Introduction to Docker

Docker is the tool used to create and manage containers.

Key Commands:

docker build: Assembles the container image.

docker run: Runs the container.

docker pull: Downloads Docker images from the Docker Hub.

docker ps: Lists running containers, providing information about their status and configurations.

Running a Container with Docker

Run an Nginx container:

```
docker run --name my-nginx -p 8080:80 nginx
```

Open a browser and visit localhost:8080 to see the Nginx welcome page.

Running a Container with Docker

```
docker run --name my-nginx -p 8080:80 nginx
```

- --name my-nginx: Names the container instance my-nginx.
- -p 8080:80: Maps port 80 in the container to port 8080 on the host, making the application accessible locally.
- nginx: Specifies the image to use (in this case, the latest Nginx image from Docker Hub).

Part 2: Kubernetes

Why Do We Need Kubernetes?

Complexity at Scale: Managing multiple containers across distributed environments is resource-intensive and complex.

- Scaling Needs: Adjusting container numbers manually to match traffic spikes or dips requires significant oversight.
- Configuration Drift: Without consistent orchestration, container configurations can drift across environments, causing reliability issues.

Why Do We Need Kubernetes?

High Availability & Resilience:

- Fault Tolerance: Detecting and recovering from container or node failures promptly is crucial to avoid downtime.
- Service Discovery: Ensuring containers can locate each other (especially when they're dynamically scheduled across nodes) is challenging without orchestration.

Why Do We Need Kubernetes?

Load Distribution:

- Traffic Management: Balancing incoming traffic across containers to avoid overload or resource wastage is difficult to manage manually.

Kubernetes Overview

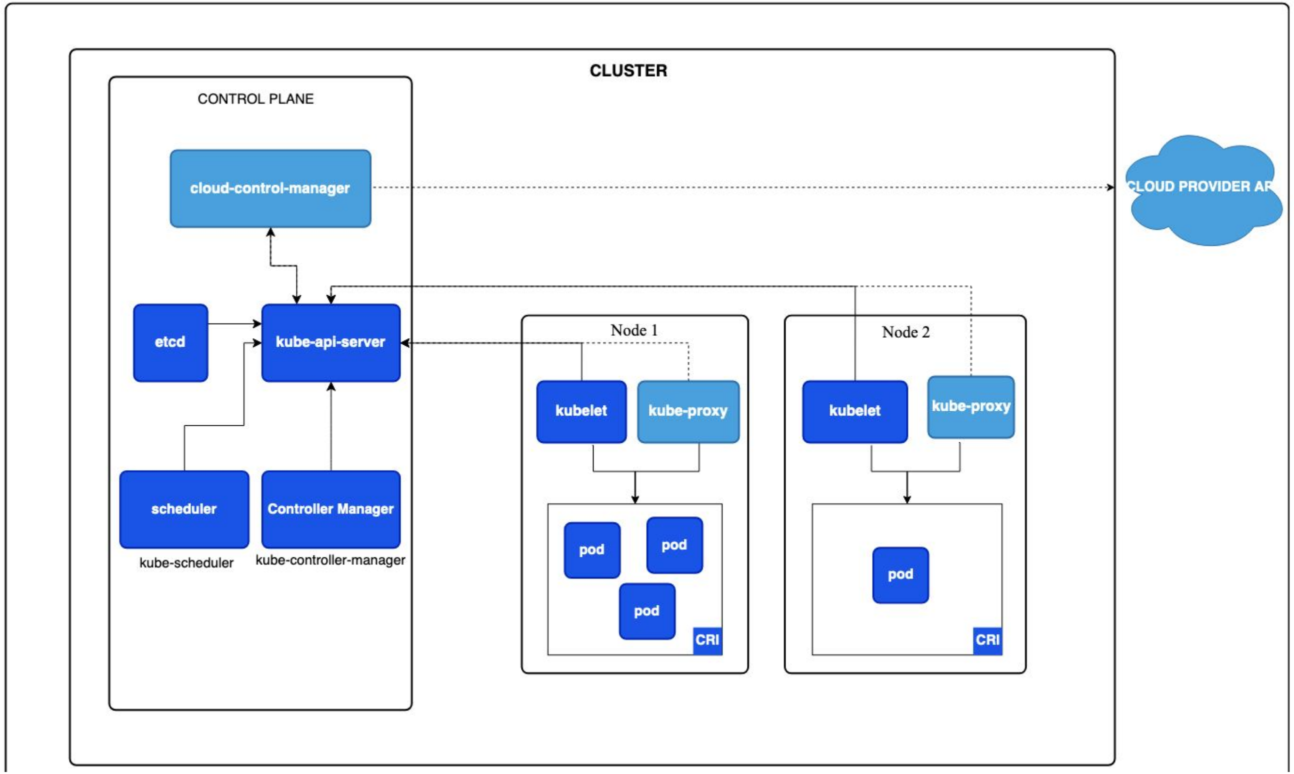
An orchestration platform that automates the deployment, scaling, and management of containerized applications.

Main Responsibilities:

- Scaling: Automatically adjusts the number of running containers based on real-time demand, adding or removing instances as needed.
- Load Balancing: Distributes incoming network traffic evenly across containers, ensuring efficient resource utilization and application performance.
- Self-Healing: Monitors container health, automatically replacing or restarting containers that fail, maintaining application availability and reliability.

Part 3: Kubernetes Architecture

Kubernetes Architecture



Pic credits: https://kubernetes.io/docs/concepts/architecture/

Kubernetes Control Plane

Control Plane: The core management layer of Kubernetes that maintains the desired state of the cluster.

- API Server: The central component that exposes the Kubernetes API and serves as the communication point for all Kubernetes operations.
- Scheduler: Assigns Pods to available nodes based on resource requirements and policies.
- Controller Manager: Ensures the actual state matches the desired state, handling tasks such as maintaining replicas and managing node lifecycles.
- etcd: A highly available key-value store that stores all cluster data, configurations, and states.

API Server

The API Server (kube-apiserver) is the main communication point for the entire Kubernetes cluster.

- It handles all REST API requests from users, CLI tools (kubectl), and other Kubernetes components, serving as the front-end interface for interacting with the cluster.
- Responsibilities:
    - Validates and processes REST requests.
    - Acts as the gatekeeper for all operations (CRUD) on Kubernetes objects.
- Security: It enforces authentication, authorization, and admission control policies, ensuring secure interactions with the cluster.

Scheduler

The Scheduler (kube-scheduler) is responsible for assigning new Pods to nodes within the cluster based on resource availability and scheduling policies.

Responsibilities:

- Monitors unscheduled Pods and assigns them to nodes with sufficient resources (CPU, memory).
- Applies scheduling rules, such as affinity and anti-affinity (e.g., ensuring two specific Pods are scheduled together or kept separate).
- Optimizes workload distribution to maintain efficient resource usage and avoid overloading nodes.

Decision Criteria: The Scheduler considers factors like resource requirements, node capacity, and existing workloads to make scheduling decisions.

Controller Manager

The Controller Manager (kube-controller-manager) oversees various controller processes that regulate and reconcile the state of Kubernetes objects.

Responsibilities:

- Node Controller: Monitors node health and reacts if nodes become unresponsive.
- Replication Controller: Ensures the specified number of replicas are running for each application.
- Endpoint Controller: Manages endpoint objects and links Services to Pods.
- Service Account & Token Controller: Manages default accounts and API tokens for newly created namespaces.

Reconciliation Loop: The Controller Manager constantly compares the desired state with the current state and makes adjustments as needed, following a control loop process.

etcd

etcd is a distributed key-value store that serves as Kubernetes' primary data store, holding all cluster configuration data, metadata, and the state of the entire system.

Responsibilities:

- Stores configuration data such as API objects, deployment details, and cluster states.
- Provides high availability with data replication, ensuring that the cluster's state is accessible across multiple master nodes.

Data Consistency: Since Kubernetes heavily relies on etcd for storing the cluster's desired state, it's critical to back up etcd regularly to prevent data loss.

Kubernetes Worker Nodes

Worker Nodes: Nodes that run containerized applications and provide compute capacity to the cluster.

- Kubelet: An agent that manages Pod operations on each node, ensuring containers are running and healthy.
- Kube-proxy: A network proxy that manages network communication, routing traffic to the appropriate Pods within the cluster.

Kubelet

The Kubelet (kubelet) is an agent that runs on every worker node, acting as the primary node-level controller.

Responsibilities:

- Receives Pod definitions from the API Server and ensures they are running as specified.
- Monitors container health and reports node status back to the control plane.
- Works with container runtimes (e.g., Docker, containerd) to manage container lifecycle events like starting, stopping, and restarting containers.

Node Monitoring: The Kubelet continually monitors the state of Pods and containers, sending status updates back to the API Server.

Kube-proxy

The Kube-proxy is a network proxy that runs on each worker node, managing networking rules to allow seamless communication between services and Pods within the cluster.

Responsibilities:

- Configures IP tables or IPVS rules to route traffic to the correct Pods based on Service endpoints.
- Balances traffic across multiple Pods within a Service, enabling load balancing and ensuring even resource utilization.
- Facilitates communication between internal and external clients by handling NodePort or LoadBalancer configurations, making services accessible outside the cluster.

Traffic Routing: The Kube-proxy is essential for forwarding and routing traffic to the appropriate endpoints, ensuring Kubernetes applications remain accessible.

Container Runtime

The container runtime is the underlying software responsible for running containerized applications on each worker node. Common runtimes include Docker, containerd, and CRI-O.

Responsibilities:

- Pulls container images from repositories, creates containers, and executes them within Pods.
- Provides the environment required for running applications within isolated containers.

CRI (Container Runtime Interface): Kubernetes supports multiple container runtimes through the CRI, enabling flexibility and compatibility with different runtime technologies.

Part 4. Kubernetes Concepts

Pods - The Basic Unit

A Pod is the smallest deployable unit in Kubernetes, consisting of one or more tightly coupled containers that share storage, network, and a single IP address.

Key Features:

- Shared Resources: Containers within a Pod share network interfaces and storage volumes.
- Use Case: Often used to run a single container, but can contain multiple containers that need to work closely together.

Example: A web application Pod might contain both the main web server container and a sidecar container for logging or monitoring.

Deployments

A Deployment is a Kubernetes object that defines and manages the desired state for a set of Pods, specifying the number of replicas, the container image version, and the update strategy.

Key Features:

- Scaling: Automatically increases or decreases the number of replicas based on demand.
- Rolling Updates: Ensures zero-downtime updates by gradually replacing old versions of Pods with new ones.
- Self-Healing: Automatically replaces failed Pods to maintain the specified replica count.

Example: A Deployment for a front-end application might specify three replicas, ensuring that three instances of the application are always available.

Services

A Service is a Kubernetes resource that defines a stable IP and DNS name for a set of Pods, enabling communication between different parts of an application or external access to Pods.

Types of Services:

- ClusterIP: Exposes the Service within the cluster, allowing communication between internal Pods.
- NodePort: Exposes the Service on a static port on each node, allowing external access.
- LoadBalancer: Provisions an external load balancer, useful for cloud-based applications.

Example: A Service can expose a group of web server Pods to other Pods within the cluster or to external users, ensuring continuous access even if individual Pods are replaced or rescheduled.

Part 5: Kubernetes in Action

Demo

- Deploy a cluster using Kind
- Create an nginx webserver deployment
- Expose the deployment using a Service object
- Scale the Deployment
- Delete a pod and watch the deployment

Part 6: Resources and Q/A

Recap & Further Resources

- Containers: Provide consistent, portable application packaging by bundling code and dependencies. They enable reliable application deployment across diverse environments.
- Kubernetes: An orchestration platform that automates container management, ensuring scalability, fault tolerance, and efficient resource allocation for containerized applications.
    - Kind: A lightweight tool for running Kubernetes clusters locally, ideal for development, testing, and hands-on learning with Kubernetes.

Recap & Further Resources

- Docker Tutorials: Resources for understanding Docker and containerization fundamentals – https://www.docker.com/101-tutorial/

- Kubernetes Documentation: In-depth guides on Kubernetes concepts, architecture, and components – kubernetes.io/docs

- Kind Documentation: Learn more about setting up and managing local clusters – https://kind.sigs.k8s.io/docs/user/quick-start/

Recap & Further Resources

- Link to slides: https://github.com/savitharaghunathan/101-kubernetes


Q & A

THANK YOU :) :)