Assignment 3.2: Container Orchestration

Lab Group 32

Ivy Rui Wang (iwa211), Delong Yuan (dyu208)

I. Project overview

This project was deployed on three virtual machines provided by the University of Amsterdam, utilizing the latest Kubernetes version 1.29.2. The environment setup included installing Docker and Kubernetes on each machine, followed by node configuration with one master node and two worker nodes.

A. Application deployment

1) **MySQL Database:** Used for storing user login and URL mapping data.
2) **Nginx Reverse Proxy Server:** Facilitated unified access to all services through port 30000.
3) **URL Shortening Service and User Authentication Service:** Deployed with three replica pods across two worker nodes.
4) **Kubernetes Secrets:** Used to securely store database credentials.
5) **Persistent Storage:** Implemented using PersistentVolume (PV) and PersistentVolumeClaim (PVC) for data persistence.
6) **Deployment and Service:** Utilized for the MySQL database, URL shortening service, user authentication service, and Nginx.

B. Implementation Highlights

1) **Unified Access Point:** The Nginx reverse proxy simplifies user access by providing a single entry point.
2) **Data Security and Persistence:** Secure storage of database credentials using Secrets and persistent data storage using PV and PVC ensure reliability and security.
3) **Scalable and Resilient Architecture:** Deployment of replicated pods across multiple nodes ensures that the applications are fault-tolerant and available.

II. Setting Up K8S Environment

For this assignment, we utilized three virtual machines provided by the University of Amsterdam, accessed via remote SSH to deploy Kubernetes (k8s), with the latest version being **1.29.2**. The first step involved installing Docker and Kubernetes on all three machines. Then, we proceeded to configure the nodes. In setting up a Kubernetes cluster, nodes are generally classified into two types: master and worker nodes. On the master node, the command *kubeadm init* was used to initialize the cluster's control plane. Upon completion, there was an output with instructions on how to join worker nodes to the cluster. Then, each worker node was joined to the cluster using the *kubeadm join* command.

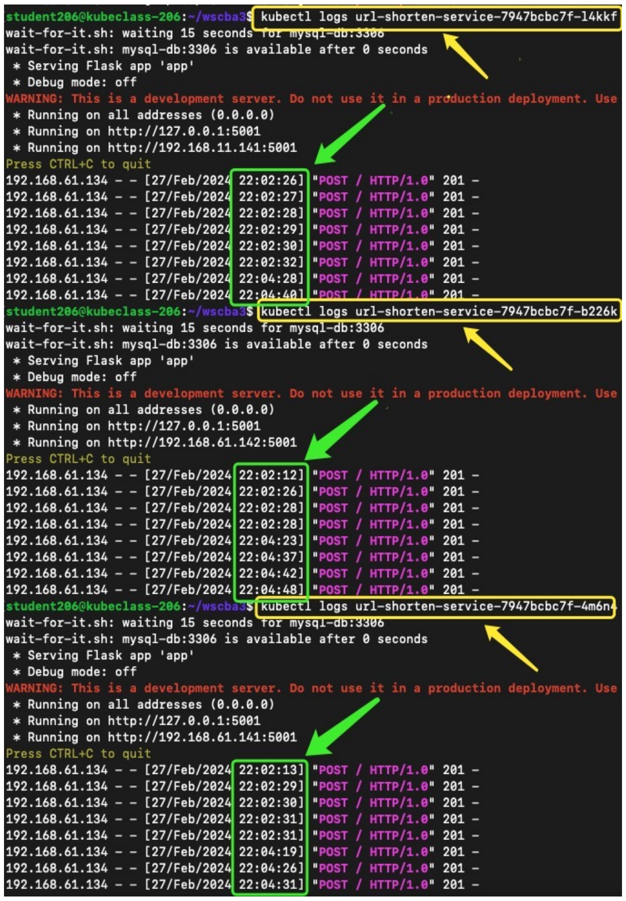
We encountered several challenges. Some commands in the guide were obsolete due to updates. Instead of downgrading our cluster, we decided to troubleshoot through trial and error. Eventually, we identified the issues were due to outdated configuration parameters and missing necessary configuration files. After adjustments, we successfully deployed a three-node Kubernetes cluster. Two essential configuration files are needed in the latest version: one for configuring *containerd* with *config.toml* (Kubernetes documentation on container runtimes), where we needed to configure the *systemd cgroup driver* by switching *SystemdCgroup=false* to *SystemdCgroup=true*. Another necessary file was for Calico network manager configuration (Calico troubleshooting documentation), which involved creating a */etc/NetworkManager/conf.d/calico.conf* file and copying the content from the documentation. This NetworkManager must be configured before attempting to use Calico networking. These files had to be configured on all nodes to prevent *kubeadm init* failures (error of connection refused at port 6443) and Calico failures.

III. Observations for load balancing

When a request reaches a Kubernetes Service, the Service decides which pod replica will handle the request based on Kubernetes' internal load balancing mechanisms. A Kubernetes Service acts as an abstraction over the pod replicas, providing a stable IP address and port number. Through this address and port, the Service can forward incoming requests to any of the backend pod replicas. The key to understanding this process lies in how Kubernetes selects a specific pod replica to handle the request.

To observe how requests are distributed across pod replicas in a Kubernetes environment, we sent multiple requests to the Service in succession and used *kubectl get pods* to list all related pods, and used *kubectl logs* to retrieve logs for each pod to see which pod handles which requests.

Figure 1 demonstrates the process of sending a series of POST requests to the URL shortening service and then using 'kubectl logs' to inspect the timestamps at which each pod processed the requests. By examining the timestamps for request handling across different pods, it's observable that each consecutive request is allocated to a different pod than the previous one. This pattern suggests that the Kubernetes Service responsible for routing traffic to the URL shortening service



```
student206@kubeclass-206:~/wscba3$ kubectl logs url-shorten-service-7947bcbc7f-l4kkf
wait-for-it.sh: waiting 15 seconds for mysql-db:3306
wait-for-it.sh: mysql-db:3306 is available after 0 seconds
 * Serving Flask app 'app'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5001
 * Running on http://192.168.11.141:5001
Press CTRL+C to quit
192.168.61.134 - - [27/Feb/2024 22:02:26] "POST / HTTP/1.0" 201 -
192.168.61.134 - - [27/Feb/2024 22:02:27] "POST / HTTP/1.0" 201 -
192.168.61.134 - - [27/Feb/2024 22:02:28] "POST / HTTP/1.0" 201 -
192.168.61.134 - - [27/Feb/2024 22:02:29] "POST / HTTP/1.0" 201 -
192.168.61.134 - - [27/Feb/2024 22:02:30] "POST / HTTP/1.0" 201 -
192.168.61.134 - - [27/Feb/2024 22:02:32] "POST / HTTP/1.0" 201 -
192.168.61.134 - - [27/Feb/2024 22:04:28] "POST / HTTP/1.0" 201 -
192.168.61.134 - - [27/Feb/2024 22:04:40] "POST / HTTP/1.0" 201 -
student206@kubeclass-206:~/wscba3$ kubectl logs url-shorten-service-7947bcbc7f-b226k
wait-for-it.sh: waiting 15 seconds for mysql-db:3306
wait-for-it.sh: mysql-db:3306 is available after 0 seconds
 * Serving Flask app 'app'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5001
 * Running on http://192.168.61.142:5001
Press CTRL+C to quit
192.168.61.134 - - [27/Feb/2024 22:02:12] "POST / HTTP/1.0" 201 -
192.168.61.134 - - [27/Feb/2024 22:02:26] "POST / HTTP/1.0" 201 -
192.168.61.134 - - [27/Feb/2024 22:02:28] "POST / HTTP/1.0" 201 -
192.168.61.134 - - [27/Feb/2024 22:02:28] "POST / HTTP/1.0" 201 -
192.168.61.134 - - [27/Feb/2024 22:04:23] "POST / HTTP/1.0" 201 -
192.168.61.134 - - [27/Feb/2024 22:04:37] "POST / HTTP/1.0" 201 -
192.168.61.134 - - [27/Feb/2024 22:04:42] "POST / HTTP/1.0" 201 -
192.168.61.134 - - [27/Feb/2024 22:04:48] "POST / HTTP/1.0" 201 -
student206@kubeclass-206:~/wscba3$ kubectl logs url-shorten-service-7947bcbc7f-4m6n4
wait-for-it.sh: waiting 15 seconds for mysql-db:3306
wait-for-it.sh: mysql-db:3306 is available after 0 seconds
 * Serving Flask app 'app'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5001
 * Running on http://192.168.61.141:5001
Press CTRL+C to quit
192.168.61.134 - - [27/Feb/2024 22:02:13] "POST / HTTP/1.0" 201 -
192.168.61.134 - - [27/Feb/2024 22:02:29] "POST / HTTP/1.0" 201 -
192.168.61.134 - - [27/Feb/2024 22:02:30] "POST / HTTP/1.0" 201 -
192.168.61.134 - - [27/Feb/2024 22:02:31] "POST / HTTP/1.0" 201 -
192.168.61.134 - - [27/Feb/2024 22:02:31] "POST / HTTP/1.0" 201 -
192.168.61.134 - - [27/Feb/2024 22:04:19] "POST / HTTP/1.0" 201 -
192.168.61.134 - - [27/Feb/2024 22:04:26] "POST / HTTP/1.0" 201 -
192.168.61.134 - - [27/Feb/2024 22:04:31] "POST / HTTP/1.0" 201 -
```

Fig. 1: We retrieved logs for each pod using the kubectl logs command. By examining the timestamps for request handling across different pods, it's observable that each consecutive request is allocated to a different pod than the previous one.

pods looks like round-robin load balancing mechanism. In this type of approach, each new request is forwarded to the next pod in sequence, ensuring an even distribution of load across all available pods.

IV. Basic requirements

A. *Deployment and Service*

1) **Purpose:** Deployments ensured that the desired number of replicas for each service was maintained, handling scaling and self-healing in case of failures. Services provided stable network endpoints for communication between the components and with external users, abstracting away the details of the pod networking and ensuring seamless service discovery and load balancing.
2) **Implementations:** Utilized for the MySQL database, URL shortening service, user authentication service, and Nginx. For URL shortening service and user authentication service, use NodeAffinity to deploy on nodes labeled app=wscb-a3-app, and configure environment variables DB_USERNAME and DB_PASSWORD in Deployment using Kubernetes Secrets for secure database access.

V. Bonus of assignment

A. *MySQL Database*

1) **Purpose:** Whatever pods get requests, ensure the requests can be handled and return correct results to client users. Also it serves as the relational database system for storing user login details and the data for URL mappings.
2) **Implementations:** A specific Deployment configuration was created to manage the database pod, alongside a Service to provide a stable endpoint for other services to connect to the database.

B. *Nginx*

1) **Purpose:** To route requests from a single entry point on port 30000 to the appropriate backend service, whether it was the URL shortening service or the user authentication service.
2) **Implementations:** Created a Deployment configuration along with a Service to expose it. For the nginx, a NodePort service type was utilized to expose the nginx service on a port 30000.

C. *Secrets*

1) **Purpose:** Ensured that the database username and password were not hard-coded into the application code or deployment configurations, enhancing security.
2) **Implementations:** Secrets were mounted into the pods that required access to the database, such as the URL shortening service and the user authentication service.

D. *Persistent Volume*

1) **Purpose:** Data was not lost when the pod was terminated or moved.
2) **Implementations:** Using HostPath as a storage type is a simple method in Kubernetes for implementing local storage persistence in single-node environments. Since it's tied to a specific node, it's not suitable for high-availability production environments. First, define a PersistentVolume to specify the path on the host machine. This configuration creates a persistent volume named mysql-pv, pointing to the host's /mnt/data/mysql directory, with 1GB of storage space. ReadWriteOnce indicates the volume can be mounted in read-write mode by a single node. Next, create a PersistentVolumeClaim to request the aforementioned PV. In the MySQL Deployment configuration, this PVC is used to ensure data persistence. The MySQL container will use the mysql-pv PVC, meaning MySQL data will be stored in the host's /mnt/data/mysql directory, thus achieving data persistence.

VI. Responsibilities

Task	**Ivy Wang**	**Delong Yuan**
Code Implementation	Collaborator	Collaborator
Writing Report	Collaborator	Collaborator

TABLE I: Division of responsibilities between team members