**ECE573 PROJECT – 6 REPORT**

**Lakshmi Narayanan Giridharan**

**Section II A20544099**

**1) What are the Pods used by Chaos Mesh? Make an educated guess about their functionalities.**

Chaos Mesh uses several specialized Pods to execute and manage chaos engineering experiments effectively in Kubernetes environments. These include the Controller Manager Pod, the Chaos Daemon Pod, and, optionally, the Dashboard Pod.

* Controller Manager Pod: This Pod is the central component responsible for orchestrating and managing chaos experiments. It communicates with the Kubernetes API server to create and handle custom resource definitions (CRDs), which define the desired configurations for chaos experiments. The Controller Manager ensures that the specified chaos actions, such as network faults, Pod deletions, or resource constraints, are correctly implemented and maintained throughout the lifecycle of the experiment. It also reconciles the state of experiments to ensure they meet their intended objectives.
* Chaos Daemon Pod: This Pod is deployed as part of a daemon set, ensuring that an instance of the Chaos Daemon runs on every node in the cluster. The Chaos Daemon is responsible for executing the actual chaos operations at the node level. It applies disruptions directly to target resources, such as Pods or containers, based on the specifications in the CRDs. Common operations performed by this Pod include introducing network delays or packet loss, injecting CPU or memory stress, or killing specific application Pods to simulate failures.
* Dashboard Pod: The Dashboard Pod provides a user-friendly graphical interface for managing and visualizing chaos experiments. It allows users to interact with Chaos Mesh through a web browser, offering a clear overview of the experiments, their configurations, and their outcomes. This interface simplifies the process of designing and running experiments, making chaos engineering more accessible to users without extensive knowledge of Kubernetes commands or YAML configuration files.

Together, these Pods form the backbone of Chaos Mesh, ensuring the smooth execution, monitoring, and management of chaos experiments within Kubernetes clusters.

**2) Where are you going to find the service name and port required by kubectl port-forward?**

The service name and port required for the kubectl port-forward command can be identified by inspecting the Kubernetes services deployed in the namespace where Chaos Mesh is installed. This can be achieved using the following steps:

1. **List the Available Services**:  
   Use the kubectl get services -n <namespace> command to display a list of all services in the specified namespace. This command will return the service names, cluster IPs, and ports exposed by each service. For instance, services like chaos-dashboard or chaos-controller-manager might be listed along with their ports.

**Example output:**

**scss**

**NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE**

**chaos-dashboard ClusterIP 10.96.3.45 <none> 2333/TCP 10d**

**chaos-controller-manager ClusterIP 10.96.5.78 <none> 8080/TCP 10d**

1. **Get Detailed Information About a Service**:  
   To gather more specific details about a particular service, use the kubectl describe service <service-name> -n <namespace> command. This provides information about the service’s ports, target Pods, and configurations, helping identify the correct port to forward. For example, you can confirm whether the chaos-dashboard service is running on port 2333.

**Example output:**

**yaml**

**Name: chaos-dashboard**

**Namespace: chaos-mesh**

**Labels: app.kubernetes.io/component=dashboard**

**Selector: app.kubernetes.io/component=dashboard**

**Type: ClusterIP**

**IP: 10.96.3.45**

**Port: <none> 2333/TCP**

1. **Confirm Target Pod and Port Mappings**:  
   After identifying the service and its port, verify the associated Pod by using kubectl get pods -n <namespace>or kubectl describe pod <pod-name> -n <namespace>. This ensures the port is correctly mapped and the service is actively forwarding traffic to the desired Pod.
2. **Execute Port Forwarding**:  
   Use the kubectl port-forward command with the identified service name and port.

For example:

**bash**

**kubectl port-forward service/chaos-dashboard -n chaos-mesh 8080:2333**

This command forwards local port 8080 to the service’s port 2333, allowing access to the Chaos Mesh dashboard or other components locally.

**SECTION III**

**1. In pod-failure.yml, which part defines where the faults happen? I.e. how to define which Pods are affected?**

The section of the pod-failure.yml file that determines the scope of fault injection is defined under the spec field, specifically in the selector and the mode configuration. For example:

**yaml**

**spec:**

**selector:**

**labelSelectors:**

**app: kafka**

**mode: fixed**

**value: 2**

* selector.labelSelectors: This field ensures that the fault is targeted specifically at Pods with a designated label. In this case, Pods labeled app: kafka will experience the fault. This is essential for focusing the test on Kafka-related resources rather than impacting unrelated components.
* mode and value: These define how the fault will be distributed among the targeted Pods. The mode: fixed ensures a specific number of Pods are affected, while value: 2 means that two Kafka Pods will randomly encounter the fault during the experiment.

By configuring these fields, we can precisely control the impact and focus the test on a subset of the Kafka Pods, replicating realistic fault scenarios such as network latency or Pod-level unresponsiveness.

**2. Read pod-kill.yml and perform an experiment with it using kubectl apply and kubectl delete. Explain how Kafka reacts to this fault.**

The pod-kill.yml file introduces a more aggressive fault by terminating selected Pods. The experiment was executed using the following command:

**bash**

**kubectl apply -f pod-kill.yml**

Observations:

The producer logs showed consistent performance even after the fault injection.

**For example:**

**plaintext**

**2024/11/22 18:18:07 test: 1000 messages published**

**...**

**2024/11/22 18:22:48 test: 23000 messages published**

Despite the termination of Kafka Pods, the producer continued to publish messages to the cluster without delays or errors, thanks to Kafka's dynamic leader reassignment.

On the consumer side, message consumption remained uninterrupted. Logs confirmed the retrieval of messages across partitions:

**2024/11/22 18:23:24 Message: topic=test partition=0 offset=8530 value=0.510906**

**...**

Kafka Behavior:

Kafka's fault tolerance is evident in this scenario. When a Pod is terminated, Kubernetes restarts it automatically, and Kafka seamlessly handles the reassignment of partition leaders to healthy brokers. This ensures continuous availability for producers and consumers, with no significant degradation in service.

**3. What is the difference from the two fault types pod-failure and pod-kill?**

* **Pod-Failure:**

This scenario simulates a state where Pods become unresponsive but are not physically removed from the cluster. Examples include network isolation, memory contention, or CPU throttling. The Kafka cluster remains intact, but some Pods cannot process requests effectively. For instance, during a pod-failure fault, the affected Pods might still appear in the list of active brokers but fail to respond to producer or consumer requests.

* **Pod-Kill:**

Here, targeted Pods are forcefully terminated. The affected Pods are removed from the cluster and subsequently restarted by Kubernetes. While this introduces a momentary delay as Kubernetes recreates the Pods, Kafka's replication mechanism ensures that data availability and processing continuity are maintained. For example, if kafka-0 is killed, Kafka reassigns its partitions to other brokers until the Pod is restored.

**4. Use kubectl apply to inject pod-failure.yml again and then start the clients. Are producer and consumer working properly? Modify clients.yml so the clients can work when two random Kafka Pods fail. Don't forget to remove the fault by kubectl delete before you would like to inject it again.**

In this experiment, a fault was introduced to the Kafka cluster using the pod-failure.yml configuration. To evaluate the impact, the clients.yml configuration for the producer and consumer was modified to enhance fault tolerance.

Steps Taken:

1. Inject pod-failure.yml:

**bash**

**kubectl apply -f pod-failure.yml**

This introduced a failure to two randomly selected Kafka Pods.

1. Start Producer and Consumer Clients:

**bash**

**kubectl apply -f clients.yml**

1. Enhance Client Resilience:  
   The clients.yml file was updated to include multiple Kafka brokers in the KAFKA\_BOOTSTRAP\_SERVERSenvironment variable. This ensured that clients could route requests to any available broker:

**yaml**

**env:**

**- name: KAFKA\_BOOTSTRAP\_SERVERS**

**value: "kafka-0.kafka-service.default.svc.cluster.local:9092,kafka-1.kafka-service.default.svc.cluster.local:9092,kafka-2.kafka-service.default.svc.cluster.local:9092"**

By leveraging multiple brokers, the producer and consumer were insulated from individual broker failures.

1. Remove the Fault:

**bash**

**kubectl delete -f pod-failure.yml**

**IMPLEMENTATION:**

A screenshot of a computer

Description automatically generatedA screenshot of a computer screen

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated