Table of Contents

[Solutions 2](#_Toc184248765)

[1 Chaos Engineering with Chaos Mesh 2](#_Toc184248766)

[1.1 Completed Tasks Screenshots 2](#_Toc184248767)

[1.2 Questions 4](#_Toc184248768)

[1.2.1 Question 1 4](#_Toc184248769)

[1.2.2 Question 2 5](#_Toc184248770)

[2 Pod Faults 5](#_Toc184248771)

[2.1 Completed Tasks with Screenshots 5](#_Toc184248772)

[2.2 Questions 8](#_Toc184248773)

[2.2.1 Question 1 8](#_Toc184248774)

[2.2.2 Question 2 8](#_Toc184248775)

[2.2.3 Question 3 9](#_Toc184248776)

[2.2.4 Question 4 11](#_Toc184248777)

**List of Figures**

[Figure 1: Installing Chaos Mesh. 2](#_Toc184248778)

[Figure 2: Kind nodes verification. 2](#_Toc184248779)

[Figure 3: Chaos-Mesh pods. 3](#_Toc184248780)

[Figure 4: Chaos-Mesh Services. 3](#_Toc184248781)

[Figure 5: Port Forwarding on Terminal. 3](#_Toc184248782)

[Figure 6: Dashboard - Forwarded port as seen on the browser. 3](#_Toc184248783)

[Figure 7: Starting kafka. 5](#_Toc184248784)

[Figure 8: Successful run of build.sh. 6](#_Toc184248785)

[Figure 9: Creation of the producer and consumer. 6](#_Toc184248786)

[Figure 10: Verification of the successful run of Producer and Consumer. 6](#_Toc184248787)

[Figure 11: Deletion of the producer and consumer. 7](#_Toc184248788)

[Figure 12: Inspection of the details of topic test. 7](#_Toc184248789)

[Figure 13: Application of Pod Failure. 7](#_Toc184248790)

[Figure 14: Deletion of Pod Failure. 7](#_Toc184248791)

Solutions

# Chaos Engineering with Chaos Mesh

## Completed Tasks Screenshots

A screenshot of a computer

Description automatically generated

Figure 1: Installing Chaos Mesh.

The above Figure 1 represents the successful installation of Chaos Mesh using the command ./reset\_cluster.sh.

A black screen with white text

Description automatically generated

Figure 2: Kind nodes verification.

Figure 2 shows the first command of the series, kind get nodes to verify that everything is working and running.

Figure 3 represents the check of pods in the chaos-mesh since they stay in their own namespace and need to be checked separately. Meanwhile Figure 4 lists the services that are run by the chaos-mesh.

A screenshot of a computer

Description automatically generated

Figure 3: Chaos-Mesh pods.

A screen shot of a computer

Description automatically generated

Figure 4: Chaos-Mesh Services.

The following Figure 5 shows the port forwarding of the chaos-mesh which comes with a dashboard feature. This forwarding enables the connection to be brought from the cluster to the VM first and then VM to the host.

A black screen with white text

Description automatically generated

Figure 5: Port Forwarding on Terminal.

A screenshot of a computer

Description automatically generated

Figure 6: Dashboard - Forwarded port as seen on the browser.

The above Figure 6 shows a glimpse at the dashboard that is hosted after the port being forwarded as originally shown in Figure 5.

## Questions

### Question 1

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

Chaos Mesh Pod Analysis

Every element of the Chaos Mesh plays a complex part in the ecology of chaos engineering:

1. chaos-control-manager

The controller-manager, a stateless service that keeps chaotic experiments in the intended state, is the orchestration core of chaotic Mesh. It uses several specific controllers to manage various facets of chaotic experiments:

* Workflow controller for controlling the sequences of experiments.
* For timing and synchronization, use a schedule controller.
* State controller to keep the experiment in progress.
* Using a resource controller to regulate system resources.
* Recovery controller for managing the repair of faults.

1. chaos-daemon

This component performs low-level system activities at the kernel level with elevated privileges:

* Controls iptables rules for chaotic networks.
* cgroup interfaces for resource limitation.
* Responds to system requests for process modification.
* Filesystem operations are implemented for I/O chaos.
* Regulates network equipment to simulate latency and bandwidth.

1. chaos-dashboard

The dashboard offers a feature-rich online interface with several uses:

* Visualization of experiment states in real time.
* Creation and alteration of graphic experiments.
* Analysis and reporting of historical data.
* Monitoring and linkage of event timelines.
* Features for user management and access control.

1. chaos-dns-server

This specific part controls DNS-related chaotic situations:

* Manipulates DNS responses.
* Regulates the delay of DNS resolution.
* Mimics DNS server malfunctions.
* Controls the behavior of the DNS cache.
* Collaborates with other elements to create a complete network chaos.

### Question 2

**Q:** Instead of searching online, where are you going to find the service name and port required by **kubectl port-forward**?

The Kubernetes cluster configuration contains the service information directly. The following comprehensive service specifications are provided by the system when kubectl get services -n chaos-mesh is executed as shown in Figure 4:

* Service type (ClusterIP/NodePort)
* Internal and external port mappings
* Service endpoints and cluster IPs
* Protocol specifications (TCP/UDP)
* Service labels and selectors

This knowledge is essential for comprehending how to access different Chaos Mesh components, especially the dashboard service, which opens ports 2334 and 2333 for API connection and web interface access, respectively.

# Pod Faults

## Completed Tasks with Screenshots

A screen shot of a computer

Description automatically generated

Figure 7: Starting kafka.

Figure 7 shows the successful starting of kafka. While Figure 8 shows the successful run of the command ./build.sh which builds and initializes the clients.

A screenshot of a computer

Description automatically generated

Figure 8: Successful run of build.sh.

The following Figure 9 shows the successful starting of the clients where the producer and consumer are created.

A black background with white text

Description automatically generated

Figure 9: Creation of the producer and consumer.

A screenshot of a computer

Description automatically generated

Figure 10: Verification of the successful run of Producer and Consumer.

The above Figure 10 shows the successful run of Producer and Consumer which are verified. Subsequently in the following Figure 11, the clients are removed to preserve resources.

A black background with white text

Description automatically generated

Figure 11: Deletion of the producer and consumer.

The following Figure 12 shows the inspection of the details of the topic test.

A screen shot of a computer

Description automatically generated

Figure 12: Inspection of the details of topic test.

A screenshot of a computer screen

Description automatically generated

Figure 13: Application of Pod Failure.

A screenshot of a computer

Description automatically generated

Figure 14: Deletion of Pod Failure.

The subsequent test is done in introducing kafka pod failure voluntarily and then checking how kafka adapts to faulty environment. There is also the check on the pods and the test topic. This is shown in the above Figure 13. Further to this, the pod-failure is deleted, and the pods and the topic test is checked again for their respective statues. This is shown in the above Figure 14.

## Questions

### Question 1

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

Several essential elements define the fault injection based on the pod-failure.yml:

selector:

labelSelectors:

app: kafka

Pods that use the label app Kafka are the focus of this setup. Included in the standard are:

* value: "2" and mode: fixed, signifying that precisely two pods will be impacted.
* action: pod-failure indicating the fault type.
* duration: 3600s setting the fault duration.
* gracePeriod: 0, which denotes instantaneous operation without delay.

### Question 2

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

Kafka's resilience features are demonstrated in the pod-kill.yml experiment, which shows how the distributed messaging system manages unexpected broker failures while preserving data integrity and service availability.

* Initial State Analysis:

A cluster state of health was demonstrated by the regular operation of all three Kafka pods (kafka-0,1,2). The system's leadership assignments across partitions showed the best possible load distribution.

Topic 'test' had balanced leadership distribution:

* + Partition 0: Leader 1
  + Partition 1: Leader 0
  + Partition 2: Leader 2
  + Partition 3: Leader 1

This distribution pattern demonstrates Kafka's astute approach to leadership assignment, which makes sure no broker is overburdened with leadership duties. The cluster was perfectly synchronized as every node was listed in the ISR (In-Sync Replicas) list.

* Fault Impact:

The fault tolerance features of the system were instantly triggered upon the application of pod-kill. The events that followed were as follows:

* + Kafka-2 was stopped and resumed right away, as shown by the age being reset to 25.
  + For most divisions, leadership stayed constant.
  + The ISR list remained constant (0,1,2).
  + Pod recreation happened automatically without human assistance.

This behavior illustrates how Kafka may manage unexpected broker failures without compromising cluster stability. Kubernetes' instantaneous pod recreation enhances Kafka's built-in fault tolerance features.

* Recovery Behaviour:

The recuperation stage demonstrated the resilience of the Kubernetes and Kafka orchestrations:

* + Instantaneous pod reproduction reduced downtime.
  + Kafka's controller automatically rebalanced the leadership.
  + Following recovery, the leadership of Partition 2 returned to Kafka-2.
  + The ISR list was the same for every partition.
  + The system remained completely operational during the procedure.

While preserving data consistency throughout the cluster, this recovery strategy guarantees constant service availability.

### Question 3

**Q:** What is the difference from the two fault types, pod-failure and pod-kill?

The tests show how different failure kinds affect Kafka's operational features and recovery procedures, exposing basic behavioral differences.

* Pod Kill Characteristics:

The pod-kill method shows how to handle broker failures in a neat and effective way:

* + Quick system recuperation by immediate termination and recreation  
      
    keeps the cluster stable by using well-coordinated failover procedures.
  + Reduced service interruption and speedy recovery using pod recreation.
  + Maintains the integrity of the ISR list and guarantees data consistency.
  + Maintaining a balanced workload with minimal interference to partition leadership.

Kafka's capacity to manage clean broker failures while preserving system stability is demonstrated by this behavior. The instantaneous pod recreation aids in preserving the system redundancy and desired replica count.

* Pod Failure Characteristics:

On the other hand, pod-failure poses a more difficult situation for the cluster:

* + Causes persistent unavailability by forcing pods into the CrashLoopBackOff state.
  + Cluster stability is more severely impacted by frequent crash-restart cycles.
  + Drastically lowers the number of brokers accessible, impacting system capacity.
  + Forces surviving nodes to consolidate their leadership.
  + Significantly lowers ISR membership, which affects data redundancy.

Kafka's fault tolerance algorithms are put to the test in a more complex recovery scenario due to the prolonged nature of pod-failure.

* Key Differences in System Impact:

1. Recovery Pattern
   * Pod Kill: A clean end that allows for quick recreation and predictable recovery routes.
   * Pod Failure: Prolonged instability caused by numerous crash attempts and sustained unavailability.

System dependability and maintenance plans are greatly impacted by the differences in recovery patterns.

1. Leadership Behaviour
   * Pod Kill: A brief change in leadership that automatically rebalances.
   * Pod Failure: Consolidation to the only broker left might result in congestion.

Performance and system stability in failure circumstances are impacted by this disparity in leadership management.

1. Data Consistency
   * Pod Kill: Preserves greater data redundancy by maintaining a larger ISR membership.
   * Pod Failure: Increases the chance of data loss by reducing ISR to a single node.

The impact on data consistency has direct implications for system reliability and data durability guarantees.

### Question 4

**Q:** 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.

* Current Behaviour Analysis:

The experimental findings highlight several drawbacks in the standard client setup:

* + Broker unavailability causes the producer to fail entirely, demonstrating a lack of fault tolerance.
  + The consumer tries to connect but has problems, suggesting insufficient failover measures.
  + Maintaining service continuity requires more than one broker setup.

The necessity of strong client-side fault tolerance setups is highlighted by these findings.

* Required Modifications for clients.yml:
  + Producer Configuration:

- name: KAFKA\_BROKER

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"

With this setup, a strong failover mechanism is implemented, allowing the producer to continue operating even if certain brokers are unavailable.

* + Consumer Configuration:

- name: KAFKA\_BROKER

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"

This setup guarantees that consumers can process messages even in the event of broker failures, just like the producer does.