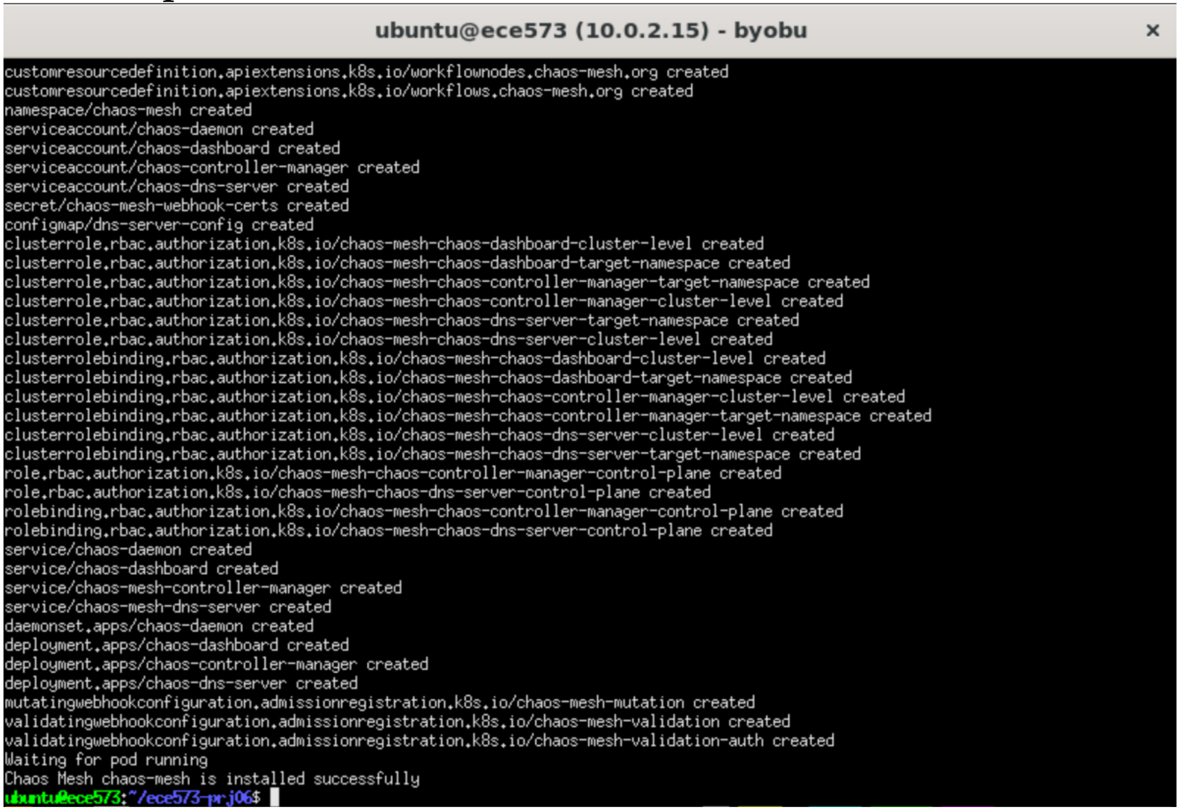
Table of Contents

List of Figures

Solutions

1 Chaos Engineering with Chaos Mesh

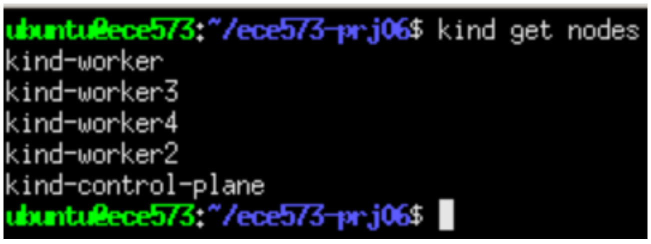
1.1 Completed Tasks Screenshots



```
ubuntu@ece573 (10.0.2.15) - byobu
customresourcedefinition.apiextensions.k8s.io/workflownodes.chaos-mesh.org created
customresourcedefinition.apiextensions.k8s.io/workflows.chaos-mesh.org created
namespace/chaos-mesh created
serviceaccount/chaos-daemon created
serviceaccount/chaos-dashboard created
serviceaccount/chaos-controller-manager created
serviceaccount/chaos-dns-server created
secret/chaos-mesh-webhook-certs created
configmap/dns-server-config created
clusterrole.rbac.authorization.k8s.io/chaos-mesh-chaos-dashboard-cluster-level created
clusterrole.rbac.authorization.k8s.io/chaos-mesh-chaos-dashboard-target-namespace created
clusterrole.rbac.authorization.k8s.io/chaos-mesh-chaos-controller-manager-target-namespace created
clusterrole.rbac.authorization.k8s.io/chaos-mesh-chaos-controller-manager-cluster-level created
clusterrole.rbac.authorization.k8s.io/chaos-mesh-chaos-dns-server-target-namespace created
clusterrole.rbac.authorization.k8s.io/chaos-mesh-chaos-dns-server-cluster-level created
clusterrolebinding.rbac.authorization.k8s.io/chaos-mesh-chaos-dashboard-cluster-level created
clusterrolebinding.rbac.authorization.k8s.io/chaos-mesh-chaos-dashboard-target-namespace created
clusterrolebinding.rbac.authorization.k8s.io/chaos-mesh-chaos-controller-manager-cluster-level created
clusterrolebinding.rbac.authorization.k8s.io/chaos-mesh-chaos-controller-manager-target-namespace created
clusterrolebinding.rbac.authorization.k8s.io/chaos-mesh-chaos-dns-server-cluster-level created
clusterrolebinding.rbac.authorization.k8s.io/chaos-mesh-chaos-dns-server-target-namespace created
role.rbac.authorization.k8s.io/chaos-mesh-chaos-controller-manager-control-plane created
role.rbac.authorization.k8s.io/chaos-mesh-chaos-dns-server-control-plane created
rolebinding.rbac.authorization.k8s.io/chaos-mesh-chaos-controller-manager-control-plane created
rolebinding.rbac.authorization.k8s.io/chaos-mesh-chaos-dns-server-control-plane created
service/chaos-daemon created
service/chaos-dashboard created
service/chaos-mesh-controller-manager created
service/chaos-mesh-dns-server created
daemonset.apps/chaos-daemon created
deployment.apps/chaos-dashboard created
deployment.apps/chaos-controller-manager created
deployment.apps/chaos-dns-server created
mutatingwebhookconfiguration.admissionregistration.k8s.io/chaos-mesh-mutation created
validatingwebhookconfiguration.admissionregistration.k8s.io/chaos-mesh-validation created
validatingwebhookconfiguration.admissionregistration.k8s.io/chaos-mesh-validation-auth created
Waiting for pod running
Chaos Mesh chaos-mesh is installed successfully
ubuntu@ece573:~/ece573-prj06$
```

Figure 1: Installing Chaos Mesh.

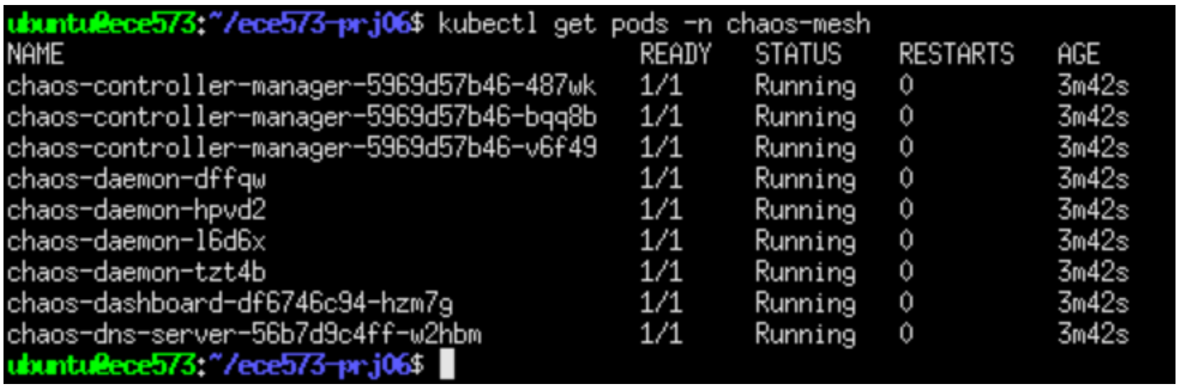
The above Figure 1 represents the successful installation of Chaos Mesh using the command `./reset_cluster.sh`.



```
ubuntu@ece573:~/ece573-prj06$ kind get nodes
kind-worker
kind-worker3
kind-worker4
kind-worker2
kind-control-plane
ubuntu@ece573:~/ece573-prj06$
```

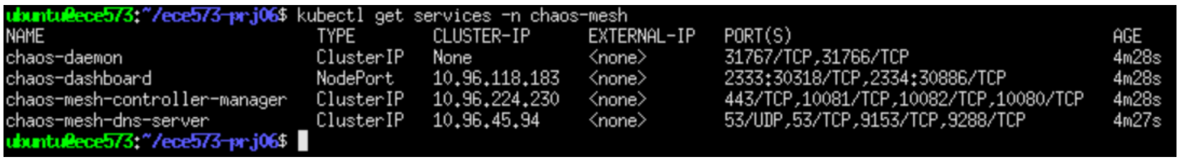
Figure 2: Kind nodes verification.

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.



```
ubuntu@ece573:~/ece573-prj06$ kubectl get pods -n chaos-mesh
NAME                                        READY   STATUS    RESTARTS   AGE
chaos-controller-manager-5969d57b46-487wk   1/1     Running   0          3m42s
chaos-controller-manager-5969d57b46-bqq8b   1/1     Running   0          3m42s
chaos-controller-manager-5969d57b46-v6f49   1/1     Running   0          3m42s
chaos-daemon-dffqw                          1/1     Running   0          3m42s
chaos-daemon-hpvd2                          1/1     Running   0          3m42s
chaos-daemon-l6d6x                          1/1     Running   0          3m42s
chaos-daemon-tzt4b                          1/1     Running   0          3m42s
chaos-dashboard-df6746c94-hzm7g             1/1     Running   0          3m42s
chaos-dns-server-56b7d9c4ff-w2hbm           1/1     Running   0          3m42s
ubuntu@ece573:~/ece573-prj06$
```

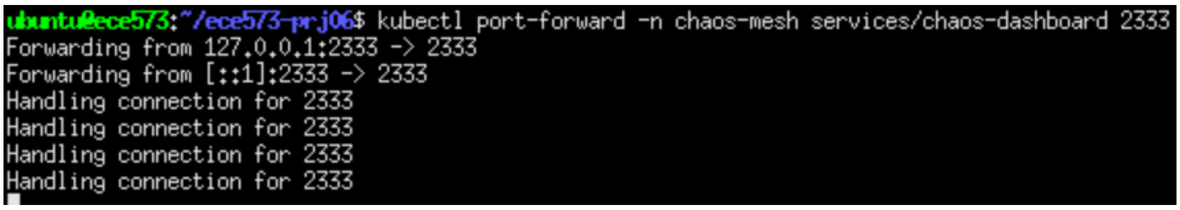
Figure 3: Chaos-Mesh pods.



```
ubuntu@ece573:~/ece573-prj06$ kubectl get services -n chaos-mesh
NAME                          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)                                 AGE
chaos-daemon                  ClusterIP   None            <none>        31767/TCP,31766/TCP                     4m28s
chaos-dashboard               NodePort    10.96.118.183   <none>        2333:30318/TCP,2334:30886/TCP           4m28s
chaos-mesh-controller-manager ClusterIP   10.96.224.230   <none>        443/TCP,10081/TCP,10082/TCP,10080/TCP   4m28s
chaos-mesh-dns-server         ClusterIP   10.96.45.94     <none>        53/UDP,53/TCP,9153/TCP,9288/TCP         4m27s
ubuntu@ece573:~/ece573-prj06$
```

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.



```
ubuntu@ece573:~/ece573-prj06$ kubectl port-forward -n chaos-mesh services/chaos-dashboard 2333
Forwarding from 127.0.0.1:2333 -> 2333
Forwarding from [::1]:2333 -> 2333
Handling connection for 2333
Handling connection for 2333
Handling connection for 2333
Handling connection for 2333
```

Figure 5: Port Forwarding on Terminal.

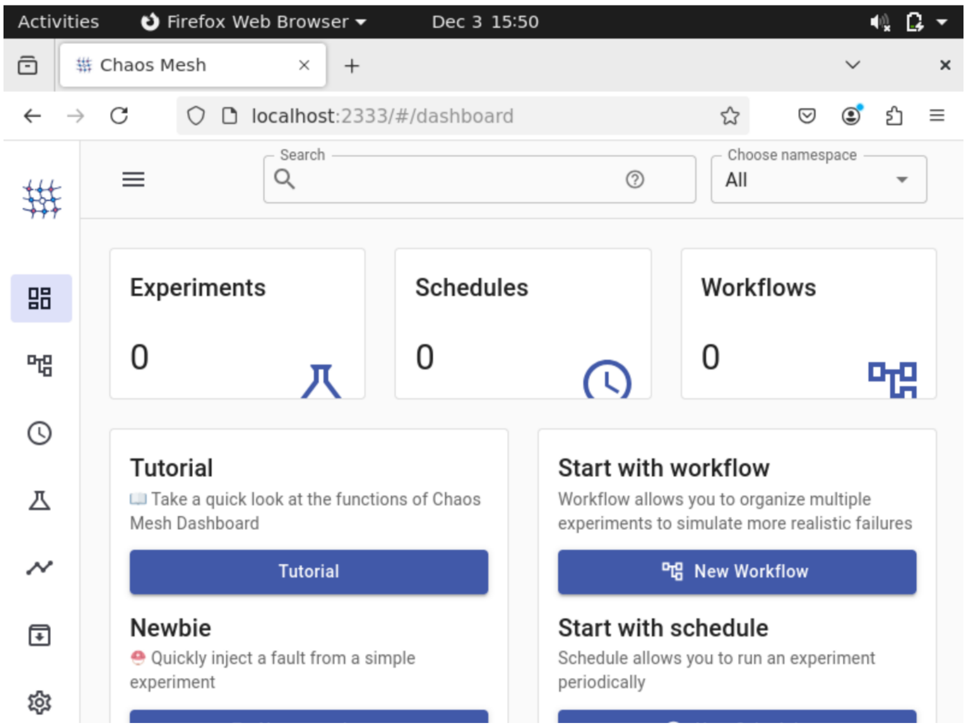

Activities
Firefox Web Browser
Dec 3 15:50
Chaos Mesh
localhost:2333/#/dashboard
Search
Choose namespace
All
Experiments
0
Schedules
0
Workflows
0
Tutorial
Take a quick look at the functions of Chaos Mesh Dashboard
Tutorial
Newbie
Quickly inject a fault from a simple experiment
Start with workflow
Workflow allows you to organize multiple experiments to simulate more realistic failures
New Workflow
Start with schedule
Schedule allows you to run an experiment periodically

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.

1.2 Questions

1.2.1 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.

2. 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.

3. 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.

4. 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.

1.2.2 Question 2

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

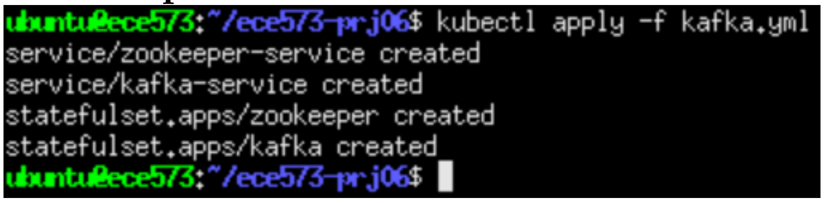
The Kubernetes cluster configuration contains the service information directly. The following comprehensive service specifications are provided by the system when `kubectl get services -n chaos-mesh` is executed as shown in Figure 4:

- 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.

2 Pod Faults

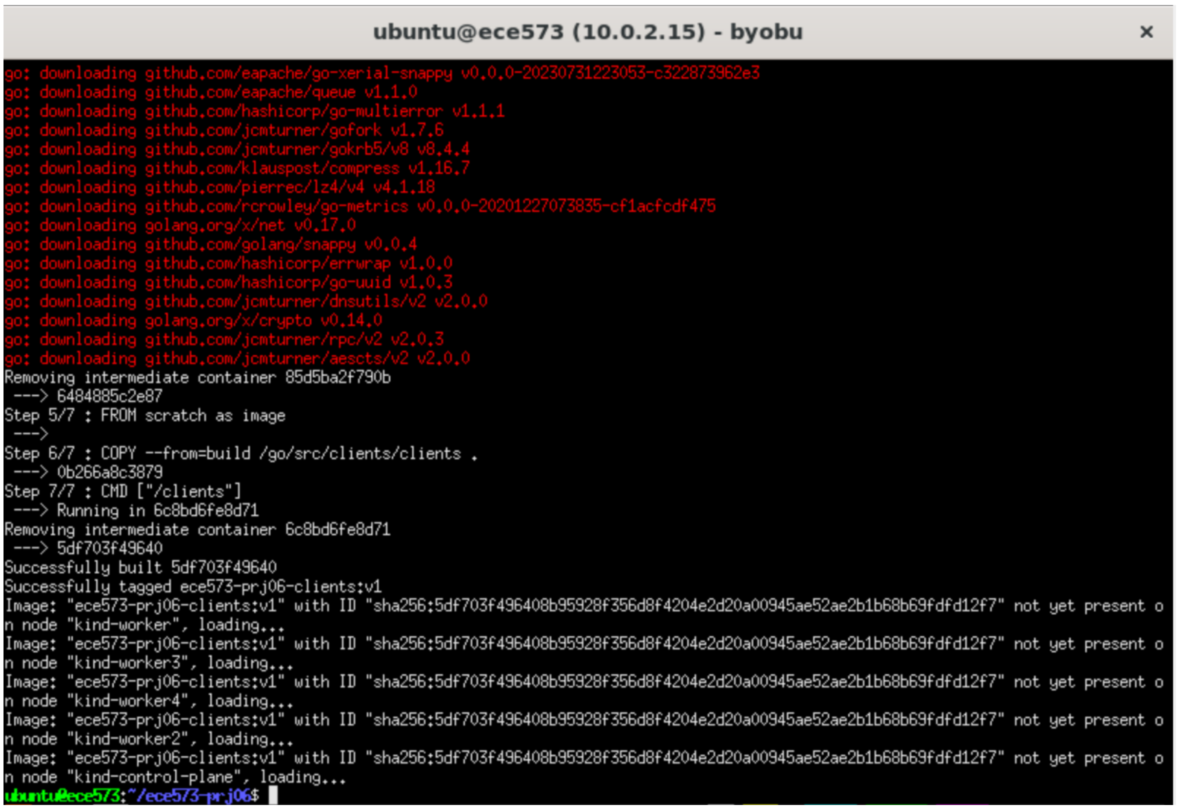
2.1 Completed Tasks with Screenshots



```
ubuntu@ece573:~/ece573-prj06$ kubectl apply -f kafka.yml
service/zookeeper-service created
service/kafka-service created
statefulset.apps/zookeeper created
statefulset.apps/kafka created
ubuntu@ece573:~/ece573-prj06$
```

Figure 7: Starting kafka.

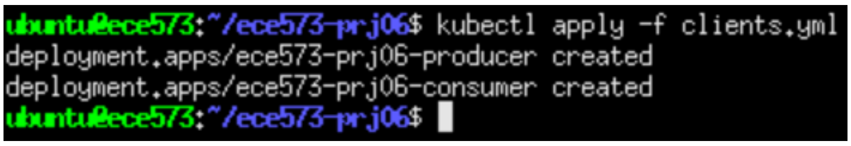
Figure 7 shows the successful starting of kafka. While Figure 8 shows the successful run of the command ./build.sh which builds and initializes the clients.



```
go: downloading github.com/eapache/go-xerial-snappy v0.0.0-20230731223053-c322873962e3
go: downloading github.com/eapache/queue v1.1.0
go: downloading github.com/hashicorp/go-multierror v1.1.1
go: downloading github.com/jcmturner/gofork v1.7.6
go: downloading github.com/jcmturner/gokrb5/v8 v8.4.4
go: downloading github.com/klauspost/compress v1.16.7
go: downloading github.com/pierrec/lz4/v4 v4.1.18
go: downloading github.com/rcrowley/go-metrics v0.0.0-20201227073835-cf1acfcdf475
go: downloading golang.org/x/net v0.17.0
go: downloading github.com/golang/snappy v0.0.4
go: downloading github.com/hashicorp/errwrap v1.0.0
go: downloading github.com/hashicorp/go-uuid v1.0.3
go: downloading github.com/jcmturner/dnsutils/v2 v2.0.0
go: downloading golang.org/x/crypto v0.14.0
go: downloading github.com/jcmturner/rpc/v2 v2.0.3
go: downloading github.com/jcmturner/aescts/v2 v2.0.0
Removing intermediate container 85d5ba2f790b
 ---> 6484885c2e87
Step 5/7 : FROM scratch as image
 --->
Step 6/7 : COPY --from=build /go/src/clients/clients .
 ---> 0b266a8c3879
Step 7/7 : CMD ["/clients"]
 ---> Running in 6c8bd6fe8d71
Removing intermediate container 6c8bd6fe8d71
 ---> 5df703f49640
Successfully built 5df703f49640
Successfully tagged ece573-prj06-clients:v1
Image: "ece573-prj06-clients:v1" with ID "sha256:5df703f496408b95928f356d8f4204e2d20a00945ae52ae2b1b68b69fdfd12f7" not yet present o
n node "kind-worker", loading...
Image: "ece573-prj06-clients:v1" with ID "sha256:5df703f496408b95928f356d8f4204e2d20a00945ae52ae2b1b68b69fdfd12f7" not yet present o
n node "kind-worker3", loading...
Image: "ece573-prj06-clients:v1" with ID "sha256:5df703f496408b95928f356d8f4204e2d20a00945ae52ae2b1b68b69fdfd12f7" not yet present o
n node "kind-worker4", loading...
Image: "ece573-prj06-clients:v1" with ID "sha256:5df703f496408b95928f356d8f4204e2d20a00945ae52ae2b1b68b69fdfd12f7" not yet present o
n node "kind-worker2", loading...
Image: "ece573-prj06-clients:v1" with ID "sha256:5df703f496408b95928f356d8f4204e2d20a00945ae52ae2b1b68b69fdfd12f7" not yet present o
n node "kind-control-plane", loading...
ubuntu@ece573:~/ece573-prj06$
```

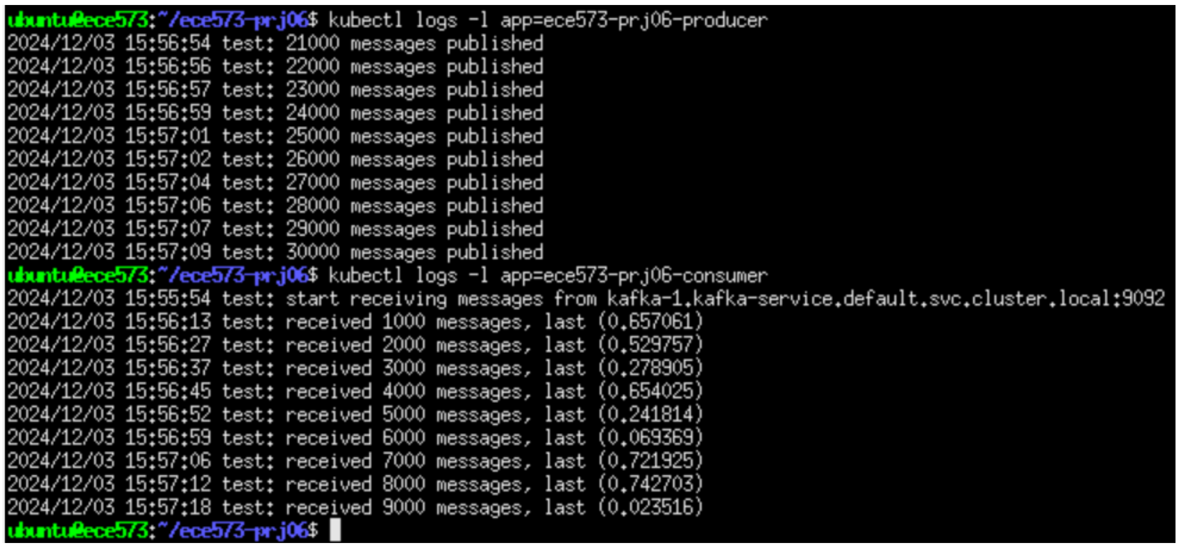
Figure 8: Successful run of build.sh.

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



```
ubuntu@ece573:~/ece573-prj06$ kubectl apply -f clients.yml
deployment.apps/ece573-prj06-producer created
deployment.apps/ece573-prj06-consumer created
ubuntu@ece573:~/ece573-prj06$
```

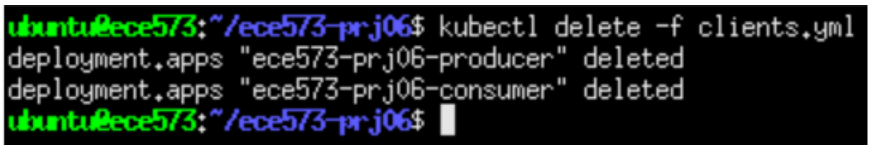
Figure 9: Creation of the producer and consumer.



```
ubuntu@ece573:~/ece573-prj06$ kubectl logs -l app=ece573-prj06-producer
2024/12/03 15:56:54 test: 21000 messages published
2024/12/03 15:56:56 test: 22000 messages published
2024/12/03 15:56:57 test: 23000 messages published
2024/12/03 15:56:59 test: 24000 messages published
2024/12/03 15:57:01 test: 25000 messages published
2024/12/03 15:57:02 test: 26000 messages published
2024/12/03 15:57:04 test: 27000 messages published
2024/12/03 15:57:06 test: 28000 messages published
2024/12/03 15:57:07 test: 29000 messages published
2024/12/03 15:57:09 test: 30000 messages published
ubuntu@ece573:~/ece573-prj06$ kubectl logs -l app=ece573-prj06-consumer
2024/12/03 15:55:54 test: start receiving messages from kafka-1.kafka-service.default.svc.cluster.local:9092
2024/12/03 15:56:13 test: received 1000 messages, last (0.657061)
2024/12/03 15:56:27 test: received 2000 messages, last (0.529757)
2024/12/03 15:56:37 test: received 3000 messages, last (0.278905)
2024/12/03 15:56:45 test: received 4000 messages, last (0.654025)
2024/12/03 15:56:52 test: received 5000 messages, last (0.241814)
2024/12/03 15:56:59 test: received 6000 messages, last (0.069369)
2024/12/03 15:57:06 test: received 7000 messages, last (0.721925)
2024/12/03 15:57:12 test: received 8000 messages, last (0.742703)
2024/12/03 15:57:18 test: received 9000 messages, last (0.023516)
ubuntu@ece573:~/ece573-prj06$
```

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

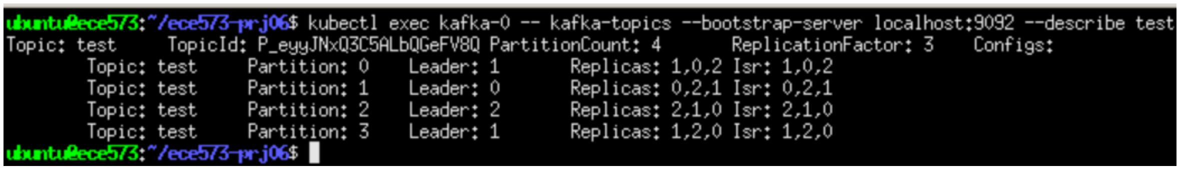
The above Figure 10 shows the successful run of Producer and Consumer which are verified. Subsequently in the following Figure 11, the clients are removed to preserve resources.



```
ubuntu@ece573:~/ece573-prj06$ kubectl delete -f clients.yml
deployment.apps "ece573-prj06-producer" deleted
deployment.apps "ece573-prj06-consumer" deleted
ubuntu@ece573:~/ece573-prj06$
```

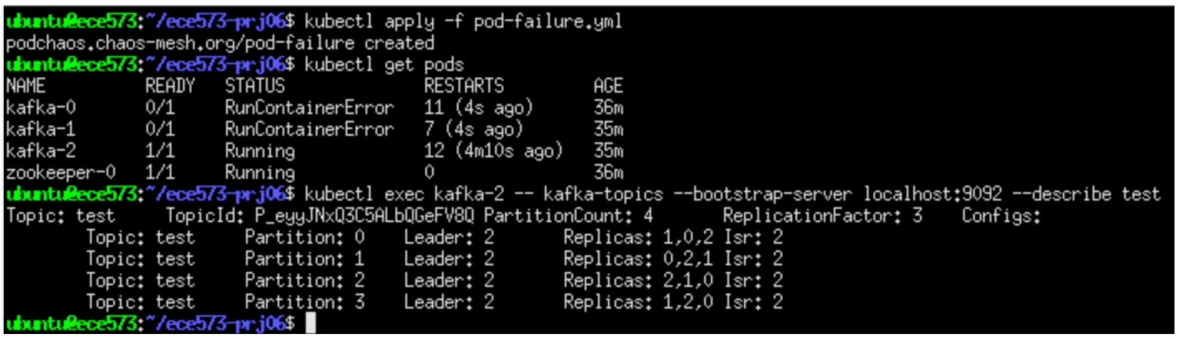
Figure 11: Deletion of the producer and consumer.

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



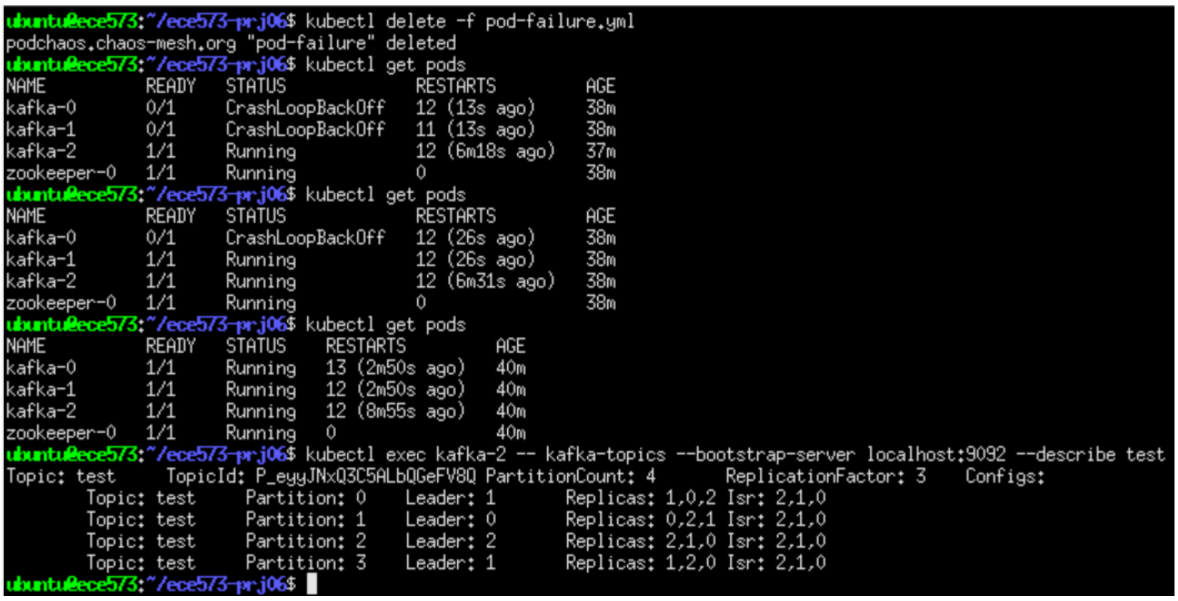
```
ubuntu@ece573:~/ece573-prj06$ kubectl exec kafka-0 -- kafka-topics --bootstrap-server localhost:9092 --describe test
Topic: test     TopicId: P_eyyJNxQ3C5ALbQGeFV8Q PartitionCount: 4       ReplicationFactor: 3    Configs:
        Topic: test     Partition: 0    Leader: 1       Replicas: 1,0,2 Isr: 1,0,2
        Topic: test     Partition: 1    Leader: 0       Replicas: 0,2,1 Isr: 0,2,1
        Topic: test     Partition: 2    Leader: 2       Replicas: 2,1,0 Isr: 2,1,0
        Topic: test     Partition: 3    Leader: 1       Replicas: 1,2,0 Isr: 1,2,0
ubuntu@ece573:~/ece573-prj06$
```

Figure 12: Inspection of the details of topic test.



```
ubuntu@ece573:~/ece573-prj06$ kubectl apply -f pod-failure.yml
podchaos.chaos-mesh.org/pod-failure created
ubuntu@ece573:~/ece573-prj06$ kubectl get pods
NAME          READY   STATUS              RESTARTS        AGE
kafka-0       0/1     RunContainerError   11 (4s ago)     36m
kafka-1       0/1     RunContainerError   7 (4s ago)      35m
kafka-2       1/1     Running             12 (4m10s ago)  35m
zookeeper-0   1/1     Running             0               36m
ubuntu@ece573:~/ece573-prj06$ kubectl exec kafka-2 -- kafka-topics --bootstrap-server localhost:9092 --describe test
Topic: test     TopicId: P_eyyJNxQ3C5ALbQGeFV8Q PartitionCount: 4       ReplicationFactor: 3    Configs:
        Topic: test     Partition: 0    Leader: 2       Replicas: 1,0,2 Isr: 2
        Topic: test     Partition: 1    Leader: 2       Replicas: 0,2,1 Isr: 2
        Topic: test     Partition: 2    Leader: 2       Replicas: 2,1,0 Isr: 2
        Topic: test     Partition: 3    Leader: 2       Replicas: 1,2,0 Isr: 2
ubuntu@ece573:~/ece573-prj06$
```

Figure 13: Application of Pod Failure.



```
ubuntu@ece573:~/ece573-prj06$ kubectl delete -f pod-failure.yml
podchaos.chaos-mesh.org "pod-failure" deleted
ubuntu@ece573:~/ece573-prj06$ kubectl get pods
NAME          READY   STATUS             RESTARTS        AGE
kafka-0       0/1     CrashLoopBackOff   12 (13s ago)    38m
kafka-1       0/1     CrashLoopBackOff   11 (13s ago)    38m
kafka-2       1/1     Running            12 (6m18s ago)  37m
zookeeper-0   1/1     Running            0               38m
ubuntu@ece573:~/ece573-prj06$ kubectl get pods
NAME          READY   STATUS             RESTARTS        AGE
kafka-0       0/1     CrashLoopBackOff   12 (26s ago)    38m
kafka-1       1/1     Running            12 (26s ago)    38m
kafka-2       1/1     Running            12 (6m31s ago)  38m
zookeeper-0   1/1     Running            0               38m
ubuntu@ece573:~/ece573-prj06$ kubectl get pods
NAME          READY   STATUS    RESTARTS       AGE
kafka-0       1/1     Running   13 (2m50s ago)   40m
kafka-1       1/1     Running   12 (2m50s ago)   40m
kafka-2       1/1     Running   12 (8m55s ago)   40m
zookeeper-0   1/1     Running   0                40m
ubuntu@ece573:~/ece573-prj06$ kubectl exec kafka-2 -- kafka-topics --bootstrap-server localhost:9092 --describe test
Topic: test     TopicId: P_eyyJNxQ3C5ALbQGeFV8Q PartitionCount: 4       ReplicationFactor: 3    Configs:
        Topic: test     Partition: 0    Leader: 1       Replicas: 1,0,2 Isr: 2,1,0
        Topic: test     Partition: 1    Leader: 0       Replicas: 0,2,1 Isr: 2,1,0
        Topic: test     Partition: 2    Leader: 2       Replicas: 2,1,0 Isr: 2,1,0
        Topic: test     Partition: 3    Leader: 1       Replicas: 1,2,0 Isr: 2,1,0
ubuntu@ece573:~/ece573-prj06$
```

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.

2.2 Questions

2.2.1 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.

2.2.2 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.

2.2.3 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.

2. 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.

3. 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.

2.2.4 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.de-
fault.svc.cluster.local:9092"
```

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