Table of Contents

List of Figures

List of Code Snippets

List of Tables

Solutions

1 Kafka Services

1.1 Screenshot of Completed tasks

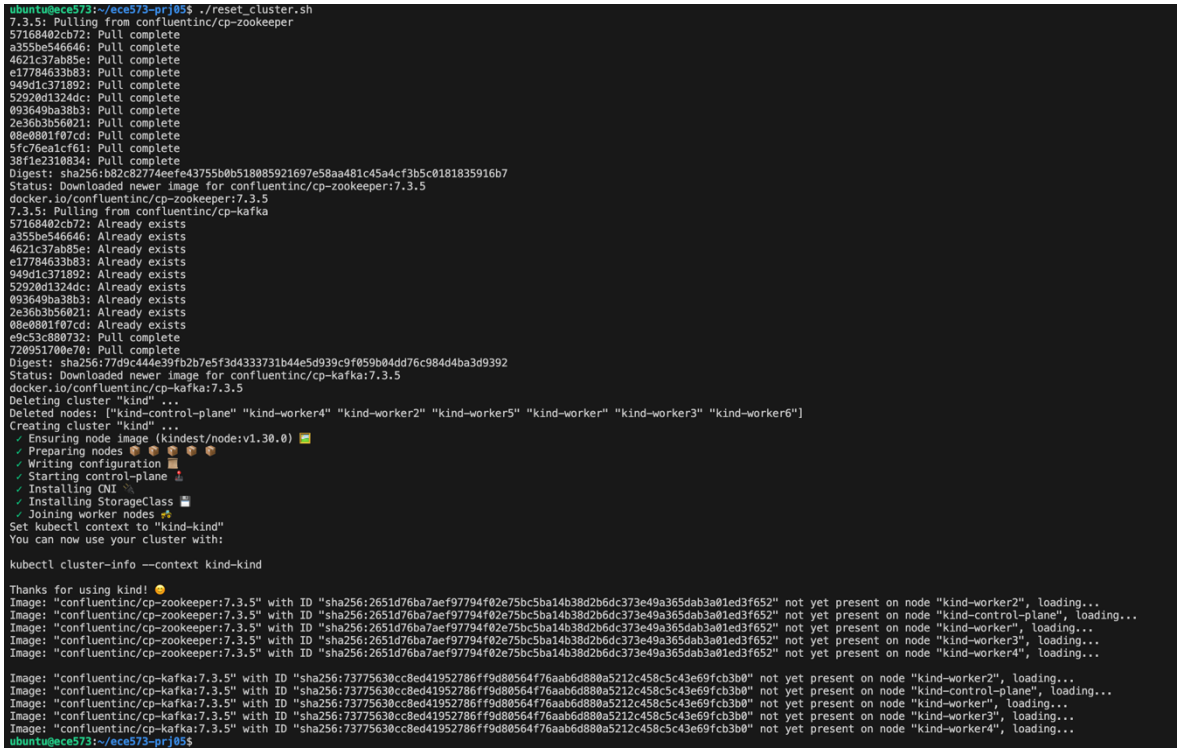

Figure 1: Running ./reset_cluster.sh

From the Figure 1, the pull of the required images using reset_cluster.sh, then use the cluster.yml script to build a new type of cluster, load images onto nodes, and remove any existing ones.

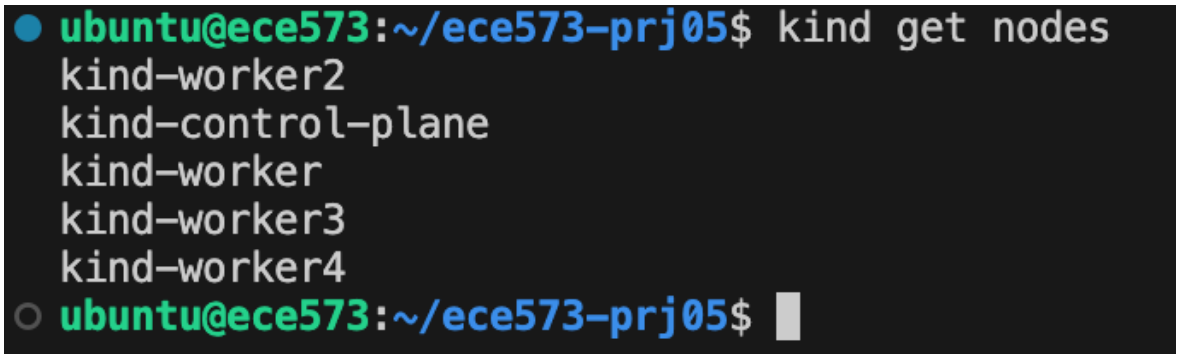

Figure 2: Verification of running cluster using king get nodes command.

The above Figure 2 shows the cluster is up and running with the worker nodes and the control place active status by running the command `kind get nodes`.

```
ubuntu@ece573:~/ece573-prj05$ kubectl apply -f kafka.yml
service/zookeeper-service created
service/kafka-service created
statefulset.apps/zookeeper created
statefulset.apps/kafka created
ubuntu@ece573:~/ece573-prj05$
```

Figure 3: Starting the services.

The above Figure 3 represents the start the of the services i.e. zookeeper, kafka and the stateful sets as well.

```
ubuntu@ece573:~/Desktop/ece573-prj05$ kubectl get services
NAME                TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
kafka-service       ClusterIP   None         <none>        9092/TCP   22s
kubernetes          ClusterIP   10.96.0.1    <none>        443/TCP    10m
zookeeper-service   ClusterIP   None         <none>        2181/TCP   22s
ubuntu@ece573:~/Desktop/ece573-prj05$ kubectl get statefulsets
NAME        READY   AGE
kafka       3/3     43s
zookeeper   1/1     44s
ubuntu@ece573:~/Desktop/ece573-prj05$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
kafka-0       1/1     Running   0          49s
kafka-1       1/1     Running   0          43s
kafka-2       1/1     Running   0          37s
zookeeper-0   1/1     Running   0          49s
ubuntu@ece573:~/Desktop/ece573-prj05$
```

Figure 4: Verification of the status of the services.

Figure 4 shows the run of the command `kubectl get services` where the services that are running are displayed with their statuses and age of the time since they are running.

```
ubuntu@ece573:~/ece573-prj05$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
kafka-0       1/1     Running   0          18m
kafka-1       1/1     Running   0          18m
kafka-2       1/1     Running   0          18m
zookeeper-0   1/1     Running   0          18m
ubuntu@ece573:~/ece573-prj05$
```

Figure 5: Checking and verifying again for all pods to be available.

A while later, the command `kubectl get pods` is run to ensure that all pods are running and available. Thus, as shown in Figure 5, some time is left for the process to happen and then the status of the pods are visible when the command is run.

Figure 6: Checking for irregularity with kubectl logs.

Based on Figure 6, the kubectl logs is further explored to hunt for any issues based on the log messages stating an error and to further troubleshoot any such errors.

Figure 7: Ensuring zookeeper and kafka's proper running by listing them.

In Figure 7, the services are listed to ensure the zookeeper and kafka are running appropriately from their respective pods.

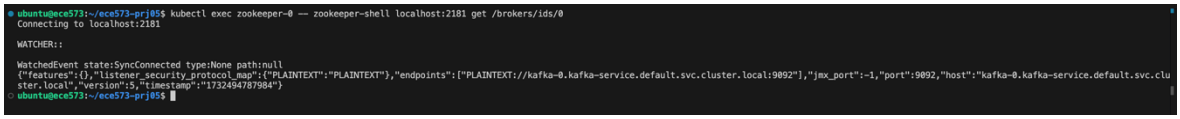

Figure 8: Detailed view of each state of zookeeper and kafka.

From the above Figure 8, it is viewed that, the three brokers 0, 1 and 2 are available and their detail is listed using the `get` attribute.
The next command in the following Figure 9 shows the list of all topics that are running which makes it easier to manage them.

Figure 9: Listing out all the topics for easier management.

However, from Figure 9, it is seen that there is no output after running the command since there are no topics. Nevertheless, this command also helps view if there are any errors present – which clearly there isn't any now.

1.2 Questions

1.2.1 Question 1

Q: How do Kafka brokers know the location of ZooKeeper Service that they can connect to? How do Kafka brokers know each other so they can collaborate to complete tasks like replication? (Hints: refer to kafka.yml)

Several important settings in the kafka.yml file enable communication between ZooKeeper and Kafka brokers:

1. ZooKeeper Connection:

```
env:
- name: KAFKA_ZOOKEEPER_CONNECT
  value: "zookeeper-service.default.svc.cluster.local:2181"
```

With this setup, ZooKeeper can be found by Kafka brokers using:

- The ZooKeeper service's fully qualified domain name (FQDN)
- The headless service that permits direct pod connection (clusterIP: None)
- The standard port for ZooKeeper clients is 2181.

2. Inter-Broker Communication:

```
env:
- name: KAFKA_ADVERTISED_LISTENERS
  value:                                    "PLAINTEXT://$(POD_NAME).kafka-
service.default.svc.cluster.local:9092"
- name: POD_NAME
  valueFrom:
    fieldRef:
      fieldPath: metadata.name
```

Brokers find and interact with one another by:

- Using the hostname of the pod to assign the broker ID dynamically:

```
command:
  - sh
  - -c
  - "export KAFKA_BROKER_ID=`hostname | awk -F '-' '{print $2}'` &&
/etc/confluent/docker/run"
```

- Pod names controlled by StatefulSet (kafka-0, kafka-1, and kafka-2).
- Direct pod addressing is made possible by the headless Kafka service.
- Listeners that broadcast the network address of each broker were advertised.

3. Replication Configuration:

```
env:
- name: KAFKA_NUM_PARTITIONS
  value: "4"
- name: KAFKA_DEFAULT_REPLICATION_FACTOR
  value: "3"
```

These configurations guarantee appropriate replication and data distribution throughout the cluster.

1.2.2 Question 2

Q: Both zookeeper-shell and kafka-topics refer to localhost (but with different ports) to make connections. What does localhost refer to? Do they refer to the same thing for both cases? (Hints: is localhost referring to a K8s node or a Pod?)

The different network environments are indicated by the localhost references in various commands:

1. ZooKeeper Shell Command

```
kubectl exec zookeeper-0 -- zookeeper-shell localhost:2181
```

This localhost is associated with:

- The zookeeper-0 pod's network namespace.
- Within the pod, the ZooKeeper process is listening on port 2181.
- Defined by the ZooKeeper StatefulSet's containerPort configuration:

```
kubectl exec zookeeper-0 -- zookeeper-shell localhost:2181 get /brokers/ids/0
```

2. Kafka Topics Command

```
kubectl exec kafka-0 -- kafka-topics --bootstrap-server localhost:9092
```

This localhost is associated with:

- Within the Kafka-0 pod, the network namespace.
- Within the pod, the Kafka broker process is listening on port 9092.
- Defined by the Kafka StatefulSet's containerPort configuration:

```
ports:
  - containerPort: 9092
```

The main difference is that each localhost reference is located within the network namespace of its corresponding pod, not the cluster network or Kubernetes node. The containerPort parameters in the StatefulSet definitions and the network isolation of the pod both enforce this.

2 Topics and Messages

2.1 Screenshots of Completed Tasks

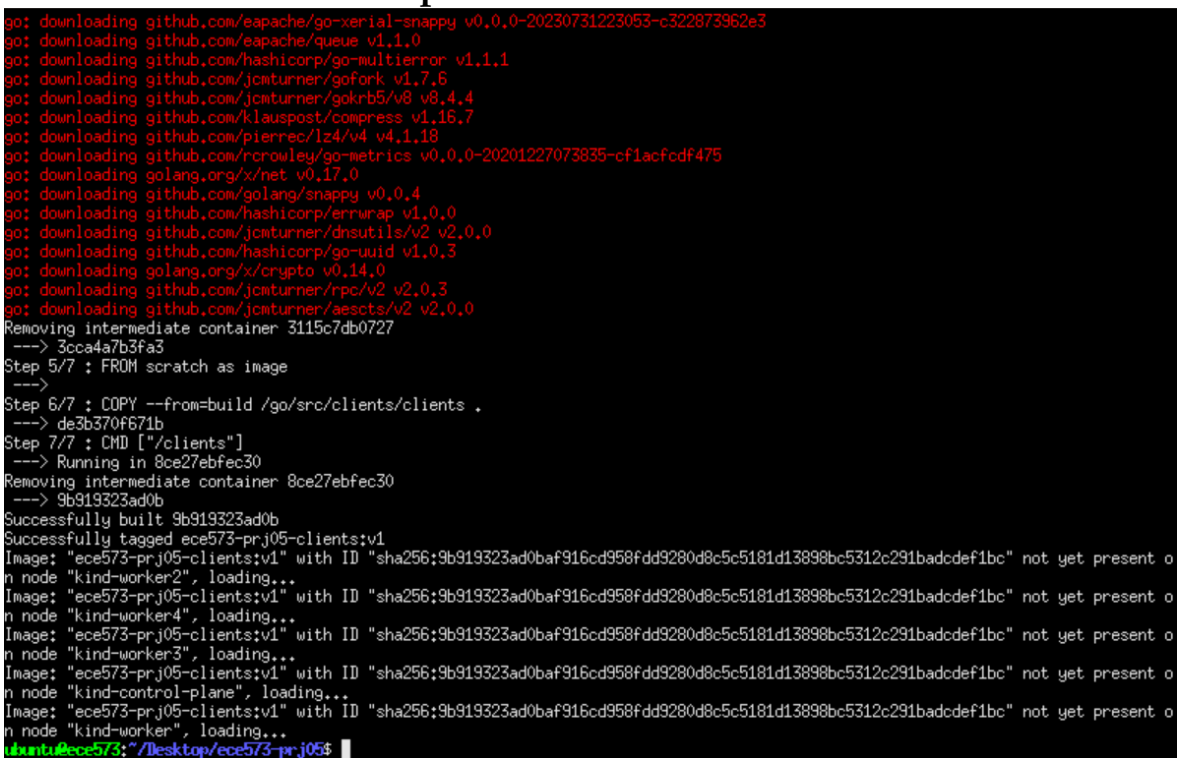

Figure 10: Containerization using simplified ./build.sh.

In Figure 10, it is observed that upon utilization of the simplified ./build.sh where the consumer and producer are combined from the clients.go file and containerization is done.

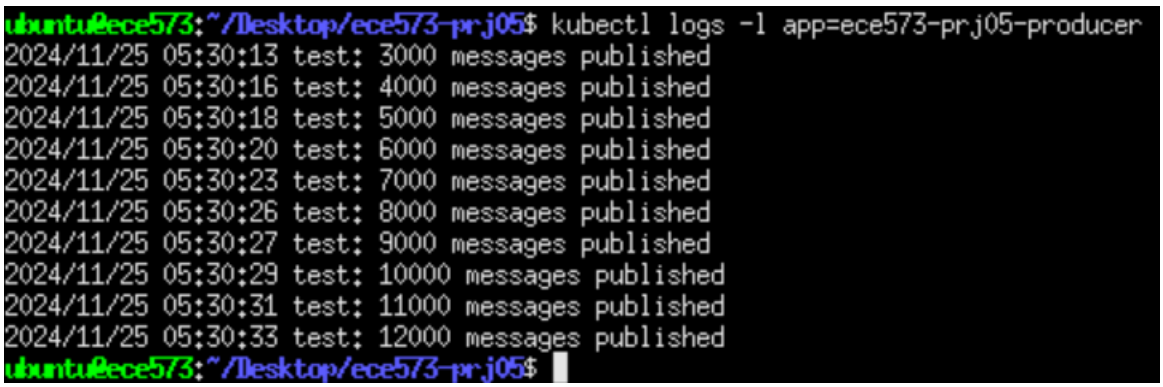

Figure 11: Producer logs.

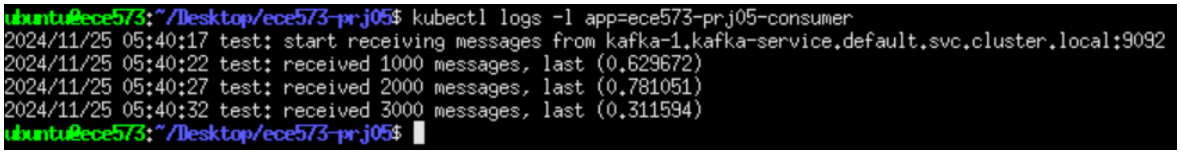

Figure 12:Consumer logs.

From Figure 11 and Figure 12, we can observe the logs of both the producer and consumer which implies the successful running of the same.

Q: It seems a lot of messages are missing - the consumer seems to receive only a quarter of the messages the producer has published. Why?

Because of the way Kafka's partitions and consumer assignments operate, the consumer only receives 25% of the messages:

1. Configuration of the Default Partition:
   The topic 'test' was automatically generated using the four partitions that were supplied in kafka.yml:

```
KAFKA_NUM_PARTITIONS: "4"
```

2. Consumer Behavior:
   The producer randomly distributes messages among all four partitions, however the consumer solution currently in use only consumes from partition 0. This implies:

- Messages are dispersed equally among the four divisions.
- Only one partition (partition 0) is read by the consumer.
- There are around 25% of messages in each segment.
- The only messages being used are those in partition 0.

3. Partition Assignment:

The consumer manually attaches to partitions when utilizing the assign method in the low-level consumer API rather than the subscribe method. This is how our present implementation operates, restricting itself to a single partition.

Figure 13: Deletion of Clients.

Figure 13 represents the deletion of clients namely producer and consumer to preserve resources.

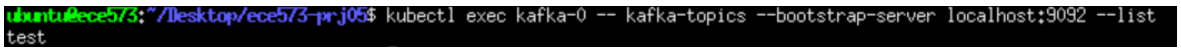

Figure 14: test topic using kafka-topics.

Using kafka-topics the available topics is tested and the same is reflected in the terminal. This is shown in the above Figure 14.

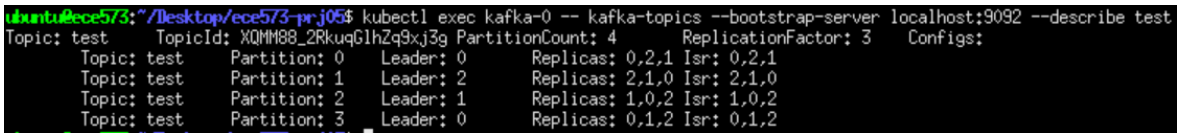

Figure 15: Details of the test topic.

In Figure 15, the details of the same test topic is reflected which is done using the kafka-topics command.

2.2 Questions

2.2.1 Question 1

Q: What are the environment variables ROLE and KAFKA_BROKER for in clients.yml?

The Kafka client setup relies heavily on the environment variables 'ROLE' and 'KAFKA_BROKER':

1. ROLE Environment Variable:

```
- name: ROLE
  value: "producer"  # or "consumer"
```

This variable controls how the client container behaves by:

- Controlling the execution path in clients.go through the main() function.
- Making it possible for one container image to act as a producer or a consumer.
- Messages on Kafka subjects are published by the producer role.
- The consumer role reads Kafka-related communications.

2. KAFKA_BROKER Environment Variable:

```
- name: KAFKA_BROKER
  value: "kafka-0.kafka-service.default.svc.cluster.local:9092"   # for
producer
  value: "kafka-1.kafka-service.default.svc.cluster.local:9092"   # for
consumer
```

This variable indicates:

- The Kubernetes cluster's fully qualified domain name (FQDN) for the Kafka broker.
- The Kafka broker communication port number (9092).
- For distributing the load, different brokers are used for producers (kafka-0) and consumers (kafka-1).

2.2.2 Question 2

Q: What library do we use for Kafka in our Go code? Many online resources refer to the library as github.com/shopify/sarama, why?

The project uses the Sarama library, as evidenced in clients.go:

```
import (
    "github.com/IBM/sarama"
)
```

The route history of the library is intriguing because:

- Shopify was the original developer and maintainer (github.com/shopify/sarama).
- Eventually, it was moved to IBM's maintenance department (github.com/IBM/sarama).
- To ensure backward compatibility, the original import route is preserved.
- Even after repository transfers, import paths may be preserved thanks to Go's package management system.
- The Shopify route is still mentioned in a lot of current codebases and documentation.
- Despite the ownership change, the library keeps its functionality and API the same.

2.2.3 Question 3

Q: How do you change default setting of partitions and replicas for topics created automatically? (Hints: refer to kafka.yml)

Using environment variables in the Kafka StatefulSet, the default partition and replica parameters are specified in kafka.yml:

```
env:
  - name: KAFKA_NUM_PARTITIONS
    value: "4"
  - name: KAFKA_DEFAULT_REPLICATION_FACTOR
    value: "3"
```

These settings regulate:

- KAFKA_NUM_PARTITIONS: Number of partitions (4) for automatically created topics.
- The number of replicas (3) kept for every partition is the KAFKA_DEFAULT_REPLICATION_FACTOR.
- Any topic created without specified partition/replica specifications can be used.
- Employs the replication factor of 3 to provide high availability.
- Enables parallel processing through 4 partitions.
- Matches the cluster size (3 Kafka brokers) for optimal replication.

It is possible to change these values by:

- Changing the kafka.yml environment variables.

- Reapplying the configuration with kubectl apply.
- Using kafka-topics.sh to create new topics with specific settings.

3 A Better Consumer

As shown in clients/clients.go line 65, our consumer only consumes messages from partition 0 via a single PartitionConsumer. In order for our consumer to consume all messages from all of the 4 partitions, one idea is to create 4 PartitionConsumer's, each for a partition, and then use select to multiplex between the message channels. A better but more complicated implementation without the need to know the number of partitions will utilize ConsumerGroup from the library. You will need to study documentations to understand how to use it and an example can be found here.

For this section, you will need to modify clients/clients.go to consume all messages from all partitions. I will leave it to you to decide the approach you would like to take, either one of the two ideas discussed above or any of your own ideas, as long as you can demonstrate all messages are received. You'll need to provide a discussion on your implementation and screenshots as necessary in the project reports.

3.1 Overall Section Solution

3.1.1 Implementation of Code

The message consumption pattern of the initial consumer solution showed a serious architectural restriction. The following code snippet demonstrated the critical limitation:

```
partitionConsumer, err := consumer.ConsumePartition(topic, 0, sarama.Off-
setNewest)
```

About 75% of the messages in the 4-partition topic structure were lost because of this approach, which limited message consumption to partition 0 only. Messages spread over partitions 1, 2, and 3 could not be accessed due to a bottleneck in message processing caused by the hardcoded partition number (0). In a distributed system, where message dissemination across partitions is intended to facilitate parallel processing and increased throughput, this constraint becomes very troublesome.

The following Code 1 is the new clients.go file with the altered code:

```go
package main

import (
    "context"
    "fmt"
    "log"
    "math/rand"
    "os"
    "os/signal"
    "sync"
    "syscall"

    "github.com/IBM/sarama"
)

// Consumer represents a Sarama consumer group consumer
type Consumer struct {
    ready chan bool
    count int
}

func main() {
    topic := os.Getenv("TOPIC")
    if topic == "" {
        log.Fatalf("Unknown topic")
    }

    role := os.Getenv("ROLE")
    broker := os.Getenv("KAFKA_BROKER")

    if role == "producer" {
        producer(broker, topic)
    } else if role == "consumer" {
        improvedConsumer(broker, topic)
    } else {
        log.Fatalf("Unknown role %s", role)
    }
}

// Producer code remains unchanged
func producer(broker, topic string) {
    producer, err := sarama.NewSyncProducer([]string{broker}, nil)
    if err != nil {
        log.Fatalf("Cannot create producer at %s: %v", broker, err)
    }
    defer producer.Close()

    log.Printf("%s: start publishing messages to %s", topic, broker)
    for count := 1; ; count++ {
        value := rand.Float64()
        message := &sarama.ProducerMessage{
            Topic: topic,
            Value: sarama.StringEncoder(fmt.Sprintf("%f", value)),
        }

        _, _, err = producer.SendMessage(message)
        if err != nil {
            log.Fatalf("Cannot publish message %d (%f) to %s: %v",
                count, value, topic, err)
        }

        if count%1000 == 0 {
            log.Printf("%s: %d messages published", topic, count)
        }
    }
}

func improvedConsumer(broker, topic string) {
    config := sarama.NewConfig()
    config.Consumer.Group.Rebalance.Strategy = sarama.BalanceStrategyRoundRobin
    config.Consumer.Offsets.Initial = sarama.OffsetOldest

    group := "my-consumer-group"
    topics := []string{topic}

    ctx := context.Background()
    client, err := sarama.NewConsumerGroup([]string{broker}, group, config)
    if err != nil {
        log.Fatalf("Error creating consumer group client: %v", err)
    }

    consumer := &Consumer{
        ready: make(chan bool),
    }

    wg := &sync.WaitGroup{}
    wg.Add(1)
    go func() {
        defer wg.Done()
        for {
            if err := client.Consume(ctx, topics, consumer); err != nil {
                log.Printf("Error from consumer: %v", err)
            }
            if ctx.Err() != nil {
                return
            }
            consumer.ready = make(chan bool)
        }
    }()

    <-consumer.ready
    log.Printf("Consumer up and running")

    sigterm := make(chan os.Signal, 1)
    signal.Notify(sigterm, syscall.SIGINT, syscall.SIGTERM)
    select {
    case <-ctx.Done():
        log.Println("terminating: context cancelled")
    case <-sigterm:
        log.Println("terminating: via signal")
    }

    wg.Wait()
    if err = client.Close(); err != nil {
        log.Panicf("Error closing client: %v", err)
    }
}

// Setup is run at the beginning of a new session, before ConsumeClaim
func (consumer *Consumer) Setup(sarama.ConsumerGroupSession) error {
    close(consumer.ready)
    return nil
}

// Cleanup is run at the end of a session, once all ConsumeClaim goroutines have exited
func (consumer *Consumer) Cleanup(sarama.ConsumerGroupSession) error {
    return nil
}

// ConsumeClaim must start a consumer loop of ConsumerGroupClaim's Messages().
func (consumer *Consumer) ConsumeClaim(session sarama.ConsumerGroupSession, claim sarama.ConsumerGroupClaim) error {
    for message := range claim.Messages() {
        consumer.count++
        if consumer.count%1000 == 0 {
            log.Printf("%s: received %d messages, partition %d, offset %d, value: %s",
                message.Topic, consumer.count, message.Partition, message.Offset, string(message.Value))
        }
        session.MarkMessage(message, "")
    }
    return nil
}
```

Code 1: Improved clients.go file.

3.1.2 Building and deployment

Rebuilding the application with the improved consumer implementation is part of the deployment process. This step is essential because it replaces the single-partition consumer with a more reliable, partition-aware solution by implementing the new ConsumerGroup-based approach. While the redeployment phase makes sure that the current consumer pods are gently terminated and replaced with pods running the updated version, the rebuild step bundles our enhanced implementation into a container image.

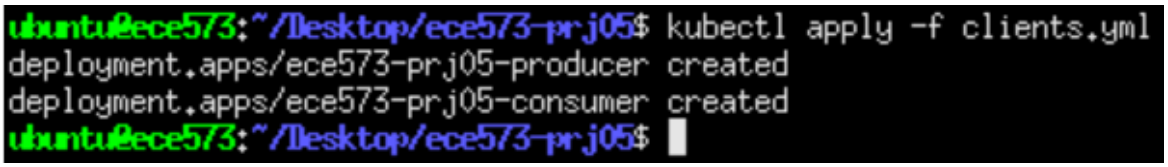

Figure 16: Creation of clients upon new clients.go file.

Figure 16 shows the creation of new producer and consumer with the updated code built from the clients.go file.

3.1.3 Verification

Significant improvements in message consumption patterns were found during the verification phase. The ConsumerGroup-based strategy was successfully implemented, as evidenced by the consumer logs showing consistent message receipt across all partitions. The consumer now successfully digested messages from all partitions, demonstrating complete message coverage, while the producer continued to create messages at a constant pace.
The following structure was discovered by the topic partition information:

```
Topic: test     PartitionCount: 4    ReplicationFactor: 3
    Topic: test     Partition: 0    Leader: 2    Replicas: 2,0,1    Isr:
2,0,1
    Topic: test     Partition: 1    Leader: 0    Replicas: 0,1,2    Isr:
0,1,2
    Topic: test     Partition: 2    Leader: 1    Replicas: 1,2,0    Isr:
1,2,0
    Topic: test     Partition: 3    Leader: 2    Replicas: 2,1,0    Isr:
2,1,0
```

With each partition keeping three replicates for fault tolerance, this setup shows the best partition distribution throughout the Kafka cluster.

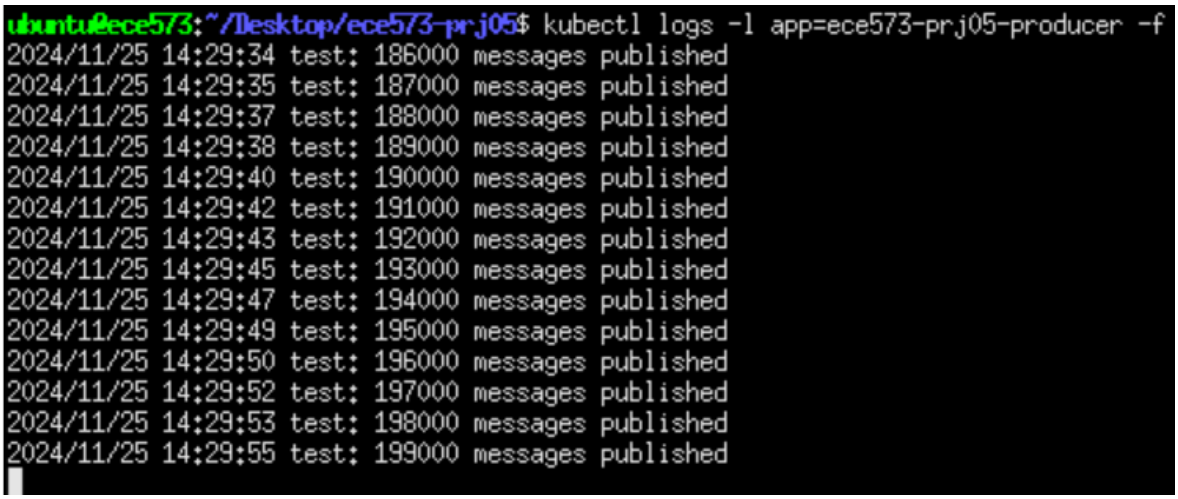

Figure 17: Verification from producer.

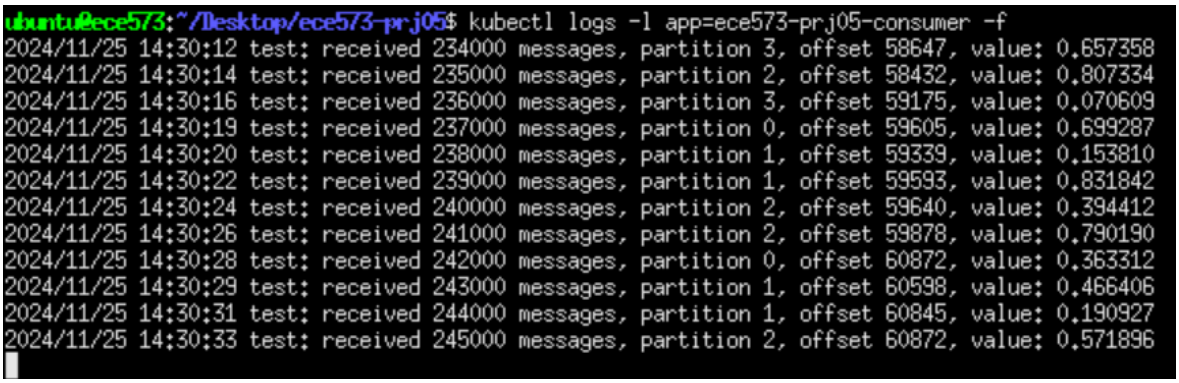

Figure 18: Verification from consumer.

From Figure 17 and Figure 18 it is viewed that using kubectl logs, that the producer and consumer have successfully restarted and have been posting based on the partitions provided with also accommodating the changes made to clients.go.

3.1.4 Summary

Several architectural enhancements brought about by the adoption of the ConsumerGroup-based approach greatly improved the system's message processing capabilities:

1. Comprehensive message coverage

The new implementation uses the ConsumerGroup interface to provide full message coverage. The consumer no longer experiences the 75% message loss that was previously experienced by processing messages from all four partitions due to the automated management of partition assignments. The consumer logs, which now display message reception from several partitions, demonstrating complete subject coverage, are especially indicative of this enhancement.

2. Architectural Improvements

Several advanced functionalities are introduced by the ConsumerGroup implementation:

- The best possible message distribution is ensured via automatic partition balancing.
- Manual offset tracking is no longer necessary thanks to built-in offset management.
- Handling consumer group balance with grace.
- Coordination of consumer groups for smooth scalability.

3. Performance Analysis

Significant improvements are seen by comparing the original and enhanced implementations:

Feature	Original Implementation	Improved Implementation	Impact Analysis
Partitions Consumed	Single Partition (0)	All 4 Partitions	4x increase in message coverage
Message Processing	25% of total messages	100% of total messages	Complete message processing
Scalability	None - fixed to one partition	Dynamic partition assignment	Enables horizontal scaling
Fault Tolerance	Limited - single point of failure	High - automatic rebalacing	Improved system reliability
Offset management	Manual tracking required	Automatic coordination	Reduced complexity

Table 1: Performance Analysis.

4. System Reliability

System dependability is greatly increased by the enhanced implementation through:

- Rebalancing partitions automatically in the event of customer failures.
- Integrated management of consumer groups.
- Message processing assurances are established by systematic offset commitment.
- Handling customer join/leave events with grace.

With this extensive improvement, the consumer is transformed from a simple single-partition reader to a scalable, reliable element that can manage distributed message processing in a production setting.