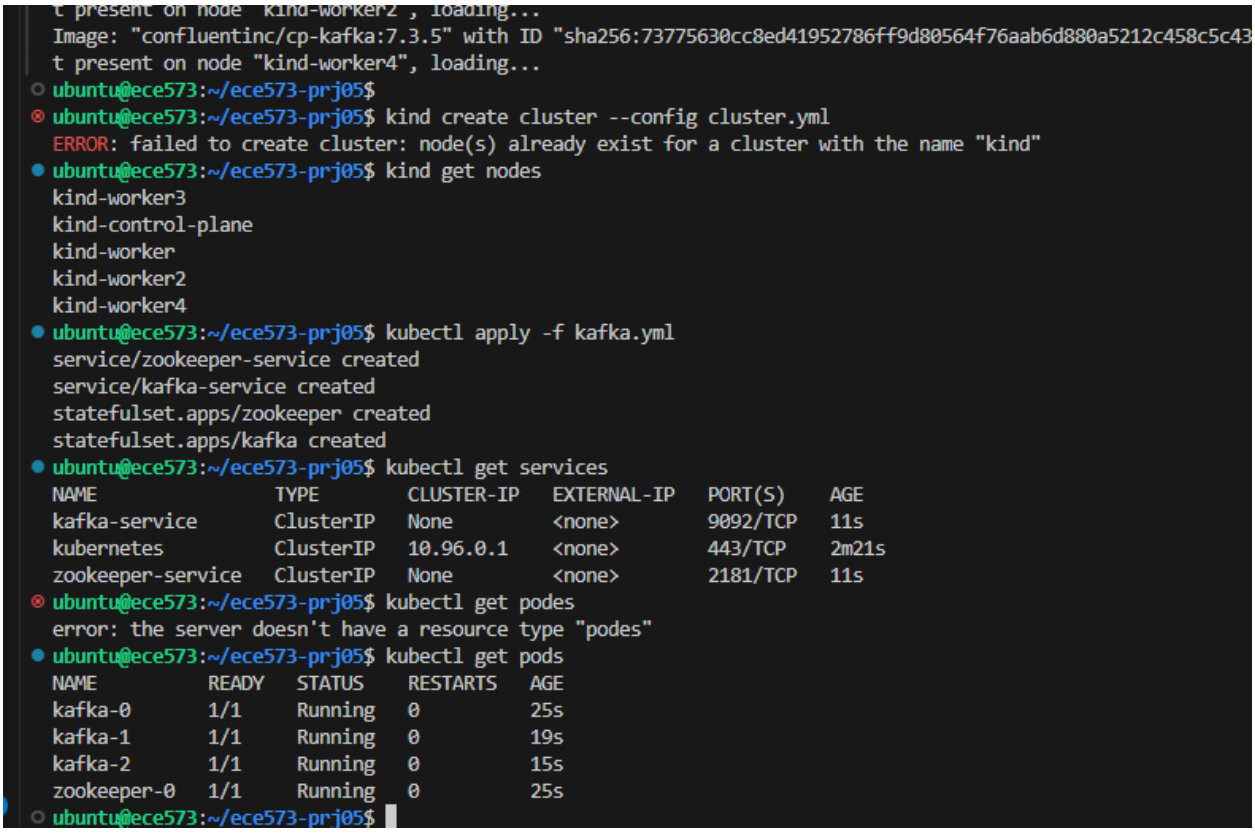
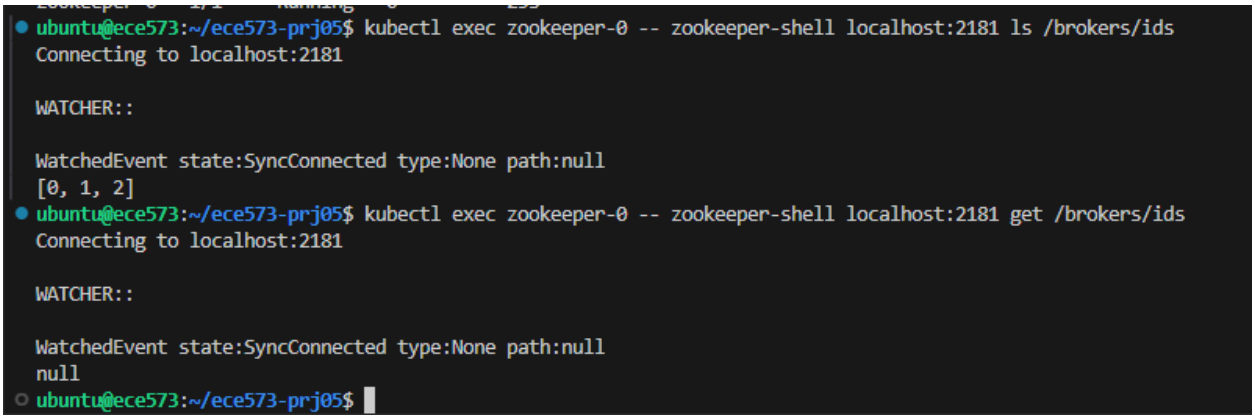
ECE 473/573 Fall 2024 - Project 5

Distributed Message Queues

Samman Chouhan A20561414

II. Kafka Services





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)

ZooKeeper Location: The KAFKA ZOOKEEPER _ _ CONNECT environment variable in the kafka.yml file provides the Kafka brokers with the location of the ZooKeeper service. The service name zookeeper-service, a Kubernetes service that encapsulates the underlying ZooKeeper pods, is assigned to this variable.

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

Here, 2181 is the default ZooKeeper port, and zookeeper-service is the ZooKeeper service's DNS name.
Broker Cooperation:
The KAFKA ADVERTISED LISTENERS environment _ variable in the kafka.yml file is how Kafka brokers are aware of one another. The advertised address and listener for broker communications are specified by this variable. Each broker has a unique ID, and the advertised address contains the Kafka service's DNS name (kafka-service).

- name: KAFKA_ADVERTISED_LISTENERS
Value:
"PLAINTEXT://$(POD_NAME).kafka-service.default.svc.cluster.local:9092"
- name: KAFKA_HEAP_OPTS
value: "
-Xms512m -Xmx512m"
- name: KAFKA_NUM_PARTITIONS

value: "4"
- name: KAFKA_DEFAULT_REPLICATION_FACTOR
value: "3"

By using the designated kafka-service DNS name and port 9092, this setup guarantees that Kafka brokers may cooperate and exchange information.

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?)

Zookeeper-shell: The ZooKeeper service operating in the same Pod as the zookeeper-shell command is denoted by localhost:2181 when utilizing zookeeper-shell to communicate with ZooKeeper. This is due to ZooKeeper's usual local access from within its own pod.

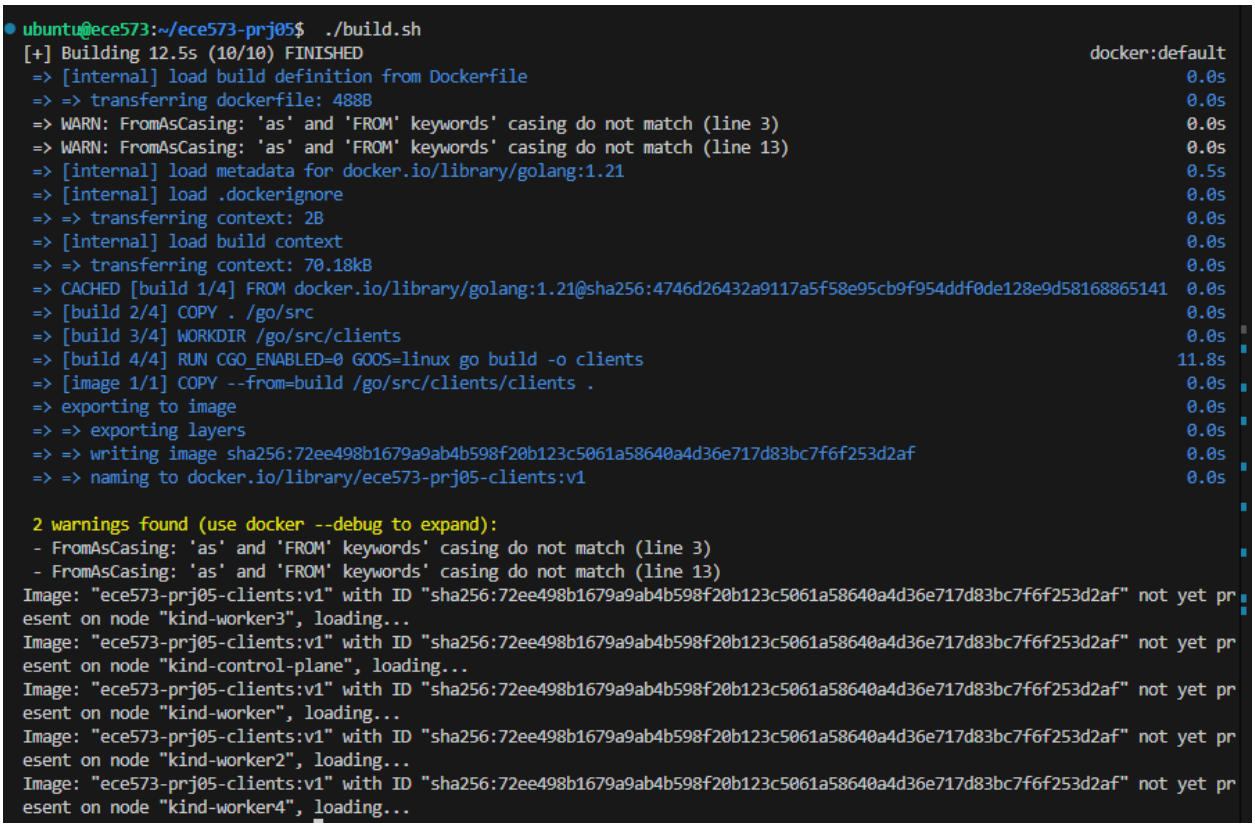
kubectl exec zookeeper-0 -- zookeeper-shell localhost:2181 ls /brokers/ids

Kafka-topics: Likewise, localhost:9092 denotes the Kafka service operating in the same Pod as the kafka-topics command when utilizing kafka-topics to manage Kafka topics. Kafka topics can be handled at this address.

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

In conclusion, localhost—rather than the K8s node—is a reference to the local environment within each of the Pods that the zookeeper-shell and kafka-topics employ. The Kubernetes cluster's service names (kafka-service and zookeeper-service) permit communication between the services, which are accessed internally within the same Pod.

III. Topics and Messages



```
ubuntu@ece573:~/ece573-prj05$ kubectl apply -f clients.yml
deployment.apps/ece573-prj05-producer created
deployment.apps/ece573-prj05-consumer created
ubuntu@ece573:~/ece573-prj05$  kubectl logs -l app=ece573-prj05-producer
2024/11/23 07:35:35 test: 6000 messages published
2024/11/23 07:35:36 test: 7000 messages published
2024/11/23 07:35:37 test: 8000 messages published
2024/11/23 07:35:38 test: 9000 messages published
2024/11/23 07:35:39 test: 10000 messages published
2024/11/23 07:35:39 test: 11000 messages published
2024/11/23 07:35:40 test: 12000 messages published
2024/11/23 07:35:41 test: 13000 messages published
2024/11/23 07:35:41 test: 14000 messages published
2024/11/23 07:35:42 test: 15000 messages published
ubuntu@ece573:~/ece573-prj05$  kubectl logs -l app=ece573-prj05-consumer
2024/11/23 07:35:54 test: received 15000 messages, last (0.332517)
2024/11/23 07:35:55 test: received 16000 messages, last (0.769466)
2024/11/23 07:35:56 test: received 17000 messages, last (0.647130)
2024/11/23 07:35:57 test: received 18000 messages, last (0.682368)
2024/11/23 07:35:57 test: received 19000 messages, last (0.604648)
2024/11/23 07:35:58 test: received 20000 messages, last (0.076494)
2024/11/23 07:35:59 test: received 21000 messages, last (0.413029)
2024/11/23 07:36:00 test: received 22000 messages, last (0.603241)
2024/11/23 07:36:01 test: received 23000 messages, last (0.206967)
2024/11/23 07:36:02 test: received 24000 messages, last (0.669142)
ubuntu@ece573:~/ece573-prj05$ kubectl delete -f clients.yml
deployment.apps "ece573-prj05-producer" deleted
deployment.apps "ece573-prj05-consumer" deleted
ubuntu@ece573:~/ece573-prj05$ kubectl exec kafka-0 -- kafka-topics --bootstrap-server localhost:9092 --list
test
ubuntu@ece573:~/ece573-prj05$ kubectl exec kafka-0 -- kafka-topics --bootstrap-server localhost:9092 --describe test
Topic: test     TopicId: 4wN_arQ8RF-y9KbIxUG8lQ PartitionCount: 4       ReplicationFactor: 3    Configs:
        Topic: test     Partition: 0    Leader: 1       Replicas: 1,2,0 Isr: 1,2,0
        Topic: test     Partition: 1    Leader: 0       Replicas: 0,1,2 Isr: 0,1,2
        Topic: test     Partition: 2    Leader: 2       Replicas: 2,0,1 Isr: 2,0,1
        Topic: test     Partition: 3    Leader: 1       Replicas: 1,0,2 Isr: 1,0,2
ubuntu@ece573:~/ece573-prj05$
```

What are the environment variables ROLE and KAFKA clients.yml?

in relation to the project's customers.YML file:
Environment Variable for ROLE:
Most frequently, the ROLE environment variable indicates the client's role or kind, such as "producer" or "consumer." In the Go code, it is employed to distinguish between various Kafka client kinds and ascertain the activities or behavior of the client.
KAFKA_BROKER environment variable: This variable most likely indicates the hostname or address of the Kafka broker to which the client should establish a connection. Usually, it contains the port and address of the Kubernetes cluster's Kafka broker.

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 github.com/Shopify/sarama module is frequently used in our Go programs for Kafka integration.
Its popularity can be attributed to several factors:
Rich Feature Set: among other things, Samarama supports message compression, offset management, custom partitioning, and message production and consumption.
Actively Maintained: Saraam was a dependable option for Kafka integration as of my most recent knowledge update in January 2022 because it was actively maintained.
Widespread Adoption: A plethora of community expertise and support has resulted from the successful use of sarama in Kafka-related projects by numerous Go developers and organizations.
Compatibility: It is made to work with various Kafka versions, giving users flexibility.

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

Usually, you would alter the kafka.yml file to alter the default partition and replica parameters for topics that are generated automatically. Search for settings pertaining to the topic creation parameters that are set to default. For instance:

```
# Example configuration in kafka.yml
...
env:
- name: KAFKA
CREATE
TOPICS
_
_
value: "topic1:3:1,topic2:2:2"
...
```

The default topics to be formed in this example are specified by KAFKA_CREATE TOPICS, each of which has a predetermined number of partitions and replicas.
Topic:partitions:replicas is the format. These settings can be changed to suit your needs.

IV. A Better Consumer

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.

```go
func consumer(broker, topic string) {
    consumer, err := sarama.NewConsumer([]string{broker}, nil)
    if err != nil {
        log.Fatalf("Cannot create consumer at %s: %v", broker, err)
    }
    defer consumer.Close()

    partitions, err := consumer.Partitions(topic)
    if err != nil {
        log.Fatalf("Cannot create partition for the topic %s:%v", topic, err)
    }

    var wg sync.WaitGroup
    wg.Add(len(partitions))
    for _, partition := range partitions {
        go func(partition int32) {
            defer wg.Done()
            partitionConsumer, err := consumer.ConsumePartition(topic, partition, sarama.OffsetNewest)
            if err != nil {
                log.Printf("Error in creating partition consumer fr partition %d:%v", partition, err)
                return
            }
            defer partitionConsumer.Close()
            log.Printf("%s: start receiving messages from Partition %d", topic, partition)
            for count := 1; ; count++ {
                select {
                case msg := <-partitionConsumer.Messages():
                    if count%1000 == 0 {
                        log.Printf("%s: received %d messages from partition %d, last (%s)", topic, count, partition,
                    }
                case err := <-partitionConsumer.Errors():
                    log.Printf("Error consuming from partition %d:%v", partition, err)
                }
            }
        }(partition)
    }
}
```

```
ubuntu@ece573:~/ece573-prj05$  kubectl logs -l app=ece573-prj05-producer
2024/11/23 07:35:35 test: 6000 messages published
2024/11/23 07:35:36 test: 7000 messages published
2024/11/23 07:35:37 test: 8000 messages published
2024/11/23 07:35:38 test: 9000 messages published
2024/11/23 07:35:39 test: 10000 messages published
2024/11/23 07:35:39 test: 11000 messages published
2024/11/23 07:35:40 test: 12000 messages published
2024/11/23 07:35:41 test: 13000 messages published
2024/11/23 07:35:41 test: 14000 messages published
2024/11/23 07:35:42 test: 15000 messages published
ubuntu@ece573:~/ece573-prj05$  kubectl logs -l app=ece573-prj05-consumer
2024/11/23 07:35:54 test: received 15000 messages, last (0.332517)
2024/11/23 07:35:55 test: received 16000 messages, last (0.769466)
2024/11/23 07:35:56 test: received 17000 messages, last (0.647130)
2024/11/23 07:35:57 test: received 18000 messages, last (0.682368)
2024/11/23 07:35:57 test: received 19000 messages, last (0.604648)
2024/11/23 07:35:58 test: received 20000 messages, last (0.076494)
2024/11/23 07:35:59 test: received 21000 messages, last (0.413029)
2024/11/23 07:36:00 test: received 22000 messages, last (0.603241)
2024/11/23 07:36:01 test: received 23000 messages, last (0.206967)
2024/11/23 07:36:02 test: received 24000 messages, last (0.669142)
```

Discussion:

Initialization of the Consumer: Samarama is used to initialize the consumer.NewConsumer using the Kafka broker addresses that are supplied.
The program ends with a fatal log message if there is a problem creating the consumer.

```go
func consumer(broker, topic string) {
    consumer, err := sarama.NewConsumer([]string{broker}, nil)
    if err != nil {
        log.Fatalf("Cannot create consumer at %s: %v", broker, err)
    }
    defer consumer.Close()
```

Partition Retrieval: Using a consumer, the application obtains the partition list for the designated Kafka topic.partitions (subject).
A fatal log message is displayed when the software terminates if there is a problem retrieving partitions.

```
partitions, err := consumer.Partitions(topic)
if err != nil {
    log.Fatalf("Cannot create partition for the topic %s:%v", topic, err)
}
```

Goroutine Concurrency: A Sync.WaitGroup is used to hold off on leaving until all partition consumers have completed their work. A goroutine is started for every partition in order to manage message consumption simultaneously. Every time a goroutine is launched, the WaitGroup counter is increased.

```
var wg sync.WaitGroup
wg.Add(len(partitions))
```

Partition Consumer Goroutine: Each goroutine uses consumer to build a new PartitionConsumer.ConsumePartition. In order to listen for messages from the partition, the goroutine enters a continuous loop. Log statements give details on when each partition's message receiving began.
for count := 1; ; count++

```
log.Printf("%s: start receiving messages from Partition %d", topic, partition)
for count := 1; ; count++ {
    select {
    case msg := <-partitionConsumer.Messages():
        if count%1000 == 0 {
            log.Printf("%s: received %d messages from partition %d, last (%s)", topic, count, partition, string(msg.Value))
        }
    case err := <-partitionConsumer.Errors():
        log.Printf("Error consuming from partition %d:%v", partition, err)
    }
```