Project Milestone
Data Ingestion Software: Kafka Clusters
Cloud Computing

Danial Asghar

100671850

EDA

Event Driven Architecture is a software design pattern and architecture. Unlike a tradition Request-response model, EDA focuses on capture, ingestion, communication, and processing of events. Following an event-based methodology allows organizations to detect "events" and action upon them in near real-time.

Producers are the entities responsible for capturing and communicating the events to an event broker, i.e., Kafka. Consumers are the entities who consume these events from the broker to action upon them. EDA is a loosely coupled architecture, producers and consumers do not know the other party. This makes EDA a great approach to employ for modern distributed systems with M:N connections.

Advantages	Disadvantages
Loose coupling: producers and consumers are not dependent on each other and can be added/removed/upgraded without any issues.	Error handling: error handling is more complex in a large-scale system with decoupled parties.
Fault tolerance: loose coupling ensures that the system is more fault tolerant to specific services failing. For example, communication can be queued in a broker until a subscriber is repaired.	Traditional methods of using API hit numbers in analytics can not be replicated using EDA.
Scalability: asynchronous and non-blocking systems are best suited for high scalability.	
Concurrency: Kafka's immutable events makes handling concurrency in distributed systems easier.	

Kafka Terminology

Cluster: A Kafka cluster is a set of more than one servers, Kafka brokers, running together.

Broker: A broker is a server which runs Kafka within a cluster. Brokers can be either a leader, follower, or controller.

Topic: Kafka topics are a method of categorizing events. Using storage as an analogy, a topic can be thought of as a folder and the events the files stored within. Each topic name is unique within a cluster. Topics can be read and written to by any number of producers/consumers. The events within a topic are immutable and retained based on the retention policy.

Replica: Replica in Kafka means keeping multiple copies of the same data. Within a cluster, events are replicated across all the brokers. This helps achieve high availability because if one broker fails, the data can be accessible as it should have been replicated to the other brokers. Events are only available to consumers once it has been replicated to all brokers.

Partition: Topics within Kafka are partitioned to separate the logs across each node of the cluster. Each partition is a single log file where events are ammended in an immutable fashion.

Zookeeper: The zookeeper is responsible for managing Kafka brokers, it tracks the status of brokers and used to notify parties if a broker is added or removed from the cluster. It also keeps a track of topics and partitions.

Controller: Each Kafka cluster has one elected controller responsible for managing partition states, replicas and administrative duties. When a cluster is spun up, every broker tries to register with the Zookeeper as the controller. The first one is assigned as the controller and rest are notified that a controller exists. If the controller fails or disconnects from Zookeeper, the same process happens again.

Leader: Each partition in Kafka has one leader, while the others behave as followers. The leader is responsible for R/W operations for the specific partition. The followers query the leader for updates and replicate the data from the leader.

Consumer: A consumer is a client application subscribed to a Kafka topic which consumes events. There can be any number of consumers within Kafka subscribed to one or more topics.

Producer: Producers are clients who publish data into a Kafka cluster. Producers write the data to a specific topic, and they can also load balance across brokers based on partitions.

Consumer Group: Consumer groups are a set of consumers who cooperate in consuming data from a topic by joining a group using the same group-id. Partitions of topics are divvied up amongst the members of a group. As such, when a message is consumed within a consumer group it is delivered to exactly one consumer.

Kafka Video

The following video will show Kafka topics being generated, messages being produced and consumed, in both NodeJS, Python and possible combinations:

https://drive.google.com/file/d/1LiZmV4MyIuoJE-3HbEVc-SeHi4oztV0T/view?usp=sharing

Docker Persistent Volumes

Docker containers by default are in-memory and do not persist any data between instantiations. The preferred method of persisting data within a container is by using Docker Volumes. According to Confluent, Kafka and Zookeeper require externally mounted volumes where the data is persisted whenever the container is started/stopped. Volumes are the preferred method because they keep the data outside of the container, hence not bloating the size of the container. The volume's contents are independent of the container's lifecycle.

The following image shows the absolute path of the local volume mounted to the Kafka Zookeeper data and Zookeeper log folders.

```
volumes:
  - "/C:/Users/Danial Asghar/Downloads/Lab 2/zk-data:/var/lib/zookeeper/data"
  - "/C:/Users/Danial Asghar/Downloads/Lab 2/zk-txn-logs:/var/lib/zookeeper/log"
```

The following images show the volumes mounted to each separate broker to persistent their data.

```
volumes:
- "/C:/Users/Danial Asghar/Downloads/Lab 2/kafka-data-1:/var/lib/kafka/data"
```

```
volumes:
- "/C:/Users/Danial Asghar/Downloads/Lab 2/kafka-data-2:/var/lib/kafka/data"
```

```
volumes:
- "/C:/Users/Danial Asghar/Downloads/Lab 2/kafka-data-3:/var/lib/kafka/data"
```

Confluent Video

The following video will show the usage of Confluent CLI to create a topic, consumer and producer. The second part uses Python scripts to create the producer and consumer:

https://drive.google.com/file/d/1SfwvxJtsOhn0kDVCVQtzYz2c2yKLCj_F/view?usp=sharing