Apache Kafka (Message Queue) with Code

Agenda

- Apache Kafka introduction
- Messaging system
- Apache Kafka as a Message system
- Apache Kafka Architecture
- Apache Kafka Work flow
- Apache kafka Core API
- Apache kafka Components
- Apache kafka Use cases
- RabbitMQ Vs Kafka
- Implementation Code

Apache Kafka introduction

Apache Kafka is a software platform which is based on a distributed streaming process.
It is a publish-subscribe messaging system which let exchanging of data between applications, servers, and processors as well.

Apache Kafka was originally developed by **LinkedIn in 2010**, and later it was donated to the Apache Software Foundation. Currently, it is maintained by **Confluent** under Apache Software Foundation.

In the year **2011** Kafka was made public.

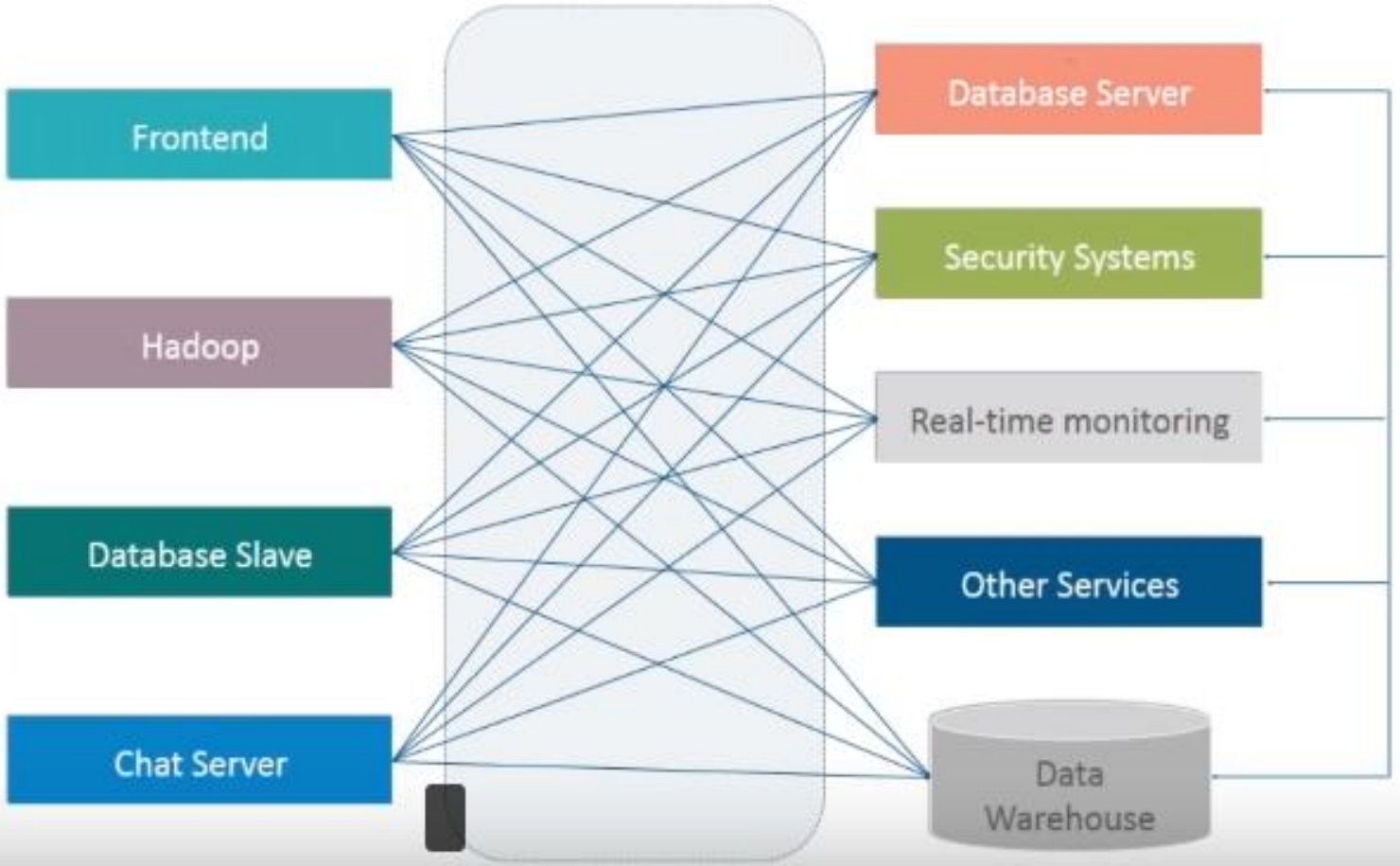
Kafka works well as a replacement for a more traditional message broker.
Apache Kafka has resolved the lethargic trouble of data communication between a sender and a receiver.

Agenda

- Apache Kafka introduction
- Messaging system
- Apache Kafka as a Message system
- Apache Kafka Architecture
- Apache Kafka Work flow
- Apache kafka Core API
- Apache kafka Components
- Apache kafka Use cases
- RabbitMQ Vs Kafka
- Implementation Code

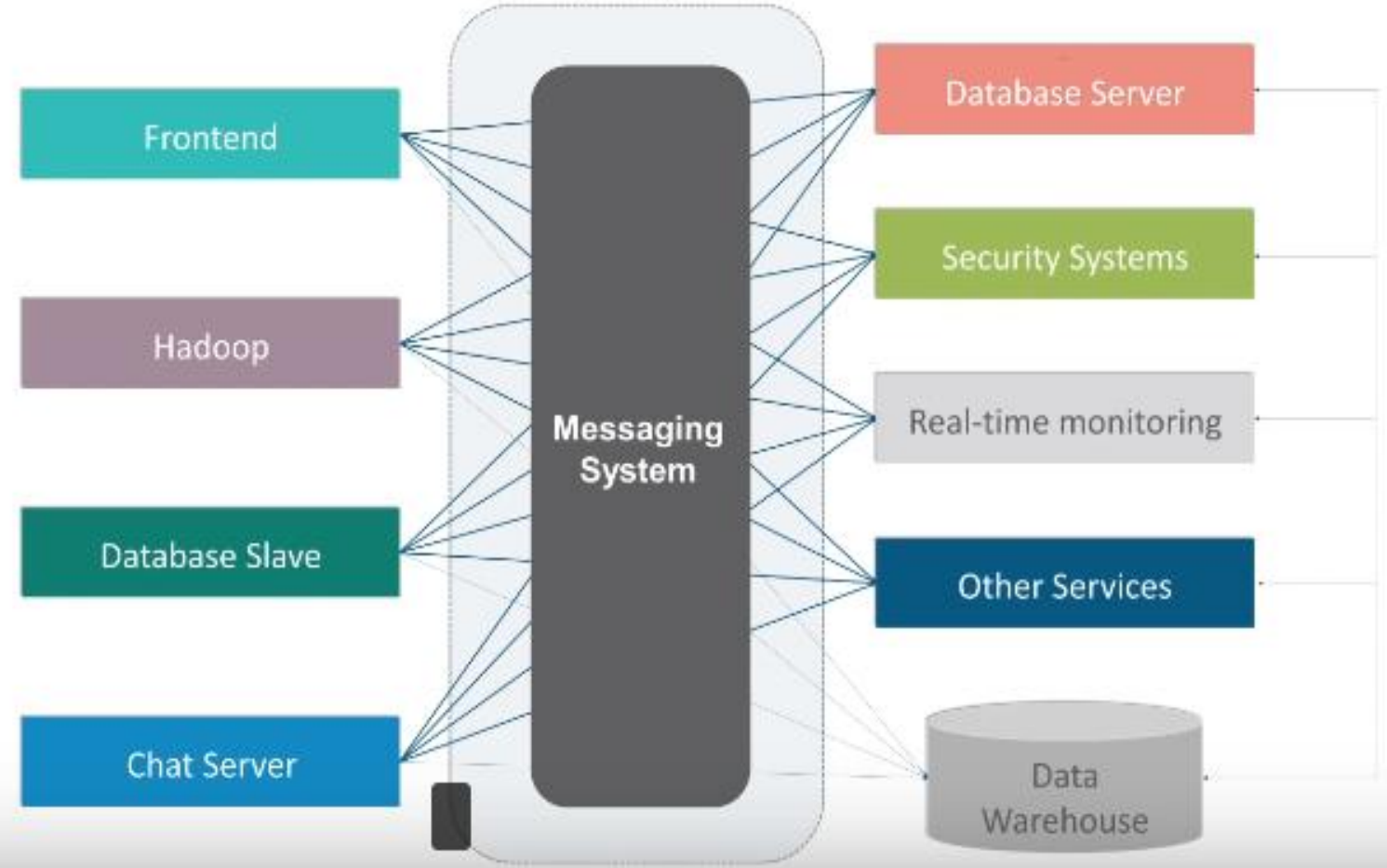
Complex Data Pipelines

Similarly, applications may also be communicating with Real-time monitoring and Other services in real-time scenario



Solution to the Complex Data Pipelines

Messaging Systems helps managing the complexity of the pipelines



Messaging System

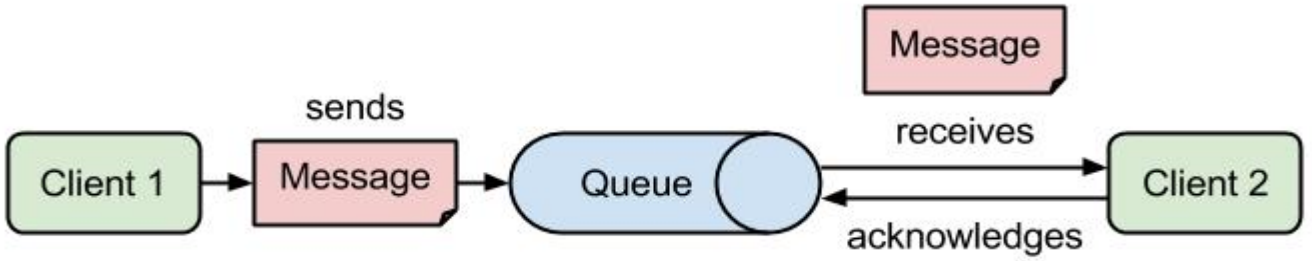
There are two types of Messaging System:

1. **Point to Point System**
2. **Publish-Subscribe System**

1. Point to Point System

Messages are persisted in a queue, but a particular message can be consumed by a maximum of one consumer only. Once a consumer reads a message in the queue, it disappears from that queue.

The typical example of this system is an Order Processing System, where each order will be processed by one Order Processor, but Multiple Order Processors can work as well at the same time. The following diagram depicts the structure.

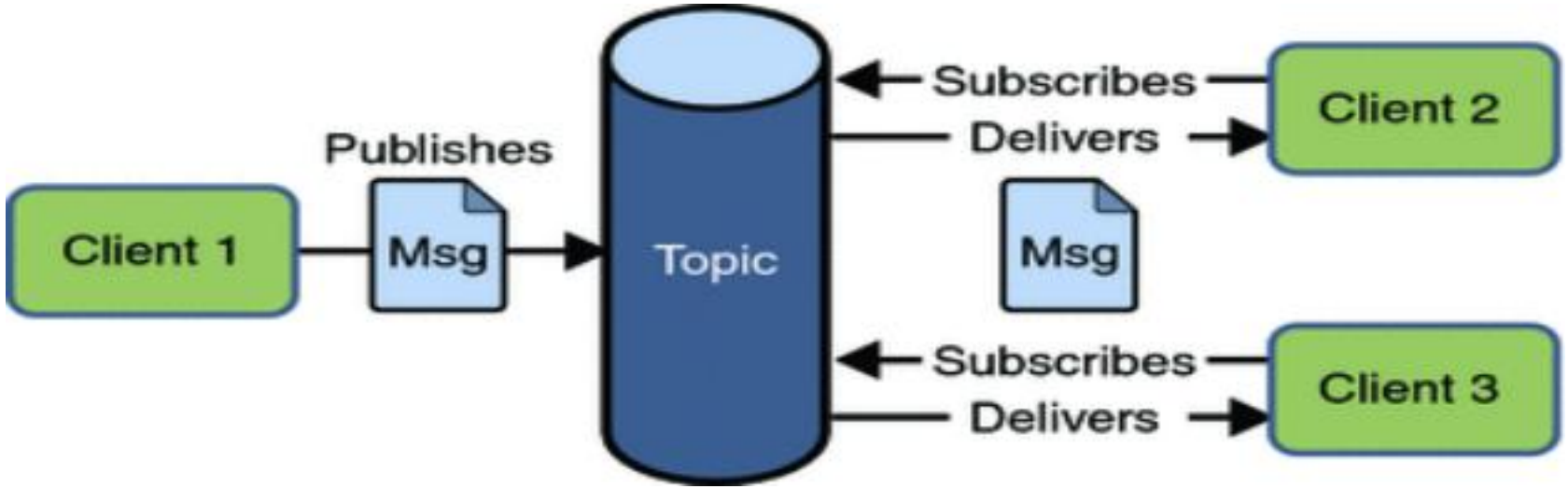


Messaging System

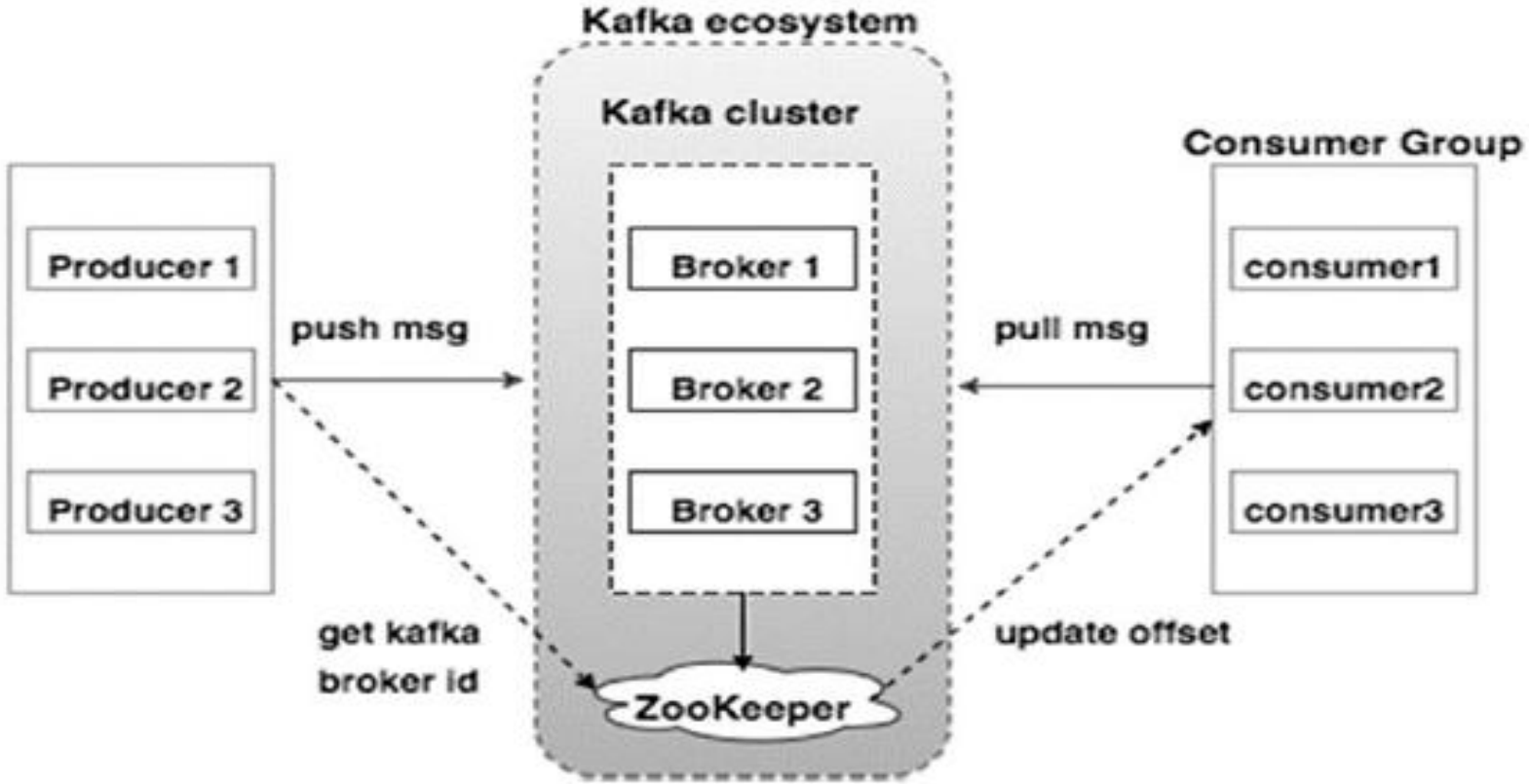
2. Publish-Subscribe System

Messages are persisted in a topic. Unlike point-to-point system, consumers can subscribe to one or more topic and consume all the messages in that topic. In the Publish-Subscribe system, message producers are called publishers and message consumers are called subscribers.

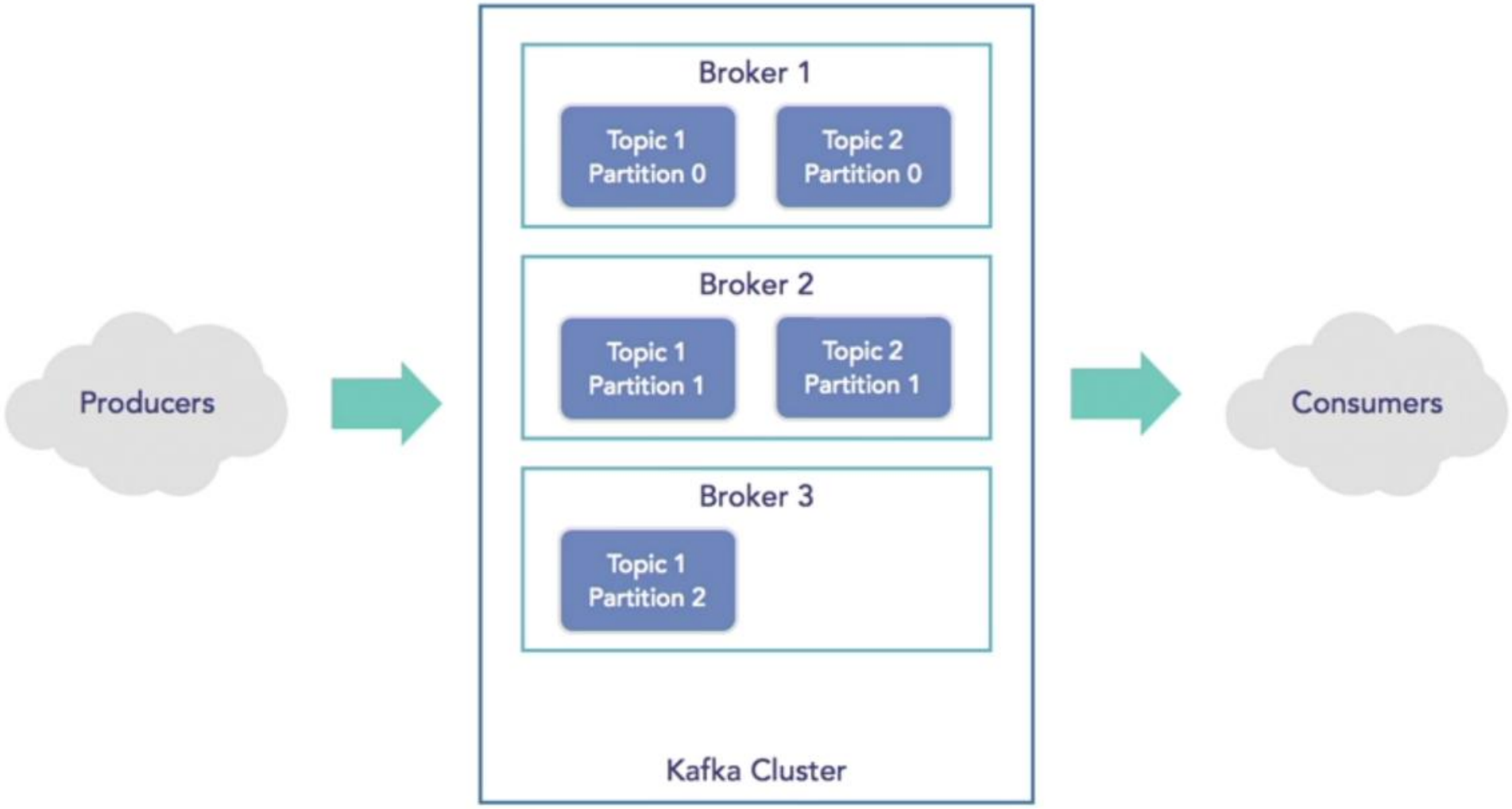
A real-life example is Dish TV, which publishes different channels like sports, movies, music, etc., and anyone can subscribe to their own set of channels and get them whenever their subscribed channels are available.



Apache Kafka as a Messaging System



Apache Kafka Architecture



Apache Kafka Architecture Cont...

Kafka is a distributed, replicated commit log. Kafka does not have the concept of a queue which might seem strange at first, given that it is primary used as a messaging system. Queues have been synonymous with messaging systems for a long time. Let's break down "distributed, replicated commit log" a bit:

Distributed because Kafka is deployed as a cluster of nodes, for both fault tolerance and scale
Replicated because messages are usually replicated across multiple nodes (servers).

Kafka is so powerful regarding throughput and scalability that it allow you to handle continuous stream of messages.

Commit Log because messages are stored in partitioned, append only logs which are called Topics. This concept of a log is the principal killer feature of Kafka.

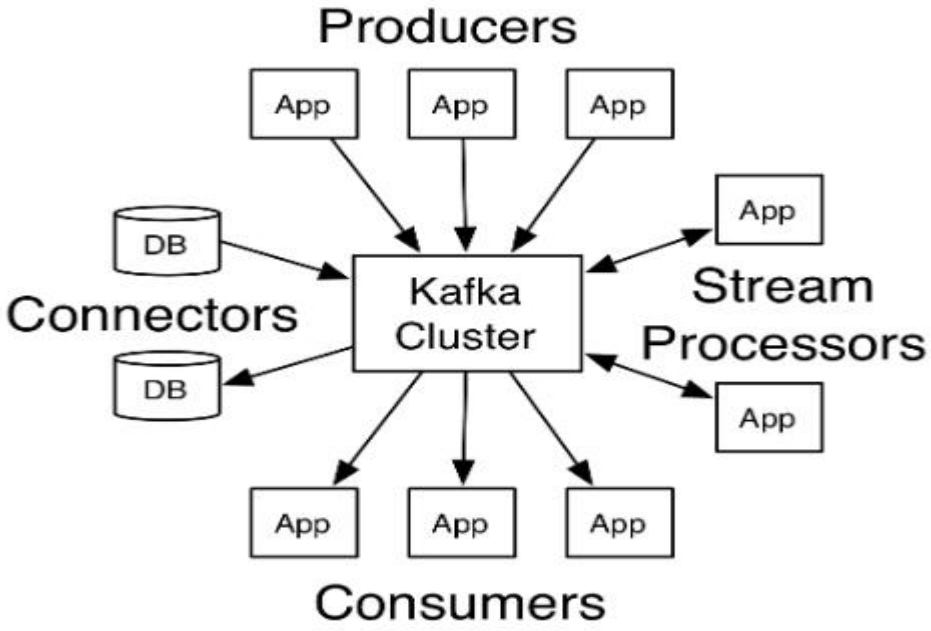
Apache Kafka Work Flow

Following is the step wise workflow of the Pub-Sub Messaging –

- Producers send message to a topic at regular intervals.
- Kafka broker stores all messages in the partitions configured for that particular topic. It ensures the messages are equally shared between partitions. If the producer sends two messages and there are two partitions, Kafka will store one message in the first partition and the second message in the second partition.
- Consumer subscribes to a specific topic.
- Once the consumer subscribes to a topic, Kafka will provide the current offset of the topic to the consumer and also saves the offset in the Zookeeper.
- Consumer will request the Kafka in a regular interval (like 100 Ms) for new messages.
- Once Kafka receives the messages from producers, it forwards these messages to the consumers.
- Consumer will receive the message and process it.
- Once the messages are processed, consumer will send an acknowledgement to the Kafka broker.
- Once Kafka receives an acknowledgement, it changes the offset to the new value and updates it in the Zookeeper. Since offsets are maintained in the Zookeeper.
- This above flow will repeat until the consumer stops the request.
- Consumer has the option to rewind/skip to the desired offset of a topic at any time and read all the subsequent messages

Apache Kafka Core API

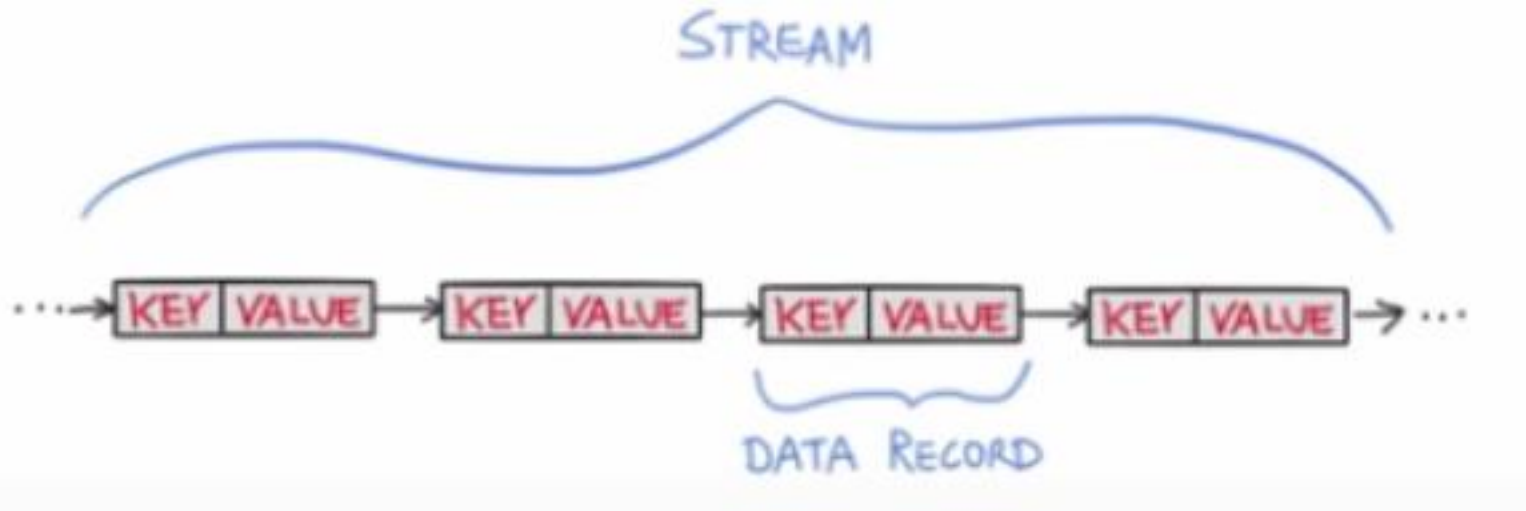
The Admin API is for manage and inspect topics, brokers and other kafka objects.
The Producer API allows an application to publish a stream of records to one or more Kafka topics.
The Consumer API allows an application to subscribe to one or more topics and process the stream of records produced to them.
The Streams API allows an application to act as a *stream processor*, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams.
The Connector API allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems. For example, a connector to a relational database might capture every change to a table.



What is Streams

Think of s Stream as an infinite. Continuous real-time flow of data.
data are a key-value pairs.

In **Kafka stream API** transform and increases data.
- Support per-record stream processing with milisecond.



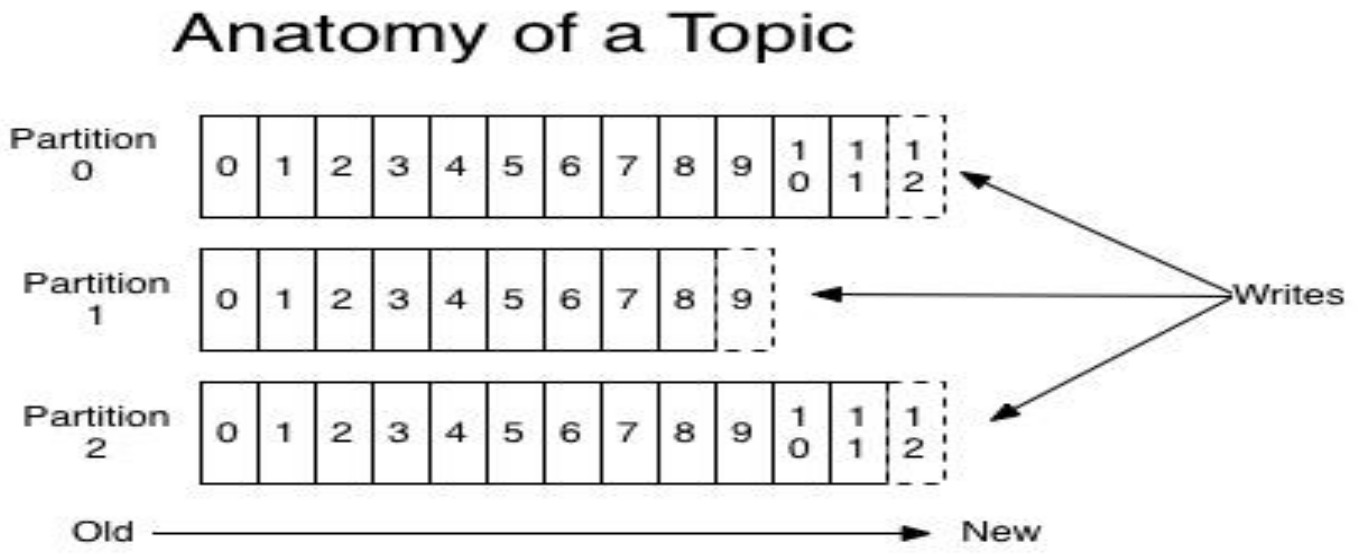
Kafka Components

Using the following components, Kafka achieves messaging:

1. Topic

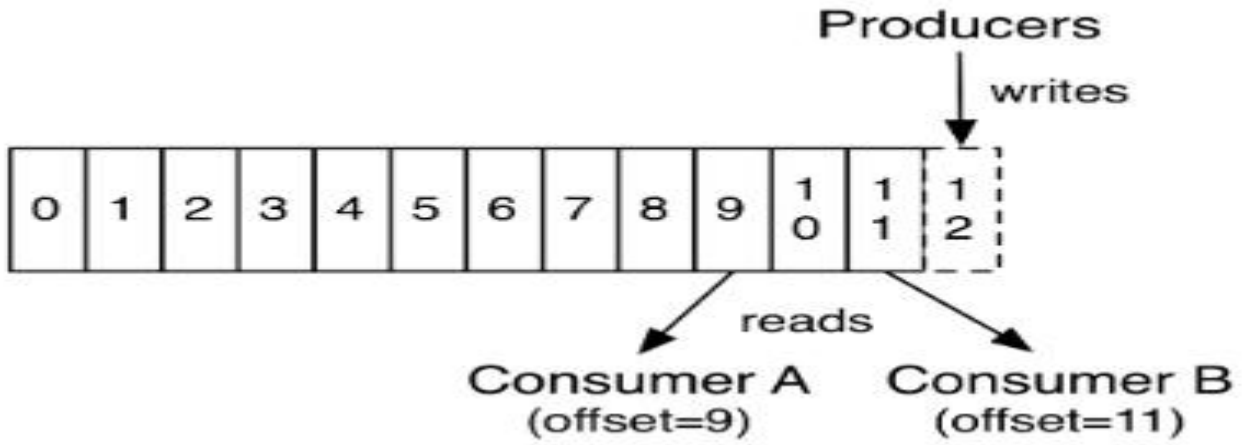
Basically, A Topic is a unique name for Kafka Stream. Topic is a category or feed name to which records are published, and stores messages. Topics in Kafka are always multi-subscriber; that is, a topic can have zero, one, or many consumers that subscribe to the data written to it.

Each partition is an ordered, **immutable** sequence of records that is continually appended to—a structured commit log. The records in the partitions are each assigned a sequential id number called the offset that uniquely identifies each record within the partition.



Kafka Components

The Kafka cluster durably persists all published records—whether or not they have been consumed—using a configurable retention period. For example, if the retention policy is set to two days, then for the two days after a record is published, it is available for consumption, after which it will be discarded to free up space. **Kafka's performance is effectively constant with respect to data size so storing data for a long time is not a problem**.
This is one of the biggest difference between RabbitMQ/ActiveMQ and Kafka.



Kafka Components

2. Kafka Producer
It publishes messages to a Kafka topic. The producer is responsible for choosing which record to assign to which partition within the topic.

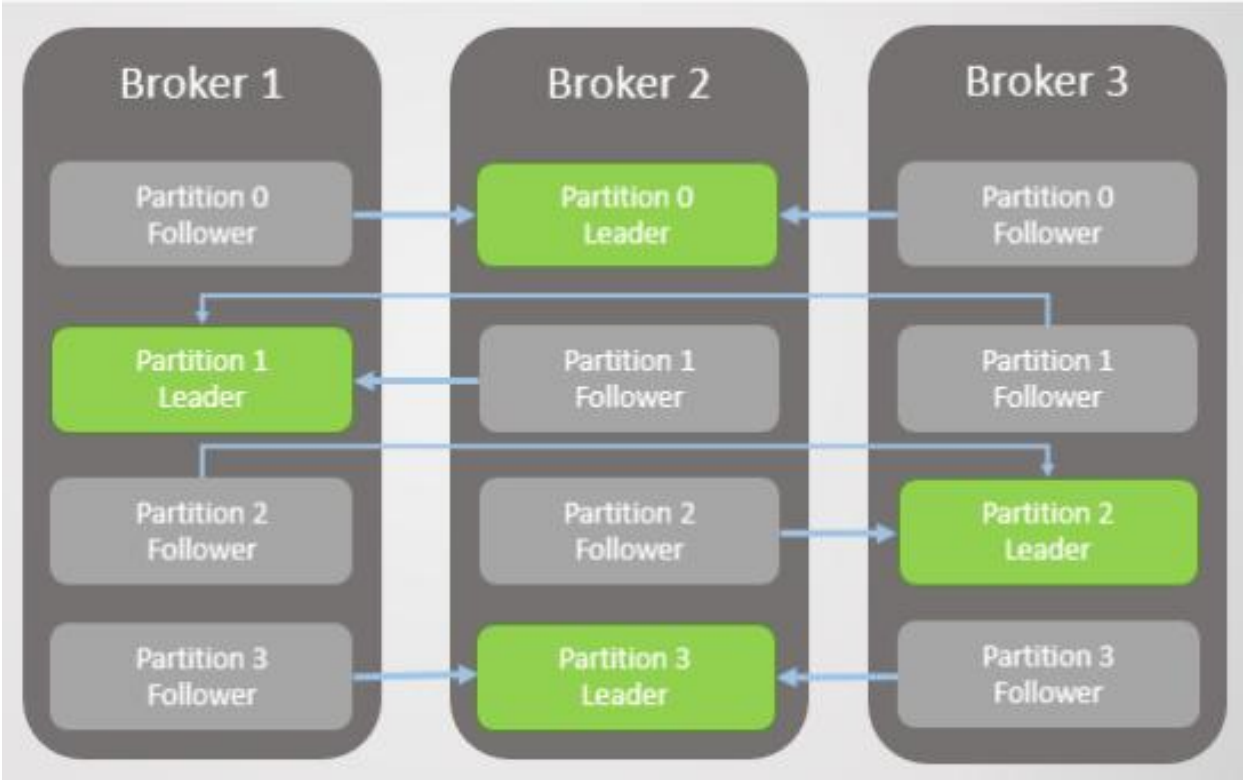
3. Kafka Consumer
This component subscribes to a topic(s), reads and processes messages from the topic(s).

4. Kafka Broker
Kafka Broker manages the storage of messages in the topic(s). If Kafka has more than one broker, that is what we call a Kafka cluster.

5. Kafka Zookeeper
To offer the brokers with metadata about the processes running in the system and to facilitate health checking and managing and coordinating, Kafka uses Kafka zookeeper.

Kafka Components

- Partitions for the same topic are distributed across multiple brokers in the cluster.
- Partitions are replicated across multiple servers; number of replicas is a configurable parameter.
- Each Partition has one server as a *leader* and a number of servers as *followers.*
- Each Server acts a leader for some of its partitions and as a follower for some other.
- The Producers are responsible for choosing which message to assign to which partition within the topic based on key assigned to message.



Kafka Use Cases

There are several use Cases of Kafka that show why we actually use Apache Kafka.

Messaging
For a more traditional message broker, Kafka works well as a replacement. We can say Kafka has better throughput, built-in partitioning, replication, and fault-tolerance which makes it a good solution for large-scale message processing applications.

Metrics
For operational monitoring data, Kafka finds the good application. It includes aggregating statistics from distributed applications to produce centralized feeds of operational data.

Event Sourcing
Since it supports very large stored log data, that means Kafka is an excellent backend for applications of event sourcing.

Kafka Features

- **High Throughput:** Provides Support for Hundreds of thousands of messages with modest hardware
- **Scalability:** Highly scalable distributed systems with no downtime
- **Data Loss:** Kafka ensures no data loss once configured properly
- **Stream Processing:** Kafka can be used along with real time streaming applications like Spark and Storm
- **Durability:** Provides support to persisting messages on disk
- **Replication:** Messages can be replicated across clusters, which supports multiple subscribers

APACHE kafka

RabbitMQ Vs Kafka

Let's see how they differ from one another:

i. Features

Apache Kafka– Basically, Kafka is distributed. Also, with guaranteed durability and availability, the data is shared and replicated.

RabbitMQ– It offers relatively less support for these features.

ii. Performance rate

Apache Kafka — Its performance rate is high to the tune of 100,000 messages/second.

RabbitMQ — Whereas, the performance rate of RabbitMQ is around 20,000 messages/second.

iii. Processing

Apache Kafka — It allows reliable log distributed processing. Also, stream processing semantics built into the Kafka Streams.

RabbitMQ — Here, the consumer is just FIFO based, reading from the HEAD and processing 1 by 1.

iv. Replay

When your application needs access to stream history, delivered in partitioned order at least once. Kafka is a durable message store and clients can get a "replay" of the event stream on demand, as opposed to more traditional message brokers where once a message has been delivered, it is removed from the queue.

Implementation of Kafka

Dependency uses:

```
<dependency>
        <groupId>org.springframework.kafka</groupId>
        <artifactId>spring-kafka</artifactId>
</dependency>
```

Implementation of Kafka

Define the KafkaSender class to send message to the kafka topic named as techacademyvisits-topic: - Producer

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Service;

@Service
public class KafkaSender {

        @Autowired
        private KafkaTemplate<String, String> kafkaTemplate;

        String kafkaTopic = "techacademyvisits-topic";

        public void send(String message) {

           kafkaTemplate.send(kafkaTopic, message);
        }
}
```

Implementation of Kafka

Define a Controller which will pass the message and trigger the send message to the Kafka Topic using the KafkaSender class.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import com.hcl.service.KafkaSender;

@RestController
@RequestMapping(value = "/techacademyvisits-kafka/")
public class ApacheKafkaWebController {

		@Autowired
		KafkaSender kafkaSender;

		@GetMapping(value = "/producer")
		public String producer(@RequestParam("message") String message) {
				kafkaSender.send(message);

				return "Message sent to the Kafka Topic techacademyvisits-topic Successfully";
		}

}
```

Implementation of Kafka

Finally Define the Spring Boot Class with @SpringBootApplication annotation - Producer

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringBootHelloWorldApplication {

        public static void main(String[] args) {

        SpringApplication.run(new Object[] {
                                SpringBootHelloWorldApplication.class
                                }, args);

       }
}
```

Implementation of Kafka

We are done with the required Java code. Now lets start Apache Kafka. As we had explained in detail in the Getting started with Apache Kafka perform the following.

Download the Apache Kafka from this link:

https://kafka.apache.org/downloads

Start Apache Zookeeper-

zookeeper-server-start.bat c:\shareData\development\appachekafka\kafka_2.12-2.0.0\config\zookeeper.properties

or
zkserver

Start Apache Kafka-

kafka-server-start.bat c:\shareData\development\appachekafka\kafka_2.12-2.0.0\config\server.properties

Implementation of Kafka

Next start the Spring Boot Application by running it as a Java Application.

CREATE TOPIC ON KAFKA SERVER

Also Start the consumer listening to the **techacademyvisits-topic**

kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic **techacademyvisits-topic**

PRODUCER

kafka-console-producer.bat --broker-list localhost:9092 --topic **techacademyvisits-topic**

-OR-

http://localhost:8080/techacademyvisits-kafka/producer?message="test"

CONSUMER

kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic **techacademyvisits-topic** --from-beginning

-OR-

IMPLEMENTATION OF CONSUMER

- **Step 1 - Consumer Spring boot app should have required kafka dependency.**
- **Step 2 - Define the application.properties**
  - spring.kafka.consumer.key-deserializer = org.apache.kafka.common.serialization.StringDeserializer
  - spring.output.ansi.enabled= always
  - spring.kafka.consumer.group-id=my-group
  - spring.kafka.consumer.auto-offset-reset=earliest

  - spring.kafka.consumer.value-deserializer=org.springframework.kafka.support.serializer.ErrorHandlingDeserializer
  - spring.kafka.consumer.properties.spring.deserializer.value.delegate.class=org.springframework.kafka.support.serializer.JsonDeserializer

  - spring.kafka.consumer.bootstrap-servers=localhost:9092

Step 3 - IMPLEMENTATION OF CONSUMER - Define the KafkaConfig class


```
import org.apache.kafka.clients.admin.NewTopic;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.annotation.EnableKafka;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.config.TopicBuilder;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;
import org.springframework.kafka.support.serializer.JsonDeserializer;
@EnableKafka
@Configuration

//Class
public class KafkaConfig {
```

Step 3 - IMPLEMENTATION OF CONSUMER - Define the KafkaConfig class - Cont...

```
@Bean
	public ConsumerFactory<String, Order> consumerFactory() {
		// Creating a map of string-object type
		Map<String, Object> config = new HashMap<>();
		// Adding the Configuration
		config.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
		config.put(ConsumerConfig.GROUP_ID_CONFIG, "my-group");
		config.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
		config.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);
		// Returning message in JSON format
		return new DefaultKafkaConsumerFactory<>(config, new StringDeserializer(), new
JsonDeserializer<>(Order.class));
	}
```

Step 3 - IMPLEMENTATION OF CONSUMER - Define the KafkaConfig class - Cont...


```
// Creating a Listener
        @Bean
        public ConcurrentKafkaListenerContainerFactory<String, Order> orderListener() {
                ConcurrentKafkaListenerContainerFactory<String, Order> factory = new
ConcurrentKafkaListenerContainerFactory<>();
                factory.setConsumerFactory(consumerFactory());

                return factory;
        }
}
```

Step 4 - IMPLEMENTATION OF CONSUMER - Define the ConsumerService


```
import java.util.Optional;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Component;
@Component
public class ConsumerService{
@Autowired
private KafkaTemplate<Long, Order> template;
@Autowired
private ProductRepository productRepository;
        @KafkaListener(topics = "orders-batch5", groupId = "my-group", containerFactory = orderListener")
        // Method
        public void consume(Order order) {
                // Print statement
                System.out.println("message = " + order);
}
}
```

Questions