Apache Kafka is an open-source distributed event streaming platform that facilitates real-time data processing. It enables applications to send and receive messages seamlessly, acting as a middleman to prevent data loss. Kafka's architecture supports a publish-subscribe model, allowing multiple applications to communicate efficiently while reducing connection complexities.

##### You may be interested in these questions:

- [What are the key features of Apache Kafka?](#related)

- [How does Kafka handle data loss?](#related)

- [What is the publish-subscribe model in Kafka?](#related)

Highlights:

00:10 Kafka is an open-source distributed event streaming platform that facilitates real-time data processing. It allows for the continuous generation and handling of data streams effectively.

-Understanding event streaming involves creating and processing real-time data streams. This is essential for applications that require instant data handling and analysis.

-Using Paytm as an example illustrates how Kafka handles numerous transactions simultaneously. It showcases the capacity of Kafka to manage vast amounts of events efficiently.

-Kafka's architecture is designed to support high-volume data streams. This makes it suitable for various applications across different industries that require real-time data processing.

02:22 Real-time event streaming involves continuously processing messages from a Kafka server to handle specific tasks like transaction limits for users. This system ensures timely notifications and effective data management.

-The client application monitors transaction counts for users, ensuring they don't exceed the limit of 10 transactions per day. This helps maintain operational integrity.

-Continuous listening and processing of Kafka messages allows for real-time data validation and user notifications. This capability is crucial for maintaining user engagement and satisfaction.

-Kafka serves as a distributed event streaming platform, facilitating load balancing across multiple servers. This distribution minimizes downtime and enhances service reliability.

04:41 Kafka serves as a middleman that ensures data delivery between applications without loss, even when one end is unavailable. This mechanism prevents data loss and enhances communication efficiency.

-Kafka was developed at LinkedIn and open-sourced in 2011, becoming part of the Apache Software Foundation. It is designed to handle real-time data feeds efficiently.

-The parcel delivery analogy illustrates how Kafka prevents data loss, similar to how a letterbox safely holds parcels when the recipient is unavailable. This ensures information is retained.

-By acting as a buffer, Kafka allows applications to communicate seamlessly. This is particularly important in scenarios where applications may be temporarily offline or busy.

07:14 Kafka serves as a messaging system that ensures data is not lost between applications, especially when one application is temporarily unavailable. It acts like a letterbox, allowing messages to be stored and retrieved later.

-The need for a messaging system arises when applications must communicate reliably without losing data during downtime. Kafka fulfills this role effectively by storing messages until they can be retrieved.

-In scenarios with multiple applications, managing numerous connections can become complex. Kafka simplifies communication and helps in handling diverse data formats from different applications smoothly.

-As applications grow, challenges such as varying data formats and connection types can emerge. Kafka provides a robust solution to mitigate these issues and maintain efficient data flow.

09:15 Managing complex connections between multiple services in enterprise applications can be challenging and may lead to bottlenecks. Implementing a messaging system like Kafka helps simplify communication and data handling.

-Different types of connections, such as HTTP, TCP, and JDVC, complicate data management and integration among various services. Each connection type requires unique handling strategies.

-The total number of connections across multiple applications can reach significant numbers, leading to increased complexity in maintaining and managing them effectively.

-A centralized messaging system like Kafka allows applications to communicate more efficiently by managing data payloads and schemas, reducing the overall connection complexity.

11:34 The video explains the advantages of using Kafka as a messaging system, highlighting its efficiency in reducing connection counts in data processing. It emphasizes the Pub/Sub model as a core component of Kafka's functionality.

-Kafka centralizes message handling, decreasing the number of connections significantly compared to previous methods, which optimized resource usage and improved performance.

-The Pub/Sub model in Kafka involves three main components: publisher, subscriber, and message broker, facilitating organized message distribution and retrieval.

-Understanding how Kafka processes messages within its architecture is crucial, with future sessions planned to delve deeper into its components and operational mechanics.

**Kafka Architecture and Components**

In this tutorial, we will discuss Kafka architecture and its components. By understanding Kafka terminology and its purpose, you can visualize how a message flows internally from the producer to the consumer through a broker or Kafka server.

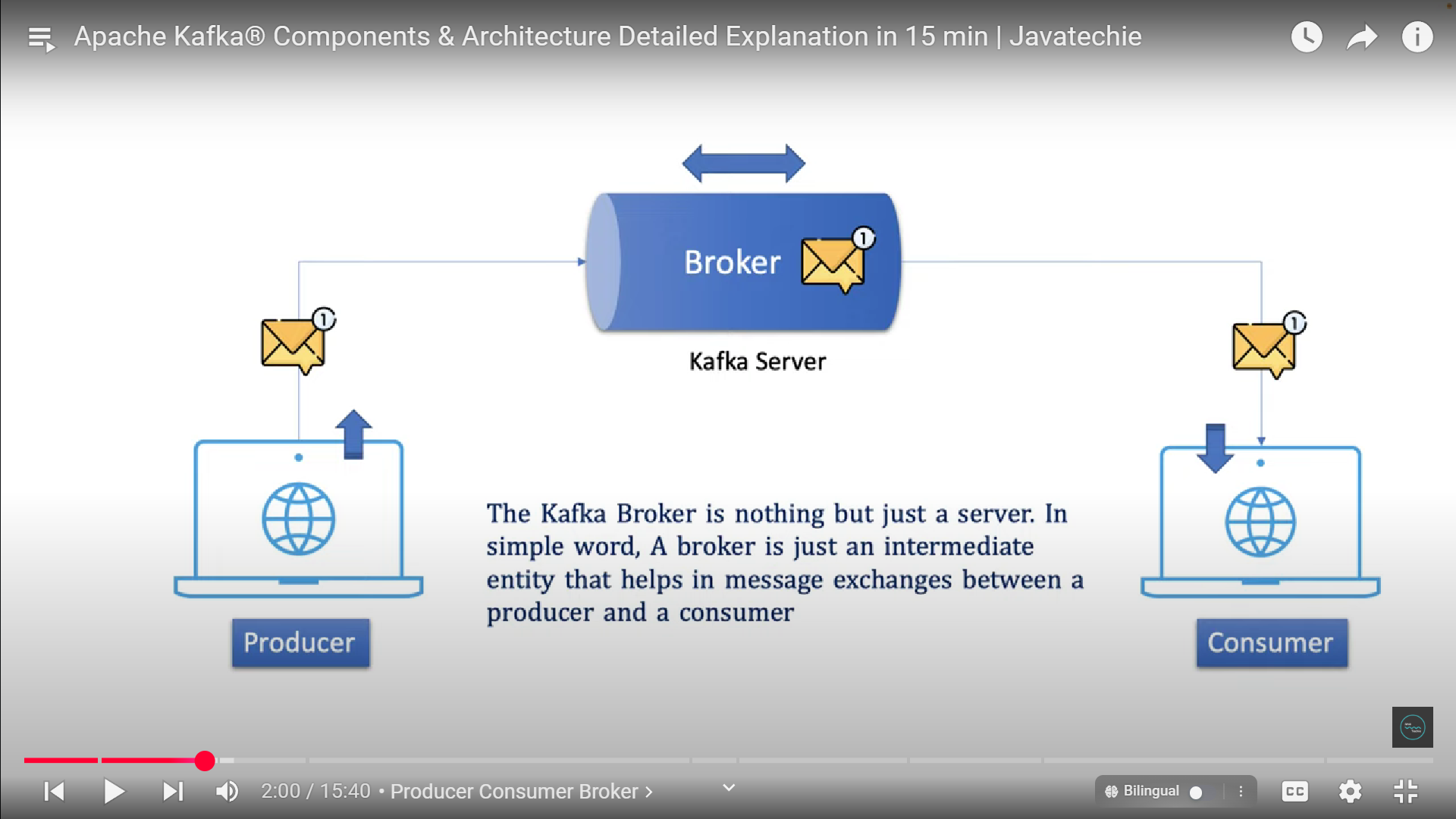
**Components of Kafka Ecosystem**

**Producer**

* The producer is the source of data that publishes messages or events.
* In a pub-sub model, the producer posts messages to the broker, which are later consumed by the consumer.
* Producers and consumers do not directly communicate. A Kafka broker acts as the middleman, facilitating message exchange.

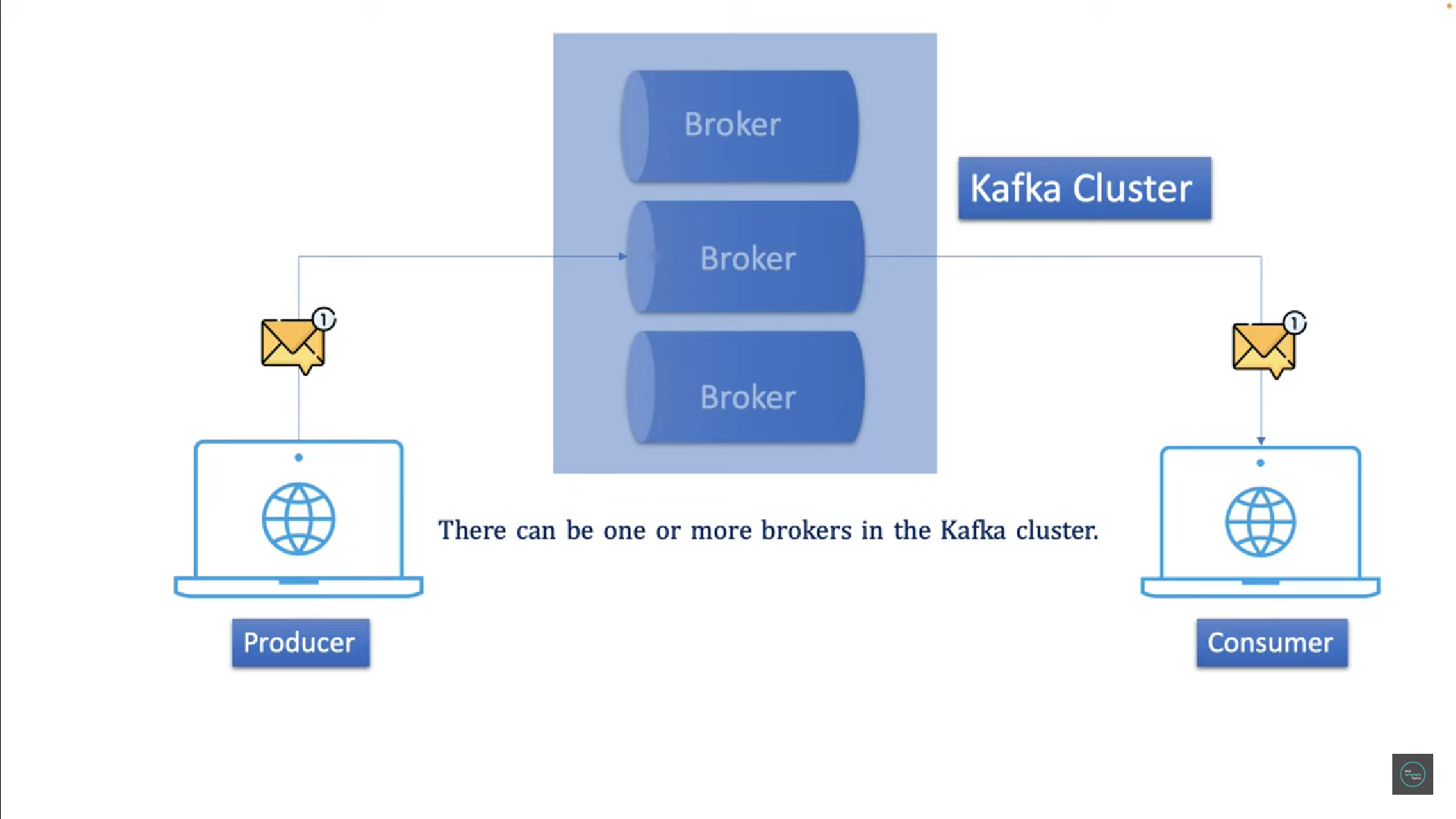
**Broker**

* A broker stores messages from producers and makes them available to consumers.
* It serves as an intermediate entity that helps in message exchange.



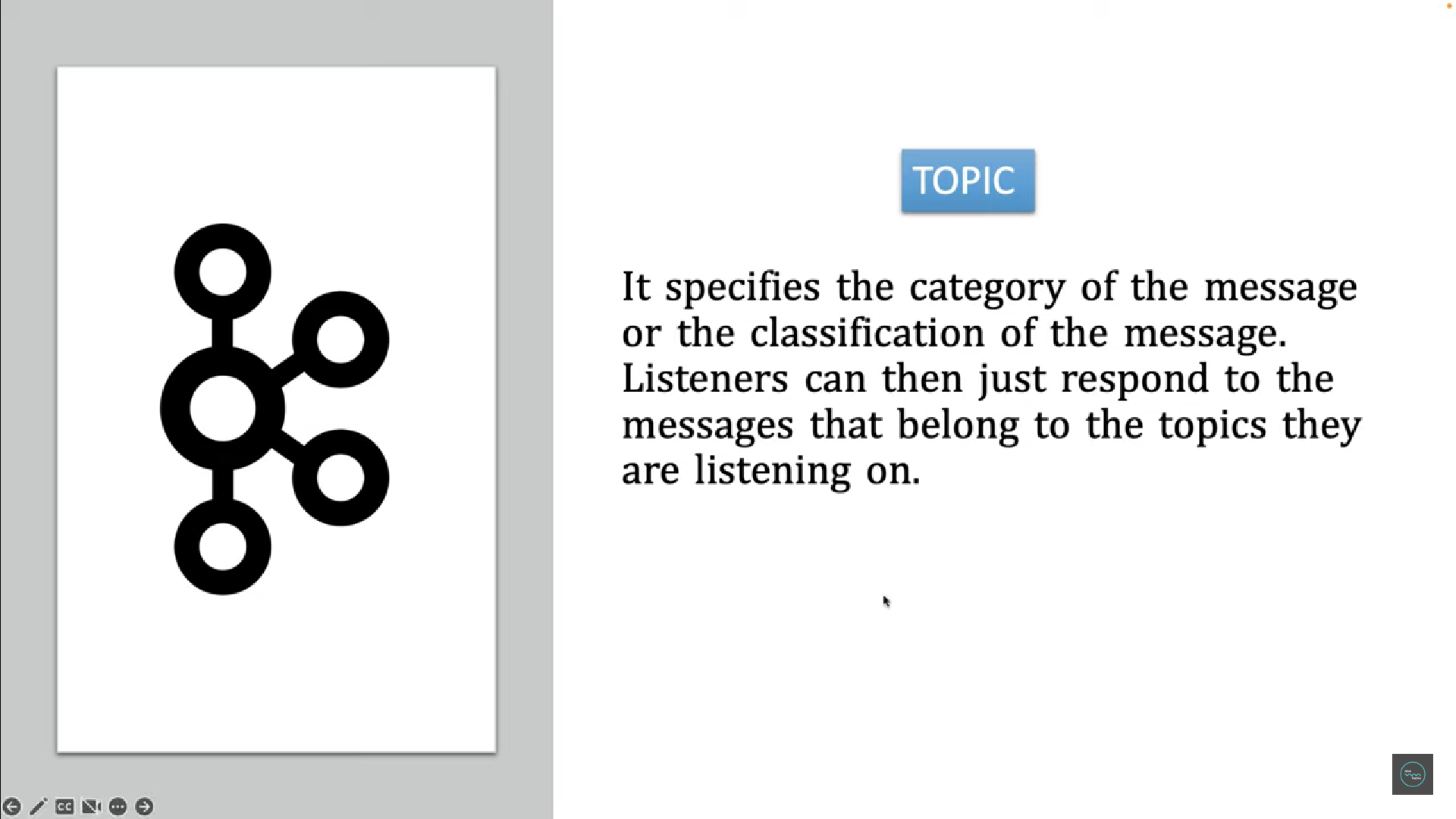
**Cluster**

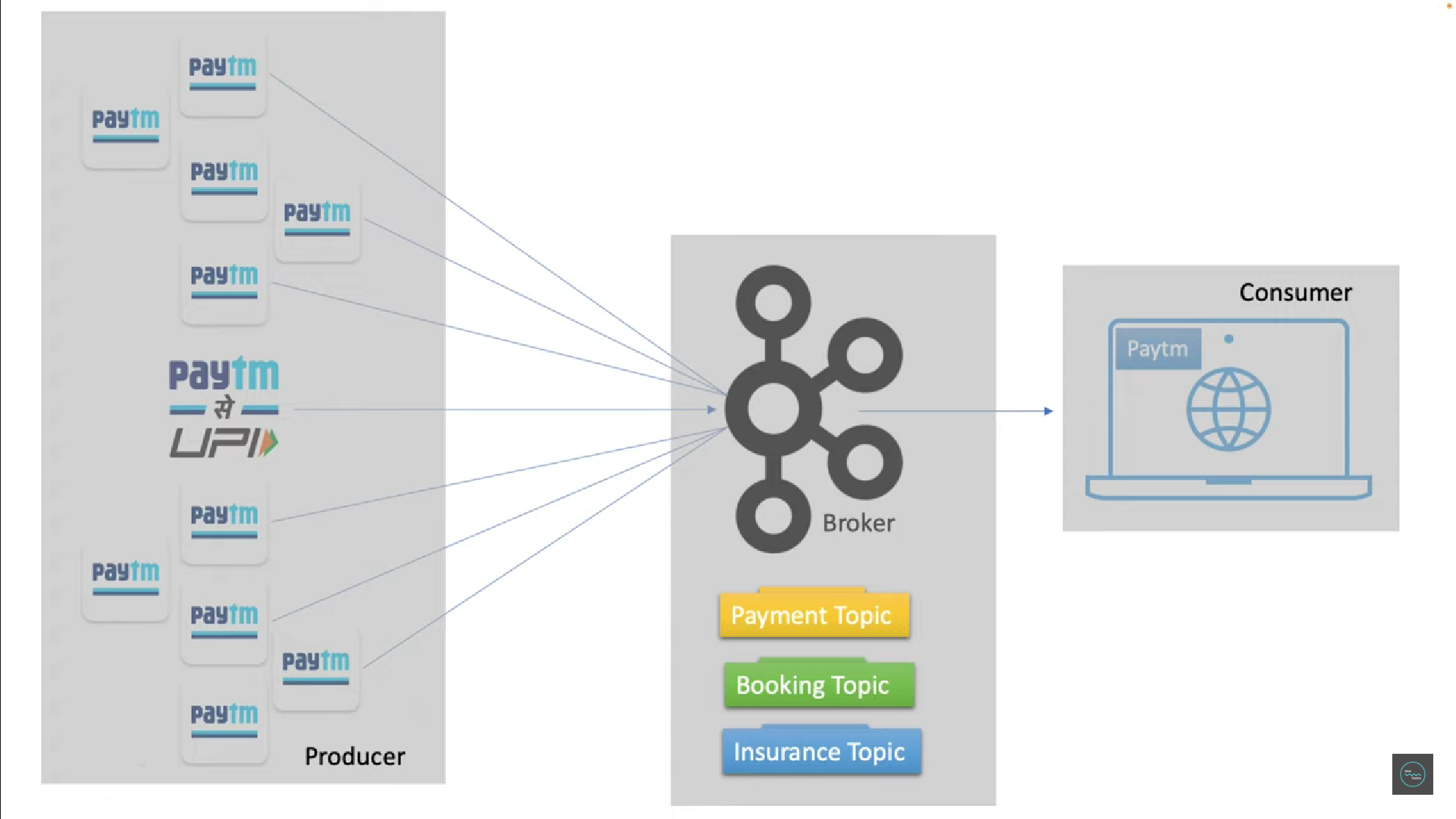
* A cluster is a group of computers or servers working together for a common purpose.
* Kafka, being a distributed system, can have multiple brokers within a single Kafka cluster.
* If the producer publishes a high volume of data, multiple brokers within a cluster can handle the load.



**Topic**

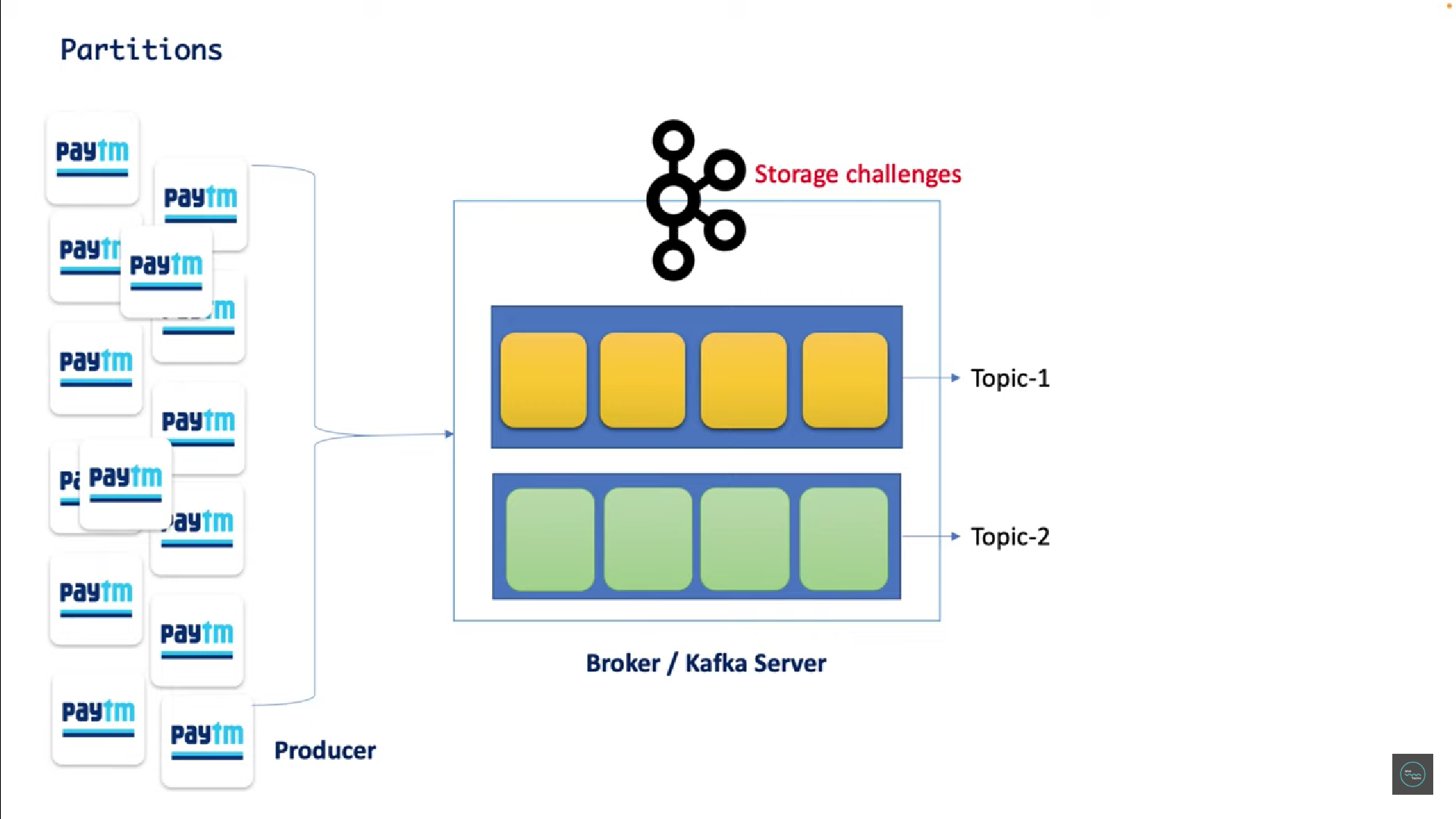
* A topic categorizes and stores messages of a similar type.
* Producers can post specific messages to corresponding topics (e.g., payment-related messages to a payment topic).
* Consumers can subscribe to specific topics and consume only relevant messages, reducing unnecessary communication with the broker.
* Topics in Kafka function similarly to tables in a database, where related data is stored together.





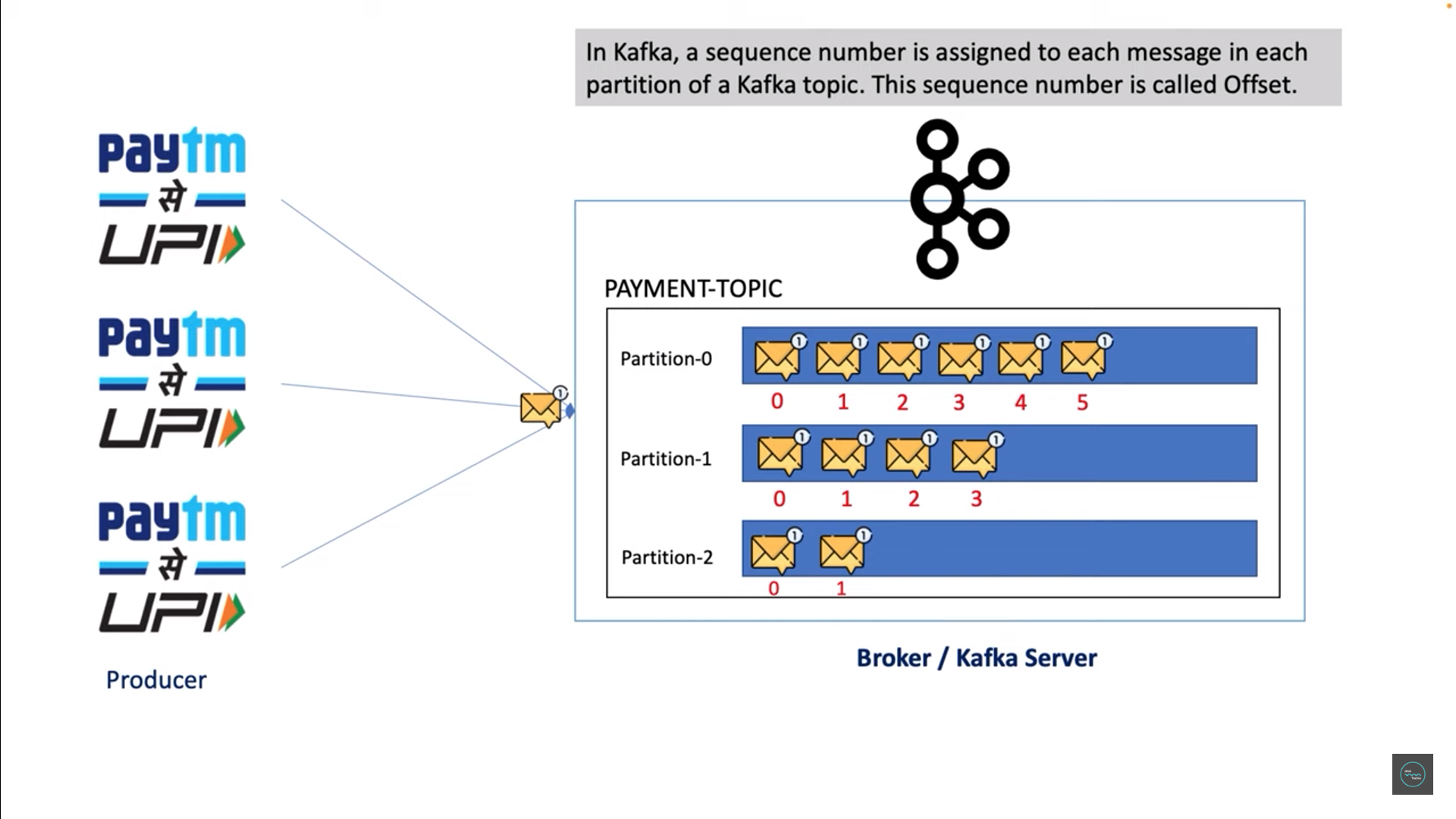
**Partition**

* A Kafka topic can be divided into multiple partitions to handle large volumes of data.
* Partitioning improves performance and availability by distributing messages across multiple machines.
* Each partition can accept messages concurrently, ensuring better throughput and load distribution.
* If a partition becomes unavailable, other partitions can handle the load without downtime.



**Offset**

* An offset is a unique sequence number assigned to each message within a partition.
* It helps track which messages have been consumed.
* If a consumer goes offline, it can resume reading messages from the last offset.



**Consumer Group**

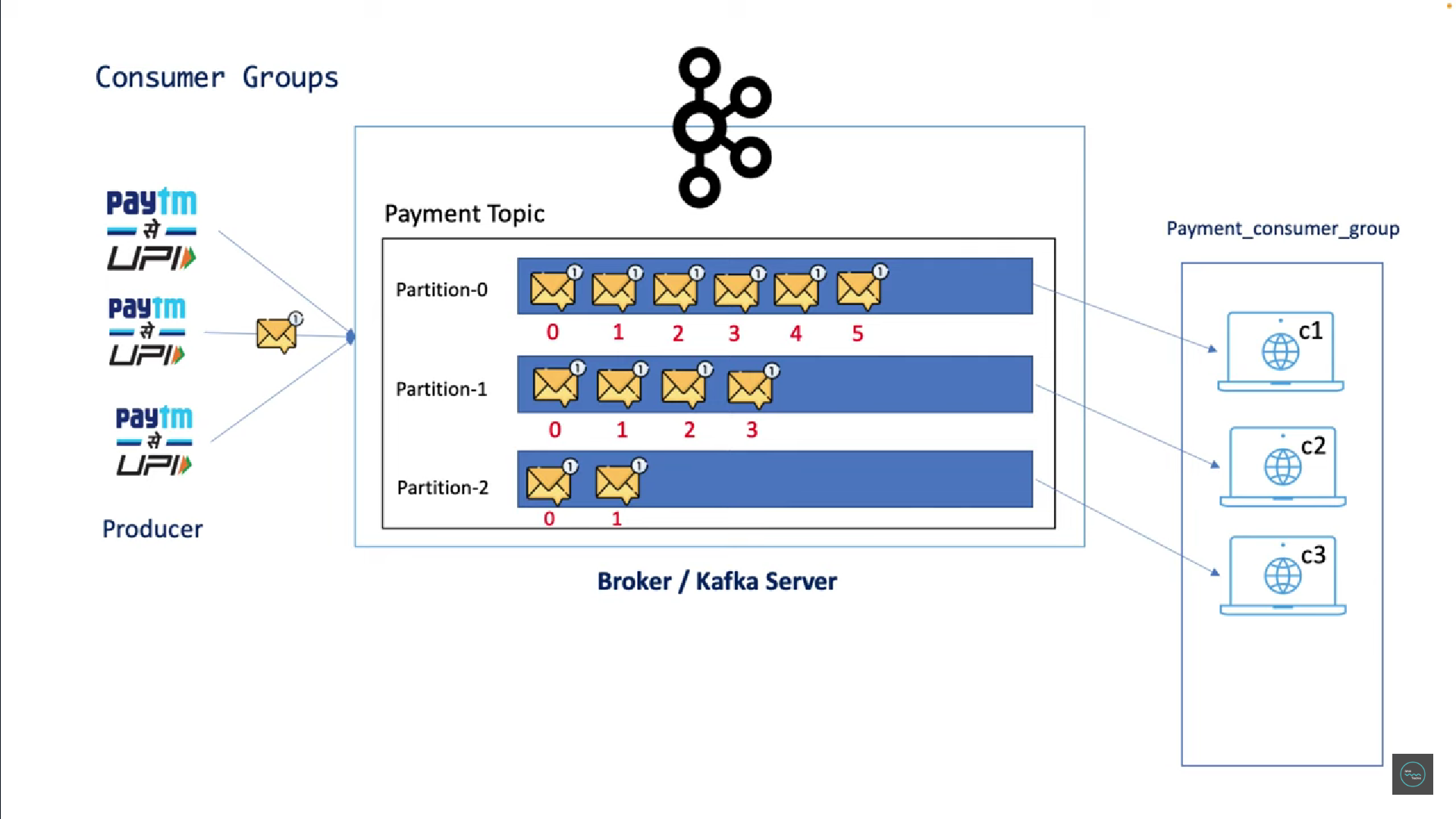
A single consumer reading from multiple partitions may lead to performance issues due to a lack of concurrency. To address this, Kafka introduces **consumer groups**.

A consumer group consists of multiple consumer instances, each assigned to one or more partitions. This enables parallel message consumption, improving throughput and performance. For example:

* If there are three partitions and three consumer instances in a group, each consumer can read from one partition.
* If a new consumer is added to the group, it remains idle unless a partition becomes available.

In case of consumer failure, another consumer in the group can take over the partition, a concept known as **consumer rebalancing**.

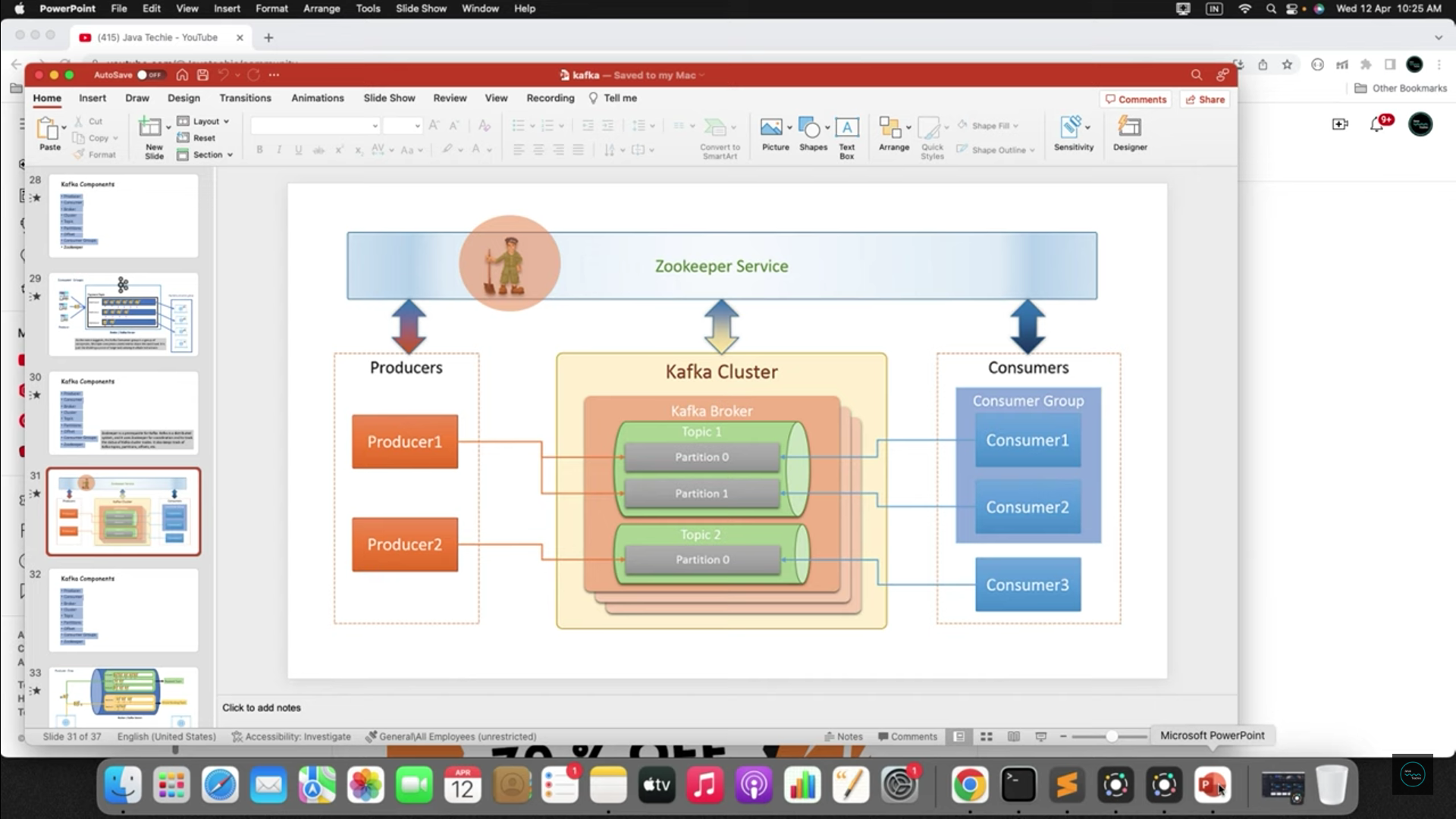
you can see here a single consumer is playing with all the three partition so definitely you don't give us the better throughput right then how we can overcome this situation it's very simple just share the workload how can I share the workload so what I will do simply I will Define n number of consumer instance now I can group all the three consumer what I Define into a single unit by specifying the group name that is called consumer group now payment consumer group have the three consumer instance now since I have multiple consumer with me I can divide the workload to each and every consumer to achieve better throughput so simply now my consumer one can read from partition 0 consumer 2 can read all the messages from partition one and consumer 3 can read from the partition two then in this approach all the three consumer instance read parallely from each and every partition which will definitely give me the better performance or we can get the better response right just keep a note we can't guarantee about consumer and partition order any consumer can talk to any partition that will be decided by the coordinator okay so you can say consumer one might listen from partition 2 or any order okay we don't have any control on it



cool now you might have a question hey I have three partition so three consumer so they will talk to each other that's correct but what if I have a fourth consumer added to my consumer group then what will be the behavior of this consumer 4 what you will do right there is no change that consumer 4 will sit ideal because there is no work for him now since all the three partitions assigned to three different consumer instance there is no partition left for him so he will simply sit ideal but in case if any consumer instance is rejected or goes off then C4 will get a chance to connect with any partition this concept is called consumer rebalancing

**Zookeeper**

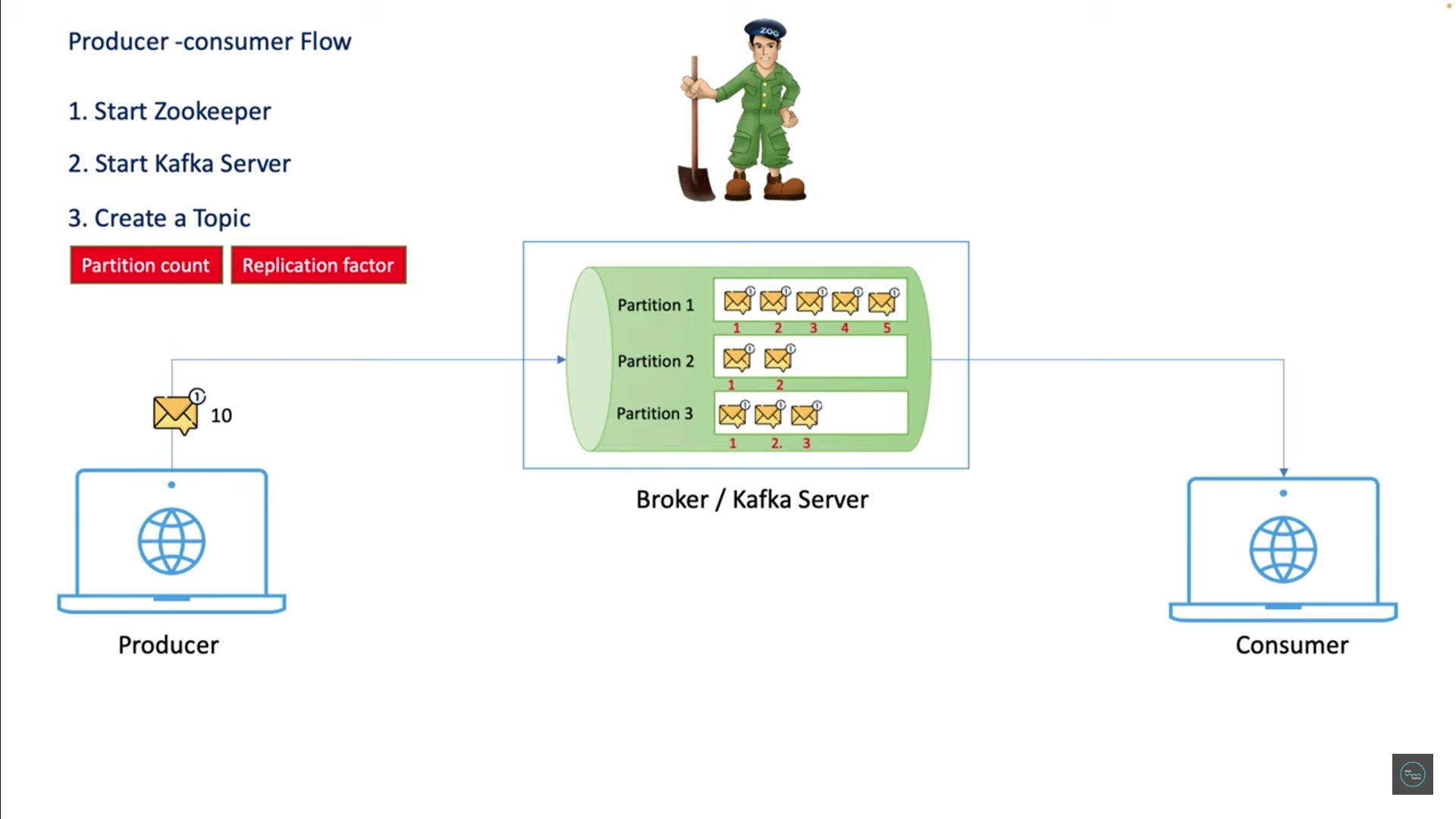
* Zookeeper is used for coordination and tracking the status of the Kafka cluster.
* It keeps track of topics, partitions, offsets, and other components of the Kafka ecosystem.
* Zookeeper acts as a manager for brokers and the cluster.



can we run Apache Kappa without zookeeper ? yes this could be possible after the Kafka 2.8 version onwards we can run Apache Kappa without using zookeeper

so before we start let me give you some hints about the role of Zookeeper and advantages of using Kafka without Zookeeper okay usually Kafka uses zookeeper to store and manage all the metadata information about the Kafka cluster Kafka also used the Zookeeper as a centralized controller that manages all the Kafka Brokers or servers however in the new Kafka version instead of storing all the server config information in Zookeeper you can store them as a topic partition inside the Kafka server itself okay so that is what the advantages of using Kafka without zookeeper so you no need to keep all the Kafka clustered metadata inside the Zookeeper rather you can store them in a cup cup topic partition itself in the Kafka server

but to start with the Kafka without zookeeper you should run Kafka with Kraft mode or Kraft metadata mode which is also called Kraft



#TO start zookeeper

C:\kafka\bin\windows\zookeeper-server-start.bat C:\kafka\config\zookeeper.properties

#to start kafka server

C:\kafka\bin\windows\kafka-server-start.bat C:\kafka\config\server.properties

#Create topic test

kafka-topics.bat --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic test

#running topic logs publisher

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

#Running logs topic consumer

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

