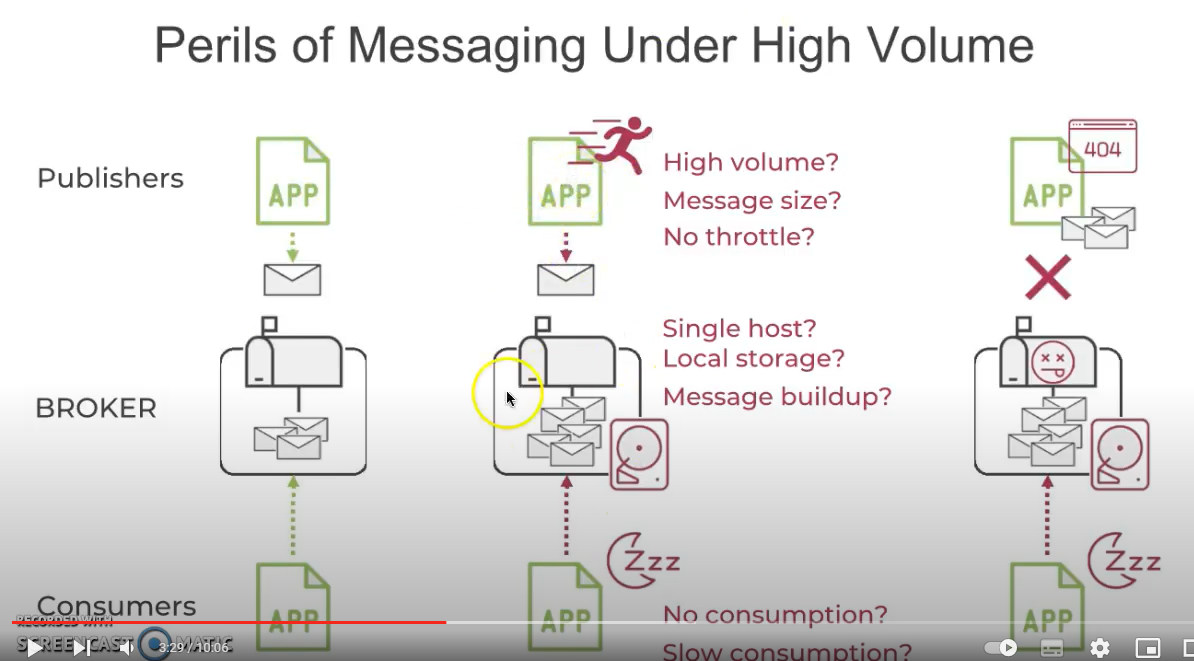


Kafka là gì:

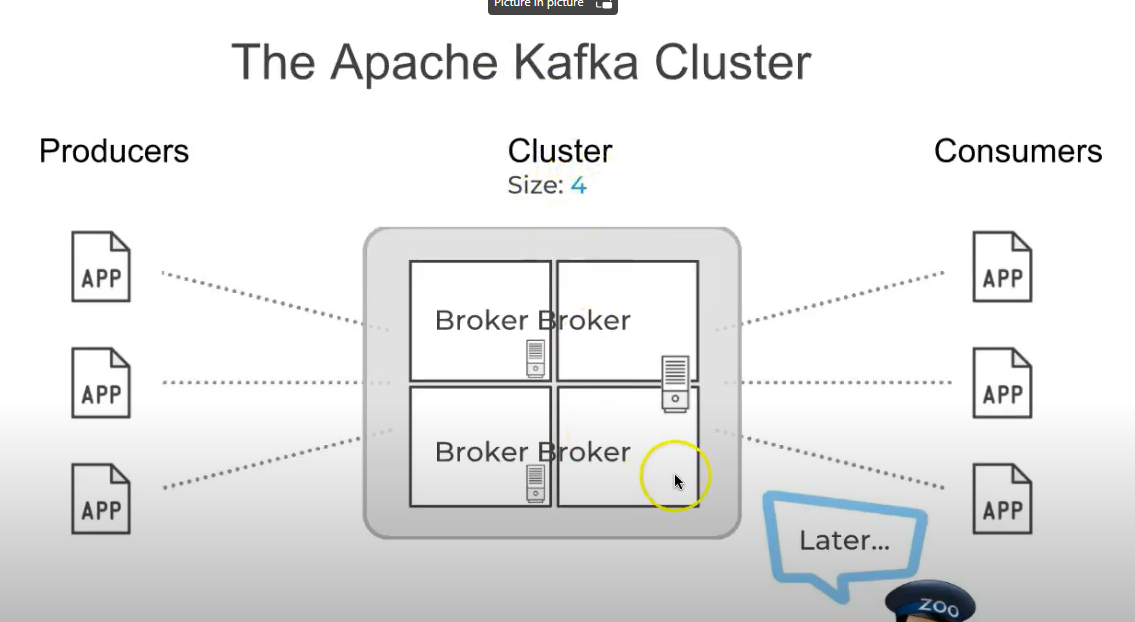


Publishers: người gửi

Brooker: hòm thư

Consumers: người nhận

* TH1: Nếu hệ thống lớn, hòm thư quá tải dẫn tới ng nhận ko thể nhận được thư hoặc hòm thư ko thể tiếp nhận quá trình gửi thư nữa
* TH2: Nếu trong quá trình hòm thư xử lý lỗi, n cần nhận lại thư nhưng thư đó đã bị mất, ko nhận lại dk



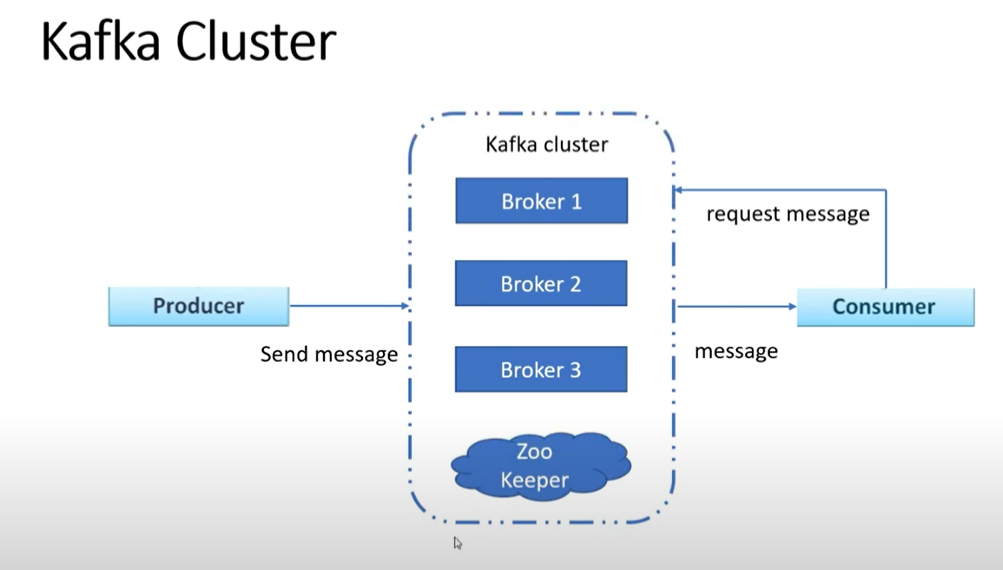
Cụm cluster là apache-kafka cluster

Broker

Apache Zookeeper là nên tảng apache kafka chạy trên đó

# Kafka Components

1. Producer
2. Consumer
3. Broker
4. Cluster
5. Topic



Kafka cluster như 1 văn phòng môi giới để chuyển dữ liệu từ ng gửi đến ng nhận

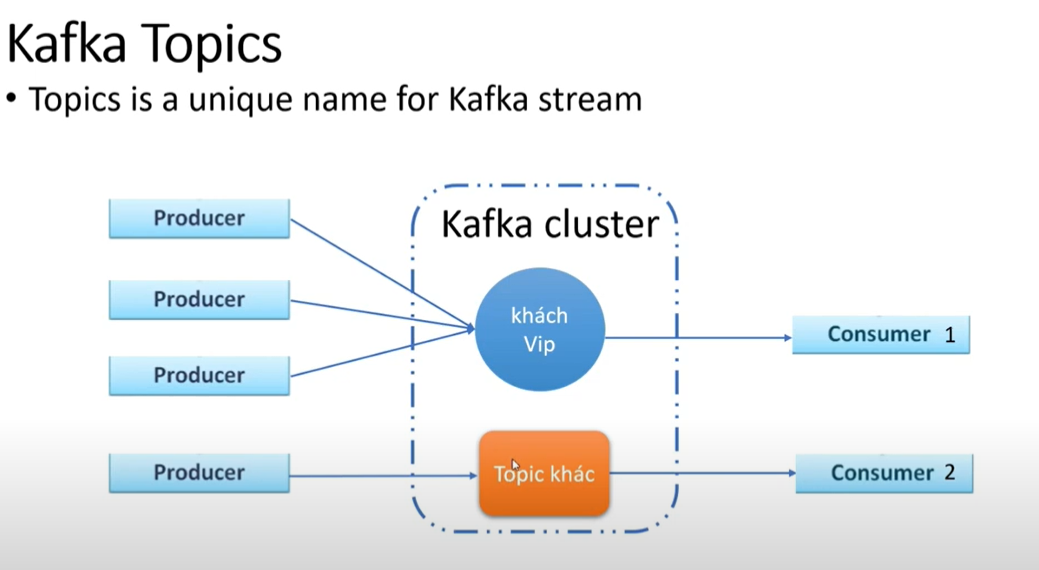
Kafka cluster : văn phòng môi giới

Zoo Keeper: Sếp

Trong 3 broker chỉ có 1 broker hoạt động (gửi nhận từ producer - consumer)

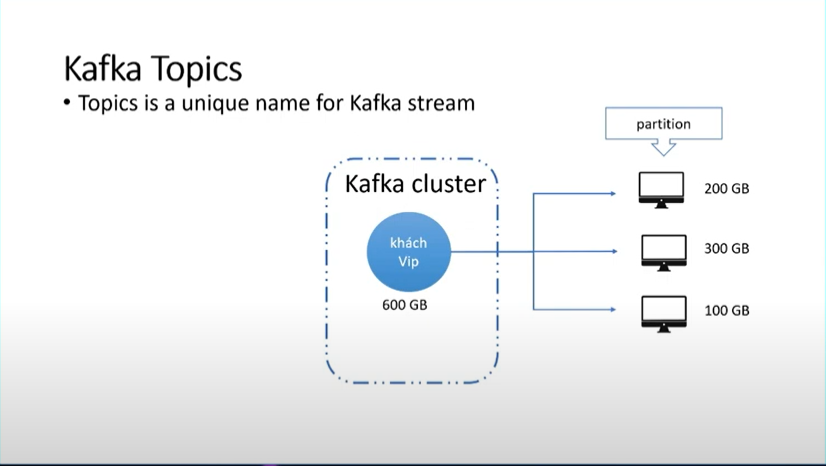
Broker 2 3, sẽ có bản copy để khi broker 1 bị hư thì broker 2, 3 sẽ thay thế

* Vấn đề đặt ra là broker 2 hay broker 3 sẽ thay thế
* Vấn đề này sẽ do thằng Zoo Keeper quyết định

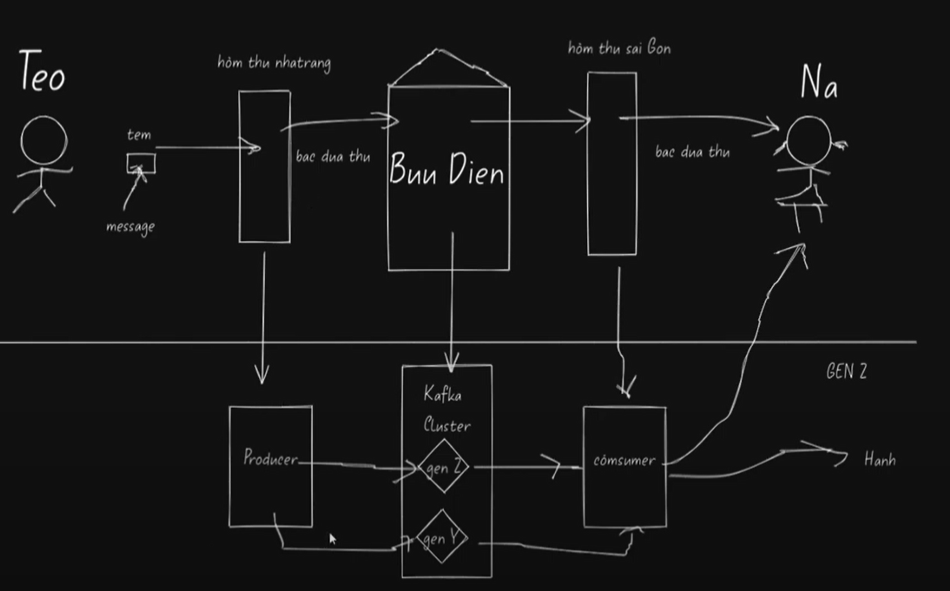


Chia ra các topic để kafka xử lý (nó có thể xét tính năng ưu tiên cho các topic)

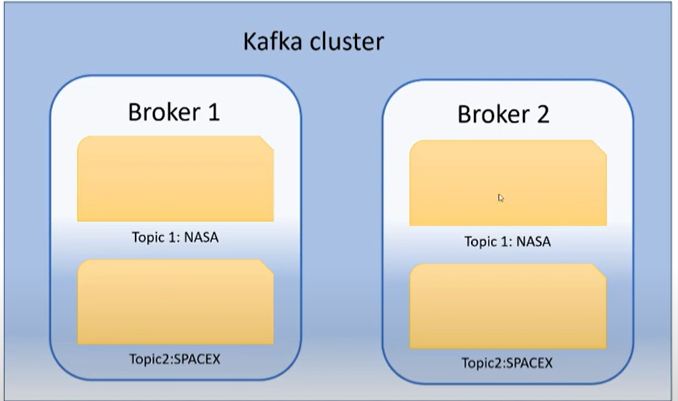
* Producer gửi thông tin cho kafka cluster
* Consumer sẽ nhận thông tin từ kafka cluster
* Trong kafka cluster sẽ có nhiều topic: khách víp, genZ, gen Y



* Partition: nghĩa là khi 1 máy quá nhiều thông tin mà n ko lưu trữ được hết trên 1 máy thì n sẽ chia làm nhiều máy khác nhau gọi là 1 partition



Gen Z, Y là các topic



1 kafka cluster có nhiều broker

1 broker có nhiều topic

Trong topic có nhiều partition

Producer: có nhiều producer

Consumer: có consumer group

# Install kafka

B1: tạo kafka Zoo Keeper vs Broker

Start the ZooKeeper service

bin/zookeeper-server-start.sh config/zookeeper.properties

Start the Kafka broker service

bin/kafka-server-start.sh config/server.properties

* tạo topic

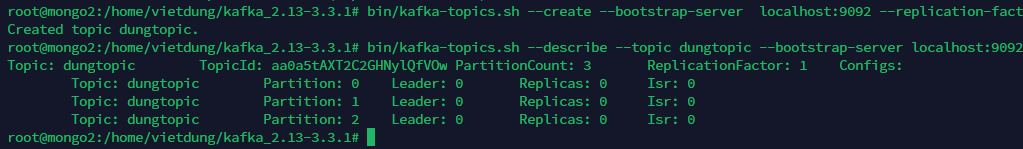
bin/kafka-topics.sh --create --zookeeper localhost:9092 --replication-factor 1 --partitions 3 --topic dungtopic

* lệnh lỗi, đổi thành

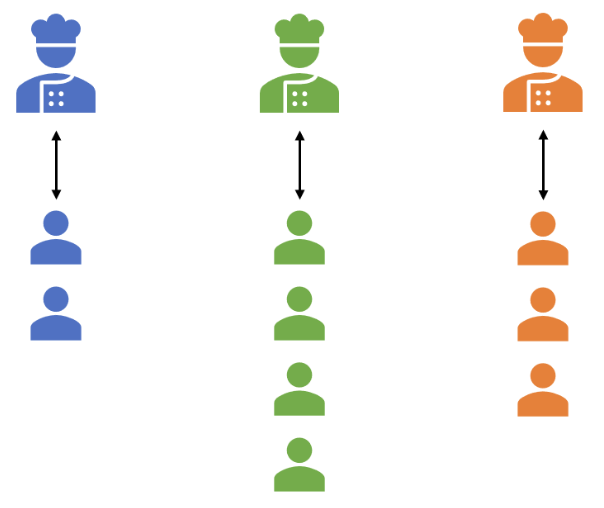
bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 3 --topic dungtopic

* kiểm tra topic phải có 3 partition ko

bin/kafka-topics.sh --describe --topic dungtopic --bootstrap-server localhost:9092

 oke

# Giải thích ý nghĩa của kafka



Ví dụ khi khách hàng order món ăn (client), và nhân viên là (server)

Thì mỗi client khi order phải đợi nhau

Giải pháp là gì: tăng nhân viên => ko hay, ko ổn

**Vấn đề đang là mỗi người phải xếp hàng đợi nhau, server tèo thì lại phải đợi, chờ server phục hồi**

* sinh ra giải pháp order, gọi món. Khách hàng ko phải chờ đợi việc order, vẫn đi chơi. Còn lại để bọn nhân viên xử lý
* giải pháp là **tách biệt request và response**.

Message-drivenprogramming bắt nguồn từ tư tưởng trên. Server và client không giao tiếp trực tiếp với nhau nữa.(tránh việc client –server: nhân viên và khách hàng đứng tán nhau làm nghẽn server hoặc server chết). Tất cả các request sẽ được gửi dưới dạng message cho bên thứ 3. Bên thứ 3 có nhiệm vụ điều hướng các message đến địa chỉ cụ thể với 2 mục tiêu:

## Message broker

Thường 1 broker sẽ là 1 server,

* Giảm tải cho các **server** bằng việc giảm các tương tác trực tiếp.
* Lưu trữ request, trong trường hợp **server** gặp sự cố.
* Phân phối request đến các nhiều **server** trong các bài toán cụ thể.
* Đơn giản hóa quá trình gửi nhận message trong môi trường multi-services.

Kafka broker này sẽ tương tác với multi service khác như SQL, REDIS qua các plugin

Hiện nay, có khá nhiều Message broker hoạt động dựa trên cách thức và nền tảng khác nhau, nhưng tựu chung lại đều chung mục đích điều hướng, trung chuyển message:

* Apache ActiveMQ.
* Apache RocketMQ.
* RabbitMQ.
* Apache Kafka.
* IronMQ.
* ZeroMQ.
* Redis, thực tế hiếm khi sử dụng. Chẳng ai đi **KIA Morning** trên cao tốc trong khi đã có **Lamborghini**.

## 3) Message distribution patterns

Tương tự với ví dụ cửa hàng KFC, Message broker cung cấp 2 patterns chính để cung cấp việc điều hướng message:

* **Point-to-point messaging**: hay còn gọi là **Queue**. Hiểu đơn giản đó là dạng phân phối message có quan hệ 1 - 1 giữa **client** và **server**, tao chỉ nói cho.. một mình mày thôi đấy. Mỗi message chỉ được gửi đến một endpoint duy nhất. Ví dụ là cuộc trò chuyện trên Skype giữa 2 người với nhau.
* **Broadcast messaging**: một message có thể được gửi tới nhiều địa chỉ khác nhau, chỉ những người subcribe nội dung đó mới nhận được message. Ví dụ như khi follow mình, hệ thống chỉ gửi thông báo khi mình có bài viết mới đến các followers. Pattern này được gọi là **Topic**.

Với từng bài toán khác nhau ta sẽ linh hoạt sử dụng **queue** hoặc **topic** để xử lý vấn đề.

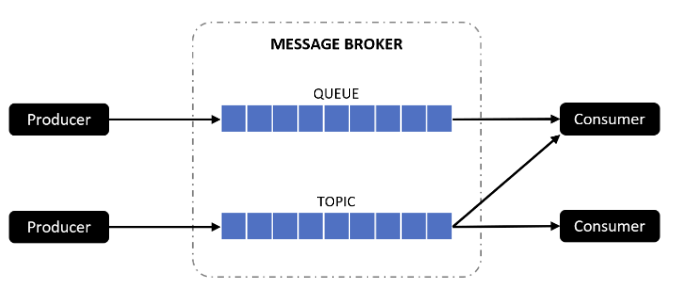
* Nhắn tin hai người có thể dùng **queue**.
* Khi nhắn tin trong group thì dùng **topic**.

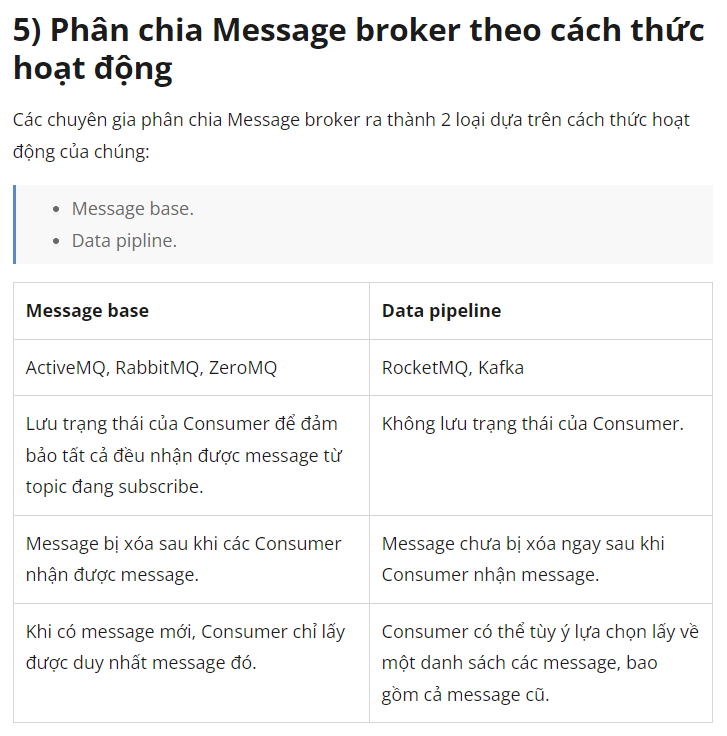
## Mô hình sử dụng Message broker

Message broker được đặt trong hệ thống bao gồm:

* **Producer/Publisher**: nơi gửi message.
* **Message broker**: hệ thống điều hướng message.
* **Consumer/Subcriber**: nơi nhận message.

Không còn khái niệm **client** và **server** mà thay vào đó là **producer/publisher** và **consumer/subscriber**. Về bản chất vẫn như nhau, một bên gửi và một bên nhận message.





2 cơ chế lưu thông tin như của facebook và whatsapp

1 cái chỉ lưu 1 lần, 1 cái có thể trace lại dk

## Apache Kafka

Không cần trình bày nhiều, các bạn có thể google với trả về 8 triệu kết quả trong 0.5s. Mình chỉ điểm qua 4 tính chất được quảng cáo của Kafka như sau:

* **High scalable**: Kafka là hệ thống phân tán - distributed system, có khả năng mở rộng rất nhanh và dễ dàng với **zero downtime** - mọi thứ vẫn hoạt động bình thường khi thêm hoặc bớt **broker**.
* **High durable**: message được lưu trên disk, đảm bảo nếu mất điện.. data vẫn còn nguyên. Ngoài ra, một message sẽ có nhiều bản sao lưu trên nhiều **broker** khác nhau, phụ thuộc vào config và set up. Nếu một **broker** die, flow vẫn hoạt động bình thường không bị ngắt quãng.
* **High reliable**: giống **durable**, lưu trữ message ở nhiều nơi. Ngoài ra có cơ chế cân bằng request trong trường hợp gặp sự cố về các **broker**. Đại khái là đáng tin cậy hơn các **message broker** hiện có trên thị trường.
* **High performance**: high throughput cho cả đầu gửi và nhận message với khả năng **scale** tuyệt vời. Nhờ vậy nó có thể xử lý hàng TB data mà không gặp nhiều vấn đề về performance.

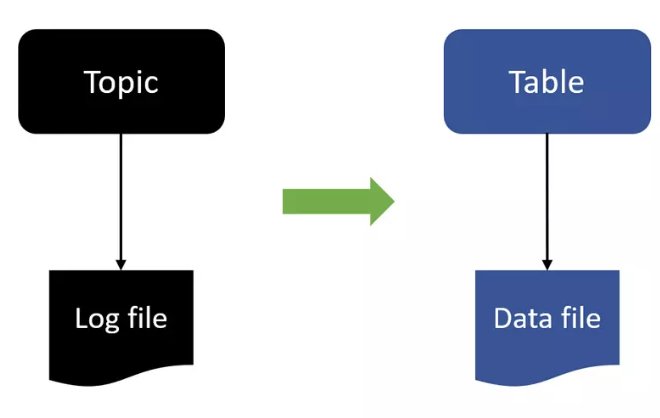
# Kafka topic, partition và offset

**Queue** và **Topic** là đại diện cho hai concept gửi nhận message từ một nguồn đến một/nhiều nguồn.

**Topic** là **stream of data**, luồng lưu trữ dữ liệu của **Kafka**. Nói thế cho nguy hiểm chứ hiểu đơn giản nó là một dãy các message nối tiếp nhau.

* Có thể coi **topic** giống như **table** trong relational databse: OracleBD, PostgreSQL... whatever.
* Table bao gồm name và row. Tương tự, sẽ có **topic name** và **message**. Các luồng dữ liệu được gửi vào **topic** giống như việc insert row vào table. Row mới được insert vào ngay sau row cũ.

Message nằm trong 1 topic

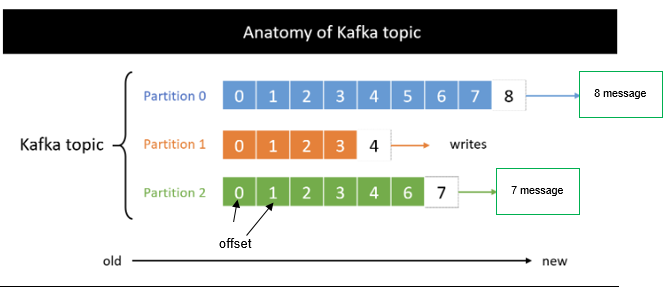


### Partition và offset

Chắc hẳn chúng ta còn nhớ đến kĩ thuật [**partition table**](https://viblo.asia/p/006-partitioning-data-voi-postgresql-p1-1VgZvr87ZAw) trong series [Performance optimization với PostgreSQL](https://viblo.asia/s/performance-optimization-voi-postgresql-OVlYq8oal8W). **Table có thể được chia thành một hoặc nhiều partition khác nhau, dữ liệu được lưu trên partition chứ không lưu trên table chính.**

**Topic** cũng giống table, nó được chia thành một hoặc nhiều **partition** và message được lưu trên đó. Khi tạo **topic** cần xác định số lượng **partition** mong muốn.

* **Partition** được order và bắt đầu từ 0.
* Các message được lưu trong partition cũng được order theo thứ tự từ cũ đến mới, append liên tục bắt đầu từ giá trị 0, được gọi là **offset**.



Toppic chứa nhiều partition, và message được lưu trong partition

Mỗi ô vuông là 1 message tương đương với 1 dữ liệu, và các message là độc lập

Với ví dụ trên, cả 3 partition không có cùng số lượng message. Số lượng message của mỗi partition là độc lập, không phụ thuộc vào nhau. Như vậy, một message trong Kafka được xác định bởi 3 yếu tố:

* Topic name.
* Partition.
* Offset.

Vài điều cần chú ý:

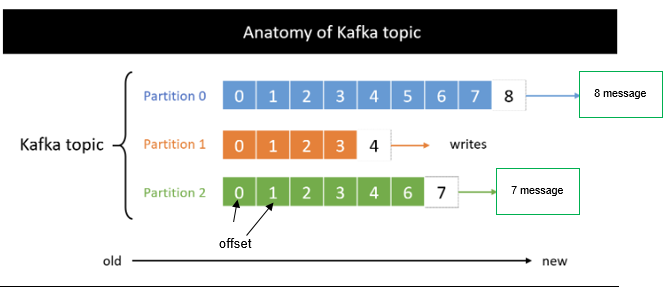
* Dù 2 message có cùng **offset** nhưng thuộc 2 **partition** khác nhau thì chúng cũng khác nhau. **Offset** chỉ có ý nghĩa trong cùng một **partition**.
* **Offset** có thứ tự, nhưng chỉ đảm bảo thứ tự trong cùng **partition**. Ví dụ trong cùng **partition 1**, message có **offset** = 3 chắc chắn đến sau message có **offset** = 2.
* Kafka là **Data pipeline**, đã giới thiệu ở bài trước. Message sau khi được consume không bị xóa ngay, default giữ lại trong 7 ngày - có thể config. Sau 7 ngày message bị xóa nhưng **offset** không reset mà tiếp tục tăng. **Offset is never go back**.
* Data sau khi lưu vào **partition** là **immutable** - bất biến, không thể thay đổi. Không thể update, không thể swap sang offset khác.

### Topic example

Mình đóng vai công ty Grab - cung cấp dịch vụ vận chuyển con người và hàng hóa. Mình muốn tracking vị trí của từng tài xế, cập nhật mỗi 20 giây. Áp dụng Kafka vào hệ thống để xử lý bài toán:

* **Tạo topic driver\_gps chứa thông tin về vị trí của toàn bộ tài xế.**
* Cứ 20 giây, thiết bị của tài xế sẽ gửi một message bao gồm: driver\_id và driver\_position lên **partition** của **driver\_gps** topic.

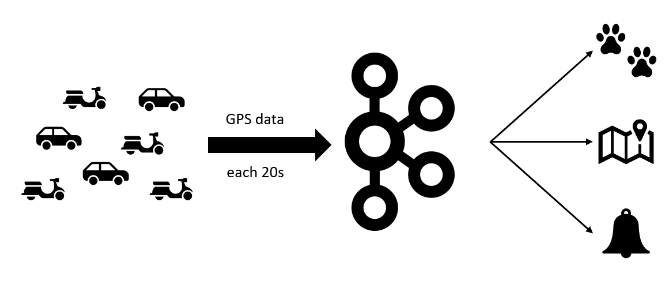
Túm cái váy lại, toàn bộ vài trăm nghìn tài xế của mình sẽ gửi message lên duy nhất một topic **driver\_gps**. Lúc này topic **driver\_gps** là **stream of data** của toàn bộ các tài xế. Khá dễ hiểu.



Sau khi message được đẩy lên Kafka, sẽ có rất nhiều **consumer** phía sau nhận message để xử lý:

* Consumer cho việc tracking location, hiển thị vị trí hiện tại của tài xế trên map.
* Consumer cho notification. Tài xế đã xuất phát chưa, đã đến nơi chưa.
* Consumer để tracking tài xế. Đang làm việc hay nghỉ ngơi, đã làm quá giờ chưa.. vân vân và mây mây.

Consumer là cái lấy dữ liệu từ kafka chuyển đến ng dùng chứ ko phải ng dùng. Người dùng trace dữ liệu từ consumer



# Kafka broker

Message được lưu tại **offset** của **partition**, **partition** được lưu ở **topic**. Vậy **topic** được lưu ở đâu?

**Topic** được lưu trữ trên file, trên disk, và tất cả đều được lưu trữ trên **server**. Và **server** là một **Kafka broker** trong **Kafka cluster**.

Vậy thường 1 broker sẽ là đại diện của 1 server

Chúng ta đã nghe nhiều về **single-point failure**: nếu chỉ deploy trên một server, không may server đó gặp sự cố thì.. còn cái lịt. Do vậy, các bài toán đều triển khai với multi-node trên multi-server, replicate hoặc cluster. Để đảm bảo 4 tính chất đã nêu ở đầu bài và xử lý single-point failure, Kafka sẽ triển khai **Kafka cluster**.

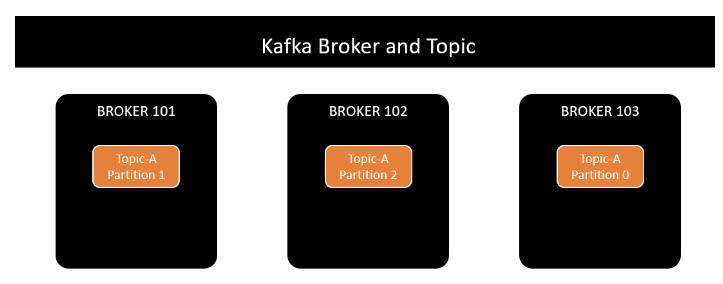
Số lượng **broker** trong **Kafka cluster** thường là 3, 5, 7, 10... Những bài toán siêu to khổng lồ có thể lên đến vài trăm **broker**, sách giáo khoa nói vậy chứ mình cũng chưa được làm với bài toán đó.

Mỗi **Kafka broker** được định danh bằng ID, là số nguyên integer. Mỗi **broker** lưu trữ một vài **partition**, không lưu trữ tất cả **partition** của **topic**.

Cuối cùng, khi connect đến bất kì **broker** nào trong mạng **Kafka cluster**, chúng ta sẽ connect tới toàn bộ mạng **cluster** đó. Không quan tâm **cluster** có bao nhiêu **broker**.

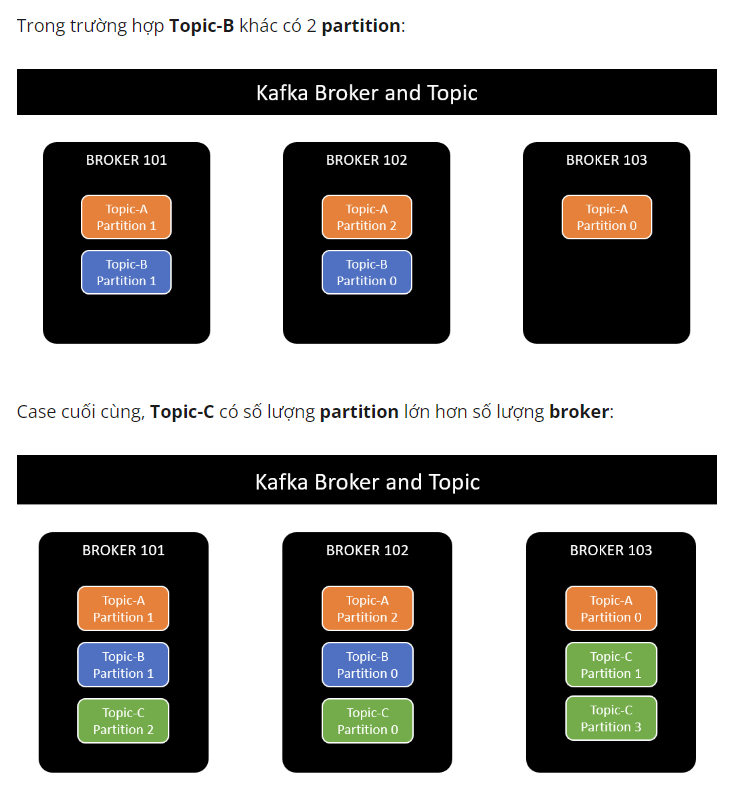
## Kafka broker & topic

Tiếp tục ví dụ trên, **Kafka cluster** của mình có 3 broker. Tạo thêm **Topic-A** cũng chia thành 3 **partition**, Kafka tổ chức chúng như sau:



Để đảm bảo **high reliable**, Kafka tự động phân tán các **partition** trên tất cả **broker** đang có. Mỗi **partition** nằm trên một **broker**. **Topic-A partition 1** có thể nằm trên bất kì **broker** nào mà không phụ thuộc thứ tự.

Tức là các partion trong 1 topic sẽ được chia đều ra các broker

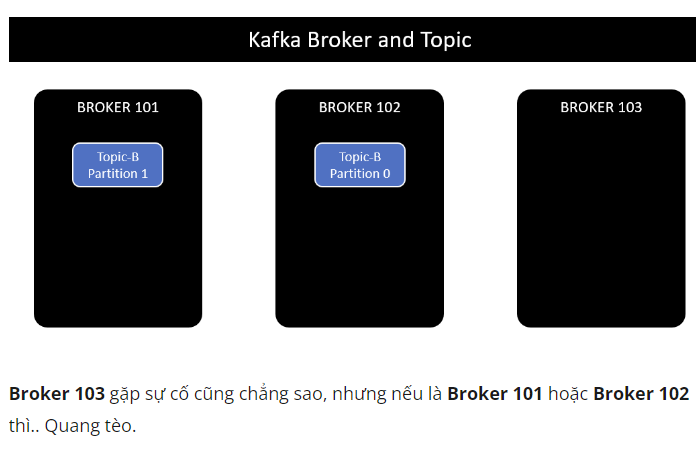


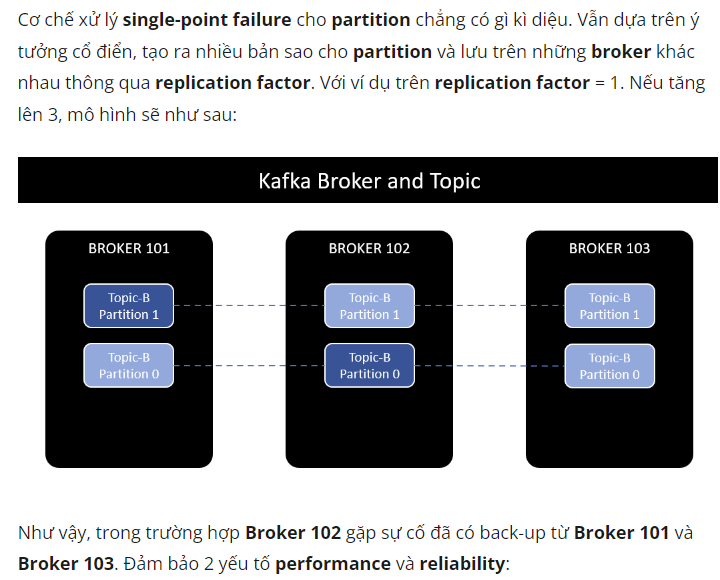
Topic C có 4 partition

Không có gì phức tạp, **Kafka cluster** sẽ phân tán các **partition** của cùng một topic ra nhiều **broker** nhất có thể. Không bỏ trứng vào cùng một giỏ.

## Topic replication

Với cách tổ chức **broker** và **partition** như trên vẫn chưa giải quyết triệt để **single-point failure**.

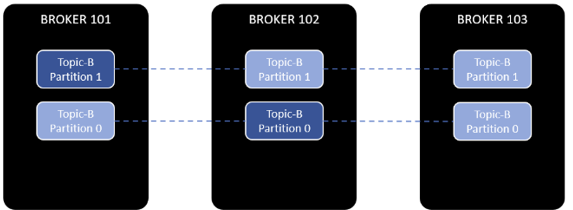




Tức là thằng này có gì, thằng kia cũng có cái đấy

# Gửi và nhận message trong Apache Kafka

Một **partition** có nhiều **replication** thì việc read/write message diễn ra trên **replication** nào hay tất cả các **replication**?



Có thể hình dung các **replication** của một **topic** là một dev team. Mỗi khi có thêm feature mới, có 2 cách để team quyết định implement luôn không:

* **Thứ nhất, cả team cùng vote**: cách này khá fair nhưng mất thời gian. Mỗi khi có feature mới lại họp hành biểu quyết - **Peer-to-Peer**.
* **Thứ hai, bầu ra leader quyết định thay cho cả team**: cách này phụ thuộc vào leader nhưng không mất thời gian họp hành. Ông leader nghỉ việc thì bầu ông khác lên thay - **Leader - Follower**

Tức là thay vì cả đống replicas tự vote bình bầu thì để thằng leader, và thằng leader chết thì thằng khác thay