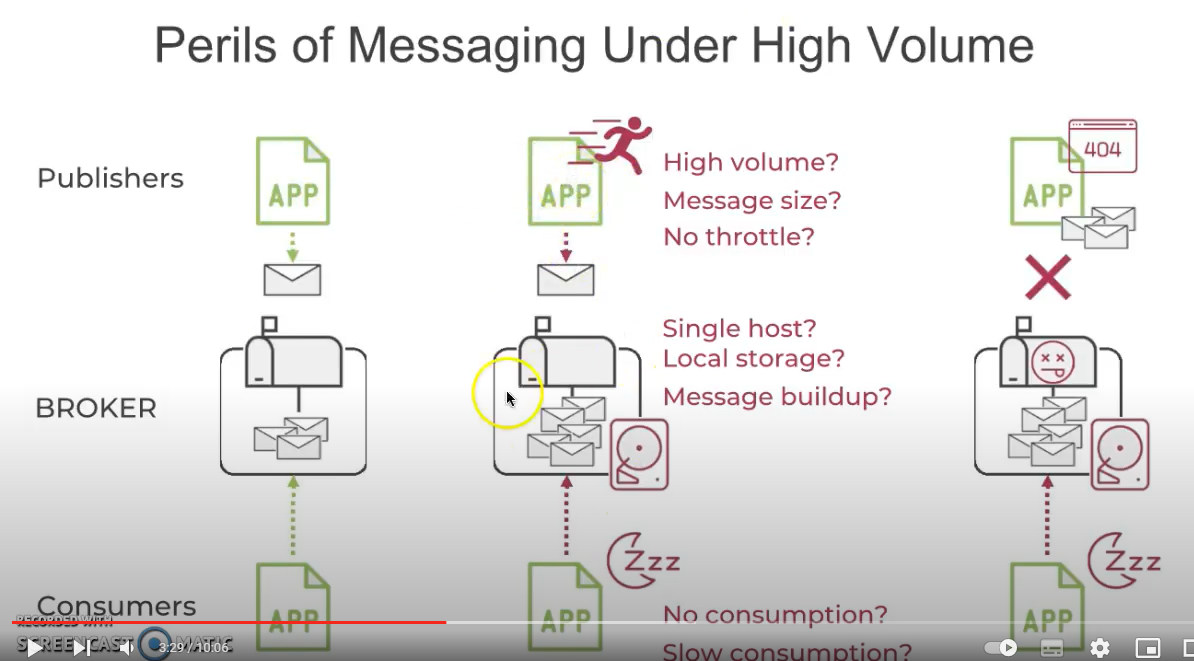


Kafka là gì:

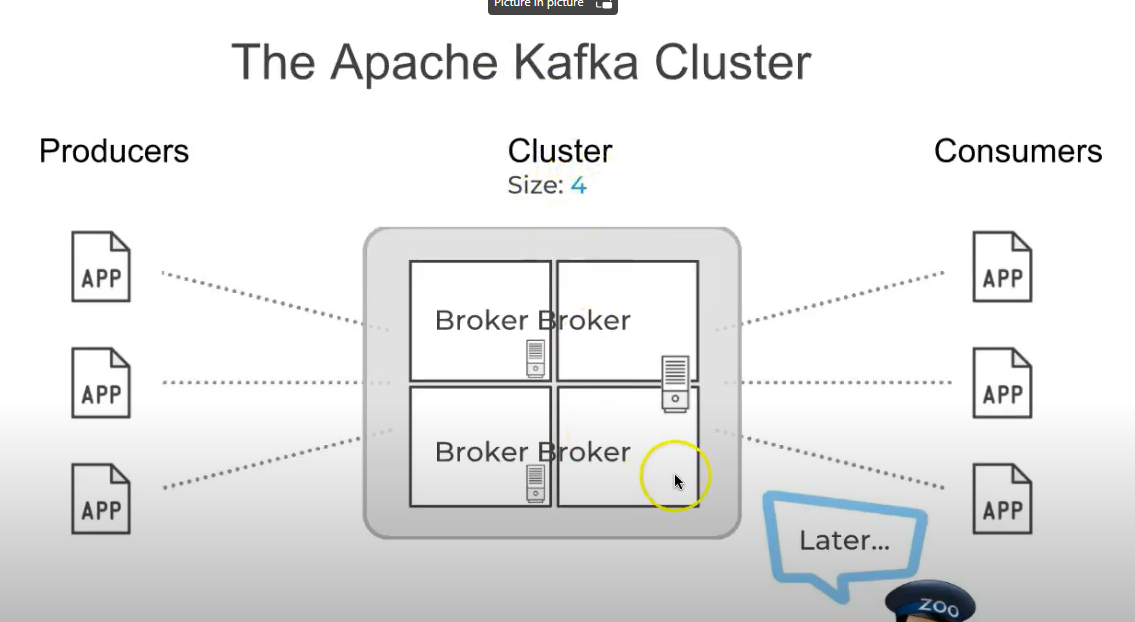


Publishers: người gửi

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



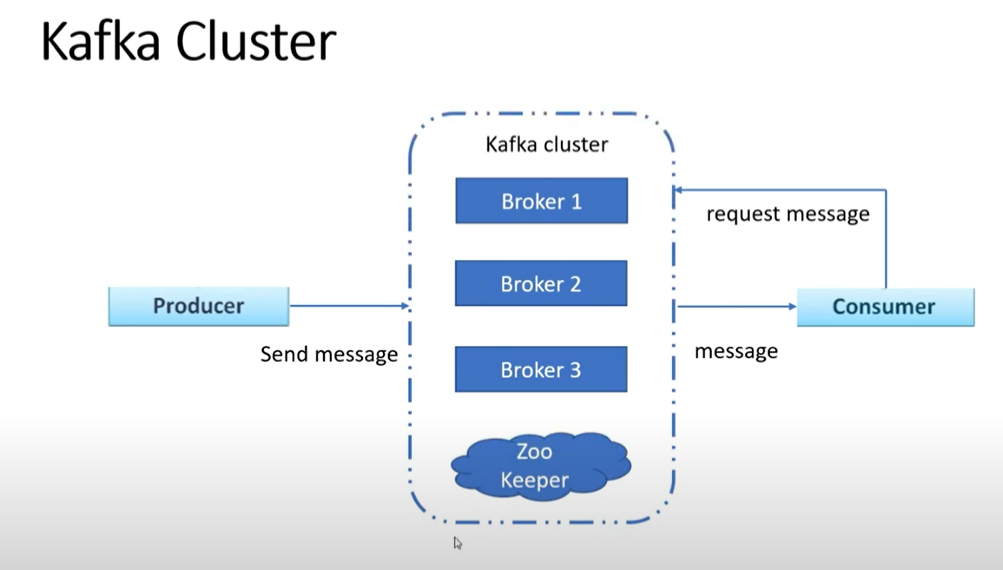
Cụm cluster là apache-kafka cluster

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

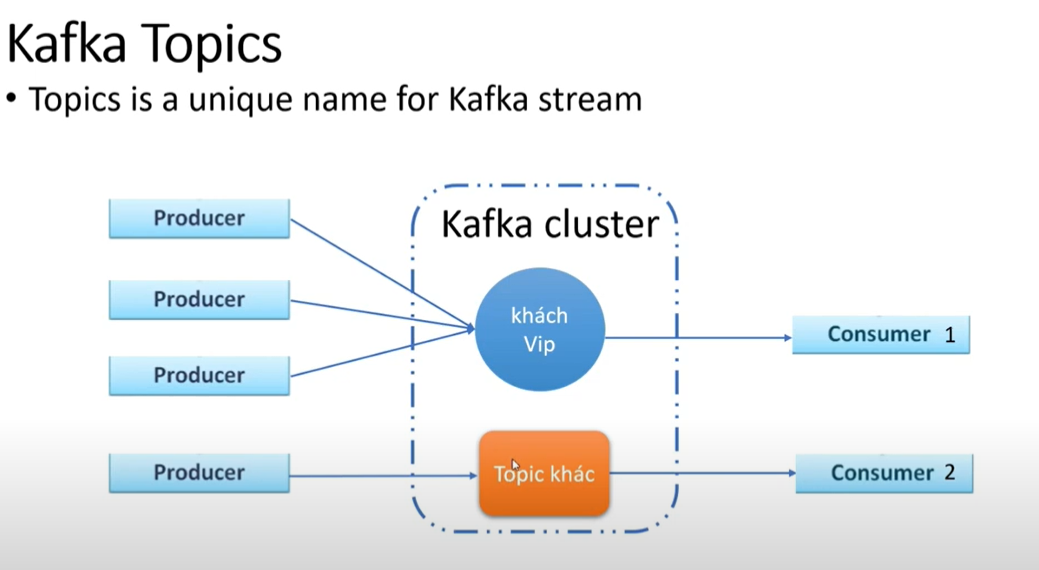
Kafka cluster : văn phòng môi giới

Zoo Keeper: Sếp

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

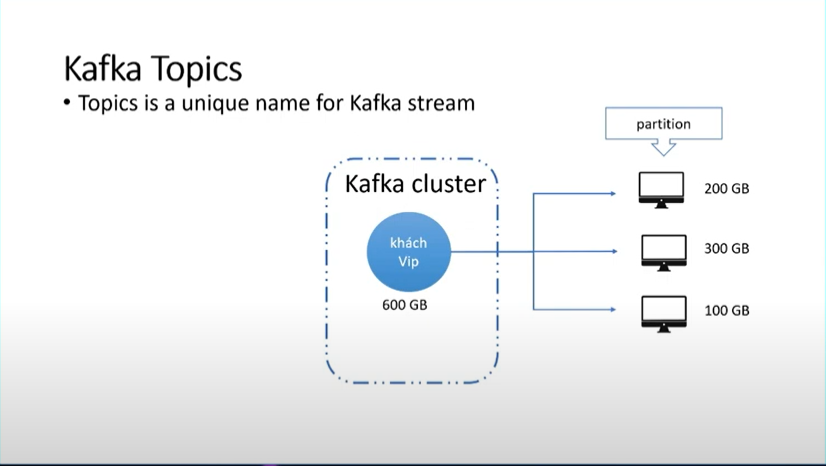
Broker 2 3, sẽ có bản copy để khi broker 1 bị hư thì broker 2, 3 sẽ thay thế

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

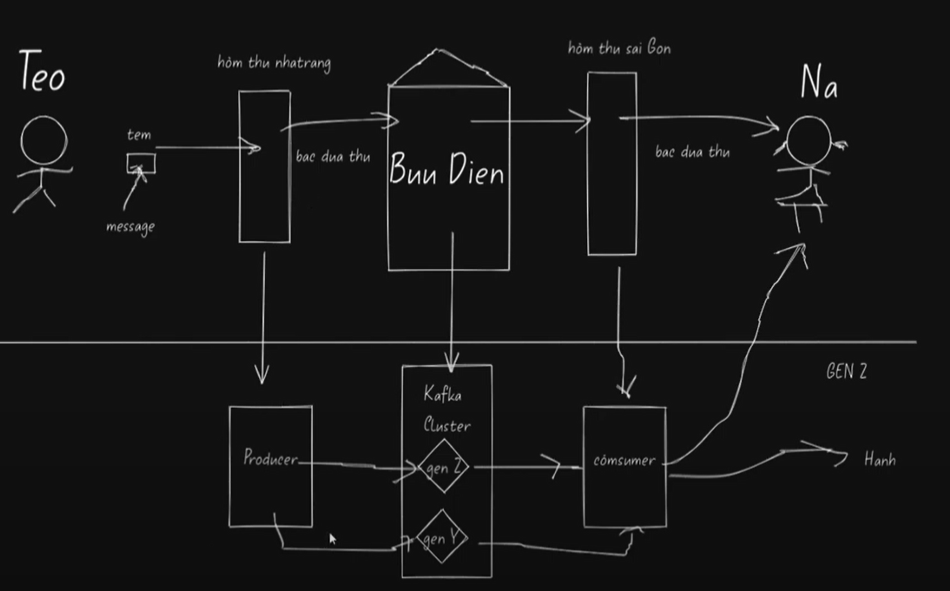


Chia ra các topic để kafka xử lý (nó có thể xét tính năng ưu tiên cho các topic)

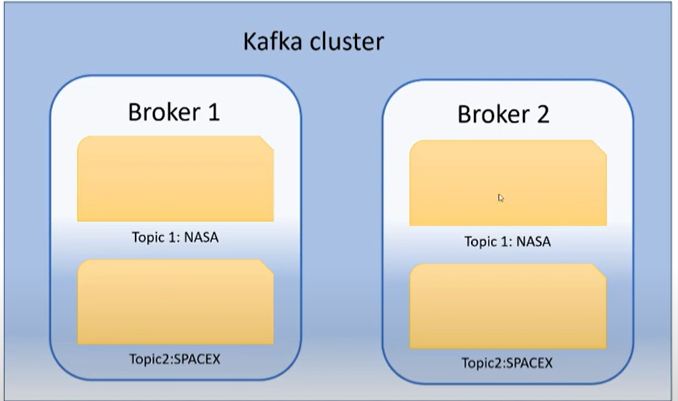
* Producer gửi thông tin cho kafka cluster
* Consumer sẽ nhận thông tin từ kafka cluster
* Trong kafka cluster sẽ có nhiều topic: khách víp, genZ, gen Y



* 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

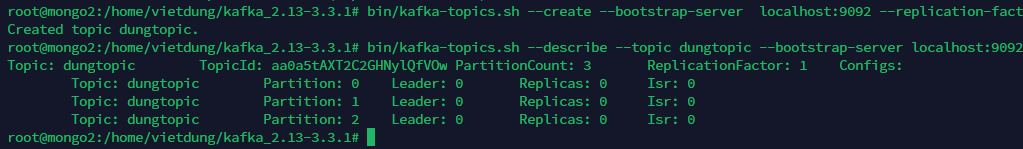
bin/kafka-topics.sh --create --zookeeper localhost:9092 --replication-factor 1 --partitions 3 --topic dungtopic

* lệnh lỗi, đổi thành

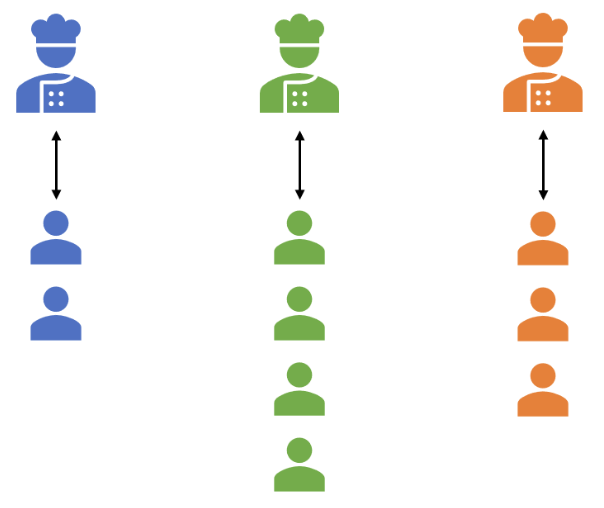
bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 3 --topic dungtopic

* 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:

Lưu ý: 1 message lưu trong 1 offset của 1 partion và sẽ sao lưu ở các partition khác trong 1 topic, 1 message ko thể bị tách ở nhiều offset ở nhiều partition, partition replicas sẽ được sao ở các broker khác

## 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.

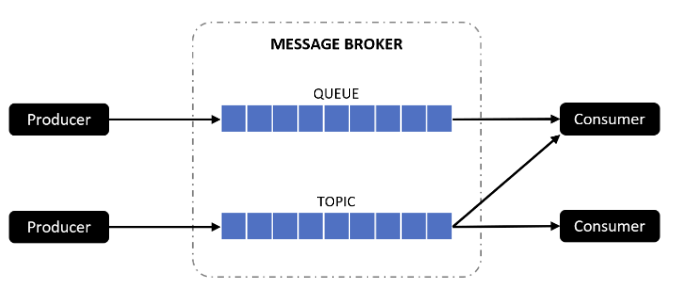
* Nhắn tin hai người có thể dùng **queue**.
* Khi nhắn tin trong group thì dùng **topic**.

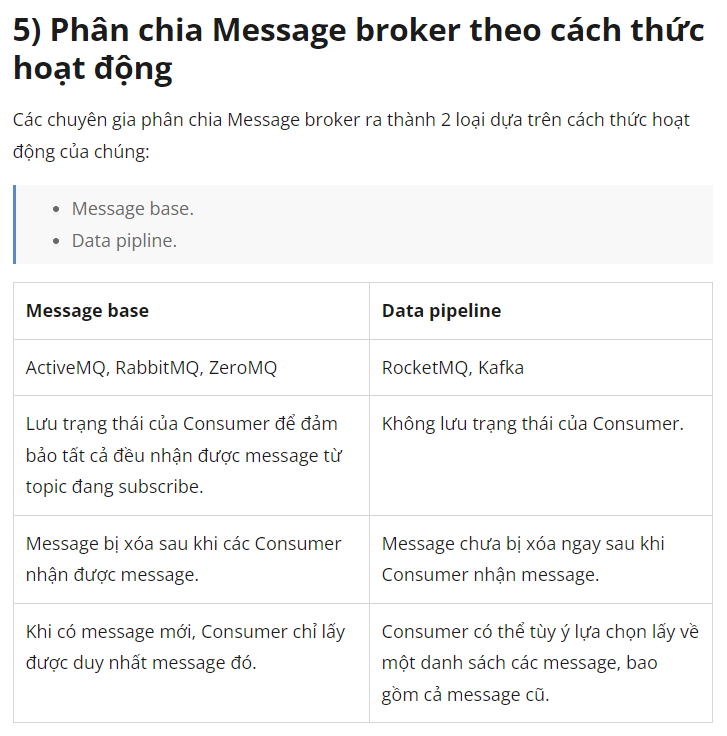
## Mô hình sử dụng Message broker

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.

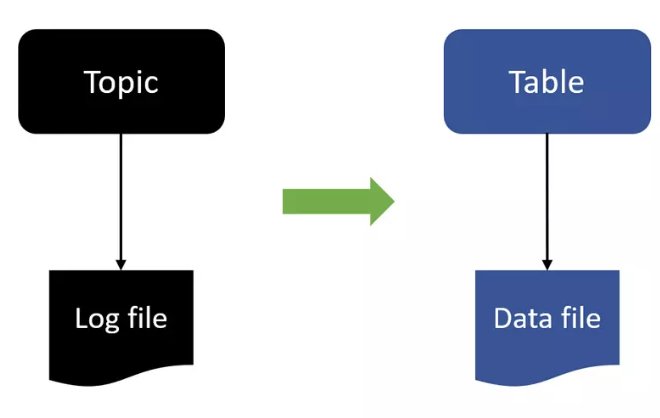
# Kafka topic, partition và offset

**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

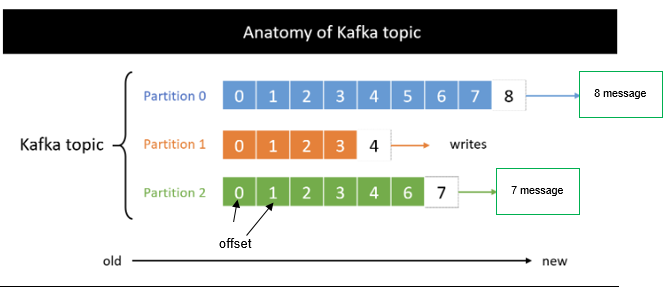


### Partition và offset

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:

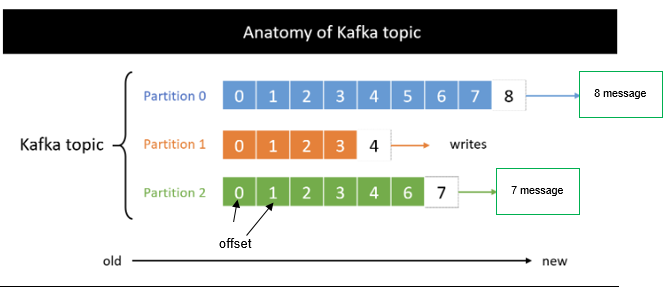
* Dù 2 message có cùng **offset** nhưng thuộc 2 **partition** khác nhau thì chúng cũng khác nhau. **Offset** chỉ có ý nghĩa trong cùng một **partition**.
* **Offset** có thứ tự, nhưng chỉ đảm bảo thứ tự trong cùng **partition**. Ví dụ trong cùng **partition 1**, message có **offset** = 3 chắc chắn đến 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.

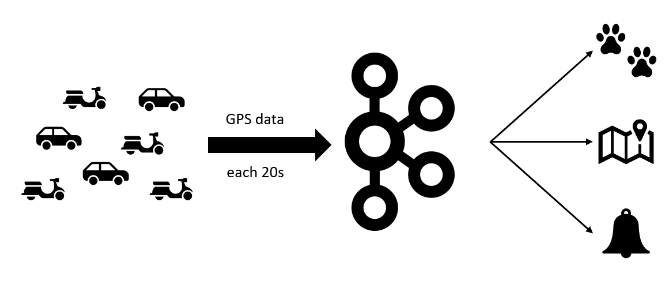
Túm cái váy lại, toàn bộ vài trăm nghìn tài xế của mình sẽ gửi message lên duy nhất một topic **driver\_gps**. Lúc này topic **driver\_gps** là **stream of data** của toàn bộ các tài xế. Khá dễ hiểu.



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**.

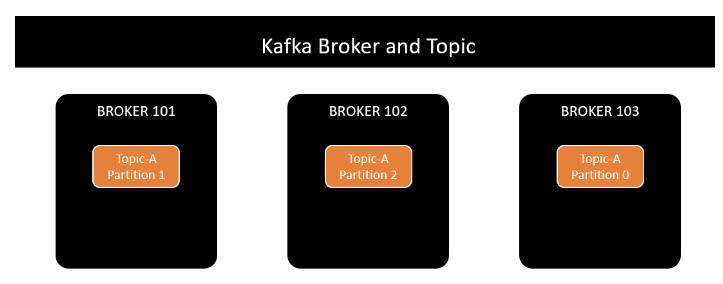
Số lượng **broker** trong **Kafka cluster** thường là 3, 5, 7, 10... Những bài toán siêu to khổng lồ có thể lên đến vài trăm **broker**, sách giáo khoa nói vậy chứ mình cũng chưa được làm với bài toán đó.

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**.

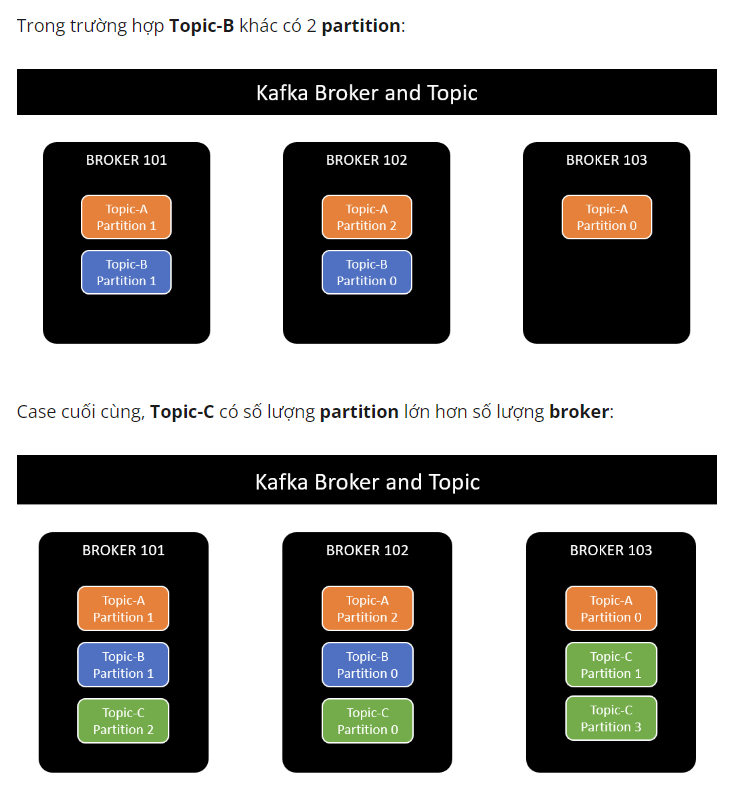
## Kafka broker & topic

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

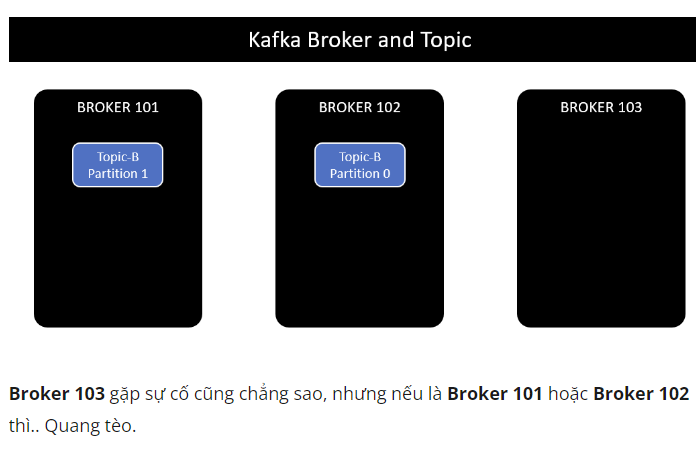


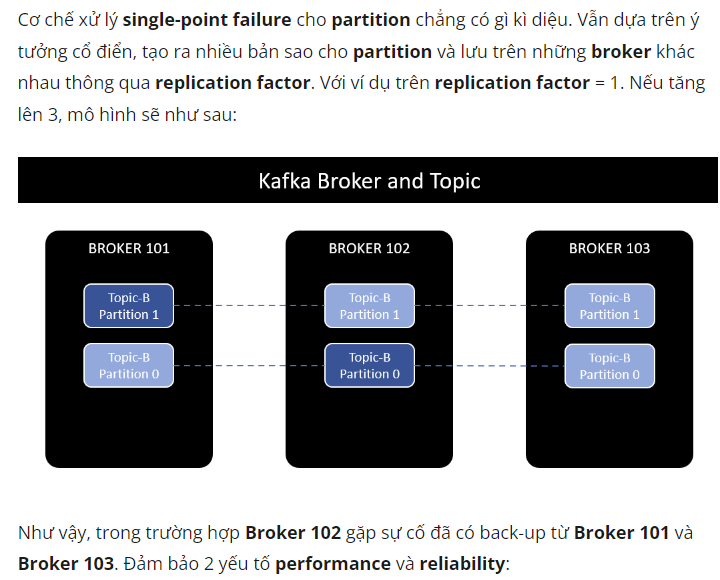
Topic C có 4 partition

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**.

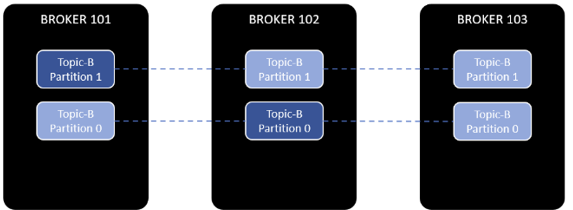




Tức là thằng này có gì, thằng kia cũng có cái đấy

# 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**?



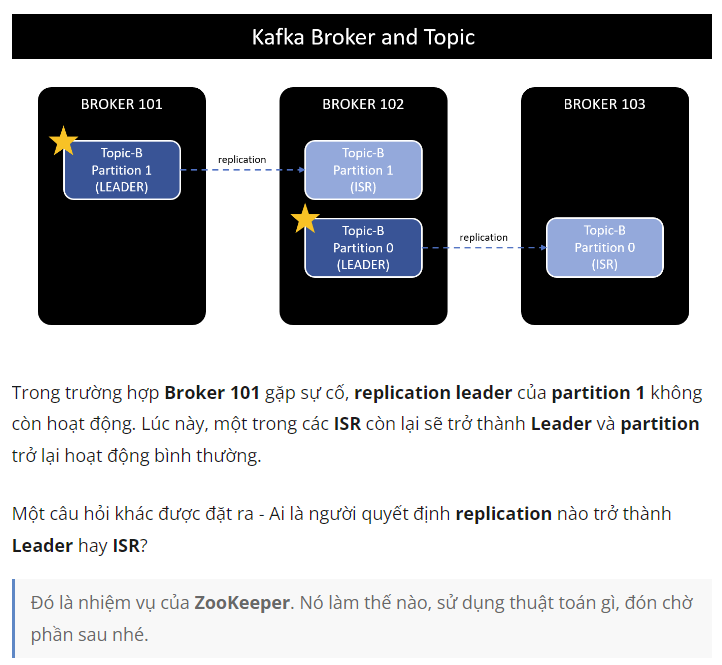
Có thể hình dung các **replication** của một **topic** là một dev team. Mỗi khi có thêm feature mới, có 2 cách để team quyết định implement luôn không:

* **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

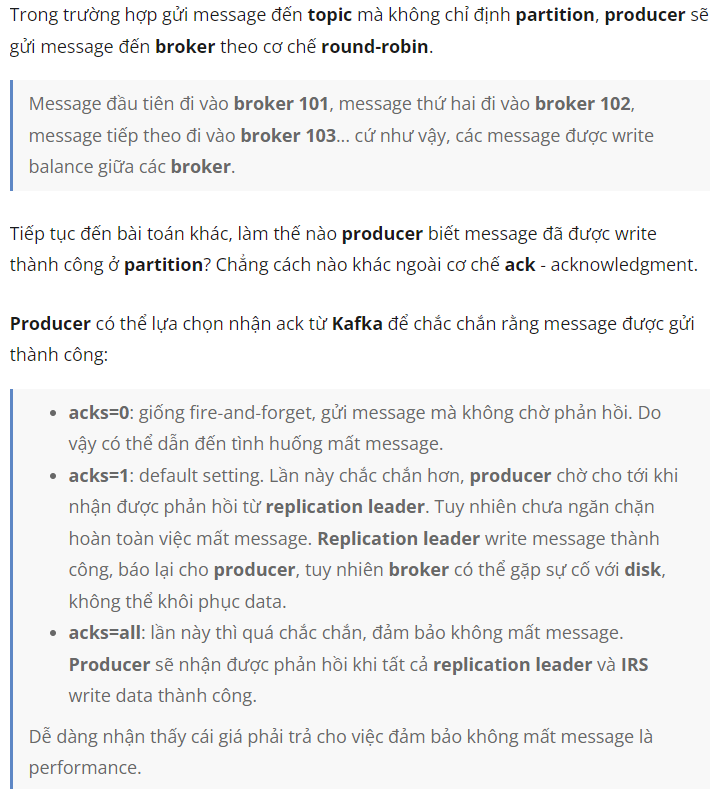
Như vậy, tất cả các thao tác read/write message của **partition** đều thông qua **replication leader**:

* Tại một thời điểm, mỗi **partition** có duy nhất một **replication leader**.
* Chỉ có thể read/write message từ **replication leader**.
* Các **replication** còn lại được gọi là **ISR**, đồng bộ message từ **replication leader**.
* Do vậy, mỗi **partition** có duy nhất một **replication leader** và một hoặc nhiều **ISR** - **in-sync replica**.

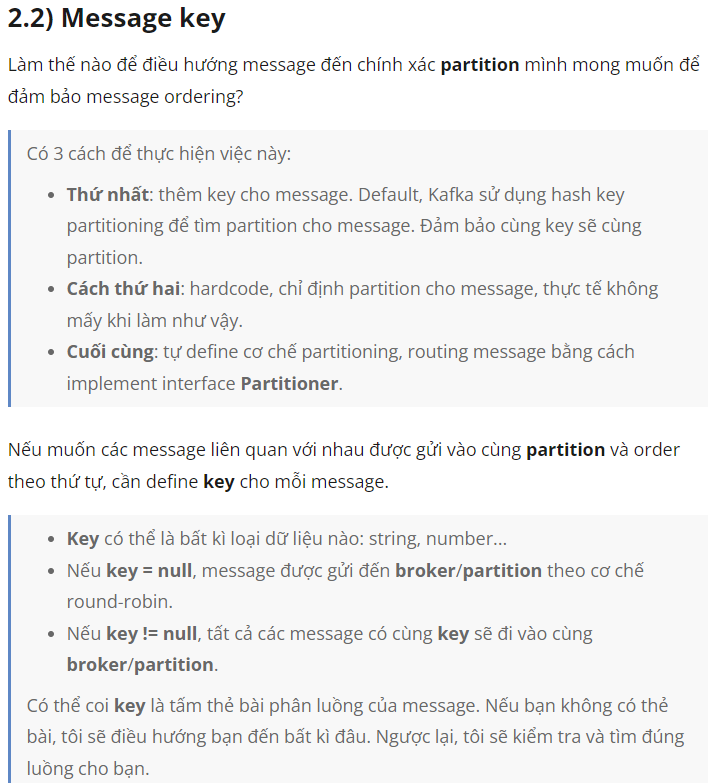


## Producer

Như ta đã biết, 1 message sẽ được lưu chọn vẹn trong 1 offset của 1 partition và sẽ được sao lưu ở partition khác trong cùng 1 topic nằm ở 1 broker khác (broker == 1 server), topic cũng được sao lưu ở các broker khác nhau



## Message key



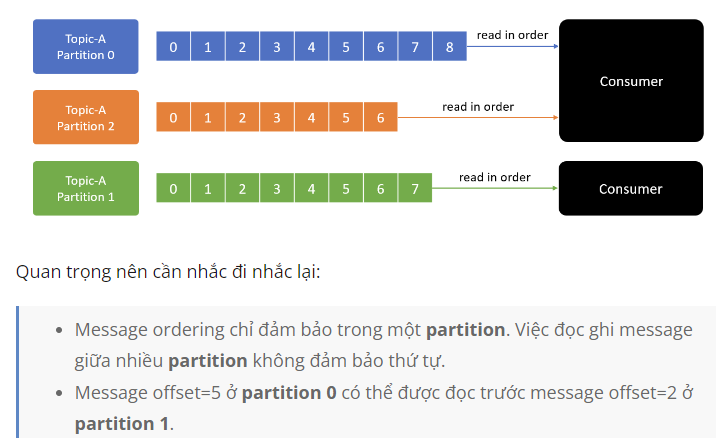
Chốt, với **producer** có 3 thứ cần quan tâm và ghi nhớ:

* Ackowledgement.
* Round-robin.
* Message key - Hash key partitioning.

## consumer

Có gửi thì có nhận, và **consumer** là đầu nhận message.

* **Consumer** đọc message từ **topic**, xác định bằng topic name.
* Đồng thời, **consumer** biết nên đọc message từ **broker** nào. Trong trường hợp chưa read xong mà **broker** gặp sự cố, **consumer** cũng có cơ chế tự phục hồi. (từ consumer mới đến ng dùng)
* Việc đọc message trong một **partition** diễn ra tuần tự để đảm bảo message ordering. Có nghĩa là **consumer** không thể đọc message **offset=3** khi chưa đọc message **offset=2**.
* Một **consumer** cũng có thể đọc message từ một hoặc nhiều hoặc tất cả **partition** trong một **topic**.



## Consumer group

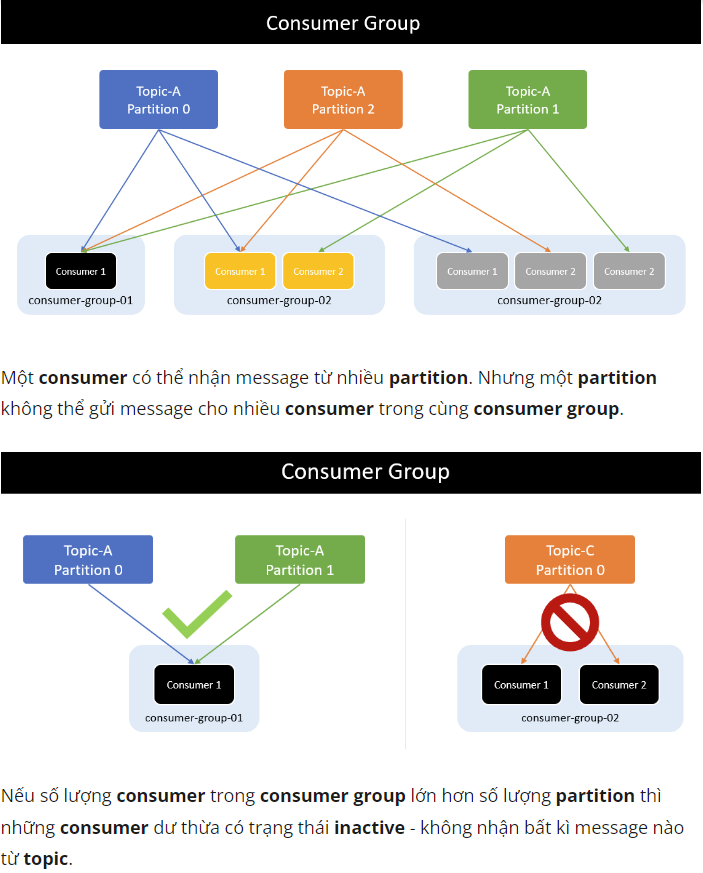
Đã có **consumer** lại mọc thêm **consumer group**. Vậy **consumer group** là gì và giải quyết vấn đề gì?

Quay lại phần trước, **consumer** là nơi đọc message từ **topic**. Có nghĩa là một **consumer** có thể đọc toàn bộ message của tất cả **partition** thuộc cùng **topic**.

Nếu số lượng **producer** tăng lên và đồng thời gửi message đến tất cả **partition** trong khi chỉ có duy nhất một **consumer** thì khả năng xử lý sẽ rất chậm, có thể dẫn tới **bottle-neck**. Giải pháp là tăng số lượng **consumer**, các **consumer** có thể xử lý đồng thời message từ nhiều **partition**. Và tất cả các **consumer** sẽ thuộc cùng một nhóm được gọi là **consumer group**.

Như vậy, **consumer group** read toàn bộ data của các **partition** và chia vào các **consumer** bên trong để xử lý.

Mỗi **consumer** thuộc **consumer group** sẽ đọc toàn bộ data của một hoặc nhiều **partition** để đảm bảo message ordering. Không tồn tại nhiều **consumer** cùng đọc message từ một **partition**.



## Consumer offset

Khi **consumer** xử lý xong message, chúng ta nên commit giá trị offset, các giá trị này sẽ lưu tại Kafka topic có tên là **\_\_consumer\_offsets**. Vì sao vậy?

Trong trường hợp **consumer** gặp sự cố và khởi động lại sau đó hoặc một **consumer** khác thay thế, nó có thể tiếp tục từ message cuối cùng được xử lý. Tránh trường hợp đọc lại toàn bộ message từ đầu.

# Kafka broker discovery

**Kafka brokers** bao gồm rất nhiều broker. Mỗi topic có thể có nhiều partition, mỗi partition được lưu trên các broker khác nhau. Tuy nhiên **consumer** chỉ cần connect tới một **broker** bất kì là có thể connect với toàn bộ **Kafka cluster**. Điều đó giúp **consumer** có thể đọc được message của topic nằm trên bất kì một **broker** nào.

Mỗi Kafka broker là một **bootstrap server**, nó có cơ chế thông báo cho client biết làm thế nào để connect tới các broker còn lại. Như vậy khi connect tới một broker là connect tới toàn bộ **Kafka brokers**.

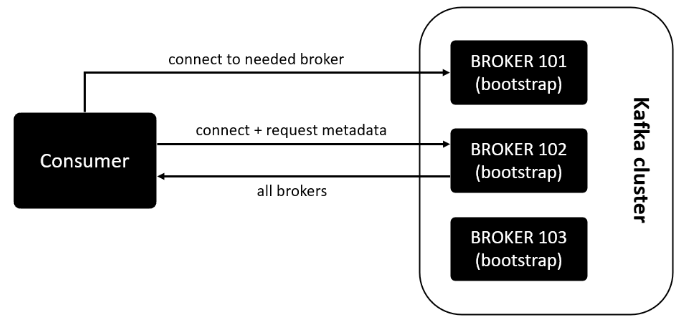
**Bootstrap servers** là một danh sách các endpoint của từng broker trong cụm cluster. Một broker có toàn bộ các thông tin của các broker còn lại.

## Cơ chế hđ các broker

Đầu tiên, **client** connect đến **Broker 102** và request **metadata**.

**Broker 102** trả về thông tin metadata và danh sách toàn bộ các **broker** hiện có trong mạng Kafka cluster.

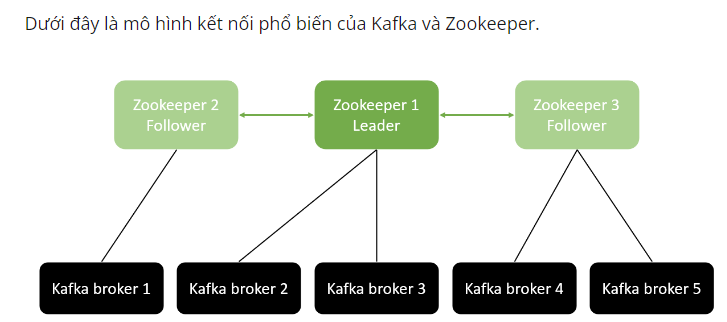
Sau khi có thông tin của toàn bộ **broker** và metadata, client sẽ quyết định connect tới **broker** nào để produce hoặc consume message.



# Zookeeper

**Zookeeper** mới chính là thứ đứng đằng sau tất cả.

* Lưu trữ tất cả thông tin của **Kafka broker**, **topic**, **partition**... Hay nói cách khác là quản lý chúng.
* Thực hiện **leader election** cho các partition.
* Gửi thông tin đến Kafka về các event phát sinh trong hệ thống: new topic, delete topic, partition die, broker die, broker comes up...



## Tổng kết

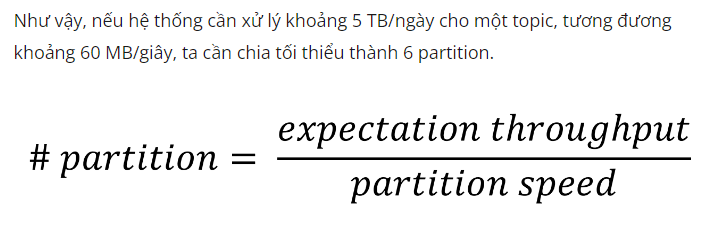
Chúng ta đã đi qua toàn bộ các thành phần của Apache Kafka và sẵn sàng practice với nó. Phần này chỉ nhằm mục đích khái quát hóa lại các tính chất của Kafka:

* Message được lưu tại **partition** theo thứ tự được gửi đi.
* **Consumer** read message theo thứ tự được lưu trong **partition**.
* Với replication factor = N, **producer** và **consumer** vẫn có thể hoạt động tốt khi N - 1 broker gặp sự cố.
* Đó cũng là lý do chúng ta hay chọn replication factor = 3 thay vì giá trị default = 1.
* Một broker có thể down cho việc maintenance.
* Một broker khác bị down do gặp sự cố.
* Nếu để giá trị replication factor cao quá cũng sẽ gặp vấn đề liên quan đến đồng bộ. Ngoài ra Kafka cũng không cho phép replication factor vượt quá số lượng broker.

Có thể coi topic giống chốt kiểm dịch Covid-19, partiton là các làn kiểm tra. Nếu được thiết kế với duy nhất một làn, các phương tiện phải nối đuôi nhau để thông quan.

Nếu tăng số lượng làn kiểm tra, số lượng phương tiện cùng được qua trạm trong một thời điểm sẽ nhiều hơn. Throughput có thể tăng từ 1 lên 5 phương tiện/phút.

Mỗi topic partition có khả năng xử lý 10 MB/s



Nghe có vẻ đơn giản , có gì đó sai sai không nhỉ?

Hệ thống sẽ chạy hoàn hảo như những gì ta mong muốn nếu các message gửi đi không define key. Lúc này Kafka sẽ round-robin message tới partition, đảm bảo số lượng message trong mỗi partition cân bằng với nhau, không chênh lệch nhiều.

Thực tế, message được gửi kèm key. Lúc này Kafka sẽ xác định partition cho message dựa trên key hashing để đảm bảo các message cùng key luôn route đến cùng partition. Như vậy sẽ có trường hợp partition có quá nhiều message, trong khi đó có partition chẳng có message nào.

Mặc dù có đến 10 làn đường nhưng chỉ có 2 làn đường cho xe đạp và xe thô sơ.

Bình thường không đến nỗi nào, dù xe cộ có đông, cả ô tô lẫn xe máy thì cũng không quá tắc đường.

Bỗng một ngày đẹp trời, người dân nổi hứng yêu mẹ thiên thiên chuyển sang đi xe đạp hết thì.. tắc như đường Trường Chinh vì chỉ sử dụng được 2 làn xe.

## File descriptor

Mỗi partition được lưu trữ tại một folder của broker. Như vậy, số lượng partition tỉ lệ thuận với số lượng file phải xử lý của broker.

Mỗi partition cần xử lý 2 files: index và actual data.

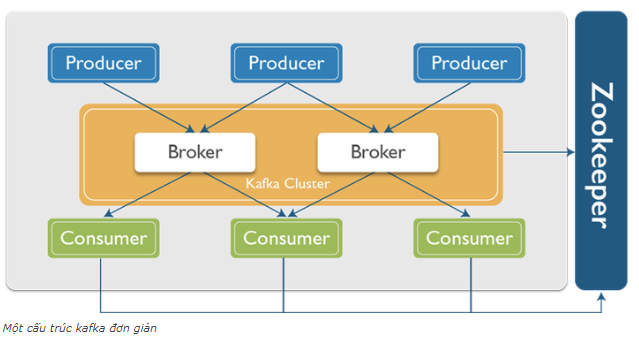
Vấn đề này liên quan tới I/O của broker, tốc độ đọc ghi ổ cứng và config của OS. Nếu số lượng open files quá lớn ta cần setting lại config phù hợp thông qua câu lệnh:

sysctl fs.file-max

Đây không phải big problem và thường xuyên xảy ra nhưng cũng cần biết về nó để tính các phương án dự phòng khi chúng ta set số lượng lớn partition.

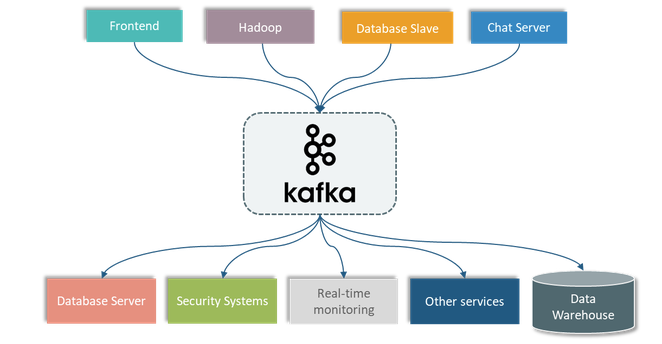
# Tổng kết – kafka

**Kafka** là gì? Là hệ thống message pub/sub phân tán (distributed messaging system). Bên pulbic dữ liệu được gọi là producer, bên subscribe nhận dữ liệu theo topic được gọi là consumer. Kafka có khả năng truyền một lượng lớn message theo thời gian thực, trong trường hợp bên nhận chưa nhận message vẫn được lưu trữ sao lưu trên một hàng đợi và cả trên ổ đĩa bảo đảm an toàn. Đồng thời nó cũng được replicate trong cluster giúp phòng tránh mất dữ liệu



* **PRODUCER**: Kafka lưu, phân loại message theo topic, sử dụng producer để publish message vào các topic. Dữ liệu được gửi đển partition của topic lưu trữ trên Broker.
* **CONSUMER**: Kafka sử dụng consumer để subscribe vào topic, các consumer được định danh bằng các group name. Nhiều consumer có thể cùng đọc một topic.
* **TOPIC**: Dữ liệu truyền trong Kafka theo topic, khi cần truyền dữ liệu cho các ứng dụng khác nhau thì sẽ tạo ra cá topic khác nhau.
* **PARTITION**: Đây là nơi dữ liệu cho một topic được lưu trữ. Một topic có thể có một hay nhiều partition. Trên mỗi partition thì dữ liệu lưu trữ cố định và được gán cho một ID gọi là offset. Trong một Kafka cluster thì một partition có thể replicate (sao chép) ra nhiều bản. Trong đó có một bản leader chịu trách nhiệm đọc ghi dữ liệu và các bản còn lại gọi là follower. Khi bản leader bị lỗi thì sẽ có một bản follower lên làm leader thay thế. Nếu muốn dùng nhiều consumer đọc song song dữ liệu của một topic thì topic đó cần phải có nhiều partition.
* **BROKER**: Kafka cluster là một set các server, mỗi một set này được gọi là 1 broker
* **ZOOKEEPER**: được dùng để quản lý và bố trí các broker.

Sử dụng kafka cho

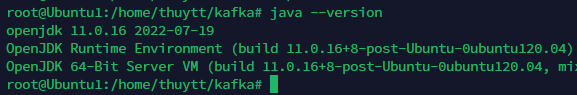


# Thực hành Apache Kafka với Kafka CLI

### Download

Đầu tiên cần set up môi trường với Kafka stand-alone (single broker).

Đảm bảo đã cài đặt sẵn JDK trên máy nhé, thấp nhất là version 8.



Sau đó download Apache Kafka [tại đây](https://kafka.apache.org/downloads), hiện tại latest version là 2.8.0.

Giải nén và bắt đầu cuộc hành trình. Mở terminal, kiểm tra thư mục hiện tại, nếu thấy output như này là ok:

tar -vxf kafka-3.3.1-src.tgz

$ ls

bin config libs LICENSE NOTICE site-docs

Tiếp theo là add PATH để thực thi cho các câu lệnh cho nhanh:

vi ~/.bashrc

****

### Start Zookeeper

Vì zookeeper quan trọng và để quản lý các broker,… , nên tạo zookeeper trước

* tạo folder data cho Zookeeper tại bất kì chỗ nào bạn muốn, hoặc tạo trong folder Kafka: (tạo để lưu data thôi, mặc định data lưu ở /tmp/zookeeper)

mkdir -p data/zookeeper

Sau đó sửa config để Zookeeper write snapshot data ra folder vừa tạo:

vi config/zookeeper.properties

Tìm dòng config dataDir=/tmp/zookeeper và thay thế bằng đường dẫn đến thư mục:

dataDir=/home/admin/kafka\_2.13-2.8.0/data/zookeeper

vi config/zookeeper.properties

Tìm dòng config dataDir=/tmp/zookeeper và thay thế bằng đường dẫn đến thư mục:

dataDir=/home/admin/kafka\_2.13-2.8.0/data/zookeeper

Start Zookeeper:

$ zookeeper-server-start.sh config/zookeeeper.properties

Zookeeper start mặc định với port 2181

Lúc này khi mở folder data zookeeper sẽ thấy folder **version-2**.

Nếu start lại zookeeper là mỗi ko start được thì vào data của zookeeper xóa đi, ví dụ cái **version-2** kia

### Start Kafka

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

Kafka start với port 9092

Có thể tạo folder lưu trữ giống zookeeper ở trên nhưng có thôi

Hiện tại mới khởi tạo kafka standlone thôi, khởi tạo trên 1 node

## Practice with CLI

### Topic

Tạo topic name bất kì với 4 required options:

* **--bootstrap-server**: địa chỉ của Kafka server.
* **--topic**: topic name.
* **--partitions**: số lượng partitions muốn tạo.
* **--replication-factor**: số lượng replication factor cho mỗi partition. Lưu ý rằng không thể tạo quá số lượng broker trong mạng cluster. Vì việc 1 broker giữ đến 2 bản sau của partition hoàn toàn không có ý nghĩa, do đó với ví dụ này, replication factor chỉ có thể = 1.

kafka-topics.sh \

--bootstrap-server localhost:9092 \

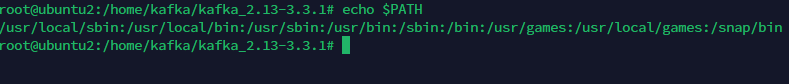
--topic my-first-topic \

--create \

--partitions 3 \

--replication-factor 1

ak quên, trước đấy tạo luôn 1 PATH cho nhanh, đỡ phải dùng /bin/kafka-topics.sh



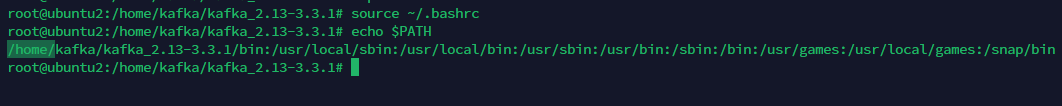
nano ~/.bashrc

add thêm

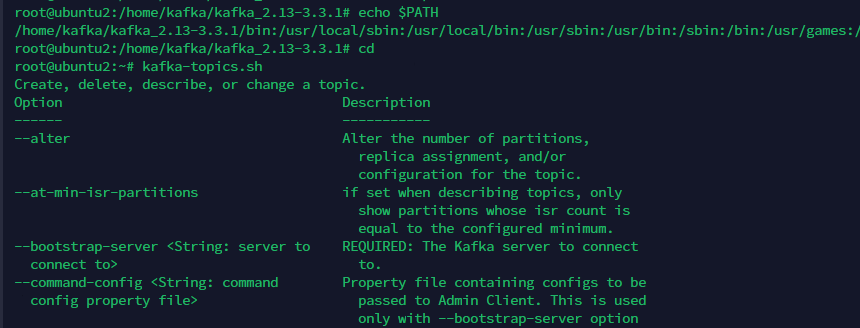
export PATH=/home/kafka/kafka\_2.13-3.3.1/bin:$PATH

source ~/.bashrc

:$PATH : tức là vẫn giữ những cái cũ của path



Oke đã có, test thử



Oke, tức là dù đang ở vị trí nào, mình vẫn gọi đến được các file trong folder /bin của kafka

* Tạo 1 topic

kafka-topics.sh \

--bootstrap-server localhost:9092 \

--topic my-first-topic \

--create \

--partitions 3 \

--replication-factor 1

replication factor chỉ có thể = 1 vì chỉ có 1 broker,

replication factor; là số bản sao partition, cho nên 1 broker có 2 bản sao partition ko để làm j cả

Để list toàn bộ topic hiện có, sử dụng **--list** option:

kafka-topics.sh \

--bootstrap-server localhost:9092 \

--list

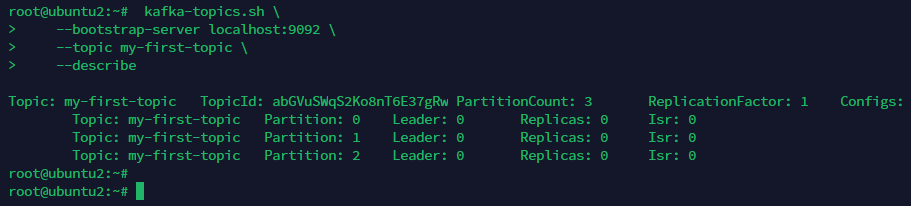
Sử dụng **--describe** options trong trường hợp muốn xem nhiều thông số hơn về topic:

kafka-topics.sh \

--bootstrap-server localhost:9092 \

--topic my-first-topic \

--describe



Dòng đầu tiên có các thông tin chung về topic như **topic name**, số lượng **partition**, **replication-factor** và các additional config.

Các dòng tiếp theo diễn tả về thông số của từng partition. Nếu để ý sẽ thấy cả 3 thông số Leader, Replicas và Isr đều bằng 0. Vì nó là **broker id**, không phải số lượng. Với các hệ thống có nhiều hơn 1 broker, con số này có thể khác.

Nếu muốn xóa topic, sử dụng option **--delete**:

kafka-topics.sh \

--bootstrap-server localhost:9092 \

--topic my-first-topic \

--delete

### Producer

Tạo **topic** thành công, bước tiếp theo là tạo **producer** và gửi message với 1 required option:

* **--bootstrap-server**: list địa chỉ Kafka broker dưới dạng host1:port1,host2:port2.

kafka-console-producer.sh \

--bootstrap-server localhost:9092 \

--topic my-first-topic

>

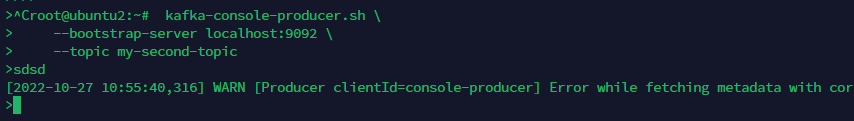
Nếu thấy output như trên tức là đã tạo thành công **producer**, tiến hành gửi vài message:

hello world

this is Kafka CLI practice

this is third message

bye now



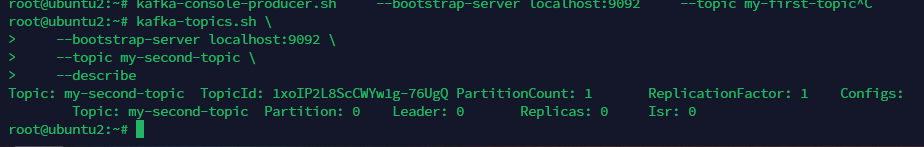
Trường hợp topic ko tồn tại thì message đầu tiên sẽ bị lỗi

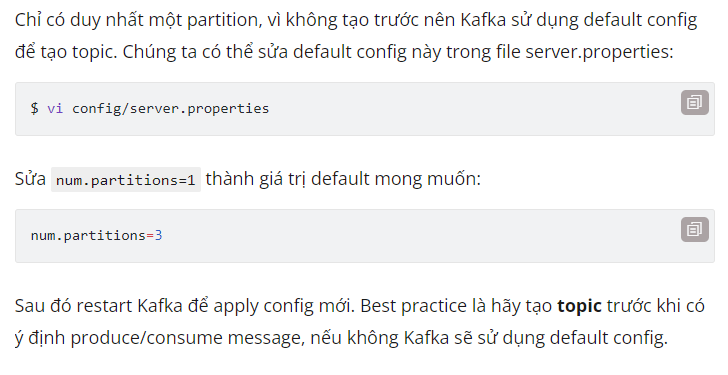
Các lần sau n sẽ ko bị warning nữa vì n đã tự tạo

vậy trong trường hợp producer produce message đến topic chưa tồn tại, Kafka sẽ tự động tạo topic. Dẫn đến 2 question:

* Message đầu tiên gửi đến có thành công không?
* Nếu Kafka tự tạo topic thì số lượng partition và replication-factor là bao nhiêu?

Sử dụng option **--describe** để kiểm tra:





### Consumer

Phần này sẽ practice Kafka console consumer để verify message đã gửi ở phần trước cũng với 1 required option là **--bootstrap-server**

### **command**

* **Xem message ở luồng stream**

kafka-console-consumer.sh \

--bootstrap-server localhost:9092 \

--topic my-first-topic

* Tức là phải mở bên

kafka-console-producer.sh \

--bootstrap-server localhost:9092 \

--topic my-second-topic

Bên này gửi message thì bên kia mới nhận được, nhận kiểu live stream

* Xem message từ đầu

kafka-console-consumer.sh \

--bootstrap-server localhost:9092 \

--topic my-first-topic \

--from-beginning

Vì sao consumer không đọc toàn bộ message từ đầu?

Thử tưởng tượng topic có vài trăm nghìn hoặc hàng triệu message, chúng ta sẽ mất kha khá thời gian xử lý hết trước khi đến message hiện tại. Lý do thứ hai là vì design của Kafka hướng đến hệ thống stream, real-time processing, vì vậy default không đọc lại toàn bộ message của topic.

### Consumer group