Table of Contents

[**RabbitMQ và Kafka - Hai hệ thống truyền tin khác nhau** 2](#_Toc126154423)

[RabbitMQ 2](#_Toc126154424)

[Exchange và Queue 2](#_Toc126154425)

[SỰ BẢO ĐẢM 4](#_Toc126154426)

[PUSH VÀ CONSUMER PREFETCH 5](#_Toc126154427)

[Định tuyến 5](#_Toc126154428)

[Dead Letter Exchange 6](#_Toc126154429)

[Ephemeral Exchange và Queue 7](#_Toc126154430)

[Plug-Ins 7](#_Toc126154431)

[Apache Kafka 7](#_Toc126154432)

[Phân vùng và nhóm Consumer 10](#_Toc126154433)

[PUSH VS PULL 11](#_Toc126154434)

[Publish/Subscribe 12](#_Toc126154435)

[Log Compaction 15](#_Toc126154436)

[Các thông tin thêm về Thứ tự của các Message 15](#_Toc126154437)

[Cấu trúc của RabbitMQ và các messaging pattern khi sử dụng RabbitMQ 16](#_Toc126154438)

[Các loại định tuyến cơ bản của RabbitMQ 16](#_Toc126154439)

[Fanout Exchanges 16](#_Toc126154440)

[Direct Exchanges và Default Exchange 17](#_Toc126154441)

[Topic Exchange 18](#_Toc126154442)

[Header Exchange 19](#_Toc126154443)

[Consistent Hashing Exchange 20](#_Toc126154444)

[Dead Letter Exchange 21](#_Toc126154445)

[Ephemeral Exchanges và Queue 21](#_Toc126154446)

[Alternate Exchange - Exchange thay thế 22](#_Toc126154447)

[Priority Queue - Hàng đợi ưu tiên 22](#_Toc126154448)

[CC và BCC 22](#_Toc126154449)

[Khai báo các Exchange, Hàng đợi và Liên kết 22](#_Toc126154450)

[Virtual Host 23](#_Toc126154451)

[Ví dụ về một số pattern khi sử dụng RabbitMQ 23](#_Toc126154452)

[1 - Simple Broadcast với Fanout Exchange 23](#_Toc126154453)

[2 - Nhiều lớp Exchange 23](#_Toc126154454)

[3 - Hệ thống định tuyến email với Header Exchange 25](#_Toc126154455)

[4 - Public Message Exchange, Private Consumer Exchange 27](#_Toc126154456)

[5 - Point-to-point Messaging 28](#_Toc126154457)

[6 - Xử lý đơn hàng nhạy cảm 28](#_Toc126154458)

[7 - Dữ liệu cục bộ 30](#_Toc126154459)

[8 - Định tuyến phân cấp 30](#_Toc126154460)

[9 - Lớp dịch vụ với các Hàng đợi ưu tiên 31](#_Toc126154461)

[10 - Lớp dịch vụ với các Topic Exchange 32](#_Toc126154462)

[11 - Định tuyến If/Else với Alternate Exchange 32](#_Toc126154463)

[12 - KHÔNG ĐỊNH TUYẾN với Alternate Exchange 33](#_Toc126154464)

[13 - Delayed Retry với Cascading Exchange và Queue 33](#_Toc126154465)

[14 - Delayed Retry với Ephemeral Exchanges và Queue 35](#_Toc126154466)

[15 - Delayed Delivery với Delayed Message Exchange Plug-In 37](#_Toc126154467)

[16 - Delayed Delivery với Cascading Exchange và Queue 37](#_Toc126154468)

[17 - Delayed Delivery với Private Publisher Exchange 37](#_Toc126154469)

[18 - RPC và hàng đợi trả lời 38](#_Toc126154470)

[Vòng đời thông điệp 42](#_Toc126154471)

[Cấu trúc của Kafka và các messaging pattern khi sử dụng Kafka 43](#_Toc126154472)

[Ví dụ về một số pattern khi sử dụng Kafka 44](#_Toc126154473)

[#1 Gửi thông điệp với Publish Subscribe đơn giản kết hợp tùy chọn quay vòng 44](#_Toc126154474)

[#2 Event-Driven Data Integration 44](#_Toc126154475)

[#3 - Lưu lượng truy cập cao, xử lý các ứng dụng nhạy cảm 50](#_Toc126154476)

[#4 Dữ liệu cục bộ 50](#_Toc126154477)

[#5 Failures, Retries và Message Lifecycle 51](#_Toc126154478)

1. **RabbitMQ và Kafka - Hai hệ thống truyền tin khác nhau**

Trong phần này, chúng ta sẽ cùng tìm hiểu xem RabbitMQ và Apache Kafka là gì và cách tiếp cận truyền thông điệp của chúng. Mỗi loại đều có các cách tiếp cận khác nhau trong thiết kế về mọi khía cạnh, cả hai đều có điểm mạnh và điểm yếu. Chúng ta sẽ không cố đưa ra bất kỳ một kết luận nào khẳng định rằng một trong số chúng tốt hơn cái còn lại trong phần này, thay vào đó hãy nghĩ về điều này như là một bước đệm để chúng ta đi sâu hơn trong các phần tiếp theo.

## **RabbitMQ**

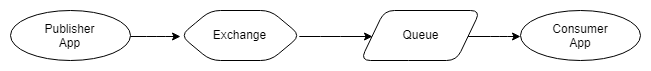
RabbitMQ là một hệ thống message queue phân tán. Nó được cho là phân tán là bởi vì nó thường chạy như một cụm các node, với các hàng đợi được trải rộng trên các node và có thể nhân rộng tùy biến để đem lại khả năng chịu lỗi và tính sẵn sàng cao. Nó được thiết kế dựa theo AMQP 0.9.1 và cung cấp các giao thức khác như STOMP, MQTT và HTTP thông qua các plug-in.

RabbitMQ có cách tiếp cận truyền thông điệp vừa truyền thống và cũng rất mới mẻ. Truyền thống là bởi nó được định hướng xung quanh các hàng đợi thông điệp, còn mới ở đây là khả năng định tuyến rất linh hoạt của nó. Khả năng định tuyến này là tính năng nổi bật nhất của RabbitMQ. Việc xây dựng một hệ thống truyền thông điệp phân tán nhanh, có khả năng mở rộng cao, và đáng tin cậy là một trong những điều đáng chú ý của RabbitMQ, tuy nhiên khả năng định tuyến thông điệp linh hoạt mới là điều làm cho RabbitMQ thực sự nổi bật trong vô số các công nghệ truyền thông điệp hiện nay.

### Exchange và Queue

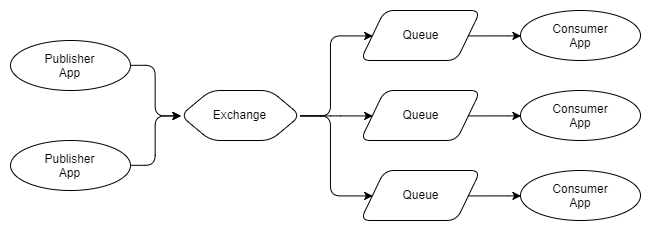
Tổng quan:

* Các Publisher gửi thông điệp đến các Exchange
* Các Exchange định tuyến cácthông điệp đến hàng đợi và các Exchange khác
* RabbitMQ gửi ack đến các Publisher ứng với mỗi thông điệp nhận được
* Các Consumer duy trì kết nối TCP liên tục với RabbitMQ và khai báo (các) hàng đợi mà chúng tiêu thụ
* RabbitMQ push thông điệp đến các Consumer
* Các Consumer gửi ack khi tiêu thụ thành công hoặc thất bại
* Các thông điệp được xóa khỏi hàng đợi khi đã tiêu thụ thành công



Hình 1: Một publisher và một consumer

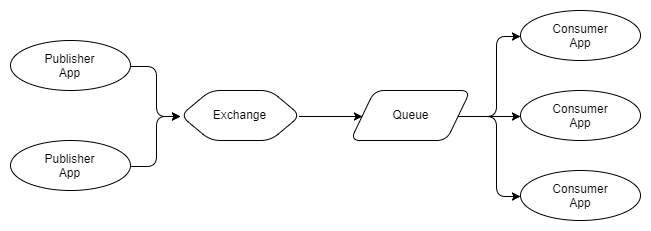
Điều gì sẽ xảy ra nếu chúng ta có nhiều Publisher của cùng một thông điệp? Ngoài ra, nếu chúng ta có nhiều Consumer mà mỗi một trong số chúng đều muốn tiêu thụ mọi thông điệp thì sao?



Hình 2: Nhiều Publisher, nhiều Consumer độc lập

Như chúng ta có thể thấy, các Publisher gửi thông điệp của chúng đến cùng một Exchange, Exchange này định tuyến mỗi thông điệp đến ba queue, mỗi queue trong số đó có một Consumer duy nhất.

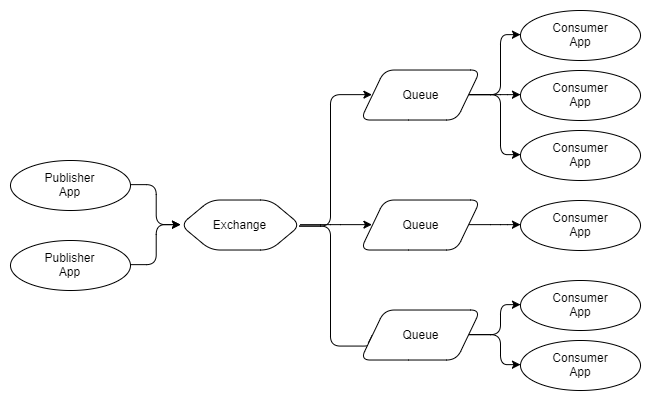
RabbitMQ cũng cho phép các Consumer khác nhau cùng tiêu thụ một queue.



Hình 3: Nhiều Publisher, một queue với nhiều Consumer cạnh tranh nhau

Trong hình 3, ta có ba Consumer cùng tiêu thụ từ một hàng đợi duy nhất. Đây là những Consumer cạnh tranh, chúng cạnh tranh để tiêu thụ các thông điệp của một hàng đợi duy nhất. Chúng ta thường sẽ mong đợi rằng trung bình mỗi Consumer sẽ tiêu thụ một phần ba số thông điệp của hàng đợi này. Chúng ta có thể sử dụng những Consumer cạnh tranh để mở rộng quy trình xử lý thông điệp, và với RabbitMQ việc này rất đơn giản, chỉ cần thêm hoặc xóa Consumer theo yêu cầu nghiệp vụ bài toán. Cho dù bạn có bao nhiêu Consumer cạnh tranh nhau đi nữa, thì RabbitMQ cũng sẽ đảm bảo rằng các thông điệp được gửi đến chỉ một Consumer duy nhất.

Chúng ta có thể kết hợp hình 2 và 3 để có nhiều bộ Consumer cạnh tranh:



Hình 4: Nhiều Publisher, nhiều queue mới các Consumer cạnh tranh

Các mũi tên giữa các Exchange và Queue được gọi là các ràng buộc (binding) và chúng ta sẽ xem xét kỹ hơn những phần trong phần 2 của loạt bài này.

### SỰ BẢO ĐẢM

RabbitMQ đưa ra đảm bảo "tối đa gửi một lần" (at most once delivery) và "ít nhất gửi một lần" (at least once delivery), nhưng không đảm bảo "chính xác gửi một lần" (exactly once delivery). Chúng ta sẽ xem xét kỹ hơn các khả năng đảm bảo gửi thông điệp này trong các bài viết sau.

Thông điệp được gửi theo thứ tự đến hàng đợi đích của chúng (đây chính xác là định nghĩa của hàng đợi). Điều này không đảm bảo rằng trật tự hoàn thành xử lý thông điệp khớp với đúng trật tự của thông điệp khi ta có các Consumer cạnh tranh. Đây không phải là lỗi của RabbitMQ mà là một thực tế dễ hiểu trong việc xử lý một tập hợp các thông điệp theo thứ tự song song. Vấn đề này có thể được giải quyết bằng cách sử dụng Exchange Consistent Hashing như bạn sẽ thấy trong phần tiếp theo về các mẫu và cấu trúc liên kết.

### PUSH VÀ CONSUMER PREFETCH

RabbitMQ push các thông điệp đến cho các Consumer trong một Stream. Có một Pull API nhưng nó có hiệu năng thấp vì mỗi thông điệp yêu cầu một request/response round-trip.

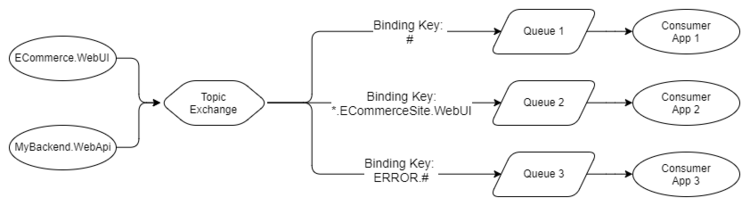
Các Push-based system có thể làm tràn các Consumer nếu như các thông điệp đến hàng đợi nhanh hơn việc Consumer có thể xử lý chúng. Vì vậy, để tránh điều này, mỗi Consumer có thể cấu hình giới hạn prefetch (còn được gọi là giới hạn QoS). Về cơ bản, đây là số lượng thông điệp chưa được gửi ack mà một Consumer có thể có vào một thời điểm bất kì. Điều này đóng vai trò như một công tắc ngắt an toàn khi Consumer bắt đầu hụt hơi và không theo kịp tốc độ tăng tiến số lượng message trong queue.

Tại sao lại là **Push** mà không phải là **Pull**? Trước hết, vì push cho độ trễ thấp. Thứ hai, lý tưởng là khi ta có các Consumer cạnh tranh của một queue duy nhất, ta muốn phân tải đồng đều giữa chúng. Nếu mỗi Consumer pull thông điệp thì tùy thuộc vào số lượng thông điệp mà chúng pull được thì công việc sẽ có thể trở nên phân phối không đồng đều. Việc phân phối thông điệp càng không đồng đều thì độ trê càng nhiều và việc mất thứ tự thông điệp trong thời gian xử lý thông điệp càng cao. Vì lý do đó, Pull API của RabbitMQ chỉ cho phép pull một thông điệp một lần, nhưng điều đó lại ảnh hưởng nghiêm trọng đến hiệu năng. Tổng hợp lại những yếu tố này làm cho RabbitMQ nghiêng về phía cơ chế push. Tuy nhiên, đây lại là một trong những hạn chế về khả năng mở rộng của RabbitMQ. Và nó chỉ được cải thiện bằng cách có thể nhóm các ack lại với nhau.

### Định tuyến

Về cơ bản Exchange là bộ định tuyến của thông điệp đến các Queue và/hoặc các Exchange khác. Để một thông điệp chuyển từ một Exchange sang một Queue hoặc Exchange khác, cần có một ràng buộc (binding). Các Exchange khác nhau sẽ yêu cầu các binding khác nhau. Có bốn loại Exchange và các binding liên quan:

* **Fanout**: Định tuyến đến tất cả các Queue và các Exchange có liên quan đến Exchange đó. Đây là mô hình pub/sub tiêu chuẩn.
* **Direct**: Định tuyến thông điệp dựa trên một **Routing Key** mà thông điệp mang theo cùng với nó, Routing Key này được Publisher thiết lập. Routing Key là một chuỗi kí tự ngắn. Direct Exchange sẽ định tuyến các thông điệp đến các Queue/Exchange có khóa ràng buộc (**Binding Key**) khớp với routing key.
* **Topic**: Định tuyến thông điệp dựa trên một routing key, nhưng cho phép wildcard matching.
* **Header**: RabbitMQ cho phép tùy chỉnh các header được thêm vào các thông điệp. Header Exchange định tuyến các thông điệp theo các value bên trong headerr. Mỗi binding bao gồm header value phù hợp. Nhiều value có thể được thêm vào một binding với **BẤT KÌ** hoặc **TẤT CẢ** các giá trị cần thiết để so sánh.
* **Consistent Hashing**: Đây là một Exchange băm routing key hoặc message header và định tuyến đến chỉ một Queue duy nhất. Điều này hữu ích khi bạn cần đảm bảo thứ tự xử lý với Consumer bị thu nhỏ.



Hình 5: Ví dụ Topic Exchange

Ở trên là một ví dụ về Topic Exchange. Publisher publish các log lỗi với một Routing Key có định dạng là LEVEL.AppName.

* Queue 1 sẽ nhận tất cả thông điệp vì nó sử dụng kí tự # (multi-word wildcard).
* Queue 2 sẽ nhận bất kỳ level log nào của ứng dụng ECommerce.WebUI. Nó sử dụng ký tự \* khi chỉ định level log.
* Queue 3 sẽ thấy tất cả các thông điệp có cấp độ ERROR từ bất kỳ ứng dụng nào. Nó sử dụng ký tự # để chỉ tất cả các ứng dụng.

Với bốn cách định tuyến thông điệp, và cho phép các Exchange định tuyến đến các Exchange khác, RabbitMQ cung cấp một bộ mẫu gửi thông điệp mạnh mẽ và linh hoạt. Tiếp theo, chúng ta sẽ xem qua về các **Dead Letter Exchange**, **Ephemeral(không bền vững) Exchange** và Queue, và ta sẽ bắt đầu thấy được sức mạnh của RabbitMQ.

### **Dead Letter Exchange**

Ta có thể định cấu hình các queue gửi thông điệp đến một Exchange theo các điều kiện sau:

* Queue vượt quá số lượng thư được định nghĩa trong cấu hình.
* Queue vượt quá số byte được định nghĩa trong cấu hình.
* Message Time To Live (TTL) hết hạn. Publisher có thể thiết đặt lifetime của thông điệp, và Queue cũng có thể có TTL cho thông điệp.

Ta tạo một hàng đợi có một binding đến Dead Letter Exchange và những thông điệp này được lưu trữ ở đó cho đến khi hành động tiếp theo được thực hiện.

Giống như với nhiều chức năng của RabbitMQ, Dead Letter Exchange cung cấp thêm các mẫu không được đề cập đến ban đầu. Ta có thể sử dụng TTL của thông điệp và Dead Letter Exchange để implement các delay Queue và retry các Queue.

### **Ephemeral Exchange và Queue**

Các Exchange và Queue có thể được tạo động và được cung cấp các đặc điểm để tự động xóa. Sau một khoảng thời gian nhất định, chúng có thể tự hủy.

### Plug-Ins

Plug-in đầu tiên mà bạn sẽ muốn cài đặt là Management Plug-In, nó cung cấp một HTTP server với giao diện người dùng web và REST API. Nó thực sự dễ cài đặt và cung cấp cho bạn một giao diện người dùng dễ sử dụng để giúp bạn bắt đầu và chạy. Triển khai script thông qua REST API cũng thực sự dễ dàng.

Một số Plug-In khác bao gồm:

* Consistent Hashing Exchange, Sharding Exchange,...
* Các giao thức như STOMP và MQTT
* Web hooks
* Thêm các loại Exchange khác
* Tích hợp SMTP

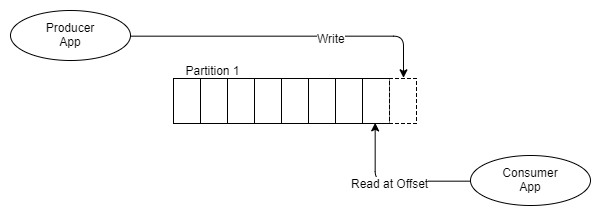
## **Apache Kafka**

Kafka là hệ thống nhân rộng commit log phân tán. Kafka không có khái niệm về một hàng đợi có vẻ lạ lẫm lúc đầu vì nó được sử dụng chính như một hệ thống truyền thông điệp. Hàng đợi đã được đồng nghĩa với hệ thống truyền thông điệp trong một thời gian dài:

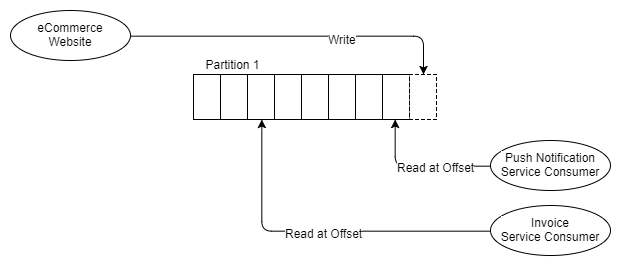
* **Phân tán**: vì Kafka được triển khai dưới dạng một cụm các node, cho cả khả năng chịu lỗi và khả năng mở rộng
* **Nhân rộng**: vì các thông điệp thường được nhân rộng trên nhiều node (máy chủ).
* **Commit Log**: vì các thông điệp được lưu trữ trong phân vùng (partitioned), và chỉ nối thêm log vào cuối, khái niệm về nối thêm log này chính là tính năng nổi bật nhất của Kafka.

Hiểu được log (Topic) và các phân vùng của nó là chìa khóa để hiểu Kafka. Vậy log được phân đoạn khác với một tập hợp các hàng đợi như thế nào? Chúng ta cùng xem hình sau:

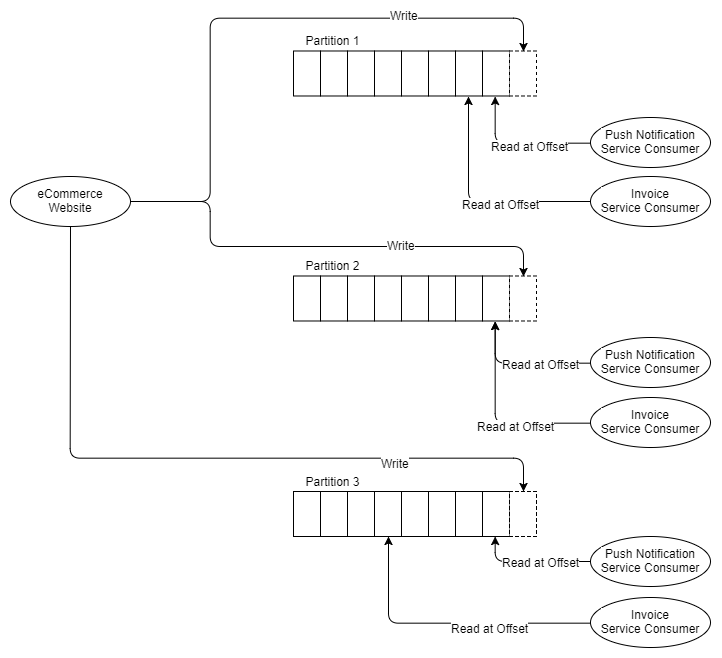
Hình 6: Một producer, một partition, một consumer



* + Thay vì đặt thông điệp trong hàng đợi FIFO và theo dõi trạng thái của thông điệp đó trong hàng đợi như RabbitMQ, Kafka chỉ gắn nó vào log.
  + Thông điệp vẫn được đặt ở đó cho dù nó được tiêu thụ một lần hoặc một ngàn lần. Nó được loại bỏ theo chính sách lưu trữ dữ liệu (thường là một khoảng thời gian).
  + Vậy một Topic được tiêu thụ như thế nào? Mỗi Consumer sẽ tự theo dõi vị trí (offset) của nó ở trong log, nó có một con trỏ trỏ tới thông điệp cuối cùng được tiêu thụ và con trỏ này được gọi là **offset**. Các Consumer duy trì offset này thông qua các client library và phụ thuộc vào phiên bản của Kafka mà offset được lưu trữ hoặc ở trong ZooKeeper hoặc chính Kafka.
  + ZooKeeper là một công nghệ đồng thuận phân tán được sử dụng bởi nhiều hệ thống phân tán cho những thứ như bầu cử leader (leader election). Kafka dựa vào ZooKeeper để quản lý trạng thái của cluster.
  + Điều đáng ngạc nhiên về mô hình log này là nó loại bỏ ngay lập tức nhiều sự phức tạp xung quanh trạng thái gửi thông điệp, và quan trọng hơn cho Consumer là nó cho phép chúng tua và quay trở lại tiêu thụ các thông điệp từ offset trước đó. Ví dụ, hãy tưởng tượng bạn triển khai một service tính toán các hóa đơn booking của khách hàng (invoice-service). Vào một ngày đẹp trời, service này có lỗi và tính toán tất cả các hóa đơn không chính xác trong vòng 24 giờ. Với RabbitMQ, cách tốt nhất bạn sẽ cần làm là phải bằng cách nào đó republish những booking đó và chỉ cho invoice-service biết điều đó. Nhưng với Kafka bạn chỉ cần dịch chuyển offset cho Consumer đó trở lại sau 24 giờ.

Hình 7: Một producer, một partition, hai consumer độc lập

* + Như bạn có thể thấy từ hình trên, hai Consumer độc lập đều sử dụng cùng một phân vùng, nhưng chúng đang đọc từ các offset khác nhau. Điều này có thể được hiểu là invoice-service mất nhiều thời gian xử lý thông điệp hơn push-notification-service hoặc có thể invoice-service đã ngừng hoạt động trong một thời gian và bắt kịp, hoặc có thể đã xảy ra lỗi và offset của nó phải được chuyển lại sau vài giờ.
  + Bây giờ, giả sử invoice-service cần phải được scale-out lên 3 instance vì nó không thể theo kịp với tốc độ tăng tiến của số lượng thông điệp.
    - Với RabbitMQ, ta đơn giản triển khai thêm hai ứng dụng invoice-service tiêu thụ từ hàng đợi.
    - Nhưng với Kafka, nó không hỗ trợ Consumer cạnh tranh trên cùng một phân vùng, chúng ta sẽ phải sử dụng nhiều phân vùng của một Topic. Vì vậy, nếu chúng ta cần ba Consumer cho invoice-service, ta cần ít nhất ba phân vùng:



Hình 8: Ba phân vùng và ba tập consumer

### Phân vùng và nhóm Consumer

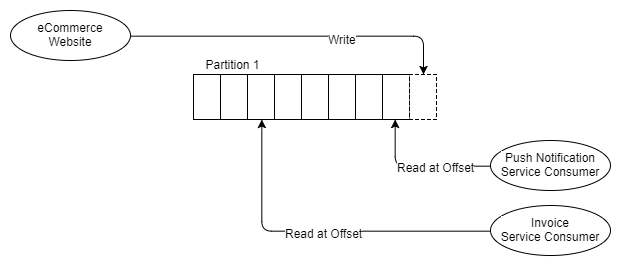
* + Mỗi phân vùng là một tệp dữ liệu riêng biệt đảm bảo thứ tự của thông điệp. Điều quan trọng cần nhớ là: thứ tự của thông điệp chỉ được đảm bảo trong cùng một phân vùng. Điều này có thể dẫn đến một số vấn đề giữa nhu cầu thứ tự của thông điệp và nhu cầu hiệu năng. Một phân vùng không thể hỗ trợ Consumer cạnh tranh, vì vậy invoice-service của ta chỉ có thể có một instance tiêu thụ mỗi phân vùng.
  + Các thông điệp có thể được định tuyến tới các phân vùng theo cân bằng tải hoặc thông qua một hàm băm: hash(message key) % số lượng partition. Sử dụng hàm băm sẽ hữu ích khi chúng ta có thể thiết kế message key sao cho các thông điệp đều thuộc cùng một loại thực thể, ví dụ như booking, có thể luôn nằm trong cùng một phân vùng. Điều này cho phép ta áp dụng được nhiều pattern khi thiết kế, cũng như đảm bảo được thứ tự của thông điệp.
  + Consumer Groups giống như Consumer cạnh tranh của RabbitMQ. Mỗi Consumer trong nhóm là một instance của cùng một ứng dụng và sẽ xử lý một tập con của tất cả các thông điệp trong Topic. Trong khi các Consumer cạnh tranh của RabbitMQ đều tiêu thụ thông điệp từ cùng một hàng đợi, mỗi Consumer trong một Consumer Groups tiêu thụ từ một phân vùng khác nhau của cùng một Topic. Vì vậy, trong các ví dụ trên, ba instance của invoice-service đều thuộc về cùng một Consumer Groups.
  + Ở điểm này, RabbitMQ trông linh hoạt hơn một chút với sự đảm bảo về thứ tự thông điệp trong một hàng đợi và khả năng ít gián đoạn của nó khi đối phó với việc thay đổi số lượng Consumer cạnh tranh. Với Kafka, cách bạn phân vùng log của mình sẽ rất quan trọng.
  + Có một lợi thế nhỏ nhưng quan trọng mà Kafka đã có ngay từ đầu, tuy nhiên RabbitMQ sau này mới có, nó liên quan đến thứ tự thông điệp và xử lý song song:
    - RabbitMQ duy trì trật tự thông điệp trong toàn bộ hàng đợi nhưng không có cách nào để duy trì thứ tự đó trong quá trình xử lý song song của hàng đợi đó.
    - Kafka không thể cung cấp thứ tự thông điệp của toàn bộ Topic (trong các phân vùng khác nhau), nhưng nó cung cấp thứ tự ở cấp phân vùng. Vì vậy, nếu bạn chỉ cần thứ tự của các thông điệp liên quan thì Kafka cung cấp cả việc gửi thông điệp theo thứ tự và xử lý thông điệp theo thứ tự.
    - Hãy tưởng tượng bạn có các thông báo hiển thị trạng thái booking mới nhất của khách hàng, vì vậy bạn luôn muốn xử lý các thông điệp booking một cách tuần tự (theo thứ tự thời gian). Nếu bạn phân vùng theo BookingId, thì tất cả các thông điệp của một booking nhất định sẽ đến cùng một phân vùng (đảm bảo thứ tự thông điệp). Vì vậy, bạn có thể tạo một số lượng lớn các phân vùng, làm cho quá trình xử lý của bạn có tính đông thời cao và cũng nhận được sự đảm bảo mà bạn cần cho thứ tự của thông điệp.
  + Khả năng này cũng tồn tại trong RabbitMQ thông qua Consistent Hashing Exchange, nó phân phối thông điệp trên các hàng đợi theo cùng cách. Mặc dù Kafka thực thi xử lý theo thứ tự này bởi thực tế chỉ có một Consumer trong mỗi Consumer Groups có thể tiêu thụ một phân vùng duy nhất. Trong khi đó, với RabbitMQ bạn vẫn có thể có nhiều Consumer tiêu thụ cạnh tranh từ cùng một hàng đợi và bạn sẽ phải thực hiện nhiều công sức hơn nếu muốn đảm bảo điều đó không xảy ra.

### PUSH VS PULL

* + RabbitMQ sử dụng push model và ngăn chặn các Consumer áp đảo thông qua giới hạn prefetch được cấu hình bởi Consumer. Điều này tốt cho việc truyền thông điệp với độ trễ thấp và hoạt động tốt cho kiến ​​trúc dựa trên hàng đợi của RabbitMQ.
  + Kafka sử dụng một pull model để Consumer tự yêu cầu thông điệp từ một offset đã cho.Pull model có ý nghĩa đối với Kafka do mô hình gồm các phân vùng của nó, Vì Kafka đảm bảo thứ tự thông điệp trong một cùng phân vùng không có Consumer cạnh tranh, ta có thể tận dụng việc này để gửi các thông điệp hiệu quả hơn, cung cấp cho ta thông lượng cao hơn.
  + Điều này không có ý nghĩa nhiều đối với RabbitMQ vì ý tưởng là chúng ta muốn cố gắng phân phối thông điệp một cách nhanh nhất có thể để đảm bảo công việc được xử lý song song, đồng đều và các thông điệp được xử lý gần với thứ tự mà chúng đến trong hàng đợi.

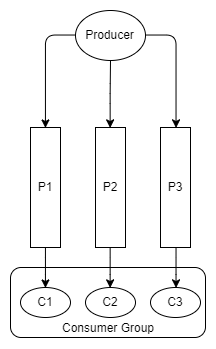
### Publish/Subscribe

* + Kafka hỗ trợ pub/sub cơ bản với một số pattern liên quan đến thực tế đó là log và có phân vùng. Các Publisher nối thêm thông điệp vào cuối phân vùng log và Consumer có thể được định vị với offset của chúng ở bất kỳ đâu trong phân vùng.



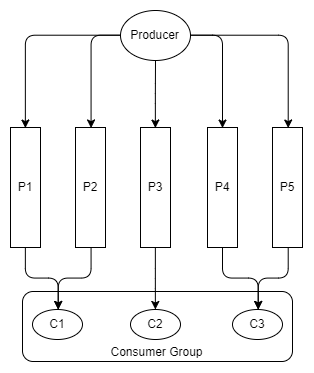
Hình 9: Các Consumer với các offset khác nhau

* + Kiểu sơ đồ này không dễ dàng để diễn giải nhanh khi có nhiều phân vùng và Consumer Groups, vì vậy đối với phần còn lại của sơ đồ cho Kafka chúng ta sẽ sử dụng kiểu sau:



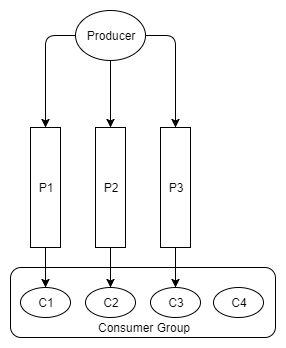
Hình 10: Một Producer, ba Partition và một Consumer Group gồm ba Consumer

* + Chúng ta không nhất thiết phải có số lượng Consumer trong Consumer Groups bằng với số lượng phân vùng:



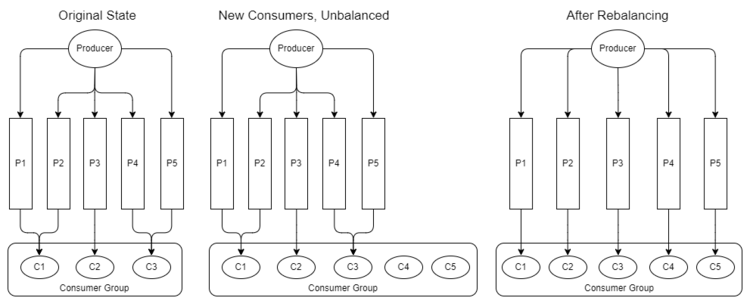
Hình 11: Một vài Consumer đọc nhiều hơn một Partition

* + Các Consumer trong một Consumer Groups sẽ điều phối việc tiêu thụ phân vùng, đảm bảo rằng một phân vùng không được tiêu thụ bởi nhiều hơn một Consumer của cùng một Consumer Groups.
  + Tương tự như vậy, nếu chúng ta có số lượng Consumer nhiều hơn số lượng phân vùng, các Consumer mới được thêm vào sẽ ở chế độ chờ - dự bị.



Hình 12: Một Consumer nhàn rỗi

* + Sau khi thêm và loại bỏ các Consumer, Consumer Groups có thể trở nên không cân bằng. Việc tái cân bằng lại các Consumer càng đồng đều sẽ càng tốt trên các phân vùng.



Hình 13: Thêm mới Consumer sẽ yêu cầu tái cân bằng

* + Việc tái cân bằng được tự động kích hoạt sau khi:
    - Một Consumer tham gia vào một Consumer Groups
    - Một Consumer rời khỏi một Consumer Groups (nó shutsdown hoặc được xem là đã chết)
    - Phân vùng mới được thêm vào
  + Việc tái cân bằng sẽ gây ra một khoảng thời gian trễ ngắn vì các Consumer ngừng đọc thông điệp và được chỉ định cho các phân vùng khác nhau. Vào thời điểm này, bất kỳ trạng thái bộ nhớ nào được duy trì bởi Consumer cũng có thể không còn hợp lệ.

### Log Compaction

* + Các chính sách lưu trữ dữ liệu tiêu chuẩn thường là các chính sách dựa trên thời gian lưu trữ và dung lượng lưu trữ.: Ví dụ như lưu trữ đến tuần cuối cùng của thông điệp hoặc dung lượng lưu trữ lên đến 50GB chẳng hạn. Nhưng có một loại chính sách lưu trữ dữ liệu khác tồn tại đó là **Log Compaction**. Khi một một log được nén lại thì chỉ có thông điệp mới nhất cho mỗi message key là được giữ lại, phần còn lại sẽ bị xóa.
  + Hãy tưởng tượng rằng chúng ta nhận được một thông điệp có chứa trạng thái booking mới nhất của người dùng. Với mỗi lần thay đổi được thực hiện đối với booking, một sự kiện mới sẽ được tạo ra với trạng thái mới nhất của booking đó. Topic có thể có một vài thông điệp cho một booking đại diện cho các trạng thái của booking đó kể từ khi nó được tạo ra. Sau khi Topic được nén gọn, chỉ thông điệp mới nhất liên quan đến booking đó sẽ được giữ lại.
  + Tùy thuộc vào khối lượng booking và kích thước dung lượng của mỗi booking, mà về lý thuyết thì bạn có thể lưu trữ mãi mãi tất cả các booking bên trong Topic. Bằng cách thực hiện nén Topic định kì, chúng ta đảm bảo rằng chúng ta chỉ lưu trữ một thông điệp cho mỗi booking.
  + Tính năng Log Compaction cho phép ta thực hiện một số các pattern khác nhau, chúng ta sẽ khám phá chúng trong các phần sau.

## **Các thông tin thêm về Thứ tự của các Message**

* + Ta đã đề cập đến việc cả RabbitMQ và Kafka có thể scale-out và duy trì thứ tự thông điệp, nhưng Kafka sẽ thực hiện dễ dàng hơn rất nhiều. Với RabbitMQ, chúng ta bắt buộc phait sử dụng Consistent Hashing Exchange và thực hiện logic thủ công trong Consumer bằng cách sử dụng một service đồng thuận phân tán như **ZooKeeper** hoặc **Consul**.
  + Tuy nhiên, RabbitMQ có một khả năng thú vị mà Kafka không có đó là cho phép các Subscriber sắp xếp các nhóm sự kiện tùy ý.
  + Các ứng dụng khác nhau không thể chia sẻ một Queue bởi vì chúng sẽ cạnh tranh nhau để tiêu thụ các thông điệp. Chúng cần các hàng đợi riêng. Điều này cho phép các ứng dụng tự do trong việc cấu hình các hàng đợi sao cho phù hợp nhất. Chúng có thể định tuyến nhiều loại sự kiện từ nhiều Topic đến các hàng đợi. Điều này cho phép các ứng dụng duy trì được thứ tự của các sự kiện có liên quan. Các nhóm event có thể được cấu hình kết hợp lại theo các cách khác nhau đối với từng ứng dụng.
  + Điều này lại là không thể với một hệ thống thông điệp dựa vào log giống như Kafka, vì log là tài nguyên được chia sẻ. Nhiều ứng dụng sẽ đọc từ cùng một log. Vì vậy, việc thực hiện nhóm các sự kiện liên quan lại với nhau vào trong một Topic duy nhất sẽ phải được thực hiện ở mức kiến ​​trúc hệ thống tông quan hơn.

# Cấu trúc của RabbitMQ và các messaging pattern khi sử dụng RabbitMQ

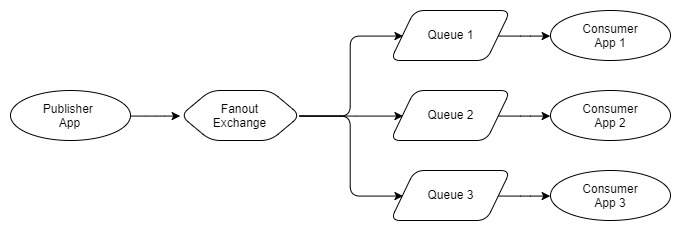
* + Trong phần này chúng ta sẽ không đề cập chi tiết đến tầng thấp trong các giao thức, mà thay vào đó sẽ tập trung vào các mô hình ở tầng cao hơn và các cấu trúc liên kết thông điệp có thể đạt được trong RabbitMQ. Trong phần tiếp theo, chúng ta sẽ làm tương tự cho Apache Kafka.
  + Đầu tiên chúng ta hãy xem qua các khối nền tảng hoặc các định tuyến cơ bản của RabbitMQ:
    - Các loại exchange và liên kết
    - Queue
    - Dead letter exchange (DLX)
    - Ephemeral exchange và queue
    - Các **Alternate Exchange**
    - Hàng đợi ưu tiên
  + Sau đó, chúng ta sẽ kết hợp tất cả chúng thành một tập hợp các mẫu ví dụ.

## 

## Các loại định tuyến cơ bản của RabbitMQ

### *Fanout Exchanges*

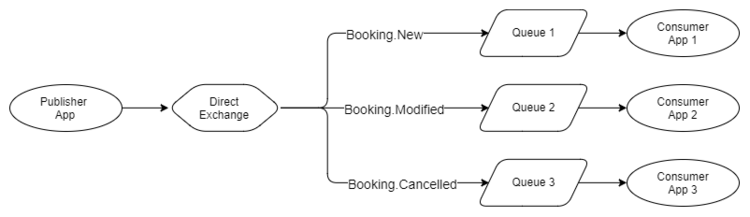
* + Những exchange này cung cấp cấu trúc liên kết đăng ký xuất bản điển hình. Một thông điệp được gửi đến một Fanout Exchange sẽ được phát tới cho tất cả các hàng đợi và các exchange có liên kết đến exchange này.



* + Trong sơ đồ trên, mỗi consumer độc lập với những consumer khác và nhận được các bản sao riêng của tất cả các thông điệp. Để mở rộng ứng dụng Consumer App 1, nhiều phiên bản ứng dụng đó sẽ cần được triển khai, tiêu thụ từ cùng một Queue 1.
  + **Fanout Exchange** là exchange nhanh nhất trong các exchange vì chúng không cần phải kiểm tra bất kỳ routing key nào hoặc kiểm tra header của thông điệp. Mặc dù một exchange có thể gửi một thông điệp duy nhất cho nhiều hàng đợi, trong thực tế nó không thường xuyên sao chép các thông tin. Nó có thể lưu thông điệp vào cơ sở dữ liệu **Mnesia** và chỉ cần đăng ký một con trỏ tới thông điệp trong mỗi hàng đợi.

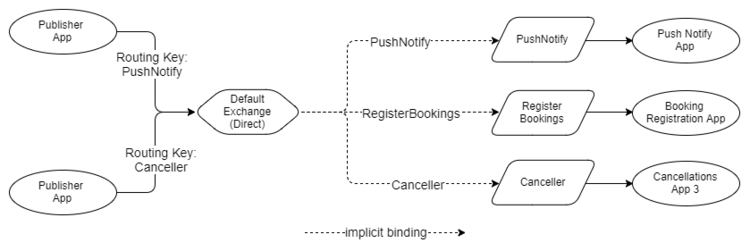
### *Direct Exchanges và Default Exchange*

* + **Direct Exchanges** định tuyến thông điệp bằng cách sử dụng Routing Key của thông điệp. Routing key được thiết đặt bởi các publisher tương ứng với thông điệp. Chúng là các chuỗi gồm nhiều từ được phân cách với nhau bằng dấu chấm. Ví dụ như, "booking.new", "booking.modified" và "booking.cancelled".
  + Liên kết giữa một Queue hoặc một Exchange với một **Direct Exchange** chứa một binding key, giá trị này sẽ được so sánh chính xác tuyệt đối.



Hình 2: Direct exchanges định tuyến bằng cách so sánh chính xác Routing Key và Binding Key

* + **Direct exchange** là loại exchange nhanh thứ hai vì chúng chỉ thực hiện so sánh chính xác các chuỗi với nhau.
  + Có một loại exchange đặc biệt được gọi là Default exchange, nó cũng là một loại Direct echange. **Default exchange** có mối liên kết ngầm với tất cả các hàng đợi trong virtual host của nó. Mối liên kết ngầm tới mỗi queue sẽ có binding key là tên của các queue đó. Điều này nghĩa là bạn có thể gửi thông điệp trực tiếp đến các queue bằng tên của queue tương ứng.



Hình 3: Default exchange có mối liên kết ngầm đến mỗi queue

* + Điều này có thể hữu ích nếu publisher muốn chọn chính xác consumer mà nó muốn xử lý thông điệp của nó, thay vì dựa vào các liên kết được cấu hình bởi các consumer. Thông thường, chúng ta sẽ muốn tách rời hoàn toàn publisher và subscriber với nhau, đây là khả năng mà các loại exchange khác đều có thể cung cấp. Nhưng trong trường hợp bạn muốn gửi thông điệp theo kiểu point-to-point thì Default exchange sẽ cung cấp cho bạn khả năng này.

### Topic Exchange

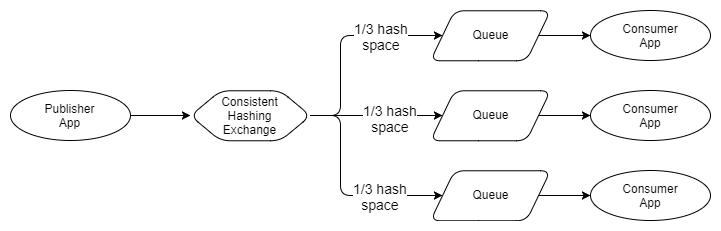
* + Các exchange này cũng định tuyến bằng routing key, nhưng các **Topic Exchange** cung cấp việc sử dụng hai loại ký tự đại diện trong Khóa Binding.
  + Ký tự \* đại diện khớp với một từ trong routing key. Ví dụ: routing key "booking.new" có hai từ.
  + Ký tự # phù hợp với bất kỳ số lượng từ.
  + Ví dụ: giả sử chúng ta có các routing key sau:
    - booking.new
    - booking.modified
    - booking.cancelled
    - extras.car.new
    - extras.car.modified
    - extras.car.removed
    - extras.hotel.new
    - extras.hotel.modified
    - extras.hotel.removed
  + Chúng ta có thể tạo các liên kết với các khóa liên kết sau:
    - *booking.new* - so sánh chính xác từ khóa
    - *extras.\*.modified* - tất cả các sửa đổi đối với tính năng bổ sung khi đặt phòng (xe hơi hoặc khách sạn)
    - *extras.#* - tất cả các tính năng bổ sung
    - *#.new* - tất cả các đặt phòng mới và tính năng bổ sung
  + Với thiết kế cẩn thận các routing key và khóa liên kết, chúng ta có thể thêm các routing key mới mà không cần cập nhật các ràng buộc hiện có, làm cho hệ thống trở nên mạnh mẽ khi đối mặt với sự thay đổi.
  + **Topic Exchange** cho phép bạn định cấu hình một exchange duy nhất cho một ứng dụng để gửi thông điệp đến, sử dụng định tuyến để đảm bảo rằng các thông điệp đến đúng consumer. Điều này giúp đơn giản hóa cấu hình và triển khai các ứng dụng producer và consumer.
  + Lưu ý rằng **Topic Exchange** chậm lại khi số lượng ràng buộc tăng lên.

### 1.4 Header Exchange

* + Đây là exchange có tính năng mạnh mẽ nhất, nhưng cũng chậm nhất trong các loại exchange. Thực tế này cần phải được tính đến vì bạn có thể có vấn đề với mở rộng hệ thống với loại exchange này. **Header exchane** bỏ qua routing key và thay vào đó phân tích các tiêu đề của thông điệp. Mỗi liên kết với một **Header Exchange** có thể bao gồm nhiều đối sánh tiêu đề mà ANY hoặc ALL bắt buộc phải khớp.
  + Giả sử ứng dụng của bạn publish lên một exchange một tập hợp các thông điệp khác nhau tạo thành nhật ký thay đổi thời gian thực cho phép tích hợp với các hệ thống khác. Mỗi thông điệp có các tiêu đề thư sau:
    - entity.type*(đặt chỗ, hành khách, hành lý, thú cưng)*
    - change.type*(mới, sửa đổi, hủy bỏ, xóa, di chuyển)*
    - agent.id
    - client.id
  + Chúng ta có thể tạo các liên kết sau:
    - *entity.type=booking*, *change.type=cancelled*, *x-match=all*. Tôi muốn tất cả các thông điệp đặt phòng bị hủy.
    - *entity.type=passenger*, *x-match=all*. Tôi muốn tất cả các thông điệp hành khách.
    - *entity.type=pet*, *change.type=new*, *x-match=all*. Tôi muốn tất cả các thông điệp thú cưng mới được thêm vào.
    - *agent.id=2*, *client.id=1001*, *x-match=any*. Tôi muốn tất cả các thông điệp liên quan đến đại lý du lịch cụ thể hoặc khách hàng cuối.
  + Như bạn có thể thấy, **Header Exchange** khá mạnh mẽ.

### 1.5 Consistent Hashing Exchange

* + **- Consistent Hashing Exchange** cho phép chúng ta phân vùng một hàng đợi thành nhiều hàng đợi và phân phối thông điệp giữa chúng thông qua việc băm routing key, tiêu đề thư hoặc thuộc tính thông điệp.



*Hình 4:*

* + Điều này cung cấp cho chúng ta các mẫu như bảo đảm xử lý theo thứ tự và dữ liệu cục bộ, hai pattern bạn sẽ thấy bên dưới trong phần các pattern.
  + Có một số vấn đề với **Consistent Hashing Exchange** mặc dù. Đầu tiên, RabbitMQ không giúp bạn điều phối ~~khách hàng~~ (thông điệp) của mình qua các hàng đợi được phân vùng như Kafka. Vì vậy, đó là xuống để bạn quản lý bằng cách nào đó. Kafka cung cấp cho bạn điều này ra khỏi hộp.
  + Các vấn đề tiềm năng khác là:
    - Thứ bạn băm (routing key, message header hoặc các thuộc tính) không có qui chuẩn nào để tạo phân phối đồng đều. Nếu bạn chỉ có bốn giá trị khác nhau thì có thể gặp rủi ro và tất cả đi đến một hàng đợi.
    - Nếu có tương đối ít hàng đợi thì có thể phân phối không đồng đều trở lại.
  + Các hệ thống phân tán khác giải quyết điều này bằng cách sử dụng khái niệm các nút ảo. Ví dụ, để ngăn phân phối mất cân bằng, ~~Rịa~~ và Cassandra có các nút ảo có số lượng lớn hơn nhiều so với các nút vật lý và chúng phân phối các nút ảo này qua các nút vật lý.
  + Bằng cách đó, chúng có được phân phối tốt hơn khi một cụm có tương đối ít nút vật lý. RabbitMQ không có khái niệm này vì vậy hãy chú ý đến việc phân phối thông điệp.

### Dead Letter Exchange

* + Chúng ta có thể định cấu hình hàng đợi để đẩy một thông điệp và gửi nó đến một exchange được cấu hình theo một trong ba điều kiện:
    - Hàng đợi đã đạt đến giới hạn số lượng thông điệp. Thông báo ở đầu hàng đợi (thông điệp cũ nhất) được đẩy ra và gửi đến **Dead Letter Exchange** đã được cấu hình (DLX). Vì vậy, khi một thông điệp mới đến một hàng đợi đầy đủ, về cơ bản nó sẽ loại bỏ thông điệp cũ nhất và được thêm vào hàng đợi một cách an toàn.
    - Hàng đợi đã đạt đến giới hạn kích thước (byte). Một lần nữa, thông điệp cũ nhất bị đẩy ra.
    - Hàng đợi đã được cấu hình với một *giới hạn tồn tại của thông điệp* (TTL) và một thông điệp đã đạt đến giới hạn đó. Một thông điệp đã được cấu hình với TTL của chính nó và nó đã đạt đến khoảng thời gian đó trong hàng đợi.
  + Thông điệp chỉ bị coi là đã chết khi đứng đầu hàng đợi. Vì vậy, các thông điệp đã vượt qua TTL của chúng chỉ được chuyển tiếp đến DLX khi chúng đến đầu hàng đợi. Điều này rất quan trọng!
  + DLX chỉ là một exchange thông thường, bạn có thể tạo một trong bốn loại và liên kết bất kỳ hàng đợi hoặc exchange khác với nó.
  + Chức năng *dead letter* của RabbitMQ không chỉ cung cấp một lối thoát cho các thông điệp trong tình trạng bị mất. Nó có thể được sử dụng để retry và delay hàng đợi như chúng ta sẽ thấy trong phần ví dụ một số pattern.

### 1.7 Ephemeral Exchanges và Queue

* + Exchange có thể được cấu hình để tự động xóa tất cả các liên kết hàng đợi đã được gỡ bỏ. Các liên kết hàng đợi có thể được loại bỏ bằng cách chỉ cần loại bỏ các liên kết hoặc loại bỏ hàng đợi.
  + Hàng đợi có thể được cấu hình để tự động xóa sau khi tất cả consumer đã ngừng sử dụng hàng đợi. Điều này có thể là do consumer đã hủy đăng ký hoặc kênh đã đóng.
  + Hàng đợi có thể được cấu hình thành **hàng đợi độc quyền**. Điều này có nghĩa là chỉ consumer đã khai báo hàng đợi mới có thể tiêu thụ nó và một khi consumer hủy hoặc đóng kênh, hàng đợi sẽ tự động xóa.
  + Hàng đợi có thể được cấu hình với một **hàng đợi TTL**. Khi hàng đợi không được sử dụng trong khoảng thời gian TTL, nó sẽ bị xóa. Không sử dụng có nghĩa là không có consumer hoạt động đăng ký.
  + **Ephemeral Exchanges** và Queue có thể được sử dụng cho các mẫu như delay queue, retry queue và reply-to queue, như chúng ta sẽ thấy trong phần ví dụ về các pattern.

### 1.8 Alternate Exchange - Exchange thay thế

* + **Mỗi** exchange có thể được cấu hình với một **Alternate Exchange**. Khi một exchange không thể định tuyến một thông điệp vì không có liên kết hoặc không có liên kết nào khớp với thông điệp, thì exchange đó sẽ định tuyến thông điệp đến **Alternate Exchange** của nó.
  + Điều này cung cấp cho chúng ta một cách không làm mất các thông điệp có thể bị mất do routing key xấu hoặc cấu trúc liên kết định tuyến xấu. Nhưng nó cũng cho phép tạo ra các pattern định tuyến mới mà bốn loại echange không cung cấp. Chúng ta sẽ thấy trong ví dụ về các pattern cách mà **Alternate Exchange** có thể được sử dụng cho các tình huống định tuyến khác nhau.

### 1.9 Priority Queue - Hàng đợi ưu tiên

* + Thông điệp có thể được cấu hình với các mức độ ưu tiên khác nhau, nên sử dụng tối đa 10 cấp độ. Khi một hàng đợi được khai báo, nó có thể được khai báo như một hàng đợi ưu tiên. Nếu publisher đặt mức độ ưu tiên cho thông điệp thì vị trí của nó trong hàng đợi ưu tiên sẽ được xác định theo mức ưu tiên đó. Thông điệp có mức độ ưu tiên cao hơn sẽ được chuyển tiếp về phía trước những thông điệp có mức độ ưu tiên thấp hơn. Vì vậy, nếu một hàng đợi có 1000 thông điệp ưu tiên thấp và một thông điệp ưu tiên cao đến, nó sẽ được đặt ở đầu hàng đợi ngay lập tức.
  + Có hai cân nhắc quan trọng cần tính đến với các hàng đợi ưu tiên. Nếu một hàng đợi ưu tiên bị đầy và có một thông điệp ưu tiên cao ở đầu hàng đợi, khi có một thông điệp ưu tiên thấp đến, nó sẽ loại thông điệp ưu tiên cao. Thông điệp ưu tiên thấp sẽ được duy trì an toàn ở trong hàng đợi và thông điệp ưu tiên cao sẽ được gửi đến DLX.
  + Tương tự như vậy, các thông điệp có mức độ ưu tiên thấp hơn có thể bị kẹt vì chúng có thể nằm phía sau các thông điệp có mức độ ưu tiên cao hơn. Ngay cả việc thiết lập một thông điệp TTL cũng không giúp ích gì trong trường hợp này vì thông điệp chết luôn xảy ra ở đầu hàng đợi.
* Vì vậy, hãy sử dụng hàng đợi ưu tiên một cách cẩn thận.

### 1.10 CC và BCC

* + Các publisher của thông điệp có thể thêm các routing key bổ sung trong hai header của thông điệp (CC và BCC). Chúng hoàn toàn giống như email. Các routing key trong các header CC và BCC định tuyến thông điệp giống như routing key của thông điệp tiêu chuẩn. Tiêu đề BCC sẽ bị xóa khỏi thông điệp trước khi gửi.

### Khai báo các Exchange, Hàng đợi và Liên kết

* + Bản thân các ứng dụng có thể khai báo các exchange, hàng đợi và liên kết mà chúng muốn.
  + **Kafka** yêu cầu một số quản lý tập trung do các quyết định về phân vùng ảnh hưởng đến tất cả consumer.
  + **RabbitMQ** linh hoạt hơn trong vấn đề này và các ứng dụng có thể quản lý các thành phần RabbitMQ của riêng chúng mà không lo ảnh hưởng đến các ứng dụng khác. Các cấu trúc dựa trên quy ước là rất tuyệt khi chúng ta có thể sử dụng các quy ước đơn giản trong các ứng dụng và xây dựng các cấu trúc liên kết định tuyến tinh vi. Tuy nhiên, nếu khả năng mở rộng và hiệu năng trở thành mối quan tâm chính, thì bạn có thể cần phải cẩn thận thiết kế cấu trúc liên kết định tuyến của mình thành chuyên dụng hơn.

### Virtual Host

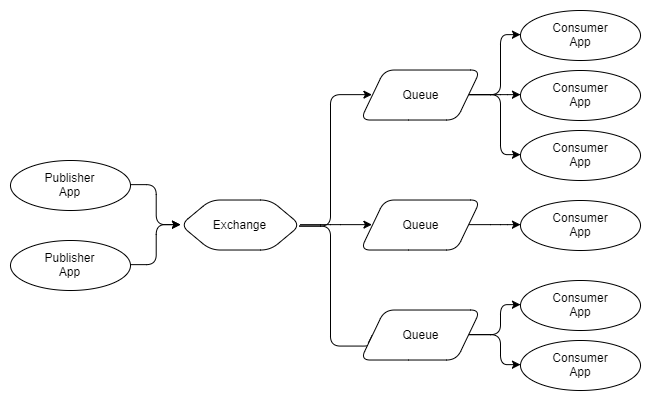
Một máy chủ ảo là một thùng chứa logic của các exchange và hàng đợi. Chúng có thể được sử dụng để kiểm soát truy cập vào các exchange và hàng đợi. Định tuyến máy chủ ảo chéo là không thể, vì vậy tất cả các ví dụ trong phần này giả định một máy chủ ảo bao gồm.

## **Ví dụ về một số pattern khi sử dụng RabbitMQ**

### 1 - Simple Broadcast với Fanout Exchange

Sử dụng Fanout Exchange để truyền tất cả thông điệp cho tất cả consumer.

Hình 5:

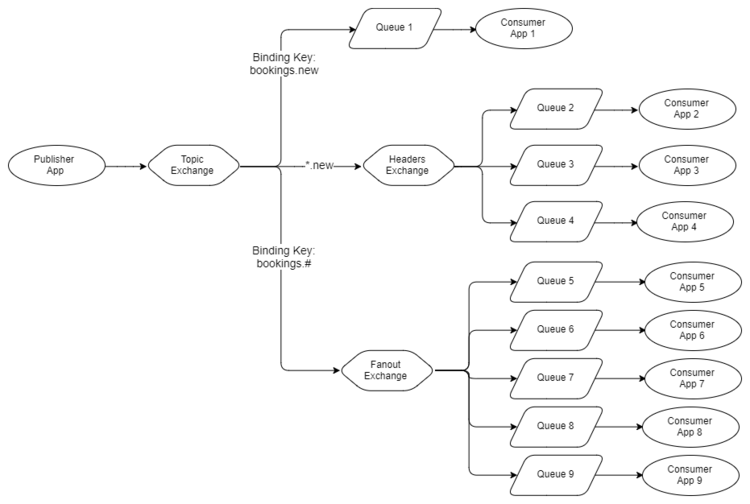


### 2 - Nhiều lớp Exchange

Để giảm chi phí định tuyến, phương pháp phân lớp có thể được áp dụng. Trong ví dụ dưới đây, chúng ta ban đầu định tuyến dựa trên một số lượng nhỏ các routing key hữu hạn đến các exchange khác. Mỗi ràng buộc cho một chủ đề làm tăng chi phí exchange đó. Trong trường hợp này, chúng ta có thể định tuyến tất cả các thông điệp đặt phòng đến một **Fanout Exchange**, nơi nó có thể được phát sóng hiệu quả đến tất cả consumer quan tâm.

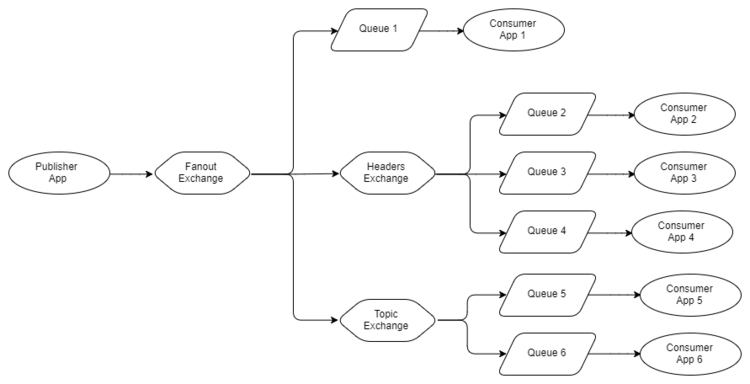
Tương tự, những consumer muốn lọc dựa trên các tiêu đề thư của một loại thông điệp nhất định có thể định tuyến thông điệp hiệu quả đến **Header Exchange** tốn kém hơn trong đó việc định tuyến dựa trên tiêu đề thư được thực hiện qua một tập hợp con các thông báo đi qua **Topic Exchange** .

Hình 6:



Nếu tất cả các exchange xuôi dòng muốn có thể định tuyến tất cả các thông điệp, điểm vào có thể là một Fanout thay thế.

Hình 7:



### 3 - Hệ thống định tuyến email với Header Exchange

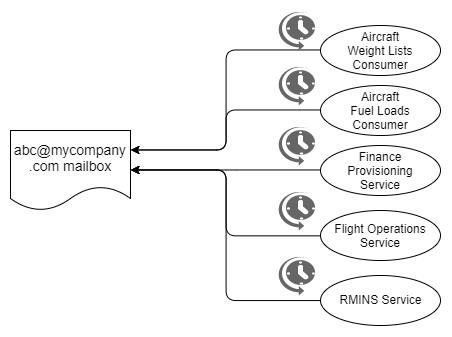
Định tuyến email không phải là một mẫu chung nhưng nó thể hiện sức mạnh của **Header Exchange**. Ví dụ này cũng cho thấy cách bạn có thể giảm sự phụ thuộc vào scheduler.

Giả sử chúng ta là một hãng hàng không và chúng ta làm việc với một đối tác tên là ABC thực hiện bảo dưỡng máy bay trên máy bay của chúng ta. Mỗi ngày hệ thống của chúng gửi cho chúng ta các email chứa thông tin bên trong email hoặc trong tệp đính kèm. Vâng chào mừng bạn đến với thế giới tích hợp thông qua SMTP, đó là sự thật.

Bạn có năm ứng dụng cần cập nhật hệ thống nội bộ với các tệp dữ liệu khác nhau mà ABC gửi cho chúng ta hàng ngày. Ví dụ, bộ phận tài chính cần biết tình trạng thành phần máy bay để tạo ra các mô hình dự đoán về chi phí trong tương lai để ngân sách cần thiết được cung cấp và hạch toán.

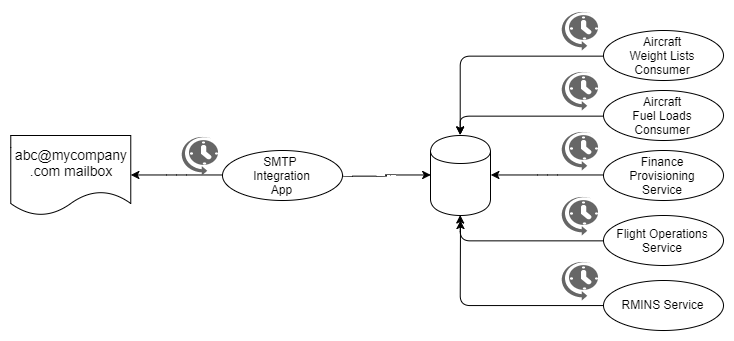
Khi tất cả năm ứng dụng đọc trực tiếp từ hộp thư, chúng ta không còn có thể dựa vào trạng thái đọc. Mỗi ứng dụng cần theo dõi những gì chúng đã đọc trước đó, bỏ qua các email mà chúng không quan tâm và được lập lịch để chạy mỗi X phút hoặc giờ. chúng ta phải thực hiện logic đọc hộp thư nhiều lần.

Hình 8:



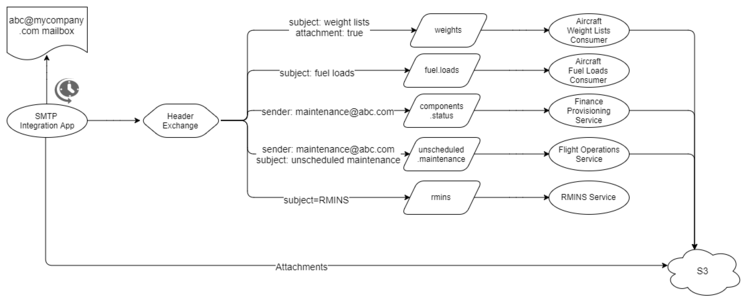
Thay vào đó, chúng ta có thể có một ứng dụng duy nhất chịu trách nhiệm đọc hộp thư ghi tất cả email và tệp đính kèm của chúng vào cơ sở dữ liệu. Sau đó, mỗi ứng dụng cần đọc từ cơ sở dữ liệu đó. Một lần nữa, mỗi ứng dụng phải theo dõi những email đã đọc, đó là đoạn code lặp đi lặp lại. Các ứng dụng này cũng cần được lên lập lịch bằng một scheduler

Hình 9:



Tùy chọn tốt hơn là có một ứng dụng được lên lịch duy nhất đọc từ hộp thư và gửi email dưới dạng thông điệp trên RabbitMQ, đến một **Header Exchange**. Tệp đính kèm có thể được lưu vào cơ sở dữ liệu hoặc dịch vụ đám mây như S3, chỉ với khóa đính kèm trong thông điệp. Các thuộc tính email như địa chỉ người gửi, địa chỉ người nhận, cc, chủ đề đều được thêm dưới dạng tiêu đề thư. Sau đó, mỗi ứng dụng chỉ cần tạo các ràng buộc phù hợp với các email mà chúng muốn tiêu thụ. Các ứng dụng không cần lịch trình khi chúng được đẩy các email từ RabbitMQ.

Hình 10:

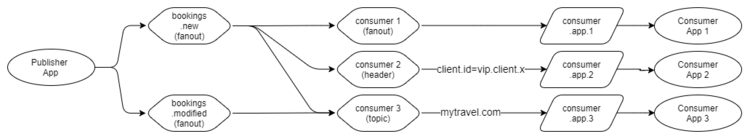


Hạn chế của **Header Exchange** là bạn chỉ có thể thực hiện các kết quả khớp chính xác. Điều này không loại trừ **Header Exchange** khá thường xuyên không may.

### 4 - Public Message Exchange, Private Consumer Exchange

Đây là một mô hình định tuyến dựa trên quy ước linh hoạt. Cấu trúc liên kết độc đáo có thể khó quản lý khi qui mô của chúng lớn hơn. Tôi có xu hướng thích các cấu trúc liên kết dựa trên quy ước vì chúng dễ quản lý. Trong pattern này, publisher của thông điệp khai báo **Fanout Exchange** dựa trên tên loại thông điệp. Consumer khi khởi động khai báo hàng đợi của riêng chúng và đối với mỗi thông điệp chúng tiêu thụ, chúng khai báo **private exchange** của chúng và liên kết nó với **message exchange** mà chúng muốn đăng ký. Sử dụng logic đơn giản này, các publisher và consumer tự động tạo ra tất cả các cơ sở hạ tầng hàng đợi mà không biết về nhau hoặc tác động lẫn nhau theo bất kỳ cách nào.

Hình 11:



Trong sơ đồ trên, publisher phát đi hai loại thông điệp: new booking và modified booking. Để định tuyến linh hoạt, nó thiết đặt kênh bán hàng là routing key (kênh bán hàng có thể là trang web chính, trang web so sánh, công ty du lịch, v.v.) và thêm một số dữ liệu thú vị khác trong tiêu đề của thông điệp.

Publisher đơn giản chỉ cần phát đi mỗi thông điệp đến exchange tương ứng của nó. Mỗi consumer đều có hàng đợi và exchange riêng. Nó liên kết exchange riêng của chính nó với messaging exchange này. Trong ví dụ của chúng ta, consumer app 1 muốn tất cả new booking. Consumer App 2 muốn tất cả new booking của một khách hàng cụ thể rất quan trọng. Consumer app 3 muốn tất cả new booking và modified booking liên quan đến MyTravel.com - nơi bán các đặt chỗ với tư cách là người bán bên thứ 3.

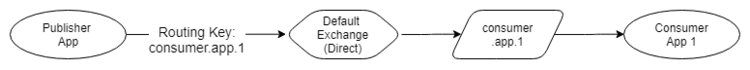
Pattern này tạo ra một cấu trúc liên kết tự quản lý trong đó yêu cầu làm sạch duy nhất là khi consumer bị xóa vĩnh viễn khỏi hệ thống. Triển khai và phát triển được đơn giản hóa khi tất cả các ứng dụng tạo ra các exchange, hàng đợi và liên kết RabbitMQ cần thiết mà chúng cần để giảm gánh nặng cho nhóm vận hành và các đường dẫn triển khai.

Một lợi ích khác của việc cung cấp cho mỗi consumer exchange riêng của chúng là các đội hỗ trợ có thể đặt "theo dõi" để xem tất cả các thông điệp được sử dụng bởi một ứng dụng. Bạn có thể tạo một hàng đợi và liên kết nó với exchange riêng của một ứng dụng consumer và nhận các bản sao của tất cả các thông điệp mà nó nhận được. Điều này cũng có thể được sử dụng để ghi log của một consumer nhất định.

### 5 - Point-to-point Messaging

Chúng ta có thể bỏ qua các tùy chọn định tuyến khác nhau và gửi thông điệp trực tiếp đến hàng đợi theo tên. Gửi thông điệp đến **Default Exchange**, với tên của hàng đợi là routing key và nó sẽ được chuyển thẳng đến hàng đợi.

Hình 12:

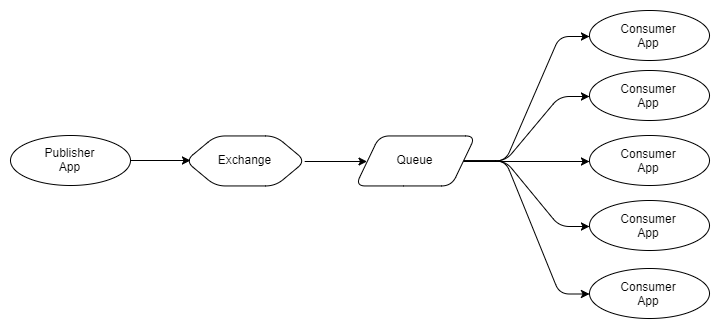


Điều này hữu ích khi publisher muốn kiểm soát consumer nào xử lý thông điệp, thay vì dựa vào định tuyến trong đó 0 đến nhiều hàng đợi có thể nhận được thông điệp. **NServiceBus** sử dụng **Default Exchange** để gửi command. **NServiceBus** chia các thông điệp thành hai loại: event và command. Event được khai báo để trao đổi nơi bất kỳ consumer nào cũng có thể đăng ký tham gia sự kiện. Các ứng dụng gửi command trực tiếp đến consumer cụ thể bằng cách sử dụng tên hàng đợi của chúng.

### 6 - Xử lý đơn hàng nhạy cảm

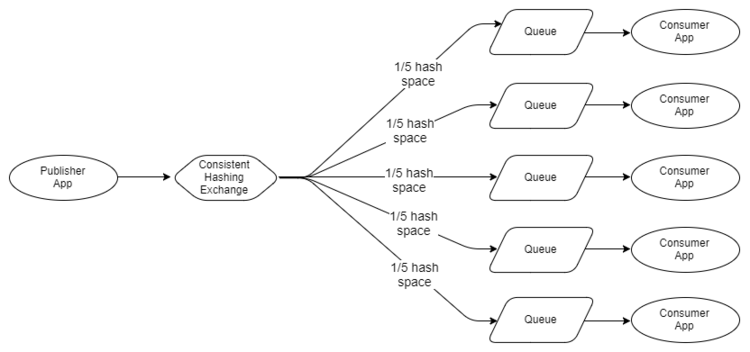
Đôi khi bạn cần phải mở rộng quy mô khách hàng của mình và duy trì xử lý thông điệp theo yêu cầu. Mặc dù **RabbitMQ** đảm bảo việc đặt hàng theo thứ tự hàng đợi, nhưng nếu có nhiều consumer cạnh tranh, mỗi người tiêu thụ nhiều thông điệp song song, thì lệnh xử lý sẽ bị mất.

Hình 13:



Một cách để khắc phục vấn đề này là sử dụng **Consistent Hashing Exchange** và phân vùng hàng đợi thành nhiều hàng đợi và định tuyến thông điệp đến các hàng đợi này dựa trên băm routing key hoặc tiêu đề của thông điệp. Thông thường thứ tự global của tất cả các thông điệp là không cần thiết, chỉ là thứ tự của các thông điệp liên quan. Ví dụ: tất cả các thông điệp liên quan đến đặt phòng nhất định phải được xử lý theo đúng thứ tự. Vì vậy, nếu chúng ta đặt booking id làm routing key hoặc làm tiêu đề thông điệp, chúng ta có thể đảm bảo rằng tất cả các thông điệp của booking id đã cho luôn đi vào cùng một hàng đợi. Sau đó, nếu chúng ta chỉ có một consumer duy nhất tiêu thụ từ hàng đợi đó, chúng ta sẽ nhận được lệnh xử lý đảm bảo chúng ta cần.

Hình 14:



Nhưng, RabbitMQ không giúp bạn phối hợp consumer của mình để khớp một consumer với một hàng đợi. Điều này tùy thuộc vào bạn muốn làm điều gì.

### 7 - Dữ liệu cục bộ

Bằng cách sử dụng **Consistent Hashing exchange** như trong khuôn mẫu trước, chúng ta cũng có được dữ liệu cục bộ. Ví dụ: tất cả các event của user\_1001 luôn chuyển đến consumer 2 vì chúng ta băm một tiêu đề thông điệp có chứa user\_id.

Điều này có nghĩa là consumer 2 có thể thực hiện một số thao tác không khả thi nếu cần các chuyến đi khứ hồi mạng. Chúng ta có thể thực hiện các bộ đếm, tập hợp thời gian thực và như vậy trong bộ nhớ .

Tuy nhiên, trong khi điều này nghe có vẻ tuyệt vời có những nguy hiểm liên quan. Nếu bạn tăng số lượng hàng đợi thì việc phân phối thông điệp sẽ thay đổi. Vì vậy, bây giờ thông điệp của user\_1001 chuyển đến consumer 4. Nhưng consumer 2 không biết rằng, nó chỉ dừng lại nhìn thấy thông điệp của user\_1001. Vì vậy, bây giờ bạn có hai consumer với trong bộ nhớ và tập hợp. Bạn có thể tránh điều này bằng cách thực hiện một số cách thông báo cho consumer về sự thay đổi trong phân phối thông điệp và khiến chúng viết ra các giá trị bộ nhớ của chúng tới một kho lưu trữ dữ liệu và mong đợi một mẩu thông điệp người dùng mới.

### 8 - Định tuyến phân cấp

Đây là một phần mở rộng của mẫu **Public Message Exchange**, **Private Consumer Exchange** và cho phép **Topic Exchange** như định tuyến bằng cách sử dụng **Fanout Exchange**.

Hãy tưởng tượng chúng ta đã chia nghiệp vụ của chúng ta thành các domain, sub-domain và action. Chúng ta có thể xây dựng một không gian tên thông điệp theo định dạng domain.sub-domain.action:

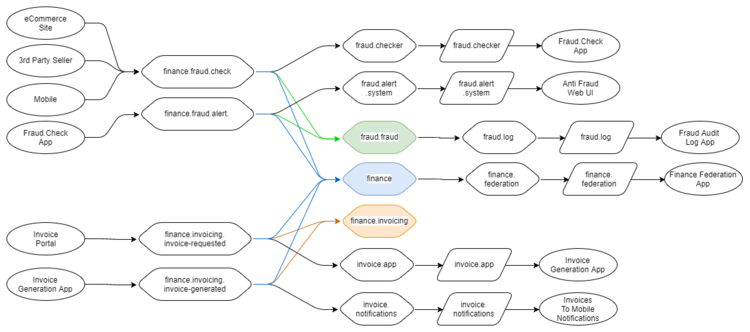
* finance.invoicing.invoice-requested
* finance.invouring.invoice-created
* finance.fraud.alert
* finance.fraud.check

Chúng ta tạo thêm message exchange:

* finance.invoicing
* finance.fraud
* finance

Chúng ta tạo các liên kết cho các exchange này theo không gian tên như dưới đây:

Hình 15:

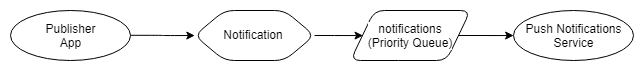


Pattern này có thể hữu ích khi bạn muốn nắm bắt thông điệp của các nhóm lớn các exchange liên quan mà không phải tạo ra số lượng lớn các liên kết. Khi các publisher khai báo exchange của thông điệp mà chúng publish, thì chúng cũng khai báo các exchange trong hệ thống phân cấp và các liên kết cần thiết. Điều này có nghĩa là một khi bạn đăng ký exchange cha, khi các exchange con mới được thêm vào, thông điệp của chúng sẽ tự động được chuyển đến bạn.

### 9 - Lớp dịch vụ với các Hàng đợi ưu tiên

Một số trường hợp của một loại thông điệp nhất định có thể có mức độ ưu tiên cao hơn các loại khác. Có lẽ một số khách hàng quan trọng hơn những khách hàng khác hoặc thông điệp có thể được gắn cờ là ưu tiên cao hơn. Một cách xử lý các thông điệp có mức độ ưu tiên cao hơn trước các mức ưu tiên thấp hơn là định cấu hình hàng đợi dưới dạng hàng đợi ưu tiên và gửi tất cả các thông điệp có mức độ ưu tiên.

Hình 16:



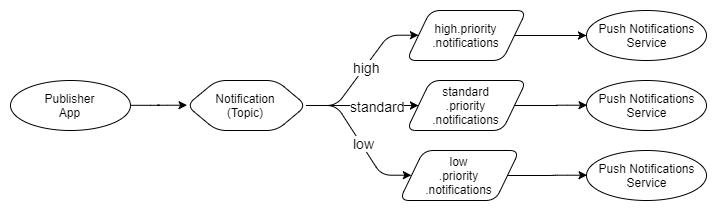
Nhìn chung, hàng đợi ưu tiên rất đơn giản. Các thông điệp ưu tiên thấp hơn có thể bị kẹt sau các thông điệp ưu tiên cao hơn và chữ chết có thể đẩy các thông điệp có mức độ ưu tiên cao có lợi cho các thông điệp có mức độ ưu tiên thấp.

Một cách tiếp cận tốt hơn cho lớp dịch vụ là sử dụng **Topic Exchange**, xem mẫu tiếp theo.

### 10 - Lớp dịch vụ với các Topic Exchange

Sử dụng một Topic với mức ưu tiên được đặt trong routing key không có các vấn đề về hàng đợi ưu tiên. Các thông điệp được chuyển đến các hàng đợi khác nhau về mặt vật lý và được xử lý bởi các trường hợp ứng dụng khác nhau. Hàng đợi ưu tiên cao có thể cung cấp độ trễ thấp hơn chỉ bởi thực tế là các thông điệp không phải chờ phía sau các thông điệp có mức độ ưu tiên thấp hơn, nhưng các thông điệp có mức độ ưu tiên cao hơn có thể được sử dụng bởi một nhóm consumer có quy mô lớn hơn trên các máy ảo lớn hơn.

Hình 17:

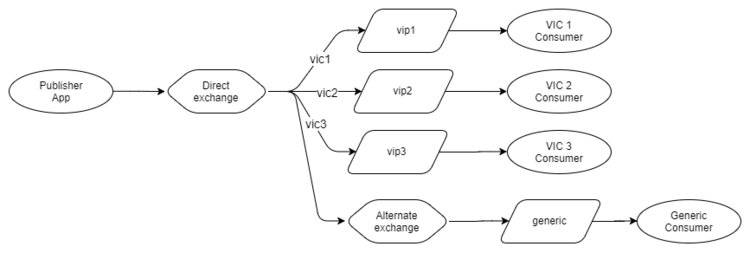


### 11 - Định tuyến If/Else với Alternate Exchange

Hãy tưởng tượng chúng ta có một số khách hàng siêu quan trọng yêu cầu hành vi tùy chỉnh cho từng thông điệp và mỗi thông điệp này cần đến một consumer dành riêng cho khách hàng đó. Các thông điệp liên quan đến hàng trăm khách hàng ít quan trọng hơn sẽ được xử lý bởi một consumer chung chung.

Chúng ta có thể đạt được điều này bằng cách đặt một mã định danh khách làm routing key và gửi thông điệp đến một **Direct Exchange** được cấu hình với một **Alternate Exchange**.

Hình 18:



**Alternate Exchange** chỉ là một exchange thông thường trong bốn loại exchange. Bạn thậm chí có thể xâu chuỗi các exchange luân phiên với nhau và thực hiện logic if, else if, else.

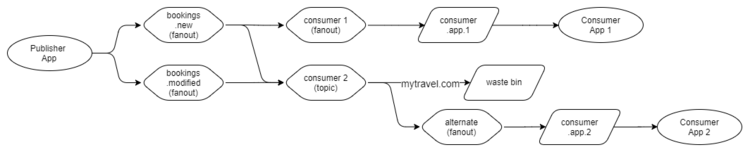
**Topic Exchange** không thể cung cấp định tuyến này vì nó không thể thực hiện OR, nó chỉ có thể thực hiện AND. Nếu chúng ta sử dụng **Topic Exchange** và sử dụng ký tự # đại diện để chụp tất cả các thông điệp, cuối cùng chúng ta sẽ xử lý các thư khách hàng rất quan trọng cũng như các máy khách ít quan trọng hơn.

### 12 - KHÔNG ĐỊNH TUYẾN với Alternate Exchange

Đôi khi bạn muốn tiêu thụ tất cả các thông điệp **ngoại trừ** một loại cụ thể. Không ai trong bốn loại exchange cung cấp kết hợp tiêu cực. Thay vào đó, chúng ta có thể tạo một hàng đợi "thùng rác" và liên kết cho thông điệp bạn không muốn. Hàng đợi này được thiết lập với một thông điệp dựa trên hàng đợi rất ngắn TTL để các thông điệp bị loại bỏ gần như ngay lập tức khi đến nơi.

Bạn định cấu hình exchange với một **Alternate Exchange** và kết nối khách hàng của bạn với một hàng đợi liên kết với **Alternate Exchange** đó. Bây giờ bạn tiêu thụ tất cả các thông điệp ngoại trừ một loại bạn không muốn.

Hình 19:



Cấu trúc liên kết này tương tự như cấu trúc trong mẫu **Public Message Exchange**, **Private Consumer Exchange**. Consumer 2 thiết lập một **Topic Exchange** private và định tuyến tất cả các booking được thực hiện từ kênh bán hàng mytravel.com đến hàng đợi "rác" và sau đó tiêu thụ phần thông điệp còn lại.

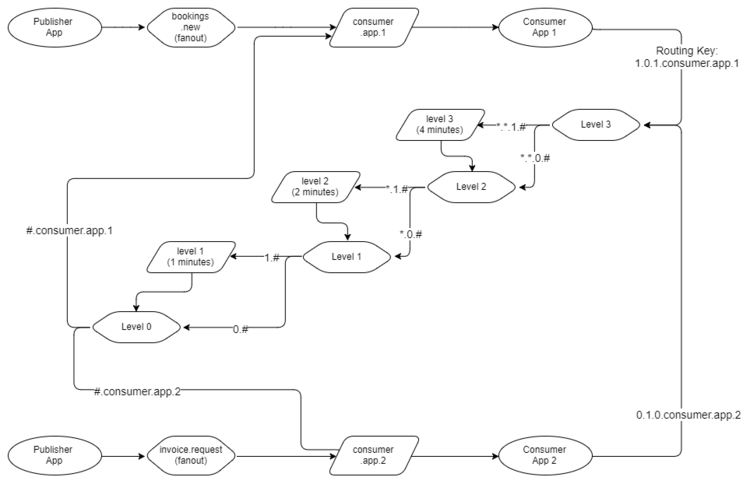
Rõ ràng bạn chỉ có thể để consumer của mình tiêu thụ mọi thông điệp và chỉ cần loại bỏ những thông điệp bạn không muốn. Nó phụ thuộc vào kịch bản cụ thể của bạn.

### 13 - Delayed Retry với Cascading Exchange và Queue

Có rất nhiều lời khuyên tồi tệ liên quan đến việc định tuyến thử lại bị trì hoãn trong RabbitMQ. Tất cả các phương pháp thử lại bị trì hoãn đều dựa vào thông báo hết hạn sử dụng và DLX. Nhiều người không tính đến việc chỉ những thông điệp ở đầu hàng đợi được gửi đến DLX. Điều này có nghĩa là bạn không thể trộn thời gian trễ trong một hàng đợi vì các thông điệp bị trì hoãn ngắn hơn bị kẹt sau các thông điệp bị trì hoãn lâu hơn. Bây giờ cảnh báo đó đã được nói, chúng ta hãy nhìn vào mô hình rất sáng tạo này.

Mẫu này được lấy từ **NServiceBus**. Nó sử dụng **Topic Exchange** xếp tầng liên kết với nhau thông qua cấu hình thư chết và định tuyến chủ đề. Ý tưởng là chúng ta tạo ra nhiều mức độ trễ, trong đó mỗi cấp chịu trách nhiệm cho khoảng thời gian trễ cố định của chính nó. Các giai đoạn này tăng lên theo cấp số 2. Cấp 1 là 1 phút, cấp 2 là 2 phút, cấp 3 là 4 phút, cấp 4 là 8 phút, sau đó sử dụng các routing key và binding key kiểu nhị phân, chúng ta có thể di chuyển một thông điệp giữa các hàng đợi trễ để đạt được bất kỳ khoảng thời gian trễ ở độ phân giải 1 phút. Bạn có thể sử dụng 20 hàng đợi hoặc hơn, với cấp 1 ở mức 1 giây và đạt được độ trễ độ phân giải thứ hai lên đến một khoảng thời gian nhiều năm!

Hình 20:



Biểu đồ trên cho thấy tầng này, cơ sở hạ tầng thử lại bị trì hoãn chia sẻ chỉ với ba cấp độ. Với ba cấp độ và cấp 1 là 1 phút, chúng ta có thể đạt được bất kỳ độ trễ nào lên đến 7 phút với độ phân giải 1 phút. Không nhiều nhưng đây là một phiên bản siêu đơn giản.

Mỗi hàng đợi chậm được cấu hình là thư chết của nó trao đổi mức dưới đây. Trong ví dụ này, chúng ta thấy rằng consumer 1 gửi thông điệp để thử lại với độ trễ là 5 phút. Chúng ta hãy xem cách thông điệp chảy qua các exchange và hàng đợi:

* Exchange cấp 3 có hai liên kết. routing key 1.0.1.consumer.app.1 khớp với khóa liên kết \*.\*.1.# chỉ. Vì vậy, nó được chuyển đến hàng đợi cấp 3. Hàng đợi đó có thông báo 4 phút được cấu hình TTL. Sau khi chờ đợi 4 phút, nó sẽ được gửi đến exchange cấp 2.
* Exchange cấp 2 có hai liên kết. Routing key 1.0.1.consumer.app.1 chỉ khớp với khóa liên kết \*.0.#, vì vậy thông điệp được chuyển đến exchange cấp 1.
* Exchange cấp 1 có hai liên kết. Routing key 1.0.1.consumer.app.1 chỉ khớp với khóa liên kết 1.#. Vì vậy, nó được chuyển đến hàng đợi Cấp 1. Hàng đợi đó có thông báo 1 phút được cấu hình TTL. Sau khi chờ đợi trong 1 phút, nó sẽ được gửi đến thư exchange cấp 0.
* Cả hai consumer đã tạo ra các ràng buộc từ hàng đợi của chúng để trao đổi này. Thông báo khớp với liên kết #.consumer.app.1 và do đó được chuyển đến hàng đợi consumer 1 nơi nó được Tiêu dùng 1 sử dụng một lần nữa, 5 phút sau khi nó gửi thông điệp đến cơ sở hạ tầng thử lại.

Một số điều cần nhớ về pattern này:

* Đây là một cách tiếp cận cơ sở hạ tầng được chia sẻ và thời gian chờ đợi có thể không chính xác dưới tải.
* Nếu thông điệp ban đầu có routing key, nó sẽ bị xóa để mẫu này hoạt động. Tôi khắc phục điều này bằng cách đặt routing key làm tiêu đề thư khi gửi thông điệp để thử lại, sau đó consumer có thể truy xuất routing key ban đầu nếu cần.
* Nếu thông điệp ban đầu có TTL và bạn muốn nó được tôn trọng thì bạn sẽ cần thêm nó làm tiêu đề thư khi bạn gửi thử lại và yêu cầu ứng dụng của bạn kiểm tra và hủy thông điệp nếu khoảng thời gian đã qua. Lý do cho điều này là khi ứng dụng khách hàng gửi thông điệp đến các trao đổi thử lại, nó không nên đặt thông báo TTL vì điều đó có thể ảnh hưởng đến thời gian thử lại. Trong mọi trường hợp, khi một thông điệp bị chết, bất kỳ thông điệp nào cũng bị xóa khỏi thông điệp.
* Retries và đặt hàng thông điệp trái ngược về cơ bản. Nếu thứ tự thông điệp là quan trọng thì việc thử lại bị trì hoãn có lẽ không phải là một ý tưởng hay trừ khi bạn thêm một số logic phát hiện các thông điệp cũ hơn.

### 14 - Delayed Retry với Ephemeral Exchanges và Queue

Chúng ta có thể đạt được kết quả tương tự như pattern trước đó với các **Ephemeral Exchange** và hàng đợi không bền. Khi một ứng dụng muốn gửi thông điệp cho Delayed retry, nó sẽ tạo ra một exchange và queue một lần với tên được đảm bảo duy nhất (ví dụ sử dụng GUID/UUID).

**Ephemeral Exchange** được cấu hình như sau:

* Fanout
* Auto-delete

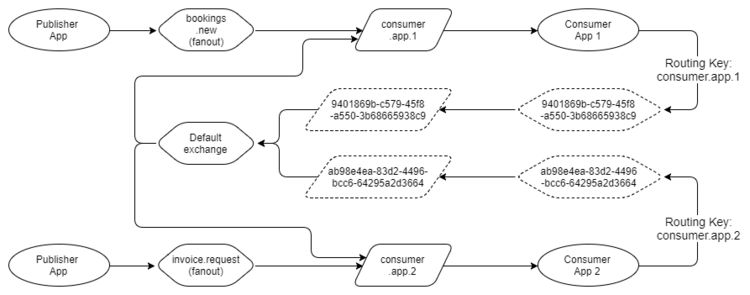
Hàng đợi không bền được cấu hình như sau:

* Một thông điệp TTL tương ứng với độ trễ bạn muốn.
* DLX của nó là **Default Exchange**.
* Một hàng đợi hết hạn một vài giây sau thông điệp TTL.
* Một liên kết đến **Ephemeral Exchange**.

Consumer khai báo exchange và hàng đợi, sau đó gửi thông điệp cho Delayed retry với routing key là tên của hàng đợi của chính nó. Một loạt các sự kiện tiếp theo xảy ra:

1. **Ephemeral Exchange** định tuyến thông điệp đến hàng đợi không bền
2. Thông điệp nằm trong hàng đợi cho khoảng thời gian thông điệp TTL
3. Thông điệp bị chết được gửi đến **Default Exchange**
4. **Default Exchange** định tuyến thông điệp này đến hàng đợi khớp với routing key
5. Hàng đợi không bền vững đạt đến TTL của hàng đợi và tự động xóa chính nó
6. **Ephemeral Exchange** thấy rằng không có hàng đợi nào liên kết với nó và nó tự động xóa chính nó.

Hình 21:

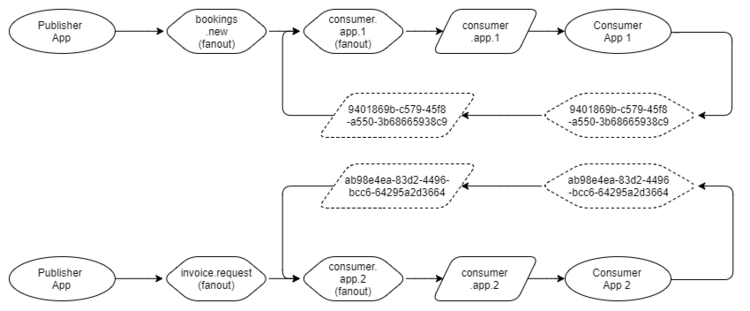


Cân nhắc cần tính đến:

* Tạo exchange, hàng đợi và liên kết là tương đối tốn kém. Nếu bạn tạo tải cao khi thử lại thì điều này có thể gây áp lực quá lớn cho cluster.
* Giống như pattern exchange xếp tầng, routing key ban đầu và thông điệp TTL được loại bỏ để thực hiện công việc này.

Nếu bạn sử dụng pattern **Public Message Exchange**, **Private Consumer Exchange pattern**, bạn hoàn toàn không cần dựa vào **Default Exchange** và có thể định cấu hình exchange riêng của consumer làm DLX. Điều này loại bỏ sự cần thiết phải thiết lập một routing key tùy biến.

Hình 22:



### 15 - Delayed Delivery với Delayed Message Exchange Plug-In

<https://github.com/rabbitmq/rabbitmq-delayed-message-exchange>

Plug-in này cung cấp cho bạn loại exchange mới, exchange thông điệp bị trì hoãn. Exchange này có thể bắt chước các loại exchange thông thường với vòng xoắn mà nếu bạn đặt tiêu đề "x-delay" trên một thông điệp, exchange sẽ trì hoãn việc gửi thông điệp cho số mili giây trong giá trị tiêu đề của bạn.

Nhược điểm của plugin này là nó không hỗ trợ tính khả dụng cao của các thông điệp đang trong thời gian trì hoãn. Vì vậy, việc mất một nút sẽ dẫn đến việc mất các thông điệp bị trì hoãn trên nút đó. Một nhược điểm khác là không hỗ trợ cờ "bắt buộc". Chúng ta sẽ đề cập đến vấn đề này trong các phần tiếp theo, nó đóng vai trò quan trọng trong việc bảo đảm gửi thông điệp.

### 16 - Delayed Delivery với Cascading Exchange và Queue

Nếu plug-in exchange thông điệp bị trì hoãn không dành cho bạn thì bạn có thể thử pattern exchange xếp hàng và hàng đợi. Đây là mô hình tương tự như mô hình **Delayed Retry với Cascading Exchange và Queue**, ngoại trừ việc các publisher gửi trực tiếp đến các trao đổi chậm trễ. Bởi vì chúng ta sử dụng một khóa nhị phân như các routing key, điều này đặt ra gánh nặng cho publisher để có thể tạo routing key chính xác. Điều này có thể hoạt động nếu bạn tạo một thư viện mã để quản lý việc tạo các thông điệp bị trì hoãn và có quyền kiểm soát của publisher.

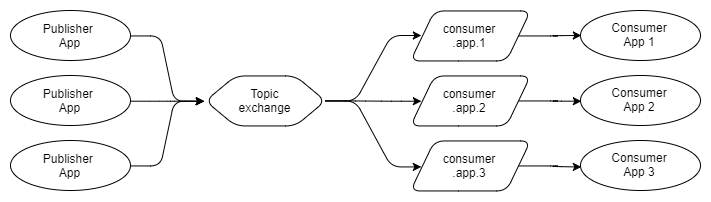
Lợi ích của phương pháp này so với trình cắm thêm là chúng ta không mất các khả năng của RabbitMQ HA.

### 17 - Delayed Delivery với Private Publisher Exchange

Nếu publisher luôn muốn có độ trễ giống nhau trên tất cả các thông điệp thì chúng có thể khai báo trao đổi và xếp hàng của riêng chúng cho mục đích thêm độ trễ.

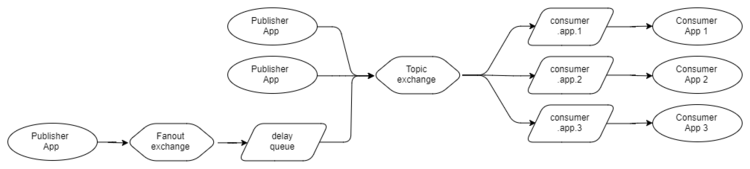
Giả sử publisher thứ ba của sơ đồ bên dưới cần trì hoãn thông điệp trước 5 phút, vì lý do kinh doanh X.

Hình 23:



Publisher thứ ba khai báo exchange và hàng đợi riêng của nó, điều đó có nghĩa là nó không ảnh hưởng đến các publisher khác. Bởi vì chúng ta luôn muốn trì hoãn 5 phút, chúng ta có thể sử dụng một hàng đợi duy nhất với DLX của nó làm **Topic Exchange** chính.

Hình 24:



Đây là phương pháp đơn giản nhất trong tất cả nhưng chỉ hoạt động với thời gian trễ tiêu chuẩn. Như đã nêu trước đó, các thông điệp chỉ được coi như đã "chết" từ đầu hàng đợi vì vậy các thông điệp có TTL ngắn hơn sẽ bị kẹt phía sau các thông điệp có TTL dài hơn. Bạn không thể pha trộn các thông điệp có độ dài khác nhau.

### 18 - RPC và hàng đợi trả lời

Để thực hiện nhắn tin theo kiểu Gọi thủ tục từ xa (RPC), nói chung, bạn phải sử dụng hàng đợi Trả lời mà người nhận thông điệp của bạn có thể trả lời. Điều này có thể khó khăn để có được đúng và bạn nên thực sự tự hỏi mình nếu bạn thực sự cần sử dụng một hệ thống nhắn tin cho RPC. Hệ thống nhắn tin được xây dựng với tâm trí không đồng bộ và độ bền. Hầu hết các lý do để sử dụng một hệ thống nhắn tin không có ở đó khi nói đến RPC. Nhưng nếu bạn vẫn muốn RPC thì bạn có một vài lựa chọn trên RabbitMQ.

Nhưng trước tiên, hãy nói chung về sự phức tạp của RPC qua một hệ thống nhắn tin. Trong trường hợp của chúng ta, RabbitMQ có một tính năng thực sự gọn gàng giúp giải quyết các vấn đề mà các hệ thống nhắn tin khác gặp phải. Trước tiên chúng ta hãy hiểu những vấn đề đó để chúng ta có thể đánh giá cao chức năng do RabbitMQ cung cấp (vui lòng bỏ qua đến cuối mẫu này để xem tính năng hay của RabbitMQ cho RPC).

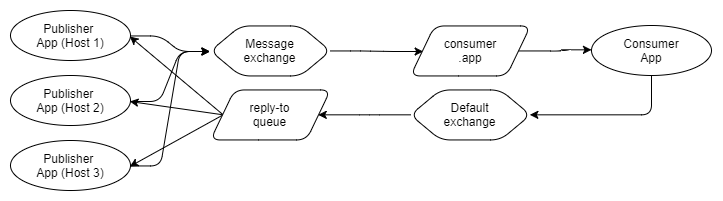
Một câu hỏi quan trọng là: bạn có muốn thực hiện cuộc gọi trong mã giống như bất kỳ lời gọi phương thức nào khác không? Nếu câu trả lời là có thì bạn cần một kiến ​​trúc **RPC** tương thích với bối cảnh trạng thái. Đó là, có trạng thái trong một luồng hoạt động trên một máy chủ cụ thể cần thông báo phản hồi để quay lại nó. Nếu chúng ta có dịch vụ mở rộng gồm 10 trường hợp, với hàng trăm luồng cho mỗi phiên bản, chúng ta cần thông báo phản hồi được gửi trở lại cùng luồng trên cùng một máy chủ. Điều này quy định một số mẫu.

Tuy nhiên, nếu chúng ta có bối cảnh không trạng thái thì chúng ta có thể tự do chọn bất kỳ mẫu RPC nào. Nếu khi thực hiện RPC, bối cảnh của việc thực thi đó kết thúc và tất cả trạng thái bị mất hoặc được lưu trong bộ đệm với correlation Id trong Redis hoặc một cái gì đó, thì chúng ta có nhiều tự do hơn để chọn các mẫu khác nhau. Tuy nhiên bối cảnh không trạng thái này không phải lúc nào cũng có thể thực hiện hoặc mong muốn.

Chúng ta hãy xem xét từng tùy chọn, cho dù nó hoạt động trong bối cảnh có trạng thái hoặc không trạng thái và có thể có những sự đánh đổi khác. Chúng ta sẽ khám phá chúng bằng cách sử dụng các exchange và hàng đợi của RabbitMQ.

#### Sửa lỗi Reply-To Queue và Correlation ID

Hình 25:



Thông điệp gửi đi bao gồm hai tiêu đề thư:

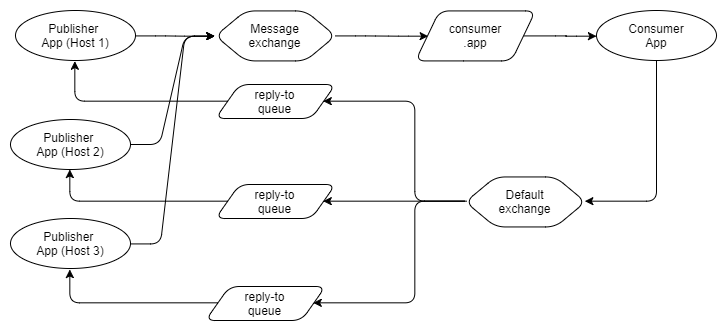
* reply-to-queue
* correlation-id

Người nhận gửi thông điệp đến **Default Exchange** với tên reply-to-queue làm routing key. Nó cũng bao gồm tiêu đề và giá trị id tương quan tương tự. Người gửi ban đầu sử dụng trả lời này để xếp hàng và có thể truy xuất bất kỳ trạng thái nào từ một cửa hàng trạng thái bằng cách sử dụng id tương quan.

Rõ ràng điều này không tương thích với bối cảnh nhà nước của chúng ta. Bất kỳ trong số 10 trường hợp ứng dụng của chúng ta đều có thể sử dụng thông điệp.

#### Hàng đợi cho mỗi ứng dụng và Correlation ID

Hình 26:

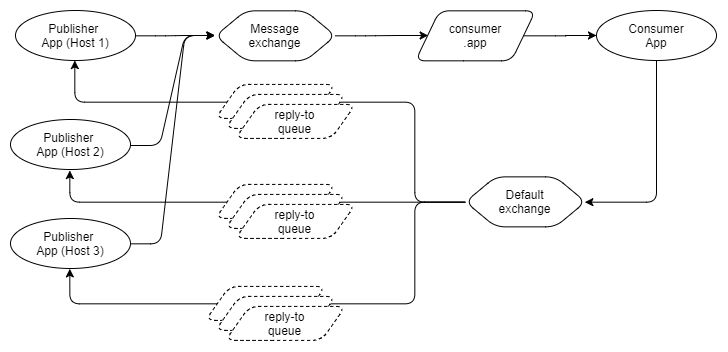


Điều này giống với mẫu Hàng đợi cố định ngoại trừ mỗi phiên bản ứng dụng có một hàng đợi riêng. Điều này có thể tương thích với bối cảnh trạng thái nếu bạn đi qua một số vòng để đến đó. Trước hết, nếu ứng dụng của bạn là một luồng, chỉ có một yêu cầu hoạt động tại một thời điểm thì bạn có thể chắc chắn rằng luồng chờ phản hồi sẽ là luồng nhận được thông báo phản hồi. Nhưng những ứng dụng này không phổ biến. Nhiều khả năng nó là một ứng dụng web và do đó đa luồng với nhiều yêu cầu hoạt động. Trong trường hợp này, nó phụ thuộc vào khả năng của ngôn ngữ lập trình.

Tuy nhiên, đây không phải là code đơn giản để có thể viết và có thể là một lựa chọn rủi ro tùy thuộc vào kỹ năng lập trình đồng thời của bạn. Lỗi sẽ khó chẩn đoán và sửa chữa. Nếu bạn có thể tìm thấy một thư viện để làm điều đó cho bạn thì đây có thể là một lựa chọn hợp lý.

#### Ephemeral Reply-To Queue với No Correlation Id

Hình 27:



Điều này hoàn toàn tương thích với bối cảnh lưu trạng thái và là mô hình đơn giản nhất. Người gọi khai báo một hàng đợi với GUID/UUID làm tên và chuyển nó dưới dạng tiêu đề của thông điệp trong thông điệp request. Hàng đợi nên được thực hiện một hàng đợi tự động xóa để nó tự động xóa chính nó sau khi người gửi đã ngừng sử dụng nó. Người nhận sẽ gửi một thông điệp response đến hàng đợi này như được đã được chỉ ra trong tiêu đề của thông điệp request. Khi người gửi đã sử dụng thông điệp response, nó sẽ hủy đăng ký đến hàng đợi này và hàng đợi này sẽ tự động xóa.

Vậy đâu là sự đánh đổi với thiết kế đơn giản và thanh lịch này? Hiệu suất. Tạo hàng đợi có thể tốn kém. Nếu bạn có một lưu lượng truy cập kha khá thì bạn sẽ cần phải thực hiện các thử nghiệm để xem liệu việc tạo liên tục và tự động xóa hàng đợi có gây quá nhiều gánh nặng cho cluster hay không. Các hệ thống thông điệp khác thậm chí không có exchange phù hợp và hàng đợi để giải quyết vấn đề này!

#### Tính năng đánh bại RPC của RabbitMQ - Direct Reply-To

Cuối cùng, chúng ta cũng có được tính năng RPC thú vị của RabbitMQ. Trả lời trực tiếp bên cạnh tất cả các vấn đề này bằng cách cung cấp cho bạn một cái gì đó rất giống với mẫu phù du nhưng không có vấn đề về hiệu suất. Đây là cách nó hoạt động:

* Người gửi đặt tên của một "hàng đợi giả" được gọi là amq.rabbitmq.reply-to trong tiêu đề thư trả lời. Đây là một hàng đợi giả vì nó hoàn toàn không phải là một hàng đợi, nhưng nó có thể được coi là một hàng đợi.
* Người gửi tiêu thụ amq.rabbitmq.reply-to queue này ở chế độ không ack.
* Người nhận sẽ gửi thông điệp trả lời tới **Default Exchange** với routing key là tên của hàng đợi giả này.
* Consumer được đẩy thông điệp từ nút RabbitMQ trực tiếp mà không bao giờ được ghi vào hàng đợi.

Rõ ràng là chúng ta mất các đảm bảo tính sẵn sàng cao vì chúng ta không thể sử dụng phản chiếu hàng đợi để sao chép thông báo qua các nút, nhưng với RPC, chúng ta không thực sự cần điều đó. Ngoài ra nếu người gửi bị ngắt kết nối thì hàng giả sẽ biến mất và không thể gửi phản hồi. Nhưng điều đó không khác gì RPC so với HTTP.

Về cơ bản, giống như với HTTP nếu có lỗi xảy ra, bạn chỉ cần thử lại yêu cầu và thậm chí sử dụng các mẫu như bộ ngắt mạch nếu cần.

Vì vậy, RabbitMQ có một hành vi tùy chỉnh đặc biệt để thực hiện RPC để tránh tất cả sự đau đớn của hàng đợi trả lời thực sự.

## Vòng đời thông điệp

Chúng ta có thể tạo vòng đời thông điệp đầy đủ xử lý các lỗi tạm thời và không nhất thời theo cách đảm bảo rằng chúng ta không bị mất thông điệp và chúng ta có thể phản hồi các lỗi xử lý theo cách được kiểm soát và quản lý.

Vòng đời bao gồm những gì? Về cơ bản nó là một luồng công việc của các đường dẫn có thể mà một thông điệp có thể thực hiện. Nó bắt đầu từ việc xuất bản một thông điệp, sau đó bao gồm xử lý thành công thông điệp hoặc thử lại trong trường hợp thất bại thoáng qua, một nơi để gửi thông điệp không thể xử lý, một nơi để lưu trữ thông điệp thất bại, tùy chọn loại bỏ, trả lại thông điệp thất bại cho consumer ban đầu, vv

Cốt lõi của khái niệm này cũng là các thông điệp thất bại mang theo tất cả thông tin thực tế có thể để chẩn đoán sự thất bại. Điều này có nghĩa là:

* Ứng dụng tiêu dùng
* Thông báo ngoại lệ hoặc lỗi
* Máy chủ
* Thời gian
* Ai công bố thông điệp
* Thông điệp này đã được xử lý bao nhiêu lần rồi?

Chúng ta xây dựng vòng đời vào thư viện mã nhắn tin của mình với API mã siêu đơn giản để các nhà phát triển sử dụng. Ngoài ra, chúng ta đảm bảo rằng API mã buộc chúng phải suy nghĩ về bản chất của sự thất bại - nó là nhất thời hay tồn tại? chúng ta có thực hiện thử lại ngay lập tức, thử lại chậm trễ, loại bỏ thông điệp hoặc gửi nó đến một hàng đợi thông điệp thất bại?

Đọc bài viết đầy đủ của tôi về việc xây dựng vòng đời thông điệp với RabbitMQ.

Các mẫu thông điệp khác Có nhiều mẫu liên quan đến nhắn tin nói chung mà tôi chưa đề cập đến như:

* Sagas
* Command / Event
* Xử lý thông điệp Kiểm tra nguồn cấp dữ liệu
* Thay đổi nguồn cấp dữ liệu thu thập dữ liệu (CDC)

# Cấu trúc của Kafka và các messaging pattern khi sử dụng Kafka

Trong phần này chúng ta sẽ xem xét Kafka, và đối chiếu nó với RabbitMQ để tìm ra sự khác biệt giữa hai loại. Lưu ý rằng, sự so sánh ở đây được giới hạn trong bối cảnh của một kiến ​​trúc ứng dụng hướng sự kiện hơn là **data processing pipeline**, mặc dù hai loại kiến trúc này khá giống nhau.

Sự khác biệt đầu tiên ta có thể nghĩ đến đó là các pattern retry và delay không có ý nghĩa với Kafka. Các thông điệp trong RabbitMQ chỉ là các thông điệp là tạm thời, chúng được di chuyển và sau đó biến mất. Vì vậy, việc ta thêm lại các thông điệp giống với những lần tiêu thụ thông điệp trước đó hoàn toàn là usecase thực tế. Nhưng trong Kafka thì log mới là trung tâm của toàn bộ hệ thống. Việc ta thêm lại một thông điệp trùng lặp với lần trước đó cho lần xử lý thông điệp thứ hai sẽ không mang ý nghĩa gì cả và điều này sẽ chỉ làm cho log bị lỗi. Một trong những điểm mạnh của Kafka đó là tính đảm bảo trật tự của nó bên trong một phân vùng log, việc thêm các bản sao sẽ làm rối tung tất cả. Trong RabbitMQ, bạn có thể thêm lại một thông điệp vào hàng đợi mà một consumer đơn lẻ có thể tiêu thụ, nhưng Kafka là một hệ thống log thống nhất mà tất cả consumer đều tiêu thụ. Việc trì hoãn không đi ngược lại với khái niệm của log nhưng Kafka không cung cấp cho ta cơ chế này.

Một điểm khác biệt lớn thứ hai ảnh hưởng đến các mô hình mà RabbitMQ và Kafka có thể thực hiện đó là trong RabbitMQ các thông điệp được gửi không bền vững, còn trong Kafka thì bền vững hơn. Trong RabbitMQ một khi một thông điệp được sử dụng, nó sẽ biến mất, không còn dấu vết của thông điệp đó. Nhưng trong Kafka, mỗi thông điệp được lưu vào log và vẫn ở đó cho đến khi được dọn sạch. Việc làm sạch dữ liệu thực sự phụ thuộc vào lượng dữ liệu bạn có, dung lượng bạn có thể cung cấp cho nó và các pattern mà bạn muốn thực hiện được. Ta có thể thực hiện theo cách tiếp cận thời gian, sử dụng ngày/tuần/tháng cuối của thông điệp hoặc ta có thể sử dụng Log Compaction để cung cấp cho ta trạng thái gần nhất theo cách tiếp cận khóa thông điệp.

Dù bằng cách nào, chúng ta đều cho phép consumer quay lại và tái tiêu thụ các thông điệp cũ. Nghe có vẻ rất giống với hành vi thử lại mặc dù không giống như cách thực hiện của RabbitMQ.

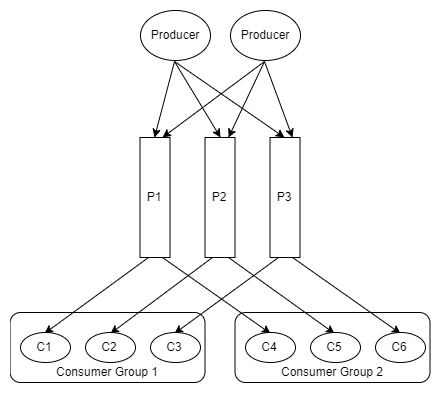
Trong khi RabbitMQ di chuyển các thông điệp và cung cấp cho ta những cách thức rất mạnh mẽ để có thể tạo ra các mẫu định tuyến sáng tạo, thì Kafka lại cho ta cách lưu trữ trạng thái hiện tại và lưu trữ lịch sử của một hệ thống, nó có thể được sử dụng như một nguồn dữ liệu đáng tin cậy mà RabbitMQ không thể đáp ứng được.

## Ví dụ về một số pattern khi sử dụng Kafka

### #1 Gửi thông điệp với Publish Subscribe đơn giản kết hợp tùy chọn quay vòng

Trường hợp sử dụng đơn giản nhất cho cả RabbitMQ và Kafka là kiến trúc publish-subscribe. Một publisher hoặc nhiều publisher chỉ cần ghi thông điệp vào phân vùng log và một hoặc nhiều nhóm consumer tiêu thụ những thông điệp đó.

Hình 1



Ngoài các chi tiết về việc quản lý các publisher gửi thông điệp đến đúng phân vùng và các nhóm consumer tự phối hợp với nhau, thì còn lại không khác gì một cấu trúc liên kết Fanout exchange trong RabbitMQ.

Tính năng này (sử dụng log thay vì queue) rất hữu ích để phục hồi khi có lỗi. Trong nhiều trường hợp, các thông điệp đã biến mất, nhưng thông thường chúng ta có thể lấy dữ liệu gốc từ hệ thống của bên thứ ba và publish lại các thông điệp, chỉ cho một thuê bao có vấn đề. Điều này đòi hỏi chúng ta phải xây dựng cơ sở hạ tầng re-publish tùy biến. Nếu chúng ta đã có Kafka, nó sẽ đơn giản như thay đổi khoảng trống cho ứng dụng đó.

### #2 Event-Driven Data Integration

Mẫu này về cơ bản là event sourcing mặc dù không thuộc phạm vi của một ứng dụng. Có hai loại event sourcing, cấp ứng dụng và cấp hệ thống và pattern này liên quan đến loại thứ hai.

#### Event Sourcing cấp ứng dụng

Ứng dụng quản lý trạng thái của chính nó như là một chuỗi các sự kiện thay đổi bất biến. Những sự kiện này được lưu trữ trong một event store. Để có được trạng thái hiện tại của một thực thể có nghĩa là thực hiện lại hoặc kết hợp các sự kiện của thực thể đó theo đúng thứ tự. Nó được kết hợp tự nhiên với **CQRS**. Đối với một số ứng dụng, đây có thể là kiến ​​trúc phù hợp nhưng nó thường bị hiểu sai và áp dụng sai, thêm độ phức tạp không cần thiết, trong khi một ứng dụng **CRUD** truyền thống sẽ ổn.

Chúng ta sẽ bỏ qua loại này.

#### Event Sourcing cấp hệ thống

Các ứng dụng có thể quản lý trạng thái của riêng chúng theo bất kỳ cách nào chúng muốn. Nếu chúng có thể lưu trữ trạng thái của chúng trong cơ sở dữ liệu quan hệ và có được sự thống nhất giao dịch ngay lập tức đi kèm với điều đó thì thật tuyệt - chúng ta nên thực hiện điều đó. Không nhất thiết phải là mối quan hệ nhưng tính nhất quán ngay lập tức với đảm bảo ACID thực sự là "chén thánh" cho các hệ thống **OLTP**. Cuối cùng, chúng ta chỉ chuyển đến kiến ​​trúc nhất quán khi tính nhất quán ngay tức thì không mở rộng.

Nhưng các ứng dụng thường cần dữ liệu của nhau và nhu cầu đó có thể dẫn đến các kiến ​​trúc không tối ưu như cơ sở dữ liệu dùng chung, ranh giới domain không tồn tại hoặc phụ thuộc vào API REST tồi.

Có một [podcast](https://softwareengineeringdaily.com/2016/10/14/kafka-event-sourcing-with-neha-narkhede/), trong đó họ mô tả một kịch bản event sourcing với một dịch vụ hồ sơ trong mạng xã hội. Có một loạt các dịch vụ liên quan khác như tìm kiếm, biểu đồ xã hội, recommendation service,... Tất cả cần phải biết về các thay đổi của hồ sơ. Theo kinh nghiệm của tôi, ví dụ về một hệ thống đặt vé của một hãng hàng không, chúng ta có một vài hệ thống phần mềm lớn với vô số dịch vụ nhỏ hơn quay quanh các hệ thống lớn hơn đó. Các dịch vụ phụ trợ cần dữ liệu đặt chỗ và dữ liệu hoạt động bay. Mỗi lần đặt chỗ được thực hiện, sửa đổi, một chuyến bay bị trì hoãn hoặc hủy các dịch vụ này cần phải đi vào hoạt động.

Đây là nơi event sourcing có ích. Nhưng trước tiên, hãy xem xét một số vấn đề phổ biến phát sinh trong các hệ thống phần mềm lớn và sau đó xem cách event sourcing có thể giải quyết điều đó.

Hệ thống doanh nghiệp phức tạp lớn thường phát triển có hệ thống; có sự chuyển đổi sang công nghệ mới và kiến ​​trúc mới có thể không đạt tới 100% hệ thống. Dữ liệu được truyền tải khắp doanh nghiệp ở nhiều nơi khác nhau, các ứng dụng dùng chung cơ sở dữ liệu để tích hợp nhanh chóng và không ai chắc chắn làm thế nào tất cả phù hợp với nhau.

#### Phân tán dữ liệu lộn xộn

Dữ liệu được phân phối và quản lý ở nhiều nơi khiến bạn khó hiểu:

* Dữ liệu phát sinh qua nghiệp vụ của bạn như thế nào
* Thay đổi trong một phần của hệ thống có thể ảnh hưởng đến các phần khác như thế nào
* Nhiều bản sao của dữ liệu tồn tại, phân mảnh theo thời gian và xuất hiện các xung đột dữ liệu

Không có các ranh giới domain rõ ràng, các thay đổi sẽ tốn kém và rủi ro vì điều đó có nghĩa là phải đụng chạm vào nhiều hệ thống.

#### Dùng chung cơ sở dữ liệu

Dùng chung cơ sở dữ liệu có thể gây ra một vài vấn đề đau đầu:

* Không thực sự tối ưu hóa cho bất kỳ ứng dụng nào. Nó có thể là tập dữ liệu đầy đủ và được chuẩn hóa hoàn toàn. Một số ứng dụng phải viết các truy vấn lớn để có được dữ liệu chúng cần.
* Các ứng dụng có thể ảnh hưởng đến hiệu suất của nhau.
* Các thay đổi đối với schema có nghĩa là các dự án migrate quy mô lớn, trong đó việc phát triển bị tạm dừng trong vài tháng để đưa tất cả các ứng dụng cùng được cập nhật, tất cả cùng một lúc.
* Không có thay đổi nào cho schema được thực hiện. Sự thay đổi mà mọi người muốn sẽ không bao giờ được thực hiện vì nó quá tốn kém.

#### Phụ thuộc vào các REST API kém

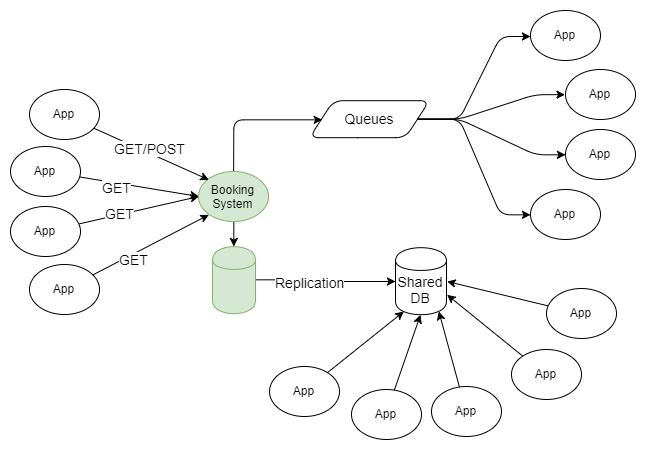
Mỗi API REST có thể có một kiểu dáng và quy ước khác nhau. Để có được dữ liệu mà bạn muốn có thể bạn sẽ cần phải sử dụng rất nhiều HTTP request và điều này có thể khá rắc rối. **GraphQL** có thể giúp bạn việc này nhưng nó vẫn không thể hỗ trợ bạn lấy được tất cả dữ liệu mà bạn muốn để lưu trữ trong cơ sở dữ liệu cục bộ, nơi bạn có thể xử lý dữ liệu bằng SQL.

Chúng ta đang chuyển nhiều hơn đến các kiến ​​trúc trung tâm API và có nhiều lợi ích cho kiến ​​trúc đó, đặc biệt là khi các API này nằm ngoài tầm kiểm soát của chúng ta. Có rất nhiều API hiện nay mà chúng ta không phải xây dựng quá nhiều phần mềm như trước đây. Tuy nhiên, nó không phải là công cụ duy nhất trong bộ công cụ và đối với kiến ​​trúc phụ trợ nội bộ, có những lựa chọn thay thế khác.

#### Kafka như là một Event Store

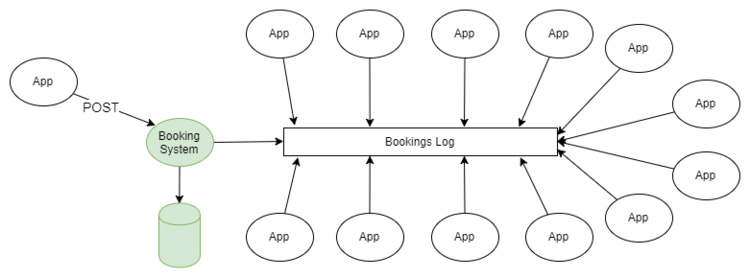
Hãy lấy một ví dụ đơn giản. Chúng ta có một hệ thống quản lý đặt vé được quản lý trong một cơ sở dữ liệu quan hệ. Hệ thống sử dụng tất cả các thuộc tính ACID được cung cấp bởi cơ sở dữ liệu để quản lý trạng thái của nó một cách hiệu quả và nói chung mọi người đều hài lòng với nó. Không sử dụng **CQRS**, không event sourcing, không microservice, chỉ là một ứng dụng truyền thống được xây dựng khá hợp lý. Tuy nhiên, có vô số dịch vụ phụ trợ (có lẽ là microservice) liên quan đến đặt vé như: push notifications, email, chống gian lận, tích hợp với bên thứ ba, chương trình khách hàng thân thiết, lập hóa đơn, hủy vé,... Tất cả chúng đều cần dữ liệu đặt vé và có nhiều cách khác nhau để chúng có được nó. Chúng cũng có thể tự sản xuất dữ liệu của riêng mình, điều này rất hữu ích cho các ứng dụng khác.

Hình 2:



Một kiến ​​trúc thay thế khác đặt Kafka ở trung tâm. Trên mỗi lần đặt vé mới hoặc sửa đổi đặt vé, hệ thống đặt vé sẽ publish toàn bộ trạng thái hiện tại của đặt vé đó cho Kafka. Chúng tôi sử dụng nén log để giảm các thông báo xuống chỉ trạng thái mới nhất của các đặt vé, giữ cho kích thước của log được kiểm soát.

Hình 3:



Theo như tất cả các ứng dụng khác có liên quan thì đây là nguồn dữ liệu đáng tin cậy và chúng tiêu thụ nguồn dữ liệu duy nhất đó. Đột nhiên chúng ta đi từ một mạng lưới phụ thuộc và công nghệ phức tạp để sản xuất và tiêu thụ đến/từ các Topic Kafka.

Kafka như là một event store:

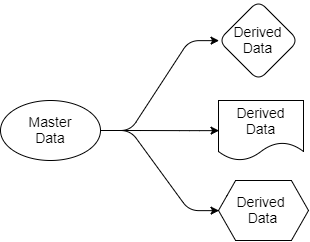
* Nếu dung lượng lớn không phải là một vấn đề, thì Kafka có thể lưu trữ toàn bộ lịch sử của các sự kiện, điều đó có nghĩa là một ứng dụng mới có thể được triển khai và tự khởi động từ log của Kafka. Các sự kiện thể hiện đầy đủ trạng thái của thực thể có thể được nén bằng Log Compaction làm cho phương pháp này khả thi hơn trong nhiều tình huống.
* Các sự kiện cần phải được phát lại theo đúng thứ tự. Miễn là các thông điệp được phân vùng chính xác, chúng ta có thể phát lại các thông điệp theo thứ tự và áp dụng các bộ lọc, biến đổi, v.v để chúng ta luôn kết thúc với đúng dữ liệu ở cuối. Tùy thuộc vào "khả năng phân vùng" của dữ liệu, chúng ta có thể xử lý dữ liệu song song cao, theo đúng thứ tự.
* Mô hình dữ liệu có thể cần phải thay đổi. Có thể chúng ta phát triển một chức năng lọc/biến đổi mới và muốn phát lại trên tất cả các sự kiện hoặc các sự kiện từ tuần trước.

Kafka có thể được cung cấp không chỉ bởi các ứng dụng trong tổ chức của bạn xuất bản tất cả các thay đổi trạng thái của họ (hoặc kết quả của thay đổi trạng thái) mà còn từ tích hợp với các hệ thống của bên thứ ba:

* Các ETL định kỳ trích xuất dữ liệu từ các hệ thống của bên thứ ba và đổ dữ liệu vào Kafka
* Một số dịch vụ của bên thứ ba cung cấp các móc nối web và các dịch vụ web REST được kích hoạt của bạn có thể ghi vào Kafka.
* Một số dịch vụ của bên thứ ba cung cấp các messaging system interface có thể được đăng ký và ghi vào Kafka.
* Một số hệ thống bên thứ ba phân phối dữ liệu qua CSV, có thể được ghi vào Kafka.

Quay trở lại các vấn đề tôi đã giải thích trước đó. Với kiến ​​trúc trung tâm Kafka, chúng ta đơn giản hóa việc phân phối dữ liệu. Chúng ta biết nguồn gốc tin cậy là gì và tất cả các ứng dụng tiếp theo đều hoạt động với các bản sao xuất phát của dữ liệu. Luồng dữ liệu từ publisher đến consumer. Dữ liệu chủ được sở hữu bởi publisher đơn lẻ nhưng những người khác có thể làm việc trên các phép chiếu của dữ liệu đó một cách tự do. Chúng có thể lọc nó, biến đổi nó, gia tăng nó với các nguồn dữ liệu khác và lưu trữ nó trong cơ sở dữ liệu của riêng chúng.

Hình 4:



Mỗi ứng dụng cần đặt chỗ và dữ liệu hoạt động chuyến bay hiện có ứng dụng cục bộ vì nó đăng ký các chủ đề Kafka có chứa dữ liệu đó. Các ứng dụng có thể sử dụng sức mạnh của SQL, Cypher, JSON hoặc bất kỳ ngôn ngữ truy vấn nào. Dữ liệu có thể được cắt và cắt nhỏ hiệu quả vì nó được lưu trữ trong một cấu trúc và một kho lưu trữ dữ liệu được tối ưu hóa cho nhu cầu của nó. Chúng ta có thể thay đổi schema mà không lo ảnh hưởng đến các ứng dụng khác.

Không phải tất cả các ứng dụng cần lưu trữ dữ liệu đó và chỉ có thể phản ứng với dữ liệu có trong mỗi sự kiện. Điều đó là tốt, event sourcing hỗ trợ cả hai mô hình.

Tại thời điểm này, bạn có thể hỏi: Tại sao tôi không thể làm điều này với RabbitMQ? Câu trả lời là bạn có thể sử dụng RabbitMQ để xử lý sự kiện theo thời gian thực, nhưng không phải là một nền tảng event sourcing. RabbitMQ chỉ là một giải pháp hoàn chỉnh để phản ứng với các sự kiện đang xảy ra hiện nay. Khi bạn thêm một ứng dụng mới cần một lát cắt riêng cho tất cả dữ liệu đặt chỗ, ở định dạng được tối ưu hóa cho vấn đề mà nó giải quyết, thì RabbitMQ sẽ không giúp bạn. Với RabbitMQ, chúng tôi quay lại cơ sở dữ liệu dùng chung hoặc sử dụng dịch vụ web REST được cung cấp bởi nguồn dữ liệu.

Thứ hai, thứ tự xử lý sự kiện là rất quan trọng. Với RabbitMQ ngay khi bạn thêm consumer thứ hai vào hàng đợi, bạn sẽ mất mọi đảm bảo về trật tự. Vì vậy, bạn có thể giới hạn RabbitMQ cho một consumer và nhận được trật tự thông điệp chính xác, nhưng cách này lại không có tính mở rộng.

Mặt khác, Kafka có thể cung cấp tất cả dữ liệu mà ứng dụng mới này cần để xây dựng bản sao dữ liệu của riêng mình và luôn cập nhật, với các đảm bảo xử lý thông điệp.

Bây giờ hãy nghĩ lại về kiến ​​trúc trung tâm API. API cho mọi thứ vẫn có vẻ là ý tưởng tốt nhất? Tôi nghĩ rằng khi cần chia sẻ dữ liệu readonly thì tôi thích kiến ​​trúc event sourcing. Chúng tôi tránh các thất bại xếp tầng và mức độ thời gian hoạt động thấp hơn đi kèm với số lượng phụ thuộc lớn hơn vào các dịch vụ khác. Chúng tôi tăng khả năng cắt và xúc xắc dữ liệu một cách sáng tạo và hiệu quả. Nhưng đôi khi bạn cần thay đổi dữ liệu trong một hệ thống khác một cách đồng bộ và sau đó API có ý nghĩa. Một số API thích các phương thức không đồng bộ hơn và tôi nghĩ rằng điều đó có ích.

Cách tiếp cận event sourcing này để tích hợp dữ liệu hoàn toàn phù hợp với microservice và **Self-Contained Systems (SCS)**. Cũng có một [podcast](https://softwareengineeringdaily.com/2017/01/03/self-contained-systems-with-eberhard-wolff/) nói về SCS bạn có thể tham khảo qua.

### #3 - Lưu lượng truy cập cao, xử lý các ứng dụng nhạy cảm

Có một vấn đề có thể xảy ra khi sử dụng RabbitMQ, một ví dụ cụ thể đó là khi các consumer tiêu thụ một queue RabbitMQ mà nguồn cung cấp dữ liệu là từ một bên thứ ba. Số lượng dữ liệu lớn và theo tự nhiên ứng dụng được scale-out để xử lý số lượng thông điệp. Vấn đề ở đây là do dữ liệu trong cơ sở dữ liệu không nhất quán và nó gây ra vấn đề trong nghiệp vụ.

Vấn đề là đôi khi hai dữ liệu liên quan đến cùng một thực thể và chúng đến cách nhau trong vòng vài giây. Cả hai đều được xử lý và do tải trên một máy chủ, thông điệp thứ hai cuối cùng được ghi vào cơ sở dữ liệu và thông điệp đầu tiên sau đó sẽ ghi đè lên thông điệp thứ hai. Vì vậy, thành ra chúng ta sẽ có được dữ liệu không như mong muốn. RabbitMQ đã thực hiện đúng công việc của mình, nó đã gửi các thông điệp theo đúng thứ tự, nhưng cuối cùng ta vẫn chèn dữ liệu theo một thứ tự sai.

Ta có thể giải quyết nó bằng cách kiểm tra nhãn thời gian trên bản ghi hiện có và không chèn nếu thông điệp cũ hơn. Ta cũng có thể giải quyết điều này bằng cách sử dụng Consistent Hashing exchange và phân vùng hàng đợi giống như cách Kafka sử dụng các phân vùng.

Kafka lưu trữ các thông điệp trong một phân vùng theo thứ tự mà chúng được gửi đến nó. Thứ tự thông điệp chỉ tồn tại ở cấp độ của phân vùng. Trong ví dụ ở trên, với Kafka, chúng ta sẽ sử dụng hàm băm trên id thực thể để chọn phân vùng. Ta đã có một loạt các phân vùng, quá đủ để mở rộng quy mô. Lệnh xử lý sẽ đạt được vì mỗi phân vùng chỉ được tiêu thụ bởi một consumer. Điều này khá đơn giản và hiệu quả.

Kafka có một số lợi thế so với RabbitMQ về phân vùng thông qua chức năng băm. Với RabbitMQ, không có gì ngăn cản bạn có những consumer cạnh tranh trong một hàng đợi được cung cấp bởi một Consistent Hashing exchange. RabbitMQ không giúp điều phối consumer để đảm bảo rằng chỉ có một consumer tiêu thụ từ mỗi hàng đợi. Kafka làm điều đó cho bạn với các nhóm consumer và một node điều phối. Vì vậy, chúng ta có thể yên tâm rằng với Kafka, ta có được đảm bảo là một consumer trên mỗi phân vùng và có sự đảm bảo về thứ tự xử lý.

### #4 Dữ liệu cục bộ

Bằng cách sử dụng chức năng băm để định tuyến thông điệp đến các phân vùng, Kafka cung cấp cho chúng ta dữ liệu cục bộ. Ví dụ: các thông điệp liên quan đến id người dùng 1001 luôn đến tay consumer 3. Vì các sự kiện của người dùng 1001 luôn đến consumer 3 có nghĩa là consumer 3 có thể thực hiện một số hoạt động không khả thi nếu network round-trip là cần thiết. Chúng ta có thể thực hiện các bộ đếm, tập hợp thời gian thực và tương tự như trong bộ nhớ. Đây là nơi ranh giới giữa các ứng dụng hướng sự kiện và stream processing pipeline bắt đầu hợp nhất.

Điều này nhắc nhở ta về sticky session trong quá khứ khi mọi người thường lưu trữ dữ liệu phiên làm việc web trong bộ nhớ và sử dụng stateful load balancing để định tuyến các yêu cầu của một người dùng cụ thể đến cùng một máy chủ mỗi lần. Điều đó gây ra tất cả các loại vấn đề như vấn đề mở rộng và tải không cân bằng.

Sticky session có nghĩa là nếu 10 máy chủ của chúng ta bị quá tải và chúng ta đã thêm 10 máy chủ khác, thì chúng ta không thể chuyển qua người dùng hiện tại sang các máy chủ khác vì phiên của họ tồn tại trên 10 máy chủ ban đầu. Chúng ta chỉ có thể di chuyển người dùng mới vào các phiên bản mới được triển khai. Điều này có nghĩa là 10 máy chủ ban đầu của chúng ta vẫn bị quá tải và 10 máy chủ mới không được sử dụng đúng mức. Ngày nay, chúng ta thường có xu hướng đưa session vào **Redis**.

Vậy làm thế nào để Kafka tránh những cạm bẫy tương tự như sticky session? Với Kafka, chúng ta không mở rộng quy mô và trong các phân vùng của mình. Trước hết bạn không thể "chia tỷ lệ" số lượng phân vùng; một khi bạn có 10 phân vùng, bạn không thể xuống 9. Nhưng thứ hai là không cần thiết. Mỗi consumer có thể tiêu thụ 1 đến nhiều phân vùng, vì vậy có rất ít lý do để muốn giảm số lượng phân vùng. Thêm vào đó các phân vùng trong Kafka giới thiệu các đột biến độ trễ trong khi tái cân bằng tải xảy ra, vì vậy chúng ta có xu hướng mở rộng các phân vùng theo tải trọng cao nhất và nhân rộng nhu cầu của consumer.

Nhưng nếu chúng ta cần tăng số lượng phân vùng và consumer cho mục đích mở rộng thì chúng ta chỉ cần trả một chi phí trễ tạm thời trong khi tái cân bằng xảy ra. Lưu ý rằng dữ liệu trong các phân vùng sẽ được đặt nếu chúng ta thêm dữ liệu mới. Không có dữ liệu được cân bằng lại, chỉ có consumer. Nhưng các thông điệp mới đến bây giờ sẽ được định tuyến khác và các phân vùng mới sẽ bắt đầu nhận thông điệp. Điều này cũng có nghĩa là các thông điệp của người dùng 1001 hiện có thể chuyển đến consumer 4 (có nghĩa là người dùng 1001 hiện có hai phân vùng).

Vì vậy, chúng ta cần có khả năng duy trì trạng thái đó ở đâu đó ngay khi tái cân bằng xảy ra vì tất cả các cược đều liên quan đến việc ai sẽ nhận được phân vùng nào sau khi tái cân bằng. Sau khi reloadbalancing kết thúc, chúng ta có thể truy xuất trạng thái của các thông điệp mới đến và tiếp tục hoạt động trong bộ nhớ.

### #5 Failures, Retries và Message Lifecycle

Trong [phần 2](https://dodangquan.blogspot.com/2018/11/rabbitmq-va-kafka-phan-2-cau-truc-cua-rabbitmq-va-cac-messaging-pattern-khi-su-dung-rabbitmq.html), tôi đã kết thúc bài viết với một vòng đời thông điệp hoàn chỉnh xử lý các lỗi thông qua sự kết hợp của:

* Thử lại ngay lập tức
* Tạm dừng tiêu thụ trong một khoảng thời gian cấu hình
* Hoãn thử lại
* Loại bỏ thông điệp
* Đóng gói các thông điệp dưới dạng các Failed Event và chuyển đến các kỹ sư hỗ trợ để phân loại và phản hồi.

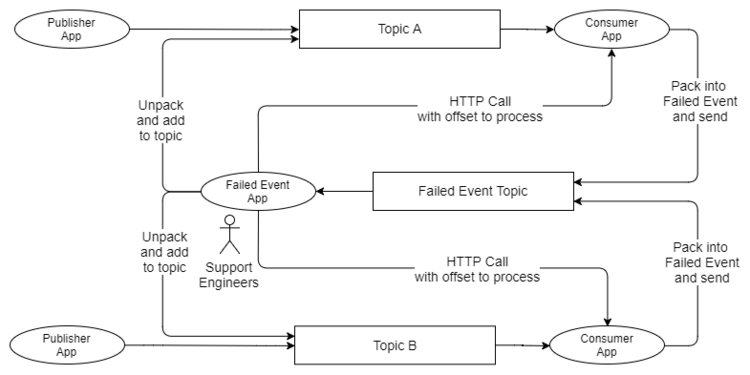
Với Kafka chúng ta có thể có một cách tiếp cận tương tự. Sự khác biệt là bản thân các thông điệp không di chuyển bất cứ nơi nào, chúng được đặt trong phân vùng log của chúng. Mặc dù vậy, chúng có thể được dọn sạch, vì vậy chúng ta sẽ phải tính đến cả hai tình huống đó. Pattern delay-retry không có ý nghĩa nhiều với Kafka. Thay vào đó, chúng ta có thể dựa vào thử lại ngay lập tức, tạm dừng tiêu thụ và gửi thông điệp đến Failed Event topic.

Thử lại ngay lập tức sẽ giải quyết các vấn đề như các vấn đề sẵn có trong thời gian rất ngắn với các dịch vụ khác và những thứ như bế tắc cơ sở dữ liệu.

Nếu vấn đề còn tồn tại lâu hơn thế thì consumer có thể tạm dừng một phút. Nó có thể thử lại thông điệp cứ sau 1 phút, hoặc một số loại backoff có thể được thực hiện. Nếu vấn đề là API không khả dụng trong vài phút thì chiến lược này sẽ giải quyết vấn đề.

Nếu sự cố vẫn còn trong một khoảng thời gian nhất định thì các thông báo sẽ được đưa đến các kỹ sư hỗ trợ để họ có thể bắt đầu điều tra vấn đề.

Hình 5:



Đôi khi vấn đề không nhất thời và chỉ ảnh hưởng đến một số ít thông điệp. Có lẽ thông điệp có dữ liệu xấu gây ra bởi một lỗi trong publisher gửi thông điệp. Có lẽ có một lỗi trong consumer chỉ xảy ra khi xử lý một tập hợp nhỏ các thông điệp. Trong trường hợp này, chúng ta có thể thích chuyển sang các thông điệp tiếp theo và tiếp tục tiêu thụ các thông điệp. Chúng ta có thể đóng gói thông điệp thất bại trong một Failed Event và gửi nó đến Failed Event topic.

Những sự kiện thất bại này chứa:

* Ứng dụng tiêu thụ
* Thông báo ngoại lệ hoặc lỗi
* Máy chủ
* Thời gian
* Ai công bố thông điệp
* Thông điệp này đã được thử bao nhiêu lần rồi

Điều này làm cho chẩn đoán đơn giản hơn nhiều so với log lỗi tương quan với dấu thời gian thông báo. Từ topic đó, các kỹ sư hỗ trợ có thể sử dụng ứng dụng xử lý thông điệp, phản hồi và chọn:

* Không làm gì cả
* Nếu sự cố được giải quyết và ứng dụng khách hàng sẽ hiển thị dịch vụ web REST, ứng dụng xử lý có thể gọi dịch vụ chuyển qua phần bù của thông báo sẽ được thử lại.
* Nếu sự cố được giải quyết và thông điệp gốc đã được xóa, ứng dụng xử lý có thể giải nén thông điệp gốc từ Sự kiện thất bại và gửi lại cho chủ đề ban đầu.

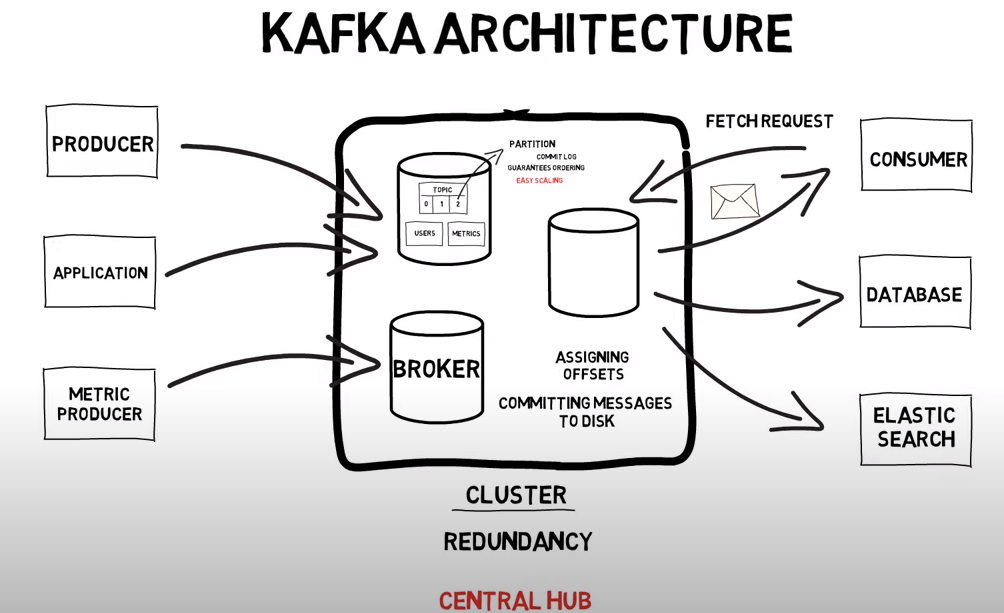
Khi có sự cố xảy ra, cách chúng tôi phản ứng phụ thuộc hoàn toàn vào domain cụ thể đó là gì. Nếu đó là một thông báo rằng chuyến bay của bạn sẽ khởi hành sau 30 phút thì có nghĩa là thử lại thông điệp 4 giờ sau không? Hoặc có thể đó là một loại thông điệp mà trong mọi trường hợp không thể bị mất. Cho phép các đội hỗ trợ xem những thông điệp nào không thể được xử lý và để họ đưa ra quyết định dựa trên kiến ​​thức về domain của họ đôi khi là lựa chọn duy nhất. Các kỹ sư hỗ trợ có thể tự động hóa các phản ứng đối với các sự cố cụ thể theo thời gian có thể giảm bớt gánh nặng. Nhưng các nhà phát triển ứng dụng cũng có thể tự động hóa quá trình ra quyết định này. Sử dụng cùng một ví dụ "30 phút cho đến khi thông báo chuyến bay của bạn", nhà phát triển có thể tạo logic khiến consumer loại bỏ thông điệp, ghi lại lỗi và tiếp tục.

# OTHER

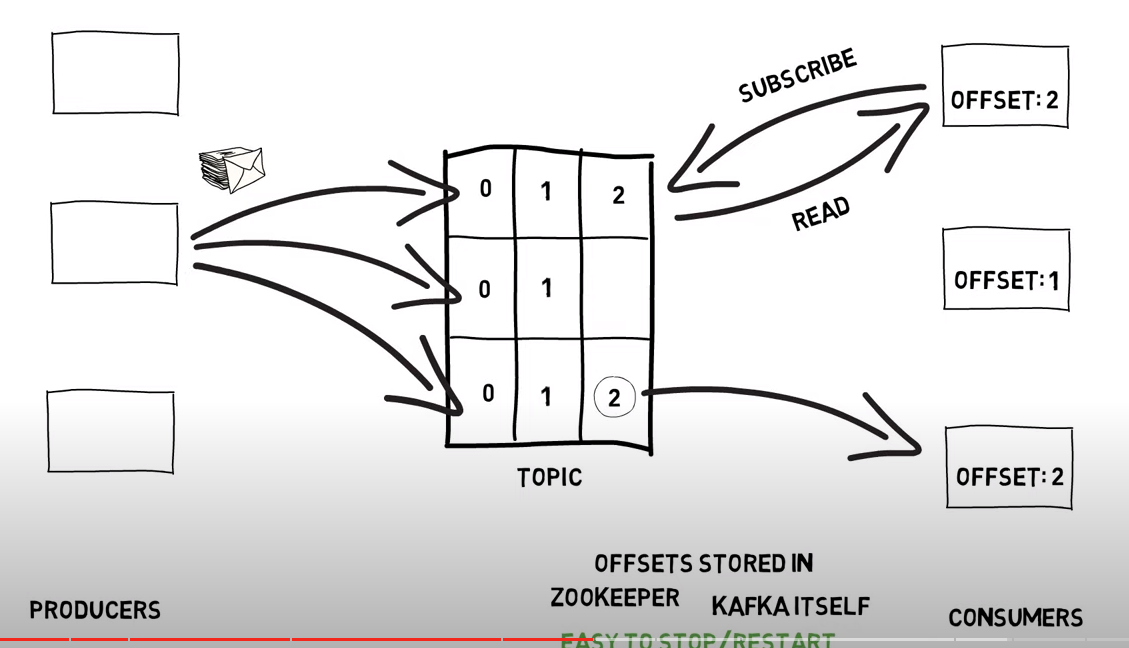
## What is kafka

* + is open-source, publis/subscriber message system, distributed event log(imutable message)
  + have **retention policy**( pesisten on DISK) -> can be hybid system: database + message system
  + Architecture: Producer(publishers) -> TOPIC -> Consumer
  + Reliable and high-throwput platform
  + Realtime data strem and data pipelines
  + Can be single place for storing and distributing event
  + Using by many big company: twitter, netflix…

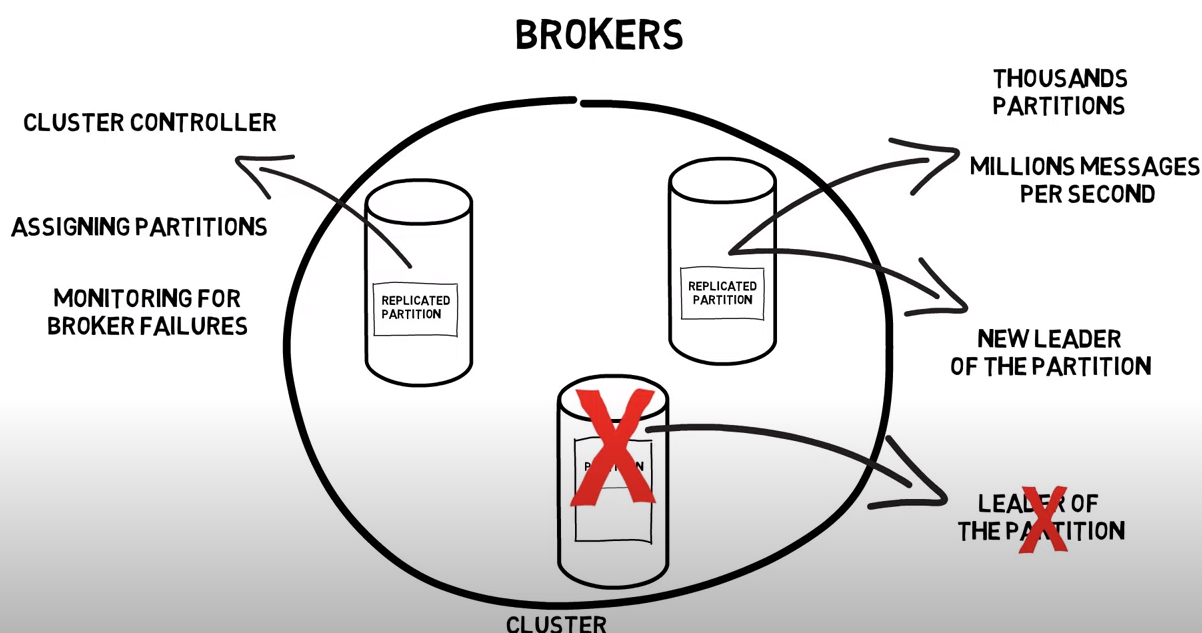
## Kafka Architecture



* + **MESSAGE:**  is a single unit of data can be send and receive
    - Just a byte array, can have optional key to control routing way to mutil partition of TOPIC
    - No ordering guarantee in topic

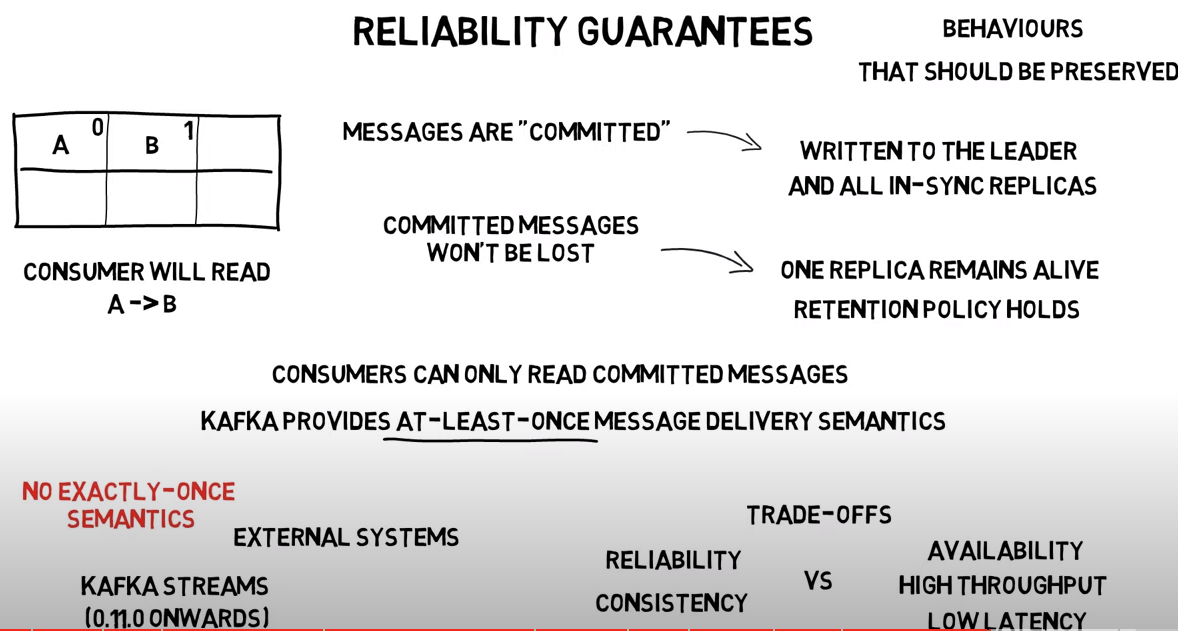


H1. Message store in TOPIC and push for subsciber

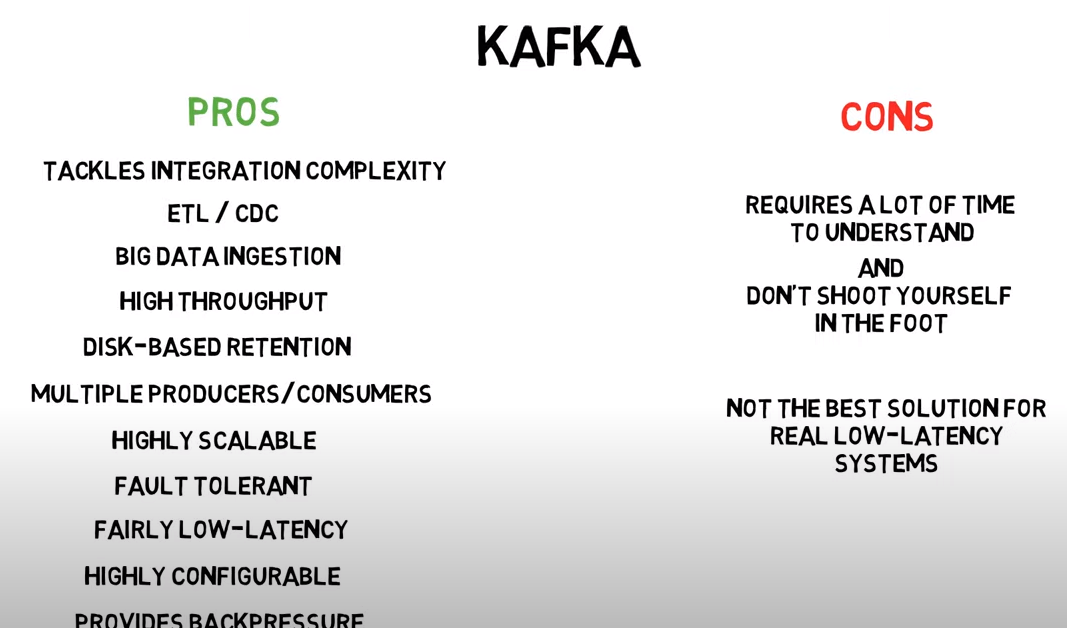


H2. Cluster and voting master merchandise in KAFKA

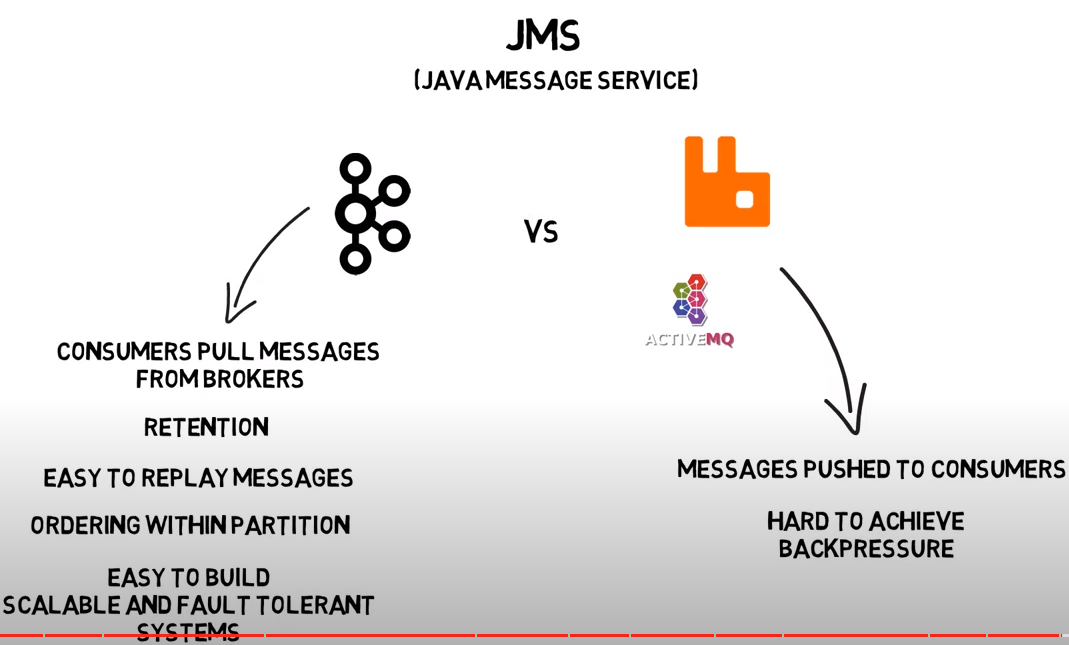
* Rentention:
  + Rentention setting for topic: time, size
  + Durables storage rage of message: oldest message are expired and delete,
* Reliability Guarantees:



* PROS and CONS



* JMS: KAFKA vs RABBITMQ



## Cluster and HA RabbitMQ

### Cluster

* + Mặc định các queue sẽ chỉ nằm trên 1 node của cluster, còn exchange và binding luôn nằm trên mọi node
  + Ta cũng có thể cấu hình queue sẽ nằm trên mọi node sử dụng cơ chế “mirror queue” trong đó:
    - bao gồm 1 master và nhiều slave
    - slave đầu tiên sẽ lên làm master mới nếu master cũ bị mất
    - Message được publish tới master sẽ được sao bản sang các slaver, tuy nhiên slave sẽ drop các message này
    - Mirror queue chỉ đảm bảo độ sẵn sàng cao chứ không đóng vai trò chia tải
  + *Link tham khảo:* [*https://www.rabbitmq.com/clustering.html*](https://www.rabbitmq.com/clustering.html)

RabbitMQ cluster là một nhóm logic các các node khác nhau mà chúng cùng chung user, máy chủ ảo, queue, exchange, bindings, tham số môi trường và các trạng thái phân tán.

### Sự hình thành cluster:

* + Các phương pháp tạo RabbitMQ Cluster
    - Declaratively by listing cluster nodes in *config file*
    - Declaratively using *DNS-based discovery*
    - Declaratively using *AWS (EC2) instance discovery* (via a plugin)
    - Declaratively using *Kubernetes discovery* (via a plugin)
    - Declaratively using *Consul-based discovery* (via a plugin)
    - Declaratively using *etcd-based discovery* (via a plugin)
    - Manually with *rabbitmqctl*
  + Tất cả các RabbitMQ broker đều khởi động lên như một node đơn lẻ, chúng có thể được tập hợp vào cùng một cụm(cluster) và tách ra thành các node đơn lẻ trở lại.

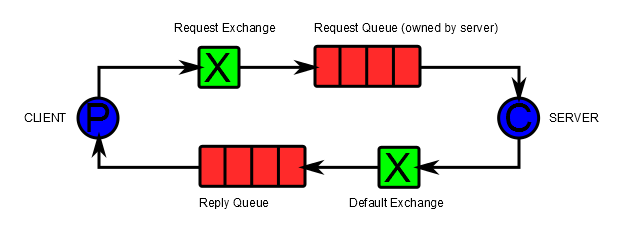
### Định nghĩa tên các node

* + RabbitMQ node được định danh dựa trên tên node, tên node bao gồm 2 phần: tiền tố (thông thường là rabbit) + hostname. Ví dụ rabbit@node1.messaging.svc.local
  + Tên node trong 1 cụm phải là duy nhất, trường hợp các node trên cùng máy chủ thì sẽ phải khác tiền tố. Ví dụ: rabbit1@hostname và rabbit2@hostname
  + Trong 1 cụm, các node định danh và liên lạc với nhau qua tên node vì thế tên node phải được khai báo và map với IP của máy chủ
  + Khi 1 node được khởi động lên, nó sẽ check xem đã được gán cho tên node, nó được thực hiện thông qua biến môi trường RABBITMQ\_NODENAME, nếu chưa giá trị nào được cấu hình, nó sẽ xử lý hostname và thêm tiền tố rabbit để tạo ra rabbit hostname

### Yêu cầu về hình thành cụm

* + Xử lý hostname

## Request-Respose Model with Message Broker

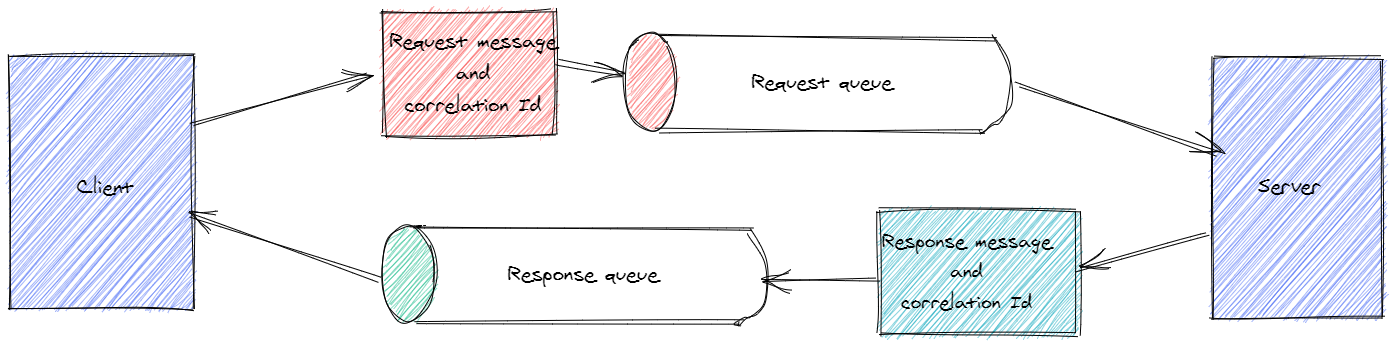


*Hinh: Mô hình request-response sử dụng message queue*

* + Với các hệ thông điệp thông thường, mô hình thường được sử dụng là gửi-nhận. Các producer sẽ gửi các thông điệp vào các queue và được xử lý bởi các consumer.
  + Sẽ không có khái niệm rõ ràng về client-server bởi vì 1 node có thể đóng vai trò là client hoặc server hoặc cả 2.
  + Mô hình request-response kết hợp với phần mềm message queue bằng cách: các client node sẽ tạo các các “reply queue” để chứa các phản hồi cho các request nó gửi đến server. Các client sẽ phải thiết lập thông tin về reply queue bằng cách thêm thông tin trường “reply\_to” vào trong request nó gửi đi.
  + Server sẽ kiểm tra thông tin “reply\_to ” và thực hiện gửi các phản hồi tương ứng vào “reply queue” thông qua default exchange, các phản hồi này sẽ được gửi cho client.
  + Với chiều client gửi request cho server thì mô hình đơn giản, tuy nhiên có một số vấn đề với chiều server gửi phản hồi cho client:
    - Nên sử dụng một queue/ request hay sử dụng chung 1 queue cho các request?
    - Vấn đề về xóa queue: hàng đợi chỉ nên độc quyền cho 1 kênh hay không. Chú ý rằng queue thực thi sẽ bị xóa trong trường hợp kết nối bị đóng.
* Chúng ta sẽ đi xem xét ưu và nhược điểm của các thiết kế:
  + Mô hình 1 reply queue/ 1 request: mỗi request sẽ tạo ra 1 hàng đợi để chứa thông tin được phản hồi cho nó.
    - Ưu điểm:
      * Lợi ích của việc này là dễ triển khai, không cần phải quan tâm đến việc tạo ràng buộc giữa request và response
    - Nhược điểm:
      * Trường hợp mất kết nối trước khi client nhận được phản hồi thì các response trong queue sẽ bị loại bỏ
      * Vấn đề về hiệu năng khi liên tục phải tạo và xóa các queue
  + Mô hình 1 reply queue/1client:Mỗi client kết nối đến sẽ duy trì 1 reply queue được sử dụng chung bởi các request.
    - Ưu điểm:
      * giải quyết được vấn để hiệu năng khi phải tạo và xóa quá nhiều queue
      * Không cần quan tâm đến việc xóa queue khi đóng hoặc mất kết nối giữa client và server vì các broker sẽ làm điều đó
    - Nhược điểm:
      * Client phải theo dõi reply queue để tìm ra được response phù hợp với request. Các cơ bản nhất là sử dụng 1 id tương ứng giữa request và reponse
      * Các queue sẽ tự động xóa trong trường hợp mất kết nối vì thế có thể dẫn đến trường hợp mất phẩn hồi
  + Mô hình reply queue tập trung:
    - Ưu điểm:
      * Giải quyết được vấn đề mất reponse trong trường hợp mất kết nối
    - Nhược điểm
      * Ngoài vấn đề phải map được request với response thì phải map được client tương ứng

## Request/Response patter với Spring AMQP

* + [Request/Response Pattern with Spring AMQP (reflectoring.io)](https://reflectoring.io/amqp-request-response/)
* Request/response pattern là gì: Là dạng giao tiếp giữa client và server: client gửi yêu cầu đến server , server thực hiện yêu cầu và phải hồi cho client một cách đồng bộ: một ví dụ phổ biến hiện nay là giao tiếp qua giao thức HTTP
* Tại sao chúng ta cần mẫu req/res bất đồng bộ: các hệ thống phần mềm hiện tại được xây dựng với rất nhiều thành phần và chúng có thể giao tiếp với nhau.
* Giao tiếp bất đồng bộ: khi ta sử dụng thông điệp cho req và res, việc giao tiếp trở nên bất đồng bộ
  + mô hình sẽ như sau:



* + - B1: client gửi yêu cầu vào kênh yêu cầu
    - B2: server sẽ tiêu thụ (consumer) các yêu cầu từ kênh yêu cầu
    - B3: Server gửi phản hồi đến các kênh phản hồi
    - B4: client tiêu thụ các phản hồi từ kênh phải hồi
  + Khi client gửi 1 request néo sẽ đợi res bằng việc nghe từ kênh response,
  + Trong trường hợp client gửi nhiêu req và yêu cầu tương ứng mỗi res với các req đó thì làm sao client có thể phân biệt được res nào của req nào. Để giải quyết trường hợp này thì client sẽ phải gửi ID cùng với req, ID này phải là duy nhất. Server sẽ dùng ID này và thêm vào response và client có thể lấy ra response tương ứng
  + Một điểm cần lưu ý là client phải có 1 trạng thái, client phải tạo rea id duy nhất, gửi cùng với req và phải lưu id này trong mem hoặc DB, sau đó client đợi và tìm kiếm response tương ứng trong kênh phản hồi dựa vào ID.

# Cấu hình và cài đặt

## Lệnh RabbitMQ

* Run RabbitMQ in Docker

docker run --rm -it -p 15672:15672 -p 5672:5672 rabbitmq:3-management