Why Do We Need a Message Queue?

BYTEBYTEGO
AUG 10, 2023 · PAID

In this issue, we're diving deep into a widely-used middleware: the message queue.

Message queues have a long history. They are often used for communication between different systems. Figure 1 illustrates the concept of a message queue by comparing it to how things work at Starbucks.

At Starbucks, the cashier takes the order and collects money, then they write the customer's name on a coffee cup to hand over to the next step. The coffee maker picks up the order and the cup and makes coffee. The customer then picks up the coffee at the counter. The three steps work asynchronously. The cashier just drops the order in the form of a coffee cup and does not wait for its completion. The coffee maker just drops the completed coffee on the counter and does not wait for the customer to pick it up.

When you place an order at Starbucks, the cashier takes the order and scribbles your name on a cup and moves to the next customer. A barista then picks up the cup, prepares your drink, and leaves it for you to collect. The beauty of this process is that each step operates independently. It is much like an asynchronous system.

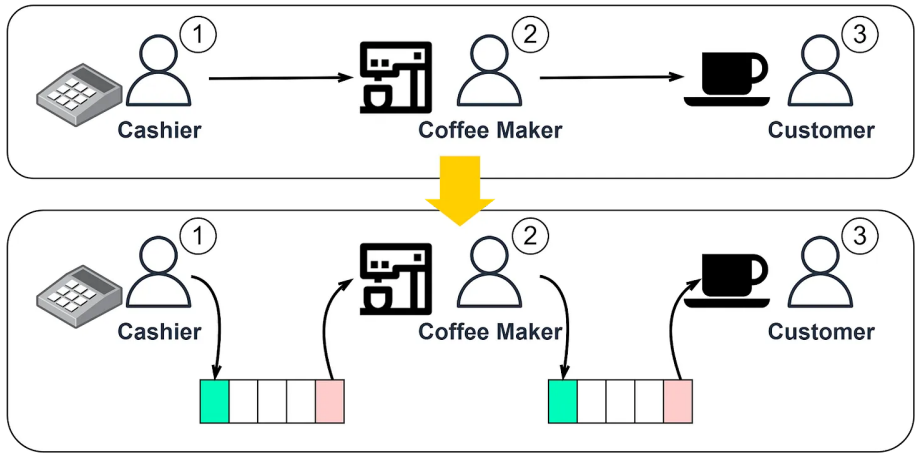


Figure 1 Starbucks as an analogy for message queues

This asynchronous processing, where each step doesn't have to wait for the previous one to complete, significantly increases the throughput of the system. For instance, the cashier doesn't wait for your drink to be made before taking another order.

A Message Queue Example

Now, let's shift our focus to a real-world example: flash sales in e-commerce. Flash sales can strain systems due to surge in user activity. Many strategies are employed to manage this demand, and message queues often play a pivotal role in backend optimizations.

A simplified eCommerce flash sale architecture is listed in Figure 2.

Steps 1 and 2: A customer places an order to the order service.

Step 3: Before processing the payment, the order service reserves the selected inventory.

Step 4: The order service then sends a payment instruction to the payment service. The payment service fans out to 3 services: payment channels, notifications, and analytics.

Steps 5.1 and 6.1: The payment service sends the payment instruction to the payment channel service. The payment channel service talks to external PSPs (Payment Service

Providers) to finalize the transaction.

Steps 5.2 and 6.2: The payment service sends a notification to the notification service, which then sends a notification to the customer via email or SMS.

Step 5.3: The payment service sends transaction details to the analytics service.

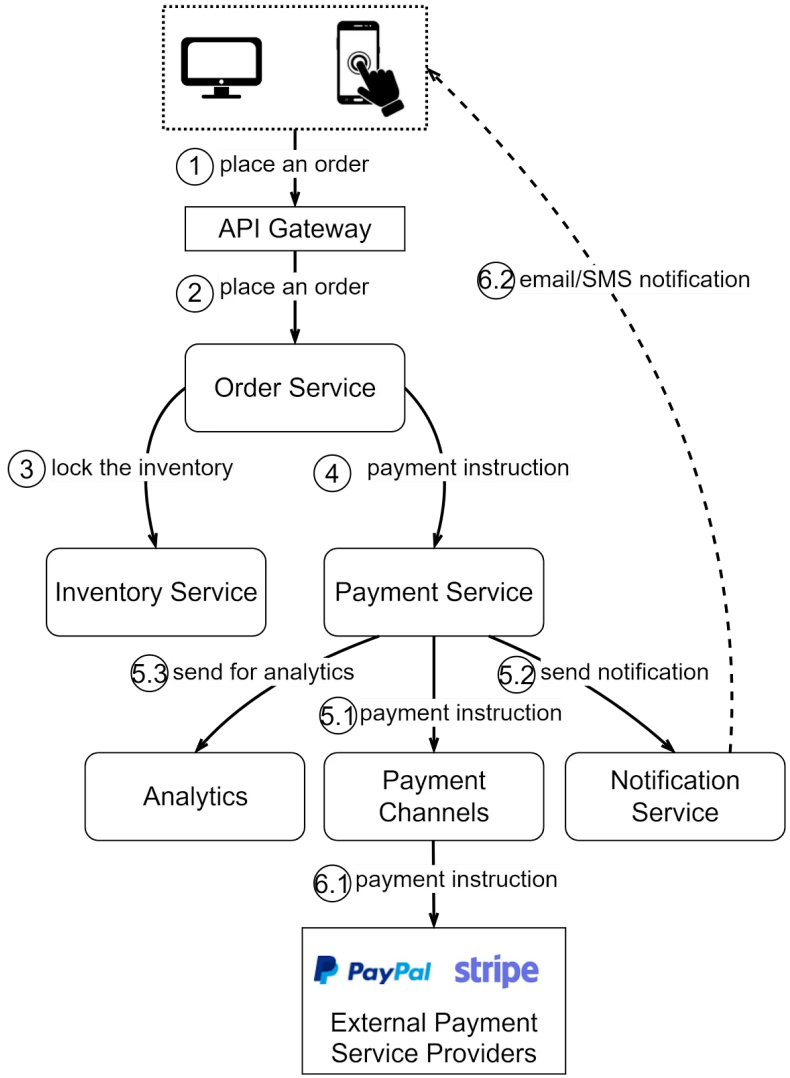


Figure 2 A simplified eCommerce flash sale architecture

A key takeaway here is that a seamless user experience is crucial during flash sales. To maintain service responsiveness despite high traffic, message queues can be integrated at multiple stages to ensure optimal performance.

Benefits of Message Queues

Fan-out

The payment service sends data to three downstream services for different purposes: payment channels, notifications, and analytics. This fan-out method is like someone shouting a message across a room; whoever needs to hear it, does. The producer simply drops the message on the queue, and the consumers process the message at their own pace.

Asynchronous Processing

Drawing from the Starbucks analogy, just as the cashier doesn't wait for the coffee to be made, the order service does not wait for the payments to finalize. The payment instruction is placed on the queue, and the customer is notified once it's finalized.

Rate Limiting

In a flash sale, there can be tens of thousands of concurrent users placing orders simultaneously. It is crucial to strike a balance between accommodating eager customers and maintaining system stability. A common approach is to cap the number of incoming requests within a specific time frame to match the capacity of the system. Excess requests might be rejected or asked to retry after a short delay. This approach ensures the system remains stable and doesn't get overwhelmed. For requests that make it through, message queues ensure they're processed efficiently and in order. If one part of the system is momentarily lagging, the order isn't lost. It's held in the queue until it can be processed. This ensures a smooth flow even under pressure.

Decoupling

Our design uses message queues in various places. The overall architecture is different from the simplified version presented in Figure 2. Services interact with each other using well-defined message interfaces rather than depending tightly on each other. Each service can be modified and deployed independently. Each

component can be developed in a different programming language. This brings flexibility to the architectural design.

Horizontal Scalability

Since the services are decoupled, we can scale them independently based on demand. Each service can serve in a different capacity, so we can scale based on their planned QPS (query per second) or TPS (transaction per second).

Message Persistence

Message queues can also be used as middleware that stores messages. If the upstream service crashes, the downstream service can always pick up the messages from the message queue to process. In this way, the recovery function is moved out of each service and becomes the responsibility of the message queue.

Batch Processing

Sometimes in the processing flow, we need to batch the data to get the summary. For example, when the payment service sends updates to the analytics service, the analytics service does not need to perform real-time updates but rather set up a tumbling window to process in batches. The batch processing is the requirement of the downstream services, so there is no need for the payment service to know about it, just drop the messages into the queue.

Message Ordering

In a flash sale, there is a limited number of inventory items. For example, a flash sale offers only 10 iPhones, but there are over 10,000 users who place the order. How do we decide on the order? Having a message queue to keep all the orders will have a natural order: The first 10 in the queue will get the iPhone.

In Figure 3 we put everything together, where the services are connected via message queues and decoupled. In this way, the architecture can achieve higher throughput.

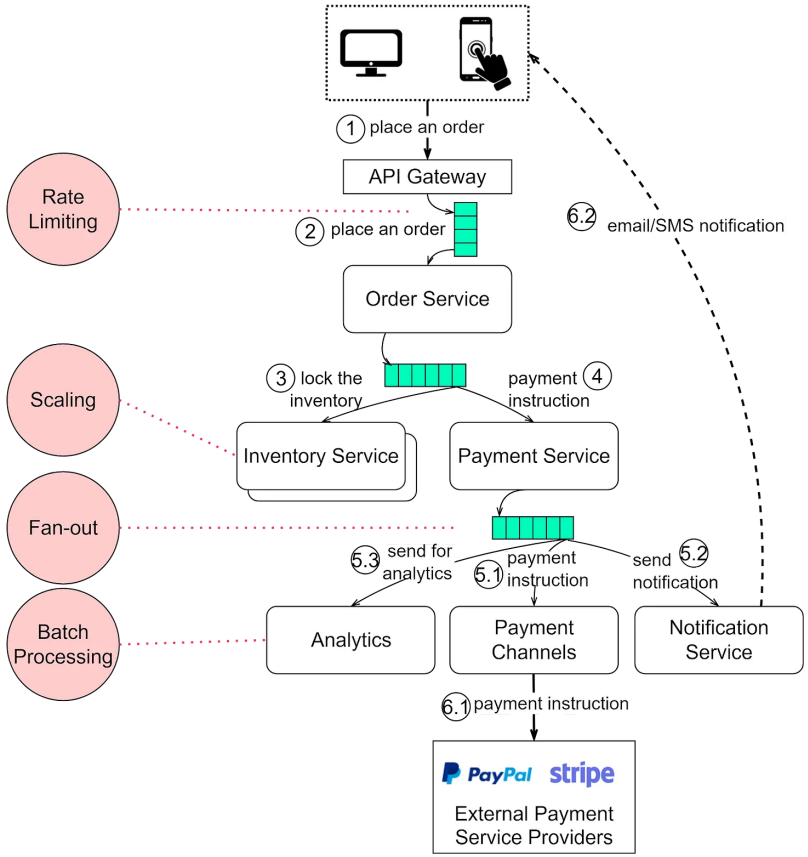


Figure 3 Use message queues for flash sale architecture

A Brief History of Message Queues

The last section discussed the benefits of adopting message queues in the architecture. Message queues, however, are not always the Kafka architecture we are familiar with. Let's walk through the history of message queues.

Figure 4 shows a brief history of message queues over the past 30 years.

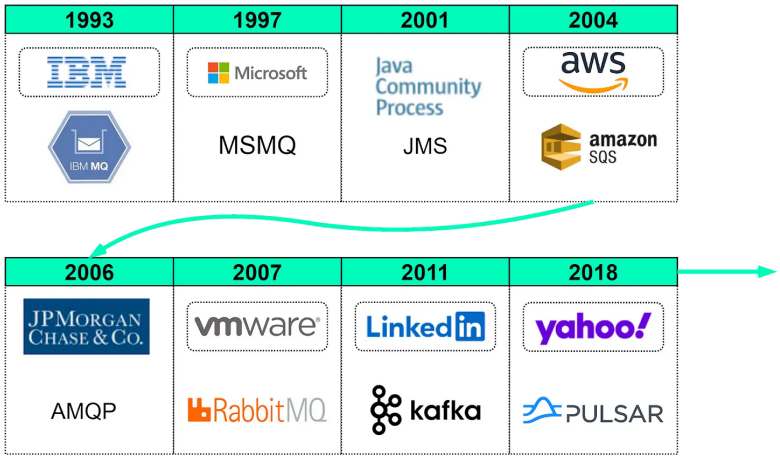


Figure 4 A brief history of message queues

IBM MQ was launched in 1993. It was originally called MQSeries and was renamed WebSphere MQ in 2002. It was renamed to IBM MQ in 2014. IBM MQ is a very successful product widely used in the financial sector. Its revenue still reached 1 billion dollars in 2020. Figure 5 shows the key concepts in IBM MQ.

The queue manager is a logical container for message queues. It transfers data to other queue managers via message channels. Queue stores messages. The message is an abstraction for the data to be transmitted. The message header holds the routing, storage, and delivery information.

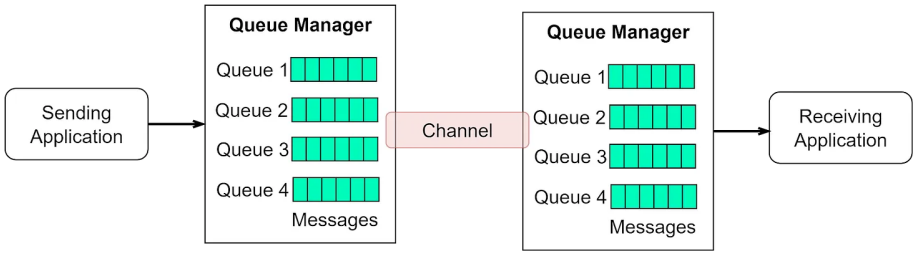
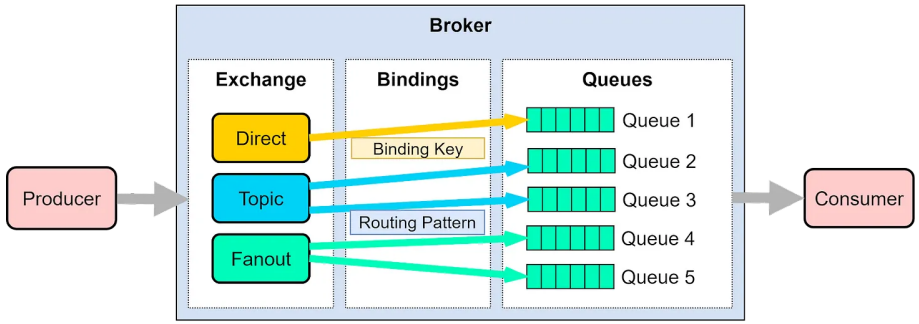


Figure 5 Key concepts in IBM MQ

There are other non-open source message queues, like MSMQ (1997) and SQS (2004), both serving well in their ecosystem.

In 2003, multiple institutions in the financial sector wanted to develop a standardized messaging protocol, so AMQP (Advanced Message Queuing Protocol) was born in JP Morgan Chase. Unlike JMS (Java Messaging Service), which is standardized at the API level, AMQP is a wire-level protocol, which means it specifies the format of the data to be transferred. As one of the implementations of AMQP, RabbitMQ was developed in 2007 by Rabbit Technologies, which was acquired by VMWare later.

Figure 6 shows RabbitMQ architecture. As we can see, it differs from IBM MQ and is more similar to Kafka concepts. The producer publishes a message to an exchange with a specified exchange type. It can be direct, topic, or fanout. The exchange then routes the message into the queues based on different message attributes and the exchange type. The consumers pick up the message accordingly.



Source: https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html

Figure 6 RabbitMQ Architecture

Although RabbitMQ has a lot of modern message queue concepts, it was developed nearly 20 years ago. At that time, the distributed systems were not as mature as today, so the architecture is restricted to handle scenarios with high volumes and a large number of concurrent requests.

In early 2011, LinkedIn open sourced Kafka, which is a distributed event streaming platform. It was named after Franz Kafka. As the name suggested, Kafka is optimized for writing. It offers a high-throughput, low-latency platform for handling real-time data feeds. It provides a unified event log to enable event streaming and is widely used in internet companies. Figure 7 shows a simplified Kafka architecture.

We will go through Kafka's architecture in more detail, but in general, Kafka defines producer, broker, topic, partition, and consumer. Its simplicity and fault tolerance allows it to replace previous products like AMQP-based message queues.

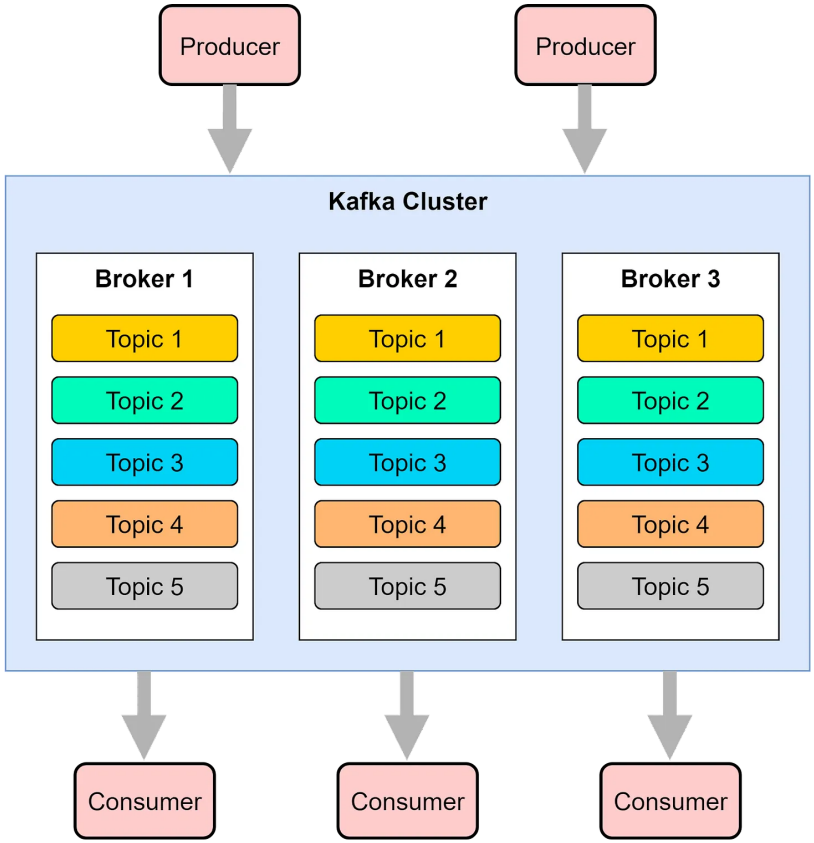


Figure 7 Simplified Kafka architecture

Pulsar, developed originally by Yahoo, is an all-in-one messaging and streaming platform. Compared with Kafka, Pulsar incorporates many useful features from other products and supports a wide range of capabilities. Also, Pulsar architecture is more cloud-native, providing better support for cluster scaling and partition migration, etc. Figure 9 shows a simplified version of Pulsar architecture.

Similar to Kafka, Pulsar has the concept of the topic. The URI looks like this:

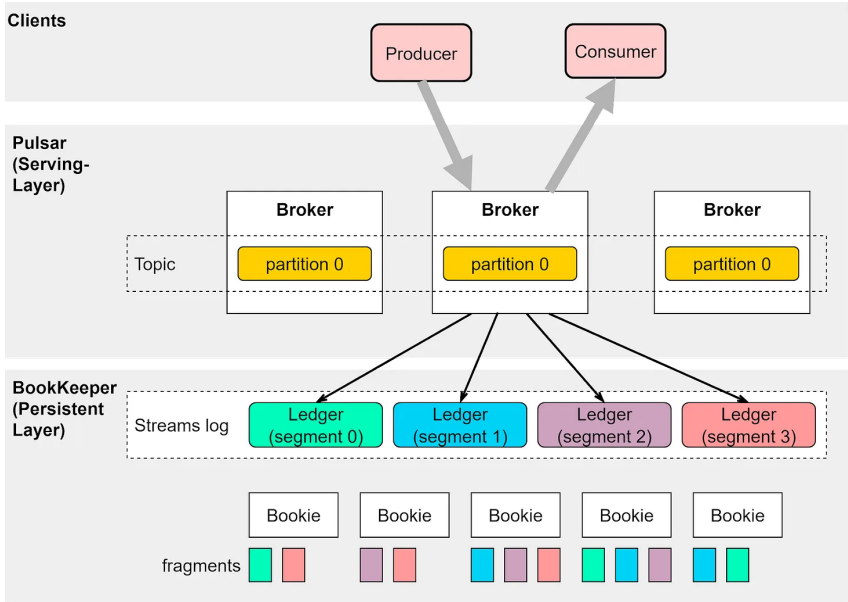

```
{type}://{tenant}/{namespace}/{topic}
```

Note that there is a tenant element in the URI, which means Pulsar supports a multi-tenant environment.

Pulsar also supports either persistent or non-persistent topics. Persistent topics persist and replicate messages on disks, while non-persistent topics reside in the memory and might be lost if there is a failure.

There are two layers in Pulsar architecture: the serving layer and the persistent layer. The serving layer is composed of multiple brokers, handling incoming and outgoing messages. The serving layer is stateless. It leverages the persistent layer to store messages via Apache BookKeeper.

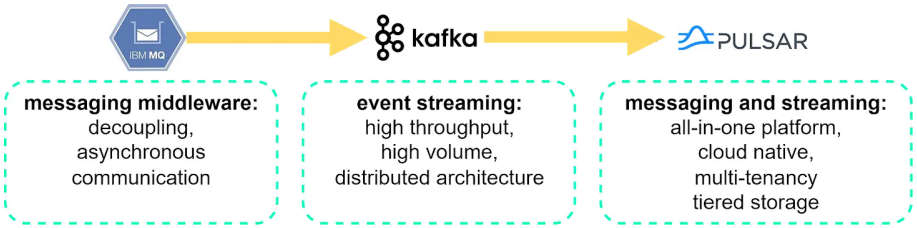
Another interesting design is that Pulsar natively supports tiered storage, where we can leverage cheaper object storage like AWS S3 to persist messages for a longer term.



https://medium.com/streamthoughts/introduction-to-apache-pulsar-concepts-architecture-java-clients-71f1a30b75d6

Figure 9 Pulsar architecture

In conclusion, message queues evolved from messaging middleware to stream handling. Modern message queues normally combine the two functions together and support fault tolerance in a distributed environment. We close this issue with the diagram below: the birth of each popular product changes the programming paradigm of message queues and solves a business pain point.



318 Likes · 7 Restacks

5 Comments

Write a comment...

AJ Renold Aug 11, 2023

The source post for the Apache Pulsar architecture image has some great content to dive deeper on that particular system - https://medium.com/streamthoughts/introduction-to-apache-pulsar-concepts-architecture-java-clients-71f1a30b75d6

LIKE (2) REPLY SHARE

rocky Aug 13, 2023

I dont understand why queue is placed between api gateway and order service

LIKE (1) REPLY SHARE

3 replies

3 more comments...