Jahong Liu

System Design Basics 19: Message Queues

Message Queues

Message queues offer a solution for scenarios when an application server faces a high volume of requests that it can't process simultaneously. Why not just scale our server horizontally or vertically? While this approach is possible, it's not always cost-effective or practical. Additionally, there might be instances where immediate processing of theses requests isn't necessary, allowing them to be queued for later handling.

Message queues serve to decouple producers (app events) and consumers (application servers), functioning as a buffer for managing surges in data.

Example

Payment processing serves as a good illustration of a system where message queues can provide considerable benefits.

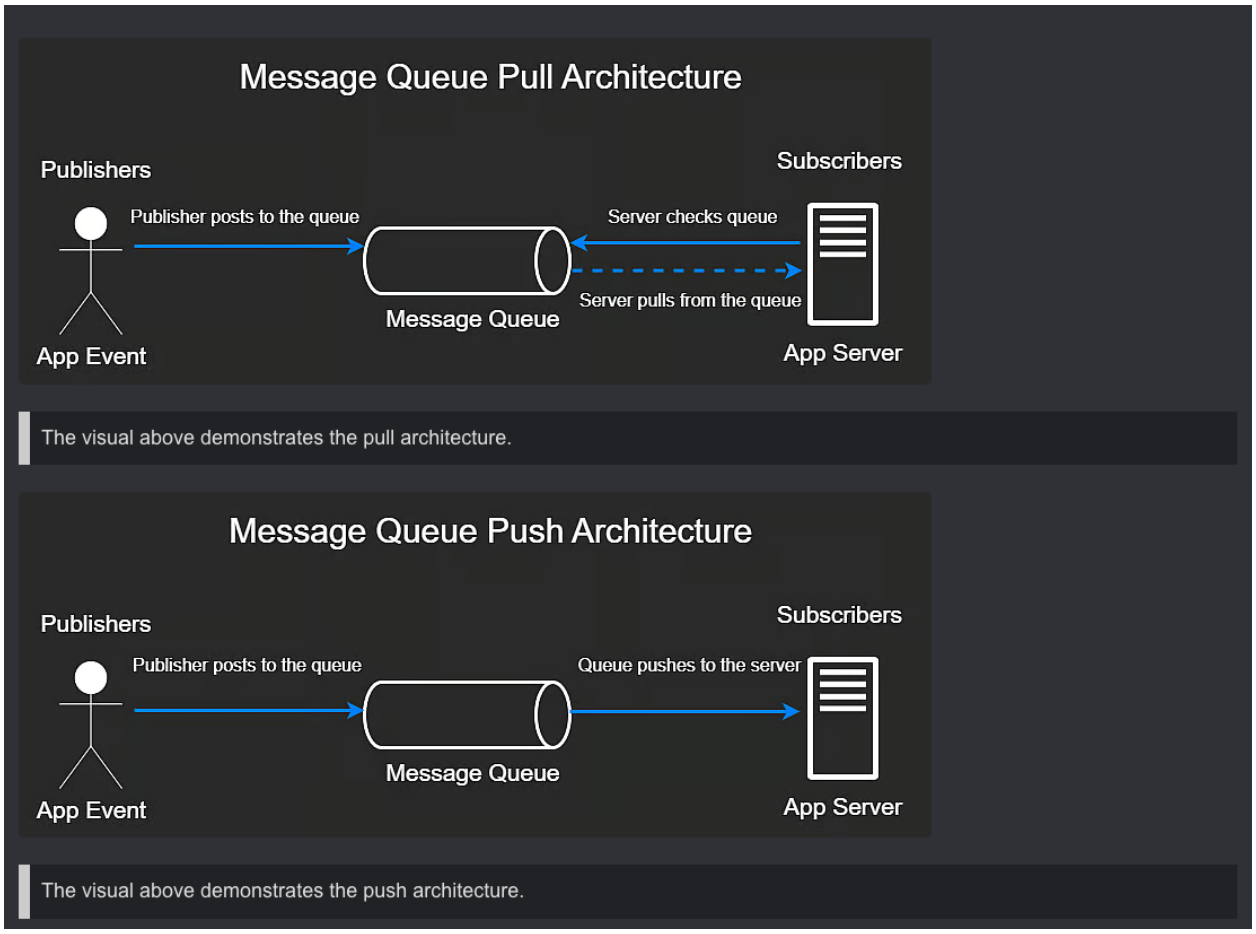
1. **Handling Peaks in Load**: During high usage periods, such as major sale, the number of payment requests typically increases significantly. If these requests were to be processed synchronously, it could result in a poor user experience due to prolonged wait times and potential timeouts. However, with message queues, payments can be stored and processed asynchronously.
2. **Decoupling Services**: When a new order is placed, a message can be published to the queue. The payment service, acting as a subscriber, can then process the payment and update the order status.

Push/Pull model

There are several methods through which a message queue can interact with the application server. Depending on the service and the system's architecture, updates can be either pull-base or push-based.

1. **Pull-Based Model**: In this scenario, the application is responsible for monitoring the message queue for any new messages. If new messages are present and the app server has the capacity, it "pull" from the queue. This approach can be more efficient in terms of managing the server-side load. However, if the queue is empty, it may introduce latency.

2. **Push-Based Model**: In this case, the queue takes on the responsibility of pushing messages to the server. But, this strategy might overload the server if the rate of incoming messages is excessively high.



When the message queue dispatches a message to the application server, the consumer or the server sends an acknowledgement after successfully processing a message (similar to what we discussed in the networking section).

If the queue does not receive an acknowledgement for a message within a specific time frame, it can infer that the message was not processed, prompting the queue to resend it. This approach, akin to what we discussed in the networking chapter, ensures message delivery, even in the event of temporary server issues.

Push/Sub model

The publisher/subscriber model provides decoupling of publishers, eliminating the need for either to be aware of each other's existence. This allows for easy system scalability and ensures messages are not lost if a subscriber is temporarily unable to process them.

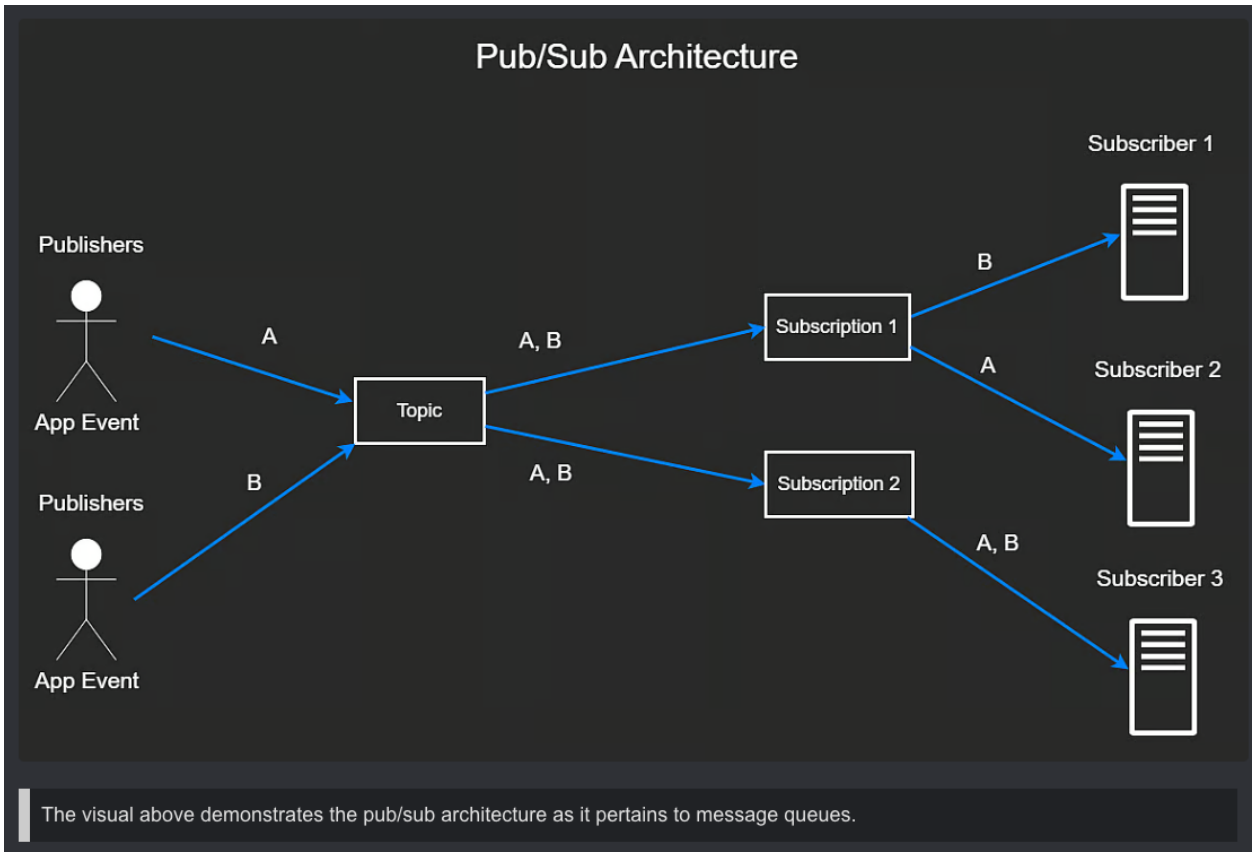
In the context of the message queue, the pub/sub will go through the following steps.

1. The publisher dispatches messages to a specific queue or topic.
2. One or more subscribers listen to the specific queue or topic.
3. The message broker ensures all messages published to a topic are successfully delivered to all subscribers of that topic. Subscribers process messages independently and at their own pace.

In the context of the pub/sub model, publisher dispatch their messages to specific topics, and subscribers indicate their interest by subscribing to these topics. A “topic” is a category or label that serves as a conduit for similar messages. It helps categorize the vast array of messages that can be received. We can have multiple subscribers subscribing to the same topic. From there, our application logic receives data from these subscribers.

In our payment example, an “OrderPlaced” topic could have multiple subscribers such as the inventory service (updating the stock level) and a billing. Service (which charges the customer). All these subscribers need to be notified when an order is placed.

Another benefit of the pub/sub model is that we can introduce a completely different API as a subscriber without needing to alter our pub/sub architecture. Thus, new subscribers can be added to a topic without modifying the publishers. This makes the system more flexible and adaptable to changing requirements.



The visual above demonstrates the pub/sub architecture as it pertains to message queues.

Popular message queues are:

1. Rabbit MQ
2. Kafka
3. GCP Pub/Sub