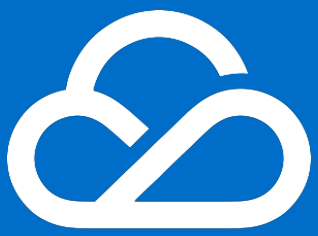
Cloud Message Queue

Best Practices

Product Introduction



Tencent Cloud

Copyright Notice

Trademark Notice

Tencent Cloud

All trademarks associated with Tencent Cloud and its services are owned by Tencent Cloud Computing (Beijing) Company Limited and its affiliated companies. Trademarks of third parties referred to in this document are owned by their respective proprietors.

Service Statement

This document is intended to provide users with general information about Tencent Cloud's products and services only and does not form part of Tencent Cloud's terms and conditions. Tencent Cloud's products or services are subject to change. Specific products and services and the standards applicable to them are exclusively provided for in Tencent Cloud's applicable terms and conditions.

Contents

Best Practices

Push or Pull

Tencent Cloud CMQ supports both Pull and Push. What scenarios are the two methods applicable to? We will briefly analyze the advantages and disadvantages of Push and Pull models in different scenarios.

Difference Between Push and Pull

In the Push model, when the server receives a message sent by the Producer, it will immediately deliver the message to the Consumer; while in the Pull model, after the server receives a message, it will do nothing but wait for the Consumer to read the message, that is, the Consumer will perform "pulling".

Scenario 1: The speed rate of Producer is higher than that of the Consumer

In the case that the speed rate of Producer is higher than that of the Consumer, there are two possibilities. The first is that, the efficiency of the Producer itself is higher than that of the Consumer (for example, the business logic of message processing on the Consumer side may be very complicated, or it may involve disk, network or other I/O operations). The second is that, message consuming is impossible or is hindered in a short time due to failure of Consumer.

In the Push model, the server cannot know the state of the current message consumer, so it will continuously push the data generated. The Consumer may receive a heavier load and even crash under the circumstances above (for example, the Producer is flume, collecting massive logs, and the Consumer is HDFS+hadoop which is lagging behind the Producer in processing efficiency), unless the Consumer has an appropriate feedback mechanism to inform the server of its state.

While it will be easier if Pull is used. Since the Consumer pulls data from the server by itself, the load can be reduced by decreasing its access frequency. For example, if a log collection service such as flume is at the frontend, which continuously produces and sends messages to CMQ that delivers these messages to the backend, the data analysis service or other services at the backend may have a lower efficiency than the Producer.

Scenario 2: Emphasize the real-timeness of messages

In the Push model, the server can immediately push a message to the Consumer once it arrives, which provides a desirable real-timeness obviously. In the Pull model, in order not to put pressure on the server (especially when the data volume is insufficient, continuous polling makes no sense), it is important to control its own polling interval, which will definitely have impact on real-timeness to a certain extent.

Scenario 3: Long polling of Pull

What is the problem with the Pull model? As the Consumer has the initiative, the Consumer cannot accurately decide when to pull the latest messages. If any messages are pulled, the Consumer can continue to pull; if no message is pulled, the Consumer needs to wait for a while to pull again.

But it is difficult to know how long the wait time will be. You might say that you can have xx dynamic adjustment algorithm for pull time. But the point is that the Consumer is not able to decide when a message should arrive. Perhaps 1,000 messages arrive consecutively within 1 minute, and then no new message is generated within half an hour. The most probable time point of the next message's arrival calculated with your algorithm is 31 minutes later or 60 minutes later, however, the next message arrives in 10 minutes in fact. Isn't it quite frustrating?

Of course, it doesn't mean that there is no solution to latency. A proven approach in the industry is to start with a short time interval (that will not place too much burden on CMQ broker), and then increase waiting time exponentially. For example, wait 5 ms at the beginning, and then wait 10 ms, 20 ms, 40 ms...until the arrival of a message, and then get back to 5 ms. Even so, the latency problem still exists: assuming that a message arrives right after 50 ms (between 40 ms and 80 ms), the message is delayed 30 ms, and for messages arriving every half an hour, such cost is just a waste.

Tencent Cloud CMQ provides an optimized approach of long polling to balance the respective drawbacks of pull/push model. The basic method is: if the Consumer fails to pull, it will not directly return, but hold the connection there to wait; when the server receives any new messages, it will pull up the connection and return the latest messages.

Scenario 4: Part or all of Consumers are not online

In the message system, the Producer and the Consumer are completely decoupled. When the Producer sends messages, the Consumer is not required to be online, and vice versa, which is the main difference between messaging and RPC communication. There are many circumstances for Consumer not online.

In case of accidental crash or offline of the Consumer, the Producer will not be affected and can continue to consume the last message when the Consumer goes online, and the message data is not lost. But if the Consumer is down for a long time or cannot be rebooted due to failures, there will be issues to be addressed, such as, whether the server needs to keep data for the Consumer, and how long the data is kept.

In the Push model, since it is impossible to know whether the Consumer is down or offline transiently or persistently, if all the historical messages since the crash are kept for the Consumer, the data cannot be cleaned up even if all the other Consumers have finished consuming, and the queue will be longer and longer over time. At that time, no matter whether the messages are temporarily stored in memory or are persistent on disks (for systems using Push model, the message queue is generally maintained in memory in order to ensure the performance and real-timeness of push, which will be discussed in detail later), they will put huge pressure on the CMQ server, and even affect the normal consumption of other Consumers, especially when the message producing speed is very fast. However, if the data is not kept, the data may be lost when the Consumer is rebooted.

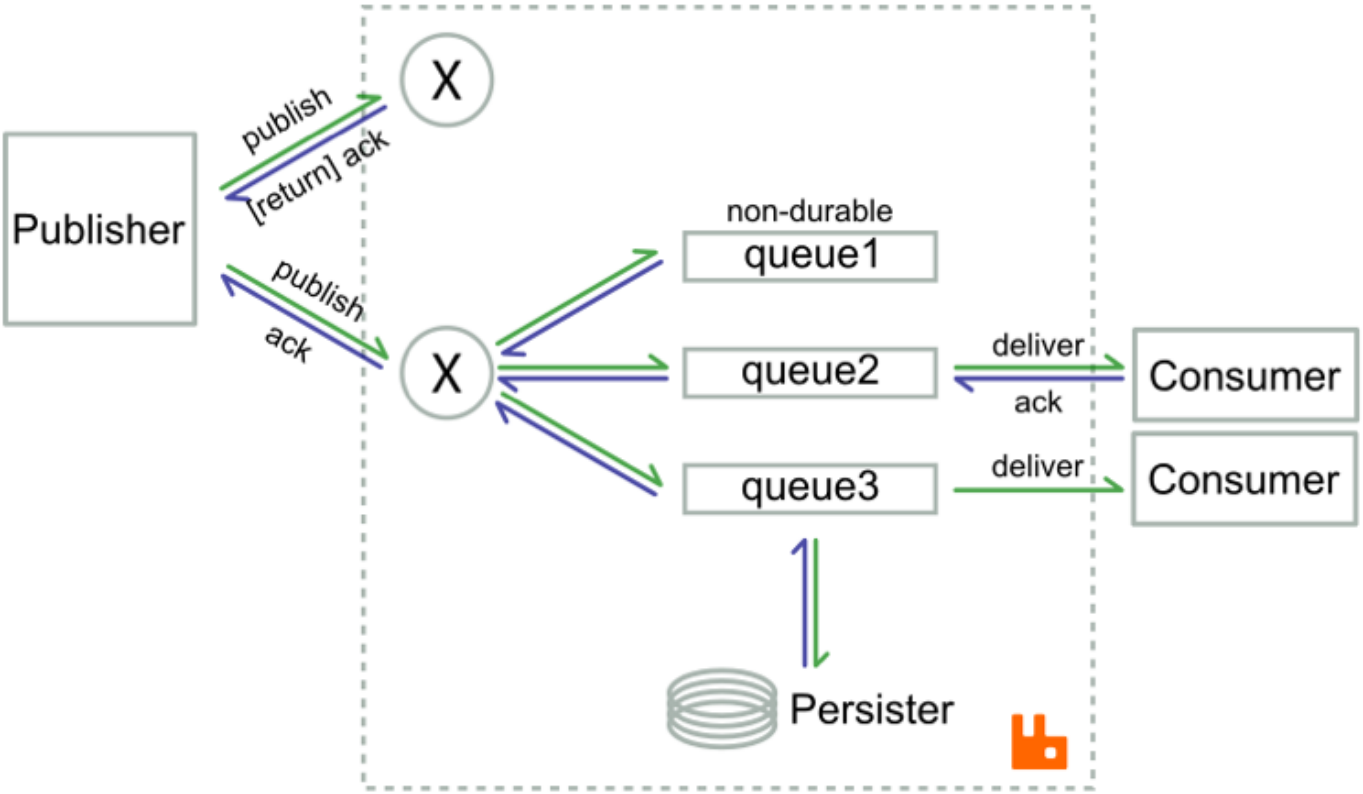
A compromise solution is that CMQ sets a timeout period for data, and it will clean up the data when the Consumer's downtime exceeds this threshold, but the threshold is not easy to determine.

In the Pull model, it gets better. The server no longer cares about the state of Consumer, but only provides services when it comes; the server will not guarantee whether the Consumer can consume the data in a timely manner (there is also a timeout period for clean-up).

Removing Duplicate Messages

The best solution to duplicate messages is to change the duplicate messages to a different ones directly.(repeated message consumption has no impact on the service). If it is not allowed to change duplicate messages, you should remove them on the consumer side.

Ⅰ. Causes for Duplicate Messages



Messages may be lost due to network exception, server crash, etc. To avoid losing message and ensure reliable delivery, CMQ applies the message production and consumption acknowledgement mechanisms.

Message production acknowledgement: the producer sends a message to CMQ for acknowledgement; CMQ persists the message to disk, and then returns the acknowledgement to the producer. Otherwise, the producer needs to resend the message to CMQ in cases such as the producer request timeout, CMQ return failure.

Consumer Acknowledgement: CMQ delivers the message to the consumer and set it to invisible; during the invisibility period, the consumer uses the handle to delete the message. If the message is

not deleted within the invisibility period, it will become visible again.

Since the message acknowledgement mechanism guarantees "at-least-once delivery", the producer/consumer may produce/consume repeatedly in cases such as network jitter, producer/consumer exceptions.

Ⅱ. How to Remove Duplicates

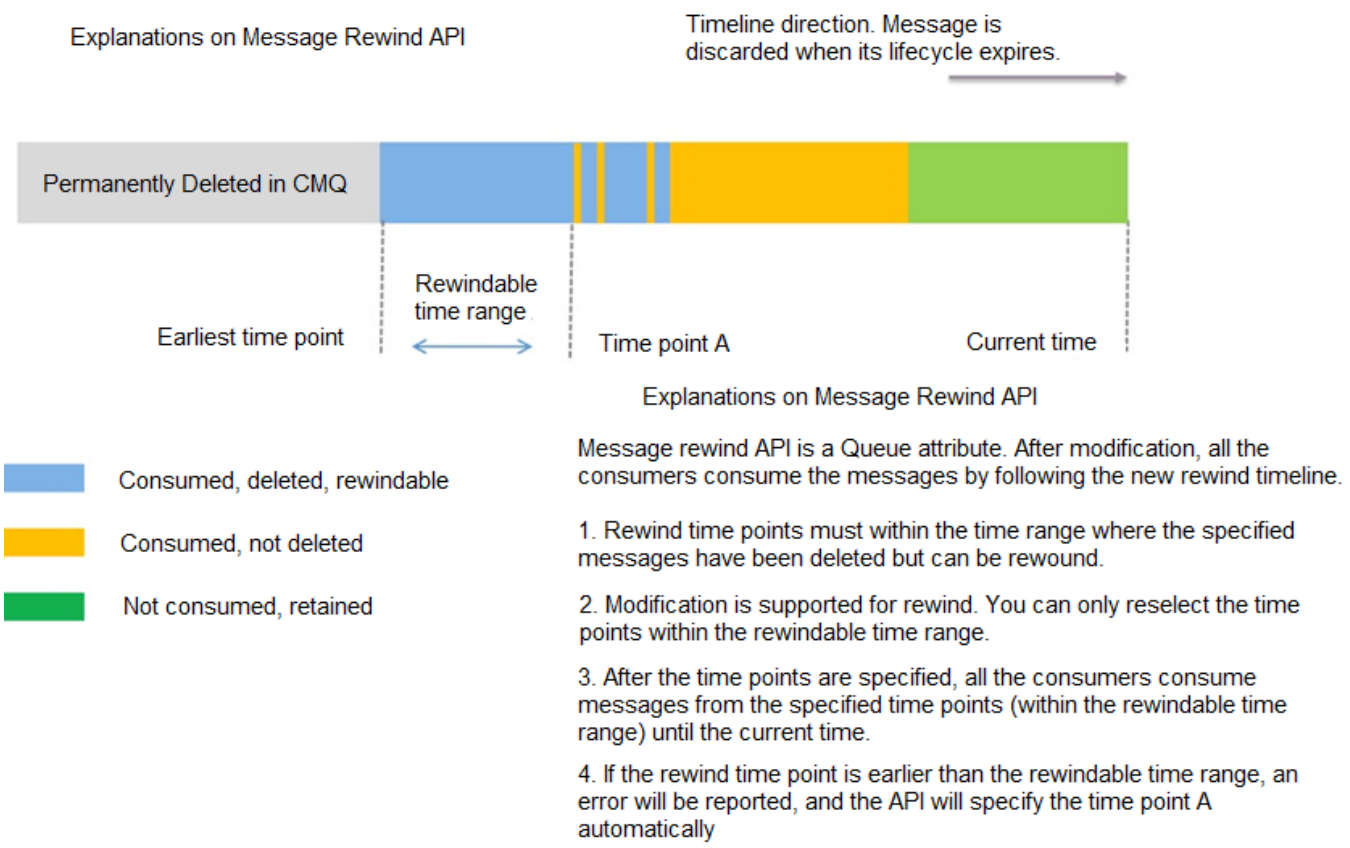
You should identify duplicate messages before you remove the duplicates. A common method is to insert a Remove Duplicates key in the message body during production for consumers to identify the duplicates via the Remove Duplicates key. The Remove Duplicates key is a unique value composed of .

If there is only one consumer, you can store the consumed Remove Duplicates key in cache (such as KV), and check if the Remove Duplicates key is consumed for each consumption. The Remove Duplicates key cache expires based on the maximum validity period of message. You can use the minimum unconsumed message time (min_msg_time) of the queue provided by CMQ, as well as the maximum retry time for producing message provided by service, to identify when the cache expires. If there are multiple consumers, a distributed Remove Duplicates key cache should be used.

• Calculate the expiration time of the key based on the maximum validity period of message:
current_time - max_retention_time - max_retry_time - max_network_time

• Calculate the expiration time of the key based on the minimum unconsumed message time of CMQ:
min_msg_time - max_retry_time - max_network_time

Notes:



CMQ provides a maximum message validity period of 15 days to meet the demands of different services.

All messages before the minimum unconsumed message time of CMQ Queue (the furthest point of time in the figure above) are deleted, and the messages after that may not.

Ⅲ. Examples

To avoid duplicate submission:

Scenario: A is a producer, B is a consumer, and CMQ is a broker. A transferred 10 CNY to B and sent the message to CMQ. CMQ successfully received the message, but failed to response to client A due to a flash of interruption or client A crash. A thought the request failed and re-produced the message, resulting in duplicate submission.

Solution: A adds information like timestamp when producing messages, to generate a unique Remove Duplicates key. If, due to network failures, current delivery failed, producer A will re-send the

message, and consumer B will use the previous Remove Duplicates key to remove the duplicates. (The case also illustrates that the message ID of CMQ cannot be used for removing duplicates, because the two messages have different IDs but the same body.)

Note that before sending messages, producer A should persist the Remove Duplicates key to disk, to avoid loss due to power failure.

To avoid messages with the same body from being filtered out:

Scenario: A transferred 10 CNY to B and sent 5 requests with the same body. If consumer B removes the duplicates based on body, B will deal with 1 request instead of 5 requests.

Solution: A adds information like timestamp when producing messages. In such case, even if A repeatedly sends the same message, different Remove Duplicates keys will be generated, which allows removing duplicate messages with the same body.