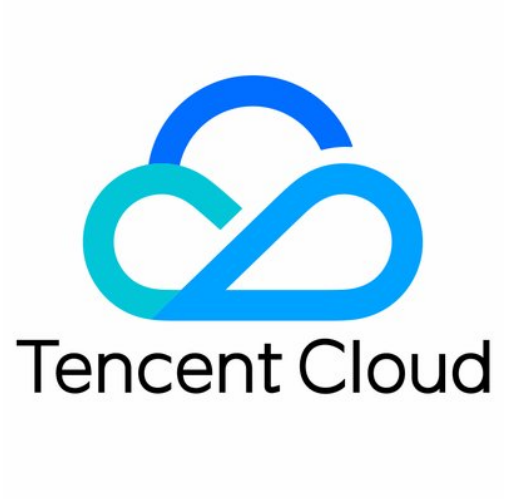
TDMQ for RocketMQ

Development Guide

Product Documentation



Contents

Development Guide
- Message Types
  - General Message
  - Scheduled Message and Delayed Message
  - Sequential Message
  - Transactional Message
- Message Filtering
- Consumption Mode
- Message Retry
- Dead Letter Queue

Development Guide
Message Types
General Message

Last updated：2022-02-11 14:33:40

General message is a basic message type, where a message is delivered to the specified topic by the producer and then consumed by the consumer subscribed to the topic. As general messages are not sequential in a topic, you can use multiple partitions to improve the message production/consumption efficiency, and they deliver the best performance when the throughput is huge.

General message is different from scheduled, delayed, sequential, and transactional message. The topics corresponding to these types of messages cannot be mixed and can only be used to send and receive messages of the same type. For example, a general message topic can only be used to send and receive general messages but not other types of messages.

Scheduled Message and Delayed Message

Last updated：2023-09-12 16:09:41

This document describes the concepts and usage of scheduled message and delayed message in TDMQ for RocketMQ.

Relevant Concepts

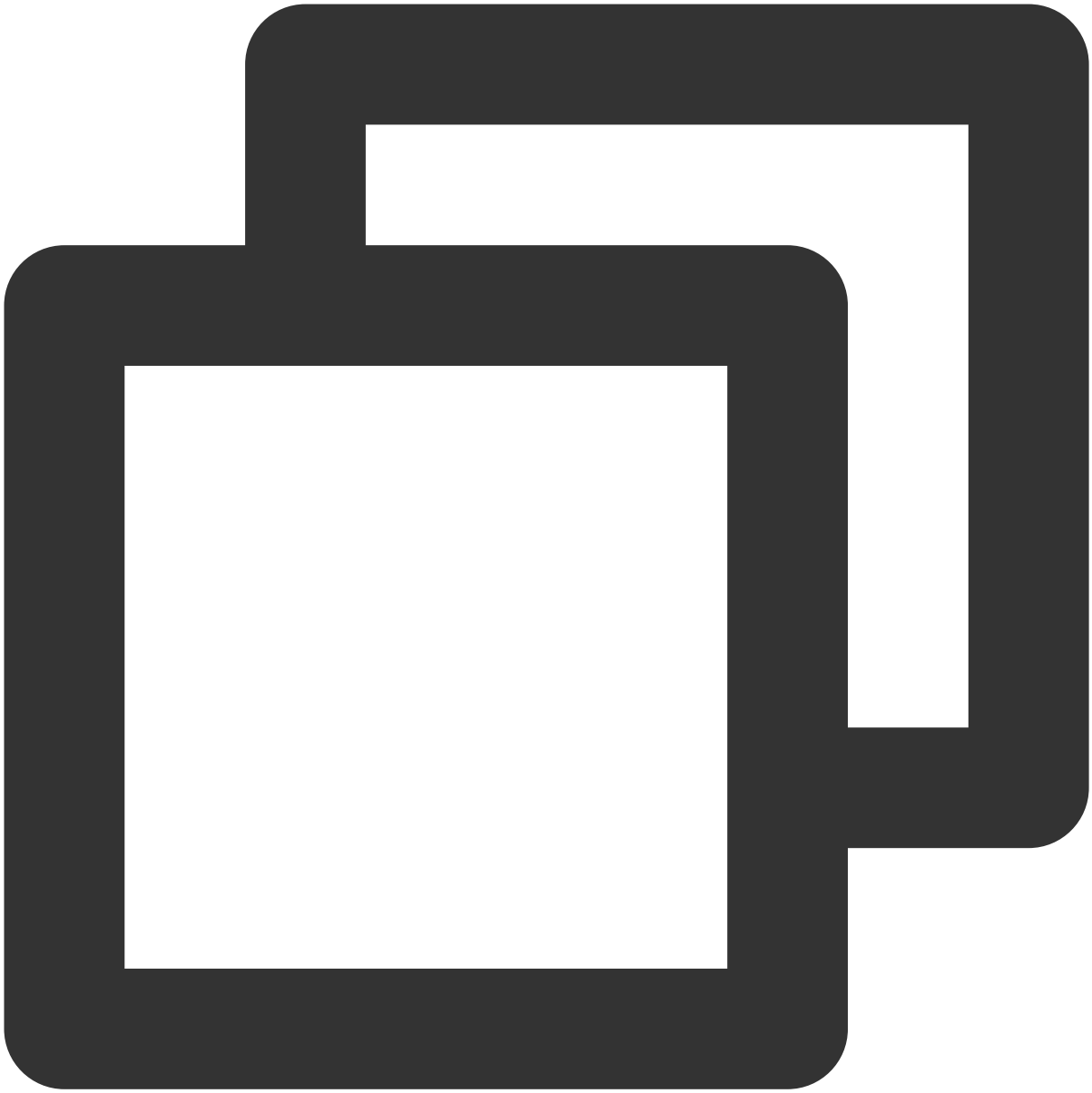
Scheduled message: After a message is sent to the server, the business may want the consumer to receive it at a later time point rather than immediately. This type of message is called "scheduled message".
Delayed message: After a message is sent to the server, the business may want the consumer to receive it after a period of time rather than immediately. This type of message is called "delayed message".
Actually, delayed message can be regarded as a special type of scheduled message, which is essentially the same thing.

Directions

Apache RocketMQ does not provide an API for you to freely set the delay time. In order to ensure compatibility with the open-source RocketMQ client, TDMQ for RocketMQ has designed a method to specify the message sending time by adding the property key-value pair to the message. You only need to add the `__STARTDELIVERTIME` property value to the `property` of the message that needs to be sent at a scheduled time (within 40 days). For delayed messages, you can first calculate the time point for scheduled sending and then send them as scheduled messages.
A code sample is given below to show how to use scheduled and delayed messages in TDMQ for RocketMQ. You can also view the complete sample code >>

Scheduled message

To send a scheduled message, simply write a standard millisecond timestamp to the `__STARTDELIVERTIME` property before sending it.

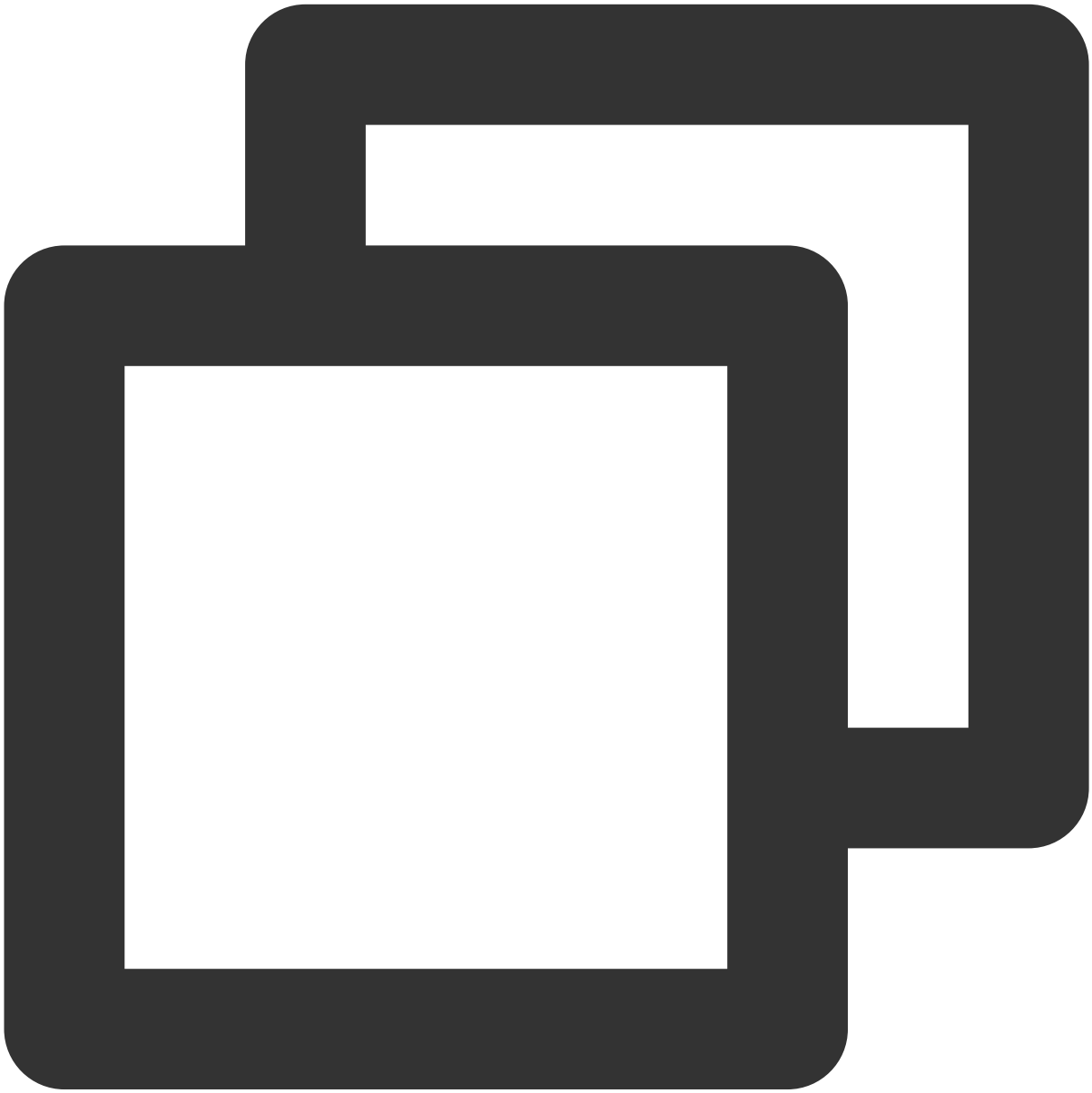


```
Message msg = new Message("test-topic", ("message content").getBytes(StandardCharse
// Set the message to be sent at 00:00:00 on 2021-10-01
try {
    long timeStamp = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").parse("10/1/2021 0
    // Set `__STARTDELIVERTIME` into the property of `msg`
    msg.putUserProperty("__STARTDELIVERTIME", String.valueOf(timeStamp));
    SendResult result = producer.send(msg);
    System.out.println("Send delay message: " + result);
} catch (ParseException e) {
    // TODO: Add the method for handling the timestamp parsing failure
    e.printStackTrace();
```

```
}
```

Delayed message

For a delayed message, its scheduled sending time point is first calculated by `System.currentTimeMillis() + delayTime` , and then it is sent as a scheduled message.



```
Message msg = new Message("test-topic", ("message content").getBytes(StandardCharse

// Set the message to be sent after 10 seconds
long delayTime = System.currentTimeMillis() + 10000;
```

```
// Set `__STARTDELIVERTIME` into the property of `msg`
msg.putUserProperty("__STARTDELIVERTIME", String.valueOf(delayTime));

SendResult result = producer.send(msg);
System.out.println("Send delay message: " + result);
```

Use Limits

When using delayed messages, make sure that the time on the client is in sync with the time on the server (UTC+8 Beijing time in all regions); otherwise, there will be a time difference.
There is a precision deviation of about 1 second for scheduled and delayed messages.
The maximum time range for scheduled and delayed messages are both 40 days.
When using scheduled messages, you need to set a time point after the current time; otherwise, the message will be sent to the consumer immediately.

Sequential Message

Last updated：2023-09-12 16:32:32

Sequential message is an advanced message type provided by TDMQ for RocketMQ. For a specified topic, messages are published and consumed in strict accordance with the principle of First-In-First-Out (FIFO), that is, messages sent first are consumed first, and messages sent later are consumed later.

Sequential messages are suitable for scenarios that have strict requirements on the sequence of message sending and consumption.

Use Cases

The comparison between sequential message and general message is as follows:

Message Type	Consumption Sequence	Performance	Applicable Scenarios
General message	No sequence	High	Huge-throughput scenarios with no requirements for production and consumption sequence
Sequential message	All messages in the specific topic follow the FIFO rule	Average	Average-throughput scenarios that require publishing and consuming all messages in strict accordance with the FIFO rule

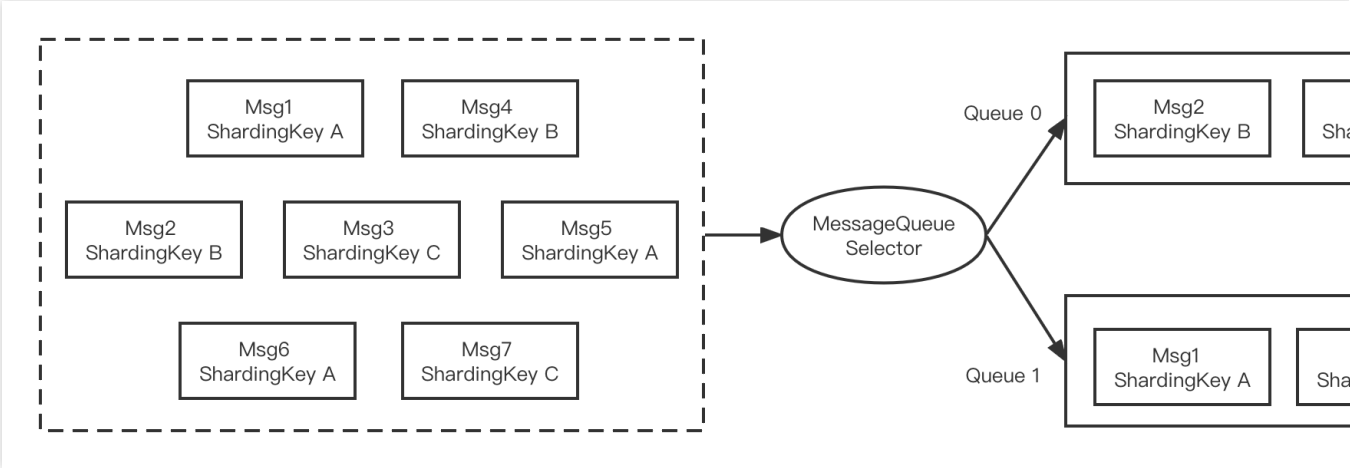
Sequential messages are often used in the following business scenarios:

Order creation: In some ecommerce systems, an order's creation, payment, refund, and logistics messages must be produced or consumed in strict sequence, No the order status will be messed up during consumption, which will affect the normal operation of the business. Therefore, the messages of this order must be produced and consumed in a certain sequence in the client and message queue. At the same time, the messages are sequentially dependent, and the processing of the next message must be dependent on the processing result of the preceding message.
Log sync: In the scenario of sequential event processing or real-time incremental data sync, sequential messages can also play a greater role. For example, it is necessary to ensure that database operations are in sequence when MySQL binlogs are synced.

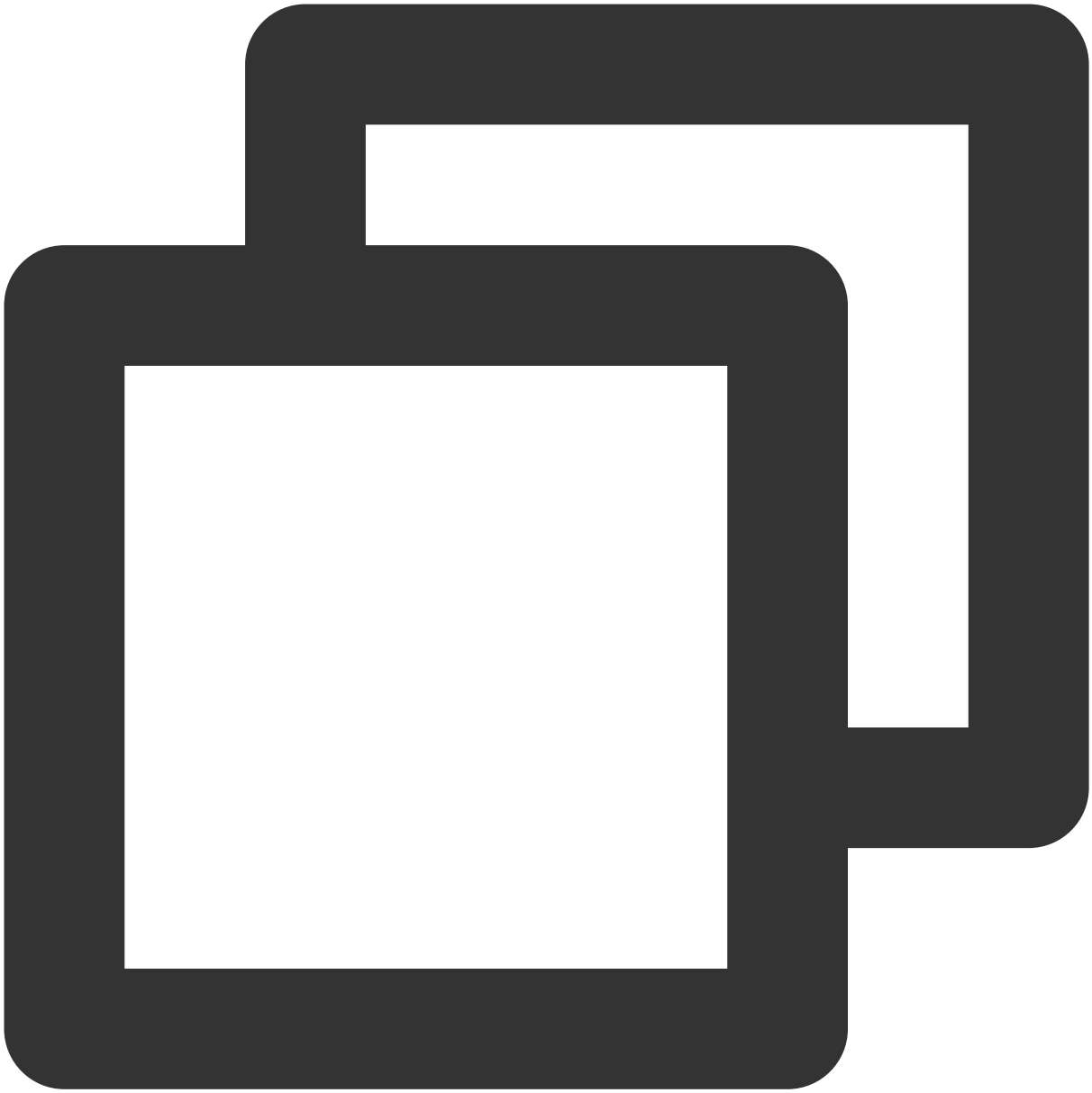
Financial scenarios: In some matchmaking transaction scenarios like certain securities transactions, the first bidder is given priority in the case of the same biding price, so it is necessary to produce and consume sequential messages in a FIFO manner.

How It Works

In TDMQ for RocketMQ, the principle of sequential messages is shown in the figure below. You can partition messages according to a certain standard (such as ShardingKey in the figure), and messages of the same ShardingKey will be assigned to the same queue and consumed in sequence.



The code of sequential message is as shown below:



```
public class Producer {
    public static void main(String[] args) throws UnsupportedEncodingException {
        try {
            DefaultMQProducer producer = new DefaultMQProducer("please_rename_uniqu
            producer.start();

            String[] tags = new String[] {"TagA", "TagB", "TagC", "TagD", "TagE"};
            for (int i = 0; i < 100; i++) {
                int orderId = i % 10;
                Message msg =
                    new Message("TopicTest", tags[i % tags.length], "KEY" + i,
```

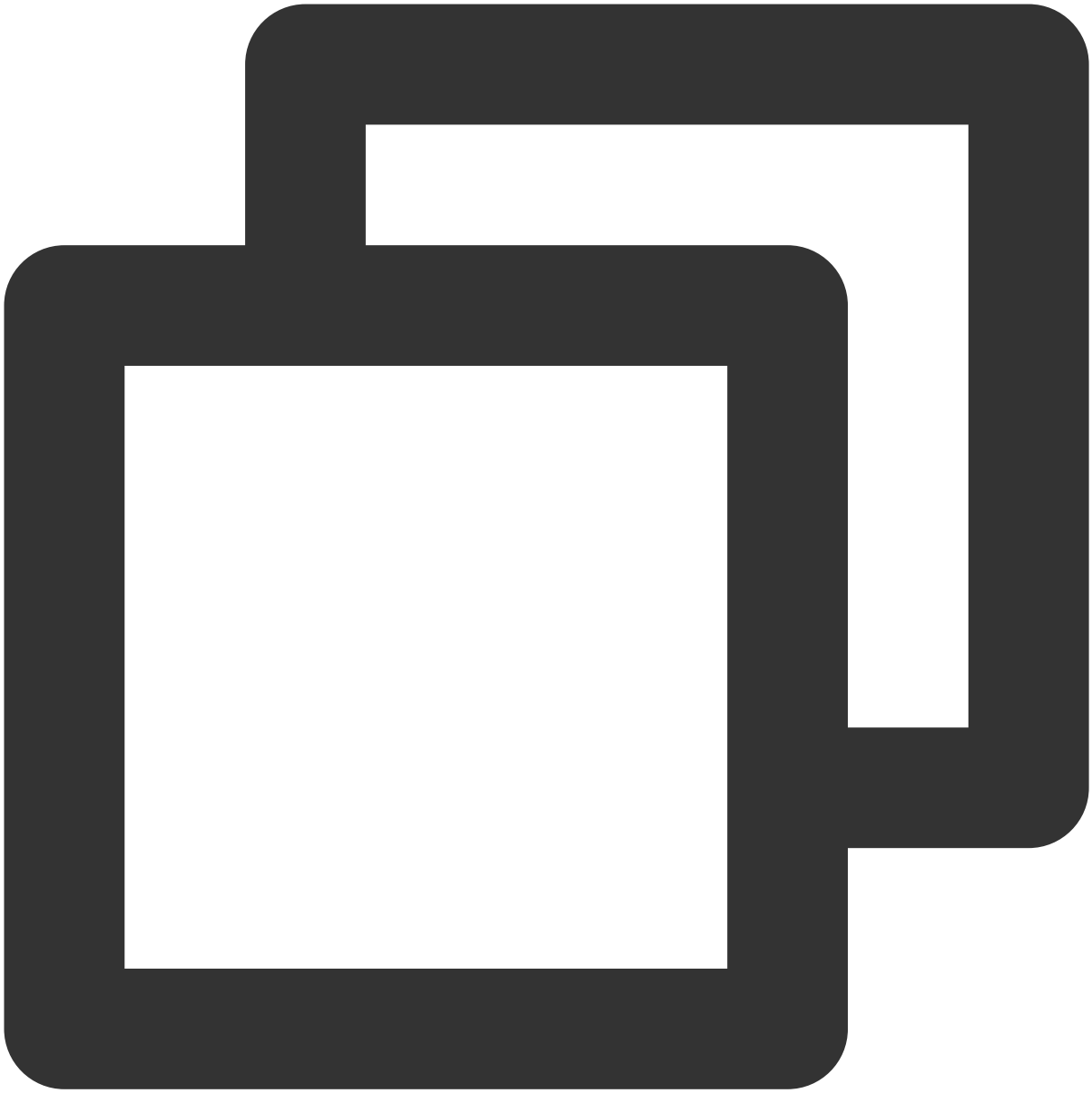
```
                        ("Hello RocketMQ " + i).getBytes(RemotingHelper.DEFAULT_CHA
                SendResult sendResult = producer.send(msg, new MessageQueueSelector
                    @Override
                    public MessageQueue select(List<MessageQueue> mqs, Message msg,
                        Integer id = (Integer) arg;
                        int index = id % mqs.size();
                        return mqs.get(index);
                    }
                }, orderId);

                System.out.printf("%s%n", sendResult);
            }

            producer.shutdown();
        } catch (MQClientException | RemotingException | MQBrokerException | Interr
            e.printStackTrace();
        }
    }
}
```

The main difference here is that the `SendResult send(Message msg, MessageQueueSelector selector, Object arg)` method is called, `MessageQueueSelector` is the queue selector, and `arg` is a Java object, which can be passed in as the classification standard of the message sending partition.

The `MessageQueueSelector` API is as follows:



```
public interface MessageQueueSelector {
    MessageQueue select(final List<MessageQueue> mqs, final Message msg, final Obje
}
```

Among them, `mqs` is the queue that can be sent, `msg` is the message, `arg` is the object passed in the above `send` API, and the queue to which the message needs to be sent is returned. In the above sample, `orderId` is used as the partition classification standard, and the remainder of all queue numbers is used to send messages with the same `orderId` to the same queue.

In the production environment, we recommend that you select the most fine-grained partition key for splitting. For example, when the order ID and user ID are used as the partition key keywords, the messages of the same end user will be processed in sequence, while those of different users will not.

Note

In order to ensure the high availability of messages, TDMQ for RocketMQ currently doesn't support "globally sequential messages" in a single queue (if you have already created globally sequential messages, you can use them normally); if you want to ensure global sequence, you can use consistent ShardingKey to do so.

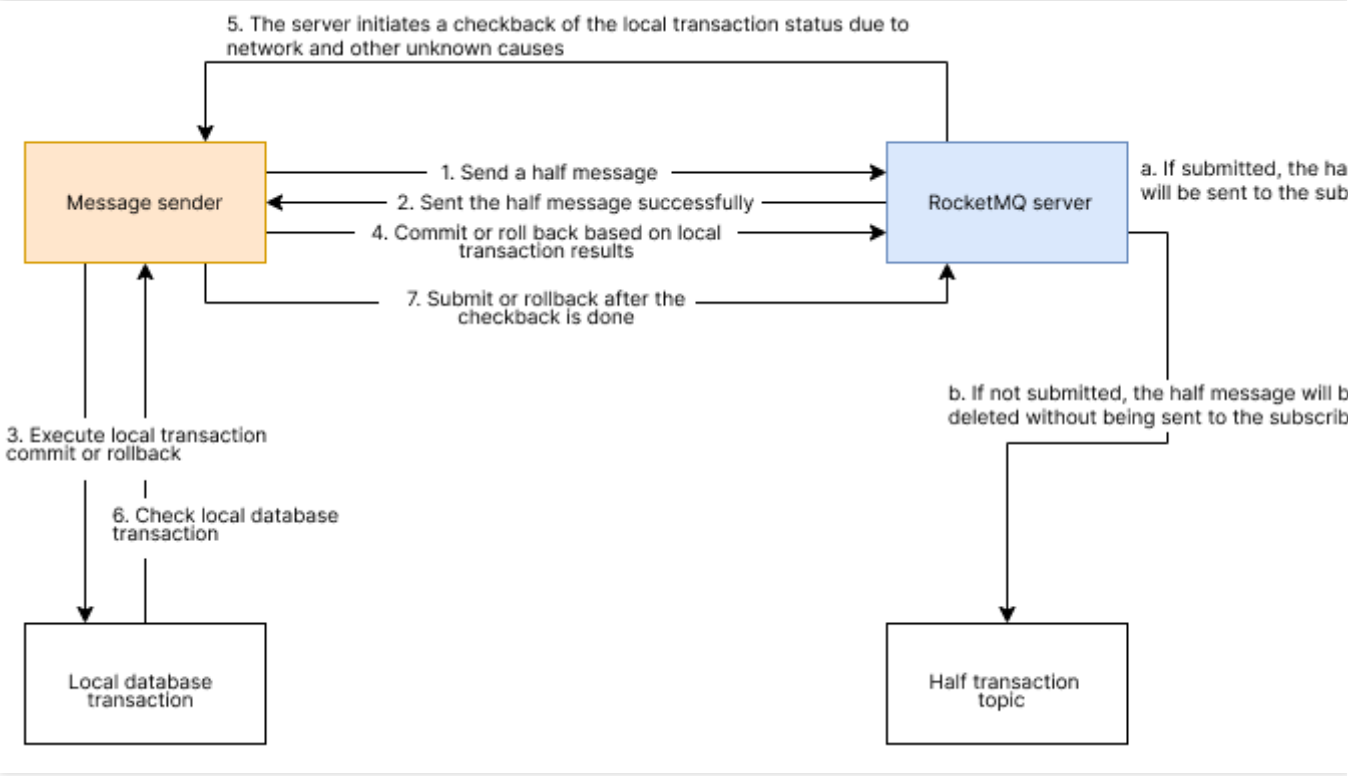
Transactional Message

Last updated：2023-04-14 16:59:12

This document describes the concept, technical principle, use cases, and usage methods of transactional messages in the TDMQ for RocketMQ.

Description

The transactional message solves the atomicity problem of local transaction execution and message sending, ensuring the eventual consistency between them. It provides users with the distributed transaction feature similar to X/Open XA, so users can achieve the eventual consistency of the distributed transaction in TDMQ for RocketMQ.



1. The producer sends a message to RocketMQ (1).
2. After receiving the message, the server stores the message in the half message topic (2).
3. Local transaction execution is done (3).
4. The producer actively sends the transaction execution result to RocketMQ (4).
5. If the local transaction execution result has not been returned after a certain period of time, RocketMQ will execute the checkback logic (5).

6. After receiving the message checkback, the producer needs to check the final result of the local transaction execution of the corresponding message and give feedback (6, 7). There are three transaction execution status:
TransactionStatus.COMMIT: Commits the transaction, and consumers can consume the message.
TransactionStatus.ROLLBACK: Rolls back the transaction, and the message is discarded without being consumed by consumers.
TransactionStatus.UN_KNOW: Unknown status, indicating the waiting of another checkback.
7. When the transaction is successfully executed, RocketMQ submits the transactional message to the real topic for consumption by consumers (a).

Use Cases

The transaction messages of TDMQ for RocketMQ can be used to process transactions, which can greatly improve processing efficiency and performance. A billing system often has a long transaction linkage with a significant chance of error or timeout. TDMQ's automated repush and abundant message retention features can be used to provide transaction compensation, and the eventual consistency of payment tips notifications and transaction pushes can also be achieved through TDMQ.

Message Filtering

Last updated：2022-02-11 14:34:21

This document describes the feature, use cases, and usage of message filtering in TDMQ for RocketMQ.

Feature Overview

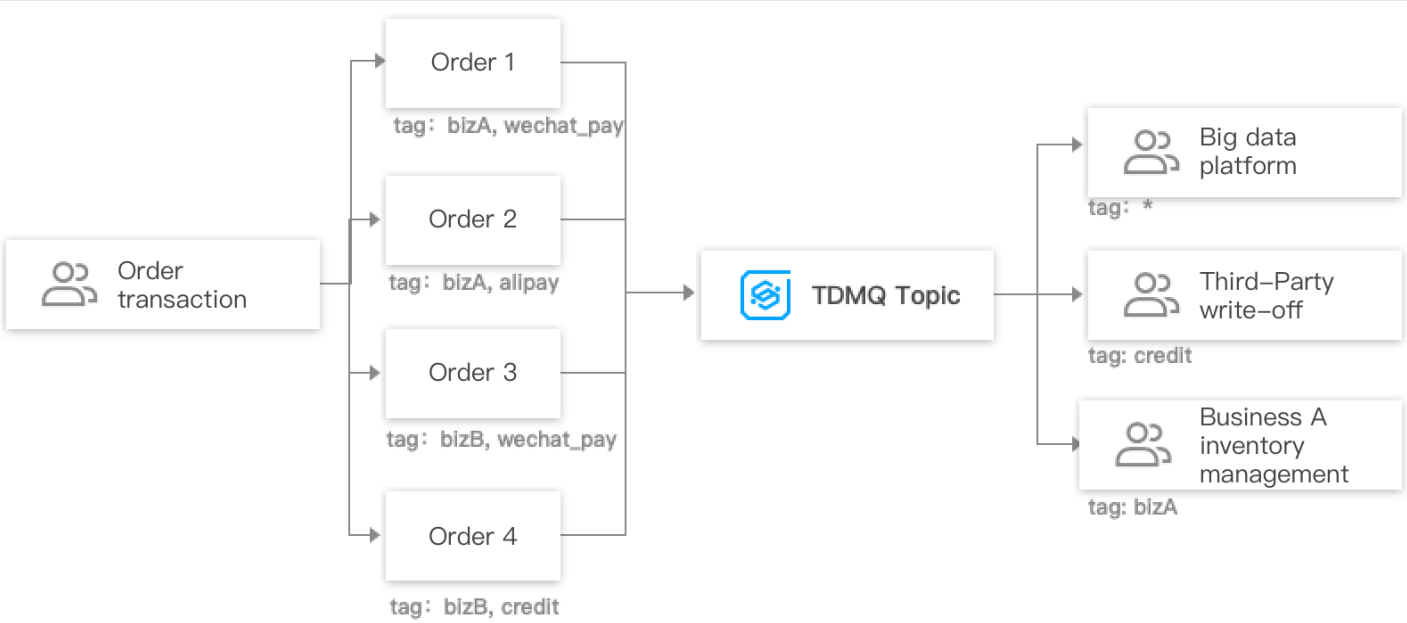
In message filtering, a message producer configures message attributes to group messages before sending them to a topic, and a consumer subscribed to the topic uses message attributes to filter the messages so that only eligible messages are delivered to the consumer for consumption.

If a consumer sets no filter conditions when subscribing to a topic, no matter whether filter attributes are set during message sending, all messages in the topic will be delivered to the consumer for consumption.

Use Cases

Generally, messages with the same business attributes are stored in the same topic; for example, when an order transaction topic contains messages of order placement transactions, payment transactions, and delivery transactions, and if you want to consume only one type of transaction messages in your business, you can filter them on the client, but this will waste bandwidth resources.

To solve this problem, TDMQ supports filtering on the broker. You can set one or more tags during message production and subscribe to specified tags during consumption.



Usage

Sending message

You must specify tags for each message when sending it.

```
Message msg = new Message("TOPIC","TagA","Hello world".getBytes());
```

Subscribing to message

- Subscribe to all tags:

  If a consumer wants to subscribe to all types of messages under a topic, an asterisk (*) can be used to represent all tags.

  ```
  consumer.subscribe("TOPIC", "*", new MessageListener() {
  public Action consume(Message message, ConsumeContext context) {
  System.out.println(message.getMsgID());
  return Action.CommitMessage;
  }
  });
  ```

- Subscribe to one tag:

  If a consumer wants to subscribe to a certain type of messages under a topic, the tag should be specified clearly.

```
consumer.subscribe("TOPIC", "TagA", new MessageListener() {
public Action consume(Message message, ConsumeContext context) {
System.out.println(message.getMsgID());
return Action.CommitMessage;
}
});
```

- Subscribe to multiple tags:

  If a consumer wants to subscribe to multiple types of messages under a topic, two vertical bars (||) should be added between two tags for separation.

```
consumer.subscribe("TOPIC", "TagA||TagB", new MessageListener() {
public Action consume(Message message, ConsumeContext context) {
System.out.println(message.getMsgID());
return Action.CommitMessage;
}
});
```

Consumption Mode

Last updated：2023-09-13 11:40:40

This document describes the features and use cases of clustering consumption and broadcasting consumption in TDMQ for RocketMQ.

Feature Overview

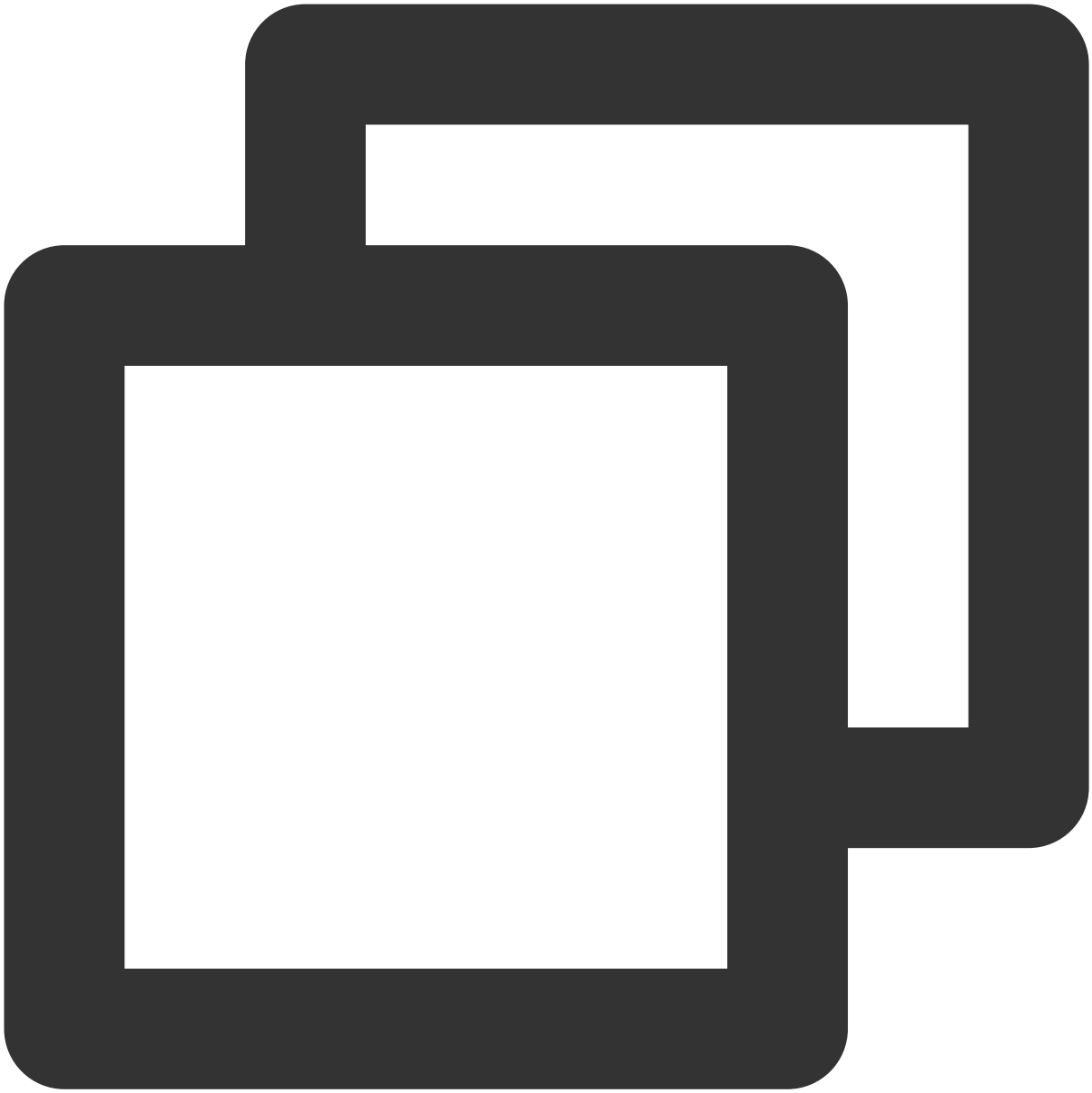
Cluster consumption: if the cluster consumption mode is used, any message only needs to be processed by any consumer in the cluster.
Broadcast consumption: if the broadcast consumption mode is used, each message will be pushed to all registered consumers in the cluster to ensure that the message is consumed by each consumer at least once.

Use Cases

Clustering consumption is suitable for scenarios where each message only needs to be processed once.
Broadcasting consumption is suitable for scenarios where each message needs to be processed by each consumer in the cluster.

Sample Codes

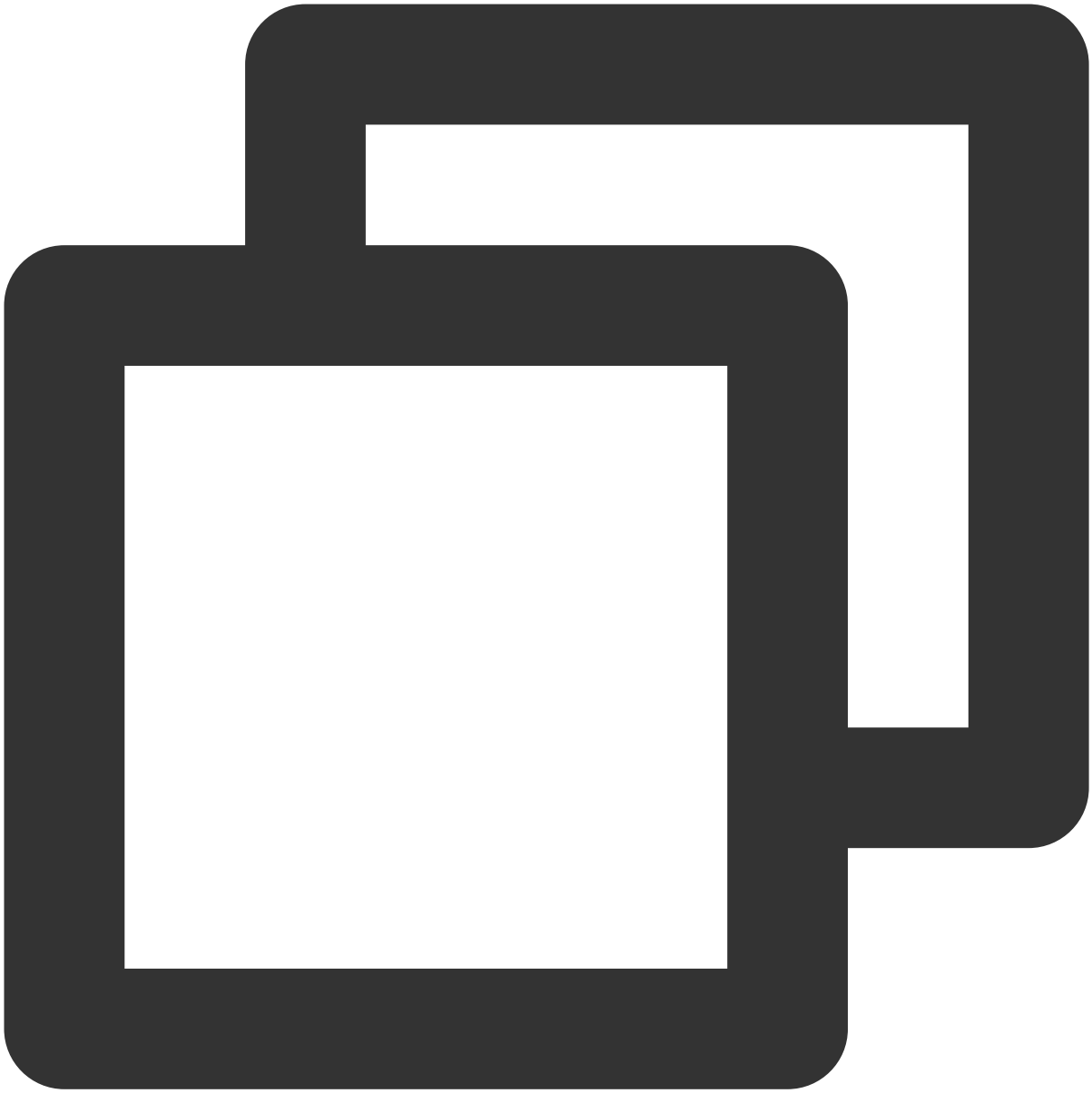
Cluster subscription
All consumers identified by the same group ID will evenly share messages for consumption. For example, if a topic has nine messages, and a group ID identifies three consumer instances, then each instance will consume only three messages evenly in the clustering consumption mode.



```
// Set the cluster subscription mode (which is the default mode if you don't specif
properties.put(PropertyKeyConst.MessageModel, PropertyValueConst.CLUSTERING);
```

Broadcast subscription

A message will be consumed once by all consumers identified by the same group ID. In the broadcasting consumption mode, for example, if a topic has nine messages and a group ID identifies three consumer instances, each instance will consume nine messages.



```
// Set the broadcast subscription mode
properties.put(PropertyKeyConst.MessageModel, PropertyValueConst.BROADCASTING);
```

Note

You need to ensure that all consumer instances under the same group ID have the same subscription relationships.

Message Retry

Last updated：2023-09-13 11:41:02

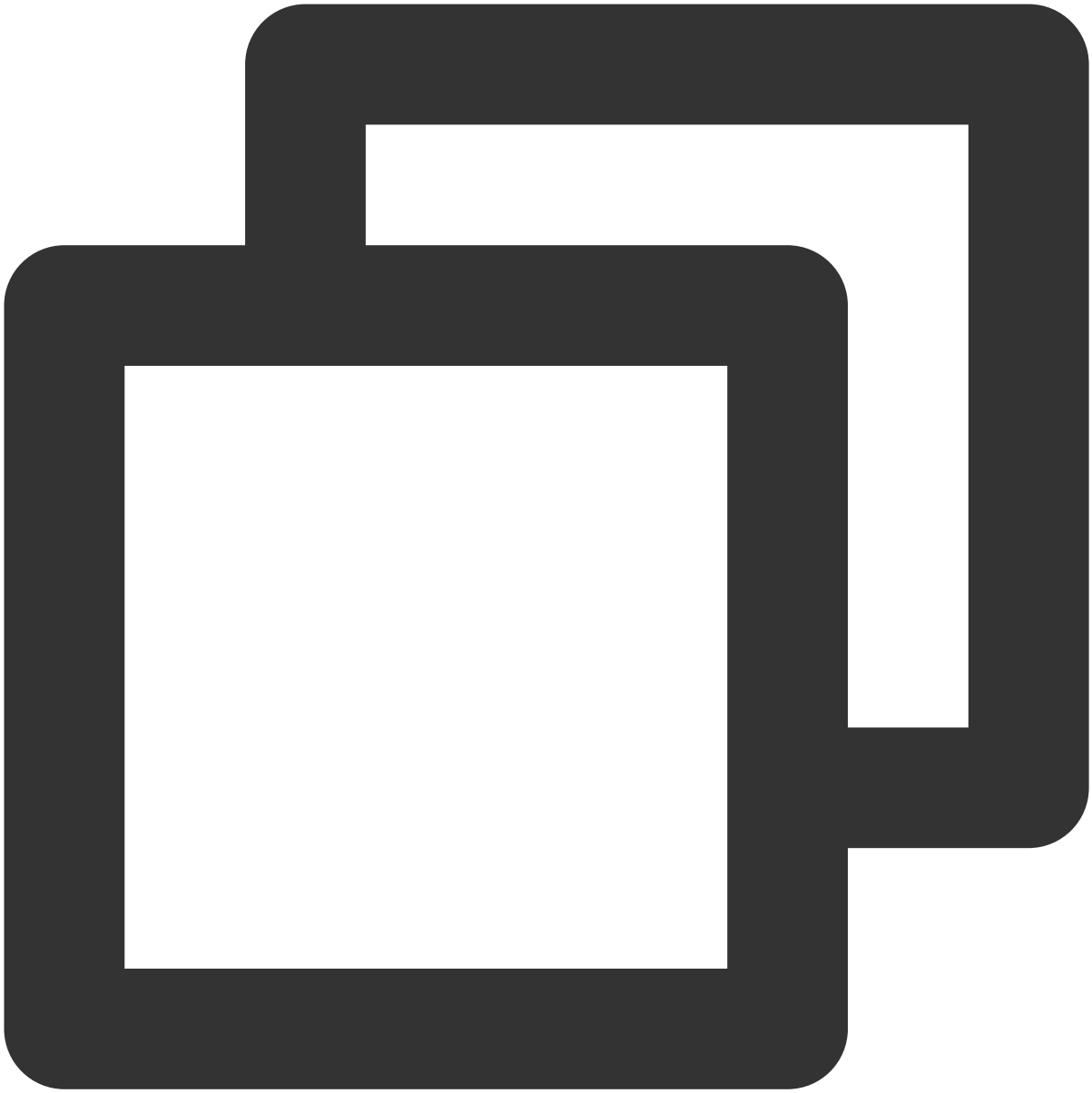
This document describes the message retry mechanisms and their usages in TDMQ for RocketMQ.

Feature Overview

When a message is consumed for the first time by a consumer and fails to get a normal response, or when it is requested by users to deliver again in the server, TDMQ for RocketMQ will automatically retry delivering this message through the message retry mechanism until it is consumed successfully. When the number of retries reaches the specified value but the message is still not consumed successfully, retry will stop, and the message will be delivered to the dead letter queue.
After the message enters the dead letter queue, TDMQ for RocketMQ can no longer process it automatically. At this point, human intervention is generally required. You can write a dedicated client to subscribe to the dead letter queue to process such messages.
Note
The broker will automatically retry in the cluster consumption mode but not the broadcast consumption mode.
The following results are considered as consumption failure, and the message will be retried accordingly:
1. The consumer returns `ConsumeConcurrentlyStatus.RECONSUME_LATER` .
2. The consumer returns null.
3. The consumer actively/passively throws an exception.

Number of Retries

When a message needs to be retried in TDMQ for RocketMQ, set the "messageDelayLevel" parameter as follows to configure the number of retries and retry intervals:



```
messageDelayLevel=1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h
```

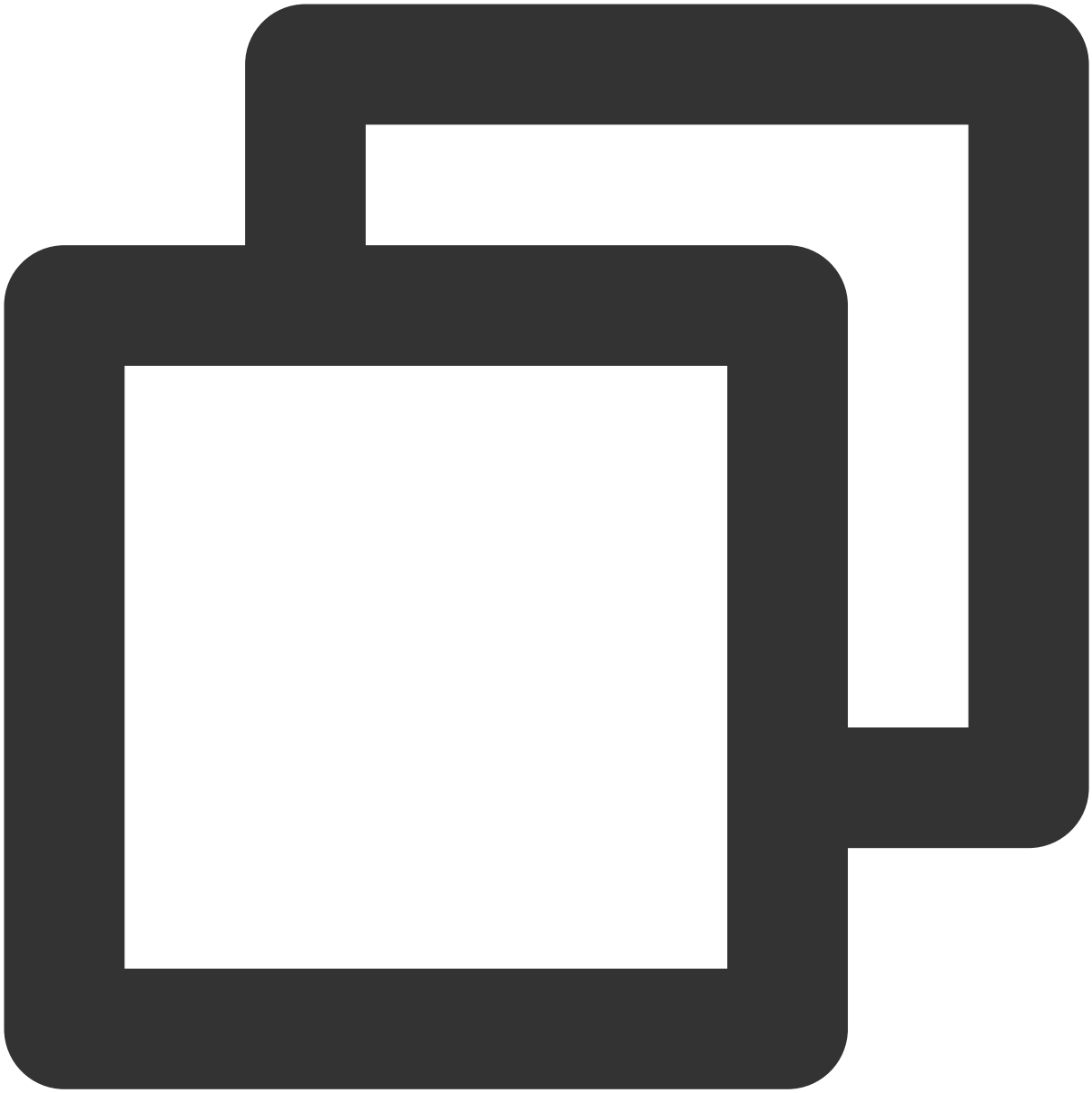
The number of retries and retry intervals have the following relationships:

Retry No.	Time Interval Since Last Retry	Retry No.	Time Interval Since Last Retry
1	1 second	10	6 minutes
2	5 seconds	11	7 minutes
3	10 seconds	12	8 minutes

4	30 seconds	13	9 minutes
5	1 minute	14	10 minutes
6	2 minutes	15	20 minutes
7	3 minutes	16	30 minutes
8	4 minutes	17	1 hour
9	5 minutes	18	2 hours

Instructions

If you need to adjust the number of retries by yourself, you can set the parameters of the consumer.



```
pushConsumer.setMaxReconsumeTimes(3);
```

Dead Letter Queue

Last updated：2023-09-12 16:42:15

This document describes the dead letter queues and their usages in TDMQ for RocketMQ.

Feature Overview

When a message is consumed for the first time by a consumer and fails to get a normal response, or when it is requested by users to deliver again in the server, TDMQ for RocketMQ will automatically retry delivering this message through the message retry mechanism until it is consumed successfully. When the number of retries reaches the specified value but the message is still not consumed successfully, retry will stop, and the message will be delivered to the dead letter queue.
After the message enters the dead letter queue, TDMQ for RocketMQ can no longer process it automatically. At this point, human intervention is generally required. You can write a dedicated client to subscribe to the dead letter queue to process such messages.

Notes

Messages in the dead letter queue must be processed manually or by new code logic, whereas messages in the retry queue can be consumed automatically.
Messages in the dead letter queue are only valid for three days by default and are deleted after that.
The dead letter queue starts with %DLQ%, which corresponds to the consumer group one by one. Therefore, a dead letter queue contains all the dead letter messages corresponding to the group ID, no matter which topic the message belongs to.