

TDMQ for Apache Pulsar

Developer Guide

Product Documentation



Copyright Notice

©2013-2024 Tencent Cloud. All rights reserved.

Copyright in this document is exclusively owned by Tencent Cloud. You must not reproduce, modify, copy or distribute in any way, in whole or in part, the contents of this document without Tencent Cloud's the prior written consent.

Trademark Notice



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

Developer Guide

How It Works

- Pulsar Topic and Partition

- Pulsar Multi-AZ Deployment

- Message Storage Principle and ID Generation Rule

- Message Replica and Storage Mechanisms

Best Practices

- Subscription Mode

- Scheduled Message and Delayed Message

- Message Tag Filtering

- Message Retry and Dead Letter Mechanisms

- Client Connection and Producer/Consumer

Developer Guide

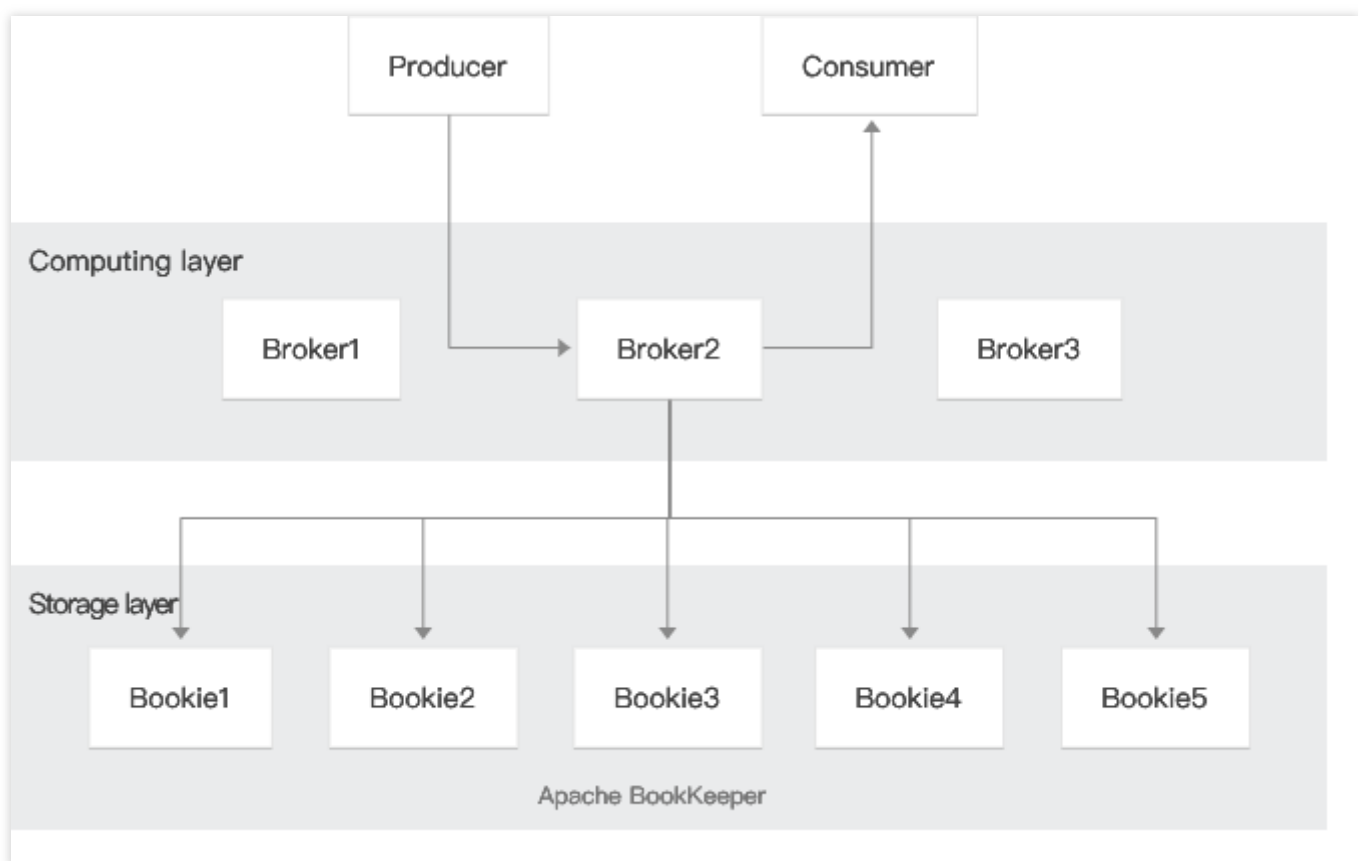
How It Works

Pulsar Topic and Partition

Last updated : 2024-06-28 11:29:49

Apache Pulsar Architecture

Apache Pulsar is a messaging system built on the publish-subscribe pattern, which consists of broker, Apache BookKeeper, producer, consumer, and other components.



Producer: message producer, which is responsible for publishing messages to topics.

Consumer: message consumer, which is responsible for subscribing to messages from topics.

Broker: stateless service layer, which is responsible for receiving and transferring messages, balancing cluster load, and performing other tasks. It does not store metadata persistently, so it can be connected and disconnected quickly.

Apache BookKeeper: stateful persistence layer, which consists of a set of Bookie storage nodes and can store messages persistently.

Apache Pulsar adopts the computing-storage separation architecture, where the computing logic related to message publishing and subscription is implemented in brokers, and data is stored on the bookie nodes in an Apache BookKeeper cluster.

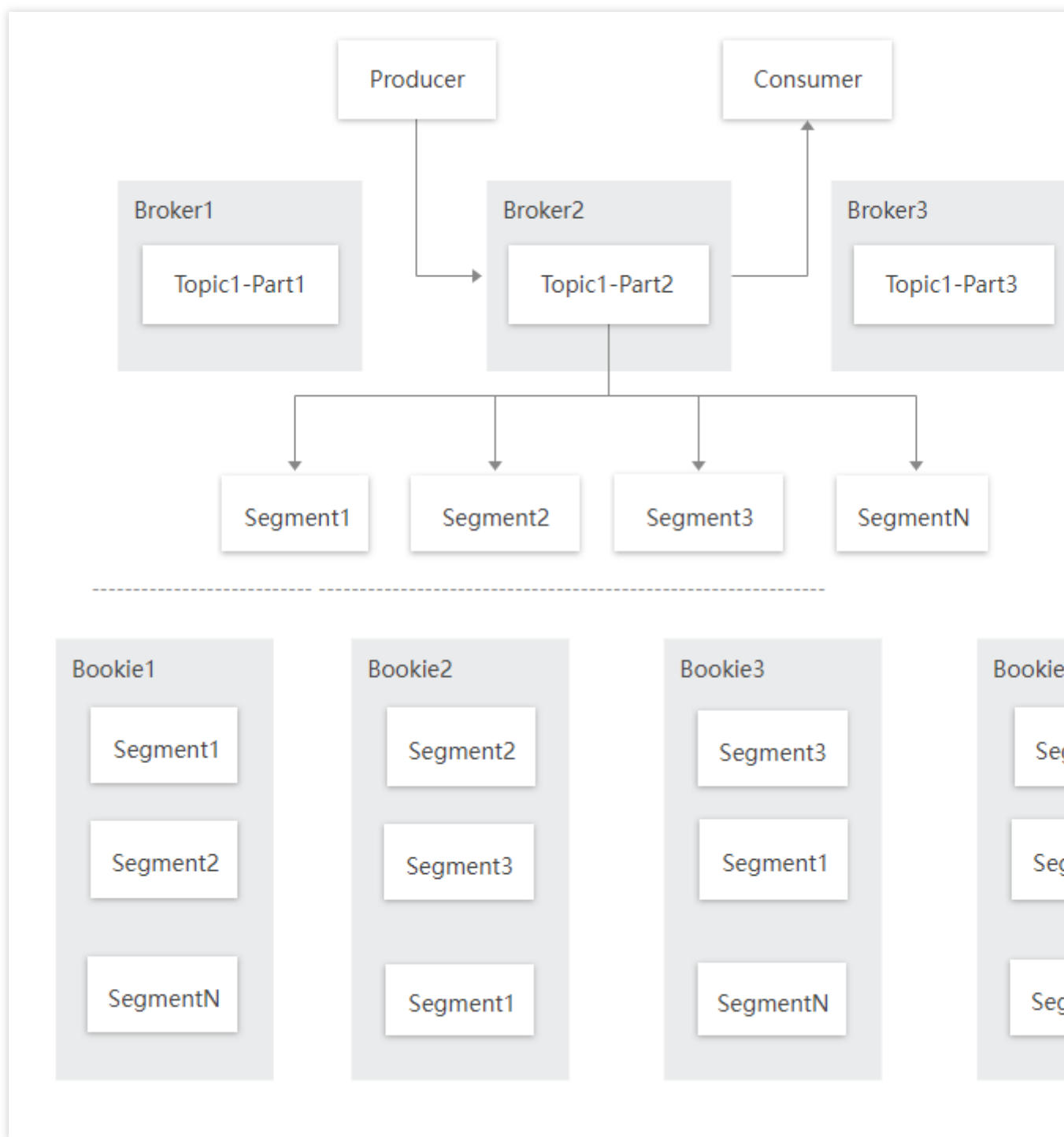
Topic and Partition

Topic is a category name where messages are stored and published. Producers write messages to topics, and consumers read the messages from these topics.

Pulsar topics are divided into partitioned topic and non-partitioned topic, with the latter referring to a topic with 1 partition. In fact, topic is a virtual concept in Pulsar, and a 3-partition topic actually refers to three partitioned topics, and messages sent to a 3-partition topic will be sent to these three partitioned topics.

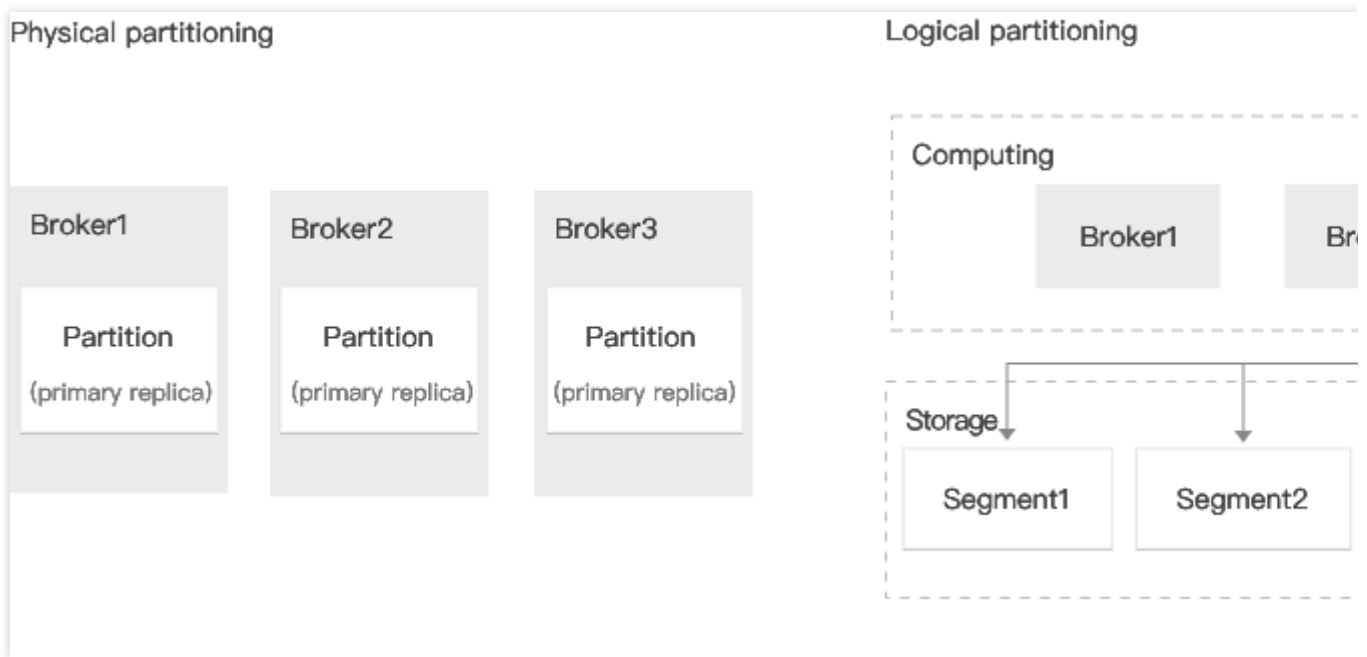
For example, if a producer sends a message to a 3-partition topic named `my-topic`, the message is actually sent to the 3 partitioned topics `my-topic-partition-0`, `my-topic-partition-1`, and `my-topic-partition-2` evenly or according to a rule (if a key is specified).

When partitioned topics persistently store data, partition is a logical concept, and the actual storage unit is segment. As shown below, data in the `Topic1-Part2` partition consists of N segments, each of which is evenly distributed and stored on multiple bookie nodes in the Apache BookKeeper cluster and has 3 replicas.



Physical Partition and Logical Partition

Logical partitions and physical partitions are compared as follows:



Physical partition: computing and storage are coupled, fault tolerance requires copying physical partitions, and capacity expansion requires migrating physical partitions to implement load balancing.

Logical partition: physical segment where the computing layer is isolated from the storage layer. With this structure, Apache Pulsar has the following strengths:

Brokers and bookies are independent of each other, making it easier to implement separate capacity expansion and fault tolerance.

Brokers are stateless, so that they can be connected and disconnected quickly, making them more suitable for cloud native scenarios.

Partitioned storage is not limited by the storage capacity of a single node.

Partitioned data is evenly distributed. In this way, the excessive amount of data in a single partition will not cause the entire cluster to overflow.

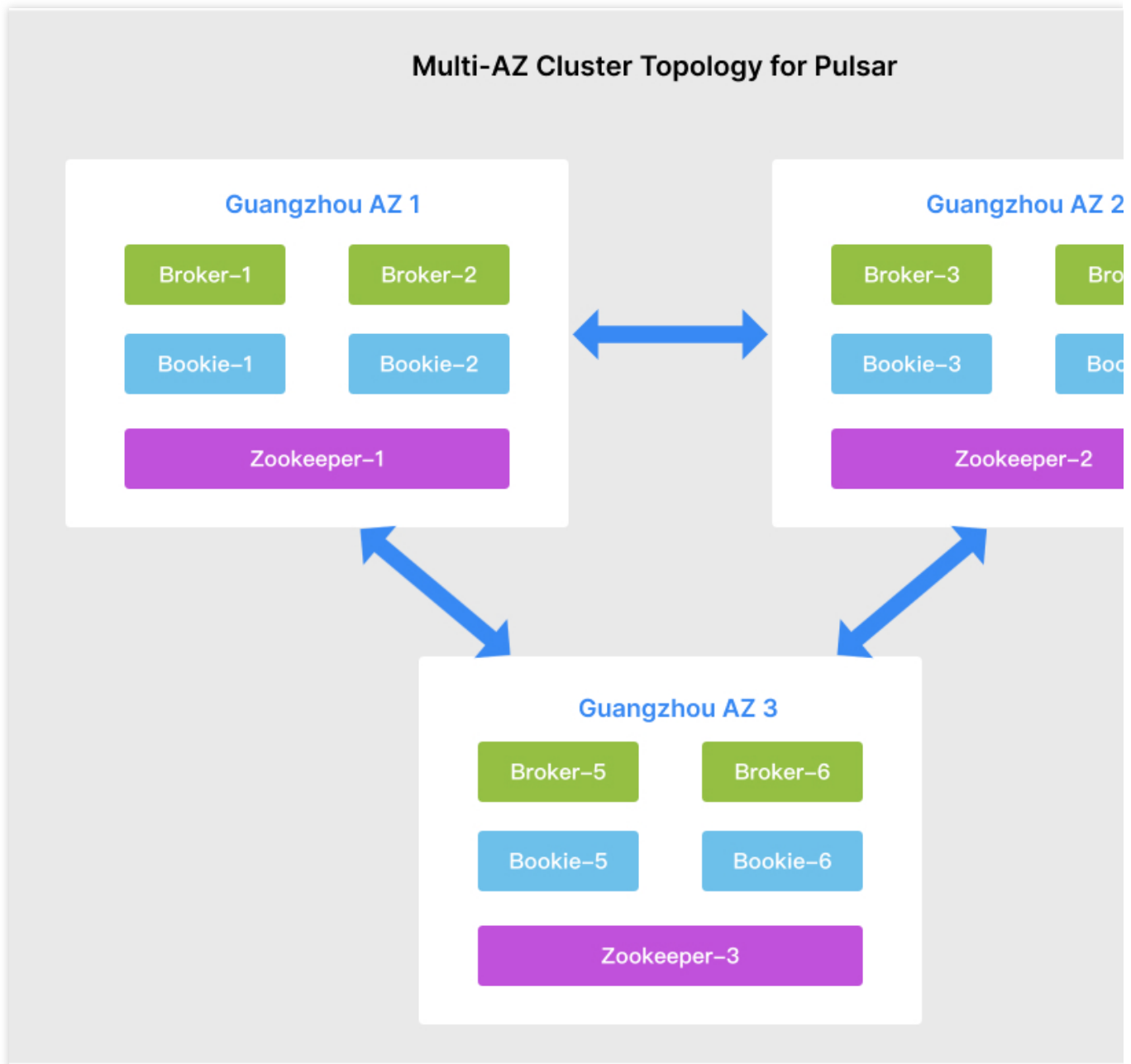
During storage capacity expansion, new nodes can be quickly added to share the storage load.

Pulsar Multi-AZ Deployment

Last updated : 2024-06-28 11:29:49

Pulsar Multi-AZ Deployment

Pulsar supports multi-AZ deployment. When you purchase a Pulsar cluster in a region with three or more AZs, you can select up to three AZs for multi-AZ instance deployment. The partition replicas of this instance will be forcibly distributed to the nodes in each AZ. This deployment mode allows your instance to normally provide services even if it is unavailable in a single AZ.



How multi-AZ deployment works

Pulsar implements cross-AZ disaster recovery based on rack awareness. Although the service nodes of different components are deployed in different racks in different AZs, they are essentially in a single Pulsar cluster.

As a type of ensemble placement policy, rack awareness is the algorithm used by BookKeeper client to select ensembles based on the network topology attribute.

How to select bookie

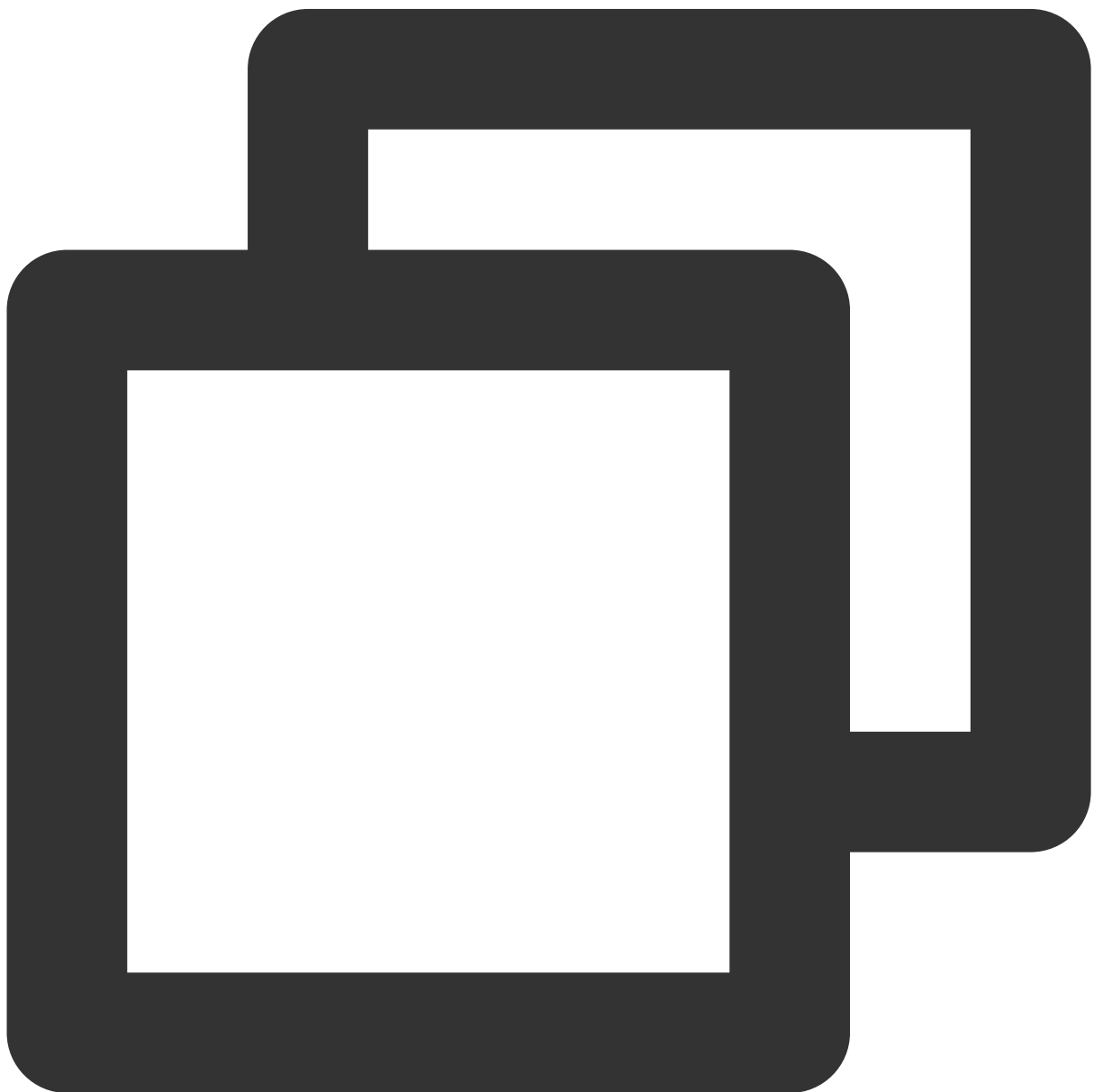
When a bookie cluster is deployed for the first time, each bookie will register a temporary zk-node to the current ZooKeeper, and the BookKeeper client will discover the bookie list via ZooKeeper. When a bookie changes or fails, the ensemble placement policy will be notified of this through the `onClusterChanged (Set, Set)` API and

reconstruct a new network topology. Subsequent operations, such as `newEnsemble` (used to generate a group of bookies), will be performed based on the new network topology.

Network topology

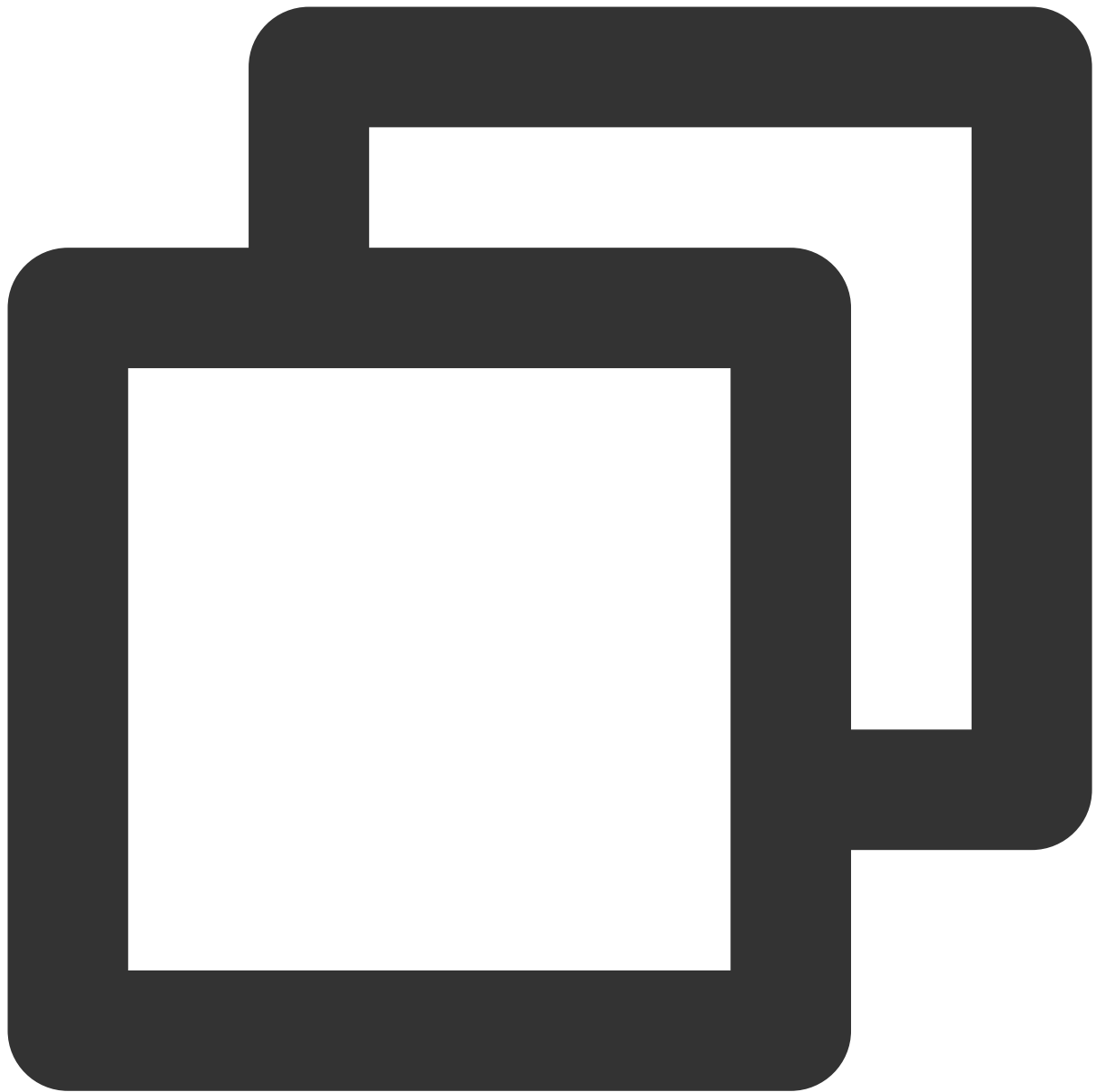
Network topology shows the bookie node information in a cluster in a tree-like hierarchical structure. A bookie cluster may consist of many data centers in different regions. A data center contains bookie nodes distributed on different racks. In the tree-like structure, the leaf nodes show the bookie information.

Example 1: There are three bookies: bk1, bk2, and bk3 in region A. Their network locations are `/region-a/rack-1/bk1`, `/region-a/rack-1/bk2`, and `/region-a/rack-2/bk3`, respectively, as shown in the network topology below:



```
    root
    |
  region-a
    /  \
rack-1 rack-2
  /  \    \
bk1  bk2  bk3
```

Example 2: There are four bookies: bk1, bk2, bk3, and bk4 in region A and region B. Their network locations are /region-a/rack-1/bk1, /region-a/rack-1/bk2, /region-b/rack-2/bk3, and /region-b/rack-2/bk4, respectively, as shown in the network topology below:



```
      root
     /   \
  region-a region-b
    |       |
  rack-1   rack-2
  /  \     /  \
bk1 bk2 bk3 bk4
```

The network locations are resolved with the `DNSToSwitchMapping` method from domain names or IP addresses. They must be in the format of `"/region/rack"`, where the first slash (/) represents root, "region" represents data center, and "rack" represents rack information.

Note:

The `DNSToSwitchMapping` method implemented in Pulsar is called `BookieRackAffinityMapping`. It uses ZooKeeper to store rack information and only supports the rack awareness capability currently.

Use Limit

Currently, only pro clusters support multi-AZ deployment. For more information, see [Product Selection](#).

Note

This feature is currently in beta testing and is free of charge.

Message Storage Principle and ID Generation Rule

Last updated : 2024-06-28 11:29:49

MessageID Generation Rules

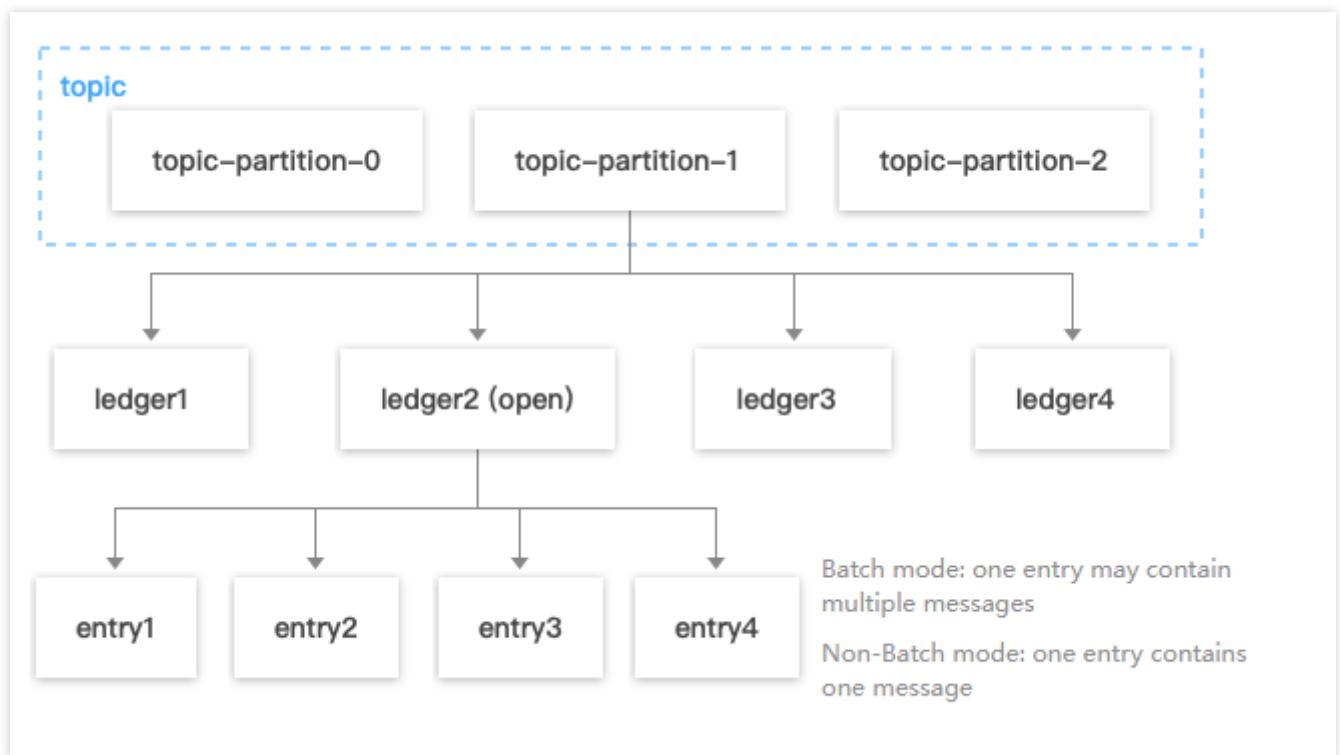
In Pulsar, each message has its own ID, namely `MessageID`, which consists of four parts:

`ledgerId:entryID:partition-index:batch-index` :

partition-index: Partition number, which is -1 for non-partitioned topics.

batch-index: -1 for non-batch messages.

The `MessageID` generation rules are determined by Pulsar's message storage mechanism as shown below:



As shown above, each topic partition in Pulsar corresponds to a series of ledgers, with only one ledger is in open (i.e., writable) status and each ledger storing only messages in its corresponding partition.

When Pulsar stores a message, it will first find the ledger used by the current partition and then generate an entry ID for the message. Entry IDs are incremental in the same ledger. After the existence duration of a ledger or number of entries stored in it exceeds the threshold, new messages will be stored in the next ledger in the same partition.

In case of batch message production, one entry may contain multiple messages.

In case of non-batch message production, one entry contains one message (this parameter can be configured on the producer side, and batch production is used by default).

Ledger is just a logical dimension for data assembly and has no actual entity. Bookies write, find, and get data only by entry.

Ledger and Entry

Ledger

Pulsar stores message data in a ledger format on bookie nodes in a BookKeeper cluster. A ledger is an append-only data structure with a single writer that is assigned to multiple bookies. Ledger entries are replicated to multiple bookies, and relevant data is written to ensure data consistency.

The data that BookKeeper needs to save include:

Journal

A journal file stores BookKeeper transaction logs. Before any ledger update is made, its description information needs to be persisted to a journal file first.

BookKeeper provides a separate sync thread to roll the current journal file according to its size.

Entry log file

An entry log file stores real data. Entry data from different ledgers is cached in the memory buffer first and then batch flushed to an entry log file.

By default, data from all ledgers is aggregated and then sequentially written to the same entry log file to avoid random disk writes.

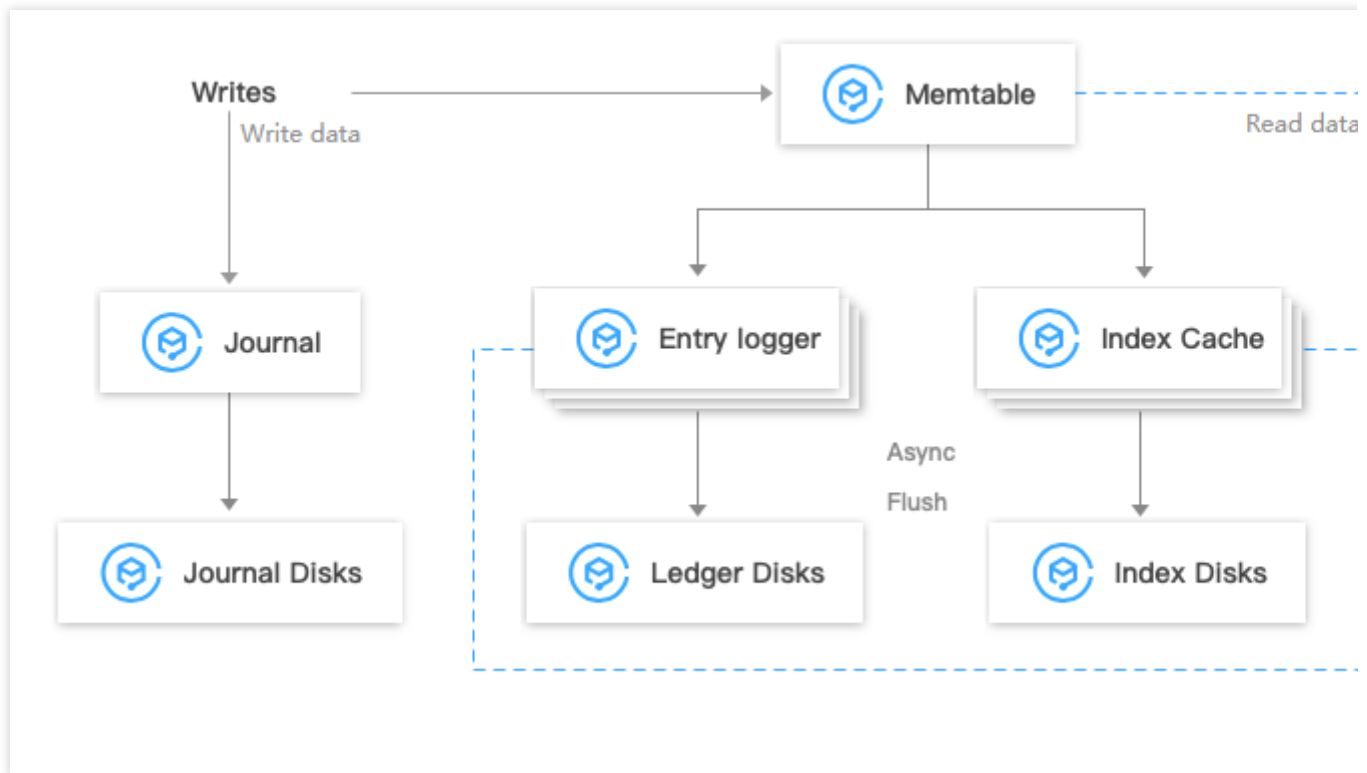
Index file

Entry data from all ledgers is written to the same entry log file. To accelerate data writes, a mapping from

`ledgerId` + `entryId` to the file offset will be performed, which will be cached in the memory called "index cache".

When the index cache capacity reaches the upper limit, it will be flushed to the disk by the sync thread.

The read and write interactions of the three types of data files are as shown below:



Entry

Entry data write

1. Data will be first written to the journal (stored in the disk in real time) and memtable (read/write cache) at the same time.
2. After the data is written to the memtable, a response will be given to the write request.
3. After the memtable is full, data will be flushed to the entry logger (for data storage) and index cache (for data index information storage).
4. The backend thread will store the data in the entry logger and index cache to the disk.

Entry data read

Tailing read request: Entries will be read directly from the memtable.

Catch-up read (lagged consumption) request: The index information will be read first, and then the index will read entries from the entry logger file.

Data consistency guarantee: LastLogMark

The written entry log and index are cached in the memory first and then flushed to the disk periodically according to particular conditions, which causes an interval between memory caching and disk storage. If the BookKeeper process crashes during this interval, after it is restarted, the data needs to be recovered from the journal file. In this case, the `LastLogMark` marks the position in the journal from where to recover.

`LastLogMark` is actually stored in the memory. When the index cache is flushed to the disk, its value will be updated. It will also be persisted to the disk periodically for data recovery from the journal when the Bookkeeper process starts.

Once `LastLogMark` is persisted to the disk, the preceding index and entry log have been persisted to the disk, and the data in the journal before the `LastLogMark` can be cleared.

Message Replica and Storage Mechanisms

Last updated : 2024-06-28 11:29:49

Message Metadata Composition

In Pulsar, the message data in each partitioned topic is stored in the form of ledger on bookie storage nodes in the BookKeeper cluster. Each ledger contains a set of entries, and bookies write, find, and get data only by entry.

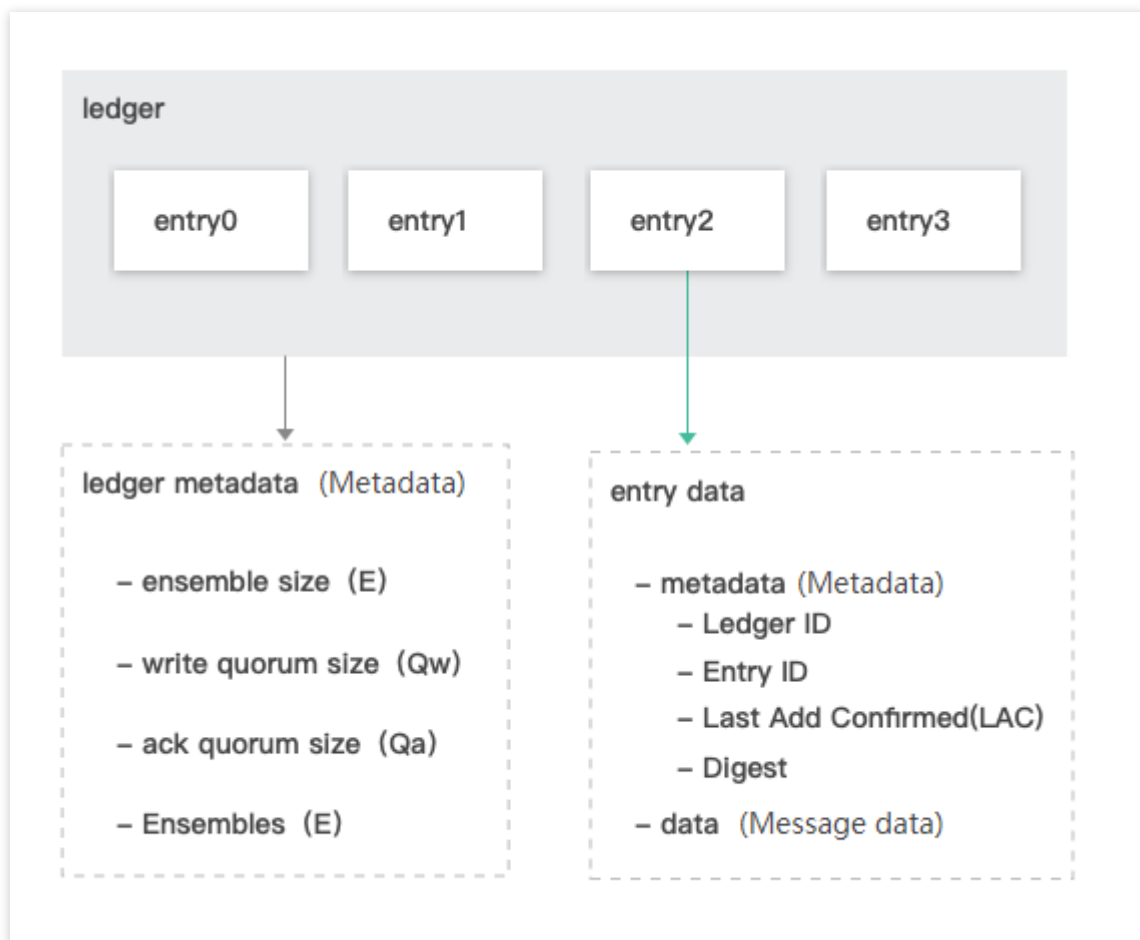
Note:

In case of batch message production, one entry may contain multiple messages, so the entries do not necessarily correspond to messages one by one.

Ledgers and entries correspond to different metadata respectively.

The metadata of ledgers is stored in ZooKeeper.

In addition to message data, entries also contain metadata. The data of entries is stored on the bookie nodes.



Type	Parameter	Description	Data Storage Location

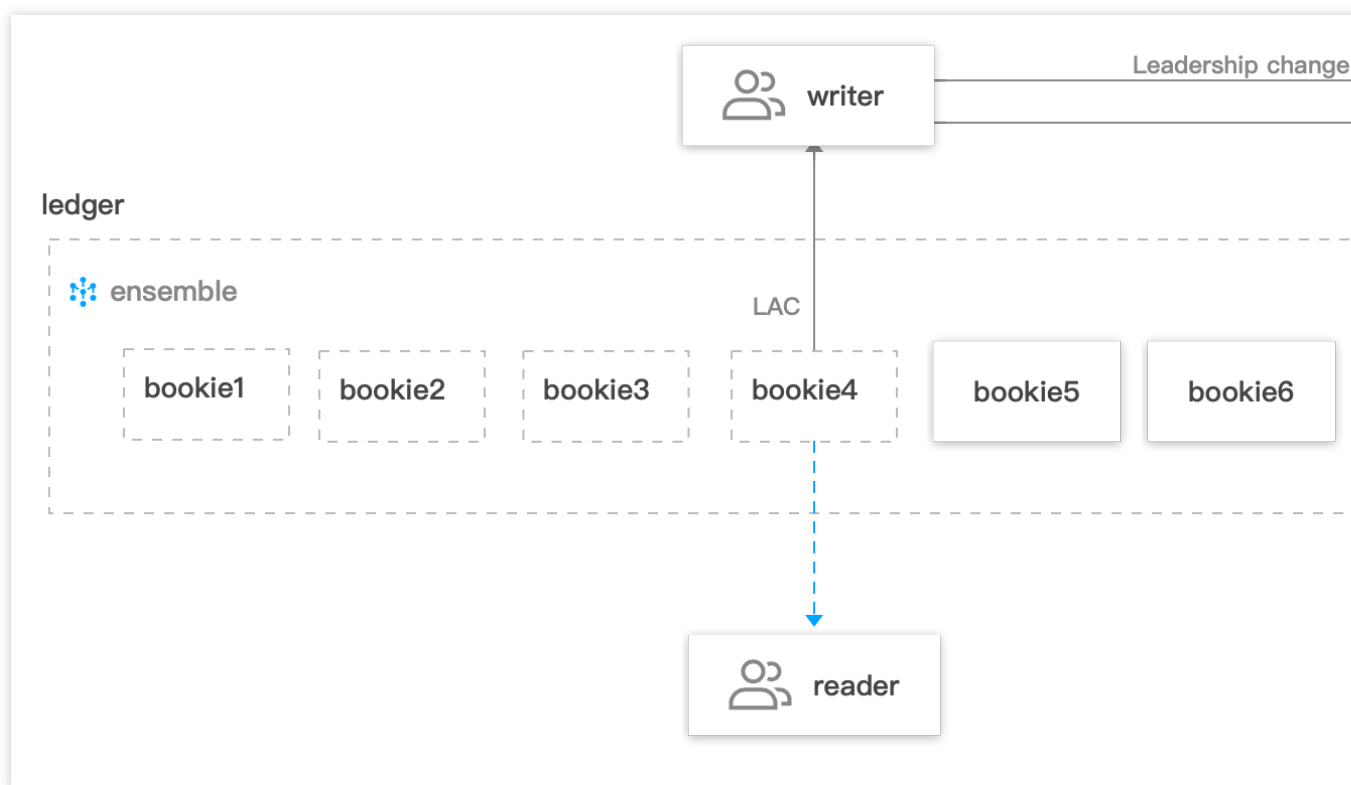
Ledger	ensemble size (E)	Number of bookie nodes selected by each ledger	Metadata is stored in ZooKeeper
	write quorum size (Qw)	Number of bookies to which each entry needs to send write requests	
	ack quorum size (Qa)	Number of write acks that should be received before a write can be considered successful	
	Ensembles (E)	The ensemble list used in the format of <entry id,="" ensembles=""> tuple key (entry ID): the entry ID at the start of the ensemble list value (ensembles): the list of IPs of the bookies selected by the ledger. Each value contains E IPs Each ledger may contain multiple ensemble lists and can have at most one ensemble list in use at any time	
Entry	Ledger ID	ID of the ledger of the entry	Data is stored on bookie nodes
	Entry ID	ID of the current entry	
	Last Add Confirmed	Entry ID of the known latest write ack when the current entry is created	
	Digest	CRC	

During the creation of each ledger, E bookie nodes will be selected from the list of existing candidate bookie nodes in writable status in the BookKeeper cluster. If there are not enough candidate nodes, the

`BKNotEnoughBookiesException` exception will be thrown. After the candidate nodes are selected, this information will be combined to form the <entry id, ensembles> tuple and stored in ensembles of the ledger's metadata.

Message Replica Mechanism

Message write process



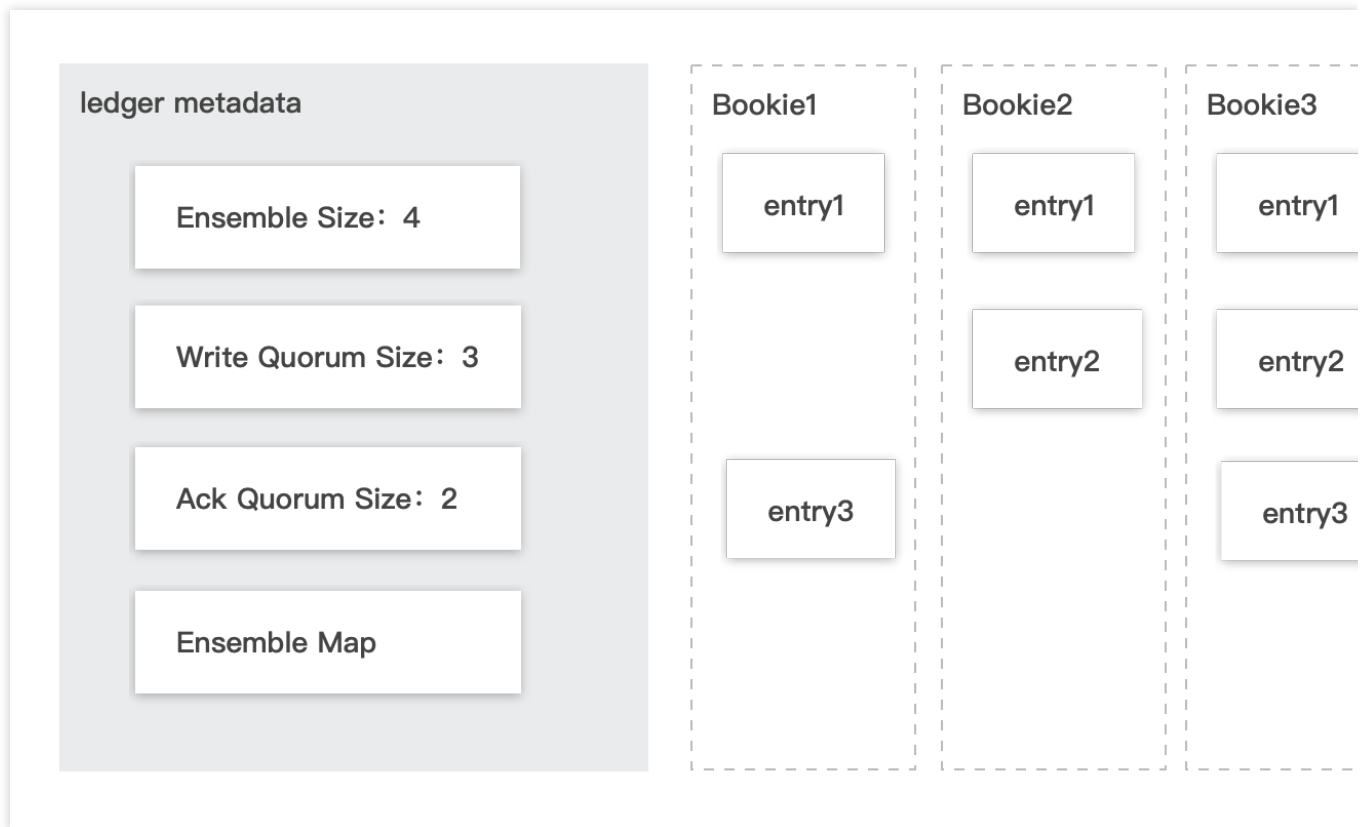
When the client writes a message, each entry will send a write request to Qw bookie nodes in the ensemble list currently used by the ledger. After Qa write acks are received, the current message will be considered written and stored successfully, and LAP (lastAddPushed) and LAC (LastAddConfirmed) will be used to mark the position currently pushed and the position where the storage ack is received respectively.

The value of the LAC metadata in each entry being pushed is the position value of the last received ack when an entry sending request is created at the current moment. The position of LAC and preceding messages are visible to the read client.

In addition, Pulsar uses the fencing mechanism to prevent multiple clients from writing to the same ledger at the same time. This is suitable for scenarios where the ownership of one topic/partition is transferred from one broker to another.

Message replica distribution

When each entry is written, the set of Qw bookie nodes in the ensemble list with which the current entry needs to be written will be determined based on the entry ID of the current message and the entry ID at the start of the currently used ensemble list (key value). Then, the broker will send a write request to these bookie nodes. After Qa write acks are received, the current message will be considered written and stored successfully. At this point, Qa message replicas can be guaranteed at the least.



As shown above, the ledger selects 4 bookie nodes (bookies 1–4), a message is written to 3 nodes each time, and after 2 write acks are received, the message will be considered stored successfully. The ensemble selected by the current ledger uses bookies 1, 2, and 3 to write entry 1 and uses bookies 2, 3, and 4 to write entry 2, and entry 3 will be written to bookies 3, 4, and 1 according to the calculation result.

Message Recovery Mechanism

By default, each bookie in a Pulsar BookKeeper cluster is started with the recovery service enabled automatically, which performs the following tasks:

1. Auditor election (by AuditorElector).
2. Task replication (by ReplicationWorker).
3. Failure monitoring (by DeathWatcher).

Bookie nodes in a BookKeeper cluster will elect a master bookie through ZooKeeper's temporary node mechanism, which mainly performs the following tasks:

1. Monitor changes in bookie nodes.
2. Mark the ledgers on failed bookies in ZooKeeper `Underreplicated`.
3. Check the number of replicas of all ledgers (the default cycle is one week).
4. Check the number of replicas of entries (not enabled by default).

The data in a ledger is recovered by fragment (each fragment corresponds to a set of ensemble lists in the ledger; if there are multiple lists, multiple fragments need to be processed).

To perform recovery, the first step is to determine which storage nodes in which fragments of the current ledger need to be replaced with new candidate nodes for data recovery. If there is no entry data on an associated bookie node in a fragment (determined based on whether the entries at the start and end exist by default), the bookie node needs to be replaced, and the fragment requires data recovery.

After the data in the fragment is recovered with the new bookie node, the original data of the ensemble list in the current fragment will be updated in the metadata of the ledger.

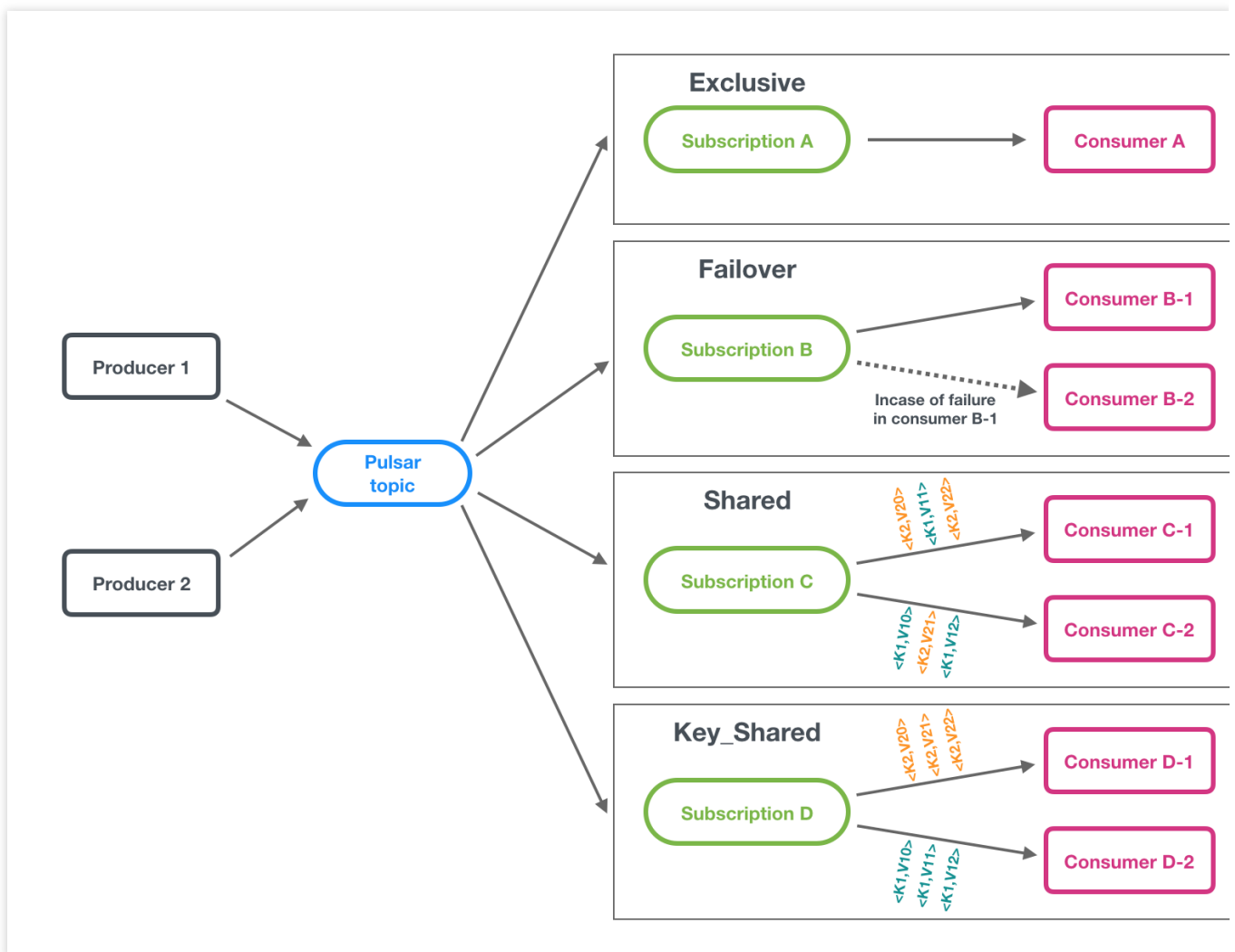
In scenarios where the number of data replicas is reduced by failures of bookie nodes, after this process is completed, the number will be gradually restored to Qw (i.e., the number of replicas specified on the backend, which is 3 in TDMQ by default).

Best Practices

Subscription Mode

Last updated : 2024-06-28 11:29:49

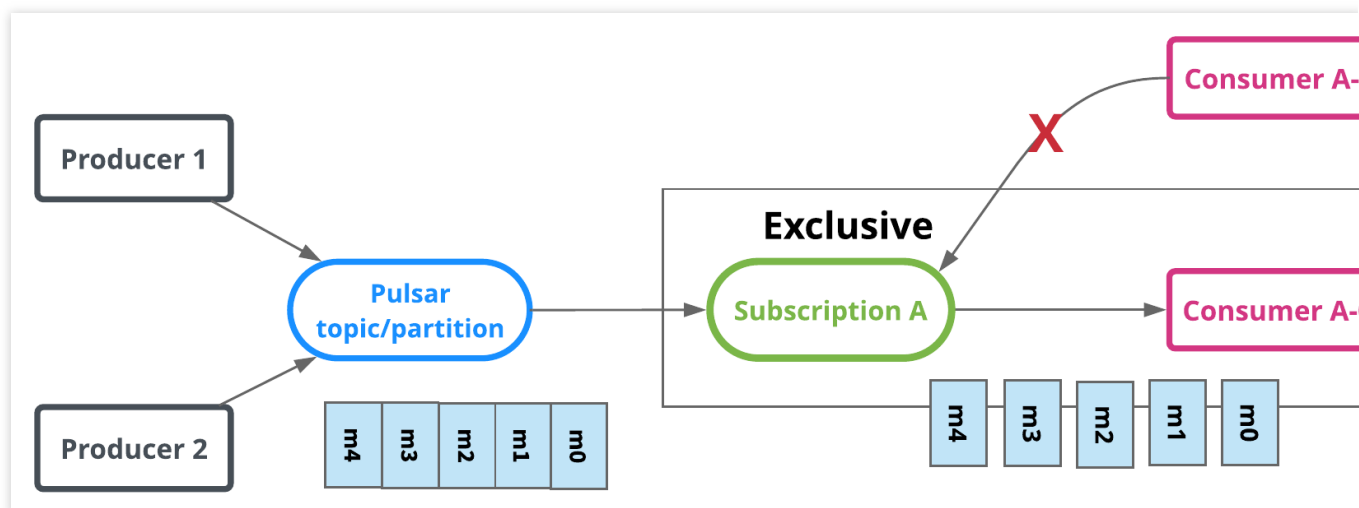
In order to meet the needs of different use cases, TDMQ for Apache Pulsar supports four subscription modes: exclusive, shared, failover, and key_shared.

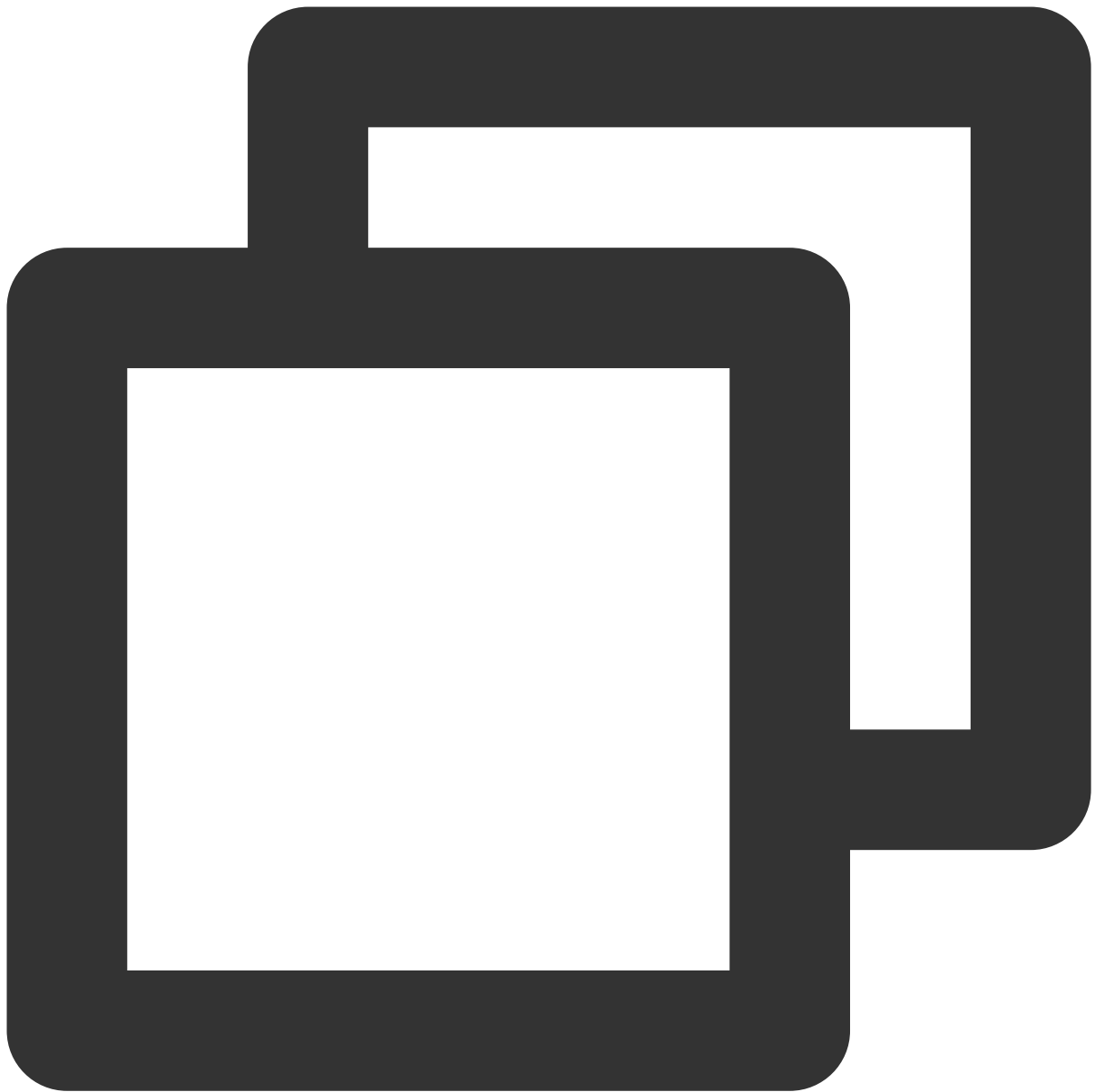


Exclusive Mode

Exclusive mode (default): A subscription can only be associated with one consumer. Only this consumer can receive all messages in the topic, and if it fails, consumption will stop.

In the exclusive subscription mode, only one consumer in a subscription can consume messages in the topic. If multiple consumers subscribe to the topic, an error will be reported. This mode is suitable for globally sequential consumption scenarios.





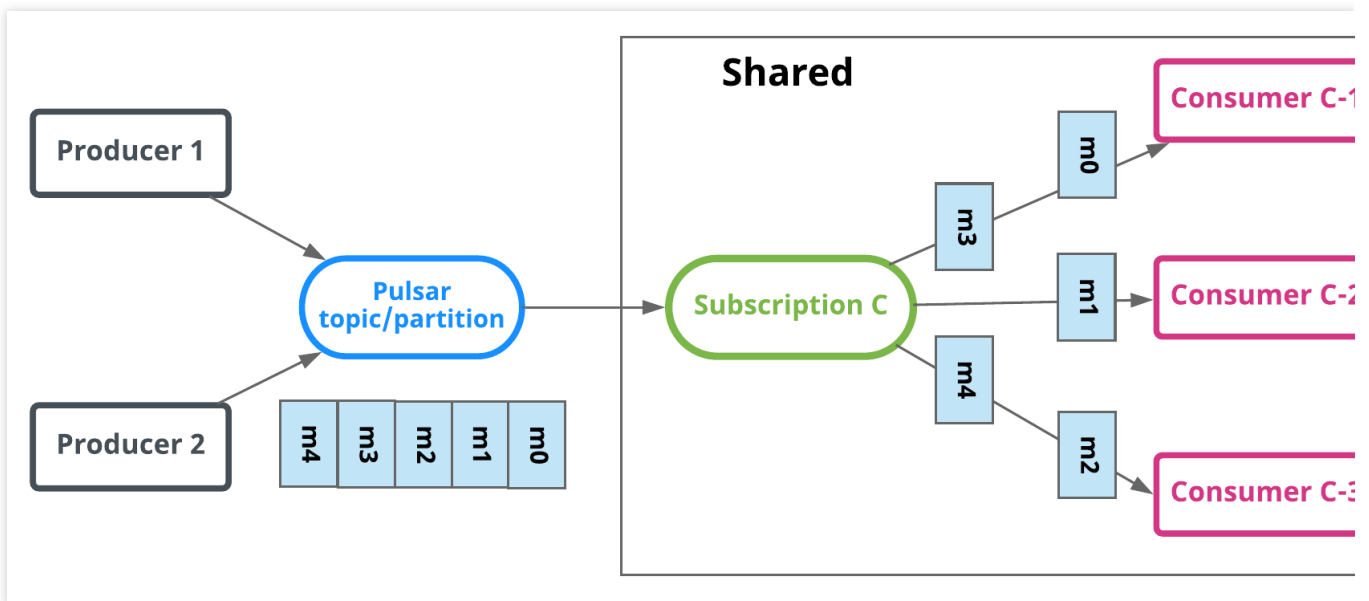
```
// Construct a consumer
Consumer<byte[]> consumer = pulsarClient.newConsumer()
    // Complete path of the topic in the format of `persistent://cluster (tenant) I
    .topic("persistent://pulsar-xxx/sdk_java/topic1")
    // You need to create a subscription on the topic details page in the console a
    .subscriptionName("sub_topic1")
    // Declare the exclusive mode to be the consumption mode
    .subscriptionType(SubscriptionType.Exclusive)
    .subscribe();
```

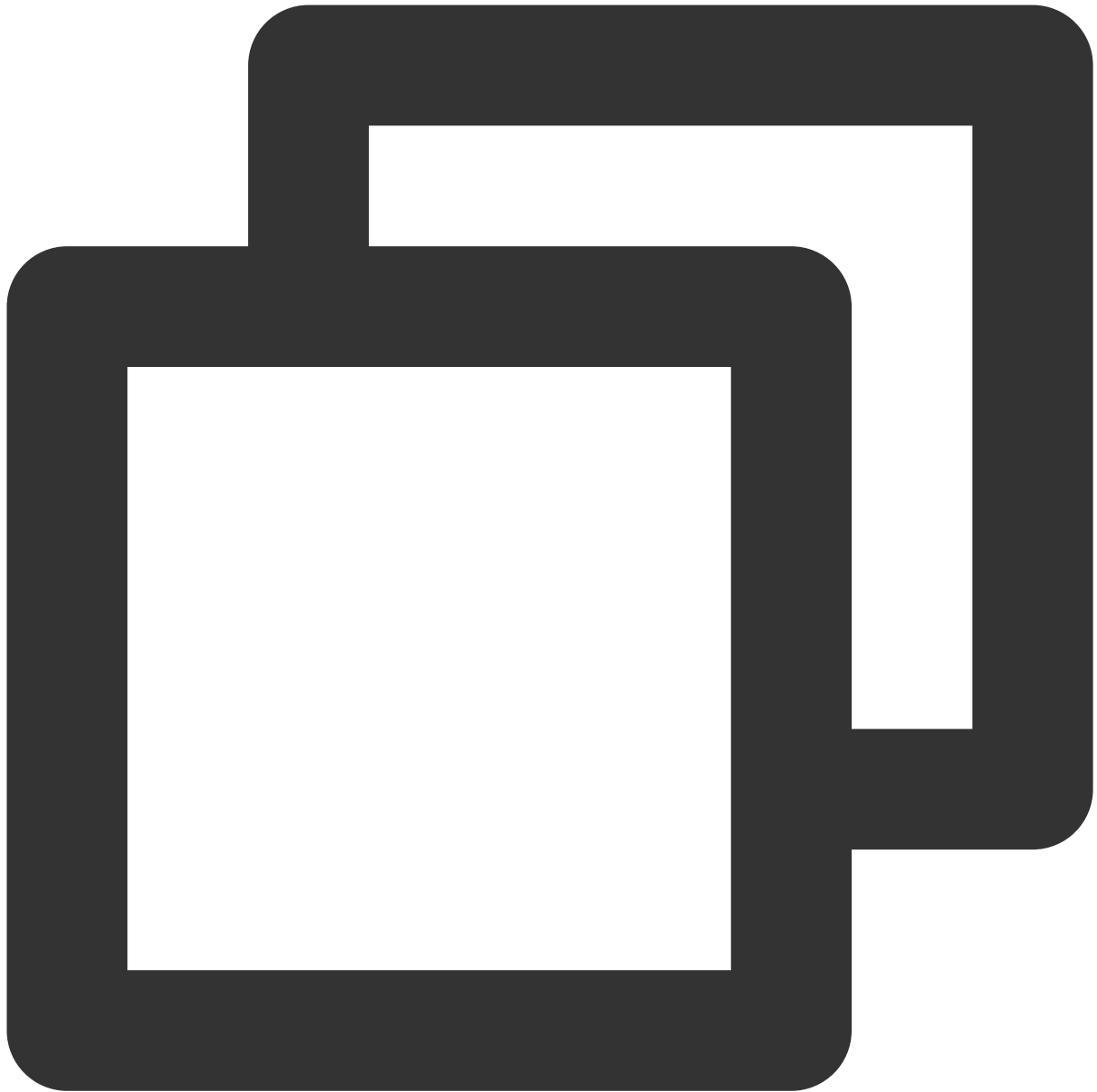
If multiple consumers are started, an error will be reported.

```
\java.exe ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/ for further details.
>> pulsar client created.
Exception in thread "main" org.apache.pulsar.client.api.PulsarClientException$ConsumerBusyException: Failed to subscribe persistent
Exclusive consumer is already connected
    at org.apache.pulsar.client.api.PulsarClientException.unwrap(PulsarClientException.java:946)
    at org.apache.pulsar.client.impl.ConsumerBuilderImpl.subscribe(ConsumerBuilderImpl.java:102)
    at com.tencent.cloud.tdmq.pulsar.simple.ListenerConsumer.main(ListenerConsumer.java:41)
```

Shared Mode

Messages are distributed to different consumers through a customizable round robin mechanism, with each message going to only one consumer. When a consumer is disconnected, any messages delivered to it but not acknowledged will be redistributed to other active consumers.





```
// Construct a consumer
Consumer<byte[]> consumer = pulsarClient.newConsumer()
    // Complete path of the topic in the format of `persistent://cluster (tenant) I
    .topic("persistent://pulsar-xxx/sdk_java/topic1")
    // You need to create a subscription on the topic details page in the console a
    .subscriptionName("sub_topic1")
    // Declare the shared mode to be the consumption mode
    .subscriptionType(SubscriptionType.Shared)
    .subscribe();
```

There can be multiple consumers in the shared mode.

```

Run: SharedConsumer1
\java.exe ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www. for further details.
>> pulsar client created.
>> pulsar consumer1 created.
receive msg 67004431:20:0:0,value:my-sync-message-1
receive msg 67004430:26:2:0,value:my-sync-message-3
receive msg 67004432:21:1:0,value:my-sync-message-5
receive msg 67004431:22:0:0,value:my-sync-message-7
receive msg 67004430:28:2:0,value:my-sync-message-9
Process finished with exit code 0

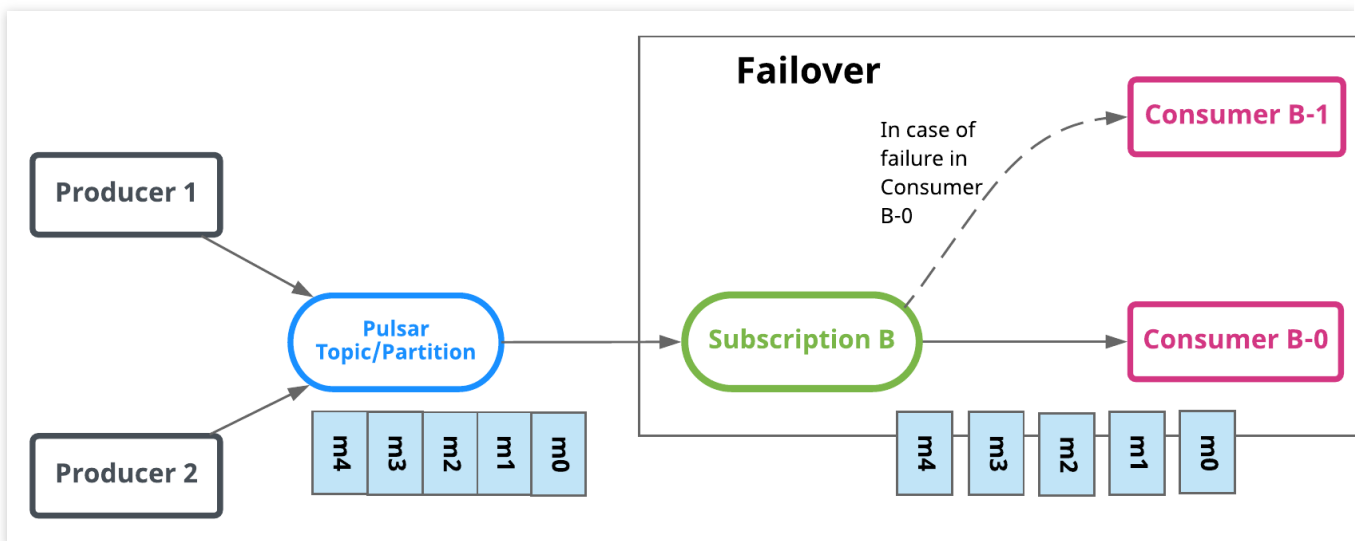
SharedConsumer2
\java.exe ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www. for further details.
>> pulsar client created.
>> pulsar consumer2 created.
receive msg 67004430:25:2:0,value:my-sync-message-0
receive msg 67004432:20:1:0,value:my-sync-message-2
receive msg 67004431:21:0:0,value:my-sync-message-4
receive msg 67004430:27:2:0,value:my-sync-message-6
receive msg 67004432:22:1:0,value:my-sync-message-8
Process finished with exit code 0

SimpleProducer
\java.exe ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www. for further details.
>> pulsar client created.
>> pulsar producer created.
deliver msg 67004430:25:2:0,value:my-sync-message-0
deliver msg 67004431:20:0:0,value:my-sync-message-1
deliver msg 67004432:20:1:0,value:my-sync-message-2
deliver msg 67004430:26:2:0,value:my-sync-message-3
deliver msg 67004431:21:0:0,value:my-sync-message-4
deliver msg 67004432:21:1:0,value:my-sync-message-5
deliver msg 67004430:27:2:0,value:my-sync-message-6
deliver msg 67004431:22:0:0,value:my-sync-message-7
deliver msg 67004432:22:1:0,value:my-sync-message-8
deliver msg 67004430:28:2:0,value:my-sync-message-9
Process finished with exit code 0

```

Failover Mode

In this mode, when there are multiple consumers, they will be sorted lexicographically, and the first consumer is initialized to be the only one who can receive messages. When the first consumer is disconnected, all the unacknowledged and upcoming messages will be distributed to the next consumer in the queue.





```
// Construct a consumer
Consumer<byte[]> consumer = pulsarClient.newConsumer()
    // Complete path of the topic in the format of `persistent://cluster (tenant) I
    .topic("persistent://pulsar-xxx/sdk_java/topic1")
    // You need to create a subscription on the topic details page in the console a
    .subscriptionName("sub_topic1")
    // Declare the failover mode to be the consumption mode
    .subscriptionType(SubscriptionType.Failover)
    .subscribe();
```

There can be multiple consumers in the failover mode.

```

FailoverConsumer1
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www. for further details.
>> pulsar client created.
>> pulsar consumer created.
receive msg 67004430:48:2:0,value:my-sync-message-0
receive msg 67004431:36:0:0,value:my-sync-message-1
receive msg 67004430:49:2:0,value:my-sync-message-3
receive msg 67004431:37:0:0,value:my-sync-message-4
receive msg 67004430:50:2:0,value:my-sync-message-6
Process finished with exit code 0

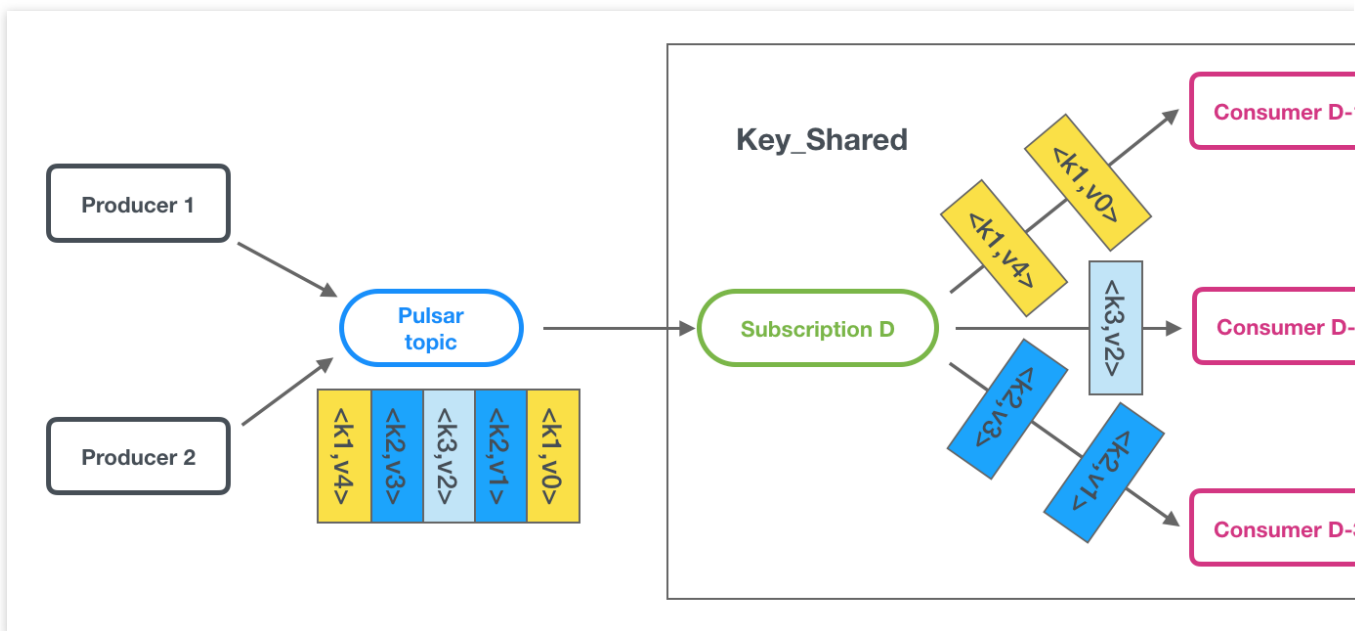
FailoverConsumer2
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www. for further details.
>> pulsar client created.
>> pulsar consumer created.
receive msg 67004432:39:1:0,value:my-sync-message-2
receive msg 67004432:40:1:0,value:my-sync-message-5
receive msg 67004431:38:0:0,value:my-sync-message-7
receive msg 67004432:41:1:0,value:my-sync-message-8
receive msg 67004430:51:2:0,value:my-sync-message-9
Process finished with exit code 0

SimpleProducer
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www. for further details.
>> pulsar client created.
>> pulsar producer created.
deliver msg 67004430:48:2:0,value:my-sync-message-0
deliver msg 67004431:36:0:0,value:my-sync-message-1
deliver msg 67004432:39:1:0,value:my-sync-message-2
deliver msg 67004430:49:2:0,value:my-sync-message-3
deliver msg 67004431:37:0:0,value:my-sync-message-4
deliver msg 67004432:40:1:0,value:my-sync-message-5
deliver msg 67004430:50:2:0,value:my-sync-message-6
deliver msg 67004431:38:0:0,value:my-sync-message-7
deliver msg 67004432:41:1:0,value:my-sync-message-8
deliver msg 67004430:51:2:0,value:my-sync-message-9
Process finished with exit code 0

```

Key_Shared mMode

If there are multiple consumers, messages will be distributed by key, and messages with the same key will only be distributed to the same consumer.



Note:

The key_shared mode has certain use limits. It is continuously iterated in the community due to its complex engineering implementation. Therefore, it doesn't have the same level of stability as exclusive, failover, and shared modes. We recommend that you first select the other three modes if they can meet your business needs.

Pro clusters can guarantee the sequential delivery of messages with the same key, while virtual clusters cannot.

Suggestions for the key_shared mode

When to use the key_shared mode

Choose the shared mode for general production/consumption scenarios.

If you want messages with the same key to be distributed to the same consumer, you cannot use the shared mode.

You have two options:

Choose the key_shared mode.

Use a multi-partition topic + failover mode.

Where to use the key_shared mode

There are a lot of message keys with even message distribution.

Consumption is fast with no message heap.

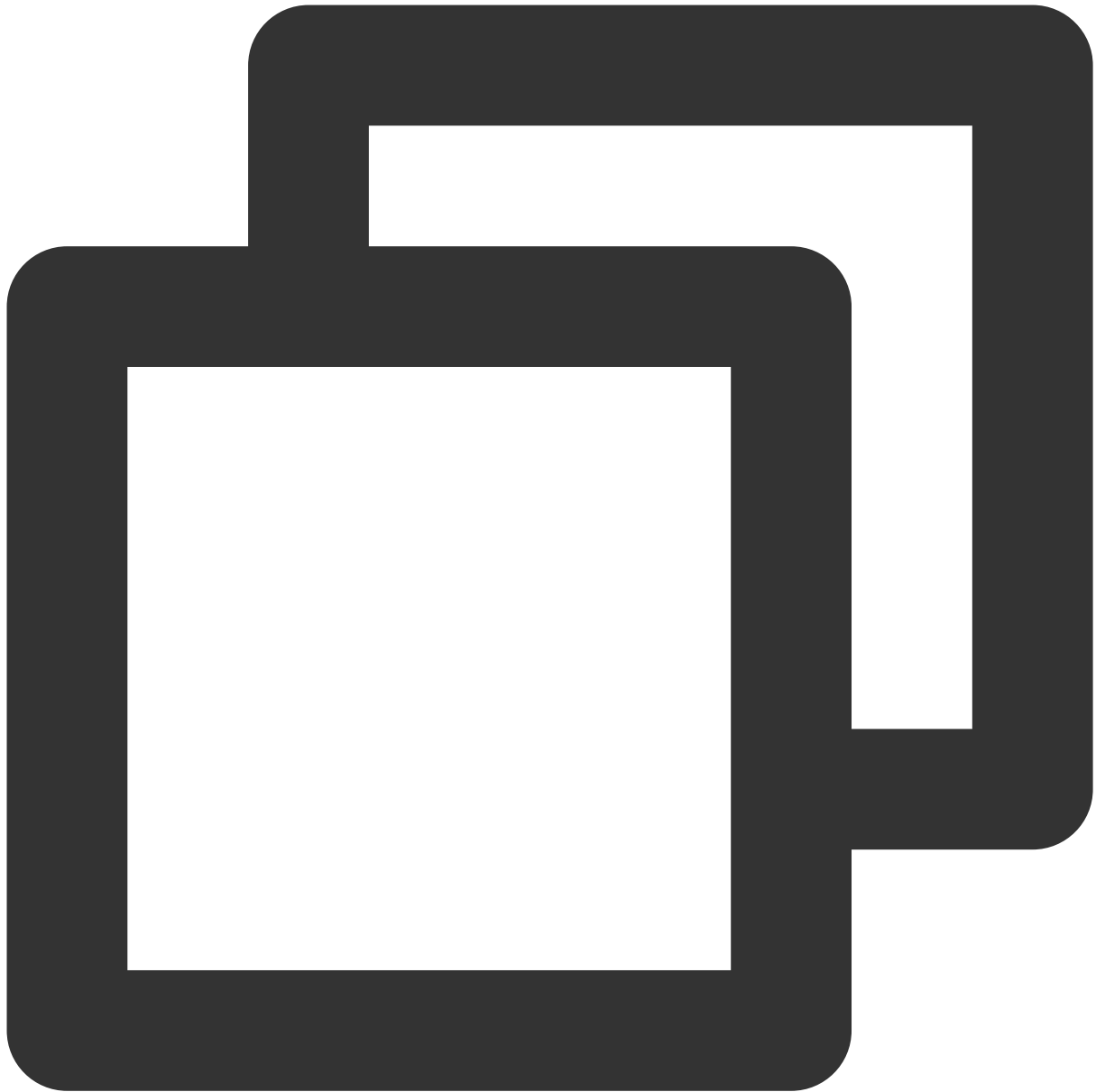
If the above two conditions cannot be met in the production process, we recommend that you use the combo of multi-partition topic and failover mode.

Sample code

Sample key_shared subscription

By default, Pulsar enables the batch feature when producing messages, and batch messages are parsed on the consumer side. Therefore, a batch of messages on the broker side is treated as one entry. Since messages with different keys may be packaged into the same batch, the key_shared mode will become ineffective in this case because it achieves sequential subscription based on the same message key. There are two ways to avoid this when creating a producer:

1. Disable the batch feature.

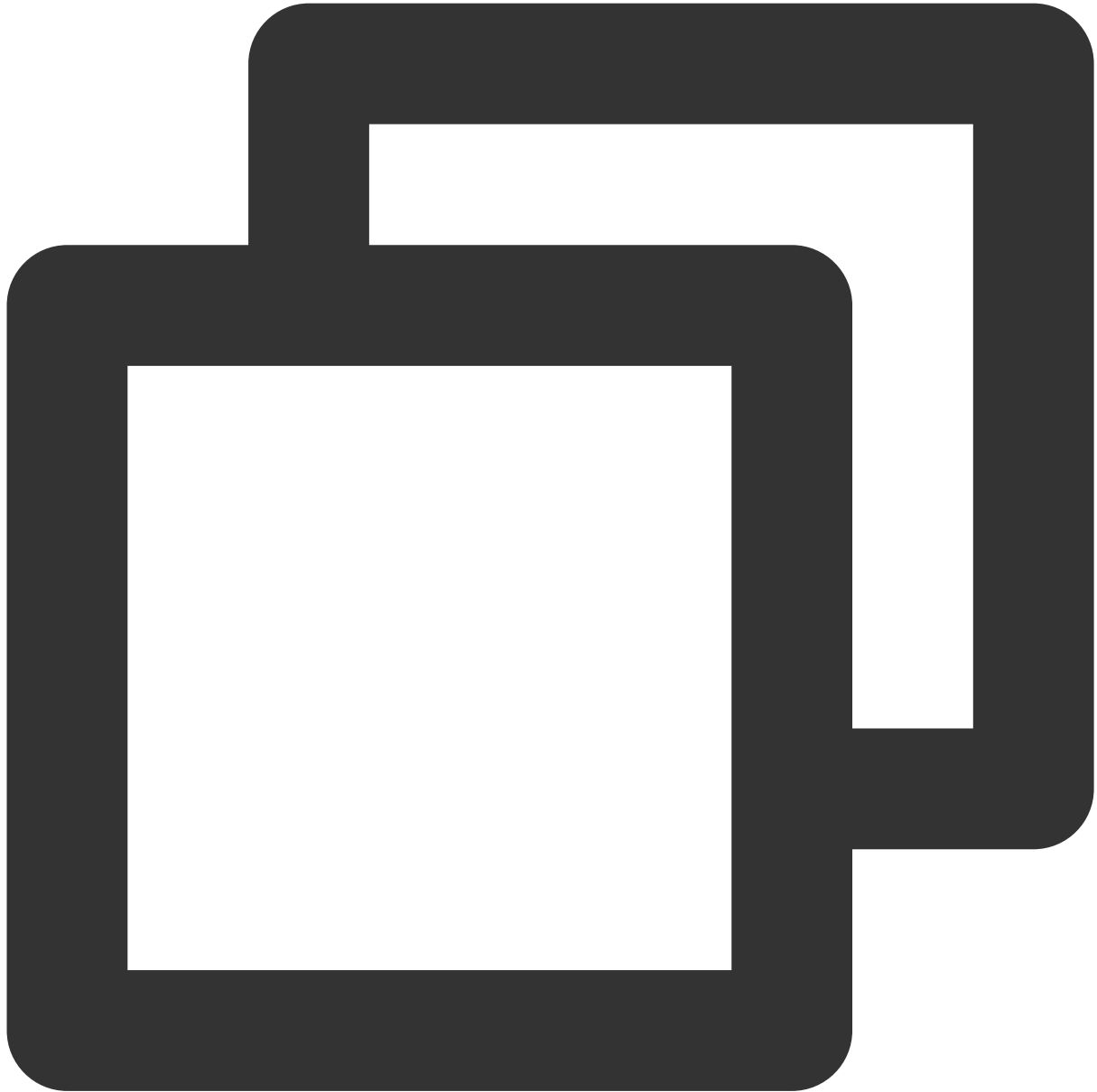


```
// Construct a producer
Producer<byte[]> producer pulsarClient.newProducer()
    .topic(topic)
    .enableBatching(false)
    .create();
// Set the key when sending messages
MessageId msgId = producer.newMessage()
    // Message content
    .value(value.getBytes(StandardCharsets.UTF_8))
    // Set the key here. Messages with the same key will only be distributed to the sa
    .key("youKey1")
```



```
.send();
```

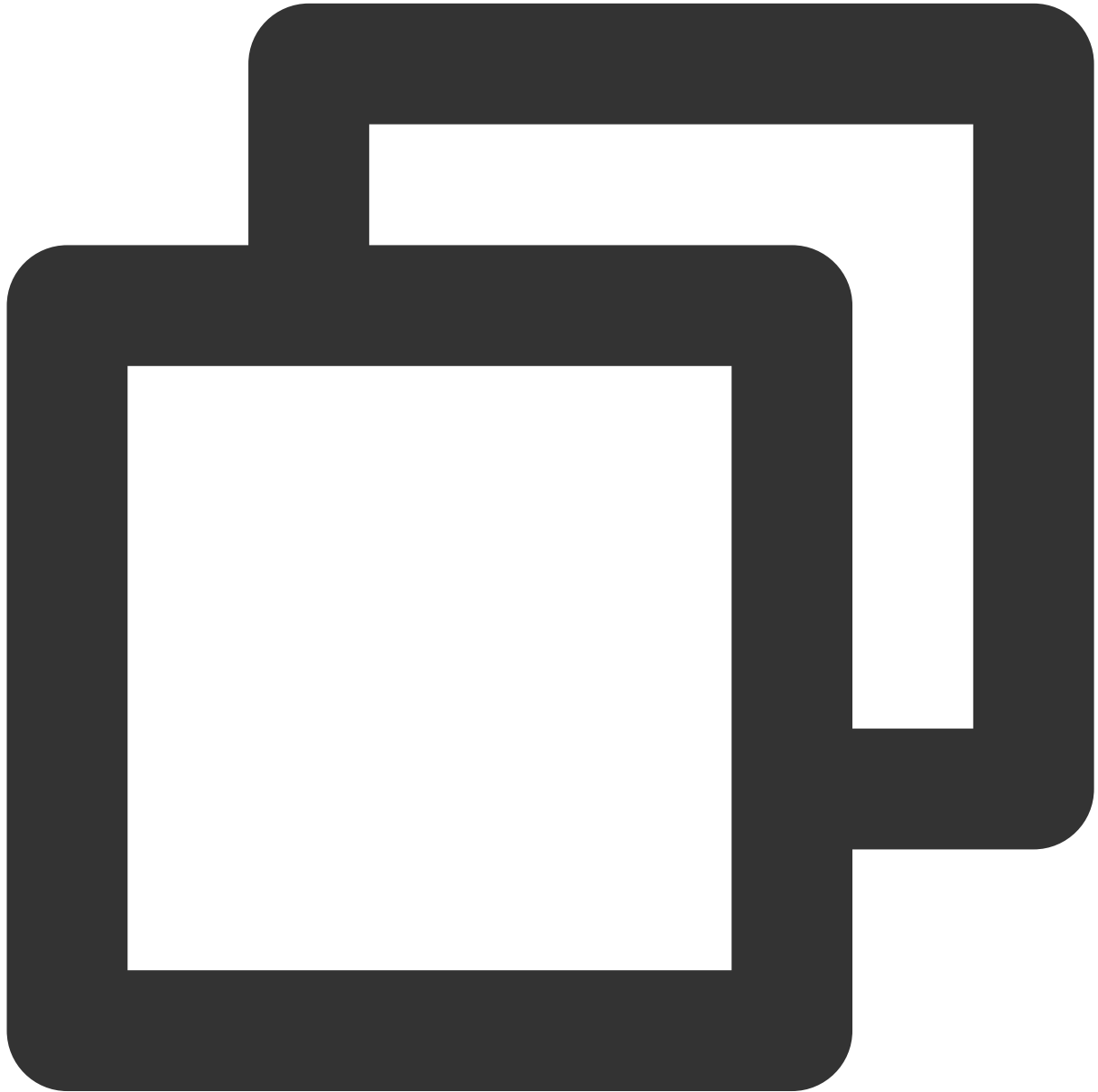
2. Use the key_based batch type.



```
// Construct a producer
Producer<byte[]> producer = pulsarClient.newProducer()
    .topic(topic)
    .enableBatching(true)
    .batcherBuilder(BatcherBuilder.KEY_BASED)
    .create();
// Set the key when sending messages
MessageId msgId = producer.newMessage()
```

```
// Message content
.value(value.getBytes(StandardCharsets.UTF_8))
// Set the key here. Messages with the same key will only be distributed to the sa
.key("youKey1")
.send();
```

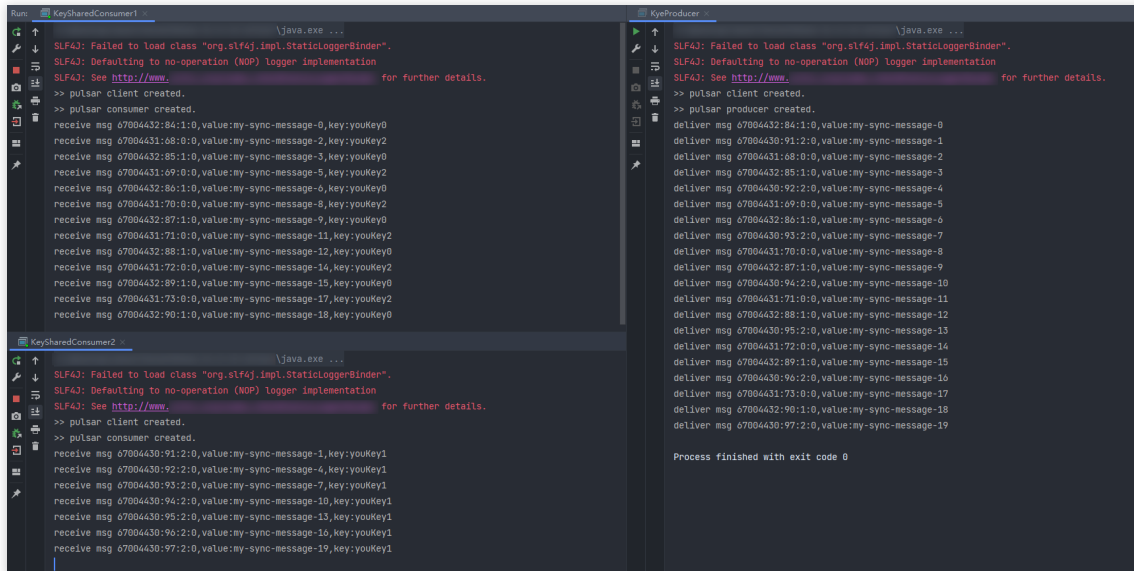
Sample code for the consumer:



```
// Construct a consumer
Consumer<byte[]> consumer = pulsarClient.newConsumer()
    // Complete path of the topic in the format of `persistent://cluster (tenant) I
    .topic("persistent://pulsar-xxx/sdk_java/topic1")
```

```
// You need to create a subscription on the topic details page in the console a
.subscriptionName("sub_topic1")
// Declare the key_shared mode to be the consumption mode
.subscriptionType(SubscriptionType.Key_Shared)
.subscribe();
```

There can be multiple consumers in the key_shared mode.



The screenshot displays two side-by-side Java IDE windows. The left window, titled 'KeySharedConsumer1', shows logs for a consumer that has received 18 messages with keys 'youKey0' and 'youKey2'. The right window, titled 'KeyProducer', shows logs for a producer that has delivered 19 messages with keys 'youKey0' and 'youKey1'. Both windows show the same initial log messages: 'SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".', 'SLF4J: Defaulting to no-operation (NOP) logger implementation', and 'SLF4J: See http://www. for further details.'.

Sample "multi-partition topic + failover" subscription

Note:

In this mode, each partition will be assigned to only one consumer instance at a time. When there are more consumers than partitions, the excessive consumers cannot consume messages. This problem can be solved by adding more partitions.

Try to ensure an even key distribution when designing keys.

The delayed message is not supported in failover mode.

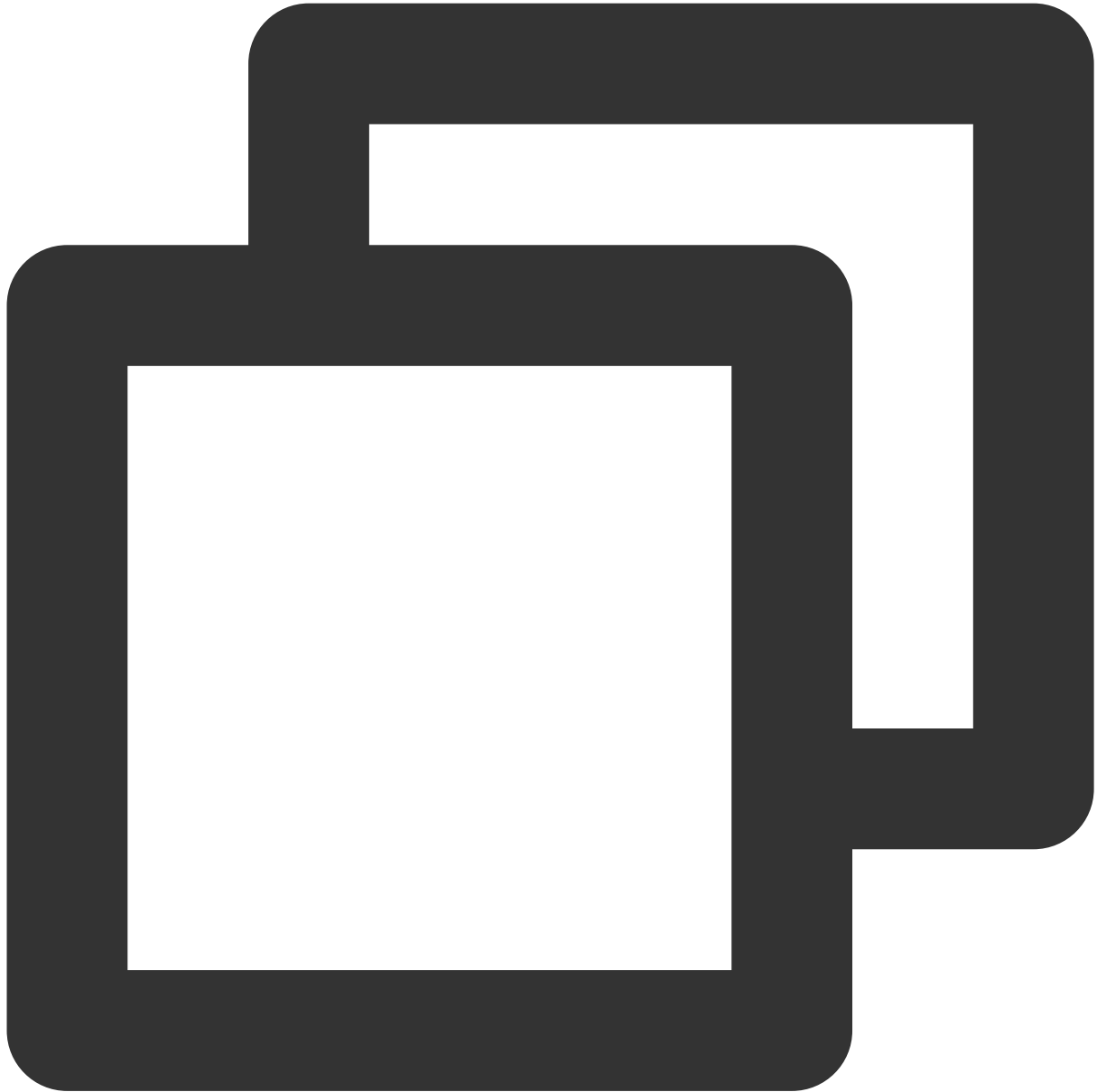
1. Sample code for the producer.



```
// Construct a producer
Producer<byte[]> producer pulsarClient.newProducer()
    .topic(topic)
    .enableBatching(false) // Disable the batch feature
    .create();
// Set the key when sending messages
MessageId msgId = producer.newMessage()
    // Message content
    .value(value.getBytes(StandardCharsets.UTF_8))
    // Set the key here. Messages with the same key will be sent to the same partition
    .key("youKey1")
```

```
.send();
```

2. Sample code for the consumer.

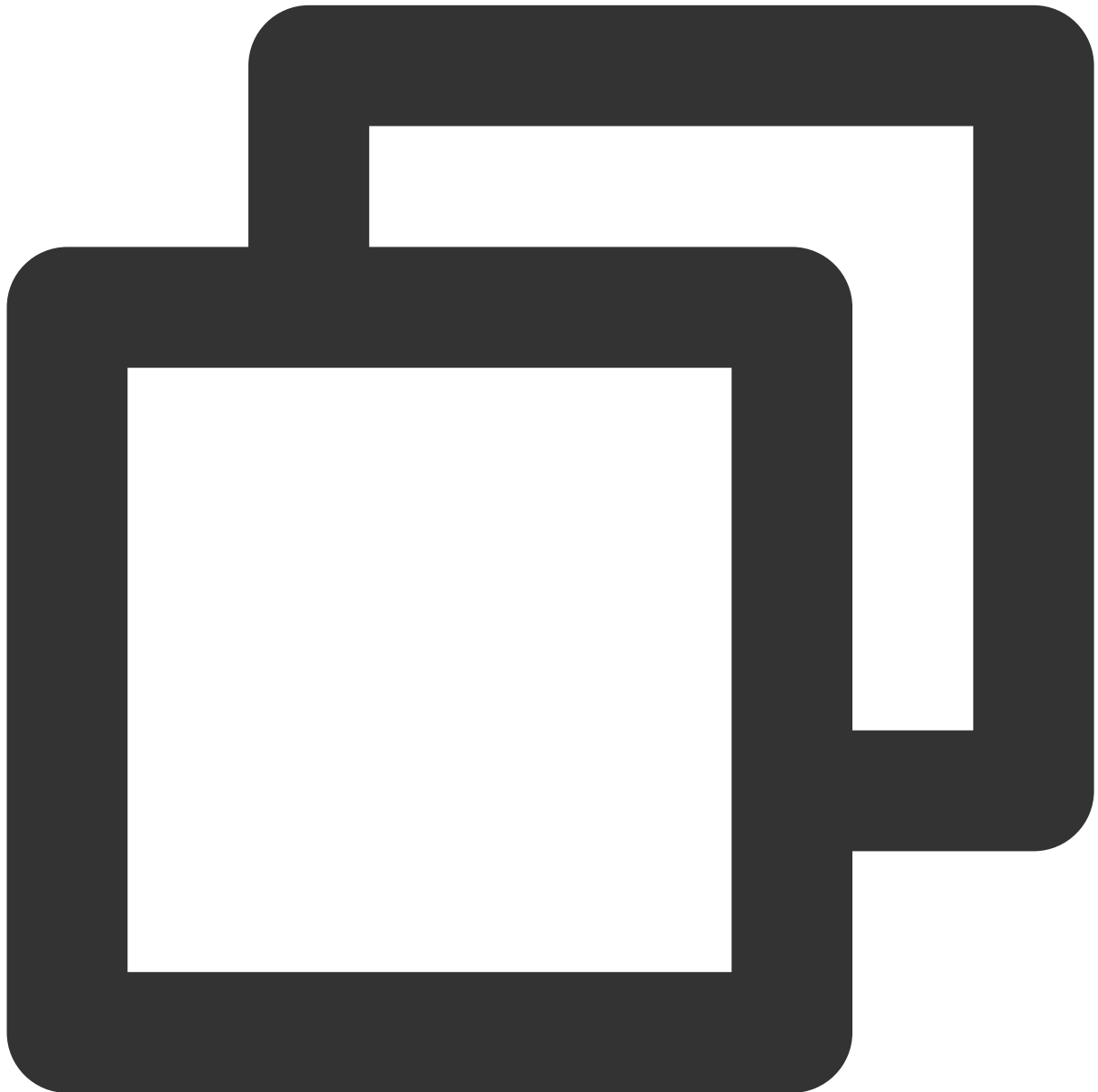


```
// Construct a consumer
Consumer<byte[]> consumer = pulsarClient.newConsumer()
    // Complete path of the topic in the format of `persistent://cluster (tenant) I
    .topic("persistent://pulsar-xxx/sdk_java/topic1")
    // You need to create a subscription on the topic details page in the console a
    .subscriptionName("sub_topic1")
    // Declare the failover mode to be the consumption mode
    .subscriptionType(SubscriptionType.Failover)
```

```
.subscribe();
```

Enabling sequence guarantee

TDMQ for Apache Pulsar 2.9.2 clusters support the sequential message delivery by key. To enable the sequence guarantee feature, you need to specify `keySharedPolicy` when creating the consumer instance.



```
// Construct a consumer
Consumer<byte[]> consumer = pulsarClient.newConsumer()
    // Complete path of the topic in the format of `persistent://cluster (tenant) I
    .topic("persistent://pulsar-xxx/sdk_java/topic1")
```

```
// You need to create a subscription on the topic details page in the console a
.subscriptionName("sub_topic1")
// Declare the key_shared mode to be the consumption mode
.subscriptionType(SubscriptionType.Key_Shared)
    // Set to require sequence
    .keySharedPolicy(KeySharedPolicy.autoSplitHashRange().setAllowOutOfOrderDel
.subscription();
```

Note:

The sequence guarantee feature is not supported for clusters on v2.7.2, which may lead to message push congestion and subsequent consumption failure.

If the sequence guarantee feature is enabled, the consumption may slow down and messages may be heaped after the consumer is restarted. This is because the feature requires the restarted consumer to consume messages sequentially, starting with earlier received messages (with all the consumption acknowledged) and moving on to any later received messages.

Scheduled Message and Delayed Message

Last updated : 2024-06-28 11:29:49

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, scheduled message can be regarded as a special type of delayed message, which is essentially the same thing.

Use Cases

There is no difference between implementing delay from your business code or a third-party component if your system uses a monolithic design. However, if your system has a large distributed architecture with dozens or even hundreds of microservices, implementing the delay logic through the application may result in a variety of issues, and if a node running the delay program fails, the entire delay logic will be affected.

Given this, it will be a good option to deliver delayed messages to a message queue based on their attributes, as the delay period can be calculated in a unified manner, while the retry and dead letter mechanisms ensure that messages will not get lost.

Directions

The TDMQ for Apache Pulsar SDK provides dedicated APIs to implement scheduled and delayed messages.

For a scheduled message, you need to specify a moment to send it.

For a delayed message, you need to specify a period of time as the delay.

Scheduled Message

Scheduled messages are implemented by the producer's `deliverAt()` method. Below is the sample code:



```
String value = "message content";
try {
    // You need to convert the explicit time to a timestamp first
    long timeStamp = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").parse("2020-11-1
    // Call the producer's `deliverAt` method to implement the scheduled message
    MessageId msgId = producer.newMessage()
        .value(value.getBytes())
        .deliverAt(timeStamp)
        .send();
} catch (JsonParseException e) {
    // TODO adds the method for handling the timestamp parsing failure
```

```
e.printStackTrace();  
}
```

Note:

The time range for scheduled messages is any time point within 864,000 seconds (10 days) starting from the current time; for example, starting from 12:00 on October 1, it can be set to 12:00 on October 11 at the most.

Scheduled messages cannot be sent in batch mode, so you need to set the `enableBatch` parameter to `false` when creating the producer.

Scheduled messages can be consumed only in the shared mode; otherwise, scheduling will not work (even in the key-shared mode).

Delayed message

Delayed messages are implemented by the producer's `deliverAfter()` method. Below is the sample code:



```
String value = "message content";

// You need to specify the length of the delay
long delayTime = 10L;
// Call the producer's `deliverAfter` method to implement the delayed message
MessageId msgId = producer.newMessage()
    .value(value.getBytes())
    .deliverAfter(delayTime, TimeUnit.SECONDS) // You can select a unit freely
    .send();
```

Note:

The time range for delayed messages is 0–864,000 seconds (10 days); for example, starting from 12:00 PM on October 1, it can be set to 12:00 PM on October 11 at most.

Delayed messages cannot be sent in batch mode, so you need to set the `enableBatch` parameter to `false` when creating the producer.

Delayed messages can be consumed only in the shared mode; otherwise, delaying will not work (even in the key-shared mode).

Use Instructions and Limits

When you use scheduled or delayed messages, we recommend you send them to a topic different from that for general messages, so you can manage them easily later while improving system stability.

When using scheduled and delayed messages, make sure that the client clock stays in sync with the server; otherwise, there will be a time difference.

There is a precision deviation of about 1 second for scheduled and delayed messages.

Scheduled and delayed messages do not support batch mode (i.e., batch sending), as that mode will cause message retention. To be on the safe side, you need to set the `enableBatch` parameter to `false` when creating the producer.

Scheduled and delayed messages can be consumed only in the shared mode; otherwise, scheduling and delaying will not work (even in the key-shared mode).

The maximum time range for scheduled and delayed messages are both 10 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.

After the scheduled time point is set, the maximum message retention period (i.e., TTL) will still be calculated starting from the message sending time point; for example, if the message is scheduled to be sent in two days, but the TTL of the message is set to one day, then the message will be deleted after one day. In this case, you should make sure that the TTL is longer than the scheduled time (i.e., two days); otherwise, the message will be deleted after the TTL elapses. This is also the case for delayed messages.

In general topics, you can send scheduled and delayed messages through API as instructed in [Directions](#) and receive them.

Message Tag Filtering

Last updated : 2024-07-19 14:17:20

This document describes how to use message tag filtering in TDMQ for Apache Pulsar.

Feature Description

A message tag is used to categorize messages under a topic. When a producer in TDMQ for Apache Pulsar sends messages with specified tags, the consumer needs to subscribe to those messages by tag.

If a consumer configures no tags when subscribing to a topic, all messages in the topic will be delivered to the consumer for consumption.

Note:

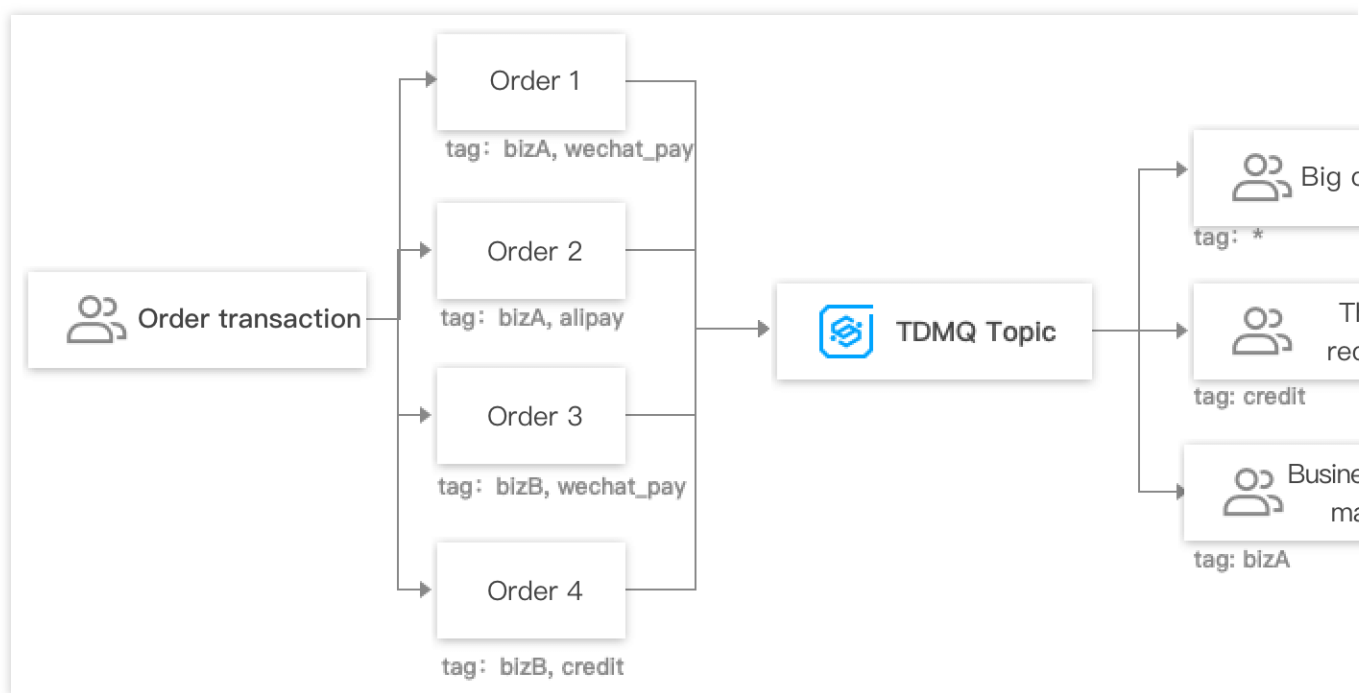
In a subscription:

1. A single consumer can use multiple tags, and the relationship between multiple tags is OR.
2. Multiple consumers need to use the same tag. If different consumers use different tags in a subscription, there will be an issue of filtering rule overlap, affecting business logic.

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 placements, payments, and deliveries, 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 for Apache Pulsar supports filtering on the broker. You can set one or more tags during message production and subscribe to specified tags during consumption.

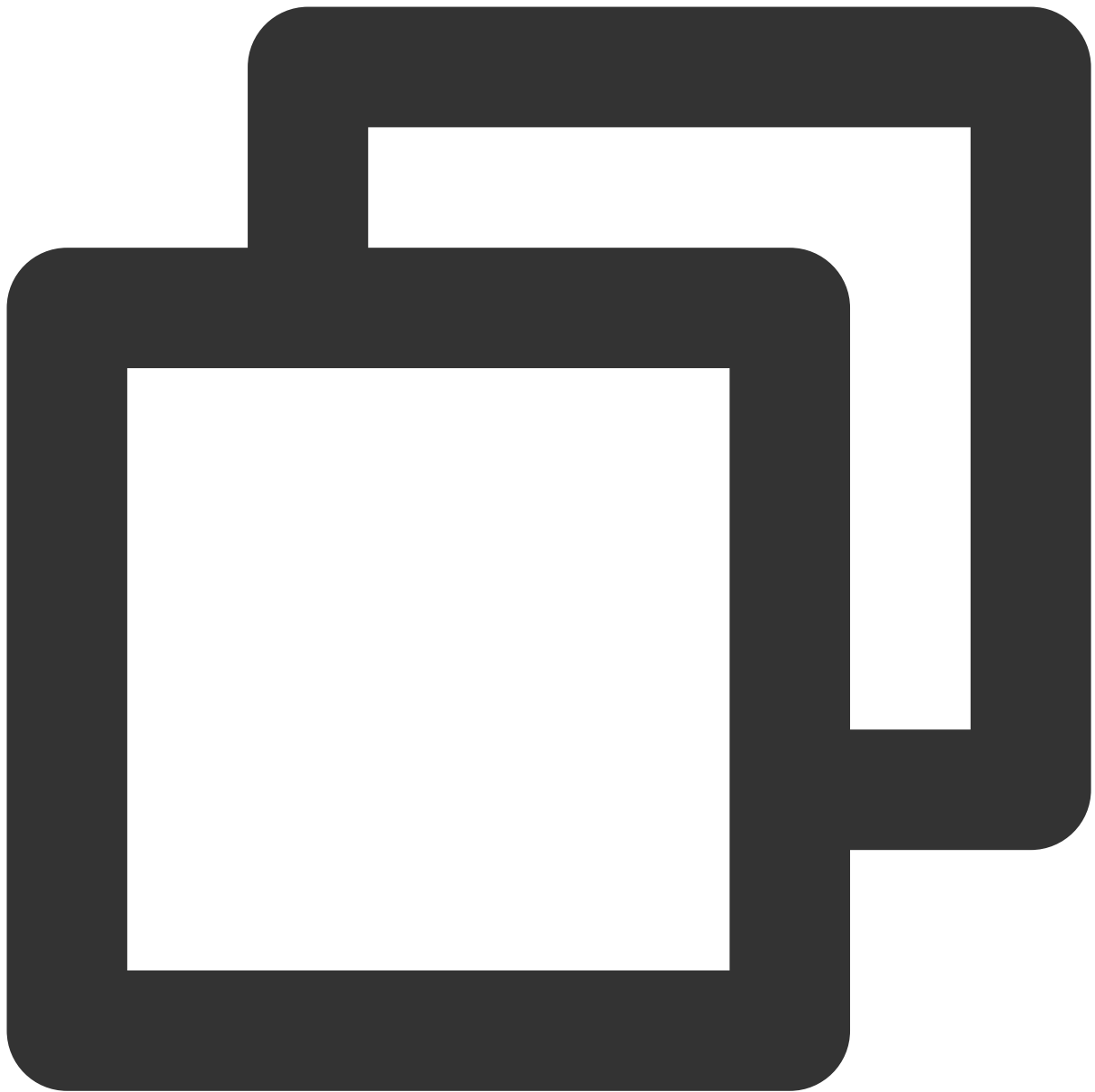


Use Instructions

Tagged messages are passed in through `Properties` and can be obtained as follows:

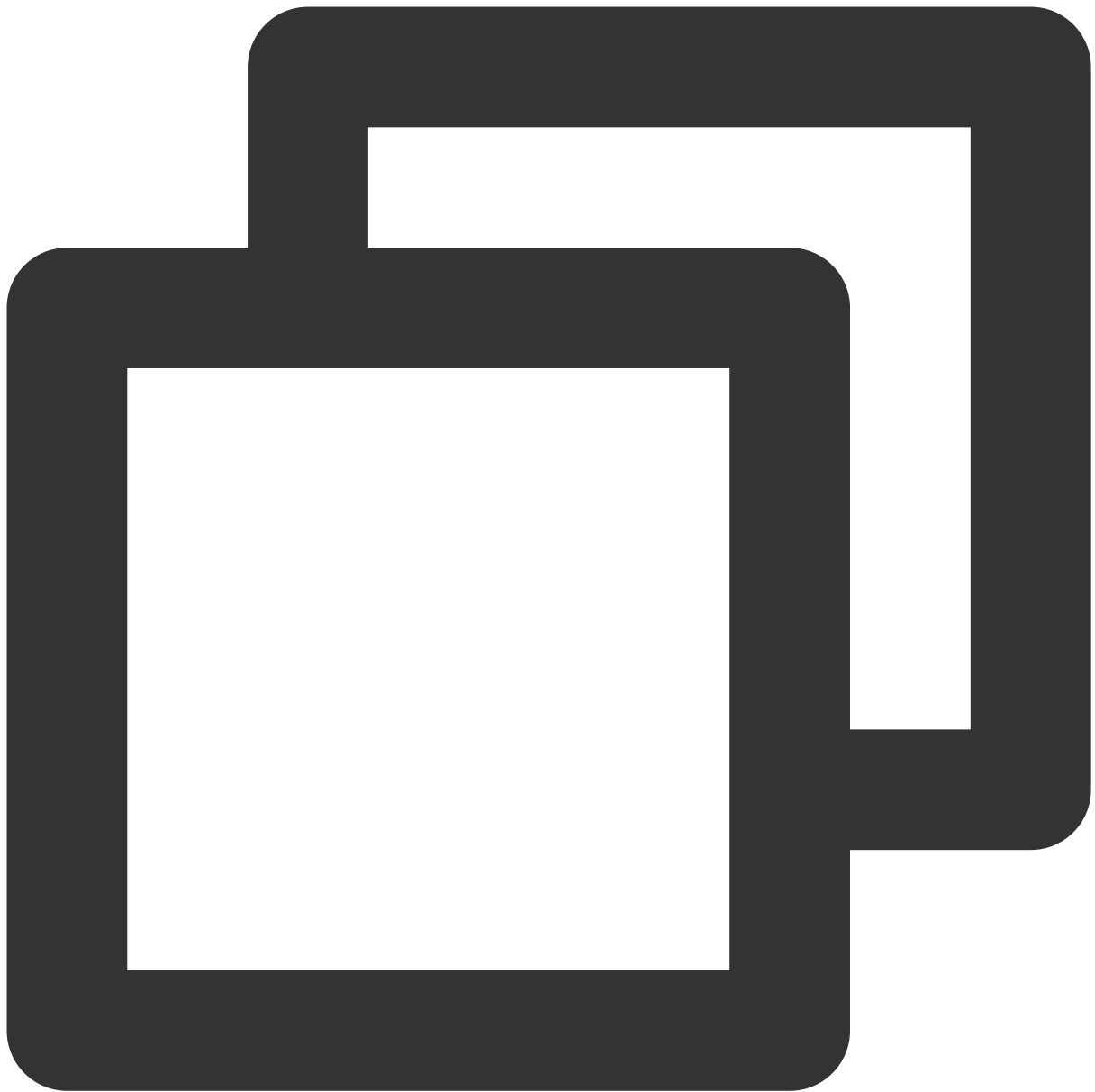
Java

Go



```
<dependency>
  <groupId>org.apache.pulsar</groupId>
  <artifactId>pulsar-client</artifactId>
  <version>2.10.1</version> <!-- Recommended -->
</dependency>
```

v0.8.0 or later is recommended.



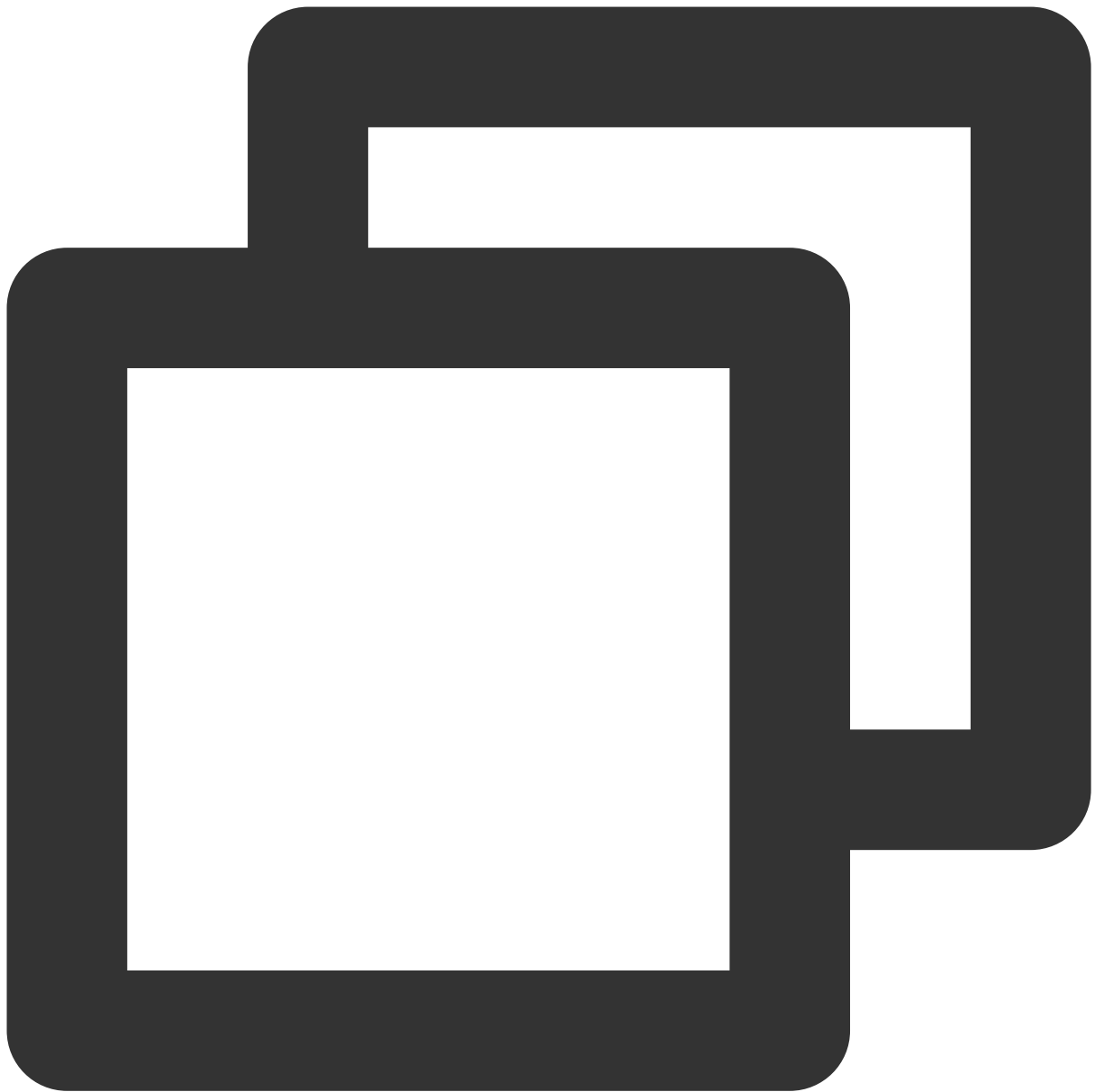
```
go get -u github.com/apache/pulsar-client-go@master
```

Use Limits of tagged messages

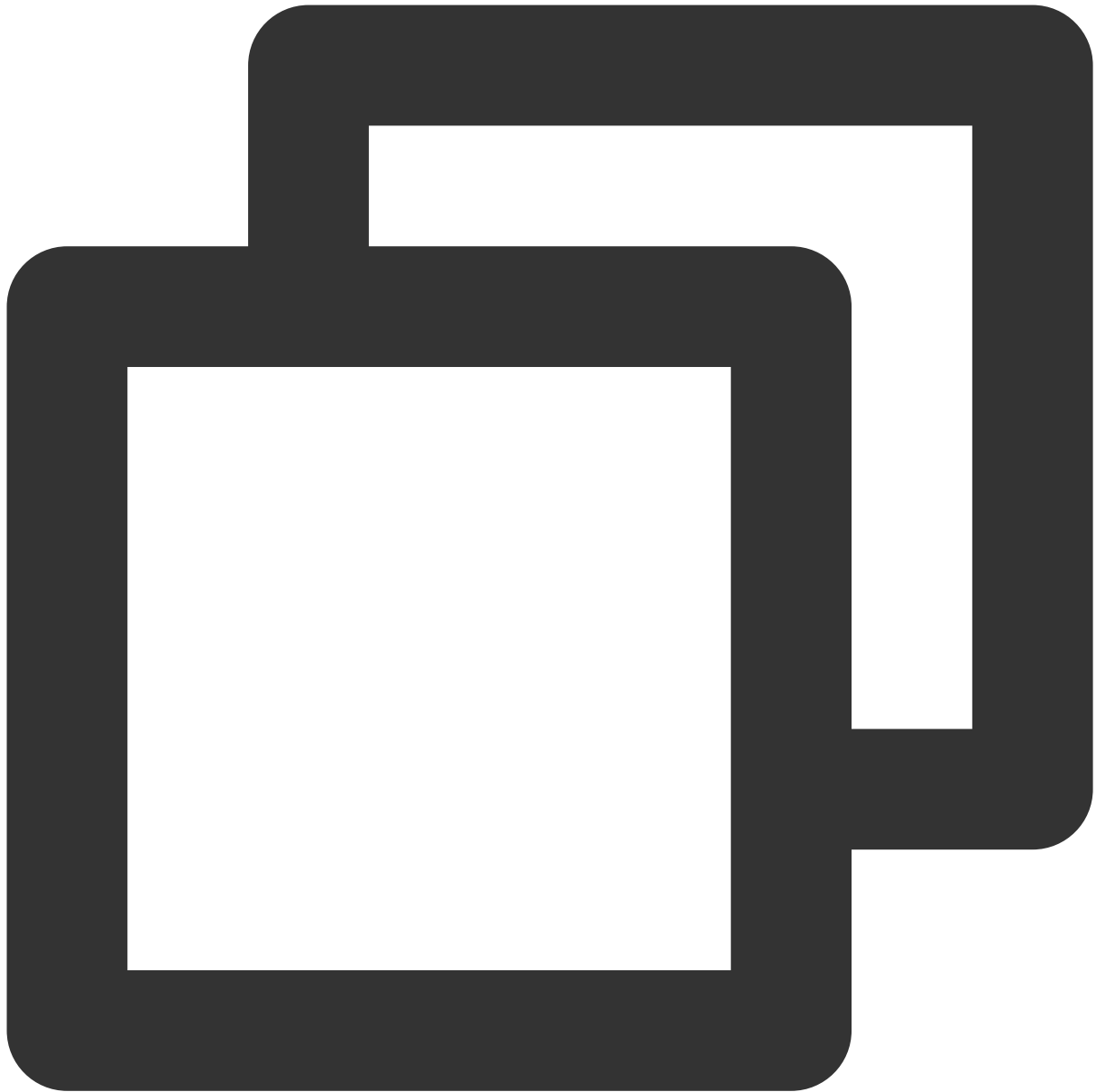
Tagged messages don't support batch operations. The batch operation feature is enabled by default. To use tagged messages, you need to disable it in the producer as follows:

Java

Go



```
// Construct a producer
Producer<byte[]> producer = pulsarClient.newProducer()
    // Disable batch operation
    .enableBatching(false)
    // Complete path of the topic in the format of `persistent://cluster (ten
    .topic("persistent://pulsar-xxx/sdk_java/topic2").create();
```



```
producer, err := client.CreateProducer(pulsar.ProducerOptions{
    DisableBatching: true, // Disable batch operation
})
```

Tagged message filtering only takes effect for messages with tags; that is, messages without tags won't be filtered and will be pushed to all subscribers instead.

To enable tagged message, set the `Properties` field in `ProducerMessage` when sending messages and set the `SubscriptionProperties` field in `ConsumerOptions` when creating consumers.

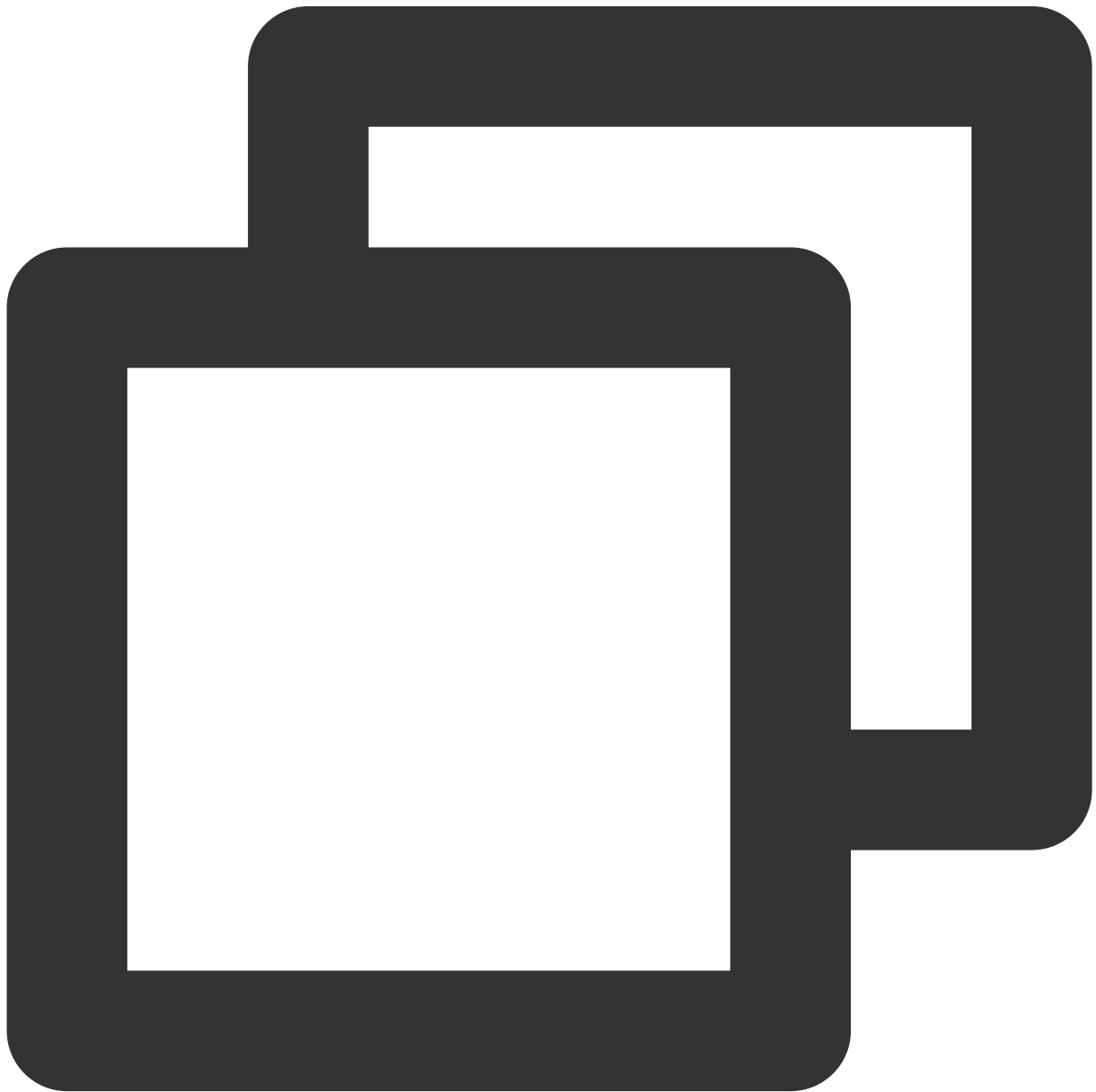
When you set the `Properties` field in `ProducerMessage`, the `key` is the tag name, and the `value` is fixed to `TAGS`.

When you set the `SubscriptionProperties` field in `ConsumerOptions`, the `key` is the tag name to be subscribed to, and the `value` is the tag version, which is reserved for feature extension in the future and has no meaning currently. You can configure as follows:

Specify one tag

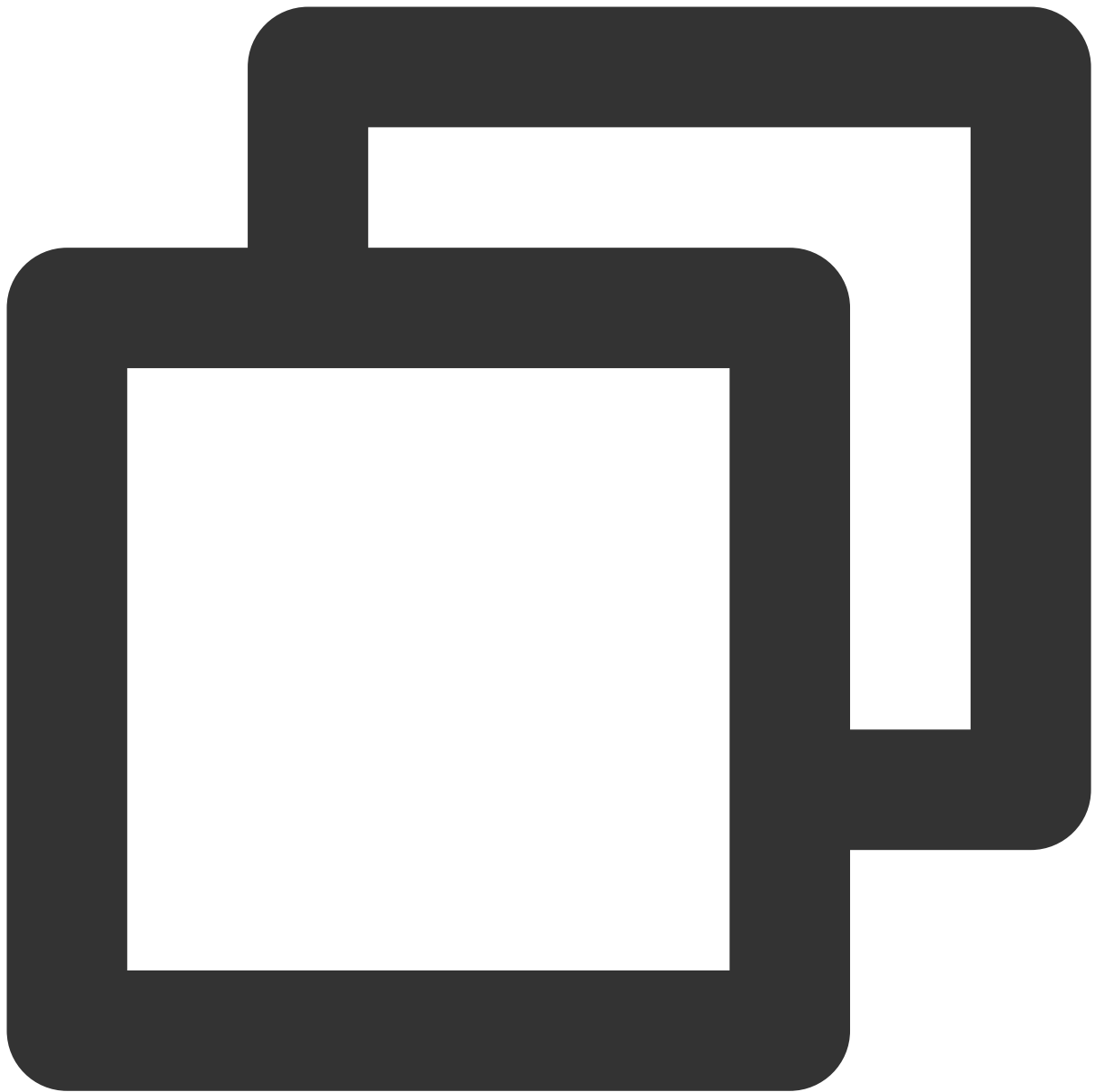
Java

Go



```
// Send the message
MessageId msgId = producer.newMessage()
    .property("tag1", "TAGS")
    .value(value.getBytes(StandardCharsets.UTF_8))
    .send();

// Subscription parameters, which can be used to set subscription tags
HashMap<String, String> subProperties = new HashMap<>();
subProperties.put("tag1","1");
// Construct a consumer
Consumer<byte[]> consumer = pulsarClient.newConsumer()
    // Complete path of the topic in the format of `persistent://cluster (tenant)
    .topic("persistent://pulsar-xxxx/sdk_java/topic2")
    // You need to create a subscription on the topic details page in the console
    .subscriptionName("topic_sub1")
    // Declare the shared mode as the consumption mode
    .subscriptionType(SubscriptionType.Shared)
    // Subscription parameters for tag subscription
    .subscriptionProperties(subProperties)
    // Configure consumption starting at the earliest offset; otherwise, historic
    .subscriptionInitialPosition(SubscriptionInitialPosition.Earliest).subscribe(
```



```
// Send the message
if msgId, err := producer.Send(ctx, &pulsar.ProducerMessage{
    Payload: []byte(fmt.Sprintf("hello-%d", i)),
    Properties: map[string]string{
        "tag1": "TAGS",
    },
}); err != nil {
    log.Fatal(err)
}

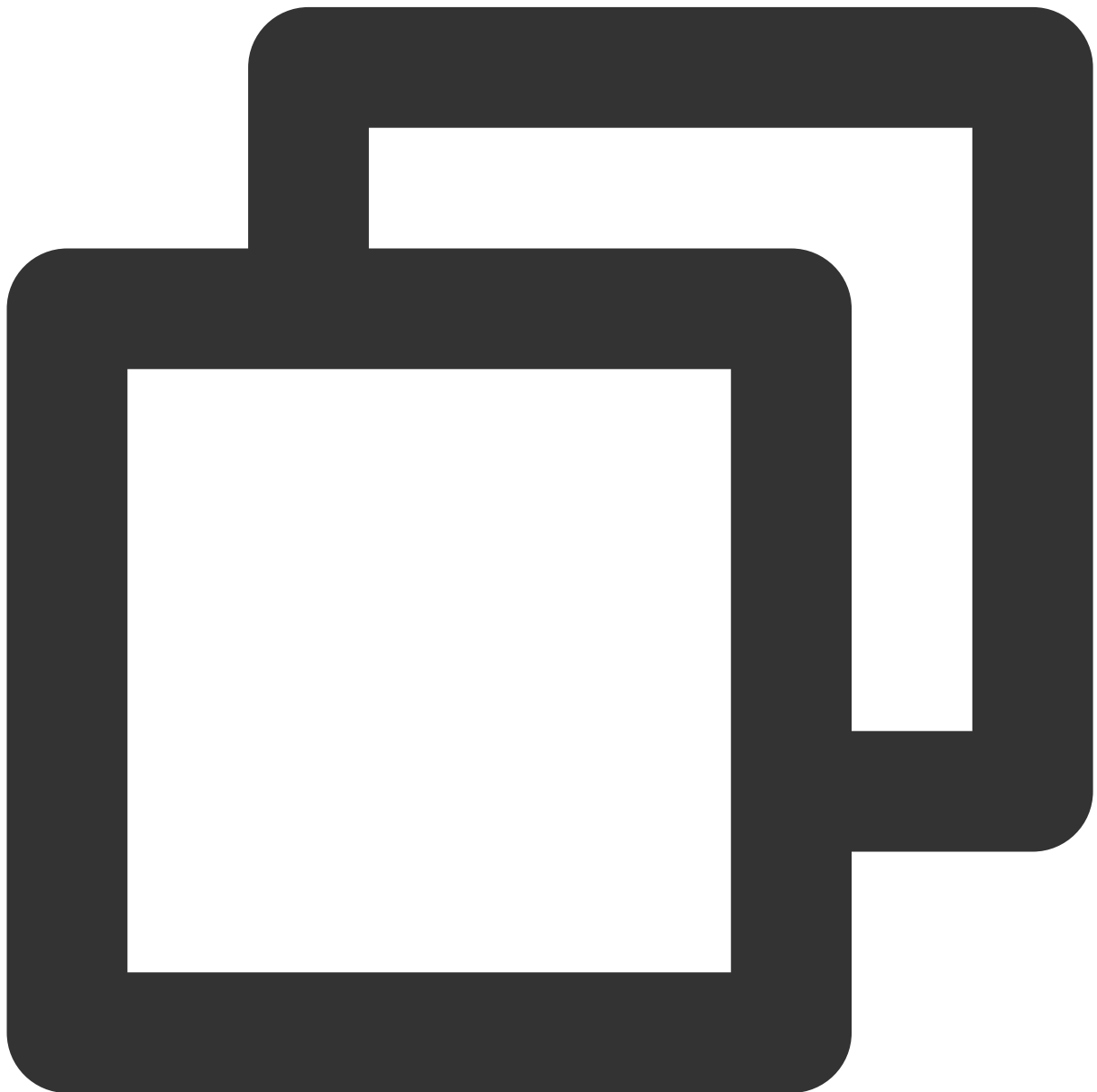
// Create a consumer
```

```
consumer, err := client.Subscribe(pulsar.ConsumerOptions{
    Topic:          "topic-1",
    SubscriptionName: "my-sub",
    SubscriptionProperties: map[string]string{"tag1": "1"},
})
```

Specify multiple tags

Java

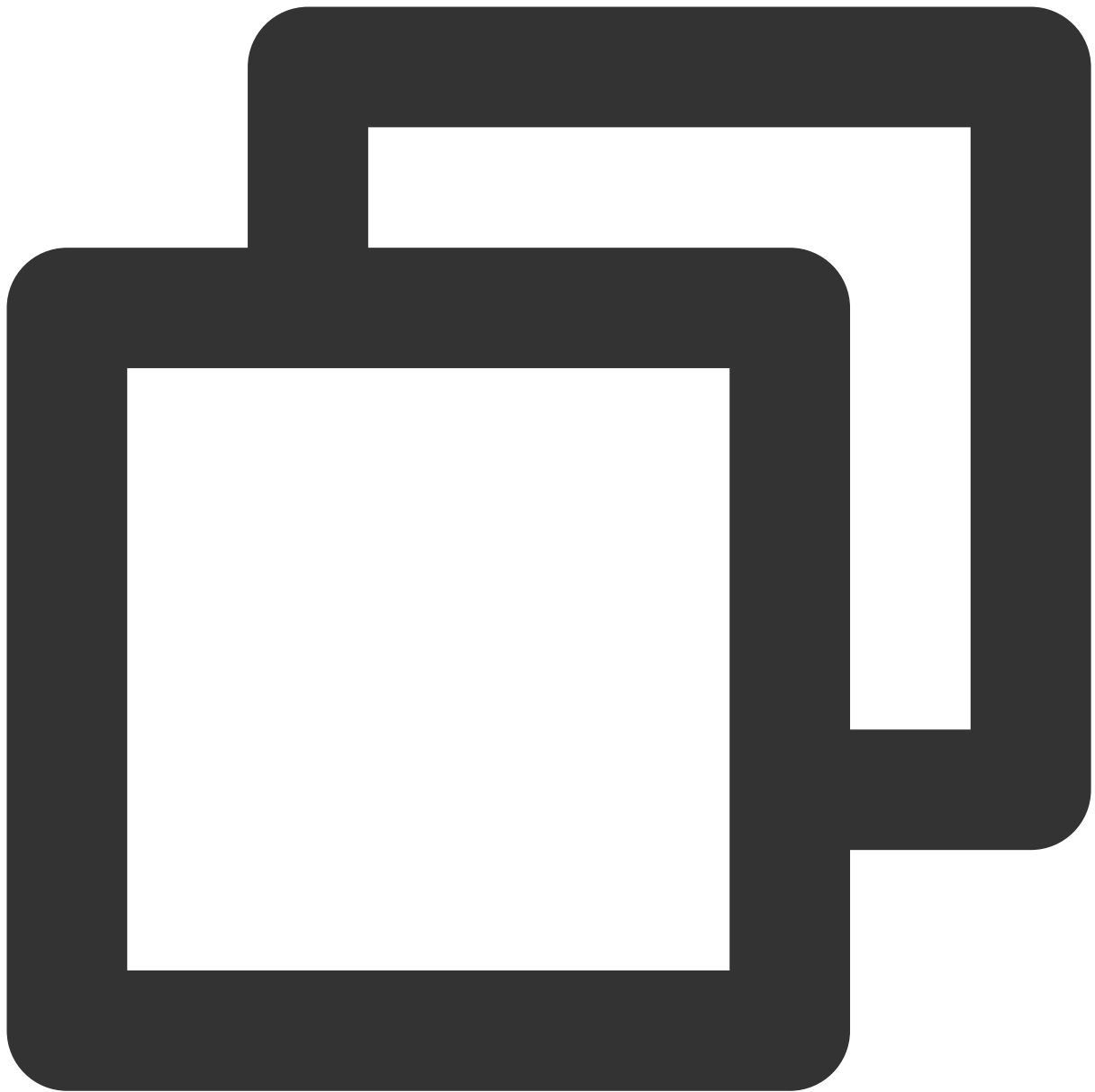
Go



```
// Send the message
```

```
MessageId msgId = producer.newMessage()
    .property("tag1", "TAGS")
    .property("tag2", "TAGS")
    .value(value.getBytes(StandardCharsets.UTF_8))
    .send();

// Subscription parameters, which can be used to set subscription tags
HashMap<String, String> subProperties = new HashMap<>();
subProperties.put("tag1", "1");
subProperties.put("tag2", "1");
// Construct a consumer
Consumer<byte[]> consumer = pulsarClient.newConsumer()
    // Complete path of the topic in the format of `persistent://cluster (tenant)
    .topic("persistent://pulsar-xxxx/sdk_java/topic2")
    // You need to create a subscription on the topic details page in the console
    .subscriptionName("topic_sub1")
    // Declare the shared mode as the consumption mode
    .subscriptionType(SubscriptionType.Shared)
    // Subscription parameters for tag subscription
    .subscriptionProperties(subProperties)
    // Configure consumption starting at the earliest offset; otherwise, historical
    .subscriptionInitialPosition(SubscriptionInitialPosition.Earliest).subscribe()
```



```
// Create a producer
if msgId, err := producer.Send(ctx, &pulsar.ProducerMessage{
    Payload: []byte(fmt.Sprintf("hello-%d", i)),
    Properties: map[string]string{
        "tag1": "TAGS",
        "tag2": "TAGS",
    },
}); err != nil {
    log.Fatal(err)
}
```

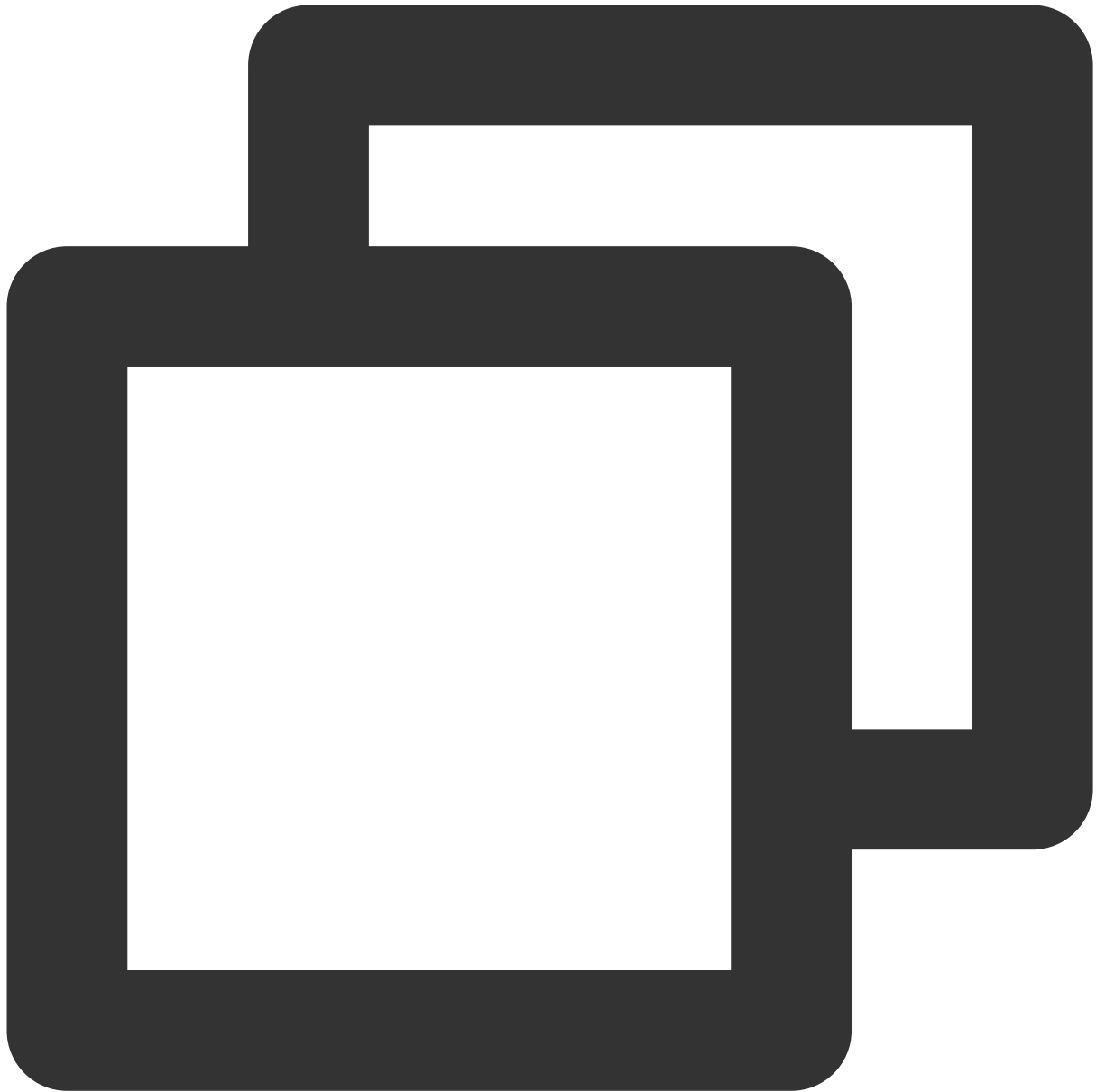


```
// Create a consumer
consumer, err := client.Subscribe(pulsar.ConsumerOptions{
    Topic:          "topic-1",
    SubscriptionName: "my-sub",
    SubscriptionProperties: map[string]string{
        "tag1": "1",
        "tag2": "1",
    },
})
```

Mix tags and properties

Java

Go



```
// Send the message
MessageId msgId = producer.newMessage()
    .property("tag1", "TAGS")
    .property("tag2", "TAGS")
    .property("xxx", "yyy")
    .value(value.getBytes(StandardCharsets.UTF_8))
    .send();

// Subscription parameters, which can be used to set subscription tags
HashMap<String, String> subProperties = new HashMap<>();
subProperties.put("tag1", "1");
```

```
subProperties.put("tag2", "1");
// Construct a consumer
Consumer<byte[]> consumer = pulsarClient.newConsumer()
    // Complete path of the topic in the format of `persistent://cluster (tenant)
    .topic("persistent://pulsar-xxxx/sdk_java/topic2")
    // You need to create a subscription on the topic details page in the console
    .subscriptionName("topic_sub1")
    // Declare the shared mode as the consumption mode
    .subscriptionType(SubscriptionType.Shared)
    // Subscription parameters for tag subscription
    .subscriptionProperties(subProperties)
    // Configure consumption starting at the earliest offset; otherwise, historical
    .subscriptionInitialPosition(SubscriptionInitialPosition.Earliest).subscribe()
```



```
// Create a producer
if msgId, err := producer.Send(ctx, &pulsar.ProducerMessage{
    Payload: []byte(fmt.Sprintf("hello-%d", i)),
    Properties: map[string]string{
        "tag1": "TAGS",
        "tag2": "TAGS",
        "xxx": "yyy",
    },
}); err != nil {
    log.Fatal(err)
}
```

```
// Create a consumer
consumer, err := client.Subscribe(pulsar.ConsumerOptions{
    Topic:          "topic-1",
    SubscriptionName: "my-sub",
    SubscriptionProperties: map[string]string{
        "tag1": "1",
        "tag2": "1",
    },
})
```

Note:

Once the `SubscriptionProperties` field is set in the consumer, the tags processed by the subscription cannot be modified. To modify tags, unsubscribe the current subscription first and create a new subscription.

Message Retry and Dead Letter Mechanisms

Last updated : 2024-07-19 14:13:13

In messaging scenarios, it is common to encounter message sending failures and message heap exceeding expectations, leading to messages not being consumed normally. To deal with these scenarios, TDMQ for Pulsar provides message retry and dead letter mechanisms.

Automatic Retry

An automatic retry topic is designed to ensure that messages are consumed normally. If no normal response is received after some messages are consumed by the consumer for the first time, messages will enter the retry topic. And after a specified number of failed retries, they will stop retrying and will be delivered to the dead letter topic.

Relevant concepts

Retry Topic : Each retry topic corresponds to one subscription name (the unique identifier of a subscriber group) and exists in TDMQ for Pulsar in the form of a topic. When you create a subscription in the console and enable `automatically create retry & dead-letter queues` , a retry topic will be automatically created, which can independently implement the message retry mechanism.

This topic is named as follows:

For clusters on v2.9.2: [Topic name]-[subscription name]-RETRY

For clusters on v2.7.2: [Subscription name]-retry

For clusters on v2.6.1: [Subscription name]-retry

How It Works

When consumption fails, calling `consumer.resumeLater` API will cause the client to check the retry count of the message internally. If the specified maximum retry count is reached, the message will be delivered to the dead letter queue. (Messages in the dead letter queue cannot be consumed automatically. If consumption is needed, users need to create extra consumers.) If the maximum retry count is not reached, the message will be delivered to the retry queue. The retry interval is implemented via the delayed message. The message in the retry queue is essentially a delayed message, and the delay time is specified by users through `resumeLater` API.

Note:

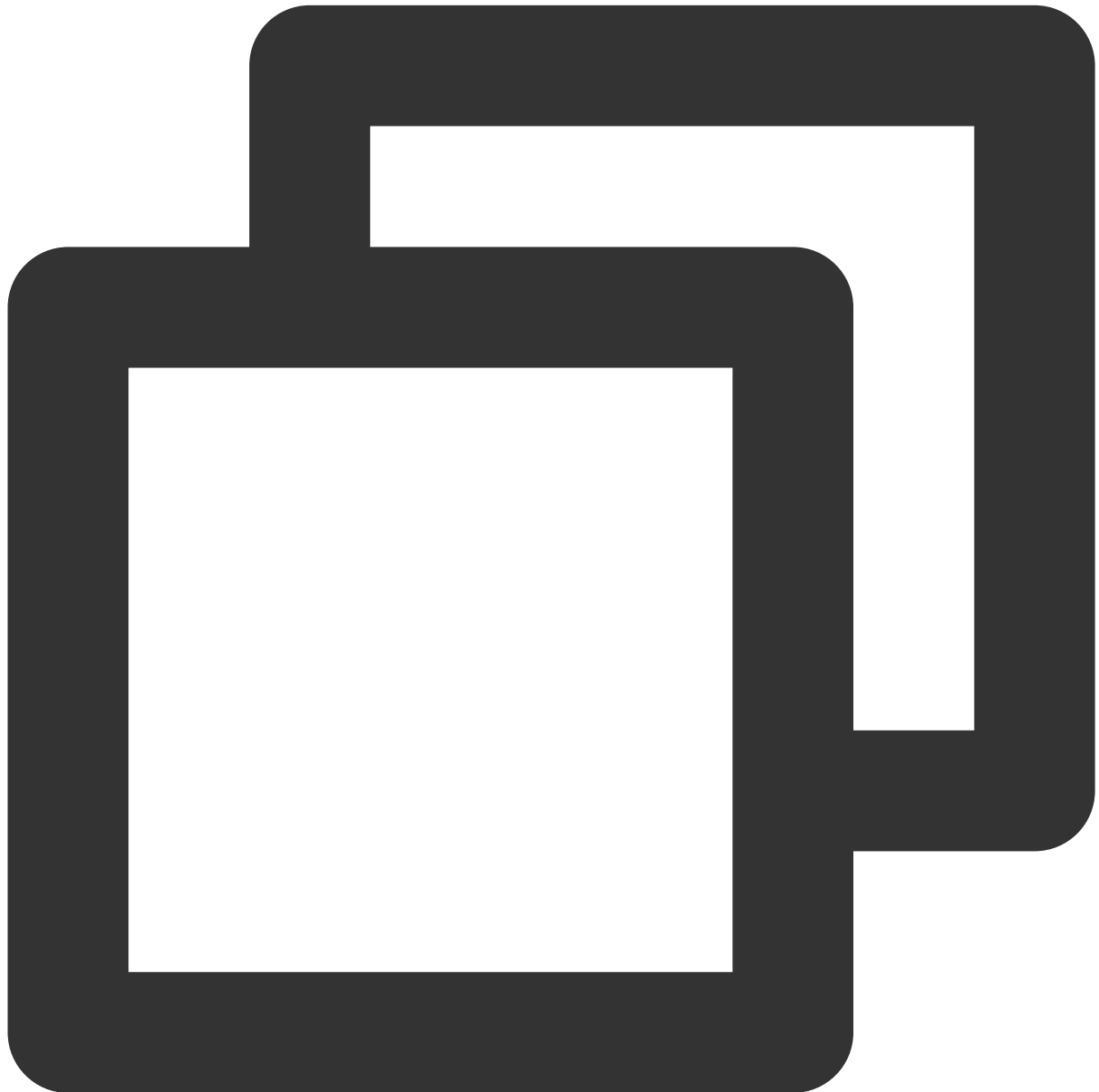
Only the shared mode (including Key sharing) supports the automatic retry and dead letter mechanisms.

If the subscription mode is Exclusive or Failover, the specified retry interval is invalid, and there will be an immediate retry. This is because the retry interval is implemented via the delayed message, which is not supported under Exclusive or Failover mode.

Note that the client version must match the cluster version for the client to accurately recognize the automatically created retry and dead letter queues.

During the use of tokens to access the retry/dead letter queue, roles used by consumers need to be granted permission to produce messages.

Here, a Java client is used as an example. The `sub1` subscription is created in `topic1`, and the client uses the `sub1` subscription name to subscribe to `topic1` and enables `enableRetry` as shown below:



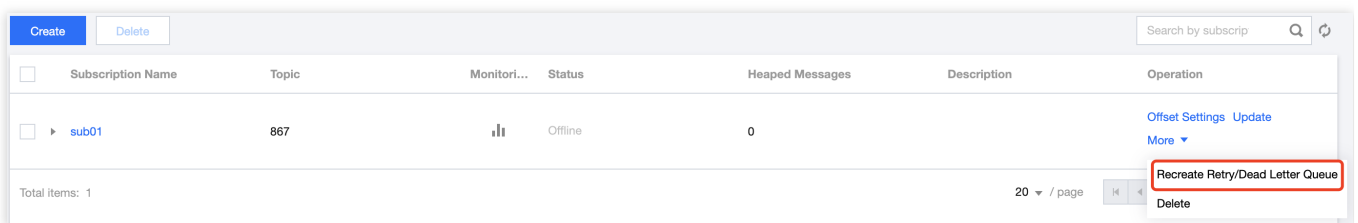
```
Consumer<byte[]> consumer = client.newConsumer()  
    .topic("persistent://1*****30/my-ns/topic1")  
    .subscriptionType(SubscriptionType.Shared) // Only the shared consumption mode s
```

```
.enableRetry(true)
.subscriptionName("sub1")
.subscribe();
```

At this point, the `sub1` subscription to `topic1` forms a delivery model with the retry mechanism, and `sub1` will automatically subscribe to the retry letter topic created automatically during subscription creation (which can be found in the topic list in the console). If no ack is received from the consumer after a message in `topic1` is delivered for the first time, the message will be automatically delivered to the retry letter topic, and as the consumer has subscribed to this topic automatically, the message will subsequently be consumed again according to particular [retry rules](#). If it still fails to be consumed after the specified maximum number of retries, it will be delivered to the dead letter topic and wait for manual processing.

Note:

If the subscription is automatically created by the client, you can click [Topic Management](#) > **More** > **View Subscription** in the console to enter the **Subscription Management** page and manually recreate the retry letter and dead letter topics.



Custom parameter settings

A set of retry and dead letter parameters are configured for TDMQ for Apache Pulsar by default as follows:

Clusters on v2.9.2

Clusters on v2.7.2

Clusters on v2.6.1

Specify the number of retries as 16 (after 16 failed retries, the message will be delivered to the dead letter topic at the 17th time)

Specify the retry letter topic as `[topic name]-[subscription name]-RETRY`

Specify the dead letter topic as `[topic name]-[subscription name]-DLQ`

Specify the number of retries as 16 (after 16 failed retries, the message will be delivered to the dead letter topic at the 17th time)

Specify the retry letter topic as `[subscription name]-RETRY`

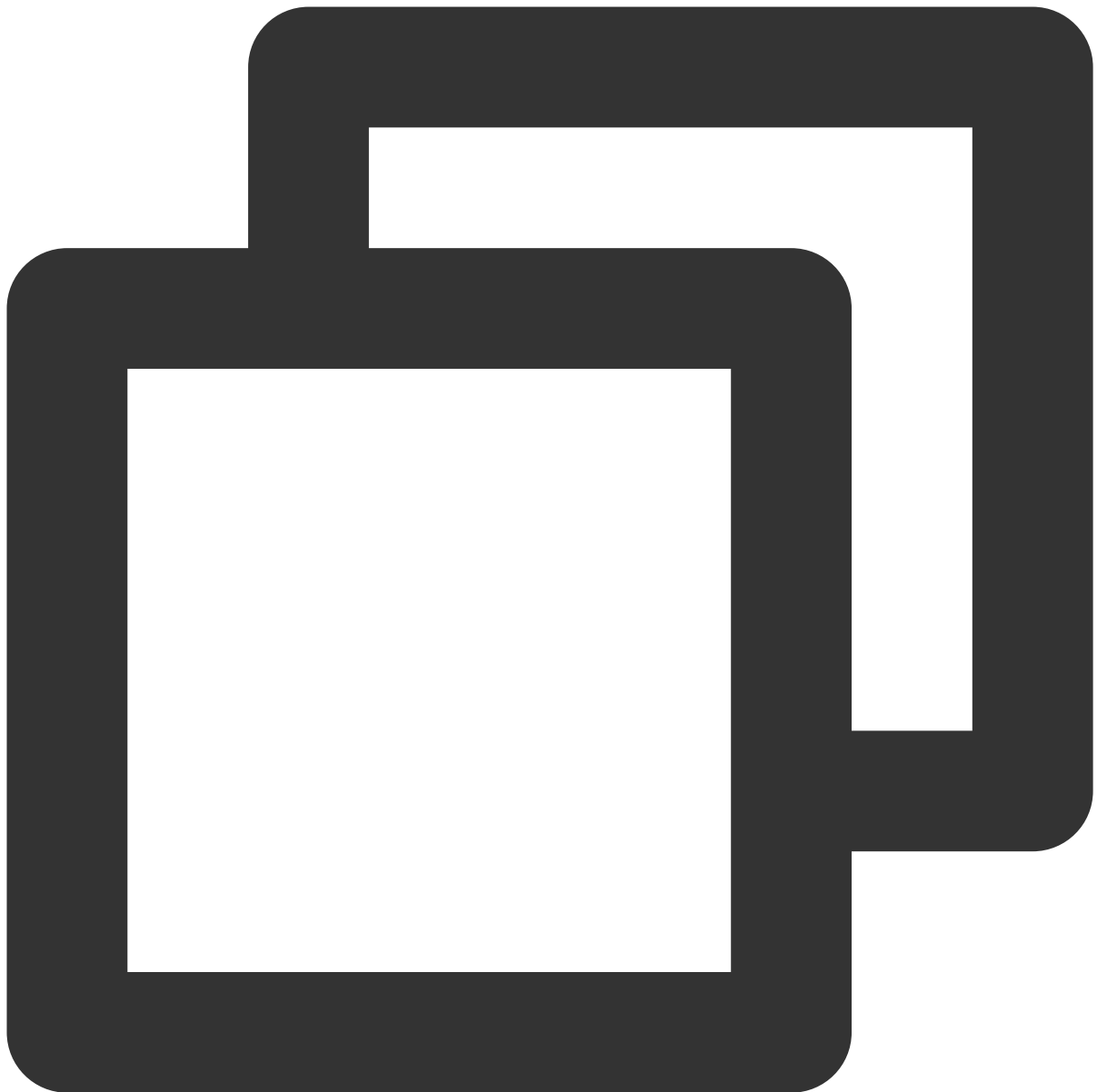
Specify the dead letter topic as `[subscription name]-DLQ`

Specify the number of retries as 16 (after 16 failed retries, the message will be delivered to the dead letter topic at the 17th time)

Specify the retry letter topic as `[subscription name]-retry`

Specify the dead letter topic as `[subscription name]-dlq`

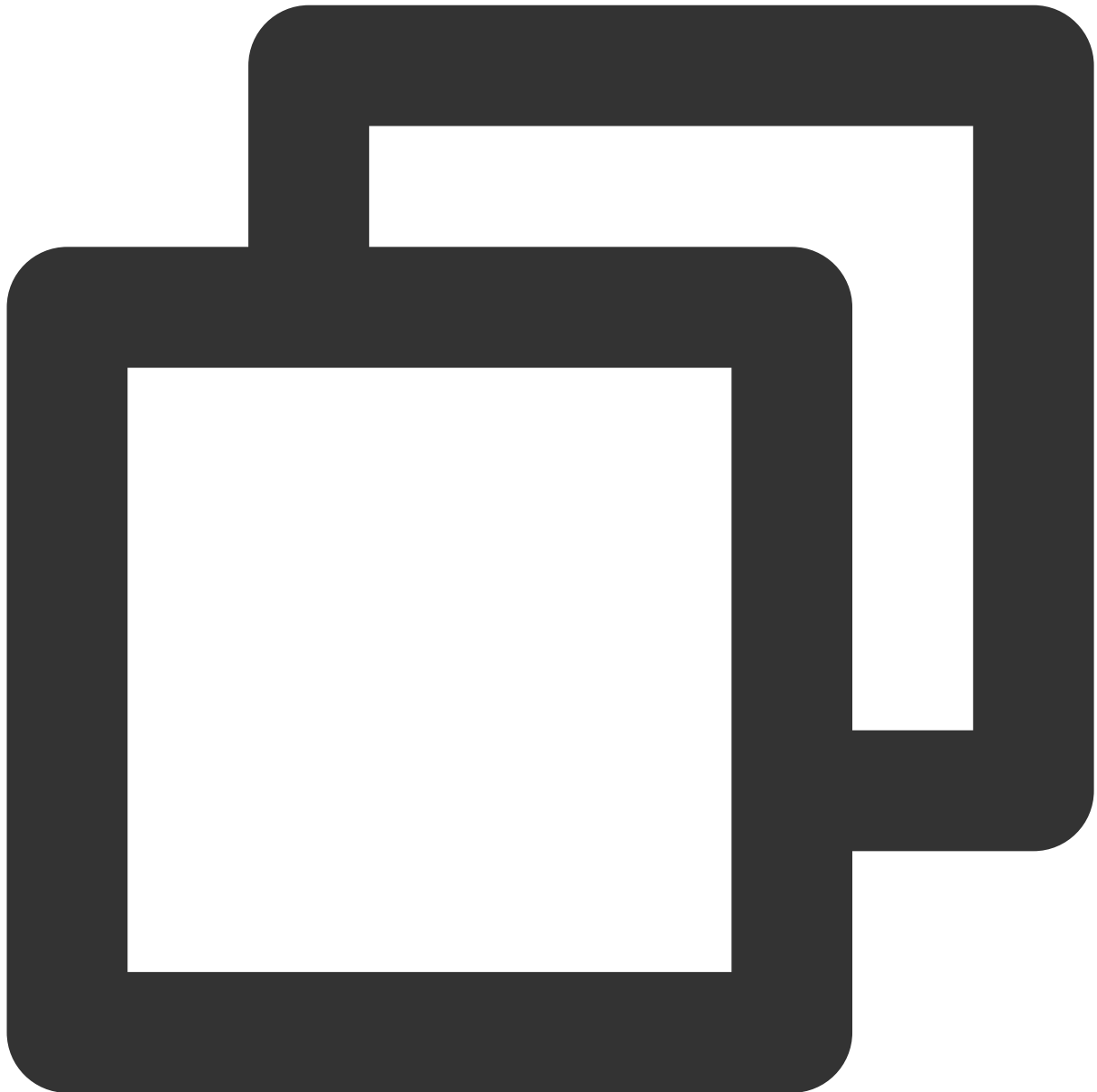
You can call the `deadLetterPolicy` API to customize these parameters. Below is the sample code:



```
Consumer<byte[]> consumer = pulsarClient.newConsumer()  
    .topic("persistent://pulsar-****")  
    .subscriptionName("sub1")  
    .subscriptionType(SubscriptionType.Shared)  
    .enableRetry(true) // Enable consumption retry  
    .deadLetterPolicy(DeadLetterPolicy.builder()  
        .maxRedeliverCount(maxRedeliveryCount) // Specify the maximum number of re  
        .retryLetterTopic("persistent://my-property/my-ns/sub1-retry") // Specify  
        .deadLetterTopic("persistent://my-property/my-ns/sub1-dlq") // Specify the  
        .build())  
    .subscribe();
```

Retry rules

The retry rules are implemented through the `reconsumeLater` API in three modes:



```
// Specify any delay time
consumer.reconsumeLater(msg, 1000L, TimeUnit.MILLISECONDS);
// Specify the delay level
consumer.reconsumeLater(msg, 1);
// Increase with level
consumer.reconsumeLater(msg);
```

Mode 1: specify any delay time. Enter the delay time in the second parameter and specify the time unit in the third parameter. The delay time is in the same value range as delayed message, which is 1–864,000 seconds.

Mode 2: specify any delay level (for existing users of the Tencent Cloud SDK only) . Its implementation effect is basically the same as that of mode 1, but it allows you to manage the delay time in distributed systems more easily. The delay level is as described below:

1.1 The second parameter in `reconsumeLater(msg, 1)` is the message level.

1.2 By default, the `MESSAGE_DELAYLEVEL = "1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h"` constant determines the respective delay time of each level; for example, level 1 corresponds to 1s and level 3 to 10s. You can customize the default value if it cannot meet your actual business needs.

Mode 3: increase with level (for existing users of the Tencent Cloud SDK only) . Different from the implementation effect of the above two modes, this mode adopts backoff retry; that is, the retry interval is 1s after the first failure, 5s after the second failure, and so on. The more the failures, the longer the interval. The specific retry interval is also determined by the `MESSAGE_DELAYLEVEL` parameter as described in mode 2.

This retry mechanism often has more practical applications in business scenarios. If consumption fails, the service generally will not be recovered immediately, therefore using such progressive retry mechanism makes more sense.

Note:

If you use the SDK provided by the Pulsar community, the delay level and increase with level modes are not supported.

Attributes of retry message

A retry message carries the following attributes:



```
{  
  REAL_TOPIC="persistent://my-property/my-ns/test,  
  ORIGIN_MESSAGE_ID=314:28:-1,  
  RETRY_TOPIC="persistent://my-property/my-ns/my-subscription-retry,  
  RECONSUMETIMES=16  
}
```

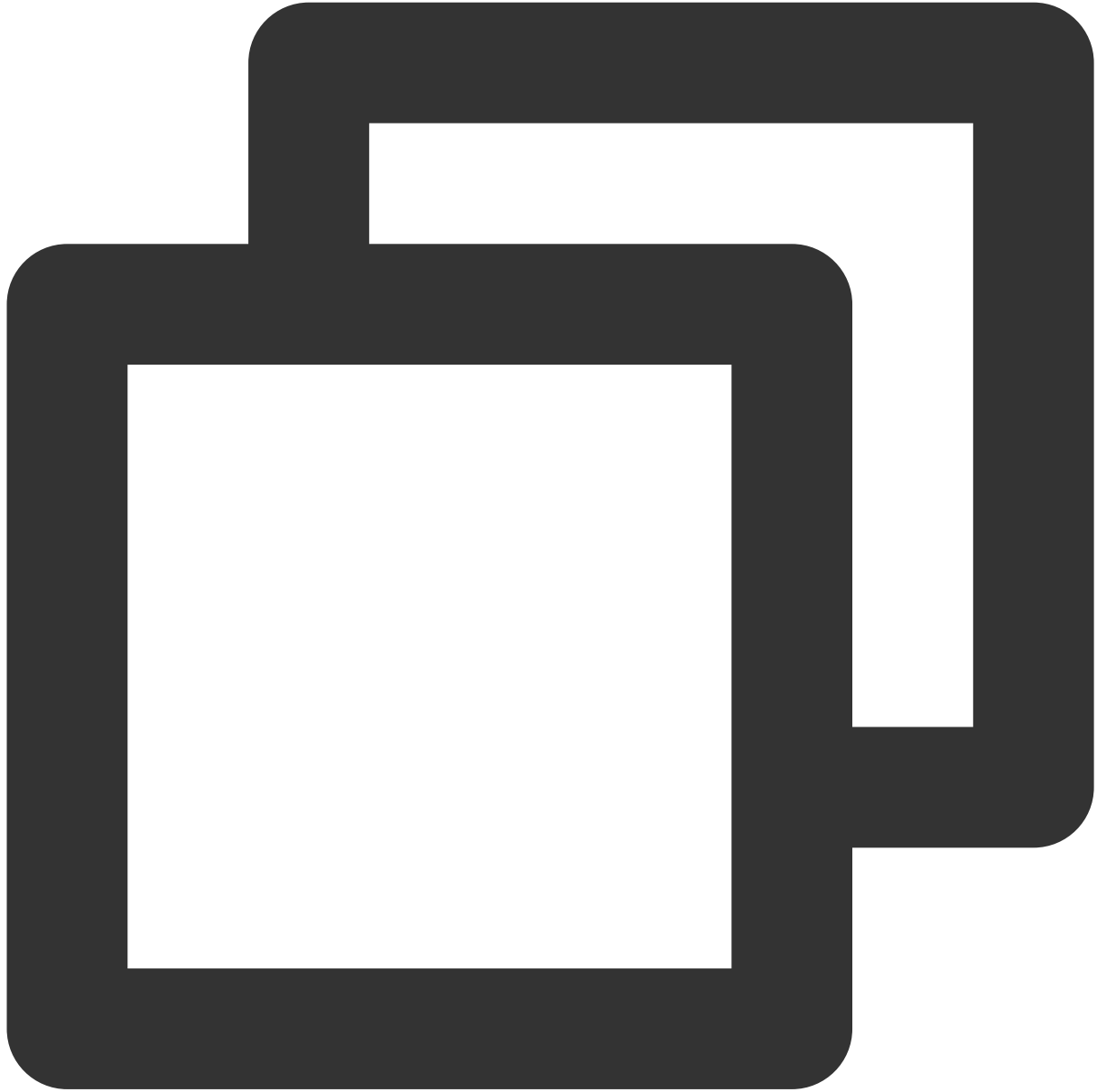
`REAL_TOPIC` : original topic

`ORIGIN_MESSAGE_ID` : ID of the initially produced message

`RETRY_TOPIC` : retry letter topic

`RECONSUMETIMES` : number of retries performed for the message

Method to Obtain Retry Count



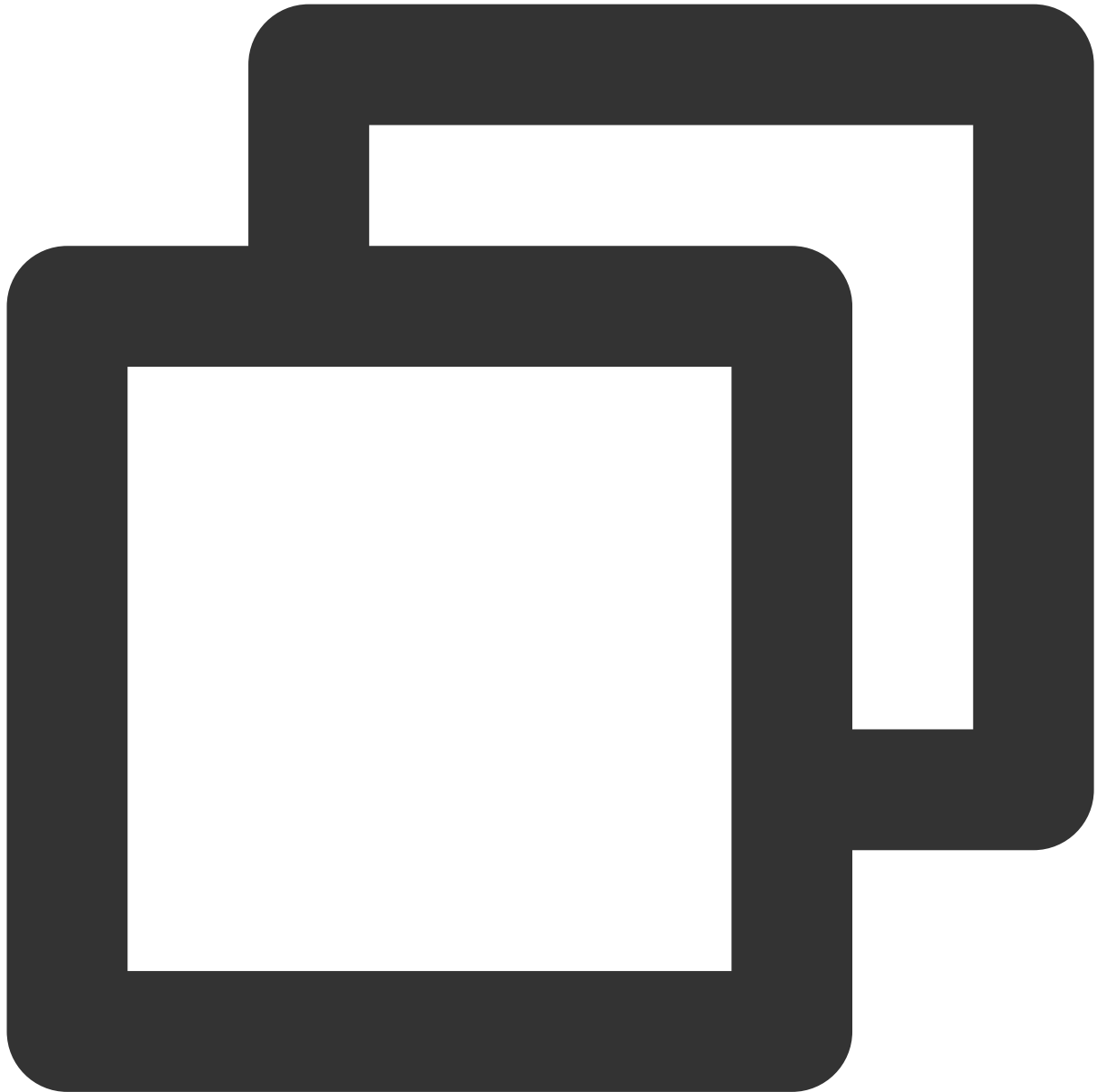
```
msg.getProperties().get("RECONSUMETIMES")
```

Note:

The number of retries obtained through the `msg.getRedeliveryCount()` API corresponds to the number under the `negativeAcknowledge` retry mode.

Flow of retry message ID

The message ID flow process is as shown below, and you can analyze relevant logs based on it.



```
Original consumption: msgid=1:1:0:1
1st retry: msgid=2:1:-1
2nd retry: msgid=2:2:-1
3rd retry: msgid=2:3:-1
.....
16th retry: msgid=2:16:0:1
17th write to the dead letter topic: msgid=3:1:-1
```

Complete sample code

The retry letter (-RETRY) topic must be enabled first in the consumer (`enableRetry(true)`), which is disabled by default. Then, the `reconsumeLater()` API needs to be called before messages can be sent to the retry letter topic.

The dead letter topic (-DLQ) must call `consumer.reconsumeLater()` . After `reconsumeLater()` is executed, the message of the original topic will be acked and transferred to the retry letter topic. After the retry reaches the upper limit, the message will then be transferred to the dead letter topic. The Pulsar client subscribes to the retry topic automatically, but not to the dead letter topic, which users must subscribe to on their own.

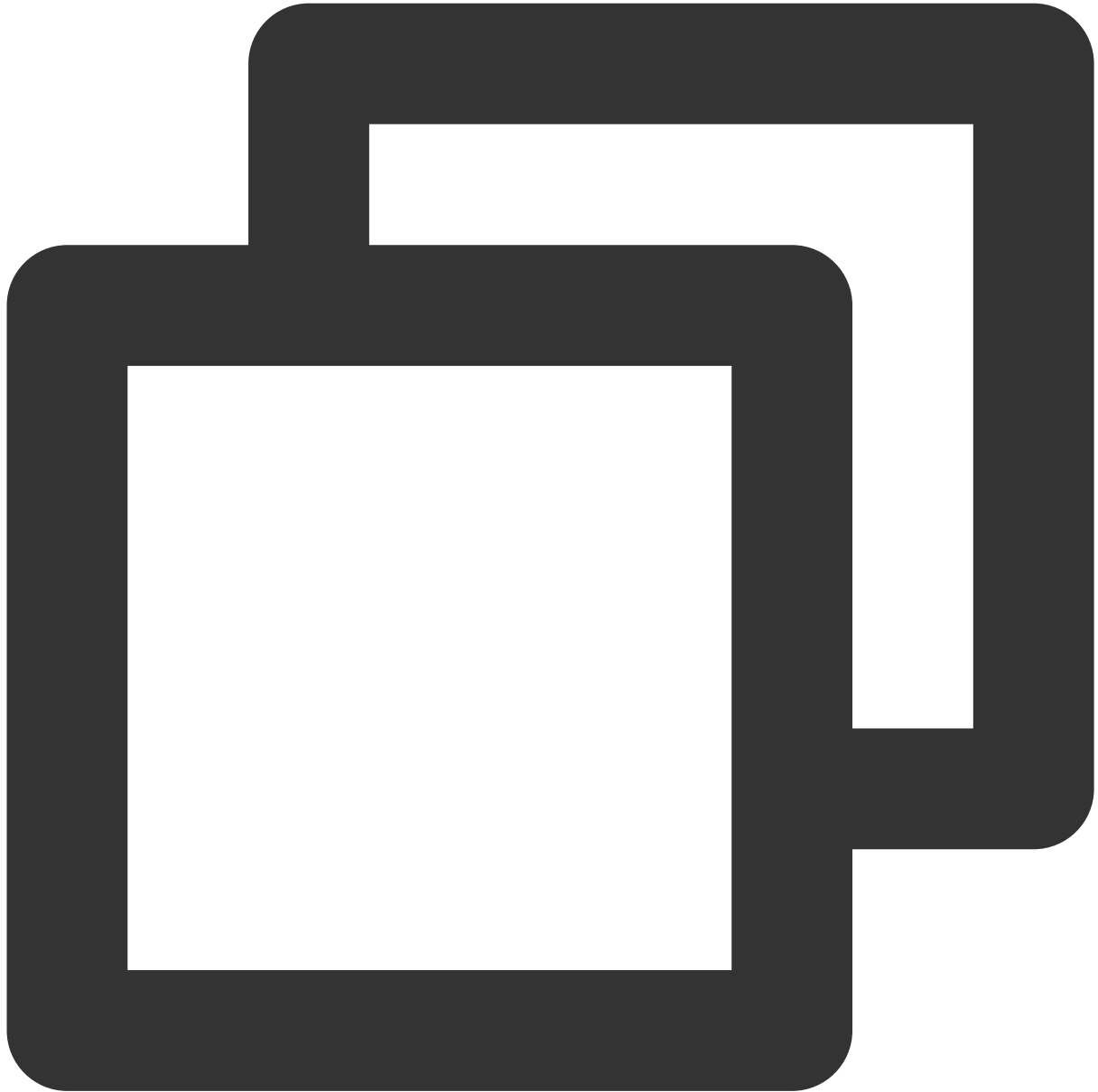
Below is the sample code for implementing the complete message retry mechanism in TDMQ for Apache Pulsar:

Subscribe to a topic



```
Consumer<byte[]> consumer1 = client.newConsumer()  
    .topic("persistent://pulsar-****")  
    .subscriptionName("my-subscription")  
    .subscriptionType(SubscriptionType.Shared)  
    .enableRetry(true) // Enable consumption retry  
    // .deadLetterPolicy(DeadLetterPolicy.builder()  
    //     .maxRedeliverCount(maxRedeliveryCount)  
    //     .retryLetterTopic("persistent://my-property/my-ns/my-subscriptio  
    //     .deadLetterTopic("persistent://my-property/my-ns/my-subscription  
    //     .build())  
    .subscribe();
```


Executing consumption



```
while (true) {  
    Message msg = consumer.receive();  
    try {  
        // Do something with the message  
        System.out.printf("Message received: %s", new String(msg.getData()));  
        // Acknowledge the message so that it can be deleted by the message bro  
        consumer.acknowledge(msg);  
    } catch (Exception e){
```

```
// select reconsume policy
consumer.reconsumeLater(msg, 1000L, TimeUnit.MILLISECONDS);
//consumer.reconsumeLater(msg, 1);
//consumer.reconsumeLater(msg);
}
}
```

Active Retry

If the client fails to consume a message and wants to reconsume it, the consumer can call the `negativeAcknowledge` API. The message will be obtained again after a period of time. The interval for reobtaining the message can be specified by consumers through the configuration of `negativeAckRedeliveryDelay` API.

Brief Description of the Implementation Mechanism

The clients cache the message ID of `negativeAcknowledge` and the list of `negativeAcknowledge` messages (the scan interval is `negativeAckRedeliveryDelay` x 1/3) is regularly scanned for the clients. When messages reach the time specified by `negativeAckRedeliveryDelay`, the client notifies the server to resubmit the messages that need to be retried. Upon receiving the client's redelivery request, the server re-pushes the corresponding messages to the client.

Note:

1. The actual time to receive the message again may be 1/3 more than the time specified by `negativeAckRedeliveryDelay`. This is related to the client's implementation logic.
2. In this method, no new messages are generated.

Below is the sample code in Java for active retry:



```
Consumer<byte[]> consumer = client.newConsumer()  
    .topic("persistent://pulsar-****")  
    .subscriptionName("my-subscription")  
    .subscriptionType(SubscriptionType.Shared)  
    // Default 1 min  
    .negativeAckRedeliveryDelay(1, TimeUnit.MINUTES)  
    .subscribe();  
  
while (true) {  
    Message msg = consumer.receive();
```

```
try {
    // Do something with the message
    System.out.printf("Message received: %s", new String(msg.getData()));
    // Acknowledge the message so that it can be deleted by the message broker
    consumer.acknowledge(msg);
} catch (Exception e){
    // Message failed to process, redeliver later
    consumer.negativeAcknowledge(msg);
}
}
```

Precautions

1. In the retry method of `negativeAcknowledge`, the messages that need to be retried are still unacked in the server's view.
2. In the retry method of `negativeAcknowledge`, there is no default maximum retry count. However, it can be achieved by configuring the maximum retry count and the dead letter queue. In this method, after a message has been retried for the specified number of times, it will be delivered to the dead letter queue.



```
Consumer<byte[]> consumer = client.newConsumer()  
    .topic("persistent://pulsar-****")  
    .subscriptionName("my-subscription")  
    .subscriptionType(SubscriptionType.Shared)  
    // Default 1 min  
    .negativeAckRedeliveryDelay(1, TimeUnit.MINUTES)  
    .deadLetterPolicy(DeadLetterPolicy.builder()  
        .maxRedeliverCount(5) // Specify the maximum number of retries  
        .deadLetterTopic("persistent://my-property/my-ns/sub1-dlq") // Specify  
        .build())  
    .subscribe();
```

```
while (true) {
    Message msg = consumer.receive();
    try {
        // Do something with the message
        System.out.printf("Message received: %s", new String(msg.getData()));
        // Acknowledge the message so that it can be deleted by the message broker
        consumer.acknowledge(msg);
    } catch (Exception e) {
        // Message failed to process, redeliver later
        consumer.negativeAcknowledge(msg);
    }
}
```

3. In the retry method of `negativeAcknowledge`, the retry count can be obtained from `msg.getRedeliveryCount()`. However, note that if all consumers under a subscription are offline, the message retry count will be reset to 0 (typical scenario: if there is only one consumer under a subscription, after the consumer restarts, the retry count of the messages will be reset to 0).

Client Connection and Producer/Consumer

Last updated : 2024-07-04 16:06:41

This document describes the relationship between the TDMQ for Apache Pulsar client and connection and between the client and producer/consumer, as well as how to use the client reasonably for higher efficiency and stability.

Core principles:

One PulsarClient is sufficient for one process.

Producers and consumers are thread-safe. They can be reused and are recommended to be reused for the same topic.

Client and Connection

The TDMQ for Apache Pulsar client (PulsarClient) is a basic unit that connects an application to TDMQ for Apache Pulsar, and one PulsarClient corresponds to one TCP connection. Generally, one application or process on the user side uses one PulsarClient, and the number of clients is equal to the number of application nodes. If an application node of the TDMQ for Apache Pulsar service is unused for a long time, its client should be repossessed to reduce the resource usage (currently, the connection limit in TDMQ for Apache Pulsar is 200 client connections per topic).

Note:

If there are many business topics, and multiple clients need to be created, reuse client objects in the following way:

1. Reuse the same client for producers or consumers of the same topic respectively.
2. If the above approach fails to meet the needs, try reusing the same client for multiple topics.
3. This limit only applies to virtual clusters. In pro clusters, the limit is still 200 by default but can be adjusted as needed.

Client and Producer/Consumer

You can create multiple producers and consumers under one client to produce and consume quickly. Typically, this is done by using multiple threads, which can isolate data among different producers and consumers.

Currently, the limits on the number of producers/consumers in TDMQ for Apache Pulsar are as follows:

Up to 1,000 producers per topic.

Up to 2,000 consumers per topic.

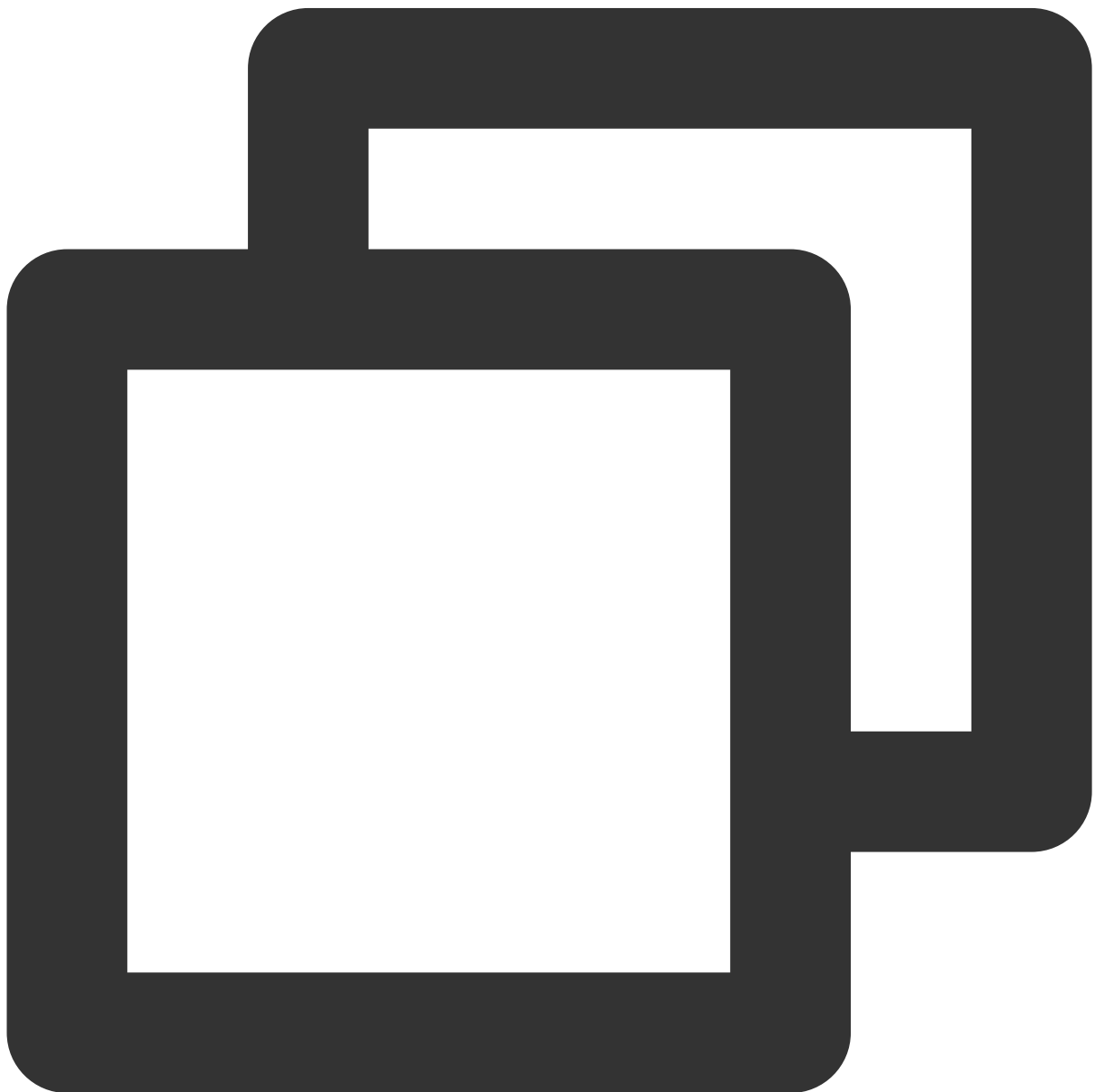
Practical Tutorial

The number of producers/consumers does not necessarily depend on the business object. They can be reused and are uniquely identified by name.

Producer

For 1,000 business objects to produce messages simultaneously, you don't necessarily need to create 1,000 producers. Since one single producer can often get the most out of the hardware configuration of each application node, it can be shared by multiple application nodes for production (singleton mode) if the messages are delivered to the same topic.

Below is the sample code in Java for message production.




```
// Get the `serviceURL` access address, token, full topic name, and subscription name
@Value("${tdmq.serviceUrl}")
private String serviceUrl;
@Value("${tdmq.token}")
private String token;
@Value("${tdmq.topic}")
private String topic;

// Declare a client object and producer object
private PulsarClient pulsarClient;
private Producer<String> producer;

// Create client and producer objects in an initialization program
public void init() throws Exception {
    pulsarClient = PulsarClient.builder()
        .serviceUrl(serviceUrl)
        .authentication(AuthenticationFactory.token(token))
        .build();
    producer = pulsarClient.newProducer(Schema.STRING)
        .topic(topic)
        .create();
}
```

Directly import the `producer` for message sending in the business logic of message production.



```
// Directly import the following code into the business logic of message production
public void onProduce(Producer<String> producer){
    // Add the business logic
    String msg = "my-message";// Simulate getting a message from the business logic
    try {
        // Schema verification is enabled in TDMQ for Apache Pulsar by default. The
        MessageId messageId = producer.newMessage()
            .key("msgKey")
            .value(msg)
            .send();
        System.out.println("delivered msg " + messageId + ", value:" + value);
    }
}
```

```
    } catch (PulsarClientException e) {
        System.out.println("delivered msg failed, value:" + value);
        e.printStackTrace();
    }
}

public void onProduceAsync(Producer<String> producer){
    // Add the business logic
    String msg = "my-async-message";// Simulate getting a message from the business
    // Send the message asynchronously, which avoids thread jamming and improves th
    CompletableFuture<MessageId> messageIdFuture = producer.newMessage()
        .key("msgKey")
        .value(msg)
        .sendAsync();
    // Check whether the message has been sent successfully from the async callback
    messageIdFuture.whenComplete(((messageId, throwable) -> {
        if( null != throwable ) {
            System.out.println("delivery failed, value: " + msg );
            // You can add the logic of delayed retry here
        } else {
            System.out.println("delivered msg " + messageId + ", value:" + msg);
        }
    }));
}
```

You need to call the `close` method to disable a long idle producer or client instance, in case they consume resources or jam up the connection pool.



```
public void destroy(){
    if (producer != null) {
        producer.close();
    }
    if (pulsarClient != null) {
        pulsarClient.close();
    }
}
```

Consumer

Like producers, consumers are recommended to be used in singleton mode, and a single consumption node needs only one client instance and one consumer instance. Generally, the consumption performance bottleneck of a message queue occurs somewhere in the consumer's own message handling process, rather than in the action of receiving messages. Therefore, when the consumption performance is poor, check the consumer's network bandwidth usage first. If the upper limit is not reached as seen from the trend, you should then analyze the message processing duration based on the logs and message trace information.

Note:

In shared or key-shared mode, the number of consumers is not necessarily less than or equal to the number of partitions. A module on the server responsible for delivering messages will deliver the messages to all consumers in a particular way (round robin by default in shared mode and round robin in the same key by default in key-shared mode).

In shared mode, if production is suspended on the producer, messages at the end may be distributed unevenly.

When multithreaded consumption is used, even if one consumer object is reused, the order of messages cannot be guaranteed.

Below is the complete sample code in Java for multithreaded consumption by using a thread pool based on the Spring Boot framework.



```
import org.apache.pulsar.client.api.*;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ExecutorService;
```

```
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

@Service
public class ConsumerService implements Runnable {

    // Get the `serviceURL` access address, token, full topic name, and subscription
    @Value("${tdmq.serviceUrl}")
    private String serviceUrl;
    @Value("${tdmq.token}")
    private String token;
    @Value("${tdmq.topic}")
    private String topic;
    @Value("${tdmq.subscription}")
    private String subscription;

    private volatile boolean start = false;
    private PulsarClient pulsarClient;
    private Consumer<String> consumer;
    private static final int corePoolSize = 10;
    private static final int maximumPoolSize = 10;

    private ExecutorService executor;
    private static final Logger logger = LoggerFactory.getLogger(ConsumerService.class);

    @PostConstruct
    public void init() throws Exception {
        pulsarClient = PulsarClient.builder()
            .serviceUrl(serviceUrl)
            .authentication(AuthenticationFactory.token(token))
            .build();
        consumer = pulsarClient.newConsumer(Schema.STRING)
            .topic(topic)
            //.subscriptionType(SubscriptionType.Shared)
            .subscriptionName(subscription)
            .subscribe();
        executor = new ThreadPoolExecutor(corePoolSize, maximumPoolSize, 0, TimeUnit.SECONDS,
            new ThreadPoolExecutor.AbortPolicy());
        start = true;
    }

    @PreDestroy
    public void destroy() throws Exception {
        start = false;
        if (consumer != null) {
            consumer.close();
        }
    }
}
```

```

        if (pulsarClient != null) {
            pulsarClient.close();
        }
        if (executor != null) {
            executor.shutdownNow();
        }
    }

    @Override
    public void run() {
        logger.info("tdmq consumer started...");
        for (int i = 0; i < maximumPoolSize; i++) {
            executor.submit(() -> {
                while (start) {
                    try {
                        Message<String> message = consumer.receive();
                        if (message == null) {
                            continue;
                        }
                        onConsumer(message);
                    } catch (Exception e) {
                        logger.warn("tdmq consumer business error", e);
                    }
                }
            });
        }
        logger.info("tdmq consumer stopped...");
    }

    /**
     * Write the consumption business logic here
     *
     * @param message
     * @return return true: message ack; return false: message nack
     * @throws Exception Message nack
     */
    private void onConsumer(Message<String> message) {
        // Business logic of delay operation
        try {
            System.out.println(Thread.currentThread().getName() + " - message received");
            Thread.sleep(1000); // Simulate business logic processing
            consumer.acknowledge(message);
            logger.info(Thread.currentThread().getName() + " - message processing success");
        } catch (Exception exception) {
            consumer.negativeAcknowledge(message);
            logger.error(Thread.currentThread().getName() + " - message processing failed", exception);
        }
    }

```



```
}  
}
```