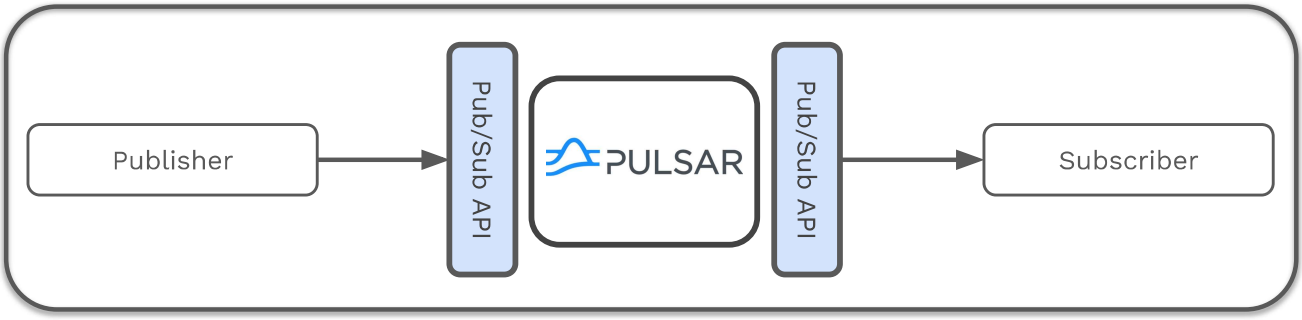
Messaging

Pulsar is built on the publish-subscribe pattern (often abbreviated to pub-sub). In this pattern, producers publish messages to topics; consumers subscribe to those topics, process incoming messages, and send acknowledgments to the broker when processing is finished.



When a subscription is created, Pulsar retains all messages, even if the consumer is disconnected. The retained messages are discarded only when a consumer acknowledges that all these messages are processed successfully.

If the consumption of a message fails and you want this message to be consumed again, you can enable message redelivery mechanism to request the broker to resend this message.

Messages

Messages are the basic "unit" of Pulsar. The following table lists the components of messages.

Component	Description
Value / data payload	The data carried by the message. All Pulsar messages contain raw bytes, although message data can also conform to data schemas.

Component	Description
Key	The key (string type) of the message. It is a short name of message key or partition key. Messages are optionally tagged with keys, which is useful for features like topic compaction.
Properties	An optional key/value map of user-defined properties.
Producer name	The name of the producer who produces the message. If you do not specify a producer name, the default name is used.
Topic name	The name of the topic that the message is published to.
Schema version	The version number of the schema that the message is produced with.
Sequence ID	Each Pulsar message belongs to an ordered sequence on its topic. The sequence ID of a message is initially assigned by its producer, indicating its order in that sequence, and can also be customized. Sequence ID can be used for message deduplication. If `brokerDeduplicationEnabled` is set to `true`, the sequence ID of each message is unique within a producer of a topic (non-partitioned) or a partition.
Message ID	The message ID of a message is assigned by bookies as soon as the message is persistently stored. Message ID indicates a message's specific position in a ledger and is unique within a Pulsar cluster.
Publish time	The timestamp of when the message is published. The timestamp is automatically applied by the producer.
Event time	An optional timestamp attached to a message by applications. For example, applications attach a timestamp on when the message is processed. If nothing is set to event time, the value is `0`.

The default size of a message is 5 MB. You can configure the max size of a message with the following configurations.

- In the `broker.conf` file.

```
# The max size of a message (in bytes).
maxMessageSize=5242880
```

- In the `bookkeeper.conf` file.

```
# The max size of the netty frame (in bytes). Any messages received larger than this value are rejected. The default value is 5 MB.
nettyMaxFrameSizeBytes=5253120
```

> For more information on Pulsar messages, see Pulsar binary protocol.

Producers

A producer is a process that attaches to a topic and publishes messages to a Pulsar broker. The Pulsar broker processes the messages.

Send modes

Producers send messages to brokers synchronously (sync) or asynchronously (async).

Mode	Description
Sync send	The producer waits for an acknowledgment from the broker after sending every message. If the acknowledgment is not received, the producer treats the sending operation as a failure.
Async send	The producer puts a message in a blocking queue and returns immediately. The client library sends the message to the broker in the background. If the queue is full (you can configure the maximum size), the producer is blocked or fails immediately when calling the API, depending on arguments passed to the producer.

Access mode

You can have different types of access modes on topics for producers.

Access mode	Description
`Shared`	Multiple producers can publish on a topic. This is the **default** setting.
`Exclusive`	Only one producer can publish on a topic. If there is already a producer connected, other producers trying to publish on this topic get errors immediately.

Access mode	Description
	The "old" producer is evicted and a "new" producer is selected to be the next exclusive producer if the "old" producer experiences a network partition with the broker.
`ExclusiveWithFencing`	Only one producer can publish on a topic. If there is already a producer connected, it will be removed and invalidated immediately.
`WaitForExclusive`	If there is already a producer connected, the producer creation is pending (rather than timing out) until the producer gets the `Exclusive` access. The producer that succeeds in becoming the exclusive one is treated as the leader. Consequently, if you want to implement a leader election scheme for your application, you can use this access mode. Note that the leader pattern scheme mentioned refers to using Pulsar as a Write-Ahead Log (WAL) which means the leader writes its "decisions" to the topic. On error cases, the leader will get notified it is no longer the leader *only* when it tries to write a message and fails on appropriate error, by the broker.

> **NOTE**
>
> Once an application creates a producer with `Exclusive` or `WaitForExclusive` access mode successfully, the instance of this application is guaranteed to be the **only writer** to the topic. Any other producers trying to produce messages on this topic will either get errors immediately or have to wait until they get the `Exclusive` access. For more information, see PIP 68: Exclusive Producer.

You can set producer access mode through Java Client API. For more information, see `ProducerAccessMode` in ProducerBuilder.java file.

Compression

Message compression can reduce message size by paying some CPU overhead. The Pulsar client supports the following compression types:

- LZ4
- ZLIB
- ZSTD
- SNAPPY.

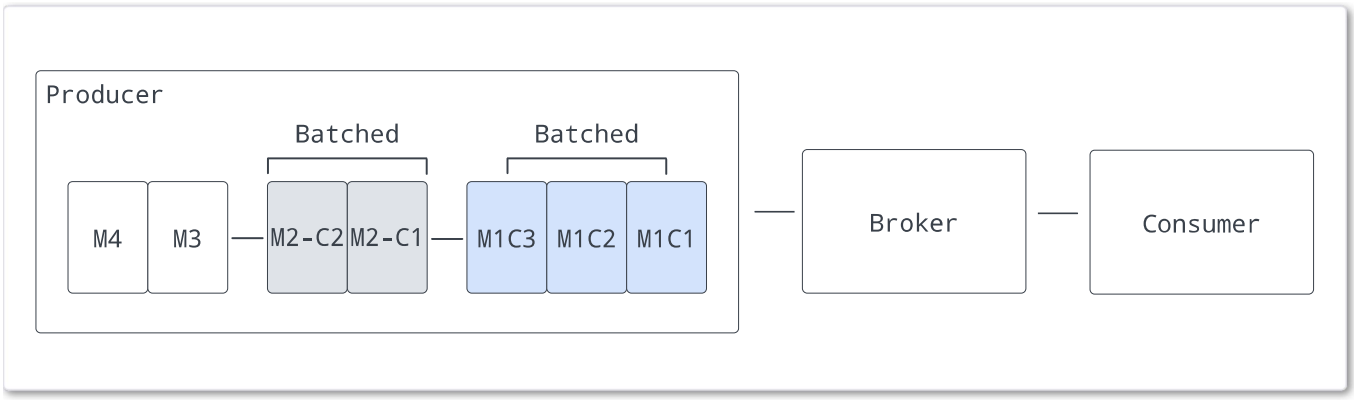
Compression types are stored in the message metadata, so consumers can adopt different compression types automatically, as needed.

The sample code below shows how to enable compression type for a producer:

```
client.newProducer()
    .topic("topic-name")
    .compressionType(CompressionType.LZ4)
    .create();
```

Batching

When batching is enabled, the producer accumulates and sends a batch of messages in a single request. The batch size is defined by the maximum number of messages and the maximum publish latency. Therefore, the backlog size represents the total number of batches instead of the total number of messages.



In Pulsar, batches are tracked and stored as single units rather than as individual messages. Consumers unbundle a batch into individual messages. However, scheduled messages (configured through the `deliverAt` or the `deliverAfter` parameter) are always sent as individual messages even when batching is enabled.

In general, a batch is acknowledged when all of its messages are acknowledged by a consumer. It means that when **not all** batch messages are acknowledged, then unexpected failures, negative acknowledgments, or acknowledgment timeouts can result in a redelivery of all messages in this batch.

To avoid redelivering acknowledged messages in a batch to the consumer, Pulsar introduces batch index acknowledgment since Pulsar 2.6.0. When batch index acknowledgment is enabled, the consumer filters out the batch index that has been acknowledged and sends the batch index acknowledgment request to the broker. The broker maintains the batch index acknowledgment status and tracks the acknowledgment status of each batch index to avoid dispatching acknowledged messages to the consumer. The batch is deleted when all indices of the messages in it are acknowledged.

By default, batch index acknowledgment is disabled (`acknowledgmentAtBatchIndexLevelEnabled=false`). You can enable batch index acknowledgment by setting the `acknowledgmentAtBatchIndexLevelEnabled` parameter to `true` at the broker side. Enabling batch index acknowledgment results in more memory overheads.

Batch index acknowledgment must also be enabled in the consumer by calling `.enableBatchIndexAcknowledgment(true);`

For example:

```
Consumer<byte[]> consumer = pulsarClient.newConsumer()
        .topic(topicName)
        .subscriptionName(subscriptionName)
        .subscriptionType(subType)
        .enableBatchIndexAcknowledgment(true)
        .subscribe();
```

Chunking

Message chunking enables Pulsar to process large payload messages by splitting the message into chunks at the producer side and aggregating chunked messages at the consumer side.

With message chunking enabled, when the size of a message exceeds the allowed maximum payload size (the `maxMessageSize` parameter of broker), the workflow of messaging is as follows:

1. The producer splits the original message into chunked messages and publishes them with chunked metadata to the broker separately and in order.
2. The broker stores the chunked messages in one managed ledger in the same way as that of ordinary messages, and it uses the `chunkedMessageRate` parameter to record chunked message rate on the topic.
3. The consumer buffers the chunked messages and aggregates them into the receiver queue when it receives all the chunks of a message.
4. The client consumes the aggregated message from the receiver queue.

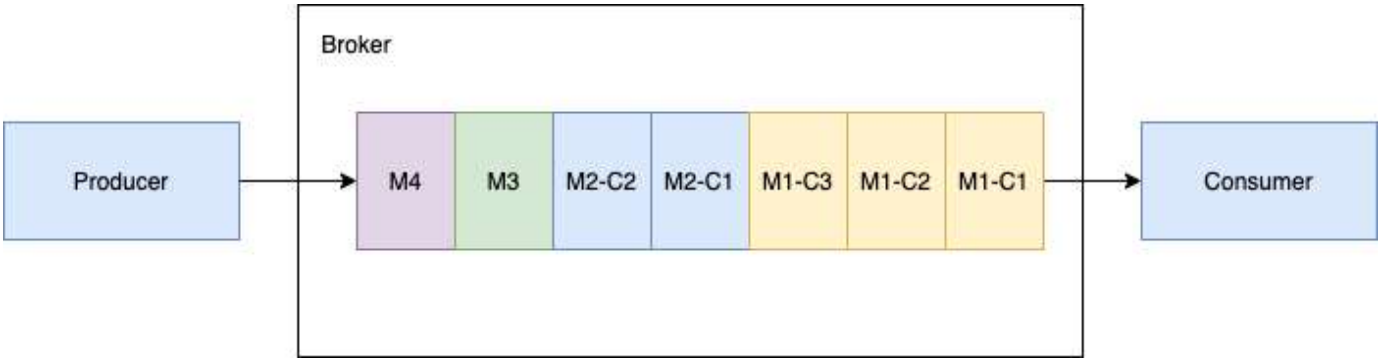
Limitations:

- Chunking is only available for persisted topics.
- Chunking is only available for the exclusive and failover subscription types.
- Chunking cannot be enabled simultaneously with batching.

Handle consecutive chunked messages with one ordered consumer

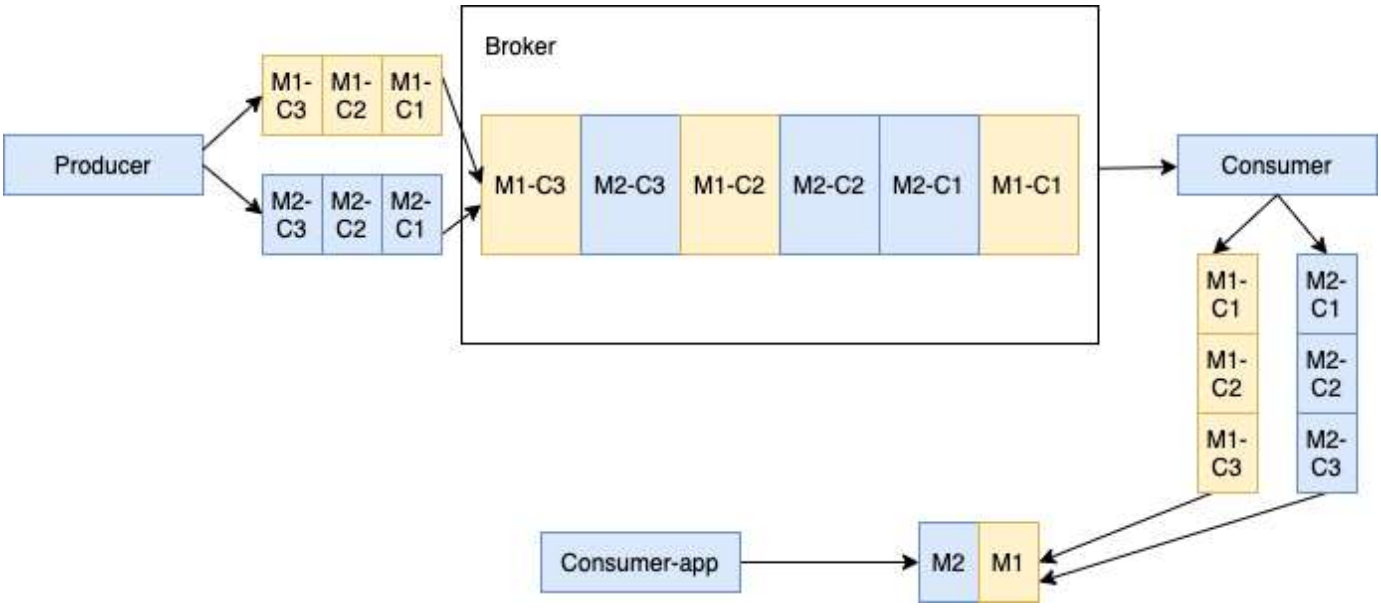
The following figure shows a topic with one producer that publishes a large message payload in chunked messages along with regular non-chunked messages. The producer publishes message M1 in three chunks labeled M1-C1, M1-C2 and M1-C3. The broker stores all the three chunked messages in the managed ledger and dispatches them to the ordered (exclusive/failover) consumer in the same order. The consumer buffers all the chunked

messages in memory until it receives all the chunked messages, aggregates them into one message and then hands over the original message M1 to the client.



Handle interwoven chunked messages with one ordered consumer

When multiple producers publish chunked messages into a single topic, the broker stores all the chunked messages coming from different producers in the same managed ledger. The chunked messages in the managed ledger can be interwoven with each other. As shown below, Producer 1 publishes message M1 in three chunks M1-C1, M1-C2 and M1-C3. Producer 2 publishes message M2 in three chunks M2-C1, M2-C2 and M2-C3. All chunked messages of the specific message are still in order but might not be consecutive in the managed ledger.



> **NOTE**
>
> In this case, interwoven chunked messages may bring some memory pressure to the consumer because the consumer keeps a separate buffer for each large message to aggregate all its chunks in one message. You can limit the maximum number of chunked messages a consumer maintains concurrently by configuring the `maxPendingChunkedMessage` parameter. When the threshold is reached, the consumer drops pending messages by silently acknowledging them or asking the broker to redeliver them later, optimizing memory utilization.

Enable Message Chunking

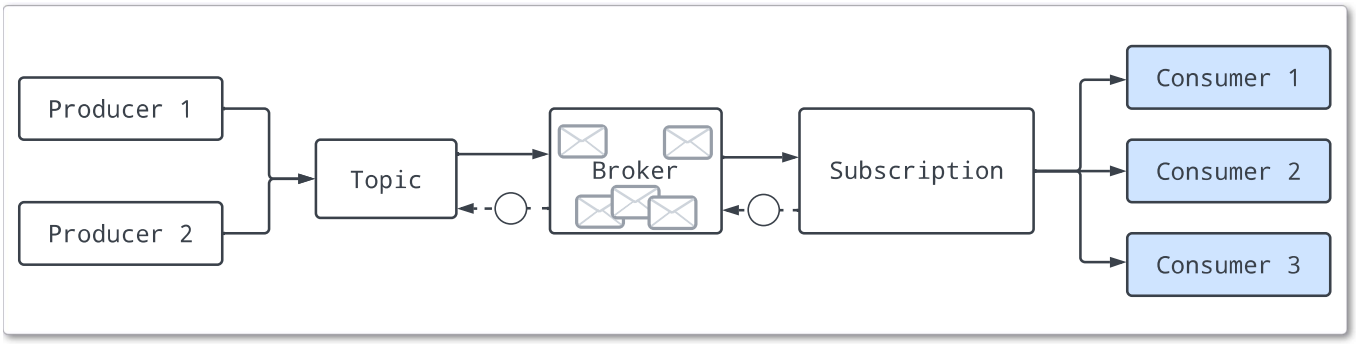
Prerequisite: Disable batching by setting the `enableBatching` parameter to `false`.

The message chunking feature is OFF by default. To enable message chunking, set the `chunkingEnabled` parameter to `true` when creating a producer.

> **NOTE**
>
> If the consumer fails to receive all chunks of a message within a specified period, it expires incomplete chunks. The default value is 1 minute. For more information about the `expireTimeOfIncompleteChunkedMessage` parameter, refer to org.apache.pulsar.client.api.

Consumers

A consumer is a process that attaches to a topic via a subscription and then receives messages.



A consumer sends a flow permit request to a broker to get messages. There is a queue at the consumer side to receive messages pushed from the broker. You can configure the queue size with the `receiverQueueSize` parameter. The default size is `1000`). Each time `consumer.receive()` is called, a message is dequeued from the buffer.

Receive modes

Messages are received from brokers either synchronously (sync) or asynchronously (async).

Mode	Description
Sync receive	A sync receive is blocked until a message is available.
Async receive	An async receive returns immediately with a future value—for example, a `CompletableFuture` in Java—that completes once a new message is available.

Listeners

Client libraries provide listener implementation for consumers. For example, the Java client provides a MesssageListener interface. In this interface, the `received` method is called whenever a new message is received.

Acknowledgment

The consumer sends an acknowledgment request to the broker after it consumes a message successfully. Then, this consumed message will be permanently stored, and deleted only after all the subscriptions have acknowledged it. If you want to store the messages that have been acknowledged by a consumer, you need to configure the message retention policy.

For batch messages, you can enable batch index acknowledgment to avoid dispatching acknowledged messages to the consumer. For details about batch index acknowledgment, see batching.

Messages can be acknowledged in one of the following two ways:

- Being acknowledged individually. With individual acknowledgment, the consumer acknowledges each message and sends an acknowledgment request to the broker.
- Being acknowledged cumulatively. With cumulative acknowledgment, the consumer **only** acknowledges the last message it received. All messages in the stream up to (and including) the provided message are not redelivered to that consumer.

If you want to acknowledge messages individually, you can use the following API.

```
consumer.acknowledge(msg);
```

If you want to acknowledge messages cumulatively, you can use the following API.

```
consumer.acknowledgeCumulative(msg);
```

> **NOTE**
>
> Cumulative acknowledgment cannot be used in Shared subscription type, because Shared subscription type involves multiple consumers which have access to the same subscription. In Shared subscription type, messages are acknowledged individually.

Negative acknowledgment

The negative acknowledgment mechanism allows you to send a notification to the broker indicating the consumer did not process a message. When a consumer fails to consume a message and needs to re-consume it, the consumer sends a negative acknowledgment (nack) to the broker, triggering the broker to redeliver this message to the consumer.

Messages are negatively acknowledged individually or cumulatively, depending on the consumption subscription type.

In Exclusive and Failover subscription types, consumers only negatively acknowledge the last message they receive.

In Shared and Key_Shared subscription types, consumers can negatively acknowledge messages individually.

Be aware that negative acknowledgments on ordered subscription types, such as Exclusive, Failover and Key_Shared, might cause failed messages being sent to consumers out of the original order.

If you are going to use negative acknowledgment on a message, make sure it is negatively acknowledged before the acknowledgment timeout.

Use the following API to negatively acknowledge message consumption.

```
Consumer<byte[]> consumer = pulsarClient.newConsumer()
                .topic(topic)
                .subscriptionName("sub-negative-ack")

.subscriptionInitialPosition(SubscriptionInitialPosition.Earliest)
                .negativeAckRedeliveryDelay(2, TimeUnit.SECONDS) // the default
value is 1 min
                .subscribe();

Message<byte[]> message = consumer.receive();

// call the API to send negative acknowledgment
consumer.negativeAcknowledge(message);

message = consumer.receive();
consumer.acknowledge(message);
```

To redeliver messages with different delays, you can use the **redelivery backoff mechanism** by setting the number of retries to deliver the messages. Use the following API to enable `Negative Redelivery Backoff`.

```
Consumer<byte[]> consumer = pulsarClient.newConsumer()
        .topic(topic)
        .subscriptionName("sub-negative-ack")
        .subscriptionInitialPosition(SubscriptionInitialPosition.Earliest)
        .negativeAckRedeliveryBackoff(MultiplierRedeliveryBackoff.builder()
            .minDelayMs(1000)
            .maxDelayMs(60 * 1000)
            .multiplier(2)
```

```
            .build())
        .subscribe();
```

The message redelivery behavior should be as follows.

Redelivery count	Redelivery delay
1	1 seconds
2	2 seconds
3	4 seconds
4	8 seconds
5	16 seconds
6	32 seconds
7	60 seconds
8	60 seconds

> **NOTE**
>
> If batching is enabled, all messages in one batch are redelivered to the consumer.

Acknowledgment timeout

> **NOTE**
>
> By default, the acknowledge timeout is disabled and that means that messages delivered to a consumer will not be re-delivered unless the consumer crashes.

The acknowledgment timeout mechanism allows you to set a time range during which the client tracks the unacknowledged messages. After this acknowledgment timeout (`ackTimeout`) period, the client sends `redeliver unacknowledged messages` request to the broker, thus the broker resends the unacknowledged messages to the consumer.

You can configure the acknowledgment timeout mechanism to redeliver the message if it is not acknowledged after `ackTimeout` or to execute a timer task to check the acknowledgment timeout messages during every `ackTimeoutTickTime` period.

You can also use the redelivery backoff mechanism to redeliver messages with different delays by setting the number of times the messages are retried.

If you want to use redelivery backoff, you can use the following API.

```
consumer.ackTimeout(10, TimeUnit.SECOND)
        .ackTimeoutRedeliveryBackoff(MultiplierRedeliveryBackoff.builder()
            .minDelayMs(1000)
            .maxDelayMs(60 * 1000)
            .multiplier(2)
            .build());
```

The message redelivery behavior should be as follows.

Redelivery count	Redelivery delay
1	10 + 1 seconds
2	10 + 2 seconds
3	10 + 4 seconds
4	10 + 8 seconds
5	10 + 16 seconds
6	10 + 32 seconds
7	10 + 60 seconds
8	10 + 60 seconds

NOTE

- If batching is enabled, all messages in one batch are redelivered to the consumer.
- Compared with acknowledgment timeout, negative acknowledgment is preferred. First, it is difficult to set a timeout value. Second, a broker resends messages when the message processing time exceeds the acknowledgment timeout, but these messages might not need to be re-consumed.

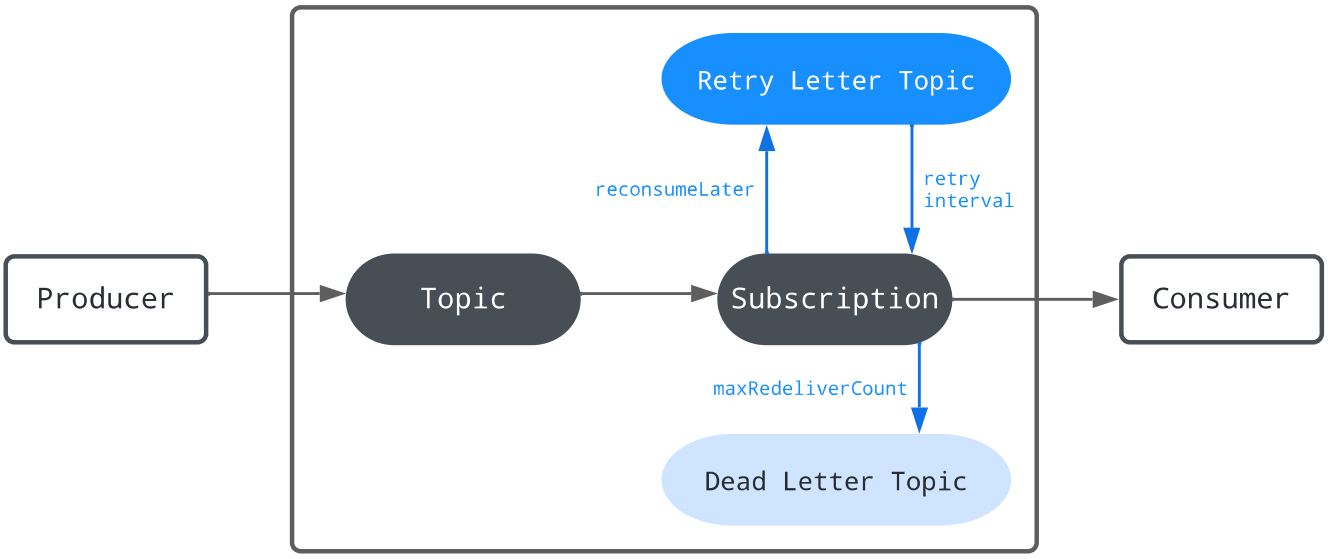
Use the following API to enable acknowledgment timeout.

```
Consumer<byte[]> consumer = pulsarClient.newConsumer()
                .topic(topic)
                .ackTimeout(2, TimeUnit.SECONDS) // the default value is 0
                .ackTimeoutTickTime(1, TimeUnit.SECONDS)
                .subscriptionName("sub")

.subscriptionInitialPosition(SubscriptionInitialPosition.Earliest)
                .subscribe();

Message<byte[]> message = consumer.receive();

// wait at least 2 seconds
message = consumer.receive();
consumer.acknowledge(message);
```

Retry letter topic

Retry letter topic allows you to store the messages that failed to be consumed and retry consuming them later. With this method, you can customize the interval at which the messages are redelivered. Consumers on the original topic are automatically subscribed to the retry letter topic as well. Once the maximum number of retries has been reached, the unconsumed messages are moved to a dead letter topic for manual processing. The functionality of a retry letter topic is implemented by consumers.

The diagram below illustrates the concept of the retry letter topic.



The intention of using retry letter topic is different from using delayed message delivery, even though both are aiming to consume a message later. Retry letter topic serves failure handling through message redelivery to ensure critical data is not lost, while delayed message delivery is intended to deliver a message with a specified time delay.

By default, automatic retry is disabled. You can set `enableRetry` to `true` to enable automatic retry on the consumer.

Use the following API to consume messages from a retry letter topic. When the value of `maxRedeliverCount` is reached, the unconsumed messages are moved to a dead letter topic.

```
Consumer<byte[]> consumer = pulsarClient.newConsumer(Schema.BYTES)
                .topic("my-topic")
                .subscriptionName("my-subscription")
                .subscriptionType(SubscriptionType.Shared)
                .enableRetry(true)
                .deadLetterPolicy(DeadLetterPolicy.builder()
                        .maxRedeliverCount(maxRedeliveryCount)
                        .build())
                .subscribe();
```

The default retry letter topic uses this format:

```
<topicname>-<subscriptionname>-RETRY
```

Use the Java client to specify the name of the retry letter topic.

```
Consumer<byte[]> consumer = pulsarClient.newConsumer(Schema.BYTES)
        .topic("my-topic")
        .subscriptionName("my-subscription")
        .subscriptionType(SubscriptionType.Shared)
        .enableRetry(true)
        .deadLetterPolicy(DeadLetterPolicy.builder()
                .maxRedeliverCount(maxRedeliveryCount)
                .retryLetterTopic("my-retry-letter-topic-name")
                .build())
        .subscribe();
```

The messages in the retry letter topic contain some special properties that are automatically created by the client.

Special property	Description
`REAL_TOPIC`	The real topic name.
`ORIGIN_MESSAGE_ID`	The origin message ID. It is crucial for message tracking.
`RECONSUMETIMES`	The number of retries to consume messages.
`DELAY_TIME`	Message retry interval in milliseconds.

Example

```
REAL_TOPIC = persistent://public/default/my-topic
ORIGIN_MESSAGE_ID = 1:0:-1:0
RECONSUMETIMES = 6
DELAY_TIME = 3000
```

Use the following API to store the messages in a retrial queue.

```
consumer.reconsumeLater(msg, 3, TimeUnit.SECONDS);
```

Use the following API to add custom properties for the `reconsumeLater` function. In the next attempt to consume, custom properties can be get from message#getProperty.

```
Map<String, String> customProperties = new HashMap<String, String>();
customProperties.put("custom-key-1", "custom-value-1");
customProperties.put("custom-key-2", "custom-value-2");
consumer.reconsumeLater(msg, customProperties, 3, TimeUnit.SECONDS);
```

> **NOTE**
>
> - Currently, retry letter topic is enabled in Shared subscription types.
> - Compared with negative acknowledgment, retry letter topic is more suitable for messages that require a large number of retries with a configurable retry interval. Because messages in the retry letter topic are persisted to BookKeeper, while messages that need to be retried due to negative acknowledgment are cached on the client side.

Dead letter topic

Dead letter topic allows you to continue message consumption even when some messages are not consumed successfully. The messages that have failed to be consumed are stored in a specific topic, which is called the dead letter topic. The functionality of a dead letter topic is implemented by consumers. You can decide how to handle the messages in the dead letter topic.

Enable dead letter topic in a Java client using the default dead letter topic.

```
Consumer<byte[]> consumer = pulsarClient.newConsumer(Schema.BYTES)
                .topic("my-topic")
                .subscriptionName("my-subscription")
                .subscriptionType(SubscriptionType.Shared)
```

```
                .deadLetterPolicy(DeadLetterPolicy.builder()
                      .maxRedeliverCount(maxRedeliveryCount)
                      .build())
                .subscribe();
```

The default dead letter topic uses this format:

```
<topicname>-<subscriptionname>-DLQ
```

Use the Java client to specify the name of the dead letter topic.

```
Consumer<byte[]> consumer = pulsarClient.newConsumer(Schema.BYTES)
                .topic("my-topic")
                .subscriptionName("my-subscription")
                .subscriptionType(SubscriptionType.Shared)
                .deadLetterPolicy(DeadLetterPolicy.builder()
                      .maxRedeliverCount(maxRedeliveryCount)
                      .deadLetterTopic("my-dead-letter-topic-name")
                      .build())
                .subscribe();
```

By default, there is no subscription during DLQ topic creation. Without a just-in-time subscription to the DLQ topic, you may lose messages. To automatically create an initial subscription for the DLQ, you can specify the `initialSubscriptionName` parameter. If this parameter is set but the broker's `allowAutoSubscriptionCreation` is disabled, the DLQ producer will fail to be created.

```
Consumer<byte[]> consumer = pulsarClient.newConsumer(Schema.BYTES)
                .topic("my-topic")
                .subscriptionName("my-subscription")
                .subscriptionType(SubscriptionType.Shared)
                .deadLetterPolicy(DeadLetterPolicy.builder()
                      .maxRedeliverCount(maxRedeliveryCount)
                      .deadLetterTopic("my-dead-letter-topic-name")
                      .initialSubscriptionName("init-sub")
                      .build())
                .subscribe();
```

Dead letter topic serves message redelivery, which is triggered by acknowledgment timeout or negative acknowledgment or retry letter topic.

> **NOTE**
>
> Currently, dead letter topic is enabled in Shared and Key_Shared subscription types.

Topics

As in other pub-sub systems, topics in Pulsar are named channels for transmitting messages from producers to consumers. Topic names are URLs that have a well-defined structure:

```
{persistent|non-persistent}://tenant/namespace/topic
```

Topic name component	Description
`persistent` / `non-persistent`	This identifies the type of topic. Pulsar supports two kind of topics: persistent and non-persistent. The default is persistent, so if you do not specify a type, the topic is persistent. With persistent topics, all messages are durably persisted on disks (if the broker is not standalone, messages are durably persisted on multiple disks), whereas data for non-persistent topics is not persisted to storage disks.
`tenant`	The topic tenant within the instance. Tenants are essential to multi-tenancy in Pulsar, and spread across clusters.
`namespace`	The administrative unit of the topic, which acts as a grouping mechanism for related topics. Most topic configuration is performed at the namespace level. Each tenant has one or more namespaces.
`topic`	The final part of the name. Topic names have no special meaning in a Pulsar instance.

NOTE

You do not need to explicitly create topics in Pulsar. If a client attempts to write or receive messages to/from a topic that does not yet exist, Pulsar creates that topic under the namespace provided in the topic name automatically. If no tenant or namespace is specified when a client creates a topic, the topic is created in the default tenant and namespace. You can also create a topic in a specified tenant and namespace, such as `persistent://my-tenant/my-namespace/my-topic`. `persistent://my-tenant/my-namespace/my-topic` means the `my-topic` topic is created in the `my-namespace` namespace of the `my-tenant` tenant.

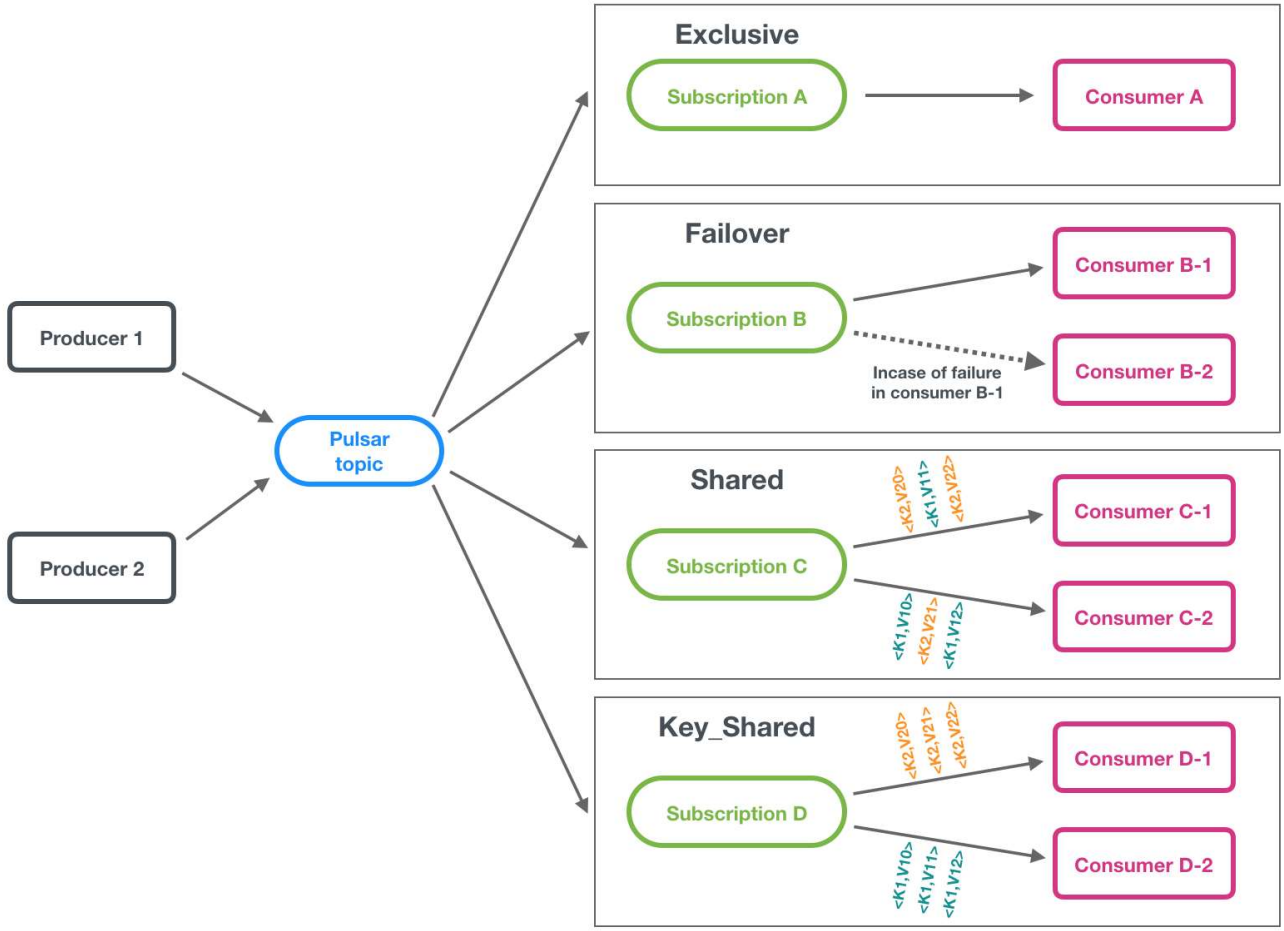
Namespaces

A namespace is a logical nomenclature within a tenant. A tenant creates namespaces via the admin API. For instance, a tenant with different applications can create a separate namespace for each application. A namespace allows the application to create and manage a hierarchy of

topics. The topic `my-tenant/app1` is a namespace for the application `app1` for `my-tenant`. You can create any number of topics under the namespace.

Subscriptions

A subscription is a named configuration rule that determines how messages are delivered to consumers. Four subscription types are available in Pulsar: exclusive, shared, failover, and key_shared. These types are illustrated in the figure below.



TIP

Pub-Sub or Queuing In Pulsar, you can use different subscriptions flexibly.

- If you want to achieve traditional "fan-out pub-sub messaging" among consumers, specify a unique subscription name for each consumer. It is an exclusive subscription type.
- If you want to achieve "message queuing" among consumers, share the same subscription name among multiple consumers(shared, failover, key_shared).
- If you want to achieve both effects simultaneously, combine exclusive subscription types with other subscription types for consumers.

Subscription types

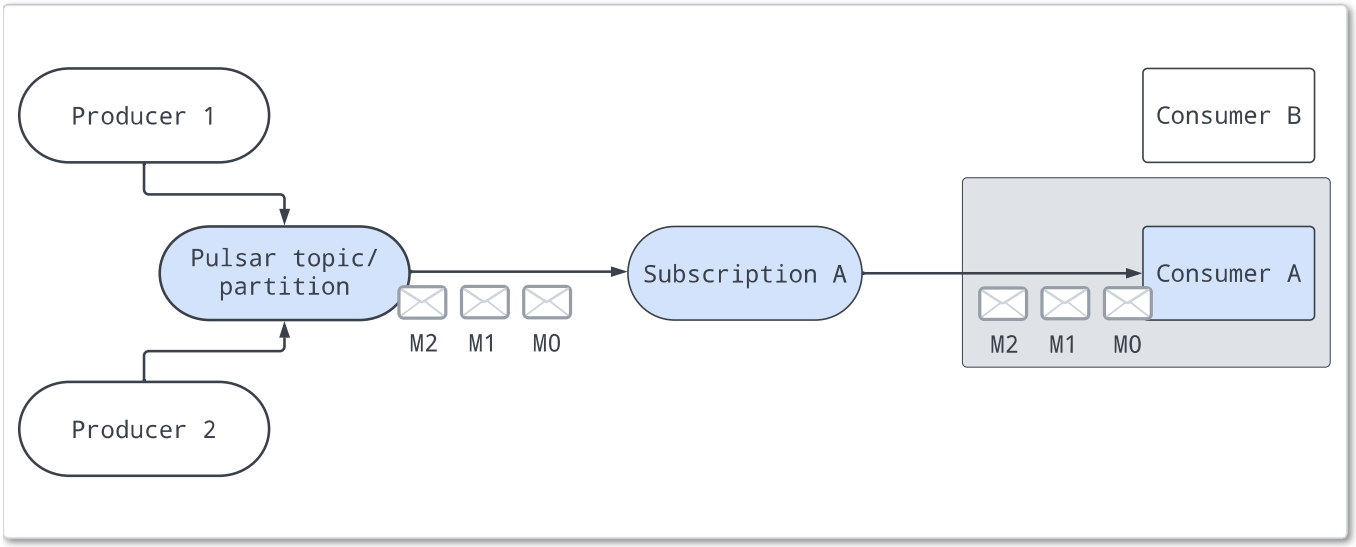
When a subscription has no consumers, its subscription type is undefined. The type of a subscription is defined when a consumer connects to it, and the type can be changed by restarting all consumers with a different configuration.

Exclusive

In the *Exclusive* type, only a single consumer is allowed to attach to the subscription. If multiple consumers subscribe to a topic using the same subscription, an error occurs. Note that if the topic is partitioned, all partitions will be consumed by the single consumer allowed to be connected to the subscription.

In the diagram below, only **Consumer A** is allowed to consume messages.

> **TIP**
>
> Exclusive is the default subscription type.



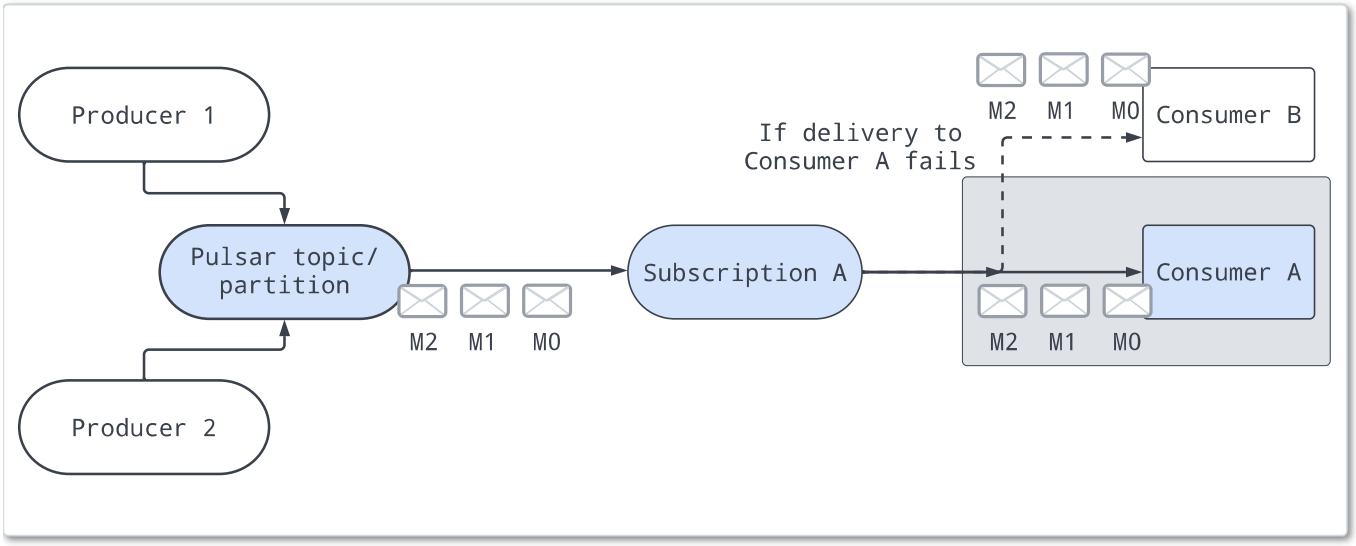
Failover

In the *Failover* type, multiple consumers can attach to the same subscription. A master consumer is picked for a non-partitioned topic or each partition of a partitioned topic and receives messages. When the master consumer disconnects, all (non-acknowledged and subsequent) messages are delivered to the next consumer in line.

- For partitioned topics, the broker will sort consumers by priority level and lexicographical order of consumer name. The broker will try to evenly assign partitions to consumers with the highest priority level.
- For non-partitioned topics, the broker will pick consumers in the order they subscribe to the non-partitioned topics.

For example, a partitioned topic has 3 partitions, and 15 consumers. Each partition will have 1 active consumer and 4 stand-by consumers.

In the diagram below, **Consumer A** is the master consumer while **Consumer B** would be the next consumer in line to receive messages if **Consumer A** is disconnected.

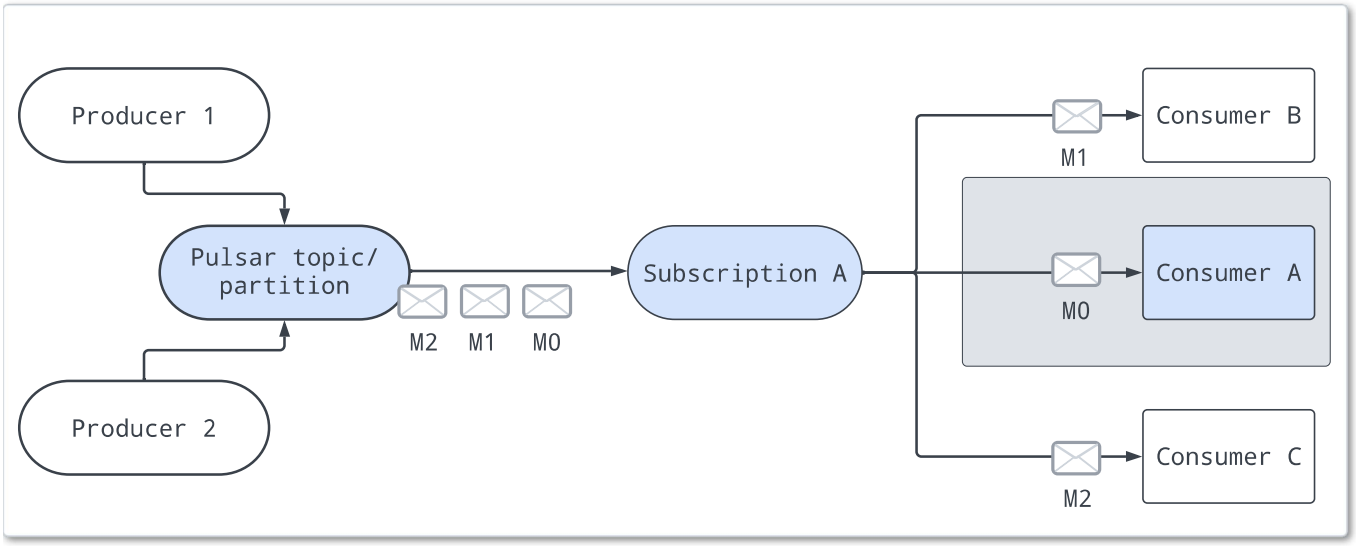


Shared

In *shared* or *round robin* type, multiple consumers can attach to the same subscription. Messages are delivered in a round-robin distribution across consumers, and any given message is delivered to only one consumer. When a consumer disconnects, all the messages that were sent to it and not acknowledged will be rescheduled for sending to the remaining consumers.

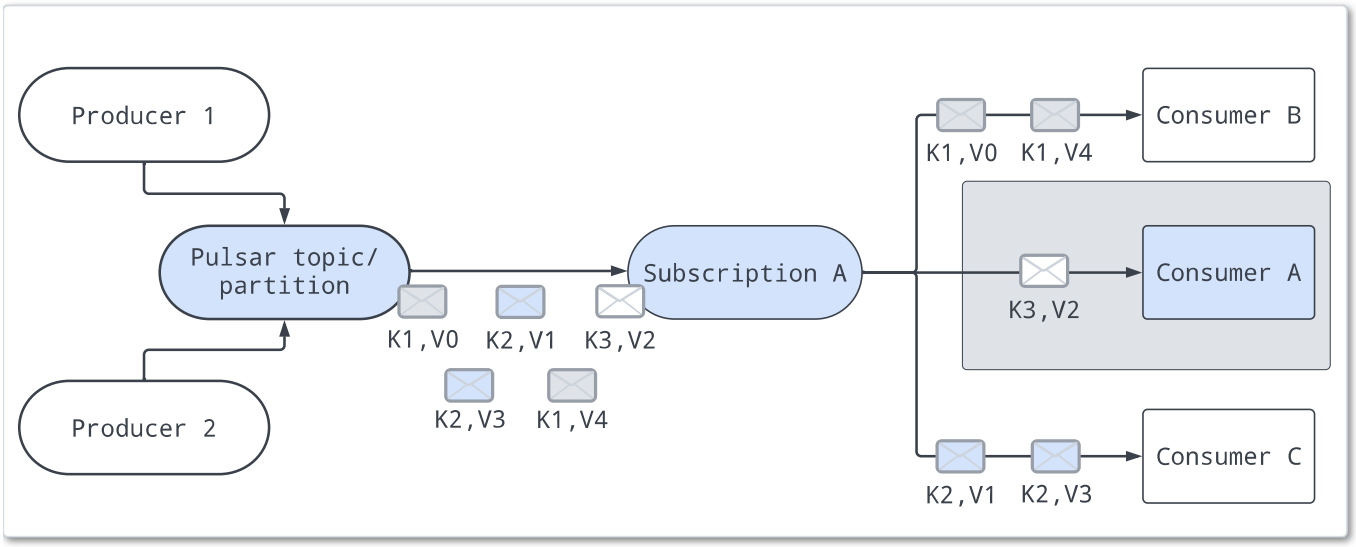
In the diagram below, **Consumer A**, **Consumer B** and **Consumer C** are all able to subscribe to the topic.

> **NOTE**
>
> **Limitations of Shared type** When using Shared type, be aware that:
>
> - Message ordering is not guaranteed.
> - You cannot use cumulative acknowledgment with Shared type.



Key_Shared

In the *Key_Shared* type, multiple consumers can attach to the same subscription. Messages are delivered in distribution across consumers and messages with the same key or same ordering key are delivered to only one consumer. No matter how many times the message is re-delivered, it is delivered to the same consumer.



There are three types of mapping algorithms dictating how to select a consumer for a given message key (or ordering key): Sticky, Auto-split Hash Range, and Auto-split Consistent Hashing. The steps for all algorithms are:

1. The message key (or ordering key) is passed to a hash function (e.g., Murmur3 32-bit), yielding a 32-bit integer hash.
2. That hash number is fed to the algorithm to select a consumer from the existing connected consumers.

```
1                    +--------------+                    +-----------
+
```

```
Message Key ----->  / Hash Function / ----- hash (32-bit) -------> / Algorithm / ----> Consumer
                  +---------------+                              +-----------+
```

When a new consumer is connected and thus added to the list of connected consumers, the algorithm re-adjusts the mapping such that some keys currently mapped to existing consumers will be mapped to the newly added consumer. When a consumer is disconnected, thus removed from the list of connected consumers, keys mapped to it will be mapped to other consumers. The sections below will explain how a consumer is selected given the message hash and how the mapping is adjusted given a new consumer is connected or an existing consumer disconnects for each algorithm.

Auto-split Hash Range

The algorithm assumes there is a range of numbers between 0 to 2^16 (65,536). Each consumer is mapped into a single region in this range, so all mapped regions cover the entire range, and no regions overlap. A consumer is selected for a given key by running a modulo operation on the message hash by the range size (65,536). The number received (0 <= i < 65,536) is contained within a single region. The consumer mapped to that region is the one selected.

Example:

Suppose we have 4 consumers (C1, C2, C3 and C4), then:

```
0              16,384           32,768          49,152            65,536
|------- C3 ------|------- C2 ------|------- C1 ------|------- C4 ------|
```

Given a message key `Order-3459134`, its hash would be `murmur32("Order-3459134") = 3112179635`, and its index in the range would be `3112179635 mod 65536 = 6067`. That index is contained within region `[0, 16384)` thus consumer C3 will be mapped to this message key.

When a new consumer is connected, the largest region is chosen and is then split in half - the lower half will be mapped to the newly added consumer and upper half will be mapped to the consumer owning that region. Here is how it looks like from 1 to 4 consumers:

```
C1 connected:
|-------------------------------- C1 --------------------------------|

C2 connected:
|--------------- C2 ----------------|---------------- C1 ---------------|

C3 connected:
|------- C3 ------|------- C2 ------|---------------- C1 ---------------|

C4 connected:
|------- C3 ------|------- C2 ------|------- C4 ------|------- C1 ------|
```

When a consumer is disconnected its region will be merged into the region on its right. Examples:

C4 is disconnected:

```
|------- C3 ------|------- C2 ------|---------------- C1 ---------------|
```

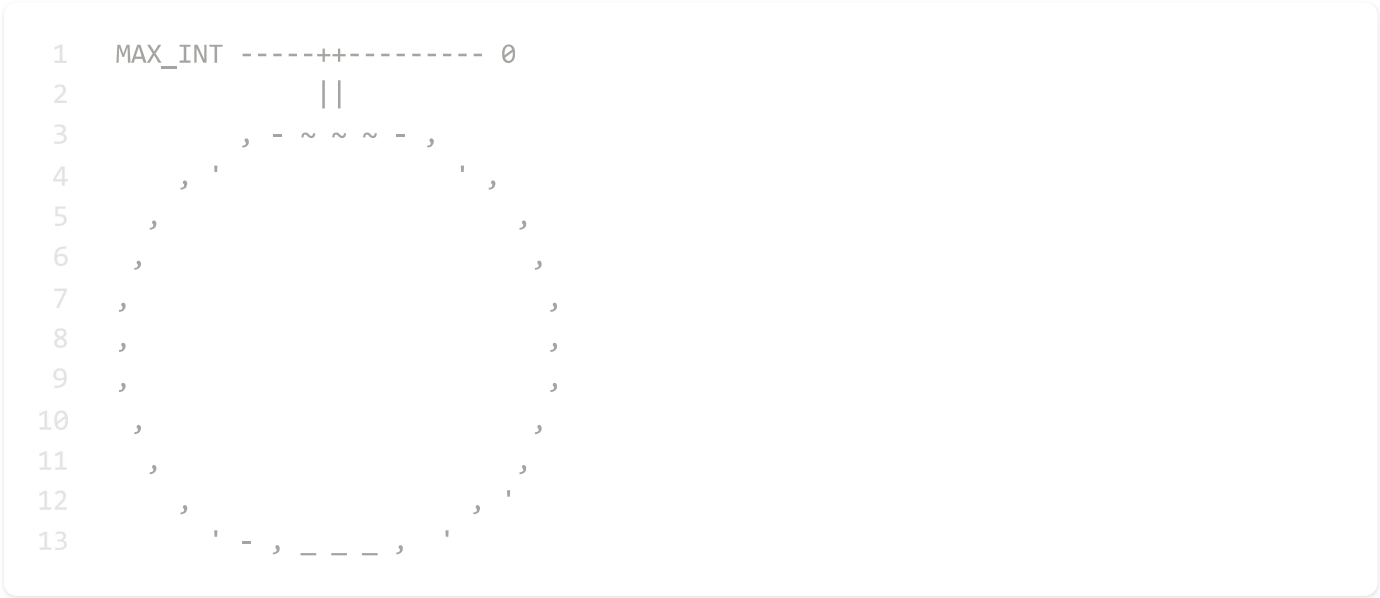
C1 is disconnected:

```
|------- C3 ------|------------------------- C2 ----------------------|
```

The advantages of this algorithm is that it affects only a single existing consumer upon add/delete consumer, at the expense of regions not evenly sized. Thi means some consumers gets more keys that others. The next algorithm does the other way around.

Auto-split Consistent Hashing

This algorithm uses a Hash Ring. It's a range of number from 0 to MAX_INT (32-bit) in which if you traverse the range, when reaching MAX_INT, the next number would be zero. It is as if you took a line starting from 0 ending at MAX_INT and bent into a circle such that the end glues to the start:

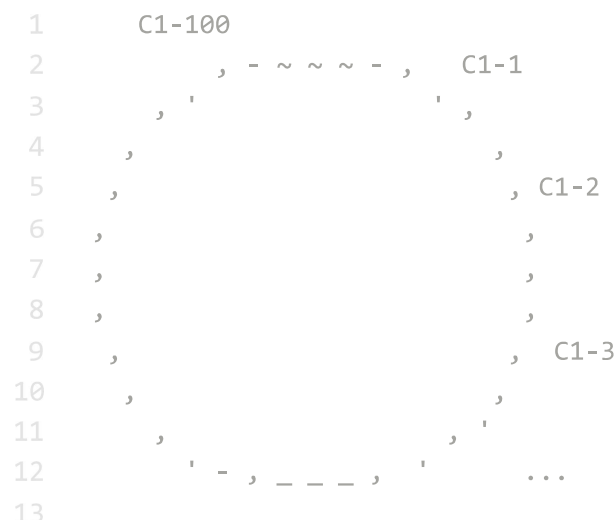


When adding a consumer, we mark 100 points on that circle and associate them to the newly added consumer. For each number between 1 and 100, we concatenate the consumer name to that number and run the hash function on it to get the location of the point on the circle that will be marked. For Example, if the consumer name is "orders-aggregator-pod-2345-consumer" then we would mark 100 points on that circle:

```
    murmur32("orders-aggregator-pod-2345-consumer1") = 1003084738
    murmur32("orders-aggregator-pod-2345-consumer2") = 373317202
```

```
    ...
    murmur32("orders-aggregator-pod-2345-consumer100") = 320276078
```

Since the hash function has the uniform distribution attribute, those points would be uniformly distributed across the circle.



A consumer is selected for a given message key by putting its hash on the circle then continue clock-wise on the circle until you reach a marking point. The point might have more than one consumer on it (hash function might have collisions) there for, we run the following operation to get a position within the list of consumers for that point, then we take the consumer in that position: `hash % consumer_list_size = index`.

When a consumer is added, we add 100 marking points to the circle as explained before. Due to the uniform distribution of the hash function, those 100 points act as if the new consumer takes a small slice of keys out of each existing consumer. It maintains the even distribution, on the trade-off that it impacts all existing consumers. This video explains the concept of Consistent Hashing quite well (the only difference is that in Pulsar's case we used K points instead of K hash functions as noted in the comments)

Sticky

The algorithm assumes there is a range of numbers between 0 to 2^16 (65,536). Each consumer is mapped to a multiple regions in this range and there is no overlap between regions. The consumer is selected by running a modulo operation on the message hash by the range size (65,536), the number received (0 <= i < 65,536), is contained within a single region. The consumer mapped to the region is the one selected. In this algorithm you have full control. Every newly added consumer specifies the ranges it wishes to be mapped to by using Consumer API. When the consumer object is constructed, you can specify the list of ranges. It's your responsibility to make sure there are no overlaps and all the range is covered by regions.

Example:

Suppose we have 2 consumers (C1 and C2) each specified their ranges, then:

```
C1 = [0, 16384), (32768, 49152]
C2 = [16384, 32768), (49,152, 65536]

 0              16,384           32,768          49,152            65,536
 |------- C1 ------|------- C2 ------|------- C1 ------|------- C2 ------|
```

Given a message key `Order-3459134`, it's hash would be `murmur32("Order-3459134") = 3112179635`, and it's index in the range would be `3112179635 mod 65536 = 6067`. That index is contained within `[0, 16384)` thus consumer C1 will map to this message key.

If the newly connected consumer didn't supply their ranges, or they overlap with existing consumer ranges, it's disconnected, removed from the consumers list and reverted as if it never tried to connect.

How to use them?

When building the consumer, you can specify the Key Shared Mode:

- AUTO_SPLIT - Auto-split Hash Range
- STICKY - Sticky

Consistent Hashing will be used instead of Hash Range for Auto-split if the broker configuration `subscriptionKeySharedUseConsistentHashing` is enabled.

Preserving order of processing

Key Shared Subscription type guarantees a key will be processed by a *single* consumer at any given time. When a new consumer is connected, some keys will change their mapping from existing consumers to the new consumer. Once the connection has been established, the broker will record the current read position and associate it with the new consumer. The read position is a marker indicating that messages have been dispatched to the consumers up to this point, and after it, no messages have been dispatched yet. The broker will start delivering messages to the new consumer *only* when all messages up to the read position have been acknowledged. This will guarantee that a certain key is processed by a single consumer at any given time. The trade-off is that if one of the existing consumers is stuck and no time-out was defined (acknowledging for you), the new consumer won't receive any messages until the stuck consumer resumes or gets disconnected.

That requirement can be relaxed by enabling `allowOutOfOrderDelivery` via the Consumer API. If set on the new consumer, then when it is connected, the broker will allow it to receive messages knowing some messages of that key may be still be processing in other consumers at the time, thus order may be affected for that short period of adding a new consumer.

Batching for Key Shared Subscriptions

> **NOTE**

When the consumers are using the Key_Shared subscription type, you need to **disable batching** or **use key-based batching** for the producers.

There are two reasons why the key-based batching is necessary for the Key_Shared subscription type:

1. The broker dispatches messages according to the keys of the messages, but the default batching approach might fail to pack the messages with the same key to the same batch.
2. Since it is the consumers instead of the broker who dispatch the messages from the batches, the key of the first message in one batch is considered as the key to all messages in this batch, thereby leading to context errors.

The key-based batching aims at resolving the above-mentioned issues. This batching method ensures that the producers pack the messages with the same key to the same batch. The messages without a key are packed into one batch and this batch has no key. When the broker dispatches messages from this batch, it uses `NON_KEY` as the key. In addition, each consumer is associated with **only one** key and should receive **only one message batch** for the connected key. By default, you can limit batching by configuring the number of messages that producers are allowed to send.

Below are examples of enabling the key-based batching under the Key_Shared subscription type, with `client` being the Pulsar client that you created.

Java **C++** **Python**

```
Producer<byte[]> producer = client.newProducer()
        .topic("my-topic")
        .batcherBuilder(BatcherBuilder.KEY_BASED)
        .create();
```

NOTE

Limitations of Key_Shared type

When you use Key_Shared type, be aware that:

- You need to specify a key or ordering key for messages.
- You cannot use cumulative acknowledgment with Key_Shared type.
- When the position of the newest message in a topic is `X`, a key-shared consumer that is newly attached to the same subscription and connected to the topic will **not** receive any messages until all the messages before `X` have been acknowledged.

Subscription modes

What is a subscription mode

The subscription mode indicates the cursor type.

- When a subscription is created, an associated cursor is created to record the last consumed position.
- When a consumer of the subscription restarts, it can continue consuming from the last message it consumes.

Subscription mode	Description	Note
`Durable`	The cursor is durable, which retains messages and persists the current position. If a broker restarts from a failure, it can recover the cursor from the persistent storage (BookKeeper), so that messages can continue to be consumed from the last consumed position.	`Durable` is the **default** subscription mode.
`NonDurable`	The cursor is non-durable. Once a broker stops, the cursor is lost and can never be recovered, so that messages **can not** continue to be consumed from the last consumed position.	Reader's subscription mode is `NonDurable` in nature and it does not prevent data in a topic from being deleted. Reader's subscription mode **can not** be changed.

A subscription can have one or more consumers. When a consumer subscribes to a topic, it must specify the subscription name. A durable subscription and a non-durable subscription can have the same name, they are independent of each other. If a consumer specifies a subscription that does not exist before, the subscription is automatically created.

When to use

By default, messages of a topic without any durable subscriptions are marked as deleted. If you want to prevent the messages from being marked as deleted, you can create a durable subscription for this topic. In this case, only acknowledged messages are marked as deleted. For more information, see message retention and expiry.

How to use

After a consumer is created, the default subscription mode of the consumer is `Durable`. You can change the subscription mode to `NonDurable` by making changes to the consumer's configuration.

Durable **Non-durable**

```
Consumer<byte[]> consumer = pulsarClient.newConsumer()
        .topic("my-topic")
        .subscriptionName("my-sub")
        .subscriptionMode(SubscriptionMode.Durable)
        .subscribe();
```

For how to create, check, or delete a durable subscription, see manage subscriptions.

Multi-topic subscriptions

When a consumer subscribes to a Pulsar topic, by default it subscribes to one specific topic, such as `persistent://public/default/my-topic`. As of Pulsar version 1.23.0-incubating, however, Pulsar consumers can simultaneously subscribe to multiple topics. You can define a list of topics in two ways:

- On the basis of a **reg**ular **ex**pression (regex), for example, `persistent://public/default/finance-.*`
- By explicitly defining a list of topics

> **NOTE**
>
> When subscribing to multiple topics by regex, all topics must be in the same namespace.

When subscribing to multiple topics, the Pulsar client automatically makes a call to the Pulsar API to discover the topics that match the regex pattern/list, and then subscribe to all of them. If any of the topics do not exist, the consumer auto-subscribes to them once the topics are created.

> **NOTE**
>
> **No ordering guarantees across multiple topics** When a producer sends messages to a single topic, all messages are guaranteed to be read from that topic in the same order. However, these guarantees do not hold across multiple topics. So when a producer sends messages to multiple topics, the order in which messages are read from those topics is not guaranteed to be the same.

The following are multi-topic subscription examples for Java.

```java
import java.util.regex.Pattern;

import org.apache.pulsar.client.api.Consumer;
import org.apache.pulsar.client.api.PulsarClient;

PulsarClient pulsarClient = // Instantiate Pulsar client object

// Subscribe to all topics in a namespace
Pattern allTopicsInNamespace =
Pattern.compile("persistent://public/default/.*");
Consumer<byte[]> allTopicsConsumer = pulsarClient.newConsumer()
                .topicsPattern(allTopicsInNamespace)
                .subscriptionName("subscription-1")
                .subscribe();

// Subscribe to a subsets of topics in a namespace, based on regex
Pattern someTopicsInNamespace =
Pattern.compile("persistent://public/default/foo.*");
Consumer<byte[]> someTopicsConsumer = pulsarClient.newConsumer()
                .topicsPattern(someTopicsInNamespace)
                .subscriptionName("subscription-1")
                .subscribe();
```

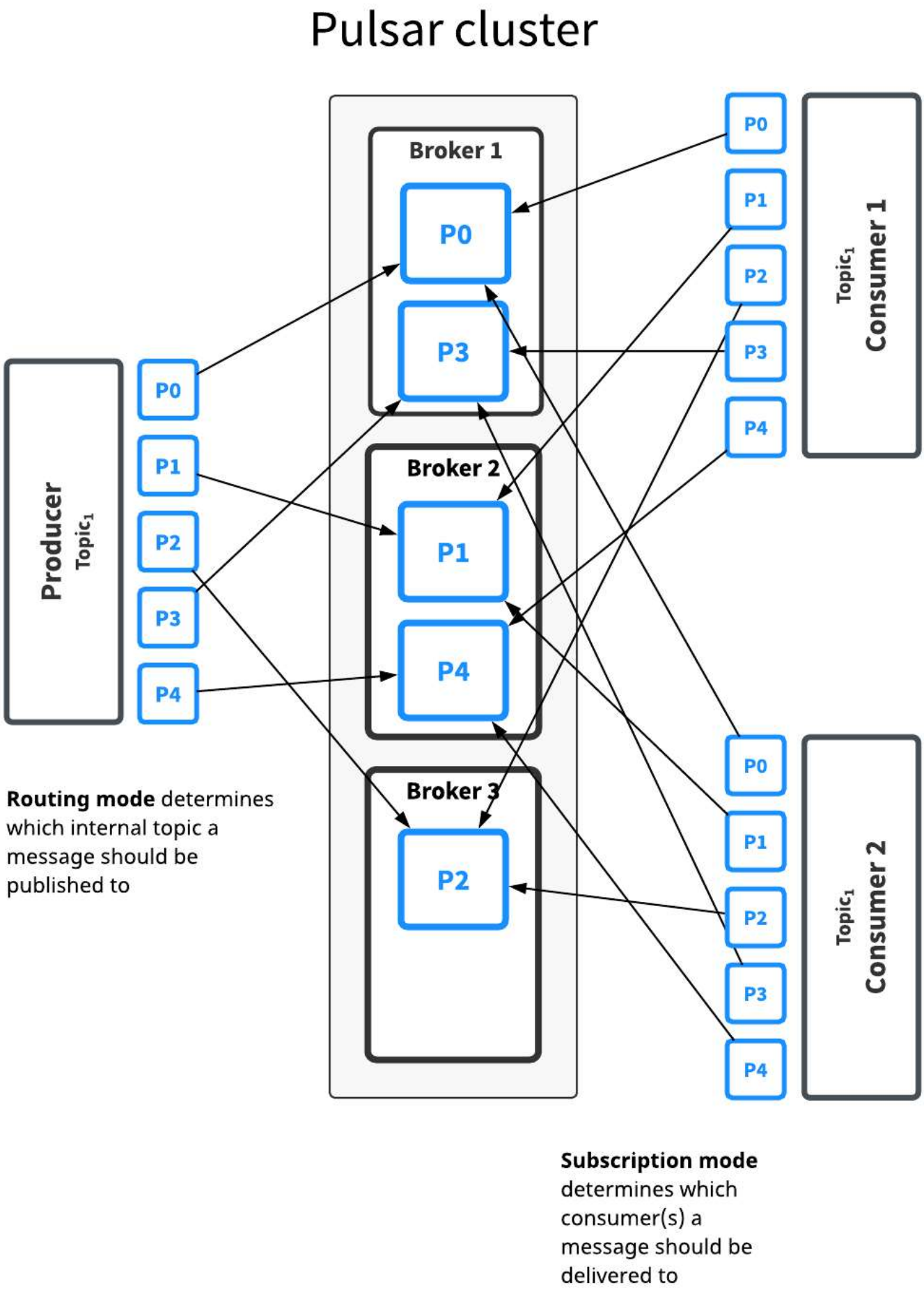
For code examples, see Java.

Partitioned topics

Normal topics are served only by a single broker, which limits the maximum throughput of the topic. *Partitioned topics* are a special type of topic handled by multiple brokers, thus allowing for higher throughput.

A partitioned topic is implemented as N internal topics, where N is the number of partitions. When publishing messages to a partitioned topic, each message is routed to one of several brokers. The distribution of partitions across brokers is handled automatically by Pulsar.

The diagram below illustrates this:



The **Topic1** topic has five partitions (**P0** through **P4**) split across three brokers. Because there are more partitions than brokers, two brokers handle two partitions a piece, while the third handles only one (again, Pulsar handles this distribution of partitions automatically).

Messages for this topic are broadcast to two consumers. The routing mode determines each message should be published to which partition, while the subscription type determines which messages go to which consumers.

Decisions about routing and subscription modes can be made separately in most cases. In general, throughput concerns should guide partitioning/routing decisions while subscription decisions should be guided by application semantics.

There is no difference between partitioned topics and normal topics in terms of how subscription types work, as partitioning only determines what happens between when a message is published by a producer and processed and acknowledged by a consumer.

Partitioned topics need to be explicitly created via the admin API. The number of partitions can be specified when creating the topic.

Routing modes

When publishing to partitioned topics, you must specify a *routing mode*. The routing mode determines which partition---that is, which internal topic---each message should be published to.

There are three MessageRoutingMode available:

Mode	Description
`RoundRobinPartition`	If no key is provided, the producer will publish messages across all partitions in round-robin fashion to achieve maximum throughput. Please note that round-robin is not done per individual message but rather it's set to the same boundary of batching delay, to ensure batching is effective. While if a key is specified on the message, the partitioned producer will hash the key and assign message to a particular partition. This is the default mode.
`SinglePartition`	If no key is provided, the producer will randomly pick one single partition and publish all the messages into that partition. While if a key is specified on the message, the partitioned producer will hash the key and assign message to a particular partition.
`CustomPartition`	Use custom message router implementation that will be called to determine the partition for a particular message. User can create a custom routing mode by using the Java client and implementing the MessageRouter interface.

Ordering guarantee

The ordering of messages is related to MessageRoutingMode and Message Key. Usually, user would want an ordering of Per-key-partition guarantee.

If there is a key attached to message, the messages will be routed to corresponding partitions based on the hashing scheme specified by HashingScheme in ProducerBuilder, when using

either `SinglePartition` or `RoundRobinPartition` mode.

Ordering guarantee	Description	Routing Mode and Key
Per-key-partition	All the messages with the same key will be in order and be placed in same partition.	Use either `SinglePartition` or `RoundRobinPartition` mode, and Key is provided by each message.
Per-producer	All the messages from the same producer will be in order.	Use `SinglePartition` mode, and no Key is provided for each message.

Hashing scheme

HashingScheme is an enum that represents sets of standard hashing functions available when choosing the partition to use for a particular message.

There are 2 types of standard hashing functions available: `JavaStringHash` and `Murmur3_32Hash`. The default hashing function for producers is `JavaStringHash`. Please pay attention that `JavaStringHash` is not useful when producers can be from different multiple language clients, under this use case, it is recommended to use `Murmur3_32Hash`.

Non-persistent topics

By default, Pulsar persistently stores *all* unacknowledged messages on multiple BookKeeper bookies (storage nodes). Data for messages on persistent topics can thus survive broker restarts and subscriber failover.

Pulsar also, however, supports **non-persistent topics**, which are topics on which messages are *never* persisted to disk and live only in memory. When using non-persistent delivery, killing a Pulsar broker or disconnecting a subscriber to a topic means that all in-transit messages are lost on that (non-persistent) topic, meaning that clients may see message loss.

Non-persistent topics have names of this form (note the `non-persistent` in the name):

```
non-persistent://tenant/namespace/topic
```

For more info on using non-persistent topics, see the Non-persistent messaging cookbook.

In non-persistent topics, brokers immediately deliver messages to all connected subscribers *without persisting them* in BookKeeper. If a subscriber is disconnected, the broker will not be able to deliver those in-transit messages, and subscribers will never be able to receive those messages again. Eliminating the persistent storage step makes messaging on non-persistent topics slightly faster than on persistent topics in some cases, but with the caveat that some core benefits of Pulsar are lost.

> With non-persistent topics, message data lives only in memory, without a specific buffer - which means data *is not* buffered in memory. The received messages are immediately transmitted to all *connected consumers*. If a message broker fails or message data can otherwise not be retrieved from memory, your message data may be lost. Use non-persistent topics only if you're *certain* that your use case requires it and can sustain it.

By default, non-persistent topics are enabled on Pulsar brokers. You can disable them in the broker's configuration. You can manage non-persistent topics using the `pulsar-admin topics` command. For more information, see `pulsar-admin`.

Currently, non-persistent topics which are not partitioned are not persisted to ZooKeeper, which means if the broker owning them crashes, they do not get re-assigned to another broker because they only exist in the owner broker memory. The current workaround is to set the value of `allowAutoTopicCreation` to `true` and `allowAutoTopicCreationType` to `non-partitioned` (they are default values) in broker configuration.

Performance

Non-persistent messaging is usually faster than persistent messaging because brokers don't persist messages and immediately send acks back to the producer as soon as that message is delivered to connected brokers. Producers thus see comparatively low publish latency with non-persistent topic.

Client API

Producers and consumers can connect to non-persistent topics in the same way as persistent topics, with the crucial difference that the topic name must start with `non-persistent`. All the subscription types---exclusive, shared, key-shared and failover---are supported for non-persistent topics.

Here's an example Java consumer for a non-persistent topic:

```
PulsarClient client = PulsarClient.builder()
        .serviceUrl("pulsar://localhost:6650")
        .build();
String npTopic = "non-persistent://public/default/my-topic";
String subscriptionName = "my-subscription-name";

Consumer<byte[]> consumer = client.newConsumer()
        .topic(npTopic)
        .subscriptionName(subscriptionName)
        .subscribe();
```

Here's an example Java producer for the same non-persistent topic:

```
Producer<byte[]> producer = client.newProducer()
                .topic(npTopic)
```

```
3                   .create();
```

System topic

System topic is a predefined topic for internal use within Pulsar. It can be either a persistent or non-persistent topic.

System topics serve to implement certain features and eliminate dependencies on third-party components, such as transactions, heartbeat detections, topic-level policies, and resource group services. System topics empower the implementation of these features to be simplified, dependent, and flexible. Take heartbeat detections for example, you can leverage the system topic for health check to internally enable producer/reader to produce/consume messages under the heartbeat namespace, which can detect whether the current service is still alive.

The following table outlines the available system topics for each specific namespace.

Namespace	TopicName	Domain	Count	Usage
pulsar/system	`transaction_coordinator_assign_${id}`	Persistent	Default 16	Transaction coordinator
pulsar/system	`__transaction_log_${tc_id}`	Persistent	Default 16	Transaction log
pulsar/system	`resource-usage`	Non-persistent	Default 4	Resource group service
host/port	`heartbeat`	Persistent	1	Heartbeat detection
User-defined-ns	`__change_events`	Persistent	Default 4	Topic events
User-defined-ns	`__transaction_buffer_snapshot`	Persistent	One per namespace	Transaction buffer snapshots
User-defined-ns	`${topicName}__transaction_pending_ack`	Persistent	One per every topic subscription acknowledged with transactions	Acknowledgments with transactions

> **NOTE**

- You cannot create any system topics. To list system topics, you can add the option `--include-system-topic` when you get the topic list by using Pulsar admin API.
- Since Pulsar version 2.11.0, system topics are enabled by default. In earlier versions, you need to change the following configurations in the `conf/broker.conf` or `conf/standalone.conf` file to enable system topics.

```
systemTopicEnabled=true
topicLevelPoliciesEnabled=true
```

Message redelivery

Apache Pulsar supports graceful failure handling and ensures critical data is not lost. Software will always have unexpected conditions and at times messages may not be delivered successfully. Therefore, it is important to have a built-in mechanism that handles failure, particularly in asynchronous messaging as highlighted in the following examples.

- Consumers get disconnected from the database or the HTTP server. When this happens, the database is temporarily offline while the consumer is writing the data to it and the external HTTP server that the consumer calls are momentarily unavailable.
- Consumers get disconnected from a broker due to consumer crashes, broken connections, etc. As a consequence, unacknowledged messages are delivered to other available consumers.

Apache Pulsar avoids these and other message delivery failures using at-least-once delivery semantics that ensure Pulsar processes a message more than once.

To utilize message redelivery, you need to enable this mechanism before the broker can resend the unacknowledged messages in Apache Pulsar client. You can activate the message redelivery mechanism in Apache Pulsar using three methods.

- Negative Acknowledgment
- Acknowledgment Timeout
- Retry letter topic

Message retention and expiry

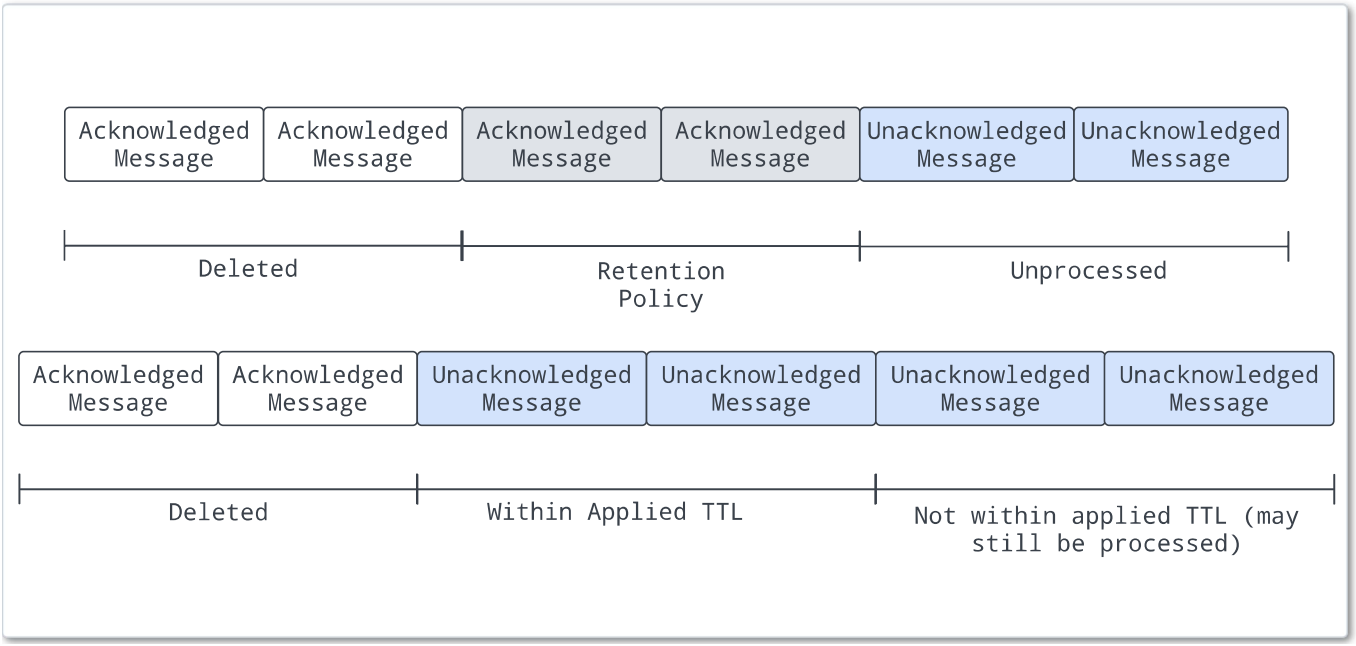
By default, Pulsar message brokers:

- immediately delete *all* messages that have been acknowledged by a consumer, and
- persistently store all unacknowledged messages in a message backlog.

Pulsar has two features, however, that enable you to override this default behavior:

- Message **retention** enables you to store messages that have been acknowledged by a consumer
- Message **expiry** enables you to set a time to live (TTL) for messages that have not yet been acknowledged

> **TIP**
>
> All message retention and expiry are managed at the namespace level. For a how-to, see the Message retention and expiry cookbook.



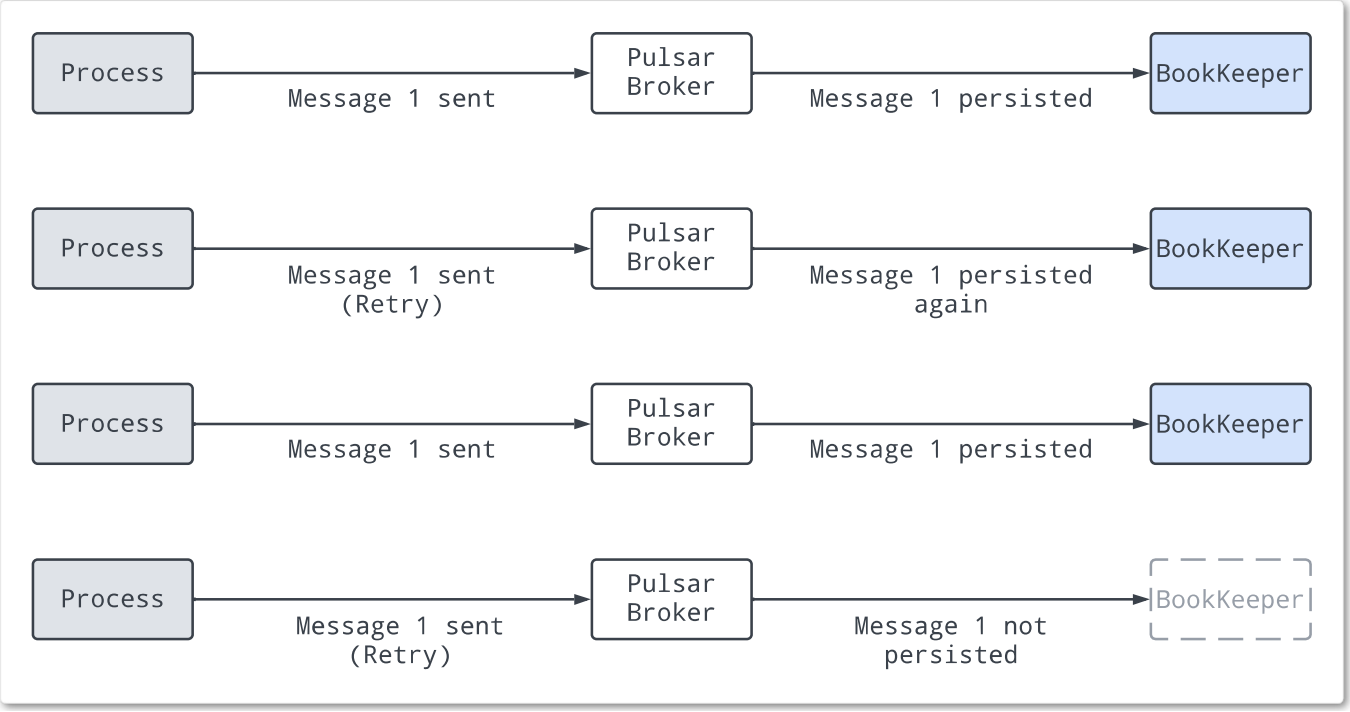
With message retention, shown at the top, a retention policy applied to all topics in a namespace dictates that some messages are durably stored in Pulsar even though they've already been acknowledged. Acknowledged messages that are not covered by the retention policy are deleted. Without a retention policy, all of the acknowledged messages would be deleted.

With message expiry, shown at the bottom, some messages are deleted, even though they haven't been acknowledged, because they've expired according to the TTL applied to the namespace (for example because a TTL of 5 minutes has been applied and the messages haven't been acknowledged but are 10 minutes old).

Message deduplication

Message duplication occurs when a message is persisted by Pulsar more than once. Message deduplication is an optional Pulsar feature that prevents unnecessary message duplication by processing each message only once, even if the message is received more than once.

The following diagram illustrates what happens when message deduplication is disabled vs. enabled:



Message deduplication is disabled in the scenario shown at the top. Here, a producer publishes message 1 on a topic; the message reaches a Pulsar broker and is persisted to BookKeeper. The producer then sends message 1 again (in this case due to some retry logic), and the message is received by the broker and stored in BookKeeper again, which means that duplication has occurred.

In the second scenario at the bottom, the producer publishes message 1, which is received by the broker and persisted, as in the first scenario. When the producer attempts to publish the message again, however, the broker knows that it has already seen message 1 and thus does not persist the message.

> Message deduplication is handled at the namespace level or the topic level. For more instructions, see the message deduplication cookbook. You can read the design of Message Deduplication in PIP-6.

Producer idempotency

The other available approach to message deduplication is to ensure that each message is *only produced once*. This approach is typically called **producer idempotency**. The drawback of this approach is that it defers the work of message deduplication to the application. In Pulsar, this is handled at the broker level, so you do not need to modify your Pulsar client code. Instead, you only need to make administrative changes. For details, see Managing message deduplication.

Deduplication and effectively-once semantics

Message deduplication makes Pulsar an ideal messaging system to be used in conjunction with stream processing engines (SPEs) and other systems seeking to provide effectively-once processing semantics. Messaging systems that do not offer automatic message deduplication require the SPE or other system to guarantee deduplication, which means that strict message

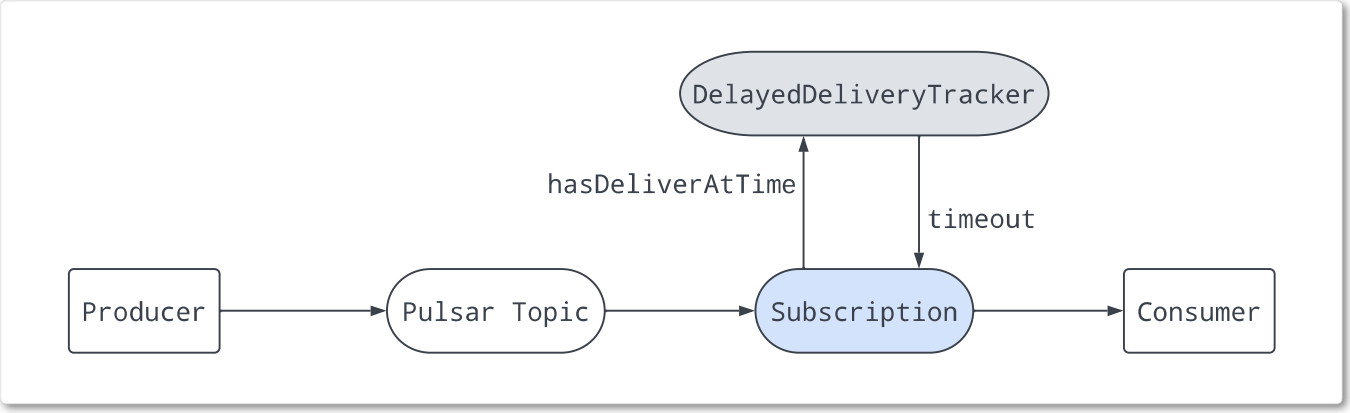
ordering comes at the cost of burdening the application with the responsibility of deduplication. With Pulsar, strict ordering guarantees come at no application-level cost.

Delayed message delivery

Delayed message delivery enables you to consume a message later. In this mechanism, a message is stored in BookKeeper. The `DelayedDeliveryTracker` maintains the time index (time -> messageId) in memory after the message is published to a broker. This message will be delivered to a consumer once the specified delay is over.

Delayed message delivery only works in the Shared subscription type. In the Exclusive and Failover subscription types, the delayed message is dispatched immediately.

The diagram below illustrates the concept of delayed message delivery:



A broker saves a message without any check. When a consumer consumes a message, if the message is set to delay, then the message is added to `DelayedDeliveryTracker`. A subscription checks and gets timeout messages from `DelayedDeliveryTracker`.

Broker

Delayed message delivery is enabled by default. You can change it in the broker configuration file as below:

```
# Whether to enable the delayed delivery for messages.
# If disabled, messages are immediately delivered and there is no tracking overhead.
delayedDeliveryEnabled=true

# Control the ticking time for the retry of delayed message delivery,
# affecting the accuracy of the delivery time compared to the scheduled time.
# Note that this time is used to configure the HashedWheelTimer's tick time for the
# InMemoryDelayedDeliveryTrackerFactory (the default DelayedDeliverTrackerFactory).
# Default is 1 second.
delayedDeliveryTickTimeMillis=1000
```

```
11
12  # When using the InMemoryDelayedDeliveryTrackerFactory (the default
DelayedDeliverTrackerFactory), whether
13  # the deliverAt time is strictly followed. When false (default), messages may be
sent to consumers before the deliverAt
14  # time by as much as the tickTimeMillis. This can reduce the overhead on the
broker of maintaining the delayed index
15  # for a potentially very short time period. When true, messages will not be sent
to consumer until the deliverAt time
16  # has passed, and they may be as late as the deliverAt time plus the
tickTimeMillis for the topic plus the
17  # delayedDeliveryTickTimeMillis.
18  isDelayedDeliveryDeliverAtTimeStrict=false
```

Producer

The following is an example of delayed message delivery for a producer in Java:

```
1  // message to be delivered at the configured delay interval
2  producer.newMessage().deliverAfter(3L, TimeUnit.Minute).value("Hello
Pulsar!").send();
```

Edit this page