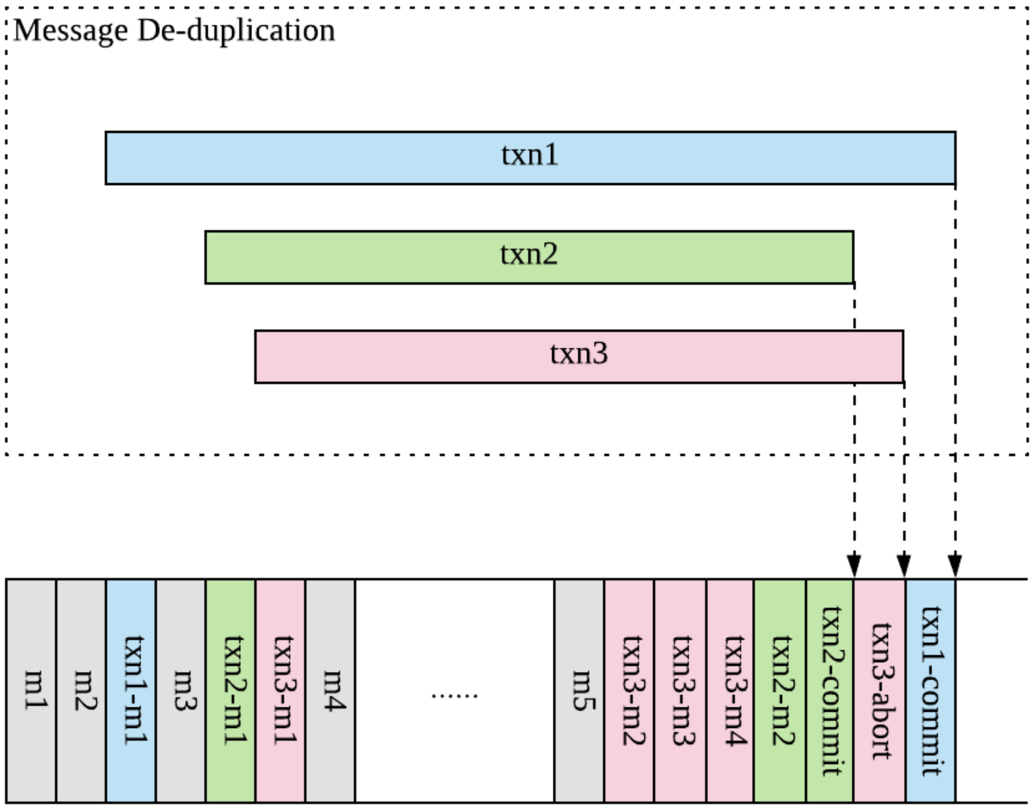
Background knowledge

As early as the PIP-6, we implemented exactly-once delivery for single messages from producers. We introduced the concepts of producerName and sequenceId, where the producerName is globally unique and the sequenceId defaults to start from 0, incrementing for each message. Of course, we also allow users to set the producerName and sequenceId themselves. When the message deduplication feature is enabled, the broker filters out duplicate messages based on the sequenceId and producerName. Moreover, since Pulsar's message resending mechanism always sends to the same partition, even for multi-partition topics, single-message delivery is exactly-once.

However, this mechanism does not guarantee atomicity for sending multiple messages. To address this issue, we introduced transactions in the PIP-31 proposal. Before diving deep into this proposal, it is highly recommended to read PIP-6 and PIP-31 better to understand the design of idempotent producers and Pulsar Transaction.

In the design of Pulsar transactions, messages sent using a transaction will carry the transaction ID of that transaction. When a transaction is committed or aborted, we write a commit or abort marker into all topics the transaction has sent messages to. Importantly, a transaction's messages do not become immediately visible to consumers upon its commitment. We can determine if a message is a transactional message by checking whether it carries a transaction ID. Prior to the marker being written, all messages following the transactional message are invisible to consumers. For example, in a given scenario, before the txn1-commit marker is written, all messages after txn1-m1 are unreadable. Even if txn2-commit and txn3-abort markers have been written to the topic, all messages from txn2 and txn3 related to this topic remain invisible to consumers.



Motivation

In some use cases, we may face challenges. Imagine a scenario where we need to start a task to process messages. The task's logic is similar to opening a transaction, using it to process some messages, and then committing the transaction. However, due to some failures, we lost contact with the task and had to restart another task to work. In this situation, there will be two tasks processing the same job. If the previously disconnected task1 has already terminated, it is possible that, as shown in the previous example, the committed messages of txn2 are not visible to consumers due to txn1 not being finished. In this case, we can only wait for txn1 to time out before task 2 can work normally. If task 1 is only disconnected but not terminated, the situation is even worse, as both tasks receive duplicate messages from the same topic and use transactions to cumulatively ack these messages, then commit the transactions. This would result in duplicate message processing. Note that cumulative ack does not have conflict ack, so that duplicate consumption can occur. In the case I mentioned earlier, tasks use cumulative ack, and the acks mentioned below refer to cumulative ack.

Therefore, we hope to immediately fence the previous task's transaction after a task starts a new transaction. That is, the broker should immediately end the transaction started by the previous task and reject all requests coming from that transaction.

Goal

In Scope

- Provide users with a transaction key mechanism to ensure that only one active transaction and one active connection is associated with a given transaction key.
- Abort previous transactions associated with the same transaction key and prevent them from performing further operations.

This will create a clean working environment for the newly started transaction.

Out of Scope

- This proposal does not consider the expansion of transaction coordinators.
- Addressing duplicate messages caused by message reconnection, redelivery, or failover consumers is not covered in this proposal. Separate proposals will be responsible for these issues.

High-Level Design

The principal objective of this proposal is to ensure that only one active connection and transaction are associated with a given transaction key. To achieve this, we propose to maintain a map within the Transaction Coordinator (TC). This map will record the transaction keys managed by the TC, and the corresponding epochs of their connections, as well as the transaction IDs.

When a connection is established, it will bring along the epoch from the last connection. Once the connection is successfully established, the TC will return a new epoch that will be stored on the client side. When a connection is made for the first time, the epoch will be set to -1L.

The TC will only accept the connection from the most recent epoch and will close the old connection and abort the old transaction as soon as a new connection is established.

The transaction key, the connection's epoch, and the transaction ID will be recorded in real-time in a newly added system topic, the `*_transaction_key*`. This serves to persistently track and manage the associations between transaction keys, connection epochs, and transaction IDs, ensuring that each transaction key is associated with only one newest active connection and transaction at a time.

Detailed Design

Design & Implementation Details

Key Concepts

- **Transaction key:** The transaction key consists of an owner identifier and a user-defined key. Users provide this key when creating a PulsarClient. For a given owner and transaction key, there can only be one active transaction at a time. After hashing the transaction key, it is bound to a transaction coordinator, which is responsible for handling all transaction operations for this transaction key.

- **Connection epoch:** Every transaction key is associated with a unique, incrementing long integer, initialized at -1, serving as the epoch. When a client initiates a connection with a specific transaction key, the transaction coordinator responsible for managing this key increments the epoch and communicates the updated value back to the client. At any given time, only one active connection is allowed for each transaction key. If a new connection is established, the system terminates the preceding connection associated with the earlier epoch and aborts the ongoing transaction linked to that transaction key.

- **System Topic '__transaction_key_':** Each Transaction Coordinator maintains a system topic named `` `__transaction_key` ``, used to store information related to all transaction keys managed by the coordinator. We initially considered including the transaction key as a part of the TransactionMetadataEntry and writing it to the `` `__transaction_log_` ``. However, since the `*TransactionMetadataEntry*` is cleared upon transaction completion, we decided against this persistence approach.

Public-facing Changes

Public API

PulsarClientBuilder

To support the proposed changes, we will introduce a new configuration option in the PulsarClient builder for setting the transaction key. Users can use this option to specify the transaction key when creating a PulsarClient instance.

With regard to transaction keys, each user can have multiple transaction keys, but these keys must be unique within the same user role. This ensures that different transactions initiated by the same user role can be distinguished from one another. Similarly, for authenticated users without a specific role, all transaction keys should also be unique to prevent any interference between transactions.

It's important to note that each transaction key must not contain the '&' symbol. This is because the broker uses the combination of owner and transaction key (`owner&transaction key`) as the key mapping to associate the transaction with the corresponding epoch. By doing so, the broker can efficiently manage and differentiate transactions for different users or roles. Including the '&' symbol in a transaction key could lead to incorrect or ambiguous mappings, resulting in unexpected behavior or errors.

Java
```java
public class PulsarClientBuilder {
    ...
    // Add a new field for the transaction key
    private String transactionKey;

    ...

    // Add a new method to set the transaction key
    // The transactionKey should not contain the symbol `&`.
    public PulsarClientBuilder transactionKey(String 
transactionKey) {
        this.transactionKey = transactionKey;
        return this;
    }

    ...
}
```
Then, when users create a PulsarClient, they can set the transaction key using the following code:

Java
```java
PulsarClient client = PulsarClient.builder()
    .serviceUrl("pulsar://localhost:6650")
    .transactionKey("user-defined-transaction-key")
    .build();
```
By including the transaction key in the PulsarClient builder, users can easily configure and manage their transaction keys when working with Pulsar transactions.

ExpiredTransactionException

We will introduce a new exception called `ExpiredTransactionException` to handle cases where an expired transaction is used for operations. This exception will provide more informative error messages to users and allow them to handle such cases in their applications.

it's also important to note that we don't need to add a new exception for expired connections. If a transaction coordinator client attempts to connect using an old connection epoch, it will receive a `NotAllowedException`. This exception will efficiently handle the scenario, negating the need for a separate exception for expired connections.

The definition of the new exception is as follows:

```java
package org.apache.pulsar.client.api.exceptions;

/**
 * Exception thrown when an expired transaction is used for
operations.
 */
public static class ExpiredTransactionException extends
PulsarClientException {

    public ExpiredTransactionException(String message) {
        super(message);
    }

    public ExpiredTransactionException(String message, Throwable
cause) {
        super(message, cause);
    }

    public ExpiredTransactionException(Throwable cause) {
        super(cause);
    }

}
```

This exception will be thrown in the relevant parts of the code where an expired transaction is detected, and the corresponding operation should be rejected. Users may encounter the ExpiredTransactionException when calling one of the following two asynchronous APIs:

```java
1. producer.newMessage(transaction).sendAsync();
```

```
2. consumer.acknowledgeCumulativeAsync(messageId, transaction);
```

When these APIs are called using an expired transaction, the returned CompletableFuture will complete exceptionally with an ExpiredTransactionException. Users can handle this exception in the exceptional handler, for example:

```java
producer.newMessage(transaction).sendAsync().thenAccept(messageId
-> {
    System.out.println("Message sent: " + messageId);
}).exceptionally(e -> {
    if (e instanceof ExpiredTransactionException) {
        System.out.println("Attempted to use an expired
transaction: " + e);
    } else {
        System.out.println("Failed to send messages: " + e);
    }
    return null;
});
```

Binary protocol

We propose to add a new field to record the epoch in the `` `CommandTcClientConnectRequest` `` and `` `CommandTcClientConnectResponse` `` messages in the binary protocol. This change will allow the system to handle connections based on the appropriate epoch associated with a given transaction key.
The modified protobuf message definitions would be:
For the request:

```java
message CommandTcClientConnectRequest {
    required uint64 request_id = 1;
    required uint64 tc_id = 2 [default = 0];
    required uint64 epoch = 3;  // Added field to record epoch
}
```

And for the response:

```java
message CommandTcClientConnectResponse {
    required uint64 request_id = 1;
    optional ServerError error  = 2;
    optional string message     = 3;
    required uint64 epoch = 4;  // Added field to record epoch
}
```

Configuration

None

Admin API & CLI

To provide better support for managing transaction keys and their associated transactions, we will extend the admin API and the CLI with the following new endpoints and commands:

Admin API Endpoints

1. `` `GET /admin/v2/transactions/transaction-keys` ``: Retrieve a list of all transaction keys in the system.
2. `` `GET /admin/v2/transactions/transaction-keys/{transaction_key}` ``: Retrieve information about a specific transaction key, such as the current epoch and associated transaction IDs.
3. `` `DELETE /admin/v2/transactions/transaction-keys/{transaction_key}` ``: Remove a transaction key from the system, effectively aborting any associated transactions.

CLI Commands

1. \``pulsar-admin transactions list-transaction-keys`\`: List all transaction keys in the system.
2. \``pulsar-admin transactions get-transaction-key <transaction_key>`\`: Retrieve information about a specific transaction key, such as the current epoch and associated transaction IDs.
3. \``pulsar-admin transactions delete-transaction-key <transaction_key>`\`: Remove a transaction key from the system, effectively aborting any associated transactions.

By adding these new admin API endpoints and CLI commands, users will have better visibility and control over the transaction keys and their associated transactions. This will help ensure the correct functioning of the transaction key feature and allow users to manage their transaction keys more effectively.

Metrics

To better monitor and manage the new feature, we will add some related metrics. These metrics can help users better understand the usage and performance of the new feature and make necessary optimizations and adjustments. Specifically, we will add the following metrics:

Name	Label	Type	Description
pulsar_txn_transaction_key_count		Gauge	Records the number of transaction keys currently used in the system, which can help users understand the system's load status.
pulsar_txn_transaction_key_epoch	Transaction key	Gauge	Records the most recent epoch used for each transaction key, in order to identify and resolve any potential epoch conflicts.
pulsar_txn_transaction_key_age	Transaction Key	Summary	Records the active time of each transaction

			key in order to identify and clean up unused transaction keys.

Security Considerations

No security considerations are needed.

Backward & Forward Compatability

The newly added transaction key feature has been carefully designed and implemented to ensure that it does not introduce any disruptive changes. This means that it is fully compatible with previous versions of the system and can be seamlessly integrated into existing environments.

Upgrade

Users can confidently update to the version that includes this new feature and enjoy its benefits without worrying about any adverse effects on their existing systems and business. The development team is committed to ensuring that the new feature works together with existing features and architectures, enabling smooth transitions and continuous improvements.

Revert

If users need to downgrade to a version that does not use the new feature in some cases, it is straightforward to do so. Simply delete the transaction key-related configuration in the Pulsar client configuration. This way, the system will no longer use the newly added transaction key feature and instead revert to the processing method used in the previous version. This flexibility allows users to switch features freely according to their actual needs and scenarios, ensuring the stability and reliability of the system.

Testing

The new feature has been carefully designed and implemented to ensure that it does not introduce any breaking changes. This means that it is fully compatible with previous versions of the system and can be seamlessly integrated into existing environments. Users can confidently upgrade to a version that includes this new feature and enjoy its benefits without worrying about any adverse effects on existing systems and businesses. The development team is committed to ensuring the new feature works with existing functionality and architecture to achieve smooth transitions and continuous improvements.

The new feature will undergo unit, chaos, and stress testing. Each testing method focuses on different goals and scenarios to ensure the stability and performance of the new feature.

- Unit testing: Unit testing focuses mainly on the independent components of the new feature to ensure that they work as intended. By writing test cases for each component, we can validate the implementation of the new feature at a lower level.

- Chaos testing: Chaos testing focuses on ensuring system reliability in the event of component failures and unstable environments. By simulating consumer, producer, proxy, and other component failures in various states, we can verify that the new feature can still meet basic guarantees under these conditions and ensure system stability.

- Stress testing: Stress testing focuses on evaluating the performance of the new feature under high-load conditions. By simulating large amounts of concurrent requests, high traffic, or other pressure scenarios, we can identify performance bottlenecks and verify that the new feature can continue to function properly under these conditions.

By applying these three testing methods, we can comprehensively evaluate the implementation of the new feature and ensure that it provides stable performance in various scenarios and conditions.

Alternatives

None

General Notes

Note

Links

<!–
Updated afterwards
–>
* Mailing List discussion thread:
* Mailing List voting thread: