Message Service

Best Practices

Best Practices

How to enable a one-to-multiple pull-based message consumption model

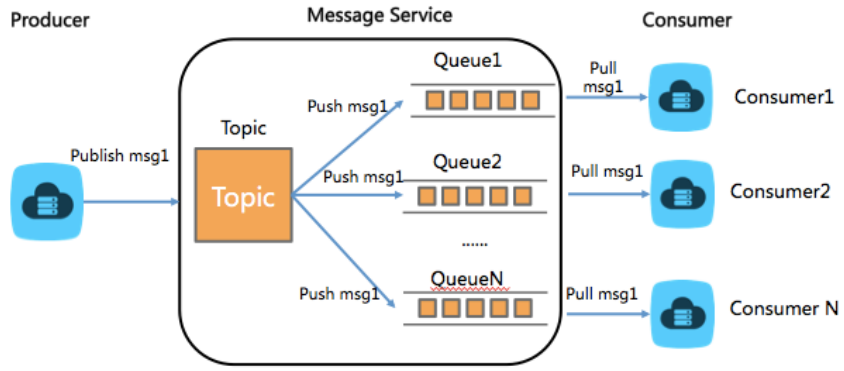
Context

Alibaba Cloud Message Service provides the queue and topic models. The queue provides a one-to-multiple sharing message consumption model, and adopts active **pull** by the client. The topic provides a one-to-multiple broadcast message consumption model, and adopts active **push** by the server. The preceding two models can meet requirements in most application scenarios.

The push mode features good instant performance. However, the client’ s address must be exposed to receive messages pushed from the server. In some circumstances, for example, in the intranet of an enterprise, the push address cannot be exposed. The pull mode is required accordingly. The one-to-multiple pull-based message consumption model can be enabled based on the queue and topic, although Message Service does not provide this consumption model. The solution is as follows:

Solution

The topic pushes the message to a queue, from which the consumer pulls the message. In this way, one-to-multiple broadcast messaging is enabled, and the consumer’ s address is not exposed, as shown in the following figure.



Interface description

CloudPullTopic in the latest Java SDK supports the above solution by default. Where,MNSClient provides the following two interfaces for you to rapidly create CloudPullTopic:

```
public CloudPullTopic createPullTopic(TopicMeta topicMeta, Vector<String> queueNameList, boolean
needCreateQueue, QueueMeta queueMetaTemplate)

public CloudPullTopic createPullTopic(TopicMeta topicMeta, Vector<String> queueNameList)
```

Where, TopicMeta specifies the meta settings for creating a topic; queueNameList specifies the queue name list to be pushed by the topic; needCreateQueue specifies whether queueNameList needs to be created; and queueMetaTemplate specifies the queue meta parameter required for creating a queue.

Demo code

```
CloudAccount account = new CloudAccount(accessKeyId, accessKeySecret, endpoint);
MNSClient client = account.getMNSClient();

// build consumer name list.
Vector<String> consumerNameList = new Vector<String>();
String consumerName1 = "consumer001";
String consumerName2 = "consumer002";
String consumerName3 = "consumer003";
consumerNameList.add(consumerName1);
consumerNameList.add(consumerName2);
consumerNameList.add(consumerName3);
QueueMeta queueMetaTemplate = new QueueMeta();
queueMetaTemplate.setPollingWaitSeconds(30);

try{
//producer code:
// create pull topic which will send message to 3 queues for consumer.
String topicName = "demo-topic-for-pull";
TopicMeta topicMeta = new TopicMeta();
topicMeta.setTopicName(topicName);
```

```
CloudPullTopic pullTopic = client.createPullTopic(topicMeta, consumerNameList, true, queueMetaTemplate);

//publish message and consume message.
String messageBody = "broadcast message to all the consumers:hello the world.";
// if we sent raw message,then should use getMessageBodyAsRawString to parse the message body correctly.
TopicMessage tMessage = new RawTopicMessage();
tMessage.setBaseMessageBody(messageBody);
pullTopic.publishMessage(tMessage);

// consumer code:
//3 consumers receive the message.
CloudQueue queueForConsumer1 = client.getQueueRef(consumerName1);
CloudQueue queueForConsumer2 = client.getQueueRef(consumerName2);
CloudQueue queueForConsumer3 = client.getQueueRef(consumerName3);

Message consumer1Msg = queueForConsumer1.popMessage(30);
if(consumer1Msg != null)
{
System.out.println("consumer1 receive message:" + consumer1Msg.getMessageBodyAsRawString());
}else{
System.out.println("the queue is empty");
}

Message consumer2Msg = queueForConsumer2.popMessage(30);
if(consumer2Msg != null)
{
System.out.println("consumer2 receive message:" + consumer2Msg.getMessageBodyAsRawString());
}else{
System.out.println("the queue is empty");
}

Message consumer3Msg = queueForConsumer3.popMessage(30);
if(consumer3Msg != null)
{
System.out.println("consumer3 receive message:" + consumer3Msg.getMessageBodyAsRawString());
}else{
System.out.println("the queue is empty");
}

// delete the fullTopic.
pullTopic.delete();
}catch(ClientException ce)
{
System.out.println("Something wrong with the network connection between client and MNS service."
+ "Please check your network and DNS availablity.");
ce.printStackTrace();
}
catch(ServiceException se)
{
se.printStackTrace();
}

client.close();
```

You can get more MNS service error code in Error codes.

How to set the message processing duration to adaptive

Context

In Alibaba Cloud Message Service specifications, each message has a default visibility timeout. After a worker receives a message, the timeout time starts.If the worker does not complete message processing within the timeout time, the message may be received and processed by another worker.

Advantage of the timeout: DeleteMessage must be displayed after message processing completes. If a worker process is crashed, the message can be processed by another worker.

Some users set the default VisibilityTimeout of a queue to a long period to ensure that the message is not released due to timeout before the worker completes message processing.

Problem

In some application scenarios, it is assumed that:

The VisibilityTimeout of a queue is set to six hours.

A worker receives a message M1. However, the worker process is crashed or the system restarts after the worker completes message processing.

The message M1 can only be received and processed by another worker in at least six hours. If you want to write code to handle Failover, the program is complex.

Objective

In some scenarios where the real-time requirement is high and each message needs to be rapidly responded to, it is expected that:

The VisibilityTimeout of a queue is short, for example, five minutes, so that the incomplete message can be received and processed by another worker in most five minutes after the current worker is crashed.

Message processing may cost more than five minutes, which cannot be timed out.

Solution

For such scenarios, a best practice with C# demo is provided (refer to the attachment).When a worker is processing a message, it regularly checks whether ChangeVisibility is required for the message. The work actively deletes the message after it completes processing.

If you have any question, you can post it in the forum, or join the TradeManager group of official Message Service technical support (group ID: 51222373).

NOTE:

1. Before running the program, set the accessId, accessKey, and endPoint.

2. Variable description:

- MessageMinimalLife indicates the minimum Life length required when a message is registered. If the remaining timeout time is only 0.1 second when a message is registered, ChangeVisibility cannot be run to extend the message life. Therefore, MessageMinimalLife ensures that the message sustains until ChangeVisibility is run. You can set this variable according to your service pressure.
- TimerInterval indicates the interval of the timer in the Manager. You only need to ensure that the timer is enabled before the message reaches MessageMinimalLife. You can set this variable to a short period (which requires frequent check).
- QueueMessageVisibilityTimeout indicates the default timeout time of the message in the Sample, which is the attribute of the queue. Each time the Sample performs ChangeVisibility, it sets the value of VisibilityTimeout to that of QueueMessageVisibilityTimeout. Therefore, the value of QueueMessageVisibilityTimeout must be greater than that of TimerInterval+MessageMinimalLife to ensure that the message is not timed out.
- MessageTimeout indicates the timeout time of the message in the Manager. If a worker is stuck, message processing is not complete in five hours (it is assumed that this duration is far beyond the normal message processing duration). The Manager does not perform ChangeVisibility for the message, but enables message visibility timeout.

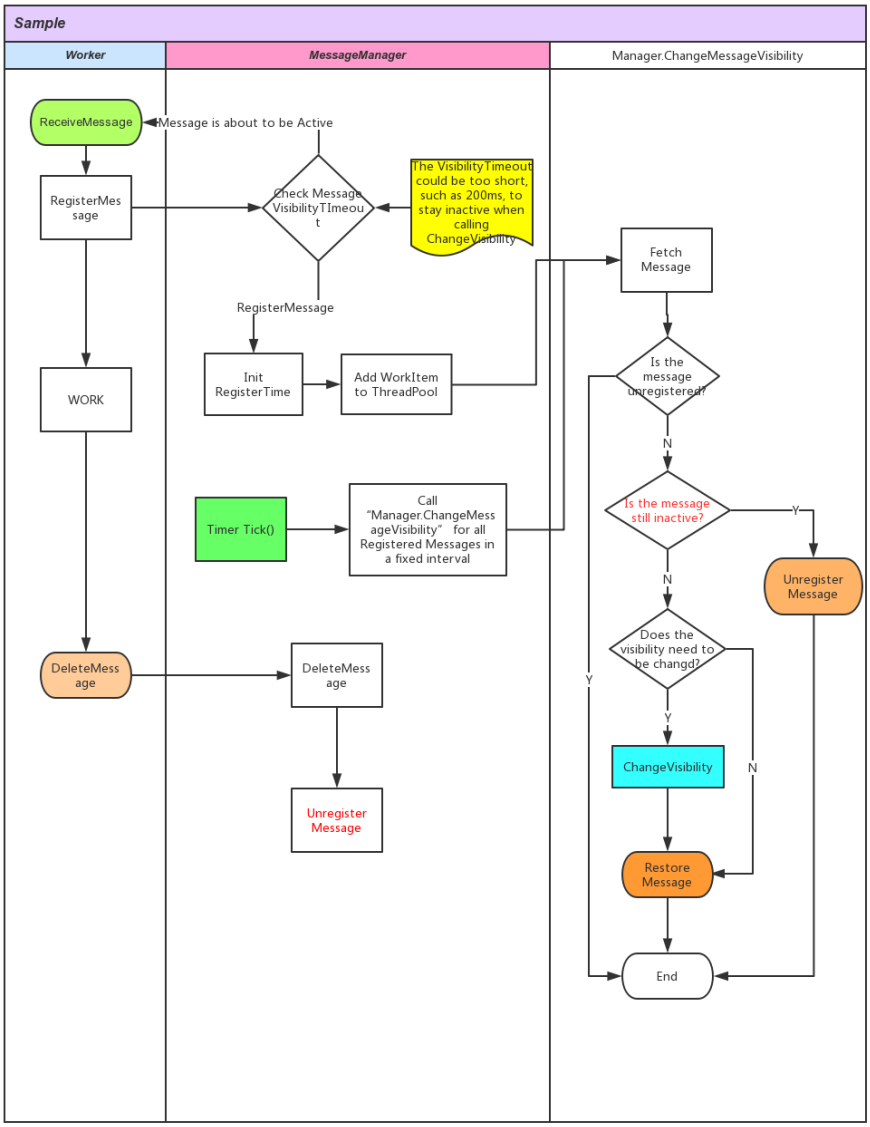
3. Process description

After receiving a message, the worker registers the message, processes the message, and calls DeleteMessage of the Manager.

After a message is registered to the Manager for the first time, the Manager calls ThreadPool to schedule a ChangeVisibilityTask to check whether ChangeVisibility is required, and then adds the message to the internal message list.

The timer in the Manager regularly calls Parallel to start ChangeVisibilityTask to check all messages in the message list.

The process related to “Manager.ChangeMessageVisibility (ChangeVisibilityTask)” is displayed in the following flowchart.



Download the sample code: ChangeMessageVisibilitySample

LongPolling

Context

Message Service enables message receiving in LongPolling mode. That is, WaitSecond is set to a

value between 1 and 30 when messages are received. By LongPolling, the server holds the request sent by the client and returns the response only when there is a message for the client. This ensures message receiving in the first time, and prevents a large number of invalid requests from the client. LongPolling is recommended in Message Service.

LongPolling requires that the server holds the persistent connection at the HTTP layer. For the server, however, resources of the persistent connection at the HTTP layer are limited. To prevent malicious attacks, Message Service provides a limited number of LongPolling connections for a single user.

Problem description

Some users enable hundreds of threads in a single client to simultaneously access the Message Service server for messages. When no message is available in the queue, the client has hundreds of requests in LongPolling mode. If the users have multiple clients, the users may need to send thousands of LongPolling requests simultaneously.

In this scenario, when the users send LongPolling requests, the Message Service server may directly return an error “Message does not exist”, instead of holding the requests.

The users cannot obtain expected LongPolling results. Some users send consecutive LongPolling requests in a While loop but do not handle exceptions. After a period, the users find that a large number of requests have been sent.

Solution

LongPolling does not need to be attached to hundreds of threads simultaneously accessing the server, if there is no message in the queue. It can be attached only to threads between 1 and N. When threads attaching LongPolling find messages in the queue, they can wake other threads to receive the messages for fast response.

LongPolling sample code is a best practice where MessageReceiver is used to receive messages. A MessageReceiver instance is created for all threads receiving messages, and receiver.receiveMessage is used to receive messages.

The receiver provides an exclusive mechanism for LongPolling. If a thread is performing LongPolling, other threads can wait for messages.

How to encrypt MNS messages to be transmitted

Context

Alibaba Cloud Message Service (MNS) provides services that are accessible over Internet using HTTP. For messages containing sensitive information, how to further increase the security of the network links between a user’ s client programs and Alibaba Cloud services? Currently, there are two solutions:

1. The HTTPS domain name for MNS is made available to users, which is being scheduled to happen mid April.
2. Messages to transmit are encrypted to avoid being intercepted.

Here are the best practices to encrypt MNS messages to transmit.

Solution

Encrypt messages before they are transmitted;

Decrypt the messages at the recipient end before they are consumed;

Sample code (Java): SecurityQueue.zip

1. SeurityQueue.java: provides putMessage, popMessage and deleteMessage interfaces.

   Before sending a message to the server, putMessage encrypts the message with the specified key and encryption algorithm.

   After receiving the message from the server, popMessage decrypts the message following the specified method and returns the decrypted message.

   SecurityKeyGenerator.java is used to generate secretKey to encrypt and decrypt messages.

   SecurityQueueDemo.java provides a demo program showing how to use SeurityQueue.

See ReadMe in the attachment for more information.

NOTE

Encrypting and decrypting messages can somehow compromise performance.

Please do not push unencrypted messages to the encrypted message queue.

How to keep a strictly ordered queue based on Message Service

Context

The queue provided by Alibaba Cloud Message Service features high reliability, high availability, and high concurrency. Data of each queue is persistently written with three copies in Alibaba Cloud Apsara distributed platform. Each queue contains at least two servers providing services, and each server supports highly-concurrent access. With such distributed characteristics, the Message Service queue cannot ensure strict first-in-first-out (FIFO) of messages as the traditional single-host queue, and can only be basically ordered.

If there are multiple message senders in a queue, strict message ordering is meaningless due to concurrency and network delay, as the actual sending order of messages from multiple senders and the real order in which messages arrive at the server are unknown. Likewise, when multiple receivers concurrently receive messages, the real message processing order is unknown.

Therefore, the message ordering is meaningful when there is only one sender (a thread or multiple threads) and one receiver; and the actual sending and receiving orders of messages can be aware and recorded.

Solution

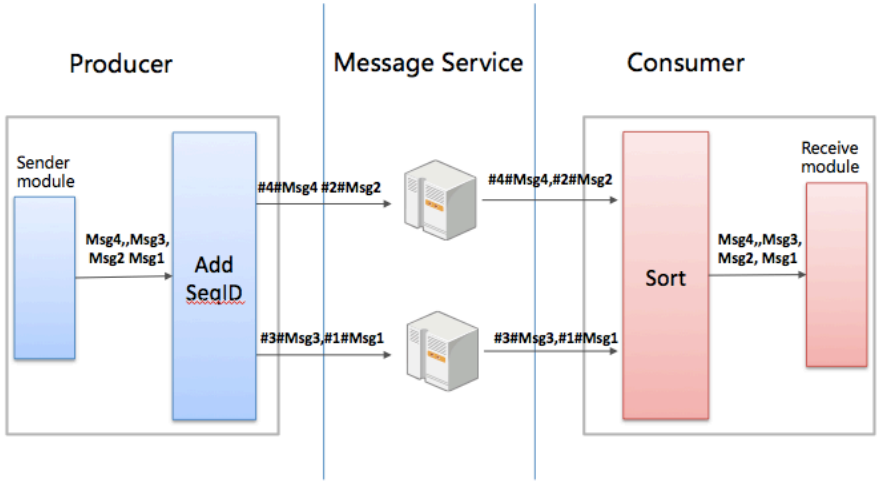
The following solution is designed based on the above assumptions to meet users' requirements on the message order and ensure that messages are received and consumed in an order they are sent.

Procedure:

The sender colors messages and adds SeqID (for example, #num#) to the messages.

The receiver restores the messages, sorts them by SeqID, and returns the messages to the upper layer. Backend threads are provided for messages that have been received to ensure that the messages will not be repeatedly consumed.

To avoid SeqID loss due to sender or receiver failure, SeqID will be persistently stored in a local disk file. (SeqID can be saved to other storage devices or databases, such as OSS, Table Store, or ApsaraDB for RDS.)



Program description

Attachment is a solution in Python version (relying on the Message Service Python SDK). In which, oredered_queue.py files in OrderedQueueWrapper type are provided to wrap common Message Service queues to ordered queues.

OrderedQueueWrapper provides the SendMessageInOrder() and ReceiveMessageInOrder() methods. SendMessageInOrder() is used to color messages, while ReceiveMessageInOrder() is used to restore messages, and return them to the receiver in order.

In addition, send_message_in_order.py and receive_message_in_order.py provide program samples for the sender and receiver to use OrderedQueueWrapper.

```
send_message_in_order.py :
#init orderedQueue
seqIdConfig = {"localFileName":"/tmp/mns_send_message_seq_id"} # Specifies the disk file to persistently send the SeqID.
seqIdPS = LocalDiskStorage(seqIdConfig)
orderedQueue = OrderedQueueWrapper(myQueue, sendSeqIdPersistStorage = seqIdPS)
orderedQueue.SendMessageInOrder(message)

receive_message_in_order.py :
#init orderedQueue
seqIdConfig = {"localFileName":"/tmp/mns_receive_message_seq_id"} #Specifies the disk file to persistently receive the SeqID.
seqIdPS = LocalDiskStorage(seqIdConfig)
orderedQueue = OrderedQueueWrapper(myQueue, receiveSeqIdPersistStorage = seqIdPS)
recv_msg = orderedQueue.ReceiveMessageInOrder(wait_seconds)
```

Running method:

Configure g_endpoint, g_accessKeyId, g_accessKeySecret, and g_testQueueName in send_message_in_order.py and receive_message_in_order.py.

Run send_message_in_order.py.

Run receive_message_in_order.py (you can perform this step without completion of step 2.)The sender sends 20 messages, and the receiver receives the 20 messages in order.

```
orderedqueuewrapper receive message body is 0
orderedqueuewrapper receive message body is 1
orderedqueuewrapper receive message body is 2
orderedqueuewrapper receive message body is 3
orderedqueuewrapper receive message body is 4
orderedqueuewrapper receive message body is 5
orderedqueuewrapper receive message body is 6
orderedqueuewrapper receive message body is 7
orderedqueuewrapper receive message body is 8
orderedqueuewrapper receive message body is 9
orderedqueuewrapper receive message body is 10
orderedqueuewrapper receive message body is 11
orderedqueuewrapper receive message body is 12
orderedqueuewrapper receive message body is 13
orderedqueuewrapper receive message body is 14
orderedqueuewrapper receive message body is 15
orderedqueuewrapper receive message body is 16
orderedqueuewrapper receive message body is 17
orderedqueuewrapper receive message body is 18
orderedqueuewrapper receive message body is 19
Finish receive 20 messages
```

You can also run the test case of oredered_queue.py to compare the ordered queue with the common Message Service queue (the endpoint and AK need to be configured):

Command: $python oredered_queue.py

Non-strictly ordered: (Entire messages are ordered while some adjacent messages are disordered. This also indicates that multiple servers in a single queue of Message Service simultaneously provide services.)

```
Test with common queue:
Send Message Succeed with commont queue. message body is :1
Send Message Succeed with commont queue. message body is :2
Send Message Succeed with commont queue. message body is :3
Send Message Succeed with commont queue. message body is :4
Send Message Succeed with commont queue. message body is :5
Send Message Succeed with commont queue. message body is :6
Send Message Succeed with commont queue. message body is :7
Send Message Succeed with commont queue. message body is :8
Send Message Succeed with commont queue. message body is :9
Send Message Succeed with commont queue. message body is :10
Receive Message Succeed with common queue! message body is 1
Receive Message Succeed with common queue! message body is 3
Receive Message Succeed with common queue! message body is 2
Receive Message Succeed with common queue! message body is 5
Receive Message Succeed with common queue! message body is 4
Receive Message Succeed with common queue! message body is 7
Receive Message Succeed with common queue! message body is 6
Receive Message Succeed with common queue! message body is 9
Receive Message Succeed with common queue! message body is 8
Receive Message Succeed with common queue! message body is 10
```

Strictly ordered:

```
Test with OrderedQueueWrapper:
orderedqueuewrapper: send message body 1
orderedqueuewrapper: send message body 2
orderedqueuewrapper: send message body 3
orderedqueuewrapper: send message body 4
orderedqueuewrapper: send message body 5
orderedqueuewrapper: send message body 6
orderedqueuewrapper: send message body 7
orderedqueuewrapper: send message body 8
orderedqueuewrapper: send message body 9
orderedqueuewrapper: send message body 10
orderedqueuewrapper: receive message body is 1
orderedqueuewrapper: receive message body is 2
orderedqueuewrapper: receive message body is 3
orderedqueuewrapper: receive message body is 4
orderedqueuewrapper: receive message body is 5
orderedqueuewrapper: receive message body is 6
orderedqueuewrapper: receive message body is 7
orderedqueuewrapper: receive message body is 8
orderedqueuewrapper: receive message body is 9
orderedqueuewrapper: receive message body is 10
```

NOTE:

> This post mainly aims to provide message ordering solutions. Code in this post has not been strictly tested. You are not advised to directly use the code in production environments without test. Meanwhile, the program is complete in a rush, which may contain errors. Looking forward to your correction.
>
> In normal conditions, the SeqIDs of the sender and receiver should match those of the messages (colored) in the queue. When the queue is deleted for recreation, ensure that the SeqID in the disk file is consistent with that in the queue. You are not advised to send a non-colored message to a queue with colored messages.
>
> A short message validation period in the queue and the actual processing result of each message may affect the message order. You need to process such disordered messages in your program.

Download the sample program:

Python sample code for an ordered queue

How to transmit messages without size limit

using MNS and OSS

Context

An Alibaba Cloud Message Service (MNS) queue has a maximum message size of 64 KB. This is about enough to meet the needs of messages used as a control-flow information exchange channel under normal circumstances. However, in some special scenarios where messages are bigger in size than that, message slices will be used.

So how can we transmit messages larger than 64 KB with MNS without using message slices? There is a solution.

Solution

If a message is found to be larger than 64 KB, the producer first uploads it to OSS before sending it to MNS;

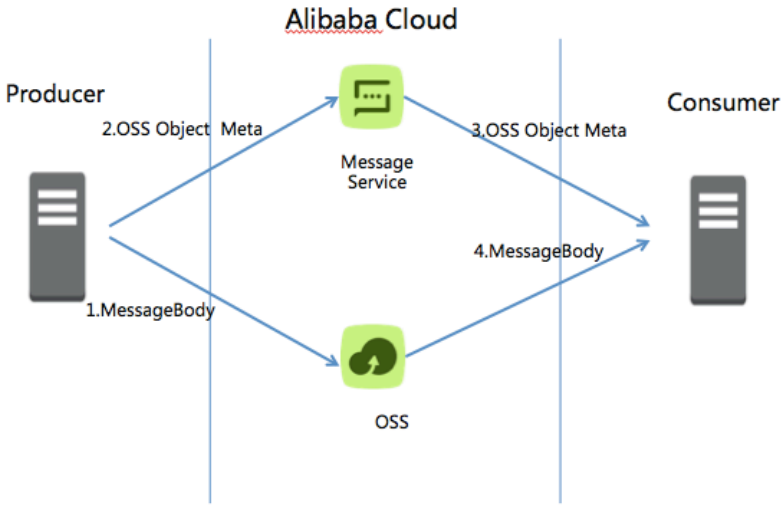
Then, the producer sends to MNS the appropriate object information for the message data;

When reading a message from the MNS queue, the consumer determines whether the message is OSS object information;

If the message is OSS object information, then the consumer can download the appropriate object information from OSS, and return it as a message to the program at the upper layer.

The messages smaller than 64 KB are still transmitted directly using MNS.

The specific process is shown below:



Program implementation

Sample code for large messages is the Java implementation of the solution above. Its main functionalities are encapsulated into a class, namely BigMessageSizeQueue

BigMessageSizeQueue provides the following public method:

```
public BigMessageSizeQueue(CloudQueue cq, OSSClient ossClient, String ossBucketName)
//Constructor, cq is a common mnsqueue object; ossClient and ossBucketName include OSS region and bucket information needed for relaying large messages
public Message putMessage(Message message) // Send a message
public Message popMessage(int waitSeconds) // Receive a message
public void deleteMessage(String receiptHandle) // Delete a message
public void setBigMessageSize(long bigMessageSize) // Set a large message threshold (a message larger than this threshold will be transmitted through OSS), which by default is 64 KB;
public void setNeedDeleteMessageObjectOnOSSFlag(boolean flag) // Choose whether to delete messages in OSS, which is set to Yes by default;
```

For details about the sample code, see the code in Demo.java attached.

NOTE

Large messages consume a considerable amount of network bandwidth. Therefore, when using this solution to send large messages, the network bandwidth for the producer and consumer may be a bottleneck.

Transmitting large messages takes longer time and is more susceptible to network fluctuations. Thus, necessary retries are recommended at the upper layer.

How to implement Message Service-based transaction messages

Context

In some cases, local operations must be consistent with message sending transactions. That is, if a message is successfully sent, the local operation succeeds; if a message fails to be sent, the local operation fails (the message needs to be rolled back even if it is successfully sent). This prevents situations where an operation succeeds but the message fails to be sent, or an operation fails but the message is successfully sent.

In addition, a message must be successfully processed once on the consumer end to avoid manual reset of consumption progress as a result of failed message processing caused by consumer program crash.

Solution

You can use the message delay function of Message Service to implement transaction messages.

Preparation

Create two queues:

1. Transaction message queue

The message validity period is shorter than the message delay time. That is, if the producer does not actively modify (submit) the message visibility time, the message is invisible to the consumer.

1. Operation log queue

The operation log queue records operations on transaction messages. The message delay time is the timeout time for the transaction operation. After a message in the log queue is confirmed (deleted), the message is invisible to the consumer.

Procedure

The producer sends a transaction preparation message to the transaction message queue.

The producer writes an operation log message to the operation log queue. The log contains

the handle of the message in Step 1.

The producer performs a local transaction operation.

If the operation in Step 3 succeeds, the producer submits the message (which is visible to the consumer). Otherwise, the producer rolls back the message.

The producer confirms the operation log in Step 2 (and deletes the log message).

After Step 4 is complete, the consumer can receive the transaction message.

The consumer processes the message.

The consumer confirms and deletes the message.

Refer to the following figure.

Exception analysis:

Exceptions of the producer (for example: process restart)

A. Read the log that is not confirmed due to operation log queue timeout.

B. Check the transaction result.

C. If the check result indicates that the transaction is successful, submit the message. (Repeated message submission does not affect the system. Messages with the same handle can only be successfully submitted once.)

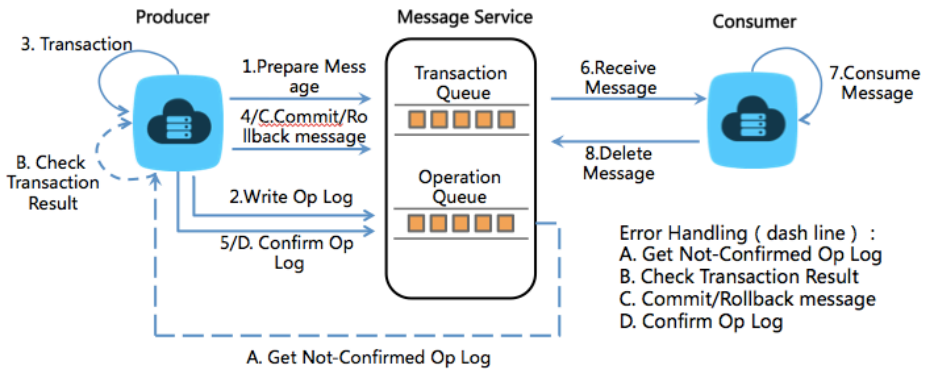
D. Confirm the operation log.

Exceptions of the consumer (for example: process restart)

A message must be processed on the consumer at least once. If Step 8 fails, the message is visible to the consumer after a period of time, and can be processed by the current or another consumer.

Unreachable Message Service (for example: network disconnection)

Message sending and receiving status and operation logs are stored in Message Service, which features high reliability and high availability. After the network connection is recovered, transactions can be continuously implemented, ensuring the consistency between operations and message sending transactions.



Code implementation:

TransactionQueue of the latest Message Service Java SDK (1.1.5) supports the preceding transaction message solution. You can easily implement transaction messages by adding the service operation and check logic to TransactionOperations and TransactionChecker.

Demo code

```
public class TransactionMessageDemo{
public class MyTransactionChecker implements TransactionChecker
{
public boolean checkTransactionStatus(Message message)
{
boolean checkResult = false;
String messageHandler = message.getReceiptHandle();
try{
//TODO: check if the messageHandler related transaction is success.
checkResult = true;
}catch(Exception e)
{
checkResult = false;
}
return checkResult;
}
}

public class MyTransactionOperations implements TransactionOperations
{
public boolean doTransaction(Message message)
{
boolean transactionResult = false;
String messageHandler = message.getReceiptHandle();
String messageBody = message.getMessageBody();
try{
//TODO: do your local transaction according to the messageHandler and messageBody here.
transactionResult = true;
}catch(Exception e)
{
transactionResult = false;
}
return transactionResult;
```

```
}
}

public static void main(String[] args) {
System.out.println("Start TransactionMessageDemo");
String transQueueName = "transQueueName";
String accessKeyId = ServiceSettings.getMNSAccessKeyId();
String accessKeySecret = ServiceSettings.getMNSAccessKeySecret();
String endpoint = ServiceSettings.getMNSAccountEndpoint();

CloudAccount account = new CloudAccount(accessKeyId, accessKeySecret, endpoint);
MNSClient client = account.getMNSClient(); //this client need only initialize once

// create queue for transaction queue.
QueueMeta queueMeta = new QueueMeta();
queueMeta.setQueueName(transQueueName);
queueMeta.setPollingWaitSeconds(15);

TransactionMessageDemo demo = new TransactionMessageDemo();
TransactionChecker transChecker = demo.new MyTransactionChecker();
TransactionOperations transOperations = demo.new MyTransactionOperations();

TransactionQueue transQueue = client.createTransQueue(queueMeta, transChecker);

// do transaction.
Message msg = new Message();
String messageBody = "TransactionMessageDemo";
msg.setMessageBody(messageBody);
transQueue.sendTransMessage(msg, transOperations);

// delete queue and close client if we won't use them.
transQueue.delete();

// close the client at the end.
client.close();
System.out.println("End TransactionMessageDemo");
}

}
```