Alibaba Summer of Code Final Report

Project name

Apache RocketMQ -- Apache RocketMQ rebalancing architecture optimization

Project task description

The current rebalancing mechanism for the cluster mode of RocketMQ is shown below:

Each consumer instance starts a rebalance service thread, which periodically obtains a list of consumer instances under the group from the broker, and sorts the message queue under the topic and consumer instances under the group, and then uses related strategy to calculate the message queue to be pulled.

To be specific, RocketMQ guarantees data consistency through a full link method. For example, each broker regularly registers with each name server to ensure that the routing information between name servers are the same. The heartbeat combined with the subscription information is sent to each broker through the consumer instance to ensure that the subscription information between brokers are the same.

However, such architecture design will bring inconsistent subscription views between each consumer in scenarios such as unreliable networks and delays. In terms of load balancing, consumers need to obtain routing information (topic-queue list) and subscription information (cid list). Each consumer instance will obtain the routing information from a random name server, and the subscription information from a random broker. The inconsistency of the view between each consumer instance will lead to unbalanced load.

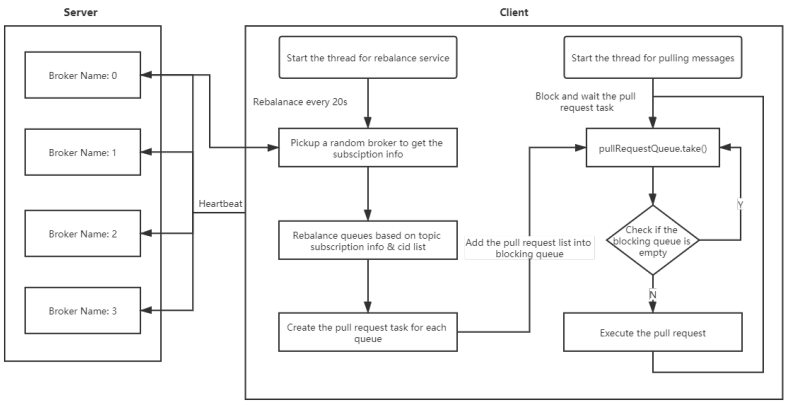
Except for the problem of the load balancing architecture, there is also another problem of queue allocating strategies implemented in RocketMQ, which existing algorithms lack of stickiness so that previous assignments of queues would not be considered. While the number of queues increases or decreases, this would lead to a large amount of overhead in the process of reassigning queues to consumers.

Implementation plan

Move the rebalancing calculation to the broker, and the client requests the broker to obtain the allocation result

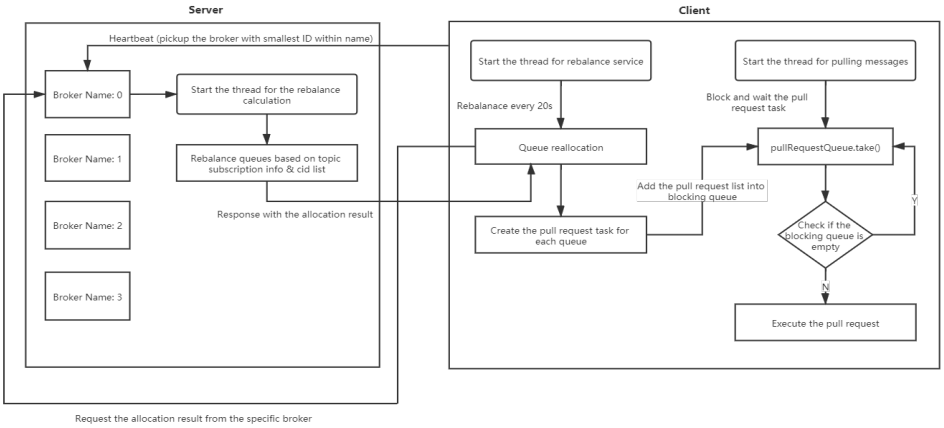
To implement this part, the initial rebalancing process would be re-design. The sketch of the

original architecture is as follows:



As could be seen from the figure, RocketMQ would initiate two threads while the RocketMQ consumer start, which one thread executes the rebalance service regularly, and the other one performs the pull message task. The rebalance service thread would obtain the subscription information and consumer ID of the consumer group from a random broker node every 20 second as clients would maintain heartbeats with each broker, and then distribute them by the specific rebalancing strategy. Later, the pullRequest object would be encapsulated with the distributed information and be put in the pullRequestQueue which is a blocking queue that it remains block state until there is a new pull request sent in. The pull message thread will then pull the task from the pullRequestQueue and execute it while the queue is not empty.

The problem of such rebalancing mechanism has been stated in the problem description section above. In this part, some required changes will be discussed. The design of new rebalancing architecture is shown below:



The figure above had presented that the server side is mainly responsible for the rebalancing calculation work. There are some main differences between these both structures. As could be seen, each consumer would send the heartbeat to all brokers by applying the original method; however, consumers under the new architecture had now been designed to maintain its

heartbeat, which carries the subscription information, with the same broker that has the smallest ID number within name. The pattern could be applied to the broker failover either. Also, as the thread which is responsible for the rebalancing calculation had now been switched to the server side, the client side would only need to launch a thread in order to request the result of rebalancing calculation from the broker side before creating the pull request task for each queue. There is one thing should be noticed in the new rebalancing architecture, as the sticky rebalancing algorithm is planned to be implemented at the next stage, so a cache mechanism needs to be developed at the broker side for storing the previous result of rebalancing calculation. The cache would contain the previous mapping relations between consumer and queues, which enables the broker calculates the queue allocation for each subsequent consumer based on the previous result to enhance the algorithm stickiness.

Those changes mentioned above are believed to help reducing the possibility of inconsistency due to the unreliable network. However, although there are several benefits for adopting this kind of rebalancing structure, it might lead to more pressures to the broker side. To solve this problem, a switch between old and new rebalancing mechanism should be developed for users to select the method that is more suitable for their projects.

Provide a sticky rebalancing algorithm for Apache RocketMQ

The purpose of this work is to solve the issue mentioned in "Problem Description" section above, which none of current queue allocation strategies of RocketMQ consider what queue assignments were before rebalancing, as if they are going to execute a fresh allocation task. Reserving existing allocations could reduce the amount of overheads of a reassignment. For instance, RocketMQ consumers retain pre-fetched messages for queues assigned to them before a reallocation. Thus, preserving the queue allocation could save on the number of messages delivered to consumers after a reassignment. There is also another benefit which the algorithm would reduce the need to cleanup local queue state between each rebalancing work.

To achieve the sticky rebalancing algorithm, there are two main requirements should be satisfied:

1. Queues are distributed as balanced as possible.
2. Queues maintain their previously allocated consumers as much as possible.

The first goal mentioned above need to take precedence over the second one, which means some queues cannot be allocated to the same consumer as previous assignments and should be switched to another consumer to guarantee the balanced allocation as possible. The meaning of the balance assignment here could be evaluated by two aspects:

1. The number of queues allocated to consumers differ by at most one.

2. None of queues allocated to consumer A could be allocated to consumer B if A has over two fewer queues assigned to it compared to B.

With the sticky rebalancing strategy, the reallocation process would be designed as follows:

1. All existing queue allocation are reserved.
2. Delete all mappings of queue and consumer that became invalid regarding to the change that triggers the reallocation.
3. Allocate unassigned queues in a manner that balances consumers' overall allocations of queues, which each queue would be assigned to the consumer which has the lowest number of queues in the group during the iteration.

To be specific, an example would be given below. Assume that there are 8 queues (Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8) with the same topic, and 3 consumers (C0, C1, C2) in the same group which have subscribed the topic.

1. Start from applying the sticky rebalancing strategy for the initial allocation, consumers would be sorted by the ascending order regarding to the number of queues assigned to them. If none of them has been allocated with queues, consumers will be sorted based on their ID numbers within name like below:

Consumer List
C0
C1
C2

2. During the whole rebalancing process, the number of queues allocated by each consumer changes dynamically, and this change will be reflected in the ranking of each consumer. For example, initially C0 is ranked first. If a queue is assigned to C0 at this time, then C0 will be ranked last, because it has the most queues. The allocation process of queues mentioned above is as follows:

Consumer List	Queue List
C0	Q1
C1	
C2	

Consumer List	Queue List
C1	
C2	
C0	Q1

Consumer List	Queue List
C1	Q2
C2	
C0	Q1

Consumer List	Queue List
C2	
C0	Q1
C1	Q2

Consumer List	Queue List
C2	Q3
C0	Q1
C1	Q2

The final allocation result:

Consumer List	Queue List
C0	Q1, Q4, Q7
C1	Q2, Q5, Q8
C2	Q3, Q6

3. Now, the consumer C1 has left the group due to the device failure. This will make mappings of queue and consumer attached to C1 (Q2, Q5, Q8) become invalid.
4. The rebalance service has been triggered and the sticky rebalancing strategy has been applied. First, it will reserve the previous allocation and arrange all consumers by the number of queues they have in ascending order. If the number is the same, consumers will be sorted based on their ID numbers within name.

Consumer List	Queue List
C2	Q3, Q6
C0	Q1, Q4, Q7

5. After, traverse each unassigned queue in turn and assign to each existing consumer. It should also be noted that during the rebalancing process, the number of queues allocated by the consumer is dynamic, and this change will be reflected in the order of the consumer.

Consumer List	Queue List
C2	Q3, Q6, Q2
C0	Q1, Q4, Q7

Consumer List	Queue List
C0	Q1, Q4, Q7
C2	Q3, Q6, Q2

Consumer List	Queue List
C0	Q1, Q4, Q7, Q5
C2	Q3, Q6, Q2

Consumer List	Queue List
C2	Q3, Q6, Q2
C0	Q1, Q4, Q7, Q5

Consumer List	Queue List
C2	Q3, Q6, Q2, Q8
C0	Q1, Q4, Q7, Q5

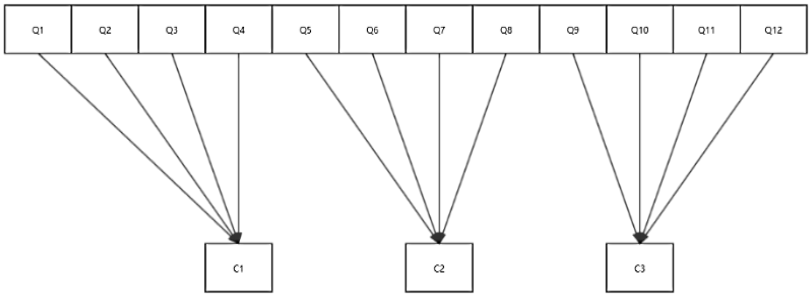
The final result of applying the sticky rebalancing algorithm:

Consumer List	Queue List
C0	Q1, Q4, Q5, Q7
C2	Q2, Q3, Q6, Q8

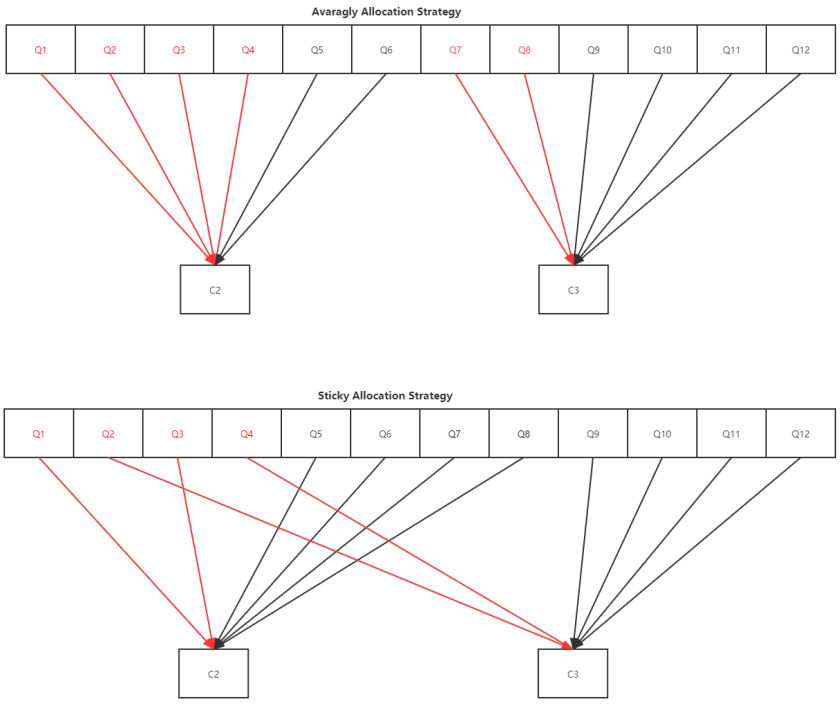
In the above queue allocation process, it could be seen that due to the continuous allocation of queues, the number of queues owned by each consumer is constantly changing, so its ranking is also changing. However, in the end, each queue is evenly distributed to each consumer, and it also ensures the queue which has been consumed will not be assigned to other consumers, which shows the consistency of the algorithm.

There are some differences between current rebalancing strategies and the sticky one. A simple comparison between averagely and expected sticky rebalancing algorithm would be provided below:

1. Assume there are 12 queues in a single topic (Q1 – Q12).
2. There is a consumer group containing 3 consumers (C1 – C3)
3. The initial allocation of both strategies is shown below:



4. Now C1 has left the consumer group. The expected result of applying both strategies for the rebalancing process would be as follows:



According to figures shown above, comparing to the averagely allocation strategy, fewer assignments need to be switched while applying the sticky rebalancing algorithm.

Implementation of rebalances evaluation method in OpenMessaging-Chaos framework

OpenMessaging-Chaos could be used to measure the performance of RocketMQ while the device failure occurs, which enables simulating the situation that triggers the rebalancing process. In this part, it would discuss the standard for evaluating the result of rebalancing strategies and the initial evaluation plan.

There are three aspects of queue reallocation strategies need to be assessed:

1. Balance degree (average degree)

   To calculate the balance degree, which is also known as average degree, for each rebalancing strategy, the calculation of standard deviation could be applied to measure the amount of variation or dispersion of the queue allocation result.

N = Total number of consumers

x = Number of queues allocated to the consumer i

μ = Average number of the queue allocation

$$Balance\ Degree = \sqrt{\frac{1}{N}\sum^{N}(x_i - \mu)^2}$$

2. Stickiness

   To obtain the stickiness of each rebalancing strategy, the rate of queues which have maintained the connection with same consumers after rebalancing should be calculated. The closer the value to 1, the higher the stickiness.

$N = Total\ number\ of\ queues$

$M = Total\ number\ of\ consumers$

$x =$ Number of queues maintaining the same connection with consumer i after rebalancing

$$Stickiness = \frac{\sum_{i=1}^{M} x_i}{N}$$

3. Time consumption

   As a more complex implementation needed to be applied to develop the sticky rebalancing algorithm, the efficiency is also an essential measurement for it. The difference between starting and ending time of the process would be recorded to obtain the time consumption of the reallocation.

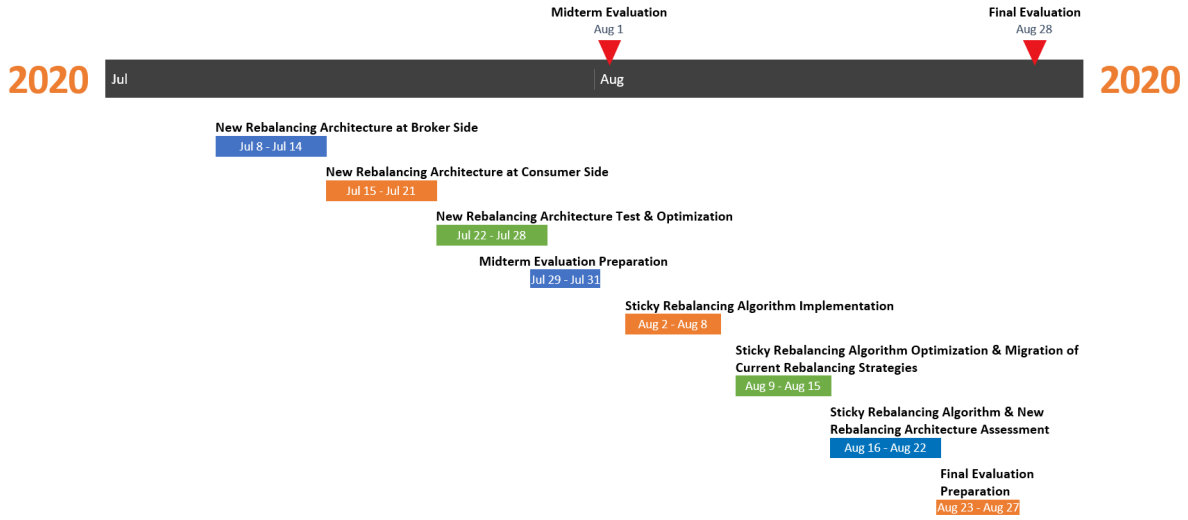
$$Time\ Consumption = Rebalance\ end\ time\ (ms) - Rebalance\ start\ time\ (ms)$$

The evaluation plan would involve several functions of the OpenMessaging Chaos framework, which could test both rebalancing architecture and strategies under certain chaotic situation. The fault types that would be applied to the project are described in the following table:

Fault Type	Description
random-loss	The function would select nodes and make it lose packets, which can simulate the situation of network loss.
random-kill (minor-kill, major-kill, fixed-kill)	The function would randomly kill node processes and restart them, which can simulate the situation of system breakdown or failure.
random-suspend (minor-suspend, major-suspend, fixed-suspend)	The function would stop random(minor, major, fixed) nodes with SIGSTOP and make it continue working with SIGCONT, which can simulate the situation while the node has been paused.

Milestone review

Date	Time	Description
25 May – 7 Jul	N/A	• Finish the detailed proposal. • Discuss the targeted idea with mentors and RocketMQ community. • Read documents and code of RocketMQ.
8 Jul – 14 Jul	1 week	• Implement the rebalancing calculation thread at the broker side.
15 Jul – 21 Jul	1 week	• Develop the routing algorithm for all consumers under a consumer group sending requests for requiring rebalancing results to the same broker. • Develop a switch for users selecting the preferred rebalancing architecture.
22 Jul – 28 Jul	1 week	• Write and perform unit tests for the new rebalancing architecture.
29 Jul – 31 Jul	3 days	• Improve the unfinished development. • Prepare for the midterm evaluation.
1 Aug	N/A	• Midterm evaluation
2 Aug – 8 Aug	1 week	• Implement the first version sticky rebalancing algorithm. • Develop the cache mechanism at the broker side for the sticky rebalancing algorithm.
9 Aug – 15 Aug	1 week	• Migrate current rebalancing strategies to the broker side and perform required changes. • Improve the sticky rebalancing algorithm based on the code review by community members.
16 Aug – 22 Aug	1 week	• Assess sticky rebalancing algorithm and new rebalancing architecture by writing reliability test. • Generate the evaluation report.
23 Aug – 27 Aug	5 days	• Improve the unfinished development. • Prepare for the final evaluation.
28 Aug	N/A	• Final evaluation



Project deliverables

Move the rebalancing calculation to the broker, and the client requests the broker to obtain the allocation result

- Developed the rebalance service thread at the broker side, which is similar to the implementation of the cluster case of rebalanceByTopic() method in RebalanceImpl().
- Developed the routing algorithm to make all consumers under a consumer group sending requests for rebalancing result to the same broker.
- Developed a switch for users selecting the preferred rebalancing architecture.
- Wrote and performed unit tests for the new rebalancing architecture.

Provide a sticky rebalancing algorithm for Apache RocketMQ

- Implemented the sticky rebalancing algorithm.
- Developed a cache mechanism at the broker side for the sticky rebalancing algorithm, which enables storing the current rebalancing result for the next calculation.
- Migrated current rebalancing strategies to the broker side and perform required changes.
- The sticky rebalancing strategy now can only be implemented on the broker side, as there will have some compatibility issues while implementing on the client side.

Implementation of rebalances evaluation method in OpenMessaging-Chaos framework

- The topic had been replaced by reliability test since OpenMessaging-Chaos framework still has no ability to simulate the rebalancing situation.
- Some situations such as restarting random consumers or brokers, disconnecting the network between consumer and broker, and the number of consumers exceeds message queues while rebalancing are simulated by the reliability test.
- The following table is a simple time performance contrast between new rebalancing structure & sticky algorithm v.s. existed rebalancing structure:

Number of messages: 1000000

Number of name servers: 1

Number of brokers: 1

Number of message queues: 4

Number of consumers: 4

Testing method:

Running the program contained the process of:

1. Registered producers
2. Producers sent messages to the broker
3. Registered consumers
4. Consumers pulled messages from the broker

	Rebalance By Broker (Sticky)	Rebalance By Broker (Averagely)	Rebalance By Client (Averagely)
Rebalance once	600181	598141.2	596016.4
Rebalance twice	772540.2	769566.4	767317.8

According to the testing result, the new rebalancing architectures is slightly more time consuming than the original one, however, it could solve the potential issue with rebalancing on the client side. Furthermore, the comparison between rebalancing with the sticky strategy and the existing strategies also should be noticed. Within the figure shown above, though the sticky strategy spend more time on the rebalancing process, it can preserve more connections between consumers and queues from the last-time allocation. There might have more evaluations of this rebalancing structure or sticky algorithm in the future.

* The code of deliverables mentioned above can be reviewed at the pull request referenced to issue #2149: https://github.com/apache/rocketmq/pull/2169
* Few features mentioned in the implementation plan section have not be implemented due to some compatibility reasons, which might be developed in the future.

Project highlights

- Implemented the new rebalancing architecture, which could avoid the inconsistency of views obtained by each consumer while the network is unstable.
- Implemented the new rebalancing strategy, which focuses on the stickiness of the rebalancing process to avoid a large amount of overhead while consumers or queues increasing/decreasing.
- Implemented the reliability test for evaluating the performances of new and old rebalancing architectures and the sticky algorithm.

Experience

I had a wonderful time during the whole Alibaba Summer of Code period with the fantastic

community at RocketMQ. Besides making a huge improvement on my coding skills, conversational and teamwork skills had also been developed after the coding phase.

Back to the time when Alibaba Summer of Code started, I was so nervous since the topic I chose was too difficult to be implemented. Fortunately, I had a professional and patient mentor – Rongtong Jin, who had guided me through the project in details and provided me a lot of ideas on building suitable solutions. The cooperation between us was pleasant and efficient, which enables me reaching the targeted progress in most of the time.

Although I had some open source experiences for the past two years, Alibaba Summer of Code still offers me a more thorough understanding of open source culture. It is unbelievable that the code written by me might be used by thousands of people and companies around the world. Contributing to such a famous open source project is different from developing the personal project, which I need to consider many aspects such as reliability, robustness, and performance of the system. Also, doing the code review with other community members is an essential final step to ensure the quality of the code, which made me learn a lot from it.

Eventually, I would like to give thanks to Alibaba Cloud for offering me this chance to take such an impressive journey. Open source is a spirit that deserves to be persisted. I will keep contributing to the RocketMQ community or even other open source projects after the event.