Dealing Distributed Transactions with 2PC, 3PC, Local Transaction Table with MQs

Adrian LIU Distributed Systems, Software Architecture 2021年July月26日2021年August月3日 3 Minutes

In a monolithic system, transactions can be managed by the DB, e.g. lock, redo log, undo log, DB's ACID characteristics, and etc. When it comes to distributed systems, multiple systems involved, the situation becomes more complicated compared to the scenario in a monolithic system.

I am going to start with the some theories and architecture design to solve **Distributed Transaction** requirements. When I have time in the future, I will write some posts about implementation details, high concurrent and low latency approaches.

This blog will be structured with the following sections:

1. Key takeaway architecture design mentality
2. Terminology
3. 2PC—Two-Phase Commit
4. 3PC—Three-Phase Commit
5. Eventual Consistency with Message Queue, Schedule Tasks and Local Transaction Table

Key takeaway architecture design mindset

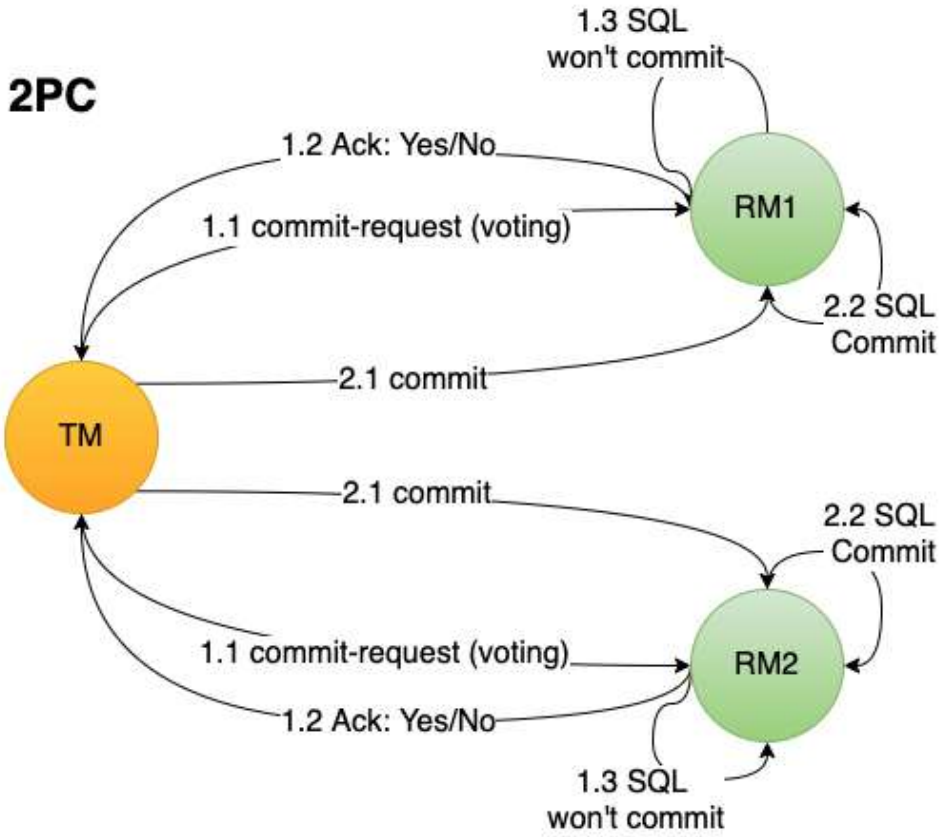
> *You can't rollback a task once it is committed/consumed in a third party systems so writing to local DB before sending message to MQ is the key.*

Terminology

TM = Transaction Manager, also called coordinator

RM = Resource Manager, also called cohort, you can take it as services such as payment service, order service

2PC — Two-Phase Commit



2PC is widely used in many organisations.

Only TM has timeout. TM timeout means after a timeout period the TM has receive response from a specific RM.

Work Flow

1. TM sends commit requests, i.e. pre commit requests, to all involved RMs. During this phase, RMs that receive the requests will start the transaction but not commit it. Resource is locked once the transaction is started on the RMs.
   - It blocks other requests that require the access to the locked resource.
2. It will have 2 scenarios on the second phases
   - If all RMs success on the previous phase, then TM sends commit instruction to all RMs;
   - If one or more of the RMs are failed on the first phase, no matter due to network failure, pre commit failure, TM timeout, or other errors, then TM will send out abort instructions to all RMs.

Pros

Better commit success rate compared to a service directly calling other parties. This is because the pre-commit phase already verify that it is able to commit before the TM sending out commit request. If the pre-commit success, the server and network are most likely in a good state.

Cons

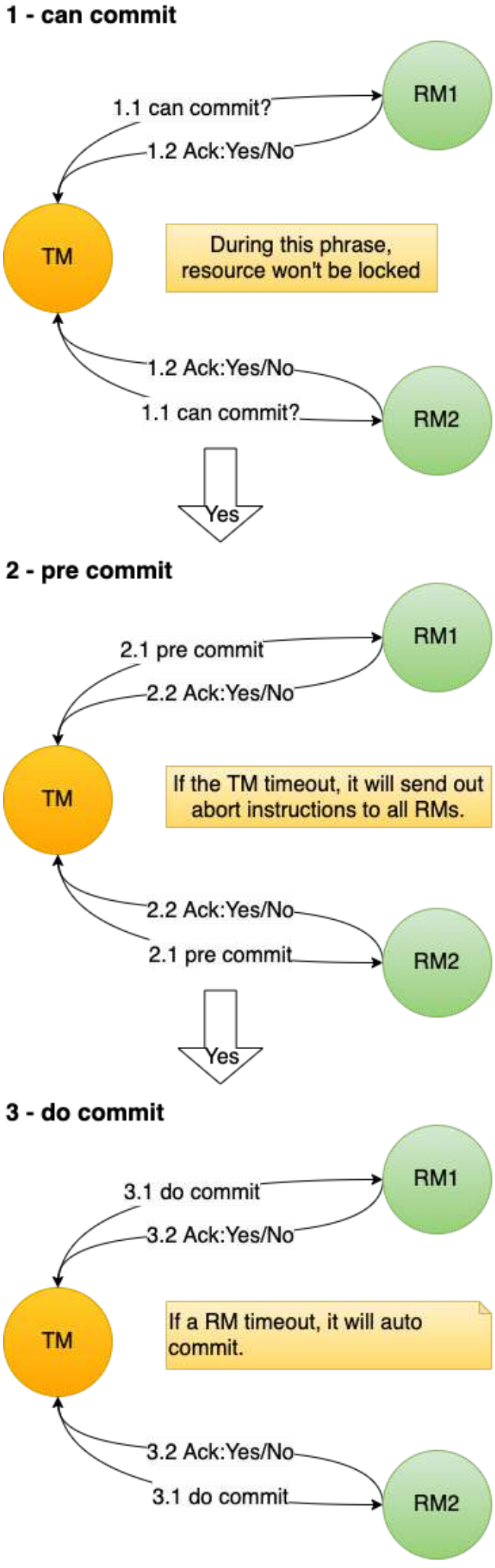
1. Single node failure. The entire transaction workflow is managed by the TM. If the TM down during the process, the transaction is broken.
2. Blocking the distributed system. Resource is locked between phase one and phase two. If there are network issues or server issues, e.g. STW, it could block other requests to access the same resource for long.
3. Data inconsistent. If in the 2 phase, the TM has sent commit request to RM1 and RM1 has committed, at the same time both TM and RM1 down before RM2 receive the commit request from the TM, then there will be data inconsistent because RM1 committed while RM2 hasn't. Also the standby instance of TM might not has yet synchronised the transaction state from the Pre-Active TM.

3PC — Three-Phase Commit

3PC has relatively **higher transaction success rate** and **lower resource lock time** compared to 2PC.

In 3PC, both TM and RM have timeout.

Work Flow


1 - can commit
RM1
1.1 can commit?
1.2 Ack:Yes/No
TM
During this phrase, resource won't be locked
1.2 Ack:Yes/No
1.1 can commit?
RM2
Yes
2 - pre commit
RM1
2.1 pre commit
2.2 Ack:Yes/No
TM
If the TM timeout, it will send out abort instructions to all RMs.
2.2 Ack:Yes/No
2.1 pre commit
RM2
Yes
3 - do commit
RM1
3.1 do commit
3.2 Ack:Yes/No
TM
If a RM timeout, it will auto commit.
3.2 Ack:Yes/No
3.1 do commit
RM2

Key changes

1. In the 2nd phase, if the TM hasn't got the response from certain RMs, then it will assume those RMs are failed to pre-commit. Then the TM will send out abort instructions to all RMs.
2. In the 3rd phase, if RMs are timeout, then they will continue to commit.

Pros

1. Higher Transaction success rate and lower resource lock time because the can-commit phase tests the network and servers' state is in good condition.
2. Both TM and RM have timeout mechanism. This reduces the probability of data inconsistent.

Cons

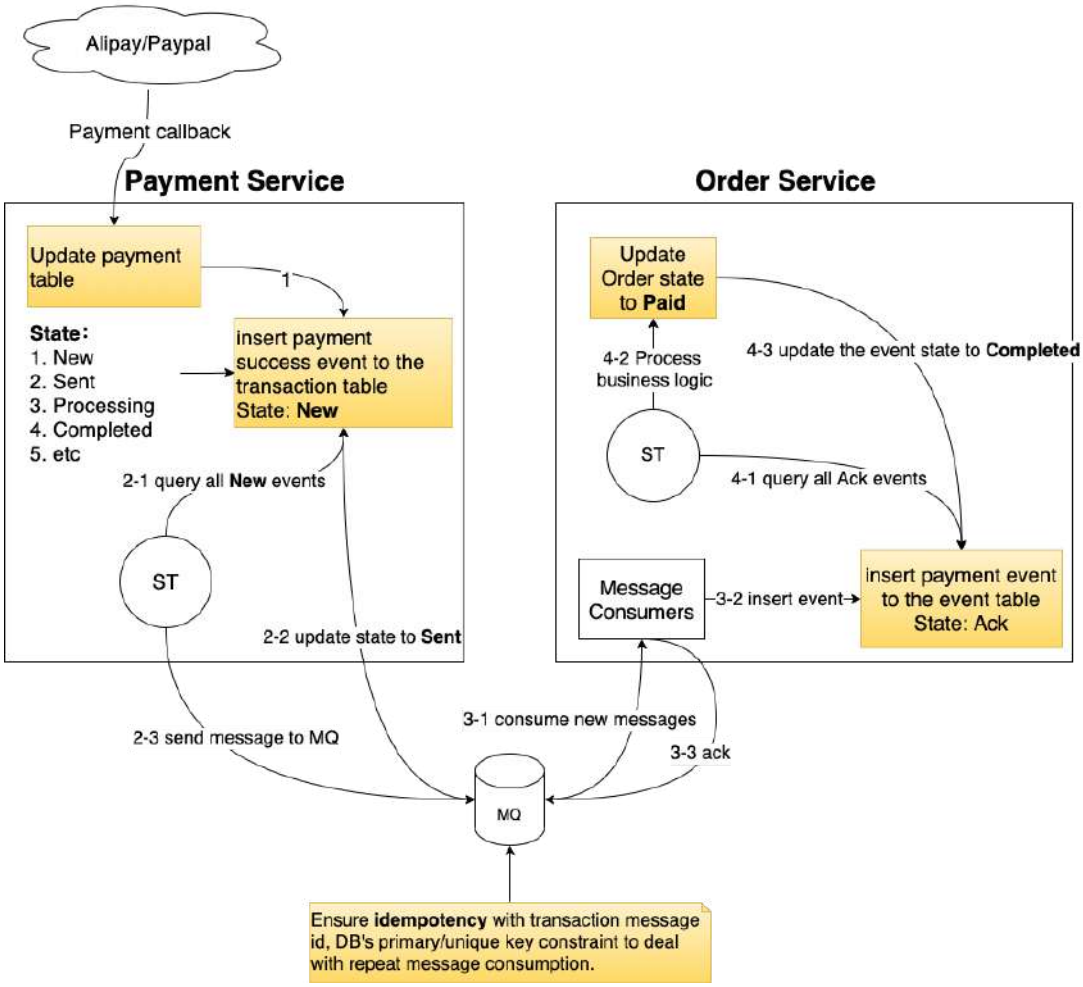
1. It hasn't solved all the issues of 2PC but rather just mitigate their severity.
2. Data inconsistent. If the TM fail and some of the RMs has pre-committed while others are not, then there will be data inconsistent.

Compensation Mechanisms

Here are the typical ways of applying compensation to deal with the data inconsistent issues of 2PC and 3PC.

1. Human intervention compensation
2. Schedule tasks compensation
3. Cron jobs compensation

Eventual Consistency with Message Queue, Schedule Tasks and Local Transaction Table



This approach is suitable for those system without high concurrent requests. It's enough for many small to medium businesses or internal systems. Both the schedule service and MySQL are the bottlenecks of this system. Frequently read/write would place overloaded stress on MySQL DBs. Scaling is also an issue because here services are tightly coupled with the db.

Work Flow

Both phase 1 and 2 are local DB transaction execution.

You can't rollback a task once it is committed/consumed in a third party systems so writing to local DB before sending message to MQ is the key.

Therefore, we first write to local DB before sending message to MQ.

1. Anything fail before sending to the MQ won't impact other services.
2. If the MQ is down
   - If the MQ hasn't sent out message to the order service, then it's fine because when its standby instance take place, it will send out the message.
   - If the MQ has sent out the message to the order service, we will have mechanism described in the above diagram to ensure **Idempotency** so no need to worry about repeat consumption issues.

Tagged:
2PC,

3PC,
MQ,
SAGA,
TCC,
Transaction

Published by Adrian LIU

View all posts by Adrian LIU

One thought on “Dealing Distributed Transactions with 2PC, 3PC, Local Transaction Table with MQs”

Pingback: Domain Driven Design(DDD) and Microservice decomposition – Adrian's Blog