Chapter 6
2PC+: A High Performance Protocol for Distributed Transactions of Micro-service Architecture

Pan Fan, Jing Liu, Wei Yin, Hui Wang, Xiaohong Chen, and Haiying Sun

6.1 Introduction

With the rapid development of the Internet field, the traditional single-service architecture is facing huge challenges. Micro-service architecture [1] has become extremely popular today. The core of the micro-service is to implement separate deployment, operation and maintenance, and expansion according to the business module. Unfortunately, there are many new problems in micro-services compared to traditional single-service architectures. Among them, distributed transaction in micro-services is one of the most common challenge [2].

In the traditional single-service architecture, whole the business modules are concentrated on the same data source. Therefore, it is convenient to implement the local transaction mechanism to ensure data consistency within the system, such as 2PC [3], MVCC, etc. However, in the micro-service architecture, service modules often straddle heterogeneous distributed system, and they are deployed across services and resources [1]. With the rapid growth of the number of micro-service nodes, 2PC and MVCC exist in their performance bottlenecks. So that 2PC and MVCC are hard to reach the processing performance of micro-services. In the evaluation experiment in Section 4, the latency of 2PC dropped below 35% of its maximum as contention increased.

P. Fan · J. Liu (✉) · W. Yin · X. Chen · H. Sun
East China Normal University, Shanghai, China
e-mail: jliu@sei.ecnu.edu.cn; yin_wei@careri.com; xhchen@sei.ecnu.edu.cn; hysun@sei.ecnu.edu.cn

H. Wang
Shanghai Avionics co. Ltd., Shanghai, China
e-mail: wang_hui@careri.com

H. Gao, Y. Yin (eds.), *Intelligent Mobile Service Computing*, EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-50184-6_6

In this paper, a distributed transaction concurrency control optimization protocol 2PC+ is proposed, which can extract more concurrency in the case of high competition than previous methods. 2PC+ is based on the traditional two-phase commit protocol, combined with transaction thread synchronization blocking optimization algorithm SAOLA.

We apply proposed 2PC+ protocol to the case of MSECP micro-service platform. Through experimental result, 2PC+ has obvious improvement in RT and TPS performance compared to 2PC for distributed transaction.

The rest of this paper is organized as follows. Section 2 elaborates the basic concepts and knowledge background in distributed transactions. In Section 3, we introduce the design details of the 2PC+ protocol and an optimization algorithms SAOLA based on 2PC protocol. The TLA+ verification process of the SAOLA is given in Section 4, and its feasibility is verified by running the results of TLC. Section 5 gives a comparison experiment between 2PC+ and 2PC. Section 6 discusses some related work, and Section 7 summarizes this paper.

6.2 Preliminaries

2PC and OCC Low-Performance Approaches Developers prefer the strongest isolation level serializability, in order to simplify the process of controlling distributed transactions. To guarantee the rules in transaction, traditional distributed systems (e.g. relational database) typically run standard concurrency control schemes, such as two-phase commit (2PC) combined with optimistic concurrency control (OCC) [11]. However, in many conflict transaction scenarios, 2PC combined with OCC measures perform poorly under highly competitive workloads.

For example, Table 6.1 shows a snippet of a new order transaction that simulates a customer purchasing two items from the same store. The transaction consists of two threads working on fragments P1 and P2, each of which reduces the inventory of different materials. Although each fragment can be executed automatically on its own machine, distributed control is still required to prevent fragmentation of the fragments between services. For example, suppose we keep the inventory of goods a and b constants and always sell the two together. In the absence of distributed transaction control, one customer can buy a but not b, while another customer can buy b but not a. This is due to the presence of locks in 2PC, so transactions can be aborted or even fail due to long thread blocks.

X/Open DTP X/Open [4] is the most widely used distributed transaction solution model in the single architecture. The key point is to provide a distributed transaction specification protocol: XA protocol. It uses the 2PC protocol to manage distributed transactions. The XA protocol specifies a set of communication interfaces between the resource manager (RM) and the transaction manager (TM). It is mainly composed of three main modules, i.e., RM, TM, and application (AP). Among them, RM is specifically responsible for the resource groups actually involved in

Table 6.1 A fragment of new-order transaction containing two pieces

transaction new_order_fragment
#simplified new-order "buys" 1 of a, b
input: a and b
begin
...
P_1:
R(tab=" Inventory ", key= a) → number
if (number > 1):
W(name=" Inventory ", key= a) ← number - 1 ...
P_2:
R(name=" Inventory ", key= b) → number
if (number > 1):
W(name=" Inventory ", key= b) ← number - 1 ...
end
...

the system, such as database resources and system operation platform resources. TM controls the distributed transaction process globally within the system, including the execution cycle of distributed transactions and coordination of RM resources.

6.3 Design

Based on the defects in the distributed transaction solution under the 2PC protocol, and optimizing the processing performance in the micro-service architecture for its problem, we propose an optimized 2PC+ protocol. It highly reduces the time cost of thread synchronization blocking when the service node processes distributed transaction in the system.

In the process based on the 2PC protocol, the thread participating in the transaction needs to be blocked after the two commits. When all the participants have completed the commit transaction, the blocking lock can be released. Thread synchronization blocking optimization in resource manager is the key to improving the performance of the original solution. In this section, the algorithm SAOLA is proposed and given the specific design and implementation.

Percolator Transaction Percolator is developed by Google to handle incremental web indexing and provides a strong and consistent way to update indexing information in a cluster of machines under distributed systems [5]. Two basic services are provided in percolator transaction, namely, a timestamp identifying the order of transactions and a distributed lock that detects the state of the process. It applies a single-row atomicity mechanism in big table. The commit and rollback operations in the original multi-row, multi-column distributed transaction are converted to simple

single-line transactions [6]. Percolator uses lock, data, and write to execute the process, lock to store data information, write to save the final write data, and data to save the current timestamp version.

The Two-Level Asynchronous Lock Mechanism Converting synchronous blocking in a transaction commit process to asynchronous non-blocking is key in the optimization process. We refined the fine-grained of the lock into a second-level optimistic lock [7, 8] and proposed this optimization algorithm SAOLA (Secondary Asynchronous Optimistic-Lock Algorithm). The specific implementation of the SAOLA algorithm is shown below:

1. Initialize settings in the transaction properties. It includes *value*, which is the current actual value of the transaction; *beginVersion*, which is the version serial number of the transaction at beginning; *commitVersion*, which is the version serial number of the transaction commit; and *lock*, which stores the uncommitted transaction, and lock is divided into two different level locks, namely, *firstLock* and *secondLock*. They represent the two phases of the lock, respectively. And specify that the *secondLock* includes the *firstLock* information.
2. BeginTransaction. As shown in Table 6.2. At the beginning, transaction object T_1 obtains the value of *beginVersion* as *b_v* and then determines whether there is a lock in the transaction object. If it does not exist, try to obtain some latest version directly from *b_v*. And the committed transaction obtains its current latest data value through its *beginVersion*. Otherwise, if there is a lock, the following three cases will occur:
   - There is another transaction object T_2 is committing, the data value of T_2 is locked, and then it is necessary to wait for T_2 to complete the commit, finally polling for retry (the number of polling can be configured).
   - In the first case, if the T_1 wait time has passed a certain threshold WAIT_TIME, the value of T_2 is still locked. It can be determined that T_2 has network fluctuations or unforeseen exceptions such as service downtime, and it is directly considered that T_2 has been interrupted. In this condition, T_1 can release the lock.
   - The *lock* in T_2 may be abnormally cleared without exception. T_2's *firstLock* has completed committing and has been successfully released. However, an exception occurred in *secondLock* that caused unsuccessful commit and remains. At this condition, it can release the *lock* directly.
3. PreCommit. The algorithm is shown in Table 6.3. At this time, T_1 has obtained all the latest values and can start executing transaction precommit. The process is divided into the following three branches:
   - The first case is for all transaction objects (only T_1 and T_2 transaction objects are assumed). The *value* of T_1 and T_2 is after the version sequence *b_v*, and it is judged whether there is a write operation of other transaction objects T_x. If it exists, then T_x has updated the latest data value, and the current T_1 and T_2 are directly rolled back, so the process can be ended.

Table 6.2 Algorithm of BeginTransaction

Algorithm 1 BeginTransaction

Input List<TransactionItem> txGroup

Output: newestValue

1: **for** T_{start} *and* T_{else} **to** txGroup
2: **if** T_{start}.isLocked == TRUE **then**
3: *waiting for* T_{else} *update to commit*
4: **if** *poll until* T_{start}.isLocked == FALSE *within* WAIT_TIME
5: break;
6: **end if**
7: **else**
8: *releaseLock*(T_{start})
9: **if** !firstLock. isLocked && secondLock.isLocked **then**
10: *releaseLock*(T_{start})
11: **end if**
12: **end if**
13: **else** // 不存在lock
14: **Long** b_v = T_{start}.beginVersion
15: newestValue = *getByBeginVersion*(b_v)
16: **return** newestValue

Table 6.3 Algorithm implementation of PreCommit

Algorithm 2 PreCommit

```
# items represents the list of all transaction objects
1: for T to items
2: If T.hasWriteData() == true || T.isLocked() == true then
3:    group.doRollBack()
4:    return
5: end if
6: else
7:      T.isLocked = true
8:      write to newestItem.value to T
9:      group.doCommit ()
10: end else
```

Table 6.4 Algorithm implementation of Second-Commit

Algorithm 3 Second-Commit

```
# items represents the list of all transaction objects
1: for T to items
2:   if T.firstLock.isLocked() == true then
3:     group.commit ()
4:   end if
5:   else
6:     group.rollBack()
7:     T.secondLock.ansyReleaseLock()
8:  end else
```

- The second case is to judge whether the locks in T_1 and T_2 are locked. If they match, both T_1 and T_2 need to be rolled back.
- The third case is that there is no write of the new object T_x; T_1 and T_2 do not exist *lock*. The *firstLock* of them is set to locked, and the latest data *value* is written, so the transaction group can be committed.

4. Second-Commit. The algorithm is shown in Table 6.4. After completing the precommit, all transaction objects (assuming that there are still only T_1 and T_2) can perform the final two-phase commit step. First, T_1 and T_2 will judge whether their *firstLock* is in the locked state. If they match, they can commit the transaction. Otherwise, it represents that another has cleared the *firstLock* of T_1 or T_2 and then execute the rollback operation.

   The transaction's main process has been completed. For *secondLock*, it can be completely separated from the main process, using thread asynchronous mode to clear *secondLock* and commit the transaction. Thus, even if an exception occurs in the operation step, T_{next} for the next transaction object, it finds that *firstLock* in T_1 or T_2 has been cleared, but *secondLock* still exists; T_{next} will automatically clear *secondLock* for T_1 or T_2 and commit the transaction.

6.4 Correctness

The formalized method TLA+ language is applied for verify the optimized scheme, and the rigorous mathematical logic is used to detect the logical feasibility of the optimized algorithm SAOLA. We give the TLA+ verification steps as follows:

1. Set the two invariants in the distributed transaction to constant: the current actual value of all participating transaction and all participant transaction RMs. TLA+ can be expressed as:

```
CONSTANTS  VALUE        CONSTANTS  RM
```

2. Set variables in the process: transaction status of the occurrence of RM, represented by the variable *rm_status*, i.e., "beginning," "preparing," "precommit," "committed," "cancel," etc.; The variable *rm_v* represents RM's current version and the global version sequence *ascend_v*. For the two locks of the control version in RM, i.e. *firstLock* and *secondLock*, represented by the variable rm_lock, two versions of the sequence appearing successively in the transaction flow *beginVersion* and *commitVersion* are represented by the variables *begin_v* and *commit_v*; respectively, the variables *first_val* and *second_val* are used to represent the RM values corresponding to the two versions; the variable *rm_data* is used to save the current actual value in the RM. Finally, the *committed_v* is used to save the committed transaction version.

```
VARIABLES  rm_status      VARIABLES  rm_v
VARIABLES  rm_lock        VARIABLES  begin_v
VARIABLES  commit_v       VARIABLES  rm_value
VARIABLES  ascend_v       VARIABLES  committed_v
```

3. Next, initializing the values of the individual variables. At the beginning, all variables are staying at initial value.

```
Init ==
    /\  rm_status = [r \in RM|-> "beginning"]
    /\  rm_v = [r \in RM| -> [begin_v |-> 0, commit_v |-> 0]]
    /\  rm_value = [r \in RM| -> [first_val|-> "",]]
    /\  second_val|->""]]
    /\  rm_lock = [v \in VALUE |->{}]
    /\  rm_data = [v \in VALUE |-> {}]
    /\  ascend_v = 0
    /\  committed_value = [v \in VALUE |-> <<>>]
```

4. When the process begins, the variable transaction state *rm_status* is "beginning," and the next state of *rm_status* does not contain "preparing." The *ascend_v* is in an ascending state, as shown in line 3. And the next state of the constraint *rm_value* cannot satisfy the getVal condition. In the next state of the current version of the RM, the start *versionbegin_v* is not equal to the next state of the version, as shown in the last line in TLA+.

```
Begin(r) ==
    /\  rm_status[r] = "beginning"
    /\  rm_status' = [rm_status  EXCEPT ![r] = "preparing "]
    /\  ascend_v' = ascend_v + 1
    /\  rm_value' =  [rm_value  EXCEPT ![r] = getVal]
    /\  rm_v' = [rm_v  EXCEPT !. begin_v = ascend_v']
```

5. The TLA+ below indicates that the transaction initially loads the process. At this time, *rm_status* stays at the "preparing" state. If the *preCommit* can be executed in the stage, and the next state of the *rm_status* cannot be "precommit"; otherwise, it is judged whether the resettable lock *isResetLock* condition is satisfied: if so, the reset lock can be executed.

```
Loading(r) ==
      /\  rm_status[r] = "preparing"
```

```
/\  IF  isPreCommit(r)      THEN
          /\  rm_status ' = [rm_status EXCEPT ![r] "]
          /\  = "precommit"]
          ELSE  IF  isResetLock (r)   THEN
           /\  resetLock(r)
          ELSE
             /\  rm_status' = [rm_status EXCEPT ![r]
             /\ = "cancel"]
```

6. TLA+ statement of the transaction precommit process. At this time, *rm_status* is the "precommit" state. If the final commit process *canCommit* can be executed, the global version *ascend_v* is incremented. The next state of *rm_v* is not allowed to be *ascend_v*, and the next state of the constraint *rm_status* does not contain "committing." Then it determines whether the RM can lock all values, i.e., *isAllLock*: if it matches, lock it. Otherwise, the constraint *rm_status* next step state does not contain "cancel" state. The TLA+ language in Commit is shown below, representing the two-phase commit step of the algorithm.

```
PreCommit(r) ==
      /\  rm_status[r] = " precommit "
      /\  IF  canCommit(r)      THEN
         /\  ascend_v' = ascend_v + 1
             /\  rm_v' = [rm_v  EXCEPT !. commit_v
              = ascend_v']
             /\  rm_status ' = [rm_status  EXCEPT ![r]
                       "committing"]
              ELSE   IF  isAllLock(r)      THEN
              /\  allLock(r)
         ELSE
            /\  rm_status ' = [rm_status EXCEPT ![r]
                           = "cancel"]

Commit(r) ==
          /\  rm_status[r] = "committing"
          /\  IF  isCommitFirstVal(r)     THEN
          /\  commitFirstVal(r)
          /\ rm_status '=[rm_status EXCEPT ![r]="committed"]
ELSE
              /\ rm_status '=[rm_status EXCEPT![r]= "cancel"]
```

7. Finally, we define the Next operation in the process. Obviously, it must complete four processes in sequence, as shown in the following TLA+:

```
Next ==
      \E r \in RM:
      Begin(r) \/ Loading(r) \/ preCommit(r) \/ commit(r)
```

We run the complete TLA+ program statement on TLC and get the result as shown in the Fig. 6.1. This algorithm generates a total of 1296 states, 324 different states have been found, and "no errors are found," which can verify the correctness of the algorithm SAOLA.

```
Semantic processing of module Naturals
Semantic processing of module RealClock
Implied-temporal checking--satisfiability problem has 2 branches.
Finished computing initial states: 324 distinct states generated.
--Checking temporal properties for the complete state space...
Model checking completed. No error has been found.
  Estimates of the probability that TLC did not check all reachable states
  because two distinct states had the same fingerprint:
    calculated (optimistic):  1.7072281088825747E-14
    based on the actual fingerprints:  2.0015351251421523E-14
1296 states generated, 324 distinct states found, 0 states left on queue.
The depth of the complete state graph search is 1.
```

Fig. 6.1 TLA+ result with running in TLC

6.5 Evaluation

6.5.1 Experimental Setup

Unless otherwise mentioned, all experiments are conducted on the Kodiak test bed. Each machine has a single core 2.7 GHz Intel Core i5 with 8GB RAM and 500GB SSD. Most experiments are bottlenecked on the server CPU. We have achieved much higher throughput when running on a local testbed with faster CPUs.

6.5.2 Experimental Case

We evaluate 2PC+'s performance under the data consistency of the Ctrip e-commerce platform (MSECP). As shown in the Fig. 6.2 below, the experimental case of MSECP contains a total of three micro-service modules: COMS, CSMS, and AMS. In the beginning, the customer initiates a request to create a new order. As the initiator of COMS, RPC remotely calls the gateways in CSMS and AMS to complete the order delivery and deduction. The CSMS and AMS then return the results of the operations in the respective service units. Finally, COMS returns the final response to the customer based on the returned results. If the operation is all successful, the order is created successfully. Otherwise, the order creation fails and the data is rolled back immediately. The MSECP is based on the spring cloud micro-services framework [24]. Finally, we deploy three micro-service module clusters.

6.5.3 RT Experiment

The SAOLA algorithm in 2PC+ needs to determine the performance improvement after optimization based on the response time (RT) of the service. In this test experiment, the createOrder interface was called concurrently with 10, 20, 50, 100,

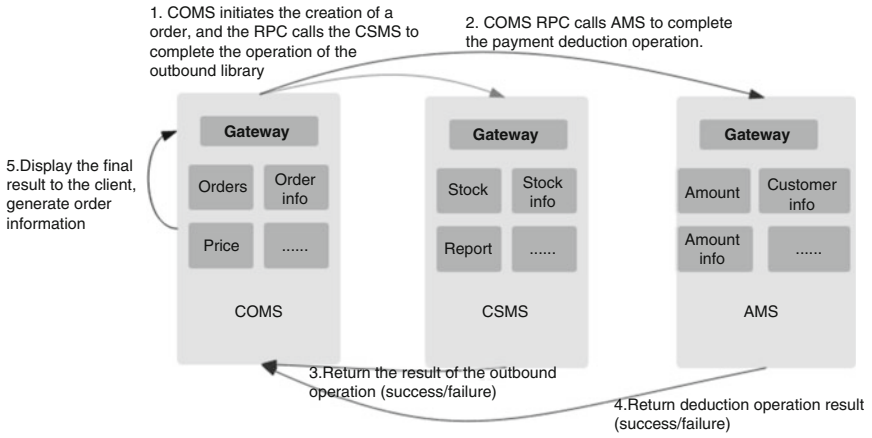

Fig. 6.2 A case of Ctrip MSECP

200, 300, and 500 sets of threads. A result of ten sets of comparison experiments was performed to calculate the RT mean values. The experimental results are shown in Fig. 6.2.

According to the analysis of the experimental results, when the number of threads concurrency is between 10 and 50, 2PC+ does not show a significant advantage on RT. However, when the thread concurrency reaches 100, it is shortened from the original 241.7ms to 83.4ms, which is 34.5% of the original RT duration. As thread concurrency continues to grow, the RT performance of the optimization scheme becomes more apparent. When the number of thread concurrency reaches 300, the RT value of the original 2PC is 820.5ms, and the RT value of 2PC+ is only 217.6ms, which is 26.5% of 2PC time. When the number of thread concurrent requests reaches 500, the 2PC's RT value is 1357.8ms, and 2PC+ is shortened to 473.6ms, which is 34.8% of the original time.

In summary, in the higher concurrent thread request scenario, the optimization algorithm in 2PC+ performs well in RT performance. Compared with 2PC, the RT performance is improved by 2.87 times to 3.77 times.

6.5.4 TPS Experiment

Similarly, transactions per second (TPS) is also one of the indicators for evaluating performance. The experimental results are shown in Fig. 6.3 below.

Through the experimental results, it can be known that when the number of concurrent threads is less than 50, 2PC+ has no obvious advantage in TPS performance. However, as the concurrency of threads increases, the TPS performance advantages of the optimization scheme gradually emerge. When the number of concurrent threads is between 100 and 200, 2PC+ can be maintained at TPS between 627.0 and

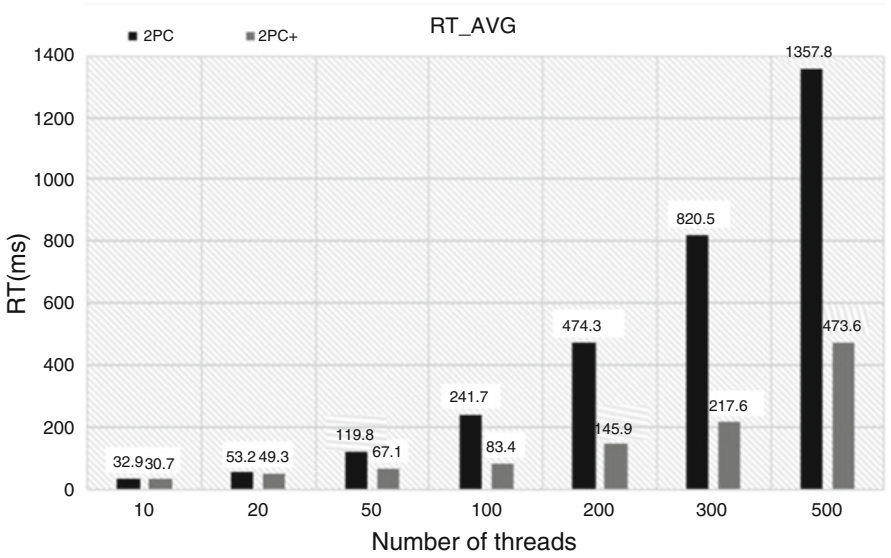

Fig. 6.3 RT experiment comparison result

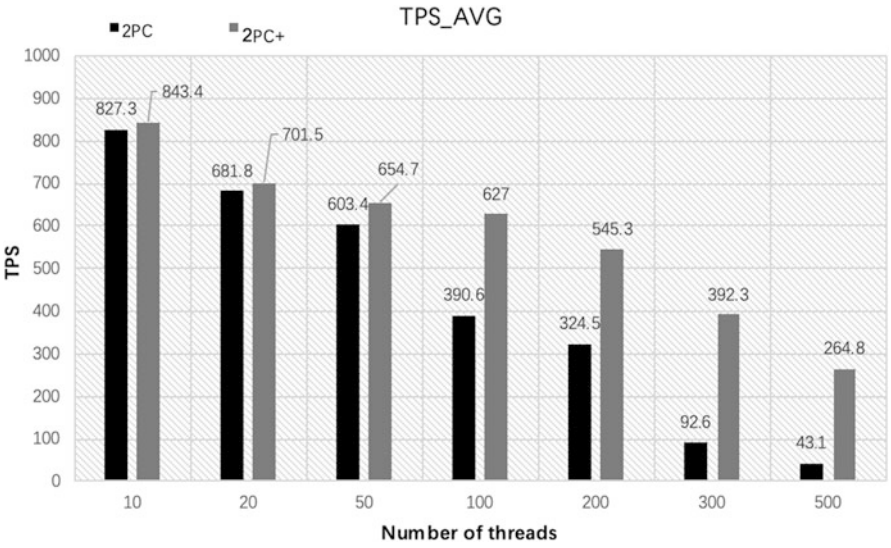

Fig. 6.4 TPS experiment comparison result

545.3, which is about 60.5% to 68.0% higher than that of 2PC's 390.6 and 324.5. When the number of concurrent threads reaches 300 to 500, the TPS of 2PC+ can be maintained at 392.3 to 264.8, which is 323.7% to 514.4% higher than the TPS of 92.6 and 43.1 in 2PC (Fig. 6.4).

As a summary, 2PC+ performs equally effective in TPS performance in higher concurrent thread request scenarios. Especially when the number of concurrent threads reaches 300 to 500, the TPS performance optimization is roughly 4.24 times to 6.14 times that of 2PC.

6.5.5 Related Work

Transaction of distributed database In academia, Kallman R et al. implement distributed transactions in H-Store memory data through serialization methods [9]. Later, Bailis P et al. developed a high availability solution to solve common faults such as network delays and partitions in distributed transactions [9]. In 2014, Mu S et al. developed a more concurrency control protocol ROCOCO [10] based on 2PL protocol and OCC, which has higher performance in dealing with distributed transaction conflicts. Guerraoui R et al. pointed out that the core rules for distributed transaction commit [11], that is, atomicity, must meet the conditions for the final agreement of all nodes in a distributed system. In the industry, Internet companies including Google, eBay, Alibaba, and PingCAP have been developing transaction solutions in distributed systems in recent years. In 2012, Google released the Spanner distributed database [12] and then in 2017 released the world's first commercial cloud data support for distributed transactions Cloud Spanner. Since then, Alibaba has developed the distributed database OceanBase [13] based on Spanner's design ideas, which solves the problem of data consistency and cross-database table transactions in distributed systems. It has extremely high processing performance. PingCAP [14] also released a distributed database VoltDB [15], which supports horizontal elastic extension, ACID transaction, standard SQL, and MySQL syntax and protocol, with high data consistency and high availability, and can support distributed transactions.

6.6 Conclusion

This paper presented 2PC+, a novel concurrency control protocol for distributed transactions in micro-service architecture. 2PC+ optimizes the synchronization blocking situation of transaction threads and reduces the probability of conflict between transactions due to high concurrency in micro-service architecture. And through the specific experimental data verification, compared to 2PC, 2PC+ has more efficient performance in RT and TPS.

Acknowledgment Our deepest gratitude goes to the anonymous reviewers for their valuable suggestions to improve this paper. This paper is partially supported by funding under National Key Research and Development Project 2017YFB1001800, NSFC Project 61972150, and Shanghai Knowledge Service Platform Project ZF1213.

References

1. B. Familiar, Microservice architecture [J] (2015)
2. F. Rademacher, S. Sachweh, A. Zündorf, Analysis of service-oriented modeling approaches for viewpoint- specific model-driven development of microservice architecture [J] (2018)
3. B.W. Lampson, D.B.A Lomet, New presumed commit optimization for two phase commit [C]. in *International Conference on Very Large Data Bases* (1993)
4. Qi Z, Xiao X, Zhang B, et al. Integrating X/Open DTP into Grid services for Grid transaction processing [C]. in *IEEE International Workshop on Future Trends of Distributed Computing Systems* (2004)
5. D. Peng, F. Dabek, Large-scale incremental processing using distributed transactions and notifications. in *Operating Systems Design and Implementation* (Oct. 2010)
6. A Dey, A Fekete, R Nambiar, et al., YCSB+T: Benchmarking web-scale transactional databases [C]. in *IEEE International Conference on Data Engineering Workshops* (2014)
7. S.J. Mullender, A.S. Tanenbaum, A distributed file service based on optimistic concurrency control [M]. *ACM SIGOPS Operating Systems Review* (2017), pp. 51–62
8. T. Neumann, T. Mühlbauer, A. Kemper, Fast serializable multi-version concurrency control for main-memory database systems [C].in *ACM Sigmod International Conference on Management of Data* (2015)
9. P. Bailis, A. Davidson, A. Fekete, et al., Highly available transactions: Virtues and limitations (extended version) [J]. Proc. Vldb Endowment **7**(3), 181–192 (2013)
10. S Mu, Y Cui, Y Zhang, et al., Extracting more concurrency from distributed transactions [C]. in *Usenix Conference on Operating Systems Design & Implementation* (USENIX Association, 2014)
11. R. Guerraoui, J Wang, How fast can a distributed transaction commit [C]. in *The 36th ACM SIGMOD-SIGACT-SIGAI Symposium* (ACM, 2017)
12. J.C. Corbett, J. Dean, M. Epstein, Spanner: Google's globally-distributed database [C]. in *Proceedings of OSDI* (2012)
13. OceanBase. https://oceanbase.alipay.com/
14. TiDB. https://pingcap.com/
15. D. Bernstein, Today's Tidbit: VoltDB [J]. IEEE Cloud Comput. **1**(1), 90–92 (2014)