Fan et al. Journal of Cloud Computing: Advances, Systems and Applications (2020) 9:40
https://doi.org/10.1186/s13677-020-00183-w

Journal of Cloud Computing: Advances, Systems and Applications

RESEARCH Open Access

2PC*: a distributed transaction concurrency control protocol of multi-microservice based on cloud computing platform

Pan Fan[1], Jing Liu[1*], Wei Yin[2], Hui Wang[2], Xiaohong Chen[1] and Haiying Sun[1]

Abstract

The two-phase commit (2PC) protocol is a key technique for achieving distributed transactions in storage systems such as relational databases and distributed databases. 2PC is a strongly consistent and centralized atomic commit protocol that ensures the serialization of the transaction execution order. However, it does not scale well to large and high-throughput systems, especially for applications with many transactional conflicts, such as microservices and cloud computing. Therefore, 2PC has a performance bottleneck for distributed transaction control across multiple microservices. In this paper, we propose 2PC*, a novel concurrency control protocol for distributed transactions that outperforms 2PC, allowing greater concurrency across multiple microservices. 2PC* can greatly reduce overhead because locks are held throughout the transaction process. Moreover, we improve the fault-tolerance mechanism of 2PC* using transaction compensation. We also implement a middleware solution for transactions in microservice support using 2PC*. We compare 2PC* to 2PC by applying both to Ctrip MSECP, and 2PC* outperforms 2PC in workloads with varying degrees of contention. When the contention becomes high, the experimental results show that 2PC* achieves at most a 3.3x improvement in throughput and a 67% reduction in latency, which proves that our scheme can easily support distributed transactions with multi-microservice modules. Finally, we embed our middleware scheme in the PaaS cloud platform and demonstrate its strong applicability to cloud computing through long-term analysis of the monitoring results in the cloud platform.

Keywords: 2PC, Distributed transactions, Microservices, Protocol optimization, Cloud computing, PaaS platform

Introduction

With the rapid and iterative development of Internet products, the traditional monolithic-service architecture is not suitable for the current large-scale data business scenarios. Therefore, microservice architecture has gradually replaced the traditio-nal approach and has become an important topic in industrial and academic circles. Microservice architecture [1] is widely used in large-scale cloud computing platforms and applica-tion development. The key to microservice architecture is to provide flexibility for program development and the reuse of fine-grained services. Microservices can be developed by different-domain teams to support business applications, and they can be implemented in various languages and access multiple underlying databases. In a distributed system, we usually deploy various microservice modules, which are homogeneous or heterogeneous systems composed of service clusters. Meanwhile, most microservices are used in cross-service and cross-resource scenarios. In other words, they need to access multiple databases in different environments when processing business. When an application invokes multiple microservices, distributed transactions are needed to support the consistent updating of these underlying databases and to ensure data consistency throughout the system. In recent years, academia and industry have carried out

* Correspondence: jliu@sei.ecnu.edu.cn
[1]East China Normal University, Shanghai 200062, China
Full list of author information is available at the end of the article

© The Author(s). 2020 **Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

conside-rable amounts of research on the data consistency of certain distributed databases, such as Spanner [2] and OceanBase (https://oceanbase.alipay.com/). Fortunately, microservices are closely related to these distri-buted databases in transactions, although supporting data-consistent distributed transactions across multi-microservice modules is even more challenging research [3].

The traditional technique is to implement a distributed transaction using the two-phase commit (2PC) protocol [4]. Unfortunately, this does not work well in large-scale and high-throughput systems, especially for applications with a large number of transaction conflicts [5]. The reason is that locks are held throughout the 2PC process. However, the number of modules across microservices is large, and it is necessary to support users with extremely high numbers of concurrent requests. For this reason, due to the limitations of the 2PC protocol, the performance of the original business will be seriously reduced, possibly rendering it unusable. Other approaches include persistent message queue patterns for loosely coupled distributed transactions [6, 7], which require additional application logic to compensate for failed transaction steps, thus increasing the cost of the system and possibly affecting the experience of users. In many business scenarios, such as e-commerce and e-finance, to ensure that the entire system always has strong data consistency, it must be controlled with the strictest consistency protocols, e.g., 2PC. Although systems such as YugaByte [8] and Found-ationDB [9] can support distributed transactions for a single database, this is not yet scalable in distributed systems with multi-microservice modules.

In view of the above problems, we aim to find a solution suitable for consistently distributed transactions in the micro-service architecture. On the one hand, it is strictly required to meet the basic principles of data consistency in the hetero-geneous distributed system of the microservice architecture. On the other hand, the solution must achieve the same level of throughput as the original microservice business. In other words, it is expected to obtain a high processing performance under the scenario of an extremely high number of concurrent user requests.

In the paper, we propose 2PC*, which is a novel distributed transaction control protocol that can extract more concurrent processing capabilities under high-intensity competitive wor-kloads than previous approaches for a multi-microservice. 2PC* is an optimized protocol based on the traditional 2PC. It utilizes a two-level asynchronous lock to reduce the over-head of synchronous blocking caused by a surge in the num-ber of transactions, thereby avoiding deadlocks. To achieve this, we propose a novel optimistic lock, i.e., the SAOL. Additionally, 2PC* uses a runtime protocol for transaction concurrency control to reduce the probability of conflicts between transactions.

In a distributed system environment where microservices are located, especially in cloud computing platforms, there are uncontrollable factors in service, e.g., service loss and network delays. Moreover, microservices are often called remotely through a gateway, such as XML-RPC or SOAP (simple object access protocol), which increases the proba-bility of these conditions occurring. Therefore, we improve the transaction compensation mechanism to achieve the ulti-mate consistency of distributed transactions across micro-services.

Finally, we implement a middleware solution for transa-ctions distributed across microservices based on 2PC* and deploy it on a specific PaaS cloud platform. Specifically, we use the Netty framework [10] to complete RPCs (remote procedure calls) [11] between transaction roles. Netty is based on Java NIO (non-blocking input and output), so its I/O operation is asynchronous and non-blocking. Therefore, the throughput and stability of RPCs can be greatly improved. We adapt our middleware to two popular microservice frameworks, Spring Cloud and Dubbo [12, 13]. We implement 2PC* and evaluate its performance using a case of the MSECP platform. 2PC* outperforms the original 2PC in workloads with varying degrees of contention. When the contention is high, 2PC*'s throughput is nearly 10 times that of 2PC. As the system scales across microservices and cont-ention increases, the throughput of 2PC* continues to grow, while the 2PC throughput drops to almost zero with no capacity to scale.

The rest of this paper is organized as follows. Section "Overview" elaborates on the overview of our scheme and the classic approach. In Section "Design", we introduce the design details of the 2PC*, including the design of SAOL and a runtime protocol. The implementation of the middleware solution based on 2PC* will be introduced in Section "Implementation". In Section "Evaluation", we will give the experimental data of the middleware solution implemented with 2PC* and compare it with 2PC. Section "Related Work" discusses some related work. Finally, Section "Conclusion" concludes this paper.

Overview

Low performance traditional approaches

2PC and OCC

In consistent transactional processing for traditional distributed databases, developers prefer the trans-action's strongest isolation level, such as serializability [14], to simplify the correctness criteria for concurrent transactions. Therefore, to ensure strict serializability, trad-itional distrib-uted storage systems usually run standard transactional concurrency control schemes, such as the

OCC (optimistic concurrency control) [15] combines with 2PC.

Unfortunately, 2PC and OCC perform poorly under com-petitive workloads in large-scale conflict transactions. We introduce a case that simulates the process of a customer buying two items from a store (Table 1). The process contains two fragments, F_1 and F_2, each of which reduces the stock quantity of different items. Each fragment can be executed atomically on the server where it is located. However, in the entire distributed system, an additional distributed transac-tion control protocol is needed to prevent fragmented transa-ctions across servers from being non-serializable and interla-ced. For example, suppose the store keeps the stock quantities of $item_1$ and $item_2$ unchanged and always sells the two items in a bundle. In the absence of a distributed transaction control protocol, the user can purchase $item_1$ but not $item_2$, while another user can purchase $item_2$ but not $item_1$.

We evaluate the performance of 2PC combined with OCC with two transactions, T_1 and T_2. Both purchase the same $item_1$ and $item_2$ stored on different services. When OCC detection is performed during the execution of 2PC, any interleaving of T_1 and T_2 will cause the process to abort. For example, if T_2 reads the stock number after T_1 reads it but before T_1 commits its update to $item_1$, T_2 will not be able to verify the process and abort later. We introduce another example where both T_1 and T_2 are aborted during 2PC because their corresponding 2PC *precommit* instructions are processed by the service in different orders.

However, the performance of 2PC combined with OCC under high-intensity workloads is far from satisfactory, especially across high-concurrency microservices. 2PC acqu-ires locks on data access for each transaction. When threads perform a conflicting operation, they serialize the transac-tions' execution order. In the example described above, once T_1 modifies the stock quantity of $item_1$, T_2 must be blocked until T_1 completes its entire process and commits successfully. In addition to blocking, 2PC also prevents deadlock by pass-ive thread abort [2]. However, as the number of competing threads increases, so does the probability of deadlocks. In addition, effective deadlock prevention mechanisms [16] (e.g., wound-wait) have many false positives. As a result, even without a real deadlock, most threads will still be abort-ed unexpectedly.

Table 1 A fragment of new-order transaction containing two pieces

transaction *new_order_fragment:*

#simplified new-order "buys" one of $item_1$, $item_2$

input: $item_1$ and $item_2$

begin

F_1: *# reduce stock level of $item_1$*

Read(tab = "Stock", key = *$item_1$*) ⟶ stock if (stock > 1):

Write(tab = "Stock", key = *$item_1$*) ⟵ stock - 1 ...

F_2: *# reduce stock level of $item_2$*

Read (tab = "Stock", key = $item_2$) ⟶ stock if (stock > 1):

Write (tab = "Stock", key = $item_2$) ⟵ stock - 1 ...

end

New distributed Transaction's characteristics in microservice architecture

ACID [17] is a design concept for transactions in traditional databases to ensure the correctness of data and avoid errors such as Read-Committed and Repeatable-Read. However, in distributed systems, especially at the application level, it is more important to meet business requirements than to pursue strict system characteristics. According to the CAP principle, strong consistency (C), availability (A), and partition tolera-nce (P) [18] cannot be met simultaneously. However, BASE theory adopts a completely different design idea than ACID. BASE sacrifices strong consistency for high availability and eventual consistency [19], which can be achieved through appropriate methods and is consistent with the characteristics of distributed systems in reality. On this basis, distributed transactions are mostly focused on the application layer for microservices. Therefore, it is necessary to not only ensure the data's eventual consistency but also obtain high availab-ility in the system.

2PC* optimize lock in two-phase commit

According to the details described below, our optimized transmission control protocol 2PC* avoids lock-blocking du-ring the two-phase commit between transactions, especially under scenarios with high-level contention.

2PC* optimizes the inefficient synchronization-blocking lock in 2PC and replaces it with a novel secondary asyn-chronous optimistic lock (SAOL). Borrowing from the design of the MVCC (multi-version concurrency control) [20], the SAOL allocates an ever-growing sequence of ver-sions to each transaction step, which is similar to the *snapshot*. The fine granularity of locks can be broken down into two levels of optimistic locks, borrowing from OCC (optimistic concurrency control) [15]. The SAOL allows multiple transactions to perform updates to the same transaction frag-ment concurrently, with two specific snapshots controlling the order in which transactions are executed, i.e., *begin-Version* and *commitVersion*. Meanwhile, the SAOL adopts a two-level optimistic lock (i.e., one composed of a *firstLock* and *second-Lock*) to control the transaction commit. Using the

BASE mechanism, the *firstLock* is responsible for controlling the resources in the main process between multiple transactions; thus, the *secondLock* can separate from it and compensate for the unfinished process in the *firstLock*.

2PC* simplifies conflicts to commit

2PC* is able to change and optimize the order of execution for transactions because it uses two phases of indicators to commit. In the *beginning* phase, when multi-microservice modules participate in the same distributed transaction, 2PC* does not immediately execute the subsequent process but instead adds the order relations (i.e., conflicts) between trans-actions to the *neighbourList* of the directed graph, which can be denoted as *GraphNode*. We conduct a preliminary merge and de-duplication of conflicts for the *neighbourList*. In the *commit phase*, *GraphNode* then combines all the conflict information and distributes it to all the microservices. 2PC* then further simplifies the transaction conflicts for the *neighbourList* so that it performs better under high-competition transaction scenarios, such as those involving microservice architecture.

A middleware based on cloud platform for distributed transaction control with 2PC*

Based on 2PC*, we implement a middleware scheme that supports consistent distributed transactions for microservices. Our prototype contains over 23,000 lines of Java code, 17000 of which are for transaction concurrency control, and is based on the Spring-Boot framework. In particular, we adopt the Netty framework to complete the underlying communication between transaction roles. We deploy it in a specific PaaS cloud platform. Experimental data prove that our scheme has very good performance for multi-microservice scenarios with high concurrent requests from users, including higher throu-ghput and lower latency.

Design

The design of the 2PC* protocol includes a novel optimistic lock mechanism (i.e., the SAOL), a concurrency control prot-ocol for transactions and a compensation measure.

In this section, we first explain the design of the SAOL. Then, we introduce a concurrency control protocol in 2PC*, an optimizing strategy, and a verification of its correctness. Finally, we provide a fault-tolerant mechanism.

Secondary asynchronous optimistic-lock

The novel 2PC*'s SAOL replaces the synchronous blocking lock with a high-performance secondary asynchronous lock. The process in each transaction fragment is identified by a unique version number, i.e., *snapshot*. We adopt Twitter's distributed identification number generation algorithm *Snow-flake* [21], which contains a timestamp to identify the execu-tion order of the transaction fragment. The specific design of the SAOL is described below.

Transaction property initialization

In the SAOL, we first define four key attribute fields in the transaction object, which are represented as follows.

- ***value***: This indicates the current actual value of the transaction object.
- ***beginVersion***: That is the version sequence number at the beginning of the process.
- ***commitVersion***: That is the version at the time the transaction was committed.
- ***lock***: This is responsible for locking up the resources of uncommitted transactions. The *lock* is divided into two different levels, namely the *firstLock* and the *secondLock*, which represent the two phases of *lock* respectively. And we specify that in the *secondLock*, it needs to contain *firstLock* information.

Begin transaction process

First, we need obtain the *begin-Version* in the transaction object T_1, denoted *b_v*, and then determine whether there is a lock in T_1. If no lock exists, we try to obtain the latest committed transaction object directly from the version number interval [0, *b_v*] and obtain the current latest value through its beginVersion. Otherwise, the subsequent process is executed according to the following three branches.

- *Case 1*. If there is another transaction object T_2 committing a transaction, then the value of T_2 is locked. At this point, we need to wait for T_2 to finish committing transactions, then poll and retry to obtain the current latest *value*.
- *Case 2*. In *case 1*, if the waiting time of T_1 has passed a certain threshold, denoted WAIT_TIME_OUT, but T_2's *value* is still locked, it can be determined that *T*2 is facing some unexpected exceptions, e.g., a network delay or downtime. We can simply assume that T_2 has been interrupted and it can release *lock* directly.
- *Case 3*. T_2's *lock* may have been remained because it has not been released properly. In this case, T_2's transaction was committed and its *firstLock* released successfully, but some unforeseen exceptions occurred in the *secondLock*, so the transaction could not be committed, thus causing the *lock* to remain.

Transaction pre-commit process

At this point, T_1 has completely obtained its latest *value*, so we can perform the T_1 *first-phase* commit, denoted preCommit. This process can be divided into three branches as follows.

- *Case 1*. For any two transaction objects T_1 and T_2, in the version sequence interval (b_v, + ∞) corresponding to their *value*s, we determine whether another transaction object T_x is updating the data. If this condition matches, it means that T_x might have changed the *value*. At this point, both T_1 and T_2 need to roll back their transactions directly, and then we can complete the process.
- *Case 2*. Determine whether a *lock* exists in both T_1 and T_2, that is, whether their transaction resources are locked. If so, T_1 and $T2$ need to directly roll back transactions and we can terminate the process directly.
- *Case 3*. If neither of the above two cases matches, we set the *firstLock* of T_1 and T_2 to locked, write the latest data to *value* at this time and commit the transaction.

Transaction second-commit process

First, we determine whether the *firstLock* of T_1 or T_2 is locked; if so, we commit the transaction directly. The *secondLock* that belongs to T_1 and T_2, it can be completely separated from the main process, and we can then use an asynchronous mode of the thread to release the *secondLock* and commit transactions in the second phase. Therefore, even if an exception has been occurred throughout T_1 or T_2, as the subsequent transaction object, denoted $T_{\text{next,}}$ it observes that the *firstLock* belonging to T_1 or T_2 has been released, but the *secondLock* still unexpectedly remains, thus T_{next} will automatically release the lock in their *secondLock*s and commit the transactions.

Concurrency control protocol

Under the microservice architecture, business modules are often deployed on multiple machines in a distributed cluster to achieve scalability and high availability. When multiple microservices act as participants and concurrently execute the same group of distributed transactions, conflicts are inevitable, and they greatly affect the performance between concurrent transactions. To reduce the probability of conflict between concurrent transactions, a novel concurrency control protocol is proposed based on the 2PC in this subsection. Our scheme is able to avoid the additional performance overhead caused by conflicts between frequent transactions.

Similar to 2PC, the design of the basic protocol is divided into two phases, i.e., the *begin phase* and the *commit phase*, which are summarized below.

Begin phase

When the distributed transaction process offici-ally begins, the *TCM* (transaction coordination manager) generates a globally unique transaction number TID for the transaction group. Then, this transaction group does not imm-ediately perform the subsequent process but instead caches it in a temporary area (such as Redis). Finally, we determine whether the transaction group matches a specific condition (which will be described in detail later); if so, the next step is performed. The key to the protocol is to maintain a graph data structure in the appropriate microservice module *MS*, which is used to record all conflict dependencies from the transact-tion group and is denoted as *GraphNode*. Its implementation code is shown in Table 2, in which *data* is the generic Java type corresponding to vertex *E* in the Graph, and it records the key information for transaction object *T*, including the *status* of the transaction, which can be summarized as the three kinds of situations below.

- BEGUN. This indicates that the transaction has started execution.
- COMMITTING. This indicates that the transaction is performing the committing process.
- FINISHED. This indicates that the workflow of each transaction has been determined.

The execution order of these statuses needs to be strictly guaranteed to be serial. Thus, we restrict BEGUN to be earlier than COMMITTING and COMMITTING to be earlier than FINISHED; i.e., BEGUN < COMMITTING < FINISHED.

The field *visited* of *GraphNode* is set to the Boolean type, and it indicates whether transaction object T is finished with the corresponding MS. It is TRUE if done, FALSE otherwise.

The *neighbourList* of *GraphNode* corresponds to the subset of V*V in the direct graph, denoted *E*, which is represented by the ArrayList type in Java. The sequence of all concurrent transactions in the *neighbourList* is recorded. For example, the *MS* may receive a transaction request initially from transaction T_1 and then accept another request from T_2. Therefore, T_1 and T_2 are prevented from participating in the same group; they conflict and need to be added to the *neighbourList*.

Finally, when *T* prepares to participate in the transaction, *MS* does not immediately perform subsequent logical operat-ions but first enumerates all other transaction objects *T'* that collide in parallel with *T* and adds them to the *neighbourList*. The *neighbourList* is

Table 2 GraphNode data structure

Algorithm 1 GraphNode.class

```
class GraphNode <T> {
    T data;
    List<GraphNode<T>> neighbourList;
    Boolean visited;
}
```

then loaded into the *TCM* to perform initial processing of these conflicting transactions, including aggregation and deduplication (Table 3). Finally, we save the *neighbourList* to the corresponding *GraphNode*.

Commit phase

After the *begin phase*, the conflicts of all transaction objects have been saved in the *TCM* and recorded in the corresponding *GraphNode*. The vertex data and the edge *neighbourList* belonging to the *GraphNode* have executed preliminary data aggregation processing. The *TCM* further aggregates the *status* of all transaction objects to preserve the latest version. As shown in the pseudo-code (Table 4), we first determine whether the *status* of the local *T* is BEGUN. If so, the status can be updated to COMMITTING and synchronized to the local *GraphNode*.

We now describe the judgement condition mentioned in the *begin phase*, that is, whether the transaction object *T* needs to be reordered. We first calculate the ancestor T_{root} of *T*, and then wait for the *status* of T_{root} to be set to COMMITTING or FINISHED. This is summarized in the following two situati-ons:

- T_{root} is in the *MS* where *T* is located. The *MS* will eventually receive the transaction request of T_{root} through the *TCM*, and no additional operations are required to wait for the request.
- T_{root} is not in the *MS* where *T* is located. At this point, the *MS* needs to initiate an inquiry request to the *MS'* where T_{root} is located. When the *MS'* observes that the *status* of *T'* is COMMITTING or FINISHED, it responds to *MS* and returns the *GraphNode* to the *MS*.

The *while* operation is performed in both cases until all ancestors' statuses of the transaction object *T* in the *MS* are set to COMMITTING or FINISHED to jump out of the loop. We calculate the *MS*'s *strongly connected component* (*SCC*) through the *GraphNode*. According to the definition of the *SCC*, in a directed graph *GraphNode*, for each pair of vertices V_i and V_j (V_i and V_j do not belong to the same vertex), it is guaranteed that there are paths from V_i to V_j and V_j to V_i.

We utilize the classic *Tarjan* algorithm [22] to calculate the *SCC* of the *GraphNode*. The core design is to maintains the graph based on the *depth-first search* (*DFS*) algorithm, and each *SCC* is a *subtree* in the *search-tree*. The nodes belonging to the *DFS* that are not traversed are added to a stack. When backtracking, we determine whether the *top-to-middle* node is a *strongly connected component*. In the best case, only the vertices of the *SCC* belonging to the *GraphNode* are traversed, thus the time complexity is no more than O(V). In the worst case, all the vertices and edges in the *GraphNode* need to be traversed in turn, with time complexity is O(V + E).

Table 3 Algorithm implementation of Begin phase

Algorithm 2 Begin

```
1: function Microservice MS: begin (t , GraphNode):
# t stands for a transaction object.
2: MS. GraphNode[t.group].status = BINGUN
3:   for t' received by MS
4:      if t' conficts with t then
5:         add { t'.group → t.group} to MS. GraphNode
6:         merge and de-duplication in MS. GraphNode
7:      end if
8:   end for
9: return  (MS. GraphNode , output )
```

Table 4 Algorithm implementation of Commit phase

```
Algorithm 3 Commit
1: function Microservice MS: commit (t , GraphNode):
# t stands for a transaction object, t' stands for another
transaction object, MS' represents the microservice that
initiates the request to MS.
2: for t' received by MS
3:   if MS. GraphNode[t.group].status == BEGUN then
4:       MS. GraphNode[t.group].status = COMMITTING
5:     if t' is not belong to MS then
6:         MS request MS'
7:     end if
8:     if (MS'. GraphNode[t'.group].status == COMMITTING
9:       || MS'.GraphNode[t'.group].status == FINISHED) then
10:          MS' response to MS
11:    end if
12:    # The SCC was obtained through Tarjan, denoted t_info
13:    t_info = getSccByTarjan (t)
14:      if t_info conflict with MS.otherScc then
15:        if ( MS. otherScc.status == FINISHED &&
16:          MS. GraphNode[t. ancestor].status ==FINISHED
17:              && t'.vistited == TRUE ) then
18:          MS decide execution order on t in cache
19:          t.visited = TRUE
20:        end if
21:      end if
22:    end if
23: return   ( MS.GraphNode , output )
```

The *SCC* of *T* is simply referred to as T_{SCC}. If there are no conflicts with the *MS*, the T_{SCC} contains only *T* individual nodes. Otherwise, the following three conditions must be met.

- The *MS* sets the *status* update in all transaction objects belonging to T_{SCC} to FINISHED.
- The *MS* waits for the *status* of other ancestors in *GraphNode* to be FINISHED.
- The *MS* waits for the *visited* in its associated ancestor T_{root} to be TRUE.

When all of the above conditions are met, the *MS* determines the eventual order based on the global transaction number TID in each *T*.

The *MS* sets the field *visited* in each *T* belonging to T_{SCC} to TRUE; this is executed sequentially in the order in which they are arranged. Finally, the output is sent to the *TCM*, and the *TCM* then notifies the transaction initiators of the final execution result. We can determine the time complexity of algorithm 3 as O(n*(V + E)) in the worst case and O(n*V) in the best case (where n stands for the number of *MS*s).

Table 5 Algorithm implementation of SimplifyConflicts

```
Algorithm 4 SimplifyConflicts
1: function simplifyConflicts (t, GraphNode):
# t stands for a transaction object, t'→t is a LRCD.
2: for {t'→t} to GraphNode
3:   if GraphNode contains ({t' ⇒ t}).length >
           ({t'→t}).length then
4:       remove { t'→t} from GraphNode
5:   end if
6: end for
```

Correctness

In this subsection, we present the correctness verification of 2PC*. We utilize the formal specification language TLA+ [23] to validate the transaction's locks through the protocol, ensuring that 2PC* is strictly serialized. We only show the core TLA+ of the protocol, a

more rigorous version of the TLA+ language is available at [24].

Constant

We set the two invariants in the transaction to constants, i.e., the data of all transaction participants, denoted as *value*, and the participant transaction object, denoted as *RM*.

Variable

We represent all transaction status in *RM* as the variable rm_status. The current version sequence number is represented by the variable rm_v, which includes a collection of all the states of the transaction, i.e., {"beginning", "preparing", "precommit", "committed", "cancel"}. We define the global version sequence ascend_v, which is a *snapshot* and is always increasing. The information about two locks in *RM* that control the version, firstLock and secondLock, is represented by the variable rm_lock.

Begin

At this point, the transaction status *rm_status* is at "beginning" and its next state is expected to be "preparing". The data *rm_value* in the transaction object is obtained by the *getVal* method.

```
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']
```

Loading

At this stage, *rm_status* is in the "preparing" state and its next state is bound to be "precommit". We determine whether the condition of resetting lock is met. If so, we then perform the *resetLock* process. Otherwise, the next state of *rm_status* is set to "cancel".

```
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"]
```

PreCommit

We present the TLA+ language for the *pre-commit* phase of the transaction. At this point, *rm_status* is "precommit", and we determine whether the *second-commit* of the transaction is performed, if so, *ascend_v* needs to be incremented, and next state of *rm_v* is restricted to be the latest value of *ascend_v*, and *rm_status*'s next state is limited to "committing". Otherwise, we need determine whether these transaction's values are all locked, if so, we perform the *allLock* method to lock all values. Otherwise, we set *rm_status* to "cancel".

```
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"]
```

SecondCommit

Finally, we present the TLA+ language for the *second-commit* of the transaction. We determine whether the first value of the transaction can be committed, i.e., *isCommitFirstVal,* if so, we can commit the transaction and then set *rm_status* to "committed". Otherwise, the transact-tion aborts and we set *rm_status* to "cancel".

```
SecondCommit(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"]
```

Next

The entire validation process follows the four steps above.

```
Next ==
 \E r \in RM:
     Begin(r) \/ Loading(r) \/ PreCommit(r) \/ SecondCommit(r)
```

Consistent

Finally, we present the consistency constraint for the transaction. These two constraints are met: the version sequence number *committed_v* for all committed transactions satisfies the ascending ordering rule, and the lock of *first_v* in *RM* can be released when the transaction committed succe-ssfully.

We run the complete TLA+ language in the TLC [25] tool and analyze the result as shown in Fig. 1. It generates 1296 states, of which 324 is distinct. More importantly, based on the results, i.e., "*No error has been found*", which can prove that the 2PC* protocol will not occur deadlock and ensure the strict serialization during the transaction process.

Simplifing NeighborList of GraphNode

As the number of transaction participant increases, the amo-unt of data stored in the *GraphNode* becomes increasingly cumbersome, which significantly affects the subsequent wor-kflow and potentially makes it unavailable. For example, the *Trajan* algorithm is used to calculate the *SCC* steps. However, much useless information

```
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. 1 TLA+ result with running in TLC

has been recorded in the *Graph-Node*, mainly related to the concurrent conflict dependencies of transaction objects, i.e., the *neighbourList*, which is irrel-evant for the subsequent process. Moreover, it is unnecessary to transfer the entire *GraphNode* between *micro-services*. To solve these problems, we optimize the runtime protocol further.

LRCD design

To simplify the neighbourList belonging to the GraphNode, only the *least recent conflict dependence* (*LRCD*) between transactions needs to be recorded Table 5. The *LRCD* is defined as follows: for any conflicting relationship path $T' \rightarrow T$ in *GraphNode*, if the path $T' \Rightarrow T$ does not exist in GraphNode, the number of paths $T' \Rightarrow T$ is not less than two, and $T' \rightarrow T$ is called an *LRCD* in *GraphNode*. According to the explanation in Fig. 2 below, $T1 \rightarrow T2$, $T2 \rightarrow T3$ is an *LRCD*, while $T1 \Rightarrow T3$ is not an LRCD; thus, $T1 \Rightarrow T3$ can be removed. **The location of** Table 5 Algorithm implementation of SimplifyConflicts

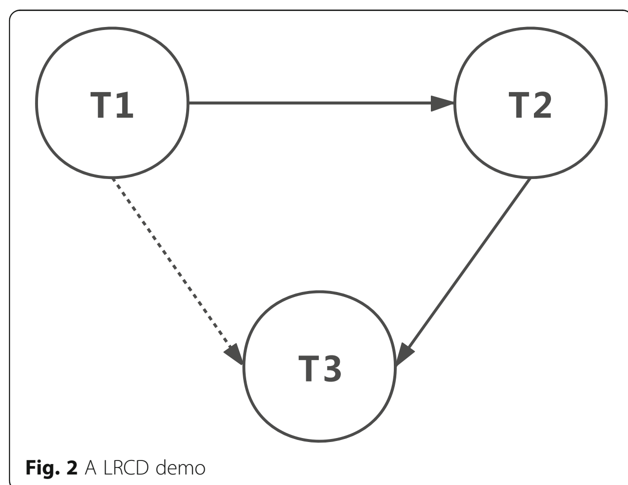

Fig. 2 A LRCD demo

In the original protocol, the following three locations need to be simplified in the neighbourList and saved only as *LRCD*s:

In the original protocol, the following three locations need to be simplified in the *neighbourList* and saved only as *LRCDs*:

- In the *begin phase*, the *TCM* received a response from the *MS* and then simplified the *neighbourList* that belonged to the response message.
- During the *commit phase*, the *MS* received a request from the *TSM*. The *MS* simplified the *neighbourList* in the request body with its local value.
- During the *commit phase*, the *MS* received an inquiry response from the other *MS'*. The *MS'* simplified the *neighbourList* in the request body with its local value.

Simplify MS swaps with GraphNode steps

For the *MS* and *TCM* corresponding to *T*, we only need to obtain the *GraphNode* of *T'*. Therefore, the *MS* and *TCM* only need to obtain the *SCC* whose *status* is FINISHED and whose *GraphNode* contains all transaction objects.

Similarly, when the *MS* receives the query request from *MS'*, the *MS* first detects the *status* of the *GraphNode* under the local environment. If the status is not FINISHED, we calcu-late the *SCC* for all the transactions contained in *GraphNode* belonging to the *MS*. Otherwise, if *status* becomes FINISHED, this means that *T* and its *SCC* have been obtained by the *MS* and its response is returned directly to *MS'*. The process is shown in pseudo-code in Table 6 below.

Fault tolerance

To achieve fault tolerance, we persisted the transaction logs in each coordination and transaction participant service to the disk. Moreover, we used the Paxos-based

Table 6 Algorithm implementation of SimplifySwapGraphNode

Algorithm 5 SimplifySwapGraphNode
1: **function** simplifySwap (t, GraphNode)
 # t represents a transaction object, t'→t is a LRCD, t'⇒ t is not a LRCD.
2: **if** MS. GraphNode[t.group].status == FINISHED **then**
3: t_info = *getSccByTarjan* (t)
4: **else if** ({ t'⇒ t} : { t'→t} *in* MS.GraphNode && MS.GraphNode[t'].status != FINISHED) **then**
5: t_info = { t'⇒ t }
6: **end if**
7: **end if**
8: **return** (GraphNode , output)

replication protocol [26] to synchronize the log data across multiple machines.

First, we execute a scheduled thread-pool task, denoted as *STPT*. *STPT* polls the logs in the disk at intervals. It filters out the key information in the failed transactions, including the TID that identifies the transaction, and the list of participating transactions. Then, we push the information into a *circle message queue* (*CMQ*). Finally, the *CMQ* repeats the request to the corresponding business method through the asynchronous polling until the response returns successfully.

The *CMQ* design is shown above (Fig. 3), it is a circular message queue with a high latency and has 2000 slots. Each transaction compensation can be regarded as a task that is added to the *Set* without repetition, i.e., Set<Task>. We then push the Set<Task> into the tail of the *CMQ*. There are two key pieces of information stored in *Set*, i.e., *layer_num* and *Function* < *Task*>. The *layer_num* represents the number of *CMQ* layers in which the task resides, and the *Function* < *Task* > indicates the target function of the task. The *CMQ* uses a pointer to specify the index of the currently running task, denoted *cur_index*.

Reliability and idempotency

We use a local message table to record the relevant data of the transaction compensation task, which includes the TID for the transaction, and the current state of execution (i.e., *status*). To ensure reliability, The *TCM* returns the result of the asynchronous message with the field status record and sets it to TRUE if successful and FALSE if not. Then, the failed transaction steps need to be pushed to the next task queue until it is compensated for successfully. To ensure idempotency, the *TCM* determines whether the message is duplicated by the TID. If the TID already exists and its *status* is TRUE, this step is skipped. Moreover, the *TCM* provides a remote message record query interface, thus the transaction participant can invoke this interface to determine whether the current message has been consumed. If so, this compensation process is skipped.

Implementation

We implemented a middleware solution with transactional in distributed microservices support using 2PC*. We design the annotation *@TxTransactional* to inject distributed transactio-nal functionality into specified microservices business with spring's AOP (aspect oriented programming), which achiev-es decoupling from the original business module. Obviously, our prototype consists of three functional modules, i.e., *transaction initiator*, *transaction actor*, and *transaction coordinator*. In particular, their network communication is implemented through high-performance Netty framework.

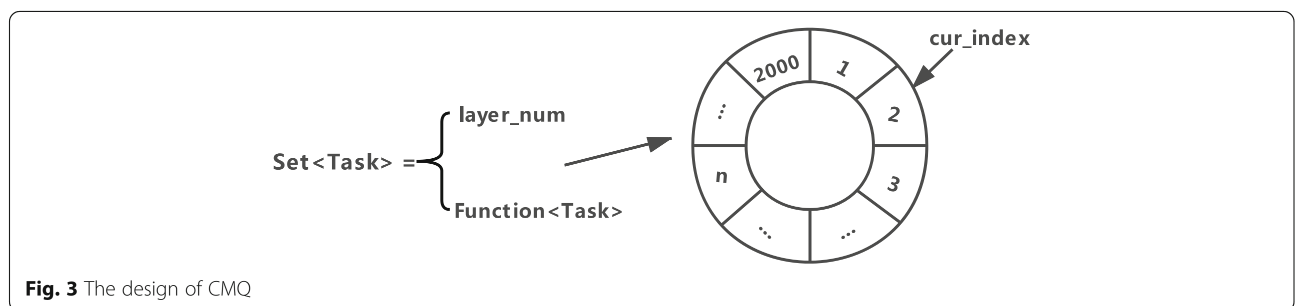

Fig. 3 The design of CMQ

Transaction initiator

The role of the transaction initiator is an active part of the process. In addition to creating transaction group information and executing the local transaction, it also notifies the coordi-nator to perform commit or rollback operations on the trans-action group (Fig. 4). The implementation process can be summarized as follows.

1) Generating the transaction group identifier TX_ID through the *snowflake* algorithm to ensure that it is unique in the entire distributed system, and then creating a new transaction group. If this step is successful, we can execute step 2. Otherwise, we just throw the runtime-exception and end the process.
2) We determine the transaction propagation mechanism type of the initiator. If it is PROPAGATION_NEVER, which means that there is no transaction requirement from the initiator. If it succeeds, we perform step 3. Otherwise, end the process. If the type of propagation is PROPAGATION_REQUIRES_NEW, it means that the new transaction was initiated. We similarly execute the transaction group's *pre-commit* process, and if successful, we perform step 4, otherwise, jump to step 5.
3) Following step 2 above, there is no transaction request in this initiator, and after performing the *pre-commit*, the coordinator is asynchronously notified to complete the *second-commit*. We adopt the *CompletableFuture* interface [27] provided by the JDK1.8. When these multiple threads attempt to complete or cancel it at the same time, only one thread is guaranteed to succeed. In practice, we use the *runAsync*, a method without a return value, to construct the initiator and coordinator as Netty transmission object in the asynchronous mode, then complete the *second-commit*'s notification step between the initiator and coordinator.
4) Similar to step 3, we also use the *CompletableFuture* to asynchronously notify the coordinator to execute the *second-commit* after the transaction group *pre-commit* succeeds. Because the transactional requirement of the initiator is newly opened, it must first commit the local transaction, thus we can adopt the *PlatformTransactionManager* (*PTM*) interface [28] provided by Spring to execute it.
5) In this situation, the transaction group failed during the *pre-commit* phase and the initiator's transaction is newly opened. Then, we perform the local transaction rollback, executing the *PTM*'s rollback function and closing the process.
6) If some unexpected exceptions occurred in steps 4 or 5, as shown in the dotted box of Fig. 4. First, we use the *PTM*'s rollback function to complete the local transaction rollback. The coordinator is then asynchronously notified by *CompletableFuture* to complete the rollback of the transaction group.

Transaction actor

The transaction actor is a passive role in distributed transacti-on processing. Its main functions include adding the micro-service's business module to the distributed

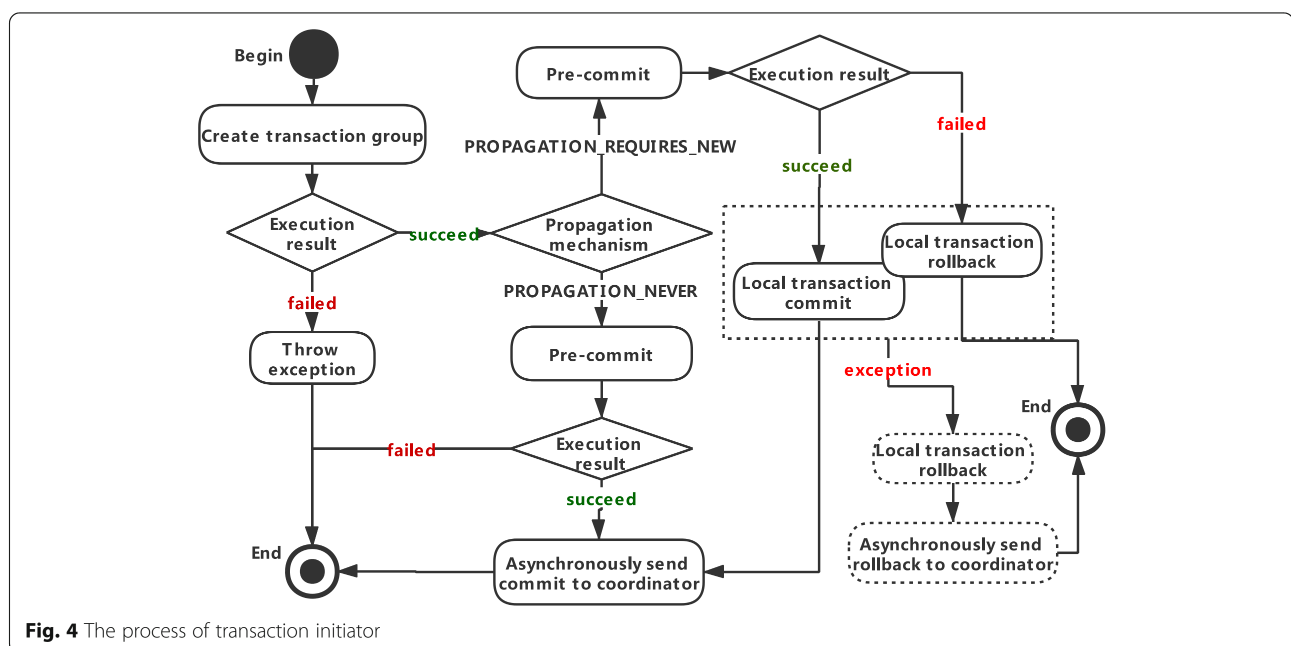

Fig. 4 The process of transaction initiator

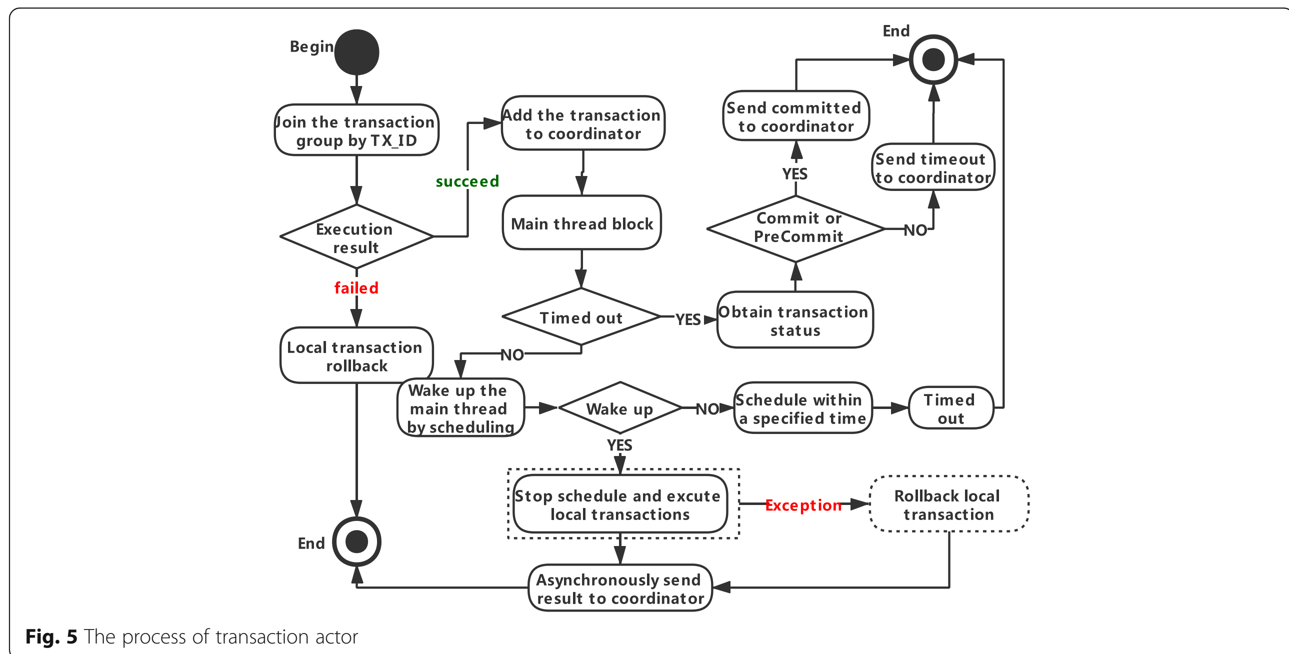

Fig. 5 The process of transaction actor

transaction group and completing local transactions through the coordinator's instruction (Fig. 5). The process of the transaction actor can be summarized as follows.

1) The actor joins the corresponding transaction group according to TX_ID. If the execution fails, the local transaction needs to rollback and then the process should be terminated. Otherwise, it initiates an add-to-transaction request to the coordinator. It's similar to the initiator, the communication between actors and coordinators is also based on Netty.
2) At this point, the initiator's thread is blocked. We use the interface provided in JDK called *ReentrantLock* [29] to lock the main thread and wake it up in combination with Condition's signal method. The initiator then waits for the coordinator's response within the specified time threshold, denoted WAIT_TIME_OUT. If the response time is no more than WAIT_TIME_OUT, step 3 is executed directly. Otherwise, we need to perform step 6.
3) We then create a scheduled task to wake up the main thread at specified intervals within the time threshold denoted TASK_TIME_OUT. If this process is successful, the task can be closed and step 4 is executed. Otherwise, we need to skip to step 5.
4) Following step 3, the subsequent process is based on the coordinator's response, which is obtained from the thread's asynchronous callback function. If the response is second-commit, we first commit the local transaction, and then the output is asynchronously notified to the coordinator via Netty. Otherwise, the local transaction needs to rollback, and then also asynchronously notify the coordinator of the result through Netty.
5) Continuing with step 3, we repeat the scheduling task for the specified time TASK_TIME_OUT until it succeeds. Otherwise, a timeout occurs and the process can be terminated.
6) Next, following step 2, the coordinator's response is timed out. The initiator needs to proactively obtain the transaction group's status through Netty. If the status is either pre-commit or second-commit, the transaction group has successfully committed and then the actor sends the commit notification asynchronously to the coordinator. Otherwise, there are some exceptions have occurred in the execution, and actor can asynchronously send the timeout exception notification to the coordinator. Finally, we need to wake up the main thread that is blocked by the *Condition*'s signal.
7) If some exceptions occurred to the initiator during the process of the local transaction's commit, as shown in the dotted box in Fig. 5, we should rollback this transaction and notify the coordinator asynchronously through Netty.

Transaction coordinator

The transaction coordinator is at the core of the hub in the distributed transaction processing. On the one hand, it deals with the corresponding business according to the notification requested by the initiator and the actor. On the other hand, it sends the instruction response to the

initiator and the actor at a specified time. We first design the transaction management interface *TransactionManangerService*, denoted *TMS*. *TMS* provides basic functionality related to the persistence of the transaction, i.e., CRUD (Create, Retrieve, Update and Delete). We then describe the transaction coordinator's two core functions, i.e., *precommit* and *rollback*.

PreCommit

The process can be divided into the following steps.

1) With the *updateItemStatus* method of *TMS*, we update the current transaction's status to COMMIT.
2) Through the *listByGroupId* method in *TMS*, we obtain the list of all transaction objects under the current transaction group number TX_ID, denoted *items*. Determine whether the *items* are empty. If so, it can terminate the process. Otherwise, we execute the next step.
3) Following the above steps, we now perform specific preliminary filtering of *items*. The filtering principle is that we remove the transaction objects that have been committed by the initiator from the *items*, that is, avoiding duplicate the communication between these transactions. By the filter function, we divide *items* into a list of transactions under the local domain environment, denoted *currentItems*, and another list under other domains, denoted *elseItems*.
4) Detecting whether Netty's channels of *currentItems* are activated. If so, we run the *excuteCommit* method to commit the transaction, otherwise, run the specific *excuteRollBack* method to complete the transaction rollback. The *excuteCommit* and *excuteRollBack* will be described later.

Rollback

Similarly, its process can be divided into the following steps.

1) We update the current transaction's status to ROLLBACK through the *updateItemStatus* method in *TMS*.
2) It is the same as described in step 2 of the *PreCommit*.
3) Similarly, it is the same as described in *PreCommit*'s step 3.
4) Finally, we execute the *excuteRollBack* method to complete the transaction rollback.

In the process of *PreCommit* and *Rollback*, they both need to run two specified methods, i.e., *excuteCommit* and *excute-Rollback*. Next, we describe the implementation details for each.

ExcuteCommit

This method applies to the distributed tran-saction's commit process that is divided into the following steps.

1) First, we iterate through the list of local transaction groups to be committed, i.e., *currentItems*.
2) Then, we build Netty's *ChannelBean* object and load it in the *HeartBeat*, and set the transaction status to COMMIT.
3) Determining if the channel in the *ChannelBean* object is empty. If so, we record the transaction object's TX_ID in the Error log. Otherwise, pushing the *HeartBeat* to *Queue* and refresh it.
4) Finally, we execute the remote request method with the *elseItems* and set the transaction status to COMMIT. Because these coordinators are clustered, thus the channels in *elseItems*' transaction objects may be connected to different coordinators. There are two functions in the remote request method. On the one hand, we observe the status of the transaction coordinator channel under the local domain and notify it to perform the transaction commit. On the other hand, we connect to the clusters of transaction coordinators under other remote domains and similarly notify them of committing transactions.

ExcuteRollback

It is responsible for the rollback process of distributed transactions, which can be roughly divided into the following steps.

1) First, we determine whether the *currentItems* are empty. If it matches, we just skip to the last step. We then load the list of transaction groups into the specified *ThreadPool* array, i.e., *CompletableFuture*, which asynchronously performs the tasks of the subsequent multi-transaction groups. To achieve this, we then execute the asynchronous method in the *CompletableFuture*, i.e., *runAsync*. Meanwhile, we build the Netty's *ChannelBean* object, load it into a *HeartBeat*, and set status to ROLLBACK.
2) Determining if the channel exists in the *ChannelBean*. If not, we just skip this step, otherwise, execute the *writeAndFlush* method for *ChannelBean*'s channel. Finally, we can push the *HeartBeat* object to *Queue* and refresh it.
3) Until all transaction objects have been loaded into Netty's channel and have been executed asynchronously through *CompletableFuture*. At this point, we execute the *allOf* method, which can acquire the execution result of all transaction objects.

4) Finally, the approach is roughly similar to the last step in *ExcuteCommit*. The only difference is that we set the transaction status to ROLLBACK and perform the transaction rollback.

Evaluation

Experimental setup

In order to minimize the extra impact of CPU's performance bottlenecks on the experiment, we chose higher performance machines to build service clusters. Each machine has an eight-core 2.7 GHz Intel Core i7 with 8GB RAM and 500GB SSD. Therefore, we have achieved much higher throughput when running on a local testbed with faster CPUs.

Experimental case

We applied our middleware solution based on 2PC* to the microservice case, that is, the online e-commerce transaction platform MSECP of an Internet company. In this paper's experimental case, we primarily consider three microservices, i.e., the *OrderMicroservice* (*COMS*), the *StockMicroservice* (*CSMS*), and the *AccountMicroservice* (*AMS*). The distribu-ted transaction process between the microservices can be summarized as follows (Fig. 6): A user initiates an order creation request from *COMS*, which then invokes *CSMS* and *AMS* through an RPC to complete the business of item out-bound and account deduction, respectively. Only when the business of these three microservice modules is successfully executed can we return a successful response to the user and commit transactions in the respective microservice modules. Otherwise, if exceptions occurred in one of these micro-services, the user is notified that the purchase failed; thus, the transactions of the respective microservices need to be rolled back immediately.

We adapted the middleware solution to two popular micro-services frameworks, i.e., Spring Cloud and Dubbo. In this section, we only present the experimental case of Dubbo, whose overall architecture is shown in Fig. 7. We adopted Dubbo to implement the three microservice modules, i.e., *COMS*, *CSMS* and *AMS*, which deployed three physical mac-hine clusters for each module. They completed the service registry on the Zookeeper [30] and used the Nginx [31] server to complete the service's reverse proxy. To ensure the high availability of the coordination service (denoted as *CS*), we adopted Eureka [32] to achieve service registry and service renewal and transferred the transaction entities to the cluster-ed Redis database. In particular, the network communication between the *CS* and multi-microservice modules is based on Netty's persistent connection.

Functional experiment

In this subsection, we design a specific test case with high randomness and a wide range to prove the functional reliab-ility of our middleware solution with distributed transactions in the multi-microservice modules, which can be described as follows.

- In *COMS*, the unit-price of item is randomly generated between $100 and $10,000, and the order number is set to be randomly generated between 1 and 1000, and the quantity purchased by the user is randomly generated between 1 and 100. We initialize the account balance in *AMS* to be $0, and the amount of stocks in *CSMS* to be 0.
- In *COMS*, *CSMS*, and *AMS*, we embed a *runtime exception* to abort the transaction process in the business code for order numbers 100, 200 and 500, respectively.
- Then, we continuously invoked the *createOrder* API interface, making a total of approximately 50,000 calls.

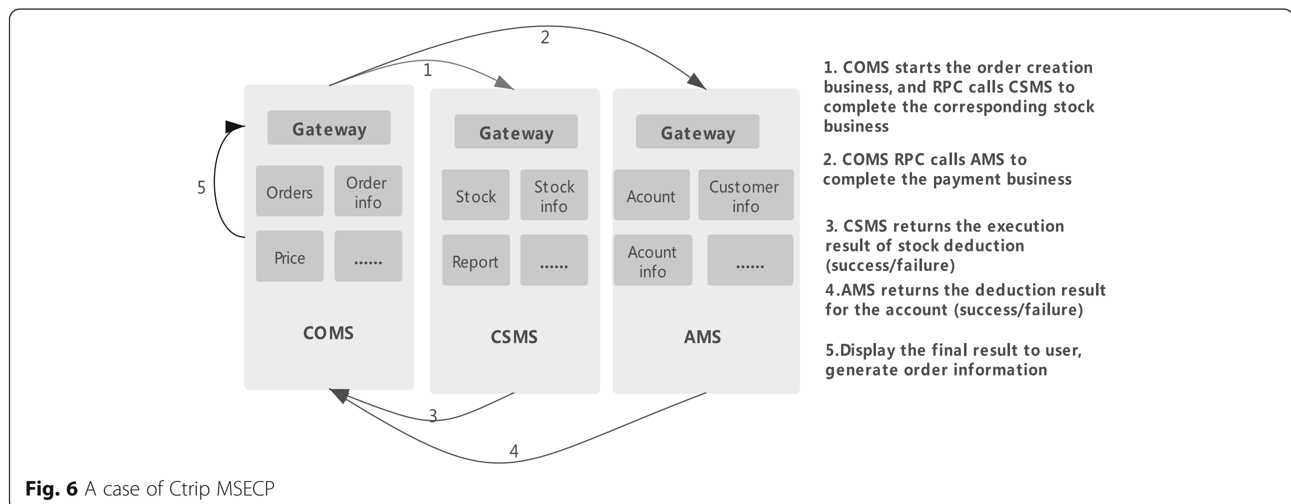

Fig. 6 A case of Ctrip MSECP

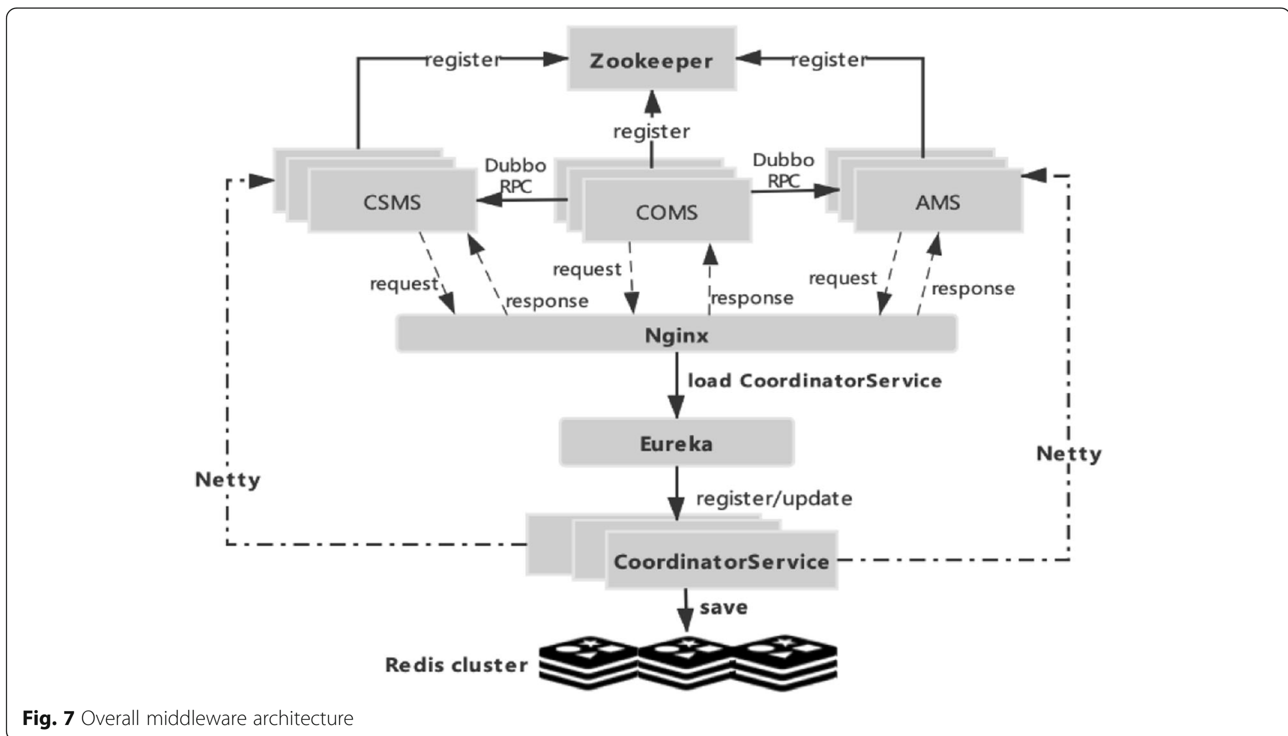

Fig. 7 Overall middleware architecture

- Finally, we determined that the data-consistent distributed transaction in this case should meet the following two conditions: the sum of the order amount and the account balance should be 0, and the sum of the order quantity and stock quantity should be 0.

According to the five groups of experimental results shown in Table 7 above, the *createOrder* interface was called approximately 5000 times, and its orders were successfully created approximately 2,450,000 to 3,180,000 times. More importantly, all experimental data met the above two condi-tions, i.e., Order amount + Account balance = 0; Order number + Stock number = 0. Therefore, our scheme is able to achieve consistent distributed transactions for multiple applications involving microservices.

Performance experiment

In this section, the evaluation of our scheme explores three key questions:

1) How does the throughput and latency of the optimized 2PC* compare with the traditional 2PC approach at varying levels of contention across microservices?
2) Can 2PC* guarantee its commit rate under the scenario of high-level contention?
3) Can our optimization scheme compensate for failed transaction steps?

Throughput

In this experimental case, we evaluated the throughput performance of our scheme through the indicators of TPS (transactions per second). We compare the TPS of the optimized protocol 2PC* and 2PC and adopt the number of local transactions of the database (i.e., Mysql) as a reference. We ran 10, 20, 50, 100, 200, 300, and 500 concurrent threads to call COMS's *CreateOrder* interface, and each thread exe-cuted 10 comparison experiments. Finally, we calculated and recorded the TPS averages.

Table 7 Data consistency experiment results

No.	Call times	Order amount	Account balance	Order number	Stock number
1	50,142	24,842,305,224	−24,842,305,224	2,850,098	−2,850,098
2	51,203	27,514,201,548	−27,514,201,548	3,183,214	−3,183,214
3	50,893	22,870,237,842	−22,870,237,842	2,581,249	−2,581,249
4	50,071	24,847,024,109	−24,847,024,109	2,708,291	−2,708,291
5	50,019	23,787,312,291	−23,787,312,291	2,457,219	−2,457,219

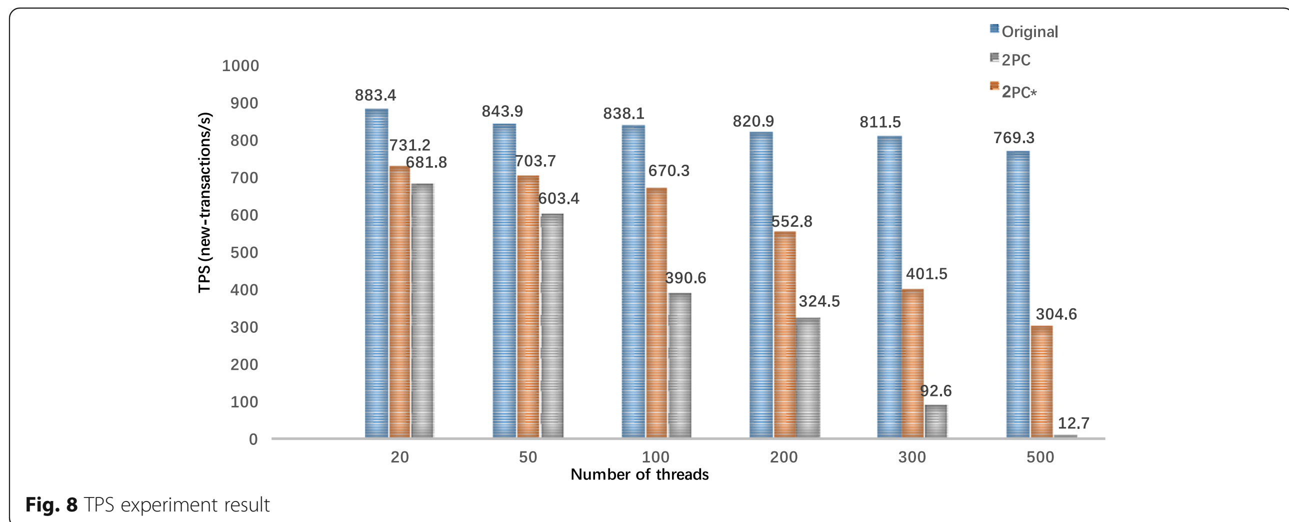

Fig. 8 TPS experiment result

Through the experimental data shown below (Fig. 8), when the number of concurrent threads is between 20 and 50, the distributed transactions are under low-level contention. Compared with 2PC, the 2PC* protocol has no significant advantage in terms of the TPS metric. However, when the number of concurrent requests increases from 50 to 100, the contention of the distributed transactions increases from low to moderate, and the performance gap between 2PC* and 2PC gradually widens. At this point, when the number of concurrent threads is between 100 and 200, the TPS of 2PC* can still be maintained at a relatively optimistic level, i.e., 670.3 and 552.8 new transactions/s, respectively, while 2PC's TPS is reduced to 391.6 and 324.5, respectively. Compared to transactions from the local database, 2PC*'s TPS drops by ~ 32.7% when the number of concurrent requests is 200, while 2PC's drops by ~ 60.5%; the throughput of 2PC* improved by ~ 70.4% compared to the original approach under moderate contention.

In the scenario with high-level contention, i.e., when the number of concurrent requests is from 300 to 500, 2PC* shows significant advantages. Under the scenarios with 300 concurrent requests, 2PC*'s TPS reduces to 401.5 new trans-actions/s, which is still half the performance of the local transactions, while 2PC's TPS reduces to only 92.6. In parti-cular, when the number of concurrent requests peaked at 500, the throughput of 2PC reached a performance bottleneck, while our scheme could still scale out. As the experimental data show, the transaction throughput performance of 2PC* is still quite high compared to that of 2PC; the TPS values are 304.6 new transactions/s and 12.7 new transactions/s, respectively. We abandoned the low-performance synchronous blocking lock in the control of transaction resources and repl-aced it with a novel second-level asynchronous lock, i.e., the SAOL, which can greatly reduce the blocking caused by the surging number of transactions; this is the key factor in the obvious advantage of the TPS performance of 2PC*.

In summary, our scheme has a significant improvement of throughput compared to the traditional approach, especially in scenarios of high-level contention. In other words, when the number of concurrent requests from users reaches a peak, the throughput of 2PC* can still maintain excellent perfor-mance, and it can be applied to highly concurrent requests for microservices with distributed transactions.

Latency

Transaction latency is another key indicator in our evaluation. We adopt the service's RT (response time) parameter to evaluate the latency performance of 2PC*. Similar to the experimental case in the "*Throughput*" sub-section, we ran 10, 20, 50, 100, 200, 300, and 500 concurrent threads to call *COMS*'s *CreateOrder* interface, performed 10 comparison experiments for each thread and calculated their RT averages.

Through the analysis of the experimental results (Fig. 9), when the transactions are under low-level contention, i.e., the number of concurrent requests is between 20 and 50, 2PC*'s RT has no obvious advantage compared with that of 2PC; for example, when the number of requests reaches 50, our scheme only reaches 18.3% improvement. The RT of 2PC is more sensitive to the increase in contention. When the number of concurrent requests grew to 200, the transactions reached moderate-level contention, and the latency of 2PC* achieved a significant performance advantage. In detail, when the number of concurrent requests reaches 100, the RT of 2PC* is only half that of 2PC. Compared to the original businesses' RT value, our scheme drops by ~ 39.6%. The latency superiority of 2PC* under high

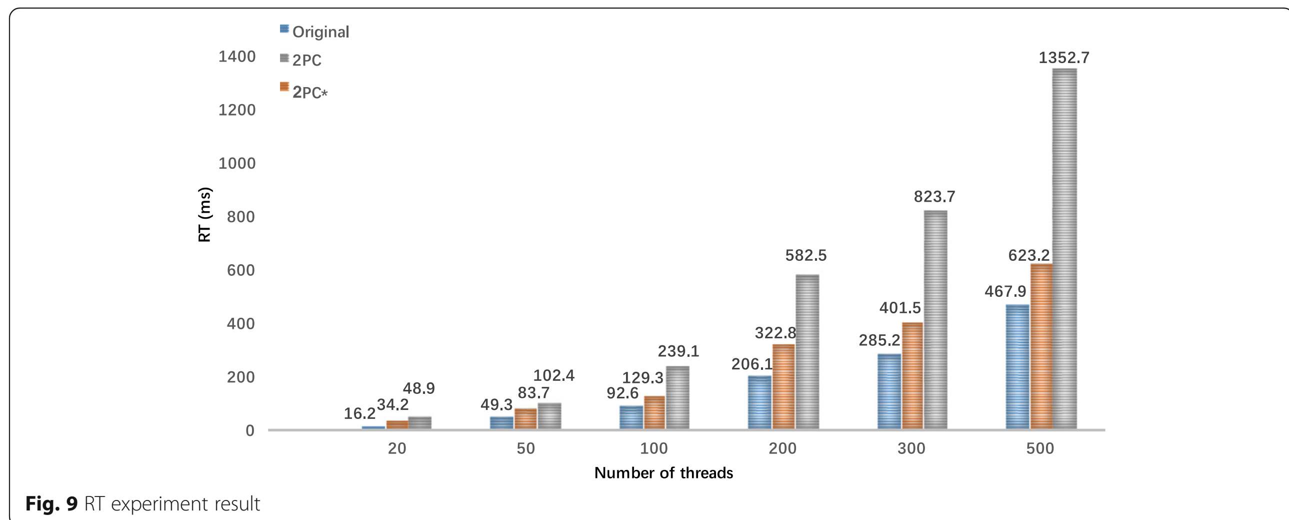

Fig. 9 RT experiment result

contention becomes more obvious. When the number of concurrent requests reaches 300, the RT of 2PC is 823.7 ms, which is 2.89 times that for the original business, while our scheme only increases to 401.5 ms. In particular, when the contention reaches its peak, 2PC is no longer suitable for the distributed transaction across microservices because its RT surpasses one second; it is 1352.7 ms. 2PC* is less sensitive to the increase in contention; the RT drops by ~ 33.4% with the original business and the latency is reduced by more than half compared with that of 2PC.

Similar to TPS, the factors affecting RT performance are closely related to the blocking rates between transactions. In the runtime protocol, 2PC* adopts an optimization algorithm based on a directed graph to aggregate and reorder the depen-dencies between transactions, which is able to reduce the conflict probability and avoid deadlocks and aborts between transactions. Additionally, the specific implementation utiliz-es the persistent connection network communication mode of the Netty framework, and we choose asynchronous threads in the coding; these are the key factors that show the obvious advantages of our proposed scheme in the experimental results of latency.

In summary, 2PC* demonstrates a better latency performa-nce with high-level contention than the traditional approach. Moreover, under the high-concurrency business scenario of microservices, our scheme can create less overhead due to latency.

Committing rate

In the three cases of low, moderate, and high contention, we evaluate the transactions' commit rate under our scheme. As shown in Fig. 10, 2PC* guarantees that the transactions are committed successfully even when the number of concurrent requests reaches a peak, while 2PC cannot be extended. When the transactions reached high con-tention, 2PC's commit rate dropped to almost zero—from 0.31 to 0.03.

Under the scenario with 500 concurrent requests, we count the instances of each of the three states of the transaction, i.e., RUNNABLE, WAITING and BLOCKED. We then calculate their blocking rates. According to Fig. 11, the blocking rates of 2PC* and 2PC both peaked at 450 ms and were 31% and 97%, respectively; thus, 2PC* is more than 3 times better than 2PC in terms of the committing rate. At this point, almost all threads in 2PC were blocked, which is the key reason for its commit rate almost reaching zero. 2PC* is also affected by the increase to high-level contention, although it is less sensitive than 2PC because it avoids aborting and retrying transactions. Compared with the traditional approach, 2PC* improves the performance of transaction committing by 3 to 4.5 times.

Transaction compensation

We continuously request the *CreateOrder* interface in the weak network environment with 100 concurrent threads, for which the running time is 30 s. As shown in the experimental data (Fig. 12), the transactions under 2PC* and 2PC occurred with 23,853.6 and 21,976.1 exceptions, respectively. We then recover the net-work environment and perform the transaction compensation process.

Our scheme can continuously compensate for exceptional transactions, which takes less than 3 s in total. 2PC* can be maintained at approximately 4831.4 groups per second, while 2PC has little ability to compensate. Based on *BASE* theory, 2PC* uses the *CMQ* asynchronously to comp-ensate for these failed transaction steps. In other words, it improves the fault tolerance of the system, which is also essential in large distributed systems, such as microservices.

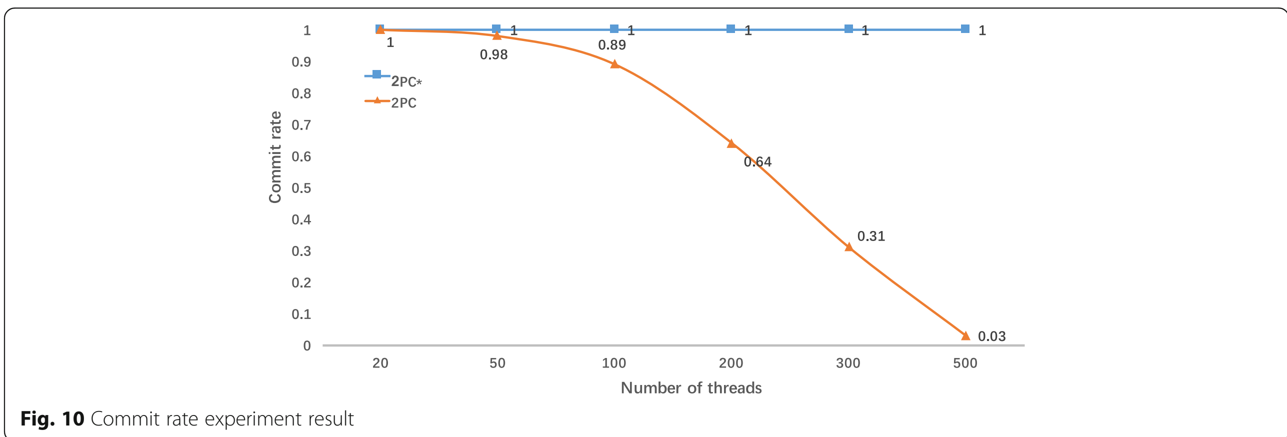

Fig. 10 Commit rate experiment result

Deployment and operations in cloud platform

Finally, we deployed our middleware solution in Ctrip's intelligent PaaS (Platform as a Service) cloud platform [33], called CPaaS Fig. 13. Its core includes three basic modules, namely, the big data platform, microservice ap-plication platform and application integration platform. The microservice business is deployed in the microser-vice application platform. Mean-while, application per-formance monitoring is responsible for managing the monitoring of the service interfaces. More importantly, our transactional middleware is deployed in the applica-tion integration platform, which is responsible for the distributed transaction control of microservices deployed in the CPaaS platform.

Through the monitoring system of the CPAAS cloud plat-form, we obtained the performance data of our scheme over 3 months. The results are shown in Table 8 below. TPS can be maintained at ~ 700 new transactions/s with an RT of no more than 96 ms. Most importantly, the transaction commit rate is always 100%, and transaction compensation can be successfully completed. Therefore, the long-term monitoring results prove that our scheme is stable and has universal app-licability in cloud computing.

Related work

In academia and industry, much of the recent work is still focused on transaction concurrency control using 2PC comb-ined with OCC in distributed databases, such as H-Store [32], VoltDB [34] by Michael and Samuel et al. In both H-Store and VoltDB, where data is assigned to different partitions according to specific rules, one of the great contributions of H-Store is that most transactions can be performed in a single partition, thus greatly reducing the additional overhead of concur-rency control. For example, H-Store can avoid the over-head of concurrent protocols with single-thread model for absolute single-partition transaction. It can supple-ment a few cross-partition transactions with the light-weight concur-rency protocol [35] to ensure possibility of serializability. However, as we have mentioned in this paper, these dist-ributed databases can only be applied to the single database and cannot be extended across

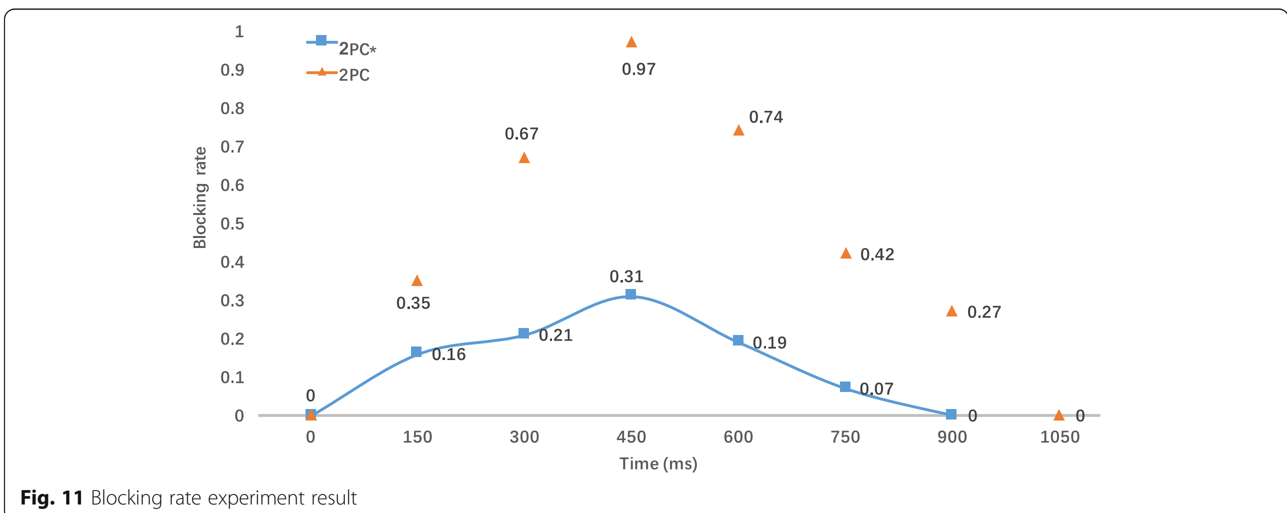

Fig. 11 Blocking rate experiment result

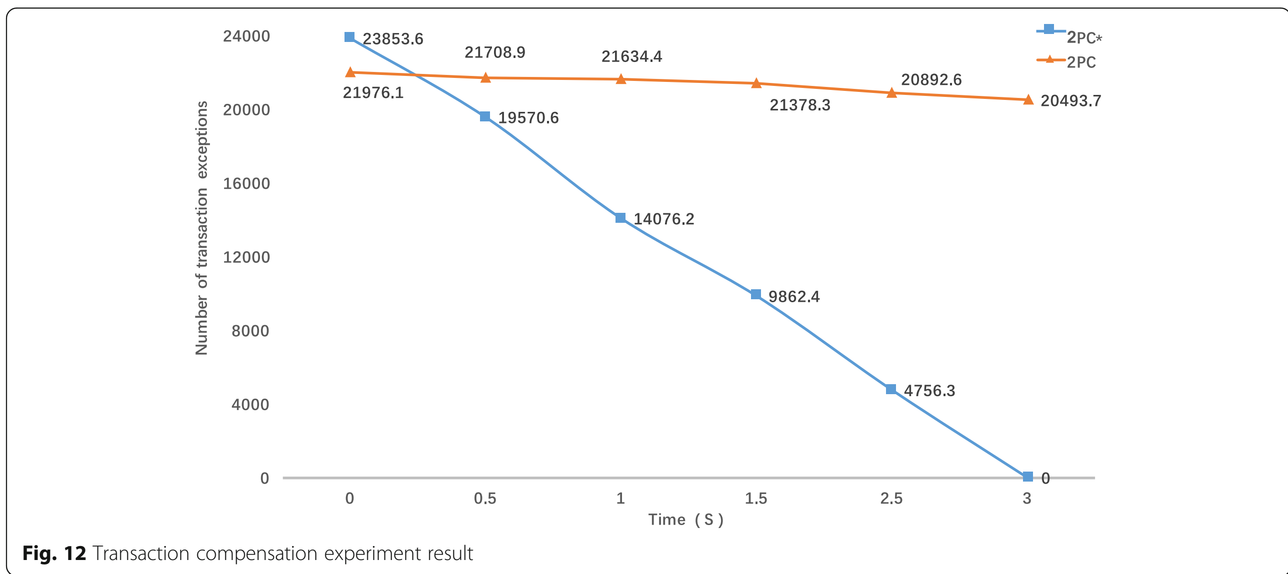

Fig. 12 Transaction compensation experiment result

multiple microservices, while 2PC* improved this function.

The multi-data-center consistency (MDCC) proposed by Tim Kraska et al. [36] used a commit method based on optimistic control, which was frequently used for storage across data centers. Therefore, MDCC does not require a global master node or a static data partitioning approach, and provides additional overhead similar to the design of eventual consistency. MDCC is based on Generalized Paxos [37] design, combined with *Commutative Operations* support. Therefore, MDCC performs better than any synchronous commit method. The reason is that it requires only single message to commit most transactional requests between multiple data centers.

Google's proposed Percolator [2] adopts OCC to support Snapshot Isolation [38]. Percolator compensates for the lack of batch processing of document updates in systems such as MapReduce [39]. It supports incremental document processi-ng and has been deployed by Google in its internal web sear-ch system. To improve throughput, Percolator allows multip-le clients to fetch documents simultaneously, and to provide isolation between different clients, it uses 2PC combined with MVCC for transaction support. Percolator has improved the timeliness of Google web search results by 50% since Google deployed it. The SAOL locking mechanism in 2PC* is also borrowed from the Percolator's design. SAOL uses the snapshot to breaking down locks in transactions into multiple levels of optimistic control, thereby reducing the blocking overhead of synchronous locks in a transaction.

In the Sinfonia [40] system built by Marcos K Aguilera et al., the concept of Min Transaction was innovatively

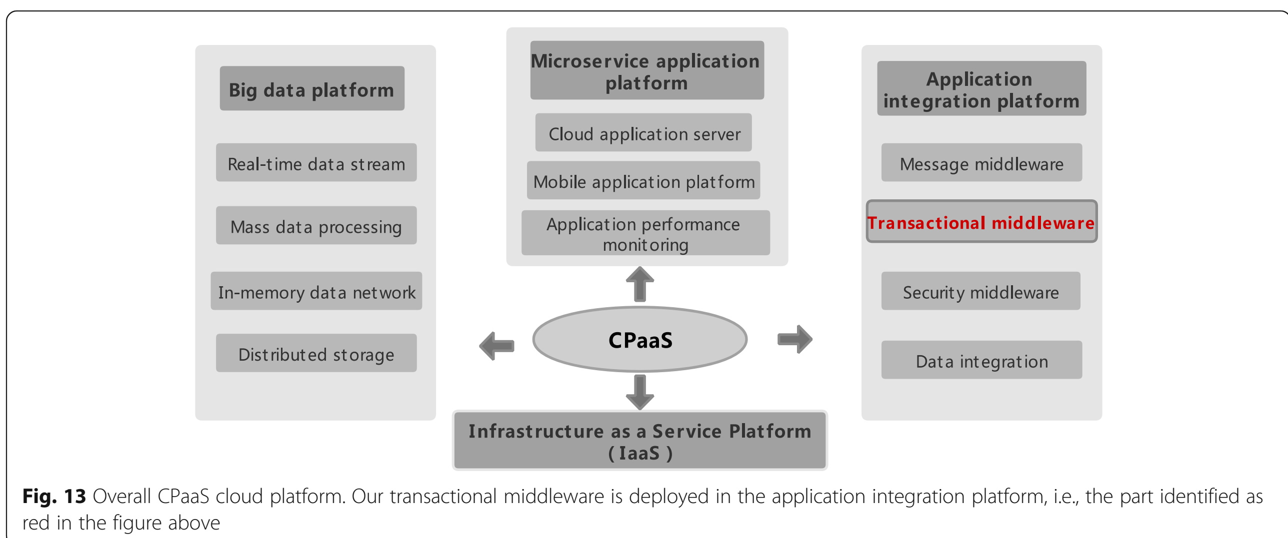

Fig. 13 Overall CPaaS cloud platform. Our transactional middleware is deployed in the application integration platform, i.e., the part identified as red in the figure above

Table 8 Data consistency experiment data

No.	Date	TPS (new-transactions/s)	RT (ms)	Commit Rate	Compensation
1	August	680.5	95.8	100%	984
2	September	701.3	89.3	100%	879
3	October	692.7	98.5	100%	1034

propos-ed, and its transaction implementation was also based on the 2PC protocol. Min Transaction is performed by locking the access object in the first phase of the Transaction and then committing it in the second phase. Sinfonia perfected the 2PC mechanism and came up with the concept of 1PC so that a single message exchange could commit the entire transaction for the coordinator. Unfortunately, the 1PC protocol of the Sinfonia is not able to be scaled for multiple microservices, as microservices often exist in the form of cross-services and cross-resources, which is the reason that 2PC* is still based on the two-phase commit protocol.

Andrei Furda et al. migrated microservices into the cloud computing environment and addressed several key issues in the process, one of which was data consistency [41]. That is, the challenges encountered in migrating legacy code runs can be summarized as operating decentralized data repositories from a centralized data repository to the microservices. Guy Pardon et al. studied the BAC theorem (backup, availability, and/or consistency) [42], which is an effective solution for the consistent disaster recovery for microservices, and is inspired by the CAP theorem. We also improved the fault tolerance of the 2PC* protocol, which is similar to the even-tual consistency they proposed, except that we borrowed from the BASE theory.

Zhang et al. proposed GRIT's distributed transaction model across microservices [3], which utilizes deterministic database technology and OCC to process data consistently. During the execution phase, transactions are optimally exec-uted by capturing their read and write operations. Then, at commit time, this method performs a conflict check and makes a global commit decision. Logically committed trans-actions are first transferred to the log and then executed asyn-chronously to carry out the database business. GRIT works at the procedural language level of the different data-bases, and 2PC* also adopts OCC's optimistic control, but we focus on transaction concurrency optimization and data consistency constraints.

Conclusion

This paper proposed 2PC*, a novel concurrency control protocol for distributed transactions in multi-microservice modules. For this purpose, we designed a novel secondary asynchronous optimistic lock, which can avoid the locks that are held in the transaction process. 2PC* utilizes a novel transaction concurrency control protocol, which is able to reduce the probability of concurrent conflicts among multiple transactions. Compared to the original 2PC, 2PC* can extract greater concurrency across multiple microservices. Finally, we implemented a middleware prototype based on 2PC* and applied it to a case of Ctrip MSECP deployed in the CPaaS cloud platform. The experimental results demonstrate that in high-level contention scenarios, our scheme has a higher throughput and lower latency than 2PC. Additionally, through long-term application performance monitoring by the CPaaS cloud platform, our scheme effectively supports distributed transaction concurrency control in a multi-microservice system.

In addition, we intend to continue some of our research in future work. We will adapt our scheme to some microservice frameworks in addition to Spring Cloud and Dubbo, which were discussed in this paper. As cloud computing becomes more popular, we can deploy it in DevOps [43, 44] in the PaaS [33, 45] cloud platform. In the IoT (Internet of Things) [27, 29], where cloud computing takes place, our scheme can also be extended. We also need to improve the QoS (quality of service) [29, 46] of microservices under various scenarios for mobile social networks [47] and hybrid networks [48] in the future.

Abbreviations

2PC: Two-phase commit; OCC: Optimistic concurrency control; MVCC: Multi-version concurrency control; TCM: Transaction coordination manager; RPC: Remote procedure call

Acknowledgements

Our deepest gratitude goes to the anonymous reviewers for their valuable suggestions to improve this paper.

Authors' contributions

Pan Fan designed and developed the proposed protocol as well as implemented the middleware solution based on cloud computing platform. Jing Liu directed working research and paper writing, and gave the experimental scheme. Wei Yin and Hui Wang provided experimental environments and designed use cases. Xiaohong Chen and Haiying Sun gave guidance on the grammar and abstract of this paper. The authors read and approved the final manuscript.

Authors' information

Pan Fan is studying for a master's degree in engineering at East China Normal University, China. His research interests include distributed computing, cloud computing, and formal method.
Jing Liu is currently a professor of computer science with East China Normal University, China. In recent years, she has been involved in the area of

model-driven architecture. She currently focuses on the design of real-time embedded systems and cyber-physical systems.
Wei Yin graduated from Northwestern Polytechinical University, China, he is currently a Senior Engineer of SAVIC Ltd. In recent years, he is involved in the area of Avioncs design, Software Engineering, System Engineering etc. His current work focuses on the design of the software and airworthness of Avionics.
Hui Wang received a master's degree in Nanjing University of Aeronautics and Astronautics, China. He is currently a researcher of SAVIC Ltd. His research interests include software verification and formal method.
Xiaohong Chen received a Ph.D. degree in Chinese Academy of Sciences, China, she is currently working as a master's tutor at East China Normal University, China. Her research interests include formal method, cyber-physical systems and artificial intelligence.
Haiying Sun received a Ph.D. degree in East China Normal University, China, her research interests include formal method, system simulation and model-driven engineering.

Funding
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.

Availability of data and materials
The full experimental data and the description of the experiment setup are provided in this manuscript in section *Evaluation* and our github, https://github.com/Leofan93/2pc-star.

Competing interests
The authors declare that they have no competing interests.

Author details
[1]East China Normal University, Shanghai 200062, China. [2]Shanghai Avionics co. Ltd, Shanghai, China.

Received: 30 January 2020 Accepted: 23 June 2020
Published online: 23 July 2020

References

1. Larrucea X, Santamaria I, Colomo-Palacios R, Ebert C (2018) Microservices. IEEE Softw 35(3):96–100. https://doi.org/10.1109/MS.2018.2141030
2. Corbett JC, Dean J, Epstein M, Fikes A, Frost C, Furman J, Ghemawat S, Gubarev A, Heiser C, Hochs-child P et al (2013) Spanner: googles globally distributed data-base. ACM Transact Comput Syst (TOCS) 31(3):8. https://doi.org/10.1145/2491245
3. Zhang G, Ren K, Ahn JS et al (2019) GRIT: consistent distributed transactions across polyglot microservices with multiple databases[C]. In: 2019 IEEE 35th international conference on data engineering (ICDE). IEEE. https://doi.org/10.1109/ICDE.2019.00230
4. Mohan C, Lindsay B, Obermarck R (1986) Transaction management in the R* distributed database management system. ACM Trans Database Syst (TODS) 11(4):378–396
5. Thomson A, Diamond T, Weng SC et al (2012) Calvin: fast distributed transactions for partitioned database systems[C]. In: Acm Sigmod international conference on management of data. ACM. https://doi.org/10.1145/2213836.2213838
6. Hwang E, Kim S, Yoo TK et al (2015) Resource allocation policies for loosely coupled applications in heterogeneous computing systems[J]. IEEE Trans Parallel Distributed Syst:1–1. https://doi.org/10.1109/TPDS.2015.2461154
7. Du J, Sciascia D, Elnikety S, Zwaenepoel W, Pedone F (2014) Clock-RSM: low-latency inter-datacenter state machine replication using loosely synchronized physical clocks. 2014 44th annual IEEE/IFIP international conference on dependable systems and networks, Atlanta, pp 343–354. https://doi.org/10.1109/DSN.2014.42
8. YugaByte. https://www.yugabyte.com/. Accessed 10 May 2019.
9. FoundationDB. https://www.cockroachlabs.com/. Accessed date 10 May 2019.
10. Zhang S, Zhu S (2013) Server structure based on netty framework for internet-based laboratory [C]. In: Control and automation (ICCA), 2013 10th IEEE international conference on. IEEE. https://doi.org/10.1109/ICCA.2013.6564990
11. Bershad BN, Anderson TE, Lazowska ED, Levy HM Lightweight remote procedure call. ACM Trans Comput Syst 8(1):37–55. https://doi.org/10.1145/74850.74861
12. He S, Zhao L, Pan M (2018) The Design of Inland River Ship Microservice Information System Based on spring cloud [C]. In: 5th international conference on information science and control Engin-eering (ICISCE). https://doi.org/10.1109/ICISCE.2018.00120
13. Dubbo. http://dubbo.apache.org/. Accessed 9 Mar 2019.
14. Herlihy MP, Wing JM (1990) Linearizability: a correctness condition for concurrent objects. ACM Transact Program Lang Syst (TOPLAS) 12(3):463–492
15. Herlihy M Apologizing versus asking permission: optimistic concurrency control for abstract data types [J]. ACM Trans Database Syst 15(1):96–124. https://doi.org/10.1145/77643.77647
16. Nishi T, Yushin et al (2017) An efficient deadlock prevention policy for noncyclic scheduling of multicluster tools [J]. In: IEEE transactions on automation science and engineering. https://doi.org/10.1109/TASE.2017.2771751
17. Breitbart Y, Garcia-Molina H, Silberschatz A (1992) Overview of multidatabase transaction management. VLDB J 1:181–239. https://doi.org/10.1007/BF01231700
18. Ardekani MS, Sutra P, Shapiro M (2013) Non-monotonic snapshot isolation: scalable and strong consistency for geo-replicated transactional systems[C]. https://doi.org/10.1109/SRDS.2013.25
19. Attiya H, Ellen F, Morrison A Limitations of highly-available eventually-consistent data stores [J]. In: IEEE transactions on parallel and distributed systems, p 1. https://doi.org/10.1109/TPDS.2016.2556669
20. Luo C, Okamura H, Dohi T (2013) Modeling and analysis of multi-version concurrent control [C]. In: 2013 IEEE 37th annual computer software and applications conference. IEEE. https://doi.org/10.1109/COMPSAC.2013.11
21. snowflake. https://github.com/twitterarchive/snowflake. Accessed 11 June 2019.
22. Tarjan (2008) Depth-first search and linear graph algorithms [C]. In: Symposium on Switching & Automata Theory. IEEE. https://doi.org/10.1109/SWAT.1971.10
23. Chaudhuri K, Doligez D, Lamport L et al (2010) The TLA+ proof system: building a heterogeneous verification platform [M]// theoretical aspects of computing – ICTAC 2010. Springer, Berlin Heidelberg. https://doi.org/10.1007/978-3-642-14808-8_3
24. Github. https://github.com/Leofan93/2pc-star. Accessed 11 June 2019.
25. Taibi, Toufik Formal specification and validation of multi-agent behaviour using tla+ and tlc model checker. Int J Artificial Intel Soft Comput 1(1):99. https://doi.org/10.1504/ijaisc.2008.021266
26. Huang X, Zhang Y, Xing C et al (2012) Paxos-based memory data replication in stock trading system [C]. In: IEEE computer software & applications conference. IEEE Computer Society. https://doi.org/10.1109/COMPSAC.2012.46
27. CompletableFuture. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html. Accessed 11 June 2019.
28. PlatformTransactionManager. https://docs.spring.io/spring/docs/5.1.3.RELEASE/spring-framework-reference/data-access.html#transaction-strategies. Accessed 11 June 2019.
29. ReentrantLock. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html. Accessed 11 June 2019.
30. Artho C, Gros Q, Rousset G et al (2017) Model-based API testing of apache ZooKeeper [C]. In: 2017 IEEE international conference on software testing, verification and validation (ICST). IEEE. https://doi.org/10.1109/ICST.2017.33
31. Ngnix. https://www.nginx.com/resources/wiki/. Accessed 15 June 2019
32. Eureka. https://spring.io/projects/spring-cloud-netflix. Accessed 15 June 2019.
33. Yin Y, Lu C, Xu Y, Wan J, Zhang H, Mai Z (2019) QoS prediction for service recommendation with deep feature learning in edge computing environment. In: Mobile networks and applications. https://doi.org/10.1007/s11036-019-01241-7
34. Bernstein D (2014) Today's tidbit. VoltDB [J] 1(1):90–92. https://doi.org/10.1109/MCC.2014.25
35. Pedreira P, Lu Y, Pershin S (2018) Rethinking concurrency control for in-memory OLAP DBMSs. In: IEEE 34th international conference on data engineering (ICDE). https://doi.org/10.1109/ICDE.2018.00164
36. Kraska T, Pang G, Franklin M, Madden S (2012) MDCC: multi-data center consistency. In: Proceedings of the 8th ACM European conference on computer systems, EuroSys 2013. https://doi.org/10.1145/2465351.2465363
37. Pires M, Ravi S, Rodrigues R (2017) Generalized Paxos made byzantine (and less complex) [C]. In: International symposium on stabilization, safety, and security of distributed systems. https://doi.org/10.1007/978-3-319-69084-1_14

38. Wei H, Huang Y, Lu J (2017) Parameterized and runtime-tunable snapshot isolation in distributed transactional key-value stores [C]. In: 2017 IEEE 36th symposium on reliable distributed systems (SRDS). IEEE. https://doi.org/10.1109/SRDS.2017.11
39. Xu X, Tang M (2017) A new approach to the cloud-based heterogeneous MapReduce placement problem [J]. IEEE Trans Serv Comput 9(6):862–871. https://doi.org/10.1109/TSC.2015.2433914
40. Aguilera MK, Merchant A, Shah MA, Veitch AC, Karamanolis CT (2009) Sinfonia: a new paradigm for building scalable distributed systems. ACM Trans Comput Syst 27(3). https://doi.org/10.1145/1294261.1294278
41. Furda CF, Zimmermann O, Kelly W, Barros A (2018) Migrating Enterprise legacy source code to microservices: on multitenancy, Statefulness, and data consistency. IEEE Softw 35(3):63–72. https://doi.org/10.1109/MS.2017.440134612
42. Pardon G, Pautasso C, Zimmermann O (2018) Consistent disaster recovery for microservices: the BAC theorem. IEEE Cloud Comput 5(1):49–59. https://doi.org/10.1109/MCC.2018.011791714
43. Gao H, Duan Y, Shao L, Sun X (2019) Transformation-based processing of typed resources for multimedia sources in the IoT environment. Wirel Netw. https://doi.org/10.1007/s11276-019-02200-6
44. Gao H, Xu Y, Yin Y, Zhang W, Li R, Wang X (2019) Context-aware QoS prediction with neural collaborative filtering for internet-of-things services. IEEE Internet Things J. https://doi.org/10.1109/JIOT.2019.2956827
45. Ebert C, Gallardo G, Hernantes J et al (2016) DevOps [J]. IEEE Softw 33(3):94–100. https://doi.org/10.1109/MS.2016.68
46. Bernstein D (2014) Cloud foundry aims to become the OpenStack of PaaS. Cloud Comput IEEE 1(2):57–60. https://doi.org/10.1109/MCC.2014.32
47. Yin Y, Xia J, Yu L, Xu Y, Xu W, Yu L (2019) Group-wise itinerary planning in temporary Mobile social network. IEEE Access 7:83682–83693. https://doi.org/10.1109/ACCESS.2019.2923459
48. Gao H, Liu C, Li Y, Yang X (2020) V2VR: reliable hybrid-network-oriented V2V data transmission and routing considering RSUs and connectivity probability. In: IEEE transactions on intelligent transportation systems(T-ITS). https://doi.org/10.1109/TITS.2020.2983835

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Submit your manuscript to a SpringerOpen journal and benefit from:

► Convenient online submission
► Rigorous peer review
► Open access: articles freely available online
► High visibility within the field
► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com