Computers and Mathematics with Applications 63 (2012) 458–468

Contents lists available at SciVerse ScienceDirect

Computers and Mathematics with Applications

journal homepage: www.elsevier.com/locate/camwa

An efficient deadlock prevention approach for service oriented transaction processing

Feilong Tang[a,*], Ilsun You[b], Shui Yu[c], Cho-Li Wang[d], Minyi Guo[a], Wenlong Liu[e]

[a] *Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China*
[b] *School of Information Science, Korean Bible University, Nowon-gu, Seoul, South Korea*
[c] *School of Information Technology, Deakin University, Burwood, VIC 3125, Australia*
[d] *Department of Computer Science, The University of Hong Kong, Hong Kong*
[e] *School of Information and Communication Engineering, Dalian University of Technology, Dalian, China*

A R T I C L E   I N F O

Keywords:
Distributed transaction processing
Deadlock prevention
Service oriented architecture (SOA)
Replication
Two-phase commit (2PC)

A B S T R A C T

Transaction processing can guarantee the reliability of business applications. Locking resources is widely used in distributed transaction management (e.g., two phase commit, 2PC) to keep the system consistent. The locking mechanism, however, potentially results in various deadlocks. In service oriented architecture (SOA), the deadlock problem becomes even worse because multiple (sub)transactions try to lock shared resources in the unexpectable way due to the more randomicity of transaction requests, which has not been solved by existing research results. In this paper, we investigate how to prevent *local deadlocks*, caused by the resource competition among multiple sub-transactions of a global transaction, and *global deadlocks* from the competition among different global transactions. We propose a replication based approach to avoid the local deadlocks, and a timestamp based approach to significantly mitigate the global deadlocks. A general algorithm is designed for both local and global deadlock prevention. The experimental results demonstrate the effectiveness and efficiency of our deadlock prevention approach. Further, it is also proved that our approach provides higher system performance than traditional resource allocation schemes.

1. Introduction

Business applications have necessitated distributed transaction management technologies. Existing distributed transaction models widely use the resource locking mechanism for keeping the consistency of transaction systems, among which two-phase commit (2PC) [1] is the most representative coordination protocol through requiring sub-transactions to lock resources before the transaction commit. Unfortunately, 2PC-like protocols potentially induce various deadlocks when multiple (sub-)transactions try to lock the same resource at the same time.

Service oriented architecture (SOA) presents new requirements and challenges to the transaction management. With the success of SOA, many large-scale information systems have been set up to provide business services simultaneously [2,3]. In SOA environments, the deadlock problem due to the resource competition among multiple (sub)transactions gets worse because of the randomicity of transaction requests and the uncontrollability of transaction execution order [4,5]. The deadlock in SOA environments will occur more often than that in traditional distributed systems. As a result, new deadlock prevention [6] approaches are needed for improving the performance of service-oriented systems.

[*] Corresponding author.
E-mail address: tang-fl@cs.sjtu.edu.cn (F. Tang).

0898-1221/$ – see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.camwa.2011.07.063

Deadlock control technologies can be categorized as *deadlock avoidance* [7–9], *deadlock detection* [10,11] and *deadlock prevention* [12–14]. The *deadlock avoidance* strategy only accepts the requests that will lead to safe states. Although it allows more concurrency [15,16], it has to know the number and type of all resources before the actual resource allocation. For many systems, however, it is impossible to know all resources and their states in advance. The *deadlock detection* [17] tracks resource allocation and process states, and restarts one or more processes to remove deadlocks. To detect deadlocks introduced by concurrent resource accesses, some researchers proposed wait-for-graph-based detection algorithm [10]. Timeout based probabilistic analysis [18] is also used to detect global deadlocks in distributed systems, however, the timeout itself is different to be decided [19]. On the other hand, after a deadlock is detected, one of transactions in the wait cycle must be aborted. Wang et al. [20] proposed guaranteed deadlock recovery based on run-time dependency graph and incorporated it into distributed deadlock detection algorithm. Although the deadlock detection is effective, it costs more time. Moreover, with regard to local deadlocks defined in this paper, nothing can be done even if a deadlock is detected. Finally, the *deadlock prevention* mechanism often removes the "hold and wait" condition by requiring processes to request all the needed resources before starting up. Ezpeleta. et al. [12] proposed a Petri net based deadlock prevention policy based on the liveness or the reachability of Petri nets. An advantage of the deadlock prevention is that it does not need to know details of all resources available and requested. So, the deadlock prevention approach is more suitable for dynamical and open service-oriented environments.

In this paper, we propose an algorithm to prevent *local deadlocks*, caused by the resource competition among multiple sub-transactions of a transaction, and *global deadlocks* due to the resource competition among different global transactions. In our scheme, we control concurrent resource accesses through the resource manager, which is particularly useful for business transactions in service-oriented environments. We describe in brief our contributions in this paper as follows.

(1) We propose a replication based approach to avoid local deadlocks. In traditional deadlock prevention schemes, when two or more sub-transactions of a global transaction compete for the same resource, the global transaction will have to be aborted. On the other hand, whenever the global transaction restarts, it will inevitably fail again due to the same resource competition.
(2) We propose a timestamp based approach to prevent global deadlocks. In our scheme, the conflicted transactions that compete for the same resource are selectively aborted after a timeout, based on their transaction ID. Consequently, our approach avoids the live-locks due to resource competition among global transactions.
(3) A general algorithm is designed for preventing both local and global deadlocks, based on the solutions proposed above.
(4) We design an intelligent resource manager by merging the deadlock prevention function with the resource management function. The resource manger can detect and prevent local and global deadlocks and allocate appropriate lock(s) for each transaction.

The experimental results demonstrate that our replication based mechanism completely avoids the local deadlocks. On the other hand, our timestamp based mechanism significantly reduces the global deadlocks and live-locks.

The remainder of this paper is organized as follows. In Section 2, we briefly introduce the related background and formally define the local and global deadlocks. Section 3 presents our replication based approach for avoiding the local deadlocks, the timestamp based mechanism to prevent global deadlocks, and the general algorithm for local and global deadlock prevention. The implementation and performance evaluations are reported in Section 4. Finally, we conclude this paper in Section 5.

2. Preliminaries

The deadlock in the transaction processing is highly relevant to transaction commit protocols. Our deadlock prevention is designed for 2PC-like protocols. In this section, we briefly describe the 2PC protocol and its locking implementations, and then formally define local and global deadlocks.

2.1. Background

2PC was designed for coordinating distributed atomic transactions with the following properties: atomicity, consistency, isolation and durability. Usually, a distributed atomic transaction is managed by a transaction manager (TM) together with a set of resource managers (RM) responsible for allocating individual resources for corresponding sub-transactions. A TM controls multiple RMs involved in a global transaction. On the other hand, an RM also can be shared by multiple TMs for concurrent transactions.

A distributed transaction contains a set of sub-transactions executed in different networked nodes [21,22]. Each of them works as a participant under the control of the TM. The 2PC protocol guarantees that a transaction is either successfully committed or not performed at all. In 2PC-based transaction processing, the TM coordinates all the sub-transactions in the following two phases.

Phase 1. All participants (sub-transactions) receive instructions from a transaction coordinator (i.e., TM) to prepare for commit. In most cases, it is achieved through locking needed resources. If a resource manager can lock the needed resources for a corresponding sub-transaction, it votes OK; that means it is ready to commit. Otherwise, it responds Failed to the TM.

Phase 2. The TM sends a commit instruction to all RMs if all the participants voted OK. Otherwise it requires the RMs, which are ready to commit, to roll back. Then all the resource managers commit or roll back according to the instruction from the TM.

The above 2PC protocol ensures that every participant takes the same action to achieve the "all or nothing" property.

2.2. Problem statement

2PC protocol potentially causes deadlocks when multiple transactions (sub-transactions) compete for the same resource. In this section, we formally define *local deadlock* and *global deadlock* problems.

We assume that there are m global transactions $T = \{T_i|1 \leq i \leq m\}$ and r resources $\mathfrak{R} = \{R_k|1 \leq k \leq r\}$ in a system. Each resource R_k is under the control of its own resource manager M_k. Further, let a distributed transaction T_i consist of n sub-transactions such that $T_i = \{T_{i,j}|1 \leq j \leq n\}$. We use $A_{i,j}^k$ to indicate the relationship between a sub-transaction $T_{i,j}$ and a resource R_k. $A_{i,j}^k = 1$ if $T_{i,j}$ needs to access R_k during the transaction processing; otherwise, $A_{i,j}^k = 0$. Similarly, A_i^k indicates the relationship between a global transaction T_i and a resource R_k. $A_i^k = 1$ if the global transaction T_i accesses R_k during the execution of T_i, which means some sub-transactions of T_i will lock the resource R_k; otherwise, $A_i^k = 0$, which means no sub-transaction of T_i needs to access R_k. Moreover, we use $L_{R_k} = 1$ to indicate that the resource R_k was locked; otherwise, $L_{R_k} = 0$. In particular, $L_{R_k}^{T_i} = 1$ means that the R_k was locked by the global transaction T_i; otherwise, $L_{R_k}^{T_i} = 0$.

Accordingly, we design two matrices: *local access matrix* $A_L = (A_{i,j}^k)_{n\times r}$ $(1 \leq j \leq n, 1 \leq k \leq r)$ and *global access matrix* $A_G = (A_i^k)_{m\times r}$ $(1 \leq i \leq m, 1 \leq k \leq r)$. As a result, $\sum_{j=1}^n A_{i,j}^k$ represents how many sub-transactions $T_{i,j}$ of a global transaction T_i will simultaneously access the same resource R_k while $\sum_{j=1}^n A_i^k$ indicates how many global transactions simultaneously requests the R_k.

Definition 1. A local deadlock occurs if at least a resource R_k is requested by two or more sub-transactions $T_{i,j}$ of a global transaction T_i, i.e., $\exists\ k,\ \sum_{j=1}^n A_{i,j}^k > 1$, before T_i commits.

When multiple sub-transactions of a global transaction T_i try to lock the same resource R_k, only the first requestor can lock the R_k while others will be blocked in traditional 2PC-based transaction processing schemes. Consequently, T_i enters the deadlock status because 2PC protocol waits for votes from all sub-transactions before the commit. On the other hand, if T_i is aborted through a timeout mechanism and is restarted again, it will still fail to commit due to the same competition on the resource R_k.

When a global deadlock occurs, there is a *wait loop* among the conflicted global transactions. By the wait loop, we mean each conflicted global transaction T_i occupies some resources but still needs other resource(s), which have (has) been locked by other transaction(s) T_j $(1 \leq i, j \leq m; i \neq j)$. As a result, these transactions in the wait loop mutually wait resources locked by others. We use $d_{i,j}$ to represent the resource demand of a sub-transaction $T_{i,j}$. A global deadlock can be defined as follows.

Definition 2. A global deadlock occurs if (1) for any global transaction T_i in the wait loop, $\exists j$, $\sum_{k=1}^r A_{i,j}^k < d_{i,j}$, which means at least one of sub-transactions of T_i needs to lock other locked resource(s) and (2) $\sum_{k=1}^r A_i^k \geq 1$, which means any global transaction in the wait loop has locked at least one resource.

2.3. Deadlock scenarios

In this section, we exemplify a local deadlock and a global deadlock.

(1) *Local deadlock scenario* (*scenario* 1)

During the 2PC-based transaction coordination, resource managers must hold the requested resources in the first phase of 2PC. Otherwise, other concurrent transactions may access intermediate results and lead to system inconsistency. For this purpose, resource managers often lock the resources and do not release their locks until the transaction commit. On the other hand, during the first stage, the transaction coordinator waits for all the sub-transactions (participants) to vote for their states. As a result, if two or more sub-transactions request the same resource(s), a local deadlock is inevitable.

For example, $T_{1,1}$ and $T_{1,2}$ are sub-transactions of a transaction T_1, and they both need to access the same resource R, as shown in Fig. 1. Due to 2PC protocol, R will be held by one of the two sub-transactions according to the scheduling rule (e.g., first come first serve). We assume that $T_{1,1}$ gets the lock on R, and then it votes *OK* to the coordinator. The coordinator has to wait for $T_{1,2}$'s response. However, $T_{1,2}{}'$ request to the same resource R will be blocked until $T_{1,1}$ release the lock on R. If the timeout mechanism is not considered in the first phase, $T_{1,1}$ will not release its lock because the coordinator cannot make a final conclusion. In this case, a wait-for cycle is formed and the local deadlock happens. On the other hand, if a timeout is merged in the 2PC protocol, T_1 can be aborted after the timeout. Unfortunately, T_1 will still fail to commit after it restarts again due to the same competition on R.

In the above example, each sub-transaction requests only one resource and further all the sub-transactions need to access the same resource. We can easily extend it to the scenario where some sub-transaction(s) need to access multiple resources. Fig. 2 illustrates such an extended local deadlock, where $T_{1,3}$ has locked R_2 but it has to wait for R_1 locked by $T_{1,1}$. Further, $T_{1,2}$ is blocked because R_2 has been locked by $T_{1,3}$.

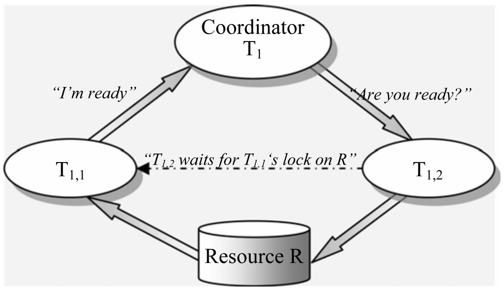

Fig. 1. Local deadlock.

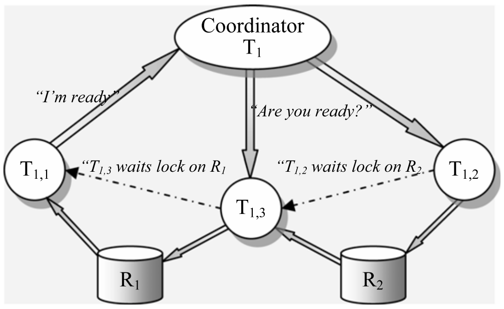

Fig. 2. Extended local deadlock.

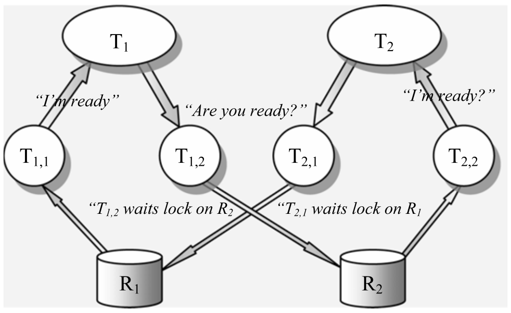

Fig. 3. Global deadlock.

(2) *Global deadlock scenario* (*scenario* 2)

A global deadlock will occur when multiple concurrent global transactions compete for the same resource(s). Assume that there are two global transactions T_1 and T_2 in a system. Each transaction consists of two sub-transactions such that $T_1 = \{T_{1,1}, T_{1,2}\}$ and $T_2 = \{T_{2,1}, T_{2,2}\}$. $T_{1,1}$ and $T_{2,1}$ need to access R_1 and $T_{1,2}$ and $T_{2,2}$ have to access R_2, respectively. As a result, a global deadlock is caused when concurrent T_1 and T_2 request the two resources R_1 and R_2 in the following order.

- Transaction managers of T_1 and T_2 require sub-transactions to prepare corresponding resources, respectively.
- $T_{1,1}$ requests the R_1 and successfully sets a lock on the R_1.
- $T_{2,2}$ requests the R_2 and successfully sets a lock on the R_2.
- $T_{1,2}$ begins to request R_2, but it has to wait for the lock on R_2.
- $T_{2,1}$ begins to request R_1, but it has to wait for the lock on R_1.

This resource request flow can be shown in Fig. 3. In this scenario, $T_{1,1}$ waits for T_1's final decision, T_1 waits for $T_{1,2}$'s vote, and $T_{1,2}$ waits for $T_{2,2}$ to release the R_2. Unfortunately, it is true of T_2 and its sub-transactions $T_{2,1}$ and $T_{2,2}$. So, T_1 and T_2 will mutually wait for the resource occupied by the other side.

3. Deadlock prevention approach

In this section, we present two prevention mechanisms for local deadlocks and global deadlocks in service-oriented environments, respectively, and then propose an algorithm to implement the two mechanisms.

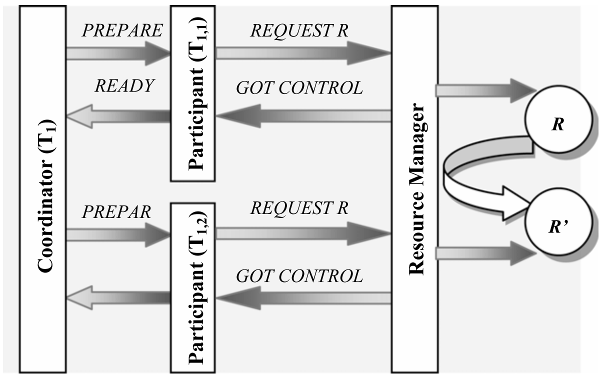

Fig. 4. Replication based mechanism for local deadlock.

3.1. Replication-based local deadlock prevention

As described above, a local deadlock occurs when some sub-transactions (called *conflicted sub-transactions*) of a global transaction try to lock the same resource, which can be locked by only one of the conflicted sub-transactions in existing 2PC-based transaction models. Consequently, others of the conflicted sub-transactions are blocked unless the global transaction is aborted. On the other hand, once the global transaction restarts, it will enter the deadlock state again due to the same resource competition. As a result, the local deadlock cannot be simply prevented by using traditional resource allocation schemes [23,24], in which a resource is locked by only one sub-transaction.

Based on the above observation and analysis, we propose a new replication-based mechanism to prevent local deadlocks, which works based on the following ideas.

(1) A resource manager replicates a copy of the competed resource when more than one sub-transactions request it through the resource manger.
(2) The replicated resource copy is shared (i.e., read and write) by all the conflicted sub-transactions.
(3) The resource copy will be updated to its original resource once the global transaction commits.

Distinguishing from traditional transaction models where each sub-transaction locks a resource separately, these conflicted sub-transactions in our scheme share a lock on the replicated resource before the global commits.

Every global transaction has a unique root ID. When a sub-transaction is distributed to a node, its manager keeps the root ID of its parent and generates its own sub-ID. We use $ID(T_i)$ and $ID(T_{i,j})$ to denote the IDs of a global transaction T_i and its sub-transaction $T_{i,j}$, respectively. In our scheme, the $ID(T_{i,j})$ consists of two parts: the ID $ID_{parent}^{T_{i,j}}$ of its parent transaction generated by the *Coordinator* and the sub-ID $ID_{sub}^{T_{i,j}}$ generated by the corresponding *Participant* (see Fig. 4), such that $ID(T_{i,j}) = ID_{parent}^{T_{i,j}} + ID_{sub}^{T_{i,j}} = ID(T_i) + ID_{sub}^{T_{i,j}}$. So, every resource manager knows the root ID of any sub-transaction, and can distinguish whether any two sub-transactions belong to the same global transaction or not, based on their root IDs (i.e., $ID(T_i)$).

Fig. 4 illustrates how our scheme prevents local deadlocks, where a global transaction T_1 includes two sub-transactions such that $T_1 = \{T_{1,1}, T_{1,2}\}$. Both $T_{1,1}$ and $T_{1,2}$ access the same resource R. Without losing generality, we assume that resource manager *RM* first receives $T_{1,1}$ request to the R and then $T_{1,2}$ request to the R. When RM receives the request from $T_{1,1}$, it locks the R for $T_{1,1}$ immediately. However, RM does not reject the request from $T_{1,2}$ although the R has been locked by $T_{1,1}$. Instead, it replicates a copy of the R (marked as R') and from then on, all requests from sub-transactions with the same root ID are shifted to the replicated resource R'. Note that RM immediately releases the lock on the original resource R after the replication. The general approach for preventing local deadlocks is described in Fig. 5, where we assume that the resource R_k is locked by the transaction T_l when $L_{R_k} = 1$. So, $T_{i,j}$ is one sub-transaction of T_l when $ID_{parent}^{T_{i,j}} = ID(T_l)$. Our approach not only prevents local deadlocks but also improves the system concurrency through allowing other global transactions to operate on the original resource R concurrently.

3.2. Timestamp-based global deadlock prevention

In this section, we investigate how to prevent global deadlocks, which happens more often than local deadlocks. Existing technologies for global deadlock prevention are generally based on sequential resource access. It is a pessimistic static resource allocation scheme that needs to exploit prior knowledge of transaction access patterns [25].

3.2.1. Pre-check based approach for preventing global deadlocks

In service-oriented environments, each business transaction knows what resources it will request. So, it is appropriate to make sure whether resources needed by a transaction are available or not before starting the transaction. We propose a

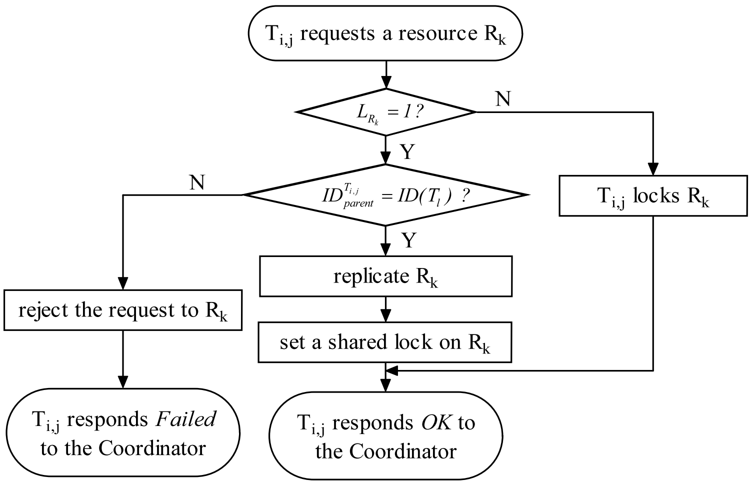
$T_{i,j}$ requests a resource R_k
$L_{R_k} = 1$?
N
Y
$ID^{T_{i,j}}_{parent} = ID(T_l)$?
N
Y
$T_{i,j}$ locks R_k
replicate R_k
reject the request to R_k
set a shared lock on R_k
$T_{i,j}$ responds *Failed* to the Coordinator
$T_{i,j}$ responds *OK* to the Coordinator

Fig. 5. Resource allocation and local deadlock prevention.

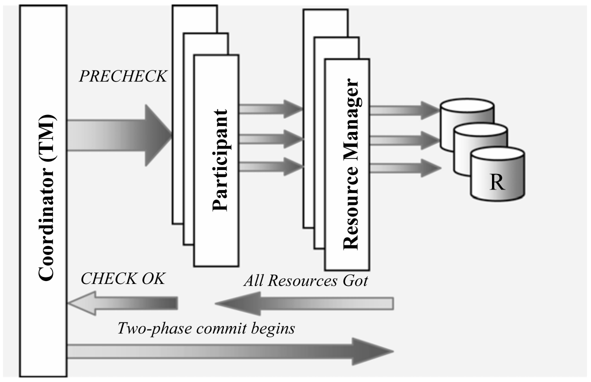

Fig. 6. Global deadlock resolution.

pre-check based approach to prevent potential global deadlocks. The basic idea is that each global transaction has to check and then hold all the necessary resources if they are available at the beginning of the transaction execution.

We extend the 2PC protocol through adding a new phase called *Pre-Check phase*. In the Pre-Check stage, the coordinator delivers all the sub-transactions to participants, and then these participants communicate with resource managers to check the state of resources. If these resources are available, the participant will hold them and at the same time return OK to the coordinator. Otherwise, it will return a failed message. After receiving OK messages from all participants, the coordinator will start the standard two-phase commit.

Our pre-check phase includes the following three steps (see Fig. 6).

Step 1. *Transaction delivery*. After receiving a transaction request, transaction manager (TM) produces a unique root transaction ID, which can be a function of current time to distinguish starting time of transactions. Next, TM divides the task into sub-transactions and distributes them to different sites which host specified services.

Step 2. *Resource pre-check*. On receiving the pre-check instruction, each participant begins to check all the needed resources through their resource managers. We still exemplify the scenario 2 in Section 2.3. Assume that $T_{1,1}$ successfully holds R_1 through M_1 and $T_{2,2}$ locks R_2 through M_2. M_1 and M_2 cache root transaction IDs $ID(T_1)$ and $ID(T_2)$, separately. Then, $T_{1,2}$ tries to lock R_2 through M_2. M_2 checks its cache and finds that R_2 has been locked by $T_{2,2}$ with the root ID $ID(T_2)$ ($ID(T_1) \neq ID(T_2)$). As a result, M_2 notifies $T_{1,2}$ that it cannot lock the R_2. However, $T_{1,2}$ will not be blocked. Instead, it immediately returns pre-check failed message to the TM of T_1.

Step 3. *Pre-check decision*. If the coordinator receives OK messages from all participants, it decides to send a *ready-for-prepare* message to them, and the two-phase commit begins. Otherwise, the coordinator decides to abort the transaction. In our example, T_1 gives up and releases its lock on R_1. On the other hand, if $T_{2,1}$ requests the resource R_1 after T_1 releases R_1, it can acquire the lock successfully and finally commit.

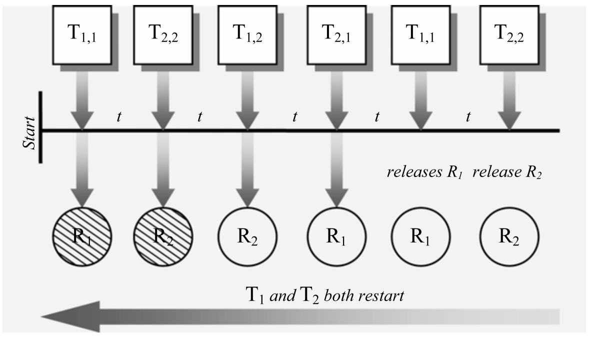

Fig. 7. A live-lock.

```
Input: Resource state $L_{R_k}$, transaction ID
Output: Resource lock or lock failure
1: if ($L_{R_k}$=0)
2:     $M_k$ directly locks $R_k$ for $T_{i,j}$;
3: else {
4:     if ($ID^{T_{i,j}}_{parent}$=ID($T_l$)) //for local deadlock
5:       { replicate $R_k$ as a $R'_k$;
6:         set a share lock on $R'_k$ for $T_{i,j}$;
7:         votes OK to the Coordinator; }
8:     else {          // for global deadlock
9:       if ( ID($T_i$) <ID($T_l$) ) {
10:          reject the request to $R_k$; }
11:          vote Failed to the Coordinator;
12:    else {
13:        $L^{T_{i,j}}_{R_k}$:=false;
14:        while (not $L^{T_{i,j}}_{R_k}$) and ($t<t_{timeout}$)
15:            if ( $L_{R_k}$=0 ) {
16:                lock $R_k$ for $T_{i,j}$;
17:                $L^{T_{i,j}}_{R_k}$:=true;
18:                votes OK to the Coordinator; } }
```

Fig. 8. A deadlock-free resource allocation algorithm.

3.2.2. Timestamp-based restart policy for global live-lock prevention

Our pre-check mechanism is able to prevent potential global deadlocks by releasing all competed resources when a resource conflict among multiple global transactions is detected; however, a live-lock may happen if these transactions restart and compete for resources simultaneously again. We exemplify a live-lock still using the scenario 2. As shown in Fig. 7, both $T_{1,2}$ and $T_{2,1}$ abort to request the resources R_2 and R_1, respectively, in terms of our pre-check policy, which in turn results in $T_{1,1}$ and $T_{2,2}$ also release the held resources R_1 and R_2, respectively. Finally, T_1 and T_2 fail to commit. After a little while, if T_1 and T_2 restart simultaneously, they potentially fail again due to the same resource competition. As a result, a live-lock occurs even though the resources R_1 and R_2 are free.

To avoid such live-locks, we selectively abort parts of conflicted transactions instead of rejecting all of them. For the fairness, the transactions with earlier starting time are paid higher execution priorities. So, we develop a timestamp based restart mechanism to choose which transaction should be aborted when a resource competition occurs. As we mentioned above, each transaction has a unique ID. The system can distinguish which transaction starts earlier in terms of their IDs.

We improve the Step 2 (i.e., *resource pre-check*) in our pre-check based deadlock prevention algorithm through introducing a timestamp-based restart policy. The basic idea behind this approach is the *first input first lock (FIFL)* in which the more early a transaction requests a resource, the more preferentially the transaction can lock the resource. Note that a transaction with an earlier starting time has a bigger transaction ID. For example, T_1 has a bigger transaction ID than T_2 because it starts earlier in the scenario 2. When a resource manager M_k receives a request to R_k locked by T_j previously, from a transaction, T_1, M_k does not reject the T_i' request if and only if T_i has a bigger transaction ID than T_j. Instead, it keeps the request for a timeout. If the locked R_k is released within the timeout, T_i will be able to lock the R_k. In this way, a transaction with an earlier starting time can have more priority to locking a resource. In our scenario 2, M_2 receives request to R_2 from $T_{1,2}$ and knows that it owns a bigger transaction ID than $T_{2,2}$. If T_2 is just at the pre-check phase, M_2 will wait for a timeout for $T_{1,2}$. On the other hand, M_1 finds that $T_{2,1}$ has a smaller transaction ID than T_1 so that it directly rejects $T_{2,1}$. Therefore, T_2 is aborted and $T_{1,2}$ can hold the resource R_2 if $T_{2,2}$ release its lock on R_2 in time. In this way, T_1 can lock both R_1 and R_2 and finally commit.

Combining the above two approaches for preventing both local deadlocks and global deadlocks, we propose a deadlock-free allocation algorithm, as shown in Fig. 8. The algorithm is executed on a resource manger M_k. In Step 4 of the algorithm, $ID^{T_{i,j}}_{parent} = ID(T_i)$ means that the resource R_k has been locked by other sub-transaction(s) of T_i. On the contrary, R_k has been locked by another global transaction T_l in Step 8. In that case, $T_{i,j}$ waits for T_l to release R_k within the timeout $t_{timeout}$. More specifically, if T_l cannot release R_k within $t_{timeout}$, $T_{i,j}$ will be rejected to access R_k.

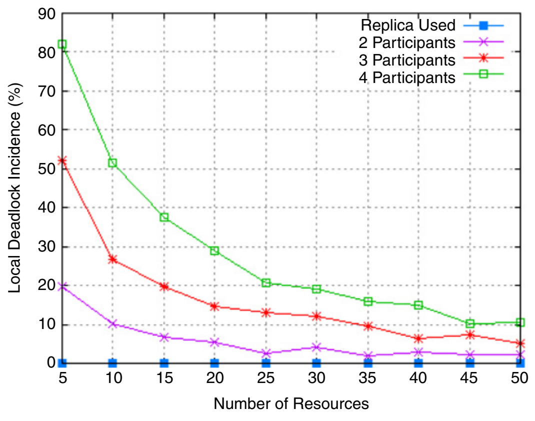

Fig. 9. Local deadlock incidence.

4. Experiments and performance evaluations

We have developed a system based on Globus Toolkit 4, in which deployed a few of business services. In our system, a coordinator (for a global transaction) communicates with a set of participants (each for a sub-transaction of the global transaction) based on remote procedure call, which guarantees that every node will not be blocked after remote calls. Once a node finishes transaction preparation or transaction commit, it sends a corresponding message to the caller. So we can use timeout mechanism to check if a deadlock occurs. Through this system, we comprehensively evaluated our resource allocation approaches for preventing both local deadlocks and global deadlocks.

4.1. Performance evaluation on the replication-based local deadlock prevention mechanism

In this experiment, we measure system performance using a *deadlock incidence*, which is a ratio of the number of deadlocked transactions to the number of all transactions in the system. We compared our replication-based local deadlock prevention mechanism (called *Replica Used*) with the traditional local resource allocation algorithm, in which a sub-transaction $T_{i,j}$ of T_i will be blocked if its resource request has been locked by another sub-transaction $T_{i,k}$ ($T_{i,j}, T_{i,k} \in T_i$; $i \neq j$). We tested how the deadlock incidence of two approaches changes with the number of resources as well as with the number of sub-transaction (i.e., participants in Fig. 9). In the current experiment, we varied the number of sub-transactions in a distributed transaction as 2, 3, and 4.

Fig. 9 indicates that the deadlock incidence in the traditional resource allocation scheme rapidly increases as the number of resources decreases. The result shows that if the number of resources is fewer than the number of participants (i.e., sub-transactions), the transaction is inevitably deadlocked in the traditional resource allocation scheme. Particularly, such transactions will become deadlocked again even though it restarts again. From Fig. 9, we also find that no deadlock happens in our replication-based local deadlock prevention approach. The reason is that if a resource is requested more than once by different sub-transactions of a global transaction, our scheme will duplicate the resource and all the sub-transactions share the replicated resource. As a result, the local deadlock is avoided no matter how many resources can be used.

4.2. Performance evaluation on the timestamp-based global deadlock prevention mechanism

In this part, we analyzed and compared the *deadlock incidence* and *average transaction processing time* in our timestamp based deadlock prevention mechanism and the traditional global resource allocation scheme in which a global transaction T_i will be aborted if its resource request has been locked by another global transaction T_j. For removing the affect of local deadlocks, we replicate a resource copy when multiple sub-transactions in a global transaction compete for a resource, as mentioned in Section 3.1.

(1) *Global deadlock incidence*

In this experiment, each global transaction includes 5 sub-transactions. We tested how the global deadlock incidence varies with the number of available resources as well as with the number of global transactions. According to Fig. 10, we can find that there are less conflicts among global transactions as the number of available resources increases. Also, the global deadlock incidence grows up as the number of global transactions increases. In the worst case, when 4 transactions, each with 5 sub-transactions, compete with each other for only 5 resources, the deadlock incidence goes up to 98%. On the contrary, in the best case, there is only 1 global transaction and no deadlock occurs because replication-based local deadlock prevention mechanism is also used.

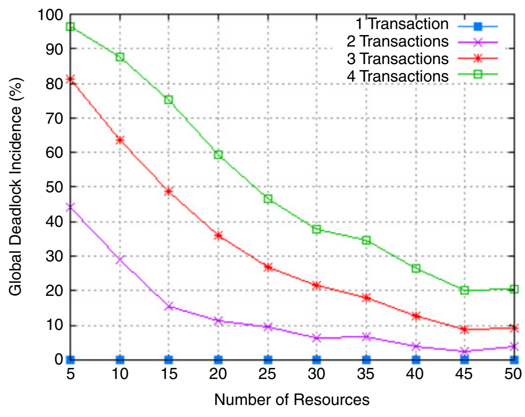

Fig. 10. Global deadlock incidence.

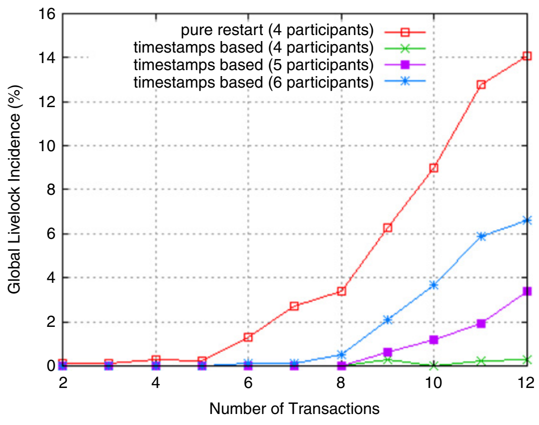

Fig. 11. Global live-lock incidence.

(2) *Global live-lock incidence*

We tested how many live-locks will occur if resource managers use the *pure restart policy*,in which conflicted global transactions that compete for the same resources are aborted immediately. With this policy, these conflicted global transactions potentially form a live-lock even though they restart again. So, we evaluate how much our timestamp-based mechanism can improve the live-lock incidence. In the pure restart policy, we set the restart times as 5, which means each global transaction may restart at most 5 times if it is not able to be ready for a transaction commit.

Fig. 11 shows the live-lock percentage of the *pure restart policy* and our *timestamp-based scheme*. The global live-lock incidence in both schemes grows up as the number of concurrent transactions increases. However, our timestamp-based scheme always outperforms over the pure restart policy, and the more a global transaction has sub-transactions, the higher the global live-lock incidence becomes. In particular, when there are only 4 sub-transactions in each global transaction, almost no live-lock occurs in our timestamp-based scheme. On the other hand, our timestamp-based scheme cannot completely avoid live-locks. The reason is that our timestamp-based scheme cannot always guarantee that each transaction gets all the needed resources.

In a word, the experiments demonstrate that our timestamp-based restart policy significantly reduces global deadlocks and global live-lock percentage, in spite that it cannot avoid live-lock completely.

(3) *Average processing time*

To evaluate how much our solution improves the system performance, we tested *average processing time* in the two solutions. Fig. 12 illustrates that the average processing time in our timestamp-based scheme is always lower than that in pure restart policy. It means that our scheme can achieve a better system throughput because the lower the average processing time is in a transaction system, the higher throughput the system will achieve.

5. Conclusions

We have presented a replication based approach to avoid local deadlocks, and a timestamp based approach to greatly mitigate global deadlocks for SOA environments. We, then, designed a general algorithm for both local and global deadlock prevention.

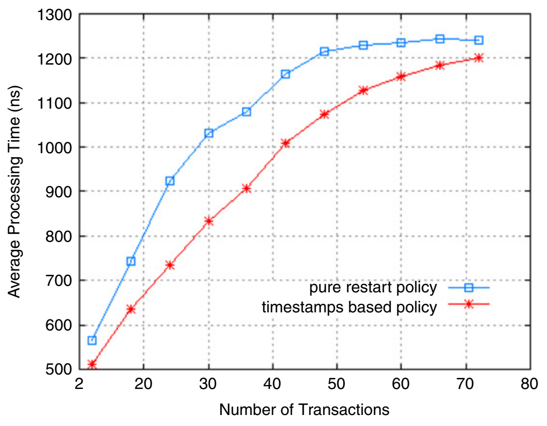

Fig. 12. Average processing time.

The experiment results demonstrate the effectiveness and efficiency of our solutions. First, our replication based approach completely eliminates local deadlocks. Next, our timestamp based scheme approach can significantly reduce the incidence of global deadlocks and corresponding global live-locks. And at the same time, it also improves the system performance.

Acknowledgments

Feilong Tang thanks The Japan Society for the Promotion of Science (JSPS) for providing excellent research environment for his JSPS Postdoctoral Fellow (ID No. P 09059) Program in The University of Aizu, Japan.

This work was supported by the National Natural Science Foundation of China (NSFC, No. 61073148), and Hong Kong RGC (No. HKU 717909E).

References

[1] G.K. Attaluri, K. Salem, The presumed-either two-phase commit protocol, IEEE Transactions on Knowledge and Data Engineering 14 (5) (2002) 1190–1196.
[2] F.L. Tang, M.L. Li, J. Huang, Real-time transaction processing for autonomic Grid applications, Engineering Applications of Artificial Intelligence 17 (7) (2004) 799–807.
[3] Ved P. Kafle, Masugi Inoue, Locator ID separation for mobility management in the new generation network, Journal of Wireless Mobile Networks, Ubiquitous Computing and Dependable Applications 1 (3) (2010) 3–15.
[4] L. Bai, M. Liu, Fuzzy sets and similarity relations for semantic web service matching, Computers & Mathematics with Applications 61 (8) (2011) 2281–2286.
[5] X.F. Di, Y.S. Fan, Y.M. Shen, Local martingale difference approach for service selection with dynamic QoS, Computers & Mathematics with Applications 61 (9) (2011) 2638–2646.
[6] Z.W. Li, M.C. Zhou, M.D. Jeng, A maximally permissive deadlock prevention policy for fms based on petri net siphon control and the theory of regions, IEEE Transactions on Automation Science and Engineering 5 (1) (2008) 182–188.
[7] S. Reveliotis, E. Roszkowska, J.Y. Choi, Correctness verification of generalized algebraic deadlock avoidance policies through mathematical programming, IEEE Transactions on Automation Science and Engineering 7 (2) (2010) 240–248.
[8] S.A. Reveliotis, E. Roszkowska, J.Y. Choi, Generalized algebraic deadlock avoidance policies for sequential resource allocation systems, IEEE Transactions on Automatic Control 52 (12) (2007) 2345–2350.
[9] S.A. Reveliotis, E. Roszkowska, On the complexity of maximally permissive deadlock avoidance in multi-vehicle traffic systems, IEEE Transactions on Automatic Control 55 (7) (2010) 1646–1651.
[10] A.K. Elmagarmid, A survey of distributed deadlock detection algorithms, ACM SIGMOD Record 15 (3) (1986) 37–45.
[11] J.R.G. Mendívil, J.R. Garitagoitia, A model for deadlock detection based on automata and languages theory, Computers & Mathematics with Applications 25 (6) (1993) 47–55.
[12] J. Ezpeleta, J.M. Colom, J. Martinez, A Petri net based deadlock prevention policy for flexible manufacturing systems, IEEE Transactions on Robotics and Automation 11 (2) (1995) 173–184.
[13] D. Taniar, S. Goel, Concurrency control issues in Grid databases, Future Generation Computer Systems 23 (2007) 154–162.
[14] S.G. Wang, C.Y. Wang, Y.P. Yu, Comments on siphon-based deadlock prevention policy for flexible manufacturing systems, IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans 41 (2) (2011) 338–340.
[15] J.M. Martinez-Rubio, P. Lopez, J. Duato, A cost-effective approach to deadlock handling in wormhole networks, IEEE Transactions on Parallel and Distributed Systems 12 (7) (2001) 716–729.
[16] T.W. Kuo, Y.T. Kao, C.F. Kuo, Two-version based concurrency control and recovery in real-time client/server databases, IEEE Transactions on Computers 52 (4) (2003) 506–524.
[17] C.T. Lu, J. Dai, Y. Jin, J. Mathuria, GLIP: a concurrency control protocol for clipping indexing, IEEE Transactions on Knowledge and Data Engineering 21 (5) (2009) 714–728.
[18] M. Hofri, On timeout for global deadlock detection in decentralized database systems, Information Processing Letters 51 (6) (1994) 295–302.
[19] M. Dotoli, M.P. Fanti, G. Iacobellis, Comparing deadlock detection and avoidance policies in automated storage and retrieval systems, in: Proceedings of IEEE International Conference on Systems, Man and Cybernetics, 2004 pp. 1607–1612.
[20] Y. Wang, M. Marritt, A. Romanovsky, Guaranteed deadlock recovery: deadlock resolution with rollback propagation, in: Proceedings of Pacific Rim Int'l Symp. Fault-Tolerant Systems, 1995, pp. 92–97.

[21] G.K. Attaluri, K. Salem, The presumed-either two-phase commit protocol, IEEE Transactions on Knowledge and Data Engineering 14 (5) (2002) 1190–1196.
[22] I. Foster, K. Czajkowski, D.E. Ferguson, et al., Modeling and managing state in distributed systems: the role of OGSI and WSRF, Proceedings of the IEEE 93 (3) (2005) 604–612.
[23] H. Wu, WN. Chin, J. Jaffar, An effcient distributed deadlock avoidance algorithm for the AND model, IEEE Transactions on Software Engineering 28 (2002) 18–29.
[24] Y.B. Ling, S.G. Chen, C.Y. Chiang, On optimal deadlock detection scheduling, IEEE Transactions on Computers 55 (9) (2006) 1178–1187.
[25] S. Lee, J.L. Kim, Performance analysis of distributed deadlock detection algorithms, IEEE Transactions on Knowledge and Data Engineering 13 (4) (2001) 623–636.