社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2008－DBS－146 (2)
2008／9／21

A Database Replication Middleware with Fine-Grained Concurrency Control

Takeshi MISHIMA † and Hiroshi NAKAMURA †

† Research Center for Advanced Science and Technology, The University of Tokyo
4-6-1 Komaba, Meguro-ku, Tokyo, 153-8904, Japan

e-mail: {mishima,nakamura}@hal.rcast.u-tokyo.ac.jp

Abstract Database replication is an essential art for high performance and availability. If replication functionalities can be implemented only in a middleware layer with no changes to existing database servers, we can distill a lot of benefits from the setting. However, it is a big challenge to realize the middleware without sacrificing throughput and consistency. In this paper, we propose LEFOMA, a new synchronous database replication middleware with tuple based concurrency control guaranteeing snapshot isolation, which needs no changes to database servers. The control avoids the deadlock against which synchronous replication middlewares commonly come up. Moreover, providing fine-grained locking, the control brings higher throughput. We show LEFOMA is not only practical but also low cost since we implemented a prototype on top of PostgreSQL, whose code size is very small. Our experimental results using TPC-C benchmark reveal that, compared to one of synchronous replication techniques, LEFOMA provides higher throughput for high concurrent execution environment.

Keywords synchronous replication, middleware, fine-grain, snapshot isolation, deadlock

1. Introduction

Database replication is an essential art for high performance and availability. If replication functionalities can be implemented only in a middleware layer with no changes to existing database servers, we can distill a lot of benefits from the setting. First, we can circumvent vendor lock-in, which we are dependent on a particular vendor, and construct a cost-effective replication system using existing any database server without modification at low cost. Second, considering difficulty of modifying database servers, because the source code of database servers is very huge and complex, the separation of replication functionalities from database servers promotes their independent evolution smoothly. Third, we can easily realize a heterogeneous database system, which helps us migrate from an old database system to a new one.

Middleware-based database replication approaches can be classified into *asynchronous* and *synchronous* replication. In asynchronous replication, a middleware updates only master replica in real time and later extracts a writeset[1] from the master and then applies it to slave replicas. Note that a temporal inconsistency between the master and the slaves can occur, which originates in asynchronous updates. In synchronous replication, a middleware keeps all replicas exactly synchronized by updating all the replicas simultaneously.

Middleware-based asynchronous replication has several shortcomings, especially, the writeset handling and the temporal inconsistency being serious, which limit the advantage of database replication, high performance and availability. First, the writeset handling with no modification of database servers causes the deterioration of throughput. To keep all replicas consistency, the order of updating master replica must be the same as the order of updating all slave replicas. Unfortunately, neither the writeset extraction nor application is a standard feature of database servers. Therefore, asynchronous middlewares must decide the order of the writesets by executing commit operations in a serialized fashion on a master and then apply the writesets to all slaves in the same order serially. Second, load balancing may not work out well because the latest data may be only in a master. All operations of update transactions[2] must be executed on a master, which may make the master bottleneck. To reduce the master's load, existing asynchronous approaches make middlewares pick out read-only transactions and then slaves execute them. However, a middleware can not distinguish read-only and update transactions unless applications do not inform the transaction's type to the middleware, which means that modification of existing API or application is needed. Even if read-only transactions can be submitted to slaves, their response times may increase because they can not be executed until the latest writesets are applied to the slaves. Moreover, both application of writesets and

[1] A writeset contains information necessary to recreate a transaction's modification on the master replica.

[2] An update transaction has at least one write operation.

execution of transactions can not be executed concurrently. Finally, a master being single point of failure, a master's fault may lost the latest data and destroy consistency between replicas.

Synchronous replication does not suffer from the problems mentioned above, having neither the writeset handling nor the temporal inconsistency. However, synchronous replication has another problem of concurrency control between replicas. If two write operations *conflict*[3] but the order of the operations in one replica is not the same as the orders in others, deadlock occurs. Surprisingly, there are very few middlewares which solve the deadlock. Although *Distributed versioning(DV)* [2] is an existing synchronous middleware which solves the deadlock, it has some shortcomings, the most serious being table based locking. This coarse-grained locking limits concurrent execution of operations and consequently incurs deterioration of throughput.

Unfortunately, the coarse-grained locking also spoils popular snapshot isolation (SI) [3] widely used in practice. This is one of the reasons that no existing synchronous replication middleware provides SI.

In this paper, we tackle the problem of the synchronous middlewares and contribute as follows:

- we propose LEFOMA, a new synchronous database replication middleware with tuple based concurrency control guaranteeing snapshot isolation, which needs no changes to database servers. The control avoids the deadlock against which synchronous replication middlewares commonly come up. Moreover, providing fine-grained locking, the control brings higher throughput.
- We implemented LEFOMA using PostgreSQL without no modification. The code size of LEFOMA is very small, less than 2000 lines in C language. These mean that LEFOMA is not only practical but also low cost.
- The experiments using TPC-C show that, compared to DV, LEFOMA provides higher throughput for high concurrent execution environment.

The remainder of this paper is organized as follows. Section 2 presents the features of database servers that we use. Section 3 addresses the deadlock against which synchronous middlewares commonly come up and the existing solution of the deadlock. Section 4 proposes LEFOMA, a new synchronous database replication middleware with tuple based concurrency control guaranteeing snapshot isolation. Section 5 presents experimental results. Section 6 talks about related work. Section 7 concludes the paper.

[3] Two write operations try to change the same data item simultaneously.

2. Features of Database Servers

In this section, we address features of database servers that we use, which provides SI with the *First Updater Wins* rule.

2.1 Snapshot Isolation

SI is a multi-version concurrency control [3]. A transaction T_i executing under SI acquires its own snapshot, the committed state of the database, at the time that T_i started, designated as $start(T_i)$. In popular database servers such as Oracle and PostgreSQL, $start(T_i)$ is the time just before the first read or write operation of T_i has been executed and its snapshot is given by executing the first operation implicitly.

Read and write operations of T_i are executed on its own snapshot. The snapshot is changed by its own write operations, so if T_i reads data it has previously written, it will read its own output. The snapshot is not altered by any write operations of any other transactions. Therefore, T_i may read stale data that another transaction has changed and committed after $start(T_i)$. This is the reason why SI is weaker than one-copy-serializability (1SR)[4][4]. Changes of T_i's snapshot are reflected to the original database at the time T_i committed successfully, represented as $end(T_i)$. We say that two transactions T_1 and T_2 are *concurrent* if the interval in time from start(T_1) to end(T_1) and that from start(T_2) to end(T_2) overlap.

Although SI is weaker than 1SR, SI is implemented as the strongest transactional guarantee in popular database servers such as Oracle and PostgreSQL. Under SI, read operations never are blocked by write operations and vise versa. Only conflicting write operations may be blocked. This increases the concurrent execution of operations and thereby brings higher performance than 1SR.

2.2 The First Updater Wins Rule

we assume that database servers use *First Updater Wins* rule[7], which is implemented popular database servers such as Oracle and PostgreSQL, to prevent the lost updates anomaly.

If transactions T_i and T_j are concurrent, and T_i updates the data item x, then it will take a write lock on x; if T_j subsequently attempts to update x while T_i is still active, T_j will be prevented by the lock on x from making further progress. If T_i then commits, T_j will abort; T_j will be able to continue only if T_i drops its lock on x by aborting. If, on the other hand, T_i and T_j are concurrent, and T_i updates x but then commits before T_j attempts to update x, there will be no delay due to locking, but T_j will abort immediately when it attempts to update x (the abort does not wait until T_j attempts to commit).

[4] The resulting schedules are equivalent to a serial schedule on a single database.

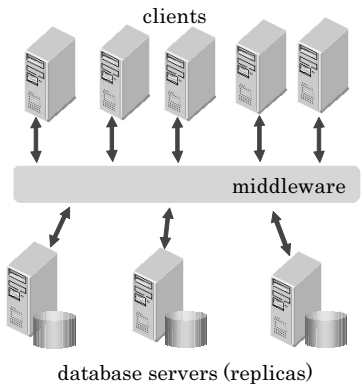

Fig.1 Model of a Replicated Database System

In short, only *winner* write operations can make progress and a transaction will successfully commit only if no other concurrent transaction causing conflict has already been committed.

3. Synchronous Replication Middlewares

In this section, we address the deadlock against which synchronous middlewares commonly come up and the existing solution of the deadlock.

3.1 Model

Figure 1 shows the model of a replicated database system with a synchronous replication middleware. It has a cluster of off-the-shelf database servers (replicas) each running on a commodity computer with no shared disk. Replicas may be heterogeneous but all replicas have the same database. A middleware lies between clients and replicas, sending a request which includes a read, write, commit or abort operation from a client to replicas. A synchronous replication middleware commonly uses a *read-one-write-all* approach, in which a read operation can be executed on any replica whereas a write operation is executed on all replicas. Receiving a request which includes a write operation from a client and sending it to all replicas, the middleware does not send back a response to the client until it receives responses from all replicas.

3.2 Deadlock

We address the deadlock against which synchronous replication middlewares commonly come up. Consider two replicas R_1 and R_2 with the First Updater Wins rule, each of which executes two conflicting write operations w_a of a transaction T_a and w_b of a transaction T_b. If, in R_1, w_a gets a write lock and therefore w_b waits for the lock, the middleware receives an only response of w_a from R_1. Otherwise, if, in R_2, w_b gets a write lock and therefore w_a waits for the lock, the middleware receives an only response of w_b from R_2. In T_a, the middleware does not make progress until it receives a response of w_a from R_2. However, the middleware does not receive it until T_b commits/aborts and then the write lock is given to w_a in R_2. Inversely, in R_1, w_b waits T_a commits/aborts. In summary, T_a waits for completion of T_b and T_b waits for completion of T_a, the middleware suffering from deadlock.

3.3 Solution of Existing Middlewares

Surprisingly, there are very few middlewares which solve the deadlock. Neither [5] nor [8] provides the solution of the deadlock. Therefore, with these middlewares, the more frequency of write operations grows, the more probability of the deadlock increases. Because it is difficult to find out and then get rid of the deadlock, they are not useful for applications with write operations.

Distributed versioning(DV)[2] is an existing synchronous middleware which solves the deadlock. DV needs its own peculiar programming model: programmers insert a *pre-declaration* of the tables which will be accessed at the beginning of each transaction, and a *last-use-declaration* after the last use of a particular table. DV consists of three types of processes: *scheduler*, *sequencer*, and some *database proxies*. A scheduler distributes incoming requests on replicas and delivers responses to clients. When the scheduler receives a pre-declaration, the sequencer assigns unique version numbers to the tables accessed by each transaction. Receiving an operation, the scheduler picks out the unique version number of the table which the operation will access, and submits the operation with the number to database proxies. Each database proxy regulates access to a replica by letting the only operation with an active version number proceed to a replica. An operation without an active version number waits until its version is available. New versions become available when a database proxy receives a last-use-declaration. In this way, all operations on a particular table are executed at all replicas in version number, and consistency between replicas is kept synchronously.

Unfortunately, DV has some shortcomings, the most serious being table based locking. This coarse-grained locking incurs unnecessary waiting. Consider two non-conflicting write operations w_a of a transaction T_a and w_c of a transaction T_c which access the same table. If w_a has an active version number of the table, w_c without an active version number waits although w_c never conflict with w_a.

4. LEFOMA

In this section, we propose LEFOMA, a new synchronous database replication middleware with tuple based concurrency control guaranteeing SI, which needs no changes to existing database servers. We assume that every replica provides SI with the First Updater Wins rule as described in Sect.2.

4.1 Basic Concept

Our main idea is to control requests submitted from clients to replicas with the two rules as follows.

- **Snapshot Creation Rule:** LEFOMA makes all replicas create the same snapshot for any pair of transactions on different replicas.
- **Write Operations Rule:** LEFOMA makes all replicas execute conflicting write operations on the same snapshots in the same order.

4.2 Snapshot Creation Control

Our first key idea to realize the snapshot creation rule is that when LEFOMA submits commit operations to all replicas, it never does the first operations until it receives all responses of the commit operations from all of them, and vice versa. That is, LEFOMA makes the relative order of the first and commit operations of different transactions be the same in all replicas.

As stated in Sect.2, transaction T_1 gets T_1's snapshot at $start(T_1)$ and T_1's changes to the snapshot are reflected to original database at $end(T_1)$. Thus, considering another transaction T_2, it depends the relative order of T_1's commit and T_2's first operation whether T_2's snapshot includes T_1's changes or not. That is, if T_1's commit operation proceeds T_2's first operation, T_2's snapshot must include T_1's changes. Otherwise, if T_2's first operation proceeds T_1's commit operation, T_2's snapshot never include T_1's changes. Thus, the relative order of T_1's commit and T_2's fist operation influences T_2's snapshot.

Consequently, if the relative order of commit and the first operations in different replicas is the same, all replicas must create the same snapshot for any pair of transactions on different replicas.

4.3 Write Operations Control

Our second key idea to realize the write operations rule is to use tuple based locking on only conflicting write operations. However, it has been considered very difficult because a middleware can not know whether write operations will conflict or not in advance, even if it parses SQL statements. Otherwise, we can imagine to extend DV from table based locking to tuple based locking but it is impractical to manage version numbers per tuple. These are the reasons why existing synchronous middlewares can not achieve tuple based locking but table based locking.

To solve this difficulty, LEFOMA delegates one replica called *leader* to decide the execution order of write operations and to use tuple based locking on conflicting write operations, and then provides *virtual tuple based locking* to the others called *followers*.

Our proposed write operations control is that LEFOMA regulates write operations as follows:

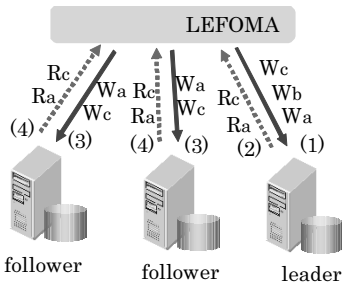

Fig.2 Write Operations Control

(1) sends all write operations only to the leader,
(2) receives responses from the leader,
(3) sends write operations of which LEFOMA has received responses to all the followers,
(4) receives responses from all the followers, and
(5) sends back one response to each client.

Consider, for instance, LEFOMA sends three write operations W_a, W_b and W_c corresponded to transaction T_a, T_b and T_c, respectively shown in Fig.2. We assume that W_a and W_b will conflict but of course LEFOMA can not know it in advance. After receiving write operations W_a, W_b and W_c from clients, LEFOMA sends all the write operations only to the leader (step(1)). Then LEFOMA receives responses R_a and R_c corresponded to W_a and W_c, respectively (step(2)). However, LEFOMA cannot receive the response R_b corresponded to W_b since W_a and W_b conflict and R_a is returned to the coordinator. Recall the First Updater Wins rule as described in Sect.2: the only write operation that can get a write lock can progress but the others have to wait for the lock. At this time, LEFOMA knows that write operations corresponded to responses which LEFOMA has received will not cause conflict in the followers, i.e., W_a and W_c will not conflict. Then LEFOMA performs *virtual tuple based locking* to all the followers: LEFOMA sends only W_a and W_c to all the followers and keeps W_b in LEFOMA (step(3)). This is as if W_a got a write lock but W_b waited for the lock in the LEFOMA layer. Then LEFOMA receives responses from all the followers (step(4)) and sends back one response to each client (step(5)). W_b in the leader can not progress until T_a commits or aborts, and W_b in the LEFOMA layer can not progress until LEFOMA receives R_b.

In summary, LEFOMA knows which write operations are winner or loser of conflict by using the feature of the First Updater Wins rule, responses of only winner write operations are returned, and then emulates it in the LEFOMA layer.

If any replica executes non-conflicting write operations on the same snapshot in any order, the final result is the

same. Therefore, LEFOMA makes all replicas execute non-conflicting write operations in any order, possibly concurrently (step(3)). This brings higher throughput than any other existing synchronous middlewares.

4.4 Load Balancing

Rather than asynchronous replication, LEFOMA can make only one replica execute any read operation, even if it belongs update transactions, because LEFOMA makes all replicas have the latest data synchronously. This means LEFOMA can distribute requests over replicas with per-operation load balancing unlike any asynchronous replication with per-transaction. Of course, LEFOMA need not be informed the transaction type, a read-only or an update transaction.

4.5 Availability

If the leader falls into malfunction, LEFOMA re-selects a new leader from followers. Clients must re-submit the transactions that have not been finished. If a follower stops due to some trouble, only LEFOMA detaches the follower from the system, i.e., the LEFOMA does not send any requests after that. LEFOMA masks follower's failure but does not the leader's. However, remained replicas keep consistency unlike asynchronous replication. This means LEFOMA provides fail-safe.

4.6 Dummy Read Operation

If the first operation is a write operation, LEFOMA may encounter dead-lock. Consider a write operation w_1 is the first operation of a transaction T_1 and waits for a lock which has been gotten by w_2 of a transaction T_2. With the First Updater Wins rule, w_1 can not get the lock until T_2 commits or aborts. However, T_2 can not commit until the first operation w_1 has ended.

To avoid this problem, if the first operation is a write operation, LEFOMA submits dummy read operations as the first operations to create snapshots to all replicas, receives responses and then submits the write operations.

4.7 Advantages

LEFOMA can guarantee SI by making all replicas create the same snapshot and execute conflicting write operations on the snapshot in the same order. Moreover, LEFOMA can avoid the deadlock by making all follower replicas execute conflicting write operations in the order that the leader does. Furthermore, LEFOMA can bring higher throughput by making all replicas execute non-conflicting write operations concurrently and any replica execute a read operation. To the best of our knowledge, LEFOMA is the first middleware which brings these advantages.

5. Evaluation

In order to clarify the effectiveness of LEFOMA, we conducted the performance evaluation with TPC-C by comparing throughput of LEFOMA with that of DV.

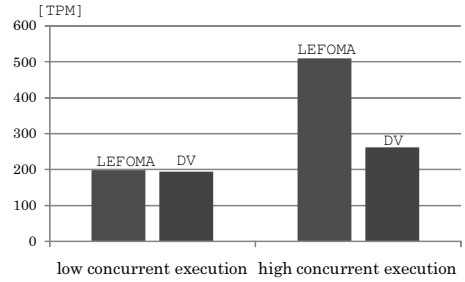

Fig.3 Throughput Comparison

5.1 Prototypes

We implemented a prototype of LEFOMA on top of PostgreSQL without modification, whose code size is less than 2000 line in C language. Because our proposal need not modify existing database servers, not only is the portability of LEFOMA high but the cost of LEFOMA also is low.

In addition, we implemented another prototype of DV on top of PostgreSQL so as to compare throughput of LEFOMA with that of DV and discuss the effectiveness of LEFOMA. Originally, DV has been proposed to guarantee 1SR. To compare LEFOMA and DV fairly, we modified the original DV to guarantee SI and implemented it with the modification.

5.2 Experimental Setup

Each prototype comprises two replicas, one middleware node and one client simulator node. All nodes are the same specification (two 2.4GHz Xeon, 2GBytes RAM and one 70GB SCSI HDD) and connected by 1Gb/s Ethernet. We used PostgreSQL, version 8.3.3, as replicas providing SI. We set the `default_transaction_isolation` to SERIALIZABLE[5], and did not change the other parameters. To measure the performance of the prototypes, we have implemented the main parts of TPC-C benchmark.

5.3 Results and Discussion

To observe how the difference between concurrency control of LEFOMA and that of DV has an influence on throughput, we measured two cases, *high concurrent execution environment* and *low concurrent execution environment*. To change the degree of concurrency, we adjusted the number of *warehouse* and *Think Time*. We selected 10 and 12000 as *warehouse* and *Think Time* respectively for the low concurrent execution environment, and 30 and 1000 as those for the high concurrent execution environment. In a preliminary experiment, we had observed that in the case of the low concurrent execution environment, there was very little concurrent execution, and in the case of the high concurrent execution environment, there was much concurrent execution.

[5] SI is called SERIALIZABLE in PostgreSQL.

Figure 3 shows the measured throughputs of LEFOMA and DV prototypes for the high and low concurrent execution environments. In the low concurrent execution environment, there is slight difference between LEFOMA and DV throughputs. This is because, in the absence of concurrent execution, it is not important whether fine-grained or coarse-grained concurrency control. In the high concurrent execution environment, LEFOMA and DV throughputs are approximately 511 TPM and 263 TPM, respectively. LEFOMA outperformed DV by approximately 94 % in throughput. This confirms that LEFOMA achieves higher throughput than DV. This is because LEFOMA has fain-grained concurrency control, executing non-conflicting write operations concurrently.

6. Related Work

We have already described middleware based approaches can be categorized into two categories and asynchronous middlewares [14, 11, 1, 9, 10, 12, 13, 6] suffer from serious problems. LEFOMA differs from them in that it does not incur the problems because LEFOMA is a synchronous replication, which has neither the writeset handling nor the temporal inconsistency.

The main difference between LEFOMA and the other synchronous middlewares is that LEFOMA can execute finer grained locking and guarantee SI. Details of other features are as follows.

Cecchet et al. [5] introduce C-JDBC which does not have concurrent control for avoiding the deadlock and keeping consistency. These are responsible to users. Fujiyama et al. [8] introduce a middleware being similar to C-JDBC. So, it can not avoid the deadlock nor keep consistency.

Amza et al. [2] propose *distributed versioning*, which can avoid the deadlock and guarantee 1SR. However, its concurrency control is coarse grain.

7. Conclusion

In this paper, we proposed LEFOMA, a new synchronous database replication middleware with tuple based concurrency control guaranteeing snapshot isolation, which needs no changes to database servers. Not only does the control avoid the deadlock against which synchronous replication middlewares commonly come up, but the control also provides fine-grained locking.

In order to clarify the effectiveness of LEFOMA, we made prototypes of LEFOMA and DV and performed a experiment using TPC-C in terms of throughput. The experimental results on the prototypes showed that LEFOMA outperformed DV by approximately 94% for the high concurrent execution environment. This means that fine-grained concurrency control is effective for applications with high concurrent execution of write operations. Moreover, the code size of LEFOMA is very small. It is concluded that LEFOMA is very practical and effective.

References

[1] Fuat Akal, Can Türker, Hans-Jörg Schek, Yuri Breitbart, Torsten Grabs, and Lourens Veen. Fine-grained replication and scheduling with freshness and correctness guarantees. In *VLDB*, pages 565–576, 2005.

[2] Cristiana Amza, Alan L.Cox, and Willy Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *Middleware*, pages 282–304, 2003.

[3] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *ACM SIGMOD*, 1995.

[4] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, Massachusetts, 1987.

[5] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. C-jdbc: Flexible database clustering middleware. In *USENIX*, 2004.

[6] Sameh Elnikety, Steven Dropsho, and Fernando Pedone. Tashkent: Uniting durability with transaction ordering for high-performance slalable database replication. In *EuroSys*, 2006.

[7] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 30(2):492–528, June 2005.

[8] Kenichiro Fujiyama, Nobutatsu Nakamura, and Ryuichi Hiraike. Database transaction managemen for high-availability cluster system. In *PRDC*, 2006.

[9] Yu Lin, Bettina Kemme, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *ACM SIGMOD*, 2005.

[10] Marta Patiño-Martínez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. Middle-r: Consistent database replication at the middleware level. *ACM Transactions on Computer Systems*, 23(4):375–423, November 2005.

[11] Chirstian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In *Middleware*, 2004.

[12] Christian Plattner, Gustavo Alonso, and Özsu. Dbfarm: A scalable cluster for multiple databases. In *Middleware*, 2006.

[13] Christian Plattner, Gustavo Alonso, and Özsu. Extending dbmss with satellite databases. *VLDB Journal*, 2006.

[14] Uwe Röhm, Klemens Böhm, Hans-Jörg Schek, and Heiko Schuldt. Fas – a freshness-sensitive coordination middleware for a cluster of olap components. In *VLDB*, 2002.