Data Synchronization and Resynchronization for Heterogeneous Databases Replication in Middleware–based Architecture

Guoqiong Liao
School of Information Technology, Jiangxi University of Finance and Economics, Nanchang 330023, China
Jiangxi Key Laboratory of Data and Knowledge Engineering, Nanchang 330023, China
liaoguoqiong@163.com

Abstract—Currently more and more web applications and telecommunication applications are required to be separation with database layer. A general solution is to employ a middleware-based architecture, including an application client tier, a database front end (DB-FE) tier, also called middleware tier, and a database backend (DB-BE) tier. This paper mainly focuses on the issues of data synchronization and resynchronization in the case of failures for the architecture. Firstly, a synchronous update everywhere replication model and a replication 2-phase commit protocol (R2PC) are discussed, which can increase update success ratio of replication transactions through relaxing the commit constraints, and enhance DB-BEs’ availability using transaction retry and discard mechanisms. Then, a novel method using request logs is suggested for data resynchronization. The method only records the missing transaction requests for the unavailable DB-BEs in the DB-FEs and links the logs belonging to a site together, to reduce the resynchronization time. Moreover, the method doesn’t interrupt the normal transaction processing in the DB-BEs, so system throughput using request logs is higher than using transaction logs. Experiences validate that the suggested methods have better performance.

Index Terms—synchronization replication, data resynchronization, heterogeneous databases, middleware-based architecture

I. INTRODUCTION

With the growing numbers of subscribers and the introduction of new additional services, a large of web applications and telecommunication applications such as home location register (HLR) and home subscriber service (HSS), are required to be separation with database layer and high availability. A general solution is to employ a middleware-based replication architecture shown as Figure 1.1,2,3

The architecture consists of three tiers:

- The database backend (DB-BE) tier, which holds the actual user/subscriber data, executes the transaction requests from BE-FEs, and returns results/responses back to the DB-FEs. For guaranteeing high availability of databases, N (N≥3, Figure 1 shows a case of N=3) DB-BEs are organized into a cluster for data redundancy, which could be heterogeneous and co-located or spread out to a wider geographic region.

- The database front end (DB-FE) tier, also called middleware tier, which is responsible of receiving the requests from ACs, routing the requests to DB-BEs, and passing on the results, either a success or failure, back to the ACs.

- The application client (AC) tier, which initiates requests from clients. An AC may connect to one or more DB-FEs for better performance, e.g. load balancing and connection redundancy.

It is assumed that these elements, including the ACs, DB-FEs and DB-BEs, are all connected to form a network and they are supported by some network protocols, e.g. Ethernet, TCP/IP, ATM, or etc. Some characteristics of such a network are:

- Redundant connection. There are redundant physical connections between ACs and DB-FEs, DB-FEs and DB-BEs. With redundant connections, any single point of failure will not have impact to the overall system service availability.

- Replicated Data. All data objects in the databases are fully replicated in N DB-BEs among one cluster to make data continuously available to the DB-FEs in the event of a component failure, which could be due to a hardware node, software or a connection failure. Whenever an object in the database is to be updated, the update has to be...
replicated in all of the N instances to ensure a consistent database view, i.e., synchronous replication.

The purpose of synchronous replication is to guarantee all replicas of every data object in a cluster always have the same image. But one problem of synchronous replication is that it may lead to an increased response time due to waiting for the completion of all the replicas’ updates. Another problem is an update request has to be rejected if one replica can’t be updated successfully, to ensure data consistency across all replicas. The increased level of update failure will have negative impact to the high availability concept for such architecture.

In the other hand, there will be instances of network or other failures that stop data from being replicated to all replicas. When this occurs, the replicas on the failed servers will become out of synchronization. Therefore, a resynchronization process must take place to ensure the failed servers can catch up of the missed updates. How to “catch up” the missed updates but without interrupting the services in the active servers is also a challenge for this architecture.

This paper mainly focuses on data synchronous and data resynchronization in the case of failures in the middleware-based replication architecture. The remaining parts of the paper are organized as following. Section 2 is the introduction of related works. Section 3 gives a synchronous update everywhere replication model and a commit framework for the architecture. Section 4 gives some rules and mechanisms for replication transactions. Section 5 describes a replication 2-phase commit protocol, including commit state graphs and algorithms both for the coordinator and participants. Section 6 suggests a novel resynchronization method using request logs. Section 7 is performance testing and evaluation. Section 8 concludes the paper.

I. RELATED WORKS

In general, database replication protocols can be categorized into two classes by the time when update propagation to databases takes place, eager vs. lazy[6]. In eager replication schemes, i.e. synchronous replication, updates are propagated within the boundaries of a transaction. In this scheme, the user does not receive the commit notification until sufficient copies in the system have been updated. Lazy replication schemes, i.e. asynchronous replication, on the other hand, update a local copy, commit and only some time after the commit, the propagation of the changes take place. The first approach provides consistency in a straightforward way but it is expensive in terms of message overhead and response time. Lazy replication allows a wide variety of optimizations, however, since copies are allowed to diverge, inconsistencies might occur.

In the other hand, by the dimension of where the updates can take place, replication protocols can also be classified into primary copy (master) and update everywhere (group) approaches[7]. The primary copy approach requires all updates to be performed first at one copy (the primary or master copy) at a master server and then at the other copies (the secondary copies) at secondary servers. This simplifies replica control at the price of introducing a single point of failure and a potential bottleneck. The update everywhere approach allows any copy to be updated, thereby speeding up access but at the price of making coordination more complex.

In most present industrial products, one standard solution for synchronous replication is the Two-Phase Commit (2PC) protocol such as XA model[8], which makes global decision by following rules:

- all vote “YES” ⇒ global commit
- one votes “NO” ⇒ global abort

But for replication transactions, the rule of making global commit is too strict since it requires all replicas can be updated successfully, and the rule of making global abort is too relaxed since it will lead to aborting a lot of transactions due to a single site’s failure. Moreover, an update transaction can’t be committed if one replica always say “NO”. Therefore, the rules are inconsistent with the requirement of high availability.

In addition, as long as one site votes “NO”, the transaction will be aborted regardless of the reason of voting “NO”. In general, if a replication transaction can commit in one site, it should also execute successfully in all other available sites if there is no site failures and link failures. However, it is possible for a replication transaction that one site votes “YES” while another site votes “NO” due to some temporal software or hardware failures, like temporary memory shortage, deadlock, or software bug. In this case, the whole transaction will also be aborted, which will lead to wasting a lot of system resources.

There are many 2PC-related protocols, like presumed commit (PC), presumed abort (PA), three-phase commit (3PC)[9], and etc, which are designed to enhance the performance of 2PC. Although 3PC eliminate the blocking in 2PC, all of them don’t give any special treatment for replication transactions.

Now almost all databases replication products, including disk-resident databases (such as Oracle, DB2, Sybase, etc.) and In-Memory Database (IMDB) (such as Timesten), use transaction logs to handle resynchronization. That is, when receiving a resynchronization request from a failed server, the active DB-BE will send the logs of the missed committed transactions (called resynchronization logs) to it. Then the failed server can replay the missed updates by the logs.

Since it is necessary for each transaction to write its logs to a stable log file before it commits, no additional log overhead needs for resynchronization in the method based on transaction logs. But the method has some disadvantages for resynchronization [10-12]:

- As one replication server has lost communication for a long time, a great amount of logs need be stored in the active DB-BE, even if many checkpoint and backup operations have been completed.
- The transaction logs are essentially designed for recovery processing, to undo the uncommitted transactions and redo the committed transactions.
Since all transactions are recorded into the logs by appending data at the end of the log, the fetcher of the resynchronization logs in the active server will take time to distinguish which logs are required for resynchronization. Also the update process in the failed servers will spend time to analyze which part in a log is the after image of the missed data.

- The procedure of resynchronization may interrupt the normal transaction processing in the active server, since the transaction logs must be accessed exclusively by both the transaction manager (log write) and the fetcher of the resynchronization logs (log read) in the active server.
- The methods can’t be extended to the replication environment with heterogeneous databases, since heterogeneous databases have different log formats.

So far as we know, no any special resynchronization solution has been suggested for the three-tier architecture.

II. REPLICATION MODEL IN THE MIDDLEWARE-BASED ARCHITECTURE

A. Synchronous update everywhere replication model

In the middleware-based architecture, the middleware (DB-FE) tier acts as an important role for all replication transactions. When a DB-FE receives a request from an AC, it will determine which cluster the request should be forwarded according to data distributed information, and then select one or more DB-BEs in the cluster to process the request.

We use synchronous update everywhere replication model for the architecture since it needs less response time than the primary copy model (Figure 2). In the model, a replication procedure can be divided into five phases:

Figure 2. Synchronous update everywhere replication model

**Phase 1** (Request forwarding): After receiving an update request from a Client, the DB-FE will forward the request to all available DB-BEs in the specified cluster.

**Phase 2** (Lock Coordination): One of the database servers sends a lock request to all other servers for asking whether to grant the lock or not. If the lock can be granted by all sites, the transaction can proceed. If not, the transaction will be delayed till all sites agree to grant the lock.

**Phase 3** (Execution): When all the locks are granted, the operation will be executed at all sites.

**Phase 4** (Commit Coordination): During the commit coordination phase, a commit protocol is used to make sure that all sites can commit the transaction. The commit coordinator is acted by DB-FE.

**Phase 5** (Client Response): When the transaction is completed, the DB-FE returns a response, either success or failure to the Client.

In this model, a DB-FE can forward transaction requests by their type:

- If the request is a read request, the DB-FE will forward it to one of the available DB-BEs in the cluster by the load balance strategy used.
- If the request is an update request, the DB-FE will forward it to all available DB-BEs in the cluster, instead of a master server.

Therefore, updates on all replicas of a data object in the model are done almost simultaneously, so it can reduce the response time largely. However, in order to ensure the consistency of all copies, distributed locking approaches should be used to resolve data access conflicts, i.e., an object can only be accessed after all of its replicas have been locked. Therefore, for each update transaction, a lock coordinator phase should be added, which will increase transaction overhead.

B. Middleware-based Commit Framework

In the model mentioned above, a new kind of commit protocol should be designed to synchronize all N replicas in a cluster. The DB-FE in the middleware tier will act as a coordinator, and all DB-BEs in the cluster are the participants of the replication transactions. The commit framework for the middleware-based replication is shown as Figure 3.

Figure 3. The commit framework for the middleware-based replication

In this framework, before a DB-FE forwards a request, it should know the up-to-date state of all DB-BEs. Therefore each DB-FE should keep heartbeats with all DB-BEs. In the view of DB-FEs, a DB-BE should be in one of the following states at any time (Figure 4):

- **DOWN** state indicates the connection between the DB-FE and the DB-BE is broken, i.e., without any heartbeat message.
• **OUT-OF-SYNC** state indicates the connection between the DB-FE and the DB-BE is linked but the data in the DB-BE is out of synchronization. When a DB-BE is in “OUT-OF-SYNC” state, it can’t become “SYNC” till catching up with all the missed updates through a resynchronization process.

• **SYNC** state indicates the data in the DB-BE is consistent with other DB-BEs in the same cluster.

![State graph of DB-BEs](image)

**III. RULES AND MECHANISM FOR REPLICATION TRANSACTIONS**

In order to ensure high availability of the replication databases, we should design a new replication commit protocol for the middleware-based replication transactions with synchronous requirements.

**A. Base commit rules for replication transactions**

It has been discussed in Part I that the commit rules of 2PC are not suitable for the synchronous replication transactions. First, two base commit rules for replication transactions are suggested as following:

**RR1.** If one participant votes “YES”, the coordinator can make a global commit decision.

**RR2.** Only if all participants vote “NO”, the coordinator can make a global abort decision.

According to the new rules, a replication transaction may commit as long as one site votes “YES”, and whether it can abort must wait for a global abort decision. This is just opposite to the standard 2PC, which allows a site to abort unilaterally, and it must wait for a global commit decision after voting “YES”.

However, although the participants are allowed to commit unilaterally in the end, it can’t commit till receiving a global commit decision, to ensure all replicas can be updated synchronously.

**RR3.** The participant who votes “YES” can commit only receiving a global commit decision from the coordinator.

**B. Transaction retry mechanism**

In order to reduce the probability of transaction abort due to temporary failures, if at least one participant has voted “YES”, the participant who has voted “NO” should retry.

Figure 5 is an example of retry mechanism. Both DB-BE1 and DB-BE2 voted “YES”, and DB-BE3 voted “NO”. The coordinator should send a retry message to DB-BE3. When C receives a retry message, it will retry the transaction till timeout or success to revote “YES”. No matter what DB-BE3 votes, the coordinator can make a global commit decision and send the decision to both DB-BE1 and DB-BE2. The additional work is, if DB-BE3 cannot success till timeout, the coordinator will indicate its state as “OUT-OF-SYNC”, and does not forward any new request to it until it has gotten the missed updates. Therefore, DB-BE1 and DB-BE2 can keep database services unceasingly, and the inconsistent replica in DB-BE3 will not be seen by applications.

Therefore, we have following rule:

**RR4.** If there is any participant voting “YES”, the participants voting “NO” should retry till timeout or success to revote “YES”.

![Transaction retry mechanism](image)

**C. Transaction discarding mechanism**

In replication environment, two kinds of DB-BEs should become unavailable, i.e., the sites in “DOWN” state or in “OUT-OF-SYN” state. For ensuring the applications can always see the consistent replicas of all data objects, the DB-FEs never forward requests to the unavailable DB-BEs. Therefore, in order to improve the availability, the DB-FEs can discard following participants in the decision phase:

- The DB-BEs become “DOWN” because of broken link.
- The DB-BEs vote “NO” or timeout after retrying.

As to the first kind of transactions, since their sites can’t communicate with DB-FEs, the data objects in the sites cannot be accessed by DB-FEs. When the sites resume connection to DB-FEs, they will enter “OUT-OF-SYN” state first, and then do resynchronization to get the missed updates, which will be discussed in Part V.

As to the second kind of transactions, when they are discarded, the states of their sites will be marked as “OUT-OF-SYN” at once, and the DB-FE doesn’t forward any new request to it until it becomes “SYNC”.

In Figure 6, DB-BE3 is in “DOWN” state. Since the coordinator cannot receive any vote from DB-BE3 as timeout, it can decide to commit the transactions on DB-BE1 and DB-BE2, and discard the replication transaction in DB-BE3. But it should identify the state of DB-BE3 as “DOWN”, and does not forward any read or write requests to it.
RR5. If the coordinator finds a participant in “DOWN” state during commit processing, the participant will be discarded.

RR6. If there are at least a participant who votes “YES” and a participant who votes “NO”, the coordinator will discard the participants voting “NO”, and denote their states as “OUT-OF-SYNC”.

IV. REPLIITIOIN TWO-PHASE COMMIT PROTOCOL (R2PC)

A two-phase commit protocol integrating the features discussed in Part VI is called a replication two-phase commit protocol (R2PC), which can also be divided into two phases:

- Vote phase: the participants vote their execution results “YES” or “NO” to the coordinator.
- Decision phase: the coordinator makes a global commit or abort decision and forward it to the participants

A. Coordinator’s state transition diagram and commit algorithm

Figure 7 is the state transition diagram for the coordinator, including INITIAL, WAIT, COMMIT, and ABORT states. After a coordinator sends “prepare” message, it enters a WAIT state. Whether it enters COMMIT or ABORT state is determined by the votes it has received.

The commit algorithm of the coordinator can be described as follows.

BEGIN

\[
\text{Pa} := \{\text{all available sites}\};
\]

if \(\text{Pa} \neq \text{Null}\) {

\begin{align*}
\text{initiates a global transaction;} \\
\text{writes START to global log and flush;} \\
\text{send transaction request to the sites in Pa;} \\
\text{wait until receives commit request from AC;} \\
\text{send “PREPARE” request to DB-BEs;} \\
\text{while NOT receive all “YES” votes OR not timeout} \\
\text{if has received a “YES” vote then} \\
\text{if receives a “NO” vote then} \\
\text{send RETRY message to it}
\end{align*}

\[
\text{Py} := \{\text{participants voting “YES”}\}; \\
\text{Pn} := \{\text{participants voting “NO”}\}; \\
\text{if Py} = \text{Pa} \{ \text{// all sites vote “YES”} \\
\text{write COMMIT to global log and flush;} \\
\text{send global commit to the participants in Py;}
\}
\]

else {

\begin{align*}
\text{if Pn} = \text{Pa} \{ \text{// all sites vote “NO”} \\
\text{write ABORT to global log and flush;} \\
\text{send global abort to the participants in Pn;}
\}
\end{align*}

else {

\begin{align*}
\text{send global commit to the participants in Py;} \\
\text{identify the states of the sites in Pn as “OUT-OF-SYNC”;} \\
\text{identify the states of the sites not in Pn and Pa as “DOWN”}
\end{align*}

write DONE to global log and flush;

END

B. Participant’s state transition diagram and commit algorithm

Figure 8 is the state transition diagram for the participants, which includes INITIAL, WAIT, READY, and ABORT states. After a participant votes commit, it enters READY state, and if it votes “NO” or timeout, it enters WAIT state. The WAIT state can be changed to READY state if the participant retry successfully. Participants will take different actions as timeout in different states.

The commit algorithm of the participants can be described as follows.

BEGIN
RETRY_FLAG=0;
start transaction;
if receives PREPARE request {
RP: if executes successfully {
writes PREPARE COMMIT log and flush;
votes "YES" to coordinator;
waits till receive the global decision or timeout;
writes COMMIT to local log and flush;
commits transaction;
return
}
else {
if RETRY_FLAG=0 {
writes PREPARE ABORT log and flush;
votes "NO" to coordinator;
wait till receive the global decision or timeout;
if receive RETRY command {
RETRY_FLAG=1;
re-execute the transaction;
goto RP;
}
if receive global abort decision {
write ABORT to local log and flush;
abort transaction
}
}
}
END

V. RESYNCHRONIZATION METHOD BASED ON REQUEST LOGS

The purpose of resynchronization procedure is to get the missed updates before the failed system resumes services. If a replication transaction can execute successfully on the replicas over all sites, the update information of the transaction needn’t be resynchronization since nobody missed it. Therefore, only the information of committed write transactions which are missed by some sites need be kept.

Instead of the transaction logs, request logs are suggested for the middle-based replication architecture to handle resynchronization, i.e., only the requests of the write replication transactions missed by failed sites should be recorded, such that resynchronization can be accomplished by re-executing the missed requests in the resynchronization sites.

A. Resynchronization procedure based on request logs

The DB-FE is both a request forwarder and a transactions coordinator, so it knows about which requests have been forwarded, whether and when the requests have committed or aborted, and which requests are missed by the DB-BEs. Thus, whenever a DB-FE makes a global commit decision in the decision phase of R2PC, it will check whether the transaction is missed by any DB-BEs. If it is, the DB-FE will record the transaction request both into its memory and a stable storage.

The format of request logs should be as <USN, TR, SBE>. Of which, USN is the update sequence number, TR is the transaction request, SBE is the set of DB-BE missing the update. The requests belonging to a DB-BE can be linked together.

The resynchronization procedure based on request logs is shown as Figure 9:

1) After the DB-FE receives a heartbeat message from an “OUT-OF-SYNC” DB-BE, it sends a resynchronization command to the DB-BE;
2) The DB-BE starts a resynchronization procedure and send a ready message to the DB-FE;
3) The DB-FE fetches all missing request logs of the DB-BE, and sends them to it;
4) The DB-BE executes the missing request by the order recorded in the log;
5) After the DB-BE finishes resynchronization, it sends an end resynchronization message to the DB-FE;
6) The DB-FE identifies the state of the DB-BE as “SYNC”;
7) The DB-BE can resume normal service.

B. Resynchronization order of requests

The replay order of the missed requests is very important for guaranteeing the correctness of resynchronization in the failed sites. It is not necessary for the failed DB-BEs to replay the missed requests strictly by the same order as they have committed at the active servers, but it must ensure all conflicted requests can be replayed with the same order as they have committed.

For example, say 3 missed transactions Ta, Tb and Tc, and Ta is conflicted with Tc. Their commit order in an active DB-BE is <Ta, Tb, Tc>. The replay orders both of <Ta, Tb, Tc> and <Tb, Ta, Tc> are said to be right for the failed DB-BE, since all of them can ensure Ta is replayed before Tc.

In fact, a DB-FE doesn’t know the actual commit order of the transactions in the DB-BEs. But it knows the order of the global commits of all requests for it is the coordinator. If the replication two-phase locking (R2PC) is used to ensure the consistency of databases, the order of making global commit can be used as the replay order in the failed servers, since the conflicted transactions can’t vote YES at the same time.

C. An example

Figure 9 is an example of request logs, which keep the missed update records for both DB-BE2 (i.e., U001–U004) and DB-BE3 (i.e., U003 and U004). An update sequence number (USN) is the global commit sequence of a transaction request, which determines the replay order in the failed DB-BEs. Figure 10 shows the resynchronization orders of the missed requests for both DB-BE2 and DB-BE3.

Figure 9. An example of request logs
VI. PERFORMANCE TESTING

In this section, the performances both of R2PC and the resynchronization method using request logs are tested.

For R2PC, we first compare it with the standard 2PC protocol on update success ratio. Then, we will test the impact of transaction retry mechanism in R2PC.

For the resynchronization method, we first compare it with the method using transaction logs on resynchronization time and transaction throughput of the system.

The performance evaluation is performed on a simulation replication environment, including 3 ACs, 2 DB-FEs, 2 DB-BE Clusters, which form a local network.

A. Experimental results

Figure 11 shows the comparison results of update success ratio under various transaction generating ratio. For the commit condition is relaxed in R2PC, i.e., as long as one DB-BE votes “YES”, then the replication transaction can commit. But in 2PC, only all DB-BEs in a cluster vote “YES”, the replication transaction can commit. Therefore, the update success ratio of R2PC is higher than that of 2PC.

Figure 12 shows the impact of retry mechanism on the availability ratio of DB-BEs. In R2PC, if there is a DB-BE votes “YES”, the replication transaction can commit. But if the sites voting “NO” or without voting are denoted as an unavailability state: “OUT-OF-SYNC” or “DOWN”, which will reduce the availability of DB-BEs. In order to reduce the probability of unavailability due to temporal failures, R2PC allows the site voting “NO” re-executes the transaction if some one else has votes “YES”. Therefore, the retry mechanism can increase the availability of DB-BEs.

Figure 13 shows the comparison of resynchronization time using different methods. For each replication transaction in the experience only updates one or 2 data objects, so they are generally short transactions. That is, the times of resynchronization operations using both the transaction logs and the request logs are almost the same. But since all transactions logs are append-only logs, the fetcher of in the active server will take time to distinguish which logs are required for resynchronization, and the update process in the failed servers have to spend time to analyze which part in a log is the after image of the missing data. In the method using request logs, all logs belonging to one site are linked together, so it is easy for the DB-FEs to fetch the desired logs, and the format of the request logs is uniform. Therefore, it can save the resynchronization time.

Figure 14 shows the impact on system throughput using different logs. The resynchronization procedure using transaction logs may interrupt the normal transaction processing in the active server, since the transaction logs must be accessed exclusively by both the
transaction manager (log write) and the fetcher of the resynchronization logs (log read) in the active server. But the resynchronization method using request logs doesn’t interrupt normal transaction processing in the “SYNC” DB-BEs, so no any overhead needs be added for resynchronization processing in the “SYNC” DB-BEs.

Figure 14. Comparison of transaction throughout

VII. CONCLUSIONS

This paper focuses on the synchronous and resynchronization mechanism for the heterogeneous databases replication under a middleware-based architecture. The main advantages can be gained from the suggested methods as following:

- Because a replication transaction will be aborted only if all “SYNC” sites have voted “NO”, the availability of systems is greatly enhanced.
- It always provides a consistent view to clients, since only the replicas in the “SYNC” sites can be accessed by clients.
- The number of transactions who are aborted due to the temporal failures can be reduced largely through the transaction retry mechanism.
- The suggested resynchronization method doesn’t interrupt normal transaction processing in the “SYNC” DB-BEs.
- No any overhead is added for the “SYNC” DB-BEs during resynchronization procedure. In other words, the DB-BEs care nothing about resynchronization.
- The resynchronization procedure is independent of DB-BEs’ failures. That is, even if all other DB-BEs in the cluster have failed, the “OUT-OF-SYNC” DB-BEs can still become “SYNC” after finishing the resynchronization with a DB-FE.

ACKNOWLEDGMENT

This work is supported by National Natural Science Foundation of China (No.60863016), Natural Science Foundation of Jiangxi, China (No.2008GQS0019)

REFERENCES


Guoqiong Liao, born in 1969, received his Ph. D in Computer Software and Theory from Huazhong University of Science and Technology (HUST), Wuhan City, China in 2003. He worked in HUST for post-doctor research till 2005. Then, he worked as a Research Scientist in Siemens Corporate Technology in China. Now he is an associate Professor in the School of Information Technology, Jiangxi University of Finance and Economics, Nanchang, China. He has published a number of papers in the area on transaction processing and high availability on real-time and mobile databases. His research interests involved embedded, real-time and mobile databases, and more recently on RFID data management and middleware.

© 2012 ACADEMY PUBLISHER