Synchronization in an Embedded DBMS Environment

Sang-Wook Kim

Division of Information and Communications
Hanyang University, Korea

Abstract — *The embedded DBMS is a lightweight DBMS for effective management of quite small databases contained in tiny mobile devices. Synchronization is a core function of the embedded DBMS to preserve the consistency of data replicated in the server and client databases. This paper presents a framework for synchronization in embedded DBMS environment. We first address key issues for realizing synchronization, and then propose solutions to them obtained from our development. The main issues touched here are (1) classifying conflicts, (2) identifying changes in a client database, (3) detecting conflicts, and (4) resolving conflicts. The proposed framework would help reduce the trial-and-errors of embedded DBMS developers in implementing their synchronization server.*

Index Terms — conflict, data consistency, data replication, embedded DBMS, synchronization.

I. INTRODUCTION

THE advent of the post-PCs' era makes small-sized mobile devices such as personal digital assistants(PDA), mobile phones, hand-held PCs(HPC), and pocket PCs(PPC) ubiquitous in the world. The advances of such mobile devices combined with wireless Internet technology also enable people to enjoy a lot of useful information regardless of times and locations. The embedded database management system (embedded DBMS) is defined as a light-weight DBMS that targets effective management of small databases stored in such mobile devices[1-5].

A mobile device equipped with an embedded DBMS has a small capacity for storage, and thus is not suitable for managing a large database shared by a large number of users. Instead, in embedded DBMS environment, a server DBMS maintains such a large server database, and an embedded DBMS manages small data, a part of the server database, downloaded from the server DBMS. This data downloading makes the same data stored in both of the server and embedded DBMSs, thus incurs data replication.

This replicated data can be independently updated by both the server and embedded DBMSs. If such updates are not consistently applied into the databases maintained by both DBMSs, applications referring to the data go into the wrong way due to the problem of the data inconsistency. The synchronization is a core function of the embedded DBMS that is in charge of the consistency of the replicated data in both sides of DBMSs.

This paper discusses synchronization in embedded DBMS environment. Some commercial DBMS vendors provide embedded DBMSs that basically support synchronization[6][9][14][16]. However, for other developers, it is hard to refer to their technical solutions in detail since they just briefly describe their functions rather than solutions in a form of a brochure or a manual.

Embedded DBMS Team at 4DHomeNet Inc. and Data & Knowledge Engineering Lab. at Kangwon National University[1] have been working together under the support of the Ministry of Information and Communications in Korea to develop an embedded DBMS that targets mobile devices and information appliances. The purpose of this paper is to share key issues and our solutions obtained from developing our synchronization manager with other developers or scientists working in the similar area. This paper proposes a framework for synchronization in embedded DBMS environment. We first point out major issues to be solved for supporting synchronization, and then present our approaches as solutions to them.

This paper is organized as follows. As a background, Section II defines a typical synchronization model for embedded DBMS environment, and classifies types of conflicts. Section III suggests the design goals that our synchronization manager tries to meet. Section IV presents the data structures, synchronization steps, and conflict detection & resolution strategies of our synchronization manager. Finally, Section V summarizes and concludes this paper.

II. BACKGROUND

As a background of this research, this section briefly explains a synchronization model and types of conflicts in embedded DBMS environment.

A. Synchronization

A typical synchronization model in embedded DBMS environment is shown in Fig. 1. A number of clients are connected to a server via the wired or wireless network. A mainframe computer or a workstation would be considered as

[1] The author's prior work.

a server depending on the weight of applications, and small-sized mobile devices such as PDAs, HPCs, PPCs, and mobile phones would be the typical clients.

The server manages the server database shared by all the clients via the server DBMS. The client maintains the client database, which is a concern of the mobile device user and also a part of the server database, via the client DBMS embedded in the mobile device. The client database can be built by the mobile device user. However, it is typically built by downloading a part of a server database from the server. As a result, data replication occurs since the downloaded data exist in both sides of the server and client databases.

In general, the client is disconnected from the server after the data downloading. The client database can be changed by an embedded DBMS in the client after the disconnection. For data consistency, this change has to be applied to the server database when the client is connected to the server. For the same reason, the change occurred in the server database during the disconnection also has to be applied to the client database when they are connected to each other. The synchronization manager is a core component in embedded DBMS environment that preserves the consistency of such

Fig. 1. Embedded DBMS Environment.

replicated data.

B. Conflicts

Let us consider that more than one client download the same data from a server database. They may change the data in different ways, thereby making the data diverge. In this case, the changes in both client databases cannot be applied into the server database correctly. This problematic situation is called a conflict[12]. In this paper, we classify the conflict into three types: the insertion, deletion, and update conflicts.

The insertion conflict happens if two clients insert the same record R into their own client databases. Two versions for R in two client databases correspond to a same entity in a real-world, however, they may have different attribute values. One client, which first tries to synchronize the client database containing R with the server database, successfully reflects the insertion of R on the server database. However, the other client, which next tries to perform synchronization, finds that R already exists in the server database, thus encounters the problematic situation. If it reflects its insertion of R by force without any concern, two Rs, which have the same value for the key but different values for other attributes, coexist in the server database.

The deletion conflict occurs when one client tries to delete record R from its client database while the other client tries to modify R in its own client database. As in the previous case, one client, which first tries to perform synchronization to delete R, successfully reflects its own changes. However, the other client, which next tries to perform synchronization to modify R, recognizes that R does not exist in the server database.

The update conflict happens when two different clients try to change the same record R in their own ways. In this case, one client, which first tries to perform synchronization, successfully reflects the update on R in the server database. However, the other client, which next tries to perform synchronization, is not able to update R in the server database since R in the server database is not the version the client has updated in its client database.

When synchronization between a server and a client starts, the synchronization manager detects what kind of conflicts happen, and then resolves each conflict in a way that is pre-defined by an application.

The synchronization methods employed in commercial embedded DBMSs are classified into two categories: one is to use the time-stamp value[11][16], and the other is to use the

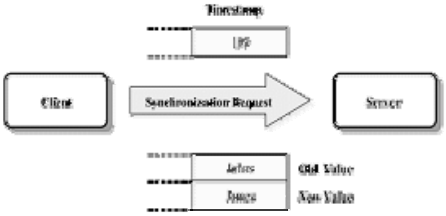

Fig. 2. The TS- and OV-Methods.

old value before the change.[2] In this paper, we simply call them the TS-method and the OV-method by taking their initials. Fig. 2 illustrates their basic concepts.

The TS-method adds a time-stamp field to each record existing both in the server and clients to be synchronized. The time-stamp has a value representing the latest time when the record was changed in the server. The client downloads the time-stamp value together with other attribute values within each record from the server. The embedded DBMS in the client keeps the time-stamp value, and detects conflicts by referring to it during synchronization.

The OV-method employs the old values of attributes in each record downloaded from the server. The embedded DBMS in the client keeps the old values independently of their new values even when it changes them, and detects the

[2] In most references including [11] and [16], they just mentioned the update conflict without the insertion and deletion conflicts.

conflict by referring to them during synchronization.

III. DESIGN GOALS

In this section, we present the design goals that we tried to satisfy in developing our synchronization manager.

(1) To minimize the storage overhead in a client for synchronization: Inherently, the mobile devices such as a mobile phone, a PDA, and a HPC have small main memory and/or disk. Thus, it is impractical if a method requires additional storage too much for synchronization.

(2) To minimize the amount of information transferred during synchronization between a server and a client via wired or wireless network: A large volume of such information delays synchronization, and also makes the possibility of a synchronization failure due to network problems much higher. So, it is very important to minimize such information required in synchronization.

(3) To make our synchronization manager independent of any server DBMSs: If a synchronization manager is dependent on a server DBMS, it is not applicable to others. So, in this case, we have to develop a different synchronization manager for each server DBMS. Thus, the development of a server-DBMS-independent synchronization manager is very important for portability.

(4) To provide various strategies for resolving conflicts: The straightforward one is to ignore all the changes by a client and to reflect all the changes by a server on the client database when detecting a conflict. However, some applications want the changes by a client with a high priority to be reflected on a server database even if a conflict occurs. Therefore, it is desirable that the synchronization manager provides various strategies for resolving conflicts, and allows applications to choose some of them by their preferences.

IV. PROPOSED APPROACH

This section presents our approach for synchronization that meets the design goals in Section III. Section IV-A presents its main data structures, and Section IV-B details the synchronization steps. Section IV-C suggests the way to detect conflicts, and finally Section IV-D presents the strategies to resolve conflicts.

A. Data Structures

The OV-method has a storage overhead larger than the TS-method. The TS-method adds only one field of the time-stamp to each record to be synchronized while the OV-method has to keep old and new values against every attribute of a record. Thus, we decided to follow the TS-method in order to detect conflicts by considering the design goal (1) on the storage overhead [3].

[3] We note that references [6][9][14][16] just briefly describe the concept and functions of synchronization, and do not provide a concrete description as proposed in this paper.

In our approach, every record in a server database has an additional field of the time-stamp. The time-stamp indicates the latest time of a record to be changed in a server database. Such changes come from the clients' synchronization request or a server's change request. Each time-stamp value has a unique meaning on its own record. That is, even if two records have an identical value for their time-stamps, this does not imply that they change at the same time.

TABLE I
CHANGES OF TIME-STAMP VALUES IN A SERVER DATABASE

Operation	Insertion	Update	Deletion
Time-stamp	1	Time-stamp+1	Record Deletion

When a new record is inserted into a server database, its time-stamp is set to 1, the minimum value. When a record in a server database is changed, its time-stamp increases by one. This implies that the record changes into a new version. When a record in a server database is required to delete, the record is removed at the time without the changing of its time-stamp. These are summarized in Table I.

In our implementation, we use the INTEGER as a type of a time-stamp. The INTEGER occupies space smaller than the time-stamp type supported by commercial server DBMSs. Also, we can simply compare and change the values of the INTEGER type without special functions. This allows us to detect changes and conflicts quite efficiently. The INTEGER type covers a range of 1 to MAX(in case of the 4-byte INTEGER, it is 2^{31}-1). So, it can support the sufficient number of changes. Even if it overflows due to a lot of changes, we can simply handle this situation by resuming its increase from 1. Thus, the INTEGER type does not restrict the number of changes in a server database.

In a client database, every record contains two additional fields: one is the time-stamp field for detecting conflicts, and the other is the status field for keeping track of the changes in the client.

The time-stamp of a record in a client database does not change even if the record changes due to the insertion, deletion, or update. This time-stamp represents a kind of a version number of the corresponding record in a server database. Its current value corresponds to the version fetched by the client for the server at the latest synchronization. The synchronization manager compares this value with that of the corresponding record in the server database so as to detect conflicts during the next synchronization.

During synchronization, only the records, which have been changed in the client database, cause conflicts. In the proposed approach, we employ the strategy to transfer only the records updated in the client into the server in order to minimize the communication overhead in synchronization. This strategy enables the proposed approach to meet the design goal (2). Therefore, the embedded DBMS has to

identify such records updated after the most-recent synchronization.

For this, we use the status field in each record to keep track of the changes in our approach. The status field is a CHAR type of one byte, and it represents the type of the last operation performed on the corresponding record in a client database. As shown in Table II, the status field has "I" for an insertion, 'U' for an update, 'D' for a deletion, and 'N' for none. During synchronization, the synchronization manager extracts only those records whose value in the status field is not 'N' in order to detect conflicts. This process is done simply by performing the SQL statement supplied by a client DBMS.

TABLE II
CHANGES OF STATUS FIELD VALUES IN A CLIENT DATABASE

Operation	Insertion	Update	Deletion	None
Status Field	I	U	D	N

Let us consider a situation where a record is inserted and then changed in a client database after the recent synchronization. In this case, its status field is kept as 'I' rather than 'U'. This is because this record has to be regarded as a newly-inserted one in the server database in the next synchronization. Also, if a record is requested to delete in a client database, it is not actually deleted at that moment. Instead, its value of the status field is changed into 'D'. The reason for this is, we have to preserve this record in a client database in order to eliminate it from a server database during the next synchronization. Since this kind of records are not allowed to show as a query result in a client, such records whose value in the status field is 'D' have to be discarded during query processing. Finally, let's consider a situation where a record is inserted and then deleted in a client database after the recent synchronization. In this case, the record is immediately deleted from the client database without handling of the status field. This is because the record does not exist in a server database, so it is not necessary to keep the information on the record until the next synchronization.

B. Synchronization Steps

Fig. 3 sketches the overall process for synchronization. The synchronization proceeds by performing a transaction that runs both in the client and server DBMSs. This means that the synchronization manager performs the standard SQL statements supported by the client and server DBMSs when accessing the records in the client and server databases for synchronization. By this, our synchronization manager is independent of any specific server DBMS, thereby meeting the design goal (3) of the platform independence. In this section, we discuss the detailed synchronization steps of the proposed approach:

(1) If a record changes in a client database, its status field is set as in Table II. In a client database, such changes would repeatedly occur.

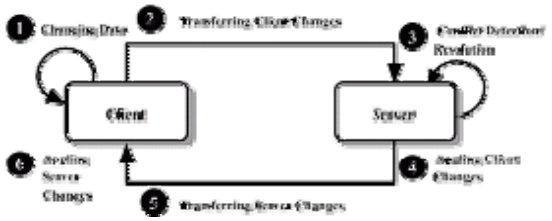

Fig. 3. Synchronization Process

(2) The client requests synchronization of the server. The client first identifies the records to be reflected in a server database from its own database. These records have changed since the latest synchronization, and their status field values are not 'N'. For synchronization, the client transfers two kinds of lists into the server.

One is the client-change-records-list that has multiple entries, each of which corresponds to the record changed after the latest synchronization. The entry has all the attribute values and a time-stamp field value for that record. This client-change-records-list is used to apply the changes occurred in a client into the server database. We note that it does not include the status field in each entry. This is to minimize the communication overhead between the client and the server in synchronization, thereby satisfying the design goal (2). In the client database, the status field is necessary to identify four types of changes of the insertion, deletion, update, and none. However, we just need to identify just three types of changes of the insertion, deletion, and update since the record un-changed in the client database does not exist in the client-change-records-list. The time-stamp field suffices for identifying three types of changes by setting it to 0 for an insertion, to a negative value for a deletion, and to a positive value for an update.

The other is the client-entire-records-summary-list that has entries, each of which corresponds to each record in a client database, and consists of the <primary key, time-stamp field> pair. This list is useful in examining the new changes that were made in a server database after the recent synchronization, and thus has to be applied to a client database at this synchronization. We present the way to detect new changes done in a server database by using this entire-records-summary-list in step (4) in more detail.

(3) The server examines and detects conflicts by using the client-change-records-list transferred from the client. If it detects a conflict, it resolves the conflict in the way defined by an application. We will elaborate more the detection and resolution of conflicts in Sections IV-C and IV-D.

(4) The server applies the changes occurred in the client database into the server database by referring to the client-change-records-list. That is, by examining the time-stamp field of each record in the client-change-records-list, it performs an update for the time-stamp of a positive value, a deletion for the time-stamp of a negative value, and an insertion for the time-stamp of 0 on the server database.

Finally, it re-adjusts the time-stamps of the records changed in the server database by following the rules in Table I.

(5) The server builds the server-change-records-list as in the below by referring to the client-entire-records-summary-list and transfers it to the client. This list contains such records changed in a server database after the client's latest synchronization. This list represents the new changes to be applied into the client database. The structure of the server-change-records-list is identical to that of the client-change-records-list.

Record update: For each entry of <primary key PK, time-stamp field TS> in the client-entire-records-summary-list, the server examines whether the record R with PK as the primary key in the server database has the same value as TS. The different value implies that there have been some updates on the corresponding record in the server database since the client's latest synchronization. In this case, the server adds the updated record into the server-change-records-list. The time-stamp of this record in the list has the same positive value as that of the original record in the server database.

Record insertion: From the server database, the server finds such records whose primary keys do not exist in the client-entire-records-summary-list. The existence of such records implies that there have been new insertions after the client's latest synchronization. Also, the server adds each inserted record into the server-change-records-list, and sets its time-stamp in the list to a negative value whose absolute value is the same as that of its corresponding record in the server database. Note that such records in the client-change-records-list have 0 as their time-stamp. In case of the server-change-records-list, we need to know which version of the record in the server database gets into the client database for conflict detection.

Record deletion: For each entry of <primary key PK, time-stamp field TS> in the client-entire-records-summary-list, the server examines whether the record R with PK as the primary key exists in a server database. If there is no such a record, it means that the record has been deleted from the server database since the client's latest synchronization. In this case, the server adds the record into the server-change-records-list, and sets its time-stamp in the list to 0. We remember that such a record in the client-change-records-list has the same value as that of the client database. The reason for this is that the record just has to be removed from the client database without any conflict detection.

(6) The client applies the changes occurred in a server database into a client database by referring to the server-change-records-list. That is, by examining the time-stamp field of each record in the server-change-records-list, it performs an update for the time-stamp of a positive value, a deletion for the time-stamp of 0, and an insertion for the time-stamp of a negative value in the server database. It also adjusts the time-stamp of the records inserted at this time to return to the positive one.

Finally, synchronization ends after deleting all the records whose status field is 'D' from the client database. These records are not necessary to be kept in the client database anymore after terminating successful synchronization.

C. Conflict Detection

The synchronization request by a client makes the server examine whether conflicts occur or not, by referring to the client-change-records-list transferred by the client.

The first one is the detection of insertion conflicts. If the record in the client-change-records-list to be inserted has the same primary key as the record in a server database, this insertion incurs an integrity constraint violation in the server DBMS. Thus, the server database cannot accept this insertion. The synchronization manager uses the primary key of records for detecting insertion conflicts. That is, it detects insertion conflicts by examining if a record of the time-stamp of 0 in the client-change-records-list has the same primary key as the record in a server database.

The second one is the detection of update conflicts classified into two types. The first type is the case that a client tries to update the record that has already been deleted and thus does not exist in the server database. The second type is as follows: (1) Two clients A and B downloaded the same version of record R; (2) Client A updated R in its client database, then reflected the update on the server database through its synchronization; (3) Client B has updated R in its client database, then tries to reflect the update on the server database during synchronization.

In the former case, we can detect the conflict by examining if the record of the positive time-stamp in the client-change-records-list exists in a server database. If the record does not exist in the server database, this implies that the server or other client has already deleted it. In the latter case, we examine the records of the positive time-stamp in the client-change-records-list as follows: (1) For record R with the primary key PK, we first find its corresponding record R' with the same PK in the server database; (2) Then, we compare the time-stamp of R with that of R'. If the two time-stamps are different, there has been a update on R' after the client's previous synchronization, thus we regard R' as having been changed into a different version. Therefore, we can detect two kinds of update conflicts safely.

The third one is the detection of deletion conflicts. A typical situation is as follows: (1) Clients A and B downloaded the same version of record R in their client databases; (2) Client A updated R in its client database, and reflected it on the server database via synchronization; (3) After that, client B deleted R from its client database and tries to apply it into the server database during synchronization. For detecting this conflict, we first look for record R whose time-stamp has a negative value in the client-change-records-list, find R's corresponding record R' in the server database where the primary keys of R and R' are the same, and then examine if the absolute values of the time-stamps in R and R' are identical. If they are different, it means that R' in the server database has already changed into a different version. So, we regard the

deletion conflict as having occurred.

D. Conflict Resolution

In case a client detects conflicts during synchronization, a unique decision to guarantee perfect consistency of the server database is to just give up the changes causing the conflicts to be applied to the database. Otherwise, such changes incurring conflicts would make the server database inconsistent. Therefore, our approach employs a give-up strategy as the basic one for resolving conflicts.

However, the major problem in the give-up strategy is to abandon all the changes performed by a user in the client. In this case, the user would complain that he/she has to do a tedious job again for applying the changes into the server database after synchronization. Some applications think much of users' satisfaction rather than keeping complete consistency with the database.

So, other than the give-up strategy, we employ another strategy to allow users to decide whether they accept or abandon the changes when conflicts occur. Of course, the user (or client) should have the authority appropriate for such decisions. For this, we categorize users into multiple classes according to their authority. This is to support various strategies for resolving conflicts according to users' preference, thereby making our approach meet the design goal (4).

V. CONCLUSIONS

Synchronization is a core function in embedded DBMS environment to sustain the consistency of replicated data in the client and server databases. Some existing commercial embedded DBMSs have been supporting synchronization [6][9][14][16], however, they do not provide their technical solutions in detail as a form of research papers. Therefore, it is not easy for developers to refer to their experiences and techniques.

The Data & Knowledge Engineering Lab. at Kangwon National University and 4DHomeNet Inc. have been working together for developing an embedded DBMS since 2001. In this paper, we have presented a framework for synchronization in embedded DBMS environment, and also discussed its related issues and solutions. Major issues touched are (1) classifying conflicts, (2) identifying changes in a client database, (3) detecting conflicts, and (4) resolving conflicts. The purpose of this paper is to share our experiences obtained in developing a synchronization manager with other embedded DBMS developers. Our contributions would help reduce their trial-and-errors significantly.

ACKNOWLEDGMENT

This work was partially supported by Seoul R&BD Program under the title “Implementation of RBDMS on Flash”. We would like to thank Se-Bong Oh, Woo-Seok Jang, Gray Noh, Yeong-Ho Kang, Byung-Dae Jung, and Sung-Yong Son who have actively participated in developing our embedded DBMS. Also, we would like to thank Jung-Hee Seo, Suk-Yeon Hwang, Grace(Joo-Young) Kim, and Joo-Sung Kim for their encouragement and support.

REFERENCES

[1] Sang-Yun Lee et al., "Synchronizing Techniques for Mobile DBMSs," Database Research, Vol. 17, No. 3, pp. 29-41, 2001.
[2] Yun-Seok Choi, "Oracle9i Lite: A Light-Weight DBMS for Mobile Environment," Database Research, Vol. 17, No. 3, pp. 103-107, 2001.
[3] Do-Yeon Kim, “"IBM Informix Cloudscape for Data Management in the Post-PC Era," Database Research, Vol. 17, No. 3, pp. 109-114, 2001.
[4] Michael A. Olson, "Selecting and Implementing an Embedded Database System," IEEE Computer, Vol. 33, No. 9, pp 27-34, September 2000.
[5] Sixto Ortiz, Jr., "Embedded Databases Come out of Hiding," IEEE Computer Magazine, Vol. 33, No. 3, pp 16-19, March 2000.
[6] Oracle, Oracle 8i Introduction, Oracle's White Paper.
[7] Oracle, Oracle Lite User's Guide.
[8] Oracle, Oracle 8i Replication.
[9] IBM, DB2 Solutions for Mobile Computing, IBM's White Paper.
[10] IBM, DB2 EveryPlace Brochure.
[11] IBM, IBM DB2 Replication Guide and Reference.
[12] IBM, IBM DB2 Sync Server Administration Guide.
[13] IBM, Have Your Database Everyplace, IBM's White Paper.
[14] Informix, Informix CloudSync, Informix's White Paper.
[15] Informix, Informix Cloudscape, Informix's White Paper.
[16] Sybase, Synchronization Technologies for Mobile and Embedded Computing, A White Paper from Sybase, Inc.
[17] Sybase, Adaptive Server Anywhere Getting Started.
[18] Sybase, Introducing SQL Anywhere Studio.