DISTRIBUTED DISCONNECTED DATABASES

Leslie Klieb
P. O. Box 26097
New Orleans, LA 70186-6097

KEYWORDS: mobile computing, databases, optimistic peer-to peer synchronization, conflict resolution

ABSTRACT

Given are two databases (consisting of one or more tables) with similar structure in different places that are asynchronously updated. We consider how to synchronize them so that they are 'more or less the same' for 'most of the time.' Topics discussed are: how to synchronize them, what 'clashes' are possible, and ensuring consistency, if not correctness. As the systems are physically separated, record locking cannot be used to ensure consistency. The systems form a distributed database system but the classic lock and commit protocol is not useful here. This proposal does not require hooks into the programs used to manipulate the tables or the operating system. However, it requires the use of reference files, effectively doubling the storage requirements for each table.

1. INTRODUCTION

The dream of mobile group computing is still a dream. Who hasn't wished to use a computer as you would a telephone booth to get the latest news from whomever you want? You walk up to any machine in front of you, identify yourself and you edit and use your computer files without considering where they are physically located. Your custom written programs are available. You see the changes that coworkers have made in your files in the same way they tell you about current events on the phone. Even if you could not use an arbitrary computer, your own laptop computer could be coupled to others by cellular modems. However, we will have to live for a while with the reality of much more limited resources, in which notebook computers allow us to physically move the computer but in which the interacting members of a group are not connected most of the time.

This paper describes how the dream of using any computer for whatever you want to do and still get your coworkers' changes was partially implemented. We developed an information system (customer list, inventory, accounting) for a small company with traveling sales people who make frequent nationwide trips. They take their laptops on the road and update their databases, finding themselves stuck with similar files on different computers. It is necessary that they can update files at all times. Customers might call any of the sales people involved, with, for instance, a change of address, and the information is updated on the computer at hand. Regularly the new information from different sites needs to be merged. This problem is bound to occur more with database files than with text files. It is also easier to solve with database files.

This paper considers how to set up a system for merging database records and how good such a system can be. Obviously, without record locking, clashes between sites are possible; we consider these 'not too harmful' when: a) every clash is detected and logged; and b) inconsistencies are removed by the system.

The theory and the implementation chosen here is independent of the type of application, but biased towards a database system with a dBase (.DBF) type of file format. The implementation was in FoxPro 2.5 under MS-DOS.

2. TRACKING CHANGES TO DATABASE FILES

The dBase file format is very popular on PCs. dBase-like tables and other PC databases have the following traits:

1. Fields in tables are typed. Common data types provided are: strings of a certain length, numbers, truth values, and dates. Usually records are fixed record type with some allowance for variable length strings (memo fields.)

2. There is no interpretation or meaning attached to the different type of fields.

3. The database engine does not require a primary key. However, good programming practice dictates that every table has a key expression which is different for every record. This key expression can be one field, a combination of fields, or a function of a combination of fields.

4. One to one, and one to many relationships between records in two tables can be set up.

5. Uniqueness of records, database normalization and referential integrity are not enforced by the database engine but have to be programmed into the application.

There are several ways to see what has changed when data are recorded or updated. One is to put a hook into the program. Another way is to provide hooks into the operating system at the file write level: under DOS, to provide a TSR; under Windows or UNIX, to have a link with another program. Ref.

[3] uses such an approach for Novell. Another is to use a reference file to see what has changed.

The 'hook' approach in a program has the disadvantage that it requires separate programming for each program. Also, it cannot cope with updates to a table outside the program, as is easily possible with most PC systems. Hooks into the operating system, like a TSR under DOS, are difficult to program. Also, the TSR must be operational all the time, something not enforceable under DOS. An important advantage of making changes to the operating system can be that often various kinds of files can be reconciled, even directory entries [2],[5],[6]. However, if the system does not know about the internal structure of a file and its semantics, it is difficult to handle simultaneous or conflicting updates at two sides.

The approach here is to limit the problem to the possibilities of using a reference file to track updates to database files. Requirements of the system were: the number of updates is relatively small compared to the size of the databases; all information had to flow to all interacting systems; and the number of interacting system is reasonably small. The overall philosophy that emerged is close to the XEROX Palo Alto Bayou project [7]. Our terminology follows Ref. [1]

Consider a table with two fields, a reference number that serves as primary key and a name.

We can say that a record is **changed** if the reference number stays the same but the name is different in the table and in the reference file. If a record with a new reference number is found in the table but not in the reference file, then this record is **new**. If a reference number disappeared in the table but was in the reference table, it was **deleted**. The following 'theorems' are basic observations that follow easily by mapping on sets.

Theorem 1: Added, changed and deleted records are disjunct.

Theorem 2: The additions, changes and deletions, when applied to the reference table, yield the 'new' table.

Corollary 1: If there is another unique key for both tables, the changes, additions and deletions found by using this key yield the same 'new' table when applied to the reference file as using the first unique key.

Corollary 2: Different keys can yield different sets for changes, additions and deletions. An example: Assume the name is changed. Using the reference number as a key, the record is changed. Using the name as a key, the record with the old name and reference number is deleted and a record with a new name and the old reference number is added.

3. SYNCHRONIZING FILES: NO CLASHES

Consider two machines where both hold an identical table and a reference file. Users are free to make *any* change they like to the tables, but not to the reference files. However, the key values are required to stay unique.

Changes without clashes: Changes can be determined by comparing the reference file and the current table with respect to a chosen key expression. We can send a file with a small table containing the key values and the new value of the field, and an indication which field has to be updated. On the other side for every record in the transmitted table we can find the record that has the same key value, and replace the contents of the changed field with the new one. We do this both in the table and in the reference file. On the original side, we update the record in the reference table as well.

Additions without clashes: The key values of the added records for both parties are all different. So additions can be applied. All fields are sent with the value of the key. Additions are again added to both the table on the other site and both reference files.

Deletions: A small file with the value of the key of the deleted records can be sent and the records with these key values can then be deleted at the other side. This process is carried out, again, both in the table and in the associated reference files.

When the changes are also applied to the reference file, the files and the reference files on both machines are identical (in the database sense: up to physical ordering) after this patching process. The order of the operations is irrelevant. The additional storage requirement of the reference files is offset by the smaller file transfers.

4. SYNCHRONIZING FILES: CLASHES

It is possible that both sides make a change to the same record. If they make a different change, there is a clash. Clashes are annoying but unavoidable without record locking, semaphores or a similar construct.

While physically connected distributed databases have the potential to prevent clashes, they are often not better in dealing with the problems that underlie a clash. Assume a customer notifies two persons with access to the same database of a change of address. The fact that the record is locked during an update, so that the information is always consistent after each update, still does not mean that information in the database is correct. If both persons enter a slightly different address then it is only a matter of chance that the correct one is entered last.

Frequent clashes can have many causes. For many PC type of applications the odds for clashes are low [5],[6],[7].

The types of interactions that are possible when a field gets updated are:

Side 2 \ Side 1	Change	Delete	Add
Change	Depends †	Y	—
Delete	Y	N	—
Add	—	—	Depends ‡

† Okay if changed to same field value; otherwise clash
‡ Okay if same field value added; otherwise clash

Table I. Clashes

Four pairings out of nine are mathematically impossible. An example, a change by Side 1 means that a record with this key value was already present in the tables. So Side 2 cannot add a record without violating the uniqueness of the key values. If the operation is possible, it is indicated if a clash is possible. Two delete operations do not lead to a clash because both sides agree on what they want with the record.

Consider now two systems connected by phone, null modem cable, shared memory, disks, or other means. Additions are transferred from Side 1 to Side 2. It is easy to check, while adding record by record to the table on Side 2, that a record with the same key value has already been added on Side 2 and to log if there is a difference in field contents. Consistency, however, requires a choice. This is done by designating one side as 'winner' and one as 'loser.' This can either be user configured, or some asymmetry of the system (like who initiates the connection) can be used. The reader should verify that it makes no difference who first attempts an update. For instance, if the losing side first adds a record to the winning side, it will be rejected. Then the losing side will receive the record from the winning side, which it has to accept.

For changes a list of the differences with the key values of the records is transferred. While looking up these records in the table on the other side, it is easy to spot if changes are contradictory. If they are, one side has to win again to ensure consistency. It should be decided beforehand which side wins. Either side can try first.

Deleting the same record on both sides does not form a clash. However, it means that the key value of the record is found in the reference table, but not in the master table when trying to delete the record at the other side. Therefore the program must notice that this is happening.

Finally, consider the case of Side 1 deleting a record that was changed at Side 2. We transfer the list of deletions to Side 2. If we discover at Side 2 that the same record was changed at 2, then for consistency we either have to add the changed record back at Side 1, or delete the changed record at Side 2. As the system is not able to figure out who is right anyway, and the clash can be easily flagged and logged, it is much easier to delete the record. In that way a 'carry back' is avoided. Again, it should be checked that the order of all operations leads to the same final result, a deleted record.

The following table summarizes all allowed operations on records:

Side 1	Side 2	Outcome
Add	Add	Predefined side wins; added
Delete	Delete	No clash; deleted
Change	Change	Predefined side wins; changed
Change	Delete	Side 2 wins; deleted
Delete	Change	Side 1 wins; deleted

Table II: Operations on records

Each side logs the clashes it detects and writes it to a text file. This file is sent to the other site after synchronization. In practice a complete log of all changes, not only clashes, is sent over to allow review.

Summarizing, this scheme has the following advantages even in the presence of clashes:

- Updates can be made in any order. Either side can start, and additions, deletions and changes can be applied in any order.

- No transfer back of information is necessary. Changes, additions and deletions are transferred unidirectionally.

- Decisions who is right when clashes are detected do not have to be made during the synchronization of the information. Corrections can be made afterwards on either side with the help of the log file. They will propagate at the next synchronization. Information after synchronization is always consistent.

5. OTHER TYPE OF OPERATIONS ON FIELDS

Unfortunately, just replacement of field contents, additions, and deletions are not always enough for a correct update. Consider an inventory database. One field of a table shows the type of merchandise. Another field, numeric, holds the number of articles of this item in stock at a common central warehouse.

The company has 10 widgets in stock. Say Salesperson 1 sells 3 widgets, and Salesperson 2 sells 4 widgets. Both update their own databases during the day. At the end of the day Salesperson 1 thinks that there are 7 widgets, Salesperson 2 thinks that there are 6. When the procedures of the previous chapter are carried out, one of them 'wins.' Whoever wins in this clash is still wrong.

The problem here is that the number of items in stock has a different semantics from, say, an age or a street number. It is necessary to apply <both> changes to the inventory database. Salesperson 1 should instruct the other side to deduct 3 widgets from its end of day total of 6 widgets, Salesperson 2 should instruct Salesperson 1 to deduct 4 widgets from its current total of 7. Then they are both correct and consistent, with 3 widgets left. This problem cannot be detected by examining the database structure alone. Street number, age, and number of items in stock are all numerical fields.

This shows that the most general way of updating a remote database is to track the changes, additions, and deletions, <and> to define an operation on the field which gets updated.

Theorem 3: Just a reference file is not sufficient for updating a remote database when all possible operations on database fields are considered.

Proof: Call the input i, the original contents of the record x, the new contents y. Then, using a modifier *1,2* to indicate the sides, and a function f for the operation:

$y1=f(x1,i1)$.

The reference file contains x, the new file y, and the function f is given. In order to apply the update to the other side, we need to solve this equation for i, transfer that value to the other side and calculate $y2=f(x2,i1)$. However, this inversion is

impossible in general. (Example: consider the function $y=x$ for $x<10$, $y=x+i$ for $x>=10$).

In the cases discussed above, the inversion of f to solve for i is possible. When new input i replaces the old x on an update, the function f is given by $y=i$, which solves trivially for i in terms of y; with the widget case the function is $y=x+i$, which is also easy to solve for i.

Sufficient for consistent updates are the conditions that f is such that i can be solved; and the function f denotes a commutative operation. Consider a non-commutative operation, like adding a string to the tail of another string. After the update the two strings are inconsistent, as the following table shows, because the order in which they are added is crucial here.

	Side 1	Side 2
new file	yyxx	yyzz
reference	yy	yy
file input (i)	xx	zz
after update	yyxxzz	yyzzxx

Table III: A non-commutative update

The whole problem can usually be avoided by a different design. In the widget example both parties could as well have just their sales recorded, and the total number of widgets sold and the current stock could be calculated by the system on the fly. Note that this is a much more natural formulation, because the central warehouse has to be notified that it has to ship the orders anyway.

Potentially troublesome is also the assignment of reference numbers (key values) to label records, like time stamps, invoice numbers, *et cetera.* This can easily lead to duplicate reference numbers and cause clashes. The reference numbers can be made unique over all sites by using different ranges of numbers on every machine or by pre- or post-fixing them with a site identification. Another possibility is to make each site independent and consolidate only figures that do not contain identification, like total daily sales, and not the separate invoices, but that defeats the purpose of synchronization.

6. MORE TABLES PER DATABASE; REFERENTIAL INTEGRITY

Up to now only single tables were considered. Usually tables in relational databases have parent-child relationships. Often there are no special problems, as long as both parent and children have unique primary keys in addition to the foreign keys relating the tables.

It is, of course, mandatory to update all the tables of a database simultaneously. Otherwise, inconsistencies arise immediately. This requirement can be much more subtle than one might think. Consider the next example:

An accounting system contains invoices and payments. Once a month all customers get a review with the open balance ('posting'). An invoice should only be corrected or changed until this posting process.

After a data synchronization, let one machine carry out the posting process. Meanwhile the amount on the invoice is changed at the other machine. After synchronization the two machines are technically consistent again, for the invoice amount as well as the posting flag. The customer, however, receives a statement showing the original amount before the correction was made on the second machine.

Formally, the paper printout for the customer is also a record in another remote disconnected database, which is not updated when the invoice is changed, unless we 'resynchronize' again by mailing another corrected statement.

A parent record without a child record can be deleted on one side and a child record added at the other side. This can be prevented by adding a flag indicating there is a child record. Then a clash ensues, so that corrections are possible.

7. MORE THAN TWO CONNECTED SYSTEMS

More than two systems can be synchronized *as long as every link has its own reference file on the two machines the link connects.* This provides the means to track every change with respect to every connected machine. All changes on all machines will eventually propagate to all other machines (*cf.*[2].)

8. PRACTICAL EXPERIENCES IN IMPLEMENTATION

The ideas of data synchronization and table merging have been implemented in a FoxPro program. A user-definable 'data dictionary' holds the following information for every table and for both sides:

1. name/location of the table.

2. name/location of the reference file.

3. key expression (this can be a general FoxPro function of the fields in the database.)

4. fields to update (or * for all fields, together with a list of fields not to update - fields referenced in keys don't have to be updated for changes.)

One side only stores this table, in order to avoid conflicts. It has been arbitrarily decided that the side with this data dictionary is also the side that wins in conflict situations (this is supposedly the side with the user who knows best about what is going on.)

At this moment the system is used to share a customer list, a list of stock numbers and accounting information between two desktops and two laptops.

1. 'Tinkering' with data at the FoxPro dot prompt is nearly unavoidable in a running production system. It is next to impossible in FoxPro (or dBase, or similar languages) to write programs that guard all business rules and integrity rules. A system that generates deltas from inside, and forces to do every update via the system [8], makes itself very vulnerable. The approach with reference files has proven essential. This

assessment will probably change when rules can be maintained at the database engine level.

2. The present implementation is not very fast and does not give feedback how processing on the other side is going. This is a problem if the other side is not visible for the operator, e.g., when using the system over the telephone. Remote control technology with drive redirection would it make possible to watch progress on both sides at the same time. However, this has not been implemented yet. Therefore synchronization is carried out via a null-modem cable when the sales people meet.

3. Most clashes occur because of fuzzy boundaries in responsibilities. While it is easy to condemn this from a formal point of view, it often makes sense for the people who do it. They consider it often as 'unavoidable' - and they are willing to live with the consequences. A concrete example: new prospects are entered on a laptop computer at trade shows. When this prospect becomes an actual purchaser, his order is entered on a desktop at the main office, while the laptop computer with the trade show information can still be out on the road. Rarely is the entered information consistent on both machines; also rarely is one side completely correct and the other completely wrong. The frequency of this kind of clash depends very much on how often the information is synchronized. To give an idea: the file most susceptible for clashes is the customer/prospect list with about 3000 names. Usually with every synchronization there are a few hundred changes in this file. Between one and ten of these are clashes.

4. Looking through log files and discussing clashes has not been an annoying chore for the people using this system. Clashes lead often to useful discussions because they point to problem cases. It can also mean that more than one salesperson had contact with the customer, which usually happens only in the more intricate or difficult situations.

5. A star-topology saves disk space for the reference files. We preferred the convenience of being able to interact with every other system.

9. COMPARISON WITH OTHER SYSTEMS

Data replication and synchronization is now built into a number of systems. Lotus Notes [4] does not employ a relational database structure and does not allow database merging at the field level. dbSynchro[8] has one master site and a number of slave sites; it replaces some database operations built into the FoxPro language with its own constructs in order to maintain a log. Therefore the common data entry via a 'browse' of a table is not possible anymore. As pointed out above I think it is important that tinkering with the data in the tables is directly possible.

Data replication in larger systems has usually been in one direction only. Two-way data merging as described here makes a whole different class of applications possible.

The Bayou project [7] is also a peer to peer optimistic data exchange mechanism for databases. It uses hooks into the program to generate the deltas.

10. SUMMARY

We described a system that allows two or more same databases on different computers to interact with each other so that changes, additions and deletions made on all computers are transferred to the other systems and merged. Clashes can occur but can be flagged and logged. The system is used in one real world situation where data are entered on four computers and regularly compared. At least in this situation users can live quite well with the limitations and find it very natural to 'do as they please,' to use an arbitrary computer to enter the information they need. Many business situations of the 'traveling salesman' variety can be expected to profit from similar systems.

Acknowledgments: I want to thank Ad Aerts and Cynthia Alden for helpful discussions. The Comet communication library from Compusolve, Albrightsville, PA, has been used for the serial communication. This work was made possible by generous financial support of Michael Alden Associates, Inc. of New Orleans. Publication was aided by support from the University of Phoenix, Louisiana Campus.

References:

[1] Date, C.J. *Introduction to database systems,* Sixth Edition, Addison-Wesley Publishing Company, 1995.

[2] Guy, R. G., *Ficus, A Very Large Scale Reliable Distributed File System,* Technical Report CSD-910018, Computer Science Dept. UCLA, June 3, 1991

[3] Huizinga, D.M. and Heflinger, K. A. Two-level client caching and disconnected operation of notebook computers in distributed systems. *ACM SIGICE Bulletin,* Vol 21 p 9, July 1995.

[4] Kawell, L. Jr., Beckhardt, S., Halvorsen, T., Ozzie, R., Greif, I., Replicated Document Management in a Group Communication System, *Proceedings of the Second Conference on Computer-Supported Cooperative Work,* Portland OR, September 1988.

[5] Kumar, P., Satyanarayanan, M., Flexible and Safe Resolution of File Conflicts, *Proceedings USENIX Technical Conference,* New Orleans, LA, January, 1995. p 95-106.

[6] Reiher, P., Heidemann, J., Ratner, D., Skinner, G., Popek, G., Resolving File Conflicts in the Ficus File System, *Proceedings 1994 Summer Usenix Conference.*

[7] Terry, D. B.,Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., Hauser, C. H., Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System, to be published in *Proceedings 15th ACM Symposium on Operating Systems Principles,* December 3-6, 1995.

[8] Online Guide, Overview Section, *dbSynchro, demo version August 1994,* Eclipse Computer Solutions, Ft Wayne, IN.