A RELIABLE AND EFFICIENT SYNCHRONIZATION PROTOCOL FOR DATABASE SHARING SYSTEMS

Erhard Rahm

University Kaiserslautern, FB Informatik, Postfach 3049
D-6750 Kaiserslautern, West Germany

Abstract:
Database sharing (DB-sharing) refers to a loosely or closely coupled multiprocessor architecture where all processors share a common database at the disk level. Such systems primarily aim at high availability and high performance demanded by large applications in online transaction processing. To achieve these goals a synchronization technique is required that efficiently coordinates the processors' database accesses and that also works properly after a processor crash. The described primary copy algorithm seems to be a good candidate to meet these requirements since it permits flexible adaption to changing working conditions. Besides of the basic protocol we specify the recovery actions after a processor crash. Mechanisms are proposed that allow to continue synchronization after a crash with little interference to transaction processing. Furthermore, the required redundancy is provided with nearly no extra costs during normal operation.

1. INTRODUCTION

High availability is a major demand in online transaction processing. Large applications in banking, inventory control or flight reservation processing typically allow an outage of five minutes per year only /Gr85/. Therefore, each major software and hardware component (processor, disk, controller, inter-processor connections, etc.) should at least be duplicated to provide sufficient fault tolerance /Ki84/. Furthermore, component failures have to be transparent to the users, modifications in the software/hardware configuration should be performed online and the common database must be a consistent and up-to-date reflection of the state of the business at any time. Kim has stated in /Ki84/ that loosely coupled multiprocessors where the common database is either partitioned among the processors (**DB-distribution**) or shared (**DB-sharing**), offer the best framework for building a highly available system. In this paper we concentrate ourselves on DB-sharing systems. A comparision between DB-sharing and DB-distribution can be found in /HR86, HR87/; /Ki84/ and /Se84/ present systems that claim high availability.

Besides of high availability, a DB-sharing system also aims at high transaction rates with short response times. Whereas current database management systems (DBMS) at best achieve about 200 - 300 (short) transactions per second (tps) of the 'Debit-Credit'-type /An85/, high-volume transaction processing systems require more than 1000 tps in the near future /Gr85/. The problem for a loosely coupled DB-sharing system to reach these transaction rates (with acceptable response times) is the expensive communication with messages, even if a high-speed communication system is used (process switches, send and receive operations). To make communication more efficient, one could use a common memory partition (e.g. for synchronization) resulting in a closely coupled DB-sharing system. Here, we only discuss loosely coupled systems that provide better availability and expandability.

Fig. 1 depicts a loosely coupled DB-sharing system where all processors (DBMS) share access to a single set of databases. In such a system, transactions can always be completely executed at one processor since each CPU has direct access to the entire database (in particular, no distributed commit protocol is required). A global load control located at one or more front-ends distributes each incoming transaction to one of the processors (transaction routing). The direct attachment of the disk drives to all CPUs implies physical contiguity of the processors, but it also permits a high-speed communication system (e.g. 10 - 100 MB/sec). Examples of DB-sharing systems are the Data Sharing facility of IMS/VS /Ke82/, Computer Console's Power System /We83/, the DCS project /Sek84/ and the Amoeba project /Sh85/.

A main advantage of DB-sharing systems is flexibility. Since each processor can access the entire database, transaction load can be dynamically distributed among the processors according to current needs and

system availability. Additional processors can be added without altering the transaction programs or the database schema. Likewise, a processor failure does not prevent the surviving processors from accessing the disks or the terminals. Transactions in progress on a failed processor can be backed out and redistributed automatically among the available processors.

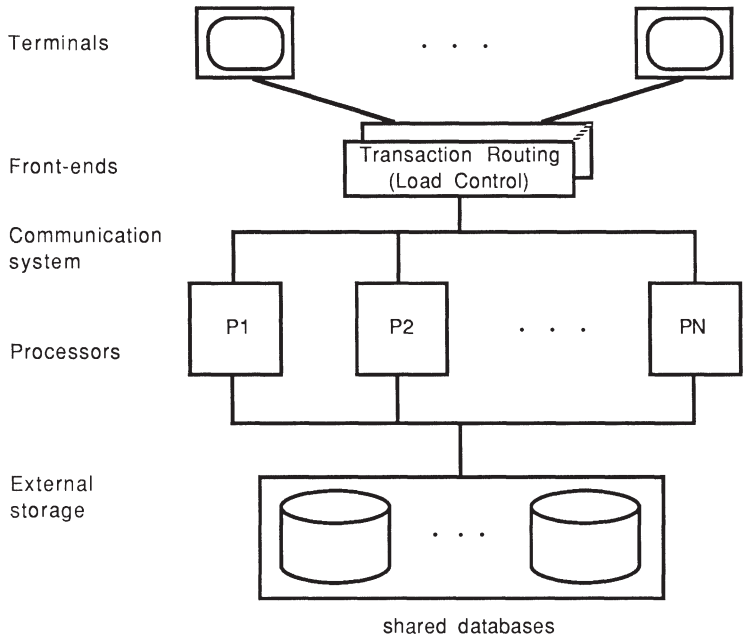


Fig. 1: Structure of a loosely coupled DB-sharing system

Naturally, the design of a DB-sharing system requires new or extended functions to be added, compared to centralized DBMS:

- The **synchronization** component has to coordinate the processors' accesses to the shared database in order to guarantee serializability of the executed transactions. Since there is no common memory, concurrency control requires message exchange among the processors which is much more time consuming than lock request handling in a centralized DBMS. Because these interprocessor communications directly influence response times and throughput, the algorithm used for concurrency control must minimize the number of synchronization messages as far as possible. Furthermore, the synchronization algorithm has to provide sufficient resiliency to maintain consistency of the database even in the presence of failures. The availability of the database has especially to be preserved across individual processor crashes. A number of conceivable synchronization techniques for DB-sharing are surveyed in /RS84,Ra86b/. Optimistic protocols are discussed in /Ra87b/.

- **Buffer control** is needed to manage the problem of **buffer invalidation** that results from the existence of a local database buffer in each processor. An update operation modifies only the processor's local copy of a database object; copies of the same object in other buffers are getting obsolete. Therefore, accesses to such invalidated objects must be avoided and a method to propagate the new contents of modified objects to other processors has to be supplied. If an update transaction writes its modified pages to the database on disk before commitment (FORCE-strategy, /HR83/), then the latest version of an object can always be read from disk. With NOFORCE, on the other hand, modified objects may be exchanged directly between the processors (via the interprocessor connections) or also across the shared disks.

- **Load control** has to find an effective strategy for transaction routing such that all processors are well utilized (however, without overloading any processor) and locality of reference (to decrease the amount of

disk-I/O and buffer invalidation) is maximized. Furthermore, a cooperation between load control and concurrency control should be possible in order to reduce the number of synchronization messages. Load control also has to react dynamically to changes in the workload and to the crash or reintegration of a processor.

The **recovery component** is responsible for system-wide logging and recovery. Each processor has to maintain a local log required for transaction undo and crash recovery. Additionally, a global log (e.g. for media recovery) is constructed by merging the local log data. Crash recovery is performed by the surviving CPUs in order to continue transaction processing. Uncommitted transactions of the failed processor are backed out and restarted on another processor.

In addition to the realization of these components, DB-sharing has to provide a 'single system image', i.e. at least end users and programmers should be relieved from the existence of multiple processors. Also very important are a high level interface to the users, ease of installation, ease of maintenance and ease of modifications. Managability and maintainability have direct influence to reliability since most system failures are caused by users and operators /Gr86/.

In this work we focus on the synchronization problem that is also related to load control, recovery and buffer control. In section 2 we describe how synchronization is performed with the primary copy algorithm under 'normal' conditions. Section 3 shows the behavior of the protocol in the presence of failures, especially after a processor crash.

The primary copy algorithm to be described is based on similar algorithms used in distributed DBMS with replication /BG81/. In the latter schemes, each data item has a primary copy controlled by one of the processors. Besides of synchronization, the primary copy processor is also responsible of updating all copies of the data item. With DB-sharing, however, no replicated data must be controlled in this way (although another kind of replication is treated by the buffer control).

2. PRIMARY COPY LOCKING (PCL)

In this approach the synchronization responsibility is distributed among all processors. Therefore, the database is logically partitioned into N disjoint parts and each of the N processors performs the global synchronization for one partition. A processor is said to have the **primary copy authority (PCA)** for its partition /RS84/. As Fig. 2 shows, each lock manager maintains a global lock table (GLT) to control the objects of its partition and a local lock table (LLT) to keep information about granted or requested locks for local transactions.

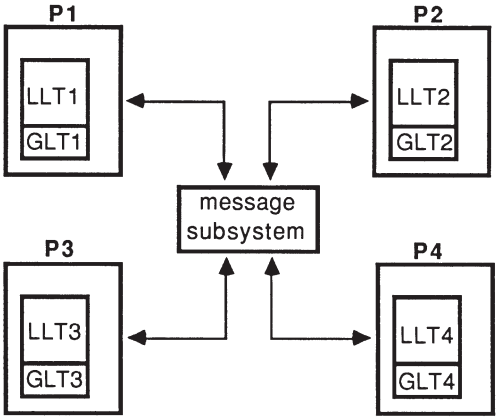


Fig. 2: Primary Copy Locking (N=4)

PCL has the obvious advantage that lock requests against the local partition can be managed without communication, regardless of external contention. Only lock requests against non-local partitions have to be sent to the authorized processor. To minimize the number of such 'long' lock requests, load control should attune the partitioning of the data and the assignment of the load appropriately instead of routing

transactions at random. It is this potential for reducing the communication overhead that makes the primary copy approach attractive for DB-sharing.

Up to now, it has not been precisely specified what kind of information is kept in the local and in the global lock tables and how synchronization can be performed using this information. We assume in the sequel that synchronization is performed on block level (page level) and that two types of locks are obtainable for a transaction: read or shared locks (S-locks) and write or exclusive locks (X-locks). The compatibility of these locks is as usual.

Data structures

Both types of lock tables (local and global lock table) use a different format for their control blocks or block entries required for synchronization. The exact layouts of the block entries are given in Fig. 3. We assume that a block entry for a certain block may simultaneously reside in the global lock table as well as in the local lock table of the same processor. This has the advantage that a redistribution of the PCAs can be done without changing the LLTs. Keeping block entries not only in the GLTs also allows a reconstruction of a GLT that was lost due to a processor crash by merging block entries from the LLTs (see next section).

```
Structure of a LLT block entry:
    BLOCK-ID: ...
    GRANTED-LOCKS-LIST: list of TA ENTRY;
    LOCAL-WAIT-LIST:    list of TA-ENTRY;

Structure of a GLT block entry:
    BLOCK-ID: ...
    INTERESTED:  array [1:n] of bit;
                 (* N processors *)
    MODE:   array [1:N] of [0, S, X];
    GLOBAL-WAIT-LIST: list of TA-ENTRY;

Structure of a TA-ENTRY:
    PNR: ... (* Identification of the processor;
                only required for GLOBAL-WAIT-LIST *)
    TA-ID: ...   (* transaction ID within processor PNR *)
    LOCK-MODE: [S, X];
```

Fig. 3: Structure of the block entries

The example situation of Fig. 4 shows the block entries for a block B1 within the lock tables of two processors. We assume that processor P1 holds the PCA for the partition to which B1 belongs. In the block entries of the LLTs only local transactions are kept, either in the list of the granted locks or in the LOCAL-WAIT-LIST. In the GLT, transactions waiting for a lock are stored within the GLOBAL-WAIT-LIST. In this list transactions of all processors can wait.

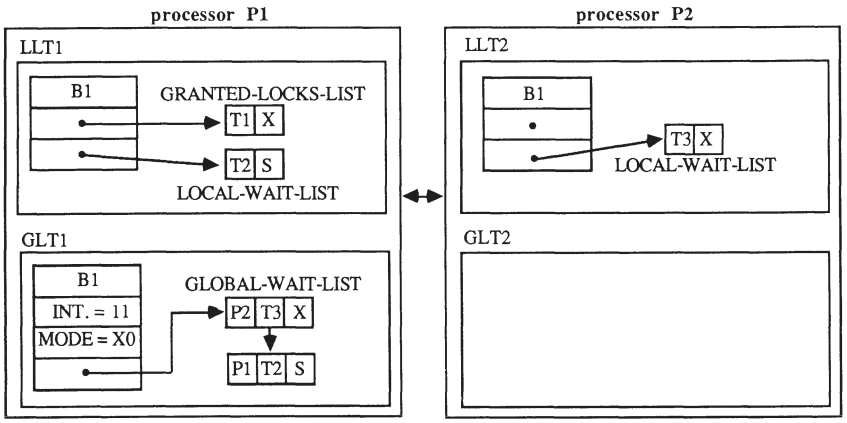


Fig. 4: Lock scenario with two processors

In addition to the GLOBAL-WAIT-LIST, a block entry in the GLT contains further information (as shown in Fig. 3) in order to process lock requests. The vector INTERESTED indicates the processors keeping a block entry for the respective block in their LLT. The vector MODE gives the mode of granted locks for the interested processors. Value MODE (P) = 0 says that processor P is not interested in the block, or that there are only transactions waiting for a lock on the block. Value MODE (P) = X indicates that an X-lock has been granted to a transaction at P, value S means that a S-lock was granted. In Fig. 4, 'INT. = 11' is used as an abbreviation of INTERESTED (1) = 1 and INTERESTED (2) = 1. This means that both processors are interested in the block. Similarly, MODE = X0 stands for MODE (1) = X and MODE (2) = 0. This says that an X-lock was granted to a transaction at P1 and that no lock for B1 is granted at P2.

Now we are in the position to explain lock request processing using the data structures just introduced.

Lock request processing

Assume transaction T at processor P has issued a lock request for block B. PCL handles this lock request as follows:
If P is the processor owning the primary copy authority for the requested block, it is checked whether or not the GLT already contains a block entry for B. If this block entry does not exist, the lock can be granted since T is the only transaction that wants to access B. In this case, block entries for B are created within the GLT and within the LLT at P, the vectors INTERESTED and MODE are initialized properly, and T is inserted into the GRANTED-LOCKS-LIST of the block entry in the local lock table.

If the GLT already holds a block entry for B, the lock request of T can be satisfied if the GLOBAL-WAIT-LIST is empty and if the required lock mode is compatible with the granted locks (decidable by using vector MODE). Otherwise, T has to wait for the desired lock and is appended to the GLOBAL-WAIT-LIST and also to the LOCAL-WAIT-LIST in the LLT.

If P does not own the primary copy authority for block B, the lock request cannot be treated locally. So, T is entered in the LOCAL-WAIT-LIST in the block entry of B in the LLT (the block entry may have to be created at first) and a lock request message is sent to the responsible processor, say P'. P' uses its GLT for processing the lock request message as just described for the local case. Only if the lock is grantable a lock response message is sent immediately. Otherwise, T is appended to the GLOBAL-WAIT-LIST in the GLT of P'. In that case, the lock manager of P' activates T (using a lock response message) at the time when the conflicting transactions have released their locks on B and T is chosen from the GLOBAL-WAIT-LIST to obtain the requested lock. After receipt of the lock response message, T is removed from the LOCAL-WAIT-LIST and inserted into the GRANTED-LOCKS-LIST.

For illustration, look at Fig. 4 once more. Assume that at the time when transaction T1 had issued its X-request for block B1, there was no interest in B1 at processor P2. Therefore, the GLT at P1 had contained INTERESTED = 10 and MODE = X0 for B1 when the X-lock was granted to T1. After that, suppose transaction T3 at P2 has wanted to modify block B1. In the LLT of P2 a block entry for B1 was created, then T3 was inserted into the LOCAL-WAIT-LIST, and finally a lock request message was sent to P1. This message resulted in a change of INTERESTED to 11, and T3 was inserted into the GLOBAL-WAIT-LIST since the requested X-lock was not compatible with the granted X-lock at P1. Fig. 4 shows the situation after the S-request of another transaction T2 running at P1. This lock request was also appended to the GLOBAL-WAIT-LIST.

In /Ra86c/ an optimization of the primary copy algorithm is proposed which provides a more effective treatment of S-locks that is especially important for level-2-consistency. Furthermore, solutions to the buffer invalidation problem are given that use additional information in the GLT and avoid extra messages as far as possible. The improved primary copy scheme has been implemented as part of a simulation system that is driven by real-life object reference strings in order to quantify the performance behavior (throughput, response times) of the algorithm. Some of our simulation results can be found in /Ra87a/.

Interaction with load control

The PCL algorithm described above should need little communication if an effective partitioning and transaction routing can be performed. To achieve this goal, load control needs a prediction of the presumable reference behavior of any transaction. Such predictions can be derived from the transaction

type identifying the subschema of the database accessed, and possibly the terminal identification and input data (parameters). At least for simple transactions this kind of information should allow a fairly precise estimation of which part of the database the transaction is going to operate on. Load control, which has to know the PCA distribution, can therefore route the transaction to that processor where most of the synchronization is local (provided this processor is not overloaded).

As shown in /Re86/, transaction routing can be done using a so-called **routing table**. The routing table is constructed for a certain load pattern and gives to each transaction type the processor(s) that allow the most efficient processing of transactions of that type. Furthermore, it guarantees for the assumed workload a good utilization of the system without overloading any processor. In /Ra86a/ some heuristics are given that allow a coordinated calculation of a routing table and a PCA distribution.

The possibility of a coordinated determining of the routing strategy and the PCA distribution allows a perfect cooperation between load control and concurrency control. In particular, it is possible to adapt the routing table and potentially the PCA distribution - when the workload changes significantly (what may be recognized by monitoring the arrival rates of the transaction types /Re86/). Similarly, the routing table and the PCA distribution can be adapted when a processor fails or is added (reintegrated) to the system. This potential of a flexible adaption to changing working conditions should generally result in a sufficient reduction of synchronization messages as needed for high transaction rates and short response times.

Note, that a redistribution of the PCAs requires that the global lock tables must be rearranged accordingly. Furthermore, after a processor crash the lost GLT of the failed processor need to be reconstructed in order to continue synchronization on the corresponding partition. The next section gives solutions to these problems.

3. RELIABLE PRIMARY COPY LOCKING

To provide continuous operation and failure transparency, the synchronization component must be capable of dealing with failures in the system. The following types of failures are usually considered in distributed environments:

a) transaction failure
b) processor failure (crash)
c) failure within the communication system
d) failure of other components (disk, controller, etc.)

Transaction failures do not pose any new problem to be solved. The effects of an uncommitted transaction can be undone by the executing processor (possibly using the local log) and the locks are released by informing the responsible lock managers.
Failure types b, c and d require sufficient redundancy and their treatment is - with the exception of the processor crash - nearly independent of the concurrency control algorithm in use. For instance, a disk failure may be recovered using a mirrored volume or alternatively by applying the global log to an archive copy. Similarly, the failure of an inter-processor connection requires a redundant connection; a reliable communication protocol must ensure that no messages are lost.

In the rest of the paper we assume a robust and fault-tolerant communication system where no messages are lost and all messages are delivered in finite time. Furthermore, it is assumed that failures are independent from each other, and that no failure occurs during the recovery of another failure. We discuss in the following how PCL can cope with processor crashes and the integration of new or repaired processors. Crash recovery requires that the local log of a failed processor is accessible by the surviving CPUs.

After a processor crash the contents of the volatile memory including the lock tables are assumed to be lost. The recovery of the failed processor must be done in cooperation with load control and includes rolling back uncommitted transactions, redoing lost effects of committed transactions, reconstruction of the lost GLT, redistribution of the PCAs, modification of the routing table and restart of the failed transactions. For simplicity it is assumed that load control is a centralized function for which a stand-by (shadow) can take over control in case of a failure. The used data structures may therefore be dually kept in independent main memory segments so that the spare load control finds a consistent copy after a failure of the primary load control. Details of such a recovery procedure for the load control are, however, beyond the scope of this paper; a suitable implementation may use similar mechanisms as described in /Sek84/.

As mentioned in the previous sections, load control performs transaction routing and load balancing. It keeps the following information:

- the routing table
- a PCA table, that holds the current PCA distribution
- further information about available processors, utilization, etc.

Fig. 5 shows examples of the routing table and the PCA table. The routing table (Fig. 5a) gives for each transaction type the processor(s) a transaction of that type should be routed to. If more than one processor is eligible (for T2 and T5 in the example), the routing decision depends on the current utilization of these processors. The PCA table is kept in each processor and determines the processor being responsible of a lock request (a lock request for a block B is directed to that processor that holds the PCA for the database fragment to which B belongs). A database **fragment** is a portion of the database (relation, segment, etc.) for which a PCA is assigned. A partition consists of all fragments that are controlled by the same processor. For example, the partition of processor P3 constitutes from fragments D1, D5 and D6.

transaction type	processors
T1	P2
T2	P2, P1
T3	P1
T4	P3
T5	P2, P3
...	...

a) routing table

database fragments	responsible processor
D1	P3
D2	P1
D3	P2
D4	P2
D5	P3
D6	P3
D7	P1

b) PCA table

Fig. 5: Routing table and PCA table (example)

Load control can also be made responsible of detecting a processor crash and of initiating and coordinating the recovery actions for a failed processor. The detection of processor failures may be done by an 'active' approach /Ki84/ where each processor sends 'I-AM-ALIVE' messages in periodic time intervals. If load control does not receive these messages from a processor P for a while, a failure is assumed and recovery actions are initiated. Similarly, the failure of load control itself can be detected by the shadow load control.

In the following we describe the actions that take place after the crash of processor P has been detected. The six steps to be described can partially be executed in parallel. These cases are explicitly mentioned and are clarified in Fig. 7.

Actions after a processor crash

Step 1 *(Broadcast of the crash event):*

Load control broadcasts that P has failed to prevent that further messages are sent to P. After that, all messages concerning partition D, for which P was owning the PCA, are buffered until the recovery actions are completed. Received messages that were sent by P before the crash are simply ignored. Transaction processing can only continue on other partitions than D.

Step 2 *(Initiation of recovery):*

Load control initiates the undo and redo operations for P (step 3) and the reconstruction of the GLT for partition D (step 4), and starts itself constructing the new routing table and the new PCA distribution. Since steps 3 and 4 are independent operations they can be performed on different processors in order to speed up the recovery process.

Step 3 *(Undo and redo operations):*

a) The recovery must undo the effects of uncommitted transactions in the database and potentially partially redo the results of committed transactions using the local log of P. In a FORCE-environment all modifications of a transaction are propagated to the database at EOT; thus, no redo actions have to take place. With NOFORCE, however, the modified pages that were only in P's buffer at crash time must be written to the database. The undo and redo operations may be performed by one of the surviving processors or by several (e.g. if long-running batch transactions are incorporated) to keep the recovery time acceptably small /Sh85/. Note that the recovering CPUs must have access to the local log of the failed processor.

b) Locks held by failed transactions have to be released in order to eliminate unnecessary lock conflicts. For partition D formerly owned by P this is done during the construction of the new GLT (step 4); for the other partitions the corresponding lock managers must release the locks.
If the PCL scheme is also used to deal with buffer invalidation (as described in /Ra86c/) then additional provisions are necessary. It has to be ensured that the latest version of each block of partition D can be read from disk. Using a FORCE-strategy, this is already given when the undo operations are completed. With NOFORCE, however, it is possible that one of the surviving processors holds in its buffer the most recent version of a page belonging to partition D whereas the copy on disk is obsolete. Those pages must be written out to prevent that the obsolete copy on disk is accessed. Furthermore, all other pages of partition D that are in the buffer of a surviving processor irrespective of whether a FORCE or a NOFORCE strategy is applied - must be considered as invalidated (unless they are in use) since the information to decide about invalidation has been lost with the GLT of the failed processor. Therefore, these pages must be removed from the buffer.

c) The original input messages of the failed transactions are read from the local log of processor P and are transmitted to load control. These transactions are restarted after the recovery has been completed and the new PCA distribution is established (step 6).

Step 4 *(Construction of the GLT for partition D):*

The GLT can be constructed by merging all block entries for blocks of partition D residing in any of the available local lock tables (LLT). First, load control determines the processor P' that has to construct the new GLT, and informs all processors within the initial broadcast message (step 1) that P' is in charge of the reconstruction of the lost GLT. Then, all surviving processors (without P', of course) send all block entries found in their local lock tables and belonging to partition D to P'. (Such a block entry indicates that a local transaction owns or waits for a lock.)

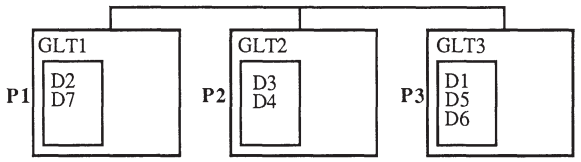
For each block B represented in the LLT of (at least) one of the surviving processors, a block entry in the new GLT is created. INTERESTED (I) is set to '1' for such a block B, if processor I has a block entry for B in its LLT, and to value '0' otherwise. The GLOBAL-WAIT-LIST is simply a union of all LOCAL-WAIT-LISTs (the order within the GLOBAL-WAIT-LIST need not be the same as in the lost GLT). Note, that no waiting transaction is left out, since a transaction is appended to the LOCAL-WAIT-LIST before the lock request is sent to another processor. Vector MODE can be constructed from the GRANTED-LOCKS-LISTs.

Since the lock information from the failed processor P is not available, the construction process automatically generates a GLT without lock requests or MODE-information concerning any (failed) transaction of P. So, the new vector MODE indicates when waiting locks are grantable. For example, assume that a transaction at P has held an X-lock for a block B, and that transactions of other processors are waiting for a lock on B. In the LLTs of these other processors, the block entry for B must have empty GRANTED-LOCKS-LISTs and LOCAL-WAIT-LISTs with the waiting transactions. Now, the construction of the new GLT generates the GLOBAL-WAIT-LIST and a vector MODE with value 0 for all processors. This indicates that lock requests from the GLOBAL-WAIT-LIST are satisfiable. However, the activation of the waiting transactions must be delayed until the recovery process is terminated and processing on partition D can be continued (step 6).
The GLT constructed in the described manner reflects a consistent synchronization state of partition D. The used block entries of the LLTs are from the point in time when the processors were informed about P's crash. These block entries remain unchanged until it is explicitly notified that processing on partition D can continue.
After having built the GLT for partition D, processor P' would be in the position to synchronize accesses to D after successful recovery. However, such a strategy would lead to an overloading of P' in most cases.

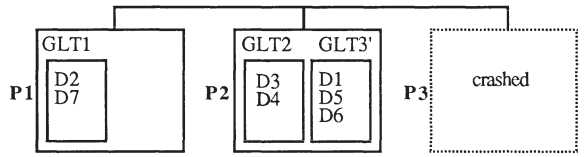
Therefore, P' merely informs the load control, which computes a new PCA distribution, that the GLT for partition D is restored.

Step 5 *(Establishing the new PCA distribution):*

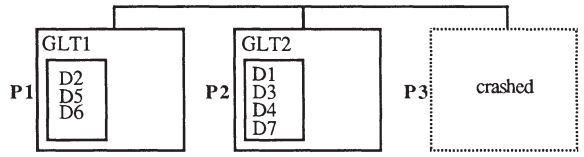
After load control has computed the new routing table and the new PCA table, and the construction of the GLT for D (step 4) has been completed, the GLTs in the system must be established according to the new PCA table (note, that step 3 may still be in progress; however, step 5 and the release of locks in step 3b should be executed in mutual exclusion). The database fragments for which the PCA should be reassigned may also belong to another partition than D. For these other fragments, transaction processing must now also be stopped; no messages concerning these fragments may be sent or processed until the new PCA distribution is established.
The new PCA distribution indicates the database fragments for which a change of the PCA processor should take place (hereto, P' can be seen as the temporary owner of the fragments belonging to partition D). All block entries in the GLTs belonging to these fragments are transmitted to the new PCA-processor. These block entries are inserted in the GLT of the receiving processor and deleted from the GLT of the sending processor. In addition to the block entries, messages (e.g. lock responses) that were buffered due to the blocking of a database fragment are also transmitted to the processor that should overtake the PCA. When the fragment can be accessed again, these messages must be processed or send.
When all GLTs are reestablished, load control is informed.



a) Situation before the crash of processor P3



b) Situation after construction of the lost GLT



c) Situation after adaption of the GLTs to the new PCA table

Fig. 6: Steps in establishing a new PCA distribution after a processor crash (example)

To illustrate the steps discussed so far, an example is given in Fig. 6. The GLTs in Fig. 6a represent the situation where all (three) processors were active and the PCA table of Fig. 5b was in use. Assume now that processor P3 has crashed and that P2 has constructed the new GLT - as described in step 4 - for the fragments formerly controlled by P3. This situation is shown in Fig. 6b. In the meantime, assume that load control has computed the following PCA table:

database fragments	responsible processor
D1	P2
D2	P1
D3	P2
D4	P2
D5	P1
D6	P1
D7	P2

To establish this PCA distribution the block entries for fragments D5 and D6 must be transmitted from P2 to P1, and those for fragment D7 from P1 to P2. Fig. 6c shows the new situation.

Step 6 *(Start processing on blocked database fragments):*

When all GLTs are established according to the new PCA distribution, load control sends the new PCA table to all processors. For partition D processing cannot be continued until recovery step 3 is completed; all other partitions are fully available now. When a database fragment is no further 'blocked', the processors are informed by the load control. Then, buffered messages for the fragment are transmitted or processed, and waiting transactions can possibly be activated if their requested locks are grantable.
When the entire database is available again, load control routes each transaction that failed on P to one of the survivors (according to the new routing table) for re-execution.

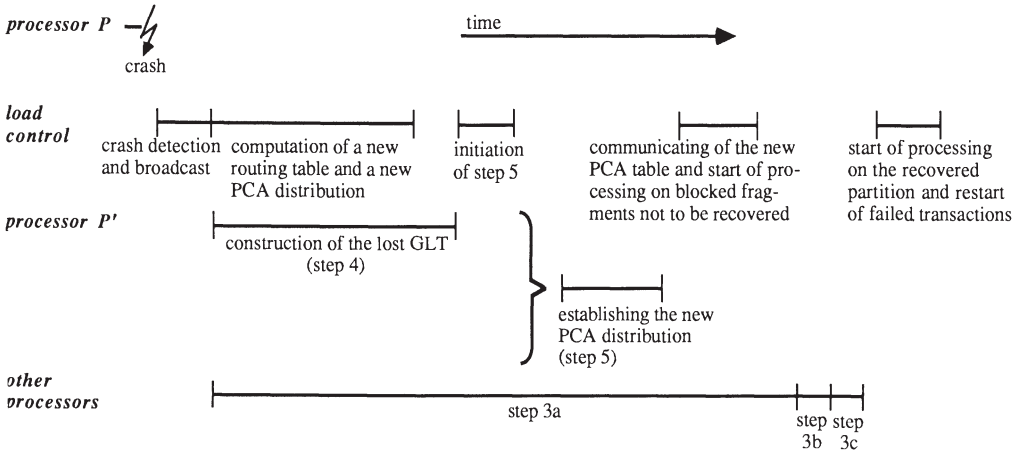


Fig. 7: Parallelism in the recovery actions after a processor crash

It should have become clear that the recovery actions after a processor crash require a coordinated effort of load control, the recovery component and the synchronization component (including buffer control). Since the recovery can be done by several processors in parallel (Fig. 7), it is supposed that the time a partition is blocked can be kept tolerably small. In most cases, especially if a NOFORCE-strategy is applied, step 3 seems to be the most time consuming one, since it may require a lot of disk writes. On the other hand, the exchange of block entries between the processors in steps 4 and 5 should only require a small fraction of the time for step 3 (fast communication system).

So, the construction of the GLT (step 4) and the establishing of the new PCA distribution (step 5) does not lengthen the recovery process, in general. In addition, only the partition to be recovered is blocked completely during recovery; other database fragments get unavailable at worst for the duration of step 5. Therefore, transaction processing should not be severely interfered after a processor crash. Furthermore, the redundancy in the LLTs used for reconstruction of a lost GLT is provided with nearly no extra costs

during normal operation. The latter point is a clear advantage over checkpointing schemes /IYD84/ that cause a high communication overhead to keep the data structures of a shadow or backup process, running on a separate processor, up-to-date.

Integration of a new (repaired) processor

It is assumed that the integration of a processor requires the intervention of the operator to inform the load control. Load control then determines the new routing table and a new PCA distribution regarding the new processor. Finally, this PCA distribution is established as described above (step 5) and the new PCA table is communicated to each processor. The use of the new routing table ensures that the new processor gets suitable transactions for execution.
The introduced method for establishing a new PCA distribution is also applicable when the load control has recognized significant changes in the workload and has modified the PCA table to adapt synchronization to the new load pattern.

4. SUMMARY

We have described the primary copy algorithm for synchronization in a loosely coupled DB-sharing system. This decentralized locking scheme also allows an integrated solution to the problem of buffer invalidation.
It was shown how primary copy locking (PCL) cooperates with load control in order to adapt synchronization to changing working conditions and to reduce the communication overhead for concurrency control. The recovery actions after a processor crash are described in some detail. Mechanisms were proposed that allow to continue synchronization after a processor failure with little interference to transaction processing during recovery. So, a method for reconstructing a lost global lock table was supplied and also for adapting the PCA distributions after a processor crash or after integration of a new processor or when the transaction load has significantly changed. A main advantage of our proposal is that the required redundancy to reconstruct lock tables after a processor crash can be provided with nearly no extra costs during normal processing.

5. REFERENCES

/An85/ Anon. et al.: A Measure of Transaction Processing Power. Datamation, April 1985

/BG81/ Bernstein, P.A.; Goodman, N.: Concurrency Control in Distributed Database Systems. ACM Comp. Surveys, Vol. 13, No. 2, 195-221

/Gr85/ Gray, J. et al.: One Thousand Transactions per Second. Proc. IEEE Spring CompCon, 1985, 96-101

/Gr86/ Gray, J.: Why Do Computers Stop and What Can Be Done About It. Proc. 5th Symp. on Reliability in Distr. Software and Database Systems, 1986, 3-12

/HR83/ Härder, T.; Reuter, A.: Principles of Transaction-Oriented Database Recovery. ACM Comp. Surveys, Vol. 15, No. 4, 1983, 287-317

/HR86/ Härder, T.; Rahm, E.: Multiprocessor Database Systems for High Performance Transaction Systems. Informationstechnik, Vol. 28, No. 4, 1986, 214-225, in German

/HR87/ Härder, T.; Rahm, E.: High Performance Database Systems - Comparision and Evaluation of Current Architectural Approaches and their Implementation. Informationstechnik, Vol. 29, No. 3, 1987, in German

/IYD84/ Iyer, B.R., Yu, P.S., Donatiello, L.: Comparitive Analysis of Fault Tolerant Architectures for Multiprocessors. IBM Research Report RC-10876, Yorktown Heights, 1984

/Ke82/ Keene, W. N.: Data Sharing Overview. IMS/VS V1, DBRC and Data Sharing User's Guide, Release 2, G30-5911-0, 1982

/Ki84/ Kim, W.: Highly Available Systems for Database Applications. ACM Comp. Surveys, Vol. 16, No.1, 1984, 71-98

/Ra86a/ Rahm, E.: Algorithms for Efficient Load Control in Multiprocessor Database Systems. Applied Informatics 4/86, 161-169, in German

/Ra86b/ Rahm, E.: Concurrency Control in DB-Sharing Systems. Proc. 16th GI Annual Conf., Informatik Fachberichte 126, Springer 1986, 617-632

/Ra86c/ Rahm, E.: Primary Copy Synchronization for DB-Sharing. Information Systems, Vol. 11, No. 4, 1986, 275-286

/Ra87a/ Rahm, E.: Performance Analysis of Primary Copy Synchronization in Database Sharing Systems. Internal Report 165/87, Dept. of Computer Science, Univ. Kaiserslautern, 1987

/Ra87b/ Rahm, E.: Design of Optimistic Methods for Concurrency Control in Database Sharing Systems. Proc. 7th Int. Conf. on Distributed Computing Systems, 1987

/Re86/ Reuter, A.: Load Control and Load Balancing in a Shared Database Management System. Proc. 2nd Data Eng. Conf., 1986, 188-197

/RS84/ Reuter, A.; Shoens, K.: Synchronization in a Data Sharing Environment. Technical report, IBM San Jose Research Lab., 1984.

/Se84/ Serlin, O.: Fault-Tolerant Systems in Commercial Applications. IEEE Computer, Aug. 1984, 19-30

/Sek84/ Sekino, A. et al.: The DCS - A New Approach to Multisystem Data Sharing. Proc. National Comp. Conf., 1984, 59-68

/Sh85/ Shoens, K. et al.: The AMOEBA Project. Proc. IEEE Spring CompCon, 1985, 102-105

/We83/ West, J.C. et al.: PERPOS Fault-Tolerant Transaction Processing. Proc. 3rd Symp. on Reliability in Distr. Software and Database Systems, 1983, 189-194

This work was financially supported by Siemens AG, Munich.