to appear in: Proc. 21st Int. Conf. on Fault-Tolerant Computing (FTCS-21), Montreal, June 1991

Recovery Concepts for Data Sharing Systems

Erhard Rahm

University Kaiserslautern, Department of Computer Science,
P.O. Box 3049, D-6750 Kaiserslautern, Germany

Abstract

Data sharing refers to a general distributed architecture for high performance transaction processing. The nodes of a data sharing system are locally coupled via a high-speed interconnect and can directly access all disks and thus the entire database. While concurrency and coherence control protocols for data sharing have been discussed in previous work, the important area of recovery has mostly been ignored. This paper discusses the new problems for crash and media recovery that have to be addressed in data sharing systems. Recovery is complicated by dependencies on other functions such as buffer management and concurrency control. Furthermore, a global log file is to be constructed where the modifications of committed transactions are reflected in chronological order. New logging and recovery protocols are proposed for loosely coupled data sharing systems that employ the primary copy approach for concurrency/coherence control. A comparison with existing data sharing systems shows that our protocols support high performance during normal processing as well as efficient recovery to provide high availability.

1. Introduction

Data sharing systems have received a great deal of attention in recent years [SUW82, WIH83, Se84, Ra86, Sh86, Fu86, Yu87, Bh88, MNP90]. They offer a promising approach for building high performance transaction systems which have to satisfy ever increasing performance, availability and growth requirements.

Data sharing embodies a locally distributed architecture where multiple transaction processing nodes are coupled via a high-speed interconnect. Each node in such a complex may be a tightly coupled multiprocessor running its own copy of the operating system and DB/DC system. Communication between different nodes typically takes place by means of message passing (loose coupling). Another option not considered in this paper would be to use shared semiconductor stores (close coupling) for data exchange to reduce the communication overhead associated with message passing. The characteristic feature of data sharing is that no partitioning of the external database is required (as in so-called 'data partitioning' systems), but that all nodes have direct physical access to the common database(s) on disk. This architecture is therefore also known as 'shared disk'. Benefits of the data sharing approach over data partitioning systems (e.g. increased flexibility for load balancing) are discussed in [Sh86, Ra91].

Existing data sharing systems and prototypes include IMS Data Sharing [SUW82, QV87], Computer Console's Power System 5/55 [WIH83], NEC's Data Sharing Control System [Se84], IBM's Amoeba prototype [Sh85], Fujitsu's Shared Resource Control Facility [Fu86], and DEC's VaxClusters [KLS86]. IBM's TPF [TPF88] also supports a 'disk sharing' by multiple (eight) nodes but without offering full DBMS capabilities and transaction management. TPF supports locking and data caching within disk control units and achieves very high transaction rates (several thousand transactions per second) [Sc87].

To take advantage of the data sharing architecture, a number of complex problems have to be addressed, notably in the areas of concurrency and coherence control [Ra88], workload allocation [Ra91] and recovery:

- In loosely coupled data sharing systems, inter-node communication is required for concurrency and coherence control. *Concurrency control* is obviously needed in order to synchronize the accesses to the shared database and to enforce global serializability. *Coherence control* is necessary because every node maintains a database buffer in main memory to cache pages from the shared database. Thus, modification of a page in one buffer makes all copies of that page in other buffers (and on disk) obsolete. Coherence control has to make sure that these *buffer invalidations* are either avoided or detected and that all transactions get access to the current versions of database objects. The number of messages for concurrency and coherence control has to be kept as low as possible to reduce the communication overhead and to limit transaction deactivations due to remote requests.
  [Ra88] summarizes and classifies previous research on concurrency and coherence control, and compares the performance of several protocols by means of detailed, trace-driven simulations. The best simulation results were observed for a primary copy locking protocol that solves the buffer invalidation problem in an integrated way to avoid additional messages for coherence control. Specific recovery considerations in this paper will therefore be based on this protocol.
- *Workload allocation* is responsible for distributing the transaction workload among the processors. This transaction routing should not be statically determined by a fixed allocation of terminals and/or programs to nodes, but should be automatic and adaptive with respect to changing conditions in the system (e.g. overload situations, node crashes, etc.). Effective workload allocation schemes not only aim at achieving load balancing (to limit resource (CPU) contention), but also at supporting an efficient transaction processing with a minimum of inter-node communication or I/O delays. The latter is comparatively easy to realize for primary copy locking by means of a so-called affinity-based routing that attempts to assign transaction types (presumably) accessing the same database portions to the same node (see below). A general discussion of workload allocation in distributed transaction systems and a classification of conceivable solutions can be found in [Ra91].
- *Recovery* is the main subject of this paper. Despite its importance, recovery has mostly been ignored in previous publications on data sharing. Crash recovery and media recovery are the major recovery forms that require new solutions for data sharing. Crash recovery for a failed node has to be performed by the surviving nodes in order to provide high availability. The realization of this recovery form depends on many factors - including the underlying protocol for concurrency and coherence control - that will be discussed in section 2 in more detail. In general, lost effects of transactions committed at the failed node have to be redone (REDO recovery) while modifications of in-progress (failed) transactions may have to be undone. Special recovery actions may be necessary to properly continue concurrency and coherence control, e.g. reconstruction of lost control information. Media recovery may require the construction of a *global log* where the modifications of all nodes are recorded in chronological order [Sh85]. *Disaster recovery* can be performed according to proposals for data partitioning systems [Ly88, KHGP90]

where all updates are asynchronously applied to a copy of the database at a remote data center.

This paper concentrates on crash and media recovery in data sharing systems. The next section discusses the major problems and design factors that appropriate solutions need to consider. Section 3 describes our logging and recovery protocols for data sharing systems that employ the primary copy approach to concurrency and coherence control. Section 4 compares our proposal with recovery schemes employed in existing data sharing systems.

2. New problems

To restrict the scope of our discussion, we make a number of assumptions that are common in existing DBMS. We generally assume an *update-in-place* policy (as opposed to shadow-page techniques) where an updated database page is written back to the same disk location from where it was read. Furthermore, we assume that concurrency control is based on locking (locks are held until the transaction is committed or aborted) and that physical logging (as opposed to logical or operation logging) is employed. To limit the number of log writes and the volume of log data, we log only the modified portions of a page rather than the entire pages. This *entry logging* also supports an effective group commit where the log data of multiple transactions can be written in a single I/O [DeW84, He87]. It is further assumed that every node in the data sharing system maintains a *local log file* where it records all modifications of locally executed transactions. Other nodes can access the local log file to perform crash recovery.

Crash recovery in data sharing systems is complicated because of close dependencies on buffer management, concurrency control and coherence control. With respect to buffer management, we distinguish between the so-called FORCE and NOFORCE alternatives as well as between STEAL and NOSTEAL policies requiring different logging and recovery protocols. Another design factor is the concurrency control granularity (page- vs. record-level locking). At the end of this section, we discuss the construction of a global log file that may be needed for crash and media recovery.

FORCE vs. NOFORCE strategy for update propagation

The buffer manager is said to employ a FORCE strategy if all pages modified by a transaction are forced (written) to the permanent database on disk before commit [HR83]. Otherwise, a NOFORCE policy is said to be in effect. FORCE simplifies crash recovery and coherence control. During crash recovery, no REDO recovery will be necessary for committed transactions. Coherence control is simpler compared to NOFORCE since the most recent version of a page can always be obtained from disk (at least when no concurrent modification of the same page at different nodes is permitted).

On the other hand, high performance requirements generally prescribe a NOFORCE policy that permits a drastically reduced I/O overhead and avoids the increase in transaction response time and lock hold time due to synchronous disk writes at EOT (end of transaction). With NOFORCE, only redo information (after-images) is written to a log file at EOT, and multiple modifications can be accumulated per page before it is written to disk. These disk writes can be performed asynchronously (before the corresponding page is selected for replacement) to avoid response time delays. For NOFORCE, crash recovery has to redo all committed modifications that were lost by a node failure. Coherence control has to keep track of as to where the most recent version of modified pages can be obtained. Modifications can either be directly transferred between nodes ('buffer-to-buffer') or across the shared disks.

STEAL vs. NOSTEAL policy for page replacement

STEAL permits the buffer manager to replace *dirty pages* (containing modifications of uncommitted transactions) and to write them to the permanent database. The use of a STEAL policy results in a considerable complication for logging and recovery since it requires before-image logging (write-ahead log (WAL) principle) and UNDO recovery after a processor crash [HR83, Mo89]. Note that FORCE usually implies STEAL since modified pages are still uncommitted when they are forced to the permanent database at EOT. Consequently, before-images have to be written to the log according to the WAL principle.

With NOSTEAL only unmodified pages or pages with committed updates can be replaced. It avoids the need to write undo information to the log; no UNDO recovery is necessary after a node crash. The modifications of transactions aborted during normal processing are backed out in main memory (no I/O). If modifications are performed on private (main memory) page copies, this can be done by simply throwing away the copies of failed transactions. Otherwise, undo information has to be kept in main memory buffers until the transaction is committed.

Concurrency control after a node crash

Recovery has to be coordinated with the concurrency control protocol to permit its proper continuation after a system crash. Lock information held in main memory of the failed system is lost and may have to be reconstructed during crash recovery. For instance, if a central lock manager (CLM) is used for concurrency control the failure of the central node requires a complete interruption of transaction processing until the global lock information is reconstructed and a new CLM is determined. A fast and simple recovery is possible if a copy of the global lock table is held on stable storage or in a stand-by node; this approach, however, causes a high checkpointing overhead during normal processing since the lock information is frequently updated. A more efficient scheme uses local lock information in the surviving nodes to reconstruct the global lock state.

Recovery processes update the database for redoing modifications of committed transactions so that they may have to acquire locks to prevent interference with transactions running on the surviving systems. The simplest approach taken in some existing systems (section 4) is to interrupt transaction processing during crash recovery, i.e. to lock the entire database exclusively. The obvious disadvantage is that availability is significantly impaired as it may take several seconds or even minutes to complete recovery. Note that no locks need to be acquired for undo recovery because the respective objects are still locked on behalf of the failed transactions. These locks must be released after recovery completion. Also, no redo recovery is necessary for objects locked by transactions running on the surviving nodes. This is because the current version of the respective objects are either in main memory of the surviving systems or on disk (guaranteed by the coherence control protocol).

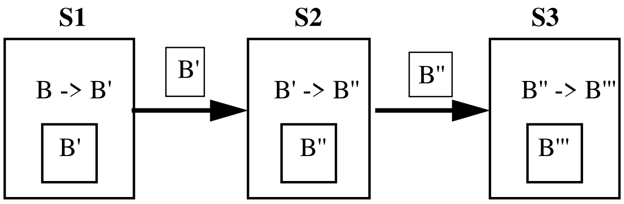
Dependencies on coherence control

Buffer invalidations can be controlled by extended lock information since locking schemes require a transaction to acquire a lock before accessing database objects [Ra88]. These integrated schemes avoid extra messages for coherence control to a large extent thus improving performance considerably, e.g. compared to broadcast invalidation schemes requiring extra messages to invalidate page copies at other nodes. On the other hand, a loss of lock tables and coherence control information can have negative consequences for recovery. If, for instance, version numbers are used to detect invalidated page copies together with the lock request processing ('on-request invalidation'), losing this information during a crash could make it necessary to clear all database buffers to avoid access to obsolete data. The broadcast invalidation scheme does not share this problem since it removes invalidated pages 'immediately' from the database buffers. Furthermore, no coherence control information has to be reconstructed during crash recovery.

Another trade-off between efficiency during normal processing and recovery exists for the method used to exchange modifications between nodes. If modified pages are always exchanged across the shared disks (and a page is not permitted to be concurrently modified in different nodes), crash recovery is simplified since it can be done with the local log file of the crashed node. This is because all modifications of other nodes are already reflected in the permanent database so that at most, updates of the crashed node may have to be redone (or undone). On the other hand, buffer-to-buffer communication is much more efficient during normal processing. It permits a page transfer time of about 1 ms as opposed to 50+ ms (2 I/Os) introduced by a page exchange across shared disks (non-volatile disk caches could permit faster transfers of about 4-8 ms).

With a direct exchange of modifications, a page can be modified at multiple nodes before it is written to disk. Hence, redoing modifications of a crashed node may require the use of redo information from remote nodes (Fig. 1). The log information for a given page is dispersed over the local log files of all systems where the page has been modified. To apply the redo information in correct order, the local log information may have to be merged in a global log. This difficulty is avoided by employing either page logging (and page locking), or FORCE or by exchanging modified pages across the shared disks. These alternatives simplify recovery at the expense of a reduced performance during normal processing.

The analogous problem exists for UNDO recovery in the case of STEAL. During crash recovery, undo log records from remote nodes may have to be applied [Ra89].



Page B has been successfully modified at S1, S2 and S3 without updating the permanent database. During crash recovery for S3 all three modifications have to be redone.

Figure 1: Redo problem with direct exchange of modified pages (NOFORCE, entry logging)

Page vs. record locking

With page locking, a database page can only be modified by one transaction at a time or it can be concurrently read by multiple transactions. While this level of concurrency is generally appropriate for most database objects, frequently accessed pages may require record-level locking to keep lock contention sufficiently low. Since record-level locking results in a higher number of lock requests, communication delays and overhead are likely to increase for data sharing. What is more, record locking considerably complicates coherence control for data sharing since different nodes can then modify different records of the same page in parallel. This has the effect that page copies in the database buffers may only partially be up-to-date so that no node holds a completely valid page version. Since writing partially up-to-date pages to the permanent database could lead to lost updates, a merging of modifications would be necessary before a modified page could be replaced from the database buffer. Coordinating disk writes and merging modifications, however, introduces additional communication requirements. Furthermore, merging modifications of different page portions may be impossible for insert or delete operations that change the page structure.

To avoid these problems, existing data sharing systems only support page locking or a restricted form of record locking. A good compromise may be a hierarchical scheme with page locking for resolving lock conflicts between nodes and for coherence control, and record locking within individual nodes to permit concurrent write accesses to different records of a page. In [Ra89], we discuss further forms of record locking for data sharing that permit a higher concurrency.

Construction of a global log

A global log can be thought of as a combination of all local log files with the log records ordered chronologically. A global log can be used for crash recovery and media recovery.

In existing data sharing systems, the global log is mostly constructed by an off-line utility that merges the local log files. This approach is not acceptable if high availability is crucial since recovery has to be delayed until the global log is completed by this utility. A simple approach to construct the global log on-line is to write the global log data during commit at the current end of the global log file. By holding the write locks until the global log write is completed it is guaranteed that the log records are in correct order. The problem with this approach is that it increases response times and thus lock hold times. Furthermore, the global log must be kept in multiple partitions in order to avoid a single hot spot that limits throughput. A shared, non-volatile semiconductor store could largely eliminate the performance problems of this scheme [Ra90].

The alternative is to merge the local log files on-the-fly, e.g. by dedicated log processors that scan the local log files or receive the log information directly from the nodes (extra messages). To sort the log information, log records must be tagged with global timestamps. *Page timestamps* (version numbers) permit us to order the log records on a per-page-basis. With page locking for update accesses, these timestamps can be maintained with no extra communication (every modification increments the version number). Alternatively, *transaction commit timestamps* allow a total ordering of all log records. A global counter can be used to assign monotonically increasing commit timestamps. If one node maintains this counter, extra communication is needed to get the timestamp; storing the counter in a common disk/storage location requires the synchronization of the increment operation. The current clock value can also be used as a commit timestamp provided the local clocks are tightly synchronized or a common hardware clock is available.

The discussion shows that there are several trade-offs involved in the design of recovery protocols for data sharing. Since efficiency during normal processing should be more important than simple and fast recovery protocols, we consider NOFORCE as mandatory for high performance data sharing. Integrated concurrency/coherence protocols and direct exchange of modifications also improve performance during normal processing (compared to broadcast invalidation and an exchange of modified pages via disk) at the expense of more complex recovery protocols. Record-level locking and STEAL do not necessarily improve performance compared to page locking and NOSTEAL, but they complicate coherence control and recovery considerably. However, at least a restricted form of record-level locking seems indispensable to prevent that lock contention becomes the performance bottleneck.

The next section presents recovery protocols for data sharing systems with primary copy locking. Our schemes will support a NOFORCE environment, entry logging and an integrated concurrency/coherence control with a direct exchange of modifications between nodes. In this paper, the protocols will be described for page-level concurrency control. The extensions needed for record-level locking cannot be presented here due to space limitations, but are described in an extended version of this report [Ra89]. For simplicity we assume a NOSTEAL environment, although our protocols can be extended to STEAL (see section 5). Ever increasing main memory sizes make NOSTEAL a reasonable choice since there should generally be enough 'non-dirty' pages that can be selected for replacement.

3. Recovery protocols for primary copy locking

Before we can discuss logging and recovery protocols, it is necessary to introduce the methods used for concurrency and coherence control with primary copy locking (PCL). Therefore, sections 3.1 and 3.2 provide a brief summary of these methods to make the paper self-contained (For a more detailed description of PCL, the reader is referred to [Ra86]. The original primary copy scheme was proposed for replicated databases [St79]). In 3.3, we describe the logging activities during normal processing, particularly during commit processing. Crash recovery and media recovery are discussed in 3.4 and 3.5, respectively.

3.1 Concurrency control

PCL is a distributed scheme where the database is divided into *logical* partitions and each node is assigned the synchronization responsibility (or *primary copy authority, PCA*) for one partition. Lock requests against the local partition can be handled without communication overhead and delay, while other requests have to be directed to the authorized processor holding the PCA for the respective partition.

To minimize the number of remote lock requests, workload allocation and PCA allocation should be coordinated such that transactions are assigned to the node where most of their database references can be locally synchronized. (This allocation goal is subject to load balancing constraints to avoid overload situations, if possible.) This approach is made possible by the prevalence of pre-planned transaction types (e.g. debit-credit transactions in banking applications) for which the reference distribution is known or can be obtained by DBMS-internal monitors. When transactions can be routed to the node where most database accesses are locally synchronized, we yield a high degree of node-specific locality of reference where each node's partition is mainly referenced by local transactions. Thus, affinity-based routing (transactions with affinity to the same database portions should be assigned to the same node) can be relatively easily supported with PCL. As trace-driven simulations [Ra88] have confirmed, this helps not only to reduce the number of remote lock requests, but also the frequency of I/Os (better hit ratios), buffer invalidations and lock conflicts with remote transactions. This resulted in substantially better performance (transaction rates, response times) than with a random distribution of the workload, uncoordinated with the PCA distribution (e.g. PCA allocation determined by a hash function). An optional improvement of PCL is the use of a so-called *read optimization* which permits a local synchronization of read accesses for which another node holds the PCA [Ra86, Ra88].

In contrast to the static data allocation in data partitioning systems, the PCA distribution can be adapted dynamically since the database itself is not partitioned. This would typically be done after a node has failed or been added, or together with an adaptation of the routing strategy, e.g. to deal with overload situations or when the load profile changes significantly. Methods to determine workload and PCA allocations are discussed in [Ra91].

3.2 Coherence control

For coherence control, various techniques can be chosen in combination with PCL [Ra86]. We focus here on one *on-request invalidation* (check-on-access) scheme that works for NOFORCE, a direct exchange of modifications between nodes and page-level concurrency control. On-request invalidation schemes use extended information in the (global) lock table to decide upon the validity of a cached page together with the lock request processing. Thus, buffer invalidations are detected without any additional communication, a main advantage compared to broadcast invalidation schemes. For the purpose of this discussion, we assume a simple approach using version or sequence numbers in the page header that are incremented by every update operation. For modified pages, the PCA lock manager maintains the version number of the current page copy in its lock table; this information can be updated when the write lock required for the modification is released (no extra message). Before a cached page is accessed, a lock generally has to be requested at the responsible PCA lock manager. The version number of the cached page copy is sent together with the lock request so that the PCA lock manager can check whether or not the copy is obsolete.

If a cached page copy is invalidated or the page is not cached at all, a transaction must be provided with the most recent version of the page. With FORCE, the transaction could simply read the page from disk. With NOFORCE, however, the page version on disk is obsolete if the page has been successfully modified at one (or more) node(s) and not been written back. One approach to deal with this difficulty is to record in the PCA node's lock table at which node a page has been modified most recently and to request the page from this processor after a lock has been granted.

Though these page requests can usually be satisfied faster than a disk I/O, they introduce extra messages and communication delays in addition to remote lock requests. This disadvantage can be avoided by an approach where the transmissions of modified pages are also combined with regular concurrency control messages. First, modified pages belonging to the partition of another node are transmitted to the responsible PCA processor together with the message required for releasing the write lock (at EOT). This has the effect that the PCA node always gets the most recent page versions for its partition. Thus, buffer invalidations are now limited to pages belonging to another node's partition. Moreover, when a lock is granted to a remote transaction, the PCA node can send the most recent page version directly to the requesting transaction, together with the lock response message. (If the PCA node does not hold a copy of the page in its buffer, it indicates in the lock response message that the page can be read from disk.) In this way, the respective transaction does not need to be deactivated again for requesting the page from another node or reading it from disk. Therefore, our coherence control scheme does not require extra messages for detecting buffer invalidations or for exchanging modified pages between different sites. The example in Fig. 2 illustrates this protocol.

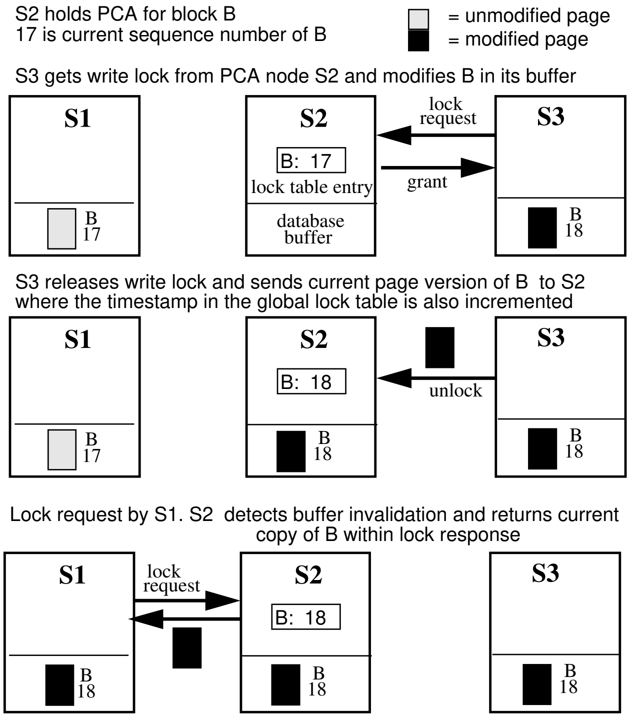


Figure 2: Example for PCL with on-request invalidation and direct exchange of modifications

Modified pages are only written out by the PCA node. This avoids an explicit coordination of disk writes which may be necessary in other schemes for NOFORCE and direct exchange of modifications to make sure that no invalidated page copy is written out.

3.3 Logging

Our logging and recovery protocols are based on physical entry logging and NOSTEAL and support the above coherence control with NOFORCE and a direct exchange of modifications. A central idea is to utilize the fact that the PCA node receives all modifications of its partition and to log these changes in the PCA node's local log file, even when the modifications have been performed on a remote processor. This approach avoids the explicit construction of a global log file since the local log files already represent a distributed form of a global log where each PCA node's log file holds the log information for the respective database partition in chronological order. This implicit global log is constructed with little extra overhead for communication and I/O and will facilitate crash recovery as well as media recovery.

Log record types

For our discussion, four log record types are relevant:

1. *Redo log records* contain the after-image for modified page portions or records. They contain the corresponding transaction and page identifier as well as the page's version number associated with the modification. The transaction identifier is assumed to consist of a node number and a local transaction number to separate redo log records of local from those of remote transactions. Redo log records of remote transactions are guaranteed to belong to committed transactions (see below), while local redo records may have been written by uncommitted transactions. The version number is stored to determine whether or not a redo log record has to be applied to a database page.

   Redo log records are first written into a main memory buffer that is written to the log file when it is full or at specific points in time (e.g. transaction commit or abort). Such a buffering permits us to write multiple log records with one I/O and to implement group commit. When a log buffer is written, all log records are appended to the current end of the local log in the order of their insertion into the log buffer.

2. *Commit log records* indicate the successful completion of an update transaction. They are only written to the local log file of the node where the transaction has been executed (with data sharing, there is no need for a distributed execution of transactions, e.g. with remote subtransactions).
3. *Abort log records* indicate the rollback of an update transaction and are only written at the transaction's execution node. All redo log records of aborted transactions will be skipped during REDO recovery. UNDO recovery will not be necessary due to the NOSTEAL assumption.
4. *Checkpoints* are taken to limit the redo work after a node crash. In our scheme, every node can take checkpoints independent of other nodes, i.e. there is no need to synchronize checkpoint activities. Checkpoints can basically be taken as in centralized database systems. Since direct checkpoints that write modified pages from the database buffer to the permanent database would lead to intolerable write delays (thousands of pages may have to be written), we use so-called fuzzy checkpoints [HR83]. They are periodically taken and record only status information in the log file such as the names of currently active transactions at the respective node and the identifier of modified pages of the local database partition. For every modified page, it is assumed that the buffer manager records the log file address (LSN or log sequence number) of the first redo log record. This START_LSN for modified pages is also written to the log during a checkpoint and marks the point on the log where redo processing has to begin. To limit the redo work, the buffer manager writes out frequently modified pages on a continuous basis (as in [Mo89]) so that their START_LSN can be reset. To take a checkpoint, a BEGIN_CHKPT record is first written to the log, followed by the status information and a END_CHKPT log record.

Commit processing

The steps during the commit of an update transaction are summarized in Fig. 3. The first commit phase is basically the same as in centralized database systems with NOFORCE. The log buffer with the commit record of the update transaction is written to the local log file (possibly delayed because of group commit). When the commit record has reached the local log file, the transaction is already guaranteed to be committed and its results can be shown to the terminal user. All redo log records of the transaction have been written to the log prior to the commit record (in previously written log buffers or in the log buffer containing the commit record).

In the second commit phase, first locks for locally controlled pages are released. If no remotely controlled data has been accessed, commit processing and transaction execution is completed at this point. Otherwise, lock release messages are sent to the responsible PCA nodes together with the modified pages and information describing which page portions have been modified. (An alternative is to transmit only the modified records/page portions. In this case, the PCA node has to copy these modifications into the page which may first require to read in the unmodified page.) To cope with transmission failures, the modified pages and log information remain buffered in the transaction's node until the PCA node has acknowledged their receipt. A buffering of the log records is also necessary to deal with situations where the PCA node crashes before the log records of remote modifications reach its log (see below).

At the PCA node, the modified pages are copied into the database buffer thereby possibly overwriting older versions of the respective pages. Furthermore, the remote redo log records are appended to the log buffer. Since every record in the log buffer will be written to a known address at the local log file, the LSN for a remote redo log record can be determined now, and START_LSN is set to this value if the respective page has not been modified before. After this, coherence control information (version numbers) is updated and the locks are released. Remote redo log records are written

```
at transaction's execution node:
  force log buffer with commit record to local log; {tx is now committed}
  release locally controlled locks;
  for remotely controlled objects do;
    keep redo log records in separate log buffer;
    send unlock message with modified pages/redo log records to
                    PCA node;
    wait for ACK by PCA node;
    clear log buffer
  end;

at PCA node:
 bring modified pages into database buffer;
 append remote redo log records to log buffer;
 increase version numbers in lock table;
 release locks;

 ACK message is returned after log buffer with remote redo log
 records has been written out {e.g. due to commit of a local transaction};
```

Figure 3: Commit Processing

asynchronously (bundled) after the write locks have been released to avoid an increase in lock contention. Similarly, ACK messages which confirm that the remote log records have been written are piggy-backed to other messages to avoid extra communication overhead. On the PCA node's log file, all redo log records for its partition are chronologically ordered since redo information for remote modifications are inserted into the log buffer before the write locks are released.

Bundling remote log operations and ACK messages permit a construction of the distributed version of the global log with almost no extra I/O or communication overhead during normal processing. In addition, remote logging does not increase response times (as perceived by the terminal user) since the transaction is already committed after a successful local logging in commit phase 1. Only remote modifications are logged to two log files while an explicit construction of the global log introduces a double logging for all modifications. In general, coordinating workload and PCA allocation should permit that most modifications are performed at the PCA node avoiding a remote logging as well as a transfer of modifications at EOT.

We note in passing that in data partitioning systems both commit phases are distributed, in general. A distributed first commit phase increases transaction response times and lock holding times.

3.4 Crash recovery

When a node fails, its lock table and buffer contents are lost. As a consequence, all further database processing with respect to the partition of the failed node must be stopped until crash recovery is completed. Blocking further accesses to the failed node's partition also avoids interferences of recovery processes with transactions running on the surviving nodes. Transaction processing on the remaining partitions can continue in parallel to the crash recovery for P. Transaction types (or terminals) assigned to P before the crash must be reassigned among the surviving nodes by the workload allocation strategy.

For the following we assume that P denotes the crashed processor and D its database partition. One of the surviving nodes, say Q, is in charge of coordinating the recovery actions; another node R is responsible for REDO recovery with P's log file (Q=R is possible). After the detection of P's crash, Q sends a broadcast message to block all further accesses to D. This message is also used to indicate that lock information and buffered redo log records for partition D that are needed for REDO recovery (see below) are to be sent to R. Apart from REDO recovery for P, crash recovery includes a reallocation of the PCAs and adaptation of the workload allocation strategy, and a reconstruction of lost lock information for partition D. The loss of coherence control information (version numbers) requires that pages of partition D cached in the surviving nodes be considered obsolete and removed, unless their validity is ensured by a granted lock. Our recovery scheme permits transactions to keep locks for D to avoid that granted locks have to be reacquired and the respective transactions aborted.

REDO recovery

The REDO recovery is performed by one of the surviving nodes, say R, with the local log file of P. Since the coherence control information for D has been lost by P's crash, it is necessary to bring all pages of D in the permanent database up to the most recent state that reflects all successful modifications at any node. This can be done almost exclusively by processing P's local log due to the logging of remote modifications at the PCA node. In addition, the taking of checkpoints should generally restrict the redo work to a small portion of D that has been modified shortly before the crash.

In the following we describe how the redo recovery for D is performed before we discuss redo recovery for P's modifications of remote partitions.

For redoing lost modifications of partition D, R processes P's log file in two passes, an analysis pass and a redo pass. The analysis pass starts at the last complete checkpoint taken by P and reads the log file until its end. This pass determines a list of *loser transactions* , i.e. transactions for which an abort log record has been found in the log, or which have been active at checkpoint time or for which redo log records have been found but no commit log record. The analysis pass also determines the pages of partition D for which redo recovery is to be performed in the second pass. These are the pages that have been recorded in the checkpoint as well as the pages of D for which redo log records have been found after the checkpoint. The checkpoint information also indicates the start point on the log for the redo pass (the minimum of all START_LSN values).

While P's log file contains all redo log records of locally committed transactions, some remotely committed modifications of D may be missing due to transmission delays and buffered log writes in P. These modifications are still in the log buffers of the surviving nodes since P has not acknowledged that they have been written to its log file. The buffered redo log records for partition D are sent to R which appends them to P's log file and adds the page identifiers to its list of pages to be redone.

All pages of D for which according to the analysis pass a redo recovery may be necessary, are read from the permanent database by R. (These disk reads can be heavily optimized by ordering them so as to minimize disk seek times and by reading, if possible, from different disks in parallel). The redo pass starts at the log address determined in the analysis pass and processes all redo log records until the end of the log file. A redo log record is applied if the following conditions hold:

- the corresponding page belongs to partition D and has been read in by R after the analysis pass
- the modifying transaction is not a loser transaction
- the version number of the log record is higher than the version number stored in the page header.

At the end of the redo pass, all updated pages of partition D are written back to the permanent database. At this point, R writes a checkpoint to P's log stating that all modifications of partition D are reflected in the permanent database.

The redo processing from the log can be avoided for pages of partition D that are cached in their current version at the surviving nodes (e.g. because of a granted lock). Instead of reconstructing these pages from the page copy on disk and the redo log information, they can be directly written to the permanent database by one of the nodes with a current copy ('dirty' page copies must not be written out due to NOSTEAL).

Committed transactions at P may have modified not only pages of partition D but also pages of remote partitions. These modifications, however, do not have to be redone, in general, since they are transmitted to the PCA node at EOT. Therefore, P's committed modifications of remote partitions are already available at the surviving nodes, except for a few modifications which may have been logged at P shortly before the crash and whose transmission to the PCA node could not be completed because of P's crash. To determine the respective pages, each surviving node informs R about which transactions from P still hold write locks on pages of their partition (only for these transactions may redo log information be outstanding). If for one of these transactions a commit record is found on P's log, its redo log records for remotely controlled pages are read from P's log and sent to the responsible node where the modifications are logged again and applied to the respective database pages and the locks are released. The locks of loser transactions are also released at the surviving nodes to avoid unnecessary lock conflicts on their database partitions.

PCA reallocation and reconstruction of lock information

If it is to be expected that node P remains unavailable for more than a few seconds, a reallocation of the PCAs is necessary to continue processing on partition D. To limit the relocation overhead, only the fragments of partition D are newly assigned among the surviving nodes while the assignments for the other partitions remain unchanged (Fragments constitute the units of PCA allocation. Since we have a logical partitioning, they can be chosen almost arbitrarily small (e.g. page ranges)). In the simplest case, the entire partition D is assigned to one node, e.g. to processor R that performed the REDO recovery and already holds many pages of D modified before the crash in its database buffer. Overload situations at R after the recovery are not necessarily introduced since only lock processing is associated with the PCA owners while the largest part of a transaction can be executed at any node (contrast this with data partitioning, where the database operations are processed where the data resides). On the other hand, assigning P's partition to one node and its transaction types to multiple processors could lead to an increased number of remote lock requests. If this appears to be a problem for the workload to be processed, partition D can be split among multiple nodes. It is assumed that the recovery coordinator Q determines the new PCA (and workload) allocation and broadcasts it to all nodes. All nodes that have received the PCA ownership for some fragments record this in their log file; in P's log file the new owners for the fragments of D are also recorded.

With the crash of P, the global lock table for partition D has been lost. A simple approach to continue concurrency control on D after the completion of recovery would be to abort all transactions that have requested or obtained locks on D before the crash. After the completion of crash recovery, these transaction would be reexecuted and all locks on D would have to be reacquired at the new PCA node(s). The rollback and reexecution of these transactions can be avoided, however, since every node keeps track (e.g. in a local lock table) of requested/granted locks of local transactions for remote partitions. For partition D, this lock information can therefore be sent to the new PCA node(s) to reconstruct the global lock table by merging this information. The global lock information (lock owners, waiting lock requests) can be completely reconstructed except for transactions from the failed node P. Their lock requests, however, are irrelevant since these transactions are failed and have to be restarted anyway. The merged lock information from the surviving nodes indicates situations where lock requests are waiting because of locks held by P's transactions before the crash. These locks are implicitly released now and the waiting lock requests can be granted after the recovery.

Coherence control information for on-request invalidation cannot be reconstructed from information of surviving nodes. Therefore, the version numbers for pages from D are set to 0 in the global lock table indicating that the current page copy can be read from the permanent database (insured by the REDO recovery). As mentioned above, pages of D already cached can be used after recovery when their validity is guaranteed by a granted lock.

Resuming processing on partition D

When REDO recovery is completed and the new PCA allocation is established, the recovery coordinator Q can broadcast the end of P's crash recovery and release the blocking of partition D. Transactions failed at P because of the crash can be distributed among the surviving nodes (according to the new workload allocation strategy) for reexecution. By using logged input messages of these transactions, the rollback and reexecution can generally be kept transparent to the terminal user.

When the crashed node is to be reintegrated again, the old PCA and workload allocation may be re-established. During the transition, partition D must be blocked and the global lock information is transferred from the current PCA owners to P. Modified pages of D in the buffers of the current PCA nodes are either written to the permanent database or also sent to P. REDO recovery and adaptation of a new PCA allocation can be done in parallel to limit the duration of crash recovery and blocking time for partition D. In our scheme, REDO recovery is very simple because of the logging of remote modifications by the PCA nodes. Its duration is mainly determined by the checkpoint frequency and the buffer manager that has to asynchronously write modified pages with low START_LSN values. The reconstruction of global lock information requires almost no extra overhead during normal processing in contrast to checkpointing schemes that maintain copies of lock tables in separate nodes or non-volatile storage.

Multiple node crashes

So far, we have only discussed crash recovery for the case of a single node crash. Our scheme is optimized for this case which is far more likely than multiple node crashes where other nodes may fail during the crash recovery of one node. Since crash recovery for a single node can largely be performed with the local log file of the failed processor, recovery from multiple crashes is similar to the recovery of multiple independent node crashes. (Crash recovery for already failed nodes may have to be started again when interrupted by a second node crash). One optimization that cannot fully be utilized any more is the main memory buffering of remote redo log records being used for redo recovery of a failed node's partition. This is because the crash of a second processor leads to the loss of its main memory log buffer that may have contained redo log records needed for the reconstruction of the first node's partition. However, since all redo log records of committed transactions are guaranteed to be in the local log file on disk, a correct redo recovery is still possible, although it takes longer to determine and read the required redo log records from the log file than using the in-memory copies. In the worst case, when all nodes have crashed, all local log files must completely be scanned and analysed to determine (using the version numbers) which redo log records are needed for the reconstruction of the local or any remote partition.

3.5 Media recovery

Media recovery is trivial with mirrored disks. Otherwise, damaged pages have to be recovered from an archive copy of the database by applying all redo log records (of committed transactions) for the respective pages in chronological order [HR83]. (This lengthy process can be avoided if a page to be recovered is buffered at its PCA node). Since the local log files typically hold only the most recent log records needed for crash recovery, older log records have to be kept on separate archive log files. Media recovery first applies redo log records from the archive log to the archive copy before the most recent modifications logged on the local log files are redone.
Archive copies and archive logs are typically maintained for entire database files or segments that constitute a collection of PCA fragments. Taking an archive copy can be done similarly as in centralized database systems, e.g. by a read-only transaction that reads every page of the segment and copies it to the archive dump. (This transaction acquires short read locks to make sure that no dirty page versions are copied). At the beginning of this transaction, the log address of the last checkpoint is recorded for every node holding the PCA for some of the fragments of the respective segment. These checkpoint addresses mark the points from where on redo log records have to be applied for this archive copy. As mentioned above, these redo log records are asynchronously copied to a separate archive log in order to limit the size of the local log files. Due to changes in the PCA allocation, the redo records for a fragment may have to be read from different local log files. Since we record the new PCA owner for a fragment in the log, the processing of redo log records can continue with the log file of this node. (When a node receives the PCA for a new fragment it can record the current log address in a main memory data structure. The processing of redo log records for media recovery or taking an archive copy can then start from this position).

4. Recovery in existing data sharing systems

As mentioned in the introduction, there is almost no discussion of recovery problems and solutions for data sharing in the literature. Although information on recovery facilities of existing data sharing systems is also very limited, we try to summarize their main features as described in the available documents.
TPF [Sc87, TPF88] is no full data sharing system since it provides only limited support for transaction management and recovery. TPF provides message logging and supports duplicating files for media recovery. Recovery procedures are to be provided primarily by the application system, e.g. the take-over by a stand-by system. High availability is only achievable with 'responsible applications' and trained system personnel. For the sake of fast restart times, the updates of some committed transactions may be lost after a node crash [Sc87].
A common feature of all 'real' data sharing systems appears to be the choice of a FORCE strategy in order to avoid the complexity of a REDO recovery after a node crash. IMS Data Sharing [QV87] employs the FORCE strategy in combination with a broadcast invalidation scheme for coherence control. It is restricted to two nodes and uses a distributed concurrency control scheme ('pass the buck') based on a logical token ring topology. Accesses can generally be synchronized at the record level, except for update accesses between nodes that are synchronized at the page level. Media recovery is based on archive copies and a global log file; a utility is offered to merge the local log files. A general observation from [QV87] is that recovery procedures for IMS Data Sharing rely heavily on operator interactions thus complicating system administration and making the recovery process susceptible to human errors and delays.
AIM/SRCF [Fu86] employs a majority consensus protocol for concurrency control where a lock request has to be granted by a majority of the nodes. Every node writes before-images (FORCE) and after-images into separate local log files. Utilities are provided for sorting and merging after-images from the local log files. Sorting is based on sequence numbers associated with database pages that are incremented for every page modification. Coherence control issues are not discussed in [Fu86].
A central lock manager for concurrency control is used in the data sharing systems of Computer Console [WIH83], NEC [Se84] and in the Amoeba prototype [Sh85]. These systems rely on FORCE and (presumably) a broadcast invalidation scheme for coherence control (coherence control is not discussed in [WIH83, Se84]). In Computer Console's system, the failure of the central lock manager results in a total 'freeze' of lock processing until a backup process has rebuilt the global lock table. In NEC's system, central lock services are provided by a special 'lock engine' that internally consists of up to eight processors. Global lock information is kept in two copies in independent main memory partitions so that after a failure a consistent copy of the global lock table is always available. In the Amoeba prototype [Sh85], it is assumed that a stand-by process can take over global concurrency control after a failure of the primary coordinator. For the creation of a global log, a special journal process runs on every node that forwards local log records to a merge processor running on a disk server. A stand-by merge process is assumed to take over if the primary merge fails.
DEC's DBMS for VaxClusters employ a distributed locking scheme for concurrency control and an on-request invalidation scheme, based on page sequence numbers, for coherence control [KLS86, ST87]. During crash recovery, all database processing is stopped ('database freeze') and failed transactions are undone (FORCE) [RSW89]. Furthermore, all locks of surviving nodes have to be reacquired after crash recovery to redistribute lock ownerships [ST87]. For media recovery, a global log file is constructed on a single disk [RSW89]. While they employ group commit to (synchronously) write the after-images to the global log file, the global log remains a potential bottleneck for transaction rates of more than 100 transactions per second.

Our protocols compare very favourably with the strategies of existing data sharing systems. Our approach primarily aims at high performance during normal processing while preserving reasonable complexity and efficiency for recovery. In contrast to current data sharing implementations, we support NOFORCE as well as a direct exchange of modifications between nodes. The broadcast invalidation scheme used for coherence control in most data sharing systems causes extra messages and response times delays for update transactions. Our on-request invalidation scheme described in 3.2, does not require extra messages for the detection of buffer invalidations or for the exchange of modifications (the DEC scheme is based on FORCE, i.e. an exchange of modified pages across disks). In simulation studies, the primary copy approach has shown to provide efficient concurrency and coherence control with less messages than (optimized) central locking strategies used in several data sharing systems [Ra88]. This resulted from the use of a read optimization, the efficient coherence control scheme, and the coordination of workload and PCA allocation (affinity-based transaction routing). Remote logging for modifications of the partition of another node permits an efficient REDO recovery and avoids the explicit construction of a global log by merging local log files. During crash recovery, only the database partition of the crashed node has to be blocked, while processing on other partitions can continue (no total database freeze). In general, remote logging is only needed for a small fraction of updates and does not introduce extra messages or disk writes during normal processing. As described in [Ra89], it is possible to extend our scheme to full record-level locking where the same page can be concurrently modified in different nodes. This level of concurrency is not supported by existing data sharing systems.

5. Conclusions

Logging and recovery algorithms for data sharing have to support high performance during normal processing as well as fast recovery from failures in order to provide high availability. Since these two subgoals often conflict with each other, it is generally necessary to find compromises. The discussion in section 2 revealed the major design issues and implementation alternatives to be considered and showed that the design of appropriate techniques is complicated by the close dependencies of logging and recovery on concurrency control, coherence control and buffer management. The new logging and recovery protocols presented in this paper are based on the primary copy approach to concurrency and coherence control. The primary focus was high performance during normal processing since node crashes, media failures or disasters should be comparatively rare events. Still, our logging technique permits a comparatively simple and fast crash recovery.

For performance reasons, we support a NOFORCE strategy for update propagation to disk, entry logging, and an on-request invalidation scheme for coherence control with a direct exchange of modifications between nodes. The recovery schemes are based on a 'double logging' of modifications of remote partitions. As in centralized DBMS, a transaction is committed as soon as its modifications and commit record are logged to the local log file. In the second commit phase, modifications of remote partitions are transferred to the PCA node where the changes are buffered, the locks are released and the modifications are logged again. By combining the transfer of modifications (and acknowledgement messages) with other messages and bundling log information, extra communication and log I/O is generally avoided by this double logging. In addition, it does not increase response times or lock holding times. The great advantage of this technique is that the log file of the PCA node holds all redo log records for its partition in chronological order (some modifications may still be buffered at remote nodes). This implicitly constructed, distributed version of a global log file permits a simple and fast redo recovery after a node crash with the local log file of the crashed node. For media recovery, an additional sorting of redo log records is also avoided.

Our approach for crash recovery exhibits several additional benefits, e.g. when compared with recovery strategies in existing data sharing systems (section 4) that are generally limited to FORCE. First of all, there is no need for a total database freeze, but only the partition of the crashed node has to be blocked during crash recovery. To speed up the recovery process, redo recovery can be performed in parallel to PCA relocation and reconstruction of lost lock information. Redo recovery can be avoided for cached pages whose validity is ensured by a transaction lock or read authorization. Locks and cached pages of the partition to be recovered can be retained by transactions on the surviving nodes so that their reacquisition after recovery completion is avoided. In addition, the recovery scheme can be extended to support full record locking [Ra89].

Further extensions of our scheme are necessary if a STEAL environment is to be supported, i.e. the replacement of pages with 'dirty' modifications. If restricted to pages of the local partition, this can easily be incorporated analogous to schemes for centralized DBMS; otherwise, undo log records also have to be transferred to and logged by the PCA node. An additional UNDO pass is then required during crash recovery to bring the partition of the failed node to the most recent transaction-consistent state.

References

[Bh88] A. Bhide: *An analysis of three transaction processing architectures* . Proc. 14th VLDB conf., 1988, 339-350

[DeW84] D.J. DeWitt et al.: *Implementation techniques for main memory database systems* . Proc. ACM SIGMOD conf., 1984, 1-8

[Fu86] *AIM SRCF functions and facilities* . Fujitsu Limited, Facom OS Techn. Manual 78SP4900E, 1986

[He87] P. Helland et al.: *Group commit timers and high volume transaction systems* . Proc. 2nd Int. Workshop on High Performance Transaction Systems, 1987 (published by Springer-Verlag, Lecture Notes in Computer Science 359, 1989)

[HR83] T. Härder, A. Reuter: *Principles of transaction-oriented database recovery* . ACM Computing Surveys,15 (4), 1983, 287-317

[KHGP90] R.P. King, N. Halim, H. Garcia-Molina, C. A. Polyzois: *Overview of disaster recovery for transaction processing systems* . Proc. IEEE 10th Int. Conf. on Distributed Computing Systems, 1990, 286-293

[KLS86] N.P. Kronenberg, H.M. Levy, W.D. Strecker: *VAX clusters: a closely coupled distributed system* . ACM Trans. on Computer Systems, 4 (2), 1986, 130-146

[Ly88] J. Lyon: *Design considerations in replicated database systems for disaster protection* . Proc. IEEE Spring CompCon, 1988, 428-430

[MNP90] C. Mohan, I. Narang, J. Palmer: *A case study of problems in migrating to distributed computing: data base recovery using multiple logs in the shared disks environment.* IBM Research Report RJ 7343, San Jose, 1990

[Mo89] C. Mohan et al.: *ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging* . IBM Research Report RJ 6649, San Jose, 1989 (revised 1990)

[QV87] M. Quenon, D. Voris: *IMS/VS Data Sharing Guidelines* . IBM Technical Bulletin, G320-0590, Dallas Systems Center, 1987

[Ra86] E. Rahm: *Primary copy synchronization for DB-Sharing* . Information Systems, 11 (4), 1986, 275-286

[Ra88] E. Rahm: *Design and evaluation of concurrency and coherency control techniques for database sharing systems* . TR 182/88, Univ. Kaiserslautern, Dept. of Computer Science, May 1988

[Ra89] E. Rahm: *Recovery concepts for data sharing systems* . TR 14/89, Univ. Kaiserslautern, Dept. of Comp. Science, Oct. 1989 (extended version of this paper)

[Ra90] E. Rahm: *Utilizing Extended Storage Architectures for High-Volume Transaction Processing* . TR 6/90, Univ. Kaiserslautern, Dept. of Comp. Science, Aug. 1990

[Ra91] E. Rahm: *A framework for workload allocation in distributed transaction processing systems* . To appear in: The Journal of Systems and Software

[RSW89] T.K. Rengarajan, P.M. Spiro, W.A. Wright: *High availability mechanisms of VAX DBMS software* . Digital Technical Journal, No. 8, Feb. 1989, 88-98

[Sc87] T.W. Scrutchin, Jr.: *TPF: performance, capacity, availability* . Proc. IEEE Spring CompCon, 1987, 158-160

[Se84] A. Sekino et al.: *The DCS - a new approach to multisystem data sharing* . Proc. National Comp. Conf., 1984, 59-68

[Sh85] K. Shoens et al.: *The Amoeba project* . Proc. IEEE Spring CompCon, 1985, 102-105

[Sh86] K. Shoens: *Data sharing vs. partitioning for capacity and availability* . IEEE Database Engineering, 9 (1), 1986, 10-16

[St79] M. Stonebraker: *Concurrency control and consistency of multiple copies in distributed Ingres* . IEEE Trans. on Software Engineering, 5 (3), 1979, 188-194

[ST87] W.E. Snaman Jr., D.W. Thiel: *The VAX/VMS distributed lock manager* . Digital Technical Journal, No. 5, Sep. 1987, 29-44

[SUW82] J. Strickland, P. Uhrowczik, V. Watts: *IMS/VS: an evolving system* . IBM Systems Journal, 21 (4), 1982, 490-510

[TPF88] *Transaction Processing Facility, Version 2 (TPF2)* . General Information Manual, Release 4.0, GH20-7450, 1988

[WIH83] J.C. West, M.A. Isman, S.G. Hannaford: *PERPOS fault-tolerant transaction processing* . Proc. 3rd IEEE Symp. on Reliability in Distr. Software and Database Systems, 1983, 189-194

[Yu87] P.S. Yu et al.: *On coupling multi-systems through data sharing* . Proc. of the IEEE, 75 (5), 1987, 573-587