Proc. of the 5th Int'l Conf. on Extending Database Technology (EDBT), March 1996, pages 388-402

Fine-granularity Locking and Client-Based Logging for Distributed Architectures

E. Panagos[1,2] and A. Biliris[2] and H.V. Jagadish[2] and R. Rastogi[2]

[1] Boston University
Computer Science Department
Boston, MA 02215
[2] AT&T Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974

Abstract. We present algorithms for fine-granularity locking and client-based logging where all transactional facilities in a distributed client-server architecture are provided locally. Multiple clients are allowed to concurrently modify different objects on the same page without synchronizing their updates. Each client has its own log disk where all log records for updates to locally cached data are written. Transaction rollback and client crash recovery are handled exclusively by the clients and local logs are not merged at any time. Clients can take checkpoints independently, and client clocks do not have to be synchronized.

1 Introduction

The proliferation of inexpensive workstations and networks has created a new era in distributed computing. At the same time, non-traditional applications, such as computer aided design (CAD), computer aided software engineering (CASE), geographic information systems (GIS), and office information systems (OIS), have placed increased demands for high-performance transaction processing on database systems. The combination of these factors creates significant performance opportunities in the design of modern client-server database systems.

Allowing clients to cache portions of the database in their local memory and retain their caches across transaction boundaries has been proven to be an effective way to reduce dependence on the server [26, 25, 4, 8]. However, existing client-server database systems do not adequately exploit client disk space. In particular, clients are not allowed to offer transactional facilities locally. The high cost of main memory and disks in the past made it more cost effective to increase the resources of the server rather than the resources of each client. In addition, client machines are not considered to have the same reliability and availability characteristics as server machines. This is because client machines may be connected to or disconnected from the network, or they may simply be turned off at arbitrary times.

Today, advances in hardware and software have resulted in both reliable network connections and reliable workstations that approach server machines regarding resources. Thus, client reliability concerns become less and less important. Concerns related to availability are more a function of the computing environment rather than of the technology. In many computing environments, such as corporate, engineering, computer aided design and manufacturing, and software development, client workstations are connected to the server(s) all the time. Of course, disconnection of these machines from the network for some reason does happen sporadically, but it can be handled in an orderly fashion. In such environments, allowing clients to offer transactional facilities locally will further reduce dependence on server resources and, thus, additional performance and scalability benefits will be gained.

In our previous work [20], we developed recovery algorithms for distributed data shipping architectures. These algorithms offer transactional facilities locally and use page-level locking. In this paper, we describe how local transaction management is carried out in a data shipping client-server architecture when fine-granularity locking is used. Our algorithms allow multiple clients to concurrently update different portions of the same database page. Each client writes log records to a local log file and logs are not merged at any time. The database state is recovered correctly even if the server and several clients crash at the same time, and if the updates performed by different clients on a page are not present on the disk version of the page, even though some of the updating transactions have committed.

The remainder of the paper is organized as follows. Section 2 states our assumptions about the distributed environment we assume when describing our algorithms. Section 3 describes our solutions to the problems associated with fine-granularity locking, and it presents our algorithms for client, server, and complex failures. We compare our work with relevant work that appears in the literature in Section 4 and, finally, we summarize in Section 5.

2 Terminology and Basic Assumptions

Transactions are executed in their entirety at the client where they are started. Data items referenced by a transaction are fetched from the server before they are accessed, and the unit of transfer is assumed to be a database page (this corresponds to the page server approach described in [6]). The buffer managers of the clients and the server follow the *steal* and *no-force* strategies [10]. Modified pages that are replaced from a client cache are sent to the server, and modified pages that are replaced from the server cache are written *in-place* to disk. Pages that were updated by a terminated transaction (committed or aborted) are not necessarily sent to the server before the termination of the transaction.

Concurrency control is based on locking and the server has a global lock manager (GLM) that employs the strict two-phase locking protocol. Each client has a local lock manager (LLM) that caches all acquired locks and forwards the lock requests that cannot be granted locally to the server. *Inter-transaction* caching [26, 25, 4, 8] of the cached pages and locks is supported, and the *callback locking* protocol [11, 13] is used for cache consistency. Fine-granularity locking is employed, and both shared and exclusive locks are retained by a client after a transaction terminates (whether committing or rolling back). For concreteness, we assume that the adaptive locking scheme presented in [3] is used. Cached locks that are called back in exclusive mode are released and exclusive locks that are called back in shared mode are demoted to shared.

Each database page consists of a header that among other information contains a *page sequence number* (PSN), which is incremented by one every time the page is modified by a transaction. The server initializes the PSN value of a page when this page is allocated by following the approach presented in [18] (i.e. the PSN stored on the space allocation map containing information about the page in question is assigned to the PSN field of the page).

The private log of each client is used for logging transaction updates, rolling back aborted transactions, and recovering from client crashes. Recovery is based on the *write-ahead log* (WAL) protocol and the ARIES [16] algorithm is employed. Log records are written to the private log before an updated page is replaced from the local cache and at transaction commit. Each client log manager associates with each log record a *log sequence number* (LSN), which is a monotonically increasing value. We assume that the LSN of a log record corresponds to the address of the log record in the private log file. Log records describing an update on a page contain among other fields the page id and the PSN the page had just before it was updated.

When the server receives a page P from some client, it merges the updates present on that page with the version of P that is present in its buffer pool. If there is no copy of P in its buffer pool, the server reads P from the disk first and then it applies the merging procedure. After the server merges two copies of the same page having PSN values PSN_i and PSN_j, respectively, it sets the PSN value of the page to be: $\max(PSN_i, PSN_j) + 1$. We add one to the maximum value to ensure monotonically increasing PSN values when two copies with the same PSN value are merged.

When a client triggers a callback for an object and the server sends the page P containing the object, the client installs the updates present on this object on the version of P that is present in its cache, if any. Similarly to the server merging procedure, the client sets the PSN of the page to be one greater than the maximum of the PSN values present on the two copies that are being merged. In this way, log records written for the same object by different clients contain monotonically increasing PSN values.

While presenting our algorithms, we assume that each client in the system writes log records for updates to pages in its own log file. We also assume that the crashed client performs restart recovery. However, our algorithms do not require that each client has a log file, nor do they require that the crashed client is the one that will recover from its failure. In particular, clients that do not have local disk space can ship their log records to the server. In addition, restart recovery for a crashed client may be performed by the server or any other client that has access to the log of this client.

3 Client-Based Logging

In this section, we first present our solutions to the recovery issues which arise when fine-granularity locking is used. Then, we present our algorithms for client, server, and complex crashes and, finally, we address issues related to log space management.

3.1 Recovery Issues in Fine-Granularity Locking

Using fine-granularity locking in a page server architecture raises the issue of managing concurrent client updates to the same pages. The *update-privilege* approach has been suggested for both shared-disks systems [17] and client-server architectures [18]. The idea is to serialize the updates by using an "update token," which is acquired before updating a page. However, this approach tends to be communication intensive due to the synchronization messages required for transferring the token and the pages that are often sent along with the transfer of the token.

A second approach is to permit multiple outstanding updates on a page and merge these updates when necessary. One way of merging these updates is by merging the log records generated for them [19, 2]. However, merging log records is an expensive and I/O intensive operation. An alternative solution is to merge the updated copies of a page. This solution involves CPU cost and usually no server disk I/O, and the price paid is only a little more book-keeping. This approach fits better with the cost parameters of a networked client-server system, and is the approach we follow here.

Merging updates that simply overwrite parts of objects residing on the same page, referred to as *mergeable* updates, is straightforward. However, updates that modify the structure of a page (by either changing the size of an object or creating new objects) cannot always be merged[3]. Consequently, only one client at a time should be allowed to perform non-mergeable updates. This is accomplished by acquiring an exclusive lock on the page whose structure is going to be altered.

Another challenge in fine-granularity locking is to be able to determine whether the updates of a log record are present on the page. When page-level locking is employed or when the update token approach is used, only one client at a time can update the PSN value of a page and write a log record containing this value. As a result, the PSN value of a page is enough to determine whether the updates

[3] Object size modifications could be made mergeable by either using some forwarding mechanism or reserving in advance enough space to accommodate any future expansions of the object. We do not consider these alternatives any further in this paper.

of a log record are present on the page. According to the assumptions made in Section 2, the updates of a log record are present on a page when the PSN value of the page is greater than the PSN value stored in the log record.

However, when a page is updated concurrently by many clients, then some log records written for this page by these clients may contain the same PSN value. Consequently, the PSN value of the page cannot be used to determine the log records that have their updates reflected on the page. Our solution to this problem consists of two parts. For handling client crashes, when a client sends a page to the server, either because of cache replacement or in response to a callback request, the server remembers the PSN value present on the page. In addition, the server remembers the PSN value present on the page the first time a client acquires an exclusive lock on the page or an object present on the page. As a result of the above technique, the following propery is guaranteed.

Property 1. The updates of a client record written for a page P are reflected on the copy of P present in the server's cache or on disk, when the PSN value stored in it is less than the PSN value the server remembers for P and this client.

For handling server crashes, the server forces to its log a *replacement* log record when it is about to write an updated page to disk. This log record contains the PSN value of the page and the list of the PSN values the server remembers for the clients that have updated the page, together with the ids of these clients. It can be easily proven that this solution has the following property.

Property 2. If the PSN value of a page P on disk is PSN_{disk} and the server's log contains a replacement log record for P whose PSN field is the same as PSN_{disk}, the PSN values stored in this log record determine the client updates that are present on the page. In particular, the updates of a client log record whose PSN value is less than the PSN value stored in the replacement log record for this client are present on the page.

Finally, because the same object may be updated by several clients before the page containing this object is written to disk, restart recovery must preserve the order in which these clients updated the object. This order corresponds to the order in which the server sent callback messages for the object, and it should be reconstructed during server restart recovery (Property 1 guarantees correct recovery in the case of a client crash). In order to be able to reconstruct the callback order, each client that triggers a callback for an exclusive lock writes a *callback* log record in its log. This log record contains the identity of the called back object, the identity of the client that responded to the callback, and the PSN value the page had when it was sent to the server by the client that responded to the callback request. Section 3.4 explains how the callback log records are used during server restart recovery.

3.2 Normal Processing

When the server receives a lock request for an object that conflicts with an existing lock on the same object or the page P conatining the object, it examines the following cases.

- *Object-level conflict.* If the requested lock mode is shared and some client C holds an exclusive lock on the object, C downgrades its lock to shared and sends a copy of P to the server, which forwards P to the requester. The same procedure is followed when the requested lock is exclusive. In this case, all clients holding conflicting locks release them, and they drop P from their cache if no other locks are held on objects residing on the page.
- *Page-level conflict.* All clients holding conflicting locks on P de-escalate their locks and obtain object-level locks; each LLM maintains a list of the objects accessed by local transactions, and this list is used in order to obtain object-level locks. After de-escalation is over, the server checks for object-level conflicts.

Clients periodically take checkpoints. Each checkpoint record contains information about the local transactions that are active at the time of the checkpointing. The checkpoint record also contains the dirty page table (DPT), which consists of entries corresponding to pages that have been modified by local transactions and the updates have not made it to the disk version of the database yet. Each entry in the DPT of a client C contains at least the following fields.

PID:	id of a page P
RedoLSN:	LSN of the log record that made P dirty

A client adds an entry for a page to its DPT the first time it obtains an exclusive lock on either an object residing on the page or the page itself. The current end of the log is conservatively assigned to the *RedoLSN* field. The *RedoLSN* corresponds to the LSN of the earliest log record that needs to be redone for the page during restart recovery. An entry is dropped from the DPT when the client receives an acknowledgment from the server that the page has been flushed to disk and the page has not been updated again since the last time it was sent to the server.

The server also takes checkpoints. Each checkpoint record contains the dirty client table (DCT), which consists of entries corresponding to pages that may have been updated by some client. Each entry in the DCT has at least the following fields.

PID:	id of a page P
CID:	id of client C
PSN:	P's PSN the last time it was received from C
RedoLSN:	LSN of the first replacement log record written for P

The server inserts a new entry into the DCT the first time it grants an exclusive lock requested by a client on either an object residing on the page or the page itself. The new entry contains the id of the client, the id of the page, the PSN value present on the page[4], and the *RedoLSN* field is set to NULL.

[4] If the client has the page cached in its local pool, the client sends the PSN value of the page when it requests an exclusive lock on an object residing on the page. It is

The server removes an entry for a particular client and page from the DCT after the page is forced to disk, and the client does not hold any exclusive locks on wither objects residing on the page or the page itself.

Every time the server forces a page P to disk, it first writes a replacement log record to its log file. The replacement log record contains the PSN value stored on the page and all the DCT entries about P. If the *RedoLSN* field of the DCT entry about P is NULL, the LSN of the replacement log record is assigned to it.

When the server receives a page P that was either called back or replaced from the cache of a client C, it first locates the entry in the DCT that corresponds to C and P and sets the value of the *PSN* field to be the PSN value present on P. Next, the server merges the updates present on P, as explained in Section 2.

Transaction rollback is handled by each client. Furthermore, clients can support the savepoint concept and offer partial rollbacks. Both total and partial transaction rollbacks open a log scan starting from the last log record written by the transaction. Since updated pages are allowed to be replaced from the client's cache, the rollback procedure may have to fetch some of the affected pages from the server. When a client needs to access again a page that was replaced from its local cache, the server sends the page to the client together with the *PSN* value present in the DCT entry that corresponds to this client and the page in question. The client ignores the PSN value sent along during normal transaction processing.

3.3 Recovery From a Client Crash

When a client fails, its lock tables and cache contents are lost. The server releases all shared locks held by the crashed client and queues any callback requests until the client recovers. Transaction processing on the remaining clients can continue in parallel with the recovery of the crashed client.

During restart recovery, the crashed client installs in its lock tables the exclusive locks it held before the failure. The recovery of the crashed client involves the recovery of the updates performed by local transactions. Since each client writes all log records for updates to pages in its own log file, all the pages that had been updated before the crash can be determined by scanning the local log starting from the last complete checkpoint. These pages correspond to the entries of the DPT which is constructed during the analysis phase of the ARIES algorithm. However, according to Property 1 and the way the DCT is updated, only the pages that have an entry in the DCT need to be recovered.

Next, the client executes the ARIES redo pass of its log by starting from the log record whose LSN is the minimum of all *RedoLSN* values present in the entries of the DPT. A page that is referenced by a log record is fetched from the server only if the page has an entry in the DPT and the *RedoLSN* value of the DPT entry for this page is smaller than or equal to the LSN of the log record. When the page is fetched from the server, the server sends along the PSN value

that PSN that will be inserted in the DCT entry. Otherwise, the PSN value present on the page that is sent to the client is assigned to the new DCT entry.

stored in the DCT entry that corresponds to this client and the client installs this PSN value on the page. The log record is applied on the page only when it corresponds to an update for an object that is exclusively locked and the PSN field of this record is greater that or equal to the PSN value stored on the page.

During the redo pass, callback log records may be encountered. These callback log records are not processed, according to the discussion presented in Section 3.1. After the redo pass is over, all transactions that were active at the time of the crash are rolled back by using transaction information that was collected during the ARIES analysis pass. Transaction rollback is done by executing the ARIES undo pass.

3.4 Recovery From a Server Crash

When the server crashes, pages containing updated objects that were present in the server cache at the time of the crash may have to be recovered. These pages may contain objects that were updated by multiple clients since pages are not forced to disk at transaction commit, or when they are replaced from the client cache, or when they are called back. During its restart recovery, the server has to (a) determine the pages requiring recovery, (b) identify the clients that are involved in the recovery of these pages, (c) reconstruct the DCT, and (d) coordinate the recovery among the involved clients.

The pages that may need to be recovered are those that have an entry in the DPT of a client and they are not present in the cache of this client. Although some of these pages may be present in the cache of some other client, it is wrong to assume that these pages contain all the updates performed on them before the server's crash. This is because fine-granularity locking is in effect. The server constructs the list of the pages that may require recovery, as well as the GLM tables, by requesting from each client a copy of the DPT, the list of the cached pages, and the entries in the LLM tables.

The clients that are involved in the recovery are identified during the procedure of determining the pages that require recovery. In particular, all clients that have an entry for a page in their DPTs and they do not have the page cached in their cache will participate in the recovery of the page.

Next, the server reconstructs its DCT. The construction of the DCT must be done in such a way that the state of a page with respect to the updates performed on this page by a client can be precisely determined. When a page is present in the cache of a client its state corresponds to the PSN value present on the page. When a page is not present in the cache of a client its state must be determined from the state of the page on disk and the replacement log records written for this page. In particular, the server executes the following steps.

1. Insert into the DCT entries of the form $< PID, CID, NULL, NULL >$ for all the pages that are present in the DPTs of the operational clients.
2. Read from disk all the pages that were determined to be candidates for recovery and remember the PSN values stored on them.
3. Update the NULL PSN and $RedoLSN$ entries in the constructed DCT in the following way:

(a) Retrieve from the log the DCT stored in the last complete checkpoint and compute the minimum of the $RedoLSN$ values stored in this table.

(b) Scan the log starting from the above computed minimum and for each *replacement* log record that corresponds to a page P having an entry in the constructed DCT do the following:

   i. If the $RedoLSN$ value of the DCT entry for P is NULL then set its value to the LSN of this log record.

   ii. If the PSN value stored in the log record is the same as the remembered PSN value computed in Step 2, then replace the PSN fields of all entries in the DCT that correspond to the client ids stored in the log record with the corresponding PSN values present in the log record.

4. Request from each operational client the pages that are present in its cache and have an entry in the DPT of this client. The updates present on these pages are merged and the PSN fields in the DCT are updated accordingly.

Finally, the server coordinates the recovery of a page P by determining for each involved client C the state of each object residing on P which had been updated by many clients before the crash. This is done in the following way.

1. Each client C_i that has P in its cache scans its log and constructs a list, referred to as $CallBack_P$, of all the objects residing on P that were called back from C. The scan starts from the location corresponding to the $RedoLSN$ value present in the DPT entry about P. $CallBack_P$ contains the object identifiers and the PSN values present in the callback log records written for these objects and the client C. If multiple callback log records are written for the same object and the same client, the PSN value stored in the most recent one is stored in $CallBack_P$.
2. The server collects all $CallBack_P$ lists and merges all the entries referring to the same object by keeping only the entry containing the maximum PSN value. The resulting list is sent to C together with P and the PSN value present in the DCT entry.

Client C installs on P the PSN value sent by the server and starts its recovery procedure for P by examining all log records written for updates to P. The starting point of the log scan is determined from the $RedoLSN$ value present in the DPT entry for P. For each scanned log record, C does the following.

1. If the log record was written for an object belonging to the $CallBack_P$ list sent by the server, the log record applied to P only when the PSN value stored in it is equal to or greater than the object's PSN value present in the above list.
2. If the log record was written for an object that does not belong to the $CallBack_P$ list, then the log record is applied to P.
3. If the log record is a callback log record that was written for an object present in the $CallBack_P$ list, the log record is skipped. Otherwise, C requests P from the server and sends the CID and PSN values present in the log record along. C continues the recovery procedure after the server sends P and C merges the updates present on it with the copy it has in its cache.

When the server receives the request for page P from C in the above Step 3, it compares the PSN value sent against the PSN values stored in the DCT for the client CID. If the latter is greater or equal to the former, then the server will send P to C. Otherwise, the server will request P from CID and then forward P to C. This situation materializes when CID is recovering P in parallel with C. In this case, CID will send P to the server only after it has processed all log records containing a PSN value that is less than the PSN value C sent to the server.

3.5 Recovery From a Complex Crash

So far, we have presented our recovery algorithms for the case of a single client or server crash. However, the server may crash while a client is in the process of recovering from its earlier failure. Similarly, a client may crash while the server is in the process of recovering from its earlier failure. In this case, operational clients will recover their updates on the pages that were present in the server's buffer pool during the crash in the same way as in the server-only crash case. Crashed clients will recover their updates in a way similar to the client-only crash case. In particular, each crashed client will scan its local log starting from the last complete checkpoint and build an augmented DPT. The server will scan its log file, starting from the minimum *RedoLSN* value present in the DCT stored in the last checkpoint record, and build the DCT entries that correspond to both the pages the crashed clients updated and the pages the operational clients had replaced. From the replacement log records and the PSN value present on each of these pages, the server will calculate the PSN value to be used while recovering those pages in the way explained in Section 3.4.

3.6 Log Space Management

Log space management becomes an issue when a client consumes its available log space and it has to overwrite existing log records. Since the earliest log record needed for recovering from a client and/or server crash corresponds to the minimum of all the *RedoLSN* values present in the DPT of this client, the client can reuse its log space only when the minimum *RedoLSN* is pushed forward. In the algorithms we have presented so far, the minimum *RedoLSN* may be pushed forward only when an entry is dropped from the DPT. But, this may not be enough to prevent the client from not having enough log space to continue executing transactions.

Our solution to the above problem is the following. When a client sends a dirty page that is replaced from its local cache to the server, the client remembers the current end of its private log file. When the server forces the page to disk, it informs all clients that had replaced the page. These clients replace the *RedoLSN* field of the DPT entry referring to the page that was forced to disk with the remembered end of the log LSN for the page. When a client faces log space problems, it replaces from its cache the page having the minimum *RedoLSN* value in the DPT and asks the server to force the page to disk. If, however, the page is not present in the client cache, the client just asks the server to force the page to disk. If the client needs more log space, it repeats the above procedure.

4 Related Work

In the following sections, we compare our work sith relevant work in the areas of client-server, shared-disks, and distributed file systems.

4.1 Client-Server Systems

A comprehensive study of performance implications related to different granularities for data transfer, concurrency control, and coherency control in a client-server environment is presented in [3]. Unlike our scheme, the authors of the above study assumed that copies of all updated data are sent back to the server at transaction commit. While concurrent updates on the same page are handled by merging individual updates, no recovery algorithms are presented in [3].

In [7], a client-server architecture that exploits client disks for extending the in-memory cache is proposed. Even though that paper's main focus is on the performance of the different ways of integrating disks with memory caches, the authors also discuss issues related to recovery when pages modified by a committed transaction are not sent to the server as part of the commit protocol. In contrast to [7], we have investigated logging and recovery issues when clients are allowed to perform local logging and they do not send pages to the server at transaction commit.

Local disk space is also used in the architecture presented in [5]. In [5], local disks are used to store relational query results that are retrieved from the server. Transaction management is carried out exclusively by the server and all updates to the database are performed at the server. Our work differs significantly in that we permit clients with local disk space to offer transactional facilities to local transactions.

Versant [24], a commercially available OODBMS, also explores client disk space. In Versant, users can check out objects by requesting them from the server and store them locally in a "personal database." In addition, locking and logging for objects stored in a personal database can be turned off to increase performance. The checked out objects are unavailable to the rest of the clients until they are checked in later on. All modified and new objects in the client's object cache must be sent to the appropriate server so that changes can be logged at transaction commit. Our architecture is more effective since it avoids generating log records at commit time, and it allows local logging of updates.

In ARIES/CSA [18], clients send all their log records to the server as part of the commit processing. Similar to our work, ARIES/CSA employs a fine-granularity locking protocol, clients do not send modified pages to the server at transaction commit, and transaction rollback is performed by clients. However, client crashes are still handled by the server and clients are not allowed to update the same page simultaneously. Unlike our algorithms, client checkpoints in ARIES/CSA are stored in the log maintained by the server and server checkpointing requires synchronous communication with all connected clients.

4.2 Shared-Disks Systems

In [21], logging and recovery protocols are presented for a shared-disks architecture employing the *primary copy authority* (PCA) locking protocol. Under the PCA locking protocol, the entire lock space is divided among the participating nodes and a lock request for a given item is forwarded to the node responsible for the item. Although PCA is similar to our work, there are several important differences. Unlike PCA that supports only physical logging, our algorithms support both physical and logical logging. PCA employs the *no-steal* buffer management policy – only pages containing committed data are written to disk – which is argued to be an inflexible and expensive policy, especially when fine-granularity locking is used [16].

Like our algorithms, PCA allows pages to be modified by many nodes before they are written to disk. However, PCA uses page level locking and commit processing involves the sending of each updated page to the node that holds the PCA for the page. Furthermore, double logging is required for every page that is modified by a node other than the PCA node. During normal transaction processing, the modifying node writes log records in its own log and at transaction commit it sends all the log records for remotely controlled pages to the PCA nodes responsible for these pages. Our algorithms do not require updated pages to be sent to the owner nodes at transaction commit time, nor do they require log records to be written in two log files.

In [17] four different recovery schemes for a shared-disks architecture are presented and analyzed. The presented algorithms were designed to exploit the fast inter-node communication paths usually found in tightly-coupled data sharing architectures. Similar to these algorithms, our algorithms work with write-ahead logging recovery, the steal no-force buffer replacement policy, and fine-granularity locking. But, unlike these algorithms, which prohibit different nodes from updating the same page concurrently, our algorithms allow multiple nodes to update different parts of the same page in parallel. Further, our algorithms are not based on the assumption that the clocks of all the clients are perfectly synchronized. Finally, our algorithms do not force pages to disk when they are exchanged between nodes as it is done in the *simple* and *medium* schemes presented in [17], nor do they require merging of the private logs at any time. Private logs have to merged in the *fast* and *super-fast* schemes presented in [17] even in the case where only a single node crashes.

The *shared data/private log* recovery algorithm presented in [14] motivated our work. However, our recovery algorithms do not require a seamless ordering of PSNs, nor do they associate for each database page extra information with the space management sub-system. Unlike the algorithm presented in [14], which requires modified pages to be forced to disk before they are replaced from a node's cache, our algorithms let the cache replacement policy force a modified page to disk. In addition, our algorithms work with fine-granularity locking.

Rdb/VMS [22] is a data sharing database system executing on a VAXcluster. Earlier versions of Rdb/VMS employed an undo/no-redo recovery protocol that required, at transaction commit, the forcing to disk of all the pages updated by the committing transaction. More recent versions offer both an undo/no-redo and an undo/redo recovery scheme [15]. In addition, a variation of the callback locking algorithm, referred to as *lock carry-over*, is used for reducing the number of messages sent across the nodes for locking purposes. However, Rdb/VMS does not allow multiple outstanding updates belonging to different nodes to be present on a database page. Thus, modified pages are forced to disk before they are shipped from one node to another.

In Rdb/VMS, each application process can take its own checkpoint after the completion of a particular transaction. The checkpointing process forces to disk all modified and committed database pages. Unlike Rdb/VMS, our algorithms support different variations of *fuzzy checkpoints* [1, 9, 12]. Those checkpoints are asynchronous and take place while other processing is going on. Another important difference is that Rdb/VMS uses only one global log file. Consequently, the common log becomes a bottleneck and a global lock must be acquired by each node that needs to append several log records to the log.

4.3 Distributed File Systems

Coda [23] is a distributed file system operating on a network of UNIX workstations. Coda is based on the Andrew File System [11] and cache coherency is based on the callback locking algorithm. However, the granularity of caching is that of entire files and directories. Coda's most important characteristic, which is closely related to our work, is that it can handle server and network failures and support portable workstations by using clients disks for logging. This ability is based on the *disconnected operation* mode of operation that allows clients to continue accessing and modifying the cached data even when they are not connected to the network. All updates are logged and they are reintegrated to the systems on reconnection.

However, Coda does not provide the same transactional semantics as our algorithms. In particular, failure atomicity is not supported and updates cannot be rolled back. Another important difference is that our algorithms guarantee that the updates performed by a transaction survive various system failures and they are altered only when a later transaction modifies them. Coda, on the other hand, guarantees permanence conditionally; updates made by a transaction may change if a conflict is discovered at the time these updates are being reintegrated into the system.

5 Conclusions

In this paper we have presented recovery algorithms that exploit client disk space to offer transactional facilities locally, while maintaining the transaction semantics associated with traditional database systems. The algorithms we have described work with fine-granularity locking and multiple clients can update concurrently different portions of the same page. The database state is recovered correctly even if several clients and the server crash, and even if the updates performed by different clients on a page have not been merged, even though some of the updating transactions have committed.

The key advantages of our algorithms are: (1) updated pages are not forced to disk at transaction commit time or when they are replaced from a client cache, (2) transaction rollback and client crash recovery are handled exclusively by the clients, (3) clients may recover the same page in parallel, (4) for objects that were updated by several clients, the synchronization log records preserve the ordering of updates, (5) private log files are never merged during the recovery process, (6) each client can take a checkpoint without synchronizing with the rest of the operational clients, and (7) client clocks do not have to be synchronized and lock tables are not checkpointed.

We believe that as the world becomes more and more distributed, it will become increasingly more important for all transaction facilities to be provided locally, even when data is shared in a global environment. In this paper, we have taken a step in this direction by showing how local logging, transaction commitment and recovery can be performed in a distributed architecture.

References

1. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.
2. M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. McAuliffe, J. F. Naughton, D. T. Schuh, and M. H. Solomon. Shoring up persistent applications. In *Proceedings of ACM-SIGMOD 1994 International Conference on Management of Data,* Minneapolis, Minnesota, pages 383 – 394, May 1994.
3. M. J. Carey, M. J. Franklin, and M. Zaharioudakis. Fine-grained sharing in a page server OODBMS. In *Proceedings of ACM-SIGMOD 1994 International Conference on Management of Data,* Minneapolis, Minnesota, pages 359–370, May 1994.
4. M.J. Carey, M. Franklin, M. Livny, and E. Shekita. Data caching tradeoffs in client-server DBMS architectures. In *Proceedings of ACM-SIGMOD 1991 International Conference on Management of Data,* Denver, Colorado, pages 357–366, May 1991.
5. A. Delis and N. Roussopoulos. Performance and scalability of client-server database architectures. In *Proceedings of the Eighteenth International Conference on Very Large Databases,* Vancouver, British Columbia, pages 610–623, August 1992.
6. D. J. DeWitt, D. Maier, P. Futtersack, and F. Velez. A study of three alternative workstation-server architectures for object-oriented database systems. In *Proceedings of the Sixteenth International Conference on Very Large Databases,* Brisbane, pages 107–121, August 1990.
7. M. Franklin, M. Carey, and M. Livny. Local disk caching for client-server database systems. In *Proceedings of the Nineteenth International Conference on Very Large Databases,* Dublin, Ireland, pages 641–654, August 1993.
8. M. Franklin, M. Carey, and Livny M. Global memory management in client-server DBMS architectures. In *Proceedings of the Eighteenth International Conference on Very Large Databases,* Vancouver, British Columbia, pages 596–609, August 1992.

9. M. Franklin, M. Zwilling, C. Tan, M. Carey, and D. DeWitt. Crash recovery in client-server EXODUS. In *Proceedings of ACM-SIGMOD 1992 International Conference on Management of Data,* San Diego, California, pages 165 – 174, June 1992.

10. T. Haerder and A. Reuter. Principles of transaction oriented database recovery — a taxonomy. *ACM Computing Surveys*, 15(4):289–317, December 1983.

11. J. H. Howard, M. Kazarand, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

12. H.V Jagadish, D.F. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. DALI: An extensible main memory storage manager. In *Proceedings of the 20th International Conference on Very Large Databases,* Santiago, Chile, September 1994.

13. C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):51–63, October 1991.

14. D. Lomet. Recovery for Shared Disk Systems Using Multiple Redo Logs. Technical Report CLR 90/4, Digital Equipment Corp., Cambridge Research Lab, Cambridge, MA., Oct. 1990.

15. D. Lomet, R. Anderson, T.K. Rengarajan, and P. Spiro. How the Rdb/VMS data sharing system became fast. Technical Report CRL 92/4, Digital Equipment Corporation Cambridge Research Lab, 1992.

16. C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

17. C. Mohan and I. Narang. Recovery and coherency-control protocols for fast intersystem page transfer and fine-granularity locking in a shared disks transaction environment. In *Proceedings of the Seventeenth International Conference on Very Large Databases,* Barcelona, Spain, pages 193–207, September 1991.

18. C. Mohan and I. Narang. ARIES/CSA: a method for database recovery in client-server architectures. *Proceedings of ACM-SIGMOD 1994 International Conference on Management of Data,* Minneapolis, Minnesota, pages 55–66, May 1994.

19. C. Mohan, I. Narang, and S. Silen. Solutions to Hot Spot Problems in a Shared Disks Transaction Environment. In *Proceedings of the 4th International Workshop on High Performance Transaction Systems*, September 1991. Also, in IBM Research Report RJ8281 (August 1991).

20. E. Panagos, A. Biliris, H.V. Jagadish, and R. Rastogi. Client-based logging for high performance distributed architectures. In *Proceedings of the 12th International Conference on Data Engineering,* New Orleans, Louisiana, February 1996. To appear.

21. E. Rahm. Recovery Concepts for Data Sharing Systems. In *Proc. 21st Int. Conf. on Fault-Tolerant Computing*, Montreal, June 1991.

22. T. Rengarajan, P. Spiro, and W. Wright. High availability machanisms of VAX DBMS software. *Digital Technical Journal 8*, pages 88–98, February 1989.

23. M. Satyanarayanan, K.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4), April 1990.

24. Versant Object Technology, Menlo Park, California. *VERSTANT System Reference Manual, Release 1.6*, 1991.

25. Y. Wang and L. A. Rowe. Cache consistency and concurrency control in client/server DBMS architecture. In *Proceedings of ACM-SIGMOD 1991 International Conference on Management of Data,* Denver, Colorado, pages 367–376, May 1991.
26. K. Wilkinson and M. A. Neimat. Maintaining consistency of client-cached data. In *Proceedings of the Sixteenth International Conference on Very Large Databases,* Brisbane, pages 122–133, August 1990.