Proc. IEEE 12th Int'l Conference on Data Engineering, February 1996, pages 344-351

Client-Based Logging for High Performance Distributed Architectures

E. Panagos*, A. Biliris, H.V. Jagadish, R. Rastogi
AT&T Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974
{thimios, biliris, jag, rastogi}@allegra.att.com

Abstract

In this paper, we propose logging and recovery algorithms for distributed architectures that use local disk space to provide transactional facilities locally. Each node has its own log file where all log records for updates to locally cached pages are written. Transaction rollback and node crash recovery are handled exclusively by each node and log files are not merged at any time. Our algorithms do not require any form of time synchronization between nodes and nodes can take checkpoints independently of each other. Finally, our algorithms make possible a new paradigm for distributed transaction management that has the potential to exploit all available resources and improve scalability and performance.

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 gives rise to significant performance opportunities in the area of distributed transaction processing.

Today, the major distributed database architectures are *client-server, shared nothing* and *shared disks*. Most of these architectures use logging for recovery. In a client-server system, both the database and the log are stored with the server and all log records generated by the clients are sent to the server. In a shared nothing system, the database is partitioned among several nodes and each node has its own log file. Each database partition is accessed only by the owning node and a distributed commit protocol is required for committing transactions that access multiple partitions. In a shared disks system, the database is shared among the different nodes. Some shared disks systems use only one log file and require system wide synchronization for appending log records to the log (e.g., Rdb/VMS [18]). Some other shared disks systems use a log file per node (e.g., [14], [11]). However, these systems either force pages to disks when these pages are exchanged between two nodes or they merge the log files during a node crash.

1.1 Paper Contributions

This paper proposes a new paradigm for distributed transaction processing. In this new approach, updates on data items performed by a node are logged locally, regardless of whether the data items are stored in a local database or in a database managed by a remote node. Local logging eliminates the need to send log records to remote nodes during transaction execution and at transaction commit.

Figure 1 shows the distributed architecture we assume while presenting our recovery algorithms. The system consists of several networked processing nodes. A node having databases attached to it, such as nodes 1 and 3, is referred to as *owner node* with respect to the items stored in these databases. All owner nodes have local logs. Nodes that do not own any database, such as nodes 2 and 4, may or may not have local logs. Although nodes with no local logs may participate in a distributed computation, our algorithms apply only to nodes that have local logs.

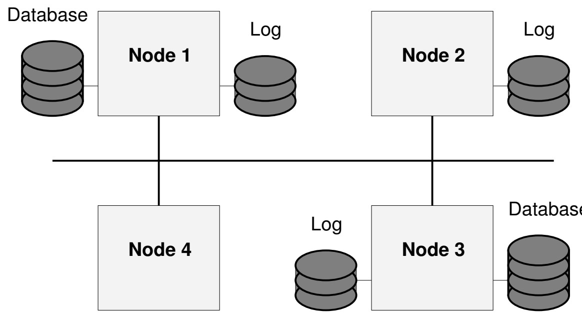

Figure 1: Distributed System Architecture

A user program running on node N accesses data items that are owned by either N or some other remote node. These data items are fetched in N's cache, i.e., we assume a data shipping architecture. Log records for data updated by N are written to the local log file and transaction commitment is carried out by N without communication with the remote nodes. To accomplish this, we have designed algorithms that correctly handle transaction aborts and node crashes, while incurring minimal overhead during normal transaction

*A PhD candidate in the Department of Computer Science at Boston University.

processing. The main characteristics of our algorithms are the following.

- Log records for updates to cached pages are written to the log file of each node.
- Transaction rollback and node crash recovery are handled exclusively by each node.
- Node log files are not merged at any time.
- Node clocks do not have to be synchronized.
- Nodes can take checkpoints independently of each other.

1.2 Applicability to Existing Distributed Architectures

The algorithms presented in this paper are of potential value to a variety of distributed computing environments. They can be applied to both shared nothing and shared disks architectures. They can also be applied to client-server architectures where client disks are used for logging and peer-to-peer architectures (e.g., Shore [3], BeSS [2]).

In existing client-server database systems, transaction management is carried out exclusively by the server. The main argument for not allowing clients to offer transactional facilities is twofold. First, client machines may not be powerful enough to handle such tasks; 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. The second, and more important, argument is data availability and client reliability – client machines could be connected to or disconnected from the network or simply 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 of 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, 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 but it is a rare event (say, once a month) and can be handled in an orderly fashion. In such environments, additional performance and scalability gains are realized when clients offer transactional facilities, because dependencies on server resources are reduced considerably.

Our algorithms can be used even in environments where some nodes are often disconnected from the rest for a long period of time, if node updates need not be available to the rest of the nodes. For instance, consider the vision of hand-held notebook computers carried by every day tradespeople. In a utility company, there may be a repair technician with a hand held notebook going out to a customer's home to attend to a complaint. Customer data is in a database attached to some other node. This data is copied into the hand-held notebook computer and cached there. Now, as the technician notes the status of the repair work, or other data, she may wish to achieve transactional durability guarantees for orders recorded in the notebook computer without repeatedly having to call the server in the central office. This can be accomplished with an algorithm such as the one we describe here where we consider the node to be a mobile machine, and the user chooses to keep the log locally to minimize communication cost and save energy. Clearly, often disconnected mobile nodes are not highly available. However, in the kind of applications described here it is unlikely that data items "checked-out" by a particular technician will need to be referenced by anyone else in the meantime.

The remainder of the paper is organized as follows. In Section 2 we state our assumptions about the distributed environment, we describe our recovery algorithms for both single and multiple node crashes, and we address issues related to log space management. We compare our work with relevant work that appears in the literature in Section 3 and, finally, we summarize in Section 4.

2 Client-Based Logging

2.1 Assumptions

Transactions are executed in their entirety in the node where they are started. Data items referenced by a transaction are fetched from the owner node before they are accessed. The unit of inter-node transfer is assumed to be a database page. Each node has a buffer pool (node cache) where frequently accessed pages are cached to minimize disk I/O and communication with owner nodes. The buffer manager of each node follows the *steal* and *no-force* strategies [7]. Pages containing uncommitted updates that are replaced from the local cache are either written *in-place* to disk or sent to the owner node, depending on whether they belong to the local database. Pages that were updated by a terminated transaction (committed or aborted) are not necessarily written to disk or sent to the owner node before the termination of the transaction.

Concurrency control is based on locking and the strict two-phase locking protocol is used. Each node has a lock manager that caches the acquired locks and forwards the lock requests for data items owned by another node to that node. Each node maintains both the cached pages and the cached locks across transaction boundaries – this is generally referred to in the literature as *inter-transaction* caching [22, 21, 4, 6]. The *callback locking* protocol [8, 10] is used for cache consistency. We also assume that both shared and exclusive locks are retained by the node after a transaction terminates (whether committing or rolling back). 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. The granularity of both locking and callback is assumed to be at the level of a database page.

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 updated. The owner node initializes the PSN value of a page when this page is allocated by following the approach presented in [15] (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 log of each node is used for logging transaction updates, rolling back aborted transactions, and recovering from crashes. Recovery is based on the *write-ahead log* (WAL) protocol and the ARIES redo-undo algorithm [13] is employed. Log records are written to the local log before an updated page is replaced from the node cache and at transaction commit. Each node log manager associates with each log record a *log sequence number* (LSN) that corresponds to the address of the log record in the local 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.

2.2 Normal Processing

When a node wishes to read a page owned by another node and not present in its cache, it sends a request for the page to the owner node. If no other node holds an exclusive lock on the page, the owner node grants the lock and sends a copy of the page to the requester. If some other node holds an exclusive lock on the page, the owner node sends a callback message to that node and waits until that node (a) downgrades/releases its lock and (b) sends the copy of the page present in its buffer pool, if any. Then, the owner node grants the lock and sends the page to the requester.

When a node wants to update a page on which it does not hold an exclusive lock, the node requests an exclusive lock from the owner node. The owner node grants the lock immediately when the page is not locked by any other node. If the page is locked by some nodes, then the owner node sends a callback message to these nodes. Once the owner node receives the acknowledgments to all callback requests, it grants the exclusive lock and sends a copy of the page to the requester, if the requester does not have the page cached in its cache.

Nodes periodically take checkpoints. Each checkpoint record contains the dirty page table (DPT) and information about the transactions that were active at the time of the checkpointing. The DPT contains entries which correspond to pages that have been modified by local transactions and the updates are not present in the disk version of the database. An entry in the DPT of a node N for a page P contains at least the following fields.

PID:	P's page id
PSN:	P's PSN the first time P was updated.
CurrPSN:	P's PSN the last time P was updated.
RedoLSN:	LSN of the log record that made P dirty.

An entry for a page P is added to the DPT of N when N obtains an exclusive lock on P and no entry for this page already exists in the table. The PSN of P is assigned to the PSN and $CurrPSN$ fields and the current end of the local 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 a page during restart recovery. Every time P is updated by a local transaction, the $CurrPSN$ value of the DPT entry is set to the PSN value of P after the update.

An entry corresponding to a page owned by N is removed from N's DPT when the page is forced to disk. An entry corresponding to a page owned by a remote node is dropped from N's DPT when N receives an acknowledgment from the owner node that the page has been flushed to disk, and the page has not been updated again after the last time it was replaced from the local cache. Dropping an entry for an updated page that is present in the local cache could result in incorrect recovery if N were to crash after taking a checkpoint. This is because the DPT stored in the checkpoint record would not contain an entry for this page.

Transaction rollback is handled by each node. Furthermore, nodes 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 node's cache, the rollback procedure may have to fetch some of the affected pages from the owner nodes.

2.3 Single Node Crash Recovery

When a node fails, its lock table and cache contents are lost. As a consequence, any further lock and data requests with respect to the data owned by the failed node are stopped until the node recovers. However, transaction processing on the remaining nodes can continue in parallel with the recovery of the crashed node.

The recovery of a crashed node involves the recovery of updates performed by locally executed transactions. In addition, the recovery of a crashed node may involve the recovery of updates performed by transactions that were executed in another node, referred to as remote transactions. This is because updated pages that are replaced from a node's cache are sent to the owner node. If the failed node does not own any data, the recovery of remote transactions is not required (For instance, in a client-server environment, the crash of a client does not involve the recovery of transactions that were executed in another client or the server). During its recovery, the crashed node has to (a) determine the pages that may require recovery, (b) identify the nodes involved in the recovery, (c) reconstruct lock information, and (d) coordinate the recovery among the involved nodes.

In the following sections we present our solutions to the above problems. While presenting our solutions, we assume that recovery is carried out by the crashed node when this node restarts. Nevertheless, our algorithms allow any node that has access to the database and the log file of the crashed node to perform crash recovery. This is realized in shared disks architectures where all nodes have access to the same database and all log files, as well as in shared nothing and client-server architectures that use hot standby nodes.

2.3.1 Determining the Pages that may Require Recovery

When a node fails, all dirty pages present in the cache of this node have to be recovered. These pages belong to two categories: pages owned by the crashed node and pages owned by a remote node. While pages belonging to the first category may have been updated by both local and remote transactions, pages in the second category have been updated only by local transactions.

Since each node writes log records for updates to pages in its own log file, the pages that were updated by local transactions can be determined by scanning the local log starting from the last complete checkpoint. These pages correspond to the entries in the DPT that is constructed during the analysis phase of the ARIES algorithm. Among these pages, the candidates for recovery are: (a) pages owned by the crashed node that are not present in the cache of any other node, and (b) pages owned by a remote node that were exclusively locked by the crashed node at the time of the crash.

The basic ARIES algorithm cannot be used to determine all dirty pages that belong to the first category. This is because under ARIES, a page is not considered dirty if it is not included in the DPT logged in the last checkpoint before the crash, and no log records for this page are logged after the checkpoint. There are two reasons that a page owned by a node is not considered dirty when it is present in the node's cache at the time of the crash. The first is that the page was updated only by local transactions and it was forced to disk before the checkpoint was taken. This case does not cause any problems because the page is no longer dirty at this point. The second reason is that the page was updated only by remote transactions after the checkpoint was taken and the page was not included in the logged DPT. In this case, no log records for updates to the page are found in the local log file.

However, according to the way each DPT is updated for pages owned by remote nodes, pages that were updated before the crash will have an entry in at least one DPT of the remaining nodes. Among these pages, the pages that may have to be recovered are only those that are present in the DPT of a node and not present in the cache of any other node. The rest of the pages, which are present in the cache of some node, contain all the updates performed on them before the owner node's crash and they do not require recovery. Thus, when the crashed node N restarts, it requests from each operational node N_r the list of all pages owned by N that are present in N_r's cache, as well as all entries in N_r's DPT that correspond to pages owned by N. After all operational nodes send the above lists to N, N is able to determine the pages that may have to be recovered based on these lists and its own DPT.

But, pages owned by the crashed node that are present in the DPTs of some nodes and the caches of some other nodes may not be recovered at all or recovered incorrectly if a node were to crash after the owner node finishes its restart recovery. Pages that are not in the DPT of the crashed node would not be recovered at all, while pages that are in the DPT would be recovered incorrectly if the disk version of them did not contain all the updates performed by the rest of the nodes in the past. Our solution to this problem is as follows. After the owner node constructs the list of the pages that may have to be recovered, it requests the pages that are present in the cache of a node and have entries in the DPTs of some other nodes from the nodes that have them in their caches. If there are multiple nodes that have the same page in their caches, only one node is notified to send the page.

2.3.2 Identifying the Nodes Involved in the Recovery

The crashed node identifies the nodes that are involved in the recovery of a page during the procedure of identifying the pages that require recovery. These nodes belong to two categories: nodes whose DPT entry for P has a PSN value greater than or equal to P's PSN value on disk, and nodes whose DPT entry for P has a PSN value less than P's PSN value on disk. Nodes in the first group have to recover their committed updates. However, some of the nodes in the second group may not have to recover P at all if their log files do not contain any log record that was written for P and whose PSN value is greater than or equal to P's PSN value. This happens when all the updates these nodes made on P took place before P was forced to disk. Thus, a node whose $CurrPSN$ value in its DPT entry for P is less than or equal to P's PSN value is not involved in the recovery process and it can drop P's entry from its DPT.

2.3.3 Reconstructing Lock Information

Before the crashed node N starts recovering the pages that were identified to require recovery, N has to reconstruct its lock information so that normal transaction processing can continue in parallel with the recovery procedure. The lock information includes all the locks that had been granted to both local and remote transactions. The locks that were granted to remote transactions are present in the lock tables of the nodes where those transactions were executed. In addition, locks that were granted to local transactions for pages owned by remote nodes are also present in the lock tables of the remote nodes.

During restart recovery, each operational node N_r releases all shared locks held by the crashed node. Exclusive locks are retained so that operational nodes are prevented from accessing a page that has not yet been recovered. The list of locks N_r had acquired from the crashed node as well as the list of exclusive locks held by the crashed node are sent to the crashed node. After all the lock lists have been sent, the crashed node can establish its lock tables. In addition, the crashed node needs to acquire exclusive locks for the pages present in its DPT that do not have a lock entry. At this point, all lock tables contain all the needed locks and normal transaction processing can continue.

2.3.4 Coordinating the Recovery Among the Involved Nodes

After the crashed node N identifies both the pages that require recovery and the nodes that will participate in the recovery of these pages, the recovery of each page P has to be done in the correct order. This order corresponds to the order in which transactions that were executed at the involved nodes updated P. Since the granularity of locking is a page, only one node can update P at a time. Hence, the PSN values stored in the log records written for P determine the order of updates. In fact, the PSN value stored in the first log record written for P by each transaction that updated P is enough for determining the order of updates. The construction of the above list of PSN values for each page that requires recovery, referred to as *NodePSNList*, is explained below.

When a remote node N_r receives the list of pages that require recovery from N, it scans its log file starting from the minimum of all *RedoLSN* values belonging to DPT entries for the pages that are included in the above list. The PSN value present in a log record examined during the scan is inserted into the *NodePSNList* when (a) the log record corresponds to an update performed on a page present in the above list, and (b) the transaction that wrote the log record is not the same as the transaction that wrote the log record whose PSN field is the last PSN inserted into the *NodePSNList*, if any. In addition, the location of this log record is remembered and it will be used during the recovery of the page. When the scan is over, N_r sends the *NodePSNList* to N.

In parallel to the above process, N constructs its own *NodePSNList* for the pages that require recovery, in the same way as the one described above[1]. Once all nodes involved in the recovery have sent their *NodePSNLists* to N, N coordinates the recovery of a page P in the following way.

1. Order the nodes involved in the recovery of P in an ascending ordering based on P's PSN values present in the *NodePSNLists* sent, including your own *NodePSNList*. Adjacent entries that correspond to the same node are merged into one entry, whose PSN value is the minimum of the two PSNs.
2. Send P to the node N_r having the minimum PSN entry in the above list. The second minimum PSN value present in the list is also sent to N_r, if any.
3. When N_r sends back P, place P in the buffer pool and remove the entry from the list.
4. Repeat the previous two steps until there are no more entries for P in the list.

When a node receives P from N together with a PSN value, it recovers P by scanning its log starting from either the log record with LSN equal to the *RedoLSN* value present in the DPT entry for P or the log record remembered in the analysis process mentioned above. The node stops the recovery process when it finds a log record written for P whose PSN value is greater than the PSN value N sent along with P, or when the entire log is scanned. In the former case, the node sends P back to N and remembers the current location in the local log. This location will be the starting point for the continuation of the recovery process for P. In the latter case, the node sends only P back to N. If no PSN value was sent along with P, the node stops the recovery process when the entire log is scanned.

During the recovery process, nodes update the DPT entries corresponding to pages that are being recovered. In particular, a node that does not apply any log record to a page drops the entry from its DPT when it does not hold a lock on the page. If the node holds a lock on the page, it sets the *RedoLSN* value of the DPT entry to the current end of the log. The former case is realized when the owner node crashes before acknowledging the writing of the page to disk. The latter case corresponds to the case where all the updates the node performed in the past are present on the disk version of the page and the node has not updated the page since.

2.4 Multiple Node Crash Recovery

So far, we have presented our recovery algorithms for the case of a single node crash. However, a second node may crash while another node is in the process of recovering from its earlier failure. Recovery from multiple node crashes is similar to the recovery from a single node crash, although it is more expensive as more log files have to be examined and processed and the recovery of a crashed node may have to be restarted. Similar to the single node crash, operational nodes may continue accessing the pages they have in their local caches while the rest of the nodes are in the process of recovering.

As in the single node crash case, we have to: (a) determine the pages that may require recovery, (b) identify the nodes that are involved in the recovery, (c) reconstruct the lock information of each crashed node, and (d) coordinate the recovery of a page among the involved nodes. Once we have determined the pages that may require recovery, the nodes that are involved in their recovery, the reconstruction of the lock information, and the coordination of the recovery among the involved nodes is done in the same way as in the single node case. Hence, the rest of this section discusses only the solution to the first problem.

As in the single node case, each crashed node has to recover the pages that had been updated by local

[1] Since N has already scanned its log during the analysis phase of ARIES, we could build part of the *NodePSNList* during this scan. Then, once all pages that require recovery are identified, N scans its log starting from the minimum of all *RedoLSN* values present in the DPT entries for these pages and stopping when the last complete checkpoint is found. A new entry is inserted into the *NodePSNList* when the two conditions mentioned above are true and the transaction that wrote the log record is not the same as the transaction that wrote the log record whose PSN field is the first PSN inserted into the *NodePSNList*, if any.

transactions, as well as the pages that it owns and which had been updated by remote transactions and were present in its cache at the time of the crash. Pages belonging to the first category can be identified from the log records written in the local log file. Unlike the single node crash case, not all pages belonging to the second category can be identified by using only the entries in the DPTs and caches of the operational nodes. The DPTs of the crashed nodes are also needed, for some of these pages may have been updated by several of these nodes.

Although each crashed node lost its DPT during the crash, a superset of each node's DPT can be reconstructed by scanning the node's log file. In particular, each crashed node scans its log by starting from the last complete checkpoint and updates the DPT stored in that checkpoint by inserting new entries for the pages that do not have an entry and are referenced by the examined log records. Once the analysis pass is done, the DPT entries that correspond to pages owned by another node are sent to the owner node. Each operational node also sends the DPT entries that correspond to pages owned by another node and the list of these pages that are present in the local cache to the owner node. The owner node merges all the received entries with the entries it has in its own DPT for the same pages, after removing all entries that correspond to pages cached in an operational node. The resulting list corresponds to the pages this node has to recover. Similar to the single node crash, pages present in the cache of an operational node and the DPT of another node are sent to the owner node.

2.5 Log Space Management

Log space management becomes an issue when a node consumes its available log space and it has to overwrite existing log records. Since the earliest log record needed for recovering from a node crash corresponds to the minimum of all the *RedoLSN* values present in the DPT of this node, the node 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 node from not having enough log space to continue executing transactions.

Our solution to the above problem is the following. When a node replaces a dirty page P from its cache, it remembers the current end of its log. When the owner node forces P to disk, it informs all nodes that had replaced P. These nodes replace the *RedoLSN* field of the DPT entry referring to P with the remembered end of the log LSN for this page. When a node faces log space problems, it replaces from its cache the page having the minimum *RedoLSN* value in the DPT and asks the node owning this page to force the page to disk. If, however, the page is not present in the node's cache, the node just asks the owner node to force the page to disk. If the node needs more log space, it repeats the above procedure. Note that the owner node may be the same as the node that needs to make space in its local log file. In this case, if the page is present in the node's cache, the page is forced to disk. Otherwise, the page is first requested from a node that has it in its cache and then it is written to disk.

3 Related Work

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

3.1 Client-Server Systems

Local disk space is used in the architecture presented in [5]. In that work, 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 transaction management facilities to the local transactions.

Versant [20], 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 all log records at commit time, and allows local transaction management.

In ARIES/CSA [15], clients send all their log records to the server as part of the commit processing. ARIES/CSA employs a fine-granularity concurrency protocol that prevents clients from updating the same page concurently by using the update token approach prersented in [14]. Similar to our schemes, ARIES/CSA 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. Unlike our algorithms, client checkpoints in ARIES/CSA are stored in the log maintained by the server and server checkpointing requires communication with all connected clients.

3.2 Shared-Disks Systems

In [17], 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 that 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 [13].

Like our algorithms, PCA allows pages to be modified by many systems before they are written to disk. However, commit processing involves the sending of each updated page to the node that holds the PCA for that 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 written for remote pages to the PCA nodes responsible for those 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 [14], four different recovery schemes for a shared disks architecture were presented and analyzed. The algorithms were designed to exploit the fast internode communication paths usually found in tightly-coupled data sharing architectures, and they use fine-granularity locking. Similar to [14], our algorithms are based on write-ahead logging and the steal no-force buffer replacement policy. However, we do not assume that the clocks of all the nodes 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 [14], nor do they require merging of the local logs at any time. Private logs have to merged in the *fast* and *super-fast* schemes presented in [14] even in the case where only a single node crashes.

The *shared data/private log* recovery algorithm presented in [11] 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 subsystem. Unlike [11], our algorithms do not force modified pages to disk before they are replaced from a node's cache.

Rdb/VMS [18] 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 [12]. 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]. 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 some log records at the end of the log.

3.3 Distributed File Systems

Coda [19] is a distributed file system operating on a network of UNIX workstations. Coda is based on the Andrew File System [8] and cache coherency is based on the callback locking algorithm. 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* 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.

4 Conclusions

In this paper we have presented a new paradigm for distributed transaction processing which has the potential to exploit every available system resource and improve scalability and performance. In particular, we have presented recovery algorithms that exploit local disk space for offering transactional facilities locally, while maintaining the transaction semantics associated with traditional database systems. Our algorithms are of potential value to a variety of distributed computing environments. They can be applied to both shared nothing and shared disks architectures. They can also be applied to client-server architectures where clients disks are used for logging and peer-to-peer architectures.

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 node cache, (2) transaction rollback and node crash recovery are handled exclusively by the nodes, (3) local log files are never merged during the recovery process, (4) each node can take a checkpoint without synchronizing with the rest of the operational nodes, and (5) clocks do not have to be synchronized across the nodes and lock tables are not checkpointed.

We have extended the work presented in this paper to include fine-granularity locking in [16]. We are currently evaluating the performance of client-based logging by implementing our algorithms in BeSS [2].

References

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[2] A. Biliris and E. Panagos. A high performance configurable storage manager. In *Proceedings of the Eleventh International Conference on Data Engineering,* Taipei, Taiwan, pages 35 – 43, March 1995.

[3] 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.

[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] 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.

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

[8] 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.

[9] H.V. Jagadish, D.F. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. Dali: A high performance main memory storage manager. In *Proceedings of the 20th International Conference on Very Large Databases,* Santiago, Chile, pages 48–59, September 1994.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] E. Panagos, A. Biliris, H.V. Jagadish, and R. Rastogi. Fine-granularity locking and client-based logging for distributed architectures. In *Proceedings of the Fifth International Conference on Extending Database Technology (EDBT),* Avignon, France, March 1996. To appear.

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

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

[19] 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.

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

[21] 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.

[22] 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.