

1996-06-13

Client-based Logging: A New Paradigm of Distributed Transaction Management

Panagos, Euthimios. "Client-Based Logging: A New Paradigm For Distributed Transaction Management", Technical Report BUCS-1996-010, Computer Science Department, Boston University, June 13, 1996. [Available from: <http://hdl.handle.net/2144/1586>]

<https://hdl.handle.net/2144/1586>

"Downloaded from OpenBU. Boston University's institutional repository."

BOSTON UNIVERSITY
GRADUATE SCHOOL

Dissertation

**CLIENT-BASED LOGGING:
A NEW PARADIGM FOR DISTRIBUTED TRANSACTION
MANAGEMENT**

by

EUTHIMIOS PANAGOS

M.A., Boston University, 1992
B.S., University of Athens, 1988

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

1996

© Copyright by

Euthimios Panagos

1996

Approved by

First Reader

Azer Bestavros, Ph. D.
Assistant Professor of Computer Science
Boston University

Second Reader

Alexandros Biliris, Ph. D.
Member of Technical Staff
AT&T Bell Laboratories

Third Reader

Wayne Snyder, Ph. D.
Associate Professor of Computer Science
Boston University

Acknowledgements

I have been very fortunate to have the opportunity to work with Alex Biliris during the past four years. Alex provided guidance and support when they were needed most during these years. His contribution is invaluable to the research presented in this thesis. I have also had the privilege of working with Professor Azer Bestavros. Azer always makes time to talk and provides different angles on problems, despite his constantly heavy workload. Also, I would like to thank Mark Crovella, Steven Homer, and Wayne Snyder for serving on my committee.

I would like to thank AT&T Bell Laboratories for providing an excellent environment for doing research during the past three years. I learned a lot from my interactions with several researchers, including H.V. Jagadish, Rajeev Rastogi, and Christos Faloutsos, among others. I would also like to thank both the many graduate students of the Computer Science department at Boston University and the rest of my friends that made Boston a fun place to be.

Finally, I would most like to thank my parents, Georgios and Ekaterini, and my brothers, Yiannis and Dimitris. Their love and support have kept me going through this long endeavor. They have encouraged me in whatever I have chosen to do and have always been there for me when I needed them. This thesis is dedicated to them.

**CLIENT-BASED LOGGING:
A NEW PARADIGM FOR DISTRIBUTED TRANSACTION
MANAGEMENT**

(Order No.)

EUTHIMIOS PANAGOS

Boston University Graduate School, 1996

Major Professor: Azer Bestavros, Assistant Professor of Computer Science

Abstract

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 challenges in the design of modern database systems. In this thesis, we propose novel techniques whose aim is to improve the performance and scalability of these new database systems. These techniques exploit client resources through *client-based* transaction management.

Client-based transaction management is realized by providing logging facilities locally even when data is shared in a global environment. This thesis presents several recovery algorithms which utilize client disks for storing recovery related information (i.e., log records). Our algorithms work with both coarse and fine-granularity locking and they do not require the merging of client logs at any time. Moreover, our algorithms support fine-granularity locking with multiple clients permitted to concurrently update different portions of the same database page. The database state is recovered correctly when there is a complex crash as well as when 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.

This thesis also presents the implementation of the proposed algorithms in a memory-mapped storage manager as well as a detailed performance study of these algorithms using the OO1 database benchmark. The performance results show that client-based logging is superior to traditional server-based logging. This is because client-based logging is an effective way to reduce dependencies on server CPU and disk resources and, thus, prevents the server from becoming a performance bottleneck as quickly when the number of clients accessing the database increases.

Contents

Acknowledgements	iv
Abstract	v
1 Introduction	1
1.1 Background and Motivation	1
1.2 Contributions of the Thesis	2
1.3 Outline of the Thesis	4
2 Client-Server Database Systems	5
2.1 Architectural Alternatives	5
2.1.1 Query Shipping vs Data Shipping	5
2.1.2 Page Server vs Object Server	7
2.1.3 Discussion	8
2.2 Utilizing Client Resources	9
2.2.1 Client Caching	9
2.2.2 Correctness Considerations	9
2.3 Related Distributed Architectures	10
2.3.1 Other Database Systems	10
2.3.2 Distributed Shared Memory	11
2.3.3 Distributed File Systems	11
3 Client-Based Logging	13
3.1 Introduction	13

3.2	Terminology and Basic Assumptions	15
3.3	Page-Level Locking	16
3.3.1	Normal Processing	16
3.3.2	Single Node Crash Recovery	17
3.3.3	Multiple Node Crash Recovery	23
3.4	Fine-Granularity Locking	24
3.4.1	Recovery Issues in Fine-Granularity Locking	25
3.4.2	Normal Processing	26
3.4.3	Recovery From a Client Crash	28
3.4.4	Recovery From a Server Crash	29
3.4.5	Recovery From a Complex Crash	31
3.5	Other Recovery Issues	31
3.5.1	Log Space Management	31
3.5.2	Media Recovery	32
3.6	Related Work	32
3.6.1	Client-Server Systems	33
3.6.2	Shared-Disks Systems	34
3.6.3	Distributed File Systems	35
3.7	Conclusions	36
4	Enabling Recovery in BeSS	38
4.1	The BeSS System Architecture	38
4.1.1	Segment and Object Structures	39
4.1.2	Preventing Database Corruption	42
4.2	Implementation Details	42
4.2.1	Interface	42
4.2.2	Operation Modes	43
4.2.3	Cache Replacement	45
4.3	Recovery in BeSS	45
4.3.1	Detecting Updates	45
4.3.2	Generating Log Records	46

4.4	Related Work	48
5	The Performance of Client-Based Logging	50
5.1	Alternative Logging and Recovery Algorithms	50
5.1.1	Server-Based Logging Read-Callback (SL-RCB)	51
5.1.2	Server-Based Logging Write-Callback (SL-WCB)	52
5.1.3	Client-Based Logging Read-Callback (CL-RCB)	53
5.1.4	Client-Based Logging Write-Callback (CL-WCB)	54
5.2	Performance Study	54
5.2.1	Database and System Model	54
5.2.2	Small Database	55
5.2.3	Large Database	58
5.2.4	Extrapolations	59
5.3	Related Work	61
5.4	Conclusions	62
6	Summary and Future Work	63
	Appendix	65
7.1	Queueing Model	65
7.1.1	Formulas for the Loads	65
7.1.2	Throughput Asymptotes	69
	Bibliography	70
	Curriculum Vitae	76

List of Tables

5.1	The different recovery algorithms studied	51
5.2	The configuration of the two databases	55
7.1	Example of load matrix	69

List of Figures

2.1	Client-Server Architectural Alternatives	6
3.1	System Architecture	14
3.2	Restart Recovery Algorithm for a Crashed Node	21
3.3	Construction of the $PSNList_{N_r}$	22
3.4	Restart Recovery Algorithm for an Operational Node	23
3.5	Reconstruction of the DCT	30
3.6	Determining the State of an Object	30
3.7	Recovery of a Page by a Client	31
4.1	The BeSS distributed architecture	39
4.2	Segment and object structure	40
4.3	Shared memory established by the node server	43
4.4	The BeSS memory mapping approach	47
5.1	UpdateOne: small database (a) Response (b) Throughput	56
5.2	UpdateAll: small database (a) Response (b) Throughput	57
5.3	UpdateRepeat: small database (a) Response (b) Throughput	58
5.4	UpdateAll: large database (a) Response (b) Throughput	59
5.5	Illustration of the accuracy of the analytical model	60
5.6	Extrapolations using the analytical model. Throughput vs number of clients. Small database with write-callback; (a) UpdateAll (b) UpdateRepeat	60

Chapter 1

Introduction

1.1 Background and Motivation

During the last decade, advances in software and hardware have changed the way distributed systems are built and operate. First, desktop computers and workstations have become more powerful and reliable with price/performance characteristics that exceed those of larger systems. As a result, a continually increasing number of organizations use desktop machines and workstations for corporate functions to reduce the dependence on more expensive systems. Second, advances in network technology have made it possible to interconnect desktop computers, workstations, and mainframes. The combination of these forces has been the driving factor in the emergence of client-server architectures.

At the same time that distributed computing was adopting the client-server model, the database community was changing direction towards object-oriented database management systems (OODBMSs). This shift occurred because OODBMSs appeared to offer substantial advantages over relational database management systems (RDBMSs) in meeting the needs of advanced applications such as computer-aided design and manufacturing (CAD/CAM), computer-aided software engineering (CASE), geographic-information systems (GIS), and office-information systems (OIS). These advanced applications are characterized by complex data structures with complex patterns of navigation, and they have severe performance demands. The restricted type system (flat tuples and attributes) and the inflexible mode of interaction with user applications (queries and cursors) made RDBMSs inadequate for these applications.

The confluence of client-server computing and OODBMSs has produced a new class of database systems. These systems use a client-server architecture to provide both high performance to user applications and the special database properties such as atomicity, isolation, and durability, that provide semantics which are free of problems associated with concurrent updates to shared data. Examples of this new class of database systems include research prototypes such as Orion [35], Exodus [15, 29], EOS [11], QuickStore [69, 70], and BeSS [10], as well as commercial products such as O2 [24], Objectivity [49], ObjectStore [36], Ontos [50], and Versant [66].

The client-server environment provides many challenges and performance opportunities for the design of these new database systems. Efficient exploitation of all available client resources is necessary to obtain high performance. The most valuable client resources are CPU, memory, and local disk space. Client CPU and memory have been adequately exploited by the *data shipping* approach and the caching of data between transactions for later reuse [71, 67, 17, 28]. Under the data shipping approach, clients request specific data items from the server and they operate on these items locally. However, client disks have not been fully exploited by existing systems and research prototypes. This thesis proposes novel techniques whose aim is to improve the performance and scalability of these new database systems. The techniques exploit client disk space for offering transactional facilities locally.

1.2 Contributions of the Thesis

Until recently, client machines were not considered to be as reliable as server machines for the following reasons. First, due to the high cost of main memory and secondary storage devices, it was more cost effective to increase the resources of the server machine rather than the resources of the client machines. Second, client machines are usually configured and managed by users as opposed to the server machine which is placed in a special machine room and controlled by specialized personnel. As a result, clients were not allowed to manage shared data without interacting with the server in order to guarantee the availability of data in the presence of client failures. Consequently, transactions had to communicate with the server for committing even in the case where they had not performed any updates.

Even though existing client-server database systems do not allow clients to perform transaction management, they use client disks in several ways. First, client disks are used through virtual memory swapping when the client's buffer pool is stored in virtual memory. However, the limited access to the virtual memory management system and the operating system's lack of knowledge about database access patterns may have a negative effect on the performance of the system [62]. Other ways of using client disks is by caching relational query results [23], or extending the in-memory cache [27], or creating a "personal database" where checked out objects are placed [66].

Today, technological advances and the reductions in computer hardware costs have resulted in powerful and reliable workstations which, quite frequently, approach server machines in terms of resources. In addition, network technology is quite mature and client-server connections via networks are reliable. 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 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 can be provided by using client disks for storing recovery related information (i.e., log records) and offering

transactional facilities locally. Transaction response times will be shortened since all recovery related operations will take place at the client. Furthermore, system performance will increase since the load of the server and the network will be reduced even more. This thesis validates the above assertion by presenting the design, implementation, and performance of new recovery algorithms that offer client-based transaction management.

The major contributions of the thesis are the following:

- **Client-based transaction management:** We propose two recovery algorithms that use client disks for offering transactional facilities locally while maintaining the transaction semantics and reliability associated with traditional database systems. Our algorithms work with both coarse and fine-granularity locking. Moreover, our algorithms support fine-granularity locking with multiple clients permitted to concurrently update different portions of the same database page. The database state is recovered correctly when there is a complex crash as well as when 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 main features of these algorithms include:

- Clients perform updates on locally cached pages and produce log records that are written to a local log file.
 - Updated pages are not forced to disk at transaction commit or when they are replaced from a client cache.
 - Local log files do not have to be merged during client and/or server restart recovery.
 - Transaction rollback and client restart recovery are handled exclusively by the clients.
 - Both the clients and the server can take checkpoints without synchronizing their actions.
 - Client clocks do not have to be synchronized and lock tables are not checkpointed.
- **Implementing recovery in a memory-mapped storage manager:** Memory-mapped storage managers allow application programs to update the database by dereferencing virtual memory pointers. Under this approach, updates are carried out in memory speeds, but the detection of the updated regions becomes more difficult than in systems that use either a function call interface or require special compiler support (e.g., Ode [6, 5]).

In this thesis, we present how recovery is implemented in BeSS, a memory-mapped object storage manager. BeSS takes advantage of current advances in operating systems and virtual memory hardware to provide fast access to persistent objects independent of their size and their physical location in a distributed client-server architecture.

- **Detailed performance study:** Existing performance studies for client-server systems either study inter-transaction caching ignoring overheads related to recovery, or they study recovery algorithms ignoring inter-transaction caching. This thesis presents a

detailed study of two classes of recovery algorithms for distributed (client-server and peer-to-peer) OODBMSs. The first assumes that the server is the only source for providing transactional facilities. The second corresponds to the client-based recovery algorithms presented in this thesis. The performance of these recovery algorithms was studied under two cache coherency algorithms (read and write callback) resulting in four different alternatives. All alternatives were implemented in BeSS, and the OO1 database benchmark was used to evaluate the study.

1.3 Outline of the Thesis

The remainder of the thesis is organized as follows. Chapter 2 describes database client-server systems in more detail. First, alternative architectures are described and evaluated, and the choice of *data shipping* is motivated. Next, we describe the performance characteristics and reliability of data shipping client-server database systems. Finally, we present other distributed architectures in which similar problems arise, and we outline the differences between them and a client-server database system.

Chapter 3 presents two new recovery algorithms for distributed client-server architectures. The algorithms exploit client disk space for providing transactional facilities locally. A detailed comparison with related work in the area of database systems and other contexts with similar characteristics is presented at the end of this chapter.

Chapter 4 presents an overview of the design and implementation of BeSS, a memory-mapped object store, and it focuses on the components of BeSS that are related to the implementation of recovery.

Chapter 5 presents a detailed performance study of the algorithms presented in this thesis using the OO1 database benchmark.

Finally, Chapter 6 summarizes the contributions of the work presented in this thesis and outlines avenues for future work.

Chapter 2

Client-Server Database Systems

2.1 Architectural Alternatives

There are two basic architectural alternatives for implementing database functionality in a client-server environment. They vary in the unit of interaction between the server process and the client application. Systems where clients send queries and operations to the server are named *query shipping* systems. Systems where clients request from the server specific data items which are operated on locally are named *data shipping* systems. Figure 2.1 shows these two alternatives.

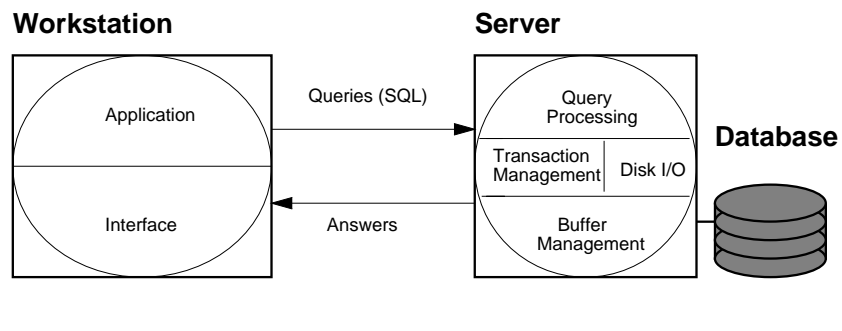
Data shipping architectures can be further classified according to the unit of data transfer between the clients and the server [25]. *Page servers* use physical entities such as pages or segments as the unit of transfer, and the application process is responsible for mapping objects to these physical entities. *Object servers* use logical entities such as objects or groups of objects, and the server is responsible for mapping objects to disk pages. *File servers* are a subclass of page servers where application processes use a remote file service (namely, NFS [47]) to access the database. The advantages of each architectural alternative are presented in the following sections.

2.1.1 Query Shipping vs Data Shipping

In traditional centralized database architectures and most of today's commercial client-server relational database systems, queries and operations are shipped from client machines to the server which processes the requests and returns back the results. In contrast, most of the client-server OODBMSs follow a data shipping approach where clients operate on the data items the server sends to them (e.g., O2 [24], ObjectStore [36], Orion [35], Exodus [15, 29], EOS [11]). The advantages of the page shipping approach for OODBMSs are mainly performance-related as indicated by the OO1 database benchmark [21] and the more recent OO7 benchmark [16]. In particular, data shipping has the following advantages:

1. System performance increases because the server is offloaded and data processing is undertaken by the clients.

Query Shipping Architecture



Data Shipping Architecture

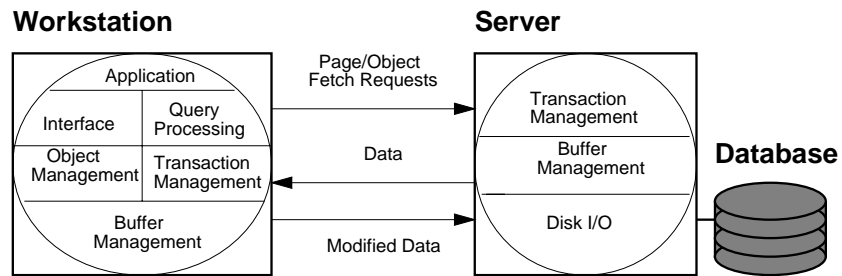


Figure 2.1: Client-Server Architectural Alternatives

2. Scalability is improved because the system grows incrementally as new clients are added to it. These clients contribute to the total system resources by utilizing part of their resources.
3. Existing libraries and tools can be used to operate on the data since the data is on the client. In this way, navigational and graphical user interfaces (GUIs) can operate directly on the data present in the application's address space offering better response time.

The advantages of query shipping are the following:

1. Since all database operations are executed by the server, existing centralized database management systems can be easily modified to function in a client-server environment.
2. Almost all RDBMSs use SQL as their query language. As a result, clients can interact with several different RDBMSs by sending SQL queries to these systems.
3. Communication costs are reduced because all the processing is undertaken by the server, and only items that match the query the client submitted are returned to the client.

2.1.2 Page Server vs Object Server

The performance tradeoffs among the three different data shipping approaches were studied in [25]. The study showed that the relative performance of each approach is workload dependent. The advantages of a page server architecture are the following:

1. Simplified server architecture because clients are responsible for mapping objects to pages.
2. Reduced communication costs and higher performance of scan queries when objects are clustered effectively into pages.
3. Existing page-level concurrency and recovery algorithms can be used with small modifications.

The advantages of an object server architecture are the following:

1. Efficient use of client buffer space because clients can request only the objects they need to access from the server.
2. The server can perform operations on objects since it knows about objects.
3. Ability to support object-level locking because the server knows which objects are accessed by which clients.

Page server OODBMSs include ObjectStore [36], O2 [24], GemStone [13], Exodus [29], EOS [11], QuickStore [69], and BeSS [10] among others. Object server OODBMSs include Orion [35], early versions of GemStone [22], Versant [66], and a prototype of O2 [65]. The Ontos system [50] can operate both as page server and object server depending on the choice made by users and Objectivity [49] uses NFS for transferring pages.

2.1.3 Discussion

Current trends in hardware and software favor the prevalence of the data shipping approach as the choice of many client-server applications. With respect to hardware, recent advances in networking provide increased network bandwidth (e.g., 100 Mbit Ethernet, FDDI, and ATMs) that reduces considerably the overhead associated with sending messages between clients and servers. Also, larger and more reliable disks, increased memory capacity, and faster processors benefit the data shipping approach because it exploits these plentiful client resources in a better way. With respect to software, the performance requirements of navigational and graphical user interfaces (GUIs) require minimal overhead when accessing raw data. This means that the data should be moved close to where the applications are running (i.e., the client workstations). Also, recent improvements in conventional operating systems such as lightweight threads, flexible memory and buffer management, and even real-time scheduling (e.g., Sun's Solaris and Microsoft's Windows NT) allow database systems to implement most, if not all, of their functionality on the client machines.

Among the available data shipping approaches, the page server benefits the most from the above-mentioned technological trends because of its simplicity. Nevertheless, page servers are often criticized for being too restrictive when it comes to concurrency and recovery because existing page servers use pages as the minimum locking granularity. However, recent algorithms show that object-level locking can be efficiently supported in a page server environment [18]. For these reasons, the work that is reported in this thesis is based on a page server architecture.

On the other hand, there are also client-server applications that would benefit if the server were executing queries. The advantage of executing queries at the server is that communication cost is reduced considerably if the result of the query against the collection is a small subset of the collection. As an example, consider an application that requires a full scan of all the fingerprints in the database so they can be compared against a target fingerprint. Having the server execute the scan would be more efficient than transferring the entire fingerprint collection to the client.

There are however several problems with this approach. First, if the server is not powerful enough the scheme may face scalability problems since the server may become the bottleneck of the system. Second, if methods can be applied on both the client and the server, their caches must be synchronized. This is because the server must be aware of the updates performed by the client before it applies methods that involve the updated objects. We plan to further investigate this approach in our future work in the context of hybrid client-server systems that combine data shipping with query shipping.

2.2 Utilizing Client Resources

In a typical client-server environment the server machine has more CPU power, memory capacity, and disk storage than any individual client machine. However, the total CPU power, memory, and disk capacity of the clients exceeds that of the server. As a result, minimizing the interaction between the clients and the server can provide both performance and scalability benefits. By reducing the need to communicate with the server, transaction response time is improved considerably since less messages have to be sent over the network. In addition, system throughput and scalability are improved as both the network and the server become less loaded and can accommodate more clients.

2.2.1 Client Caching

Caching refers to the storing of data items close to where they are needed so that the cost of accessing them is reduced. Examples of caching include computer registers, file system buffers, and database buffers, among others. In a page server environment, client caching refers to the ability to maintain pages retrieved from the server in buffers that belong to the application's address space so that secondary storage access and communication costs are kept minimal. Specifically, when an object needs to be retrieved, the page containing the object is fetched from the server, and once loaded in memory it becomes available to the transaction.

Despite its potential performance advantages, caching is not free of problems. As replicas of a page may exist in more than one client cache at the same time, a protocol must be applied to ensure cache consistency. Depending on the number of clients and the workload characteristics of the system, such a cache consistency protocol may add a significant amount of overhead to the system. In addition, the interaction of such a protocol with the particular concurrency control protocol employed by the system may overload the server, and it may increase transaction abort rates due to delayed discovery of stale pages. Thus, the choice of the cache consistency protocol may degrade system performance instead of increase it.

Recently, several cache consistency protocols have been proposed and their performance has been examined under a variety of workloads [71, 67, 17, 28, 23]. Among these protocols, the *callback locking* protocol [33, 36] offers an integrated locking and cache coherency solution that has been shown to have good performance over a wide range of workload characteristics [67, 17, 18]. Furthermore, two commercial systems, Objectstore [36] and Rdb/VMS [56, 39], already employ this protocol. For these reasons, the work reported in this thesis is based on the callback locking protocol.

2.2.2 Correctness Considerations

While client resources can be used for boosting the performance of the system, the degree to which they can be exploited is limited by the fact that client-server database systems

have to support the traditional ACID transaction properties [31, 30]:

Atomicity: Transactions are either executed in their entirety or not at all.

Consistency: Transactions preserve any integrity constraints when their execution is terminated.

Isolation: Even though transactions execute concurrently, each transaction does not see any other transaction during its execution.

Durability: Committed transaction updates survive failures.

Isolation is achieved by ordering the interleaved execution of transactions in such a way that the final output is the same as the output of a serial (i.e., non-interleaved) execution of the same set of transactions [1]. Traditionally, this ordering is realized by using either blocking or transaction restarts for resolving data conflicts.

2.3 Related Distributed Architectures

Apart from the client-server systems, many other distributed systems employ caching techniques and utilize disks for local management of data. In this section we present example architectures that have some similarities with the work presented in this thesis. Direct comparisons are made in subsequent sections for issues that are addressed in those sections. The comparisons made in this section are based on the following criteria:

- **Transactional semantics:** How the system handles concurrent users and what guarantees are given to the users about the durability of their updates?
- **Cost tradeoffs:** What are the costs associated with accessing shared data? What is the overhead of the system for maintaining consistency?
- **Target applications:** What are the characteristics of the applications for which the system is designed? Some of the important characteristics here are the amount of data accessed and the degree and the kind of sharing (i.e., read-only versus read-write).

2.3.1 Other Database Systems

Issues related to caching appear in shared-disks, shared nothing, and several hybrid client-server database systems. All of these systems provide the same transactional semantics as the page server systems.

Shared-disks systems have received a great deal of attention in recent years [54, 41, 38, 56, 39]. A shared-disks or data sharing system consists of a number of nodes that are coupled via a high-speed network link. Each node in the complex runs its own copy of the database system software and has direct access to the common database(s) stored on disk. While

shared-disks systems face the same cache coherency problem present in page server systems performing caching, the two architectures differ in several ways. First, the communication cost among the nodes of a shared-disks system is smaller than the cost involved in a client-server system. Second, the structure of a shared-disks system is peer-to-peer as opposed to the master-slave structure of a client-server system.

Local disk space is also used in a hybrid client-server architecture presented in [23]. In that work, local disks are utilized 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. Versant [66], a commercially available OODBMS, also explores client disk space. In Versant, users can check out objects by requesting them from the server. The checked out objects are stored locally in a “personal database.” Checked out objects are not available to the rest of the clients until a corresponding checkin is completed.

Shared nothing systems differ from shared-disks systems in the way the disks are allocated to the processing nodes. While in shared-disks systems all disks are shared among the processing nodes, in a shared nothing architecture the disks are partitioned among the nodes, and each node manages its local database partition. Query shipping is the method for transactions running on one node to access data stored with another node. Transactions accessing multiple partitions have to follow some kind of distributed commit protocol in order to commit their updates (e.g., the two-phase commit protocol). It is critical to partition the database in such a way that the load of the system is balanced. In addition, query optimization becomes more difficult as the query optimizer has to generate a distributed execution plan.

2.3.2 Distributed Shared Memory

Distributed Shared Memory (DSM) is an abstraction for supporting the notion of shared memory in an environment consisting of workstations and file servers [48, 37]. References to virtual memory locations that reside on a different machine are treated transparently as if they were virtual memory locations residing on the local machine. Unlike multiprocessor machines that handle cache consistency in hardware, DSM systems use messages for keeping the client caches consistent similar to the page server architectures. However, DSM systems do not offer transaction support. Consequently, node failures can be masked by implementing some kind of checkpointing mechanism.

2.3.3 Distributed File Systems

Distributed file systems use client caching for improving performance (e.g., the Andrew File System [33] and its follow-on project Coda [59], and Sprite [46], to name a few). Distributed file systems, however, cache entire files or large chunks of files rather than individual pages because of the following workload characteristics:

- Files are usually read sequentially and in their entirety.

- Updates of shared files correspond to the exception rather than the rule.
- Many files are deleted soon after they are created.

A major difference to page servers is that distributed file systems typically do not provide any transaction support. Thus, client failures may result in a loss of all the modifications that had not been propagated to the server at the time the failure occurred. Coda [59], however, provides limited transaction support as well as client-based logging for supporting disconnected clients. In Coda, updates made by disconnected clients are logged at the client and they are integrated into the system on reconnection.

Chapter 3

Client-Based Logging

This chapter proposes a new paradigm for distributed transaction processing which has the potential to exploit all available resources and improve system scalability and performance. 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.

The recovery algorithms we present in this chapter work with both page-level and fine-granularity locking. Our algorithms are of potential value to a variety of computing environments. They can be applied to shared nothing, shared-disks, and peer-to-peer architectures (e.g., Shore [14], BeSS [10]). Also, they can be applied to client-server architectures, where client disks are used for logging.

3.1 Introduction

Figure 3.1 shows the general structure of the distributed database system we assume while presenting our recovery algorithms. The system consists of several processing nodes which are connected via a local and/or wide area network. A node that has 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 – we say that this node *owns* the databases attached to it. Note that database *ownership* is a role that a node plays, and it does not characterize the machine as a whole. All owner nodes have local logs. Nodes that do not own databases, 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 do have local logs.

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

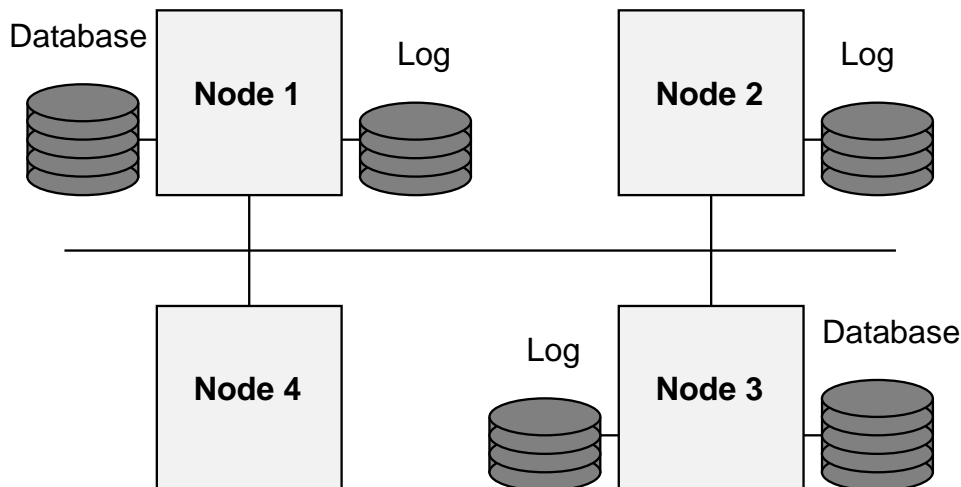


Figure 3.1: System Architecture

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.
- When fine-granularity locking is employed, multiple nodes can concurrently update different portions of the same database page.
- 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.

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. Moreover, data about different customers may co-reside on the same page. Different technicians may need information about different customers but conflict on the page they wish to cache. The fine-granularity locking algorithm we present here accomplishes this as well. Clearly, mobile nodes which

are disconnected frequently 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 chapter is organized as follows. Section 3.2 states our assumptions about the distributed environment that we assume when describing our recovery algorithms. In Section 3.3 we describe our algorithms for both single and multiple node crashes when the minimum locking granularity is a database page. Section 3.4 extends the algorithms presented in the previous section to handle fine-granularity locking. Section 3.5 covers some advanced issues regarding the algorithms of the previous sections. We compare our work with relevant work that appears in the literature in Section 3.6. Finally, we summarize in Section 3.7.

3.2 Terminology and Basic Assumptions

Transactions are executed in their entirety at the node where they are started. Data items referenced by a transaction are fetched from the owner node 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 [25]). Each node has a local 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 [31]. Modified pages that are replaced from the local cache are either written *in-place* to disk, or they are sent to the owner node depending on whether they belong to a 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 local lock manager that caches the acquired locks and forwards the lock requests for remote data items to the node owning these items. *Inter-transaction* caching [71, 67, 17, 28] of the cached pages and locks is supported, and the *callback locking* protocol [33, 36] is used for cache consistency. Both shared and exclusive locks are retained by each 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.

Each database page consists of a header which among other information contains a *page sequence number* (PSN). The PSN of a page is incremented by one every time the page is modified by a transaction. The owner node initializes the PSN value of a page when the page is allocated by following the approach presented in [42] (i.e., the PSN stored on the space allocation map, which contains 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 node crashes. Recovery is based on the *write-ahead log* (WAL) protocol and the ARIES [40] algorithm 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) 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 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.

3.3 Page-Level Locking

In this section, we discuss logging and recovery protocols that enable nodes participating in a distributed system to log their updates in a local log file. Our techniques avoid the need to ship the generated log records to the nodes that own the data. In addition, since each transaction is executed in its entirety at only one node, all log records for updates to pages are stored with this node, and locks are retained after the termination of the transaction, no distributed commit processing is required. The granularity of both locking and callback is assumed to be at the level of a database page.

3.3.1 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 take checkpoints periodically. 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. Note that we do not overwrite an existing entry because this could cause incorrect recovery when the node crashes after taking a checkpoint and before the page is forced to disk. According to ARIES, the page would not be considered dirty if no log records are written for this page after the checkpoint.

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.

3.3.2 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 for 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 the crashed node involves the recovery of the updates performed by locally executed transactions. In addition, the recovery of the crashed node may involve the recovery of updates performed by transactions that were executed in another node. 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 solve the following problems.

1. Determining the pages that may require recovery.
2. Identifying the nodes involved in the recovery of these pages.
3. Reconstructing lock information.

4. Coordinating 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 on behalf of the crashed node. This is true 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.

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. In 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 was 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 since 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 after the checkpoint record.

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, during restart the crashed node N 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 determines the pages that have to be recovered using these lists and its own DPT.

However, 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. Also, these pages may be 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 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 has 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.

Identifying The Nodes That Are 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 situation arises 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.

Reconstructing Lock Information

Before the crashed node starts recovering the pages that were identified to require recovery, the node 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 were held by both local and remote transactions before the crash. The locks that were held by remote transactions are present in the lock tables of the nodes where those transactions were executed. In addition, locks that were held by local transactions for pages owned by remote nodes are also present in the lock tables of the remote nodes.

Each operational node 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 been recovered yet. The list of locks an operational node had acquired from the crashed node, together with 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 establishes its lock tables, and it acquires exclusive locks for the pages present in its DPT that do not have a

lock entry. At this point, normal transaction processing can continue.

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 carried out in the proper order to preserve the ARIES repeating history property. This order corresponds to the order in which P was updated by transactions that were executed at the involved nodes.

Because the granularity of locking is a page, only one node at a time can update P . Consequently, only one node at a time can write a log record containing the PSN value stored on P before the update and, hence, all log records written for P in the log files of the nodes participating in the system contain distinct PSN values. In addition, since the PSN value of a page is incremented on every update by one, the ascending ordering of the PSN values stored in the log records written for P determine the order in which P was updated before the crash.

However, N can determine the recovery order for P without considering the PSN values stored in all log records written for P by the nodes involved in the recovery of this page. This is because of the following observation. Due to the strict two-phase locking protocol we use, when a transaction updates a page several times before it terminates, all log records the transaction writes for this page contain consecutive PSN values. Furthermore, log records written for the same page by two different transactions that are executed on the same node will contain consecutive PSN values when the page is not updated by another node in the interim. Thus, N can determine the recovery order of a page by collecting from each node that is involved in the recovery of the page, including itself, the list of non-consecutive PSN values stored in the node's log records written for the page, referred to as *PSNList*.

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.

Summary of the Algorithm

In this section, we summarize our algorithm, and we present several optimizations that can be employed to make the algorithm more efficient. Before presenting the steps followed by the crashed node N and each node N_r involved in the recovery process, we state the notation we are going to use.

Let $PSNList_N$ denote the *PSNList* constructed by N . Each entry in this list has

1. Initialize $PSNList_N$, $RecList$, and $CallbackList$ to NULL.
2. Without waiting for a response, request from each operational node the entries in the lock table, DPT, and cache which correspond to pages owned by N .
3. Read from your log file the DPT and the transaction table stored in the last complete checkpoint record. Insert the entry $\langle PID, N \rangle$ in $RecList$ for each page with id PID that is present in DPT.
4. Execute the ARIES analysis pass by examining all log records written after that checkpoint. For each log record written for a page P , do the following.
 - (a) If P is not present in DPT, insert $\langle PID, PSN, LSN \rangle$ in $PSNList_N$, $\langle PID, N \rangle$ in $RecList$, and $\langle PID, PSN, PSN + 1, LSN \rangle$ in DPT, where PID is P 's id, PSN is the PSN value stored in the log record, and LSN is the LSN of the log record.
 - (b) If P is present in DPT, insert $\langle PID, PSN, LSN \rangle$ in $PSNList_N$ if the $CurrPSN$ value of this entry is not the same as the PSN value stored in the log record. Next, update the $CurrPSN$ value of the entry to be one more than the PSN value stored in the log record.
5. Wait until all operational nodes have sent the lists requested in the first step. For each node N_r , do the following.
 - (a) Acquire the locks N_r holds for pages owned by N .
 - (b) For each page with id PID that is cached at N_r , remove the entries for this page from DPT, $PSNList_N$, and $RecList$. In addition, add $\langle PID, N_r \rangle$ in $CallbackList$.
 - (c) Insert $\langle PID, N_r \rangle$ in $RecList$ if PID is not present in the cache of N_r and present in its DPT.
6. Send to each node N_r the $RecList$ entries that have N_r as their second attribute. In addition, request from N_r all page whose ids belong to the entries N_r has in $CallbackList$.
7. Scan the local log from the minimum of the $RedoLSN$ values present in the DPT and update $PSNList_N$ using the algorithm shown in Figure 3.3. The scan stops when the last complete checkpoint record is reached.
8. Wait until all nodes involved in the recovery send back their $PSNLists$. Next, order the nodes involved in the recovery of a page P in an ascending ordering based on P 's PSN values present in these lists, including your own $PSNList_N$. Adjacent entries belonging to the same node are merged into one entry having the minimum of the two PSN values.
9. Read from disk each page P present in $RecList$ and remove from the merged lists computed in the previous step all entries having PSN values that are less than the PSN value of P . Next, coordinate the recovery of P in the following way.
 - (a) Send P to the node N_r having the minimum PSN in the above list. The second minimum PSN value present in the list is also sent to N_r , if any.
 - (b) When N_r sends back P , place P in the cache and remove the entry from the list.
 - (c) Repeat the previous two steps until there are no more entries for P in the list.
10. Undo all updates belonging to local transactions that were active at the time of the crash.
11. Take a checkpoint and continue normal processing.

Figure 3.2: Restart Recovery Algorithm for a Crashed Node

the form $\langle PID, LSN, PSN \rangle$, where PID is the id of a page, LSN is the LSN value of a log record written for the page, and PSN is the PSN value stored in the log record. Let $RecList$ denote the list of pages that may require recovery. This list is constructed by the crashed node during restart, and it contains entries of the form $\langle PID, NID \rangle$, where PID is the id of a page, and NID is the id a node that will recover the page. Let $CallbackList$ denote the list of pages that have to be requested from the operational nodes before the restart recovery is over. This list contains entries of the form $\langle PID, NID \rangle$, where PID is the id of a page, and NID is the id the node that will supply the page. In addition, we assume that each DPT entry contains two extra fields named $RecPSN$ and $RecLSN$. The way these fields are used is explained below.

1. Set the $RecPSN$ and $RecLSN$ fields of every DPT entry that corresponds to a page belonging to the list of pages requiring recovery to NULL.
2. Scan the local log starting from the minimum $RedoLSN$ value present in the DPT entries for the pages belonging to the above recovery list. When a log record written for a page P present in this list is encountered, do the following.
 - (a) If $PSNList_{N_r}$ is NULL add $\langle PID, LSN, PSN \rangle$ in $PSNList_{N_r}$, where PID is the id of P , LSN is the LSN value of the record, and PSN is the PSN value stored in the log record.
 - (b) If $PSNList_{N_r}$ is not NULL add $\langle PID, LSN, PSN \rangle$ in $PSNList_{N_r}$ if the PSN value stored in the log record is not the same as the $RecPSN$ field of the DPT entry for P .
 - (c) Set the $RecPSN$ of the DPT entry for P to be $PSN + 1$.

Figure 3.3: Construction of the $PSNList_{N_r}$

The steps followed by the crashed node N during its restart are shown in Figure 3.2. When the crashed node sends to an operational node N_r the list of pages that require recovery, N_r constructs its $PSNList_{N_r}$ list following the steps shown in Figure 3.3. When N_r receives a page P for recovery, it executes the steps shown in Figure 3.4.

We can optimize the algorithm used for constructing node N 's $PSNList_N$ in the following way. N maintains a number of $\langle PSN, LSN \rangle$ pairs for each page P having an entry in its DPT. When P is about to be updated by a local transaction, the $CurrentPSN$ value present in the DPT entry for this page is compared against the PSN value stored on P . If they are different, a new entry is added to the above list having as PSN the PSN value stored on P and as LSN the LSN value of the log record written for the update. When a page is removed from the DPT, all $\langle PSN, LSN \rangle$ pairs for this page are deleted. In this way, during a single node crash, the operational nodes avoid scanning their logs and the $PSNLists$ are sent to the crashed node together with the lists requested during Step 1 of the algorithm shown in Figure 3.2.

In addition, Step 9 shown in Figure 3.2 can be optimized by reading the pages present in $RecList$ asynchronously. The asynchronous disk reads for these pages can be initiated after Step 5 of the algorithm shown in Figure 3.2 has been completed.

1. If the *RecLSN* field of the DPT entry for P is NULL, start scanning the log from the *RedoLSN* value present in P 's DPT entry. Otherwise, use the *RecLSN* value. During the scan, skip all log records that were written for a page other than P . When a log record written for P is encountered, do the following.
 - (a) If the PSN value stored in the log record is less than the PSN value present on the page, skip it.
 - (b) If the PSN value stored in the log record is the same as P 's PSN value, apply the log record and increment the PSN value of the page by one.
 - (c) If the PSN value stored in the log record is greater than the PSN value of the page and less than the PSN value sent along with the page, if any, apply the log record and set the PSN value of the page to be one more than the PSN value stored in the log record.
 - (d) If the PSN value stored in the log record is greater than the PSN value sent along with the page, set the *RecLSN* value of P 's DPT entry to be the LSN of this log record and suspend the scanning.
2. If the *RecLSN* value is NULL, remove P 's entry from the DPT if N_r does not hold an exclusive lock on P . If N_r holds an exclusive lock on P , set the *RedoLSN* field of P 's DPT entry to be the current end of the log.
3. Send P back to the crashed node.

Figure 3.4: Restart Recovery Algorithm for an Operational Node

3.3.3 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 one 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 transactions. In addition, the pages that were present in its cache at the time of the crash which it owns and have been updated by remote transactions must be recovered. 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. The crashed node updates the DPT stored in that checkpoint by inserting new entries for the pages that do not have an entry, and they 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.

3.4 Fine-Granularity Locking

So far in this chapter, we have assumed that the minimum locking unit is a database page. Locking at a finer granularity provides greater concurrency and, hence, better performance for highly accessed data structures such as indices.

One simple way to obtain some of the benefits of fine-granularity locking is to use the two-level locking scheme suggested in [53]. The lock manager of each node acquires locks on objects for the locally running transactions, but it requests page locks when it communicates with the lock manager of another node. Since synchronization across nodes is still at the page level, the page-level locking recovery algorithms presented in Section 3.3 are directly applicable. However, two applications running on different nodes cannot concurrently update two objects on the same page and, thus, only some of the benefits of fine-granularity locking are obtained.

In this section, we consider a system in which fine-granularity locking is allowed even across nodes. Thus, several nodes may simultaneously update different objects residing on the same page. Our algorithms can be augmented with an adaptive locking scheme that switches between object and page-level locking depending on the degree of data conflicts, such as [56, 18]. However, the specifics of the adaptive scheme are orthogonal to our discussion here which is focused on logging and recovery. For concreteness, our description below assumes the scheme of [18].

In the remainder of this chapter we assume that the database is owned by only one node, the server. The rest of the nodes are assumed to be clients having local disk space. This assumption is made to simplify the description of the algorithms by avoiding the need to distinguish between the server and the client role of a node. In addition, 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.

3.4.1 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 [41] and client-server architectures [42]. 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. The cost of shipping pages among multiple nodes is relatively low in an environment where all the nodes are coupled via a high-speed interconnect – an assumption made in the shared disks architectures [41, 55, 54, 53]. However, in a client-server environment this cost could be significant due to the relatively high communication overhead associated with a local (or wide) area network.

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 [43, 14]. 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 requires 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. To our knowledge, no one else has studied this problem before.

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 merged¹. 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 of a log record are present on the page. According to the assumptions made in Section 3.2, 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.

¹Object 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 thesis.

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 property 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 when a client crashes). 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 3.4.4 explains how the callback log records are used during server restart recovery.

3.4.2 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 containing 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. 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 client lock manager 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 present on them 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 page², and the *RedoLSN* field is set to NULL. 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 either objects residing on the page or the page itself.

²If 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 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.

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 it sets the value of the *PSN* field to be the PSN value present on P . Next, the server merges the updates present on P 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.

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.

3.4.3 Recovery From a Client Crash

When a client fails, its lock table 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 table 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 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 to 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 than or equal to the PSN value stored on the page.

During the redo pass, callback log records may be encountered. According to the discussion presented in Section 3.4.1, these callback log records are not processed. 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.

3.4.4 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 which 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 its lock table, by requesting from each client a copy of the DPT, the list of the cached pages, and the entries in the client's lock table.

The clients that are involved in the recovery are identified during the procedure of determining the pages that require recovery. In particular, the clients that will participate in the recovery of the page are those that have an entry for the page in their DPTs, and the page is not present in their caches.

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 steps

1. Insert into the DCT entries of the form $\langle PID, CID, NULL, NULL \rangle$ 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.

Figure 3.5: Reconstruction of the DCT

shown in Figure 3.5.

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.

Figure 3.6: Determining the State of an Object

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 way shown in Figure 3.6.

A 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 executes the steps shown in Figure 3.7.

When the server receives the request for page P from C in Step 3 of the algorithm shown in Figure 3.7, 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

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.

Figure 3.7: Recovery of a Page by a Client

send P to C . Otherwise, the server will request P from CID, and then it will forward P to C . This situation materializes when CID is recovering P in parallel with C . Here, 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.

3.4.5 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 particular, 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. Starting from the minimum $RedoLSN$ value present in the DCT stored in the last checkpoint record, the server will scan its log file 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 3.4.4.

3.5 Other Recovery Issues

In this section we discuss two additional recovery issues: log space management and media recovery.

3.5.1 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. However, 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 sends a dirty page that is replaced from its local cache to the owner node, the node remembers the current end of its log file. When the owner node forces the page to disk, it informs all nodes that had replaced the page. These nodes 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 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 private 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 one of the nodes that have it in their caches and then it is written to disk.

3.5.2 Media Recovery

If a node is not able to read a page from disk because of a media error, then media recovery is required for recovering the page. The media recovery process reconstructs the current state of the page from an archival copy, along with a log of all the updates performed on the page since the archival copy was made. Since each node writes log records for updates to a page in its private log file, the log records needed for recovering the page may be located in several log files. Here, media recovery can be performed by all nodes by using the same technique as the one used for recovering a page updated by multiple nodes and which was present in the cache of the owner node at the time of the crash. However, the DPTs that will be used are those stored with the archival copy. Those DPTs correspond to the DPTs the nodes had at the time the archival process started.

Another way of performing media recovery is by merging all log records written for pages owned by a node into one log file. During normal processing, each node sends asynchronously the log records for updates to pages owned by remote nodes to these nodes. Each owner node analyzes the incoming log records and merges them into one log file. When a media failure occurs, the owner node waits until all log records are received and merged and then it redoes the updates on the affected page(s) by using the archival copy.

3.6 Related Work

Several papers have addressed recovery problems in the area of client-server and multiprocessor shared-disks (also referred to as *data sharing*) architectures, and they are relevant to

our work. Also, work in distributed file systems has some similarities with our techniques. In the next sections we present a comparison of our work with the main features of the algorithms available in the literature.

3.6.1 Client-Server Systems

In a typical client-server environment, the server is the owner of the database and the only source that provides locking and logging facilities. Consequently, transactions running on client workstations have to communicate with the server to log and commit their updates. In the following, we compare our work with techniques that utilize client resources in a client-server system in order to reduce the need to communicate with the server.

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 [18]. 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 [18].

In [27], 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 [27], we have investigated logging and recovery issues where clients are allowed to perform local logging, and they do not send pages to the server at transaction commit. In our schemes, recovery for a particular page is performed by the client itself, and the server coordinates the recovery of a page that has been updated by multiple clients.

Local disk space is also used in the architecture presented in [23]. Here, 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 [66], 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." This way, network traffic is reduced. In addition, in order to increase performance, Versant allows users to turn off locking and logging for objects stored in a personal database. The checked out objects are unavailable to the other clients till 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 transaction management.

In ARIES/CSA [42], 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.

In addition, 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.

The Shore project [14] represents a merger of object-oriented database and file system technologies by providing a file system interface to an object-oriented database. Shore employs a peer-to-peer server architecture where every participating node runs a Shore server. Shore servers provide transaction management facilities to the application programs running on the same node and they “own” local databases. Similar to Shore, our proposed architecture departs from the traditional strong client-server model, and it is designed so that it can take advantage of the plentiful and inexpensive resources provided by client workstations. Unlike Shore where the server owning an object is the one that stores all log records for the object and recovers the object, our algorithms use client disks to store the log records and clients recover all the objects that were modified by locally executing transactions.

3.6.2 Shared-Disks Systems

In a shared-disks architecture, each processing node usually has its own local log file where all log records generated by transactions running on that node are written. Transaction processing is handled by each node and inter-node communication is required only for concurrency and coherency control. To our knowledge the only papers that discuss shared-disks recovery issues to some extent are [54], [41], [38], and [56, 39]. In the following, we compare our work with the protocols presented in these papers.

In [54], 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 [40].

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 [41], 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. 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. Furthermore, 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 [41], nor do they require merging of the private logs at any time. Private logs have to be merged in the *fast* and *super-fast* schemes presented in [41] even in the case where only a single node crashes.

The *shared data/private log* recovery algorithm presented in [38] 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 [38] 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 [56] 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/redolog recovery scheme [39]. 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, 29, 34]. 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.

3.6.3 Distributed File Systems

Coda [59] is a distributed file system operating on a network of UNIX workstations. Coda is based on the Andrew File System [33] 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.

3.7 Conclusions

Although in shared nothing and several shared-disks architectures each node writes log records for updates to pages in its private log file, existing client-server database systems do not allow clients to perform local logging because they consider client machines to be unreliable. Consequently, transactions running on a client have to communicate with the server for committing even when they do not perform any updates. However, current technological advances have increased dramatically the reliability of desktop computers and workstations. In addition, clients are considered to be highly available in several application environments (including corporate and collaborative environments). As a result, extra performance benefits can be realized by exploiting client disk space for offering transactional facilities. On the one hand, transaction response time is reduced because all log records are written to a local log file and commit processing is performed locally. On the other hand, the load of the server decreases and more clients can be supported.

In this chapter we have presented a new paradigm for distributed transaction processing. Our techniques have the potential to exploit every available system resource, and they improve system 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. The key advantages of our algorithms are the following.

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 node crash recovery are handled exclusively by the nodes.
3. When fine-granularity locking is employed, multiple nodes can concurrently update different portions of the same database page.
4. Node log files are never merged during the recovery process.

5. Node clocks do not have to be synchronized.
6. Each node can take a checkpoint without synchronizing with the rest of the nodes.

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

Chapter 4

Enabling Recovery in BeSS

In this chapter, we address the challenges that arise when implementing logging and crash recovery in a memory-mapped client-server environment. In particular, we present the implementation of recovery in the **B**ell **L**aboratories **S**torage **S**ystem (BeSS) [10, 7].

BeSS is a storage manager that facilitates the development of high-performance database management systems. The architecture of BeSS is not tailored to a specific data model or language. It is possible to build relational and object-oriented database systems and persistent languages on top of BeSS as well as home-grown specialized database systems. For example, BeSS is being used as the storage engine of the AT&T's Prospector [19], a content based multimedia system that requires an extended relational interface to BeSS.

The rest of the chapter is organized as follows. Section 4.1 describes the architecture of BeSS, including storage structures, pointer dereference, and issues related to preventing database corruption caused by bad pointers. In Section 4.2 we present implementation details, including interface, modes of operation, and cache replacement. Section 4.3 describes how recovery is implemented in BeSS and Section 4.4 compares BeSS with related storage systems.

4.1 The BeSS System Architecture

BeSS operates in a multi-client multi-server environment. A typical BeSS network configuration is illustrated in Figure 4.1. Some nodes, such as nodes 1 and 3, own databases while some others, such as nodes 2 and 4, are client workstations that do not own any database. There is a BeSS server process on every node, regardless whether there is a database attached to the node, similar to the peer-to-peer architecture of Shore [14]. The presence of a BeSS server process on every node not only makes inter-transaction caching for transactions within the same application process more efficient, it also enables sharing of data across transactions that are part of different application processes running on the same node.

An application running on a node can access the entire distributed database space by communicating only with the local BeSS server. The *two phase commit* (2PC) protocol is

employed for distributed commits, and timeouts are used for distributed deadlock detection. The strict two phase locking algorithm is used for concurrency control, and recovery is based on the ARIES [40] *write-ahead log* (WAL) protocol. Moreover, client-server interaction is minimized by caching data and locks between transactions running on the same client. Cache consistency is provided by employing the *callback locking* algorithm [33, 36], which has been shown to have good performance over a wide range of workloads [67, 17].

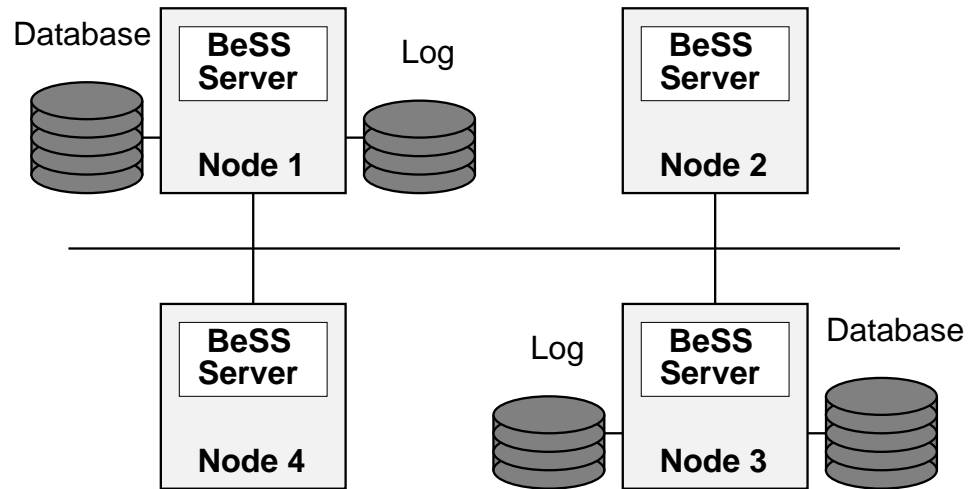


Figure 4.1: The BeSS distributed architecture

At the conceptual level, BeSS manipulates *databases* which are collections of BeSS *files*. BeSS files contain *object segments* in which objects are stored. An object segment is the clustering facility provided to users to indicate that some objects need to be co-located. BeSS files group objects so that they can be retrieved via a cursor mechanism. However, an individual object can be accessed directly without first accessing the file containing it.

At the physical level, the database consists of several *storage areas*, which are UNIX files or disk raw partitions. Storage areas, are partitioned into several *extents*, and allocation of disk segments from one of these extents is based on the binary buddy system, as described in [3]. Storage areas that correspond to UNIX files may expand in size by one extent at a time.

All objects in a BeSS file are stored on object segments that are allocated from one storage area. However, a storage area may contain objects belonging to multiple BeSS files. Initially all objects of all files in a database may be placed in the same storage area. To accommodate growth, objects within a BeSS file can be moved to another storage area, and as we shall see in the next section, without affecting existing object references.

4.1.1 Segment and Object Structures

Figure 4.2 illustrates the structure of an object segment. Each object segment consists of two parts: the *slotted segment* and the *data segment*, each of which is a sequence of one or

more contiguous pages. Slotted segments are allocated from one storage area, and they are never relocated. The slotted segment contains a fixed-size header and an array of slots. The header includes information required for managing the object segment, such as the number of objects and the available free space in the data segment.

The data segment contains the actual objects which can vary in size and consume no predetermined amount of space. For every object in the data segment, there is an object header that is stored in a slot in the slotted segment. The object header contains meta-information that is necessary for managing the object it refers to – such as a pointer to the object’s type (TP), a pointer to the object’s data in the data segment (DP), the object size, and other bookkeeping information. Type descriptors contain the offsets of pointers within the objects they describe. Data segments can be re-sized or moved to a different location in either the same or a different storage area.

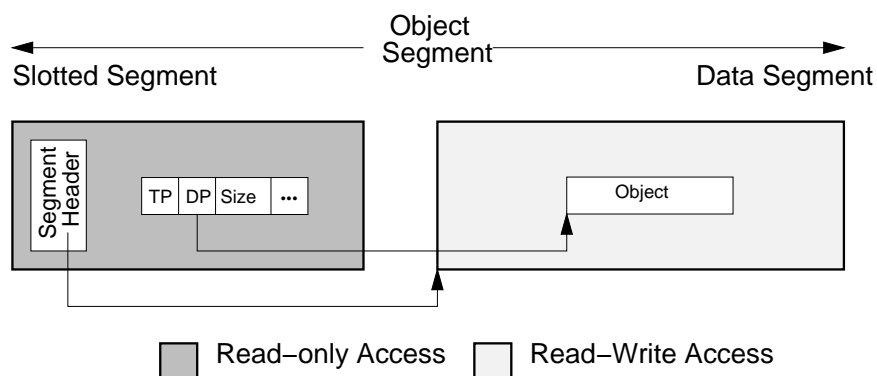


Figure 4.2: Segment and object structure

The object identifier (OID) is a 96-bit number that uniquely identifies an object in a BeSS system. Each OID contains a number to approximate unique oids. This number is stored in every slot and it is modified every time the slot is re-used. Slotted segments (and their slots) are allocated from one storage area and they are never relocated. Data segments can be re-sized or moved to a different location without affecting the validity of existing OIDs. It’s possible that huge databases may end up having one storage area exclusively for slotted segments while data segments are spread over multiple storage areas.

However, object references are not implemented through the rather long and slow OIDs. Instead, references among objects belonging to the same database are pointers to the headers (slots) of the referenced objects. References among objects belonging to different databases are implemented via a level of indirection: the reference points to a *forward* object, which is stored in the database of the referencing object, and it contains the complete address of the referenced object. Such inter-database references are handled transparently and exclusively by BeSS. After intra- or inter-database references are processed as discussed in the next paragraph, they point to the header of the referenced object, which in turn contains in its DP field the virtual memory address of data part of the referenced object.

To illustrate how the above structures are used, consider the actions taken when a *slotted segment fault* occurs. First, the slotted segment is fetched in memory, and a virtual

memory address range for its data segment is reserved and access-protected. Then, for every slot in the slotted segment representing an object in the data segment, the DP field of the slot – whose value is the address in which the object was mapped the last time it was accessed – is adjusted to point to the new (reserved) virtual memory address of the object in the data segment. This involves just two arithmetic operations.

Suppose now a *data segment fault* occurs when some object within a data segment, for which addresses have been reserved as discussed above, needs to be accessed. First, the data segment of the object is either fetched as a whole or only the pieces needed to access the object are fetched – this depends on the availability of cache space. Second, BeSS examines the type descriptor of every object O that is present in the portion of the data segment that was actually fetched, and it locates all references contained in O . For each reference to an object O_i , BeSS performs the following actions. If the slotted segment containing O_i has never been referenced before in the current transaction, an address range for the slotted segment is reserved and access-protected. (If it has been referenced before, the slotted segment is either in memory or virtual memory addresses have been reserved for it.) Then, O is modified to point to the virtual memory address of the header of O_i . When later on O_i is accessed, there will be a slotted segment fault which will trigger the actions described in the previous paragraph.

Thus, accessing an object potentially causes actions in three waves. In the first wave, address ranges for the referenced slotted segments are reserved. In the second wave, as some of these slotted segments are accessed for the first time, slotted segments are fetched in and address ranges for the corresponding data segments are reserved. Finally, accessing some objects within one of these data segments causes the data segments to be fetched. The latter may trigger another round of virtual memory address reservation and data fetch.

The advantages of the inter-object reference scheme and the segment and object structures employed by BeSS are the following:

- Databases can be re-organized on the fly without affecting object references. Reorganization includes compaction, resizing, or relocation of data segments and movement of entire files between storage areas.
- Memory address space is reserved in a less greedy fashion than the schemes presented in [36, 61, 69]. In BeSS, virtual address space for data segments is reserved only when the corresponding slotted segments are actually accessed.
- Persistent objects are manipulated directly in the segment on which they reside, without incurring any in-memory copying cost.

Regarding large objects, BeSS offers transparent access to fixed-size objects that can fit in a data segment – currently, up to 64KB. For “very” large objects, BeSS offers a class interface that includes byte range operations – such as **read**, **write**, **insert**, **delete** a number of bytes starting at some arbitrary byte position within the object, and **append** bytes at the end of the object. In anticipation of object growth, hints about the potential size of the object can be provided by the user. The large object is stored in a sequence of

variable-size segments indexed by a tree structure [3, 4], and the root of the tree is placed in the overflow segment.

4.1.2 Preventing Database Corruption

Since object references are virtual memory addresses, user code has direct access to BeSS control structures, such as slotted segments. Hence, mechanisms to prevent database corruption caused by bad pointers are of paramount importance. BeSS utilizes the standard facilities provided by the underlying hardware for detecting access protection violations. The virtual memory management hardware detects an illegal attempt to update write-protected items at the time the update is attempted, before the possible error takes place and propagates to other structures.

As shown in Figure 4.2, the slotted segment is mapped into write-protected virtual memory and thus ordinary user code cannot modify this memory section. Data segments are readable and potentially writable by user code. Before BeSS (or some other trustworthy software) updates critical control structures, it explicitly unprotects the address space containing these structures, and it reprotects the address space after the update. This scheme allows correct software to modify protected data but prevents accidental database updates by incorrect pointers. However, if malicious software manages to unprotect memory before updating it, this protection mechanism alone would not work. The major cost associated with this mechanism is an increased number of system calls [63], which for many applications is an acceptable tradeoff for the benefits gained.

4.2 Implementation Details

4.2.1 Interface

Object retrieval is implicit, via dereference¹, using a number of BeSS typed references that are based on the ODMG-93 standard [20]. For example, the C++ class `ref<T>` encapsulates a pointer to an object header as discussed in section 4.1.1. It is parameterized with the type of the pointer for which its instances can be substituted. Given a type, say, `Person` with members `name` and `spouse`, the instantiated type `ref<Person>` behaves like a pointer of type `Person`. So, a variable `p` of type `ref<Person>` can be used as if it were of type `Person*` (e.g., one can use `p->spouse->name` to refer to the `name` member of the `p->spouse` object, or pass `p` to a function argument expecting a `Person*` variable).

An object may also be retrieved explicitly by supplying to BeSS the name of the object and a pointer to the database where the object is stored. Any BeSS object can be given a name. For such so called “named” or “root” objects, BeSS maintains a directory which is implemented as a pair of hash tables. BeSS enforces the referential integrity between root objects and their names. When a root object is removed from a database so is the name

¹We use the term *dereference* for any indirect access – either through dereference operators such as `*` and `->` in C and C++, or through indexing an array or pointer variable.

of the object. Also, explicit retrieval can be performed using the class `global_ref<T>` that encapsulates an OID but access via this mechanism is somewhat slower compared to the one described in the previous paragraph.

Finally, new objects in BeSS are created by a number of overloaded functions. These functions require the size and a pointer to the type descriptor of the object being created, as well as where the object should be created – in a database, in a specific file, or in a specific object segment. They return a pointer to the object header of the newly created object, which may then be casted to the appropriate type (For example, in C++ this can be done automatically by overloading the `new` operator).

4.2.2 Operation Modes

Each BeSS server manages a cache (shown in Figure 4.3) which is used to store all pages accessed by local transactions. These pages may belong to databases managed by this server, if any, and databases managed by remote servers. The cache is created using the `mmap` UNIX system call and it is viewed as a contiguous sequence of cache frames, each of which can hold a database page. The server is responsible for forwarding to remote server requests made by local applications (e.g., for fetching remote data), and, symmetrically, it processes requests from remote servers (e.g., callback requests).

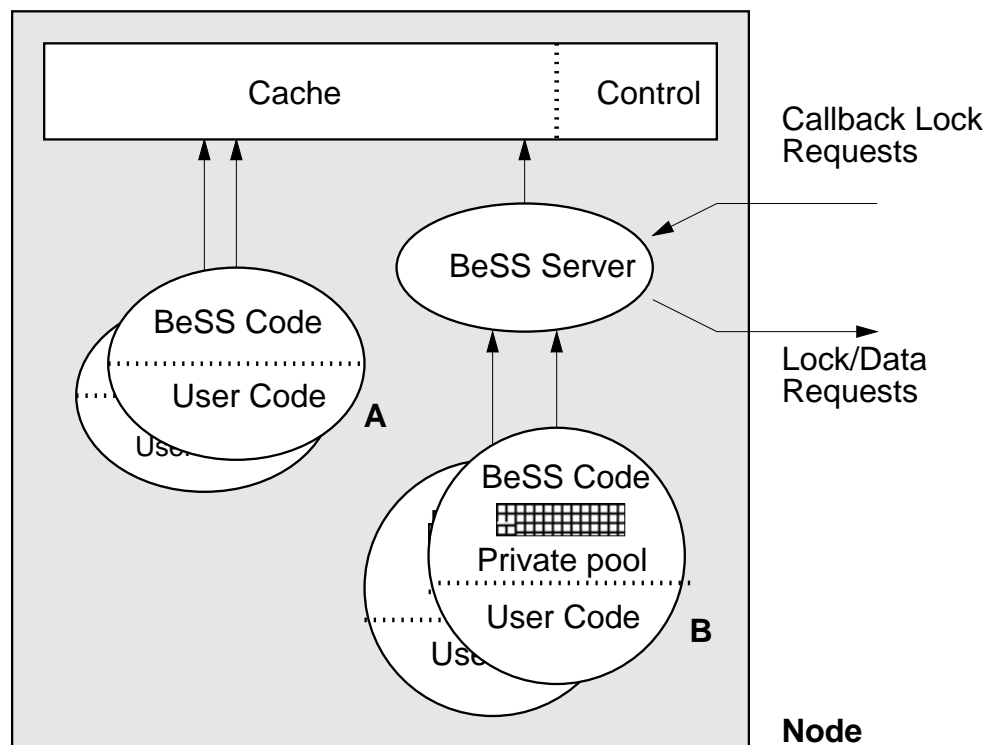


Figure 4.3: Shared memory established by the node server

A user process can access the shared cache either directly (*in-place access* or *shared*

memory) or indirectly through the server (*copy on access*). In the former case, each process gains access to the shared cache and all control data by mapping the cache, which has been established by the server, into its address space. In the latter case, each process maintains a private cache (Figure 4.3, application B) and communication with the local BeSS server is required for fetching segments. If the segment is not present in the shared cache, the server will fetch it from the appropriate BeSS server. This private cache of each process is implemented as a fixed size file divided into frames whose size is equal to the BeSS page size. The above file is mapped into the process' virtual address space using the UNIX `mmap` system call.

Copy on access has the advantage that user processes do not need to synchronize their accesses to their private caches, but inter-process communication is expensive. In-place access offers the potential for high performance, especially for short transactions, since it avoids interprocess communication and the cost of copying data to a private space and back to the cache. However, it incurs the cost of synchronizing concurrent access to the shared cache. The shared memory mode enables sophisticated users with well tested and debugged code to tailor the storage system and build multiple specialized servers, such as multimedia servers. Note also that the interface provided by the node server is the same in both modes, it is just the process boundaries that differ.

In the shared memory mode, apart from the problem of synchronizing concurrent accesses to the shared cache, pointers between database objects and their control structures, pointers among the control structures, and pointers among database objects must be valid to every application process accessing them. BeSS uses latches (atomic test-and-set) for synchronizing concurrent accesses to the shared cache. Clean-up of shared structures from process failures is handled by keeping track of process actions as in [39].

BeSS ensures the validity of the shared pointers by treating them in a uniform way as offsets from the beginning of a fictitious virtual address space, as it is outlined below. Each process maps the shared cache in a number of frames – each having size equal to database page – in the process' private virtual memory address space, referred to as PVMA. The size of PVMA may be much larger than the size of the shared cache. However, for our scheme to work all processes must reserve the same number of PVMA frames. Mapping of database pages to virtual frames is performed via a mapping table which is *shared* by all processes. This means that if a process maps a page at some frame, all processes see this page at this frame (but possibly at different addresses).

Thus, pointers in the shared space are made valid by: a) mapping each database page fetched in the shared cache to the same PVMA frame for all processes, and b) using offsets instead of virtual memory pointers. The shared mapping table in conjunction with the use of offsets gives the illusion of a shared virtual address space, referred to as SVMA. Note also, that in this scheme a pointer needs to be fixed once by the first process that fetched the corresponding page in cache. A simple BeSS template class translates pointers from the process's virtual address space to pointers in the shared address space, and vice versa.

4.2.3 Cache Replacement

Cache replacement is based on a clock-like algorithm [26]. However, BeSS does not implement the traditional clock algorithm where a bit indicates whether a slot has been accessed since the last time the clock swept over this slot. This is because the cache manager does not have enough information indicating which slots have been accessed recently due to the memory mapping architecture.

BeSS' clock algorithm is based on the state of a virtual frame. Each virtual frame may be either *invalid*, or *protected*, or *accessible*. A frame is invalid when it is access-protected and it is not mapped to any cache slot. A frame is protected when it is access-protected and it is mapped to a cache slot. Finally, a frame is accessible when it can be accessed by the application without causing an access violation. An accessible frame is always mapped to a cache slot. The clock algorithm sweeps through the virtual frames and skips all invalid frames. Accessible frames are also skipped but after they are converted to protected. The cache slot corresponding to a protected frame is selected for replacement in the copy on access mode.

In the shared memory mode, however, the cache slot of a protected frame cannot be unilaterally replaced because it may be accessed by other processes. BeSS associates a counter with each cache slot. This counter corresponds to the number of processes that can access the slot. Each process increments the counter of a slot when the process gains access to the slot. In addition, the clock algorithm is broken up in two levels. The first level is the same as the clock algorithm in the copy on access mode with the difference that the protected frames are made invalid, and the counter of the slot they correspond to is decremented by one. The second level operates on the cache slots and uses the counter as an indication whether the slot has been accessed since the last time the clock swept over it. A cache slot with counter zero is selected for replacement.

4.3 Recovery in BeSS

Memory-mapped systems allow applications to update objects by dereferencing virtual memory pointers. This approach allows applications to update objects at memory speeds with no extra overhead. However, this approach makes detecting the portions of the object that have been updated more difficult than traditional approaches. In this section, we describe how BeSS identifies the portions that have been updated and how log records are being generated.

4.3.1 Detecting Updates

In all the existing relational DBMSs and several OODBMSs, updates to the database are carried out by invoking functions provided by the storage subsystem and exported to higher level software components. Under this software approach for dealing with updates, each function acquires the appropriate locks and generates detailed recovery information (i.e.,

log records). For OODBMSs, the compiler of the programming language used to access persistent objects has to be modified so that it generates calls to the underlying storage system just before an object might become dirty (e.g., during an assignment). For instance, the compilers for E [58] and Ode [6, 5], which are both extensions of C++, generate such calls to the their storage systems, Exodus [57] and EOS [11], respectively.

Similar to other memory-mapped storage systems, BeSS detects updates by using the standard virtual memory management facilities provided by the underlying hardware. When a database page is fetched in memory it is write-protected. An attempt to modify this page signals a protection violation and a fault handler is invoked. The fault handler acquires locks, records the update, and unprotects the page for future modifications. The hardware approach offers a transparent, automatic, and fast way of detecting updates. The overhead associated with updates (locking and logging) occurs exactly once, when the page is updated for the first time. Any subsequent updates proceed at full speed. This is important for many applications, such as CAD, that typically work on many objects by repeatedly traversing relationships between these objects and updating some of them as well. However, there is one caveat. The scheme works only for granules that are integral multiples of the page size used by the virtual memory system - usually 4 K-bytes. An implication of this to recovery is that the modified regions on a page cannot be detected.

Generating a log record for the entire page (i.e., assuming the whole page is dirty) may have severe implications in the performance of the system, as shown in [70]. For this reason, BeSS follows the *page diffing* [61, 32, 70] approach, which presents an efficient way to locate modified portions on a page. Under the page diffing approach, the updated portions of a page are identified by comparing the updated copy of the page against a *clean* copy of the same page. The following section describes the implementation of page diffing in BeSS.

4.3.2 Generating Log Records

As we mentioned in Section 4.2.2, the virtual address space of a process can be viewed as a sequence of virtual frames of size equal to the cache frame size. BeSS maintains a mapping from virtual frames to physical cache frames, as explained below and illustrated in Figure 4.4. Let us assume that a transaction needs to access page A that happens to be in cache frame 0, perhaps fetched by another transaction, as shown in Figure 4.4 (a). Some virtual frame, say frame 0, is mapped to the cache frame 0, pointers within virtual frame 0 are fixed if needed, and the application gets read (but not write) permission to virtual frame 0. The transaction keeps accessing pages in the fashion described above until the number of accessed pages reaches the capacity of the cache, as shown in Figure 4.4 (b) where the transaction has mapped its virtual frames 0, 1, 2, and 3 to cache frames 0, 2, 3, and 1, respectively. The transaction then needs access to page E that is not currently in the cache. As shown in Figure 4.4 (c), page A is evicted from cache frame 0 and the virtual frame 0 is access protected; page E is fetched in cache frame 0, the virtual frame 4 is mapped to this cache frame, and the application gets read access permission to this virtual frame. The cache-to-virtual frame mapping is dynamic in the sense that the same virtual address frame may be mapped to different physical frames during the execution of a transaction.

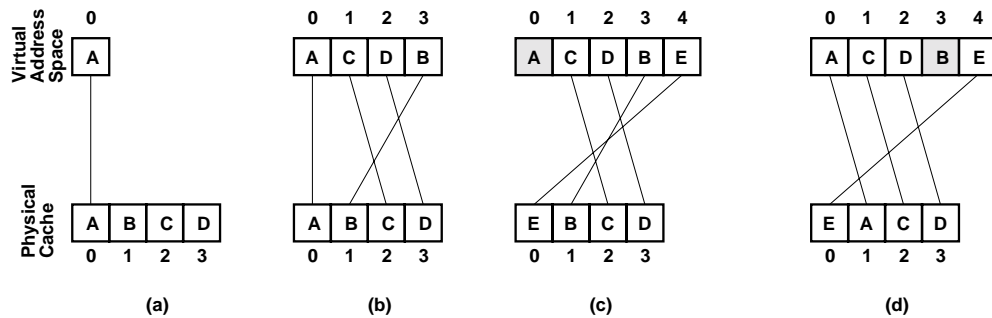


Figure 4.4: The BeSS memory mapping approach

For instance, if page A needs to be brought back in cache, it may be placed in any cache frame and the virtual frame 0 will be mapped to that new cache frame, see Figure 4.4 (d).

When a page is mapped in the application’s virtual space for the first time, a page descriptor is created. The page descriptor contains information about the mapped address of the page as well as the access permissions and it is inserted into a balanced tree, whose key is the virtual memory address of the page. When a write violation occurs in some virtual address, the BeSS fault handler is invoked. The fault handler first makes sure that the address of the fault is within some virtual frame F handled by BeSS – if not, the fault propagates to the application. Then, based on the address of F , the above tree is searched to locate the descriptor of the page P that corresponds to F so appropriate actions for recovery can take place. The following cases need to be considered:

1. F is access protected and P is not cached. P is fetched in the cache as explained by the example in Figure 4.4, and the steps described in case 3 below are taken.
2. F is access protected because of the normal activity of the cache replacement mechanism, and P is cached. The mapping between P and F is re-established, F is write-protected, and control is returned to the application process.
3. F is write-protected and P is cached. The current image of P is copied into a separate space, referred to as *recovery buffer*, and the descriptor of P is made to point to this recovery buffer. Write access is enabled for F and control is returned to the application process.

Log records are generated when the recovery buffer is full, or when an updated page is replaced from the client cache, or at transaction commit. The copy of the updated page in the recovery buffer is compared against the copy of the page in the cache for determining the regions that are different. Next, the before and after image of each modified region are used to generate the log record for the page. This log record is then inserted in an in-memory log buffer. The log buffer is flushed to the log at commit time.

4.4 Related Work

Existing storage systems prevent database corruption by providing a function call interface to the client applications (e.g., Exodus [15]). On the other hand, BeSS offers direct access to objects and utilizes the standard virtual memory facilities provided by hardware to detect access protection violations. BeSS also uses the underlying hardware to automatically detect writes, as in ObjectStore [36] and QuickStore [69].

Objects often contain references to other objects. Usually, the on-disk representation of an object differs from the in-memory representation of the object because of the way the storage manager treats inter-object references. Some systems use object identifiers (OIDs) for both the in-memory and on-disk representations [11]. Other systems use virtual memory pointers for both representations [60, 12, 36, 69]. Finally, there are systems that use OIDs for the on-disk representation and virtual memory pointers for the in-memory representation. These systems convert the OIDs to virtual memory pointers when they fetch the objects from disk – this is referred to as *swizzling* [45, 8]. The conversion can be done either in software [68, 44, 24, 50, 49] or by using the facilities provided by the underlying hardware [61, 72]. In BeSS, inter-object references are represented by virtual memory pointers to the headers of the referenced objects which in turn contain the address of the object itself. Also, since BeSS stores object headers in a different segment than the object data, segments containing object data can be moved around without affecting the inter-object references.

ObjectStore [36] is a commercial object-oriented database management system based on a memory mapped architecture. However, ObjectStore does not allow application processes to operate directly on objects that reside in the shared client cache. Instead, applications have to copy the items they need into their private space. Similar to BeSS, ObjectStore offers inter-transaction caching and automatic detection of updates. Unlike BeSS, ObjectStore does whole page logging and stores all pages modified by a transaction to the log.

QuickStore [69] is a memory-mapped client-server storage system for persistent C++ implemented on top of Exodus. Similar to BeSS, QuickStore uses page diffing for generating log records. However, the page diffing approach in QuickStore generates one log record for each modified object present in each updated page, while BeSS generates only one log record for each updated page. Another difference is that QuickStore does not support inter-transaction caching.

Texas [61] is an object-oriented storage system that adds persistence to the C++ programming language. Similar to BeSS and QuickStore, Texas is based on memory mapping, and it utilizes the page diffing scheme for generating log records. Unlike BeSS and QuickStore, Texas is a single user system and no concurrency control mechanism is supported. In addition, Texas limits the amount of data that can be accessed during a single transaction to the size of the swap space backing the application since it allocates space for pages in virtual memory. Unlike BeSS, Texas installs the updates made by an application to the database by scanning the log records generated for updates to the database and applying these log records.

BeSS differs in many respects from our previous work in the context of EOS [11, 9], although it includes components of EOS that according to our experience have worked very well – disk space allocation and large objects are some of them. First, pointer dereference in EOS is somewhat slow because inter-object references are OIDs. BeSS offers a fast pointer dereference mechanism by using virtual memory pointers. Second, object relocation in EOS is a tedious task because OIDs are physical addresses. On the other hand, BeSS uses one level of indirection which facilitates on the fly database reorganization – compaction, resizing, and relocation of data segments – without affecting existing references. Third, unlike EOS that requires explicit calls for setting locks, BeSS utilizes the virtual memory mechanisms provided by ordinary hardware for guarding against software errors, such as stray pointers, and for automatic lock acquisition. Fourth, BeSS is an open-server system and user code can be linked with the BeSS server to build application specific servers, as opposed to running the application as a client, the only alternative offered by EOS. Finally, BeSS offers to application processes running on the same machine the capability of accessing data in a shared cache.

Chapter 5

The Performance of Client-Based Logging

In this chapter we present a study of four recovery algorithms for a data-shipping client-server system. The algorithms were implemented in BeSS, and their relative performance was studied using the OO1 benchmark [21].

These recovery algorithms, which are based on the ARIES [40] redo-undo protocol, belong to two categories: *server-based logging* and *client-based logging*. In the former category, the log is stored with the server and all log records generated by transactions running on clients are sent to the server. In the latter category, which corresponds to the algorithms presented in Chapter 3, each client has its own log that contains records generated by locally executed transactions [51, 52]. For each of the above two categories, we examined two cache coherency protocols: read-callback and write-callback.

The remainder of the chapter is organized as follows. Section 5.1 describes in details the four recovery algorithms we study in this chapter. Section 5.2 compares the relative performance of these schemes using the OO1 benchmark. Section 5.3 compares our work with relevant work that appears in the literature and, finally, we present our conclusions in Section 5.4. A queuing model that can be used for extrapolations is presented in the Appendix.

5.1 Alternative Logging and Recovery Algorithms

In this section we describe the recovery algorithms we implemented and evaluated, which are server-based and client-based logging each combined with two cache coherency protocols: read-callback and write-callback.

When the read-callback protocol is employed, all updated pages that are present in the local buffer pool of a client at transaction commit are sent back to the server. Exclusive locks that have been acquired by the committing transaction are demoted to shared. For the write-callback protocol we consider in this paper, we assume that modified pages are

Callback Algorithm	Logging Site	
	<i>Server</i>	<i>Client</i>
<i>Read</i>	SL-RCB	CL-RCB
<i>Write</i>	SL-WCB	CL-WCB

Table 5.1: The different recovery algorithms studied

not sent to the server at transaction commit, nor are exclusive locks demoted to shared.

Transactions are executed in their entirety on the client where they are started. Data items referenced by a transaction are fetched from the server before they are accessed. All recovery schemes are based on the ARIES redo-undo recovery protocol and clients generate log records by using the page diffing technique. In addition, the standard two-phase locking algorithm is assumed for concurrency with the minimum locking granularity being a page.

Table 5.1 shows the names that we use in the rest of the paper for the algorithms we examine. We begin our discussion by describing the ARIES redo-undo recovery algorithm in BeSS when the log file is stored with the server and the read callback algorithm is employed, referred to as SL-RCB. The rest of the algorithms are built on top of the SL-RCB and we present the modifications that were made to the SL-RCB for each one of them in the next sections.

5.1.1 Server-Based Logging Read-Callback (SL-RCB)

The SL-RCB recovery scheme is based on the Exodus implementation of ARIES, as described in [29]. The server maintains a circular, append-only log file and uses the *steal* and *no-force* buffer management strategies [31]. As mentioned above, log records are generated at the clients using the page-diffing technique and they are sent to the server just before any updated page is sent back to the server or during transaction commit, depending on which event happens earlier. At transaction commit, all updated pages present in the client buffer pool are sent back to the server and the exclusive locks are demoted to shared.

Correct restart recovery is based on the ability to determine all the pages that were dirty at the time of the server crash. In SL-RCB, the server maintains a dirty page table (DPT) which includes entries for both the dirty pages that are present in the server’s cache and the updated pages present in the buffer pool of each active client, avoiding the synchronous communication with the clients required by ARIES/CSA during checkpointing. This approach also differs from the approach taken in Exodus. In Exodus, the DPT of the server includes entries for only the dirty pages that are present in the server’s cache. Information about the dirty pages present in the cache of a client is saved in the log by sending the list of pages updated by the committed transaction to the server at transaction commit.

DPT is maintained by analyzing the log records the clients send to the server. In

particular, the server examines the header of each log record it receives to find information about the transaction T that generated the record and the page P for which the record was written. This information is used for updating the transaction table and the DPT, as well as the header of the log record by assigning to the field that refers to the previous log record written by T the actual offset in the log file of this record. If T does not have an entry in the transaction table, a new entry is inserted and the *First* and *LastLSN* fields of this entry are set to the LSN assigned to the log record. If P does not have an entry in the DPT, a new entry is inserted and the *RecLSN* for P is set to the LSN assigned to the log record.

Transaction abort in SL-RCB can be carried out by both the server and the client. In the former case, the client purges from its cache all pages that were updated by the aborted transaction and the server undoes the updates by traversing the transaction's log records in reverse chronological order¹. In the latter case, the client will send any remaining log records to the server. Next, the server will send to the client the last log record written by the aborted transaction and the client will undo the updates by scanning the remaining log records in reverse order². If a page is not present in the client buffer pool, the page is fetched from the server. For the remaining of the chapter we will assume that transaction rollback is performed by the server.

5.1.2 Server-Based Logging Write-Callback (SL-WCB)

The SL-WCB is a modification of the SL-RCB algorithm in which clients do not send dirty pages back to the server at transaction commit, nor do they demote exclusive locks to shared. Not having to send updated pages back to the server at commit results in substantial saving when the number of updated pages is rather large. These savings have a positive impact on both transaction response time and system scalability since the load placed on the server by each client is reduced.

Another difference between SL-WCB and SL-RCB is the handling of lock callbacks. Under SL-RCB, the server always has the latest committed version of a page in its cache or on disk and clients do not have to send any pages when responding to callbacks. On the other hand, a SL-WCB client sends a dirty page present in its cache back to the server when the client has an exclusive lock cached for this page and the server requests a callback.

However, the most important difference between the two schemes is the handling of transaction aborts. While under SL-RCB a client can purge the pages updated by an aborted transaction from its buffer pool freely, doing so under SL-WCB may result in substantial performance degradation. Since dirty pages are not shipped to the server at transaction commit, the copy of a page in the server's cache or on disk may not contain

¹Since the page diffing approach corresponds to physical before-after image logging, a log record may be applied multiple times to the same page in an idempotent fashion. Consequently, the undo of a log record during transaction rollback and system restart is not a conditional operation, as is in Exodus.

²Since the server alters the headers of the log records and assigns actual offsets in the log file, the client can ask for the previous log record written by the aborted transaction by supplying the LSN present in the *PrevLSN* field of the log record. Thus, the server does not have to maintain any mapping between the LSNs assigned by the clients and the actual LSNs of the log records, as it is done in ARIES/CSA.

the updates of all transaction that have committed in the past. As a result, purging an updated page from the client's buffer pool may result in having the server redo committed updates before rolling back the aborted transaction.

The solution to this problem is to require from the clients to ship the pages updated by the aborted transactions to the server, before they purge these pages from their local buffer pools. We should note that the server would have to redo committed updates when a client crash occurs and the server does not have the latest committed versions of some pages present in the buffer pool of the client at the time of the crash.

5.1.3 Client-Based Logging Read-Callback (CL-RCB)

In this section we present the implementation in BeSS of the client-based logging algorithm presented in [51]. Under CL-RCB, each client has a local log file where log records for updates to cached pages are written. The server recovery component of the SL-RCB algorithm was used as the basis of the client and server recovery subsystems. Similar to SL-RCB, all updated pages are sent back to the server at transaction commit and exclusive locks at demoted to shared. Transaction rollback is handled by each client using the log records present in its private log. Pages that need to be recovered and which are not present in the client buffer pool are fetched from the server.

Similar to the previous two algorithms, the server in CL-RCB maintains a DPT whose entries corresponds to clients and pages that have been updated by these clients. Unlike the SL-RCB, the server does not examine any log records written by the clients and new entries are inserted in the DPT when clients acquire exclusive locks for pages. When the server writes a dirty page to disk, it removes from the table all entries for clients that either do not hold a lock on the page or hold a shared lock on it. In addition, the server notifies each of these clients by piggybacking the page id in the message header of the next reply to be sent to the client.

Each client also maintains a DPT which is stored in the log file of the client during checkpointing. Each such DPT contains entries for the dirty pages present in the client buffer pool and for the pages that have been updated by this client in the past and not yet written to stable storage. An entry for a page is removed from the DPT when the client is notified by the server that the page has been forced to disk.

When a client crashes, the server releases all shared locks held by that client. The crashed client needs to perform restart recovery only if it held exclusive locks at the time of the crash. In this case, the server will forward to the client the list of all exclusive locks held by it, together with the entries this client has in the server's DPT. Next, the client will apply the same restart logic as the SL-RCB algorithm with the difference that only log records corresponding to pages that were exclusively locked at the time of the crash are examined and the corresponding pages are fetched from the server.

5.1.4 Client-Based Logging Write-Callback (CL-WCB)

Although CL-WCB is very similar to CL-RCB, the two schemes have a number of differences, besides the callback algorithm they use. The most important difference is the way client crashes are handled. Since dirty pages are not sent back to the server at transaction commit, committed updates may have to be redone during client restart recovery. This is because a given page may have been updated by a number of transactions running on the same client before the occurrence of the failure. Details on how client crash recovery is handled in this case can be found in Section 3.3.2.

5.2 Performance Study

In this section, we present the results of the study we conducted to compare the performance of the recovery algorithms presented in Section 5.1. In this study, we used two databases of different sizes and three different operation sets in order to investigate the performance tradeoffs between the two callback algorithms and the relative merits of client-based logging.

All experiments were run on SPARCstation 10s running SunOS 4.1.3. The clients and server processes were run on separate machines and they were connected via an Ethernet network. Each client machine had 32 M-bytes of main memory and 142 M-bytes swap space. The server machine had 80 M-bytes main memory and 320 M-bytes swap space. Each machine had attached to it one Seagate/Baracuda disk drive of size 2 G-bytes and external transfer rate of 10 M-bytes per second. This disk was used to store the log of the server and the local log of each client. In addition, the server machine had a second Seagate/Baracuda disk drive of size 4 G-bytes and external transfer rate of 10 M-bytes per second. This disk was used to store the two databases. Each database was stored in a raw disk partition and the database page size was 4 K-bytes. The client the server log files were stored in regular UNIX files and `fsync()` system call was used at transaction commit for flushing any internal operating system buffers to disk. All times reported were obtained by using the `gettimeofday()` and `getrusage()` UNIX system calls and include the time to start a transaction as well as the time to commit it.

5.2.1 Database and System Model

A modified version of the OO1 benchmark [21] was used as the basis for all the experiments. We used two different database sizes in the study, referred to as *small* and *large*. Table 5.2 shows the size characteristics of these databases. Each database consists of 6 modules, each of which consists of a number of part objects, each having size equal to 128 bytes. Each part object is connected to exactly three other objects. In order to be able to traverse all objects, one connection is initially added to each object to connect the objects in a ring; the other two connections are added at random. Furthermore, one of the part objects serves as the root of the object hierarchy and it is given a name so that it can be retrieved from the database at the beginning of each experiment.

Database Name	Number of Modules	Objects per Module	Module Size M-bytes	Total Size M-bytes
<i>Small</i>	6	20,000	2.5	15.0
<i>Large</i>	6	200,000	25.0	150.0

Table 5.2: The configuration of the two databases

The experiments were performed by using three different traversals, referred to as *UpdateOne*, *UpdateAll*, and *UpdateRepeat*. All traversals retrieve the root part object of a particular module and they visit all part objects in that module. While the *UpdateOne* traversal updates only the very first part object, the *UpdateAll* traversal updates all part objects of each module, and the *UpdateRepeat* updates all part objects of each module four times. The purpose of the *UpdateRepeat* is to generate a large number of log records so that the logging overhead is increased and the differences between the server-based logging and client-based logging algorithms become more evident.

In order to avoid performance degradation due to lock conflicts and deadlocks, we had each client access a different module in the database. Thus, the number of clients in all experiments was varied from 1 to 6. During each experiment, each traversal was run as a separate transaction, which was run repeatedly so that the steady state performance of the system could be observed. In addition, due to inter-transaction caching, the server cache and the buffer pool of each client were not flushed between transactions.

Each client was given 8 M-bytes of buffer space and the server's cache was set to 32 M-bytes. In all the experiments we fixed the size of the recovery buffer used for the generation of log records by each client to 2 M-bytes, for we were not interested in measuring the overhead of logging when different sizes are used. The reader is referred to [70] for a detailed discussion regarding the tradeoffs among different size recovery buffers.

The following two sections present the collected results of all experiments. First, we concentrate on the results for the small database and then we analyze the results for the large database.

5.2.2 Small Database

This section presents the performance results collected for the three traversal operations using the small database. Since the cache of each client is 8 M-bytes and the size of each module is 2.5 M-bytes, page replacement does not take place in the client cache during the execution of each transaction. Furthermore, the server's cache is large enough (32 M-bytes) and fits all 6 modules and, thus, no paging activity is present in the server's cache either.

Figure 5.1 shows the response time and throughput versus the number of active clients for the *UpdateOne* traversal. The number of objects updated during the *UpdateOne* traversal is one and, thus, only one log record of size 84 bytes is generated by each algorithm.

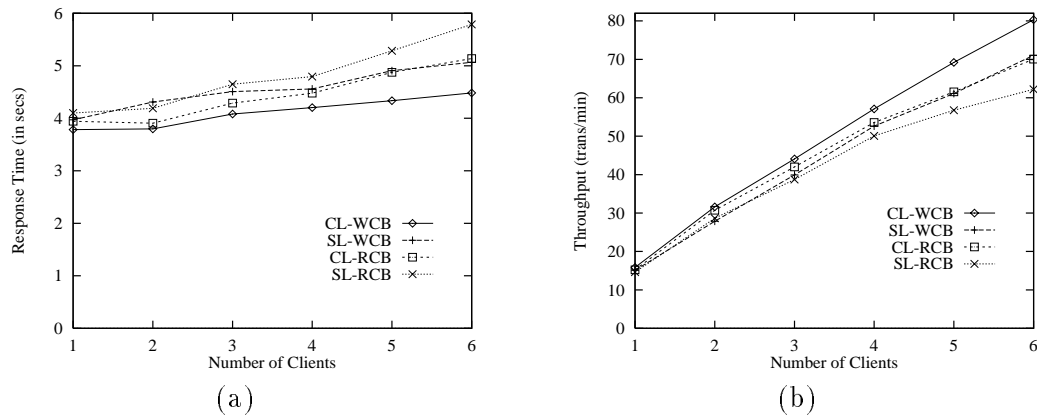


Figure 5.1: UpdateOne: small database (a) Response (b) Throughput

In addition, both SL-RCB and CL-RCB send one page back to the server at transaction commit and they also demote the exclusive lock on that page to shared.

The performance of both write callback algorithms is better than the performance of the read callback algorithms, independent of the number clients that are present in the system. This behavior was expected since both SL-RCB and CL-RCB require synchronous communication with the server during the execution of each transaction for acquiring an exclusive lock on the page containing the object that is going to be updated, besides the cost of shipping the updated page back to the server at commit. As the number of clients in the system increases, the performance difference between the read and write callback algorithms increases as well. This is because the server has to handle more data for the former algorithms and has to search its cache for placing the shipped page and its lock tables for locating and demoting the exclusive lock held on the page.

Figure 5.1 also shows that CL-RCB and CL-WCB offer better performance than SL-RCB and SL-WCB, respectively. In particular, CL-RCB offers 11.2% better transaction response time than SL-RCB and CL-WCB is 11.5% faster than SL-WCB when six clients are present in the system. This difference in performance is due to the increased number of log records the server has to handle and the fact that the log disk has to be synchronized, by calling `fsync()`, for each transaction commit message it receives from a client.

We now turn our attention to the UpdateAll traversal operation. Figure 5.2 shows the response time and throughput versus the number of active clients. Under this operation, each transaction updates all 20,000 objects that constitute each module, regardless of the recovery scheme that is employed. This is translated into updating 625 database pages. The number of log records each transaction generates is the same as the number of pages the transaction updates and the total size of these log records is 388 K-bytes.

CL-WCB has the best performance and it is followed by SL-WCB, which is 17% slower when six clients are active in the system. SL-WCB is more expensive in this case because the server log becomes a limiting factor when the number of clients increases. Unlike the

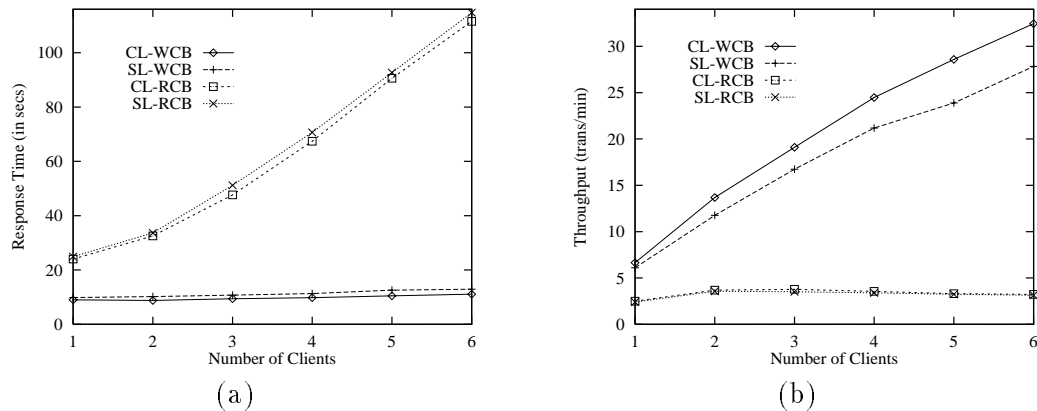


Figure 5.2: UpdateAll: small database (a) Response (b) Throughput

CL-WCB scheme whose performance remains unaffected by the number of active clients, the performance of SL-WCB degrades as the number of clients increases. Under the CL-WCB scheme, each transaction writes the same number of log records to the client log, independently of the number of active clients, and does not interact with the server at all. On the other hand, in SL-WCB each transaction has to send its log records to the server and, consequently, the logging activity of the server increases proportionally to the number of active clients.

The performance of both read callback algorithms is dramatically worse than the performance of the write callback algorithms. Under the CL-RCB and SL-RCB schemes, each transaction ships 625 dirty pages (2.5 M-bytes) to the server at commit. For each of these pages the server has to search its cache to either locate a previous copy of the page or find an empty slot. As a result, the performance of the server becomes a bottleneck. Even though the performance of CL-RCB and SL-RCB is very close, CL-RCB has better scalability characteristics. Transaction throughput starts decreasing after three clients for CL-RCB, as opposed to two clients for the SL-RCB scheme. This advantage is due to the fact that CL-RCB reduces the load of the server by avoiding to send 388 K-bytes of log data to the log file kept with the server.

The last experiment we conducted for the small database is the UpdateRepeat traversal. The purpose of this traversal was to study the scalability characteristics of each of the algorithms when the logging activity in the system is increased. However, since the UpdateRepeat differs from the UpdateAll in that each object is updated four times instead of one, the page diffing scheme would generate the same number of log records for both traversals. To avoid that, we altered the page diffing scheme so that instead of generating one log record for each updated page, four log records were created. These log records were identical.

Figure 5.3 shows the response time and throughput versus the number of active clients for the UpdateRepeat traversal. By comparing the results in Figure 5.3 against the results

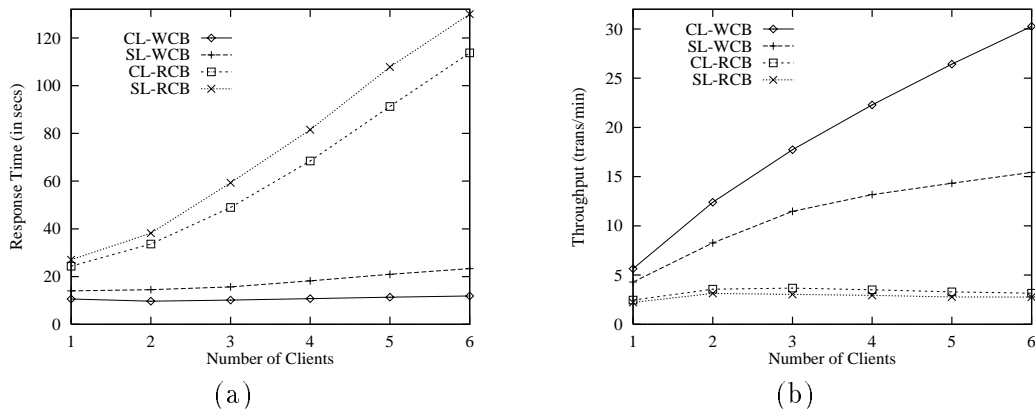


Figure 5.3: UpdateRepeat: small database (a) Response (b) Throughput

in Figure 5.2, we can see that the difference between the CL-WCB and SL-WRB schemes has increased from 17% to 96%. This dramatic improvement in transaction throughput is because CL-WCB eliminates the server’s log disk bottleneck by storing all 1.5 M-bytes of log records produced by each transaction in the local log file.

It is not a surprise that transaction throughput for both CL-RCB and SL-RCB is lower in the UpdateRepeat than the UpdateAll traversal. While the number of pages that are shipped to the server by each transaction during commit remains the same for both traversals, four times more log records are generated during the UpdateRepeat traversal. Interestingly, CL-RCB still performs better than SL-RCB and shows transaction throughput increase when three clients are active. This is because CL-RCB lessens the burden placed on the server’s log disk and CPU by storing all log records locally.

5.2.3 Large Database

This section contains the results of the experiments using the large database. Each module in the large database is 15 M-bytes in size and it does not fit in the 8 M-byte cache of each client. Consequently, all traversals experience paging activity in the client cache. Furthermore, since the cache of the server is 32 M-bytes, the server experiences paging activity when the number of active clients is more than two.

Since SL-RCB and CL-RCB send back to the server updated pages at transaction commit, they increase the burden placed on the server’s database disk and CPU. In addition, the server’s CPU is also loaded because the server has to demote all exclusive locks for each committed transaction. This is translated to 6,250 searches in the lock hash tables for each transaction. On the other hand, both SL-WCB and CL-WCB schemes reduce the amount of paging in the system maintaining the updated pages in the cache of each client and, thus, outperform the former schemes.

CL-WCB and SL-WCB perform exactly the same when the UpdateOne traversal is used. The same holds for the CL-RCB and SL-RCB algorithms. This is in contrast to the results we collected for the same operation and the small database. Nevertheless, these results were expected since the logging activity is petty compared to the paging activity that takes place in both the client buffer pools and the server cache. As we mentioned in the previous section, the UpdateOne traversal generates only one log record of size 84 bytes, as opposed to the 15 M-bytes of data that are accessed by each transaction.

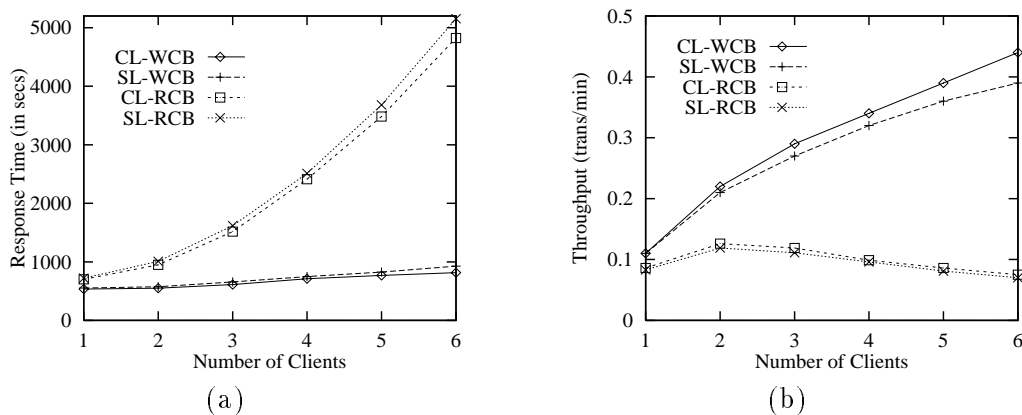


Figure 5.4: UpdateAll: large database (a) Response (b) Throughput

Figure 5.4 shows the response time and throughput versus the number of active clients for the UpdateAll traversal. Not surprisingly, CL-WCB performs better than SL-WCB. By comparing the results shown in Figure 5.4 and the results shown in Figure 5.2 we notice that the difference in performance between CL-WCB and SL-WCB is smaller when the large database is used for the UpdateAll traversal. In particular, CL-WCB is 13% faster than SL-WCB with six clients when the large database is used, as opposed to 17% when the same traversal is run against the small database. The reason for that is the paging activity in both the client buffer pools and the server’s cache. Similar to the UpdateAll traversal for the small database, the CL-RCB performs slightly better than the SL-RCB.

5.2.4 Extrapolations

We have also developed a queuing model, in Chapter 7.1, which allows us to do extrapolations for large populations of clients. There, we can illustrate the better scalability of client-based logging. The model uses standard tools from queuing networks, and specifically the ‘mean value’ analysis (MVA) [64] and the light-load and heavy-load asymptotes.

As a first step, we compare the analytical results with the experimental ones, to obtain confidence on the model. For brevity, we show the plots for the write-callback case, because it has higher throughput than the read-callback. Figure 5.5 plots the throughput as a function of the number of clients; it shows both the measured, as well as the analyti-

cally expected throughput. Two workloads are shown: (a) the UpdateAll one and (b) the UpdateRepeat one. Both workloads are on the small database. Notice that the analysis is close to the experiments, with the right trends.

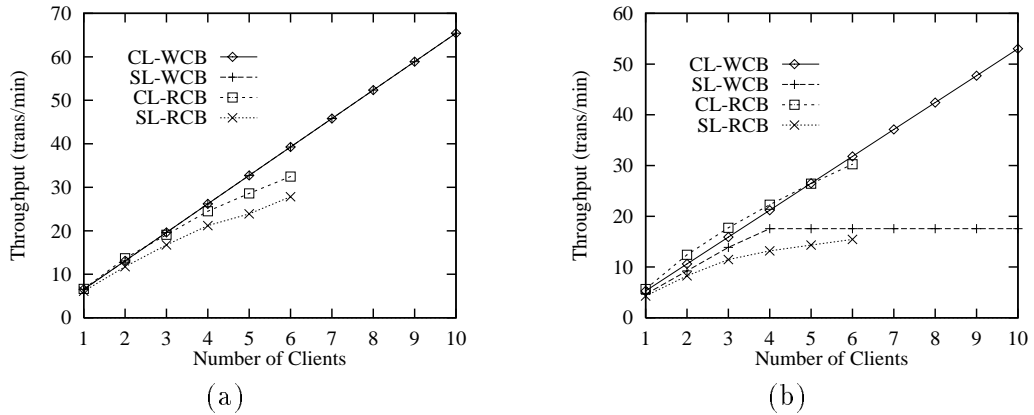


Figure 5.5: Illustration of the accuracy of the analytical model

Having obtained some confidence in the model, next we use it for extrapolations. Figure 5.6 shows the same workloads as before, for a wider range of multiprogramming level (up to 40). Notice that even in the UpdateAll case, our method eventually achieves large savings, because the server-logging method eventually hits a bottleneck at about 12 clients, and remains flat from then on.

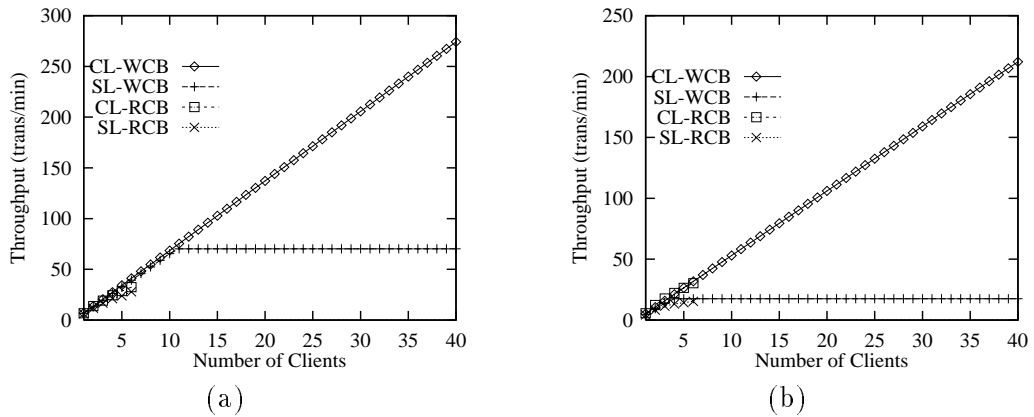


Figure 5.6: Extrapolations using the analytical model. Throughput vs number of clients. Small database with write-callback; (a) UpdateAll (b) UpdateRepeat

5.3 Related Work

A performance analysis was presented in [29] as part of the evaluation of transaction recovery in the Exodus client-server storage manager (ESM-CS). The recovery algorithm described in [29] is based on ARIES [40] and cache consistency is based on the *check on access* protocol [17]. Exodus provides a function call interface to application programs and requires all dirty pages be sent to the server at transaction commit.

In [11], the performance of a redo-only recovery algorithm was presented for the EOS client-server storage manager. EOS does not keep track of updates performed in a page and it logs the entire after image of the page, which is an expensive solution for small updates. In addition, EOS does not support inter-transaction caching.

In [2], a study of the performance trade-offs between recovery time and run-time overhead is presented for an asynchronous replica management algorithm. In this study, a shared nothing computing environment is assumed and recovery is based on ARIES. In addition, all log records generated by the nodes are shipped to a log server that is highly reliable and available. Another assumption that is made is that dirty pages are written to disk by daemon processes and the buffer manager never needs to write a page to disk for making room in the buffer pool for another page.

The ARIES/CSA architecture proposed in [42] follows the traditional client-server recovery paradigm where clients send all their log records to the server as part of the commit processing. However, ARIES/CSA clients are allowed to take checkpoints as well as roll-back aborted transactions and modified pages are not sent to the server at transaction commit. Although ARIES/CSA employs a fine-granularity concurrency protocol, clients are not allowed to update the same page simultaneously. However, it is not clear how a memory-mapped storage system could efficiently support fine-granularity locking since the memory-mapped technique is page-based due to the underlying virtual memory management subsystem.

QuickStore [69] is a memory-mapped client-server storage system for persistent C++ implemented on top of ESM-CS. QuickStore uses page-level strict two-phase locking for concurrency and the ESM-CS ARIES redo-undo for recovery. The authors presented a study of a number of recovery protocols in [70], including two algorithms based on page diffing. The page diffing approach in QuickStore generates one log record for each modified object present in each updated page while the algorithms we present in this chapter generate only one log record for each updated page. In addition, the performance studies presented in [70] do not take into account the particular callback algorithm employed by the system. Finally, [70] studied server-based logging recovery algorithms only, while we have included in our study two client-based logging algorithms.

A number of schemes for efficient generation of logging information in persistent programming languages and their relative performance were presented in [32]. Similar to our study, all of the schemes presented in [32] use diffing for generating log records. However, [32] does not examine the scalability of the logging algorithms and the effects of the concurrent control protocol employed since all studies were carried out in a single user system.

The study presented in this chapter also differs from [32] in that we examine both client-based and server-based logging schemes. We also examine the performance of the different recovery techniques we studied when a different callback algorithm is used and when the database size is very large and causes a significant amount of paging in the system.

5.4 Conclusions

The vast majority of the existing performance studies of recovery algorithms for data-shipping client-server architectures assume that the server owns both the database and the log file. Clients access the database by caching parts of it in local buffer pools, and they generate log records for the updates they perform. These log records are sent to the server, and they are stored in the log so that the durability of the committed updates is guaranteed. To increase system performance, clients are allowed to cache both data and locks across transactions. Although performance studies regarding client cache consistency exist in abundance, these studies are based on simulations and they ignore issues related to recovery.

In this chapter we have presented a performance analysis of two classes of recovery algorithms for client-server systems: server-based and client-based logging, both of which are based on the ARIES redo-undo recovery scheme. The first technique assumes that the server is the only source for providing transactional facilities. The second technique exploits client disk space for offering transactional facilities locally. The performance of these recovery algorithms was studied under the read and write callback inter-transaction caching algorithms, resulting in four different alternatives. All of these alternatives were implemented in BeSS, a memory-mapped object storage manager based on a page shipping client-server architecture and page diffing was used for generating log records.

The results of the study show that the client-based logging algorithms perform better than the server-based logging algorithms, especially when the logging activity in the system increases. Client-based logging is better than server-based logging even when a significant amount of paging is present in the system. By using local disks for logging and recovery, the load placed on the server's CPU and log disk is greatly reduced. This is translated to better transaction response times and better system scalability when the number of active clients increases, since server performance bottleneck is greatly avoided.

The performance study also compared the relative performance of the read and write callback algorithms. The results show that write callback offers much better performance than read callback, especially when the number of pages updated by each client increases. Under the read callback algorithm, clients have to communicate with the server to obtain exclusive locks during transaction execution and send all the pages updated by the transaction back to the server at transaction commit. In addition, the server has to traverse its lock tables and demote all exclusive locks held by the committed transaction to shared. Consequently, as the number of pages updates by a transaction increases and the number of clients interacting with the server also increases, the server becomes a performance bottleneck.

Chapter 6

Summary and Future Work

Current technological advances and the proliferation of inexpensive workstations and networks have made it possible to implement database functionality on desktop computers and workstations. At the same time, non-traditional applications have placed increased demands for high-performance transaction processing on database systems. The confluence of these two trends has raised significant challenges and performance opportunities for the design of new database management systems.

This thesis has presented and investigated recovery algorithms that exploit client disks for offering transactional facilities locally while maintaining the transaction semantics associated with traditional database systems. 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. This is because client-based transaction management is an effective way to reduce dependencies on server CPU and disk resources, preventing the server from becoming a performance bottleneck as quickly when the number of clients accessing the database increases.

The thesis also described the components of the BeSS memory-mapped object storage manager that are related to the implementation of recovery. In addition, the thesis presented a detailed performance study of the implementation of the proposed algorithms in BeSS. The results of the study show that the client-based logging algorithms perform better than the server-based logging algorithms particularly when the logging activity in the system increases. Client-based logging is better than server-based logging even when a significant amount of paging is present in the system. By using local disks for logging and recovery, the load placed on the server's CPU and log disk is greatly reduced. This translates to better transaction response times and better system scalability when the number of active clients increases since the server performance bottleneck is greatly avoided.

Today, the move from centralized data management environments to network-based and even mobile environments is accelerated. As a result, efficient and flexible techniques for managing data in a dynamic distributed environment will be of great importance in the coming years. We believe that future client-server database systems will combine data shipping for interactive and fast navigation with query shipping for efficient processing of large volumes of data. We plan to study and investigate the performance and architectural

tradeoffs present in these new systems.

Ongoing advances in broadband networking, image compression techniques, and storage devices have made multimedia applications to be of great interest and growing practical importance. However, most of the current research in multimedia has concentrated on algorithms for efficient disk scheduling and data placement, and it has ignored the overall system architecture. We plan to explore the technology needed to support multimedia in an environment of conventional networked workstations and PCs (as opposed to a special-purpose video on demand environment), as well as integrate multimedia services with the BeSS object storage manager we described in this thesis.

Appendix

7.1 Queueing Model

Here we present a simple queueing model, and the associated “Mean Value Analysis” [64]. In queueing theory, a *closed* queueing system consists of: (a) S queueing stations, and (b) n jobs, grouped into one or more classes, with n_i jobs for the i -th class.

What is necessary to provide is the 2-d matrix with the load $\rho_{i,j}$ that job-type i requires from the service station j . The load is the time units of service that this job-type requires from the j -th station. Once this is done, we can compute the throughput TP_i for each class, the total throughput TP and the utilization u_j of the j -th queueing station. Moreover, we can quickly estimate the light-load and heavy-load throughput asymptotes, which are optimistic bounds on the total throughput.

7.1.1 Formulas for the Loads

Here we compute the loads as a function of the hardware characteristics (disk speed, network speed, cache sizes) and of the recovery algorithm. Notice that each transaction is in a class of its own.

The queueing stations are: (a) the data disk of the server DD , (b) the log disk of the server LD , (c) the CPU of the server $CPUS$, (d) the network NET , (e) the CPU of each client ($CPUC_j$ for the j -th client), and (f) the log disk of each client, if it exists (LDC_j , for the j -th client).

We are given the following hardware characteristics:

- The average disk access time per page $DiskAccessTime$ (= 0.01sec).
- CPU speeds, in MIPS (actually, in ‘instructions per second’ IPS: $IPSS$ for the server, $IPSc$ for each of the clients; 50Mips for all machines).
- The CPU instructions to send a message over the net (just the header) $FixedMsgInst$ (=20,000), and per Kbyte $PerKByteMsgInst$ (= 2,500).
- The network transfer rate $transferRate$ (8Mbits per second).
- Cache sizes on the server (C_s) and on each of the clients (C_c).

We are also given the following characteristics of the OS and of the benchmark:

- Page size P in Kbytes (=4Kb).
- The ratio b of data pages over log-pages generated.
- CPU time to read and to update 1 page of data
($cpuSecsPerPageRead=0.005sec$, $cpuSecsPerPageUpdated=0.0065sec$).
- The number of pages read N and number of pages updated N_u per transaction.
- The module size M (in pages) that each transaction operates on.

Given the above, we need to estimate the hit ratio of a cache for the following setting: With a cache size of C , a size of $DBSIZE$ pages in the database and a stream of transactions with N page-read requests and N_u page-update requests, we need to estimate: (a) the hit ratio of the transaction, and (b) the number of dirty pages that will have to be written back. The answers are:

- Hit ratio $h = C/DBSIZE$, both for reads and writes.
- $N_u * (1 - h)$ dirty-pages of the cache will be replaced, on the average.

Let $h_c = Cc/M$, $h_s = Cs/n/M$ be the hit ratio for a client and for the server cache, respectively.

Given the above, we have the following loads for each queueing station, for each of the following four settings (read/write callback \times server/client logging).

Server-Based Logging Read-Callback

Here the i -th transaction will require the following service ('load') from each of the queueing stations:

$$\rho_{i,DD} = ((1 - h_c) * (1 - h_s) * N + (1 - h_s) * N_u) * DiskAccessTime \quad (7.1)$$

These are the actual page misses for a given transaction, given the caches at the server and the client.

$$\rho_{i,LD} = N_u/b * DiskAccessTime \quad (7.2)$$

For the network, we shall have $KbOverNet$ Kbytes to be shipped:

$$KbOverNet = (N_u/b + N_u + N(1 - h_c)) * P \quad (7.3)$$

comprising $nMessages$ messages:

$$nMessages = N_u/b + N_u + N(1 - h_c) + N_u \quad (7.4)$$

where we count the messages for log pages, pages fetched for update, cache misses for read requests, and lock requests, respectively. Notice that, since no write-locks are maintained at a client, all the N_u update requests have to be fetched from the server. Then:

$$\rho_{i,NET} = (KbOverNet + nMessages * controlMsgSize) * xferRate \quad (7.5)$$

The CPU of the server will mainly be occupied with shipping pages over the network:

$$\rho_{i,CPUS} = (FixedMsgInst * nMessages + PerKByteMsgInst * KbOverNet) / IPSs \quad (7.6)$$

For the CPU of the given client i , we have

$$\rho_{i,CPUCi} = N * cpuSecsPerPageRead + N_u * cpuSecsPerPageUpdated \quad (7.7)$$

Notice that

$$\rho_{i,CPUCj} = 0 \quad \text{if } i \neq j \quad (7.8)$$

Of course, since there is no local log disk, its load is zero.

Client-Based Logging Read-Callback

Similar to the remote-logging case, with the following differences: (a) the client does *not* send the log pages to the server; this reduces the load of the network, of the CPU at the server and of the log disk at the server; and it increases the load of the local load disk. We just list the different loads:

$$\rho_{i,LD} = 0 \quad (7.9)$$

The volume of data over the network are reduced by N_u/b , to:

$$KbOverNet = (N_u(1 - h_c) + N(1 - h_c)) * P \quad (7.10)$$

comprising $nMessages$ messages:

$$nMessages = N_u(1 - h_c) + N(1 - h_c) + N_u(1 - h_c) \quad (7.11)$$

The above two values affect the load of the network and the server's CPU. The load of the log disk of the client is also affected:

$$\rho_{i,LDCi} = N_u/b * DiskAccessTime \quad (7.12)$$

Server-Based Logging Write-Callback

It is similar to the case with read-callback and remote logging, with the difference that each client requests not N_u pages, but only the cache misses ($N_u * (1 - h_c)$). Of course, all N_u/b log records per transaction still have to be appended to the server's log disk.

$$\rho_{i,DD} = ((1 - h_c) * (1 - h_s) * N + (1 - h_s) * N_u * (1 - h_c)) * DiskAccessTime \quad (7.13)$$

$$\rho_{i,LD} = N_u/b * DiskAccessTime \quad (7.14)$$

For the network, we shall have $KbOverNet$ Kilobytes to be shipped:

$$KbOverNet = (N_u/b + N_u(1 - h_c) + N(1 - h_c)) * P \quad (7.15)$$

comprising $nMessages$ messages:

$$nMessages = N_u/b + N_u(1 - h_c) + N(1 - h_c) + N_u(1 - h_c) \quad (7.16)$$

where we count the messages for log pages, cache misses at the client for update requests, cache misses for read requests, and lock requests, respectively. Then:

$$\rho_{i,NET} = (KbOverNet + nMessages * controlMsgSize) * xferRate \quad (7.17)$$

The CPU of the server will mainly be occupied with shipping pages over the network:

$$\rho_{i,CPUS} = (FixedMsgInst * nMessages + PerKByteMsgInst * KbOverNet) / IPSs \quad (7.18)$$

For the CPU of the given client i , we have:

$$\rho_{i,CPUCi} = N * cpuSecsPerPageRead + N_u * cpuSecsPerPageUpdated \quad (7.19)$$

Notice that

$$\rho_{i,CPUCj} = 0 \quad \text{if } i \neq j \quad (7.20)$$

Of course, since there is no local log disk, its load is zero.

Client-Based Logging Write-Callback

Similar to the read-callback and remote-logging, with the difference that the N_u/b log pages are *not* shipped over the network to the log-disk server, but they are stored on the local log disk. Thus the differences are as follows:

$$\rho_{i,LD} = 0 \quad (7.21)$$

Network traffic:

$$KbOverNet = (N_u(1 - h_c) + N(1 - h_c)) * P \quad (7.22)$$

comprising $nMessages$ messages:

$$nMessages = N_u(1 - h_c) + N(1 - h_c) + N_u(1 - h_c) \quad (7.23)$$

$$\rho_{i,LDCi} = N_u/b * DiskAccessTime \quad (7.24)$$

For all the above cases, we can compute the load matrix - an example is in Table 7.1.

job	<i>LD</i>	<i>DD</i>	<i>CPUS</i>	<i>NET</i>	<i>CPUC</i> ₁	<i>LDC</i> ₁	<i>CPUC</i> ₂	<i>LDC</i> ₂
#1	0.00	0.00	0.00	0.00	0.72	11.52	0.00	0.00
#2	0.00	0.00	0.00	0.00	0.00	0.00	0.72	11.52

Table 7.1: Example of load matrix

7.1.2 Throughput Asymptotes

The exact estimation of the formulas grows rapidly with the number of classes, and may take too long. In our case, each job is in a separate class, because each job has drastically different loads, with respect to the clients' CPUs and log disks. A fast alternative is to use optimistic bounds. The two typical bounds used in queueing networks are the *light* load and the *heavy* load asymptotes. To use these bounds, we need to have a single-class model. We achieve it by equally spreading the local CPU and log-disk load of client i to all the CPUs and log-disks of all the clients. With this approximation, we have only one class of jobs, and still S queueing stations. The ' i ' subscript can thus be dropped from the load symbols: ρ_j denotes the load of the j -th queueing station due to ρ_j denotes the load of the j -th queueing station due to any one of the given jobs.

Light Load Asymptote

Here the idea is to assume that *no* job is ever waiting; thus, one job needs $\sum_j \rho_j$ seconds of service, with a throughput of $1/\sum_j \rho_j$ jobs/second; more (i.e., n) jobs lead to a throughput that is n times higher:

$$TP_{light} = n \times 1/\sum_j \rho_j \quad (7.25)$$

Heavy Load Asymptote

Here the optimistic assumption is to ignore all the queueing stations (i.e., load=0), except for the heaviest-loaded one, which will be come the bottleneck (i.e., the station with the longest queue, with utilization $> 100\%$). Let $\rho_{max} = \max_j \rho_j$ be the load of this station; then, we have

$$TP_{heavy} = 1/\rho_{max} \quad (7.26)$$

Among the two optimistic bounds we keep the smallest.

Bibliography

- [1] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.
- [2] A. Bhide, A. Goyal, H. Hsiao, and A. Jhingran. An efficient scheme for providing high availability. In *Proceedings of ACM-SIGMOD 1992 International Conference on Management of Data*, San Diego, California, pages 236–245, June 1992.
- [3] A. Biliris. An efficient database storage structure for large dynamic objects. In *Proceedings of the Eighth International Conference on Data Engineering*, Tempe, Arizona, pages 301–308, February 1992.
- [4] A. Biliris. The performance of three database storage structures for managing large objects. In *Proceedings of ACM-SIGMOD 1992 International Conference on Management of Data*, San Diego, California, pages 276–285, May 1992.
- [5] A. Biliris, S. Dar, and N. Gehani. Making C++ objects persistent: The hidden pointers. *Software Practice and Experience*, 23(12):1285 – 1303, December 1993.
- [6] A. Biliris, N. Gehani, D.F. Lieuwen, E. Panagos, and T. Roycraft. Ode 2.0 User’s Manual. Technical report, AT&T Bell Laboratories, 1993.
- [7] A. Biliris, W. O’Connell, and E. Panagos. BeSS reference manual. Technical report, AT&T Bell Laboratories, January 1995.
- [8] A. Biliris and J. Orenstein. Object storage management architectures. In A. Dogac, M. T. Ozsu, A. Biliris, and T. Sellis, editors, *Object-Oriented Database Systems*. Springer-Verlag, New York, 1994.
- [9] A. Biliris and E. Panagos. EOS User’s Guide, Release 2.0. Technical report, AT&T Bell Laboratories, May 1993.
- [10] 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.
- [11] A. Biliris and E. Panagos. Transactions in the client-server EOS object store. In *Proceedings of the Eleventh International Conference on Data Engineering*, Taipei, Taiwan, pages 308–315, March 1995.

- [12] P.A. Buhr, A.K. Goel, and A. Wai. μ Database: A toolkit for constructing memory mapped databases. In *Proceeding of the Fifth International Workshop on Persistent Object Systems*, San Miniato, Italy, pages 166–185, September 1992.
- [13] P. Butterworth, A. Otis, and J. Stein. The GemStone object database management system. *Communications of the ACM*, 34(10):51–63, October 1991.
- [14] 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.
- [15] M.J. Carey, D.J. DeWitt, G. Graefe, D. Haight, D. Richardson, D. Schuh, E. Shekita, and S. Vandenberg. The EXODUS extensible DBMS project: An overview. In S.B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, pages 474–500. Morgan Kaufmann, San Mateo, California, 1990.
- [16] M.J. Carey, D.J. DeWitt, and J.F. Naughton. The OO7 benchmark. In *Proceedings of ACM-SIGMOD 1993 International Conference on Management of Data*, Washington, D. C., pages 12–21, May 1993.
- [17] M.J. Carey, M.J. 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.
- [18] 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.
- [19] F.J. Carino, W. Sterling, and I.T. Leong. MoonBase - a complete multimedia database solution. In *Proc. of the ACM Multimedia '94 Conference, Workshop on Multimedia Database Management Systems*, pages 27 – 36, San Fransisco, California, October 1994.
- [20] R.G.G. Cattell. *Object Database Standard: ODMG-93*. Morgan Kaufmann, San Mateo, California, 1993. Contributions by T. Atwood, J. Dubl, G. Ferran, M. Loomis, and D. Wade.
- [21] R.G.G. Cattell and J. Skeen. Object operations benchmark. In *ACM Transactions on Database Systems*, pages 1–31, March 1992.
- [22] G. P. Copeland and D. Maier. Making Smalltalk a database system. In *Proceedings of ACM-SIGMOD 1984 International Conference on Management of Data*, Boston, Massachusetts, June 1984.
- [23] 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.
- [24] O. Deux et al. The O_2 system. *Communications of the ACM*, 34(10):51–63, October 1991.

- [25] D.J. DeWitt, D. Maier, P. Fattersack, 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.
- [26] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Transactions on Database Systems*, 9(4):560–595, December 1984.
- [27] M.J. Franklin, M.J. 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.
- [28] M.J. Franklin, M.J. 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.
- [29] M.J. Franklin, M. Zwilling, C. Tan, M.J. Carey, and D.J. 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.
- [30] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, California, 1993.
- [31] T. Haerder and A. Reuter. Principles of transaction oriented database recovery — a taxonomy. *ACM Computing Surveys*, 15(4):289–317, December 1983.
- [32] A.L. Hosking, E.W. Brown, and J.E.B. Moss. Update logging for persistent programming languages: A comparative performance evaluation. In *Proceedings of the Nineteenth International Conference on Very Large Databases*, Dublin, Ireland, pages 429–440, August 1993.
- [33] 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.
- [34] H.V. Jagadish, D.F. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. DALI: An extensible main memory storage manager. In *Proceedings of the Twentieth International Conference on Very Large Databases*, Santiago, Chile, pages 48–59, September 1994.
- [35] W. Kim, J. Garza, N. Ballou, and D. Woelk. Architecture of the ORION next-generation database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109–124, March 1990.
- [36] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):51–63, October 1991.
- [37] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

- [38] 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.
- [39] 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.
- [40] 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.
- [41] 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.
- [42] 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.
- [43] 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).
- [44] J.E.B. Moss. Design of the Mnome persistent object store. *ACM Transactions on Information Systems*, 8(2):103–139, April 1990.
- [45] J.E.B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(8):657–673, August 1992.
- [46] M. Nelson, B. Welch, and J. Ousterhout. Caching in the sprite network file system. *ACM Transactions on Computer Systems*, 6(1), March 1992.
- [47] *Network File System: Version 2 Protocol Specification*. Sun Microsystems, Inc., Mountain View, California, 1988.
- [48] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8), 1991.
- [49] Objectivity Inc. *Objectivity/DB Documentation V2*, 1993.
- [50] ONTOS Inc., Burlington, Massachusetts. *Ontos DB 2.2 Reference Manual*, 1992.
- [51] 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.

- [52] 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.
- [53] E. Rahm. Concurrency and Coherency Control in Database Sharing Systems. Technical Report ZRI 3/91, University of Kaiserslautern, Germany, Dec. 1991. Revised August 1992.
- [54] E. Rahm. Recovery Concepts for Data Sharing Systems. In *Proceedings 21st International Conference on Fault-Tolerant Computing*, Montreal, June 1991.
- [55] E. Rahm. Empirical Performance Evaluation of Concurrency and Coherency Control Protocols for Database Sharing Systems. *ACM Transactions on Database Systems*, 18(2):333–377, June 1993.
- [56] T. Rengarajan, P. Spiro, and W. Wright. High availability mechanisms of VAX DBMS software. *Digital Technical Journal* 8, pages 88–98, February 1989.
- [57] J. E. Richardson. Compiled item faulting: A new technique for managing I/O in a persistent language. In *Proceeding of the Fourth International Workshop on Persistent Object Systems*, Martha’s Vineyard, Massachusetts, pages 3–16, September 1990.
- [58] J.E. Richardson and M.J. Carey. Persistence in the E language: Issues and implementation. *Software Practice and Experience*, 19(12):1115–1150, December 1989.
- [59] 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.
- [60] E. Shekita and M. Zwillig. Cricket: A mapped, persistent object store. In *Proceeding of the Fourth International Workshop on Persistent Object Systems*, Martha’s Vineyard, Massachusetts, pages 89–102, September 1990.
- [61] V. Singhal, S.V. Kakkad, and P.R. Wilson. Texas: An efficient, portable persistent store. In *Proceeding of the Fifth International Workshop on Persistent Object Systems*, San Miniato, Italy, pages 11–33, September 1992.
- [62] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.
- [63] M. Sullivan and M. Stonebraker. Using write protected structures to improve software fault tolerance in highly available database management systems. In *Proceedings of the Seventeenth International Conference on Very Large Databases*, Barcelona, Spain, pages 171–180, August 1991.
- [64] K.S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. Prentice Hall, Inc., Englewood Cliffs, NJ, 1982.

- [65] F. Velez, G. Bernard, and V. Darnis. The O2 object manager: an overview. In *Proceedings of the Fifteenth International Conference on Very Large Databases*, Amsterdam, Netherlands, pages 357–366, August 1989.
- [66] Versant Object Technology, Menlo Park, California. *VERSTANT System Reference Manual, Release 1.6*, 1991.
- [67] 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.
- [68] S.J. White and D.J. DeWitt. A performance study of alternative object faulting and pointer swizzling strategies. In *Proceedings of the Eighteenth International Conference on Very Large Databases*, Vancouver, British Columbia, pages 419 – 431, August 1992.
- [69] S.J. White and D.J. DeWitt. QuickStore: A high performance mapped object store. In *Proceedings of ACM-SIGMOD 1994 International Conference on Management of Data*, Minneapolis, Minnesota, pages 395–406, May 1994.
- [70] S.J. White and D.J. DeWitt. Implementing crash recovery in QuickStore: A performance study. In *Proceedings of ACM-SIGMOD 1995 International Conference on Management of Data*, San Jose, California, pages 187–198, June 1995.
- [71] 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.
- [72] P.R. Wilson and S.V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge addresses on standard hardware. In *Int'l Workshop on Object Orientation in Operating Systems*, pages 364–377, Paris, France, September 1992.

Curriculum Vitae

Education

- Ph.D.** in Computer Science. Boston University, January 1996. Thesis Title:
“Client-based logging: a new paradigm for distributed transaction management.”
- M.Sc.** in Computer Science. Boston University, May 1992.
- B.Sc.** in Mathematics. University of Athens, 1988.

Professional Experience

- **AT&T Bell Laboratories, Murray Hill, NJ:** February 1993 - Current

BeSS: BeSS is a high-performance memory-mapped storage manager based on a peer-to-peer architecture. I designed and implemented several components of BeSS, including transaction management, buffer management, concurrency control, inter-transaction and inter-process caching, and client-server communication. BeSS uses standard virtual memory management facilities to provide fast pointer dereference, automatic detection of user updates, and protection against bad pointers and database corruption. In addition, references to persistent objects involve one level of indirection and simplify on-the-fly database reorganization. Finally, the BeSS server is intended to be an open server, capable of supporting a wide range of applications. Sophisticated users can link with the BeSS server a trusted piece of code in order to build specialized servers, like SQL and multimedia servers. BeSS is written in C++ and runs on UNIX.

EOS: EOS is a storage manager that offers fast and transparent access to persistent objects independent of their size in a client-server environment. I converted the single user version of EOS into a client-server architecture, and I designed and implemented transaction management and concurrency control, among other things. EOS objects are uninterpreted byte strings which can range in size from a few bytes to gigabytes. EOS allows many readers and one writer to access the same data item simultaneously. EOS uses a write-ahead redo-only logging scheme that offers short logs, fast recovery from system failures, and non-blocking checkpoints. EOS is written in C and C++ and runs on UNIX.

- **Boston University, Computer Science Department, Boston, MA:** Spring 1991 - Fall 1992
 - Teaching Fellow:** for courses in Pascal, C, and databases.
 - Instructor:** for an introductory course in C.
 - Grader:** for an undergraduate database course.

Refereed Conference Publications

1. W. O'Connell, I.T. Jeong, D. Schrader, C. Watson, G. Au, A. Biliris, S. Choo, P. Colin, G. Linderman, E. Panagos, J. Wang, T. Walter. "A Teradata Content-Based Multimedia Object Server for Massively Parallel Architectures." In *Proceedings of ACM-SIGMOD 1996 International Conference on Management of Data*, Montreal, Canada, June 1996.
2. A. Biliris, T. A. Funkhouser, W. O'Connell, E. Panagos. "BeSS: Persistent Objects for Virtual Environments." In *Proceedings of ACM-SIGMOD 1996 International Conference on Management of Data*, Montreal, Canada, June 1996.
3. A. Biliris, H. Chen, S. Choo, K. Ganapathy, G. Linderman, W. O'Connell, E. Panagos, D. Schrader. "Prospector: A Content-Based Multimedia Server for Massively Parallel Architectures." In *Proceedings of ACM-SIGMOD 1996 International Conference on Management of Data*, Montreal, Canada, June 1996.
4. E. Panagos, A. Biliris, H.V. Jagadish, R. Rastogi. "Client-Based Logging for High Performance Distributed Architectures." In *Proceedings of the 12th International Conference on Data Engineering*, New Orleans, Louisiana, USA, February 1996.
5. E. Panagos, A. Biliris, H.V. Jagadish, R. Rastogi. "Fine-granularity Locking and Client-Based Logging for Distributed Architectures." In *Proceedings of the Fifth International Conference on Extending Database Technology*, Avignon, France, March 1996.
6. A. Biliris, E. Panagos. "Transactions in the Client-Server EOS Object Store." In *Proceedings of the 11th International Conference on Data Engineering*, Taipei, Taiwan, March 1995.
7. A. Biliris, E. Panagos. "A High Performance Configurable Storage Manager." In *Proceedings of the 11th International Conference on Data Engineering*, Taipei, Taiwan, March 1995.
8. B. K. Hillyer, A. Biliris, E. Panagos. "The Calico Project for Continuous Media Services." In *Proceedings of the ACM Multimedia '94 Conference, Workshop on Multimedia Database Management Systems*, San Fransisco, California, October 1994.
9. A. Biliris, E. Panagos. "EOS: An Extensible Object Store." In *Proceedings of ACM-SIGMOD 1994 International Conference on Management of Data*, Minneapolis, Minnesota, May 1994.

Journal Publications

10. E. Panagos, A. Biliris. "Synchronization and Recovery in a Client-Server Storage System." *The VLDB Journal*.

Invited Papers

11. A. Biliris, E. Panagos. “BeSS: a Memory-Mapped Object Manager.” SIGMOD Record.

Technical Reports

12. A. Biliris, W. O’Connell, E. Panagos. “BeSS User’s Guide, Release 0.9.0.” AT&T Bell Laboratories, January 1996.
13. E. Panagos, A. Biliris, C. Faloutsos. “A Performance Study of Recovery Techniques for Distributed Memory Mapped Object Stores.” AT&T Bell Laboratories, November 1995.
14. A. Biliris, W. O’Connell, E. Panagos. “BeSS Reference Manual.” AT&T Bell Laboratories, January 1995.
15. A. Biliris, E. Panagos. “EOS User’s Guide, Release 2.0.” AT&T Bell Laboratories, May 1993.
16. A. Biliris, N. Gehani, D. Lieuwen, E. Panagos. “Ode 2.0 User’s Manual.” AT&T Bell Laboratories, May 1993.
17. A. Bestavros, S. Braoudakis, E. Panagos. “Performance Evaluation of Two-Shadow Speculative Concurrency Control.” Boston University, February 1993.

Pending Patents

- “Dynamic Hierarchical Resource Scheduling For Continuous Media.” With A. Biliris and B.K. Hillyer.
- “Client-Based Logging for High Performance Distributed Architectures.” With A. Biliris, H.V. Jagadish, and R. Rastogi.

Professional Activities

- Member of the Association for Computing Machinery (ACM).
- External referee for:
 - International Conference on Very Large Data Bases (VLDB) ’94
 - International Conference on High Performance Computing ’95
 - ACM SIGMOD International Conference on Management of Data ’96
 - Journal of Theory and Practice of Object Systems
 - Journal of Information Systems
 - The VLDB Journal